CCF Transactions on High Performance Computing (2023) 5:277–291
https://doi.org/10.1007/s42514-023-00155-x

REGULAR PAPER

FSGraph: fast and scalable implementation of graph traversal on GPUs

Yuan Zhang[1,2] · Huawei Cao[1,3] · Yan Liang[1,2] · Jie Zhang[1,2] · Junying Huang[1] · Xiaochun Ye[1] · Xuejun An[1,2]

Received: 11 March 2023 / Accepted: 15 May 2023 / Published online: 31 May 2023

Abstract
Graph is one of the best ways to express and process association relationship. It is widely used in various applications, including social networks, fraud detection, Internet of things, etc. As a typical graph traversal algorithm, the Breadth-First Search (BFS) performance on GPU is not desirable, due to strong data dependency, intensive irregular memory access and low computation intensity. On GPUs, the situation is even worse with unbalanced data partitioning and high communication-to-computation ratios. In this paper, we implement FSGraph that is a fast and scalable BFS implementation on GPUs. In FSGraph, we propose three optimizing techniques: GPU-friendly Compressed Sparse Row (CSR) structure, bidirectional one-dimensional (1d) partition and UM-aware communication. We have evaluated our work with extensive experiments on four T4 and four V100 GPUs. The average performance of BFS on four T4 GPUs is 132.67 Giga-Traversed Edges per Second (GTEPS), which delivers up to 1.44× improvement than that on single T4. In terms of four V100 GPUs, the BFS performance achieves 392.35 GTEPS and outperforms existing CPU-based cluster with 1024 nodes on November 2022 Graph500 list.

Keywords BFS · GPU-friendly CSR structure · Bidirectional 1d partition · UM-aware communication

1 Introduction

With the development of information society, people are facing the need to process the continuously generated data. To facilitate our lives, graph analytic becomes a new exploration in the wave of big data analysis (Bader and Madduri 2008). In reality, it is a common method to abstract and describe problems with graph. Graph is one of the most important data structures, which is widely used in various applications, such as social networks, fraud detection, Internet of things and so on (Pham et al. 2015; Li et al. 2021; Mislove et al. 2007; Ting et al. 2013).

As a typical graph algorithm, the Breadth-First Search (BFS) is the core component of high-level graph analysis algorithms, including Betweenness Centrality (BC), Connected Component (CC) and Single-Source Shortest Path (SSSP) (Murphy et al. 2010). Different from the compute-intensive workload, the typical characteristics of BFS, classified as data-intensive application, are strong data dependency, intensive irregular memory access and low computation intensity. Towards data-intensive applications, the Graph500 list [7] was introduced to rank computer performance. In the benchmark of Graph500, the BFS is one of the key kernels (Murphy et al. 2010; Ueno and Suzumura 2012).

Designed for high throughput, Graphics Processing Unit (GPU) provides high memory bandwidth and massive parallelism towards compute-intensive workloads. It is widely used in big data, deep learning and high-performance computing. With the development of graph computing, the acceleration of graph traversal on GPU is gradually becoming a research trend. However, the irregular characteristic of graph traversal and the explosive growth of graph scale hinder the high performance gain on GPU platform (Sabet et al. 2020; Dong et al. 2020). The core problems are intensive random access, workload imbalance, high communication-to-computation ratios, and limited communication bandwidth. What's worse, the problems are further deteriorated for scale-free graphs, which follow power-law distribution (Faloutsos et al. 2011). The topology of scale-free graph results in workload imbalance.

✉ Huawei Cao
caohuawei@ict.ac.cn

[1] Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100190, China

[2] University of Chinese Academy of Sciences, Beijing 100049, China

[3] University of Chinese Academy of Sciences, Nanjing 211135, China

Memory divergence creates additional challenges in BFS processing. Because of the discontinuous memory access within a warp, a large amount of load and store transactions are caused during the traversal. Besides, for large graph, unbalanced data partitioning and communication overhead are extra challenges for graph traversal on GPUs.

Considering above problems, we propose several optimizing schemes on data representation, graph partitioning and communication optimization among GPUs. Based on these optimizations, we carry out plenty of experiments and evaluate the performance of BFS implementation on multiple T4 and V100 GPUs, compare and analyze the results with some existing works to show the effective performance improvement of our work. Specifically, the main contributions are as follows:

1. In order to improve the data locality of graph traversal, GPU-friendly graph representation includes *bitmap adaptive CSR* and *adjacency list* reorganization is proposed, which improves the memory divergence of graph traversal on GPUs.

2. Due to the scale-free graph topology, there exists heavy communication overhead and workload imbalance. We develop a bidirectional 1d partition scheme, and combine it with a static shuffle method to reduce the workload unbalance and communication overhead.

3. In terms of data communication overhead among GPUs, this paper systematically evaluates the performance and scalability of a series of efficient communication libraries. Finally, we propose UM-aware communication in our implementation, which can reduce the communication overhead and increase the scalability of graph traversal.

4. We systematically evaluate our optimizations with extensive experiments on NVIDIA Tesla T4 and V100. The results show better performance and scalability on GPUs than existing GPU-based systems.

The rest of this paper is organized as follows: Sect. 2 and Sect. 3 introduce the background and discuss the challenge of graph traversal on GPUs. Section 4 presents our optimizations in FSGraph. The experiment and evaluation of our work are in Sect. 5. Then the related work is clarified in Sect. 6. Finally, we conclude the FSGraph in Sect. 7.

2 Background

BFS is a widely used graph algorithm and the fundamental building block of many graph analysis algorithms. To boost the performance of BFS, there has been a lot of work on parallel implementation of BFS on GPUs. In this section, we will present some state-of-art optimizations for BFS implementation.

2.1 CSR format

Graph is usually presented in CSR format, which can reduce the memory footprint and increase the streaming access to neighboring edges. With CSR format, the graph data can be compressed to row storage of *adjacency matrix*. Figure 1 shows the basic idea of CSR. There are two arrays named *row list* and *adjacency list*. For each vertex, all the neighboring vertices are stored in *adjacency list*. The size of *adjacency list* is equal to the total number of edges in the graph. The offset of each vertex's first neighbor in *adjacency list* is stored in *row list*, and the adjacent values in *row list* are the degree of each vertex. Taking Fig. 1 as an example, the degree of vertex 0 is 3, and the neighbors of vertex 0 are 1, 2, 3.

2.2 Top-down BFS algorithm

Algorithm 1 Top-Down BFS

Input: undirected Graph $G = (V, E)$, cur: current queue, $next$: next queue, adj: adjacency list, s: initial root vertex.
Output: $parent$: an parent array of all vertices.

```
1:  parent[v] = ∞, v ∈ V
2:  parent[s] = s
3:  cur = {s}
4:  next = {}
5:  while cur ≠ {} do
6:      for u ∈ cur do
7:          for w ∈ adj(u) do
8:              if parent[w] == ∞ then
9:                  parent[w] = u
10:                 next = next ∪ {w}
11:             end if
12:         end for
13:     end for
14:     cur = next
15:     next = {}
16: end while
```

Traditionally, BFS is performed in top-down manner. Given a graph $G=(V, E)$ and the initial root vertex s, $V=\{v_1, v_2, v_3, \ldots\}$, $E=\{(v_1, v_2), (v_1, v_5), (v_2, v_4), \ldots\}$, V is vertex set and E is edge set. The top-down BFS will visit all reachable vertices from s, then generate the final BFS spanning tree. The pseudo-code of top-down BFS is shown in Algorithm 1. At the beginning, all data structures are initialized, a random root vertex s is generated and put into *current queue*. When *current queue* is not empty, all neighbors of each vertex u

in *current queue* will be traversed. The traversal procedure is to mark the status of unvisited neighbors as visited, map their parent vertices and put these vertices into *next queue*. When all the vertices in *current queue* are processed, then the *current queue* is swapped with *next queue* and *next queue* is cleared. In this way, top-down BFS can generate the BFS searching tree at the end of this algorithm.

2.3 Bottom-up BFS algorithm

When *current queue* is small, the top-down BFS approach is very efficient. After several iterations, the size of *current queue* increases rapidly, and many vertices in *adjacency list* have been visited. Under the circumstances, if we continue to traverse the graph in top-down manner, it will lead to massive redundant edges traversal. Besides, there exist plenty of atomic operations to get global *next queue* for parallel BFS implementation on GPUs, leading to extra heavy computational overhead.

