Int J Parallel Prog (2019) 47:644–667
https://doi.org/10.1007/s10766-017-0533-y

WolfPath: Accelerating Iterative Traversing-Based Graph Processing Algorithms on GPU

Huanzhou Zhu[1] · **Ligang He**[1] · **Songling Fu**[2] · **Rui Li**[3] · **Xie Han**[4] · **Zhangjie Fu**[5] · **Yongjian Hu**[6] · **Chang-Tsun Li**[7]

Received: 15 September 2017 / Accepted: 28 September 2017 / Published online: 14 November 2017

Abstract There is the significant interest nowadays in developing the frameworks of parallelizing the processing for the large graphs such as social networks, Web graphs, etc. Most parallel graph processing frameworks employ iterative processing model. However, by benchmarking the state-of-art GPU-based graph processing frameworks, we observed that the performance of iterative traversing-based graph algorithms (such as Bread First Search, Single Source Shortest Path and so on) on GPU is limited by the frequent data exchange between host and GPU. In order to tackle the problem, we develop a GPU-based graph framework called WolfPath to accelerate the processing of iterative traversing-based graph processing algorithms. In WolfPath, the iterative process is guided by the graph diameter to eliminate the frequent data exchange between host and GPU. To accomplish this goal, WolfPath proposes a data structure called Layered Edge list to represent the graph, from which the graph diameter is known before the start of graph processing. In order to enhance the applicability of our WolfPath

✉ Ligang He
ligang.he@warwick.ac.uk

[1] Department of Computer Science, University of Warwick, Coventry CV4 7AL, UK

[2] College of Polytechnic, Hunan Normal University, Changsha, China

[3] College of Computer Science and Network Security, Dongguan University of Technology, Dongguan, China

[4] School of Electronics and Computer Science and Technology, North University of China, Taiyuan, China

[5] School of Computer and Software, Nanjing University of Information Science and Technology, Nanjing, China

[6] School of Electronic and Information Engineering, South China University of Technology, Guangzhou, China

[7] School of Computing and Mathematics, Charles Sturt University, Wagga Wagga, Australia

framework, a graph preprocessing algorithm is also developed in this work to convert any graph into the format of the Layered Edge list. We conducted extensive experiments to verify the effectiveness of WolfPath. The experimental results show that WolfPath achieves significant speedup over the state-of-art GPU-based in-memory and out-of-memory graph processing frameworks.

Keywords Graph processing · GPGPU · Parallel computing

1 Introduction

The demand for efficiently processing large scale graphs has been growing fast nowadays. This is because graphs can be used to describe a wide range of objects and computations and graph-based data structures are the basis of many applications [19,22–24,26,30,36,46,48]. Traditionally, motivated by the need to process very large graphs, many frameworks have been developed for processing large graphs on distributed systems. Examples of such frameworks include Pregel [28], GraphLab [25], PowerGraph [9] and GraphX [10]. However, since developing distributed graph algorithm is challenging, some researchers divert their attention to design the graph processing system that handle large scale graphs on a single PC. The research endeavours in this direction have delivered the systems such as GraphChi [17], PathGraph [45], GraphQ [39], LLAMA [27] and GridGraph [51]. However, these systems suffer from the limited degree of parallelism in conventional processors. GPU is renowned for its potential to offer the massive degree of parallelism in many areas [3,4,11,21,32,42–44,49,52]. Therefore, much research now resort to use GPU to accelerate the graph processing process. The exemplar GPU-based graph processing systems include Medusa [47], Gunrock [40], CuSha [16], Frog [35] and MapGraph [6].

Many of these graph processing frameworks employ iterative processing techniques [38,41,50]. Namely, graph processing involves many iterations. Some iterative graph processing algorithms use the threshold value (e.g., in the PageRank algorithm) or the number of vertices/edges (e.g., in the Minimum-cut algorithm) to determine the termination of the algorithms. In these algorithms, the iteration count is known beforehand.

However, in iterative traversing-based graph processing, the algorithm is driven by the graph structure and the termination of the algorithm is determined by the states of vertices/edges. Therefore, these algorithms need to check the state of vertices/edges at the end of every iteration to determine whether to run the next iteration. In each iteration, either synchronous or asynchronous methods can be used to compute and update the values of vertices or edges in the graph. The processing terminates when all vertices meet the termination criteria.

The aforementioned termination criteria is application-specific (e.g, the newly computed results of all vertices remain unchanged from the previous iteration). The number of iterations is unknown before the algorithm starts. Such a termination method does not cause any problem on CPU-based graph processing systems, because checking the termination condition on CPU does not incur much overhead. On the other hand, termination checking is a time consuming process in GPU-accelerated systems. This is because all the threads in different thread blocks need to synchronize their decision at this point. However, current GPU devices and frameworks (CUDA [5] and OpenCL [37]) do not support synchronization among different thread blocks during the execution of the kernel. Therefore, to synchronize between different thread blocks, the program has to exit the kernel, and copy the data back to the host and use the CPU to determine whether the computation process is complete. This frequent data exchange between host and GPU introduces considerable overhead.

To address this problem, we present WolfPath, a framework that is designed to improve the iterative traversing-based graph processing algorithms (such as BFS and SSSP) on GPU. Our development is motived by the facts that: (1) the iterative graph processing algorithm converges when the computations on all vertices have been completed; (2) the iterative traversing-based graph processing on GPU requires frequent data exchange between GPU and host memory to determine the convergence point; (3) the maximum number of iterations needed to complete a traversing-based graph processing algorithm is determined by the graph's diameter (longest shortest path).

WolfPath has following features. First, a graph in WolfPath is represented by a tree structure. In doing so, we manage to obtain very useful information, such as the graph diameter, the degree of each vertex and the traversal order of the graph, which will be used by WolfPath in graph computations. Second, we design a layered graph structure, which is used by WolfPath to optimize GPU computations. More concretely, for all vertices in the same depth of the graph tree, we group all the edges that use these vertices as source vertex into a layer. All the edges in the same layer can be processed in parallel, and coalesced memory access can be guaranteed. Last but not least, based on the information we gain from the graph model, we design a computation model that does not require frequent data exchange between host and GPU.

The rest of the paper is organised as follows. Section 2 overviews the limitation of iterative graph processing technique used by the state-of-art graph processing frameworks. Section 3 presents the graph modelling and in-memory data structure proposed in this paper. Section 4 presents the details of the WolfPath framework, including the iterative processing model and the graph partition method when the graph can not fit into GPU. Experimental evaluation is presented in Sect. 5. Section 6 discusses related work. Finally, Sect. 7 concludes this paper.

2 Motivation: the Limitation of Current Approach

Many parallel graph processing frameworks are based on the iterative processing model. In this model, the computation goes through many iterations. In each iteration, either synchronous (such as Bulk Synchronous parallel used by Pregel) or asynchronous (Parallel sliding window used by GraphChi) methods can be used to compute and update the vertex or edge values. The computation terminates when all vertices meet the application specific termination criterion (e.g, the results of all vertices remain unchanged).

In order to determine if all processes/threads meet the termination condition, the processes/threads need to communicate with each other. On CPU based parallel graph processing systems, the processes/threads can communicate through messaging passing or shared memory. On GPU, the threads are organised in thread blocks and the communication can be divided into two types: intra- and inter-block communication.

Intra-communication refers to the communications between the threads within a block, which is achieved via shared memory or global memory in GPU. On the contrary, the inter-block communication is the communications across different thread blocks. However, there is no explicit support for data communication across different thread blocks. Currently, inter-block data communication is realized through the GPU global memory followed by a barrier synchronization in CPU [5,37]. The barrier is implemented by terminating the execution of the current kernel and re-launching the kernel.

The reason for the lack of support for inter-block communications on GPU is as follows. On GPU, the number of thread blocks launched by an application is normally much larger than the number of Streaming Multiprocessors (SM). However, When a large number of threads try to communicate between different blocks, it can cause the deadlock. An example is given as follows to illustrate the deadlock issue. Suppose that there are 5 thread blocks and only 4 SMs and that each thread block will occupy all resources on a SM. Assume blocks 1–4 execute on 4 SMs. When synchronization occurs, blocks 1–4 will wait until block 5 finishes. However, block 5 will never be executed on any SM since all SMs are busy and there are no resources available. Consequently, the deadlock occurs.

