WolfGraph: the Edge-Centric graph processing on GPU

Huanzhou Zhu[a,1], Ligang He[b,*], Matthew Leeke[b], Rui Mao[c]

[a]*Department of Computing, Imperial College London*
[b]*Department of Computer Science, University of Warwick*
[c]*College of Computer Science and Software Engineering, ShenZhen University*

Abstract

There is the significant interest nowadays in developing the frameworks for parallelizing the processing of large graphs such as social networks, web graphs, etc. The work has been proposed to parallelize the graph processing on clusters (distributed memory), multicore machines (shared memory) and GPU devices. Most existing research on GPU-based graph processing employs the vertex-centric processing model and the Compressed Sparse Row (CSR) form to store and process a graph. However, they suffer from irregular memory access and load imbalance in GPU, which hampers the full exploitation of GPU performance. In this paper, we present WolfGraph, a GPU-based graph processing framework that addresses the above problems. WolfGraph adopts the edge-centric processing, which iterates over the edges rather than vertices. The data structure and graph partition in WolfGraph are carefully crafted so as to minimize the graph pre-processing and allow the coalesced memory access. WolfGraph fully utilizes the GPU power by processing all edges in parallel. We also develop a new method, called Concatenated Edge List (CEL), to process a graph that is bigger than the global memory of GPU. WolfGraph allows the users to define their own graph-processing methods and plug them into the WolfGraph framework. Our experiments show that WolfGraph achieves 7-8x speedup over GraphChi and X-Stream when processing large graphs, and it also offers 65% performance improvement over the existing GPU-based, vertex-centric graph processing frameworks, such as Gunrock.

Keywords: GPGPU, Graph Processing, CUDA, Parallel Processing

1. Introduction

The demand for efficiently processing large-scale graphs has been growing recently. This is because graphs can be used to describe a wide range of objects, and computations on graph-based data structures are the core of many applications. Motivated by the need to process very large graphs, many frameworks have been developed for processing large graphs on distributed systems. Such frameworks include Pregel[1], GraphLab[2], PowerGraph[3], GraphX [4], Chaos [5] and Gemini [6]. However, since developing efficient distributed graph algorithms is challenging, some research studies aim to design the graph processing systems that can handle large graphs (with billions of edges) on a single PC. The results of these studies are Wonderland [7], Mosaic [8], PathGraph [9], GraphQ [10], LLAMA [11] and GridGraph [12], GraphChi[13]. However, these systems suffer from limited degree of parallelism provided by conventional processors. To overcome this problem, much research employs GPU to accelerate graph processing due to its massively parallel architecture, such as Tigr [14], Frog [15], GTS [16], Gunrock [17], CuSha [18], MultiGraph [19] and Virtual Warp [20].

Currently, using GPU for efficient graph processing remains a challenging and open problem due to the following reasons. First, although GPU provides a massive degree of parallelism compared to CPU, its hardware architecture requires regular data access pattern (coalesced memory access) to achieve the peak performance. However, most graphs are of highly irregular structure, which leads to the problems of irregular (random) memory accesses and load imbalance in GPU and consequently limits the performance of graph algorithms

*Corresponding author
Email address: ligang.he@warwick.ac.uk (Ligang He)
[1]This work was done when the author studied at the University of Warwick

on GPU. For example, most existing graph processing techniques [21] [22] [23] [24] employ the vertex-centric processing and rely on the CSR (Compressed Sparse Row) graph representation. Due to the poor locality in the CSR representation, visiting a node's neighbours usually leads to random memory access. Second, the existing work of GPU-based graph processing assumes the entire graph can fit into the global memory of GPU. However, some large-scale graphs are even bigger than GPU memory, which makes these work infeasible to process those graphs. Third, because GPU has the limited global memory space compared with CPU, it requires frequent data copying between device memory and host memory, which also results in poor performance. Finally, as pointed out by [25], [26], [27] in the state-of-the-art graph processing systems, the time spent in reading a graph from hard disks to CPU memory and constructing the necessary data structure in memory for processing the graph, which is called the pre-processing time, constitutes a big proportion of the total processing time for a large graph. Reducing this pre-processing time will significantly improve the overall performance of graph processing frameworks.

In this paper, we present WolfGraph, a framework for processing large graphs on GPUs, to address the above problems. The framework partitions a graph, construct the data structures to store the graph, and parallelize the graph processing on GPU. WolfGraph provides the interface for the programmers to implement their own graph processing algorithms using CUDA (e.g., the Breadth-First-Search algorithm). WolfGraph then loads the graph to be processed from the hard disk to the host memory, partitions the graph into blocks of graph edges (each of which is called an edge block in this paper) with minimal pre-processing time, launches the kernel function (containing the programmers' graph processing implementation) with blocks of threads, and uses thread blocks to run edge blocks in parallel.

WolfGraph adapts a recently introduced graph processing model, known as edge-centric processing [28] to efficiently process the graphs on GPUs. In this model, the graph is represented as an unordered list of edges. The processing iterates over edges rather than vertices. More specifically, after the graph is split into edge blocks, each edge block contains an unordered list of edges, which are contiguously stored in memory, and a set of vertices associated with the edges in this block. Each edge block is processed by a thread block in GPU and multiple thread blocks are processed by edge blocks in parallel. Within each thread block, the edges are processed in parallel by the threads. Such allocation of edge blocks to thread blocks enables the coalesced memory access to the edges in GPU global memory. In WolfGraph, the access to vertices are still random. But the data structure for holding vertices is placed in the shared memory of GPU, the access to which is much faster than to the global memory. Moving the data structure of vertices to the share memory also significantly reduces the synchronization overhead among threads during the graph processing.

In most existing work, the graph is preprocessed before the graph processing algorithm is applied. The idea is that although it takes the time to preprocess a graph, the execution of the graph processing algorithm will take much less time than without preprocessing and therefore the overall processing time will be reduced significantly. Our survey shows that pre-processing a graph consumes a big proportion of the total processing time of the graph. The main novelty and contribution of this work are designing a GPU-based graph processing framework that endures minimal preprocessing. We carefully analyze the benefits brought by the existing graph preprocessing methods in both GPU-based [18] and CPU-based [13] graph processing frameworks, and then develop a new GPU-based method to accomplish the efficient execution with minimal preprocessing. The new method includes careful design of the GPU data structure for graph processing, full exploitation of GPU memory hierarchy and thread synchronization, and crafty strategy of thread parallelization and consequent coalesced memory access by threads. Comparing with the existing GPU-based framework [18][20], our method can achieve similar execution time although minimal preprocessing is carried out. Therefore, our graph processing framework reduces the overall processing time significantly.

The above processing handles the graph that can fit into the GPU global memory, which is called "in-memory" graph processing. For a graph larger than the GPU global memory, we develop the "out-of-memory" processing. The method developed for in-memory processing still acts as the core of the out-of-memory processing. However, we first partition the graph into sub-graphs with minimal efforts. Each sub-graph can fit into the global memory. Each sub-graph is processed in the edge-centric

manner by GPU. The advantage of such method is its minimal pre-processing time. However, it is at the expense of potential frequent copying of sub-graphs between GPU global memory and host memory. To address this problem, we develop a new method called the Concatenate Edge List (CEL), which is inspired by GraphChi [13], for out-of-memory processing. When the CEL method loads a sub-graph into the GPU global memory, for each vertex that is a destination vertex in this sub-graph, it also loads into GPU the edges that have this vertex as a source. With the CEL method, we only need to load each sub-graph into GPU once. New supporting data structure is also designed in order to enable efficient construction of the concatenated edge list.

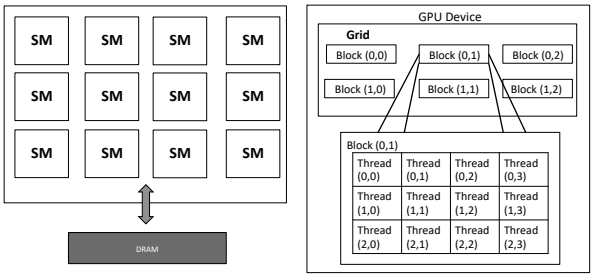
We have implemented a prototype of WolfGraph. WolfGraph employs the iterative graph processing model, where a given computation function is iteratively applied on every edge in the graph until a convergence condition is met. WolfGraph allows the developers to concentrate on programming graph processing algorithms, which will be automatically parallelized by WolfGraph.

The rest of this paper is organised as follows. Section 2.1 provides an overview of GPU hardware and CUDA programming model. Section 2.2 presents the edge-centric processing model on GPU for the graphs that can fit into GPU memory. Section 3 presents the details of the WolfGraph framework including how to split the graph into edge blocks, the allocation of edge blocks to thread blocks and the core APIs provided by WolfGraph. Section 4 presents the graph partition and CEL method for the graphs larger than GPU memory. Experimental results are presented and analyzed in Section 6. Sections 7 presents related work. Finally, Section 8 concludes this paper.

2. Background

2.1. GPGPU and CUDA

Using GPU as a general computing unit has attracted considerable attention [29] [30] [31] [32]. GPU provides massive parallel processing power. As the host for the GPU device, CPU organizes and invokes the kernel functions that execute on GPU. As shown in Figure 1a, a GPU device consists of a number of streaming multiprocessors (SM), each comprising simple processing engines, called CUDA cores in the NVIDIA terminology [33]. Each SM has its own shared memory, which is equally accessible by all CUDA cores in the SM. At any given cycle, the CUDA cores in a SM execute the same instruction on different data items. SMs communicate with each other through the global memory of GPU.



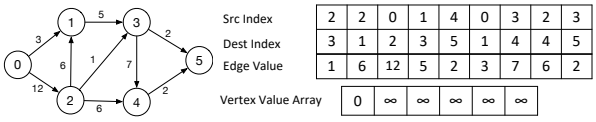
(a) The simple Cuda Hardware architecture (b) The simple Cuda programming model

Figure 1: GPGPU architecture and thread execution model in Cuda

From the programmers' perspective, the CUDA model [33] is a collection of threads running in parallel. A collection of threads (called a block) runs on a multiprocessor at a given time. Multiple blocks can be assigned to a single multiprocessor and their execution is time-shared. A single execution on a device generates a number of blocks. A collection of all blocks in a single execution is called a grid (Figure 1b). All threads of all blocks executing on a single multiprocessor divide its resources equally amongst themselves. Each thread and block is given a unique ID that can be accessed within the thread during its execution. The code running on GPU is actually executed in groups of 32 threads, what NVIDIA calls a warp. The threads within the same warp can run simultaneously on a streaming multiprocessor (SM). The programmer decides the number of blocks and threads to be executed. If the number of threads is more than the warp size, they are time-shared internally on the multiprocessor.

2.2. An Overview of Edge Centric Processing on GPU

It has been shown that because it allows sequential access to the edges, the edge-centric approach can improve I/O performance for disk-based graph processing, which requires frequent disk accessing during the execution [28]. Similarly, the sequential access to the GPU memory is critical to the performance of GPU applications [18]. This is because



(a) Graph structure (b) Edge centric representation

Figure 2: An exemplar graph and its edge centric representation

the sequential access guarantees the coalesced access to the global memory on the GPU device [18]. This section first gives an overview of edge centric graph processing on GPU and introduces the data structure used to represent the graph, and then further presents the computation model used in WolfGraph.

2.3. Edge centric Graph data structure

The input data of the edge centric processing is an unordered set of directed edges (an edge in undirected graphs can be represented by a pair of directed edges, one in each direction). A graph is represented in memory by an edge list, which consists of three one-dimensional arrays: the *vertex array*, the *source array* and the *destination array*. The *vertex array* is used to store the state of each vertex. This array is indexed by the vertex ID, that is, ith entry of the vertex array contains the state of the vertex with ID i (e.g., the distance of the path connecting to vertex i). The *source array* and the *destination array* are used to store the source and destination vertices of each edge respectively. The ith entries in the source array and the destination array form an edge in the graph. In addition, if it is needed, an array of edge weights can be added to this graph representation structure. Figure 2 illustrates the edge-centric representation of a sample graph, in which there are 6 vertices and 9 edges.

Unlike other data structure for graphs, the edge-centric graph representation does not require any pre-processing of the data because of the way the edge list is constructed in the memory. When constructing the edge list, an edge is read from the raw data in the disk each time. The edge is stored in the same order in the edge list as it is in the raw data. Therefore, the raw data is only read sequentially once to construct the edge list in the memory. Consequently, the time spent in constructing the graph data structure in memory is minimized.