Algorithm 2 Bottom-Up BFS

Input: undirected Graph $G = (V, E)$, cur: current queue, $next$: next queue, adj: adjacency list, s: initial root vertex.
Output: $parent$: an parent array of all vertices.

```
1:  parent[v] = ∞, v ∈ V
2:  parent[s] = s
3:  cur = {s}
4:  next = {}
5:  while cur ≠ {} do
6:      for w ∈ V & parent[w] == ∞ do
7:          for u ∈ adj(w) do
8:              if u ∈ cur then
9:                  parent[w] = u
10:                 next = next ∪ {w}
11:                 break
12:             end if
13:         end for
14:     end for
15:     cur = next
16:     next = {}
17: end while
```

In order to solve the bottleneck of top-down BFS approach, Beamer proposed an effective bottom-up BFS algorithm (Beamer et al. 2012). Compared with top-down approach, bottom-up approach works in the opposite way. When the size of *current queue* is large, it starts from all the unvisited vertices and checks whether their parent vertices are in *current queue*. Using bottom-up approach, as the *current queue* increases, the iteration of graph traversal will finish earlier. Thus, bottom-up approach can significantly reduce the redundant traversal of edges. The detailed pseudo-code of this algorithm is shown in Algorithm 2.

Besides, in order to find the parent of vertex in the *current queue* quickly, Yasui and Fujisawa (2015) conducted a statistical analysis depending on the degree and the frequency of vertex access in the graph. He found that the vertices with high degree had a high probability of being the parent for an unvisited vertex. Thus, the degree-aware optimization was proposed to speed up the graph traversal during bottom-up stage. This optimization could further reduce the redundant edges checking in bottom-up BFS algorithm and enhance the performance of the BFS algorithm.

However, bottom-up approach also has its drawbacks. When there are a few vertices in *current queue*, most vertices will not be expanded at this iteration, causing plenty of redundant edges checking in the parent checking. So, the bottom-up BFS is advantageous when the size of *current queue* is large, while top-down BFS is efficient when the size is small. Top-down and bottom-up approaches are complementary in graph traversal. In Beamer's direction-optimizing approach (Beamer et al. 2012), BFS algorithm started to run in top-down approach and switched to bottom-up approach when the size of *current queue* was large enough. In the last several iterations, BFS switched back to top-down approach when the size of the *current queue* became small. Under appropriate switching policy, BFS performance improves a lot.

2.4 Graph partitioning scheme

For distributed BFS implementation, the partition of graph data can usually be categorized into one dimensional (1d) partition, two dimensional (2d) partition and even three dimensional (3d) partition, based on different vertex

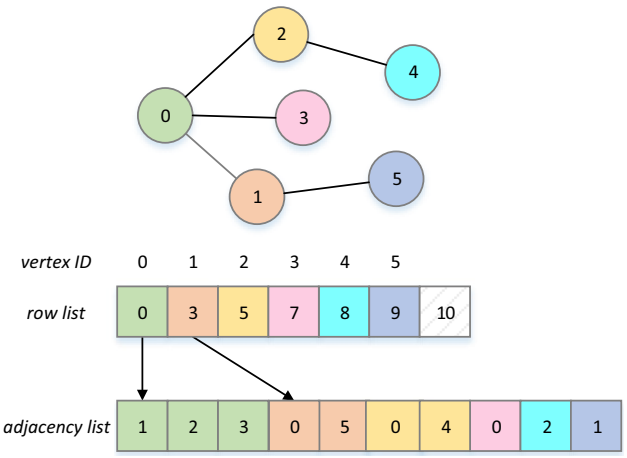

Fig. 1 Illustration of CSR format

mapping and data partitioning methods (Buluç and Madduri 2011; Yoo et al. 2005; Checconi et al. 2012). 1d partition mainly divides graph data according to the indices of vertices. The vertex information and its adjacent edges are placed on the same computing node. 1d partition has the characteristics of simple thinking and easy implementation. It works well on evenly distributed graphs. However, due to the power-law distribution of most graph (Faloutsos et al. 2011), 1d partition can induce serious workload imbalance among distributed computing nodes. The 2d partition divides the *adjacency matrix* of graph on the basis of block method. 2d partition can split the collective communications from one dimension to two dimensions, which reduces the communication overhead. The detailed 1d and 2d partition schemes are shown in Fig. 2 (Note: the original graph is shown in Fig. 1). 3d partition is realized on the basis of the 2d partition and can further reduce the communication overhead compared to 2d partition. However, this method is relatively complicated and changes the computation and communication models, which is not effective for large scale parallel BFS implementation.

2.5 Data communication on GPUs

There are several methods of data communication for the implementation of graph traversal on GPUs. Originally, NVIDIA provided a general library named Compute Unified Device Architecture (CUDA). By using the cudaMemcpy function provided by CUDA, researchers and developers can simply transfer data among GPUs and CPU. However, this method requires frequent data copy. Under the limitation of PCIe bandwidth, the communication overhead among GPUs and CPU for graph traversal are high. To facilitate and simplify data exchange between GPU and GPU, NVIDIA provides Peer to Peer (P2P) communication interface, which can eliminate two copies of data from GPU to GPU. Some works use P2P to implement data communication on multi-GPUs. Although this method enhances the performance of graph traversal to some degree, it has a low scalability for graph traversal with the number of GPUs increasing.

In 2017, Potluri et al. put forward using NVSHMEM library functions for efficient BFS on multi-GPUs systems (Potluri et al. 2017). On GPUs, NVSHMEM-based data communication is mainly relied on the storage model of OpenSHMEM, which is a single-program multi-data programming model and provides high performance and good scalability. It is a fine-grained inter-GPU communication, which can realize data synchronization in the kernel function. Although the library is advantageous over the above two methods, it is difficult to program, and the performance degrades significantly as the data communication increases. Besides, for GPU clusters, some researches focus on using Message Passing Interface (MPI) for data communication on GPUs.

3 Challenges of BFS on GPUs

In this section, we motivate our approach by identifying challenges for parallel BFS implementation on GPUs. With the explosive growth of graph data, graph processing on a single GPU can no longer achieve high performance due to insufficient GPU global memory. The BFS performance has a sharp decline when the size of graph data exceeds the memory footprint of GPU. Multi-GPUs system has large potential in exploiting the scalability of large-scale graph traversal. However, due to the irregular characteristics of BFS traversal, causing severe memory divergence, load imbalance and high communication overhead, it's difficult for efficient implementation of BFS on GPUs.

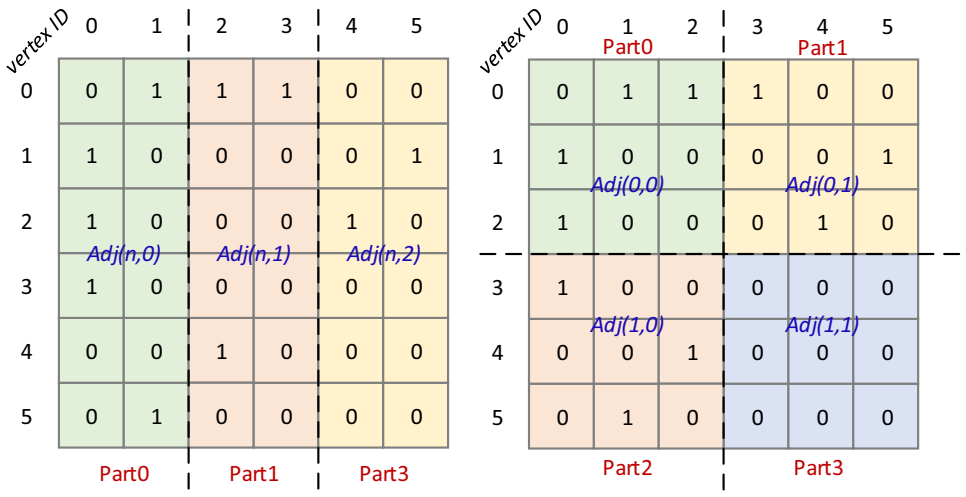

Fig. 2 The illustration of 1d partition (left) and 2d partition (right)

3.1 Memory divergence

As we all know, GPU can provide massive parallelism and high bandwidth. Generally, high bandwidth utilization of GPU is achieved by regular and sequential memory access. However, due to the power-law distribution, the social and web networks graphs are highly irregular distributed. From the work of Khorasani, the BFS implementation on GPU only achieves 12.8%~15.8% memory bandwidth utilization (Khorasani et al. 2014). The irregular memory access within the same warp can cause severe memory divergence, leading to a large amount of load and store transactions, and high latency.

Memory coalescing is always a way to improve memory efficiency on GPU. In this work, we focus on the improvement of graph data layout by taking the benefit of memory coalescing technique. Original CSR format is not GPU-friendly in bottom-up phase. To make BFS more GPU-aware, we propose an improved graph data structure to achieve high memory efficiency.

