GTS: A Fast and Scalable Graph Processing Method based on Streaming Topology to GPUs

Min-Soo Kim, Kyuhyeon An, Himchan Park, Hyunseok Seo, and Jinwook Kim
Department of Information and Communication Engineering
DGIST
Republic of Korea
{mskim,khan,chan150,hsseo,bm010515}@dgist.ac.kr

ABSTRACT

A fast and scalable graph processing method becomes increasingly important as graphs become popular in a wide range of applications and their sizes are growing rapidly. Most of distributed graph processing methods require a lot of machines equipped with a total of thousands of CPU cores and a few terabyte main memory for handling billion-scale graphs. Meanwhile, GPUs could be a promising direction toward fast processing of large-scale graphs by exploiting thousands of GPU cores. All of the existing methods using GPUs, however, fail to process large-scale graphs that do not fit in main memory of a single machine. Here, we propose a fast and scalable graph processing method GTS that handles even RMAT32 (64 billion edges) very efficiently only by using a single machine. The proposed method stores graphs in PCI-E SSDs and executes a graph algorithm using thousands of GPU cores while streaming topology data of graphs to GPUs via PCI-E interface. GTS is fast due to no communication overhead and scalable due to no data duplication from graph partitioning among machines. Through extensive experiments, we show that GTS consistently and significantly outperforms the major distributed graph processing methods, GraphX, Giraph, and PowerGraph, and the state-of-the-art GPU-based method TOTEM.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming – parallel programming; E.1 [**Data Structures**]: Graphs and networks

Keywords

Graph processing, GPUs, SSDs, Stream

1. INTRODUCTION

Graphs are widely used to model real-world objects in many disciplines such as social networks, web, business intelligence, biology, and neuroscience, due to their generality of modeling. As the sizes of real graphs are growing rapidly, fast and scalable graph processing methods have become more important than ever before.

SIGMOD '16, June 26–July 1, 2016, San Francisco, CA, USA.

DOI: http://dx.doi.org/10.1145/2882903.2915204

In order to handle large-scale graphs efficiently, there have been proposed a number of distributed graph processing methods. Apache Giraph [1, 11] is an alternative implementation of Google's Pregel [22]. It follows the Bulk-Synchronous Parallel (BSP) message passing model where all vertex kernels run simultaneously in a sequence of supersteps. Within a superstep, each kernel receives all messages from the previous superstep and sends them to its neighbors in the next superstep. Apache Spark GraphX [10, 33, 35] is a graph-parallel framework built on top of Apache Spark. PowerGraph [9, 20, 21] is a graph processing framework considering the power-law distribution of real graphs. It follows the Gather-Apply-Scatter (GAS) model, where Gather collects the information about adjacent vertices/edges, Apply updates the new value of the central vertex using the information, and Scatter updates the data on adjacent edges using the new value. Although they follow their own architectures and models, they all require a lot of machines equipped with a total of thousands of CPU cores and a few terabyte main memory for handling billion-scale graphs.

Meanwhile, the continuous advancement of GPU technology makes the theoretical computing power of modern computers ever-increasing. Due to the much higher theoretical computing performance of GPUs than CPUs, it becomes more and more important to exploit GPUs in a wide range of problems requiring high performance computing such as graph processing. Exploiting GPUs could be a promising direction toward fast processing of large-scale graphs mainly due to their implementation of the characteristics of the Parallel Random Access Machine (PRAM) abstraction. The PRAM abstraction has been widely used to investigate theoretical performance of parallel graph algorithms. It assumes an infinite number of processors and uniform memory latency. GPUs implement these characteristics with a very large number of hardware threads and uniform memory latency. In addition, GPUs have massive memory bandwidth. Conventional CPUs are far from those characteristics and only can traverse or calculate on a few vertices at a time. On the contrary, GPUs can do on a much large number of vertices at a time, without a severe memory bottleneck to shared memory and without excessive concern about different memory latency. Since many real graphs have millions or billions vertices, GPUs are potentially well suited to fast processing of such graphs.

Nevertheless, there is a major challenge associated with exploiting GPUs for processing large-scale graphs. Many real graphs do not fit in the GPU device memory with this tendency becoming more marked as the sizes of graphs are growing [7, 8]. Lack of support for large-scale graphs beyond the capacity of device memory is pointed out as one of the most critical problems of the existing graph processing methods using GPUs [7, 8, 15]. There is almost no study on solving this problem yet, in spite of its importance. To the best of our knowledge, TOTEM [7, 8] is the only work to systemat-

ically process a graph that does not fit in the GPU device memory. To solve the problem, it partitions a graph into two parts, one part in main memory and the other part in GPU device memory. GPUs process the part in GPU device memory, while CPUs process the part in main memory. Though it can handle large-scale graphs, it has many fundamental drawbacks such as underutilization of the computational power of GPUs, lack of scalability in terms of the number of GPUs, and the difficulty of optimizing performance due to a lot of options. Furthermore, it still cannot process larger-scale graphs beyond the capacity of main memory.

We propose a fast and scalable GPU-based graph processing method called GTS that can process even RMAT32 (64 billion edges) graphs very efficiently. GTS fully exploits the computational power of GPUs by processing the entire graph only using GPUs. It does not rely on the graph partitioning scheme and not require a bunch of options for optimization. To overcome the limit of GPU memory capacity and moreover the limit of main memory capacity, we propose a concept of *storing only updatable attribute data and moving topology data*. Here, the attribute data means the information about vertices and edges that are required and updated during execution of vertex kernels. The proposed method stores graphs in PCI-E SSDs and executes a graph algorithm using thousands of GPU cores while streaming topology data of graphs to GPUs via PCI-E interface. More specifically, GTS first copies attribute data to GPU device memory, and then, processes a graph algorithm by applying a user-defined GPU kernel function on each piece of topology data being copied in a streaming fashion from main memory to device memory. In GPUs, asynchronous data transfer can be achieved by using the asynchronous GPU streams (e.g., CUDA Streams), which could hide memory access latency from GPUs to main memory and so utilize GPU's computing power more. For efficient streaming, GTS adopts the slotted page format [12] that divides a graph into fixed-size units. In terms of exploiting multiple GPUs and SSDs, we also propose two strategies, (1) the strategy for performance (shortly, Strategy-P) for high performance with a limit on scalability and (2) the strategy for scalability (shortly, Strategy-S) for high scalability with a limit on performance. GTS can achieve higher performance compared with the existing methods, due to no communication overhead among machines and due to exploiting massive parallelism of GPU cores. In addition, it could achieve higher scalability compared with the existing methods, due to no data duplication from graph partitioning among machines and due to storing graphs on secondary storage, i.e., SSDs. For example, it can easily store and process an RMAT33 graph (8 billion vertices and 128 billion edges) only using 1 TB PCI-E SSDs, theoretically. GTS is also fairly scalable in terms of the number of GPUs and SSDs, and so, shows a stable speedup when adding a GPU or an SSD to the machine. It is because GTS almost uniformly distributes the units of graph data to GPUs, which again perform graph processing almost independently of each other.

The main contributions of this paper are as follows:

- We propose a novel concept of *storing only updatable attribute data and moving topology data* that is counter-intuitive in terms of the conventional models (e.g., GAS) of storing topology data and moving attribute data.
- We propose a parallel graph processing method GTS on GPUs that can perform graph algorithms very efficiently for large-scale graphs (e.g., billions vertices) by fully exploiting the asynchronous GPU streams.
- We present two strategies that can improve the performance or the scalability further by exploiting multiple GPUs and multiple SSDs: Strategy-P and Strategy-S.
- Through extensive experiments, we demonstrate that GTS consistently and significantly outperforms the major distributed graph processing methods, GraphX, Giraph, and PowerGraph, and the state-of-the-art GPU-based method TOTEM, across wide range of benchmarks.
- Especially, we show that GTS can process an RMAT32 graph within a reasonable time in a single machine that the existing distributed methods fail to process by using 30 machines of a total of about 2 TB memory.

The rest of this paper is organized follows. Section 2 reviews the data format adopted by GTS. In Section 3, we propose the GTS method. In Section 4, we present two strategies for exploiting multiple GPUs and SSDs. In Section 5, we present the cost models of GTS, and in Section 6, we address several implementation issues. Section 7 presents the results of experimental evaluation, and Section 8 discusses related work. Finally, Section 9 summarizes and concludes this paper.

2. PRELIMINARIES

In this section, we explain the data formats proposed for storing graph data, especially with focusing on the format for storing a graph on secondary storage (e.g., SSD). Most of real graphs are known as to be sparse, and various kinds of in-memory formats for a sparse graph have been proposed. They include Diagonal (DIA) [13], ELLPACK (ELL) [2], Compressed Sparse Row (CSR), Compressed Sparse Column (CSC), and Coordinate list (COO) [3]. They might have a limit on the size of graphs to process since they usually require a very long contiguous edge array in main memory for large-scale graphs.

Besides, there is an external memory (i.e., out-of-core) graph format called the *slotted page format* [12]. It represents a graph in a set of fixed-size slotted pages. A slotted page consists of two parts, *records* and *slots*, where records grow forward from the start of the page, but slots grow backward from the end of the page. A slot consists of a vertex ID (*logical ID*), denoted as VID, and the start offset of the corresponding record, denoted as OFF. A record consists of the size of the adjacency list, denoted as ADJLIST_SZ, and the adjacency list itself, denoted as ADJLIST. Here, the adjacency list again consists of a list of *record ID*s of neighbor vertices. A record ID is a *physical ID* consisting of a pair of the page ID (of 2-byte), denoted as ADJ_PID, and the slot number (of 2-byte) where the corresponding vertex is located, denoted as ADJ_OFF. By using physical IDs, graph algorithms can access to the physical locations of neighbor vertices easily during traversal. This concept of physical ID is commonly used for performance in database area [14]. The vertex IDs and record IDs are consecutive and ordered within a page.

Figure 1 shows an example graph G and the slotted pages of G. In Figure 1(a), the vertices v_0, v_1, and v_2 have a relatively small number of neighbor vertices, while v_3 has a relatively large number of neighbor ones. Such skewness of the node degree distribution is common in real graphs. The low-degree vertices like $\{v_0, v_1, v_2\}$ can be stored in a single page SP_0 as in Figure 1(b), which is called a *Small Page* (SP). In contrast, a high-degree vertex like v_3 might not be stored in a fixed-size slotted page, but can be stored in multiple pages $\{\text{LP}_1, \text{LP}_2\}$ instead as in Figure 1(c), which are called *Large Pages* (LPs).

3. STREAMING GRAPH TOPOLOGY

In this section, we present the proposed method GTS. Section 3.1 explains the concept of streaming topology, and Section 3.2 describes the streaming scheme of GTS in detail. Section 3.3 presents the scheme to process various types of graph algorithms, and Section 3.4 shows the algorithm of the GTS framework.

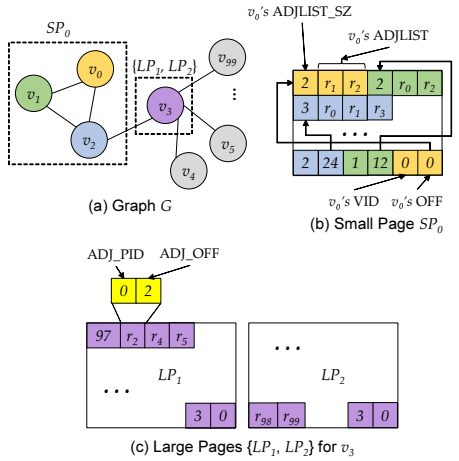


Figure 1: Example graph G and the slotted pages of G.