2.4. Computation model

The read-compute-write iterative processing model has been used in literature to process graphs. The advantage of this model is that the graph edges can be processed in any order without affecting the correctness of the final result. Therefore, the graph processing can be parallelized.

The read-compute-write model works in the following way. The model runs a loop of iterations to update the vertex/edge values until none of the vertex/edge values in the graph changes in an iteration. Each iteration consists of three phases: read, compute and write. The read phase first gathers the source and destination vertex for each edge (stored in the source and destination arrays in the edge list), and then uses the IDs of the fetched vertices to obtain the corresponding vertex values from the vertex array. The compute phase uses the data gathered in the read phase to update the values of corresponding edges/vertices. The write phase writes the updated values back to the vertex array so that the updated values can be used in next iteration. Note that an iteration in the read-compute-write iterative processing model is different from an iteration in the vertex-centric processing model. In an iteration in the vertex-centric model, the model expands as many edges as possible from a vertex.

In each iteration of the read-compute-write model, all edges in the graph need to be processed and the edge/vertex values are updated. The computations of the edges are unordered, i.e., independent of each other. Therefore, the edge computations in each iteration can be performed in parallel. It is straightforward to evenly distribute the workload across threads in such an unordered model. Note that on the contrary, the vertex-centric model, which visits the graph by vertex and represents the graph by the adjacency list, is inherently difficult to be load-balanced among threads, because each vertex is connected to a different number of edges.

3. In-memory processing engine in WolfGraph

The in-memory engine is designed for processing the graphs which can be fitted in the global memory of GPU. When designing the in-memory engine, the key is to achieve a good degree of parallelism. Therefore, in this section, we first describe how to map the workload to the GPU threads and parallelize the computation process, and then discuss how to exploit the memory hierarchy of GPU

to improve the performance. We also present the APIs provided by WolfGraph and demonstrate how to program with these APIs at the end of the section. The in-memory processing engine will serve as the core component for processing the graphs whose sizes are bigger than the global memory of GPU, which we call out-of-memory graph processing and will be discussed in Section 4.

3.1. Parallel processing in WolfGraph

As discussed in the previous section, WolfGraph is based on the read-compute-write iterative processing model, and the edge computations in each iteration can be processed in parallel on GPU. To facilitate efficient graph processing, it is crucial to develop a suitable strategy to allocate the workload to GPU threads.

In WolfGraph, an edge is allocated to a thread and the continuously indexed threads process the edges that are stored in the contiguous memory space. This way, the coalesced memory access to the edges can be achieved in GPU. Once a thread completes the computation in an iteration (i.e., updates the value of a vertex), it writes the updated vertex values to the corresponding locations in the vertex array.

We identify two problems that need to be addressed in the write phase. First, since a single vertex array is shared by all GPU threads, multiple threads may write to the same memory location during the write phase. To address this problem. WolfGraph uses the CUDA *atomic* operation to synchronize potential simultaneous writes by threads.

Second, the vertex array is constructed in the CPU memory and copied to the GPU global memory at the beginning of the graph processing. When the threads write the newly computed data to the corresponding locations in the vertex array in each iteration, these locations may not be contiguous, which causes the random access to the vertex array in global memory. Our benchmarking experiments show that this is a factor that impairs the performance. To mitigate the performance degradation caused by the random access to global memory, WolfGraph does not write the newly updated data directly to the global memory, but write them (and synchronize, if necessary) to the shared memory first and then launch a separate kernel to write the new data to the global memory. It has two benefits by doing so. First, the number of random accesses to the global memory is significantly reduced.

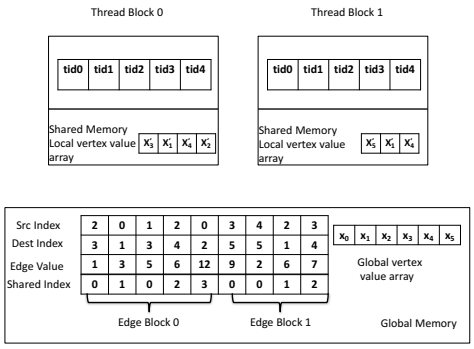


Figure 3: The Edge block representation of graph in Figure 2a

Although the writes to the shared memory is still random access, they are much faster than the random access to the global memory. Therefore, the overall performance is significantly improved compared with writing the new data directly to the global memory. This benefit is analyzed in detail later in this section and is also supported by our experiments in the later part of this paper. Second, by writing the new data first to the shared memory, some of the necessary data synchronizations are moved from the global memory to the shared memory. We have conducted the benchmarking experiments about this. We made two observations from the experimental results: 1) synchronization in shared memory is faster than synchronization in global memory *when the degree of synchronization (i.e., the number of threads that write the data simultaneously to the same location in memory) is less than a threshold*; 2) caching the new data in the shared memory can reduce both the degree of synchronization and the number of synchronizations in global memory.

3.2. In memory data structure of WolfGraph

Our research shows that the graph representation presented in Section 2.3 does not exploit the GPU memory hierarchy effectively. In this section, we present the extension to the data structure and also discuss the memory usage.

One aim of this work is to minimize the preprocessing time of a graph. We have discussed in Section 2.3 that the time spent in constructing the edge list is minimized. We also discussed that the read-compute-wirte iterative model enables us to process the edges in an unordered way, i.e., the processing of the edges can be parallelized.

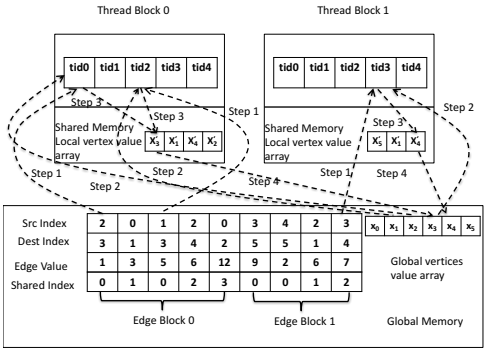


Figure 4: Global memory access inside edge blocks shown in Figure 3

Once the size of the edge block (i.e., the number of edges in an edge block) is known, the edge list can be split with minimal effort. We will present the method of determining the edge block size later in this paper.

In the hard disk, the graph is stored as a list of edges. When WolfGraph reads the graph from the hard disk, it constructs an edge list (containing three arrays: src index, dest index and edge value arrays) and a vertex-value array in the CPU memory as shown in Section 2 (Fig. 2), which will be copied from CPU memory to global memory of GPU. In a GPU, WolfGraph splits the edge list into smaller *edge blocks* (We will introduce the method of determining the size of edge blocks later in this paper), each of which consists of a set of edges. The threads that are processing the graph are organized in thread blocks. A thread block processes an edge block. Multiple thread blocks process the edge blocks in parallel. Within a thread block, an edge is processed by a thread in parallel. When a thread in a thread block processes an edge, it obtains its global thread index (assuming the thread index is i) in GPU and reads from the i-th element in the edge-value array to obtain the edge value. Then the thread obtains the index of the source node of the edge by reading the i-th element of the src index array and reads the value of the source node from the vertex-value array using the source index. After applying the user-defined graph processing algorithm (e.g., shortest path algorithm), it will generate an updated value for the destination node of the edge. In order to achieve the benefits discussed at the beginning of section 3 (i.e., reducing the number of random access to GPU global memory and also reducing the number of and the degree of data synchronization in global memory), the thread writes the updated value of the destination node first to the shared memory. In order to facilitate this, WolfGraph adds two new data structures: a *shared-index* array for the whole graph and a *local-vertex-value* array for each thread block. The local-vertex-value array is stored in local memory and used to hold the updated values of the destination nodes after processing the edges, while the shared-index array is in global memory and used to indicate to the thread which position the updated value of the corresponding destination node should be written into in the local-vertex array. As an example, the data structure and memory allocation for the graph in Fig. 2 are shown in Fig. 3. In the figure, the graph is divided into two edge blocks with 5 edges in block 0 and 4 edges in block 1, which are processed by thread block 0 and thread block 1, respectively (tidx in the thread blocks represents the thread id). The first element of the Share-index array is 0, which means that the updated value of the destination node of the edge $< 2, 3 >$ (the updated value is denoted by x'_3) should be written in the location with the index of 0 in the local vertex array (i.e., the first element of the array) in the shared memory.

3.3. Analysis of thread synchronization

As discussed above, our design requires the thread synchronisation in global memory. The thread synchronisation is implemented by the atomic operations. When designing applications on GPU, shared memory has been widely used to improve the performance due to its higher access speed than global memory. However, executing atomic operations on shared memory is rather uncommon. There has been the research indicating that thread synchronisation in shared memory is slower than that in global memory [33]. We studied this phenomenon and found that although Nvidia does not reveal the implementation details of the atomic operations in global memory and shared memory, the underlying reason for this performance discrepancy may be because the atomic operations in global memory and shared memory are implemented in different ways. In shared memory, the atomic operations are implemented using the explicit lock and unlock. When multiple threads access the same location in shared memory, they are put in a mechanism like a loop and their atomic operations are processed in sequence. When a thread invokes the atomic operation to access the data, it

locks the memory location, accesses the data and unlocks it after the operation is completed. In the global memory, however, the atomic operation is optimised and implemented with a single hardware instruction [33].

We reason that since multiple threads that are calling the atomic operations are placed in a loop and processed in sequence, the number of these threads should have impact on synchronization performance. Based on this reasoning, we conducted the following benchmarking experiments and we gain new findings.

We wrote a benchmarking program. The key kernel of the program is shown in Algorithm 1. In the kernel, each thread performs the *atomicMin* operation to write data into shared memory or global memory. The *atomicMin* operation takes two parameters as input. The second parameter is the data that the operation is writing while the first is the memory location which the data is written into. The *atomicMin* operation compares the data of the first parameter with the data in the memory location of the second parameter, and then the memory location will store the smaller value between them two. There are two arrays, *result* and *location*, in the program. An element in the *location* array holds a location that the data should be written into in the *result* array. When the arrays are defined in global memory (or shared memory), the program accesses the global memory (or shared memory). The kernel is run with a single block of 1024 threads, which is the maximum number of threads that a thread block can support in our GPU device (GTX 780TI). The synchronization degree (i.e., the number of threads that are storing the data in the same location in the *result* array) is controlled by setting the element values of the *location* array. The synchronization degree varies from 2 to 512. When the degree is 2, every two threads write to the same location of the *result* array. When the conflict degree is 512, a half of threads in the 1024 threads all write to a same location of the *result* array and the other half write to another same location. The kernel was run 10 times for both accessing shared memory and accessing global memory. The average time for running the kernel with different synchronization degrees is plotted in 5.

As can be observed from Fig. 5, the average time spent in writing the data remains approximately unchanged in global memory as the synchronization degree increases, while the time for writing

Algorithm 1: Mini Benchmark

```
1 __device__ void benchmark(int *location,
      int *values, int *result)
2 {
3   int thread_id = threadIdx.x;
4   int v = values[thread_id];
5   int l = location[thread_id];
6   atomicMin(&result[l], v);
7 }
```

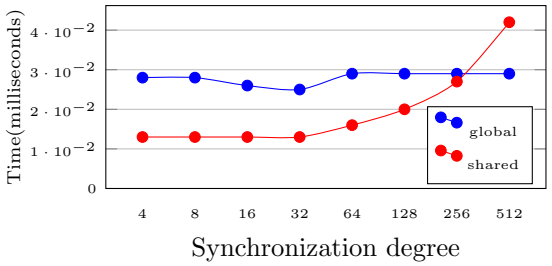


Figure 5: The run-time of the benchmark program with different synchronization degree in shared memory and global memory

data to shared memory increases as the synchronization degree increases. When the synchronization degree is higher than a certain value, writing data in shared memory takes longer than writing in global memory. This result can be explained as follows. The CUDA kernel runs in groups of 32 threads, which is called a warp. When they invoke the atomic operation for accessing shared memory, their operations are processed in sequence by the CUDA library. If some threads are accessing the same memory location, they need to wait for the lock of the memory location to be released. When more threads are accessing the same memory location (i.e., the synchronization degree is higher), the longer the threads potentially have to wait and therefore delay processing of other threads' atomic operations in the same warp. Although threads access the shared memory faster than the global memory, the longer delay caused by higher synchronization degree will eventually cancel the speed advantage of share memory.

