Master's Thesis
석사 학위논문

A large-scale graph processing method for multi-attribute graphs using GPUs and SSDs

Kyuhyon An (안 규 현 安 圭 鉉)

Department of Information and Communication Engineering
정보통신융합공학전공

DGIST

2017

Master's Thesis
석사 학위논문

A large-scale graph processing method for multi-attribute graphs using GPUs and SSDs

Kyuhyon An (안 규 현 安 圭 鉉)

Department of Information and Communication Engineering
정보통신융합공학전공

DGIST

2017

A large-scale graph processing method for multi-attribute graphs using GPUs and SSDs

Advisor : Professor Min-Soo Kim

Co-advisor : Professor Daehee Hwang

by

Kyuhyon An
Department of Information and Communication Engineering
DGIST

A thesis submitted to the faculty of DGIST in partial fulfillment of the requirements for the degree of ［Master of Science］ in the ［Department of Information and Communication Engineering］. The study was conducted in accordance with Code of Research Ethics[1]

. .

Approved by

Professor Min-Soo Kim (Signature)
(Advisor)

Professor Daehee Hwang (Signature)
(Co-Advisor)

[1] Declaration of Ethical Conduct in Research: I, as a graduate student of KAIST, hereby declare that I have not committed any acts that may damage the credibility of my research. These include, but are not limited to: falsification, thesis written by someone else, distortion of research findings or plagiarism. I affirm that my thesis contains honest conclusions based on my own careful research under the guidance of my thesis advisor.

A large-scale graph processing method for multi-attribute graphs using GPUs and SSDs

Kyuhyon An

Accepted in partial fulfillment of the requirements for the degree of [Master of Science]

. .

Head of Committee 김 민 수 (인)
Prof. Min-Soo Kim

Committee Member 황 대 희 (인)
Prof. Daehee Hwang

Committee Member 최 지 환 (인)
Prof. Jihwan Choi

Degree
201522013

안 규 현. Kyuhyoen An. A large-scale graph processing method for multi-attribute graphs using GPUs and SSDs. Department of Information and Communication Engineering. 2016. 39p. Advisors Prof. Min-Soo Kim, Prof. Co-Advisors Daehee Hwang.

Abstract

The one of characteristics is simpeness of structure that provides facile data analysis. The importance of graph processing is growing due to that. However the handling on large-scale graph using existing methods becomes harder due to the increase of data size. The distributed graph system requires a lot of machine for scalability. However network overhead increases linearly and is larger than processing time. Meanwhile GPU is used as coprocessor for handling graph data via thousands of GPU cores. However all of existing methods using GPU fail to process large-scale graphs due to lack of main memory.

In this paper, we propose the method GStream 2.0 that handles large-scale graphs which is larger than main memory using a single machine. Proposed method is streaming the graph data from secondary storage to GPU memory and launches graph algorithm on GPU device. Through streaming graph data via slotted page format which is partitioning graph data with fixed size, proposed method handles large-scale graph which is larger than GPU device memory. Performance is improved by avoiding intermediate data transfer and proposed method is using the array for result of algorithm which is sorted in GPU device memory. For supporting graph algorithm based on edge attribute, we extend slotted page format. Finally, we propose the extended model for 2-hop neighborhood algorithm.

Through experiments, we show that GStream 2.0 significantly outperforms the major methods and the state-of-the-art methods.

Keywords: Graph processing, GPUs, SSDs, Stream, Parallel programming

Contents

Lists of figures

Lists of tables

Lists of algorithms

1. INTRODUCTION

The one of real-world data analysis is graph processing, such as shortest path finding, propagation simulation, and correlation analysis. The graph processing is becoming major method in wide research area, due to generality of modeling. The graph processing methods and systems is required scalability because the size of graphs modeling real-world object increases fast with development of equipment and IoT era.

As single machine could not handle large-scale graphs, distributed system for graph processing have been researched. One of graph processing model is Bulk-Synchronous Parallel (BSP) message passing model, such as Apache Giraph [1, 11], Google's Pregel [22]. In BSP message passing model, an execution of all vertex kernels iterates via supersteps. Each vertex kernel in superstep receives all messages from the previous superstep and sends the result of the current superstep to adjacency of vertex in the next superstep. Another of graph processing model is Gather-Apply-Scatter (GAS) model, such as Apache Spark GraphX [10, 33, 35], PowerGraph [9, 20, 21]. In GAS model, Gather phase gathers the data for graph algorithm each vertex with their neighborhood. Apply phase stores the result of each vertex with Gather phase data. Scatter phase updates adjacent edge and vertices. These model handle large-scale graph using a lot of CPU and main memory with high bandwidth network.

Graph processing with coprocessor also have been proposed [7, 8]. Typical coprocessor is GPU due to high computing power with thousands of core. GPU as coprocessor is widely used to high computing via massively parallel model. For large number of core, GPU has huge memory bandwidth. Only a few vertices is calculated or traversed at a time with conventional CPU, however a large number of vertices is calculated or traversed at a time with GPU due to above characteristics of GPU. Considering the size of real-world graph data is growing up to million or billion vertices, graph processing with GPU is proposed within several years.

Existing methods with GPU outperform those with CPU, however they have major problem. Most of them

convert from in-memory processing using CPU to in-memory processing GPU. They can handle only graph which fit in the GPU device memory, however many real graphs do not fit in it. Previous proposed methods for graph processing with GPU concentrate only performance of their method without considering the size of graph. Only one method solve the lack of GPU device memory, TOTEM [7, 8]. For the problem, TOTEM partitions a graph data into two part, one part in main memory, and the other in GPU device memory. GPU process only the part in GPU device memory, and CPU process only the part in main memory. TOTEM can handle graph even if it is not fit in GPU device memory, however partition of data brings many problems such as underutilization of the computation power of GPU, partitioning overhead, message passing overhead between main memory and GPU. Moreover, TOTEM cannot handle large-scale graph which is larger than main memory due to in-memory processing.

We propose a fast and scalable disk-based graph processing method using GPUs, GStream 2.0. GStream 2.0 can handle RMAT32 (64 billion edges) graph which is failed to handle with other in-memory methods. In GStream 2.0, the total graph is processed by GPU without underutilization of the computing power of GPUs. Furthermore the partitioning of graph is not required. To overcome lack of GPU device memory and main memory, we propose a concept of storing only updatable attribute data and moving topology data. In proposed method, the entire graph data is stored in secondary storage such as SSD, the requested graph data is streaming to GPU device memory, a graph algorithm is launched using GPU. For streaming topology, GStream 2.0 adopts the slotted page format that contains topology data with fixed-size page. GStream 2.0 has three phase. In first phase, a chunk of result data is copied to GPU device memory. In second phase, required topology data is copied to GPU device memory with streaming and user-defined GPU kernel function is launched. Finally, result data is copied back to main memory. For hiding overhead from copying data, required topology data is concurrently copied via asynchronous GPU stream even user-defined GPU kernel function is being launched. For exploiting multiple GPU, we also propose two strategies. The first strategy is for performance and the other strategy is for scalability. We extend the slotted page format for the graph algorithm based on edge attribute. We also extend graph processing

model for 2-hop neighborhood algorithm. With no communication overhead and fully exploiting the GPU cores, GStream 2.0 can achieve higher performance compared with the existing methods. Moreover, with no data duplication from graph partition and storing data on SSDs, GStream 2.0 can achieve higher scalability compared with the existing methods. GStream 2.0 shows a stable scalability with increasing a GPU or a SSDs.

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 GStream 2.0 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 propose two strategies that can improve the performance or the scalability with exploiting multiple GPUs and multiple storages.
- We extend data format for edge attribute and processing model for 2-hop neighborhood algorithm.
- Through extensive experiments, we demonstrate that GStream 2.0 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 GStream 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 GStream 2.0. In Section 3, we propose the main concept of GStream 2.0. In Section 4, we propose two strategies for exploiting

multiple GPUs and storages. In Section 5, we present extended data format for edge attribute. In Section 6, we present extended processing model for 2-hop neighborhood algorithm. Section 7 presents the results of experimental evaluation, and Section 8 concludes this paper.

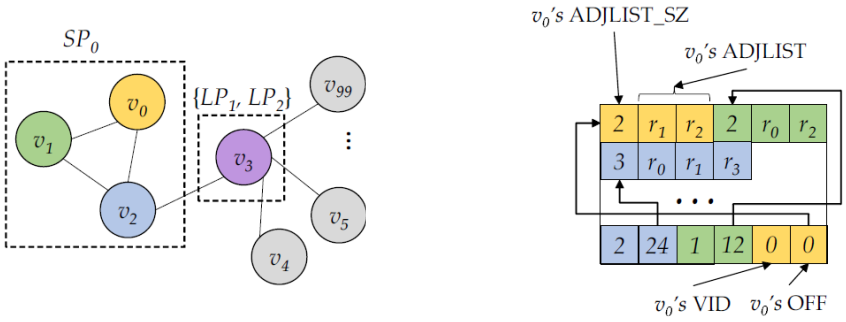
2. PRELIMINARIES

In this section, we explain the data formats in GStream 2.0. Since the limit of memory and the compression for sparse graph, the various type of in-memory format for a sparse graph have been proposed, such as Diagonal (DIA) [13], ELLPACK(ELL) [2], Compressed Sparse Row(CSR), Compressed Sparse Column (CSC), and Coordinate list (COO) [3]. They usually require a contiguous memory and it comes a limit of graph processing.