3.1 Concept

In general, graph algorithms require both graph topology data (shortly, topology data) and attribute data for vertices and/or edges. For example, in addition to topology data, PageRank requires two attribute vectors for vertices: a vector of the previous PageRank values (shortly, prevPR) and a vector of the next PageRank values (shortly, nextPR). As another example, BFS requires one attribute vector for keeping traversal levels for vertices (shortly, LV). The attribute vectors again can be divided into read-only ones and read/write ones. For example, for PageRank, prevPR is a read-only attribute vector, while nextPR is a read/write attribute vector, in a specific iteration. For BFS, LV is a read/write attribute vector.

Most of the existing graph processing systems [9, 20, 22] follows the concept that topology data is stored on main memory in a cluster of machines, and attribute data is moved among machines (e.g., the Gather-Apply-Scatter (GAS) model). Without loss of generality, the amount of attribute data to be moved is smaller than that of topology data, and so, it is beneficial to follow that concept for a distributed shared-memory system where each local memory can be accessed via a relatively slow interconnection network. However, it might not be true for a machine equipped with GPUs that have limited device memory and are connected with main memory via a much faster interconnection, i.e., PCI-E interface.

Here, we propose a concept of *storing only updatable attribute data and moving topology data*, where a relatively small amount of attribute data is stored in GPU's limited device memory, and topology data is moved via a high-speed interconnection. Moving topology data from main memory or SSDs to GPUs takes some amount of time even using PCI-E interconnection. However, we can hide that time by processing a given graph algorithm simultaneously with moving attribute data. A graph algorithm θ is executed through a corresponding user-defined GPU *kernel* function, denoted as K_θ. In our method, K_θ is executed using thousands of GPU cores for both a part of topology data and attribute data within GPU device memory.

Intuitively, GTS first copies read/write attribute data to device memory, and then, processes graph algorithms by copying read-only attribute data and topology data in a streaming fashion to device memory. GTS considers that each attribute vector is conceptually partitioned into multiple subvectors, and topology data consists of multiple fixed-size units, especially, in the slotted page format described in Section 2. For example, prevPR for PageRank can be conceptually decomposed into a lot of subvectors, where a subvector corresponds to a slotted page in terms of the range of vertex IDs. That is possible since the vertex IDs are consecutive in each slotted page. Let attribute data be X and topology data be $Y = \{y_1, \cdots, y_n\}$. Then, the amount of device memory required for processing the graph algorithm θ completely becomes $|X| + |y_i|$. It is obviously important to reduce $|X|$ or $|y_i|$ in order to process a large-scale graph with the limited size of device memory. In general, since $|X| >> |y_i|$, we focus on reducing $|X|$. Between read-only attribute vector (shortly, RA) and read/write attribute vector (shortly, WA), GTS keeps only WA in device memory for reducing the amount of device memory used. Since WA is frequently and randomly updated during graph algorithm in general, it is important to keep WA in device memory for performance. However, RA is not updated during processing, and so can be fed into device memory together with the corresponding topology data. For example, for PageRank, we keep the entire nextPR in device memory, but feed each subvector of prevPR together with the corresponding topology page to device memory.

Figure 2 shows the data flow of GTS. We suppose WA is divided into W chunks ($W = 1$ in default), and RA is divided into R subvectors. We also suppose the numbers of small pages and large pages are S and L, respectively. The number of units of RA, i.e., R is usually equal to S since most of the topology pages are SP, which will be shown in Section 7.5. RA_j represents the subvector of RA corresponding to SP_j. For WA, GTS performs the following three steps: (1) copying WA to GPU device memory; (2) processing graph algorithms while copying the topology pages $\{SP_j\}$ (or $\{LP_j\}$) together with the read-only attribute vectors $\{RA_j\}$ to GPU device memory in a streaming fashion; and (3) copying WA, which has been updated during graph processing, back to main memory for (data) synchronization.

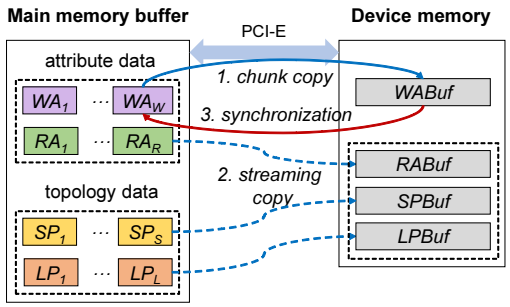


Figure 2: Basic data flow of GTS.

3.2 Asynchronous multiple streaming

GTS copies topology data from main memory to GPU device memory via the PCI-E bus asynchronously in a streaming way. For streaming topology data, GTS adopts the slotted page format, but it is not a necessary condition that GTS uses the slotted page format. GTS may adopt any format that can divide topology data into fixed-size units for streaming. GTS rather adopts the slotted page format since it considers workload balancing and coalesced memory access to some degree, which will be discussed in Section 6.2. For streaming RA, SP, and LP to device memory, GTS allocates three kinds of streaming buffers in device memory, called RABuf, SPBuf, and LPBuf, respectively, as in Figure 2. In addition, for WA, GTS also allocates a chunk buffer called WABuf.

GTS exploits multiple GPU streams for streaming. Figure 3 shows the timeline of copy operations of attribute and topology data to device memory. A CPU thread first transfers WA to WABuf. Then, it starts multiple GPU streams, each of which performs a series of operations, (1) copying SP_j (or LP_j) to SPBuf, (2) copying RA_j to RABuf, and (3) executing the kernel function, repeat-

edly. In general, transfer operations for WA, RA_j, and SP_j to device memory cannot overlap with each other, at least in the current GPU architecture [5]. Instead, they can overlap with kernel execution [5, 27]. Theoretically, the suitable number of streams k can be determined by using the ratio of the transfer time of SP_j and RA_j to the kernel execution time. For example, in Figure 3, if the kernel execution time is k times longer than transfer time, then the transfer operation for SP_{k+1} would start right after the transfer operation for RA_k at time t. Table 1 shows the ratios of the transfer time to the kernel execution time for BFS and PageRank on three real data sets used in experimental evaluation. BFS has relatively high ratios since it is not computationally intensive, while PageRank has relatively low ratios since it is computationally intensive. Thus, the optimal k seems to be dependent on graph algorithms.

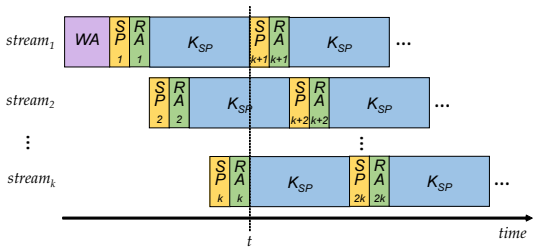
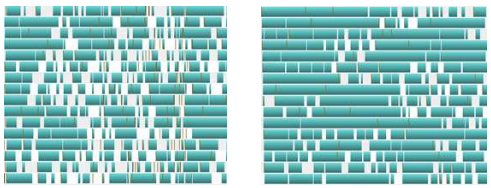


Figure 3: Timeline of copy operations in multiple streams.

Table 1: The ratios of transfer time to kernel execution time for BFS and PageRank on real data sets.

Algorithm	Twitter [18]	UK2007 [32]	YahooWeb [34]
BFS	1:3	1:1	2:1
PageRank	1:20	1:6	1:4

However, in practice, the performance continuously increases until using 32 streams, which will be shown in Section 5.4. This is because the kernel execution becomes faster when SP_j and RA_j are prepared in the queues of GPU in advance. The maximum number of streams that can execute a kernel function concurrently is 32 in the current CUDA technology [5]. After streaming and processing all $\{SP_j\}$ for WA, GTS performs streaming and processing all $\{LP_j\}$ for WA. The reason separating processing SPs from processing LPs is reducing the kernel switching overhead among SPs and LPs. After processing all data streamed to GPU is done, the updated WA is copied back to main memory for bulk synchronization, which is omitted in Figure 3. Figure 4 shows the actual timelines of copy operations for BFS and PageRank when using 16 streams on a synthetic data, which is obtained by profiling. In Figure 4, the very short red colored bars indicate copying SP_j and RA_j to device memory, while the long green colored bars indicate executing a kernel function. The timeline for PageRank in Figure 4(b) is denser than that for BFS in Figure 4(a) since PageRank is computationally intensive, whereas BFS is not.



(a) Streaming for BFS (b) Streaming for PageRank

Figure 4: Actual timeline of copy operations for BFS and PageRank when using 16 streams.

3.3 Handling BFS-like algorithms

We consider two major types of graph algorithms: (1) accessing a part of a graph via graph traversal and (2) accessing a whole graph by linear scanning vertices and edges [12]. The former algorithms are usually less computationally intensive, but causes non-coalesced memory accesses due to the irregular structure of graphs. They include Breadth-First Search (BFS), Single-Source-Shortest-Path (SSSP), neighborhood, induced subgraph, egonet, K-core, and cross-edges. BFS is the typical one [7, 8, 12], and hereafter, we denote them as BFS-like algorithms. The latter algorithms are usually computationally intensive, and the scan order of vertices and edges is not important in many cases. They include PageRank, degree distribution, Random Walk with Restart (RWR), radius estimations, and connected components. PageRank is the typical one [7, 8, 12], and hereafter, we denote them as PageRank-like algorithms.

The processing scheme described in Sections 3.1 and 3.2 is typically suitable for processing a single iteration of PageRank-like graph algorithms that access the entire topology data once. However, BFS-like algorithms requires level-by-level traversal, where a single level traversal accesses a very small portion of topology data and does not require streaming the entire topology data. Each traversal just requires streaming a set of topology pages containing the vertices to be visited. For that purpose, GTS keeps the data structure called nextPIDSet that contains the IDs of pages to be accessed at the next level. The local version of nextPIDSet is updated in each GPU during a single level traversal and copied back to main memory, and then, a CPU thread merges the local versions into a single global version. At the next level, the CPU thread copies the set of topology pages in the global nextPIDSet to GPU. As a result, GTS can integrate two types of algorithms having quite different access patterns into a single framework, which will be explained in Section 3.4.

After GTS allocates four buffers WABuf, RABuf, SPBuf, and LPBuf in the GPU device memory, there might be free memory available in GPU device memory. Especially, for BFS, since GTS allocates a small amount of WABuf due to small WA data, where WA is just a LV attribute vector for vertices, there might be a lot of free memory left in GPU device memory. In that case, GTS tries to maximize the performance of graph processing by caching topology data, i.e., SPs and LPs. The BFS-like algorithms could access the same topology pages repeatedly during traversal, and thus the caching scheme could avoid unnecessary copying from main memory to device memory. In general, the cache hit rate increases as the size of cache memory increases. When the total number of topology pages of a graph is $S + L$, a naive approximation of the cache hit rate using B pages would be $B/(S + L)$ for random graphs, though it also depends on a caching algorithm used. GTS basically adopts the LRU algorithm for the caching algorithm, but other algorithms can be used as well.

3.4 Algorithm of the framework

In this section, we present the algorithm of the GTS framework. Algorithm 1 presents the pseudo code of the framework. It performs a user-defined kernel function K_{SP} and K_{LP} for a specific graph algorithm on a graph G, where K_{SP} is a kernel for SPs and K_{LP} is a kernel for LPs. GTS requires such two kinds of kernels since SPs and LPs have a little different structure. As an initialization step, GTS loads G into main memory (MM), creates the streams for small pages and large pages for each GPU, and allocates the buffers WABuf, RABuf, SPBuf, and LPBuf in the device memory (DM) of each GPU. Then, it sets nextPIDSet, a set of page IDs to process next, depending on the type of the graph algorithm. If the graph algorithm is of BFS-like, the page ID con-

taining the start vertex is assigned to nextPIDSet. Otherwise, the constant ALL_PAGES is assigned to it. The map data structure cachedPIDMap$_i$ is initialized, which is used for storing the IDs of cached pages within GPU$_i$ (Line 8). We note that cachedPIDMap$_i$ is updated within GPU$_i$ during streaming topology data (Lines 14-27) and copied back to main memory (Line 29). Here, MMBuf indicates main memory buffer used for fetching the slotted pages from SSDs to main memory.