Our GPU device, GTX 780TI, supports running maximum 2048 threads and 32 thread blocks in a SM. Since a thread block is allocated with the separate shared memory space, running a GPU kernel with a higher number of thread blocks will lead to a lower synchronization degree. If the maximum number of thread blocks is used, the number of

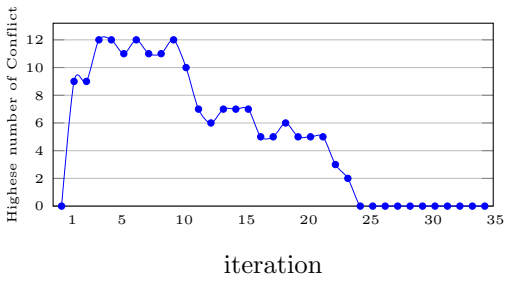
threads in each thread block is 2048/32 = 128, which means that the maximum synchronization degree in theory is 128. As shown in Figure 5, only when the synchronization degree is more than approximately 256, will the synchronization overhead in shared memory become higher than that in global memory. Therefore, we hypothesize that caching data access (and therefore synchronization) in shared memory will benefit the performance when 32 thread blocks are used to run the GPU kernel in a SM. The number 128 is the maximum synchronization degree in theory. Note that no matter how high the vertex degree is in a graph, the synchronization degree in the shared memory is limited to 128.

We also conducted the following experiments to gain the insight into the realistic synchronization degree in graph processing. In these experiments, we are running the single source shortest path (SSSP) algorithm with the following 3 real world graphs, amazon0601, webgoogle and liverJournal. The number of thread blocks are set to be 32.

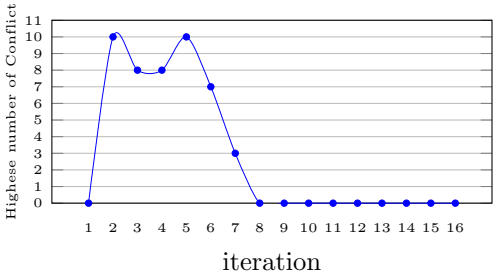
In the first experiment, we record for each graph the highest number of synchronisation among all thread blocks in each iteration, i.e., the highest number of threads that write the data simultaneously to the same shared memory location. The results are shown in Figure 6. It can be seen from the figure that the highest synchronisation degrees are only 12, 10 and 16 for the three graphs. This result suggests that when processing the three graphs, the actual synchronisation degree is much less than the threshold (256) shown in Figure 5.

After caching the data access and synchronisation in shared memory, the synchronised data access to global memory should be reduced. We conducted the experiments to show the reduction of the synchronised writes to the global memory. In the experiments, we processed the three graphs by using the shared memory and also by only using the global memory, and then recorded the number of writes in each iteration that refer to the same memory location in both cases. The results are shown in Figure 7. It can be seen from the figure that by caching the data access in shared memory, the synchronised writes to global memory, which is random writes, are significantly reduced.

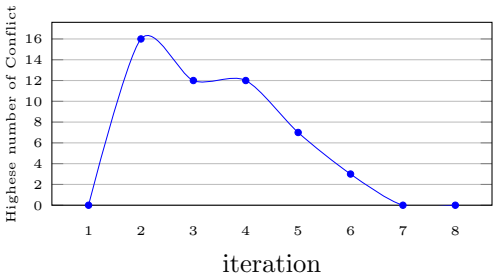
When we ran the experiments for Figure 7, we also recorded the execution time of each iteration in both cases. The results are plotted in Figure 8. As can be seen from the figure, caching the data access in the shared memory leads to much less execution time, compared with performing atomic operations entirely in global memory. These results verify our earlier hypothesis, i.e., caching the data access and synchronisation in shared memory can improve the performance.



(a) amazon0601



(b) web-Google



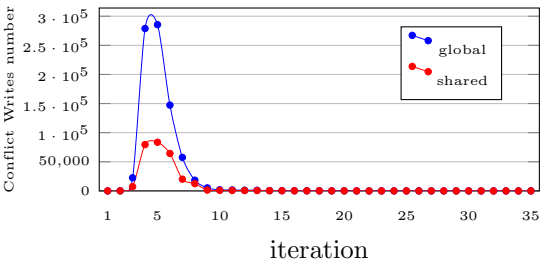
(c) LiverJournal

Figure 6: Maximum conflict among all thread blocks

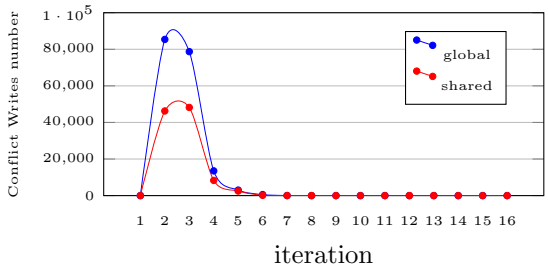
3.4. Two-level GPU processing and memory access pattern

Based on the finding above, we propose a two-level execution mode as follows to reduce random access and the synchronization overhead in global memory.

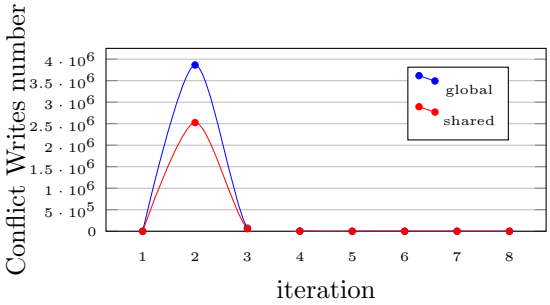
The iterative graph processing goes through a number of iterations to calculate the vertex values. In each iteration, a kernel, called the edge-processing kernel, is launched with multiple thread



(a) amazon0601

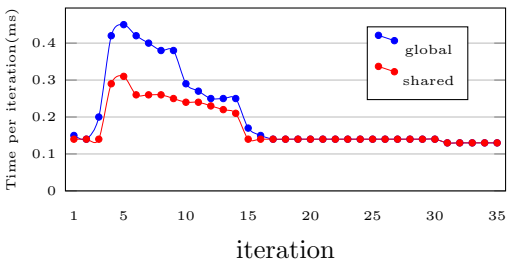


(b) webGoogle

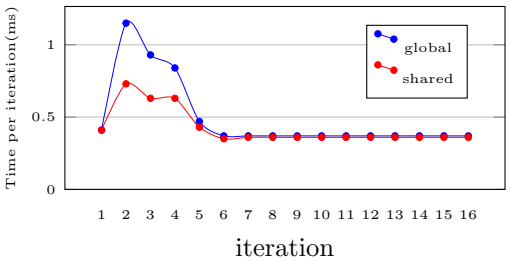


(c) liverjournal

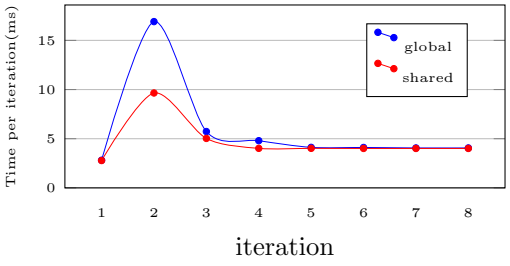
Figure 7: Conflict writes to global memory with and without using shared memory



(a) amazon0601



(b) webGoogle



(c) Liverjournal

Figure 8: Execution time per iteration. (ms)

blocks. A thread block processes an edge block. In a thread block, each thread processes one edge and calculates the value of the vertex associated with the edge in parallel. When a thread in a thread block obtains a new value for the vertex in the edge block, it accesses the shared index array and applies the atomic operation to write the new data to the local vertex-array of the edge block in shared memory. The atomic operation will perform synchronisation if more than one thread updates the data in the same location. Then the edge-processing kernel exits and another kernel, called global-updating kernel, is invoked to write the new values of the vertices in each local vertex array to the corresponding locations of the global vertex array in the global memory. In the global updating kernel, multiple thread blocks are generated, each block is used to write the data in a local vertex array to the global vertex array. In a thread block, a thread calls the atomic operation to write a data item to the global memory. Synchronisation is performed when the threads in different thread blocks update the data simultaneously to the same location in the global memory. After the updating is completed, the global updating kernel exits. Note that the synchronisation in global memory is not guaranteed before the global updating kernel exits.

With this two-level execution mode, each edge block is processed by a thread block in the GPU in the following 4 steps, which is shown in Figure 4. First, threads within each thread block read all information in the edge block in parallel. The threads with consecutive thread ID in a thread

block read the edge information residing in contiguous global memory locations, thus providing the coalesced global memory access. In the second step, based on the vertex information fetched in the first step, the threads fetch the vertex values from the global vertex array to the shared memory of the thread block. Here, the access to the Global-Vertex array is random. In Step three, the threads compute the updated value for the destination vertex of each edge and write the result back to the shared memory. Synchronisation is performed when multiple simultaneous data writing refers to the same location. The last step performs the synchronisation between different thread blocks by writing the local vertex values to the global vertex value array. As can be seen, the memory access in the steps except step 1 is not coalesced. Although the random memory access can be eliminated by more sophisticated graph partitioning methods, these partitioning methods will increase the pre-processing time significantly. For example, in CuSha, the graph is first partitioned into disjoint sets of vertices. Each partition (shard) stores all the edges whose destinations are in that set. Then, the edges in a partition are sorted based on the ID of source vertices. By partitioning the graph in this way, the consecutive threads can access the consecutive memory locations in the global memory when reading/writing the values of destination vertices into/from the shared memory. However, such partitioning method incurs much longer pre-processing time due to the activities of sorting the edges and finding the appropriate partition sizes.

3.5. Implementing GPU-based graph processing algorithms using WolfGraph

Users can implement a broad range of graph processing algorithms with WolfGraph. In this section, we take the Single Source Shortest Path (SSSP) algorithm as an example to show how to write a GPU-based graph processing program using WolfGraph.

The edge-centric approach to implementing SSSP is to iteratively update the value of the destination vertex of every edge by adding the value of an edge to the value of the source vertex of the edge. The calculation repeats until the values of the destination vertices of all edges do not change any more. Algorithm4 presents the pseudo code of the SSSP functions.

Algorithm 2 shows the part of the pseudo code that runs on the host/CPU. The host iteratively launches the GPU kernels until the results converge (i.e., *not_converge* is true). In SSSP, convergence means that the path distance from source to every vertex does not change any more. At the end of each iteration, the device copies the value of the *not_converge* variable back to the host memory (line 8) and the CPU then determines whether the graph processing is completed according to the value of *not_converge*.

Algorithm 2: Pseudo code of host function

```
1 /*host function*/
2 not_converge = true;
3 while(not_converge){
4   not_converge = false;
5   copy not_converge to GPU;
6   process_edge();
7   update_vertex();
8   copy not_converge back to CPU;
9 }
```

Algorithm 3 shows the kernel functions in WolfGraph. Each edge block is processed by a thread block. In the first kernel *process_edge*, the consecutive threads read the edge information and use this information to initialize the vertex data. (Line 5-12). The access pattern to the Global memory in these steps is shown in Figure 4. The computation is performed by invoking the *compute* method defined by the user. Since multiple threads within the block may simultaneously update the same location in the shared memory, the atomic function is used to update the destination vertex. Because the order of function invocations is non-deterministic, the compute function must be both commutative and associative.

Then a second kernel *update_vertx* is launched. In this kernel, a flag called value_updated is used to indicates whether or not the vertex values are updated. It is initially set to false (line 23). Each thread invokes the *is_update* method (lines 24), which is another user defined function to check whether or not to write the updated value back to global memory. The threads will update the contents of global memory and set values_updated to true if the not_update method returns true. If values_updated flag is set to true, the vertices in the global vertex array are updated atomically with the newly computed value and the not_converge flag is set to true. Even though the memory accesses to global vertex array are not fully coalesced in this case, it requires less number of memory transactions than directly write to global vertex array.