3.2 Workload imbalance

For parallel BFS implementation on distributed GPUs, the workload is usually divided with 1d partition based on vertex indices, and distributed evenly to available GPU nodes. Due to the power-law nature of scale-free graphs, the degree of vertex varies significantly, leading to serious workload imbalance in graph traversal. Although GPU provides strong computing power, the running time will be dominated by the vertices with high degrees.

What's worse, the above 1d partition divides neighboring edges of one vertex into different GPUs, which will induce frequent communication and synchronization during the graph traversal. To improve the communication efficiency and workload balance, we develop a bidirectional 1d partition method, and combine it with static shuffle scheme proposed in our previous work (Zhang et al. 2019).

3.3 Communication overhead

For parallel BFS implementation on GPUs, there exists frequent data communication and synchronization among host CPU and GPUs. With the increase of the graph scale, the communication overhead explodes a lot. However, due to the limited PCIe bandwidth, which is only 32 GB/s in the ideal case of PCIe Gen 4.0 x16, while the memory bandwidth of T4 is 320 GB/s [20], the data transmission among CPU and GPUs is the main bottleneck of the performance.

To reduce the communication overhead among host CPU and GPU, we propose a UM-aware communication scheme, which combines the efficient communication library and Unified Memory (UM). UM can provide unified memory address for CPU and GPU, which further breaks the barriers of PCIe performance.

4 Algorithm optimization

In this section, the detailed optimizations for BFS implementation on multi-GPUs will be illustrated. We propose three techniques regarding the challenges of BFS parallelism. These GPU-friendly optimizations are used to improve memory access, balance workload, and reduce communication overhead.

4.1 GPU-friendly CSR structure

It is well-known that intensive irregular memory access and low memory bandwidth utilization are characteristics of BFS. Those characteristics prevent efficient BFS acceleration on GPU. The main reason is that most graph data reside in GPU global memory and only a small amount of data can be cached in cache or shared memory. Thus, it is critical for BFS to achieve efficient data access in global memory. We need to relieve irregular memory access and improve global memory bandwidth utilization to facilitate BFS performance on GPU.

To enhance the efficiency of memory access on GPU, memory coalescing technique is put forward in this paper. Under the SIMT model, all threads in a warp execute the same instruction. If the memory access of all threads within a warp could be coalesced to consecutive addresses, all these memory operations can be combined into one single access transaction, which drastically decreases the latency and increases memory utilization. Different from prior works (Busato and Bombieri 2014; Luo et al. 2010; Zhong et al. 2016), we pay more attention to the improvement of graph data structure and construct a GPU-friendly CSR structure, which follows the rule of memory coalescing in bottom-up phase. Compared with original CSR, the GPU-friendly CSR structure is constructed under two optimizing techniques. One is the *bitmap adaptive CSR* and the other is the *warp-aligned adjacency list*.

4.1.1 Bitmap adaptive CSR

Firstly, the idea of *bitmap adaptive CSR* is introduced. To reduce the memory access in bottom-up stage, bitmap optimizing technique is used. For parallel BFS implementation (Agarwal et al. 2010), each thread deals with one unit of bitmap each time. The size of bitmap unit is usually 4 bytes. Combined with vertex sorting technique, good locality is offered by bitmap. When we use the technique of bitmap

optimizing on GPU, the threads in the same warp will face memory divergence problem in accessing *index mapping lists*, including *row list* and some other auxiliary structures. If the vertex indices in bitmap are remapped, the locality of the bitmap will be broken. Thus, to access *index mapping lists* efficiently, we propose *bitmap adaptive CSR*, which does not break the benefit of bitmap.

The main idea of the *bitmap adaptive CSR* is like the remix of original CSR structure. The position of vertex in the bitmap is kept as before. But in the *index mapping lists*, like the *row list*, the corresponding position of the vertex is remapped to achieve the memory coalescing. In this way, not only we can keep the locality of bitmap, but also we can improve memory access efficiency at the same time. All threads in the same warp deal with the same bit in their own bitmap unit every time. To introduce the procedure clearly, we define *b_size* as bitmap unit size and *w_size* as warp size. Generally, the *w_size* is 32. As can be seen in Fig. 3, *thread_0* (t_0) deals with vertex *b_size*0* (the 1^{st} bit of *bitmap* unit 0), *thread_1* (t_1) deal with vertex *b_size*1* (the 1^{st} bit of bitmap unit 1),..., *thread_31* (t_{31}) deal with vertex *b_size*31* (the 1^{st} bitmap of unit 31). Different from prior CSR, the vertices with ID *b_size*0*, *b_size*1*, *b_size*2*,..., *b_size*31* are put together in the *bitmap adaptive CSR*. The memory access to the *row list* in the warp can be coalesced into one transaction, which is suitable for SIMT execution. All following vertices that belong to the same bitmap unit are remixed in the same way. However, in *adjacency list*, the values are not changed. The main reason is that the technique does not reorder the vertex ID. All vertices are kept with the step size of *b_size*w_size* in the same way.

Figure 3 shows the structure of *bitmap adaptive CSR*. All *index mapping lists* are rearranged. Here, we take the *row list* as an example. The value in bitmap and *row list* is the mapped vertex ID. The locations of all vertices in bitmap keep the same, while their locations in *row list* are reordered for memory coalescing. Under SIMT execution model, all threads in a wrap can access consecutive memory in the *row list*. Equation(1) and Eq.(2) show the mapping policy between original vertex ID and rearranged location in *index mapping lists*.

$$\begin{aligned} id\_to\_loc(id) = & \frac{id}{b\_size*w\_size} * (b\_size * w\_size) \\ & + (id \bmod b\_size) * w\_size \\ & + \frac{id \bmod (b\_size*w\_size)}{b\_size} \\ & - \frac{id \bmod b\_size}{b\_size} \end{aligned} \tag{1}$$

$$\begin{aligned} loc\_to\_id(loc) = & \frac{loc}{b\_size*w\_size} * (b\_size * w\_size) \\ & + (loc \bmod w\_size) * b\_size \\ & + \frac{loc \bmod (b\_size*w\_size)}{w\_size} \end{aligned} \tag{2}$$

4.1.2 Warp-aligned adjacency list

To improve the global memory access efficiency in edge traversal procedure, the *warp-aligned adjacency list* is used. As described above, although memory coalescing is achieved in the *index mapping lists*, using original *adjacency list* still leads to severe memory divergence problem. Because all threads in the same warp execute edge traversal procedure of different vertices at the same time. To realize the coalesced memory access in *adjacency list*, we need to keep the adjacent threads in the same warp addressing the neighboring cells in global memory. Thus, the *warp-aligned adjacency list* tries to reassemble the neighbors of the vertices processed in the same warp to adhere to the memory coalescing rule.

Taking Fig. 4 as an example, adjacent *w_size* vertices will be processed by the same warp in SIMT execution model, so all vertices are formed into groups in units of *w_size*. The *warp-aligned adjacency list* is derived from original *adjacency list*. In Fig. 4, the 1st neighbor of vertex 0 are rearranged with other 1st neighboring vertices which will be processed in the

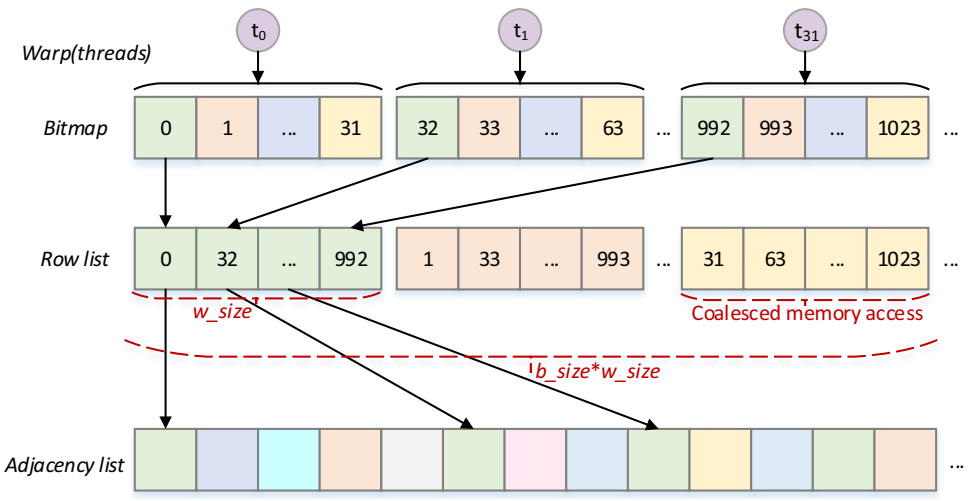

Fig. 3 The bitmap adaptive CSR structure

same warp. All following neighboring vertices belonging to the same group are remixed in the same way. Due to varied degree of vertices, the minus sign (-) is extra padding space for neighboring alignment. If we mix all neighbors of each vertex, the padding cost will occupy huge memory footprint. There is a discrepancy between memory access efficiency and memory footprint. Using degree-aware (Yasui and Fujisawa 2015) optimization, the traversal procedure based on bottom-up approach stops in a small number of neighboring vertices. As a result, the *warp-aligned adjacency list* is constructed by choosing a certain number of adjacent neighbors. Taking memory efficiency and padding cost into consideration, in our implementation, the *warp-aligned adjacency list* only includes the first 30% edges of original *adjacency list*. Based on our experiment, no more than 7% memory overhead is brought by this new structure. Using this technique, the memory accesses are coalesced in a warp, which highly enhances the memory efficiency.