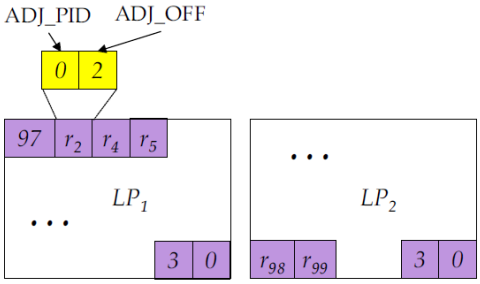
Besides, there is an external memory (i.e., out-of-core) graph format called the *slotted page format* [12]. In slotted page format, a graph is represented in a set of fixed-size slotted pages. In a slotted page, there are two parts, records and slots. The records are located from the start of the page. The slots are located from the end of the page. A record represent edges of a vertex and consist of the size of the adjacency list, denoted as ADJLIST_SZ, and the adjacency list of a vertex, denoted as ADJLIST. A slot represent vertex and consist of a vertex ID, denoted as VID, and the offset of matching record, denoted as OFF. An edge in adjacency list consist of physical ID of destination. A physical ID consist of two parts, the page ID (2-byte), denoted as ADJ_PID, and the slot number (2-byte), denoted as ADJ_OFF. The page ID is id of page which contains the slot of destination vertex. The slot number is offset of destination vertex slot. With physical ID, graph algorithms can easily access to the physical locations of neighbor vertices. This concept of physical ID is commonly used for performance in database area [14].

Figure 1 shows an example of slotted page with example graph G. In Figure 1(a), the v_0, v_1, and v_2 have a small number of edges, while v_3 has a large number of edges. In real graphs, node degree distribution follows power-law distribution, few of high-degree vertices and a lot of low-degree vertices. Figure 1(b) shows that storing of low-degree vertices in a single page SP_0, which is called a Small Page (SP). In case a high-degree vertex cannot be stored in a single page, a high-degree vertex is stored in multiple pages $\{LP_1, LP_2\}$ which are called Large

Pages (LP) in figure 1(c).



(a) Graph G **(b) Small Page SP_0**



(c) Large Pages $\{LP_1, LP_2\}$ for v_3

Figure 1. Example of Graph G and the slotted page of G

3. MAIN CONCEPT OF GSTREAM 2.0

In this section, we present the proposed method GStream 2.0. Section 3.1 explains the main concept of GStream 2.0, and Section 3.2 describes the streaming scheme of GStream 2.0 in detail. Section 3.3 explains the major types of graph algorithms, and Section 3.4 shows the algorithm of the GStream framework.

3.1 Main concept of GStream 2.0

In graph algorithms, there are two type of data, graph topology data (shortly, topology data) and attribute data for vertices or edges. For example, Breadth-First Search (BFS) requires topology data and an attribute data for levels of vertices (shortly, LV). Furthermore, PageRank requires topology data and two attribute data, one of them is data for the previous PageRank values (shortly, prevPR) and the other is data for the next PageRank values (shortly, nextPR). The attribute vectors consist of two type of vector, read-only vector and read/write vector. For example, LV, the attribute vector of BFS is the read/write vector. For PageRank, prevPR is a read-only attribute and nextPR is a read/write attribute.

The concept of most of the existing graph processing methods [9, 20, 22] is that storing topology data in main memory and passing attribute data among machines. Since the size of transferred attribute data is smaller than topology data, this concept is fitted for distributed shared-memory system with a lot of main memory and low bandwidth network. However that concept is not fitted for graph processing with GPU, since GPU is connected with main memory through high-speed interface and has the small device memory.

Here, we propose a concept of storing only updatable attribute data and moving topology data. The attribute data is stored limited device memory, and topology data is moved via a high-speed interface. Nevertheless the size of transferred topology data is larger than attribute data, the overhead of moving topology data is hidden by

concurrent process of moving data and graph algorithm. The graph algorithm is described as user-defined GPU kernel function and is launched using thousands of GPU cores.

GStream 2.0 has three phase for graph processing. First, GStream 2.0 copies read/write attribute data to device memory. Second, GStream 2.0 processes graph algorithm for read-only attribute data and topology data with streaming to device memory. Finally, read/write attribute data is copied back to main memory for synchronization. Each attribute is conceptually divided into multiple subvectors, and topology data consists of multiple fixed-size unit using the slotted page format. For example, prevPR for PageRank can be divided into multiple chunk. Each chunk of prevPR matches a slotted page based on range of vertexID. GStream 2.0 requires GPU device memory at least for attribute data and a single slotted page. In general, the size of attribute data is larger than the size of a single slotted page, it is required to reduce the size of attribute data for large-scale graph processing. The read-only attribute is not required keep in device memory since read-only attribute is not updated. Only the read/write attribute have to be stored in device memory. For reducing the size of attribute data, only read/write attribute is stored in device memory. For example, nextPR for PageRank is stored in device memory during graph processing, however each subvector of prevPR is discard after end of kernel function for each matched a slotted page.

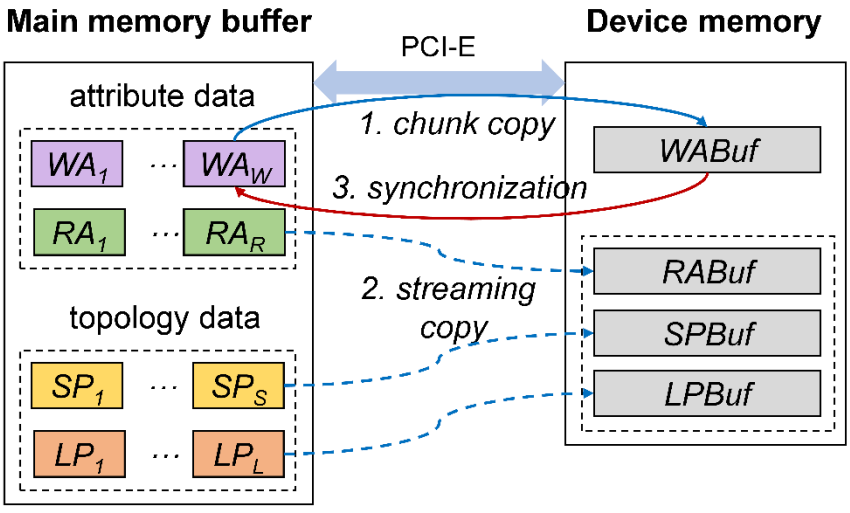


Figure 2. The data flow of GStream 2.0

Figure 2 describes for the data flow of GStream 2.0. We suppose read/write attribute (shortly, WA) is divided into *W* chunk, and read-only attribute (shortly, RA) is divided into *R* subvectors. We also suppose the numbers of small pages and large pages are *S* and *L*. The number of RA subvetors is equal to *S*. RA_i is matched SP_i. The three step in Figure 2 represents the data flow of three phase which is mentioned previous paragraph.

3.2 Asynchronous multiple streaming

For hiding overhead of topology copy, GStream 2.0 copies topology data from main memory to GPU device memory using asynchronous multiple streaming. For streaming, it is required to divide fixed-size unit, and GStream 2.0 adopts the slotted page format, one of graph format consist of fixed-size unit. In figure 2, there is three buffers for streaming RA, SP, and LP in device memory, RABuf for RA to device memory, SPBuf for SP, LPBuf for LP. WABuf in device memory is chunk buffer for WA.

GStream 2.0 exploits multiple GPU streams for streaming. Figure 3 shows the timeline of streams in GStream 2.0. First, WA is copied to WABuf. Then, each stream performs sequence of operations: (1) copying SP_i (or LP_i) to SPBuf (or LPBuf), (2) copying RA_i to RABuf, and (3) executing user-defined kernel function for graph processing. Only there is one copy operation to device memory at a time due to hardware specification [5], however kernel execution can overlap a copy operation or other kernel execution [5, 27]. The ratio between the copy time of data (SP_i, RA_i) each kernel function and the execution time of kernel function, determines the number of streams *K*. If a single kernel execution in $stream_i$ ends before data copy of the others, $stream_i$ is idle until ready to data copy. In contrast if a single kernel execution in $stream_i$ ends after data copy of the others, $stream_i$ starts next data copy immediately after kernel execution. In the former case, the performance does not improve with the increase of number of stream. In the latter case, the performance improves with the increase of number of stream. Table 1 shows the ratios of the transfer time to the kernel execution time for BFS and PageRank on three real data set used in experimental evaluation. BFS has relatively high ratios due to random access of vertex in a slotted

page, while PageRank has relatively low ratios due to full access of vertex in a slotted page. Thus the number of stream *k*, depends on graph algorithms. The maximum number of stream is 32, in current CUDA technology [5].

