Master's Thesis
석사 학위논문

GStream: A Graph Streaming Processing Method for Large-scale Graphs on GPUs

Hyunseok Seo(서 현 석 徐 玄 錫)

Department of Information and Communication Engineering
정보통신융합전공

DGIST

2015

GStream: A Graph Streaming Processing Method for Large-scale Graphs on GPUs

Advisor : Professor Min-Soo Kim

Co-advisor : Professor Seungho Choe

By

Hyunseok Seo
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]

12. 02. 2014

Approved by

Professor Min-Soo Kim (Signature)
(Advisor)

Professor Seungho Choe (Signature)
(Co-Advisor)

[1] Declaration of Ethical Conduct in Research: I, as a graduate student of DGIST, 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.

GStream: A Graph Streaming Processing Method for Large-scale Graphs on GPUs

Hyunseok Seo

Accepted in partial fulfillment of the requirements for
the degree of Master of Science.

12. 02. 2014

Head of Committee 김 민 수 (인)
Prof. Min-Soo Kim
Committee Member 최 승 호 (인)
Prof. Seungho Choe
Committee Member 은 용 순 (인)
Prof. Yongsoon Eun

MS/IC
201322007

서 현 석. Hyunseok Seo. GStream: A Graph Streaming Processing Method for Large-scale Graphs on GPUs. Department of Information and Communication Engineering. 2015. 35p. Advisors Prof. Kim, Min-Soo, Prof. Co-Advisors Choe, Seungho.

ABSTRACT

Fast processing graph algorithms for large-scale graphs becomes increasingly important as graphs become popular in a wide range of applications and the sizes of graphs are growing rapidly. Due to the relatively low cost and high computational power of GPUs, there have been many attempts to process graph applications by exploiting the massive amount of parallelism of GPUs. However, most of the existing methods fail to process large-scale graphs that do not fit in GPU device memory, mainly due to the irregular structure and the complex graph processing algorithms. Although the state-of-the-art method TOTEM can process large-scale graphs by partitioning a graph into two parts: the main memory part processed by CPUs and the device memory part processed by GPUs, it still has several fundamental problems such as a large amount of synchronization overhead and lack of scalability. We propose a fast and scalable parallel processing method GStream that processes large-scale graphs (e.g., billion vertices) beyond the capacity of GPU device memory very efficiently by exploiting the concept of so-called nested-loop join operation and the asynchronous GPU streams. It exploits multiple GPUs by uniformly distributing the workload among GPUs and also exploits available GPU device memory by caching streaming graph data. GStream is the first scalable method in terms of both the data size and the number of GPUs, to the best of our knowledge. Extensive experimental results show that GStream consistently and significantly outperforms TOTEM in terms of absolute performance, scalability, and graph size to process, for both synthetic graphs and real graphs.

Keywords: Graph processing, Large-scale, GPU, Stream

Contents

List of tables

List of figures

Ⅰ. 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 parallel methods for processing graph algorithms on them have become more important than ever before. Meanwhile, the continuous advancement of GPU technology make the computing power of modern computers ever-increasing. Due to the relatively low cost and the massive amount of parallelism to potentially largely outperform CPUs, GPUs have recently become popular as general computing device. It becomes more and more important to exploit GPUs for better performance per price and energy.

Though exploiting the computing power of GPUs is a promising direction for fast processing large-scale graphs, there are three major challenges to make it difficult. First, the irregularity of graphs causes the *workload imbalance* among threads [12]. In most real-world graphs, the distribution of degrees is highly skewed. The workload imbalance from that skewness may lead to a severe performance penalty due to the underutilization of GPU following the massive parallel architecture. Second, many graph algorithms entail *non-coalesced memory access patterns* [16]. Due to the irregular structure of graphs, accessing the neighbors of a vertex leads to a large amount of data-dependent memory references. This poor locality limits the performance of graph processing on GPUs. Third, many real world graphs do *not fit in the GPU device memory* with this tendency becoming more marked as the sizes of graphs are growing[6-7]. Lack of support for large-scale graphs beyond the capacity of device memory is one of the most critical problems of the existing graph processing methods using GPUs [6-7, 12].

There have been a number of efforts to solve the first two major problems [6-7, 10, 12, 16, 20, 24]. The virtual warp-centric technique (VWC) [12] focuses on solving workload imbalance by partitioning a warp into multiple virtual warps and letting the threads within a virtual warp process the

neighbors of a vertex in parallel. CuSha [16] focuses on overcoming the problem of non-coalesced memory access patterns. The prior works including the VWC method primarily rely on the Compresses Sparse Row (CSR) format for representing graphs in memory. However, CSR could suffer from the irregular memory accesses since it just stores vertices and edges in arrays without regard to access patterns on graphs. Instead, CuSha adopts the format known as *shards* [18] and maps GPU hardware resource on to shards so as to achieve coalesced memory access.

However, there is almost no study on solving the last problem yet, in spite of its importance. To the best of our knowledge, TOTEM [6-7] is the only work to systematically 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, where GPUs process the part in GPU device memory, while CPUs process the part in main memory. It follows the Bulk Synchronous Parallel (BSP) model [23] for synchronization between main memory and device memory.

Even though TOTEM handles much large-scale graphs than the other methods, we observe that is still has three serious drawbacks. First, it has a *large amount of synchronization overhead* between main memory and device memory. Due to the partitioning scheme to cut edges across between main memory and device memory and the BSP model to synchronize per each step, a large amount of status data is frequently copied to device memory and again copied back to main memory. Second, 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 rapidly as the number of GPUs increases, which means the amount of status data to be synchronized among main memory and GPUs also increases rapidly [8]. As a result, the speedup tends to decrease rapidly as well. Third, it has a *fundamental limit on the size of graphs* to process. TOTEM is based on the CSR format, which is an in memory data structure and does not seriously consider large-scale graphs. Since CSR basically uses an index number of 4-byte for each edge, TOTEM cannot even load a graph having a larger number of edges than 2^32=4 billions, though there is enough memory to accommodate it. The CSR

format of 8-byte might solve that issue, but it is still non-trivial to allocate and handle a large-scale contiguous edge array in main memory.

We propose a fast and scalable GPU-based graph processing method called GStream that can process even billion-vertex graphs very efficiently. GStream does graph processing only using GPUs and does not rely on the graph partitioning scheme, which incurs much less synchronization overhead. Instead, it deals with large-scale graphs by exploiting the concept of so-called nested-loop join operation and asynchronous data transfer (i.e., copy) of graph data between main memory and 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, GStream adopts the slotted page format that divides a graph into fixed-size units. As a result, GStream with only a single GPU outperforms TOTEM equipped with two CPUs and two GPUs in most cases. Furthermore, GStream is fairly scalable in terms of the number of GPUs since it performs almost independent graph processing for each GPU. There is no cut edge among main memory and the GPU device memory. In addition, the units of graph data are almost uniformly distributed to each GPU for processing. As a result, GStream could achieve more stable speedup ratios than TOTEM.

The main contributions of this paper are as follows:

- We propose a novel parallel graph processing method on GPUs that can perform graph algorithms very efficiently for large-scale graphs (e.g., billion vertices) much larger than the size of GPU device memory by fully exploiting the asynchronous GPU streams.
- We present two techniques that can improve the performance further: (1) uniformly distributing the workload among multiple GPUs by hashing fixed-size workload units and (2) caching graph data by utilizing available GPU device memory.
- We present a detailed pseudo code of the GStream framework and also its cost model that analyses the trend of the performance and explains the characteristics of GStream.
- Through extensive experiments, we demonstrate that GStream significantly outperforms the state-of-the-art method TOTEM across wide range of benchmarks for both synthetic graphs

and real graphs. GStream could process many graphs that TOTEM couldn't, and also improve the performance up to 7.64 times and 4.28 times compared with TOTEM for BFS and PageRank, respectively.

