A Bucket-aware Asynchronous Single-Source Shortest Path Algorithm on GPU

Yuan Zhang
Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China, 100190
University of Chinese Academy of Sciences, Beijing, China, 100049
zhangyuan-ams@ict.ac.cn

Huawei Cao*
Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China, 100190
University of Chinese Academy of Sciences, Nanjing, China, 211135
caohuawei@ict.ac.cn

Jie Zhang
Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China, 100190
zhangjie20s@ict.ac.cn

Yiming Sun
Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China, 100190
sunyiming20g@ict.ac.cn

Ming Dun
Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China, 100190
dunming@ict.ac.cn

Junying Huang
Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China, 100190
huangjunying@ict.ac.cn

Xuejun An
Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China, 100190
axj@ict.ac.cn

Xiaochun Ye
Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China, 100190
yexiaochun@ict.ac.cn

ABSTRACT

Single-Source Shortest Path (SSSP) algorithm is a common routine in graph processing and has been extensively studied on Graphics Processing Unit (GPU). Despite the powerful parallelism resources and high memory bandwidth provided by GPU, the performance of the SSSP algorithm is hindered by several bottlenecks, such as irregular memory access, load imbalance, and redundant operations. In this paper, three optimizations are proposed to boost the performance of the SSSP algorithm on GPU, including property-driven reordering, adaptive load balancing, and bucket-aware asynchronous execution. Property-driven reordering is employed to improve the data locality and work efficiency. Adaptive load balancing brings a higher utilization of software and hardware GPU resources. Bucket-aware asynchronous execution presents a bucket-based approach for asynchronous implementation to accelerate the convergence of SSSP search.

Extensive experimental results show that our work outperforms the state-of-the-art SSSP implementations, including GPU-based and CPU-based, with an average speedup of 5.09× and 10.32× on real-world and synthetic graphs. In addition, our SSSP algorithm indicates good scalability when the graph scale and GPU platform change.

ICPP-W 2023, August 07–10, 2023, Salt Lake City, UT, USA

ACM ISBN 979-8-4007-0842-8/23/08.
https://doi.org/10.1145/3605731.3605746

CCS CONCEPTS

• **Computing methodologies** → **Parallel algorithms**.

KEYWORDS

Graph, Single-Source Shortest Path, Parallel Computing, GPU

ACM Reference Format:
Yuan Zhang, Huawei Cao, Jie Zhang, Yiming Sun, Ming Dun, Junying Huang, Xuejun An, and Xiaochun Ye. 2023. A Bucket-aware Asynchronous Single-Source Shortest Path Algorithm on GPU. In *52nd International Conference on Parallel Processing Workshops (ICPP-W 2023), August 07–10, 2023, Salt Lake City, UT, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3605731.3605746

1 INTRODUCTION

In the era of data explosion, the graph is a fundamental data structure and plays an important role in many applications. Therefore, graph processing has drawn great attention in data analysis and mining [12, 31, 33]. As a basic graph processing algorithm, Single-Source Shortest Path (SSSP) algorithm is one of the critical routines in various applications, including road layout management [5], network routing design [8] and social network analysis [23]. Nonetheless, the SSSP algorithm suffers from irregular memory access, load imbalance, and redundant operations. These characteristics pose tremendous challenges in terms of improving the performance of the SSSP algorithm.

In recent years, numerous studies have been conducted to improve the performance of SSSP based on Graphic Processing Unit (GPU). GPU offers abundant computing resources and high memory bandwidth, making it popular in fields of high-performance computing and artificial intelligence [11]. Taking V100 GPU as an example, it has 5120 CUDA cores and the peak memory bandwidth

reaches 900 GB/s. Different from Central Processing Unit (CPU), GPU uses Single-Instruction Multiple-Thread (SIMT) execution mode to schedule software and hardware resources, which makes it efficient for parallel computing. In a word, GPU provides high throughput, high parallelism, and low energy consumption, which motivates researchers to implement SSSP on GPU.

Initially, Harish and Narayanan implement the SSSP algorithm on GPU using the CUDA model [17]. It takes advantage of the parallel resources of GPU. Based on synchronous push mode, the work efficiency and memory efficiency of this work are poor. In 2014, Davidson *et al.* present three optimizations based on Δ-stepping algorithm, including Workfront Sweep, Near-Far, and Bucketing, providing better performance than traditional Bellman-Ford algorithm on GPU [10]. It only uses two *buckets* named Near and Far, and executes SSSP search in synchronous mode, leading to work inefficiency. Furthermore, Wang *et al.* implement a sophisticated work scheduler based on an approximate priority queue [34], which improves the work efficiency and performance of the SSSP algorithm. But the bottlenecks induced by redundant operations and load imbalance have not been well solved. In addition, existing GPU-based graph processing systems [6, 25, 28, 33, 35] implement SSSP algorithm with different optimizations, such as load balancing, coalesced memory accesses, and efficient programming model. Compared with works dedicated to optimizing the SSSP algorithm, the performance of SSSP in graph processing systems is sub-optimal.

Many real-world networks follow power-law distributions [7]. This distribution in addition to irregular memory accesses, workload imbalance, and wasteful operations make SSSP a challenging problem. Furthermore, the data dependency of the traversal makes it impossible to predict offline, and thus optimizing is hard. Therefore, our work focuses on improving the data locality, enhancing the utilization of GPU, and accelerating the convergence of SSSP algorithm. To implement a high-performance SSSP on GPU, we propose three optimizations: property-driven reordering, adaptive load balancing, and bucket-aware asynchronous execution mode. The detailed optimizations are as follows:

1) Considering the properties of vertex and edge, property-driven reordering optimization improves the data locality and work efficiency of the SSSP algorithm by revising and reordering the CSR format of the graph.

2) To improve load balance, the optimization of adaptive load balancing is proposed to adaptively select static or dynamic load balancing strategies, which enhances the utilization of GPU, including software and hardware resources.

3) We design a bucket-aware asynchronous execution mode to accelerate the convergence of SSSP search, cut down the overhead of synchronous barriers, and improve the parallelism of SSSP execution.

We conduct extensive experiments, and the results show that our work outperforms the state-of-the-art SSSP implementations, including GPU-based and CPU-based, with an average speedup of 5.09× and 10.32× on real-world and synthetic graphs. Additionally, our SSSP algorithm indicates good scalability when the graph scale and GPU platforms change.

The rest of this paper is organized as follows. Section 2 introduces the background. Section 3 discusses our motivations. The detailed optimizations, experiments, and results are presented in Section 4 and Section 5. In Section 6, we state the related work. Finally, we summarize our conclusion in Section 7.

2 BACKGROUND

SSSP algorithm is a fundamental algorithm in the field of graph analysis and processing. In this section, we will state the basic SSSP algorithm, including Dijkstra's algorithm, Bellman-Ford, and Δ-stepping algorithm.

2.1 Dijkstra's and Bellman-Ford Algorithm

SSSP algorithm is designed to find all shortest paths from a starting vertex to all other vertices in the graph. The basic operation of the SSSP algorithm is shown in Algorithm 1, which is called the relaxation of an edge. For a vertex u and its adjacent edge $e(u, v, w)$, where u, v and w denote the source vertex, destination vertex, and edge weight, respectively. The relaxation operation contains three steps: computation (line 1), check (line 2), and update (line 3).

Algorithm 1 The relaxation operation in SSSP algorithm.

Input: An active vertex u, one edge $e(u,v,w)$ of its adjacent edges, the distance array *dist*.
Output: The updated distance array *dist*.
1: $new\_dist = dist[u] + w$
2: **if** $new\_dist < dist[v]$ **then**
3: $dist[v] = atomicMin(dist[v], new\_dist)$
4: **end if**
5: **return** *dist*

The typical SSSP algorithms include Dijkstra's algorithm [13] and Bellman-Ford algorithm [4]. Based on the strategy of the greedy algorithm, Dijkstra's algorithm selects an active vertex with the minimum tentative shortest distance to process in each iteration. It ends when no active vertex needs to be processed. In Dijkstra's algorithm, the priority queue is used to get the nearest active vertex, and each vertex is updated at most once, which indicates Dijkstra's algorithm is work efficient. However, the priority queue structure is not suitable for GPU. And in each iteration, only dealing with one active vertex's adjacent edges in parallel also leads to underutilization of GPU. Different from Dijkstra's algorithm, the Bellman-Ford algorithm deals with multiple active vertices in each iteration, which indicates it is parallel-friendly. But the Bellman-Ford algorithm has a serious problem in that each vertex may be updated several times, resulting in work inefficiency [2, 26]. Besides, to ensure the correctness of updates caused by race conditions, the atomic operation is needed in the Bellman-Ford algorithm.