4.2 Bidirectional 1d partition

As described in Sect. 2.4, traditional 1d partition divides graph data based on the indices of vertices. The information of vertex and its adjacent edges are placed on the same compute node. However, the irregularity of graph data restricts efficient BFS implementation on GPUs and can cause severe workload imbalance and communication overhead.

In order to reduce the communication overhead among GPUs, we propose a bidirectional 1d partition. The corresponding division is shown in Fig. 5. The left figure is column direction (column-based) 1d partition and the right figure is row direction (row-based) 1d partition.

Let $G = (V, E)$, V represents vertex set, a total of n vertices, and E represents edge set, a total of e edges. Assuming that the number of divisions in row is r, the number of divisions in column is c, the corresponding information based on row and column 1d partition are as follows:

For row-based 1d partition, the V and E are divided by:

$$V = [V_{r0}|V_{r1}|V_{r2}|...|V_{r(r-1)}] \tag{3}$$

$$E = [E_{r0}|E_{r1}|E_{r2}|...|E_{r(r-1)}] \tag{4}$$

In Eq.(3) and Eq.(4), each set of partial vertices V_{rk} and E_{rk} on the *k-th* computing node is defined by:

$$V_{rk} = \left\{ v_k \in V | k \in \left[\frac{kn}{r}, \frac{(k+1)n}{r} \right] \right\} \tag{5}$$

$$E_{rk}(v) = \{ e_{(v,u)} \in E_{out}(v) | v \in V_{rk}, u \in V \} \tag{6}$$

The $E_{out}(v)$ in Eq.(6) is the set of outgoing edges of each vertex v in the vertex set V_{rk}.

For column-based 1d partition, the V and E are divided by:

$$V = [V_{c0}|V_{c1}|V_{c2}|...|V_{c(c-1)}] \tag{7}$$

$$E = [E_{c0}|E_{c1}|E_{c2}|...|E_{c(c-1)}] \tag{8}$$

In Eq.(7) and Eq.(8), each set of partial vertices V_{ck} and E_{ck} on the *k-th* computing node is defined by:

$$V_{ck} = \{ v_k \in V \} \tag{9}$$

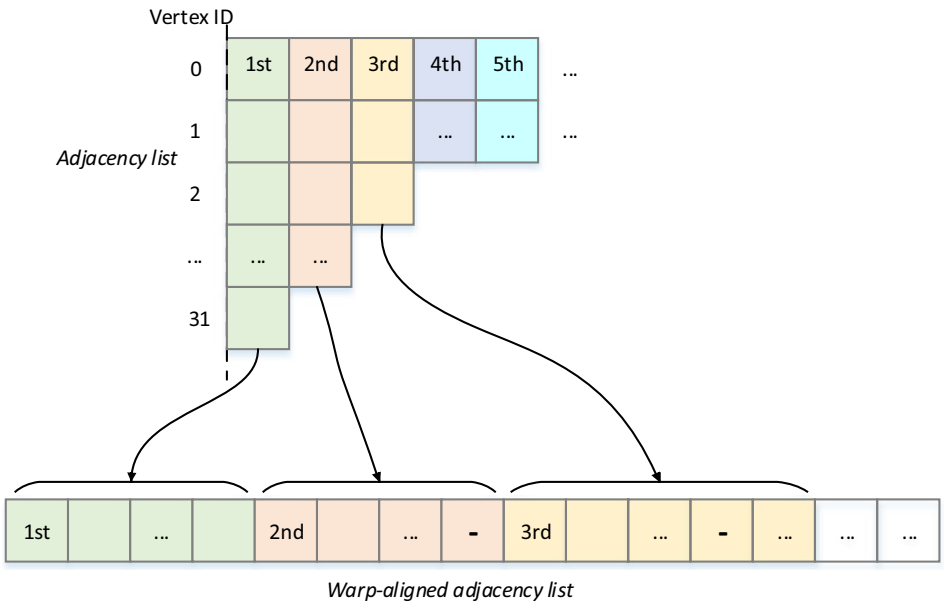

Fig. 4 The data structure of warp-aligned *adjacency list*

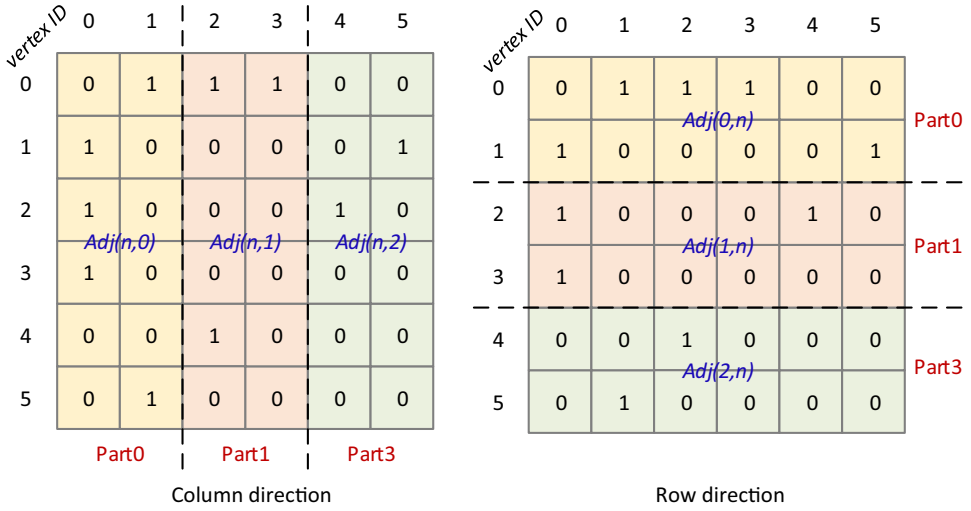

Fig. 5 The bidirectional 1d partition

$$E_{ck}(v) = \{e_{(u,v)} \in E_{in}(v) | u \in \left[\frac{kn}{c}, \frac{(k+1)n}{c}\right], v \in V_{ck}\} \quad (10)$$

The $E_{in}(v)$ in Eq.(10) is the set of ingoing edges of each vertex v in the vertex set V_{ck}.

The column direction partition divides graph data by edges. It will separate the adjacent edges of high-degree vertices into different GPU nodes. In top-down stage, column direction partition is more suitable, which can ensure initial workload balance on each GPU after partitioning. Because we assign an approximation number of edges for each GPU. However, in bottom-up stage, the graph traversal starts from vertices that have not been visited, and it will check whether the parent vertex locates in the *current queue*. If we use column-based 1d partition, the neighboring edges of the vertex will be divided into different GPU nodes, which induces frequent communication during the graph traversal. The row-based 1d partition can collect unvisited vertices and their neighboring edges together, which can eliminate the data communication in bottom-up stage among GPUs.

To reduce the communication overhead among GPUs, extra graph data need to be stored in each GPU by bidirectional 1d partition. In our experiment, bidirectional 1d partition needs 0.31 times extra memory capacity but the communication-to-computation ratios reduce 15.04 times, compared with traditional 1d partition. To further improve the workload balance in each GPU, we implement the static shuffle scheme proposed in our previous work (Zhang et al. 2019). The specific method sorts the vertices' degrees in descending order and further improves load balance in task assignment.

4.3 UM-aware communication

To reduce the overhead of communication and synchronization among GPUs, we propose a UM-aware communication scheme, which combines MPI library and data placements of UM.

As described above, the graph is divided and stored in each GPU node by bidirectional 1d partition. For the parallel BFS implementation among GPUs, the global synchronization in each iteration is needed for the calculation of *next queue*, inducing extra communication overhead. However, limited PCIe bandwidth becomes the main bottleneck for the performance of BFS. In this regard, NVIDIA has proposed P2P, GPU-based MPI, NVSHMEM and other technologies to achieve efficient data transmission among GPUs and CPU. The communication process of P2P is shown in Fig. 6.