Algorithm 1 Framework of GTS

Input: Graph G, /* input graph */
K$_{\text{SP}}$, /* GPU kernel of Q for small pages */
K$_{\text{LP}}$, /* GPU kernel of Q for large pages */

Variable: nextPIDSet, /* set of page IDs to process next */
cachedPIDMap$_{1:N}$; /* cached page IDs in GPU$_{1:N}$ */
bufferPIDMap; /* buffered page IDs in MMBuf */

1: /* Initialization */
2: Create SPStream and LPStream for GPU$_{1:N}$;
3: Allocate WABuf, RABuf, SPBuf, LPBuf for GPU$_{1:N}$;
4: **if** Q is BFS-like **then**
5: nextPIDSet ← page ID containing start vertex;
6: **else**
7: nextPIDSet ← ALL_PAGES;
8: cachedPIDMap$_{1:N}$ ← ∅;
9: **if** $|G|$ < MMBuf **then**
10: Load G into MMBuf;
11: Copy WA to WABuf of GPU$_{1:N}$;
12: /* Processing GPU kernel */
13: **repeat**
14: /* repeat Lines 15-31 for LPs */
15: **for** $j \in$ nextPIDSet.SP **do**
16: **if** $j \in$ cachedPIDMap$_{h(j)}$ **then**
17: Call K$_{\text{SP}}$ for SP$_j$ in GPU$_{h(j)}$;
18: **else if** $j \in$ bufferPIDMap **then**
19: Async-copy SP$_j$ in MMBuf to SPBuf in GPU$_{h(j)}$;
20: Async-copy RA$_j$ to RABuf in GPU$_{h(j)}$;
21: Call K$_{\text{SP}}$ for SP$_j$ in GPU$_{h(j)}$;
22: **else**
23: Fetch SP$_j$ from SSD$_{g(j)}$ to MMBuf;
24: Async-copy SP$_j$ in MMBuf to SPBuf in GPU$_{h(j)}$;
25: Async-copy RA$_j$ to RABuf in GPU$_{h(j)}$;
26: Call K$_{\text{SP}}$ for SP$_j$ in GPU$_{h(j)}$;
27: Thread synchronization in GPU;
28: Copy WA of GPU$_{1:N}$ to MMBuf;
29: Copy nextPIDSet$_{1:N}$ and cachedPIDMap$_{1:N}$ to MMBuf;
30: nextPIDSet ← $\cup_{1\le i\le N}$ nextPIDSet$_i$;
31: **until** nextPIDSet = ALL_PAGES ∨ nextPIDSet = ∅

The *repeat-until* loop (Lines 10-34) takes charge of level-by-level traversal. For a PageRank-like algorithm, this loop is performed only once. Lines 15-27 are performed for processing small pages and performed similarly for processing large pages. We note that Lines 19-20 and 24-25 asynchronously transfer a topology page SP$_j$ (or LP$_j$) in nextPIDSet to a specific GPU$_{h(j)}$ according to the return value of $h(j)$, which will be explained in Section 4. Here, before transferring the page, GTS first checks if the page already exists in the cache of GPU$_{h(j)}$ by looking up cachedPIDMaph(j) (Line 16). We note that RA$_j$ for LP is a subvector of a single attribute value since LP$_j$ deals with only a single vertex. While executing a kernel, a new set of page IDs to process at the next level is assigned to local $nextPIDSet_i$ in device memory of each GPU$_i$, which is copied back to MMBuf (Line 29), and then merged into the global nextPIDSet (Line 30). The updated set of cachedPIDMap$_{1:N}$ are also copied back to MMBuf and used in the next level traversal. In the case of PageRank-like algorithms, both nextPIDSet and cachedPIDMap$_{1:N}$ are actually not used. The detailed kernel functions for BFS and PageRank are explained in Appendix B. In the case of PageRank, since the pseudo code in Algorithm 1 is for a single iteration of PageRank, users might need to perform Lines 13-31 as many times as necessary in their applications. Here, at the end of every iteration, nextPR should be initialized after being copied to prevPR.

4. EXPLOITING MULTIPLE GPUS

In this section, we present two strategies for exploiting multiple GPUs. GTS can be easily extended to exploit multiple GPUs. We let the number of GPUs be N. Section 4.1 presents the strategy for high performance with a limit on scalability, whereas Section 4.2 presents the strategy for high scalability with a limit on performance.

4.1 Strategy for performance

The first strategy of GTS for multiple GPUs is copying the same attribute data, especially WA, to all GPUs and copying a different topology data to each GPU. Figure 5(a) shows the data flow scheme of that strategy, which consists in four different steps. In Step 1, GTS copies the same WA to all $\{\text{GPU}_1, \cdots, \text{GPU}_N\}$, which are denoted as solid arrows. In Step 2, it executes a given GPU kernel K_{SP} while streaming a different $\langle$ SP$_k$, RA$_k$ $\rangle$ to each GPU$_k$ for $1 \le k \le N$, which are denoted as dotted arrows. More specifically, it copies $\langle$ SP$_{i*N+j}$, RA$_{i*N+j}$ $\rangle$ to GPU$_j$ for $0 \le i \le \lceil \frac{S}{N} \rceil - 1$ and $1 \le j \le N$. LPs are processed in the same way. Here, each GPU can execute the same GPU kernel function independently for a different part of topology data. Steps 3 and 4 perform the data synchronization of WA that has been updated during Step 2, which are denoted as double-lined arrows. GTS exploits the peer-to-peer memory copying feature of modern GPUs. When using multiple GPUs, a naive synchronization method is performing N times synchronization from GPUs to main memory directly, one time per each GPU. This approach might suffer from synchronization overhead as N increases. GTS largely reduces such synchronization overhead by exploiting peer-to-peer memory copying among GPUs, which speed is much faster than that of between GPU and main memory. In Step 3, the WA data of each GPU is copied and merged to the device memory of a master GPU (e.g., GPU$_1$), and then, in Step 4, the updated WA data in GPU$_1$ is copied to main memory.

In terms of the framework of GTS (i.e., Algorithm 1), Step 1 corresponds to Line 11, Step 2 to Lines 16-26, and Steps 3-4 to Line 28. In Step 2, for load balancing, the function $h(x)$ returns a hash value for a page ID j of SP$_j$ (or LP$_j$) such that the page SP$_j$ (or LP$_j$) is streamed to GPU$_{h(i)}$. Typically, GTS uses the mod function for the default hash function.

This strategy of GTS potentially can achieve fairly linear parallel speedup with respect to the number of GPUs for graph processing, as long as the capability of data streaming is sufficient. Moreover, since the different topology data distributed over GPUs have almost equal sizes, i.e., almost the same amount of workload under this strategy, the speedup ratio can be fairly stable regardless of the characteristics of a graph (e.g., its size and its density). The capability of data streaming largely depends on (1) the speed of PCI-E interface and (2) the I/O performance of SSDs. Under the current computer architecture, the I/O speed of SSDs (e.g., about up to 2 GB/sec) is much slower than that of PCI-E interface (e.g., 16 GB/sec). Thus, the overall performance of this strategy is limited to the I/O performance of SSDs. To alleviate this performance problem, GTS exploits multiple SSDs. In detail, GTS stores each slotted page SP$_j$ of G ($1 \le j \le S$) in a specific SSD$_{g(j)}$, where

$g(j)$ returns a hash value for a page ID j, and fetches the corresponding page from $SSD_{g(j)}$ according to the I/O request at Line 23 in Algorithm 1. If the size of G is smaller than main memory buffer (MMBuf), fully loading G into MMBuf might be a better method (Lines 9-10), especially for BFS-like algorithms. In that case, the speed of PCI-E interface would become a performance bottleneck.

Although this strategy shows high and stable performance of fairly linear speedup with multiple GPUs, it could suffer from GPU's limited device memory. It could not process a graph algorithm requiring WA larger than the size of a single GPU's device memory. For example, this strategy can process the PageRank algorithm only up to RMAT30 using a GPU having 6 GB device memory.

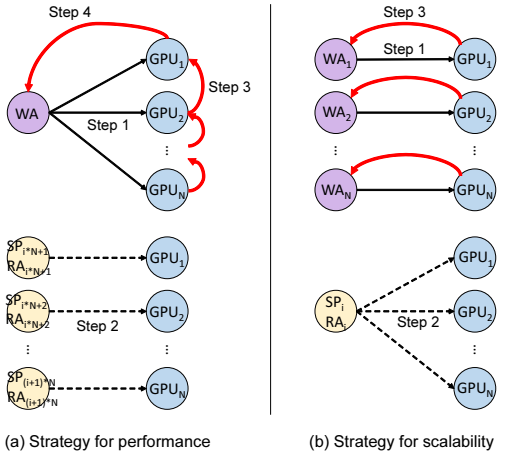


Figure 5: Two strategies of GTS exploiting multiple GPUs.

4.2 Strategy for scalability

The second strategy of GTS for multiple GPUs is copying a different attribute data, especially a different WA_i chunk to each GPU and copying the same topology data to all GPUs. Figure 5(b) shows the data flow scheme of that strategy, which consists three steps. In Step 1, GTS copies a different WA_i to each GPU_i for $1 \leq i \leq N$, which are denoted as solid arrows. In Step 2, it executes a given GPU kernel K_{SP} while streaming the same $\langle$ SP_j, RA_j $\rangle$ to all GPUs, which are denoted as dotted arrows. LPs are processed in the same way. Here, each GPU can execute the same GPU kernel function independently for a different part of attribute data. Step 3 performs the data synchronization of all $\{$ WA_i $\}$ that has been updated during Step 2, which are denoted as double-lined arrows. Since all WA_i chunks are disjoint with each other, GTS cannot use the peer-to-peer memory copying feature of GPUs. This strategy just uses a naive synchronization method of performing N times synchronization from GPUs to main memory directly, one time per each GPU.

In terms of the framework of GTS, as in Section 4.1, Step 1 corresponds to Line 11, Step 2 to Lines 16-26, and Steps 3-4 to Line 28. However, in Step 2, the function $h(x)$ returns a set $\{1, \cdots, N\}$ instead of a single hash value for a page ID j such that the page SP_j (or LP_j) is streamed to all GPUs.

This strategy of GTS tries to maximize the size of a graph to process. Especially, it can achieve linear increase of the size of a graph to process, with respect to the number of GPUs, as long as the capacity of main memory or SSDs is sufficient. Since the different attribute data distributed over GPUs have almost equal sizes, and at the same time, the same topology data is fed into those GPUs, each GPU has almost the same amount of workload under this strategy. That is, the workload of graph processing is well balanced among GPUs regardless of the characteristics of a graph. This strategy is logically analogous to using a single GPU of large device memory. Thus, although increasing the number of GPUs, the performance of graph processing itself does not change, and the capability of data streaming to GPU also does not change. If we have a large amount of main memory, and so, perform this strategy without accessing to SSDs, the speed of PCI-E interface would be a performance bottleneck. Otherwise, the I/O performance of SSDs would be a bottleneck. However, when we exploit multiple SSDs as described in Section 4.1, the gap between the I/O performance of SSDs and the logical speed of PCI-E interface is not so much due to the fixed capability of data streaming to the logical single GPU. That means the overall performance of this strategy would not increase much even though processing an entire graph in main memory.