The rest of this paper is organized as follows. Section II reviews the types of graph algorithms to consider. In Section III, we propose the GStream method. In Section IV, we discuss other techniques to extend GStream. Section V presents the results of experimental evaluation, and Section VI discusses related work. Finally, Section VII summarizes and concludes this paper.

II. PRELIMINARIES

In this section, we present the types of graph algorithms that GStream considers for processing. There are various kinds of graph algorithms, most of which can be categorized into two types: (1) accessing a part of a graph usually via graph traversal and (2) accessing a whole graph usually by scanning vertices and edges [9, 13].

The former algorithms are usually less computationally intensive, but causes non-coalesced memory accesses due to the irregular structure of graphs. The algorithms of this type include Breadth-First Search (BFS), neighbourhood, induced subgraph, egonet, K-core, and cross-edges. BFS is one of the most key algorithms among them, and the other algorithms can be processed in a similar way BFS is processed [9, 13]. Hereafter, we call this type as *BFS-like* algorithms, which are also called as *target quires* in the previous studies [9, 13].

The latter algorithms are usually computationally intensive, and the scan order of vertices and edges is not important in many cases. The algorithm of this type include PageRank, degree distribution, Random Walk with Restart (RWR), radius estimations, and discovery of connected components. PageRank is one of the most typical algorithms among them, and the other algorithms can be processed in a similar way PageRank is processed [9, 13]. Hereafter, we call this type as *PageRank-like* algorithms, which are also called *global queries* in the previous studies [9, 13].

This paper mainly focuses on the BFS and PageRank algorithms as the previous studies do [6-7, 12, 20]. However, our discussion in the paper also can be applied to many other graph algorithms mentioned above.

III. GSTREAM METHOD

In this section, we present to proposed method GStream. Section 3.1 presents the basic concept of GStream, and Section 3.2 presents the streaming scheme of GStream. In Section 3.3 and 3.4, we explain how to exploit multiple GPUs and how to cache graph data, respectively. Finally, Section 3.5 presents the overall framework of the GStream method, and Section 3.6 presents its cost model.

3.1 Basic concept of GStream

In general, graph algorithms require both graph topology data (shortly, topology data) and attribute vectors 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*). The attribute vectors again can be divided into read-only attribute vector. For example, for PageRank, *prevPR* is a read-only attribute vector, while *nextPR* is a read/write attribute vector, in a specific iteration.

Dividing graph data into topology data and attribute vectors and further dividing attribute vectors into read-only ones and read/write ones is the first step that can make it possible to process large-scale graphs beyond the capacity of GPU device memory. In a situation where all of topology data and attribute vectors cannot be accommodated in the device memory, which part of them is kept in the device memory and in which order they are processed can significantly affect the performance. They determine the parallel processing framework as well. We categorize the relationship between the size of graph data and the capacity of GPU device memory into the following three cases.

- ***Case1***. Both topology data and attribute vectors can be kept in the device memory of GPUs.
- ***Case2***. Only either topology data or attribute vectors can be kept in the device memory of GPUs.
- ***Case3***. Neither topology data nor attribute vectors can be entirely kept in the device memory of GPUs.

Among the above cases, we mainly focus on Case2 and Case3 since they are not properly handled by the existing methods. For doing that, GStream adopts the concept of "join" operation from the database area, especially, so-called *nested-loop join* [11]. For two sets $X = \{x_1, \ldots, x_p\}$ and $Y = \{y_1, \ldots, y_q\}$, nested-loop join processes all elements $\{y_1, \ldots, y_q\}$ for each element x_i with computing a user-defined function on each pair $\langle x_i, y_j \rangle$, if X is the *outer join operand*, and Y is the *inner join operand*. Conceptually, we can set topology data as the outer join operand and attribute vectors as the inner join operand, which we call it as the *topology-major* strategy, or can set attribute vectors as the outer join operand and topology data as the inner join operand, which we call it as the *attribute-major* strategy. Intuitively, the attribute-major strategy processes graph algorithms by copying the topology data (i.e., inner data) in a streaming fashion to GPU device memory for each chunk of the attribute vectors (i.e., outer data) resident in GPU device memory. Here, the number of chunks of the attribute vectors would be only one in Case2, but more than one in Case3. In contrast, the topology-major strategy processes graph algorithms by copying the attribute vectors (i.e., inner data) in a streaming fashion to GPU device memory for each chunk of the topology data (i.e., outer data) resident in GPU device memory. Between two strategies, GStream chooses the attribute-major strategy since it has more potential benefits in terms of performance such as smaller outer data and lower synchronization overhead, compared with the topology-major strategy. The amount of the attribute vectors is typically smaller than that of the topology data, which indicates a smaller number of chunks (i.e., a smaller number of iterations in outer loop) and a smaller number of streaming copies of topology data to device memory. We present the detailed performance analysis in Section 3.6.

Figure 1 shows the basic data flow of GStream following the attribute-major strategy. In Figure 1, *WA* represents read/write attribute vectors, and *RA* represents read-only attribute vectors. Here, we suppose *WA* is divided into *W* units, and similarly *RA* is divided into *R* units. Between *WA* and *RA*, GStream considers only *WA* as outer data in order to further reduce the size of outer data. Since *WA* is frequently updated during graph algorithm, 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 each chunk of outer data, i.e., WA_i, GStream

performs the following three steps: (1) copying WA_i to GPU device memory; (2) processing graph algorithms while asynchronously copying the topology data together with the corresponding parts of read-only attribute vectors to GPU device memory in a streaming fashion; and (3) copying WA_i, which has been updated during graph processing, back to main memory for (data) synchronization.

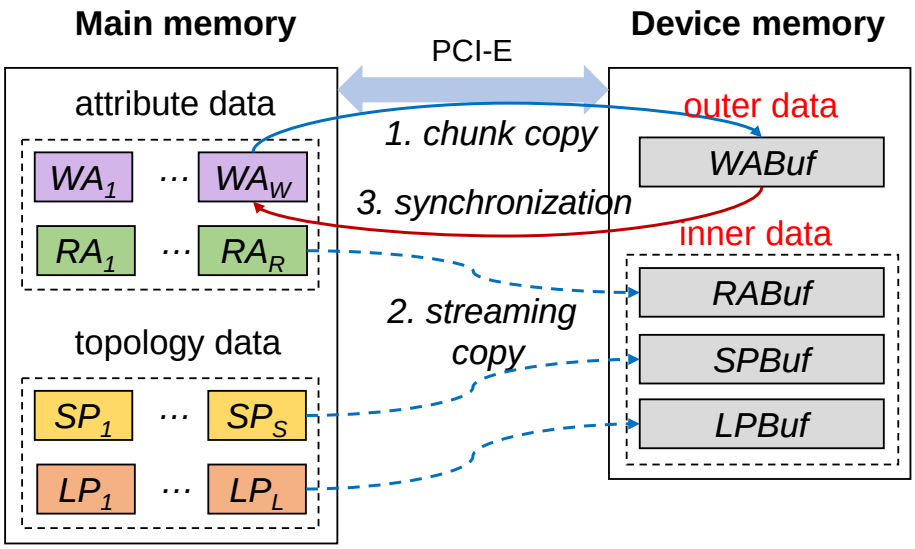


Figure 1. Basic data flow of GStream.