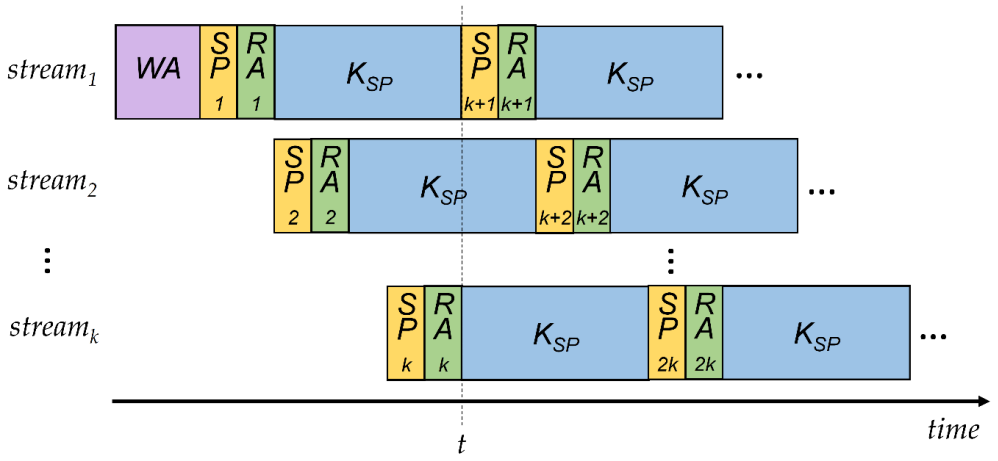
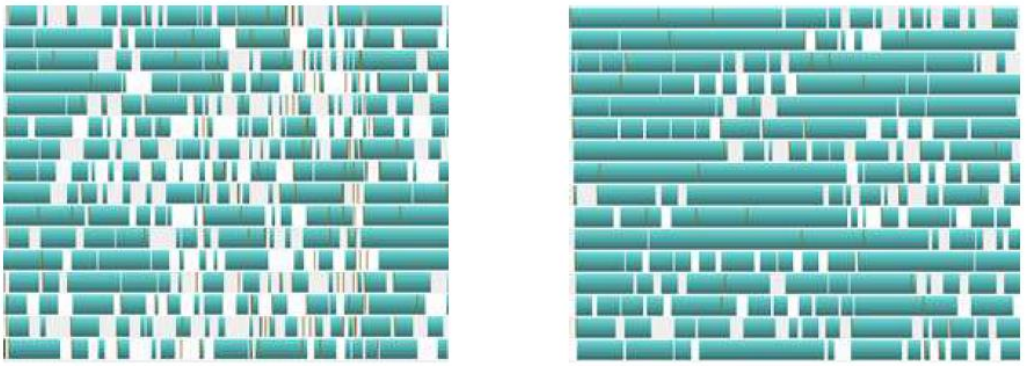


Figure 3. The time line of streams.

Table 1. The ratio of the transfer time to kernel execution time for BFS and PageRank on real data set.

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

There is kernel switching overhead among SPs and LPs due to difference of execution flow between SP and LP. For reducing overhead, GStream 2.0 divide two part of processing, one of them is to process for all of SP and the other is to process for all of LP. First, GStream 2.0 performs processing for all of SP. The processing for all of LP is started after all of processing for SP. After processing both of data, the updated WA is copied back to main memory. Figure 4 shows the actual timelines of copy operation and kernel function for BFS and PageRank when using 16 streams on a synthetic data. In Figure 4, the very short red colored bars before the long green colored bar mean copy operation for a slotted page and RA to device memory. The long green colored bars mean execution of a kernel function. The timeline for PageRank in Figure 4(b) is denser than that for BFS in Figure 4(a) due to difference of the number of access vertex between both algorithms.



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

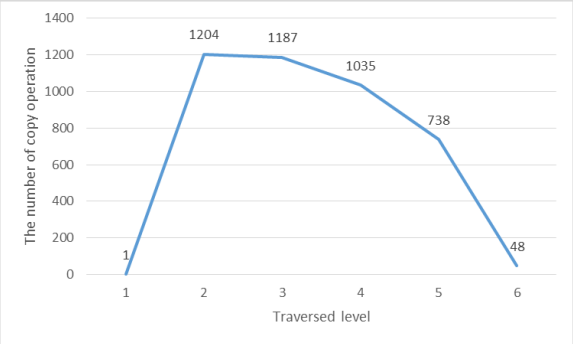
Figure 4. The actual timelines of copy operation and kernel function for BFS and PageRank.

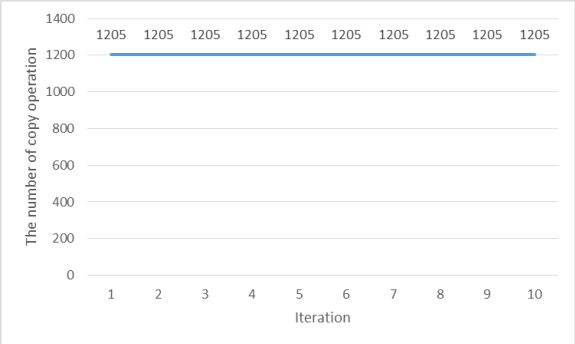
3.3 Major types of graph algorithms

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 usually tend less computationally intensive, but lead to random memory accesses due to the irregular structure of graphs. There are 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 usually tend computationally intensive, and require the linear scan of whole graph in many cases. They include PageRank, degree distribution, RandomWalk 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.

Within a single iteration of PageRank-like algorithm, the entire topology is required and copied to device memory. However, within a single level traversal of BFS-like algorithm, the part of topology is required and copied to device memory. Figure 5 shows the number of copy operation for BFS and PageRank on synthetic graph. The number of page in used graph is 1205. BFS-like algorithm requires variable size of copy operation in Figure

5 (a). PageRank-like algorithm requires whole graph each iteration in Figure 5 (b).





(a) For BFS algorithm. **(b) For PageRank algorithm.**

Figure 5. The number of copy operation for BFS and PageRank on synthetic graph.

For copy of required page on BFS-like algorithm, GStream 2.0 uses the data structure called nextPIDSet that contains the IDs of pages to be copied at the next level. The nextPIDSet is updated by traversal kernel function for BFS-like algorithm and is copied back to main memory after the execution of kernel function is done. At the next level, the set of pages in the nextPIDSet is copied to device memory.

At the beginning of next level, copied topology data at previous level still remains in device memory. For the remained topology data, it can be processed without copy from main memory. The performance of GStream 2.0 can increase, since the size of copied data is reduced by reusing remained topology data. GStream 2.0 manages topology streaming buffer, SPBuf and LPBuf as cache for reusing. GStream 2.0 allocates both buffer as large as possible to exploit cache. We suppose the size of cache is B, the naïve cache hit rate is $B / (S + L)$ for BFS-like algorithm. GStream 2.0 adopts the LRU algorithm as cache algorithm.

3.4 Algorithm of the framework

In this section, we present the algorithm of the GStream 2.0 framework. Algorithm 1 presents the pseudo code of the framework. K_{SP} and K_{LP} are a user-defined kernel function for a specific graph algorithm on a graph G. K_{SP} is a kernel for SPs and K_{LP} is a kernel for LPs. Both kernel function have to be given due to difference of structure

between SP and LP. As an initialization step, GStream 2.0 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, depending on the type of the graph algorithm. If the graph algorithm is of BFS-like, the page ID containing 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). MMBuf means main memory buffer used for fetching the slotted pages from SSDs to main memory. If the size of graph G is smaller than the size of MMBuf, then the entire graph topology is loaded to MMBuf.

Algorithm1. The framework of GStream 2.0

```
Input:      Graph G, /* input graph */
            K_SP, /* GPU kernel of Q for small pages */
            K_LP, /* GPU kernel of Q for large pages */
Variables:  nextPIDSet, /* set of page IDs to process next */
            cachedPIDMap1:N ; /* cached page IDs in GPU1:N */
            bufferPIDMap; /* buffered page IDs in MMBuf */
 1:  /* Initialization */
 2:  Create SPStream and LPStream for GPU1:N;
 3:  Allocate WABuf, RABuf, SPBuf, LPBuf for GPU1:N;
 4:  if Q is BFS-like then
 5:      nextPIDSet ← page ID containing start vertex;
 6:  else
 7:      nextPIDSet ← ALL_PAGES;
 8:  cachedPIDMap1:N ← ∅
 9:  if |G| < MMBuf then
10:      Load G into MMBuf;
11:  Copy WA to WABuf of GPU1:N;
12:  /* Processing GPU kernel */
```

```
13:  repeat
14:  /* repeat Lines 15-31 for LPs */
15:      for j ∈ nextPIDSet.SP do
16:          if j ∈ cachedPIDMap_h(j) then
17:              Call K_SP for SP_j in GPU_h(j);
18:          else if j ∈ 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_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_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 ← ∪_1≤i≤N nextPIDSet_i ;
31:  until nextPIDSet = ALL_PAGES ∨ nextPIDSet = ∅;
```

The *repeat-until* loop (Lines 13-31) 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 after small pages processing. 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 hash function, $h(j)$, which will be explained in Section 4. Before transferring the page, GStream 2.0 first checks if the page already exists in the cache of $GPU_{h(j)}$ by looking up $cachedPIDMap_{h(j)}$ (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. If the page is not in MMBuf, GStream 2.0 fetches pages from $SSD_{g(j)}$ to MMBuf. 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 nextPID-Set and cachedPIDMap$_{1:N}$ are actually not used. 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. THE STRATEGY OF GSTREAM 2.0

In this section, we present two strategies for exploiting multiple GPUs. GStream 2.0 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. Section 4.2 presents the strategy for high scalability with a limit on performance.