2.2 Δ-stepping Algorithm

To balance the parallelism and work efficiency, Δ-stepping algorithm [27] is proposed. Combining the idea of Dijkstra's and Bellman-Ford algorithm, Δ-stepping algorithm introduces the concepts of "*bucket*" and "Δ". It divides edges into *light* edges and *heavy* edges. *Light* edges are those with weight less than Δ, while *heavy* edges are the remaining ones. Additionally, it puts all vertices into different *buckets* according to their tentative distances. For an integer i ($i \geq 0$), the *bucket* B_i stores the vertices whose distances belong

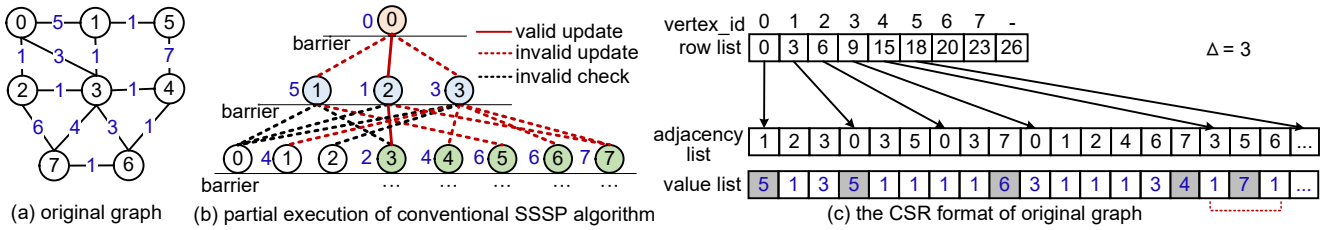

Figure 1: Illustration of graph and execution.

to $[i \cdot \Delta,\ (i+1) \cdot \Delta)$. Δ-stepping algorithm deals with all vertices for the non-empty *buckets* in the order of *bucket id*.

Δ-stepping algorithm has three *phases*. *Phase* 1: based on current *bucket*, it handles *light* edges for all active vertices in *bucket* until no active vertex fall into current *bucket*. *Phase* 2: for all vertices whose distances fall into current *bucket*, it deals with their *heavy* edges. *Phase* 3: it updates current *bucket* to next *bucket*, and then gets all vertices whose distances belong to the new current *bucket*. The Algorithm loops *phase* 1 to *phase* 3 until no vertex needs to be updated and then returns the array of distance (*dist*). In addition, *phase* 1 and *phase* 2 call the relaxation operation of Algorithm 1 multiple times to process the *light* and *heavy* edges of each active vertex respectively.

Δ-stepping algorithm can be viewed as a generalization of Dijkstra's algorithm and Bellman-Ford algorithm. For $\Delta = 1$, it is equivalent to Dijkstra's algorithm, while for $\Delta = \infty$, it is equivalent to the Bellman-Ford algorithm. Not only can it obtain good parallelism, but also improve work efficiency. Therefore, most existing works focus on boosting Δ-stepping algorithm to solve SSSP problems on GPU. Even so, during the execution of SSSP, the redundant overhead, load imbalance, and inefficient execution mode need to be reconsidered. The performance of the SSSP algorithm based on GPU can be further improved.

3 MOTIVATIONS

Due to the poor locality and irregularity of input graphs, as well as the unpredictable traversal path of the SSSP algorithm, the performance of SSSP implementation on GPU is unsatisfying. In this section, we focus on analyzing the bottlenecks of the SSSP algorithm.

we use a small graph to analyze and record the property of the graph, the execution process, and the updated frequency of the SSSP algorithm. The results are shown in Fig. 1. Fig. 1 (a) is an undirected graph with 8 vertices and 13 edges. Each vertex has a degree (the number of edges to adjacent vertices) property and each edge owns a weight property. The degrees of vertices and weights of edges are different. Fig. 1 (b) shows the partial execution of the SSSP algorithm starting from vertex 0 in synchronous mode. Among them, there are 2 valid updates, 7 invalid updates, and 5 invalid checks. An update is valid when it brings the final shortest distance of the vertex, otherwise, the update is invalid. A check is only valid if it shortens the tentative shortest distance. To a certain extent, invalid updates can further result in invalid updates and invalid checks. Besides, under the synchronous mode, the overhead of the synchronization barrier between iterations and the problem of slow convergence are issues worth considering.

3.1 Motivation 1: Good Locality

Since most graphs are sparse, the Compressed Sparse Row (CSR) format is widely used in graph processing to efficiently represent graphs in memory [14, 25, 28, 31, 39]. The basic CSR format contains *row list*, *adjacency list*, and *value list*. Using CSR format, the adjacent edges of each vertex are usually stored according to *vertex id*, which leads to unordered *value list*. In Fig. 1 (c), if the value of Δ is 3, for vertex 4, the weights of adjacent edges are 1 (< Δ), 7 (> Δ), and 1 (< Δ), respectively. We mark *heavy* edges' weight by gray background to show them explicitly in *value list*. As shown in Fig. 1 (c), the *light* and *heavy* edges of each vertex are usually mixed. Meanwhile, the execution of the SSSP algorithm highly depends on the weight property of the edge in the graph. Δ-stepping deals with *light* edges in *phase* 1 and *heavy* edges in *phase* 2. In these two *phases*, it exploits conditional branch statements to distinguish *light* or *heavy* edges. As a result, the branch and memory divergences occur frequently.

On GPU, Streaming Multiprocessor (SMs) schedules threads in a *Warp*. A *Warp* usually consists of 32 threads. 32 threads are executed in the SIMT mode, which easily leads to branch divergence problems due to different control paths. Under the circumstances, the *Warp* executes all the branches sequentially and disables threads that are not on the taken path. The divergence problem can largely reduce the utilization of SM, as well as the memory bandwidth. Thus, the reordering optimization based on the property of the graph can improve data locality and GPU utilization.

3.2 Motivation 2: Load Balancing

The degree of most real-world graphs follows power-law distribution [7]. For power-law graphs, a few vertices possess a high degree, which connects to a large number of vertices. While the majority of vertices connect to only a few vertices, which leads to severe sparse and irregularity of the graph. Meanwhile, during the execution of SSSP, most existing works use vertex-centric programming model [24, 31, 37], where each active vertex is considered as an individual processing element and its adjacent edges will be processed in parallel. Consequently, the execution of SSSP faces terrible load imbalance because of the power-law distribution.

To analyze the execution of Δ-stepping algorithm, we use two synthetic graphs with different *SCALE* 24, 25 and fixed *edgefactor* 16 to evaluate Δ-stepping implementation. Here, the parameters of *SCALE* and *edgefactor* represent the scale and average degree of the

graph, respectively. The number of vertices and edges in the graph are 2^{SCALE} and $edgefactor * 2^{SCALE}$. The Δ-stepping algorithm is implemented with Graph500 benchmark reference code [1]. We use the empirical Δ value 0.1 and choose the same starting vertex for different graphs. The number of active vertices in each *bucket* during the execution of Δ-stepping algorithm is shown in Fig. 2.

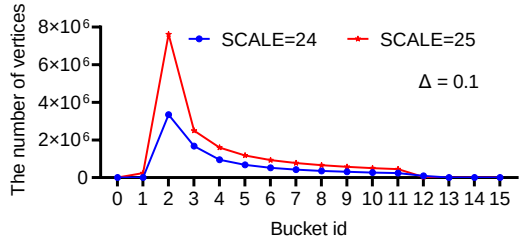

Figure 2: The active vertices in each *bucket* of Δ-stepping.

In Fig. 2, the number of active vertices in different *buckets* differs greatly. During the execution of SSSP, the number of active vertices increases dramatically in a given *bucket*, then decreases gradually in subsequent *buckets*. The number of edges that need to be processed also varies in different *buckets* and *phases* during the execution of the Δ-stepping algorithm. Therefore, SSSP suffers from severe load imbalance. Designing and using adaptive load balancing to maximize the utilization of GPU is important to improve the performance of the SSSP algorithm.