3.2 Streaming topology data

GStream copies the topology data from main memory to GPU device memory via the PCI-E bus asynchronously in a streaming way. For representing a sparse graph in memory, where most of real graphs are sparse, there have been proposed many formats such as Coordinate list (COO), Yale format, Compressed Sparse Row (CSR), and Compressed Sparse Column (CSC). However, they all are inadequate for streaming the topology data since it is hard to divide topology into smaller independent pieces that can be properly transferred and processed, especially into fixed-size ones for efficiency. The existence of hub vertices having a large number of outgoing edges makes this problem harder. For this reason, GStream adopts the slotted page format that have been developed in the database area [9, 11], which divides topology data into fixed-size pages, typically of 1MB. Here, it is no necessary for GStream to use a specific format like the slotted page format. GStream can adopt any format that can divide topology data into fixed-size units. Figure 2 shows an example graph G and

the slotted pages of G. In Figure 2(a), the vertices v_0, v_1, and v_2 have a relatively small number of neighbour vertices, and so are stored in a single page called *Small Page* (SP) as in Figure 2(b). An SP consists of two parts, *records* and their *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 and the start offset of the corresponding record. A record consists of the size of the adjacency list and the adjacency list itself. The vertex v_3 has a relatively large number of neighbour vertices, and so it stored in multiple pages called *Large Pages* (LPs), which structure is almost the same with that of SP as in Figure 2(c). The vertex IDs are sorted in the pages. More details are explained in [9]. As shown in Figure 1, we assume the number 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 is *SP*, which will be shown in Section 5.1.

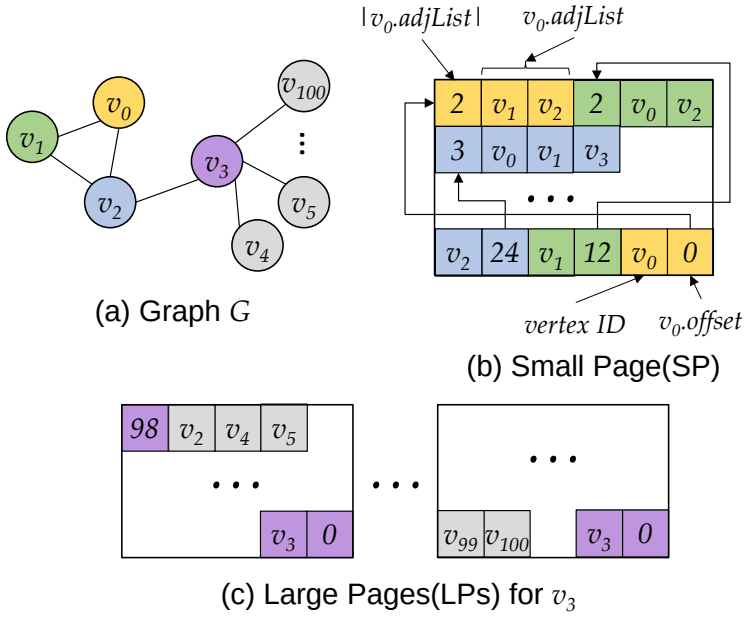


Figure 2. Example graph G and the slotted pages of G.

For streaming inner data *RA*, *SP*, and *LP* to device memory, GStream allocates three kinds of streaming buffers in device memory called *RABuf*, *SPBuf*, and *LPBuf*, respectively, as in Figure 1. In addition, for outer data *WA*, GStream also allocates a chunk buffer called *WABuf*.

GStream exploits multiple GPU streams for streaming inner data. Figure 3 shows the timeline of copy operations of inner data and outer data to device memory. Since GPU threads cannot execute a kernel function before outer data becomes available in GPU, a CPU thread first transfers WA_i to *WABuf*. After that, it starts multiple GPU streams, each of which performs a series of operations, (1) copying SP_i to *SPBuf*, (2) copying RA_i to *RABuf*, and (3) executing the kernel function, repeatedly. Here, RA_i represents the attribute sub-vectors corresponding to SP_i, which can be easily identified since vertex IDs are sorted in the pages. In general, transfer operations for WA_i, RA_i, and SP_i to device memory cannot overlap with each other, at least the current GPU architecture [1]. Instead, they can overlap with kernel execution with multiple streams [1, 21]. Theoretically, the suitable number of stream k can be determined by using the ratio of the transfer time of SP_i and RA_i to the kernel execution time. For example, in Figure 3, if the kernel execution time is k times longer than transfer time for SP_i and RA_i, 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 algorithm.

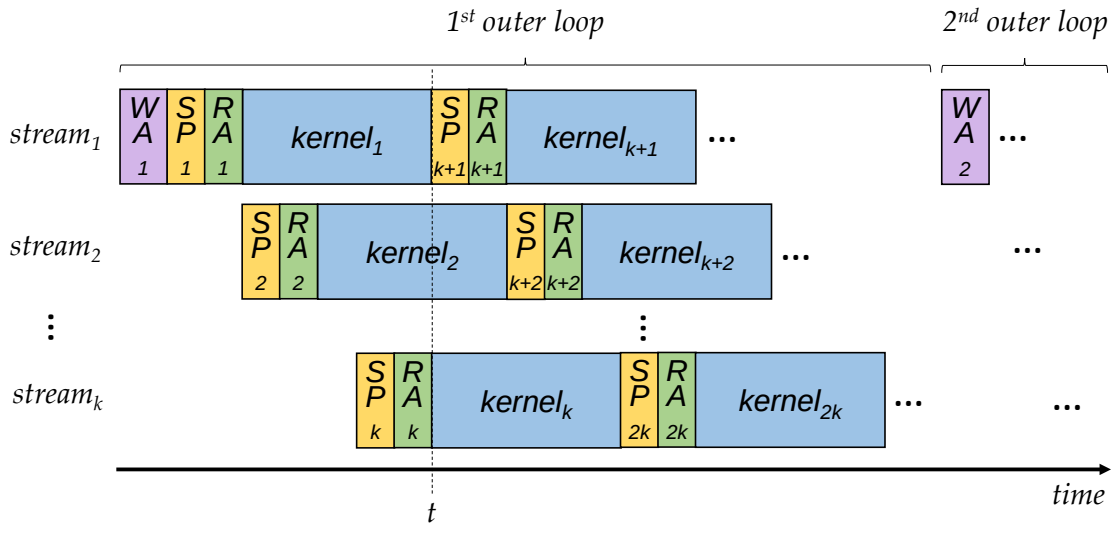


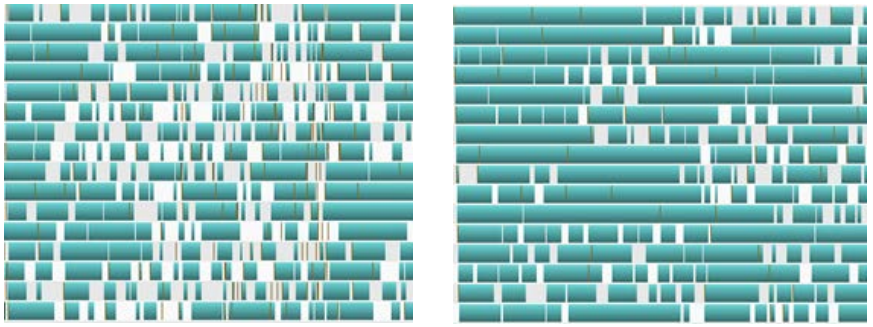
Figure 3. Timeline of copy operations in multiple streams.

However, in practice, the performance continuously increases until using 32 streams, which will be shown in Section 5.3. This is because the kernel execution becomes faster when SP_i and RA_i are prepared in the queues of GPU in advance, and also the maximum number of streams that can execute a kernel function concurrently is 32 in the current CUDA technology [1]. After streaming and processing all the SPs for WA_i, GStream performs streaming and processing all the LPs for WA_i. The reason separating processing SPs from processing LPs is reducing the kernel switching overhead among SPs and LPs. After processing all the inner data required for the first outer data WA_1 is done, the updated outer data WA_1 is copied back to main memory for *bulk* synchronization, which is omitted in Figure 3 for lack of space. Then, the second inner loop using WA_2 starts.

