This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2016.2611659, IEEE Transactions on Parallel and Distributed Systems

Optimizing Graph Processing on GPUs

Wenyong Zhong, Jianhua Sun, Hao Chen, Jun Xiao, Zhiwen Chen, Cheng Chang, and Xuanhua Shi

Abstract—Distributed vertex-centric model has been recently proposed for large-scale graph processing. Due to the simple but efficient programming abstraction, similar graph computing frameworks based on GPUs are gaining more and more attention. However, prior works of GPU-based graph processing suffer from load imbalance and irregular memory access because of the inherent characteristics of graph applications. In this paper, we propose a generalized graph computing framework for GPUs to simplify existing models but with higher performance. In particular, two novel algorithmic optimizations, lightweight approximate sorting and data layout transformation, are proposed to tackle the performance issues of current systems. With extensive experimental evaluation under a wide range of real world and synthetic workloads, we show that our system can achieve 1.6x to 4.5x speedups over the state-of-the-art.

Index Terms—GPGPU, Graph Computing, Pregel, Bulk Synchronous Model, Load Imbalance.

1 Introduction

With the rapid development of the Internet, processing very large web graphs has become a hot research issue in both the academia and industry. Large-scale graph processing frameworks are becoming increasingly important for solving problems in scientific computing, data mining, and other domains such as social networks. For example, finding the shortest paths of on-line maps, the citation relationships among twitter forwarding, and the purchasing preference in E-commerce webs, are all typical scenarios that heavily rely on efficient graph computation. However, developing graph processing algorithms over large dataset is challenging in terms of programmability and performance. Thus, a general graph programming framework that provides supports for high performance processing of a wide range of graph algorithms is often desired.

As a result, in the last several years, we have witnessed a growing interest in distributed graph processing, such as Pregel, GraphLab, PowerGraph, GPS, and Mizan, which are purposely-built distributed graph computing systems with easy-to-use programming interfaces and reasonable performance under large-scale workloads. The prominent one is Pregel [24], which was firstly proposed in 2010 as a programming model to address the challenges in parallel computing of large graphs. The high-level programming model of Pregel plays a significant role in abstracting architectural details of parallel computing from programmers. Specifically, the vertex-centric programming model proposed by Pregel greatly relieves the efforts of performing computation on large-scale data-intensive graphs, and provides high expressibility for a wide range of graph algorithms. Similar to MapReduce [13] whose programming model has been adopted successfully in many mainstream parallel environments, the advantages of Pregel have inspired the research of applying the Pregel-like graph computing model to many parallel architectures (e.g. multi-core, GPU, and heterogeneous hybrid systems), such as Grace [27], Medusa [37], and TOTEM [15].

With the advancement of GPU hardware and the introduction of GPU programming frameworks such as CUDA [5] and OpenCL [25], GPU has become a more generalized computing device. General purpose computing on GPU (GPGPU) has found its way into many fields as diverse as biology, linear algebra, cryptography, image processing, and so on, given the tremendous computational power provided by GPUs such as massive parallel threads and high memory bandwidth as compared to CPUs. In parallel to this trend, GPUs are increasingly leveraged to accelerate graph algorithms with either GPU-specific optimizations [18] or Pregel-like programming interfaces to hide the hardware intricacies [37], [7], [35], [29]. However, it is still challenging to design a GPU-based graph computing system that can exploit the hardware characteristics of GPUs efficiently, and provide a flexible user interface at the same time.

Existing graph processing systems mainly use the Compressed Sparse Row (CSR) representation of graphs due to its compact storage requirement. However, the low storage space consumption comes at a cost of non-coalesced access to GPU memory because of poor memory access locality. CuSha [7] is a graph processing framework that enables users to write vertex-centric algorithms on GPU, proposing a new graph representation called G-Shards to minimize non-coalesced memory accesses. However, this representation is not space efficient, which may exacerbate the situation when more GPU memory is needed to process large graphs. Medusa [37] is a general purpose GPU-based graph processing framework that provides high-level APIs for easy programming and scales to multiple GPUs. The achievable performance of Medusa may be limited by its internal data organization and processing logic that result in both irregular memory access and imbalanced workload distribution among GPU threads. [18] proposes techniques such as deferring outliers and dynamic workload distri-

- *Wenyong Zhong, Jianhua Sun, Hao Chen, Jun Xiao, Zhiwen Chen, Chang Cheng are with College of Computer Science and Electronic Engineering, Hunan University, Changsha, China. E-mail: {hnuzwy,jhsun,haochen}@hnu.edu.cn.*
- *Xuanhua Shi is with School of Computers, Huazhong University of Science and Technology, Wuhan, China. E-mail: {xhshi}@hust.edu.cn*

1045-9219 (c) 2016 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2016.2611659, IEEE Transactions on Parallel and Distributed Systems

bution to alleviate intra-warp divergence and achieve a balanced load among different warps. However, the improvement in performance is limited because of its relatively heavyweight implementation.

In this paper, we present a general graph processing framework for GPUs, and our goal is to provide a simpler programming model without any performance loss and expressibility reduction. In particular, two novel algorithmic optimizations, lightweight approximate sorting and data layout transformation, are proposed to tackle the performance issues in existing frameworks. The approximate sorting can efficiently alleviate load imbalance on the GPU due to non-uniform degree distribution in graphs. The data layout transformation is effective in optimizing memory representation of key data structures to coalesce memory access. We believe that the proposed optimizations are not tightly-coupled to our system, and applicable to other similar frameworks.

The main contributions of this paper include:

- In order to exploit the fine-grained parallelism on GPUs, we present a general graph computing framework using a customized Edge-Vertex programming model instead of the traditional vertex-centric model. Compared with existing GPU-based frameworks such as Medusa that employs a more complex programming model, our system offers a simpler interface without the reduction of performance. We strike a balance between the programming simplicity and the exploitation of performance on GPUs.
- We recognize that load imbalance among GPU threads in existing graph processing systems has important implications on performance. So, we introduce an efficient approach called *approximate sorting* to address this issue, which greatly alleviates imbalance at minimal cost. In addition, we observe that key data structures in GPU-based graph systems may suffer from severe non-coalesced memory accesses, which significantly hinders scalability. Thus, we propose a lightweight algorithm to transform the original row-major layout to column-major layout to mitigates non-coalesced access patterns.
- We conduct extensive performance evaluations on a set of representative graph datasets that include both real world power-law graphs and synthesized random graphs, and the results indicate that our system outperforms an existing graph processing system.

2 BACKGROUND

In this section, we present the necessary background on the Pregel programming model and GPU architecture.

2.1 Pregel Programming Model

The Pregel programming model is inspired by the Bulk Synchronous Parallel model [34]. In this model, programmers express the parallelism of graph computation by a sequence of iterations called supersteps. During each superstep, the framework invokes an user-defined function for each vertex in parallel. Inside this function, the vertices receive its incoming messages from other vertices in the prior superstep, then the vertices update their values and send messages to other vertices that will be used in the next superstep [28]. Although simple, the Pregel programming model is flexible to express many graph algorithms.

Several variants of the Pregel programming model were also proposed such as some open-source implementations Hama [2], Giraph [1] and GPS [28], which target distributed environments. At the same time, efforts are devoted to extending the original Pregel model in order to optimize the performance for other parallel architectures. For example, Medusa [37] is an efficient implementation of applying the Pregel model to GPU platforms, with some distinguishable features to accommodate the inherent characteristics of GPUs.

Medusa provides a more fine-grained programming interface than Pregel, exposing fine-grained data parallelism on edges, vertices and messages, which is called EMV model. This model enhances the vertex-centric model to provide support for efficient graph processing on GPUs. Using the APIs offered by Medusa, programmers can define computations on vertices, edges and messages respectively. However, the EMV model is still complicated for programmers compared with the vertex-centric interface. Although Medusa processes edges, vertices, and messages separately with different GPU kernels to exploit GPU parallelism, load imbalance among GPU threads within warps/thread blocks still exists, leading to GPU resource underutilization.

Medusa proposes a graph-aware buffer scheme for the message buffer. In the EMV model, messages are usually sent or received along the edge. Therefore, an edge can send one or more messages to the message buffer. The size of the message buffer is pre-defined according to the number of edges in the graph and the maximum messages that an edge will send. The locations to store messages on GPU are established on CPU by constructing a reverse graph. As a result, the write positions of messages sent to the same vertex are consecutive. However, when vertices read messages from the message buffer, the threads in a warp would read the randomly scattered messages in the global memory, violating the requirement of coalesced memory access.

2.2 GPU and CUDA Programming Framework

The current generation of GPUs have thousands of processing cores that can be used for general-purpose computing. For example, the Kepler GPU GTX780 consists of 12 Streaming Multiprocessors (SMXs), each equipped with up to 192 Stream Processors (SPs). Each SMX has 64 KB of onchip memory that can be configured as 48 KB of shared memory with 16 KB of L1 cache, or as 16 KB of shared memory with 48 KB of L1 cache, and 1536 KB L2 cache is shared by all SMXs. In addition to the L1 cache, Kepler introduces a 48 KB read-only data cache. Each SMX has 64 KB 32bit registers equally split to the threads running in one block. In contrast, the off-chip global memory has a much larger size (typically in GB range) and longer access latency.