Consequently, the strategy for scalability of GTS is suitable to process a relatively large-scale graph where its WA cannot fit in a single GPU's device memory by storing the graph on SSDs (e.g., an RMAT32 graph in a machine of 6 GB GPUs and 500 GB SSDs). On the contrary, the strategy for performance of GTS (in Section 4.1) is suitable to process a relatively small-scale graph where its WA can fit in a single GPU's device memory by storing the graph in main memory (e.g., an RMAT30 graph in a machine of 6 GB GPUs and 128 GB main memory).

5. COST MODELS

In this section, we present the cost models of GTS, which allow us to understand the performance tendency and further improve the performance later through the cost-based optimization. We only consider major factors that could affect the performance of GTS. Since PageRank-like algorithms and BFS-like algorithms show a quite different tendency, we present two cost models. For simplicity, we present the cost models for Strategy-P without I/O.

5.1 Cost model for PageRank-like algorithms

The cost model for PageRank-like algorithms is given by

$$\frac{2|WA|}{c1} + \frac{|RA|+|SP|+|LP|}{c2 \times N} + t_{call}\Big(\frac{S+L}{N}\Big) + t_{kernel}(SP_{|1|} + LP_{|1|}) + t_{sync}(N). \quad (1)$$

where N is the number of GPUs, $c1$ is the communication rate (e.g., in MB/s) between main memory and device memory in a chunk copy mode, $c2$ is the communication rate in a streaming copy mode, $t_{call}(x)$ is the time overhead of calling a kernel function x times, $t_{kernel}(y)$ is the kernel execution time to process y pages, and $t_{sync}(z)$ is the time overhead of synchronization among z GPUs. Here, $c1$ is usually higher than $c2$ for GPUs. For example, in PCI-E 3.0 x16 interface, $c1$ is about 16 GB/s, while $c2$ is about 6 GB/s. In Eq. 1, $\frac{2|WA|}{c1}$ indicates the total amount of time for copying all WA to device memory and copying the updated ones back to main memory. That time does not decrease while using multiple GPUs. The transfer time of $\frac{|RA|+|SP|+|LP|}{c2}$ is divided by N since the data is transferred concurrently to N GPUs. The time overhead of calling the kernel function $t_{call}(S+L)$ is also divided by N. The term $t_{kernel}(SP_{|1|}+LP_{|1|})$ indicates the last kernel execution time for the last single SP and the last single LP that are not hidden by data streaming. They are not negligible since PageRank-like algorithms are usually computationally intensive. They also cannot be divided by N since every GPU does the same thing. We note that the time overhead $t_{sync}(N)$ increases as N increases in order to synchronize WAs among more GPUs.

5.2 Cost model for BFS-like algorithms

The cost model is for BFS-like algorithms is given by

$$\frac{2|WA|}{c1} + \sum_{l=0}^{depth} \left(\frac{|RA_{\{l\}}| + |SP_{\{l\}}| + |LP_{\{l\}}|}{c2 \times N \times d_{skew}} \times (1 - r_{hit}) + t_{call}\left(\frac{S_{\{l\}} + L_{\{l\}}}{N \times d_{skew}}\right)\right). \quad (2)$$

where depth is the number of traversal levels, $SP_{\{l\}}$ is a set of small pages visited at an l-th level of traversal, d_{skew} is the degree of workload skewness (i.e., imbalance) among GPUs, and r_{hit} is the cache hit rate $\frac{B}{S+L}$ discussed in Section 3.3. The operations in the braces at different levels of traversal cannot overlap with each other due to synchronization barrier, and thus the total amount of time is just a sum of the times from level 0 to level depth. The transfer time of data $\frac{|RA_{\{l\}}|+|SP_{\{l\}}|+|LP_{\{l\}}|}{c2}$ is divided by N due to using N GPUs, and moreover divided by d_{skew}, which is between $\frac{1}{N}$ (most imbalanced) and 1 (most balanced). We need to consider this factor since page access patterns of BFS-like algorithms might not be quite balanced different from PageRank-like algorithms. In the most imbalanced case, the transfer time of data is the same with that of using only one GPU. The term (1-r_{hit}) represents the caching effect, where r_{hit} is between 0 (no cache hit) and 1 (all cache hits). There is no term $t_{kernel}(y)$ in this cost model since the kernel execution time of BFS-like algorithms is not a major factor. There is also no term $t_{sync}(z)$ since the size of WA to be synchronized (e.g., LV) is usually negligible. In the term $t_{call}(\frac{S_{\{l\}}+L_{\{l\}}}{N \times d_{skew}}, S_{\{l\}})$ indicates the number of small pages visited at an l-th level of traversal.

6. IMPLEMENTATION

6.1 Data format for trillion-scale graphs

In terms of the slotted page format, although the one proposed in [12] is useful for representing a graph topology data for secondary storage, there is a clear limit to the maximum size of a graph to represent. The physical ID of 4-byte (2 bytes for page ID and 2 bytes for slot number) can theoretically represent a graph of up to $2^32 = 4$ billion vertices. In practice, however, it fails to represent an RMAT30 graph of 1 billion vertices and 16 billion edges due to the two-level addressing scheme (of page ID and slot number) and the skewness of the node degree distribution. Thus, in order to handle even a trillion-scale graph, we slightly generalize the existing format such that p-byte page ID (ADJ_PID) and q-byte slot number (ADJ_OFF) are used for addressing. For example, when considering the physical ID of 6-byte, there are three possible configurations as in Table 2, where $(p = 2, q = 4)$ means a small number of large-sized pages, $(p = 3, q = 3)$ a medium number of midium-sized pages, and $(p = 4, q = 2)$ a large number of small-sized pages. In the table, the maximum page size is calculated under the assumption that ADJLIST_SZ is of 4-byte, VID of 6-byte, and OFF of 4-byte. Among configurations, we choose $(p = 3, q = 3)$ and implement our method using 64 MB page size, since both p and q are well-balanced, and the page size of 64 MB is compatible with the default block size widely used in many big data framework such as Hadoop [31] and Spark.

Table 2: Three possible configurations of physical ID of 6-byte.

p	q	max. page ID	max. slot number	max. page size
2	4	64 K	4 B	80 GB
3	3	16 M	16 M	320 MB
4	2	4 B	64 K	1.25 MB

6.2 Micro-level parallel processing

GTS mainly focuses on coarse-granular or macro-level parallel graph processing for handling large-scale graphs that do not fit in GPU device memory. However, when GTS calls a given GPU kernel on topology data page-by-page, the GPU kernel can apply various kinds of fine-granular or micro-level parallel graph processing techniques to each page. The kernel can apply a better/different technique to each page depending on the characteristics of the page (e.g., density, i.e., the ratio of the number of vertices to the number of edges within a page).

For example, we exploit the VWC technique [15] as a default technique for processing each slotted page, where the threads in each warp process the outgoing edges of each vertex simultaneously. We presents the GPU kernels for BFS and PageRank exploiting the VWC technique in Appendix B. We denote the VWC technique as edge-centric in this paper. On the contrary to edge-centric, we can use another micro-level technique that makes each GPU thread process each vertex and its entire outgoing edges. We denote that technique as vertex-centric. Then, we can consider a hybrid micro-level technique combining both edge-centric and vertex-centric. In general, the vertex-centric technique might be suitable for very sparse graphs where each vertex has only few outgoing edges, while the edge-centric one might be suitable for less-sparse graphs. The hybrid technique can handle both types of graphs by applying a different micro-level technique to each page depending on the density of the page. We will show the effectiveness of each technique in Appendix E.

7. PERFORMANCE EVALUATION

In this section, we present experimental results in four categories. First, we evaluate the performance of GTS compared with the state-of-the-art distributed graph processing methods, Apache Giraph [1, 11], Apache Spark GraphX [10, 33, 35], PowerGraph (GraphLab v2.2) [9, 20, 21], and Naiad [23, 25] to show the superiority of our method. Second, we evaluate the performance of GTS compared with the state-of-the-art CPU-based graph processing methods, Ligra [29], Ligra+ [30], and Galois [26], to show the superiority of our method. For reference, we also evaluate the performance of the parallel graph processing method using CPUs called MTGL [2], which is widely used for comparison [36]. Third, we evaluate the performance of GTS compared with the state-of-the-art GPU-based graph processing method, TOTEM [7, 8], to show the superiority of our method. To the best of our knowledge, TOTEM is the only method to process large-scale graphs that do not fit in GPU device memory and also to exploit multiple GPUs. For reference, we also evaluate the performance of Cusha [16] and MapGraph [6], which can process only the graph data that can fit in GPU memory. Fourth, we evaluate the performance of GTS while varying strategies (of Section 4), storage types (i.e., SSD and HDD), the number of streams, and the densities of graphs to show the characteristics of GTS.

7.1 Experimental setup

For experiments, we use both synthetic and real datasets. For synthetic datasets, we generate scale-free graphs following a power law degree distribution by using RMAT [4]. We generate from RMAT27 to RMAT32, where the ratio of the number of vertices to the number of edges is set to 16. For real datasets, we use three well-known graphs of Twitter [18], UK2007 [32], and YahooWeb [34], which all have different sizes and characteristics. Table 3 shows the basic statistics of those data sets. For GTS,

we use $(p = 2, q = 2)$ in Section 6.1 for storing RMAT27-29 graphs and real graphs since their sizes are relatively small. In the table, #SP and #LP mean the number of small pages and that of large pages, respectively, under the corresponding configuration $(p = 2, q = 2)$. Most of topology pages are small pages in both synthetic and real graphs. We use $(p = 3, q = 3)$ for storing RMAT30-32, where there is no LP due to the large page size of 64 MB.

Table 3: Statistics of graph datasets used in the experiments.

data	#vertices	#edges	statistics for GTS		
			(p, q)	#SP	#LP
RMAT27	128 M	2,048 M	(2,2)	9,724	58
RMAT28	256 M	4,096 M	(2,2)	19,533	62
RMAT29	512 M	8,192 M	(2,2)	38,747	937
RMAT30	1 B	16 B	(3,3)	1,786	0
RMAT31	2 B	32 B	(3,3)	3,584	0
RMAT32	4 B	64 B	(3,3)	7,175	0
Twitter	42 M	1,468 M	(2,2)	5,418	1,029
UK2007	106 M	3,739 M	(2,2)	15,484	0
YahooWeb	1,414 M	6,636 M	(2,2)	32,807	0

We summarize the statistics of the size of WA data versus the size of topology data in the slotted page format in Table 4. We can see the ratio of the WA data to the topology data is very small, which is between 1.7% and 10%. The WA data for up to RMAT32 can fit in two NVIDIA TITAN X GPUs' memory (i.e., 24 GB), except RMAT32 for CC.

Table 4: Statistics of the sizes of WA data versus topology data in the slotted page format (GByte).

data	topology	WA			
		BFS	PageRank	SSSP	CC
RMAT28	20	0.5	1	1	2
RMAT29	40	1	2	2	4
RMAT30	114	2	4	4	8
RMAT31	229	4	8	8	16
RMAT32	459	8	16	16	32

We conduct all the experiments of four distributed graph processing methods on the same cluster of one master node and 30 slave nodes connected via Infiniband QDR (40 Gbps), each node of which is equipped with two Intel Xeon 8-core 2.60 GHz CPUs, 64 GB memory, and two 3 TB HDDs (RAID 0). The cluster has a total of 480 CPU cores and 1,920 GB memory. We also conduct all the experiments of four CPU-based methods and four GPU-based methods on the same workstation equipped with two Intel Xeon E5-2687W 3.1GHz CPUs of eight cores, 128 GB main memory, two NVIDIA GTX TITAN X GPUs of 12 GB device memory, and two Fusion-io's PCI-E SSD. The CPUs and GPUs are connected with PCI-E 3.0 x16 interface. For graph processing, GTS uses only GPUs, while TOTEM uses both two CPUs and GPUs. All CPU-based methods use 16 threads after turning off the Hyper-Threading (HT) option for performance.