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

	Twitter [17]	UK2007 [3]	YahooWeb [4]
BFS	1:3	1:1	2:1
PageRank	1:20	1:6	1:4

Figure 4 shows the actual timeline of copy operations for BFS and PageRank when using 16 streams and a synthetic data, which is obtained by profiling. In the figure, very short red colored bars indicate copying SP_i and RA_i to device memory, while long green colored bars indicate executing a kernel function. The timeline for PageRank in Figure 4(b) is more dense 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 Exploiting Multiple GPUs

Section 3.1 and 3.2 describe the basic principle of GStream with assuming a single GPU. However GStream is easy to be extended to a more powerful method exploiting multiple GPUs. We let the number of GPUs be *N*. The strategy of GStream for multiple GPUs is copying the same outer data to all GPUs and copying a different inner data to each GPU, which we call it as the *outer replication* strategy. Figure 5 shows the data flow of GStream following that strategy, which consists four steps. In Step1, GStream copies the same WA_i to all $\{GPU_1, \ldots, GPU_N\}$. In Step2, it copies a different $\langle SP_{i+j}, RA_{i+j} \rangle$ to each GPU_{i+j} for $0 \le j \le N\text{-}1$. LPs are processed in the same way. Then, each GPU can execute kernel function independently for a different part of topology data. Thus, GStream potentially can achieve fairly linear parallel speedup with respect to the number of GPUs for graph processing.

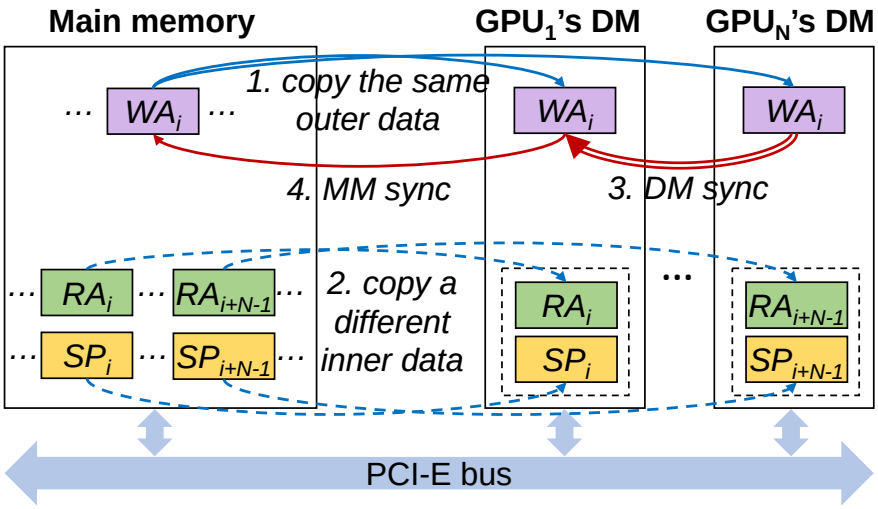


Figure 5. Data flow of GStream using multiple GPUs.

Moreover, since the different inner 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 such as its size and its density. In contrast, the state-of-the-art method is not stable in terms of the speedup ratio, which will be shown in Section 5.2. In general, for load balancing, GStream calculates a hash value *h(x)* for a page ID *i* for SP_i and then copies the page SP_i to $GPU_{h(i)}$. Typically, GStream uses the mod function for the default hash function.

When using multiple GPUs, a naïve 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. GStream largely reduces such synchronization overhead by exploiting so-called *peer-to-peer memory copying* among GPUs, which speed is much faster than the copy speed of between GPU and main memory. The Step 3 and 4 in Figure 5 show the data synchronization of GStream. The outer data of each GPU is copied to the device memory of a master GPU (e.g., GPU_1) in Step 3, and then the updated outer data in GPU_1 is copied to main memory in Step 4.

Different from the outer replication strategy, it is also possible to exploit multiple GPUs by copying a different outer data to each GPU and copying the same inner data to all GPUs, which we call it as the *inner replication* strategy. However, GStream does not adopt that strategy since it could not exploit peer-to-peer memory copying for outer data synchronization, and so slower than the outer replication strategy.

3.4 Caching topology data

After GStream allocates four buffers *WABuf*, *RABuf*, *SPBuf*, and *LPBuf* in the GPU device memory, there might be free memory available in GPU device memory. Especially, if GStream allocates a small amount of *WABuf* due to small outer data as in BFS, where *WA* is just a *visit* attribute vector for vertices, there is a lot of free memory left in GPU device memory. In that case, GStream tries to maximize the performance of graph processing by caching streaming inner data, especially SPs and LPs. The BFS-like algorithms could access the same topology page repeatedly during traversal, and thus caching 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 naïve approximation of the cache hit rate using B pages would be $B/(S+L)$ for random graphs, though it also depends on a cache algorithm used. As a cache algorithm, GStream basically adopts the LRU algorithm, but other cache algorithms can be used as well.

3.5 Framework of GStream

The processing scheme described in Sections 3.1-3.4 can process some graph algorithms consisting of only a single nested-loop join between topology data and attribute vectors, like PageRank, but not all kinds of graph algorithms. For example, BFS requires level-by-level traversal, where a single level traversal requires a single nested-loop join between the set of topology pages containing the vertices to be visited next (i.e., inner data) and the *visit* attribute vector (i.e., outer data). Thus, in general, GStream should perform multiple nested-loop join operations for processing graph algorithms. Here, we note that the previous work such as [6, 9] usually processes a single level traversal of BFS by scanning almost the entire topology data even though only a very small portion of topology data is necessary for each traversal level, which may largely degrade the performance of BFS. In contrast, GStream improves the performance by minimizing the amount of inner data, i.e., by copying only the topology pages containing the vertices to be visited next.

In terms of page access patterns, PageRank and BFS stand at the two extremes: the former just needs a single nested-loop join accessing the entire topology as inner data, while the latter needs multiple nested-loop joins, each join of which accesses a very small portion of topology as inner data. GStream integrates two extremes into a single framework. Figure 6 presents the framework of the GStream method. It performs a user-defined kernel function K_Q for a graph algorithm such as PageRank and BFS on a graph G. As an initialization step, it loads G into main memory (MM), creates the streams for small pages and large pages for each GPU, and allocate 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 K_Q is a function for BFS-like algorithm, the page ID containing the start vertex is assigned to *nextPIDSet*. Otherwise, the constant *ALL_PAGES* is assigned to it. The map for cached page IDs $cachedPIDMap_i$ for each GPU_i is initialized, which is used for caching old topology pages in GPU_i (Line 9). The `do-while` loop (Lines 10-33) takes charge of level-by-level traversal. If K_Q is a PageRank-like algorithm, this loop is performed only once. The three `for` loops (Lines 11, 13, and 21) takes charge of a single nested-loop join. In Figure 5, Steps 1, 2, 3, and 4 correspond to Line 12, Lines 13-27, Line 28, and Line 29, respectively.

Lines 13-20 are performed for processing small pages, and Lines 21-27 are performed for processing large pages. We note that the inner loop (Lines 13 and 21) asynchronously transfers a topology page SP_j (or LP_j) in *nextPIDSet* to a specific $GPU_{h(j)}$ after hashing the page ID *j*, as discussed in Section 3.3.