The schedulable execution unit on the GPU is called a *warp* formed by a group of 32 threads. Warps are grouped together into *cooperative thread arrays* (CTAs), which are correspondingly structured as a *grid*. Typically, the threads

1045-9219 (c) 2016 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2016.2611659, IEEE Transactions on Parallel and Distributed Systems

in a warp follow the same execution path and operate on distinct data in SIMT (Single Instruction Multiple Threads) fashion in order to achieve maximal parallelism. Warp divergence may occur when there are conditional branches taken on the execution path. Launching a large number of threads concurrently is a recommended way to hide the latency of global memory access and to better utilize the computational resources on the GPU.

CUDA (Compute Unified Device Architecture) is a GPGPU programming framework from NVIDIA. CUDA supports various memory spaces, such as register, constant, local, parameter, texture, shared, and global memory, which differ in size, addressability, access speed and access permissions. Perhaps the single most important performance consideration in programming for GPU architectures is the coalesced access to the global memory [5].

3 System Design

Our system is composed of two major parts: the Pregel-like high-level programming interface and the customized runtime for the GPU. The programming interface can simplify the programming task of using GPUs for graph computation. The runtime system can hide GPU hardware intricacies and specific optimizations. In this section, we first explain why existing graph computing models are not sufficient and propose the Edge-Vertex model. Then, we describe the workflow and APIs of our system. At last, we analyze the potential performance bottleneck of the proposed computing model.

3.1 Edge-Vertex Model

First, we describe the reason why the vertex-centric model can not fit in the GPU architecture directly. In the vertex-centric model, the developer needs to define a function (e.g., *compute*) to perform computations on individual vertices (such as sending or receiving messages along edges). In order to fully utilize the computational resource of GPUs, we need to map the function to each GPU thread to exploit the fine-grained parallelism of GPUs. However, real world graphs often exhibit power-law degree distribution, which indicates that the workload assigned to each GPU thread may be imbalanced, leading to suboptimal performance. To address this issue, the EMV model proposed in Medusa splits the computation into multiple components, separating the processing of vertex, edge, and message to offer the developer more flexibility in designing algorithmic optimizations. However, this benefit comes at the cost of incurring programming complexity such as exposing the structure of the message buffer to allow explicit management of messages. However, the expressibility of defining GPU-based computation on vertices and edges would be sufficient for most graph algorithms. Further, encapsulating the message buffer in the runtime system can not only reduce programming complexity but also provide the opportunity of implementing optimizations for different GPU hardware architectures.

Therefore, we partition the computation in the vertex-centric model into two methods such as *EdgeCompute* and *VertexCompute*, which we call the Edge-Vertex model. It is based on the consideration that in our model edges are responsible for sending messages to the message buffer and vertices conduct the real computation with the messages received from the buffer. In other words, how the message buffer is constructed and managed is invisible to the end user, striking a balance between the flexibility in optimizing certain graph algorithms and the complexity in writing graph applications. Furthermore, this simplification does not necessarily imply performance reduction because specifically designed optimizations can greatly improve the overall performance as will be shown in this paper.

3.2 Workflow and APIs

Figure 1 illustrates the workflow of our system. At first, the runtime system constructs the CSR representation [17] for the input graph data, and common management tasks including memory allocation on the CPU and GPU and data transfer between the host memory and GPU are also automatically fulfilled by the runtime system.

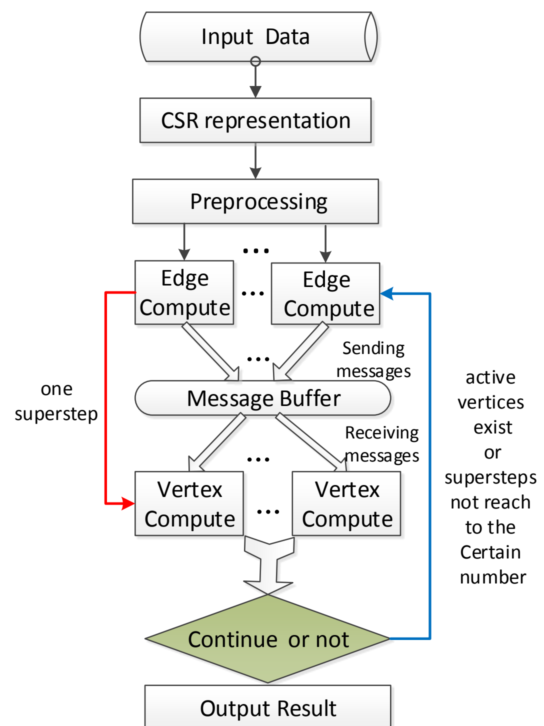


Fig. 1: The workflow of our system

3.2.1 *Preprocessing*

In our system, the preprocessing stage is managed by the runtime and is transparent to users. The internal message management module is an array-based buffer with associated operations of manipulating message delivery and reception. Before the actual graph processing, we first need to establish the accurate size for the chunked message buffer with each chunk representing the storage space for messages belonging to certain vertices. In particular, for a chunk of the message buffer that is assigned to a specific vertex, we need to calculate the positions where an edge can send messages to and a vertex can receive messages from

1045-9219 (c) 2016 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2016.2611659, IEEE Transactions on Parallel and Distributed Systems

in advance. The details will be given in section 4.2 when discussing the optimizations against the message buffering mechanism.

3.2.2 EdgeCompute stage

In the *EdgeCompute* stage, each GPU thread is responsible for one or more edges associated with a vertex and performs the operations of the following steps in order. First, the value and state of the source vertex are read. Second, the weight of the edge is obtained. Third, messages are delivered to destination vertices based on the results evaluated based on the vertex value and edge weight. Like the vertex-centric model, the state of each vertex is active at start, and the programmer can determine whether to vote to halt or not explicitly according to the algorithm's semantics.

3.2.3 VertexCompute stage

In the *VertexCompute* stage, one or more vertices are assigned to a GPU thread that takes the following three steps similar to the *EdgeCompute* stage. First, messages sent to the vertex are retrieved from the message buffer by the GPU thread. Second, the newly-obtained messages are used to compute the values for the vertex and its outgoing edges. Third, new state or value can be set to the vertex if needed. The invocation of the functions *EdgeCompute* and *VertexCompute* proceeds as a sequence of iterations until no active vertex exists or the pre-defined maximum number of supersteps are reached.

3.2.4 Message buffer

The message buffer is organized based on the total number of messages and the maximum number of messages each vertex can receive. Each edge is assigned a unique ID that is used as the index into the message buffer for both the stages. The ID values for the in-edges of a destination vertex are consecutive, which guarantees that messages sent to the same vertex are located contiguously in memory. Thus, the combiner in the VertexCompute stage can process the messages using a simple loop. This design together with the layout remapping discussed in Section 4.2 can greatly improve the overall performance.

3.2.5 Pregel-style API

In each superstep, each edge and vertex invoke the *EdgeCompute* and *VertexCompute* method provided by the user until certain conditions are met. The APIs we provide are shown in Table 1 with brief descriptions. Similar to existing frameworks, we divide the API into two categories: the user-implemented APIs and the system-provided APIs.

As for the user-implemented APIs, *initData* must be called to initialize data and configure parameters for threads and blocks on the GPU. As discussed earlier, programmers can define the computation with two separate functions *EdgeCompute* and *VertexCompute*. System-provided APIs are used as library calls to hide the GPU-specific programming details. Our system offers the method *startGPU* to invoke the device code, which in turn calls the corresponding user-defined functions. Two versions of *startGPU* are provided to control the iteration in different ways. One can be invoked with an explicit parameter setting the maximum supersteps, and the other can determine the completion of computation by checking whether there will be active vertex in the next superstep.

TABLE 1: Pregel-style APIs

user-implemented APIs
/* EdgeCompute function, processing one edge. */ ***void EdgeCompute(Edge e);*** /* VertexCompute function, processing one vertex. */ ***void VertexCompute(Vertex v);*** /* Initialize the value of edges and vertices, configure the number of threads and blocks on GPUs. */ ***void initData(void* graph,void* initalValues);***
system-provided APIs
/* Start the runtime system, and stop until reaching max-Supersteps specified by the programmer. */ ***void startGPU(int maxSupersteps);*** /* Start the runtime system, and stop until no active vertices exist. */ ***void startGPU();*** /* Set the state of the vertex to inactive. */ ***void voteToHalt(Vertex v);*** /* Set the state of the vertex to active. */ ***voteToActive(Vertex v);*** /* Set the parameter continue to true, if there are still active vertices sending messages in the next superstep. */ ***void stillContinueNextSupersteps(bool continue);*** /* Combine the incoming messages sent to the same vertex. */ ***Message Combiner(Vertex v);***

3.2.6 A Running Example

In graph algorithms, finding the single source shortest paths (SSSP) in a graph is a well-known and easy-to-understand graph algorithm [12]. For this graph algorithm, we need to specify a vertex as the single source vertex and find a shortest path between the source vertex and every other vertex in the graph. We use a running example to illustrate how the three major functions *EdgeCompute*,*VertexCompute*, and *Combiner* are defined to implement the SSSP algorithm in our system. Algorithm 1 shows the expressibility and simplicity of the Edge-Vertex model in writing graph algorithms for GPUs.