3.3 Motivation 3: Work Efficiency

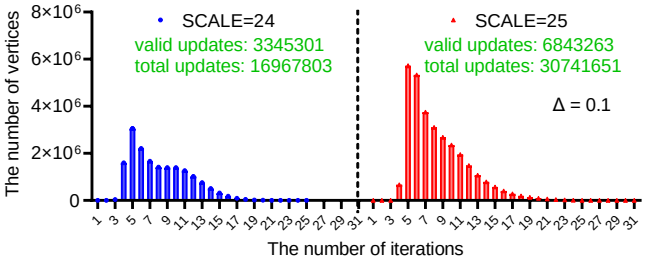

Figure 3: The detailed analysis of *phase* 1 in peak overhead of the *bucket*.

Besides, we further analyze the runtime of each *bucket* and find that the overhead of *bucket* with peak active vertices is accounting for seventy percent of the total execution time. The time consumption is mainly in *phase* 1, which processes the active vertices with synchronous mode. At the same time, the number of active vertices of each iteration in *phase* 1 is shown in Fig. 3. The iterations of these two graphs are more than 20, resulting in high synchronization overhead. The number of valid updates and total updates during *phase* 1 are also counted. For the graph with *SCALE* 25, the total updates are 30741651, which is nearly 4.49× larger than valid updates, causing work inefficiency. To accelerate the convergence and improve work efficiency, we propose bucket-ware asynchronous execution for Δ-stepping algorithm.

4 ALGORITHM OPTIMIZATIONS

Through the aforementioned analysis, we propose three optimizations to enhance the performance of SSSP. In this section, detailed optimizations will be presented, consisting of property-driven reordering, adaptive load balancing, and bucket-aware asynchronous execution.

4.1 Property-driven Reordering

The execution of SSSP highly depends on the vertex's degree property and the edge's weight property of the graph. According to the prior work [37], during the execution of SSSP, vertices with high degrees are frequently used, while the vertices with low degrees are rarely used. Hence, we reorder the vertices in descending order by degree and reassign the index for them. In this way, vertices with high degrees are assigned low *vertex id* and stored together. Besides, the workload property of the SSSP algorithm is order-sensitive [9] and the weight property has a great influence on SSSP search. Although we sort and reorder *vertex id* in descending order of degree, the weight in *value list* is still unordered, which results in branch and memory divergences. For the SSSP algorithm, the relaxation of edges with small weight values has a high possibility for valid updates. Hence, for each vertex, we further reorder the adjacent vertices in *adjacency list* and *value list* in ascending order of weight. In addition, to quickly locate the *heavy* edges in *phase* 2 of Δ-stepping algorithm, the offset of *heavy* edges is also added to *row list*.

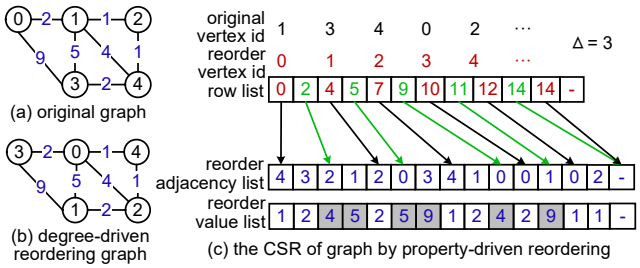

Figure 4: The example of property-driven reordering.

The example of property-driven reordering is shown in Fig. 4. Fig. 4 (a) presents the original graph. During processing, we first reorder vertices based on their degree, and then reorder the adjacency edges and edges' weights of each vertex by weight property. In Fig. 4 (a), the degree of vertices 0, 1, 2, 3, 4 are 2, 4, 2, 3, 3. According to the descending order of degree, we reorder the *original vertex id* from 0, 1, 2, 3, 4 to *reorder vertex id* 1, 3, 4, 0, 2, and the graph after reordering by degree is shown in Fig. 4 (b). The topology of the degree-driven reordering graph is the same as the original graph. Then, for each vertex, we sort the *adjacency list* and *value list* in ascending order of weight property. And the offset of each vertex's *heavy* edges is added to *row list*. Taking Fig. 4 (c) as an example, in *row list*, the values 2, 5, 9, 11, 14 are the offsets of *heavy* edges for vertices 0, 1, 2, 3 and 4 respectively. Ultimately, the CSR structure with property-driven reordering is shown in Fig. 4 (c). In Fig. 4 (b) and Fig. 4 (c), the higher degree, the lower *reorder vertex id*. And in *reorder adjacent list* and *reorder value list*, the adjacent edges of

each vertex are displayed according to the weight property. The green number in *row list* is the offset of *heavy* edges of each vertex.

Under property-driven reordering, the locality is enhanced in the SSSP algorithm. Therefore, memory efficiency and work efficiency are improved. Besides, by adding the offset of *heavy* edges to *row list*, we can quickly locate the offset of *light* edges and *heavy* edges, which is eminently suitable for Δ-stepping algorithm. What's more, the offset of *heavy* edges can be changed immediately in *phase* 1. In this way, it can adapt itself to the change of Δ value.

4.2 Adaptive Load Balancing

As discussed earlier, the workload is imbalanced in the SSSP algorithm due to the power-law distribution of graph and vertex-centric processing. Therefore, we design an adaptive load balancing by combining static and dynamic load balancing. Among them, the static load balancing allocates a fixed number of threads (e.g. 32 or 256) to process one active vertex based on the evaluation of workload, as well as the hardware and software characteristics of GPU. Dynamic load balancing takes good advantage of the dynamic parallelism on GPU, which allows GPU threads to dynamically launch other GPU threads. Dynamic parallelism is useful in applications with irregular and unpredictable nested parallelism [16, 29]. Besides, to reduce high kernel launch overhead and enhance the utilization of GPU resources, kernel fusion [25] is introduced in our optimizations. As a common optimization for a collection of iterative GPU applications, kernel fusion fuses multiple kernel functions into one function, so that the data processing can be completed with this function call.

In Δ-stepping algorithm, the workloads of *phase* 1, *phase* 2 and *phase* 3 are various for different *buckets*. For *phase* 1, the number of active vertices in different *buckets* and the number of *light* edges for different active vertices differ greatly. As a result, the traditional static workload distribution method is not sufficient to achieve good load balance for this *phase*. In this case, we propose to use dynamic parallelism to balance workload and improve the utilization of GPU. In *phase* 1, we initially launch a master kernel with a certain number of parent threads. The amount of parent threads is the same as the number of initial active vertices in *bucket*. Each parent thread deals with one active vertex. The workload of a parent thread is relevant to the number of *light* edges of the active vertex. With property-driven reordering, we can quickly calculate the number of *light* edges for each vertex. To fully use dynamic parallelism, the active vertices are divided into three categories based on the number of *light* edges, large workload list, middle workload list, and small workload list. More specifically, if the number of *light* edges is larger than parameter α, we put it into the large workload list. When the number of vertex's *light* edges is larger than parameter β and smaller than α, the vertex will be put into the middle workload list. The rest of the vertices are put into the small workload list. Allowing for the characteristics of GPU hardware and degree distribution of input graphs, the value of parameters α and β are 256 (the number of *Block* granularity threads) and 32 (the number of *Warp* granularity threads), respectively.

The detailed workload scheduling strategy of *phase* 1 is shown in Fig. 5. Initially, we classify the active vertices into three classes and put them into the corresponding workload list. Then, we launch a master kernel (grid) with a certain number of parent threads to deal with active vertices. The parent threads further create different kinds of child kernels to launch child threads. Considering GPU's architecture, all kernels execute concurrently with Hyper-Q support. The child threads are dynamically created in *Block*, *Warp* granularity depending on the number of *light* edges. For example, when an active vertex has 6 (<32) *light* edges, the parent thread will deal with these workloads instead of creating extra child threads. When an active vertex has 224 (<256) *light* edges, its parent thread will create 32 child threads (a *Warp* granularity threads) to deal with the *light* edges. When an active vertex has 4000 (<4096) *light* edges, its parent thread will create 256 child threads (a *Block* granularity threads). If the number of *light* edges of one active vertex is extremely large, which is greater than 4096, its parent thread will assign multiple *Block* granularity threads. Suppose an active vertex has n *light* edges, we will create $\lfloor n/4096 \rfloor$ *Block* threads. Besides, we limit the largest dimension of the master and child kernels to prevent the wasting of threads.