In Fig. 6, although the data communication based on P2P is still limited by the bandwidth of PCIe, it eliminates two data copies between GPU and CPU, which improves the data communication among GPUs. In this paper, the initial distributed algorithm is implemented based on P2P communication. However, the scalability of the implementation gets worse with the increases of GPU numbers and graph scale, as discussed in below performance evaluation section. We further implement parallel graph traversal based on UM-aware communication approach, which combines our BFS algorithm with CUDA-based MPI and UM technologies. In this way, we can carry out the message transfer in a pipeline, which utilizes asynchronous stream to copy data and improve communication efficiency greatly. These are the main reasons that the performance of BFS based on UM-aware communication outperforms the scheme based on P2P.

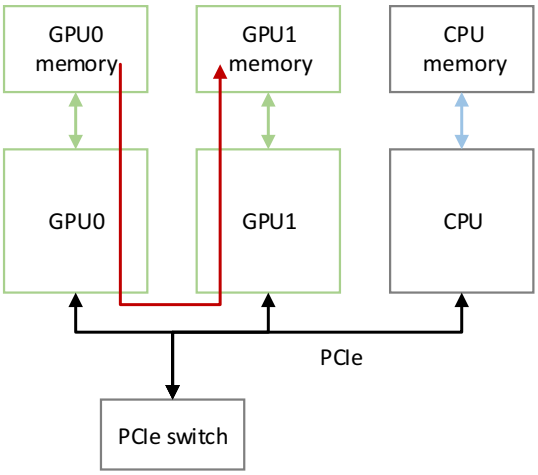

Fig. 6 Peer to Peer communication among GPUs

UM method unifies host memory and device memory into the same address spaces, and fetched required data from CPU on-the-fly. It can reduce the overhead of communication between CPU and GPU to a certain extent. We using it to transfer data that require frequent CPU-GPU interaction, such as intermediate attribute information, state data and so on. The UM method is suitable for algorithm with relatively good spatial and temporal locality. In order to improve the locality of BFS execution, we propose to use UM-aware communication, which designs UM-based data placements. Usually, UM depends on CUDA driver to implement the data movement, but the performance is cut down when it is combined with CUDA-based MPI. The main reason lies in the access of managed buffer. Under the circumstances, we pay more attention to the data placements of UM, and propose to launch a kernel on the device for migrating the memory pages to GPU from underlying CUDA driver. The detailed data process is shown in Fig. 7.

When placing data movement from CPU to CPU, the effective locations of the sending and receiving buffers are on host CPU. We only need to use CUDA-based MPI to send and receive data. When placing data movement from GPU to GPU, the effective locations of sending and receiving buffers are on GPU devices. The sending process and receiving process all need to involve with GPU kernels to send and receive data with MPI. When placing data movement from GPU to CPU, the effective locations are on host and devices, respectively. At this time, GPU device should invoke a kernel to read the relative buffer data. By this design, the communication among CPU and GPUs can take full advantage of UM, which significantly reduces the communication overhead.

5 Performance evaluation

The experimental platform is an X86 server equipped with four T4 GPUs, each of which contains 16GB global memory and 40 SMs. The specific configuration of hardware platform is shown in Table 1. Most of the experiments in this paper are running on this platform and gain good performance. The BFS algorithm on multi-GPUs is implemented with CUDA SDK and C/C++ language. The software compilation environment is NVIDIA NVCC 10.2 and GCC 6.2.0 with optimization flag of -O3.

The main datasets used in our evaluation are Kronecker graphs, which are generated using Graph500 benchmark. We use the standard parameters with (*A=0.57, B=0.19, C=0.19, D=0.05*), and the average degree of the vertex is 16. The Kronecker graph generator adjusts the scale and average degree of graph through the parameters *SCALE* and *edgefactor*. The generated graphs contain *2^ SCALE* vertices and *2^ SCALE*edgefactor* edges and follow the power-law distribution. The performance of BFS is measured in Giga-Traversed Edges per Seconds (GTEPS), by taking the ratio

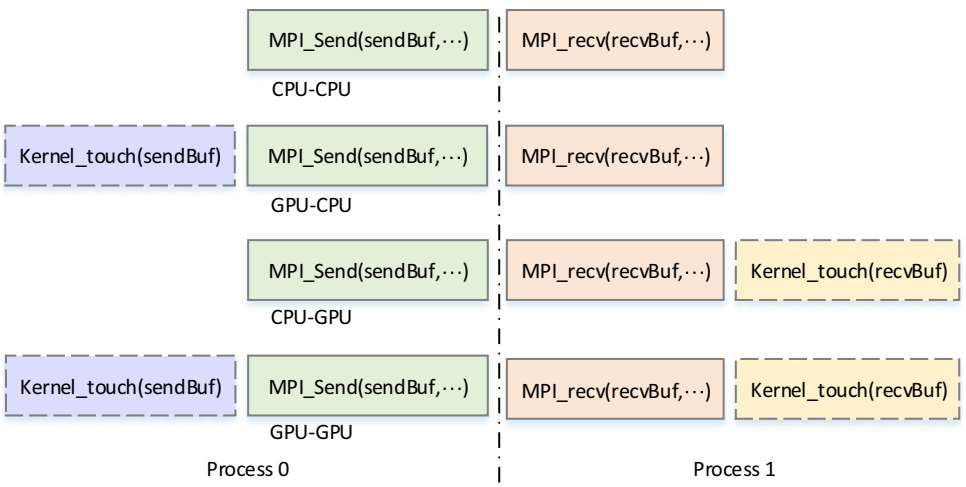

Fig. 7 The data placement scheme with UM-aware communication

Table 1 Experimental platform configuration

Processor	Xeon(R) Gold 6148 CPU	NVIDIA Tesla T4
frequency	2.40GHz	1590MHz
cores	20cores	40SMs
memory	256GB	16GB
memory type	DDR4	HBM2
L1 cache	64KB	48KB
L2 cache	1MB	4MB
L3 cache	27.5MB	–

of the number of edges over the traversal time. For each experiment, we randomly selected 64 root vertices and run 64 times BFS with different root vertices, then get the performance by taking average performance.

5.1 Performance variation on single GPU

In this section, we first evaluate our optimizations on a single GPU. Direction-optimizing BFS was chosen as the baseline (BL). Traditional optimizations (TDO) involve the existing techniques, including vertex sorting, degree-aware, and bitmap lookup approaches (Yasui and Fujisawa 2015; Zhang et al. 2019). The performance comparison of various optimizations for graphs with different *SCALE*, ranging from 2^ 21 to 2^ 26 on single GPU is shown in Fig. 8.

From Fig. 8, based on different *SCALE*, the performance of TDO outperforms BL by 1.00 to 1.66 times. These optimizations enhance the performance of BFS algorithm and achieve 16.07 GTEPS on Tesla T4. However, the performance is still lower than the state-of-art work on CPU-based platform (Zhang et al. 2019). To fully utilize the massive processing units and high bandwidth of GPU, GCS (GPU-friendly CSR structure) is proposed. It uses memory coalescing technique to improve the efficiency of memory access in bottom-up stage. Compared with the BL, GCS performance achieves up to 3.64 to 5.50 times speedup. The highest performance achieves 92.15 GTEPS with *SCALE=26* on Kronecker graph.

On single GPU, the scalability of our algorithm with varied graph *SCALE* and *edgefactor* was evaluated. The performance of graph traversal with *edgefactor* ranging from 16 to 512 is shown in Fig. 9. Due to the limited memory of T4 GPU, memory overflow occurs when processing data of *SCALE=22* and *edgefactor=512*. For this, we have added an annotation in Fig. 9. Apparently, the performance is more efficient with a larger *edgefactor*. For the graph with *SCALE=21* and *edgefactor=512*, the performance achieves 668.75 GTEPS, which is nearly 15.60 times better than the

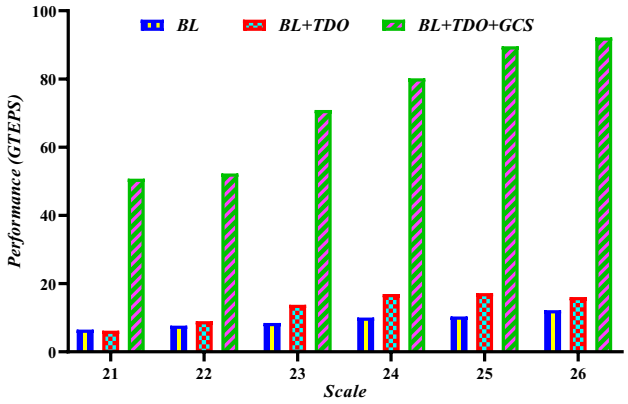

Fig. 8 BFS performance with varied optimizations

graph with *edgefactor* 16. The acceleration afforded by bottom-up approaches reduces the edge traversal required for a dense graph (larger *edgefactor*). Based on Graph500 benchmark, 16 is chosen as the default *edgefactor* in the following experiments.