Due to the lack of support for inter-block communications, implementing iterative graph computation algorithm on GPU is much more challenging than on CPU. To demonstrate this, let us first consider how the iterative computation is implemented on CPU. Algorithm 1 shows the high level structure of the iterative computation on CPU. The loop is controlled by the *flag* variable, which is set to be true at the beginning. Next, all processes/threads execute a user-defined function, *compute()*, and then invoke the *update_condition* function to check if the user-specified termination condition is met. Each process/thread has its own $flag$ variable, which is updated by *update_condition* function. The *update_condition* function returns *false* if the program reaches the termination condition and returns $true$ if otherwise. The $flag$ variable is synchronised between all the processes/threads. If its value is $false$, the iteration terminates.

Algorithm 1 Iterative Computation on CPU

```
$flag \leftarrow true$
**while** $flag == true$ **do**
    compute();
    **if** $update\_condition() == false$ **then**
        $flag \leftarrow false$
```

When executing the above code in parallel on a CPU-based system, the synchronization of the $flag$ variable can be easily achieved through shared memory or message passing. However, due to the fact that the current GPUs do not support synchronisation between different thread blocks, it takes more efforts to achieve the synchronization on GPU. The only solution that can ensure that the shared variable is properly synchronized across all thread blocks is to exit the kernel. To implement the iterative processing on GPU, many state-of-art graph processing frameworks use the following approach. The $flag$ variable is stored in the global memory of GPU. Each thread also has a local d_flag variable. If a thread meets the termination condition in the current iteration, it sets its own d_flag to *false*. Then d_flag is synchronised between all the threads within a thread block. One thread in each thread block updates the global $flag$ variable if the value of d_flag in this thread block is $false$. Next, the program exits the kernel and copies the value of $flag$ back to the host, which is used by the host program to determine whether another kernel should be launched (i.e., continuing to perform the computation). This technique is outlined Algorithms 2 and 3.

Algorithm 2 Iterative processing kernel

```
function UPDATE_KERNEL(bool $flag$)
    for all threads in parallel: do
        if $update\_condition() == false$ then
            $flag \leftarrow false$
```

Algorithm 3 Iterative processing host

```
$flag \leftarrow true$
while $flag == true$ do
    function COPY_FLAG_TO_GPU
    function UPDATE_KERNEL($flag$)
    function COPY_FLAG_TO_CPU
```

Clearly, in order to decide whether to launch the next iteration, the value of $flag$ needs to be exchanged between host and GPU frequently. These operations incur a significant overhead. If the number of iterations needed to complete the graph computation is known beforehand, the exchange of $flag$ between host and GPU can be eliminated, which can potentially improve the performance.

We conduct the experiments to investigate the extra overhead caused by exchanging the $flag$ variable. Four real world graphs, which are listed in Table 1, are used in this experiment. We determine the diameter of the graph using the BFS algorithm implementated on CuSha framework [5]. We record the computation time and the number of iterations it takes to complete the algorithm. We can prove that the number of iterations the iterative graph computation has to perform must be less than or equal to the graph diameter. Instead of using the $flag$ value to determine the termination condition of the graph computation, we terminate the computation when the number of iterations equals to the graph diameter. We re-run the modified program and record the computation time. The results are listed in Table 2.

As can be seen from Table 2, the computation time of modified program is much faster than the original version. We also noticed that the original program takes a fewer number of iterations than the modified version in some cases. This is because the computation converges fast in those cases. Therefore, using the graph diameter as the termination condition in those cases causes the extra overhead of performing unnecessary computations. In order to compare these two types of overhead, we mea-

Table 1 Real world graphs used in the experiments

Graph	Vertices	Edges
GPU In memory Graph		
RoadNet-CA [20]	1,965,206	5,533,214
amazon0601 [20]	403,394	3,387,388
Web-Google [20]	8,75,713	5,105,039
LiveJournal [20]	4,847,571	68,993,773

Table 2 The computation time comparison of Original CuSha and Modified CuSha

Graph	Original		Modified	
	Time (ms)	Iteration	Time (ms)	Iteration
RoadNet-CA	195.97	551	1.98	554
amazon0601	22.3	35	0.15	35
web-Google	12.83	21	0.14	32
LiveJournal	70.96	9	0.076	14

Table 3 Memory copy and computation time

Graph	Memory copy (ms)	Computation (ms)
RoadNet-CA	0.36	0.004
amazon0601	0.63	0.004
Web-Google	0.6	0.004
LiveJournal	7.9	0.005

sured the average time for performing data copying and the average computation time per iteration in the experiments. The results are listed in Table 3:

The results from Table 3 show that the data copy time is greater than the computation time by 90–1500 times. These results indicate that it is worth performing the excessive iterations, comparing with copying data. We observe another interesting point in Table 3. No matter which graph the algorithm is processing, only one integer (i.e., the value of the $flag$ variable) is copied between GPU and host. However, the average memory copying time per iteration is different for different graphs. This is because the synchronisation cost between thread blocks is different for different graphs. More threads are involved in synchronisation, the longer the synchronization takes.

All these results support our proposed strategy. Namely, it can improve performance to use the maximum number of iterations as the termination condition so as to eliminate the need of data copying between GPU and CPU. However, a question arises from the strategy: how to know the number of iterations needed for different graphs before the graph processing algorithm starts? This question motivates us to design a new graph representation that help determine the graph diameter and further develop the novel and GPU-friendly graph processing methods.

3 Graph Representation in WolfPath

In this section, we present the graph model to help determine the number of iterations when processing a graph on GPU iteratively. We also present the detailed data structure of our in-memory graph representation, which is designed in the way that can improve the coalescence in-memory access so as to achieve the higher performance.

3.1 Graph Modelling in WolfPath

The experimental results shown in last section suggest that using the number of iterations can improve the performance in graph processing. However, because graph processing algorithms are data driven, it is difficult to determine the number of iterations for different graph inputs before the program starts. Much research [14,15,18,28] have been conducted to tackle this problem. The research shows that when processing graph algorithms iteratively, the upper bound of the number of iterations is the diameter of a graph, i.e., the number of the nodes on the path corresponding to the graph diameter [31].

In WolfPath, we model the graph as a layered tree structure. That is, we first represent the graph as a tree-like structure, then group the vertices that in the same depth into one layer. By modelling the graph this way, the diameter of the graph is the distance from the root vertex to the deepest leaf vertex.

If some vertices in the same layer are connected, we duplicate these vertices in the next level. By duplicating these vertices, the vertex value updated in the current level can be sent to the next level. The reason for this design is as follow. The vertices in the current level and the next level form a group of edges. If a vertex is both source and destination of different edges, the calculated values of their neighbouring vertices may not settle (i.e., the calculated values are not final values of the vertices) after one iteration. Therefore, by duplicating these vertices in the next level, the updated value of their neighbouring vertices will be recomputed.

Based on the above description, given a graph $G = (V, E)$, a layered tree $T = (V_t, E_t)$ is defined as follows. $V_t \subseteq V$ and $E_t \subseteq E$. The root vertex of the tree, denoted by $v_{rt} \in V_t$, is the vertex which does not have in-edges. $degree_{in}(v)$ denotes the in-degree of vertex v. Then $degree_{in}(v_{rt}) = 0$.

$\forall v_t \in V_t$, if $degree_{in}(v_t)$ is greater than 0, then $\exists v \in V_t$ s.t. $(v, v_t) \in E_t$. If the out-degree of vertex v_t, denoted by $degree_{out}(v_t)$, is 0, then v_t is called a leaf vertex of T. Given a level L_i, $\forall v_t \in L_i$, if $\exists v \in L_i$ s.t. $e_t = (v_t, v)$, then v is also in the level L_{i+1}. Figure 1b gives the tree structure of the graph shown in Fig. 1a.

3.2 Graph Data Structure in WolfPath

As shown in the previous research [16], one of the main factors that limits the performance of graph processing on GPU is the non-coalesced memory access. This is because most graphs have highly irregular structures. Hence, it is important to design a data structure for the graph storing in the GPU global memory so as to achieve the coalesced global memory access.