The SSSP procedure works as follows. In each iteration, we first get the source node of each edge, and send messages to sink nodes if the source node is updated in the previous iteration. Then, in the vertex compute stage, the minimum value is calculated by scanning a segment of the message buffer that belongs to the current vertex. If it is less than the value from the previous iteration, the old value is updated with the new one, and the vertex votes to continue to the next step. Otherwise, the vertex deactivates itself by voting to halt. The whole procedure terminates when all vertices are simultaneously inactive.

3.3 Performance Bottleneck Analysis

Although our Edge-Vertex model can guarantee a balanced load distribution among GPU threads in the *EdgeCompute* stage because of the coalesced memory access ensured by the predetermined write positions, load imbalance still exists in the *VertexCompute* stage. For example, in the *VertexCompute* method of the SSSP algorithm, we need to combine messages belonging to a vertex. But the number of messages that a vertex receives may vary significantly due

1045-9219 (c) 2016 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

Algorithm 1 The pseudo code of the SSSP algorithm

```
void EdgeCompute(Edge e)
1: Vertex srcVertex=e.getSrcVertex();
2: if srcVertex.active then
3:    e.sendMsg(srcVertex.value+e.weight);
4: end if
---------------------------------------------
void VertexCompute(Vertex v)
1: msg_min=Combiner(v);
2: if msg_min < v.value then
3:    vertex.value=msg_min;
4:    voteToActive(v);
5:    stillContinueNextSuperstep(true);
6: else
7:    voteToHalt(v);
8: end if
---------------------------------------------
typedef: Message int
Message Combiner(Vertex v)
1: min=INF;
2: for each i ∈ v.inEdgesNum do
3:    if min > v.currentMsg() then
4:       min=v.currentMsg();
5:    end if
6:    v.nextMsg();
7: end for
8: return min;
```

to highly variable distribution of vertex degree in a graph, which causes not only divergence within a warp but also resource underutilization among blocks. In section 4.1, we propose a lightweight but effective optimization mechanism to address this issue.

Another potential performance bottleneck comes from the *Combiner* method where each GPU thread reads messages from the message buffer in a non-coalesced way, because the message buffer is initialized on the CPU that favors a row-major memory layout and transfered to the GPU with the memory structure unchanged. But on the GPU we are often recommended to use a column oriented layout for data structures to fit in with the practice of GPU memory access optimization. However, it is difficult to arrange the message buffer in a way that works well for both cases. Thus, we will present another lightweight solution to optimize memory layout in section 4.2.

4 Optimizations

In this section, we present two novel optimizations to improve the performance of the runtime system. As stated earlier, we assume the CSR representation for the graph on the GPU.

4.1 Optimizing Load Imbalance with Approximate Sort

Inherent irregularity in some applications may cause imbalanced workload distribution among GPU threads, resulting in ineffective utilization of compute resource. In our case, each vertex must receive incoming messages from the message buffer, while the maximum size of incoming messages for a vertex is determined by the in-degree of the vertex. As the in-degree distribution of a graph varies greatly, processing vertices with high variance of incoming messages in a warp or thread block will lead to workload imbalance. For example, a single thread in a warp processing comparatively large incoming messages for the vertex would introduce wasted resource as every vertex will take as many cycles as the largest one to process in the warp.

implications: Because current GPUs provide no support for fine-grained thread-level task scheduling, tackling the inefficiency incurred by workload irregularity imposes the responsibility of fine-tuning the algorithm on developers. An ideal solution to this problem should work well for a wide range of workloads. For example, it should achieve highest possible performance for highly irregular workloads, and at the same time guarantee lowest possible cost for regular workloads.

Approaches: To address this issue, one possible approach is to identify outliers in incoming messages to defer the processing of these messages to subsequent kernels [18]. For example, vertices with an excessive large/small number of incoming messages are delayed to process. However, this method incurs considerable overhead, so more lightweight solutions are preferred. Although existing work [31] indicates that it is not worthwhile to group packets with identical or similar size by GPU sorting algorithms when accelerating the network stream processing with GPUs. However, we found that pre-sorting the message buffer according to the vertex in-degree on the CPU can result in a significant performance increase for the vertex kernel. Moreover, balancing the workload within warps or thread blocks does not necessarily require a strict ordering. These investigations are the direct motivation of designing a GPU-based approximate sorting algorithm for our system as detailed below.

Before discussing the details of the approximate sorting, we first present a high-level overview about this algorithm. Conceptually, the approximate sorting maps elements to different buckets with a linear projection, and the ordering among buckets is guaranteed (all the elements in a bucket must be larger or smaller than those in its consecutive bucket). However, we do not maintain ordering in buckets (elements in a bucket are unordered, but the elements are similar in size). In comparison, traditional sorting algorithms require a precise ordering among all elements, which is not necessary for our purpose and incurs more overhead.

As shown in Figure 2, our algorithm operates in three steps. First, each element in the size array (calculated based on vertex in-degrees) is mapped into a bucket (the number of buckets is a pre-defined parameter and typically much less than the array size). In this step, we maintain an ordering among all elements that are mapped into the same buckets and a counter array that records the size of each bucket. Second, an exclusive prefix sum operation is performed on the counter array. In the third step, the results of the above two steps are combined to produce the final coordinates that are then used to transform the input vector to an approximately-sorted form.

Step 1: Similar to many parallel sort algorithms that subdivide the input into equally-sized buckets and then sort each bucket in parallel, we first map each element of the size

1045-9219 (c) 2016 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2016.2611659, IEEE Transactions on Parallel and Distributed Systems

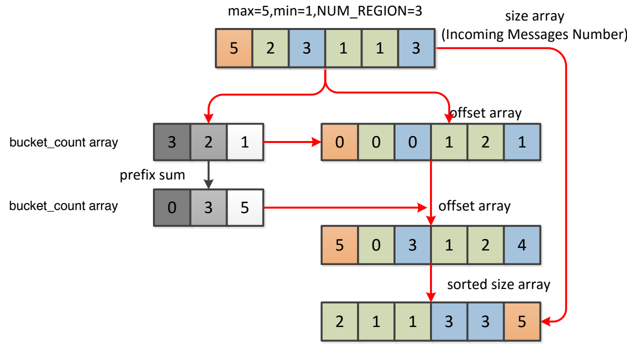


Fig. 2: Illustration of approximate sort

Listing 1: Assigning elements to buckets.

```
1  __global__ void assign_bucket(uint *input,uint lenght,uint max,uint min,
2                                uint *offset,uint *bucket_count,uint *bucket_index)
3  {
4   int idx = threadx.x + blockDim.x * blockIdx.x;
5   uint bucket_idx;
6   for (; idx < lenght; idx += total_threads)
7   {
8     uint value=input[idx];
9
10    bucket_idx = (size − min)*(NUM_BUCKETS − 1)/(max − min);
11    bucket_index[idx] = bucket_idx;
12
13    offset[idx] = atomicInc(&bucket_count[bucket_idx], length);
14  }
15 }
```

array into a bucket. As shown in Listing 1, the number of buckets is a fixed value NUM_BUCKETS, and the mapping procedure is a linear projection of each element in the input vector to one of the NUM_BUCKETS buckets. The linear projection is demonstrated at lines 10 and 11 in Listing 1, where the variables of *min* and *max* represent the minimum and maximum value in the input respectively, which can be obtained efficiently on GPUs. In this way, each bucket represents a partition of the interval [min, max], and all buckets have the same width of $\frac{max-min}{NUM_BUCKETS}$. The elements in the input vector are assigned to the target bucket whose value range contains the corresponding element. In addition, another *count array* is maintained to record the number of elements assigned to each bucket. As shown at line 13, the counting is based on an atomic function provided by CUDA, *atomicInc*, to avoid the potential conflicts incurred by concurrent writes. The function *atomicInc* returns the old value located at the address presented by its first parameter, which can be leveraged to indicate the local ordering among all the elements assigned to the same bucket. The Kepler GPUs has substantially improved the throughput of global memory atomic operations as compared to Fermi GPUs, which also has been observed in our implementation.

Step 2: Having obtained the counters for each bucket and the local ordering within a specific bucket, we perform a *prefix sum* operation on the counters to determine the address at which each bucket's data would start. Given an input array, the prefix sum, also known as scan, is to generate a new array in which each element i is the sum of all elements up to and including/excluding i (corresponding to inclusive and exclusive *prefix sum* respectively). Because

Listing 2: the key step of approximate sort.

```
1  __global__ void appr_sort(uint *key,uint *key_sorted,void *value,uint length,
2                            void *value_sorted,uint *offset,uint *bucket_count,
3                            uint *bucket_index,uint *oldToNew)
4  {
5   int idx = threadIdx.x + blockDim.x * blockIdx.x;
6   uint count = 0;
7   for (; idx < length; idx += total_threads)
8   {
9     uint key = key[idx];
10    uint value = value[idx];
11
12    uint bucket_index = bucket_index[idx];
13    count = backet_count[bucket_index];
14    uint off = offset[idx];
15    off = off + count;
16
17    key_sorted[off] = key;
18    value_sort[off] = value;
19    oldToNew[idx] = off;
20  }
21 }
```

the length of the count array (NUM_BUCKETS) is typically less than that of the length of the input, performing the scan operation on CPU is much fast than the GPU counterpart. However, due to the data transfer overhead (in our case, two transfers), and the fact that we observed devastating performance degradation when mixing the execution of the CPU-based scan with other GPUs kernels in a CUDA stream, the parallel *prefix sum* is performed on the GPU using the CUDPP library [4].

