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-
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
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 latency of global memory access and to better utilize the computational resources on the GPU.

Consideration in programming for GPU architectures is the hardware architectures.


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 it. This is achieved by maintaining a map that associates 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 it.

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](image)

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 it.
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.

<table>
<thead>
<tr>
<th>TABLE 1: Pregel-style APIs</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>user-implemented APIs</strong></td>
</tr>
<tr>
<td>/* EdgeCompute function, processing one edge */ void EdgeCompute(Edge e);</td>
</tr>
<tr>
<td>/* VertexCompute function, processing one vertex */ void VertexCompute(Vertex v);</td>
</tr>
<tr>
<td>/* Initialize the value of edges and vertices, configure the number of threads and blocks on GPUs. <em>/ void initData(void</em> graph, void* initialValues);</td>
</tr>
<tr>
<td><strong>system-provided APIs</strong></td>
</tr>
<tr>
<td>/* Start the runtime system, and stop until reaching max- Supersteps specified by the programmer. */ void startGPU(int maxSupersteps);</td>
</tr>
<tr>
<td>/* Start the runtime system, and stop until no active vertices exist. */ void startGPU();</td>
</tr>
<tr>
<td>/* Set the state of the vertex to inactive. */ void voteToHalt(Vertex v);</td>
</tr>
<tr>
<td>/* Set the state of the vertex to active. */ void voteToActive(Vertex v);</td>
</tr>
<tr>
<td>/* Set the parameter continue to true, if there are still active vertices sending messages in the next superstep. */ void stillContinueNextSupersteps(bool continue);</td>
</tr>
<tr>
<td>/* Combine the incoming messages sent to the same vertex. */ Message Combiner(Vertex v);</td>
</tr>
</tbody>
</table>

and the other can determine the completion of computation by checking whether there will be active vertex in the next superstep.

### 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
In this section, we present two novel optimizations to im-
optimize memory layout in section 4.2. Another potential performance bottleneck comes from
ineffective utilization of compute resource. In our case,
anced workload distribution among GPU threads, resulting
Inherent irregularity in some applications may cause imbal-
the GPU.

earlier, we assume the CSR representation for the graph on
GPU we are often recommended to use a column oriented
favors a row-major memory layout and transfered to the
because the message buffer is initialized on the CPU that
sages from the message buffer in a non-coalesced way,

Combiner(v)

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
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, which also has been observed in our implementation. 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 i (corresponding to inclusive and exclusive prefix sum respectively). Because 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 faster 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...
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

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 lightweight 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 $\text{Per\_Group\_Num}$ that is set to 3 in our example. As a consequence, we can obtain the number of groups using the formula $\left\lceil \frac{\text{length(size)}}{\text{Per\_Group\_Num}} \right\rceil$ 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

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...
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.

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
TABLE 2: Characteristics of workloads used in the experiments

<table>
<thead>
<tr>
<th>Workload</th>
<th>In short</th>
<th>Nodes</th>
<th>Edges</th>
<th>$\text{Max}(d)$</th>
<th>$\eta$</th>
</tr>
</thead>
<tbody>
<tr>
<td>ego-Gplus</td>
<td>ego</td>
<td>107,614</td>
<td>13,673,453</td>
<td>17058</td>
<td>99.9%</td>
</tr>
<tr>
<td>soc-Pokec</td>
<td>soc</td>
<td>1,632,803</td>
<td>30,622,564</td>
<td>6808</td>
<td>76.3%</td>
</tr>
<tr>
<td>com-LiveJournal</td>
<td>com</td>
<td>3,997,962</td>
<td>34,681,189</td>
<td>2454</td>
<td>76.7%</td>
</tr>
<tr>
<td>cit-Patents</td>
<td>cit</td>
<td>3,774,768</td>
<td>16,518,948</td>
<td>779</td>
<td>61.9%</td>
</tr>
<tr>
<td>RMAT</td>
<td>RMAT</td>
<td>1,000,000</td>
<td>16,000,000</td>
<td>555</td>
<td>52.4%</td>
</tr>
<tr>
<td>Random</td>
<td>Rnd</td>
<td>1,000,000</td>
<td>16,000,000</td>
<td>41</td>
<td>99.9%</td>
</tr>
</tbody>
</table>