In terms of software versions and configurations, we use Scala 2.11.7 and Spark 1.5.1 for GraphX, MPI ICC 14.0.0 for PowerGraph, and Hadoop 1.2.1 for all three distributed methods. For Giraph, we set the size of mapper memory to 60 GB. For Spark, we set the size of executor memory to 60 GB. Naiad requires the.NET framework, and so, we use Mono (JIT compiler version 3.2.8) for running Naiad on Linux. For MTGL, Galois, Ligra, Ligra+, TOTEM, CuSha, and MapGraph, we download their latest source codes. We compile all single-machine methods with the same optimized option of -O3 with gcc 4.9 and CUDA 7.5. If a method requires its own data format, we convert graph data to its own format (e.g., Galois, Ligra, Ligra+, CuSha, and MapGraph). Different from GTS, TOTEM requires a different set of options for each graph algorithm and each data set in order to achieve the best performance [8]. We use the sets of options recommended by the authors of TOTEM for most of experiments. We also have found Naiad often failed to process graph queries due to lack of memory, and so, adjusted its configuration to achieve its best scalability and performance (e.g., sizes of heaps and arrays).

7.2 Comparison with Distributed Methods

Figure 6 shows the comparison results among GraphX, Giraph, PowerGraph, Naiad, and GTS, for BFS and PageRank. Y-axis represents the elapsed times in seconds (in log-scale), and O.O.M means out of memory error. In the case of PageRank, we measure the total elapsed times of ten iterations. For four distributed methods, we measure the elapsed time, excluding loading and finalization times. For GTS, we measure the elapsed times between starting reading the first page from main memory (for real graphs and RMAT28-30) or SSDs (for RMAT31-32) and showing the query results. Here, for real graphs and RMAT28-30, since they can fit in main memory, we exclude loading time (Lines 1-10 in Algorithm 1) for a fair comparison. We set the buffer size of GTS to 20% of a graph size for RMAT31 and RMAT32 (e.g., 45 GB for RMAT31).

For all datasets used, GTS significantly outperforms the distributed graph processing methods using 30 machines, for both BFS and PageRank. Moreover, GTS shows the best scalability among the methods compared. Only GTS can process all graphs of up to RMAT32 for both BFS and PageRank. Among four distributed methods, Naiad shows the worst scalability, Giraph shows the worst performance, and PowerGraph the best scalability and performance, in general. The reason that the processing time of GTS rapidly increases between RMAT30 and RMAT31 is due to including I/O time of SSDs and changing the strategy from performance (of Section 4.1) to scalability (of Section 4.2). Theoretically, the processing time of GTS should increase linearly between RMAT31 and RMAT32 since GTS uses the secondary storage and the same strategy for both datasets, but it actually does not. This is because there are higher-degree vertices in RMAT32, and the performance of GPUs tends to be degraded (e.g, down-clocking) due to overheat when processing for a long time.

7.3 Comparison with CPU-based Methods

Figure 7 shows the comparison results among MTGL, Galois, Ligra, and Ligra+, and GTS, for BFS and PageRank. In the figure, except GTS, there is no results for relatively large-scale graphs such as RMAT29-30 and YahooWeb, since the CPU-based methods cannot load data into main memory or process graph algorithms due to lack of main memory. Among the CPU-based methods, Galois, Ligra, and Ligra+ have significantly outperformed the multi-threaded graph library (MTGL) in terms of both the elapsed time and the size of a graph to process, except the case of Twitter for PageRank. Among three CPU-based methotds, Ligra and Ligra+ show a better performance than Galois, except the case of UK2007 for BFS. Ligra shows a similar performance with Ligra+. However, we could not execute Ligra+ for UK2007, RMAT27, and RMAT28, due to segmentation fault errors, which were executed successfully in Ligra. We guess the Ligra+ source code is not stable yet. Compared with GTS, either Galois or Ligra slightly outperforms GTSfor relatively small graphs for BFS. This is because the CPU-based methods perform edge-level random access for traversal algorithms, while GTS performs page-level random access with data transfer overhead between main memory and GPUs. For relatively

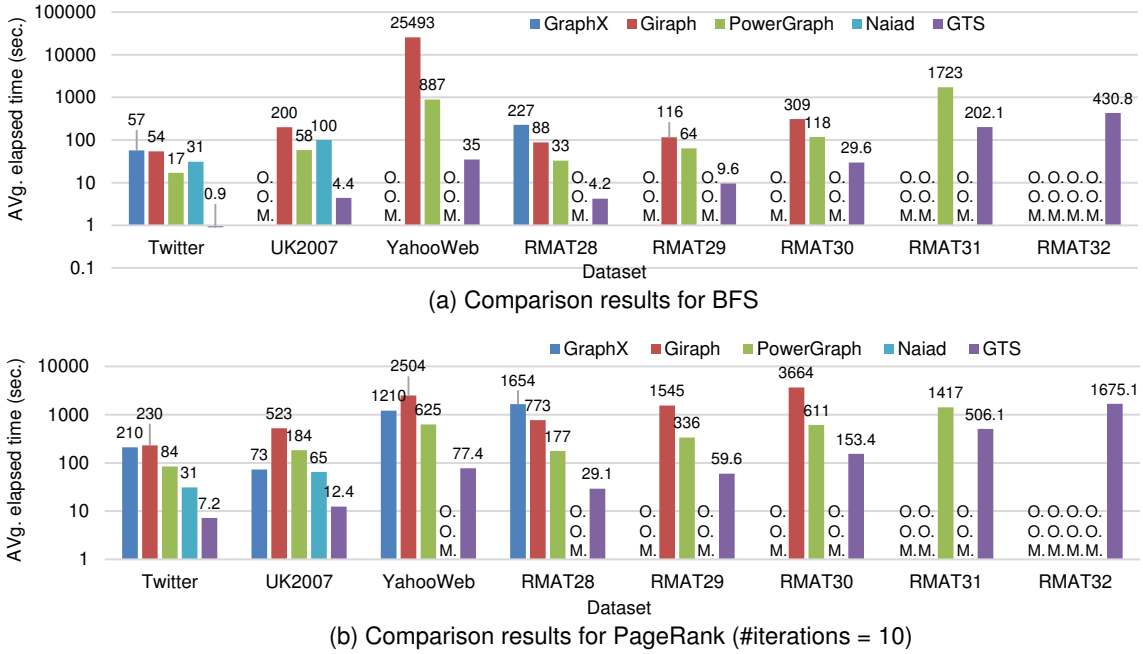


Figure 6: Comparison with GraphX, Giraph, PowerGraph, and Naiad for BFS and PageRank (Y-axis is log-scale).

large graphs (e.g., YahooWeb, RMAT29-30), only GTS could process BFS. For PageRank, GTS significantly outperforms all CPU-based methods in terms of both the elapsed time and the size of a graph to process.

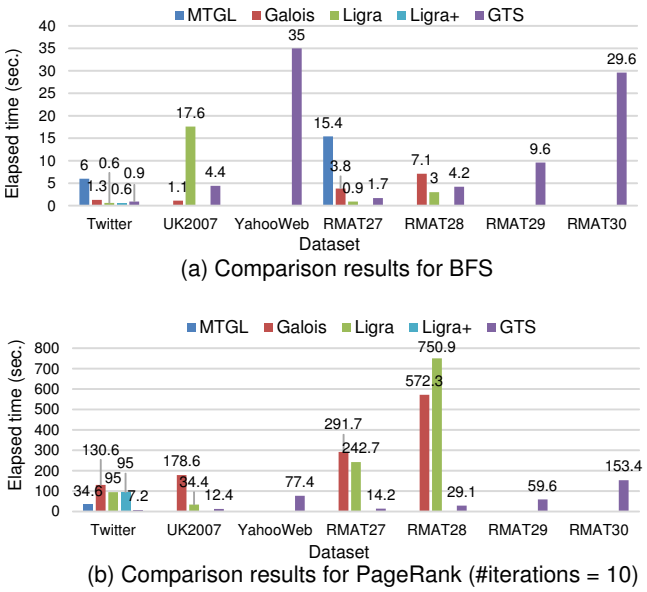


Figure 7: Comparison with MTGL, Galois, Ligra, and Ligra+ for BFS and PageRank.

7.4 Comparison with GPU-baesd Methods

Figure 8 shows the comparison results among MapGraph, CuSha, TOTEM, and GTS, for BFS and PageRank. Both CuSha and MapGraph can process only the graph data that can fit in GPU memory, and so, the size of a graph to process is very small. CuSha can process BFS only up to Twitter data. It cannot process other data (e.g., RMAT27) due to lack of GPU memory. We expected CuSha would be faster than GTS as long as a graph could fit in GPU memory. However, CuSha was slower than GTS, and even than TOTEM for Twitter. It cannot process PageRank for all graphs tested, since PageRank requires more memory than BFS due to prevPR and nextPR. MapGraph is worse than CuSha in terms of scalability. It cannot process even BFS for Twitter. It can just process a tiny graph like LiveJournal. It is because the Market Matrix format of MapGraph is less space-efficient than the G-Shard format of CuSha.

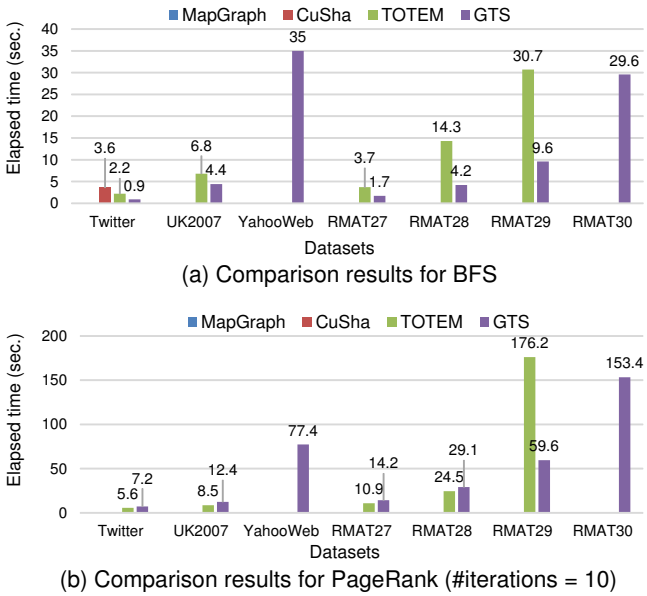


Figure 8: Comparison with TOTEM, MapGraph, and CuSha for BFS and PageRank.

We compare the performance of GTS with the best performance of TOTEM using the set of options carefully selected. In the case of TOTEM, we can minimize the amount of graph data processed by slower processors, i.e., CPUs, by fitting as much graph data as possible in device memory, and thus, maximize its performance. Appendix C shows the set of options including the ratios of graph data processed by GPUs to that by CPUs in TOTEM (GPU%:CPU%), most of which are the ones recommended by the authors. For PageRank, TOTEM slightly outperforms GTS for relatively small-scale graphs such as RMAT27, Twitter, and UK2007. GTS, however, significantly outperforms TOTEM for large-scale graphs such as RMAT29. For BFS, GTS consistently outperforms TOTEM. Here, TOTEM cannot process YahooWeb due to some bugs, and so, there is no corresponding result. In addition, TOTEM cannot process RMAT30-32 since it relies on in-memory data format requiring a contiguous array in main memory. We note that GTS processes PageRank for RMAT29 only in about 59 seconds, which indicates the graph processing speed of GTS is about 7 GB/s, since the size of RMAT29 is about 40 GB in the slotted page format, and the number of PageRank iterations is ten in the experiments. We also note that GTS shows the performance of up to 1,500 MTEPS (millions traversed edges per second) for Twitter.