4.1 Strategy for performance

The first strategy of GStream 2.0, strategy-P is using multiple GPUs for performance. The performance of this strategy increase *N* times than the performance with a single GPU, since the topology data is divided each GPU. First, GStream 2.0 is copying the same attribute data, especially WA, to all GPUs and then copying a different topology data to each GPU. Figure 6 shows the data flow scheme of that strategy, which consists in four different steps. In Step 1, GStream 2.0 copies the same WA to all $\{GPU_1, \cdots, GPU_N\}$. In Step 2, it copies different topology data and read-only data each GPU. GStream 2.0 executes a given GPU kernel K_{SP} for a different $< SP_k, RA_k >$ to each GPU_k for $1 \leq k \leq N$. In general, it copies $< SP_{i*N+j}, RA_{i*N+j} >$ to GPU_j for $0 \leq i \leq \lceil \frac{S}{N} \rceil - 1$ and $1 \leq j \leq N$. For the processing of LP is similar to the processing of SP. Each GPU can execute the same GPU kernel function independently for a different part of topology data. The synchronization between each device memory is required, since the result of GPU kernel function in each GPU is different the result in each other. Step 3 presents the data synchronization of WA between each device memory and Step 4 presents the data synchronization of WA between device memory and main memory. In Step 3, the naïve synchronization method is following: (1) repeat to copy back WA in a single GPU device memory to main memory and merge copied WA to stored WA in main memory until all of GPU done and (2) copy updated WA in main memory to each GPU device memory. However the synchronization overhead becomes since the amount of copied data grow up *N* times. For avoiding it, GStream

2.0 performs Step 3 using peer-to-peer memory copying among GPUs, which is faster than copy 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). In Step 4, the updated WA data in a master GPU is copied to main memory. In Algorithm 1, Line 11 describes Step 1, Lines 16-26 describe Step 2, and Line 28 describes both of synchronization step, Step 3 and Step 4.

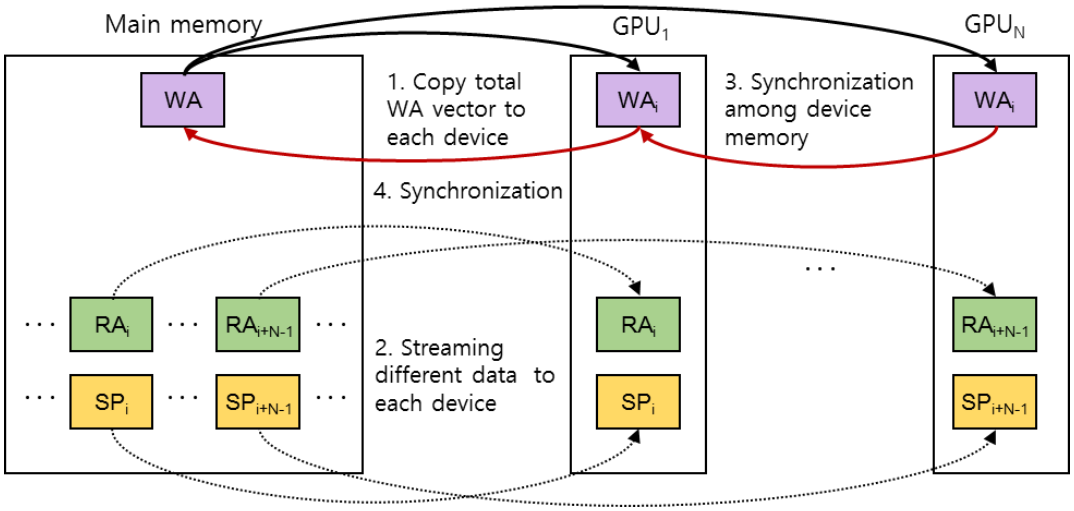


Figure 6. Strategy for performance exploiting multiple GPUs.

Since the workload is divided by the number of GPUs, this strategy can achieve linear speedup. In Step 2, the function $h(x)$ returns a hash value for input x. The page ID j of a page is used for selecting a GPU device to streaming and process, by return value of $h(j)$. GStream 2.0 uses the mod function for the default hash function.

In GStream 2.0, topology data is stored in SSDs and topology data is moved from SSDs to GPU memory via main memory. The bandwidth of data streaming is bounded on a slower one between the speed of PCI-E interface and I/O performance of SSDs. The bandwidth of data streaming determine the amount of data streaming per second, and the amount of data streaming per second has positive linear relationship with the performance of this strategy. Under the current computer architecture, the I/O speed of SSDs (e.g., about up to 2GB/sec) is much slower than that of PCI-E interface (e.g., 16GB/sec). Therefore the performance of this strategy is bounded on the I/O speed of SSDs. To relieve the limitation of this strategy, GStream 2.0 exploits multiple SSDs. Each slotted

page SP_j of graph G ($1 \leq j \leq S$), is stored in $SSD_{g(j)}$. The function $g(j)$ returns a hash value for a page ID j using mod function. GStream 2.0 fetches required page from corresponding SSDs concurrently.

Although this strategy shows high and stable performance of linear speedup with multiple GPUs, there is a major problem from GPU's limited device memory. In this strategy, GStream 2.0 cannot handle in case that the size of WA is larger than the size of single GPU's device memory. For example, this strategy can process the PageRank algorithm only up to RMAT30 using a GPU having 6GB device memory.

4.2 Strategy for scalability

The second strategy of GStream 2.0, strategy-S is using multiple GPUs for scalability. The scalability of this strategy increase N times than the scalability with a single GPU, since attirbute data is divided each GPU. First, GStream 2.0 is copying the different chunk of attribute data, especially WA_i, each GPU and then copying a same topology data to all GPUs. Figure 7 shows the data flow scheme of that strategy, which consists in three different steps. In Step 1, GStream 2.0 copies a different WA_i to each GPU_i for $1 \leq i \leq N$. In Step 2, it copies a same topology data and read-only data each GPU. GStream 2.0 executes a given GPU kernel K_{SP} for the same < SP_k, RA_k > to all of GPUs. For the processing of LP is similar to the processing of SP. Each GPU can execute the same GPU kernel function independently for a different part of attirbute data. The synchronization between each device memory is not required, since each GPU updates only for own chunk of WA. Step 3 presents the data synchronization of WA between device memory and main memory. For Step 3, GStream 2.0 uses the naïve synchronization method is following: (1) repeat to copy back WA in a single GPU device memory to main memory and merge copied WA to stored WA in main memory until all of GPU done and (2) copy updated WA in main memory to each GPU device memory.

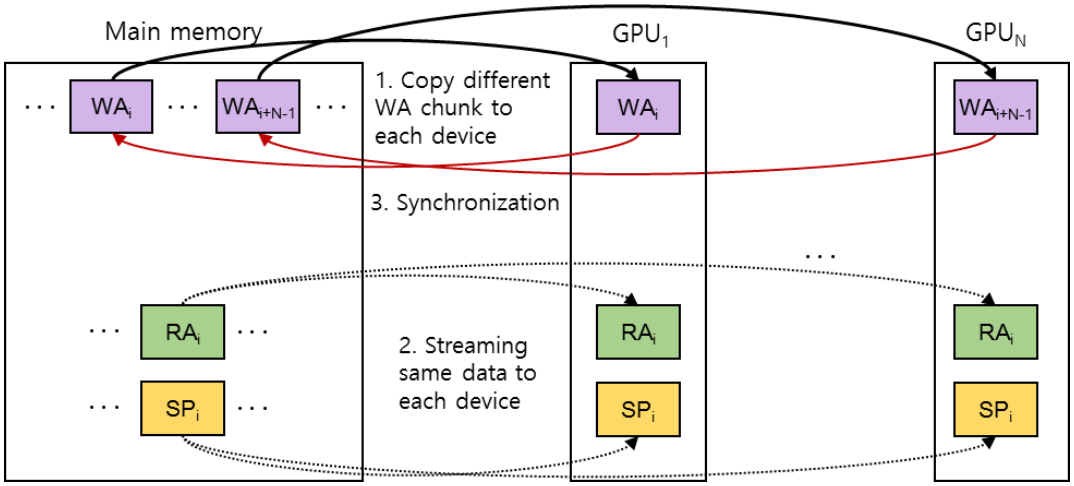


Figure 7. Strategy for scalability exploiting multiple GPUs.

In Algorithm 1 as in section 4.1, Line 11 describes Step 1, Lines 16-26 describe Step 2, and Line 28 describes synchronization step, Step 3. 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 for streaming the page to all GPUs.