The GStream Framework

Input: Graph *G* // input graph
K_Q // user-defined kernel function

Variables: *nextPIDSet* // set of page IDs to process
$cachedPIDMap_{1:N}$ // maps of cached page IDs

```
 1:  Load G into MM;
 2:  Create SPstream and LPstream for GPU_1:N;
 3:  Allocate {WABuf, RABuf, SPBuf, LPBuf} for GPU_1:N;
 4:  if K_Q is a BFS-like algorithm then
 5:     nextPIDSet ← page ID containing start vertex;
 6:  else // K_Q is a PageRank-like algorithm
 7:     nextPIDSet ← ALL_PAGES;
 8:  end if
 9:  cachedPIDMap_1:N ← ∅
10:  do   // perform level-by-level traversal
11:     for i ← 1, W do
12:        Copy WA_i to WABuf of GPU_1:N;
13:        for j ∈ nextPIDSet do // processing SPs
14:           if j ∉ cachedPIDMap_h(j) then
15:              Asyn-copy SP_j to SPBuf of GPU_h(j);
16:              Asyn-copy RA_j to RABuf of GPU_h(j);
17:           end if
18:           Call K_Q for SP_j in GPU_h(j);
19:        end for
20:        Thread synchronization in GPU_1:N;
21:        for j ∈ nextPIDSet do // processing LPs
22:           if j ∉ cachedPIDMap_h(j) then
23:              Asyn-copy LP_j to LPBuf of GPU_h(j);
24:           end if
25:           Call K_Q for LP_j in GPU_h(j);
26:        end for
27:        Thread synchronization in GPU_1:N;
28:        Copy WA_i of GPU_2:N to GPU_1;
29:        Copy WA_i of GPU_1 of to MM;
30:     end for
31:     Copy nextPIDSet_1:N and cachedPIDMap_1:N to MM;
32:     nextPIDSet ← ∪_1≤i≤N nextPIDSet_i;
33:  while nextPIDSet ≠ ALL_PAGES ∧ nextPIDSet ≠ ∅
```

Figure 6. Pseudo code of the GStream framework.

Here, before transferring the page, GStream first checks if the page already exists in the cache of $GPU_{h(j)}$ by looking up $cachedPIDMap_{h(j)}$ (Lines 14 and 22). During executing K_Q, a new set of page IDs to process at the next level is assigned to $nextPIDSet_i$ in device memory of each GPU_i, which is copied back to MM (Line 31), and then merged into the original *nextPIDSet* (Line 32). The updated $cachedPIDMap_{1:N}$ is also copied back to MM and used in the next level traversal. In the case of PageRank-like algorithms, both *nextPIDSet* and $cachedPIDMap_{1:N}$ are actually not used. In the case of BFS-like algorithms, we can easily improve the performance by maintaining WA_i in *WABuf* without copying it repeatedly for each traversal level.

3.6 Cost model of GStream

Now, we present the cost models of GStream, which would allow us to understand the performance tendency. We only consider major factors that could affect the performance of GStream. Since PageRank-like algorithms and BFS-like algorithms show a quite different tendency, we present two cost models.

The cost model for PageRank-like algorithms is given by

$$\frac{2|WA|}{c1} + W \times \left\{ \begin{array}{l} \dfrac{|RA| + |SP| + |LP|}{c2 \times N} + t_{call}\left(\dfrac{S+L}{N}\right) \\ +t_{kernel}\left(SP_{|1|} + LP_{|1|}\right) + t_{sync}\ (N) \end{array} \right\} \quad (1)$$

where *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* in GPUs. For example, in PCI-E 2.0 x16 interface, *c1* is about 8GB/s, while *c2* is about 6GB/s. In Eq.(1), $2|WA|/c1$ indicates the total amount of time for copying all WA_i to device memory and copying the updated one back to main memory. That time does not decrease with multiple GPUs due to the outer replication strategy discussed in Section 3.3. The term in the braces is the time for a single inner loop, which is performed *W* times. The transfer time of inner data $(|RA|+|SP|+|LP|)/c2$ is divided by *N*

since transmission can be performed concurrently for 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 as shown in Figure 3. That time is not short since PageRank-like algorithms are usually computationally intensive, and 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 WA_i among more GPUs.

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

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

where *depth* is the number of traversal levels, $SP_{\{l\}}$ is a set of small pages visited at an l-th level traversal, d_{skew} is the degree of workload skewness (i.e., imbalance) among GPUs, and r_{hit} is the cache hit rate $B/(S+L)$ discussed in Section 3.4. As discussed in Section 3.5, *WA* for BFS-like algorithms is usually small enough to fit in device memory and can be maintained in device memory during algorithm execution. Thus, we do not need to copy *WA* to GPU device memory for each traversal level, i.e., a total of *depth* times. Instead, *WA* is copied only once, and so there is no term of multiplication with W. The operations in the braces at different traversal levels 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 inner data $(|RA_{\{l\}}|+|SP_{\{l\}}|+|LP_{\{l\}}|)/c2$ is divided by N due to using N GPUs, and moreover divided by d_{skew}, which is between $1/N$ (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 inner 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 is usually so small. In the term $t_{call}((S_{\{l\}}+L_{\{l\}}) / (N \times d_{skew}))$, $S_{\{l\}}$ indicates the number of small pages visited at an l-th level traversal.

IV. OTHER TECHNIQUES

4.1 Fine-granular parallel processing in GStream

GStream mainly focuses on *coarse-granular* or *macro-level* parallel graph processing, as presented in Section 3. However, since GStream processes topology data page-by-page, we can apply various kinds of *fine-granular* or *micro-level* parallel graph processing techniques to each page. Even we can apply a better/different technique to each page depending on its characteristics such as density, i.e., the ratio of the number of vertices to the number of edges within a page.

One of naïve micro-level technique would be letting each GPU thread process each vertex and its outgoing edges, which we call it as *vertex-centric*. On the contrary, the VWC method [12] that the threads in a warp process the outgoing edges of a vertex simultaneously can be considered as *edge-centric*. In general, the vertex-centric strategy is suitable for every sparse graphs where each vertex as only few outgoing edges, while the edge-centric one is suitable for less-sparse graphs. In order to cope with both cases, GStream adopts the *hybrid* strategy that processes a page with the vertex-centric technique if its density is lower than a certain threshold, or processes it with the edge-centric technique otherwise.

4.2 Extending to a disk-based method

GStream can be relatively easily extended so as to process a huge-scale graphs stored in disks (e.g., HDD and SDD), since the slotted page format is originally a developed one for disk-based processing. However, the gap between I/O performance of disks and computing power of GPUs is quite large at least under the current computer architecture, and so there is a clear limit on the performance of a disk-based graph processing using GPUs. Even PCI-E type SSD (e.g., Fusion-io's, Intel's) has much lower practical I/O speed of around 1GB/s compared with the graph processing speed of GStream of around several GB/s, which will be shown in Section 5. The communication speed between main memory and GPU device memory is also much higher than the I/O speed of PCI SSD. Therefore, the

overall performance of disk-based GStream is expected to equal to the I/O speed of disks for the time being.

V. PERFORMANCE EVALUATION

In this section, we present experimental results in two categories. First, we evaluate the performance of GStream compared with the state-of-the-art method TOTEM [6-7] 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 exploits multiple GPUs. Second, we evaluate the performance of GStream while varying the number of streams, the sizes of buffers, and the densities of graphs to show the characteristics of GStream.

5.1 Experimental setup

For experiments, we use both synthetic datasets and real datasets. For synthetic datasets, we generate scale-free graphs following a power law degree distribution by using RMAT [5], which are summarized in Table 2. For small synthetic graphs that can fit in the device memory of a sing GPU, we generate RMAT24, RMAT25, and RMAT26, where the ratio of the number of vertices to the number of edges is set to 16 as in TOTEM. For large synthetic graphs that cannot fit in the device memory, we generate RMAT27, RMAT28, and RMAT29. For real datasets, we use three well-known graphs of Twitter [17], UK2007 [3], and YahooWeb [4], which all have different sizes and characteristics. Table 2 shows the basic statistics of those data sets, where most of the topology pages is SP for both synthetic graphs and real graphs.