Step 3: By combining the atomically-incremented offsets generated in step 1 and the bucket data locations produced by the *prefix sum* (as shown at lines 12-15 in Listing 2), it is straightforward to scatter the key-value pairs to proper locations (see lines 17-18). With the sorted offset array, threads in the same warp or block is able to process incoming messages that are similar in size, leading to balanced workload distribution. A side effect of the sorting operation is that we can not directly access the source vertex via an edge, because the location of the vertex has been changed. Therefore, we maintain another index array to trace the one-to-one correspondence between the old index and the new one (see line 19 in Listing 2). With the mentioned index array, we need to update the locations of the source and destination vertices.

Choosing a suitable value for the number of buckets may have important implications for the efficiency and effectiveness of our sorting algorithm. As the number of buckets increases, for inputs exhibiting uniform distribution of elements, our algorithm would approximate more closely to the ideal sorting, while the overhead of performing the prefix sum may increase accordingly. When decreasing the number of buckets, besides the effect of getting a coarse-grained approximation for the input vector, time variations for the kernel *assign-bucket* may occur as a result of using the atomic operation to resolve conflicts when multiple elements are assigned to the same bucket concurrently. We will present empirical evaluations on this in Section 5. Furthermore, the approximate sorting may incur additional overhead for workloads exhibiting abnormal degree distribution. For example, if a very large number of vertices in a graph have similar degrees, it would be not necessary to perform the approximate sorting. One solution is to examine

1045-9219 (c) 2016 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2016.2611659, IEEE Transactions on Parallel and Distributed Systems

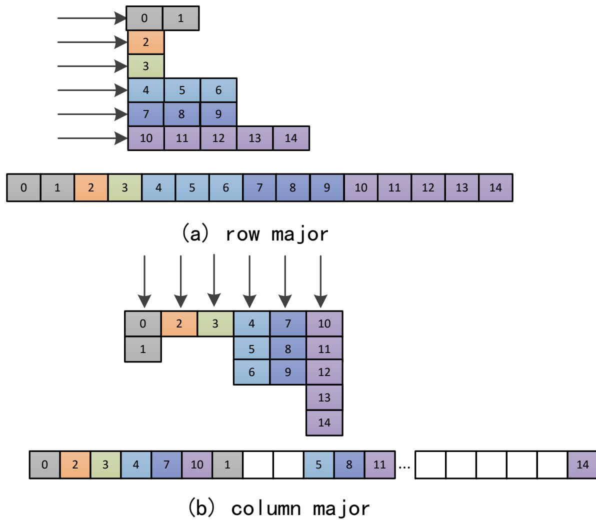


Fig. 3: Two kinds of data layouts

the difference between the maximum and minimum value. If it is below a specified threshold, the sorting operation can be disabled dynamically.

4.2 Data Layout Remapping of the Message Buffer

Given the fact that messages are always sent to the message buffer allocated and managed by the runtime system, we have two choices for the data layout of the array-based message buffer: row-major and column-major. The two types of data layouts are illustrated in Figure 3. The top half of both Figure 3 (a) and (b) is the conceptual representation, where arrows indicate the accessing threads. The actual memory layout is presented in the bottom half in Figure 3, from which we can see that threads can access contiguous memory location with the column-major layout, whereas memory accesses with the row-major layout are not coalesced. In this way, with the column-major layout, the GPU threads can access the data in a coalesced way as much as possible (some threads may have no data to process), while the row-major layout makes the global memory access among GPU threads separated with variable strides, violating the principle of coalesced memory access.

Medusa adopts an array-based buffer management mechanism for efficiently sending and receiving messages. In Medusa, the size of the message buffer is pre-determined according to the total number of edges in a graph, and then the exact locations to which edges will send messages are calculated on the CPU. As a result, the write positions of messages sent to the same vertex are guaranteed to be consecutive as shown in Figure 4. This scheme proposed by Medusa avoids dynamic memory allocation and atomic operations.

Although the programming model of attaching one vertex to one GPU thread is simple to understand and straightforward to implement, potential performance issues may arise if no proper guidance is made. One key observation is that, for certain applications (like ours), the difference of favored data layout between the CPU and GPU may result in suboptimal performance. On the CPU, applications

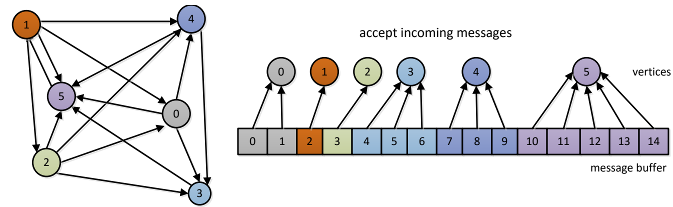


Fig. 4: The graph aware buffer scheme proposed by Medusa

may prefer row-major data layout because of its extensive support for features such as cache locality and prefetching at the hardware layer. In contrast, GPU applications benefit from memory coalescing, which requires a column-major data layout so that threads in a warp can access contiguous data and achieve better locality. To this end, we propose a data layout remapping algorithm against the message buffer to take full advantage of coalesced access to the global memory.

Implications: It is desirable to have the hardware, driver, or runtime provide mechanisms to automate the transformation of data layout, saving additional costs without compromising transparency. Unfortunately, such mechanisms do not exist in current GPUs platforms. Therefore, we need to implement our own data remapping approach that is light-weight enough to avoid both noticeable performance degradation and extra memory consumption.

Approaches: As shown in Figure 4, with the row major data layout, each GPU thread handling one vertex scans the relevant segment of the message buffer from the first element to the last iteratively, which violates the common practice of coalesced memory access. However, implementing strict column-major layout in a single array costs too much extra memory space. Because we can keep the global memory accesses within a warp as coalesced as possible, it is not necessary to maintain strict column-major data layout for all the GPU threads. The idea of our data remapping approach is to group vertices at the granularity of a warp. Therefore, the data layout within a group is column-major, and among groups the data layout is row-major. We will elaborate our data layout remapping algorithm in the following three steps, and for simplicity, we assume that three vertices consist of one group.

In the first step, similar to the approximate sorting that maps the input array into a bucket, we first split the *size array* into groups each with the same number of items. As demonstrated in Figure 5, the number of elements in each group is a constant value Per_Group_Num that is set to 3 in our example. As a consequence, we can obtain the number of groups using the formula $\lceil \frac{length(size)}{Per_Group_Num} \rceil$ and use the maximum value in each group to form the *group array*, which is implemented on the GPU to avoid the data transfer overhead between the CPU and GPU. The capacity of a vertex's incoming messages is determined by the maximum value of the group to which the vertex belongs.

The goal of step two is to produce the value of how much memory we need to allocate for the message buffer and generate the *offset array* for groups in the message buffer. Concretely, the capacity of a vertex's incoming messages

1045-9219 (c) 2016 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2016.2611659, IEEE Transactions on Parallel and Distributed Systems

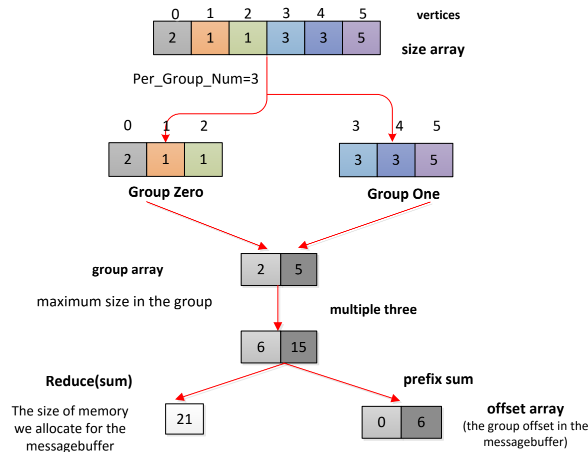


Fig. 5: Data layout remapping algorithm (step 1 - step 2)

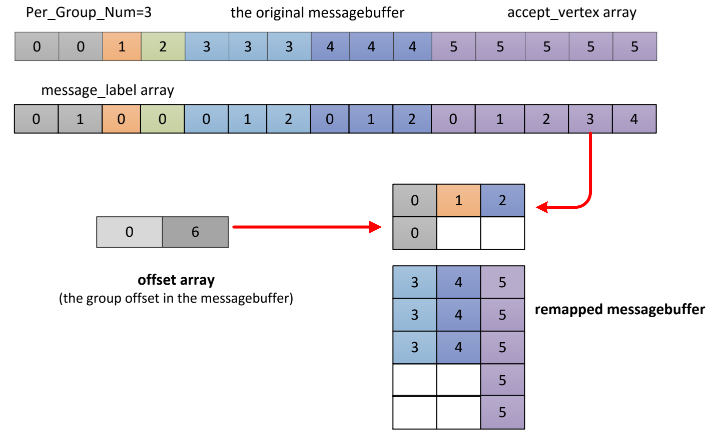


Fig. 6: Data layout remapping algorithm (step 3)

depend on which group it belongs to and the corresponding value in the *group array*. Each element in *group array* is multiplied by *Per_Group_Num*, which generates another new array. The elements in the new array indicate how much memory space each group should allocate. We can obtain the size of the message buffer by the reduce operation and use exclusive prefix sum to produce the *offset array* that represents each group's offset in the message buffer. Parallel prefix sum and reduce can be efficiently implemented on the GPU, and we rely on the CUDPP library [4] to perform these operations.