This strategy maximizes the size of a graph to process using multiple GPUs. Especially, it can achieve linear scalability of the size of a graph to process. In this strategy, the workload of each GPU is same due to streaming same data to all of GPUs, and the workload of each GPU is same as the workload of a single GPU. 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. In this strategy, there is also performance boundary due to I/O speed of SSDs and the speed of PCI-E interface. In strategy-P, the logical speed of PCI-E interface for all of GPUs has linear speedup. In contrast, in strategy-S, the logical speed of PCI-E interface for all of GPUs is fixed to the speed of PCI-E interface for a single GPU device. The gap between the I/O performance of SSDs and the logical speed of PCI-E interface for strategy-S, is less than it for strategy-P. That means the overall performance of this strategy would not increase much even though processing an entire graph in main memory.

Therefore, the strategy for scalability of GStream 2.0 is suitable to process a large-scale graph in case that 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 6GB GPUs and 500 GB SSDs). On the contrary, the strategy for performance of GStream 2.0(in

Section 4.1) is suitable to process a small-scale graph in case that 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 6GB GPUs and 128GB main memory).

5. EXTENDED SLOTTED PAGE FORMAT

In this section, we present extended data format. Section 5.1 presents extended slotted page format for trillion-scale graph. Section 5.2 presents extended slotted page format for edge attributes.

5.1 Extended slotted page format for trillion-scale graph

The slotted page format [12], is useful for representing a graph topology data for secondary storage. However there is problem of maximum size for representing graph. The physical ID of 4-byte (2 bytes for page ID and 2bytes for slot number) can theoretically represent a graph of up to 2^{32} = 4 billion vertices. However, if the number of page ID overflows the maximum number of byte for page ID, then the graph cannot be represent by the slotted page format. For example, an RMAT30 graph which has page ID over the maximum number of 2 bytes of 1 billion vertices and 16 billion edges, it cannot be represented by slotted page format.

Thus for handle even a trillion-scale graph, we generalize the existing format such that p-byte page ID (ADJ_PID) and q-byte slot number (ADJ_OFF) are used for addressing. For an RMAT 30 graph, we use the physical ID of 6-byte. The p-byte determine the maximum number of page ID, and the q-byte determine the maximum size of a page. For 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 medium-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 64MB page size, since both p and q are well-balanced, and the page size of 64MB 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.

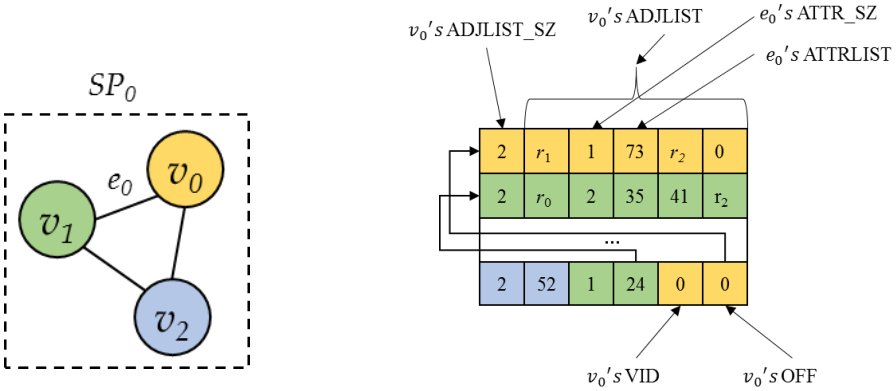
p	q	max. page ID	max. slot number	max. page size
2	4	64 K	4 B	80 GB
3	3	16 M	16 M	320 MB
4	2	4 B	64 K	1.25 MB

5.2 Extended slotted page format for edge attributes

In BFS algorithm, WA is the traversed level of each vertex. In PageRank algorithm, WA is the value of rank each vertex. Those two algorithm require the value of each vertex and update WA of each vertex. However, the graph algorithm which requires the value of each edge or stores the result of processing into edge attribute, is existing such as Single-Source-Shortest-Path (SSSP), Minimum spanning tree (MST), Betwenness centrality (BC). For example, for SSSP, the operation of kernel function is following: (1) read cost of an edge, (2) add cost of current vertex and an edge and (3) update the cost to WA of destination node.

We extend the slotted page format for supporting edge attributes. The edge attribute is located after corresponding edge. The edge attribute consists of the size of attribute list, denoted as ATTR_SZ, and the attribute list of edge, denoted as ATTRLIST. For supporting various size of attribute list, ATTR_SZ is required.

Figure 8 shows an example of extended slotted page for supporting multi-attributes of edge with example graph G. In Figure 8(a), Graph G has three vertices, the v_0, v_1 and v_2. The e_0 is an edge between v_0 and v_1, contains edge attribute. Figure 8(b) shows that storing graph topology with the edge attribute. The first edge of v_0's ADJLIST is the e_0 and after record of e_0, the edge attribute is stored. With the e_0' ATTR_SZ, kernel function can access next record.



(a) Graph G **(b) Small Page SP_0 with edge attribute**

Figure 8. Example of extended slotted page for supporting multi-attribute

6. EXTENDED PROCESSING MODEL OF GSTREAM 2.0

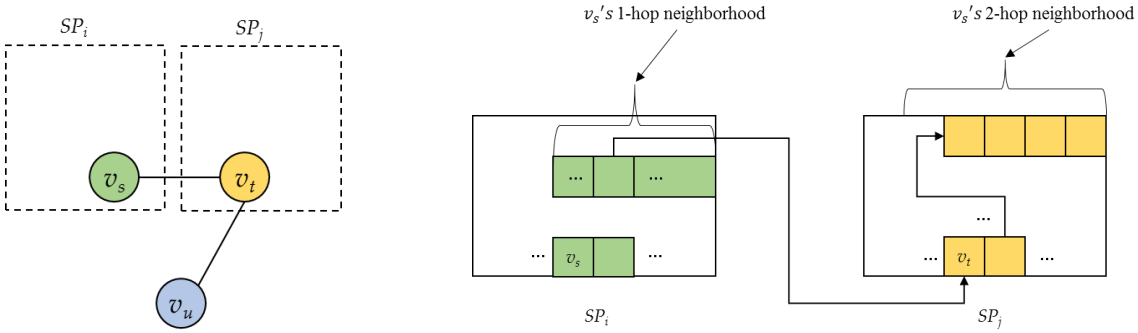
In this section, we present two types of graph algorithm extraneous in terms of BFS-like algorithm and PageRank-like algorithm. For handling those type, we explain extended processing model. In section 3.3, we explain BFS-like algorithm and PageRank-like algorithm. Both of algorithm is categorized based on pattern of transfer topology data due to characteristic of algorithm. In this time, we explain other types which is not categorized based on pattern of transfer data.

Within kernel function for BFS, each thread which handles a vertex only accesses the adjacency list of corresponding vertex. The other example, within kernel function for PageRank, each thread which handles a vertex also only accesses the adjacency list of corresponding vertex. Both of algorithm process on graph with 1-hop neighborhood. In contrast, within kernel function for triangle finding, each thread which handles a vertex, is required to access 2-hop neighborhood. 2-hop neighborhood algorithm include triangle finding, clustering coefficient, 2-hop neighborhood find. The existing graph methods handle 2-hop neighborhood algorithm with contiguous edge list in main memory or message passing to neighborhood. The former approach is commonly used in-memory methods. Nevertheless it is not suitable for GStream 2.0, since the topology structure of GStream 2.0 is fixed size unit. The latter approach is commonly used distributed system. However it is also not suitable for GStream 2.0, since the amount of intermediate data is larger than topology data. We suppose the size of message is *M*, also suppose the number of total edge in graphs is *E*. The size of intermediate data can be calculated easily as $M \times E$. This approach is also not suitable for GStream 2.0 due to transfer overhead.

We propose extended processing model of GStream 2.0 for supporting 2-hop neighborhood algorithm. In this model, kernel function can handling 2-hop neighborhood algorithm without generating intermediate data. If GStream 2.0 generates intermediate data, buffer area for intermediate data is required and the size of other buffers is decreased. In section 3.3 and section 4, we explained the importance of the size of buffers and the size

of WA. Intermediate data is created by using 1-hop neighborhood processing for handling 2-hop neighborhood algorithm. Thus extended processing model of GStream 2.0 processing on 2-hop neighborhood topology data. The streaming and processing pair of page provides to access 2-hop neighborhood topology during kernel function.

Figure 9 shows simple example of page pair processing. In Figure 9 (a), in case 2-hop algorithm processing on vertex v_s which is stored in SP_i, requires SP_j. In Figure 9 (b), kernel function is launched with pair of page, the processing on vertex v_s can access v_s's 2-hop neighborhood.



(a) Example graph of 2-hop neighborhood **(b) Example of accessing 2-hop neighborhood**

Figure 9. Simple Example of page pair processing

In this model, a part of Algorithm 1 is changed. First, nextPIDSet contains pair of pageID. User-defined kernel also is changed to handle pair of page. The unit of streaming data to device memory is pair of page instead of a single page. Line 19, Line 23, Line 24 is changed to pair of page.