Algorithm 3: Pseudo code of kernel function

```
__global__ process_edge(src_index,
    dest_index,shared_index, edge_values){
  /*parallel for edge blocks:*/
  shared share_local_vertex[N];
  /* step 1 fetch edge information,
      coalesced access */
  source_vertex = src_index[tid];
  destination_vertex = dest_index[tid];
  shared_index = shared_index[tid];
  edge_value = edge_values[tid];

  /* step 2 initialise vertex value, non-
      coalesced access*/
  share_local_vertex[shared_index] = global
      [destination_vertex];
  src_value = global_vertex[source_vertex];
  __synchronize;

  /* step 3 compute the update vertex value
      */
  parallel for each thread invoke:
  compute(src_value, share_local_vertex[
      shared_index], edge_value);
}

__global__ update_vertex(global_vertex,
    shared_local_vertex);
/* step 4 write back and check if the
    computation is converged*/
/*parallel for vertex in
    shared_local_vertex:*/
value_updated = false;
if(is_updated(shared_local_vertex[tid],
    global_vertex[destination_vertex])
{
  atomicMin(&global_vertex[
      destination_vertex],
      shared_local_vertex[tid]);
  value_updated = true;
}

if(value_updated == true)
{
  not_converge = true;
}
```

Algorithm 4 presents the functions required to compute SSSP on a graph. In SSSP, every vertex holds a value (initially set to a very large number representing ∞) standing for the shortest distance from the source. Source vertex value is set to 0. At the beginning of each iteration, the *init_shared_vertex* method loads the most updated vertex values into the block's shared memory. The *compute* function is act on every edge, it first computes the new distance for a destination vertex, then atomically choosing the minimum distance between the current and the calculated distances. The *is_update* function notifies the caller to execute the next iteration if the new distance of the destination vertex is smaller than its old value. As we can see, the user only need to provide the *compute* and *is_update* functions; hence making it easier to code graph processing algorithms using WolfGraph.

Algorithm 4: Pseudo code of SSSP implementation in WolfGrahp

```
__device__ void compute(src_value,
    share_local_vertex,  edge_value)
{
  if(share_local_vertex != INF)
  {
    atomicMin(&(share_local_vertex,
        src_value+edge_value));
  }
}

__device__ bool is_updated(
    shared_local_vertex,  global_vertex)
{
  if(shared_local_vertex < global_vertex)
  {
    return true;
  }
  return false;
}
```

4. Out-of-memory graph processing in WolfGraph

In the last section, we presented the design and implementation of in-memory processing of WolfGraph, i.e., the case where the entire graph can fit into the global memory of GPU. However, the size of GPU global memory is much smaller than the host memory. The sizes of real world graphs can vary from few gigabytes to terabytes, which are too large to be loaded into GPU global memory all at once. Therefore, in this section, we design an out-of-memory graph processing framework that can process such large-scale graphs.

4.1. The graph partition method

The general idea of designing an out-of-memory graph processing framework is to partition the graph into sub-graphs that can fit into the GPU memory and process these sub-graphs in GPU one at a time. There are two key issues that need to be addressed properly. First, we still want to minimize the pre-processing time as we do for in-memory graph processing. So the graph partition should

not incur many pre-processing efforts. Second, a common problem of processing sub-graphs one at a time is that when a sub-graph is being processed or after it has been processed, the graph processing algorithm needs to access the previously processed sub-graph to update the newly calculated results. In the GPU environment, however, this requires the data exchanges between GPU's global memory and CPU main memory, which will incur high overhead and should be reduced as much as possible. Next, we present the methods that we develop to address the above two issues.

As the graph data is read into the CPU memory, the data structure such as the one in Figure 3 is constructed in the CPU memory, including the edge list and the global-vertex-value array. To address the first issue, i.e., to minimise the time spent in graph partitioning, the out-of-memory graph processing framework splits the edge list into chunks (each chunk is a subgraph) in the similar way as we split the graph into edge blocks in the in-memory graph processing. Namely, a graph is partitioned into the equal-sized subgraphs in each of which the edges are in the same order as they are in the graph raw data. Therefore, there is no need to pre-process the graph. In the out-of-memory graph processing, we do not partition the global-vertex-value array. The reasons are two folds. First, to obtain the vertices for a subgraph, we have to search the global-vertex-value array to construct the vertex sub-array that contains the vertices in a subgraph, which incurs the pre-processing. Second, we argue there is no practical need to partition the global-vertex-value array. Since compared to the memory space occupied by edges, the memory space required by vertices is small. For example, in the graph arabic2005 [34], the edges take 10.24 GBytes memory space while the vertices only occupy 90.98 MBytes space (each vertex value is stored as an integer). The vertices of an entire graph can be easily accommodated in GPU global memory, even for very large-scale graphs.

The size of each subgraph is determined in the following way. Assume the size of GPU global memory size is G, and the entire graph has $|N|$ vertices and $|E|$ edges. Because we put the whole vertex array into the GPU global memory, the remaining memory space for holding the edge list of a subgraph is then $G - |N| * sizeof(vertex)$ (a vertex is represented as an integer or a floating point number depending on the graph processing algorithms). As shown in Figure 3, an edge is represented by

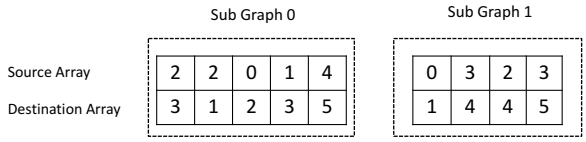


Figure 9: An example of partitioning graph in Figure 2a into 2 sub-graphs

4 elements: source index, destination index, edge value and shared index value. Therefore the size of an edge is $3 * sizeof(int) + sizeof(edge\_value)$ (an edge value is an integer or a floating point number depending on graph processing algorithms). The number of edges that GPU global memory can hold, denoted by e, can then be calculated by

$$e = \lfloor \frac{G - |N| * sizeof(vertex)}{3 * sizeof(int) + sizeof(edge\_value)} \rfloor \quad (1)$$

This graph partition method does not require any pre-processing. It simply puts the first e edges in the edge list in the first subgraph, next e edges in the second subgraph, etc, as Wolfgraph reads the graph raw data from the hard disk into the CPU memory. This process requires only one sequential read of the graph from the hard disk. As an example, the graph in Fig. 2a can be partitioned into subgraphs shown in Fig. 9.

4.2. Out-of-memory graph processing engine

After partitioning the graph into subgraphs, the out-of-memory graph processing engine processes the subgraphs one by one. The processing engine starts with loading the first subgraph into the GPU global memory, processes the subgraph using the in-memory processing engine presented in previous sections, and copies the results back to the CPU memory when the computation for the subgraph is completed. Then the processing engine moves to the next subgraph and repeats the process. In the out-of-memory processing, an iteration is defined as a round of processing all subgraphs with the in-memory processing engine. When an iteration is completed and there are still vertices whose values are updated in this iteration (i.e., the graph processing has not converged yet), the processing engine enters the next iteration and repeats the above process from the first subgraph again. This loop exits when the vertices' values in all subgraphs remain unchanged in an iteration.

Although the graph partition method in previous subsection does not require pre-processing, the partition may cause excessive data exchanges between GPU and CPU during the processing of the graph, which hampers the performance. The following example is used to illustrate this issue.

We still use the graph in Fig. 2a as the exemplar graph. Assume the SSSP (Single Source Shortest Path) algorithm is used to process this graph, and the graph is partitioned into two subgraphs as shown in Fig. 9. In this example, the SSSP algorithm requires a loop of 3 iterations as follows to finish the processing of the graph.

In the first iteration, subgraph 0 is processed first. According to the edge-centric processing, the in-memory processing engine goes through another loop to process the edges in a subgraph in parallel by taking as input the weights of the edges and the current values of their source vertices. The weights of the edges and the initial values of their source vertices are shown in Figure 2b. In the first iteration of processing subgraph 0, the thread that processes the edge $< 2, 3 >$ updates the value of vertex 3 to be 1 ($\infty + 1 = \infty$), since the weight of the edge is 1 and the current value of its source vertex (i.e., vertex 2) is ∞. Similarly, after processing the edges $< 0, 2 >$, $< 1, 3 >$ and $< 4, 5 >$ in the first iteration, the new values of vertices 1, 2, 3 and 5 (i.e., the destination vertices of these edges) are shown in the row of "iteration 1" in the sub-table of processing subgraph 0 in Table 1. Since the values of the vertices are still not stable, the processing of subgraph 0 moves to next iteration. In the second iteration, the vertices will be updated with the values shown in the row of "iteration 2". There are two issues to note in the second iteration. First, since the edges $< 2, 1 >$ and $< 1, 3 >$ are processed in parallel, the processing of the edge $< 1, 3 >$ takes as input the current value of vertex 1 (∞), not the updated value after processing the edge $< 2, 1 >$ (i.e., 18). Second, a thread that processes an edge uses the atomic operation to write the minimal value to the vertex value array when multiple threads are writing the values to the same array location simultaneously. The edges $< 2, 3 >$ and $< 1, 3 >$ have a common destination vertex (i.e., 3). Therefore, when these two threads write the new values into the position of vertex 3 in the vertex value array, the minimal value, which is 13 (not ∞), is written. Since the values are updated in the second iteration, the processing goes into the third iteration. The values of all vertices in subgraph 0 remain unchanged in the third iteration. So the in-memory processing for subgraph 0 is completed.

Table 1: The Edge-centric Processing for the graph in Figure 9

Processing subgraph 0 in Figure 9						
vertex	0	1	2	3	4	5
initial values	0	∞	∞	∞	∞	∞
iteration 1	0	∞	12	∞	∞	∞
iteration 2	0	18	12	13	∞	∞
Processing of subgraph 1 in Figure 9						
vertex	0	1	2	3	4	5
initial values	0	18	12	13	∞	∞
iteration 1	0	3	12	13	18	15

The out-of-memory processing engine then copies subgraph 1 to the GPU and invokes the in-memory processing engine to process subgraph 1. When processing subgraph 1, vertex 1 is updated with a new value 3. Other vertices are updated with the values shown in the row of "iteration 1" in the sub-table of processing subgraph 1 in table 1. After iteration 1, the values of all destination vertices remains unchanged and therefore, the processing of subgraph 1 is completed.

After the first iteration of the out-of-memory processing is finished, the out-of-memory processing moves to the second iteration. In the second iteration, when processing subgraph 0, the engine updates vertex 3 with the value 8. When processing subgraph 1, the engine updates verices 4 and 5 with the values 15 and 10, respectively. In the third iteration of the out-of-memory processing, the values of all vertices remain unchanged in both subgraphs. Therefore, the out-of-memory processing of the graph exits, returning the shortest distance from vertex 0 to all other vertices.

In this example, the out-of-memory processing goes through three iterations to complete the computation and consequently, the graph is copied to the GPU three times. The CEL (Concatenate Edge List) method is presented in next section to reduce the number of times the graph is copied to GPU in the out-of-memory processing.

4.3. The Concatenated Edge List representation

In iterative graph processing, the fundamental reason behind the repetitive loading of a subgraph is because the graph partition method essentially partition the graph randomly and it may place an edge and its child edges (Edge A is called the child

of edge B if B's destination vertex is A's source vertex) in different subgraphs. If the edge's child edges are processed before the edge, then the subgraph that contains the child edges need to be loaded and re-processed again after the parent edge is processed.

After identifying the reason behind the repetitive loading of subgraphs, a method called Concatenated Edge List (CEL) is designed in WolfGraph to reduce the number of times a graph is copied from CPU to GPU.

Inspired by Parallel Sliding Window method developed in GraphChi [13] and Concatenated Windows developed in CuSha [18], we developed a method called Concatenate Edge List (CEL) to overcome the above problem. The first step in the CEL method is similar as in Subsection 4.1. Namely, we determine the number of subgraphs that the graph has to be partitioned into, so that each subgraph can be fitted into the GPU global memory. Then we evenly split the edges into each subgraph. To reduce the data exchanges between CPU and GPU, WolfGraph transfers the concatenated edge list into GPU instead of transferring the subgraph into GPU.

The CEL is built in the following way. When WolfGraph processes a subgraph, it identifies such edges in the this current subgraph that have successor edges in the subgraphs that have been processed in this iteration of out-of-memory processing. WolfGraph selects all successor edges of the current subgraph and append them to the current subgraph, which is the CEL for the current subgraph.

Note that WolfGraph only selects the successor edges of the current subgraph from the subgraphs that have been processed in this iteration of out-of-memory processing, not from the subgraphs that have not been processed yet in this iteration. This is because if a sub-graph that contains the successor edges of the current subgraph has not been processed yet, the subgraph, including the successor edges of the current subgraph in this subgraph, will be processed later after the current subgraph in this iteration. Adding these successor edges into the CEL will not help reduce the number of iterations in the out-of-memory processing.

Due to the fact that there may be the vertices with high degrees in the graph, the constructed CEL may be too big to fit into the GPU memory. In the event this happens, WolfGraph splits the constructed CEL into smaller CELs until each split CEL can fit into the GPU memory. Then WolfGraph transfers the CEL to GPU and processes it using the in-memory processing engine.