5.2 Performance on multi-GPUs

In order to figure out the effect of bidirectional 1d partition, Fig. 10 shows the overhead of communication and computation before and after our optimization for graph with *SCALE=26* on four T4 GPUs. The communication and computation time of the original 1d partition are 159.79ms and 12.56ms respectively, while the communication and computation time with bidirectional 1d partition are 7.38ms and 9.31ms. Although bidirectional 1d partition increases the memory footprint by 1.31 times, the communication to computation ratios reduces 15.04 times compared with traditional 1d partition.

In terms of communication, this paper systematically compares graph traversal performance based on P2P communication and UM-aware communication. The comparison result is shown in Fig. 11. When there is only one GPU, the performance of using P2P and UM-aware communication is the same, which is reasonable because it does not involve communication among multi-GPUs. When the number of GPU increases to 4, it can be seen that the BFS performance based on UM-aware communication outperforms 1.64 times than that based on P2P. In addition, the BFS performance based on P2P degrades with the number of GPU increasing from two to four, so the scalability of distributed graph traversal based on P2P communication is limited. The main reason of the bad scalability of P2P is that multi-GPUs share a fixed PCIe bandwidth, leading to heavy communication latency.

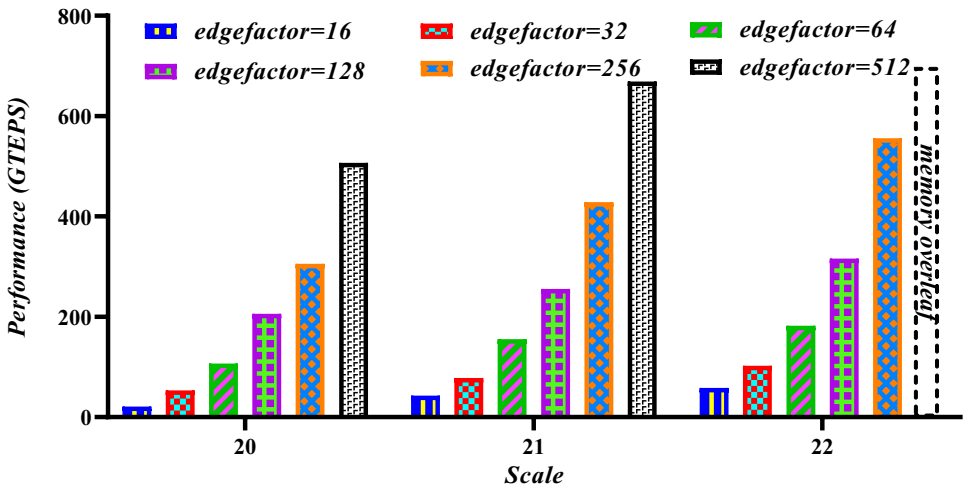

Fig. 9 BFS performance with different edgefactors

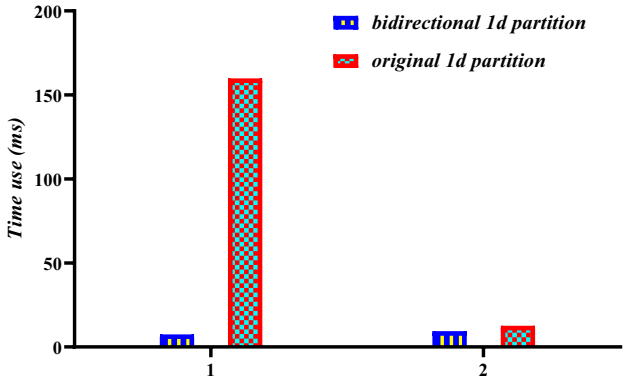

Fig. 10 Communication and computation overhead

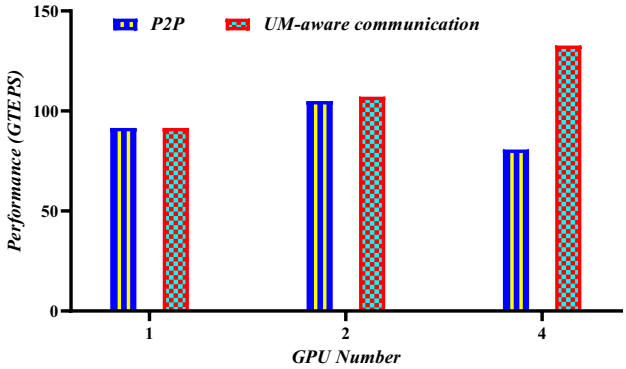

Fig. 11 BFS performance with different communication

5.3 Scalability on GPUs

In Fig. 12, we evaluate the scalability of our algorithm with varied graph *SCALE* and GPU numbers. The performance of BFS traversal increases with the GPU number, reaching the peak performance of 132.67 GTEPS at the *SCALE* of 26

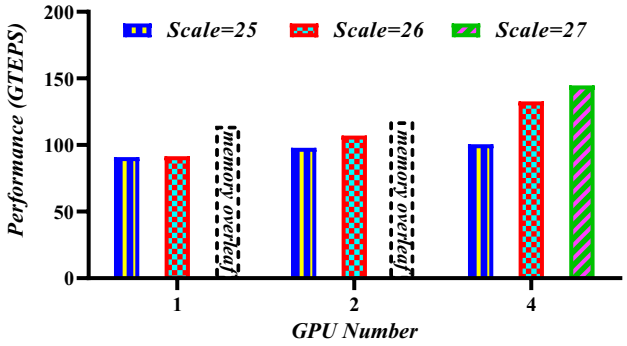

Fig. 12 BFS performance with different scale on four T4

on four T4 GPUs. It achieves 1.44 times speedup compared with that on single GPU. When the graph *SCALE* increases to 27, only the system with four GPUs can process the large-scale graph, due to the insufficient distributed GPU global memory. Under the circumstances, the peak performance is 155.37 GTEPS. Our algorithm shows good scalability for large-scale graph on multi-GPUs.

If GPU has stronger computing power or NVLink technology is supported, the performance will be further improved. The NVIDIA Tesla V100 GPU has 80 SMs, 5120 CUDA cores in total, and its memory bandwidth reaches 900 GB/s. We launch the algorithm on four V100, which support for the NVLink with the second generation and the global memory of each GPU is 16GB. The results is show in Fig. 13 (a), the peak performance for the graph with *SCALE=26* and *edgefactor=16* reaches 392.35 GTEPS. Our implementation on four V100 GPUs is in a leading position among GPU-based systems, and it is better than the existing CPU-based implementation with 1024 nodes.

In Fig. 13 (a), we also find that the performance of BFS on single GPU outperforms that on two GPUs. In this regard, we further analyze the computation and communication

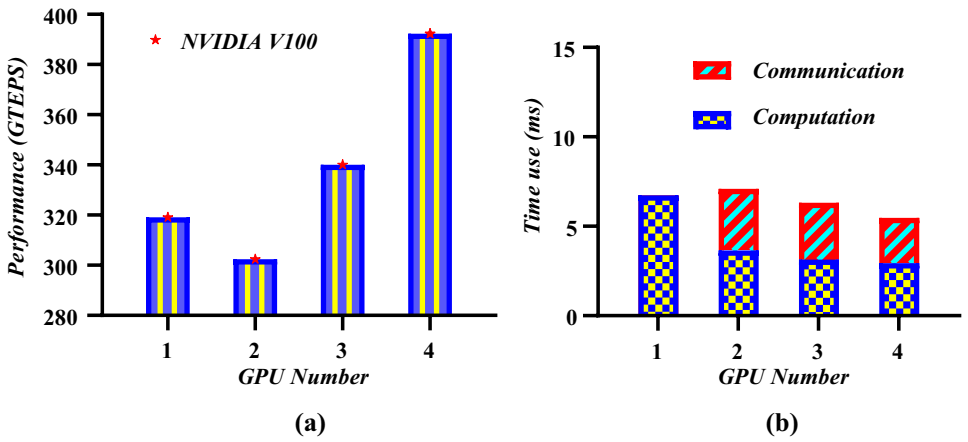

Fig. 13 The performance of BFS on four V100

overhead during the execution of BFS algorithm. The results are show in Fig. 13 (b). The parallel efficiency of BFS algorithm can be improved with the number of GPUs increasing, but it also increases communication overhead, which affects the performance of the algorithm. When the number of GPUs is 2, the improvement brought by multi-GPUs parallelism can not completely cover the time delay caused by communication among GPUs. In addition, it is supposed to have good scalability among more GPUs within one node. Due to the limited hardware environment, it is currently not possible to conduct relevant experiments.