7. EXPERIMENTAL EVALUATION

In this section, we present experimental results in six categories. First, we evaluate the performance of GStream 2.0 to show the superiority of our method, 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]. Second, we evaluate the performance of GStream 2.0 to show the superiority of our method, compared with the state-of-the-art CPU-based graph processing methods, Ligra [29], Ligra+ [30], and Galois [26]. For reference, we also evaluate the performance of the multithreaded graph library (MTGL) [2] which is widely used for comparison [36], the parallel graph processing method using CPUs. Third, we evaluate the performance of GStream 2.0 to show the superiority of our method, compared with the state-of-the-art GPU based graph processing method, TOTEM[7,8]. To the best of our knowledge, TOTEM is the only method to exploit multiple GPUs and also to process large-scale graphs that do not fit in GPU device memory. For reference, we also evaluate the performance of Cusha [16] and MapGraph [6], which can process only the graph data that is smaller than GPU memory. Fourth, we evaluate the performance of GStream 2.0 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 GStream 2.0. Fifth, we evaluate the performance of GStream 2.0 on additional graph algorithms. Finally, we evaluate the performance of GStream 2.0 for edge attribute.

7.1 Experimental setup

We use both synthetic and real datasets for experiments. We generate scale-free graphs for synthetic datasets, 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 1:16. We use three well-known graphs of Twitter [18], UK2007 [32], and YahooWeb [34], for real datasets, which all have different sizes and characteristics.

Table 3 shows the basic statistics of those data sets. For GStream 2.0, we use (p = 2, q = 2) in Section 5.1 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 64MB.

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., 24GB), except RMAT32 for CC.

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, each node of which is equipped with two Intel Xeon 8-core 2.60GHz CPUs, 64GB 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, 128GB

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

main memory, two NVIDIA GTX TITAN X GPUs of 12GB 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, GStream 2.0 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.

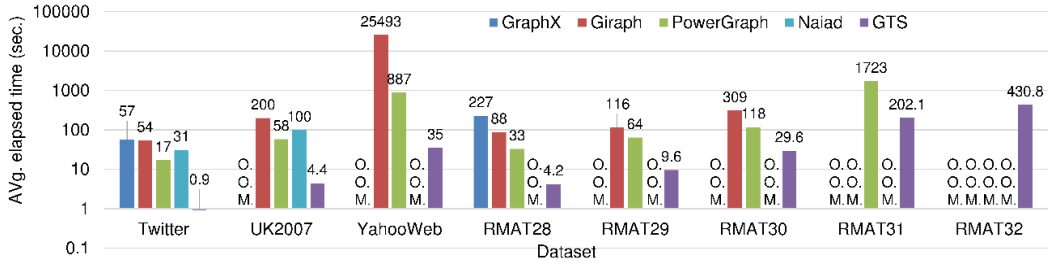
In terms of software versions and configurations, for all three distributed methods, we use Scala 2.11.7 and Spark 1.5.1 for GraphX, MPI ICC 14.0.0 for PowerGraph, and Hadoop 1.2.1. For Giraph, we set the size of mapper memory to 60GB. For Spark, we set the size of executor memory to 60GB. Naiad requires 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 use their latest source codes. We compile all single-machine methods with the same maximum 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 GStream 2.0, 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

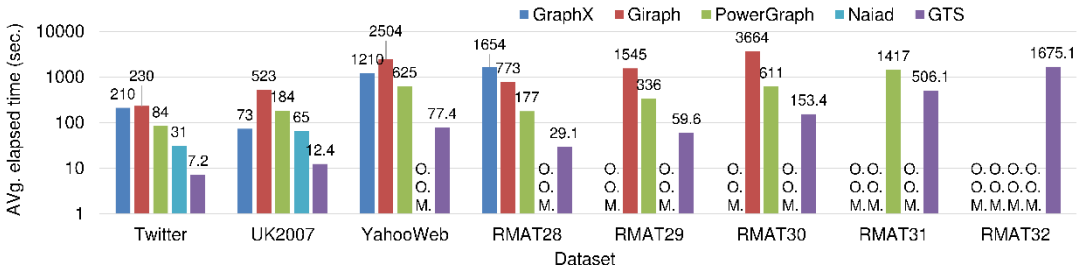
Figure 10 shows the comparison results among GraphX, Giraph, PowerGraph, Naiad, and GStream 2.0, for BFS and PageRank. Y-axis represents the elapsed times in seconds (in log-scale), and O.O.M means out of memory error. In the case of PageRank, we measure the total elapsed times of ten iterations. For four distributed methods, we measure the elapsed time, excluding loading and finalization times. For GStream 2.0, we measure the elapsed times between starting streaming the first page from main memory (for real graphs and RMAT28-30) or SSDs (for RMAT31-32) and showing the query results. For real graphs and RMAT28-30, we exclude loading time (Lines 1-10 in Algorithm 1) for a fair comparison, since they can fit in main memory. We set the main memory buffer size of GStream 2.0 to 20% of a graph size for RMAT31 and RMAT32 (e.g., 45GB for RMAT31). For all datasets used, GStream 2.0 significantly outperforms the distributed graph processing methods using 30 machines, for both BFS and PageRank. Furthermore, GStream 2.0 shows the best scalability among the methods. Only GStream 2.0 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 processing time of GStream 2.0 rapidly increases between RMAT30 and RMAT31 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 GStream 2.0 should increase linearly between RMAT31 and RMAT32 since GStream 2.0 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.

7.3 Comparison with CPU-based Methods

Figure 11 shows the comparison results among MTGL, Galois, Ligra, and Ligra+, and GStream 2.0. In the figure, since the CPU-based methods cannot load data into main memory or process graph algorithms due to lack



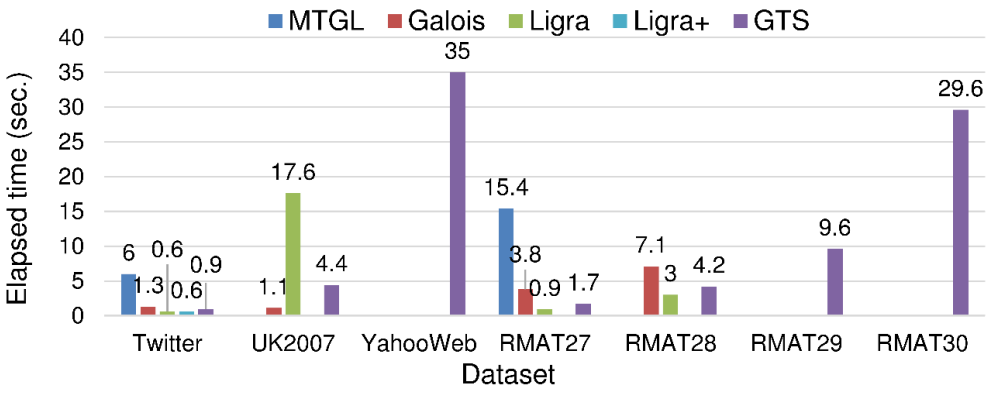
(a) Comparison result for BFS



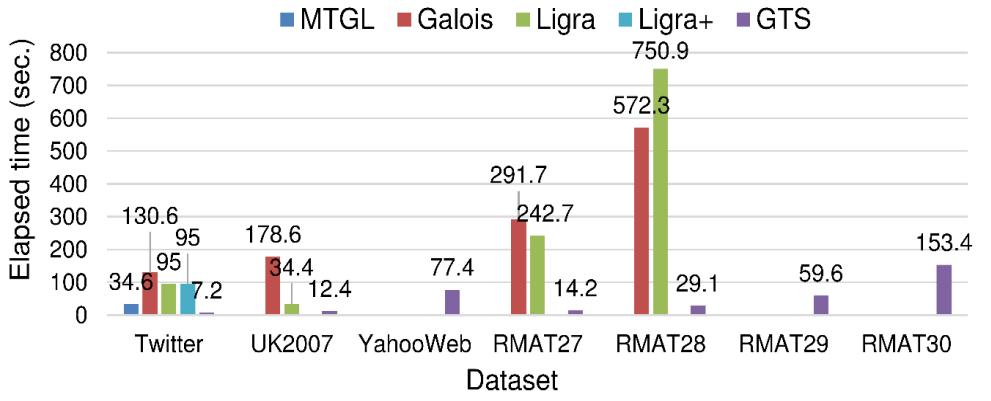
(b) Comparison results for PageRank (# iterations = 10)

Figure 10. Comparison with Graph X, Giraph, PowerGraph, and Naiad for BFS and PageRank

of main memory, there is no results for large-scale graphs such as RMAT29-30 and YahooWeb, except GTS. Among the CPU-based methods, Galois, Ligra, and Ligra+ have significantly outperformed MTGL in terms of both the performance and scalability, except the case of Twitter for PageRank. Among three CPU-based methods, 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 GStream 2.0, either Galois or Ligra slightly outperforms GStream 2.0 for relatively small graphs for BFS. This is because the CPU based methods perform edge-level random access for traversal algorithms, while GStream 2.0 performs page-level random access with data transfer overhead between main memory and GPUs. For relatively large graphs (e.g., YahooWeb, RMAT29-30), only GStream 2.0 could process BFS. For PageRank, GStream 2.0 significantly outperforms all CPU based methods in terms of both the elapsed time and the size of a graph to process.