An example is shown in Figure 10 to illustrate the construction of the CEL. In this example, we partition the graph in Figure 2a into two subgraphs. The out-of-memory processing engine processes subgraph 0 first. Since all other subgraphs (i.e., subgraph 1 in this example) have not been processed yet, there do not exist the successor edges of subgraph 0 in the subgraphs that have been processed in this iteration of out-of-memory processing. Therefore, the CEL of subgraph 0 is the same as subgraph 0. When processing subgraph 1, WolfGraph first identifies all successor edges of subgraph 1 in the subgraphs that have been processed in this iteration, which is subgraph 0 in this example. The successor edges of subgraph 1 in subgraph 0 are $< 1, 3 >$ and $< 4, 5 >$. Therefore, WolfGraph appends these two edges to subgraph 1, which is the CEL for subgraph 1. Then WolfGraph transfers the CEL to the GPU memory and processes the CEL using the in-memory processing engine.

By using the CEL method, there is no need to load the subgraphs again that have been processed in this iteration of out-of-memory processing and contain the successor edges of the subgraph that is being processed. Hence, the number of iterations and consequently the overall processing time can be reduced.

To enable the fast identification of the successor edges from other subgraphs, we assign a global index to each edge and design a mapping array to record the relationship between a vertex and the edges that start with this vertex. An example of the mapping array is shown in Figure 10. In this figure, a global index of an edge is at the left side of the edge. For example, the edges $< 2, 3 >$ and $< 2, 1 >$ have the indices of 0 and 1, respectively. The mapping array records the mapping between a vertex and the edges that start from the vertex. With the mapping array, we can quickly identify whether a destination vertex (i.e., the vertex is the destination of an edge) in the current subgraph have the child edges. If there are, the global indices of the edges can be used to quickly locate which subgraphs that these child edges are in since WolfGraph partitions the edges into subgraphs in their storing order and WolfGraph knows how many edges there are in each subgraph. For example, vertices 1 and 4 are two destination vertices in subgraph 1. It can be seen from the mapping array that there is one edge, which has the index of 3, starting from vertex

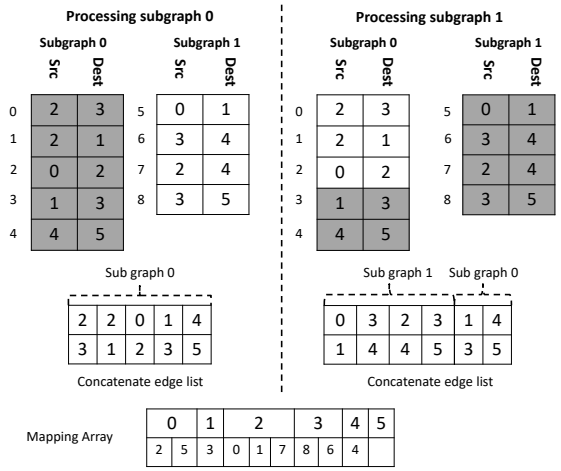


Figure 10: An example of building the CEL (concatenated edge list) from two sub-graphs. The gray area represent the edges included in the CEL when processing the current subgraph.

1 and that there is one edge, which has the index of 4, starting from vertex 4. WolfGraph knows that the edges with the indices 3 and 4 are in subgraph 0 and subgraph 0 has been processed in this iteration. Therefore, these two edges are retrieved and appended to subgraph 1 to construct the CEL for subgraph 1.

The following example is used to illustrate the effectiveness of the CEL method. We still apply the SSSP algorithm to process the graph in Figure 9. In the first iteration of out-of-memory processing, the processing of subgraph 0 is the exactly same as the previous example shown in Table 1. When processing subgraph 1, the out-of-memory processing engine first constructs the CEL for subgraph 1, the constructed CEL is shown in Figure 10. When processing this CEL, the vertices $1, 3, 4, 5$ are updated with the values of 3, 8, 15, and 10 respectively. In the second iteration of out-of-memory processing, the new values computed for these vertices are greater than or equal to the current values stored for these vertices. Therefore, the engine will not update any vertices. Namely, the values of all vertices remain unchanged in this out-of-memory processing. Therefore, the processing of the entire graph exits after the second iteration. Compared with the example presented in subsection 4.2, the CEL method reduces the number of out-of-memory processing iterations from 3 to 2, and therefore, reduces the overall processing time.

5. Implementation of WolfGraph

WolfGraph consists of three main components: Loading Engine, Data Transfer Engine, and Compute Engine. Figure 11 shows the general software architecture of WolfGraph. In this section, we describe selected details of these components.

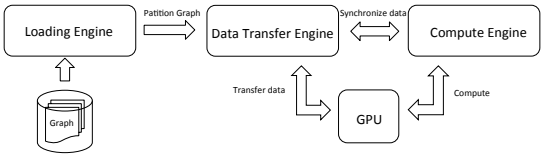


Figure 11: Architecture of WolfPath framework.

5.1. Loading Engine

The Loading Engine is responsible for (1) load-balanced edge block creation and (2) providing graph partitioning logics.

Designing an efficient format for storing the Edge Block is paramount for good performance. In WolfGraph, we store the Edge Block in the following way.

We first create an array called Edge Block List, which is a flat array of pointers, each pointer points to an Edge Block. The Edge Block consists of four arrays as described in Section 3.2. The size of Edge Block List array and each Edge Block are determined by the number of edges in the graph and number of thread blocks used in graph processing.

WolfGraph uses user defined thread block size T to determine the size of Edge Block List array and each Edge Block. Assume the input graph has E edges, the size of Edge Block List B is $\lceil \frac{E}{T} \rceil$, and the size of each Edge Block are $\lceil \frac{E}{B} \rceil$. By computing the size of each array, we can statically allocate memory for them, which can reduce the significant amount of time that used in resizing and reallocating in dynamic memory allocation. Then WolfGraph reads the graph data from the hard disk sequentially, and stores the edge information in each Edge Block in the same order. Once it fills all edges in one Edge Block, it moves to the next Edge Block. Therefore, we only need to iterate the entire graph once to construct the Edge Blocks.

5.2. Data Transfer Engine

The data transfer engine aims to transfer data between GPU/host memory and construct the Concatenate Edge List if necessary.

In data transfer engine, the data is transferred through the memory-copy operation provided by CUDA. We use CUDA stream operation to overlap the data transfer and the computation. For different Edge Block, each Stream created by the *StreamCreator* inside the Data Transfer Engine typically issues multiple $MemcpyAsync()$ operations and graph computation kernels asynchronously. Therefore, the data transfer and the computation is overlapped.

When building the Concatenate Edge List, each Edge Block needs to access edges stored in other sub-graphs. Hence, to enable fast access to edges from other Edge Block, we give a global index to each edge and use a mapping array to build the relationship between each destination vertex and the edges that start with this vertex. The mapping array is constructed while loading the graph from disk to memory. We first construct an array indexed by the vertex ID, each array element points to a link list. Because we read edges from disk sequentially and write them in Edge Blocks in exact same order, we use a counter to keep track its global index, that is, every time we add an edge to the Edge Block, we increment the counter by 1. For each edge added to the Edge Block, we use its source vertex ID to locate its position in mapping array and append its global index value to the corresponding link list. Therefore, the global index value stored in each link list is in ascending order.

5.3. Computation Engine

The Computation Engine is mainly responsible for GPU in-memory computation and to send feedback information to the Data Transfer Engine about the results and termination condition used for the next iteration. The detail of Computation Engine is discussed in Section 3.4.

6. Evaluation

In this section, we evaluate the performance of WolfGraph using two types of graph dataset: "small" graphs that can fit into the GPU memory (called in-memory graphs in the experiments) and large graphs that can fit the CPU memory, but cannot fit in the GPU memory (called out-of-memory graphs).

In-memory graphs are used to evaluate WolfGraph against other state-of-the-art GPU-based in-memory graph processing systems, including CuSha [18], Virtual Warp-centric [20] and Gunrock [17]. Out-of-memory graphs are used to compare WolfGraph with two popular CPU-based graph processing systems, GraphChi and X-Stream.

Table 2: Real world graphs used in the experiments

GPU In memory Graph		
Graph	Vertices	Edges
RoadNet-CA [35]	1965206	5533214
amazon0601 [35]	403394	3387388
Web-Google [35]	875713	5105039
LiveJournal [35]	4847571	68993773

GPU Out-of-memory Graph		
Graph	Vertices	Edges
orkut [35]	3072441	117185083
hollywood2011 [36]	2180653	228985632
arabic2005 [37]	22743892	639999458
uk2002 [38]	18520486	298113762

We used eight graphs, which are publicly available, in the experiments. The eight graphs are listed in Table 2. These graphs are abstracted from different real-world applications and natures and cover a broad range of sizes. For example, the *Live-Journal* is the directed social networks which represent friendship among the users. *RoadNetCA* is the California road network in which the roads are represented by edges and the vertices represent the intersections. *WebGoogle* is a graph released by Google in which vertices represent web pages and the directed edges are links. *orkut* is an undirected social network, in which vertices and edges represent the friendship between users. *uk-2002* is a large crawl of the .uk domains, in which vertices are the pages and edges are links. Note that some of the graphs we used in the experiments demonstrate the power-law feature, i.e., a small number of vertices in a graph have high degrees. For example, the Hollywood-2011 graph has the average degree of 105, but the maximum degree is 13107. Also, the average degree of Arabic-2005 is 28, but the maximum degree of this graph is 575618.

We choose three widely used graph processing algorithms to evaluate the performance, including Breadth First Search (BFS), Single Source Shortest Paths (SSSP) and PageRank(PG). The PageRank algorithm were set to run 10 iterations, a typical setting used in other literature [28].

A Nvidia GeForce GTX 780Ti graphic card is used in the experiments, which has 12 SMX multiprocessors and 3GB GDDR5 RAM. The host ma-

chine is the Intel Core i5-3570 CPU operating at 3.4 GHZ with 32 GB DDR3 RAM, on which the operating system Fedora 21 is installed. The algorithms are programmed using CUDA 6.5. All programs were compiled with the highest optimisation level (-O3).

6.1. Performance evaluation

6.1.1. Comparison with the GPU-based Out-of-memory Frameworks

WolfGraph (WG) is a GPU-based graph processing framework. In this section, we first compare WolfGraph (WG) with two popular CPU-based graph processing frameworks, GraphChi (GC) [13] and X-Stream (XS)[28], to evaluate the speedup of our GPU-based graph processing solution over the CPU-based solutions. WolfGraph assumes that the graph can be fitted into the CPU memory. The graphs chosen in the experiments can fit in the CPU memory, but cannot fit in the GPU memory. In doing so, GraphChi and X-Stream do not have to endure unfair hard disk operations compared with WolfGraph.

Figures 12 and 13 show the speedup of WolfGraph over GraphChi and X-Stream, respectively. It can be seen from the figures that WolfGraph achieves an average speedup of 7.48 and 8.55 over GraphChi and X-Stream (running with 4 threads), respectively, although WolfGraph has to move the data between GPU and CPU and build the Concatenate Edge List. To analyse the performance in detail, Figure 14 shows the detailed time breakdown of the three frameworks. We define the pre-processing time as the time spent in loading the graph from the hard disk to the CPU memory and constructing the designed data structures. Computation time refers to the time taken in actual execution of the algorithm. The time of building CEL records the time spent in building the concatenated edge list. Data Transfer is time taken by WolfGraph in transferring the data between host and GPU. Since X-Stream does not have a separate pre-processing stage, which overlaps the graph loading and graph processing. So we only record the total execution time for X-Stream.

As can be observed from the time breakdown shown in the Figure 14, the performance improvement of WolfGraph over GraphChi and X-Stream is attributed by the following factors. First, the computation time of WG is shorter than that of GC by orders of magnitude. This is because

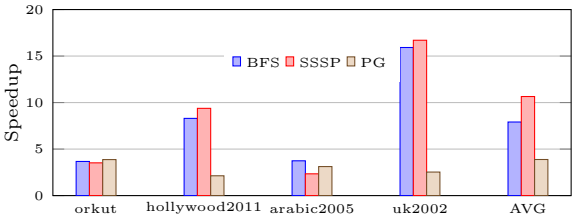


Figure 12: Speedup over GraphChi

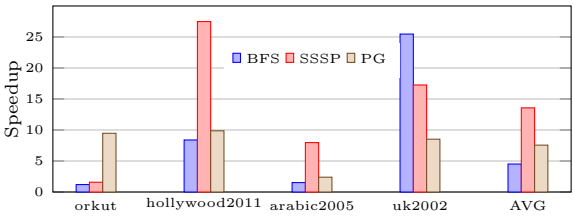


Figure 13: Speedup over X-Stream

the edge-centric processing model used by WolfGraph can fully utilize the massive parallel processing power provided by GPU, while the GraphChi and X-Stream are CPU-based, which limits the degree of parallelism. Second, the pre-processing time in WolfGraph is less than half of the pre-processing time in GraphChi. This is because the pre-processing time of WolfGraph is almost equal to the graph loading time and no other pre-processing operations are needed while GC needs to convert the raw graph into the graph stored in the "shard" structure and sort the edges in each shard.