Table 2. Synthetic and real graph datasets used for experiments.

data	#vertices	#edges	#pages (in GStream)	
			#SP	#LP
RMAT24	16M	256M	1,205	0
RMAT25	32M	512M	2,415	2
RMAT26	64M	1,024M	4,880	3
RMAT27	128M	2,048M	9,724	58
RMAT28	256M	4,096M	19,533	62
RMAT29	512M	8,192M	38,747	937
Twitter	42M	1,468M	5,418	1,029
UK2007	106M	3,739M	15,484	0
YahooWeb	1,414M	6,636M	32,807	0

We conduct all the experiments on the same workstation with two Intel Xeon E5-2687W 3.1GHz CPUs of eight cores, 128 GBytes main memory, and two NVIDIA GTX TITAN GPUs of 2,688 cores and 6 GB device memory. The CPUs and GPUs are connected to PCI-E 2.0 x16 interface due to the mainboard that not supports PCI-E 3.0 x16 interface. For TOTEM, we download the source code from [2]. All experiments are performed on the same OS of SUSE Linux Enterprise 11 SP2 and the same GPU Toolkit of CUDA 5.5. Both TOTEM and GStream are compiled with same optimized option of –O3 with gcc 4.3.4. In all experiments where a graph does not fit in device memory, GStream uses only GPUs, while TOTEM uses both all 16 CPU cores and GPUs.

5.2 Comparison with TOTEM

Figure 7 shows the comparison results between TOTEM and GStream. Figures 7(a)-(b) are for small synthetic graphs, Figures 7(c)-(d) are for large synthetic graphs, and Figures 7(e)-(f) are for real graphs. Each of Figures 7(a)-(f) shows two side-by-side results for BFS(in left) and PageRank(in right). For all figures, Y-axis represents an average elapsed time in seconds, and so lower bars indicate better performance. In the case of PageRank, we measure an average of the total elapsed time of 10 iterations.

Small synthetic graphs. We compare the performance of GStream with the best performance of TOTEM by using small synthetic graphs. The performance of TOTEM is maximized when copying the entire graph data to device memory and processing graph algorithms only by using GPUs [6-7]. In Figure 7(a)-(b), GStream improves the performance for all cases compared with TOTEM. It improves the performance up to 7.12 times for BFS (on RMAT26 using a single GPU), and up to 2.62 times for PageRank (on RMAT24 using two GPUs). The reason why the improvement of BFS is higher than that of PageRank is due to the way of accessing topology data for traversal. TOTEM as well as other major methods [6-7, 12] process BFS by scanning almost the entire topology data at each level traversal, while GStream processes BFS by streaming only the necessary topology pages to device memory at each level of traversal. We note that GStream with a single GPU outperforms TOTEM with two GPUs for all RMAT24-26 and for both BFS and PageRank. For RMAT25, TOTEM

fails to process PageRank with the error message “unspecified launch failure”, which seems to be a bug of TOTEM, and so there is no corresponding elapsed time.

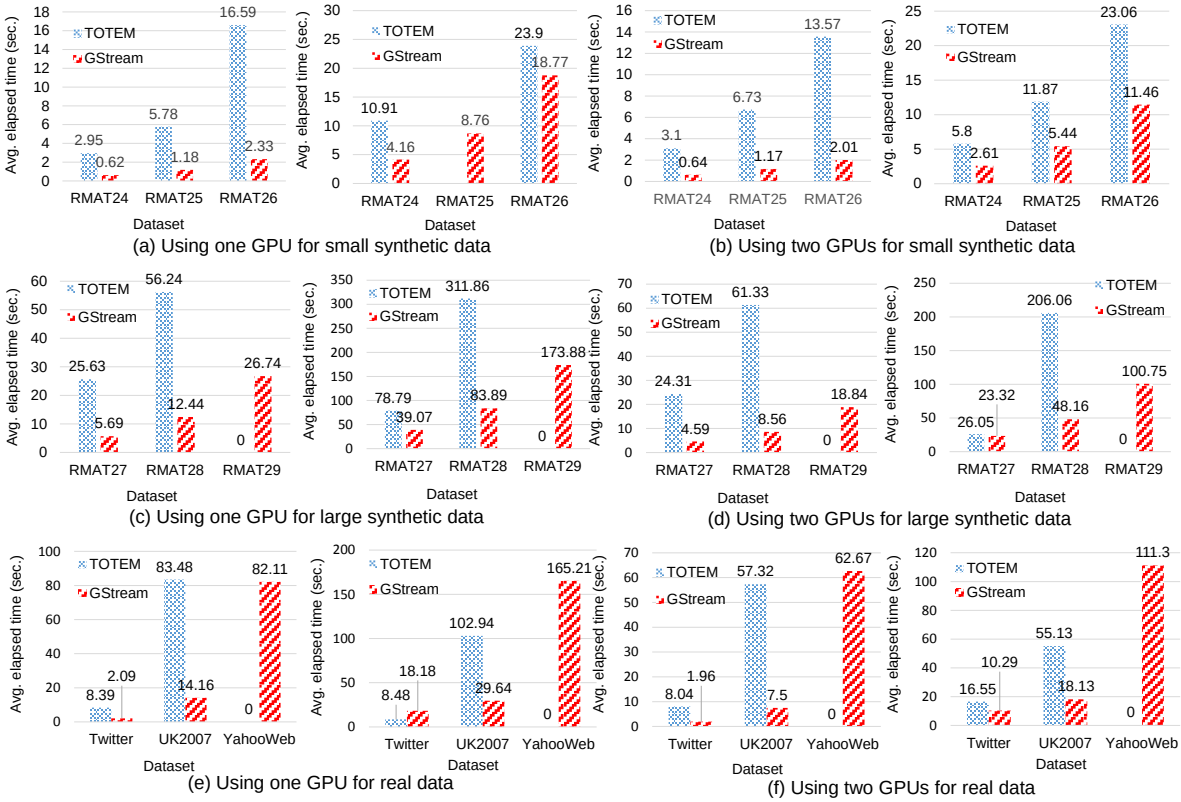


Figure 7. Comparison with TOTEM: BFS (in the left side) and PageRank (in the right side) in each of (a)-(f).

Large synthetic graphs. We compare the performance of GStream with the best performance of TOTEM for large synthetic graphs by fitting as much graph data as possible in device memory and letting GPU(s) process that data. The remaining graph data that cannot fit in device memory is in main memory and processed by CPUs. Table 3 shows the ratio of graph data processed by GPUs to that by CPUs in TOTEM (GPU%:CPU%). There is no information about RMAT29 since TOTEM cannot load it. For RMAT27-28, the portion of the GPU side for PageRank is smaller than that for BFS, since PageRank requires bigger status information than BFS.

Table 3. Ratios of partition sizes in TOTEM (GPU%:CPU%).

		RMAT27	RMAT28	RMAT29
one GPU	BFS	60:40	30:70	N/A
	PageRank	50:50	20:80	N/A
two GPUs	BFS	100:0	50:50	N/A
	PageRank	100:0	50:50	N/A

For RMAT29, the number of edges is larger than 2^32=4 billions, i.e., beyond the range of 4-byte integer. Since the CSR format of TOTEM uses an index number for each edge, TOTEM cannot even lad RMAT29 into main memory. In contrast, the slotted page format of GStream uses only vertex IDs and does not use any kinds of edge IDs, and thus GStream can load and process RMAT29 without a problem. In Figures 7(c)-(d), GStream still outperforms TOTME in all cases.It improves the performance up to 7.16 times for BFS (on RMAT28 using two GPUs), and up to 4.28 times for PageRank (on RMAT28 using two GPUs). We note that the performance gap between both methods becomes wider for large synthetic graphs. We also note that GStream succeeds in processing RMAT29, which is larger than the largest available real graph YahooWeb, in about 100 seconds by using two GPUs. Here, the graph processing speed is about 4GB/s since the size of RMAT is about 40GB, and we perform PageRank iteration 10 times for all experiments.