7.5 Characteristics of GTS

Figure 9 shows the performance of GTS while changing the strategy explained in Section 4 for RMAT30. Strategy-P indicates the strategy for performance in Section 4.1, and Strategy-S the strategy for scalability in Section 4.2. Both strategies show similar performance with each other when using 1 SSD or 2 HDDs since the I/O performance is a bottleneck. However, Strategy-P shows a slightly better performance than Strategy-S when using main memory or 2 SSDs due to no or less I/O bottleneck.

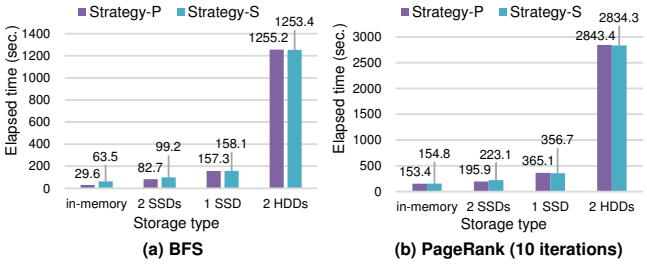


Figure 9: Comparison between two strategies for BFS and PageRank (RMAT30).

In terms of the overall performance, we note that the speed of PCI-E bus becomes a bottleneck in memory setting, and the I/O performance of PCI-E SSDs becomes a bottleneck in SSD setting. For example, for ten iterations of PageRank using RMAT30, GTS in memory setting takes about 153 seconds, which is approximately equal to $114 \times 10 \div 6 = 190$ seconds, where 6 means the communication rate in a streaming copy mode $c2$ in Section 5.1. Here, actual elapsed time of 153 seconds is smaller than the calculated time of 190 seconds due to caching mechanism described in Algorithm 1. For another example, GTS using two SSDs takes about 196 seconds, which is approximately equal to $114 \times 10 \div 5 = 228$ seconds, where 5 (GB/s) means the sequential read performance of two PCI-E SSDs. Here, actual elapsed time of 196 seconds is smaller than the calculated time of 228 seconds due to the page buffering mechanism in Algorithm 1. The performance of GTS using two HDDs is completely bound by the I/O performance of HDDs. When using two HDDs in the Strategy-P mode, its sequential read I/O bandwidth is about 330 GB. The elapsed time of PageRank for RMAT30 is about 2,843 seconds, where the calculated time is $114 \times 10 \div 0.33 = 3,454$ seconds. Here, actual elapsed time of 2,843 seconds is smaller than the calculated time of 3,454 seconds due to the page buffering mechanism.

Figure 10 shows the performance of GTS while varying the number of streams for RMAT26-29. The performance increases steadily as the number of streams increases for all data sets. Even for BFS where the ratios of transfer time to kernel execution time are much smaller than 32, it does due to the reason explained in Section 3.2.

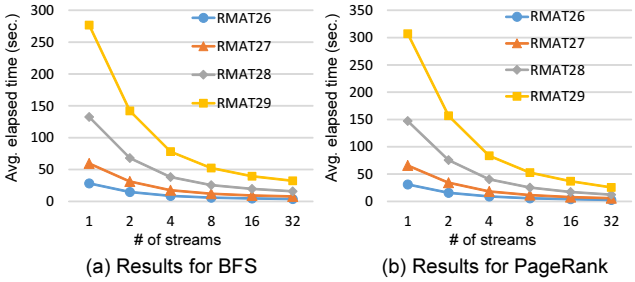


Figure 10: Performance when varying the number of streams.

Figure 11(a) shows the performance of GTS for BFS while varying the cache size from 32 MB to 5,120 MB, and Figure 11(b) shows the corresponding cache hit rates. For RMAT29, there is no result at the cache size 5,120 MB due to a large size of WABuf. We can easily adjust the size of cache since it is allocated by a CPU thread (i.e., the framework thread of GTS). For example, for the cache of 1,024 MB, GTS allocates the array of 16 slotted pages of 64 MB within GPU_i and make $cachedPIDMap_i$ maintain up to 16 page IDs. In Figure 11(b), the cache hit rates increase linearly as the cache sizes increase, but decrease linearly as the sizes of topology data increase, as discussed in Section 3.3.

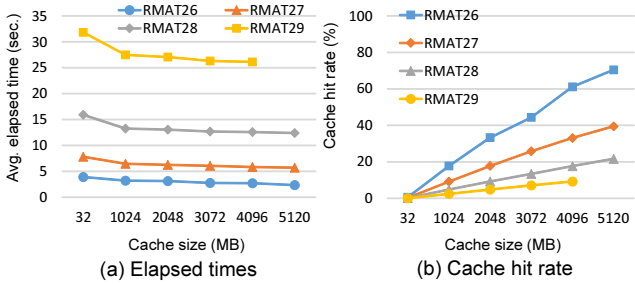


Figure 11: Effectiveness of caching for BFS.

8. RELATED WORK AND DISCUSSION

Most of the existing graph systems (e.g., Apache Giraph, PowerGraph) follow the *vertex-centric* scatter-gather model involving random access to topology data (i.e., vertices and edges), and so, require topology data to be in main memory for performance. In contrast, X-Stream [28] has proposed the *edge-centric* scatter-gather model that can exploit sequential access to edge data, and so, only requires both vertex data and update data to be in main memory. These two approaches are two extremes in terms of edge access: the former relies on only fine-grained (i.e., edge level) random access, and the latter relies on only fine-grained sequential access (i.e., streaming edges). As a result, the latter approach has a significant performance penalty for traversal algorithms (e.g., BFS, SSSP) which the former approach does not have. For a large-scale graph (e.g., YahooWeb) having a high diameter, X-Stream executes a very large number of scatter-gather iterations, each of which re-

quires streaming the entire edge list but doing little work. Accordingly, for traversal algorithms for such a graph, X-Stream did not finish in a reasonable amount of time [28]. GraphChi [19] has the similar problem, but shows a worse performance than X-Stream, due to requiring fully loading (not streaming) a shard file and no overlapping between disk I/O and computation. GTS is quite different from the above two extremes, since it exploits coarse-grained (i.e., page level) sequential access, and at the same time, coarse-grained (i.e., page level) random access. We have presented this *hybrid* mechanism supporting both sequential and random accesses in detail in Section 4.3, Algorithm 1, and Appendix A. Due to exploiting both, GTS requires streaming only the relevant pages for traversal algorithms.

We discuss how GTS is differ from the streaming mechanism used in X-Stream and other existing work in more detail. Like other methods following the scatter-gather model, X-Stream needs to update the data field of each vertex after each scatter-gather iteration. In order to do that, it tries to maintain the whole vertex data including both read-only attributes and updatable attributes in main memory. However, since the size of the whole vertex data might not fit in main memory, X-Stream partitions the vertex and edge data into multiple partitions, where each partition can fit in main memory, and performs the three phases of scatter, shuffle, and gather for each partition (not two phases of scatter and gather). In order to update the vertex data of other partitions, the shuffle phase is essential. In the shuffle phase, X-Stream builds the *update* data structure, which is used for updating the vertex data structure later. In this scheme, both vertex data and update data take a substantial amount of main memory, and the computational overhead of the shuffle phase is also considerable. In contrast, GTS separates the data fields of vertices into read-only (i.e., RA) and updatable (i.e., WA), and maintains the only and entire WA data in GPU memory. By minimizing the amount of data to be kept in memory, it can keep the entire WA data in GPU memory even for billion-scale graphs, and moreover, do not need to build the update data structure and perform the shuffle phase. If the WA data is larger than the single GPU memory, we can spread it to multiple GPUs' memory in Strategy-S. As long as the WA data can fit in GPUs' memory, GTS can perform high-performance streaming in the hybrid mechanism described above. Unlike X-Stream, it has no shuffle phase and no write operations to secondary storage. It performs read-only streaming from beginning to end, while X-Stream performs a mixture of read and write streaming. It fully exploits sequential streaming bandwidth, while X-Stream only partially exploits the bandwidth.

There are a number of graph processing methods using GPUs on a single computer [7, 8, 13, 15, 16, 24, 36]. The VWC method [15] proposes the virtual warp scheme that enables trading off between workload imbalance and ALU underutilization with a single parameter, the number of threads per virtual warp. It usually partitions a physical warp of 32 threads into multiple virtual warps of 4, 8, or 16 threads. Too large virtual warp could cause unused ALUs within a warp, which could limit the parallel performance of kernel executions. CuSha [16] adopts the shards format [19] for solving the non-coalesced memory access problem and presents two graph representations: G-Shards and Concatenated Windows (CW). It focuses on fully utilizing the GPU computing power by processing multiple shards in parallel on GPU's streaming multiprocessors. Medusa [36] proposes a programming framework that can simplify implementation of GPU programs for graph processing. [24] presents a BFS parallelization method that focuses on fine-grained task management constructed from efficient prefix sum, which achieves an asymptotically optimal $O(|V| + |E|)$ complexity. All the work mentioned above lack support for large-scale graphs that do not fit in the GPU's limited device memory. However, many techniques addressed in the above work belong to micro-level parallel processing techniques and are orthogonal to our method GTS, and so they can be applied to processing each topology page.

TOTEM [7, 8] is the only work to process large-scale graphs and exploit multiple GPUs, to the best of our knowledge. It partitions a graph into two parts: (1) the main memory part processed by CPUs and (2) the device memory part processed by GPUs. Although it can handle large-scale graphs that other methods cannot, it still has three major drawbacks. First, it completely underutilizes the computational power of GPUs. It processes only a small fraction of a graph by using GPUs. The remaining part of the graph is processing by relatively slow CPUs. This underutilization becomes more and more marked as the graph size increases, since the size of the part processed by GPUs is fixed. Second, it it is not very scalable in terms of the number of GPUs used. TOTEM demonstrates the graph processing power of GPU is higher than that of CPUs, and so it concludes that using more GPUs instead of more CPUs are required for faster graph processing. However, under the partitioning scheme like edge-cut, the number of cut edges among main memory and multiple GPUs increases as the number of GPUs increases, which means the amount of data to be communicated among main memory and GPUs also increases [9]. As a result, the speedup tends to decrease as well. Third, it is difficult for users to optimize the performance due to a lot of configuration options. Different from GTS, TOTEM requires a different set of options for each graph algorithm and for each data set in order to achieve good performance. If users do not carefully tune a bunch of options, its performance could be significantly degraded. Although TOTEM does not outperform GTS, we consider that hybrid computation using both CPUs and GPUs is potentially will be superior to our method GTS using only GPUs.