In the third step, our goal is to perform the remapping operation on the original message buffer. As depicted in Figure 6, the two arrays *accept_vertex* and *message_label* are part of the message buffer and used to facilitate the layout transformation. The former traces which vertex will read from the indicated location, and the latter records the index number for each segment (allocated for each vertex) in the message buffer. The *accept_vertex* array has the same number of elements as the message buffer, so we use it to illustrate the transformation in Figure 6. But the actual layout remapping is performed on the message buffer not the *accept_vertex* array. In this step, we also need the *offset array* deduced from the previous step. As presented in Listing 3, we first need to obtain the target vertex and the index used to access the message buffer from the *message_label* array before remapping (see lines 8-9). Next, we determine the group to which the vertex belongs and the corresponding index using $accept_vertex\%Per_Group_Num$ (see lines 11-12). After calculating the new position for the column-major layout (see lines 14-16), we can copy the content in the message buffer to new locations (line 18).

Although involving multiple kernel invocations, the data layout remapping incurs little performance overhead due to the lightweight design and the exploitation of GPU's specific features. The benefit of data remapping comes at the cost of supplementary GPU memory reserved to hold the remapped content. However, we can control the value of the parameter Per_Group_Num to manage the free memory. For example, when $Per_Group_Num = 1$, the data layout remains intact and there is no extra memory consumption.

Listing 3: **Data layout transformation kernel.**

```
1  __global__ void remap(int *accept_vertex_array, int* message_label_array,
2                        int *message_content,int length,
3                        int *offset_array,int *remap_message_content)
4  {
5    int idx = blockId.x * blockDim.x + threadIdx.x;
6    for (; idx < length; tid += total_threads)
7    {
8      int accept_vertex = accept_vertex_array[idx];
9      int message_label = message_label_array[idx];
10
11     int group_id = accept_vertex / Per_Group_Num;
12     int group_inner_id = accept_vertex % Per_Group_Num;
13
14     int offset_group = offset_array[group_id];
15     int new_position = offset_group + group_inner_id;
16     new_position += message_label * Per_Group_Num;
17
18     remap_message_content[new_position] = message_content[idx];
19   }
20 }
```

Therefore, choosing a suitable value for Per_Group_Num may have important implications for the efficiency and effectiveness of our data layout remapping algorithm. As Per_Group_Num increases, the effect on memory coalescing will be better while memory consumption increases correspondingly. In our case, we set Per_Group_Num to 32 because the memory loads by threads of a warp are coalesced. Furthermore, we plan to investigate in-place data remapping approaches to alleviate this limitation in our future work. More detailed analysis about the performance and space overhead can be found in Section 5.

Our data layout remapping is executed after the approximate sorting because the incoming messages associated with the vertices in a group are roughly equal, which can reduce memory space consumption. In the preprocessing phase as mentioned above, we need to establish the size of the message buffer and determine the positions for message delivering and receiving. Thus, the data layout remapping is integrated as part of the preprocessing phase, and it is not necessary to remap the message buffer in each superstep.

5 Experimental Results

In this section, we present the experimental results from four aspects. First, we evaluate the overall performance

1045-9219 (c) 2016 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2016.2611659, IEEE Transactions on Parallel and Distributed Systems

TABLE 2: Characteristics of workloads used in the experiments

Workload	In short	Nodes	Edges	*Max(d)*	η
ego-Gplus	ego	107,614	13,673,453	17058	99.9%
soc-Pokec	soc	1,632,803	30,622,564	6808	76.3%
com-LiveJournal	com	3,997,962	34,681,189	2454	76.7%
cit-Patents	cit	3,774,768	16,518,948	779	61.9%
RMAT	RMAT	1,000,000	16,000,000	555	52.4%
Random	Rnd	1,000,000	16,000,000	41	99.9%

TABLE 3: the feature of machines used in the experiments

Feature	Intel Xeon E5 2648L	GeForce GTX 780
Cores/Proc	8	12
Hardware Threads/core	2	192
Frequency/Core(GHz)	1.8	0.90
Main Memary/proc(GB)	8 DDR3	3 DDR5
Memory Bandwidth/Proc(GB/s)	57.6	288.4

compared with an existing GPU-based graph processing framework Medusa. Second, we explore the efficiency of the preprocessing phase in our system. Third, we investigate the effectiveness of the approximate sorting algorithm. Forth, we analyze the performance impact of the data remapping algorithm.

5.1 Experiment Setup

Real world and synthetic graphs are both considered in this paper. Table 2 summarizes the characteristics of the workloads used in our experiments. The six workloads we use are all directed graphs. We define *d* as the in-degree of a vertex and *Max(d)* as the maximum in-degree in a graph. For simplicity, $d \geq 2$ means the number of vertices whose in-degree is greater than one, its percentage is represented by the variable η. Ego-Gplus [22] consists of circles from Google+ and the η is equal to 99.9%, indicating that the in-degree of vast majority of vertices is greater than one in the graph. Soc-Pokec [33] and Com-LiveJournal [36] both come from online social networks. The η of the two graphs is approximately the same (76%). Cit-Patents [21] is maintained by the National Bureau of Economic Research and its η is much smaller, implying that 38.1% of vertices in the graph have less than two in-edges. All the graphs exhibit power law degree distribution.

RMAT and Random are two synthetically generated workloads. RMAT has a power law degree distribution, and the η is 52.4%. Random is a uniformly distributed graph workload and *Max(d)* is much smaller than others. The average degree of both of the synthetic workloads [6] is configured to 16. Because this paper focuses on the issue of the load imbalance among GPU threads and the optimization of coalesced memory access, the major workloads we choose are power-law graphs. The experiments on the Random workload show that our optimizations are also effective to non-power-law graphs. We plan to conduct experiments on more real graphs such as meshes and road networks in future work.

The hardware platform in our experiments is shown in Table 3. The workstation is equipped with an Intel Xeon E5-2648L CPU with 8 GB of DDR3 memory, and contains a GeForce GTX 780 GPU with 12 multiprocessors each with 192 processing cores. The GPU has 3GB of DDR5 memory.

5.2 Overall Performance

We implement three representative graph algorithms using the Medusa API as the baseline to compare the overall performance between our system and Medusa. The execution time contains two parts: the preprocessing time and the graph processing time. The three representative graph algorithms used for comparison are: PageRank [8], single source shortest path (SSSP) algorithm [24], and the HCC algorithm to find connected components in a graph [28].

As shown in Figure 7, the PageRank algorithm is run for 50 iterations, and we can observe that the our system achieves 1.7x to 2.3x speedup. When evaluating the SSSP algorithm, we randomly select the source vertex in the workloads. Figure 8 demonstrates that the speedup of our system ranges from 1.6x to 4.5x. In Figure 9, we can see 1.7x to 3.7x performance improvement for the HCC algorithm as compared to Medusa. The SSSP algorithm with the dataset 'com' achieves the best speedup (4.5x), mainly because of its simpler internal implementation logic as compared to the other two algorithms. Given that the 'com' graph is the most complex dataset we tested, this significant speedup for SSSP, from another perspective, reflects the effectiveness of our optimizations in handling large graphs. In summary, our system can attain 1.6x to 4.5x speedup for the three representative graph algorithms across all workloads in contrast to Medusa, and we plan to perform experimental evaluations on more graph algorithms and datasets in future work.

5.3 The Preprocessing Time Comparison

We explore the overhead of the preprocessing phase in this section. In our system, edges are responsible for sending messages to the message buffer, from which messages are consumed by corresponding vertices. Therefore, precise positions for edges and vertices to manipulate the messages need to be determined in advance. Medusa proposed a graph-aware buffer scheme to establish the positions. However, it is implemented on the CPU and can not take full advantage of the GPU's computing power. In comparison, our system conducts the preprocessing on the GPU. For further optimization, we offload the data layout remapping operation in the preprocessing phase to the GPU. In this way, we can avoid the unnecessary conversion from row-major to column-major layout for each access to the global memory and calculate the positions only once.