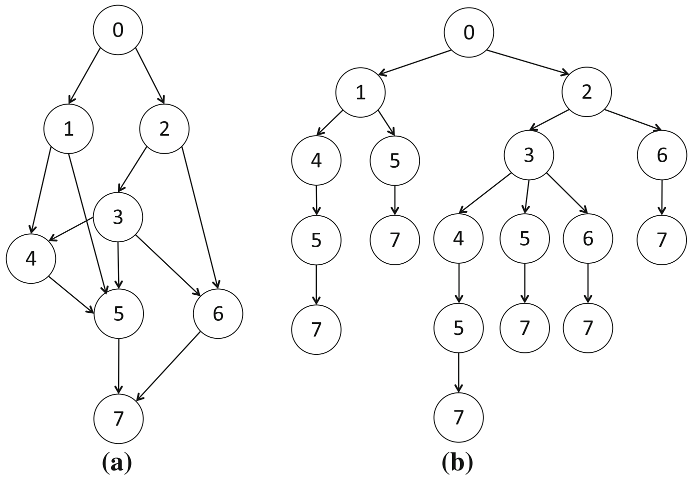


Fig. 1 An example graph and its layered tree representation. **a** Example graph and **b** Layered tree representation

Based on the layered tree structure, we design a layered edge list structure to store the graph in the memory. In this design, each level in the layered tree is represented by two arrays, *source array* and *destination array*, which are used to store the source and destination vertexes of each edge in that level respectively. The i-th entry in the source array and the i-th entry in the destination array form an edge in the level. We also create an array for each layer to store the updated values of the destination vertices in that layer. An example is shown in Fig. 2.

It provides the following benefits to use this structure to store a graph in memory. First, the consecutive threads can read/write the consecutive elements from/to the source and destination arrays in the global memory. Therefore the coalesced global memory access can be guaranteed. Second, because all the edges in each layer are independent to each other, we can process them in parallel. In addition, if one layer is too big to fit in the GPU global memory, we can split it into multiple smaller edge lists. Third, multiple layers can be combined together to fully utilise the computation power of GPU. We will discuss the last two advantages in more detail in a later section.

4 The WolfPath Framework

In this section, we first describe how WolfPath executes a traversing-based graph processing algorithm in parallel. Then we present how WolfPath tackle the issue of GPU under-utilisation and how WolfPath partition a large scale graph when it can not be fit in the GPU global memory.

4.1 Preprocessing

In this work, a preprocessing program is developed to transform a general graph format into a layered tree. This program first reads the general graph format into the CPU

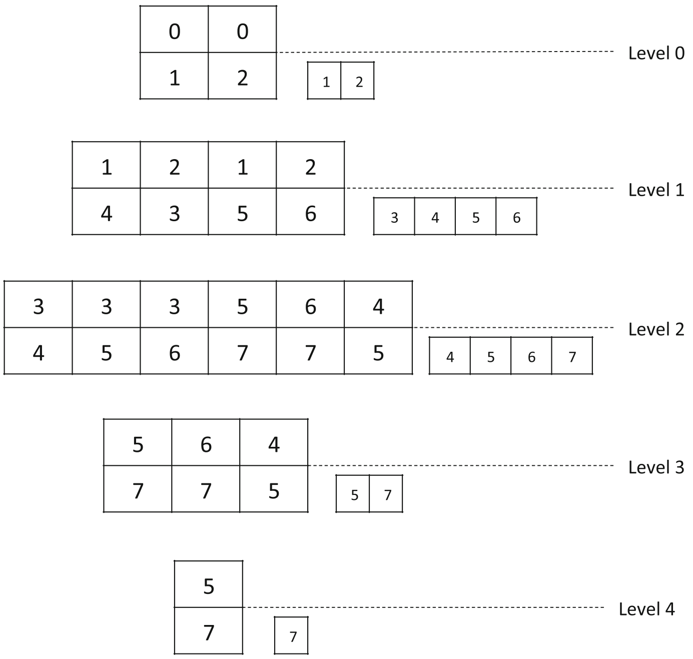


Fig. 2 An example of layered edge list structure for graph in Fig. 1a

memory and converts it into the CSR format. Next, it uses Algorithm 4 to build the layered tree, and then writes the layered tree back to a file stored in the disk. It is worth noting that this program is only run once for a graph. When processing a graph, WolfPath will first check if there exists a corresponding layered tree file. If the file exists, WolfPath will use it as the input. Otherwise, it will convert the graph into this new format. Algorithm 4 is used to build the layered tree.

Algorithm 4 Building layered tree

```
1: function BUILD_TREE(graph $G$, vertex $v_{rt}$)
2:     Tree $T \leftarrow null$
3:     Queue $Q \leftarrow null$
4:     $Q.enqueue(v_{rt})$
5:     $node\_level[V] \leftarrow 0$
6:     while $Q! = null$ do
7:         $v = Q.dequeue()$
8:         $level = node\_level[v]$
9:         for all neighbor $u$ of $v$ do
10:            if $u \notin T$ then
11:                $T.add\_edge(v, u)$
12:                $Q.enqueue(u)$
13:                $node\_level[u] + = 1$
14:            else if $u \in T$ && $u \in \{w \in T | w.level = level\}$ then
15:                $T.add\_edge(v, u)$
16:                $Q.enqueue(u)$
17:                $node\_level[u] + = 1$
```

Algorithm 4 is based on the breadth-first algorithm (BFS). The algorithm constructs an layered tree T for graph G (Line 2). It also creates Queue Q (Line 3). The algorithm starts with adding the vertex v_{rt} into Q (Line 4). In order to quickly retrieve the level information of each node, the algorithm also maintains an array called $node\_level$ (Line 5), which is used to store the level information of each node. The size of this array is equal to the number of vertices (denoted by V) in the graph G. This array is indexed by vertex id. The algorithm initialises all vertices in this array to 0 (Line 5). Then the algorithm performs the following steps iteratively (Line 6): it pops out the first vertex v from the queue (Line 7), and reads its level information from $node\_level$ (Line 8). $\forall e : v \rightarrow u \in E$ (Line 9), if $u \notin T$ (Line 10), or u has already been added into T but is in the same level as v (Line 14), the algorithm adds edge $\langle v, u \rangle$ in T (Line 11, 15). Next, the algorithm puts u in the queue by performing $enqueue(u)$ (Line 12, 16), sets the level of u to the current level plus 1 (Line 13 and 17). This process repeats until Q becomes empty.

4.2 The Computation Model of WolfPath

The computation process of WolfPath is as follows. In WolfPath, the graph is processed layer by layer. For each layer, three operations are performed by GPU in parallel: the read, compute and write operations. For i-th level in the graph, the read operation reads the updated vertex value from the global vertex array. The compute operation acts on each edge and uses the data gathered from the read operation to compute the value for its edge/vertices. The write operation writes the updated value to the global vertex value array. So the updated values can be used in next iteration by the read operation. Hence, The computation model in WolfPath is synchronous and guarantees that all updates from a previous compute phase are seen only after the write operation is completed and before the next read operation starts. The whole process terminates when all the levels have been processed, that is, the number of iterations is equal to the number of levels of the graph. This process is outlined in Algorithm 5.

Algorithm 5 Computation process of WolfPath

```
1: function PROCESSING(Layered Tree $T$, root vertex $v_{rt}$)
2:     $i \leftarrow 0$
3:     $v_{rt} = DEFAULT\_VALUE$
4:     while $i < T.level$ do
5:         if $i != 0$ then
6:             for all vertexes in parallel in level $i$ do
7:                 read vertex value from level $i - 1$
8:         for all Edges in parallel in level $i$ do
9:             compute the update value
10:        for all vertexes in parallel in level $i$ do
11:            write vertexes value to update value array
```

4.3 Combined Edge List

By representing the graph in the layered tree format, we can gain the knowledge about how many layers there are in the graph and use it to determine the number of iterations needed for most graph algorithms. However, because WolfPath processes the graph layer by layer and the graphs have the nature of irregular data structure, such representation may cause the under-utilisation of GPU.

Each layer in the graph may have different numbers of edges. This number varies dramatically between levels. For instance, consider the graph shown in Fig. 1. The first layer has 2 edges only. on the other hand, the second and third layer have 4 and 6 edges respectively. Hence, the number of threads required to process the first level is far less than the computing power (i.e., the number of processing cores) of GPU.