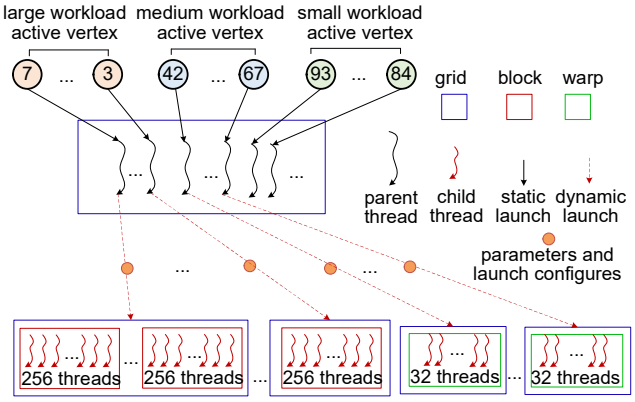

Figure 5: The detailed load balancing of *phase* 1.

Since the workloads of *phase* 2 and *phase* 3 are small, we use the static load balancing strategy by launching one master kernel. In these two *phases*, the algorithm loops all vertices, deals with *heavy* edges, and records the active vertices that fall into the next *bucket*, which makes us launch a fixed number of threads. For each thread, we coarsely assign the same number of *heavy* edges to guarantee load balancing. Besides, the logic of *phase* 2 is similar to *phase* 3. To reduce the high overhead of creating and destroying kernel, we use kernel fusion [25] technique to merge *phase* 2 and *phase* 3.

For different workload characteristics in different *phases* of the SSSP algorithm, we propose to use static and dynamic load balancing strategies. It improves the utilization of GPU and the performance of the SSSP algorithm.

4.3 Bucket-aware Asynchronous Execution

Based on the experimental results in section 3, we conclude that Δ-stepping algorithm suffers from the drawbacks of work inefficiency and slow convergence. And the main reason is that the fixed Δ value leads to different numbers of active vertices in various iteration layers during algorithm execution. Meanwhile, based on synchronous mode, the main bottleneck of *phase* 1 is the synchronization barrier.

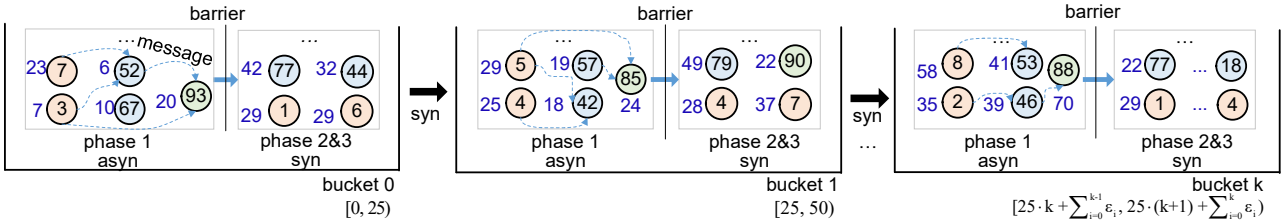

Figure 6: The execution flow of bucket-aware Δ-stepping.

In this regard, we propose a bucket-aware asynchronous execution. It contains readjusting Δ value to balance work efficiency and parallelism, as well as using asynchronous execution mode in *phase* 1 during algorithm execution.

$$\varepsilon_i = \begin{cases} 0\,, & i = 0, 1 \\ \left|\frac{C_{i-2}-C_{i-1}}{C_{i-2}+C_{i-1}}\right| * \frac{T_{i-2}-T_{i-1}}{T_{i-2}+T_{i-1}} * \Delta_0\,, & i >= 2 \end{cases} \quad (1)$$

$$\Delta_i = \Delta_{i-1} + \varepsilon_i \quad (2)$$

We design dynamic interval Δ_i of *bucket* i by adding parameter ε_i. The values of ε_i and Δ_i are calculated by Eq. (1) and Eq. (2). In Eq. (1), C_i and T_i denote the number of converged vertices and the number of threads in iteration i layer. The Δ_0 and Δ_1 value of the first and second *buckets* are fixed. Then, the following *buckets* are dynamically adjusted. It depends on the utilization of GPU and the number of converged vertices. Here, the number of threads stands for the utilization of GPU. As the utilization of GPU increases, we reduce Δ_i value, otherwise we increase Δ_i value. By dynamically readjusting the intervals of *buckets*, the work inefficiency and low parallelism between different *buckets* are alleviated.

Besides, to alleviate the synchronization overhead and accelerate the convergence of the algorithm, we propose a bucket-aware asynchronous mode. The whole execution flow of bucket-aware Δ-stepping is shown in Fig. 6. Here, we suppose that the initial Δ_0 is 25, the integral of *bucket* k is $[\Delta_{k-1},\ \Delta_k)$, where Δ_k is $25\cdot(k+1)+\sum_{i=0}^{k}\varepsilon_i$. Specifically, we use asynchronous mode in *phase* 1. While for *phase* 2 and 3, as well as processing between *buckets*, we use synchronous mode. Algorithm 2 describes the detailed bucket-aware asynchronous Δ-stepping algorithm.

In Algorithm 2, lines 1-3 initialize the algorithm based on the starting vertex s, lines 6-9 execute *phase* 1 of the algorithm in asynchronous mode, line 11 call *Update_Delta_Epsilon* function to update the changing value of Δ, lines 13-19 execute the *phase* 2 and 3 of algorithm in synchronous mode. Among them, the parameters of *Update_Delta_Epsilon* are G, ϵ, Δ, *csum* and *tsum*, of which the *csum* parameter is the number of converged vertices, *tsum* parameter records the number of threads used in this iteration. It partly reflects the utilization of GPU. *Deal_Heavy_Edges_Update_NBucket* function updates the *heavy* edges and gets the number of active vertices for next *bucket*. Next, we will focus on elaborating how *phase* 1 executes in asynchronous mode.

To implement asynchronous execution efficiently, we design manager threads and worker threads in *phase* 1. Manager threads are mainly deployed to control the flow of asynchronous execution and manage workload lists. Worker threads perform the basic operation of the SSSP algorithm to deal with the vertex's adjacent *light* edges. During the execution of worker threads, when a vertex's distance is updated, the worker thread will put the *vertex id* into the corresponding workload list based on the number of *light* edges. When a worker thread finishes all its traversal, the manager thread will pop a new work from a workload list to the worker thread. When all workload lists are empty and all worker threads complete their traversal, the *phase* 1 of the algorithm ends.

Algorithm 2 The bucket-aware asynchronous Δ-stepping.

Input: A graph *G=(V, E, W)*, the starting vertex *s*, the value of delta Δ, the distance array *dist*
Output: The final shortest distance array *dist*.

```
1:  min_delta=0, max_delta=Δ
2:  Queues: large, middle, small workload lists
3:  Queues.push(s), sum=1, csum=1, tsum= 1
4:  while sum≠0 do
5:    # deal with phase 1 in asynchronous mode
6:    while ¬ Queues.isEmpty() do
7:      # wait for processing workload lists asynchronously
8:      wait() # manager and worker threads deal with light edges
9:    end while
10:   # readjusting the value of Δ
11:   Update_Delta_Epsilon(G, ε_i, Δ, csum, tsum)
12:   # deal with phase 2 and 3 in synchronous mode
13:   for v ∈ V do
14:     if dist[v]≥min_delta then
15:       # deal with current active vertices' heavy edges
16:       # get next bucket's active vertices
17:       Deal_Heavy_Edges_Upate_NBucket(v, dist, Δ, ϵ, sum,
          Queues, csum, tsum)
18:     end if
19:   end for
20: end while
21: return dist
```

Besides, in *phase* 1, the update of one active vertex's distance and state are visible to other vertices. The newly activated vertices will be processed immediately. There is no synchronization barrier between active vertices belonging to different iterations. Taking Fig. 6 as an example, in each *bucket*, we use the color orange, light blue, and green to mark active vertices belonging to large, middle, and small workload lists respectively. For the *phase* 1 of *bucket* 0,

the algorithm processes 7, 52, 93 sequentially along the depth path, without synchronous wait between layer iterations.

By using this bucket-aware asynchronous execution optimization, the work inefficiency and low parallelism can be greatly improved. Meanwhile, the synchronization overhead is cut down, which accelerates the convergence of the algorithm.

4.4 Summarize of Optimizations

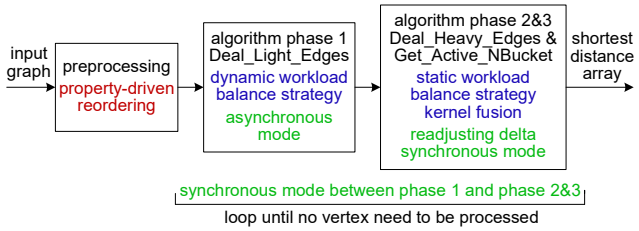

Figure 7: The overall optimization techniques in our work.