(a) Comparison results for BFS



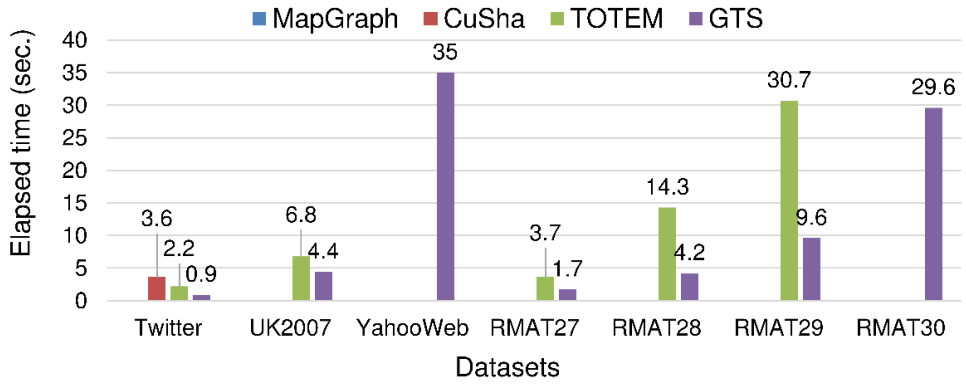
(b) Comparison results for PageRank (# iterations = 10)

Figure 11. Comparison with MTGL, Galois, Ligra, and Ligra+ for BFS and PageRank

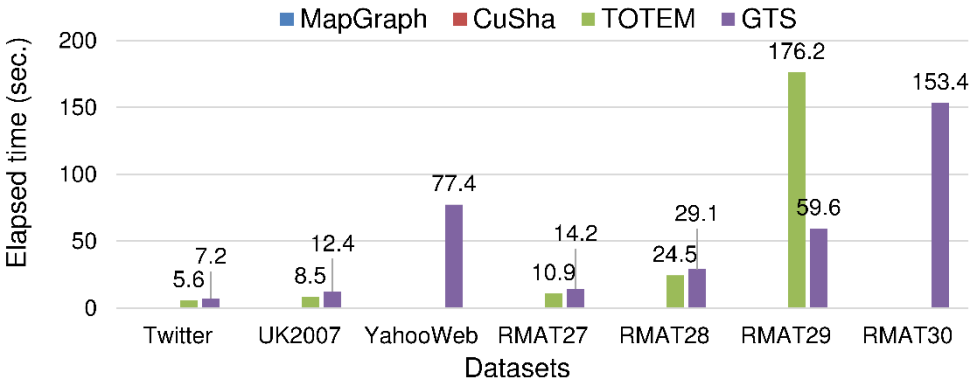
7.4 Comparison with GPU-based Methods

Figure 12 shows the comparison results among MapGraph, CuSha, TOTEM, and GStream 2.0, 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 GStream 2.0 as long as a graph could fit in GPU memory. However, CuSha was slower than GStream 2.0, 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. We compare the performance of GStream 2.0 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. For PageRank, TOTEM slightly outperforms GStream 2.0 for relatively small-scale graphs such as RMAT27, Twitter, and UK2007. GStream 2.0, however, significantly outperforms TOTEM for large-scale graphs such as RMAT29.



(a) Comparison results for BFS



(b) Comparison results for PageRank (# iterations = 10)

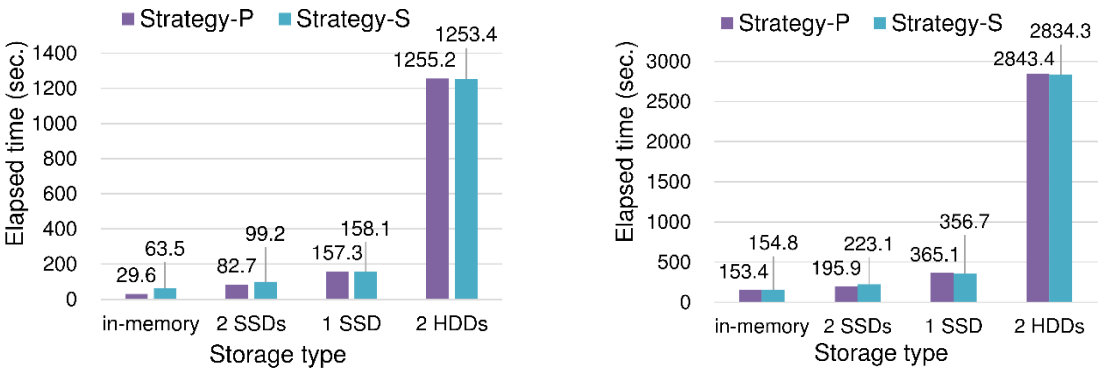
Figure 12. Comparison with TOTEM, MapGraph and CuSha for BFS and PageRank

For BFS, GStream 2.0 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 GStream 2.0 processes PageRank for RMAT29 only in about 59 seconds, which indicates the graph processing speed of GStream

2.0 is about 7GB/s, since the size of RMAT29 is about 40GB in the slotted page format, and the number of PageRank iterations is ten in the experiments. We also note that GStream 2.0 shows the performance of up to 1,500 MTEPS (millions traversed edges per second) for Twitter.

7.5 Characteristics of GStream 2.0

Figure 13 shows the performance of GStream 2.0 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.



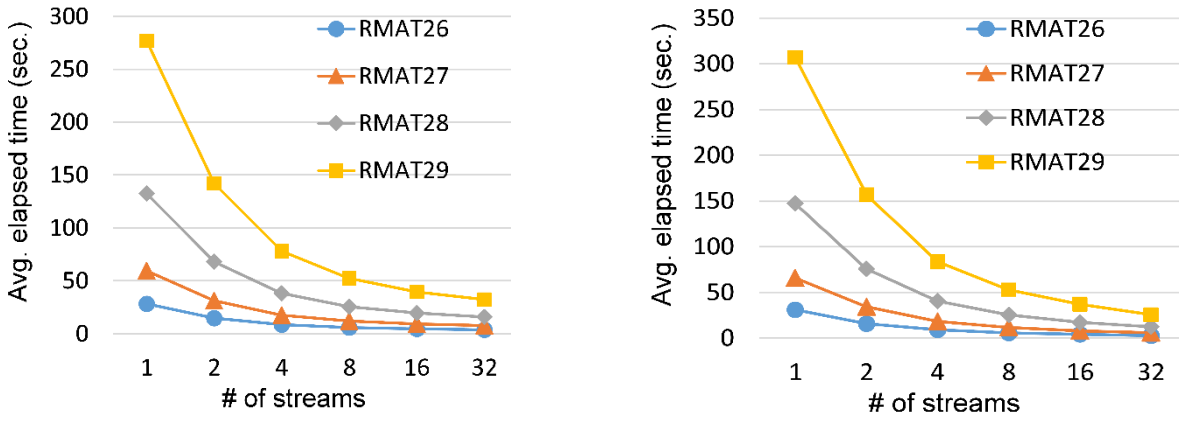
(a) BFS **(b) PageRank (10 iterations)**

Figure 13. Comparison between two strategies for BFS and PageRank (RMAT 30).

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, GStream 2.0 in memory setting takes about 153 seconds, which is approximately equal to 114 × 10 ÷ 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, GStream 2.0 using two SSDs takes about 196

seconds, which is approximately equal to 114×10÷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 GStream 2.0 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 330GB. The elapsed time of PageRank for RMAT30 is about 2,843 seconds, where the calculated time is 114 × 10 ÷ 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 14 shows the performance of GStream 2.0 while varying the number of streams for RMAT26-29. The performance increases steadily as the number of streams increases for all data sets. Even if for BFS where the ratios of transfer time to kernel execution time are much smaller than 32.

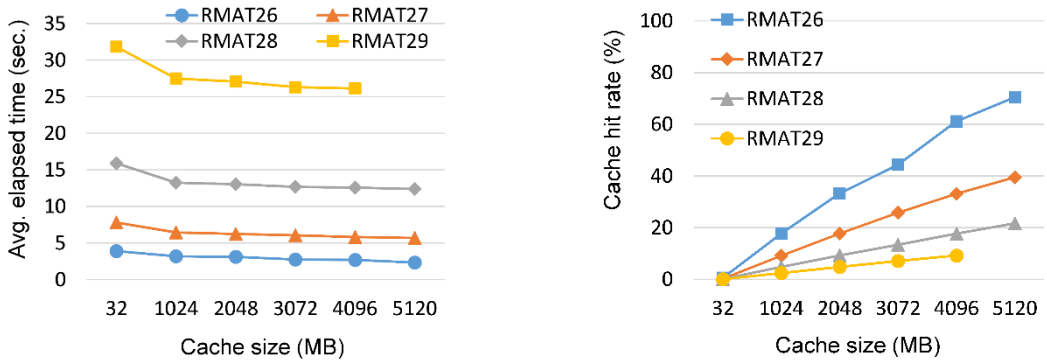


(a) Result for BFS **(b) Result for PageRank**

Figure 14. Performance when varying the number of streams.