9. CONCLUSIONS

In this paper, we proposed a fast and scalable GPU-based graph processing method called GTS that can process even RMAT32 (64 billion edges) graphs very efficiently. GTS fully exploits the computational power of GPUs by processing the entire graph only using GPUs. To overcome the limit of GPU memory capacity and moreover the limit of main memory capacity, we proposed a concept of storing only updatable attribute data and moving topology data. The proposed method stores graphs in PCI-E SSDs and executes a graph algorithm using thousands of GPU cores while streaming topology data of graphs to GPUs via PCI-E interface. For streaming topology data, GTS exploits the asynchronous GPU streams (e.g., CUDA Streams), which could hide memory access latency from GPUs to main memory and so utilize GPU's computing power more. For efficient streaming, GTS adopted and generalized the slotted page format that divides a graph into fixed-size units. In terms of exploiting multiple GPUs and SSDs, we also proposed two strategies, the strategy for performance and the strategy for scalability. GTS is fairly scalable in terms of the number of GPUs and SSDs, and so, shows a stable speedup when adding a GPU or an SSD to the machine. Through extensive experiments, we demonstrated that GTS consistently and significantly outperforms the major distributed graph processing methods, GraphX, Giraph, and PowerGraph, and the state-of-the-art GPU-based method TOTEM, across wide range of benchmarks. Especially, we demonstrated that GTS can process an RMAT32 graph within a reasonable time in a single machine that the existing distributed methods fail to process by using 30 machines of a total of about 2 TB memory.

10. ACKNOWLEDGMENTS

This work was partly supported by Institute for Information & communications Technology Promotion(IITP) grant funded by the Korea government(MSIP) (No. R0190-15-2012, High Performance Big Data Analytics Platform Performance Acceleration Technologies Development), Basic Science Research Program through the National Research Foundation of Korea(NRF) funded by the Ministry of Science, ICT & Future Planning (2014R1A1A1008227), and Samsung Research Funding Center of Samsung Electronics under Project Number SRFC-IT1401-04.

11. REFERENCES

[1] Apache Giraph. http://giraph.apache.org/, 2015.

[2] B. W. Barrett, J. W. Berry, R. C. Murphy, and K. B. Wheeler. Implementing a portable multi-threaded graph library: The mtgl on qthreads. In *IPDPS*, pages 1–8, 2009.

[3] G. E. Blelloch, M. A. Heroux, and M. Zagha. Segmented operations for sparse matrix computation on vector multiprocessors. *Technical Report CMU-CS-93-173*, 1993.

[4] D. Chakrabarti, Y. Zhan, and C. Faloutsos. R-mat: A recursive model for graph mining. In *SDM*, volume 4, pages 442–446, 2004.

[5] CUDA Stream. http://on-demand.gputechconf.com/gtc-express/2011/presentations/StreamsAndConcurrencyWebinar.pdf, 2011.

[6] Z. Fu, M. Personick, and B. Thompson. Mapgraph: A high level api for fast development of high performance graph analytics on gpus. In *GRADES*, pages 1–6, 2014.

[7] 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 *PACT*, pages 345–354, 2012.

[8] A. Gharaibeh, T. Reza, E. Santos-Neto, L. B. Costa, S. Sal-linen, and M. Ripeanu. Efficient large-scale graph processing on hybrid cpu and gpu systems. *Technical Report arXiv:1312.3018*, 2014.

[9] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. Powergraph: distributed graph-parallel computation on natu-ral graphs. In *OSDI*, volume 12, pages 17–30, 2012.

[10] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica. Graphx: Graph processing in a distributed dataflow framework. In *OSDI*, 2014.

[11] M. Han and K. Daudjee. Giraph unchained: barrierless asynchronous parallel execution in pregel-like graph processing systems. In *PVLDB*, pages 950–961, 2015.

[12] 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 *KDD*, pages 77–85, 2013.

[13] P. Harish and P. J. Narayanan. Accelerating large graph algorithms on the gpu using cuda. In *HiPC*, pages 197–208, 2007.

[14] G. M. Hector, J. D. Ullman, and J. Widom. *Database Systems: The Complete Book*. Prentice Hall, 2008.

[15] S. Hong, S. Kim, T. Oguntebi, and K. Olukotun. Accelerating cuda graph algorithms at maximum warp. In *PPoPP*, pages 267–276, 2011.

[16] F. Khorasani, K. Vora, R. Gupta, and L. N. Bhuyan. Cusha: vertex-centric graph processing on gpus. In *HPDC*, volume 23, pages 239–252, 2014.

[17] M. Kim. Towards exploiting gpus for fast pagerank computation of large-scale networks. In *EDB*, 2013.

[18] H. Kwak, C. Lee, H. Park, and S. B. Moon. What is twitter, a social network or a news media? In *WWW*, pages 591–600, 2010.

[19] A. Kyrola, G. Blelloch, and C. Guestrin. Graphchi: large-scale graph computation on just a pc. In *OSDI*, volume 12, pages 31–46, 2012.

[20] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. Distributed graphlab: A framework for machine learning and data mining in the cloud. In *PVLDB*, pages 716–727, 2012.

[21] Y. Low, J. E. Gonzalez, A. Kyrola, D. Bickson, C. E. Guestrin, and J. Hellerstein. Graphlab: A new framework for parallel machine learning. In *arXiv:1408.2041*, 2014.

[22] 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 *SIGMOD*, pages 135–146, 2010.

[23] F. McSherry, M. Isard, and D. G. Murray. Scalability! but at what cost? In *HotOS*, pages 1–6, 2015.

[24] D. Merrill, G. Michael, and A. Grimshaw. Scalable gpu graph traversal. In *PPoPP*, pages 117–128, 2012.

[25] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: a timely dataflow system. In *SOSP*, pages 439–455, 2013.

[26] D. Nguyen, A. Lenharth, and K. Pingali. A lightweight infrastructure for graph analytics. In *SOSP*, pages 456–471, 2013.

[27] S. Pai, M. Thazhuthaveetil, and R. Govindarajan. Improving gpgpu concurrency with elastic kernels. In *ASPLOS*, pages 407–418, 2013.

[28] A. Roy, I. Mihailovic, and W. Zwaenepoel. X-stream: edge-centric graph processing using streaming partitions. In *SOSP*, pages 472–488, 2013.

[29] J. Shun and G. E. Blelloch. Ligra: a lightweight graph processing framework for shared memory. In *PPoPP*, pages 135–146, 2013.

[30] J. Shun, L. Dhulipala, and G. Blelloch. Smaller and faster: Parallel processing of compressed graphs with ligra+. In *DCC*, pages 403–412, 2015.

[31] A. Thusoo, J. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive: a warehousing solution over a map-reduce framework. In *PVLDB*, pages 1626–1629, 2009.

[32] Webspam-uk2007. http://barcelona.research.yahoo.net/webspam/datasets/uk2007, 2007.

[33] R. S. Xin, J. E. Gonzalez, M. J. Franklin, and I. Stoica. Graphx: A resilient distributed graph system on spark. In *GRADES*, 2013.

[34] Yahoo webscope. ahoo! altavista web page hyperlink connectivity graph. http://webscope.sandbox.yahoo.com, 2009.

[35] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: cluster computing with working sets. In *USENIX Conference on Hot Topics in Cloud Computing*, volume 10, 2010.

[36] J. Zhong and B. He. Medusa: Simplified graph processing on gpus. In *TPDS*, 2013.

APPENDIX

In this appendix, we present the details on GPU kernel functions of GTS for two typical graph algorithms, BFS and PageRank, used in the experiments. GTS requires two different kinds of GPU kernels for processing SPs and LPs, as explained in Section 3.4, since SPs and LPs have a little different structure. Thus, we present a total of four kernels: two kernels K_{BFS_SP} and K_{BFS_LP} for BFS and two kernels K_{PR_SP} and K_{PR_LP} for PageRank. Appendix A presents the mapping table of the slotted page format required for understanding the four GPU kernels.

A. MAPPING TABLE FROM RID TO VID

Each record ID (i.e. physical ID) in ADJLIST consists of a pair of the page ID (ADJ_PID) and the slot number (ADJ_OFF) where the corresponding vertex is located. For example, in Fig. 1, ADJLIST of v_3 in LP_1 contains r_2, which points to v_2 in SP_0, and thus consists of a pair of 0 (ADJ_PID) and 2 (ADJ_OFF). Since the graph algorithms usually require vertex IDs for traversal instead of record ID, they need a method to translate a record ID to the corresponding vertex ID. For that purpose, The slotted page format maintains a kind of mapping table from RID to VID, called RVT in main memory. Fig. 12 shows RVT for G in Fig. 1. In RVT, there exists a tuple for each slotted page, and a tuple consists of a pair of START_VID and LP_RANGE, where START_VID means the first VID in the corresponding page, and LP_RANGE means the range of large page IDs. We can easily and quickly translate RID to VID by calculating RVT[ADJ_PID].START_VID + ADJ_OFF for a given RID, i.e. (ADJ_PID, ADJ_OFF). For example, r_2 is $(0, 2)$ as explained above, and so its VID is calculated by RVT[0].START_VID + 2 = 2. That is, r_2's VID is 2. In Appendix B, we denote the VID calculated using (ADJ_PID, ADJ_OFF) as ADJ_VID.

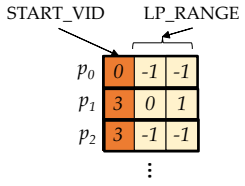


Figure 12: The RVT table for mapping RID to VID.

B. GPU KERNELS IN GTS

B.1 Kernel for BFS

We exploit the virtual warp-centric (VWC) technique [15] as a default technique for the BFS graph algorithm, where the threads in a warp process the outgoing edges of a vertex simultaneously. Algorithm 2 shows the GPU kernel K_{BFS_SP} for processing SPs. Since it is a kind of θ-join operator between topology data and attribute vector, it takes SP and LV as inputs, where LV is WA for traversal levels of vertices. It also takes a flag finished and a current traversal level curLevel, as in [15]. As the last input, it takes the local nextPIDSet maintained in each GPU, denoted as nextPIDSet_{GPU}. It is a bit vector where the number of bits is equal to the number of pages, and a bit 1 means that the corresponding page should be visited next. The overall structure of the kernel is similar with that of the kernel in [15], since both follow the same VWC technique. It first calculates the warp ID (W_ID) and the offset in the

Algorithm 2 BFS Kernel for SP

Input: SP; /* a small page */
LV; /* WA for level */
finished; /* flag for finishing traversal */
curLevel; /* current traversal level */
nextPIDSet_{GPU}; /* local nextPIDSet in GPU */

```
1: __kernel__ K_BFS_SP(SP, LV, finished, curLevel, nextPIDSet_GPU) {
2: ID ← THREAD_ID;
3: W_ID ← ID / W_SZ;
4: W_OFF ← ID % W_SZ;
5: while W_ID < SP.NUM_NODES do
6:     VID ← SP[W_ID].VID;
7:     ADJLIST_SZ ← SP[W_ID].ADJLIST_SZ;
8:     ADJLIST ← SP[W_ID].ADJLIST;
9:     if LV[VID] = curLevel then
10:        expand_warp(W_OFF, ADJLIST_SZ, ADJLIST, LV, finished,
           curLevel, nextPIDSet_GPU);
11: }
12: __device__ expand_warp(W_OFF, ADJLIST_SZ, ADJLIST, LV, finished, curLevel, nextPIDSet_GPU) {
13: for i ← W_OFF; i < ADJLIST_SZ; i ← W_SZ+i do
14:     ADJ_PID ← ADJLIST[i].PID;
15:     ADJ_OFF ← ADJLIST[i].OFF;
16:     ADJ_VID ← RVT[ADJ_PID].START_VID+ADJ_OFF;
17:     if LV[ADJ_VID] = NULL then
18:         LV[ADJ_VID] ← curLevel + 1;
19:         nextPIDSet_GPU[ADJ_PID] ← true;
20:         finished ← false;
21: __threadfence_block();
22: }
```

Algorithm 3 BFS Kernel for LP

Input: LP; /* a large page */
LV; /* WA for level */
finished; /* flag for finishing traversal */
curLevel; /* current traversal level */
nextPIDSet_{GPU}; /* local nextPIDSet in GPU */

```
1: __kernel__ K_BFS_LP(LP, LV, finished, curLevel, nextPIDSet_GPU) {
2: ID ← THREAD_ID;
3: VID ← LP.VID;
4: while ID < LP.ADJLIST_SZ do
5:     if LV[VID] = curLevel then
6:         ADJ_PID ← LP.ADJLIST[ID].PID;
7:         ADJ_OFF ← LP.ADJLIST[ID].OFF;
8:         ADJ_VID ← RVT[ADJ_PID].START_VID+ADJ_OFF;
9:         if LV[ADJ_VID] = NULL then
10:            LV[ADJ_VID] ← curLevel + 1;
11:            nextPIDSet_GPU[ADJ_PID] ← true;
12:            finished ← false;
13: }
```

warp (W_OFF) for each thread (Lines 2-4). Then, each warp processes a single vertex in SP (Lines 5-12). A warp checks the corresponding VID, ADJLIST_SZ, and ADJLIST (Lines 6-8) and then calls the expand_warp routine if the corresponding vertex should be traversed, i.e. LV[VID] is equal to curLevel (Lines 9-11).