To summarize, the process of our algorithm and corresponding core optimization techniques are shown in Fig. 7. In Fig. 7, we use property-driven reordering optimization in the preprocessing stage. During the execution of the algorithm, we use an adaptive load-balancing strategy and bucket-aware asynchronous execution mode. Among them, we use dynamic load balancing strategy and asynchronous mode in *phase* 1. For the *phase* 2 and 3 of the algorithm, we use kernel fusion, readjusting Δ value, static load balancing, and synchronous mode techniques. Our three optimizations are tightly coupled, leading to a significant improvement in the performance of the algorithm.

5 EXPERIMENTS AND RESULTS

In this section, we will first present our experimental environment and input graphs. Next, the performance evaluation, work and memory efficiency analysis, and algorithm scalability will be clarified.

5.1 Experimental Setup

5.1.1 Hardware and Software. We implement our work using C++ and CUDA on the platform of an X86 server equipped with V100 GPU. In terms of hardware, the model of the X86 server is Intel(R) Xeon(R) Platinum 8269CY, with 26 cores for each CPU and 256GB memory. Meanwhile, the V100 GPU contains 5120 CUDA cores (80 SMs) and 16GB of global memory. The detailed software compiling environments are NVIDIA NVCC 11.2, CUDA Driver 460.106.00, and CNU GCC-7.4.0 with optimization flag -O3.

5.1.2 Datasets. In our evaluations, we use both synthetic and real-world graphs. The synthetic graphs are generated based on Graph500 benchmark [1]. It uses *Kronecker* generator to generate the graphs. The theory of *Kronecker* generator is similar to R-MAT generator [7]. The parameters with *A=0.57, B=0.19, C=0.19, D=0.05* determine the skewness of degree distribution. The *SCALE* and *edgefactor* determine the size of the vertex set and the average degree of vertices respectively. The number of vertices and edges in a synthetic graph is 2^{SCALE} and $edgefactor * 2^{SCALE}$. We name synthetic graph with *SCALE=21* and *edgefactor=16* as *k-n21-16*.

For real-world graphs, we use roadNet-TX [22], amazon0601 [18], web-Google [22], com-LiveJournal [36], soc-Pokec [32], com-Orkut [36], as-Skitter [21], soc-LiveJournal1 [3, 22], wiki-Talk [19, 20] and soc-twitter-2010 [30]. They involve different domains with various properties. We count the relevant information, including the number of vertices and edges, the average degree, and the maximum diameter. The detailed information is shown in Table 1. To exhibit them better, we call them road-TX, Amazon, web-GL, com-LJ, soc-PK, com-OK, as-Skt, soc-LJ, wiki-TK, and soc-TW respectively in the following tables, figures, and paper. In addition, due to the input graphs without providing the weight property of the edge, we use the random function that follows uniform distribution to generate different edges' weight values belonging to 1 to 1000.

Table 1: The detailed information of real-world graphs.

Graph	#vertices	#edges	#avg_deg	#diameter
road-TX	1,379,917	1,921,660	1.39	1054
Amazon	403,394	3,387,388	8.39	21
web-GL	875,713	5,105,039	5.82	21
com-LJ	3,997,962	34,681,189	8.67	17
soc-PK	1,632,803	30,622,564	18.75	11
com-OK	3,072,441	117,185,083	38.141	9
as-Skt	1,696,415	11,095,298	6.540	25
soc-LJ	4,847,571	68,993,773	14.233	16
wiki-TK	2,394,385	5,021,410	2.097	9
soc-TW	21,297,772	265,025,545	12.444	18

5.1.3 Experimental method. For the performance evaluation and analysis, we select 64 different starting vertices randomly. For each starting vertex, the SSSP search is launched 10 times to get the average performance. The performance of our experiments is measured in runtime overhead with milliseconds (ms) and Giga-Traversed Edges per Second (GTEPS). GTEPS takes the ratio of the number of edges in the graph over the traversal time. Meanwhile, to show our work better, in tables and figures, we mark our work as RDBS.

5.2 Performance Evaluation

In our experiments, we first evaluate the performance of our SSSP algorithm with different optimizations. Then we compare our work with the state-of-the-art SSSP implementations on GPU and CPU platforms.

5.2.1 Performance Variation with Optimizations. In our experiments, we choose a synchronization SSSP algorithm based on push mode as baseline (BL), which uses the static load balancing strategy. BASYN stands for our bucket-aware asynchronous execution mode optimization. PRO uses the strategy of property-driven reordering. ADWL uses the optimization of adaptive load balancing. Fig. 8 shows the speedup of using different optimizations over BL on real-world and synthetic graphs. From Fig. 8, we observe that the BASYN+PRO outperforms the BL by 1.36× to 9.97× on different graphs. BASYN and PRO optimizations improve the performance of the SSSP algorithm on GPU due to the following two reasons. First, the BASYN optimization reduces the overhead of redundant operations and accelerates the speed of algorithm convergence.

Second, the PRO optimization improves the data locality thereby achieving good memory access efficiency and work efficiency.

Even so, the performance can be further enhanced due to load imbalance. To make full use of the massive processing units of GPU, we design adaptive load balancing. ADWL solves the problem of load imbalance for SSSP implementation. The speedup of BASYN+ADWL concerning BL ranges from 1.47× to 45.88×. It achieves a very high improvement on the k-n21-16 graph due to the irregular degree distribution of the synthetic graph. Finally, we systemically evaluate our work (BASYN+PRO+ADWL) with BL. The results show that our work outperforms the BL by 1.38× to 53.44×. Compared with baseline (BL), the average speedup of BASYN+PRO, BASYN+ADWL, and BASY+PRO+ADWL are 5.15×, 16.37×, 19.60×. These results indicate that our optimizations can highly improve SSSP performance on GPU.

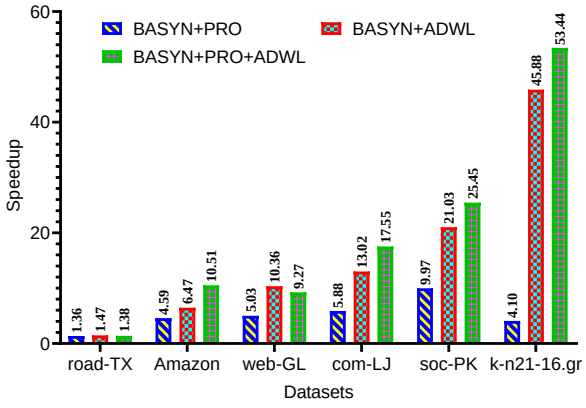

Figure 8: The speedup optimizations with various graphs.

5.2.2 Performance Comparison with Existing Work. To further show our performance improvement, we compare our work with ADDS [34] and PQ-Δ* [15]. ADDS is the state-of-the-art SSSP algorithm on GPU with asynchronous execution mode, which is implemented by Wang [34]. While PQ-Δ* is the state-of-the-art work of SSSP on the CPU. It uses a lazy-batched priority queue to enhance the efficiency and parallelism of the SSSP algorithm. We run PQ-Δ* using our host X86 server, 26 cores (1 CPU), 52 threads in total. The results are shown in Table 2. From Table 2, our Δ-stepping implementation is 4.48×, 9.81×, 5.62×, 15.13×, 17.35×, 9.53× faster than PQ-Δ* on road-TX, Amazon, web-GL, com-LJ, soc-PK, k-n21-16 graphs, respectively. The average speedup is 10.32×. Compared with PQ-Δ*, our work makes full use the GPU parallelism and high bandwidth to boost the performance of the SSSP algorithm.

For ADDS, our work outperforms it by 2.07×, 1.88×, 2.33×, 2.33×, 21.02× on Amazon, web-GL, com-LJ, soc-PK, k-n21-16 graphs, respectively. We also notice that our work is slower than ADDS on the road-TX graph. Compared to ADDS, our property-driven reordering, adaptive load balancing, and bucket-aware asynchronous execution bring good locality, higher utilization of GPU, and fast convergence of the SSSP algorithm. Additionally, for uniform-degree and high-diameter graphs, such as road-TX, the performance of our method is not as good as ADDS. The reason is that the property-driven reordering and adaptive load balancing optimizations are not fit for this type of graph. Besides, compared with real-world graphs, synthetic graphs show worse locality and irregularity characteristics, which results in bad performance using fundamental Δ-stepping.