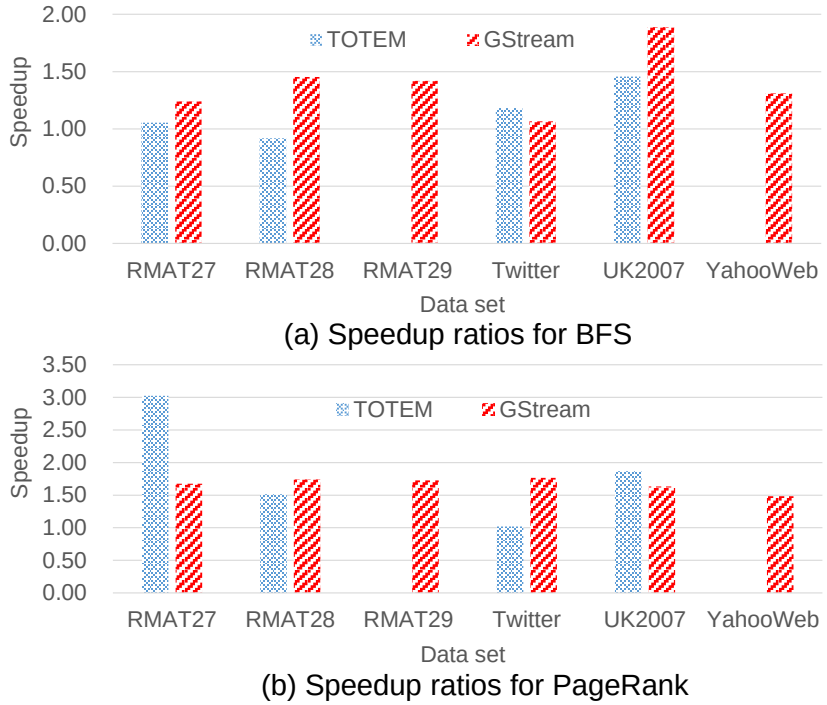
Real graphs. We compare the performance of GStream with the best performance of TOTEM for real graphs, where the ratio of the GPU portion to the CPU portion in TOTEM is summarized in Table 4. For YahooWeb, TOTEM cannot load data into main memory due to the same reason with RMAT29. In Figure 7(e)-(f), GStream outperforms TOTEM in all cases, except one (PageRank on Twitter using a single GPU). GStream improves the performance up to 7.64 times for BFS (on UK2007 using two GPUs), and up to 3.47 times for PageRank (on UK2007 using one GPU).

Table 4. Ratios of partition sizes in TOTEM (GPU%:CPU%).

		Twitter	UK2007	YahooWeb
one GPU	BFS	80:20	30:70	N/A
	PageRank	80:20	30:70	N/A
two GPUs	BFS	100:0	60:40	N/A
	PageRank	100:0	60:40	N/A

Scalability. We evaluate the speedup ratios of GStream when using two GPUs compared with that of TOTEM. Figures 8(a)-(b) show the speedup for BFS and for PageRank, respectively. In most cases, GStream shows higher speedup, and furthermore more stable speedup than TOTEM, as discussed in Section 3.3. For BFS, GStream shows the speedup ratios between 1.07 and 1.89, while TOTEM shows those of between 0.92 and 1.46, i.e., the performance is rather degraded in some cases. For

PageRank, GStream shows the fairly stable speedup ratios of between 1.63 and 1.77, while TOTEM shows the unpredictable speedup ratios of between 1.02 and 3.02. GStream shows more stable speedup ratios for PageRank than for BFS since there is almost no workload imbalance among GPUs in PageRank as discussed in Section 3.6.



(a) Speedup ratios for BFS

(b) Speedup ratios for PageRank

Figure 8. Speedup ratios when using two GPUs.

One exceptionally high speedup of TOTEM for PageRank on RMAT27 is due to a relatively bad performance of TOTEM when using one GPU. TOTEM can maintain the entire RMAT27 with two GPUs, but cannot do it with one GPU, as shown in Table 3. In terms of absolute performance, GStream still is faster than TOTEM in that case (for PageRank on RMAT27 using either one GPU or two GPUs), as we can see in Figure 7(c)-(d).

5.3 Characteristics of GStream

Figure 9 shows the performance of GStream while varying the number of streams for from RMAT26 to RMAT29. The performance increases steadily as the number of streams increases for all data sets. Even for BFS where the ratio of transfer time to kernel execution time is much smaller than 32, it does due to the reason explained in Section 3.3.

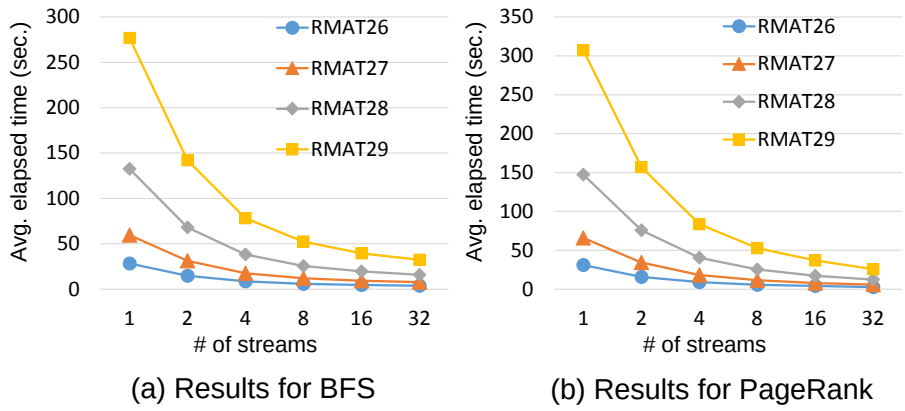


Figure 9. Performance when varying the number of streams.

Figure 10(a) shows the performance of GStream for BFS while varying the cache size from 32MB to 5,120MB, and Figure 10(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*. In Figure 10(b), the cache hit rate increases linearly according to the cache size and decreases linearly according to the size of topology data, as discussed in Section 3.4.

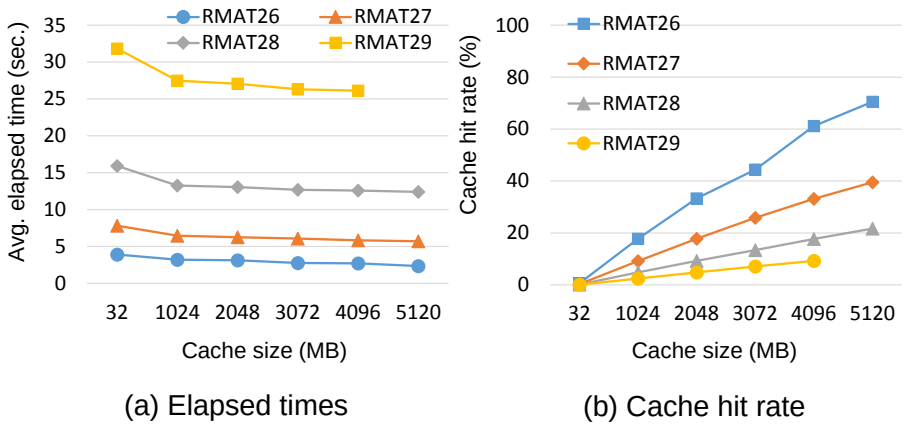


Figure 10. Effectiveness of caching for BFS.

Figure 11 shows the performance of GStream 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 to process each slotted page. The three strategies discussed in Section 4.1 show similar performance for very sparse graph of 1:4. However, the edge-centric strategy outperforms the

vertex-centric strategy largely for more dense graphs. The hybrid strategy improves the performance slightly (up to 6% for BFS and up to 24% for PageRank) compared with the edge-centric one.