To overcome this problem, we propose the combined edge list, which is a large edge list that is constructed from multiple layers of edge lists. The combined edge list is constructed in the following way. We first define a number ME, which is the minimum amount of edges to be processed by GPU. Then we add the number of edges level by level starting from the first level of the layered tree. Once the total number of edges is greater or equal to ME, we group these levels together and then re-count the edges from the next level. This process repeats until all levels have been processed.

The way of building the combined edge list ensures that each combined edge list is formed by consecutive edge lists from the layered tree. Hence, each combined edge list can be treated as a sub-graph of the original graph. Therefore, the number of iterations required to process a combined edge list is equal to the number of levels used to form this tree. So Algorithm 5 is re-designed as Algorithm 6.

Algorithm 6 Processing the Combinaed Edge List

```
1: function PROCESSING(Layered Tree $T$, root vertex $v_{rt}$)
2:     $CEL \leftarrow build\_Combined\_Edge\_List(T)$
3:     $i \leftarrow 0$
4:     $v_{rt} = DEFAULT\_VALUE$
5:     while $i < CEL.count$ do
6:         $levels \leftarrow CES[i].level$
7:         while $j < level$ do
8:             parallel process all edges in $CEL[i]$
```

It is very important that we group the consecutive levels together to form a combined edge list. Because the result of the vertices in level i depends on those in level $i-1$, the results from the previous iteration need to be passed to the next iteration, which can be easily achieved by grouping the consecutive levels together. If we group the non-consecutive levels into a list, passing results between different levels requires lot of data transferring between host and GPU memory, which will harm the performance.

There remains one question: how do we choose the value for ME? If the number of ME is too small, the resulting edge list may not fully occupy the GPU. On the contrary, if it is too large, the size of the edge list may exceed the size of the GPU memory. Since it is desired that the GPU is fully occupied, the maximum active threads can be used to determine the minimum number of edges per combined edge list. The maximum number of active threads is the number of threads that a GPU can run simultaneously in theory, which can be found by Eq. 1, where N_{sm} is the number of multiprocessors (SMs), MW_{psm} is the maximum number of resident warps per SM and the T_{pw} is the threads per warp.

$$N_{sm} * MW_{psm} * T_{pw} \tag{1}$$

4.4 Out of GPU Memory Processing

Comparing with the host platform, the GPU has a limited memory space. The size of real world graphs may be from few gigabytes to terabytes, which are too large to fit in the GPU global memory. Therefore, in this section, we develop an out-of-GPU-memory engine that can process such large scale graphs.

The general process of developing an out-of-GPU-memory engine is to first partition the graph into sub-graphs that can fit into GPU memory, and then process these sub-graphs in GPU one at a time. Therefore, our first objective in designing such an engine is to achieve good performance in graph partitioning. Ideally, the performance of this process should be as close as possible to the performance of loading the graph into memory. The second objective is that after partitioning the graph, the graph framework has to deal with the frequent data exchange between GPU and host memory. Otherwise, the performance will take hit.

Based on the design of Layered Edge List, these two goals can be achieved. The process of partitioning the graph is similar to building the combined edge list. We start with the first layer and accumulate the vertices in the layers until the size of accumulated vertices becomes larger than the GPU globa memory. We group all the accumulated vertices as a sub-graph and start the accumulating process again from the current layer. The partitioning process is complete when we have processed all layers in the graph.

The complexity of such partitioning method is $\mathrm{O}(N)$, where N is the number of layers in the graph. Given the fact that most real world graphs, especially the social network graphs, do not have a big diameter, the number of N will not be very large.

After partitioning the graph, each sub-graph is processed in order based on their positions in the graph. That is, the processing starts with the sub-graph that contains the first layer. The next sub-graph to be processed is the one that follows the first sub-graph. In addition, the updated values of the vertices in the last layer of the current sub-graph need to be passed to the next sub-graph. This can be achieved by retaining the updated vertex values in the global memory after the computation is finished. Therefore, when next sub-graph is loaded into the GPU global memory, the data needed by the sub-graph is in the global memory. To process each sub-graph, we use the same method as that for in-GPU-memory processing. Combining multiple layers into a sub-graph enables us to fully utilise the GPU.

It is possible that the size of one layer is larger than the GPU memory. in this case, we can split this layer into multiple parts and compute one part at a time. This works because all the edges in a layer are independent to each other and it is therefore safe to partition a layer into multiple small chunks and process them separately.

4.5 Optimisation for GPU

When implementing WolfPath, we take advantage of the hardware architecture of GPU to optimise the performance. Specifically, WolfPath uses the shared memory to improve the performance of random access to the memory and performs computation and communication asynchronously.

4.5.1 Shared Memory

Writing the updated value into the *updated value array* will cause the random access to the global memory. To address this issue, we use the shared memory in GPU to store the vertex array, which can enhance the performance of random access, since the access to the shared memory is much faster than global memory.

The following method is used in WolfPath to use the shared memory effectively. The shared memory in GPU is shared between all the threads in a thread block. Therefore, the input edge list is split into small blocks in WolfPath, and a vertex array is constructed for each block. Each edge block is processed by one thread block. Multiple thread blocks are processed in parallel. In each thread block, edges are processed in parallel by the threads.

During the execution, the threads first fetch the updated vertex values into the shared memory of the block. The consecutive threads of the block read consecutive elements of the local vertex value array. Hence, reading requests are coalesced to the minimum number of memory transactions. After the computation, the threads first write the newly computed values into the shared memory, and then perform synchronisation between different thread blocks by writing the data from the shared memory to the global memory.

4.5.2 Asynchronous Execution

WolfPath asynchronously performs computation and communication. Specifically, it leverages the CUDA Streams and the hardware support such as Hyper-Qs provided by NVIDIAs Kepler to enable data streaming and computation in parallel. WolfPath creates multiple CUDA Streams to launch multiple kernels and overlap memory copying and computation in order to transfer data asynchronously. This is motivated by the fact that an edge list in WolfPath can be split into many sub-arrays. Each of these sub-arrays is independent to each other. WolfPath exploits this fact and does not move the entire edge list in a single copy performed by a CUDA stream. Instead, WolfPath creates multiple CUDA Streams to move these sub-arrays to the GPU. As the result, many hardware queues in GPU are used concurrently, which improves the overall throughput.

5 Experimental Evaluation

In this section, we evaluate the performance of WolfPath using two types of graph dataset: small sized graphs that can fit into the GPU global memory (called in-memory graphs in the experiments) and large scale graphs that do not fit (out-of-memory graphs). The size of a graph is defined as the amount of memory required to store the edges, vertices, and edge/vertices values in user-defined datatypes.

Small graphs are used to evaluate WolfPath's performance in in-memory graph processing against other state-of-art in-memory graph processing systems (e.g., CuSha [16] and Virtual-Warp-Centric [13], and the large graphs are used to compare WolfPath

Table 4 Real world graphs used in the experiments

Graph	Vertices	Edges
GPU Out-of-memory Graph		
orkut [20]	3,072,441	117,185,083
hollywood2011 [2]	2,180,653	228,985,632
arabic2005 [1]	22,743,892	639,999,458
uk2002 [2]	18,520,486	298,113,762

with other out-of-core frameworks that can process large graphs on a single PC (e.g., GraphChi and X-Stream).

The eight graphs listed in Tables 1 and 4 are publicly available. They cover a broad range of sizes and sparsity and come from different real-world origins. For example, *Live-Journal* is directed social networks graph. *RoadNetCA* is the California road network. *orkut* is an undirected social network graph. *uk-2002* is a large crawl of the .uk domains, in which vertices are the pages and edges are links.

We choose two widely used searching algorithms to evaluate the performance, namely Breadth First Search (BFS) and Single Source Shortest Paths (SSSP).

The experiments were conducted on a system with a Nvidia GeForce GTX 780Ti graphic card, which has 12 SMX multiprocessors and 3 GB GDDR5 RAM. On the host side, we use the Intel Core i5-3570 CPU operating at 3.4 GHZ with 32 GB DDR3 RAM. The benchmarks were evaluated using CUDA 6.5 on Fedora 21. All the programs were compiled with the highest optimisation level (-O3).