Table 2: Running time (ms) and speedup compared with existing work on various graphs.

graphs	PQ-Δ* (CPU)	ADDS (GPU)	RDBS
road-TX	39.68 (4.48×)	8.10 (0.91×)	**8.86**
Amazon	19.62 (9.81×)	4.14 (2.07×)	**2.00**
web-GL	27.98 (5.62×)	9.34 (1.88×)	**4.98**
com-LJ	167.76 (15.13×)	25.84 (2.33×)	**11.09**
soc-PK	99.25 (17.35×)	13.34 (2.33×)	**5.72**
k-n21-16	42.60 (9.53×)	93.95 (21.02×)	**4.47**

5.3 Work and Memory Efficiency Analysis

In this section, we first analyze the work efficiency by counting and recording the ratio of total updates to valid updates and then use GPU profiling tools to further explore the work and memory efficiency of our algorithm.

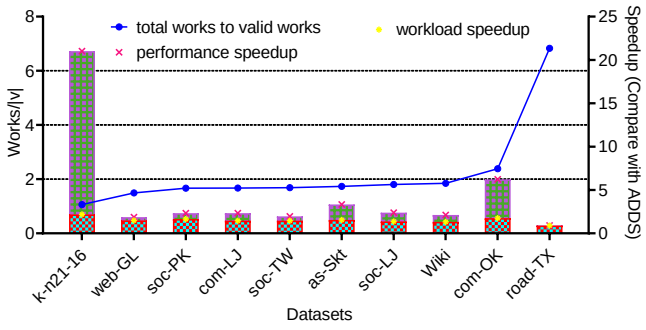

Figure 9: The work efficiency of our work.

5.3.1 Work Efficiency Analysis. Because different input graphs have different numbers of vertices, the number of update operations also varies. Thus, we use the ratio of the total number of updates to valid updates as an indicator to evaluate work efficiency. The detailed work efficiency of our work with different real-world graphs is shown in Fig. 9. The ratios of total updates to valid updates are 1.06, 1.49, 1.67, 1.67, 1.69, 1.73, 1.80, 1.85, 2.39, 6.83 on k-n21-16, web-GL, soc-PK, com-LJ, soc-TW, as-Skt, soc-LJ, Wiki, com-OK, road-TX graphs. The average ratio is 2.22. The total number of ADDS's updates is 2.18×, 1.48×, 1.65×, 1.46×, 1.46×, 1.55×, 1.37×, 1.33×, 1.75× larger than our work on k-n21-16, web-GL, soc-PK, com-LJ, soc-TW, as-Skt, soc-LJ, Wiki, com-OK graphs. Meanwhile, the performance of our algorithm performs 21.02×, 1.87×, 2.33×, 2.33×, 1.96×, 3.33×, 2.39×, 2.12×, 6.22× better than ADDS. Besides, we notice that the road-TX graph has a maximal ratio of total works to valid works and poor performance. Work inefficiency is the main

reason for bad performance. As for other graphs except for road-TX, our optimizations can reduce certain redundant operations and improve the work efficiency of the SSSP algorithm.

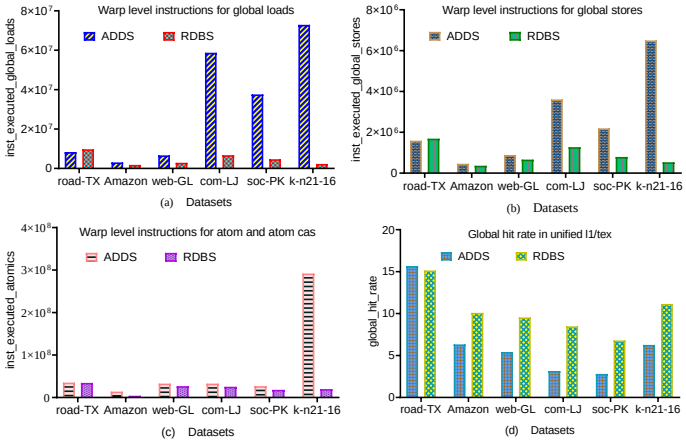

Figure 10: The profiling of RDBS and ADDS.

5.3.2 Memory Efficiency Analysis. For a deep analysis of memory efficiency and work efficiency, we use GPU profiling tools supported by NVIDIA to execute the SSSP algorithm on various graphs. We count and record the *inst_executed_global_loads* (*Warp* level instructions for global loads), *inst_executed_global_stores* (*Warp* level instructions for global stores), *inst_executed_atomics* (*Warp* level instructions for global atom and atom cas and *Warp* level shared instructions for atom and atom CAS) and *global_hit_rate* (Global Hit Rate in unified l1/tex), and the results are shown in Fig. 10. In Fig. 10 (a) and Fig. 10 (b), our algorithm's *Warp* level instructions for global loads and stores are 0.03× to 1.17× (0.41× on average), 0.082× to 1.06× (0.57× on average) against ADDS. The less *Warp* level instructions for global loads and stores, the lower memory access overhead. Meanwhile, by Fig. 10 (c) and Fig. 10 (d), we can observe that the atomic operations overhead of our algorithm reduce by 2.03% to 93.18% (39.61% on average) and the global hit rate enhances 3.59% on average compared with ADDS. Hence, our work can achieve both work efficiency and memory efficiency.

5.4 Algorithm Scalability

We evaluate the scalability of our SSSP algorithm using different graph scales and different GPU platforms.

5.4.1 Scalability with Various Graph Scales. To investigate the scalability of our work, we use synthetic graphs with different *SCALE* 22, 23, 24 and different *edgefactor* 8, 16, 32, 64 to evaluate the performance of our SSSP algorithm. Meanwhile, based on the aforementioned synthetic graphs, we compare our performance with ADDS. The results are shown in Fig. 11.

For synthetic graphs with *SCALE* 22, the performance of graphs with different *edgefactor* 8, 16, 32, 64 are 8.81, 16.78, 21.26, and 35.35 GTEPS. For synthetic graphs with *SCALE* 23, the performance of graphs with different *edgefactor* 8, 16, 32, 64 are 9.32, 20.60, 23.65, and 38.98 GTEPS. For synthetic graphs with *SCALE* 24, the performance of different graphs with *edgefactor* 8, 16, 32, 64 are 11.28, 20.16, 26.23, and 40.09 GTEPS. From the above performance results, we conclude that the higher the average degree, the better performance. Besides, with a fixed *edgefactor*, as the *SCALE* increases, the performance of our SSSP algorithm is better. For performance comparison with ADDS, we observe that with different *edgefactor* 8, 16, 32, 64, on *SCALE* 22 graphs, our performance is 13.53×, 22.93×, 27.97×, 45.35× faster than ADDS, on *SCALE* 23 graphs, our performance performs 14.82×, 31.62×, 34.86×, 58.21× better than ADDS, on *SCALE* 24 graphs, our performance is 18.45×, 33.09×, 40.87×, 68.65× faster than ADDS. The average speedup is 34.20×. Our SSSP algorithm shows good scalability as the graph scale increases.

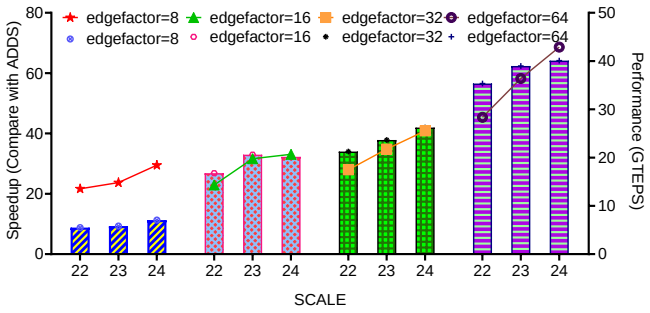

Figure 11: The speedup and performance with *edgefactor*.

In terms of performance, our SSSP algorithm gains more benefits on synthetic graphs than on real-world graphs. It achieves superior scalability and performance because reordering optimization and adaptive load balancing can work better with poor locality and irregularity of input graphs. The bucket-aware asynchronous execution further boosts the convergence of the SSSP algorithm.

5.4.2 Scalability on different GPU platforms. To further evaluate the scalability of our work on different GPUs, we use Tesla T4 to run our SSSP algorithm and compare the results with V100. The detailed results are shown in Fig. 12. The performance of SSSP on V100 outperforms that on Tesla T4, with the speedup of 2.14×, 1.47×, 2.30×, 2.35×, 2.58×, 1.51× on Amazon, road-TX, web-GL, com-LJ, soc-PK, k-n21-16 graphs.