TABLE 3: the feature of machines used in the experiments

<table>
<thead>
<tr>
<th>Feature</th>
<th>Intel Xeon ES 2648L</th>
<th>GeForce GTX 780</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cores/Proc</td>
<td>8</td>
<td>12</td>
</tr>
<tr>
<td>Hardware Threads/core</td>
<td>2</td>
<td>192</td>
</tr>
<tr>
<td>Frequency/Core(GHz)</td>
<td>1.8</td>
<td>0.90</td>
</tr>
<tr>
<td>Main Memory/proc(GB)</td>
<td>8 DDR3</td>
<td>3 DDR5</td>
</tr>
<tr>
<td>Memory Bandwidth/Proc(GB/s)</td>
<td>57.6</td>
<td>288.4</td>
</tr>
</tbody>
</table>

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 $\text{Max}(d)$ 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 $\eta$. Ego-Gplus [22] consists of circles from Google+ and the $\eta$ 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 $\eta$ of the two graphs is approximately the same (76%). Cit-Patents [21] is maintained by the National Bureau of Economic Research and its $\eta$ 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 $\eta$ is 52.4%. Random is a uniformly distributed graph workload and $\text{Max}(d)$ 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 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 $\text{Per}_\text{Group}_\text{Num}$ 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
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 547ms (439ms + 108ms) respectively. Our system takes 520ms (511ms + 9ms) 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 datasets. 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% ~ 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 (FPQquick 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 \( O(n\log n) \) 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 sorting algorithms climbs up rapidly as the problem scale increases especially for the FPQquick 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-
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.

<table>
<thead>
<tr>
<th>Data Set</th>
<th>Approx Sort</th>
<th>Radix Sort</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ego</td>
<td>5.1%</td>
<td>12.4%</td>
</tr>
<tr>
<td>Soc</td>
<td>11.4%</td>
<td>12.9%</td>
</tr>
<tr>
<td>Com</td>
<td>21.7%</td>
<td>25.1%</td>
</tr>
<tr>
<td>Cit</td>
<td>27.5%</td>
<td>32.1%</td>
</tr>
<tr>
<td>Rnd</td>
<td>12.4%</td>
<td>13.9%</td>
</tr>
</tbody>
</table>

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 \( \eta \) for power-law graphs. Workloads with smaller \( \eta \) exhibit narrower difference between the two different layouts than those with larger \( \eta \), because of the distinguishable in-degree distributions. For example, for the workload ego with the largest value of \( \eta \), 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-
TABLE 7: Comparison of global store/load efficiency for the PageRank algorithm.

<table>
<thead>
<tr>
<th></th>
<th>ego</th>
<th>soc</th>
<th>com</th>
<th>cit</th>
<th>RMAT</th>
<th>Rnd</th>
</tr>
</thead>
<tbody>
<tr>
<td>store/wi</td>
<td>64.2%</td>
<td>51.3%</td>
<td>49.4%</td>
<td>57.7%</td>
<td>50.5%</td>
<td>57.1%</td>
</tr>
<tr>
<td>th remapping</td>
<td>66.6%</td>
<td>55.6%</td>
<td>51.1%</td>
<td>60.2%</td>
<td>55.4%</td>
<td>61.9%</td>
</tr>
<tr>
<td>load/wi</td>
<td>12.7%</td>
<td>18.2%</td>
<td>19.5%</td>
<td>22.1%</td>
<td>13.8%</td>
<td>14.5%</td>
</tr>
<tr>
<td>th remapping</td>
<td>14.1%</td>
<td>93.1%</td>
<td>94.1%</td>
<td>92.2%</td>
<td>92.1%</td>
<td>90.1%</td>
</tr>
</tbody>
</table>

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 multicores 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
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 suite 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


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.

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 a member of the IEEE and ACM, a senior member of the CCF.