5.1 Performance Evaluation

5.1.1 Comparison with Existing In-Memory Graph Processing Frameworks

In this section, we compare WolfPath with the state-of-art in-memory processing solutions such as CuSha [16] and Virtual Warp Centric [13]. In the experiments, we use the CuSha-CW method, because this strategy provides the best performance in all CuSha strategies. Both CuSha and Virtual Warp Centric apply multi-level optimisations to the in-memory workloads.

We first compare the computation times among WolfPath, CuSha and VWC. Figures 3 and 4 show the speedup of WolfPath over CuSha and VWC.

We also list the breakdown performances in Fig. 5. In these experiments, the Data Transfer is time taken to move data from the host to GPU, the computation refers to the time taken for actual execution of the algorithm. We also recorded the number of iterations that each algorithm takes in different systems, these results are listed in Table 5.

As can be seen from these results, WolfPath outperforms CuSha and Virtual Warp Centric. Although WolfPath requires more iteration to complete the algorithm, the average speedup of WolfPath over CuSha is more than 100×, and 400× over VWC. This is due to the elimination of memory copy operations. Also, the performance of VWC is the worst among 3 systems, because VWC does not guarantee the coa-

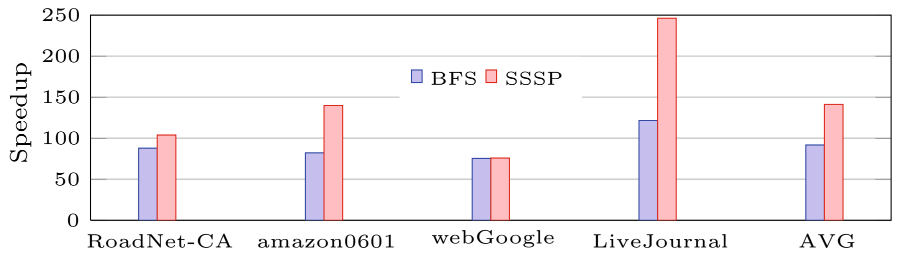


Fig. 3 Speedup over CuSha

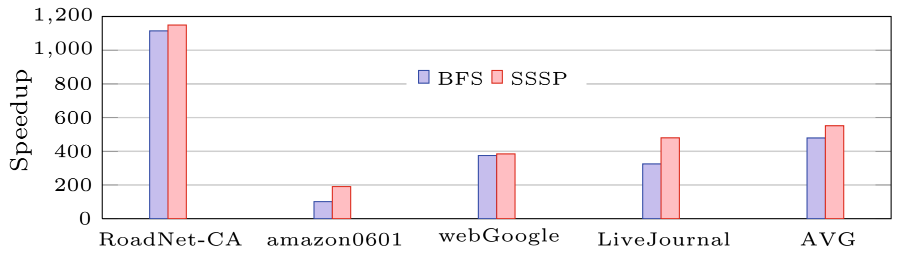


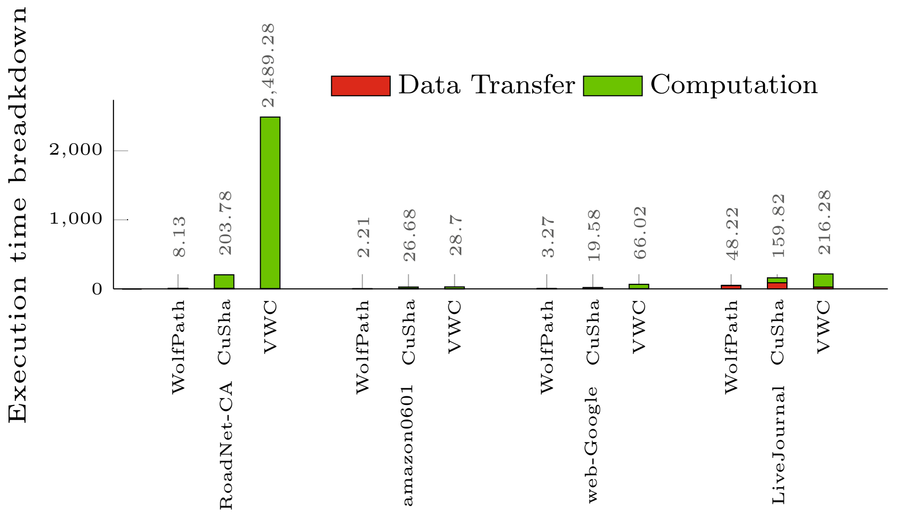
Fig. 4 Speedup over VWC

lesced memory access. On the other hand, with carefully designed data structures, both WolfPath and CuSha can access graph edge in a sequential manner, hence memory performance is much better. We also noticed that WolfPath takes more iterations than other implementations. This is because in iterative computation model, the computation can be converged very fast, but WolfPath is bounded by the graph diameter, which is the upper bound of the iteration count. However, as discussed in the previous section, compare to memory copy operation, computation is much faster. Therefore, WolfPath still outperforms other systems.

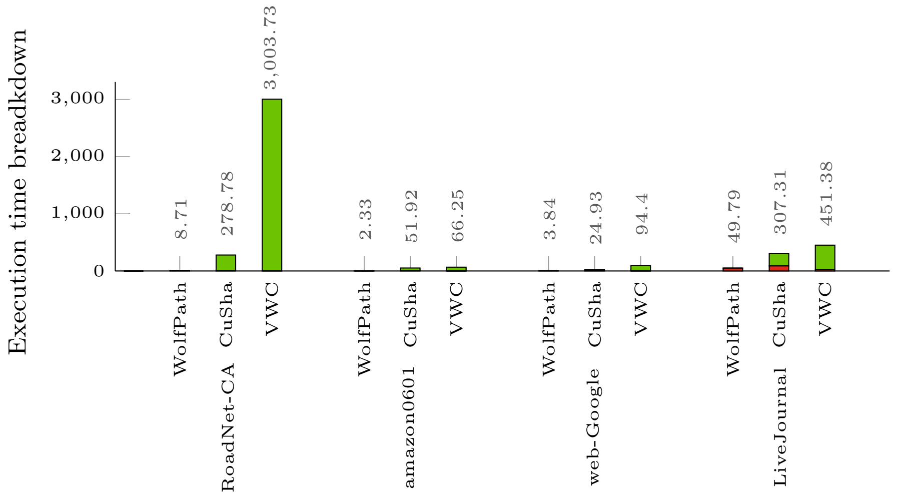
5.1.2 Comparison with GPU Out-of-Memory Frameworks

The results shown in the last section demonstrate WolfPath's performance in processing in-memory graphs. However, many real-world graphs are too large to fit in GPU memory. In this section, we examine the WolfPath's ability to process large graphs which cannot fit into GPU memory. To the best of our knowledge, the state-of-art GPU-based graph processing frameworks [6,16,47] assume that the input graphs can fit in the GPU memory. Therefore, in this work, we compare WolfPath (WP) with two CPU-based, out-of-memory graph processing framework: GraphChi (GC) [17] and X-Stream (XS) [33]. To avoid disk I/O overhead in systems such as GraphChi and X-Stream, the dataset selected in the experiments can fit in host memory but not in GPU memory.

As shown in Figs. 6 and 7, WolfPath achieves an average speedup of 3000$\times$ and 4000$\times$ over GraphChi and X-Stream (running with 4 threads), respectively, despite its need to move the data between GPU and CPU. We also list the computation time and iteration counts of three systems in Table 6. Since X-Stream does not require any



(a)



(b)

Fig. 5 Execution time breakdown of WolfPath, CuSha and VWC on different algorithms and graphs. The time unit is milliseconds. **a** BFS and **b** SSSP

pre-processing and the computation is overlapped with I/O operations, we use the total execution time of the system as the comparison.

As can be seen from the Table 6, although WolfPath performs more iterations than GraphChi and X-stream, it still outperforms them. This performance improvement is due to the massive parallel processing power provided by GPU, while GraphChi and X-Stream are CPU-based and their degrees of parallelism are limited.