As shown in Table 4, we can observe that our system significantly outperforms Medusa in the preprocessing stage even the overhead of layout remapping is considered, because of computation offloading to the GPU including the data remapping procedure. In this test, we set the parameter of *Per_Group_Num* to 32. The achieved speedup impressively ranges from 10x to 50x. One prominent feature is that the overhead of remapping remains stable over all workloads irrespective of the complexity of the graphs. For example, the overhead of ego-Gplus is close to that of soc-Pokec, while the latter has 16x more nodes and 2.2x more edges than the former. In contrast, the overhead in Medusa shows obvious causality with the scale of graphs, and it

1045-9219 (c) 2016 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2016.2611659, IEEE Transactions on Parallel and Distributed Systems

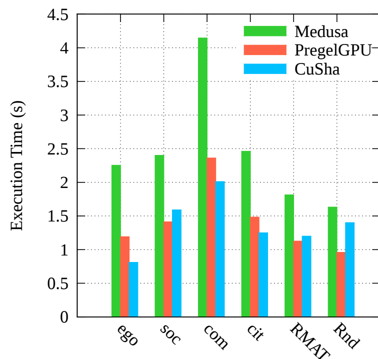


Fig. 7: Performance comparison of PageRank

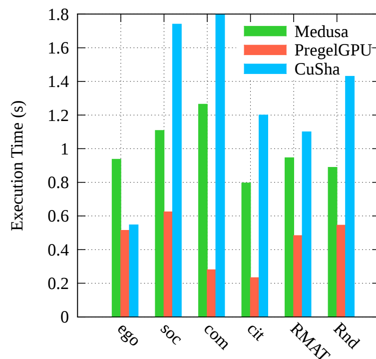


Fig. 8: Performance comparison of SSSP

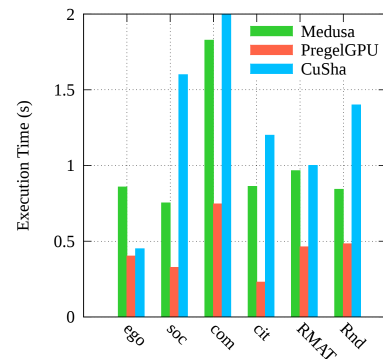


Fig. 9: Performance comparison of HCC

TABLE 4: The overhead of preprocessing (seconds)

workload	Our system	Medusa	speedup
ego-Gplus	0.016	0.21	13
soc-Pokec	0.020	0.449	22
com-LiveJournal	0.030	1.002	33
cit-Patents	0.012	0.665	55
RMAT	0.012	0.305	25
Random	0.011	0.328	30

raises with the increasing number of nodes and edges in the graphs.

5.4 Comparison with CuSha

Our original goal is to show the effectiveness of the proposed optimizations that are orthogonal to the design of existing system, so we base our prototype system on Medusa. To further demonstrate the superiority, we compare with CuSha [7] in this section. As shown in Figure 7, 8, and 9, our system outperforms CuSha in most cases. For SSSP and HCC, CuSha even underperforms the original Medusa due to the large overhead of its shard pre-processing. For example, for the SSSP algorithm, the execution time of CuSha on Random and ego-Gplus is 1423ms (1366ms + 57ms) and 547 ms (439ms + 108ms) respectively. Our system takes 520ms (11ms + 509ms) and 496ms (16ms + 480ms) for the same dataset. The preprocessing time of CuSha (collecting graph data into shard form) is much larger than ours (1366 vs 11, and 439 vs 16), but CuSha outperform our system in other parts of computation (57 vs 509, and 108 vs 480). On the whole, our system outperforms CuSha by 2.7x and 1.1x respectively on the two dataset. However, CuSha shows slightly better performance with PageRank under the ego, com, and cit dataset.

5.5 Approximate Sorting

In this part, we evaluate the effectiveness of approximate sorting by answering two questions: To what extent can we improve the performance by sorting? How fast is the approximate sorting as compared to existing GPU-based sorting algorithms? The answer to the first question can be obtained from Figure 10, which shows non-trivial improvement due to the alleviation of imbalance workload distribution. In this experiment, we run PageRank for 50 iterations. The performance can be improved by 8% $\sim$ 20% when the sorting is enabled. In particular, the improvement for power law graphs is more significant than the random graph (Random) with uniform degree distribution.

However, more attentions should be paid on the accuracy and associated overhead in order to achieve satisfactory gains in performance. Figure 11 presents the answers for the second question. We compare approximate sorting with three commonly-used and high-performance GPU-based sorting algorithms (FPGquick sorting [10], merge sorting, and radix sorting), and a uniform size distribution for all the experiments is assumed. We prepare three datasets (M means 10^6) and set BUCKET_NUMBER to 1000 in the approximate sorting. Both quick sort and merge sort are comparison based divide-and-conquer algorithms, and have average $O(nlogn)$ time complexity. Radix sort is a non-comparative integer sorting algorithm by sorting individual digits. Its time complexity is $O(kn)$ where k is the highest number of digits among the input key set. k determines the number of iterations needed to accomplish sorting. Approximate sort's complexity is $O(n)$ with only one iteration to map keys to buckets, which is different from others that require recursive or iterative executions. Having presented the complexity analysis, we compare the performance of the four algorithms next.

As shown in the Figure 11, our approach outperforms the competitors significantly. The approximate sorting consumes less than 1 millisecond, reflecting a 21.7x speedup in the best case compared to merge sorting and a 6.1x speedup in the worse case in contrast to radix sorting. It is worth noting that the overhead of the approximate sorting increases slowly with the growing scale of the datasets, while the costs of the other three sorting algorithms climbs up rapidly as the problem scale increases especially for the FPGquick sorting and merge sorting.

We use a metric *warp execution efficiency* of CUDA profiler [3] to demonstrate the microscopic impact of the approximating sorting. The *warp execution efficiency* presents the ratio of the average active threads per warp to the maximum number of threads per warp supported on a multiprocessor. With this metric, we can observe how sorting increases the number of active threads in a warp, be-

1045-9219 (c) 2016 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2016.2611659, IEEE Transactions on Parallel and Distributed Systems

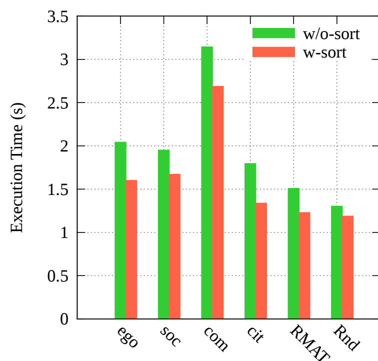


Fig. 10: with sorting vs. without sorting

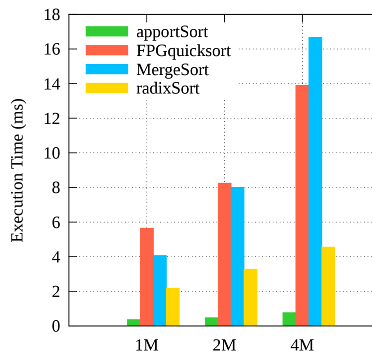


Fig. 11: Comparison with traditional sorting algorithms

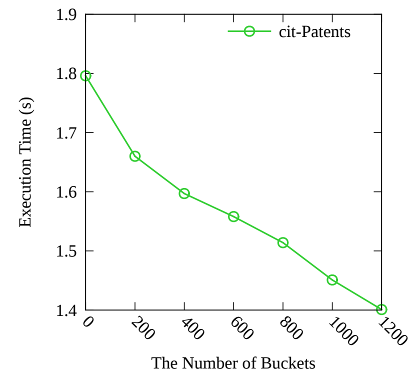


Fig. 12: Performance impact of the number of buckets.

TABLE 5: Comparison of warp execution efficiency (WEE) for the PageRank algorithm.

	ego	soc	com	cit	RMAT	Rnd
WEE/with sorting	9.1%/89.8%	21.8%/95.6%	29.4%/92.1%	37.9%/94.2%	35.7%/95.6%	68%/91.9%

TABLE 6: The improvement of overall performance of the approximate sort over radix sort.

	ego	soc	com	cit	RMAT	Rnd
apprx vs. radix	5.1%	11.4%	21.7%	25.1%	12.4%	12.9%

cause irregular load distribution would severely constrain the number of concurrently running threads. As shown in Table 5, with approximate sorting, warp execution efficiency is greatly improved across all workloads. The improvement is especially prominent for the 'ego' dataset (about 10x) that has the most irregular degree distribution.

Table 6 shows the overall performance improvement of the approximate sort over the radix sort that is the fastest traditional sort algorithm in our evaluation. Because the sorting is performed on the vertex array, the improvement ratio for 'ego' (with minimum nodes in our dataset) is marginal (5.1%), and the graph 'cit' (with the second largest number of nodes) achieves the best ratio (25.1%). Besides the vertex array, the sorting procedure also needs additional logic to adjust the edge array. Thus, the improvement of the graph 'com' (with both the largest number of nodes and edges) is less than that of 'cit'.

Next, we investigate the impact of the value of BUCKET_NUMBER on the overall performance. In Figure 12, we use the graph cit-Patents and the PageRank algorithm to evaluate the performance under different values of BUCKET_NUMBER (excluding the run time of sorting itself). BUCKET_NUMBER=0 implies that no sorting is conducted. And the increasing of the number of buckets directly translate into higher performance (see the linear decrease of the execution time in the figure).

As discussed in Section 4.1, graphs with abnormal degree distribution (e.g., a majority of nodes have similar degrees) can be handled by comparing the maximum and minimum degree. The time consumption of obtaining the maximum and minimum degree is less than 0.15 milliseconds that is already included in the preprocessing time, so the overhead of dynamic checking the degree distribution is negligible. In our current implementation, if the difference between the maximum and minimum degree is 0, we set BUCKET_NUMBER to 0 to disable the sorting. The experimental results from the Random graph (1 million nodes with min degree 1 and max degree 41) also demonstrate the effectiveness of approximating sorting on such graphs with uniform degree distribution.