The expand_warp routine processes all neighbor vertices in ADJLIST in a warp centric manner. That is, it processes the first W_SZ neighbor vertices, and then processes the next W_SZ neighbor vertices, and so on (Line 15). Each thread in a warp checks ADJ_PID and ADJ_OFF of the corresponding neighbor vertex (Lines 16-17) and calculates ADJ_VID (Line 18). If the corresponding neighbor vertex ADJ_VID is not visited yet (Line 19), the thread sets LV[ADJ_VID] to curLevel + 1 (Line 20) and sets the flag finished to false (Line 22). The thread also sets the bit $\text{nextPIDSet}_{\text{GPU}}$[ADJ_PID] to true (Line 21) such that the GTS framework could asynchronously copy the page ADJ_PID from main memory to GPU at the next level traversal. Finally, the routine calls CUDA's __threadfence_block() function to synchronize all of the threads within the warp (Line 29).

Algorithm 3 shows the GPU kernel $K_{\text{BFS\_LP}}$ for processing LPs. It is basically similar to $K_{\text{BFS\_SP}}$, except that multiple warps for a large page processes ADJLIST of the page together. Since a single warp does not need to process an entire ADJLIST of a vertex, there is no loop like Line 15 in the $K_{\text{BFS\_SP}}$ kernel. Instead, each thread performs the body of the expand_warp routine of $K_{\text{BFS\_SP}}$ directly (Lines 6-13) if the current vertex should be traversed, i.e. LV[VID] is equal to curLevel (Line 5). In $K_{\text{BFS\_LP}}$, since there is no calling of the warp-level routine like expand_warp in the kernel, there is no warp-level synchronization like __threadfence_block().

B.2 Kernel for PageRank

We exploit the edge-based method proposed in [17] in addition to the VWC technique for the PageRank graph algorithm. In the edge-based method, a GPU thread takes responsibility for partial updating the PageRank value of the source (or destination) node of an edge. Algorithm 4 shows the GPU kernel $K_{\text{PR\_SP}}$ for processing SPs. Since it is also a kind of θ-join operator between topology data and attribute vector, it takes SP and two required attribute vectors nextPR and prevPR as inputs. Since nextPR is WA, the kernel requires the entire nextPR. In contrast, prevPR is RA, and so the kernel only requires the partial prevPR corresponding to the given SP as explained in Section 3.1. We denote such a subvector of prevPR as prevPR[v:w]. Before calling the kernel $K_{\text{PR\_SP}}$, each element of nextPR is initialized to $\frac{1-df}{|V|}$ in main memory, where df means the damping factor. Thus, the kernel only needs to add the value of the remaining part of the PageRank equation to nextPR. The outline of $K_{\text{PR\_SP}}$ is similar to that of $K_{\text{BFS\_SP}}$ (Lines 2-8). A warp calls the expand_warp routine in order to update the PageRank values for the outgoing edges from VID (Line 9). The expand_warp routine processes all neighbor vertices in ADJLIST in a warp centric manner, as in $K_{\text{BFS\_SP}}$. Each thread in a warp calculates ADJ_VID of the corresponding neighbor vertex as in A.2 (Line 16) and calculates the partial PageRank value for ADJ_VID with considering the edge between VID and ADJ_VID (Line 17). The calculated value is added to nextPR[ADJ_VID]. Here, since multiple GPU threads could update nextPR[ADJ_VID] simultaneously, we should use the atomicAdd operator to avoid a race condition. CUDA supports several atomic operators including atomicAdd. At the end of the routine, the synchronization function is called as in $K_{\text{BFS\_SP}}$ (Line 19).

Algorithm 5 shows the GPU kernel $K_{\text{PR\_LP}}$ for processing LPs. It is basically similar to $K_{\text{PR\_SP}}$, except that multiple warps for a large page processes ADJLIST of the page together. As in $K_{\text{BFS\_LP}}$, each thread performs the body of the expand_warp routine of $K_{\text{PR\_SP}}$ directly (Lines 4-7).

Algorithm 4 PageRank Kernel for SP

Input: SP; /* a small page (for vertices [v:w]) */
nextPR; /* WA */
prevPR[v:w]; /* subvector of RA */

```
1: __kernel__ K_PR_SP(SP, nexPR, prevPR[v:w]) {
2: ID ← THREAD_ID;
3: W_ID ← ID / W_SZ;
4: W_OFF ← ID % W_SZ;
5: while W_ID < SP.NUM_NODES do
6:    VID ← SP[W_ID].VID;
7:    ADJLIST_SZ ← SP[W_ID].ADJLIST_SZ;
8:    ADJLIST ← SP[W_ID].ADJLIST;
9:    expand_warp(W_OFF, ADJLIST_SZ, ADJLIST, VID, nextPR, prevPR[v:w]);
10: }
11: __device__ expand_warp(W_OFF, ADJLIST_SZ, ADJLIST, VID, nextPR, prevPR[v:w]) {
12: for i ← W_OFF; i < ADJLIST_SZ; i ← W_SZ+i do
13:    ADJ_PID ← ADJLIST[i].PID;
14:    ADJ_OFF ← ADJLIST[i].OFF;
15:    ADJ_VID ← RVT[ADJ_PID].START_VID+ADJ_OFF;
16:    atomicAdd(nextPR[ADJ_VID], df * prevPR[VID] / ADJLIST_SZ);
17: __threadfence_block();
18: }
```

Algorithm 5 PageRank Kernel for LP

Input: LP; /* a large page (for vertex v) */
nextPR; /* WA */
prevPR[v]; /* subvector of v */

```
1: __kernel__ K_PR_LP(LP, nexPR, prevPR[v]) {
2: ID ← THREAD_ID;
3: while ID < LP.ADJLIST_SZ do
4:    ADJ_PID ← LP.ADJLIST[ID].PID;
5:    ADJ_OFF ← LP.ADJLIST[ID].OFF;
6:    ADJ_VID ← RVT[ADJ_PID].START_VID+ADJ_OFF;
7:    atomicAdd(nextPR[ADJ_VID], df * prevPR[v] / v.ADJLIST_S);
8: }
```

C. OPTIONS OF TOTEM

Table 5 shows the ratios of graph data processed by GPUs to that by CPUs in TOTEM (GPU%:CPU%) when following the options recommended by the authors. In general, as the size of a graph increases, the size of the GPU partition decreases. However, it is not for RMAT29 since the mapped memory options of TOTEM allocates a part of the GPU partition as mapped memory [8].

D. ADDITIONAL GRAPH ALGORITHMS

In addition to BFS and PageRank, for a wider range of benchmarks, we implement the following three additional graph algorithms using GTS: Single-Source Shortest Path (SSSP), Connected Components (CC), and Betweenness Centrality (BC). It demonstrates the adaptability of GTS. We select those three graph algorithms since Giraph, GraphX, PowerGraph, and TOTEM commonly support them. Figure 13 shows the comparison results among five methods (BC between two methods). GTS significantly outperforms other four methods for SSSP and CC, and also largely outperforms TOTEM for CC. Here, we perform the experiments of BC using the default mode, i.e., the single node mode for both methods.

Table 5: Ratios of partition sizes in TOTEM (GPU%:CPU%).

data	one GPU		two GPUs	
	BFS	PageRank	BFS	PageRank
RMAT27	65:35	60:40	80:20	80:20
RMAT28	15:85	60:40	40:60	80:20
RMAT29	50:50	15:85	75:25	30:70
Twitter	50:50	80:20	75:25	85:15
UK2007	35:65	30:70	70:30	60:40
YahooWeb	10:90	15:85	N/A	N/A

E. MICRO-LEVEL PARALLEL PROCESSING

Figure 14 shows the performance of GTS for BFS and PageRank while changing the density (i.e., #vertices : #edges) of RMAT28 from 1:4 to 1:32 and changing a micro-level parallel processing technique for each slotted page. The three techniques discussed in Section 6.2 show similar performance for very sparse graph of 1:4. However, for denser graphs, the edge-centric strategy outperforms the vertex-centric strategy largely. The hybrid strategy improves the performance slightly (up to 6% for BFS and up to 24% for PageRank) compared with the edge-centric one.

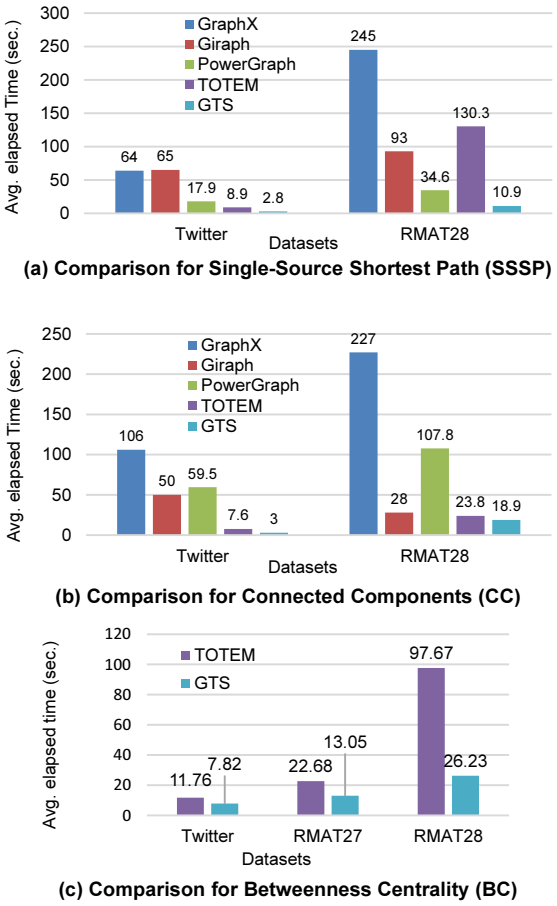


Figure 13: Comparison for additional graph algorithms: SSSP, CC, and BC.

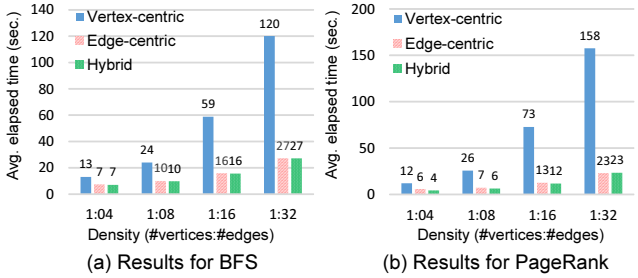


Figure 14: Performance when changing micro-level parallel processing techniques and graph density.