Table 5 Number of iterations executed by different systems and total execution times (ms)

	Iterations			Total execution time		
	WolfPath	CuSha	VWC	WolfPath	CuSha	V WC
BFS						
RoadNet-CA	554	551	486	8.1	203.75	2489.28
amazon0601	35	35	18	2.21	26.68	28.7
web-Google	32	21	18	3.27	19.58	66.02
LiveJournal	14	9	8	48.22	159.82	216.28
SSSP						
RoadNet-CA	554	550	465	8.71	278.78	3003.73
amazon0601	35	35	14	2.33	51.92	66.25
web-Google	32	20	17	3.84	24.93	94.4
LiveJournal	14	9	7	49.79	307.31	450.93

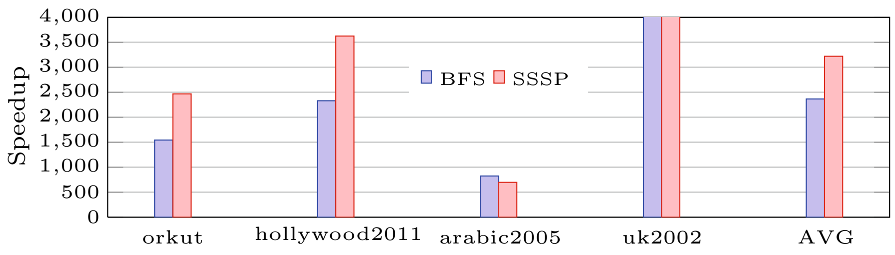


Fig. 6 Speedup over GraphChi

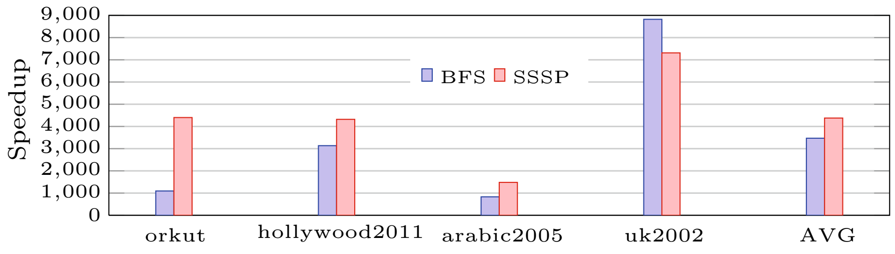


Fig. 7 Speedup over X-Stream

5.2 Memory Occupied by Different Graph Representation

From Fig. 5, we can see that VWC has the shortest data transfer time. This is because it represents the graph in the CSR format, which is memory efficient. However, in order to have sequential access to the edges, both WolfPath and CuSha represent graphs with edges, which consume more memory space than CSR. In this section, we evaluate the cost of using the Layered Edge list representation in terms of required memory space against CSR and CuSha's CW representation.

Table 6 Execution times of WolfPath, GraphChi and X-Stream on different algorithms and graphs. The time is in seconds

	Computation			Iteration		
	WolfPath	GraphChi	X-Stream	WolfPath	GraphChi	X-Stream
BFS						
orkut	0.02	30.88	21.88	7	2	2
hollywood2011	0.09	209.86	282.17	16	8	13
arabic2005	0.2	164.92	166.18	51	3	3
uk2002	0.15	715.38	1323.59	49	28	48
SSSP						
orkut	0.02	49.38	88.02	7	3	3
hollywood2011	0.1	362.56	432.24	16	9	16
arabic2005	0.23	160.56	340.94	51	3	7
uk2002	0.16	974.83	1170.32	49	31	42

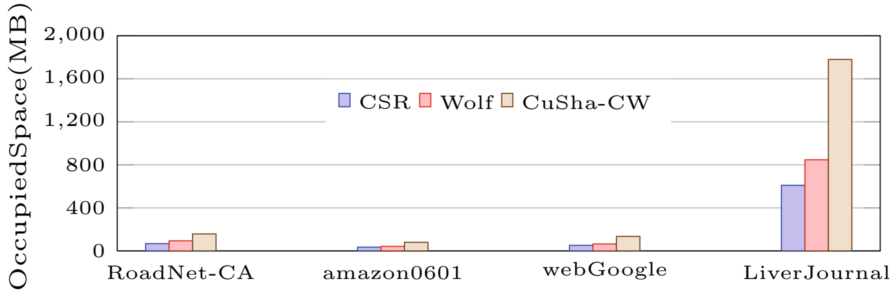


Fig. 8 Memory occupied by each graph using CSR, CuSha-CW, WolfPath

Figure 8 shows the memory consumed by WolfPath's Layered Edge List, CuSha-CW and CSR. The Layered Edge List and CuSha-CW need 1.37× and 2.81× more space on average than CSR. CuSha uses 2.05× more memory than WolfPath, because it represents each edge with 4 arrays.

5.3 Preprocessing Time

Table 7 shows the preprocessing time of WolfPath, CuSha, and GraphChi. The preprocessing time refers to the time taken to convert the graph from the raw data to the framework specified format (e.g, layered tree in WolfPath or Shard in GraphChi). It consists of the graph traversing time and the time to write the data into the storage. Because CuSha is unable to process the graph larger than GPU memory, the corresponding cells in the table are marked as NA.

The first observation from the table is that (1) for in-memory graphs, CuSha preprocesses the data faster than other two systems, and (2) WolfPath is the slowest system. This is because CuSha does not write the processed data back to the disk. GraphChi

Table 7 Preprocessing time (s)

	WolfPath	CuSha	GraphChi
Amazon0601	0.79	0.24	0.58
LiverJournal	14.68	3.8	10.15
WebGoogle	1.46	0.44	0.79
orkut	21.4	NA	22.07
hollywood	38.6	NA	34
uk2002	59.78	NA	69
arabic2005	120.3	NA	151.5

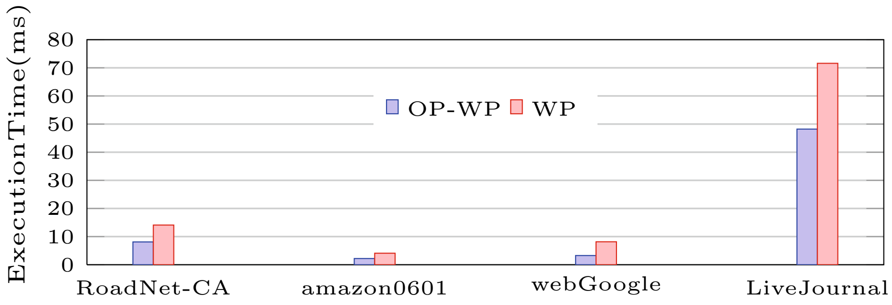


Fig. 9 Performance comparison with and without optimisation

will only write a copy of data into shard. In contrast, WolfPath traverses the data using the BFS-based algorithm and then writes the data into a temporary buffer before it writes the data to the hard disk. Therefore, the workload of WolfPath is heavier than two other systems.

For graphs larger than GPU memory, WolfPath performs better than GraphChi when processing uk2002 and arabic2005. This is because GraphChi generates many shard files for these two graphs, and hence it takes longer to write to the disk.

From this experiment, we argue that in WolfPath, although the preprocessing is time-consuming, the pre-processing is worthwhile because of the following reasons: First, for each graph, WolfPath only needs to convert it once. Second, the resultant format provides better locality and performance for iterative graph computations.

5.4 Effect of Optimisation Techniques

As shown in previous experimental results, the memory accessing operations are the dominant factor that affect the performance, and therefore are our primary target of optimization. Figure 9 shows that the performance improves significantly thanks to the 2 optimization techniques discussed in Sect. 4.5, including asynchronous execution and shared memory synchronisation. For example, without these optimization techniques, the execution time of the BFS algorithm is 71ms with the Livejournal graph. With the optimisation the execution time drops to 48ms.

6 Related Work

Using GPUs for graph processing was first introduced by Harish and Narayanan [12]. Since then, the CSR format has become the mainstream representation to store graphs on GPU. Merrill et al. [29] present a work efficient BFS algorithm. They use different approaches to minimize the workload imbalance. Virtual Warp Centric was proposed in [13] to tackle the workload imbalance problem and reduce the intra-warp divergence.

Medusa [47] is a GPU-based graph processing framework that focuses on abstractions for easy programming. MapGraph [6] implements the runtime-based optimisation to deliver good performance. Based on the size of the frontier and the size of the adjacency lists for the vertices in the frontier, MapGraph can choose from different scheduling strategies.