5.4 Comparison with existing work

In order to show the efficiency of our optimizations, we compare our algorithm with several BFS implementations on GPU platform.

For single GPU, we compare our BFS with several state-of-art works, including Enterprise Liu and Huang (2015) and Tigr (Sabet et al. 2018). Our work is primarily designed for scale-free graphs with small diameter, instead of graphs with high diameter like road networks. Except for Kronecker graph, we also evaluate some real-word small-diameter graphs, such as higgs (De Domenico et al. 2013), soc-Pokec (Takac and Zabovsky 2012), com-Orkut (Yang and Leskovec 2015) and wiki-topcats (Klymko et al. 2014; Yin et al. 2017). The results are shown in Table 2. For Kron-n24-16 (Kronecker graph with *SCALE=24* and *edge-factor=16*), our algorithm achieves 96.97 GTEPS, which is much higher than other works. For small-diameter graphs, our work performs average 1.65, 2.35 times better than Tigr and Enterprise.

We also compare our implementation with existing distributed works, including (Liu and Huang 2015; Pan et al. 2017; Bisson et al. 2015) and (Bernaschi et al. 2015). The results are taken directly from their paper due to the unopened source of their distributed code. As shown in Table 3, for Kron-n24-32 graph on four T4 GPUs, the performance of our work reaches 192.53 GTEPS, which is 2.84 times better than (Pan et al. 2017). For com-Orkut (Yang and Leskovec 2015) graph, our algorithm achieves 3.08× speedup, compared with (Bisson et al. 2015).

Considering the software and hardware performance of T4, K40, K20X GPUs, our work shows superior performance gains with regard to existing works. Tesla T4 has 40 SMs, with total 2560 CUDA cores. In terms of memory system, the peak memory bandwidth of T4 is 320 GB/s. While K40 and K20X have 2880 and 2688 CUDA cores respectively, as well the memory bandwidth of K40 and K20X are 288, 255 GB/s. Therefore, taking parallelism resources and memory bandwidth into consideration, our theoretical

Table 2 Comparison with previous single GPU BFS work (GTEPS)

Graph data	Enterprise	Tigr	Our
higgs	3.23	4.61	7.61
soc-Pokec	4.56	2.67	7.85
com-Orkut	3.16	2.75	8.86
wiki-topcats	2.60	3.96	8.76
Kron-n24-16	5.64	1.08	90.17
Kron-n25-16	7.31	0.99	96.97

Table 3 Comparison with previous multi-GPUs BFS work (GTEPS)

Ref	Graph data	Hardware	Performance
Liu and Huang (2015)	Kron-n24-32	2*K40	15.00
Pan et al. (2017)	Kron-n24-32	2*K40	77.70
Our	Kron-n24-32	2*T4	177.69
Pan et al. (2017)	Kron-n24-32	4*K40	67.70
Our	Kron-n24-32	4*T4	192.53
Bernaschi et al. (2015)	Kron-n23-16	4*K20X	1.30
Our	Kron-n23-16	4*T4	65.91
Bisson et al. (2015)	com-Orkut	4*K20X	2.67
Our	com-Orkut	4*T4	8.25

analysis suggests that the a T4 GPU give almost 1.5 to 2.0 times performance of a K40 or K20X. And the experimental results are consistent with the theoretical analysis, which shows that our BFS algorithm has substantial scalability on different GPU platforms.

6 Related work

In recent years, in order to utilize the massive parallelism and enhance the performance of graph traversal on GPUs, plenty of optimizations have been put forward, which achieve a series of improvements and promote good performance on GPU-based platforms to some certain degree.

For graph traversal on a single GPU, Harish and Narayanan put forward BFS implementation on GPU based on vertex-centric model, which identified active vertices by scanning vertices' status array (Harish and Narayanan 2007). Hong et al. proposed virtual warp to improve the workload balance on GPU (Hong et al. 2011). Under the circumstances, the *adjacency list* of each active vertex would be processed by a group of threads instead of one thread, which further improve the efficiency of graph traversal. Merrill et al. proposed an adaptive parallelization of BFS algorithm that mapped the workload of a vertex to a single thread, warp or block depending on its out-degree and achieved high performance on GPU (Merrill et al. 2012). In 2013, Beamer proposed a direction-optimizing scheme that combined the traditional top-down approach with a novel bottom-up approach (Beamer et al. 2012). It could dramatically reduce the number of redundant edges traversal.

For graph traversal on multi-GPUs, Hiragushi et al. implemented an efficient hybrid BFS implementation on GPUs, demonstrating that it was beneficial for graph traversal on GPUs (Hiragushi and Takahashi 2013). Zhong et al. proposed the Medusa graph processing system, which adapted the multi-hop replication mechanism to reduce the communication overhead between GPU and CPU (Zhong and He 2013). Khorasani and Takahashi proposed a CUDA-based graph processing framework CuSha, which used two novel graph representations G-Shared and Concatenated Windows to overcome the irregular memory accesses and underutilization of GPU resources (Khorasani et al. 2014). Pan et al. implemented a distributed BFS algorithm on GPUs and proposed a novel communication model by using global reduction for high-degree vertices, and point-to-point transmission for low-degree vertices (Pan et al. 2018). Besides, to accelerate the data interaction between CPU and GPU, some works focus on GPUs by using Unified Memory (UM), such as SubWay (Sabet et al. 2020), GRUS (Wang et al. 2021) et al.

Although plenty of optimizations have been proposed and achieved good results on GPU platform, the essential problems, such as workload imbalance, memory access inefficiency and high communication overhead, still need further improvements. Our work focus on dealing with memory access divergence, workload imbalance, communication overhead and redundant computation on GPUs, and achieve good performance and high scalability.

7 Conclusions

In this paper, we implement FSGraph, a fast and scalable BFS algorithm on GPUs. We propose three optimization techniques and study the performance of BFS implementation on NVIDIA Tesla T4 and V100 GPUs. Our algorithm demonstrates good performance and scalability. The average performance of BFS on four T4 GPUs is 132.67 GPTES with *SCALE=26* and *edgefactoer=16*, which delivers up to 1.44× improvement than that on a single T4. In terms of V100 GPUs, the BFS performance on four V100 GPUs achieves nearly 392.35 GTEPS with improved memory access, balanced data partition, efficient data communication. To the best of our knowledge, our implementation achieves the highest performance among existing GPU-based systems. In the near future, it is planned to investigate the big data extension on GPU cluster with limited memory.

Acknowledgements 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.

Data Availability The data that support the findings of this study are available from the corresponding author upon reasonable request.

Declarations

Conflict of interest No potential conflict of interest was reported by the authors

References

Agarwal, V., Petrini, F., Pasetto, D., Bader, D.:A.: Scalable graph exploration on multicore processors. In: SC'10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, pages 1–11. IEEE, 2010

Bader, D. A., Madduri, K.: Snap, small-world network analysis and partitioning: An open-source parallel graph framework for the exploration of large-scale networks. In: 2008 IEEE international symposium on parallel and distributed processing, pp. 1–12, IEEE, 2008

Beamer, S., Asanovic, K., Patterson, D.: Direction-optimizing breadth-first search. In: SC'12: Proceedings of the International