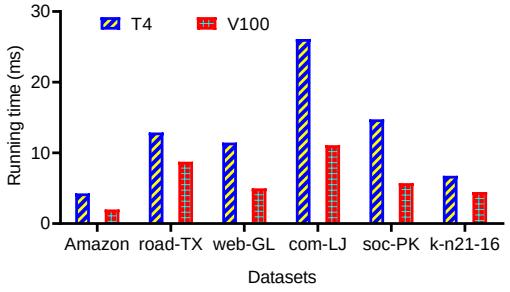

Figure 12: The running time of our SSSP on different GPUs.

Considering the software and hardware performance of these two GPUs, the performance of SSSP on T4 is also very good. Tesla

T4 has 40 SMs, with a total of 2560 CUDA cores. In terms of memory systems, the peak memory bandwidth of T4 is 320 GB/s. While V100 has 5120 CUDA cores and the memory bandwidth is 900 GB/s. Therefore, taking parallelism resources and memory bandwidth into consideration, our theoretical analysis suggests that the performance of SSSP on the V100 platform should be two to three times better than that on the Tesla T4 platform. And the experimental results are consistent with the theoretical analysis, which shows that our SSSP algorithm has substantial scalability on different GPU platforms.

6 RELATED WORK

In this section, we summarize the existing works of the SSSP algorithm based on CPU and GPU platforms.

6.1 SSSP on CPU

Ligra [31] is a simple framework for implementing graph traversal algorithms on shared-memory machines. It implements a high-performance Bellman-Ford algorithm and achieves 18.00× to 28.80× speedup on 40 cores. Julienne [12] uses work efficient bucketing skill to implement a parallel graph processing framework and Δ-stepping algorithm is implemented using the interface of this framework. To generalize existing algorithms including Δ-stepping and Raidus-stepping, MIT's researchers propose and implement an efficient stepping algorithm framework [15]. It uses a lazy-batched priority queue (LAB-PQ) to abstract the semantics of the priority queue used by the stepping algorithm. Besides, in this stepping framework, they implement Bellman-Ford, Δ*-stepping, and ρ-stepping. To alleviate the invalid operations, Zheng *et al.* [38] propose two novel filtration approaches to capture critical information for accurate and efficient vertex convergence judgment.

On the CPU platform, the optimizations of the SSSP algorithm mainly focus on improving algorithm parallelism and work efficiency, as well as implementing a graph processing framework. Considering the computing power of the CPU, we choose the GPU platform to implement the high-performance SSSP algorithm.

6.2 SSSP on GPU

Compared with the CPU, the powerful hardware and high memory bandwidth make GPU widely used in various applications. In graph processing, Gunrock presents a novel data-centric abstraction for graph processing and implements graph algorithms with several optimizations, such as kernel fusion, push-pull traversal, idempotent traversal, and priority queues [35]. SEP-Graph implements a highly efficient software framework for graph processing on GPU [33]. It automatically switches between Sync or Async, Push or Pull, and Data-driven or Topology-driven to achieve the shortest execution time. Wang *et al.* [34] present ADDS algorithm, a Δ-stepping algorithm for work efficient shortest path. The ADDS enhances memory efficiency and proposes to dynamically change Δ based on prior experiments. SSSP algorithm achieves good performance using GPU resources.

But the performance of the SSSP algorithm is sub-optimal. Gunrock suffers from slow convergence problems using synchronous mode. SEP ignores load balancing issues. Wang uses an asynchronous mode and changes Δ, which increases the difficulty of programming and ignores irregular memory access problems. Therefore, we focus on the GPU platform and solve the load imbalance, irregular memory access, and redundant operations problems to boost the performance of the SSSP algorithm.

7 CONCLUSION

The graph is a critical structure for many applications. SSSP algorithm is widely used in various domains. GPU provides powerful computing and high memory bandwidth. However, on the GPU platform, the poor locality and severe irregularity of input graphs, as well as the nondeterministic SSSP traversal bring a series of problems including irregular memory access, load imbalance, and redundant operations, resulting in bad performance of the SSSP algorithm. In this work, we implement a high-performance SSSP algorithm. Three optimizations are proposed to boost the performance of the SSSP algorithm, including property-driven reordering, adaptive load balancing, and bucket-aware asynchronous execution. We conduct plenty of experiments to evaluate our work, and the results show that our work outperforms the state-of-the-art SSSP implementations, including GPU-based and CPU-based, with an average speedup of 5.09× and 10.32×. Additionally, our work achieves good scalability with different graph scales and GPUs.

In the future, we will further explore a high-performance graph processing framework for large-scale graphs on the multi-GPUs platform.

ACKNOWLEDGMENTS

This work was supported by National Key Research and Development Program (Grant No. 2022YFB4501404), the Beijing Natural Science Foundation (4232036), CAS Project for Youth Innovation Promotion Association.

REFERENCES

[1] 2010. Graph500. http://www.graph500.org.
[2] Masab Ahmad, Halit Dogan, and Omer Khan. 2019. Speculative Task Parallel Algorithm for Single Source Shortest Path. (2019).
[3] Lars Backstrom, Dan Huttenlocher, Jon Kleinberg, and Xiangyang Lan. 2006. Group Formation in Large Social Networks: Membership, Growth, and Evolution. In *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (Philadelphia, PA, USA) *(KDD '06)*. Association for Computing Machinery, New York, NY, USA, 44–54. https://doi.org/10.1145/1150402.1150412
[4] Richard Bellman. 1958. On a routing problem. *Quarterly of applied mathematics* 16, 1 (1958), 87–90.
[5] Mariusz Bojarski, Davide Del Testa, Daniel Dworakowski, Bernhard Firner, Beat Flepp, Prasoon Goyal, Lawrence D Jackel, Mathew Monfort, Urs Muller, Jiakai Zhang, et al. 2016. End to end learning for self-driving cars. *arXiv preprint arXiv:1604.07316* (04 2016).
[6] Ajay Brahmakshatriya, Yunming Zhang, Changwan Hong, Shoaib Kamil, Julian Shun, and Saman Amarasinghe. 2021. Compiling Graph Applications for GPUs with Graphit. In *Proceedings of the 2021 IEEE/ACM International Symposium on Code Generation and Optimization* (Virtual Event, Republic of Korea) *(CGO '21)*. IEEE Press, 248–261. https://doi.org/10.1109/CGO51591.2021.9370321
[7] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. 2004. R-MAT: A recursive model for graph mining, In Proceedings of the 2004 SIAM International Conference on Data Mining. *SIAM Proceedings Series* 6, 442–446. https://doi.org/10.1137/1.9781611972740.43
[8] Marielle Christiansen, Kjetil Fagerholt, Bjørn Nygreen, and David Ronen. 2013. Ship routing and scheduling in the new millennium. *European Journal of Operational Research* 228, 3 (2013), 467–483.
[9] Vidushi Dadu, Sihao Liu, and Tony Nowatzki. 2021. PolyGraph: Exposing the Value of Flexibility for Graph Processing Accelerators. In *Proceedings of the 48th*

Annual International Symposium on Computer Architecture (Virtual Event, Spain) *(ISCA '21)*. IEEE Press, 595–608. https://doi.org/10.1109/ISCA52012.2021.00053

[10] Andrew Davidson, Sean Baxter, Michael Garland, and John Owens. 2014. Work-Efficient Parallel GPU Methods for Single-Source Shortest Paths. *Proceedings of the International Parallel and Distributed Processing Symposium, IPDPS*, 349–359. https://doi.org/10.1109/IPDPS.2014.45

[11] NVIDIA developer. 2017. *NVIDIA TESLA V100 GPU ARCHITECTURE*. Technical Report. NVIDIA.

[12] Laxman Dhulipala, Guy Blelloch, and Julian Shun. 2017. Julienne: A Framework for Parallel Graph Algorithms Using Work-Efficient Bucketing. In *Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures* (Washington, DC, USA) *(SPAA '17)*. Association for Computing Machinery, New York, NY, USA, 293–304. https://doi.org/10.1145/3087556.3087580

[13] Edsger W Dijkstra et al. 1959. A note on two problems in connexion with graphs. *Numerische mathematik* 1, 1 (1959), 269–271.

[14] Rongyu Dong, Huawei Cao, Xiaochun Ye, Yuan Zhang, Qinfen Hao, and Dongrui Fan. 2020. Highly Efficient and GPU-Friendly Implementation of BFS on Single-node System. In *2020 IEEE Intl Conf on Parallel & Distributed Processing with Applications, Big Data & Cloud Computing, Sustainable Computing & Communications, Social Computing & Networking (ISPA/BDCloud/SocialCom/SustainCom)*. IEEE, 544–553.