5.6 The Performance of Data Layout Remapping

For GPU devices of compute capability 2.x or 3.x, concurrent accesses to the global memory by the threads in a warp are coalesced into a number of transactions that is equal to the number of cache lines necessary to service all of the threads in the warp [5]. We study the effect of the data layout remapping in this section. As shown in Figure 13, we also use the PageRank algorithm (50 iterations) to perform the comparison between two cases, one with data layout remapping and the other without. Per_Group_Num is set to 32 that is the warp size.

In Figure 13, we can see that the performance improvement varies between 10% and 25% by enabling the data layout remapping. Another observation is that the performance improvement is related to the value of η for power-law graphs. Workloads with smaller η exhibit narrower difference between the two different layouts than those with larger η, because of the distinguishable in-degree distributions. For example, for the workload ego with the largest value of η, the improvement reaches 25%.

In order to show how layout remapping improves coalesced memory access from microscopic perspective, we present the results about global store/load efficiency collected using the CUDA visual profiler [3] in Table 7. As suggested by the profiler, we should minimize the metrics *gld_efficiency* and *gst_efficiency* wherever possible in order to mitigate un-coalesced memory accesses. As shown in Table 7, the store efficiency increases slightly when the sort-

1045-9219 (c) 2016 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2016.2611659, IEEE Transactions on Parallel and Distributed Systems

TABLE 7: Comparison of global store/load efficiency for the PageRank algorithm.

global efficiency	ego	soc	com	cit	RMAT	Rnd
store/with remapping	64.2%/64.6%	51.3%/55.6%	49.4%/51.1%	57.7%/60.2%	50.5%/55.4%	57.1%/61.9%
load/with remapping	12.7%/91.4%	18.2%/93.1%	19.5%/94.1%	22.1%/92.2%	13.8%/92.1%	14.5%/90.1%

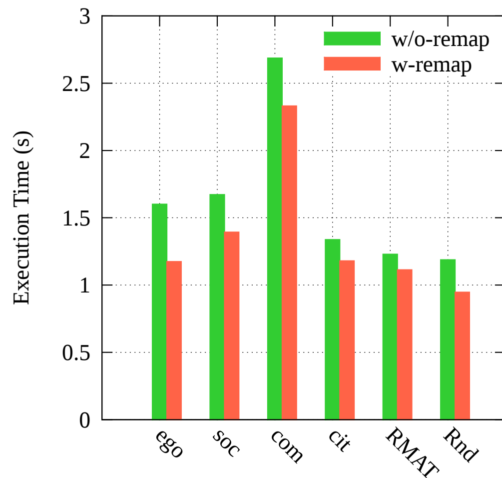


Fig. 13: Performance impact of layout remapping

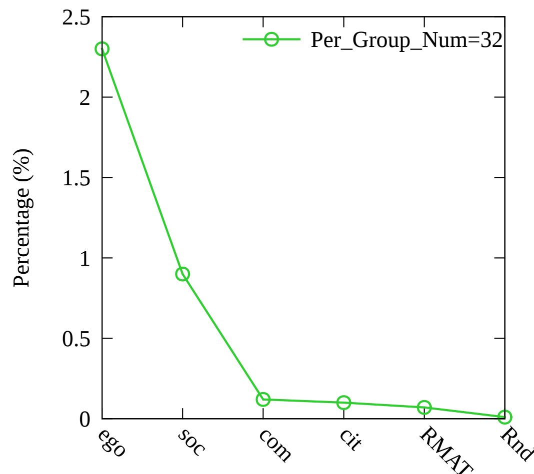


Fig. 14: Extra memory consumption

ing is enabled, while the load efficiency exhibits remarkable improvement with the approximate sorting.

Next, we analyze the extra memory consumption incurred by the data layout remapping. Overall, the extra memory needed to implement the data remapping is restricted across all the workloads. It is directly proportional to the value of *Max(d)* in power-law graphs. As shown in Figure 14, for the workload ego with $Max(d) = 17058$, 2.3% more memory space consumption is needed, while for the graph soc ($Max(d) = 6808$) the ratio decreases to 0.89%. For the graph Rnd with uniform degree distribution, the memory overhead is even much lower than others.

6 Related Work

Distributed graph processing systems. Driven by the rapid development of social networks, graph computing has been a hot issue in both academia and industry in recent years. HaLoop [9] and Pegasus [14] are graph processing systems based on Mapreduce. But due to the inherent characteristics of graph processing, it is not suitable to directly leverage the MapReduce programming paradigm to perform graph computations. To support efficient graph computation, vertex-centric programming model was proposed in Pregel and its successive variances. Compared to Pregel, GPS [28] and Mizan [19] enhances the original Pregel model by introducing new features such as adjusting the graph dynamically across nodes during the computation. Kineograph [11] performs graph computation on changing graph structure, including an incremental graph computation engine, which can handle continuous updates and produce timely results.

Multicore graph processing systems. Grace [27] implements a vertex-centric model, containing a series of graph and multicore specific optimizations that includes graph partitioning, in-memory vertex ordering, updates batching, and load-balancing. GraphChi [20] is a disk based graph processing system and designed to run on a single machine with limited memory by breaking large graphs into small parts. In GraphChi, a single PC can perform graph computations on very large graphs with performance comparable to large-scale distributed systems. TurboGraph [16] is another disk-based graph engine to process billion-scale graphs on a single PC, by fully exploiting multicore parallelism and SSD IO parallelism. GPSA [32] is a single-machine graph processing system based on the actor programming model, which can exploit the capabilities of multi-core systems as much as possible. GPSA decouples computation from message dispatching, which makes it possible to overlap the execution of the two processing procedures.

GPU based graph processing frameworks. Medusa [37] is an efficient implementation of the Pregel model for GPUs. Medusa provides a more fine-grained programming interface, exposing data parallelism on edges, vertices, and messages called EMV model. Even with the fine-grained interface, Medusa introduces load imbalance and non-coalesced memory access among threads on the GPU, which leads to underutilization of GPU computing resources. Furthermore, the EMV model is still complicated with too many details exposed to developers compared to the original vertex-centric model. In this paper, we present an Edge-Vertex model and two optimizations to address these issues. CuSha [7] is a graph processing framework, proposing new graph representations such as G-Shards and Concatenated Windows to minimize non-coalesced memory accesses and achieve higher GPU utilization for processing sparse graphs. However, the new graph representation in CuSha incurs larger memory space overhead than the conventional CSR representation. TOTEM [15] is a graph processing engine for

1045-9219 (c) 2016 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

heterogeneous many-core systems, which reduces development complexity and applies algorithm-agnostic optimizations to improve performance.

Gunrock [35] is a high-performance graph processing library targeting the GPU. Gunrock implements a data-centric abstraction, and strikes a balance between performance and expressiveness by coupling GPU computing primitives and optimization strategies with a high-level programming model. GraphReduce [29] is a scalable GPU-based framework that operates on graphs that exceed the GPU's memory capacity. GraphReduce adopts a combination of edge- and vertex-centric implementations and uses multiple GPU streams to exploit the high degree of parallelism of GPUs. Enterprise [23] is a new GPU-based BFS system that combines three techniques to remove potential performance bottlenecks: streamlined GPU threads scheduling, GPU workload balancing, and GPU based BFS direction optimization. The GPU workload balancing that classifies nodes based on different out-degrees is similar to our sorting-based load-balancing strategy, but using a different approach. Frog [30] is a light-weight asynchronous processing framework with a hybrid coloring model. It includes three parts: a hybrid graph coloring algorithm, an asynchronous execution model, and a streaming execution engine on GPUs. GraphBIG [26] is a comprehensive benchmark suites for graph computing, which supports a wide selection of workloads for both CPU and GPU, and covers a broad scope of graph computing applications.

Existing works mainly focus on a general and high-performance GPU-based framework to ease the development of graph processing algorithms. We instead target the optimizations of graph processing on GPUs based on the observations from the designs of GPU-based graph systems. Although we only perform comparison with Medusa (our work is based on Medusa), the lightweight optimization strategies proposed in this paper are orthogonal to the optimizations in other GPU graph processing systems, and may also be applicable to those systems.

7 Conclusions

In this paper, we propose a general graph computing framework for GPUs that achieves the goals of easy-to-use and good performance with a simplified programming model. Specifically, we develop a Edge-Vertex model in order to better utilize the fine-grained parallelism of GPUs; and we identify that load imbalance can be alleviated with the lightweight approximate sorting and that non-coalesced memory access can be mitigated by the data layout remapping. In addition, the integration of data remapping into the preprocessing of graph data can significantly improve the overall performance. As demonstrated by experimental evaluation, our system can achieve up to 4.5x performance speedup compared to the state-of-the-art across a wide range of workloads. As for future work, we are planning to extend our system to support multi-GPUs and distributed environments.

Acknowledgment

This research was supported in part by the National Science Foundation of China under grants 61272190, 61572179 and 61173166. Jianhua Sun is the corresponding author.