Conference on High Performance Computing, Networking, Storage and Analysis, pp 1–10, IEEE, 2012
Bernaschi, M., Carbone, G., Mastrostefano, E., Bisson, M., Fatica, M.: Enhanced gpu-based distributed breadth first search. In: Proceedings of the 12th ACM International Conference on Computing Frontiers, pages 1–8, 2015
Bisson, Mauro, Bernaschi, Massimo, Mastrostefano, Enrico: Parallel distributed breadth first search on the kepler architecture. IEEE Transact. Parallel Distrib. Syst **27**(7), 2091–2102 (2015)
Buluç, Aydin, Madduri, K.: Parallel breadth-first search on distributed memory systems. In: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, pages 1–12, 2011
Busato, Federico, Bombieri, Nicola: Bfs-4k: an efficient implementation of bfs for kepler gpu architectures. IEEE Transact. Parallel Distrib. Syst. **26**(7), 1826–1838 (2014)
Checconi, F.o, Petrini, F., Willcock, J., Lumsdaine, A., Choudhury, A. Roy, Sabharwal, Y.: Breaking the speed and scalability barriers for graph exploration on distributed-memory machines. In: SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, pages 1–12, IEEE, 2012
De Domenico, Manlio, Lima, Antonio, Mougel, Paul, Musolesi, Mirco: The anatomy of a scientific rumor. Sci. Rep. **3**(1), 1–9 (2013)
Dong, R.u, Cao, H., Ye, X., Zhang, Y., Hao, Q., Fan, D.: Highly efficient and gpu-friendly implementation of bfs on single-node system. In: 2020 IEEE Intl Conf on Parallel and Distributed Processing with Applications, Big Data and Cloud Computing, Sustainable Computing and Communications, Social Computing and Networking (ISPA/BDCloud/SocialCom/SustainCom), pp 544–553, IEEE, 2020
Faloutsos, Michalis, Faloutsos, Petros, Faloutsos, Christos: On power-law relationships of the internet topology. In: The Structure and Dynamics of Networks, pp. 195–206. Princeton University Press, New jersey (2011)
Graph500. http://www.graph500.org, (2010)
Harish, P., Narayanan, P. J.: Accelerating large graph algorithms on the gpu using cuda. In International conference on high-performance computing, Springer, pp 197–208, 2007
Hiragushi, T., Takahashi, D.: Efficient hybrid breadth-first search on gpus. In: International Conference on Algorithms and Architectures for Parallel Processing, Springer, pages 40–50, 2013
Hong, Sungpack, Kim, Sang Kyun, Oguntebi, Tayo, Olukotun, Kunle: Accelerating cuda graph algorithms at maximum warp. Acm. Sigplan. Notices **46**(8), 267–276 (2011)
Khorasani, F., Vora, Keval, G., Rajiv, B., Laxmi N., Cusha.: Vertex-centric graph processing on gpus. In: Proceedings of the 23rd international symposium on High-performance parallel and distributed computing, pages 239–252, 2014
Klymko, C., Gleich, D., Kolda, T, G.: Using triangles to improve community detection in directed networks. arXiv preprint arXiv:1404.5874,(2014)
Li, Z., Wang, H., Zhang, P., Hui, P., Huang, J., Liao, J., Zhang, J., Bu, J.: Live-streaming fraud detection: a heterogeneous graph neural network approach. In: Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, pp. 3670–3678, 2021
Liu, H., Huang, H H.: Enterprise: breadth-first graph traversal on gpus. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, pages 1–12, 2015
Luo, L., Wong, M., Hwu, Wen-m.: An effective gpu implementation of breadth-first search. In: Design Automation Conference, pages 52–55, IEEE, 2010
Merrill, Duane, Garland, Michael, Grimshaw, Andrew: Scalable gpu graph traversal. Acm. Sigplan. Notices **47**(8), 117–128 (2012)
Mislove, A., Marcon, M., Gummadi, K. P., Druschel, P., Bhattacharjee, B.: Measurement and analysis of online social networks. In: Proceedings of the 7th ACM SIGCOMM conference on Internet measurement, pages 29–42, 2007
Murphy, Richard C., Wheeler, Kyle B., Barrett, Brian W., Ang, James A.: Introducing the graph 500. Cray Use. Group (CUG). **19**, 45–74 (2010)
Nvidia. nvidia t4 70w low profile pcie gpu accelerator. https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/tesla-t4/t4-tensor-core-product-brief.pdf, (2020)
Pan, Y., Pearce, R., Owens, J, D.: Scalable breadth-first search on a gpu cluster. In: 2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS), pages 1090–1101. IEEE, 2018
Pan, Y., Wang, Y., Wu, Y., Yang, C., Owens, J. D.: Multi-gpu graph analytics. In: 2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS), IEEE,pages 479–490, 2017
Pham, T.-A. N., Li, X., Cong, G., Zhang, Z.: A general graph-based model for recommendation in event-based social networks. In: 2015 IEEE 31st international conference on data engineering, pp. 567–578, IEEE, 2015
Potluri, Sreeram, Goswami, Anshuman, Venkata, Manjunath Gorentla, Imam, Neena: Efficient breadth first search on multi-gpu systems using gpu-centric openshmem, pp. 82–96. Springer, In Workshop on OpenSHMEM and Related Technologies (2017)
Sabet, Amir Hossein N., Zhao, Zhijia, Gupta R.: Subway Minimizing data transfer during out-of-gpu-memory graph processing. In: Proceedings of the Fifteenth European Conference on Computer Systems, pages 1–16, 2020
Sabet, Amir Hossein Nodehi., Qiu, Junqiao, Zhao, Zhijia: Tigr: Transforming irregular graphs for gpu-friendly graph processing. ACM SIGPLAN Notices **53**(2), 622–636 (2018)
Takac, L., Zabovsky, M.: Data analysis in public social networks. In: International scientific conference and international workshop present day trends of innovations. Present Day Trends of Innovations Lamza Poland, 2012
Ting, Y., Yan, C., Xiang-wei, M.: Personalized recommendation system based on web log mining and weighted bipartite graph. In: 2013 international conference on computational and information sciences, pp 587–590, IEEE, 2013
Ueno, K., Suzumura, T.: Highly scalable graph search for the graph500 benchmark. In: Proceedings of the 21st international symposium on High-Performance Parallel and Distributed Computing, pages 149–160, 2012
Wang, Pengyu, Wang, Jing, Li, Chao, Wang, Jianzong, Zhu, Haojin, Guo, Minyi: Grus: Toward unified-memory-efficient high-performance graph processing on gpu. ACM Transact. Architec. Code Optimiz. (TACO) **18**(2), 1–25 (2021)
Yang, Jaewon, Leskovec, Jure: Defining and evaluating network communities based on ground-truth. Knowledge Info. Syst. **42**(1), 181–213 (2015)
Yasui, Y., Fujisawa, K.: Fast and scalable numa-based thread parallel breadth-first search. In: 2015 International Conference on High Performance Computing and Simulation (HPCS), pp 377–385, IEEE, 2015
Yin, H., Benson, A. R., Leskovec, J., Gleich, D. F.:Local higher-order graph clustering. In: Proceedings of the 23rd ACM SIGKDD international conference on knowledge discovery and data mining, pages 555–564, 2017
Yoo, A., Chow, E., Henderson, K.h, McLendon, W., Hendrickson, B., Catalyurek, U.: A scalable distributed parallel breadth-first search algorithm on bluegene/l. In: SC'05: Proceedings of the 2005 ACM/IEEE Conference on Supercomputing, pp 25–25, IEEE, 2005
Zhang, C., Cao, H., Ye, X., Wang, G., Hao, Q., Fan, D.: Highly efficient breadth-first search on cpu-based single-node system. In: 2019 IEEE 21st International Conference on High Performance

Computing and Communications; IEEE 17th International Conference on Smart City; IEEE 5th International Conference on Data Science and Systems (HPCC/SmartCity/DSS), pages 2066–2071, IEEE, 2019

Zhong, Jianlong, He, Bingsheng: Medusa: Simplified graph processing on gpus. IEEE Transact. Parallel Distrib. Syst. **25**(6), 1543–1552 (2013)

Zhong, Wenyong, Sun, Jianhua, Chen, Hao, Xiao, Jun, Chen, Zhiwen, Cheng, Chang, Shi, Xuanhua: Optimizing graph processing on gpus. IEEE Transact. Parallel Distrib. Syst. **28**(4), 1149–1162 (2016)

Yuan Zhang born in 1990. PhD candidate. Member of CCF. Her main research interests include high throughput computing architecture and parallel computing.

Huawei Cao born in 1989. PhD. Member of CCF. His main research interests include parallel computing and high throughput computing architecture.

Yan Liang born in 1984. PhD candidate. His main research interests include high throughput computing architecture and parallel computing.

Jie Zhang born in 1999. She is currently pursuing the master's degree in computer architecture from Institute of Computing Technology, Chinese Academy of Sciences. Her main research interests include high throughput computing architecture and paralleling computing.

Junying Huang received her Ph.D. degree in microelectronics and solid-state electronics from the University of Chinese Academy of Sciences in 2016. Currently she is an associate professor with the Department of High-throughput Computer Research Center, Institute of Computing Technology, Chinese Academy of Sciences. Her research interests include superconductive RSFQ logic, computer architecture, electronic design automation, and hardware security.

Xiaochun Ye received the Ph.D. degree in computer architecture from the Institute of Computing Technology, Chinese Academy of Sciences (CAS), in 2010. Currently he is a Professor Researcher, director of the High-Throughput Computer Research Center in Institute of Computing Technology, CAS. His main research interests include many-core processor structure and software simulation technology.

Xuejun An born in 1966. PhD, senior engineer, PhD supervisor. Member of CCF. His main research interests include computer system architecture and high performance interconnection network.