The graph processing solutions described above use the CSR format to represent the graph, hence suffering from the random access to the graph data. CuSha [16] addresses the inefficient memory access by introducing the G-Shard and Concatenated Windows. However, as shown in this paper, CuSha requires frequent data exchange between host and GPU, which leads to long overall execution time.

All of the approaches above make the fundamental assumption that the input graphs fit into GPU memory, which limits the usage of these solutions. However, WolfPath does not suffer from such restriction.

Most existing Out-of-Memory graph processing frameworks are CPU based, these frameworks are aiming to process graphs that do not fit into host memory. For instance, GraphChi [17] is the first disk-based graph processing framework and designed to run on a single machine with limited memory. X-Stream graph processing framework [33] uses the edge-centric processing model that takes as input a binary formatted edge-list. It does not require preprocessing, but requires frequent disk I/O to fetch data.

Totem [7,8] is a hybrid platform that uses both GPU and CPU. It statically partitions a graph into the parts of residing in GPU and host memories based on the degree of vertices. However, as the size of the graph increases, only a fixed portion of the graph is able to fit in the GPU memory, resulting in GPU underutilization. GraphReduce [34] aims to process the graphs that exceed the capacity of the GPU memory. It partitions the graph into shards and loads one or more shards into the GPU memory at a time. In GraphReduce, each shard contains a disjoint subset of vertices. The edges in each shard are sorted in a specific order.

7 Conclusion

In this paper, we develop WolfPath: a GPU-based graph processing framework for processing iterative traversing-based graph processing algorithms efficiently. A new data structure called Layered Edge List is introduced to represent the graph. This structure helps eliminate the frequent data exchange between host and GPU. We also propose a graph preprocessing algorithm that can convert an arbitrary graph into the layered structure. The experimental results show that WolfPath achieves the significant speedup over the state-of-art in-GPU-memory and out-of-memory graph processing frameworks.

Acknowledgements This work is partially supported by the EU Horizon 2020—Marie Sklodowska-Curie Actions through the project entitled Computer Vision Enabled Multimedia Forensics and People Identification (Project No. 690907, Acronym: IDENTITY), Guangzhou City Science Foundation, (Project No. 201510010275), the Natural Science Foundation of China (Grant Numbers. 61370226, 61672156), PAPD and CICAEET.

References

1. Boldi, P., Rosa, M., Santini, M., Vigna, S.: Layered label propagation: a multiresolution coordinate-free ordering for compressing social networks. In: Srinivasan, S., Ramamritham, K., Kumar, A., Ravindra, M.P., Bertino, E., Kumar, R. (eds.) Proceedings of the 20th International Conference on World Wide Web, pp. 587–596. ACM Press, New york (2011)
2. Boldi, P., Vigna, S.: The WebGraph framework I: Compression techniques. In: Proceedings of the Thirteenth International World Wide Web Conference (WWW 2004), pp. 595–601. ACM Press, Manhattan, USA (2004)
3. Chen, B., Yang, J., Jeon, B., Zhang, X.: Kernel quaternion principal component analysis and its application in rgb-d object recognition. Neurocomputing (2017). https://doi.org/10.1016/j.neucom.2017.05.047
4. Chen, C., Li, K., Ouyang, A., Tang, Z., Li, K.: Gpu-accelerated parallel hierarchical extreme learning machine on flink for big data. IEEE Trans. Syst. Man Cybern. Syst. **PP**(99), 1–14 (2017). https://doi.org/10.1109/TSMC.2017.2690673
5. Corporation, N.: NVIDIA CUDA C Programming Guide. NVIDIA Corporation, version 9.0.176 edn (2017). http://docs.nvidia.com/cuda/cuda-c-programmingguide/index.html#axzz4atgDRVPb
6. Fu, Z., Personick, M., Thompson, B.: Mapgraph: a high level api for fast development of high performance graph analytics on gpus. In: Proceedings of Workshop on GRAph Data Management Experiences and Systems, GRADES'14, pp. 2:1–2:6. ACM, New York, NY, USA (2014). https://doi.org/10.1145/2621934.2621936
7. Gharaibeh, A., Beltrão Costa, L., Santos-Neto, E., Ripeanu, M.: A yoke of oxen and a thousand chickens for heavy lifting graph processing. In: Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques, PACT '12, pp. 345–354. ACM, New York, NY, USA (2012). https://doi.org/10.1145/2370816.2370866
8. Gharaibeh, A., Costa, L.B., Santos-Neto, E., Ripeanu, M.: On graphs, gpus, and blind dating: a workload to processor matchmaking quest. In: Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing, IPDPS '13, pp. 851–862. IEEE Computer Society, Washington, DC, USA (2013). https://doi.org/10.1109/IPDPS.2013.37
9. Gonzalez, J.E., Low, Y., Gu, H., Bickson, D., Guestrin, C.: Powergraph: distributed graph-parallel computation on natural graphs. In: Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation, OSDI'12, pp. 17–30. USENIX Association, Berkeley, CA, USA (2012). http://dl.acm.org/citation.cfm?id=2387880.2387883
10. Gonzalez, J.E., Xin, R.S., Dave, A., Crankshaw, D., Franklin, M.J., Stoica, I.: Graphx: graph processing in a distributed dataflow framework. In: Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation, OSDI'14, pp. 599–613. USENIX Association, Berkeley, CA, USA (2014). http://dl.acm.org/citation.cfm?id=2685048.2685096
11. Gu, B., Sheng, V.S., Wang, Z., Ho, D., Osman, S., Li, S.: Incremental learning for v-support vector regression. Neural Netw. **67**(C), 140–150 (2015)
12. Harish, P., Narayanan, P.J.: Accelerating large graph algorithms on the gpu using cuda. In: Proceedings of the 14th International Conference on High Performance Computing, HiPC'07, pp. 197–208. Springer-Verlag, Berlin, Heidelberg (2007). http://dl.acm.org/citation.cfm?id=1782174.1782200
13. Hong, S., Kim, S.K., Oguntebi, T., Olukotun, K.: Accelerating cuda graph algorithms at maximum warp. In: Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Program-

ming, PPoPP '11, pp. 267–276. ACM, New York, NY, USA (2011). https://doi.org/10.1145/1941553.1941590