The experimental results with Hollywood-2011 and Arabic-2005 show that the impact of the power-law feature on the CEL method. When the vertex degree increases, it takes more time to construct the CEL. However, since we establish the mapping between the edge No. and the vertex, the time spent in constructing the CEL only increases linearly as the degree increases. For example, the Hollywood-2011 graph has the average degree 105 with the maximum degree being 13107. When processing this graph with WolfGraph, our experimental records show that the time spent in building the CEL is 0.89 second (the data transfer time is 0.43 second and the computation time is 0.18 second). The average degree of the graph Arabic-2005 is 28 with the maximum degree being 575618. It takes WolfGraph 2.63 seconds to build the CEL (1.98 second on data transfer and 0.49 second on computation).

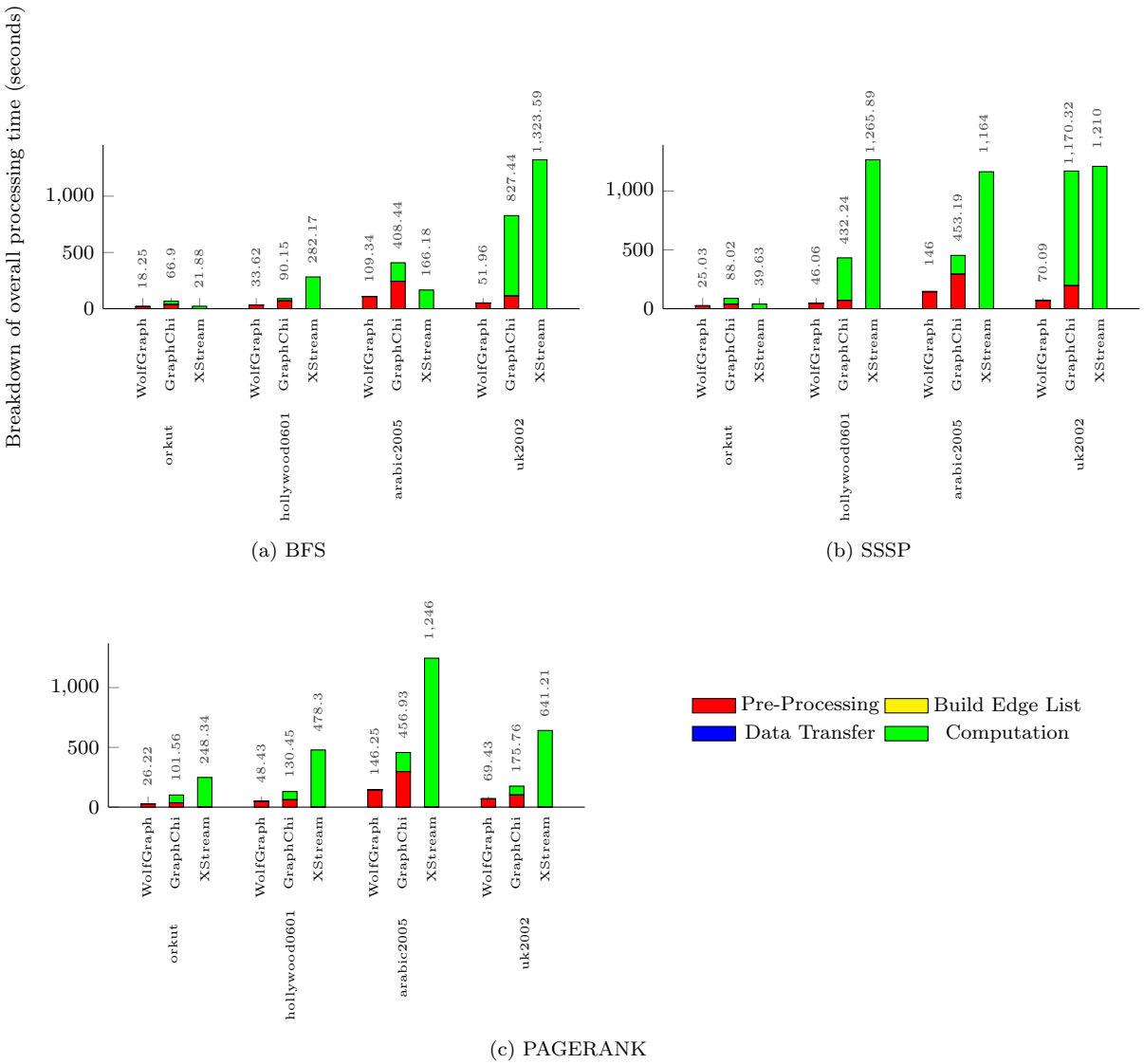


Figure 14: The execution-time breakdown of WolfGraph, GraphChi and X-Stream on out-of-GPU-memory graphs. The time is in second.

6.1.2. Comparison with the GPU-based in-memory Frameworks

The results shown in last section demonstrated the ability of WolfGraph in processing the graphs that are bigger than the size of the GPU memory. Recall that the other goal of WolfGraph is that it should perform as good as other existing in-memory graph processing frameworks. In this section, we examine the performance of WolfGraph for processing smaller graphs. We compare WolfGraph with the state-of-the-art in-memory processing solutions including CuSha [18], Virtual Warp Centric [20] and Gunrock [17]. Note that we compared with CuSha-CW (a version of CuSha) in the experiments, since it is demonstrated that the version of CuSha-CW represents the best performance for the CuSha framework.

The performances is broken down in Figure 15. As can be seen from the figure, WolfGraph outperforms other three solutions. Compared with VWC, WolfGraph not only has the shorter pre-processing time, but also achieves the 6.5x speedup on average in the computation time, thanks to the coalesced memory access to the global memory, and the

maximum usage of the GPU parallelism through the edge-centric processing model. Compared with CuSha, the computation time between WolfGraph and CuSha are very similar. But the graph representation used by CuSha incurs long pre-processing time. Hence, the overall processing time of WolfGraph is much shorter than CuSha. As for the comparison with Gunrock, the computation time achieved by Gunrock is shorter than that of WolfGraph by 10-70% with the BFS and the SSSP algorithms, but is longer by 5X with PageRank. This is because Gunrock uses a load-balancing strategy during the traversal of the graph and only computes the vertices that changed their values in the previous iteration. However, Gunrock suffers from the longest pre-processing time among all four solutions, which makes it the slowest solution after the pre-processing time is counted. Averagely, the overall execution time of WolfGraph is 65% faster than the Gunrock.

The experiments with both out-of-memory and in-memory processing show an interesting phenomenon: the computation time is much shorter than the pre-processing time. This indicates the need for reducing the pre-processing time, which is typically spent in building the user-defined data structure.

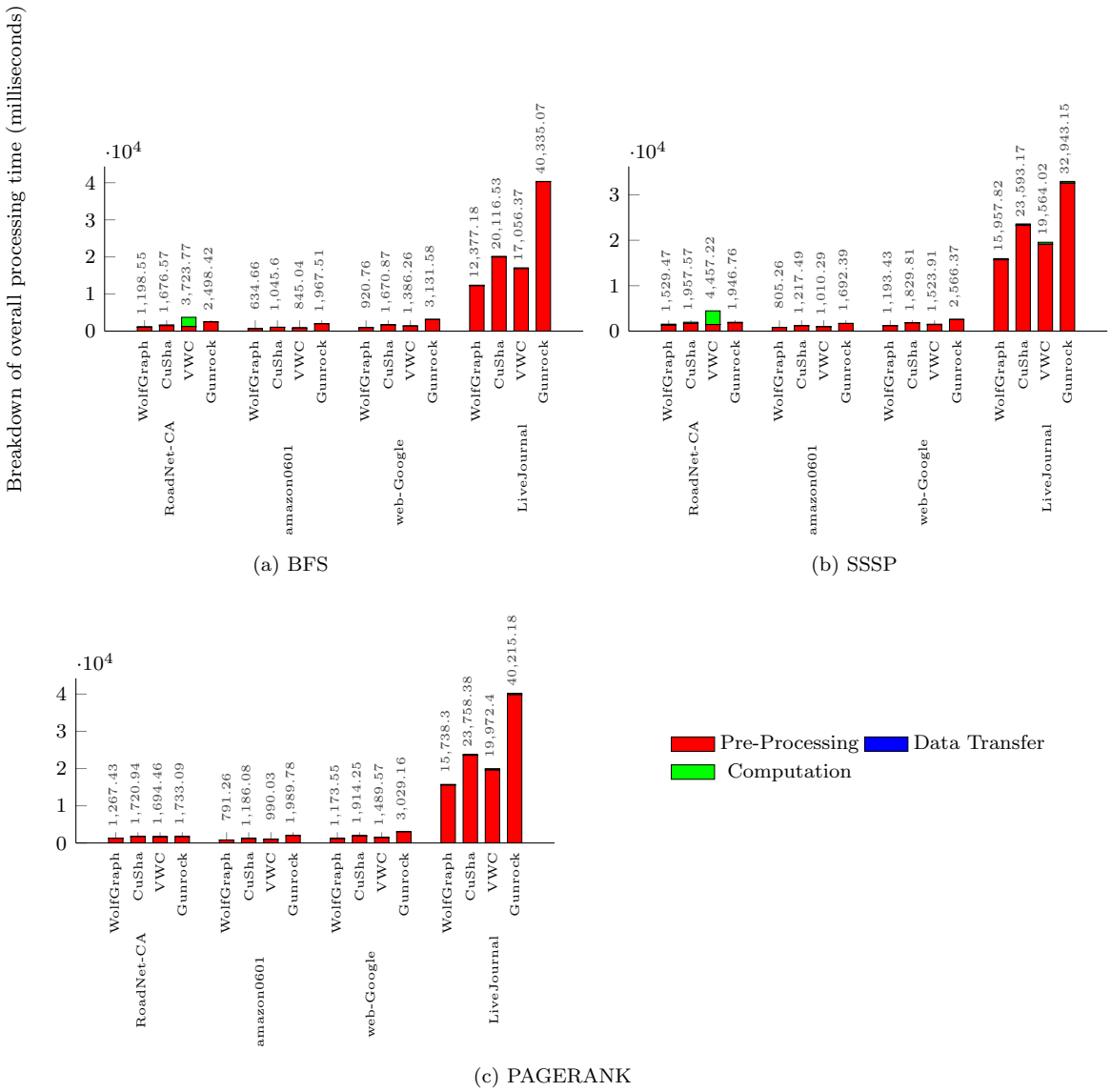
6.2. Global Memory efficiency

The reason why we adopt the edge-centric processing and represent the graphs as the edge list is because we aim to achieve the sequential access to the global memory. This section evaluates the efficiency of WolfGraph in terms of accessing the global memory. We compare WolfGraph with CuSha and Virtual Warp Centric in terms of the average global memory load efficiency, the average global memory store efficiency and the warp execution efficiency. In the experiments, we process the *LiveJournal* graph using the BFS, SSSP and PageRank algorithms. The results are shown in Figure 16.

The global memory load efficiency is the ratio of the achieved load throughput to the required load throughput in global memory, which indicates how well the threads within a kernel read from the global memory (i.e., a higher value indicates that more read operations are performed). As can be seen from Figure 16a, the global memory load efficiencies achieved by VWC and Gunrock are only 41.4% and 61.3%, respectively, on average. This is because VWC and Gunrock store the graph data with the CSR format, which is difficult to achieve the coalesced access. On the contrary, the average global memory load efficiencies achieved by CuSha and WolfGraph are 89.6% and 93.6%, respectively. This is because both frameworks provide the coalesced access to the global memory.