[15] Xiaojun Dong, Yan Gu, Yihan Sun, and Yunming Zhang. 2021. Efficient Stepping Algorithms and Implementations for Parallel Shortest Paths. In *Proceedings of the 33rd ACM Symposium on Parallelism in Algorithms and Architectures* (Virtual Event, USA) *(SPAA '21)*. Association for Computing Machinery, New York, NY, USA, 184–197. https://doi.org/10.1145/3409964.3461782

[16] Izzat El Hajj, Juan Gómez-Luna, Cheng Li, Li-Wen Chang, Dejan Milojicic, and Wen-mei Hwu. 2016. KLAP: Kernel Launch Aggregation and Promotion for Optimizing Dynamic Parallelism. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture* (Taipei, Taiwan) *(MICRO-49)*. IEEE Press, Article 13, 12 pages.

[17] Pawan Harish and Petter J Narayanan. 2007. Accelerating large graph algorithms on the GPU using CUDA. *High Performance Computing-HiPC 2007* 4873, 197–208. https://doi.org/10.1007/978-3-540-77220-0_21

[18] Jure Leskovec, Lada A. Adamic, and Bernardo A. Huberman. 2007. The Dynamics of Viral Marketing. *ACM Trans. Web* 1, 1 (may 2007), 5–es. https://doi.org/10.1145/1232722.1232727

[19] Jure Leskovec, Daniel Huttenlocher, and Jon Kleinberg. 2010. Predicting Positive and Negative Links in Online Social Networks. In *Proceedings of the 19th International Conference on World Wide Web* (Raleigh, North Carolina, USA) *(WWW '10)*. Association for Computing Machinery, New York, NY, USA, 641–650. https://doi.org/10.1145/1772690.1772756

[20] Jure Leskovec, Daniel Huttenlocher, and Jon Kleinberg. 2010. Signed Networks in Social Media. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (Atlanta, Georgia, USA) *(CHI '10)*. Association for Computing Machinery, New York, NY, USA, 1361–1370. https://doi.org/10.1145/1753326.1753532

[21] Jure Leskovec, Jon Kleinberg, and Christos Faloutsos. 2005. Graphs over Time: Densification Laws, Shrinking Diameters and Possible Explanations. In *Proceedings of the Eleventh ACM SIGKDD International Conference on Knowledge Discovery in Data Mining* (Chicago, Illinois, USA) *(KDD '05)*. Association for Computing Machinery, New York, NY, USA, 177–187. https://doi.org/10.1145/1081870.1081893

[22] Jure Leskovec, Kevin J Lang, Anirban Dasgupta, and Michael W Mahoney. 2009. Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. *Internet Mathematics* 6, 1 (2009), 29–123.

[23] Fredrik Liljeros, Christofer R Edling, Luis A Nunes Amaral, H Eugene Stanley, and Yvonne Åberg. 2001. The web of human sexual contacts. *Nature* 411, 6840 (2001), 907–908.

[24] Hang Liu and H. Howie Huang. 2015. Enterprise: Breadth-First Graph Traversal on GPUs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Austin, Texas) *(SC '15)*. Association for Computing Machinery, New York, NY, USA, Article 68, 12 pages. https://doi.org/10.1145/2807591.2807594

[25] Hang Liu and H. Howie Huang. 2019. SIMD-X: Programming and Processing of Graph Algorithms on GPUs. In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference* (Renton, WA, USA) *(USENIX ATC '19)*. USENIX Association, USA, 411–427.

[26] Saeed Maleki, Donald Nguyen, Andrew Lenharth, María Garzarán, David Padua, and Keshav Pingali. 2016. DSMR: A Parallel Algorithm for Single-Source Shortest Path Problem. In *Proceedings of the 2016 International Conference on Supercomputing* (Istanbul, Turkey) *(ICS '16)*. Association for Computing Machinery, New York, NY, USA, Article 32, 14 pages. https://doi.org/10.1145/2925426.2926287

[27] Ulrich Meyer and Peter Sanders. 1998. Delta-Stepping: A Parallel Single Source Shortest Path Algorithm. In *Proceedings of the 6th Annual European Symposium on Algorithms (ESA '98)*. Springer-Verlag, Berlin, Heidelberg, 393–404.

[28] Amir Hossein Nodehi Sabet, Junqiao Qiu, and Zhijia Zhao. 2018. Tigr: Transforming Irregular Graphs for GPU-Friendly Graph Processing. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems* (Williamsburg, VA, USA) *(ASPLOS '18)*. Association for Computing Machinery, New York, NY, USA, 622–636. https://doi.org/10.1145/3173162.3173180

[29] Mhd Ghaith Olabi, Juan Gómez Luna, Onur Mutlu, Wen-mei Hwu, and Izzat El Hajj. 2022. A Compiler Framework for Optimizing Dynamic Parallelism on GPUs. In *Proceedings of the 20th IEEE/ACM International Symposium on Code Generation and Optimization* (Virtual Event, Republic of Korea) *(CGO '22)*. IEEE Press, 1–13. https://doi.org/10.1109/CGO53902.2022.9741284

[30] Ryan A. Rossi and Nesreen K. Ahmed. 2015. The Network Data Repository with Interactive Graph Analytics and Visualization. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence* (Austin, Texas) *(AAAI'15)*. AAAI Press, 4292–4293.

[31] Julian Shun and Guy E. Blelloch. 2013. Ligra: A Lightweight Graph Processing Framework for Shared Memory. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Shenzhen, China) *(PPoPP '13)*. Association for Computing Machinery, New York, NY, USA, 135–146. https://doi.org/10.1145/2442516.2442530

[32] L. Takac and Michal Zábovský. 2012. Data analysis in public social networks. *International Scientific Conference and International Workshop Present Day Trends of Innovations* (01 2012), 1–6.

[33] Hao Wang, Liang Geng, Rubao Lee, Kaixi Hou, Yanfeng Zhang, and Xiaodong Zhang. 2019. SEP-Graph: Finding Shortest Execution Paths for Graph Processing under a Hybrid Framework on GPU. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming* (Washington, District of Columbia) *(PPoPP '19)*. Association for Computing Machinery, New York, NY, USA, 38–52. https://doi.org/10.1145/3293883.3295733

[34] Kai Wang, Don Fussell, and Calvin Lin. 2021. A Fast Work-Efficient SSSP Algorithm for GPUs. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Virtual Event, Republic of Korea) *(PPoPP '21)*. Association for Computing Machinery, New York, NY, USA, 133–146. https://doi.org/10.1145/3437801.3441605

[35] Yangzihao Wang, Yuechao Pan, Andrew Davidson, Yuduo Wu, Carl Yang, Leyuan Wang, Muhammad Osama, Chenshan Yuan, Weitang Liu, Andy T. Riffel, and John D. Owens. 2017. Gunrock: GPU Graph Analytics. *ACM Trans. Parallel Comput.* 4, 1, Article 3 (aug 2017), 49 pages. https://doi.org/10.1145/3108140

[36] Jaewon Yang and Jure Leskovec. 2012. Defining and Evaluating Network Communities Based on Ground-Truth. In *Proceedings of the ACM SIGKDD Workshop on Mining Data Semantics* (Beijing, China) *(MDS '12)*. Association for Computing Machinery, New York, NY, USA, Article 3, 8 pages. https://doi.org/10.1145/2350190.2350193

[37] Yu Zhang, Da Peng, Xiaofei Liao, Hai Jin, Haikun Liu, Lin Gu, and Bingsheng He. 2021. LargeGraph: An efficient dependency-aware GPU-accelerated large-scale graph processing. *ACM Transactions on Architecture and Code Optimization (TACO)* 18, 4 (2021), 1–24.

[38] Long Zheng, Xianliang Li, Xi Ge, Xiaofei Liao, Zhiyuan Shao, Hai Jin, and Qiang-Sheng Hua. 2021. Efficient Graph Processing with Invalid Update Filtration. *IEEE Transactions on Big Data* 7, 3 (2021), 590–602. https://doi.org/10.1109/TBDATA.2019.2921358

[39] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. 2016. Gemini: A Computation-Centric Distributed Graph Processing System. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (Savannah, GA, USA) *(OSDI'16)*. USENIX Association, USA, 301–316.

Received 22 April 2023; revised 22 June 2023; accepted 26 June 2023