14. Kang, U., Tsourakakis, C.E., Appel, A.P., Faloutsos, C., Leskovec, J.: Hadi: Mining radii of large graphs. ACM Trans. Knowl. Discov. Data **5**(2), 8:1–8:24 (2011). https://doi.org/10.1145/1921632.1921634
15. Kang, U., Tsourakakis, C.E., Faloutsos, C.: Pegasus: a peta-scale graph mining system implementation and observations. In: Proceedings of the 2009 Ninth IEEE International Conference on Data Mining, ICDM '09, pp. 229–238. IEEE Computer Society, Washington, DC, USA (2009). https://doi.org/10.1109/ICDM.2009.14
16. Khorasani, F., Vora, K., Gupta, R., Bhuyan, L.N.: Cusha: vertex-centric graph processing on gpus. In: Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing, HPDC '14, pp. 239–252. ACM, New York, NY, USA (2014). https://doi.org/10.1145/2600212.2600227
17. Kyrola, A., Blelloch, G., Guestrin, C.: Graphchi: large-scale graph computation on just a pc. In: Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation, OSDI'12, pp. 31–46. USENIX Association, Berkeley, CA, USA (2012). http://dl.acm.org/citation.cfm?id=2387880.2387884
18. Lattanzi, S., Moseley, B., Suri, S., Vassilvitskii, S.: Filtering: a method for solving graph problems in mapreduce. In: Proceedings of the Twenty-Third Annual ACM Symposium on Parallelism in Algorithms and Architectures, SPAA '11, pp. 85–94. ACM, New York, NY, USA (2011). https://doi.org/10.1145/1989493.1989505
19. Shen, J., Tan, H.W., Wang, J., Wang, J.W., Sungyoung, L.: A novel routing protocol providing good transmission reliability in underwater sensor networks. J. Internet Technol. **16**, 171–178 (2015). https://doi.org/10.6138/JIT.2014.16.1.20131203e
20. Leskovec, J., Krevl, A.: SNAP Datasets: Stanford large network dataset collection (2014). http://snap.stanford.edu/data
21. Li, J., Li, X., Yang, B., Sun, X.: Segmentation-based image copy-move forgery detection scheme. IEEE Trans. Inf. Forensics Secur. **10**(3), 507–518 (2015)
22. Li, K., Tang, X., Li, K.: Energy-efficient stochastic task scheduling on heterogeneous computing systems. IEEE Trans. Parallel Distrib. Syst. **25**(11), 2867–2876 (2014). https://doi.org/10.1109/TPDS.2013.270
23. Li, K., Tang, X., Veeravalli, B., Li, K.: Scheduling precedence constrained stochastic tasks on heterogeneous cluster systems. IEEE Trans. Comput. **64**(1), 191–204 (2015). https://doi.org/10.1109/TC.2013.205
24. Liu, J., Li, K., Zhu, D., Han, J., Li, K.: Minimizing cost of scheduling tasks on heterogeneous multicore embedded systems. ACM Trans. Embed. Comput. Syst. **16**(2), 36:1–36:25 (2016). https://doi.org/10.1145/2935749
25. Low, Y., Gonzalez, J., Kyrola, A., Bickson, D., Guestrin, C., Hellerstein, J.M.: Graphlab: a new framework for parallel machine learning. CoRR (2010). arXiv:1006.4990
26. Ma, T., Zhou, J., Tang, M., Tian, Y., Al-Dhelaan, A., Al-Rodhaan, M., Lee, S.: Social network and tag sources based augmenting collaborative recommender system. IEICE Trans. Inf. Syst. **98**(4), 902–910 (2015)
27. Macko, P., Marathe, V.J., Margo, D.W., Seltzer, M.I.: Llama: efficient graph analytics using large multiversioned arrays. In: 2015 IEEE 31st International Conference on Data Engineering, pp. 363–374 (2015). https://doi.org/10.1109/ICDE.2015.7113298
28. Malewicz, G., Austern, M.H., Bik, A.J., Dehnert, J.C., Horn, I., Leiser, N., Czajkowski, G.: Pregel: a system for large-scale graph processing. In: Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data, SIGMOD '10, pp. 135–146. ACM, New York, NY, USA (2010). https://doi.org/10.1145/1807167.1807184
29. Merrill, D., Garland, M., Grimshaw, A.: Scalable gpu graph traversal. In: Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '12, pp. 117–128. ACM, New York, NY, USA (2012). https://doi.org/10.1145/2145816.2145832
30. Pan, Z., Jin, P., Lei, J., Zhang, Y., Sun, X., Kwong, S.: Fast reference frame selection based on content similarity for low complexity hevc encoder. J. Vis. Commun. Image Represent. **40–B**, 516–524 (2016)
31. Plimpton, S.J., Devine, K.D.: Mapreduce in MPI for large-scale graph algorithms. Parallel Comput. **37**(9), 610–632 (2011). (Emerging Programming Paradigms for Large-Scale Scientific Computing)

32. Rong, H., Ma, T., Tang, M., Cao, J.: A novel subgraph k+ -isomorphism method in social network based on graph similarity detection. Soft Comput. (2017). https://doi.org/10.1007/s00500-017-2513-y
33. Roy, A., Mihailovic, I., Zwaenepoel, W.: X-stream: edge-centric graph processing using streaming partitions. In: Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles, SOSP '13, pp. 472–488. ACM, New York, NY, USA (2013). https://doi.org/10.1145/2517349.2522740
34. Sengupta, D., Song, S.L., Agarwal, K., Schwan, K.: Graphreduce: processing large-scale graphs on accelerator-based systems. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC '15, pp. 28:1–28:12. ACM, New York, NY, USA (2015). https://doi.org/10.1145/2807591.2807655
35. Shi, X., Luo, X., Liang, J., Zhao, P., Di, S., He, B., Jin, H.: Frog: asynchronous graph processing on gpu with hybrid coloring model. IEEE Trans. Knowl. Data Eng. **PP**(99), 1 (2017). https://doi.org/10.1109/TKDE.2017.2745562
36. Shi, X., Zheng, Z., Zhou, Y., Jin, H., He, L., Liu, B., Hua, Q.S.: Graph processing on gpus: a survey. ACM Comput. Surv. (2017). https://staticcuris.ku.dk/portal/files/182749408/gpusurvey.pdf
37. Stone, J.E., Gohara, D., Shi, G.: Opencl: a parallel programming standard for heterogeneous computing systems. IEEE Des. Test **12**(3), 66–73 (2010). https://doi.org/10.1109/MCSE.2010.69.
38. Wang, J., Li, T., Shi, Y., Lian, S., Ye, J.: Forensics feature analysis in quaternion wavelet domain for distinguishing photographic images and computer graphics. Multimed. Tools Appl. (2017). https://doi.org/10.1007/s11042-016-4153-0
39. Wang, K., Xu, G., Su, Z., Liu, Y.D.: Graphq: graph query processing with abstraction refinement—scalable and programmable analytics over very large graphs on a single pc. In: 2015 USENIX Annual Technical Conference (USENIX ATC 15), pp. 387–401. USENIX Association, Santa Clara, CA (2015). https://www.usenix.org/conference/atc15/technical-session/presentation/wang-kai
40. Wang, Y., Davidson, A., Pan, Y., Wu, Y., Riffel, A., Owens, J.D.: Gunrock: a high-performance graph processing library on the gpu. In: Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2015, pp. 265–266. ACM, New York, NY, USA (2015). https://doi.org/10.1145/2688500.2688538
41. Xiong, L., Xu, Z., Shi, Y.: An integer wavelet transform based scheme for reversible data hiding in encrypted images. Multidimension. Syst. Signal Process. (2017). https://doi.org/10.1007/s11045-017-0497-5
42. Yang, W., Li, K., Mo, Z., Li, K.: Performance optimization using partitioned spmv on gpus and multicore cpus. IEEE Trans. Comput. **64**(9), 2623–2636 (2015). https://doi.org/10.1109/TC.2014.2366731
43. Yuan, C., Sun, X., Lv, R.: Fingerprint liveness detection based on multi-scale lpq and pca. China Commun. **13**(7), 60–65 (2016)
44. Yuan, C., Xia, Z., Sun, X.: Coverless image steganography based on sift and bof. J. Internet Technol. **18**(2), 435–442 (2017)
45. Yuan, P., Zhang, W., Xie, C., Jin, H., Liu, L., Lee, K.: Fast iterative graph computation: a path centric approach. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC '14, pp. 401–412. IEEE Press, Piscataway, NJ, USA (2014). https://doi.org/10.1109/SC.2014.38
46. Zhang, Y., Sun, X., Wang, B.: Efficient algorithm for k-barrier coverage based on integer linear programming. China Commun. **13**(7), 16–23 (2016)
47. Zhong, J., He, B.: Medusa: simplified graph processing on gpus. IEEE Trans. Parallel Distrib. Syst. **25**(6), 1543–1552 (2014). https://doi.org/10.1109/TPDS.2013.111.
48. Zhou, Z., Wang, Y., Wu, J., Yang, C., Sun, X.: Effective and efficient global context verification for image copy detection. IEEE Trans. Inf. Forensics Secur. **12**(1), 48–63 (2017)
49. Zhou, Z., Wu, J., Huang, F., Sun, X.: Fast and accurate near-duplicate image elimination for visual sensor networks. Int. J. Distrib. Sens. Netw. **13**(2), 435–442 (2017). https://doi.org/10.1177/1550147717694172
50. Zhou, Z., Yang, C., Chen, B., Sun, X., Liu, Q., Wu, Q.J.: Effective and efficient image copy detection with resistance to arbitrary rotation. IEICE Trans. Inf. Syst. **E99–D**(6), 1531–1540 (2016)
51. Zhu, X., Han, W., Chen, W.: Gridgraph: Large-scale graph processing on a single machine using 2-level hierarchical partitioning. In: 2015 USENIX Annual Technical Conference (USENIX ATC 15), pp. 375–386. USENIX Association, Santa Clara, CA (2015). https://www.usenix.org/conference/atc15/technical-session/presentation/zhu

52. Zhu, X., Li, K., Salah, A., Shi, L., Li, K.: Parallel implementation of mafft on cuda-enabled graphics hardware. IEEE/ACM Trans. Comput. Biol. Bioinf. **12**(1), 205–218 (2015). https://doi.org/10.1109/TCBB.2014.2351801