The global memory store efficiency indicates the ratio of the global memory write throughput achieved by the kernel to the global memory store throughput actually needed by the kernel. This value shows how well the threads within a kernel write to the global memory. As we can see from Figure 16b, the average store efficiency achieved by VWC is 12.8%. The store efficiency achieved by CuSha and WolfGraph are 42.8% and 42.5% respectively. In both WolfGraph and CuSha, the store operation are performed in parallel. However, in VWC, only one thread within each virtual warp is used to update the vertex value, which results in a lower store efficiency. For all three frameworks, the average global memory store efficiency is lower than their load efficiency. This is mainly because the store operation is not fully coalesced. On the other hand, Gunrock optimises the store efficiency by performing the coalesced writes to the global memory. Therefore, it achieves the store efficiency of nearly 80%. However, the coalesced writes achieved by Gunrock is at the expense of a large amount of preprocessing.

The warp execution efficiency is defined as the ratio of the average number of active threads in a warp to the maximum possible number of active threads per warp supported by the multiprocessor in the GPU. The efficiency indicates how well the hardware resources in GPU are utilized. As shown in Figure 16c, The warp efficiencies achieved by CuSha and WolfGraph are 85.5% and 96.7% on average. VWC and Gunrock delivers a much lower warp execution efficiency of 36.9%. This is mainly due to the imbalanced workload distribution among the thread blocks. On the other hand, both CuSha and WolfGraph evenly distribute the workload to thread blocks, and consequently improve the warp execution efficiency. Moreover, in CuSha, the warp execution efficiency is bounded by the window size (the set of edges in shard j that are involved in the processing of shard i [18]). WolfGraph does not have such a limitation because each thread block is only responsible for the edge block assigned to it.



(a) BFS

(b) SSSP

(c) PAGERANK

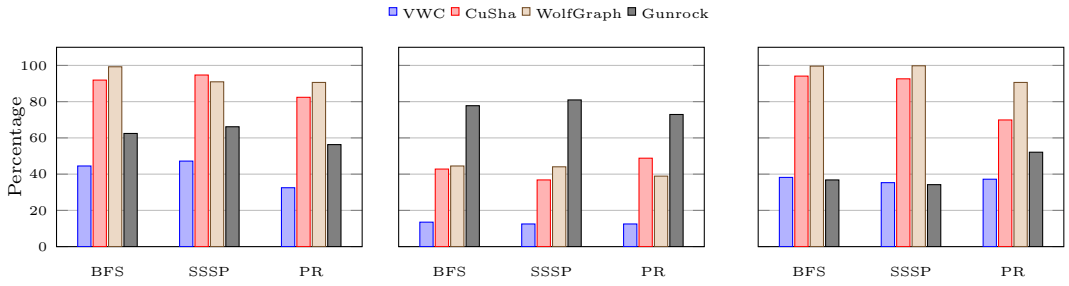
Figure 15: The execution-time breakdown of WolfGraph, CuSha, VWC and Gunrock on in-GPU-memory graphs. The time is in millisecond.

6.3. Memory occupied by different graph representations

In this subsection, we evaluate the memory consumption of different graph representations, i.e., the edge list representation used by WolfGraph, CSR by VWC and Gunrock, and the CW representation by CuSha.

The memory consumed by the edge List representation is $3 * |E| * sizeof(index) + (2 * |E| + |V|) * sizeof(Value)$, which is the same as that of the CW representation. However, the CSR representation only consumes the memory of $(|E| + |V|) * sizeof(index) + (|E| + |V|) * sizeof(value)$. Figure 17 shows the actual memory consumed by WolfGraph (the edge List), Cusha-CW and VWC and Gunrock (CSR). WolfGraph and CuSha-CW consume 2.45x and 2.46x more space on average than CSR. CuSha-CW uses slightly more memory than WolfGraph. This is because the data structure overhead of a vector is slightly more than that of an ordinary array.

The increased memory consumption leads to the



(a) Load Efficiency (b) Store Efficiency (c) Warp Efficiency

Figure 16: Comparing WolfGraph with CuSha and VWC in terms of accessing the global memory; The liveJournal graph is processed using BFS, SSSP and PageRank (PR) algorithms

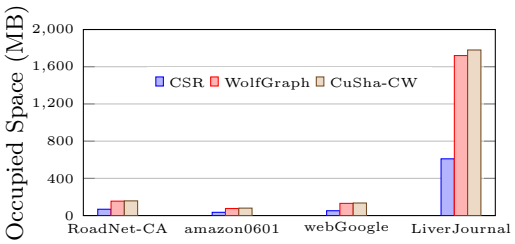


Figure 17: Memory occupied by the graphs using CSR, CuSha-CW, WolfGraph edge list representations

Table 3: Comparing the out-of-memory processing by WolfGraph with the in-memory processing by VWC due to the difference in graph representations.

Graph		BFS	SSSP
orkut	WG	18.25	25.03
	VWC	31.38	35.26
hollywood	WG	33.62	46.06
	VWC	45.98	59.58
uk-2002	WG	51.96	70.1
	VWC	58.42	74.58

following situation, it is possible that some graphs can fit into GPU memory with the CSR representation, but cannot with the edge list representation of WolfGraph and the CW representation of Cusha. To test the performance of WolfGraph in this scenario, we select the following three graphs, orkut, hollywood2001 and uk-2002, to conduct the experiments. These graphs can fit into GPU with the CSR representation, but have to be split into two or more Concatenate Edge Lists (CEL) in WolfGraph. The benchmarking algorithms used in the experiments are BFS and SSSP. The results are listed in Table 3.

As can be seen from the table, although building the concatenated edge list and transferring the data to GPU add the extra overhead, WolfGraph still outperforms VWC for both benchmarking algorithms. This is because WolfGraph has much shorter pre-processing time and also faster computation due to its GPU-friendly graph representation.

6.4. Sensitivity Analysis of WolfGraph

In this section, we examine the sensitivity of WolfGraph across different graph characteristics, including graph size and graph sparsity. The synthetic graphs are generated with the SNAP graph library [34]. The RMAT [39] model is used to ensure that the generated graphs are scale free and resemble the characteristics of real-world graphs (e.g, follows the power-law degree distributions).

We conduct the experiments by using WolfGraph and the BFS algorithm to process 10 synthetic RMAT graphs across a range of different sizes and sparsities. The experiment results are shown in Figure 18 and Figure 19. In these figures, we only record the kernel computation time.

Figure 18 shows the trend of the execution time as the number of edges and vertices in the graph increase. In this experiment, the average degree of the graphs is fixed to be 16. As can be seen from the figure, As the graph size increases, the computation time increases, which is to be expected. As the graph becomes bigger, the kernel takes more memory transactions to fetch the data from the global memory and write the results back.

In next experiment, the graphs are fixed to have 64 million edges, but the degree of the graphs increases from 8 to 128. As can be see from Figure 19, the computation time decreases as the graph degree increases. As the graph degree increases, the

number of vertices in the graph decreases. Consequently, the frequency of random access to the vertex array is reduced. The kernel performance is therefore improved.

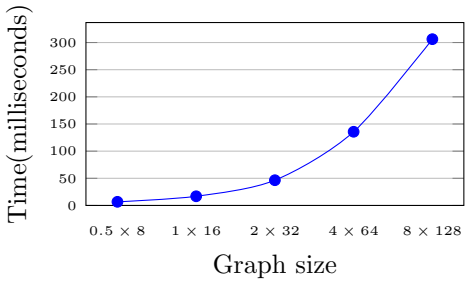


Figure 18: Execution time of WolfGraph as the graph size increases (with the graph degree being fixed to 16. $x \times y$ on the x-axis means x million vertices by y million edges

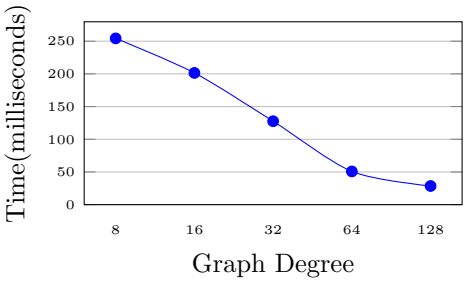


Figure 19: Execution time of WolfGraph as the graph degree increases (with the number of edges being fixed to 64 million)

7. Related work

7.1. In-Memory Graph Processing

Using GPU for graph processing was first introduced by Harish et al. [21]. Since then, the CSR format has become the mainstream representation to store graphs on GPU. Merrill et al. [40] present a work efficient BFS algorithm. They also use different approaches to minimize the workload imbalance. Virtual Warp Centric has been proposed in [20] to tackle the workload imbalance problem and reduce the intra-warp divergence.

Medusha [41] is a GPU-based graph processing framework that focuses on the abstractions for easy programming. MapGraph [42] implements the runtime-based optimisation to deliver desired performance. Based on the size of the frontier and the size of the adjacency lists for the vertices in the frontier, MapGraph can choose 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 [18] uses the G-Shard and the CW (concatenated window) representation to avoid non-coalesced memory access. The G-shard representation is the same as the shard structure in GraphChi. Because each G-shard is processed by one thread block in CuSha, the threads within one thread block will update the values of the outgoing edges in other thread blocks after the computation. However, updating vertex values in this way will cause the underutilization of GPU. To overcome this problem, CuSha develops the CW representation, which is an array that combines all incoming and outgoing edges of each G-shard. Similar to G-shard, each CW is processed by one thread block. Because all edges are grouped together, the threads do not need to access other thread blocks' data to update their values. Although CuSha's methods are effective, such representations consume 2 to 2.5 times more space than CSR, which can hinder the framework from processing very large graphs. In addition, as shown in the last section, CuSha requires a significant amount of time to pre-process the graph, which leads to long overall execution time. WolfGraph not only simplifies the pre-processing procedure, but also achieve the similar performance as CuSha.

All of the above approaches make the fundamental assumption that the input graphs fit in the GPU memory, which limits the usage of these solutions. However, WolfGraph does not have such a restriction.

7.2. Out-of-Memory Graph Processing

Most existing out-of-memory graph processing frameworks are CPU based. These frameworks aim to process the graphs that do not fit into the host memory. For instance, GraphChi [13] is the first graph processing framework that can handle the large-scale graphs on a single PC. GraphChi uses the vertex-centric model. In order to process the graphs being loaded from the hard disk, it introduces two new techniques to process large graphs in a single PC.

GraphChi uses an innovative out-of-core data structure called *shard* to reduce the amount of random access to the hard disk. Before the computation, GraphChi first pre-processes the graph data.

The input data will be partitioned into sub-graphs, each of which is called a shard. Each shard contains a set of vertices and all the inward edges of these vertices. In each shard, the edges are sorted in the ascending order of source vertex ID. The partition method used by GraphChi guarantees that the number of edges in each shard is similar and the size of each shard should be able to fit in the memory. GraphChi also developed a method called parallel sliding windows (PSW). During the computation, GraphChi loads the first shard into the memory, and then searches other shards and loads out-edges (the source vertice of these edges are the destination vertice in the current shard) of the current shard from other shards into memory as well. Once the processing of the current shard is finished, it moves to the next shard, and repeats the above process. The whole computation terminates when all shards have been processed. Organising the graph into shards and computing with PSW can guarantee the sequential read from the hard disk, and hence maximises the performance of the hard disk I/O.

TurboGraph [43], a more recent vertex-centric graph processing framework designed for SSD. It improves on GraphChi by extracting more parallelism, overlapping CPU processing and disk I/O. GridGraph [12] divides edges into smaller grids rather than shards in GraphChi and applies a 2-level hierarchical partitioning of the grids, which organizes several adjacent grids into a larger virtual grid. This way, GridGraph not only ensures data locality but also reduces the amount of disk I/O.

Mosaic [8] leverage the advantage of NVMes to achieve high throughput during the graph processing. It also employs a hybrid execution model to perform the computation more efficiently. The fundamental ideas of WolfGraph are based on the X-Stream [28]. X-Stream also uses the edge-centric processing model and takes a binary formatted edge-list as input, which does not require preprocessing. But X-Stream is a CPU-based graph processing framework.

Totem [44] [45] is a hybrid platform that uses both GPU and CPU. It statically partitions the graphs between GPU and CPU memories based on the degree of vertices. However, as the graph size increases, only a fixed portion of a graph can fit in the GPU memory, resulting in the underutilization of GPU. Groute [46] is a multi-GPU programming model and framework. The framework is designed to process graphs in a node with multiple GPUs, which has different design objectives from WolfGraph.

GraphReduce [47] also aims to process the graphs bigger than 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 sub-set of vertices. The edges in each shard are sorted in a specific order. Tigr [14] proposes a class of novel structural transformations that can effectively reduce the irregularity of graphs, so that the transformed graphs can be processed more efficiently on GPU. Graphie [48] overlaps the data transformation and computation, and processes the graph asynchronously. These arrangements may lead to longer pre-processing time. Hence, we tried to avoid such designs in WolfGraph.