Figure 15(a) shows the performance of GStream 2.0 for BFS while varying the cache size from32MB to 5,120MB, and Figure 15(b) shows the corresponding cache hit rates. For RMAT29, there is no result at the cache size 5,120MB 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 GStream 2.0). For example, for the cache of 1,024MB, GStream 2.0 allocates the array of 16 slotted pages of 64MB within GPU_i and make $cachedPIDMap_i$ maintain up to 16 page IDs. In

Figure 15(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.

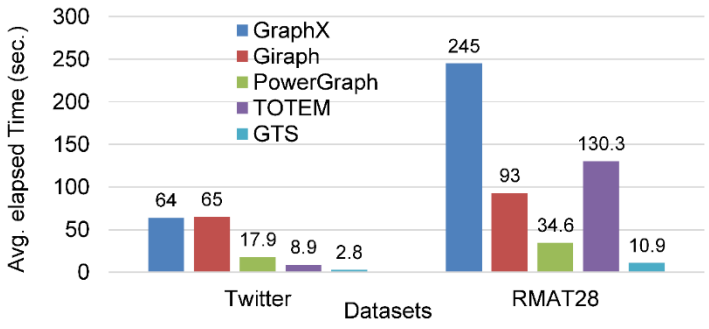


(a) Elapsed times **(b) Cache hit rate**

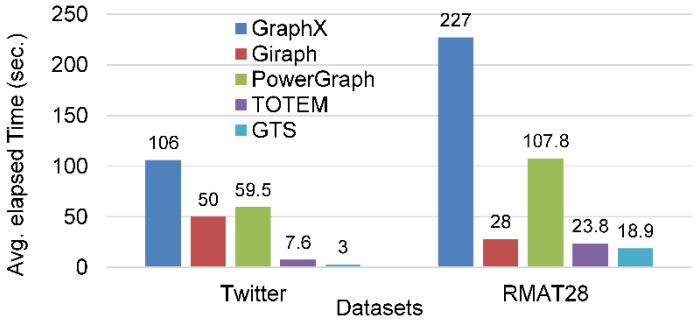
Figure 15. Effectiveness of caching for BFS

7.6 Additional graph algorithms

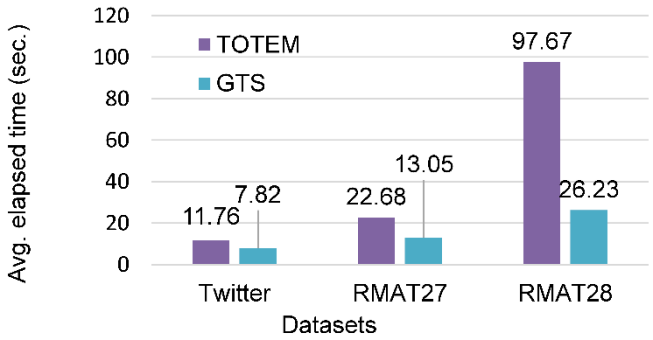
In addition to BFS and PageRank, for a wider range of benchmarks, we implement the following three additional graph algorithms using GStream 2.0: Single-Source Shortest Path (SSSP), Connected Components (CC), and Betweenness Centrality (BC). It demonstrates the adaptability of GStream 2.0. We select those three graph algorithms since Giraph, GraphX, PowerGraph, and TOTEM commonly support them. Figure 16 shows the comparison results among five methods (BC between two methods). GStream 2.0 significantly outperforms other four methods for SSSP and CC, and also largely outperforms TOTEMfor CC. Here, we perform the experiments of BC using the default mode, i.e., the single node mode for both methods.



(a) Comparison for Single-Source Shortest Path (SSSP)



(b) Comparison for Connected Components (CC)



(c) Comparison for Betweennes Centrality (BC)

Figure 16. Comparison for additional graph algorithm: SSSP, CC, and BC

7.7 Experiment for edge attribute

Figure 16 shows the experimental result of structural clustering algorithm for networks (SCAN) like algorithm for edge attribute. For experiment about edge attribute, we implement SCAN-like algorithm. The original SCAN algorithm requires 2-hop algorithm and recursive execution. However, our GStream 2.0 does not support

recursive execution in GPU kernel function, due to lack of register for a single core of GPU. For this reason, we implement SCAN-like algorithm without recursive execution using edge attribute. The reason that there are weak scalability between RMAT25 and RMAT26 is due to Peer-to-Peer communication each other GPU device.

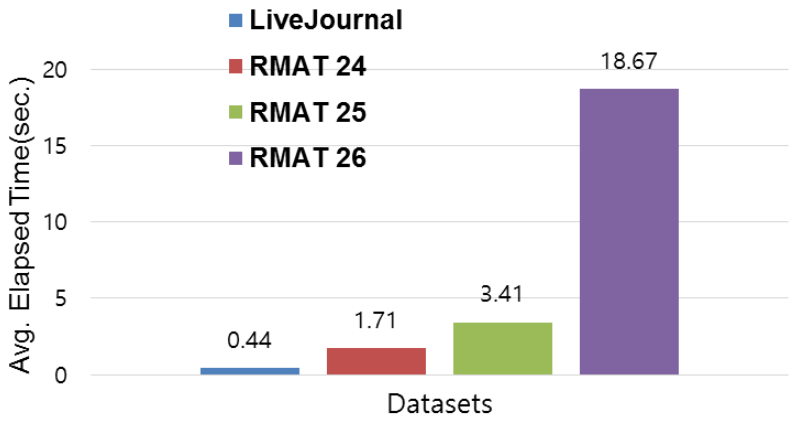


Figure 17. Experimental result for SCAN-like algorithm

8. CONCLUSION

In this paper, we proposed a fast and scalable GPU-based graph processing method called GStream 2.0 that can process even RMAT32 (64billion edges) graphs very efficiently. We proposed novel concept, a concept of storing only updatable attribute data and moving topology data to overcome the limit of GPU memory capacity and moreover the limit of main memory capacity. GStream 2.0 fully exploits the computational power of GPUs by processing the entire graph only using GPUs. 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 hiding time of transfer topology data, GStream 2.0 exploits the asynchronous GPU streams (e.g., CUDA Streams), so it utilize GPU's computing power more. For efficient streaming, GStream 2.0 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. GStream 2.0 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. We expended slotted page format for trillion-scale graph and multiple edge attribute. We also expended processing model of GStream 2.0 for handling 2-hop neighborhood algorithm efficiently. Through extensive experiments, we demonstrated that GStream 2.0 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 GStream 2.0 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.

REFERNECE

[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] Kim, M. S., An, K., Park, H., Seo, H., & Kim, J. GTS: A Fast and Scalable Graph Processing Method based on Streaming Topology to GPUs. In *SIGMOD*, pages 447-461, 2016.

[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.

요 약 문

복수 개의 GPU 와 SSD 를 이용한 대규모의 다중 속성 그래프 처리 방법

그래프 데이터의 특성 중의 하나인 구조의 간단함 때문에 데이터 분석을 쉽게 처리할 수 있어, 그래프 데이터 처리의 중요성이 커지고 있다. 하지만 그래프 데이터의 사이즈가 증가함에 따라 기존 방법들의 큰 규모의 그래프 처리는 점점 힘들어지고 있다. 분산 그래프 시스템은 확장성을 위해 많은 컴퓨터가 필요하지만 그에 비례해 네트워크 오버헤드가 그래프 처리 시간을 능가하게 된다. 이와 별도로 많은 계산 장치를 가진 GPU 를 그래프 처리를 위한 보조처리장치로 사용하는 방법들이 제안 되었다. 하지만 이전에 제안된 GPU 를 이용한 방법들은 메인 메모리의 부족으로 인해 큰 규모의 그래프 처리가 불가능했다.

본 논문에서 제안 된 GStream 2.0 은 한대의 머신에서 메인 메모리보다 큰 규모의 그래프를 처리할 수 있다. 제안된 방법은 보조기억장치에 저장된 그래프 데이터를 GPU 메모리로 스트리밍 하고, GPU 장치를 이용해 그래프 알고리즘을 실행한다. 그래프 데이터를 고정된 크기의 Page 로 나누는 Slotted Page Format 을 사용하여 스트리밍 하기에 GPU 메모리 보다 큰 그래프를 처리 할 수 있다. 연산의 결과를 저장하는 배열을 GPU 메모리에서 관리하는 하여 기존의 분산 방법들에서 성능 저하의 원인이 된 중간 데이터 교환을 없애 성능을 향상시킨다. 간선의 속성을 추가하도록 데이터 구조를 확장하여 간선의 속성을 사용하는 그래프 알고리즘을 지원한다. 또한, 인접 노드들의 인접 노드들이 필요한 알고리즘에 대해서도 지원하도록 확장하였다.

실험을 통해 GStream 2.0 은 다른 주류 방법과 최신 방법들을 비교하였다. 다른 방법들보다 성능이 뛰어남을 보였고, 다른 방법들이 지원하지 못하는 크기의 데이터도 처리함을 보여주었다.

핵심어: 그래프 처리, GPUs, SSDs, Stream, 병렬 프로그래밍