References

[1] "Apache giraph," http://hama.apache.org.
[2] "Apache hama," http://hama.apache.org.
[3] "Cuda profiler," http://docs.nvidia.com/cuda/profiler-users-guide/.
[4] "Cudpp: Cuda data parallel primitives library," http://cudpp.github.io/.
[5] "Nvidia cuda," http://www.nvidia.com/object/cuda.
[6] D. A. Bader and K. Madduri, "Gtgraph: A synthetic graph generator suite," *For the 9th DIMACS Implementation Challenge*, 2006.
[7] F. K. K. V. R. G. 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 2014)*. ACM, 2014, pp. 239–252.
[8] S. Brin and L. Page, "The anatomy of a large-scale hypertextual web search engine," *Computer networks and ISDN systems*, vol. 30, no. 1, pp. 107–117, 1998.
[9] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst, "Haloop: Efficient iterative data processing on large clusters," *Proceedings of the VLDB Endowment*, vol. 3, no. 1-2, pp. 285–296, 2010.
[10] D. Cederman and P. Tsigas, "Gpu-quicksort: A practical quicksort algorithm for graphics processors," *Journal of Experimental Algorithmics (JEA)*, vol. 14, p. 4, 2009.
[11] R. Cheng, J. Hong, A. Kyrola, Y. Miao, X. Weng, M. Wu, F. Yang, L. Zhou, F. Zhao, and E. Chen, "Kineograph: Taking the pulse of a fast-changing and connected world," in *Proceedings of the 7th ACM european conference on Computer Systems*. ACM, 2012, pp. 85–98.
[12] B. V. Cherkassky, A. V. Goldberg, and T. Radzik, "Shortest paths algorithms: Theory and experimental evaluation," *Mathematical programming*, vol. 73, no. 2, pp. 129–174, 1996.
[13] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.
[14] E. Deelman, G. Singh, M.-H. Su, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, G. B. Berriman, J. Good *et al.*, "Pegasus: A framework for mapping complex scientific workflows onto distributed systems," *Scientific Programming*, vol. 13, no. 3, pp. 219–237, 2005.
[15] A. Gharaibeh, L. B. Costa, E. Santos-Neto, and M. Ripeanu, "A yoke of oxen and a thousand chickens for heavy lifting graph processing," in *Proceedings of the the 21st International Conference on Parallel Architectures and Compilation Techniques*. ACM, 2012, pp. 345–354.
[16] W.-S. Han, S. Lee, K. Park, J.-H. Lee, M.-S. Kim, J. Kim, and 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*. ACM, 2013, pp. 77–85.
[17] P. Harish and P. Narayanan, "Accelerating large graph algorithms on the gpu using cuda," in *High performance computing–HiPC 2007*. Springer, 2007, pp. 197–208.
[18] S. Hong, S. K. Kim, T. Oguntebi, and K. Olukotun, "Accelerating cuda graph algorithms at maximum warp," *ACM SIGPLAN Notices*, vol. 46, no. 8, pp. 267–276, 2011.
[19] Z. Khayyat, K. Awara, A. Alonazi, H. Jamjoom, D. Williams, and P. Kalnis, "Mizan: A system for dynamic load balancing in large-scale graph processing," in *Proceedings of the 8th ACM European Conference on Computer Systems*. ACM, 2013, pp. 169–182.
[20] A. Kyrola, G. E. Blelloch, and C. Guestrin, "Graphchi: Large-scale graph computation on just a pc." in *OSDI*, vol. 12, 2012, pp. 31–46.
[21] J. Leskovec, J. Kleinberg, and C. Faloutsos, "Graphs over time: Densification laws, shrinking diameters and possible explanations," in *Proceedings of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining*. ACM, 2005, pp. 177–187.
[22] J. Leskovec and J. J. Mcauley, "Learning to discover social circles in ego networks," in *Advances in neural information processing systems*, 2012, pp. 539–547.
[23] H. Liu and H. H. Huang, "Enterprise: Breadth-first graph traversal on gpus," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC 2015)*. ACM, 2015, pp. 68:1–68:12.

1045-9219 (c) 2016 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

[24] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: A system for large-scale graph processing," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. ACM, 2010, pp. 135–146.

[25] A. Munshi, "Opencl: Parallel computing on the gpu and cpu," *SIGGRAPH, Tutorial*, 2008.

[26] L. Nai, Y. Xia, I. G. Tanase, H. Kim, and C.-Y. Lin, "Graphbig: Understanding graph computing in the context of industrial solutions," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '15. New York, NY, USA: ACM, 2015, pp. 69:1–69:12.

[27] V. Prabhakaran, M. Wu, X. Weng, F. McSherry, L. Zhou, and M. Haradasan, "Managing large graphs on multi-cores with graph awareness." in *USENIX Annual Technical Conference*, 2012, pp. 41–52.

[28] S. Salihoglu and J. Widom, "Gps: A graph processing system," in *Proceedings of the 25th International Conference on Scientific and Statistical Database Management*. ACM, 2013, pp. 22:1–22:12.

[29] D. Sengupta, S. Song, K. Agarwal, and 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 2015)*. ACM, 2015, pp. 28:1–28:12.

[30] X. S. Shi, J. Liang, S. Di, B. He, H. Jin, L. Lu, Z. Wang, X. Luo, and J. Zhong, "Optimization of asynchronous graph processing on gpu with hybrid coloring model," in *Proceedings of the the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 2015)*. ACM, 2015.

[31] R. Smith, N. Goyal, J. Ormont, K. Sankaralingam, and C. Estan, "Evaluating gpus for network packet signature matching," in *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*. IEEE, 2009, pp. 175–184.

[32] J. Sun, D. Zhou, H. Chen, C. Chang, Z. Chen, w. Li, and L. He, "Gpsa: A graph processing system with actors," in *Proceedings of the 44th International Conference on Parallel Processing (ICPP 2015)*. ACM, 2015, pp. 709–718.

[33] L. Takac and M. Zabovsky, "Data analysis in public social networks," in *International Scientific Conference AND International Workshop Present Day Trends of Innovations*, 2012.

[34] L. G. Valiant, "A bridging model for parallel computation," *Communications of the ACM*, vol. 33, no. 8, pp. 103–111, 1990.

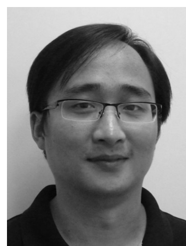
[35] Y. Wang, A. Davidson, Y. Pan, Y. Wu, A. Riffel, and J. D. Owens, "Gunrock: A high-performance graph processing library on the gpu," in *Proceedings of the 21th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 2016)*. ACM, 2016, pp. 11:1–11:12.

[36] J. Yang and J. Leskovec, "Defining and evaluating network communities based on ground-truth," in *Proceedings of the ACM SIGKDD Workshop on Mining Data Semantics*. ACM, 2012, pp. 3:1–3:8.

[37] J. Zhong and B. He, "Medusa: Simplified graph processing on gpus," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 6, pp. 1543–1552, 2013.

Wenyong Zhong is an Ph.D student at the College of Computer Science and Electronic Engineering, Hunan University, China. His research interests are in heterogeneous computing systems and graph computing.

Jianhua Sun is an Associate Professor at the College of Computer Science and Electronic Engineering, Hunan University, China. She received the Ph.D. degree in Computer Science from Huazhong University of Science and Technology, China in 2005. Her research interests are in security and operating systems. She has published more than 70 papers in journals and conferences, such as IEEE Transactions on Parallel and Distributed Systems, IEEE Transactions on Computers.



Hao Chen received the BS degree in chemical engineering from Sichuan University, China, in 1998, and the PhD degree in computer science from Huazhong University of Science and Technology, China in 2005. He is now a Professor at the College of Computer Science and Electronic Engineering, Hunan University, China. His current research interests include parallel and distributed computing, operating systems, cloud computing and systems security. He has published more than 70 papers in journals and conferences, such as IEEE Transactions on Parallel and Distributed Systems, IEEE Transactions on Computers, IPDPS, IWQoS, and ICPP. He is a member of the IEEE and the ACM.

Jun Xiao was a Master student at the College of Computer Science and Electronic Engineering, Hunan University, China. Her research interests are in GPGPU computing and graph computing systems.

Zhiwen Chen is an Ph.D student at the College of Computer Science and Electronic Engineering, Hunan University, China. His research interests are in parallel computing and multi-core systems.

Cheng Chang is working toward the PH.D. degree at the School of Computer Science and Electronic Engineering, Hunan University, China. His research interests include distributed storage system and virtualization. He is a student member of the IEEE.

Xuanhua Shi Xuanhua Shi is a professor in Big Data Technology and System Lab/ Service Computing Technology and System Lab, Huazhong University of Science and Technology (China). He received his Ph.D. degree in Computer Engineering from Huazhong University of Science and Technology (China) in 2005. From 2006, he worked as an INRIA Post-Doc in PARIS team at Rennes for one year. His current research interests focus on the cloud computing and big data processing. He published over 70 peer-reviewed publications, received research support from a variety of governmental and industrial organizations, such as National Science Foundation of China, Ministry of Science and Technology, Ministry of Education, European Union and so on. He has chaired several conferences and workshops, and served on technical program committees of numerous international conferences. He is amember of the IEEE and ACM, a senior member of the CCF.

1045-9219 (c) 2016 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.