8. Conclusion

In this paper, we develop a graph processing framework called WolfGraph. The framework applies the edge-centric computation model, and proposes the edge list structure to represent a graph. We demonstrate that it is more efficient to use the edge-centric processing model and the edge list graph representation to process the graphs on GPU. We also propose a representation called the Concatenated Edge List to process the graphs that cannot fit into the GPU memory. WolfGraph achieves the similar performance as the existing GPU-based in-memory processing frameworks and achieves the significant speedup over the existing out-of-memory implementations.

There are several future directions of our work: (1) adding the hard disk-based solution to WolfGraph, so that the graph with the size greater than the CPU memory can be processed efficiently; (2) extending the WolfGraph to support distributed architecture; (3) designing an adaptive runtime system that dynamically selects the graph representation, computation model, etc., based on the graph data.

9. Acknowledgement

This work is partially supported by the National Key R&D Program of China 2018YFB1003201, Worldwide Byte Security Information Technology Ltd, Guangdong Key Laboratory project 2017B030314073.

Reference

[1] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, G. Czajkowski, Pregel: A system for large-scale graph processing, in: Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data, SIGMOD '10, ACM, New York, NY, USA, 2010, pp. 135–146. doi:10.1145/1807167.1807184.

[2] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, J. M. Hellerstein, Distributed graphlab: A framework for machine learning and data mining in the cloud, Proc. VLDB Endow. 5 (8) (2012) 716–727. doi:10.14778/2212351.2212354.

[3] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, C. Guestrin, Powergraph: Distributed graph-parallel computation on natural graphs, in: Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12), USENIX, Hollywood, CA, 2012, pp. 17–30.

[4] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, I. Stoica, Graphx: Graph processing in a distributed dataflow framework, in: Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation, OSDI'14, USENIX Association, Berkeley, CA, USA, 2014, pp. 599–613.

[5] A. Roy, L. Bindschaedler, J. Malicevic, W. Zwaenepoel, Chaos: Scale-out graph processing from secondary storage, in: Proceedings of the 25th Symposium on Operating Systems Principles, SOSP '15, ACM, New York, NY, USA, 2015, pp. 410–424. doi:10.1145/2815400.2815408.

[6] X. Zhu, W. Chen, W. Zheng, X. Ma, Gemini: A computation-centric distributed graph processing system, in: 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16), USENIX Association, Savannah, GA, 2016, pp. 301–316.

[7] M. Zhang, Y. Wu, Y. Zhuo, X. Qian, C. Huan, K. Chen, Wonderland: A novel abstraction-based out-of-core graph processing system, in: Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '18, ACM, New York, NY, USA, 2018, pp. 608–621. doi:10.1145/3173162.3173208.

[8] S. Maass, C. Min, S. Kashyap, W. Kang, M. Kumar, T. Kim, Mosaic: Processing a trillion-edge graph on a single machine, in: Proceedings of the Twelfth European Conference on Computer Systems, EuroSys '17, ACM, New York, NY, USA, 2017, pp. 527–543. doi:10.1145/3064176.3064191.

[9] P. Yuan, W. Zhang, C. Xie, H. Jin, L. Liu, K. Lee, Fast iterative graph computation: A path centric approach, in: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC '14, IEEE Press, Piscataway, NJ, USA, 2014, pp. 401–412.

[10] K. Wang, G. Xu, Z. Su, Y. D. Liu, 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), USENIX Association, Santa Clara, CA, 2015, pp. 387–401.

[11] P. Macko, V. J. Marathe, D. W. Margo, M. I. Seltzer, Llama: Efficient graph analytics using large multi-versioned arrays, in: 2015 IEEE 31st International Conference on Data Engineering, 2015, pp. 363–374. doi:10.1109/ICDE.2015.7113298.

[12] X. Zhu, W. Han, W. Chen, Gridgraph: Large-scale graph processing on a single machine using 2-level hierarchical partitioning, in: 2015 USENIX Annual Technical Conference (USENIX ATC 15), USENIX Association, Santa Clara, CA, 2015, pp. 375–386.

[13] A. Kyrola, G. Blelloch, C. Guestrin, Graphchi: Large-scale graph computation on just a PC, in: Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12), USENIX, Hollywood, CA, 2012, pp. 31–46.

[14] A. H. Nodehi Sabet, J. Qiu, Z. Zhao, Tigr: Transforming irregular graphs for gpu-friendly graph processing, in: Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '18, ACM, New York, NY, USA, 2018, pp. 622–636. doi:10.1145/3173162.3173180.

[15] X. Shi, X. Luo, J. Liang, P. Zhao, S. Di, B. He, H. Jin, Frog: Asynchronous graph processing on gpu with hybrid coloring model, IEEE Transactions on Knowledge and Data Engineering 30 (1) (2018) 29–42. doi:10.1109/TKDE.2017.2745562.

[16] M.-S. Kim, K. An, H. Park, H. Seo, J. Kim, Gts: A fast and scalable graph processing method based on streaming topology to gpus, in: Proceedings of the 2016 International Conference on Management of Data, SIGMOD '16, ACM, New York, NY, USA, 2016, pp. 447–461. doi:10.1145/2882903.2915204.

[17] Y. Wang, A. Davidson, Y. Pan, Y. Wu, A. Riffel, J. D. Owens, Gunrock: A high-performance graph processing library on the gpu, in: Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '16, ACM, New York, NY, USA, 2016, pp. 11:1–11:12. doi:10.1145/2851141.2851145.

[18] F. Khorasani, K. Vora, R. Gupta, L. N. Bhuyan, Cusha: Vertex-centric graph processing on gpus, in: Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing, HPDC '14, ACM, New York, NY, USA, 2014, pp. 239–252. doi:10.1145/2600212.2600227.

[19] C. Hong, A. Sukumaran-Rajam, J. Kim, P. Sadayappan, Multigraph: Efficient graph processing on gpus, in: 2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT), 2017, pp. 27–40. doi:10.1109/PACT.2017.48.

[20] S. Hong, S. K. Kim, T. Oguntebi, K. Olukotun, Accelerating cuda graph algorithms at maximum warp, in: Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming, PPoPP '11, ACM, New York, NY, USA, 2011, pp. 267–276. doi:10.1145/1941553.1941590.

[21] P. Harish, P. J. Narayanan, Accelerating large graph algorithms on the gpu using cuda, in: S. Aluru, M. Parashar, R. Badrinath, V. K. Prasanna (Eds.), High Performance Computing – HiPC 2007, Springer Berlin Heidelberg, Berlin, Heidelberg, 2007, pp. 197–208.

[22] A. Davidson, S. Baxter, M. Garland, J. D. Owens, Work-efficient parallel gpu methods for single-source shortest paths, in: 2014 IEEE 28th International Parallel and Distributed Processing Symposium, 2014, pp. 349–359. doi:10.1109/IPDPS.2014.45.

[23] F. Busato, N. Bombieri, Bfs-4k: An efficient implementation of bfs for kepler gpu architectures, IEEE Transactions on Parallel and Distributed Systems 26 (7) (2015) 1826–1838. doi:10.1109/TPDS.2014.2330597.

[24] D. Li, M. Becchi, Deploying graph algorithms on gpus: An adaptive solution, in: 2013 IEEE 27th International Symposium on Parallel and Distributed Processing, 2013, pp. 1013–1024. doi:10.1109/IPDPS.2013.101.

[25] Y. Guo, A. L. Varbanescu, A. Iosup, D. Epema, An empirical performance evaluation of gpu-enabled graph-processing systems, in: 2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, 2015, pp. 423–432. doi:10.1109/CCGrid.2015.20.

[26] J. Malicevic, B. Lepers, W. Zwaenepoel, Everything you always wanted to know about multicore graph processing but were afraid to ask, in: 2017 USENIX Annual Technical Conference (USENIX ATC 17), USENIX Association, Santa Clara, CA, 2017, pp. 631–643.

[27] F. McSherry, M. Isard, D. G. Murray, Scalability! but at what cost?, in: Proceedings of the 15th USENIX Conference on Hot Topics in Operating Systems, HOTOS'15, USENIX Association, Berkeley, CA, USA, 2015, pp. 14–14.

[28] A. Roy, I. Mihailovic, W. Zwaenepoel, X-stream: Edge-centric graph processing using streaming partitions, in: Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles, SOSP '13, ACM, New York, NY, USA, 2013, pp. 472–488. doi:10.1145/2517349.2522740.

[29] W. Yang, K. Li, Z. Mo, K. Li, Performance optimization using partitioned spmv on gpus and multicore cpus, IEEE Transactions on Computers 64 (9) (2015) 2623–2636. doi:10.1109/TC.2014.2366731.

[30] X. Zhu, K. Li, A. Salah, L. Shi, K. Li, Parallel implementation of mafft on cuda-enabled graphics hardware, IEEE/ACM Transactions on Computational Biology and Bioinformatics 12 (1) (2015) 205–218. doi:10.1109/TCBB.2014.2351801.

[31] J. Liu, K. Li, D. Zhu, J. Han, K. Li, Minimizing cost of scheduling tasks on heterogeneous multicore embedded systems, ACM Trans. Embed. Comput. Syst. 16 (2) (2016) 36:1–36:25. doi:10.1145/2935749.

[32] C. Chen, K. Li, A. Ouyang, Z. Tang, K. Li, Gpu-accelerated parallel hierarchical extreme learning machine on flink for big data, IEEE Transactions on Systems, Man, and Cybernetics: Systems 47 (10) (2017) 2740–2753. doi:10.1109/TSMC.2017.2690673.

[33] Nvidia cuda, `http://www.nvidia.com/cuda`.

[34] D. A. Bader, K. Madduri, 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, 2008, pp. 1–12. doi:10.1109/IPDPS.2008.4536261.

[35] J. Leskovec, A. Krevl, SNAP Datasets: Stanford large network dataset collection, `http://snap.stanford.edu/data` (Jun. 2014).

[36] `http://law.di.unimi.it/webdata/hollywood-2011/`.

[37] `http://law.di.unimi.it/webdata/arabic-2005/`.

[38] `http://law.di.unimi.it/webdata/uk-2002/`.

[39] D. Chakrabarti, Y. Zhan, C. Faloutsos, R-MAT: A Recursive Model for Graph Mining, pp. 442–446. doi:10.1137/1.9781611972740.43.

[40] D. Merrill, M. Garland, A. Grimshaw, Scalable gpu graph traversal, in: Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '12, ACM, New York, NY, USA, 2012, pp. 117–128. doi:10.1145/2145816.2145832.

[41] J. Zhong, B. He, Medusa: Simplified graph processing on gpus, IEEE Transactions on Parallel and Distributed Systems 25 (6) (2014) 1543–1552. doi:10.1109/TPDS.2013.111.

[42] Z. Fu, M. Personick, B. Thompson, 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, ACM, New York, NY, USA, 2014, pp. 2:1–2:6.

[43] W.-S. Han, S. Lee, K. Park, J.-H. Lee, M.-S. Kim, J. Kim, H. Yu, Turbograph: A fast parallel graph engine handling billion-scale graphs in a single pc, in: Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '13, ACM, New York, NY, USA, 2013, pp. 77–85. doi:10.1145/2487575.2487581.

[44] A. Gharaibeh, L. Beltrão Costa, E. Santos-Neto, M. Ripeanu, 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, ACM, New York, NY, USA, 2012, pp. 345–354. doi:10.1145/2370816.2370866.

[45] A. Gharaibeh, L. B. Costa, E. Santos-Neto, M. Ripeanu, On graphs, gpus, and blind dating: A workload to processor matchmaking quest, in: 2013 IEEE 27th International Symposium on Parallel and Distributed Processing, 2013, pp. 851–862. doi:10.1109/IPDPS.2013.37.

[46] T. Ben-Nun, M. Sutton, S. Pai, K. Pingali, Groute: An asynchronous multi-gpu programming model for irregular computations, SIGPLAN Not. 52 (8) (2017) 235–248. doi:10.1145/3155284.3018756.

[47] D. Sengupta, S. L. Song, K. Agarwal, K. Schwan, 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, ACM, New York, NY, USA, 2015, pp. 28:1–28:12.

[48] W. Han, D. Mawhirter, B. Wu, M. Buland, Graphie: Large-scale asynchronous graph traversals on just a gpu, in: 2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT), 2017, pp. 233–245. doi:10.1109/PACT.2017.41.