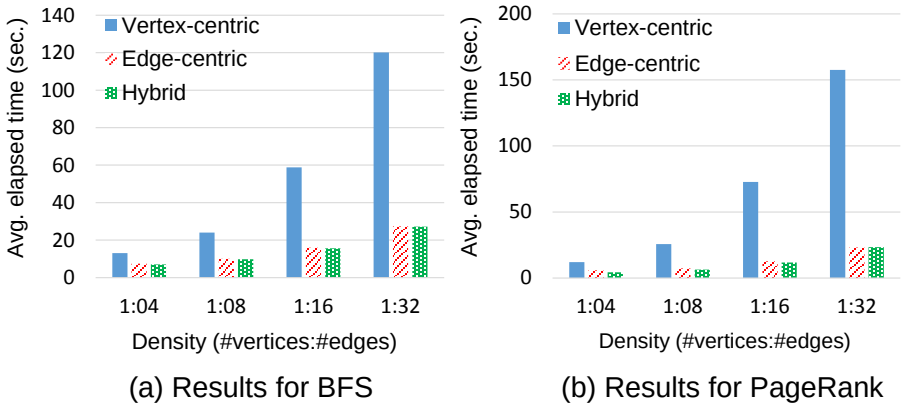


Figure 11. Performance when changing micro-level parallel processing techniques and graph density.

VI. RELATED WORK

There are a number of graph processing methods using GPUs on a single computer [6-7, 10, 12, 16, 20, 24]. The WVC method [12] 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 [18] 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 [24] proposes a programming framework that can simplify implementation of GPU programs for graph processing [20] 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 are lack of support for large-scale graphs that do not fit in the GPU's limited device memory. However, many technique addressed in the above work belong to micro-level parallel processing techniques and are orthogonal to our method GStream, and so they can be applied to processing each topology page. TOTEM [6-7] is 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. Though it can handle large-scale graphs, it still has many fundamental problems such as a large amount of synchronization overhead, lack of scalability in terms of the number of GPUs, and a limit on the size of graphs to process.

There are also a number of graph processing methods on a distributed systems of multiple computers [8, 13-15, 19, 22]. The representative methods include Pregel [19], GraphLab [15], PowerGraph[8], Trinity [22], PEGASUS [14] and GBase [13]. Pregel [19] follows the BSP message passing

model in which all vertex kernels run simultaneously in a sequence of so-called super-steps. GraphLab [15] allows a vertex kernel to be executed in asynchronous parallel on each vertex. PowerGraph [8] is basically similar to GraphLab, but it partitions graphs by exploiting the properties of power-law degree distributions of real graphs. However, there is no method yet that exploits GPUs for graph processing on a distributed systems, which would be an interesting topic for fast processing huge-scale graphs larger than the capacity of main memory of a single computer.

VI. CONCLUSIONS

Parallel graph processing on GPUs has been suffered from severe limitation on the graph size to process even though GPUs have massive computing power. In this paper, we have proposed a novel parallel graph processing method GStream that processes large-scale graphs larger than the size of GPU device memory very efficiently and is scalable in terms of both data size and the number of GPUs. It adopts the concept of nested-loop join combined with asynchronous GPU streams, where it keeps writable data in device memory and streams read-only data to GPUs. We have presented the detailed pseudo code of the GStream and its cost models. We have also presented the caching strategy for GStream and how to apply the fine-granular parallel processing strategies to our framework. Extensive experimental results with various synthetic and real data sets show that GStream consistently and significantly outperforms the state-of-the-art method TOTEM in terms of the absolute performance and the scalability. We believe that the concepts and techniques of GStream will be able to apply to the similar problems other than graph processing.

References

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

[2] TOTEM source code. https://github.com/netsyslab/Totem, 2014.

[3] Webspam-uk2007. http://barcelona.research.yahoo.net/webspam/datasets/uk2007/.

[4] Yahoo webscope. yahoo! altavista web page hyperlink connectivity graph. http://webscope.sandbox.yahoo.com.

[5] D. Chakrabarti, Y. Zhan, and C. Faloutsos. R-MAT: A recursive model for graph mining. In SDM, 2004.

[6] A. Gharaibeh, L. Beltrao Costa, E. Santos-Neto, and M. Ripeanu. A yoke of oxen and a thousand chickens for heavy lifting graph processing. In PACT, 2012.

[7] A. Gharaibeh, L. Beltrao Costa, E. Santos-Neto, and M. Ripeanu. Efficient Large-Scale Graph Processing on Hybrid CPU and GPU Systems. In CoRR, 2013.

[8] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. Powergraph: distributed graph-parallel computation on natural graphs. In OSDI, 2012.

[9] 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, 2013.

[10] P. Harish and P. J. Narayanan. Accelerating large graph algorithms on the gpu using cuda. In HiPC, 2007.

[11] G. M. Hector, J. D. Ullman, and J. Widom. Database Systems: The Complete Book (2nd Edition). Prentice Hall, 2008

[12] S. Hong, S. Kim, T. Oguntebi, and K. Olukotun. Accelerating CUDA graph algorithms at maximum warp. In PPoPP, 2011.

[13] U. Kang, H. Tong, J. Sun, C.-Y. Lin, and C. Faloutsos. Gbase: a scalable and general graph management system. In KDD, 2011.

[14] U. Kang, C. E. Tsourakakis, and C. Faloutsos. Pegasus: A peta-scale graph mining system implementation and observations. In ICdevice memory, 2009.

[15] 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. PVLDB, 2012.

[16] F. Khorasani, K. Vora, R. Gupta, and L. N. Bhuyan. CuSha: vertex-centric graph processing on GPUs. In HPDC, 2014.

[17] H. Kwak, C. Lee, H. Park, and S. B. Moon. What is twitter, a social network or a news media? In WWW, 2010.

[18] A. Kyrola, G. Blelloch, and C. Guestrin. Graphchi: large-scale graph computation on just a pc. In OSDI, 2012.

[19] 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, 2010.

[20] D. Merrill, G. Michael, and A. Grimshaw. Scalable GPU Graph Traversal. In PPoPP, 2012.

[21] S. Pai, M.J. Thazhuthaveetil, and R. Govindarajan, Improving GPGPU Concurrency with Elastic Kernels. In ASPLOS, 2013.

[22] B. Shao, H. Wang, and Y. Li. Trinity: a distributed graph engine on a memory cloud. In SIGMOD, 2013.

[23] L.G. Valiant. A bridging model for parallel computation. Communications of the ACM, 33(8), 1990.

[24] J. Zhong and B. He. Medusa: Simplified graph processing on GPUs. In TPDS, 2013.

요 약 문

GStream: GPU 에서 대규모 그래프 처리를 위한 스트리밍 방법

큰 규모 그래프를 위한 빠른 그래프 알고리즘 처리는 다양한 분야의 어플리케이션에 있어서 그래프가 인기를 얻게 되고 그래프의 규모가 급속히 증가함에 따라 점점 중요해지고 있다. 상대적으로 낮은 가격 대비 높은 계산 능력을 가지고 있는 GPU 의 대규모 병렬성을 이용하여 그래프 어플리케이션을 처리하기 위한 많은 시도가 이루어져 왔다. 그러나 현존하는 대부분의 방법들은 대개 불규칙적은 그래프 구조와 복잡한 그래프 처리 알고리즘 때문에 장치 메모리에 저장할 수 없는 큰 규모의 그래프를 처리하지 못했다. 가장 최신의 방법인 TOTEM 은 큰 규모의 그래프를 메인 메모리와 장치 메모리 부분적으로 나누어 저장하여 메인 메모리에 저장된 부분은 CPU 에 의해 계산되고, 장치 메모리에 저장된 부분은 GPU 에 계산되는 방법으로 처리한다. TOTEM 은 장치메모리 보다 큰 규모의 그래프 데이터를 처리하기는 하지만 아직 메인 메모리와 장치 메모리간의 동기화 오버헤드와 GPU 의 개수 및 데이터 규모에 따른 확장성의 부족과 같은 근본적인 문제를 가지고 있다. 우리는 빠르고 GPU 의 개수 및 데이터 규모에 대해 확장성을 가진 병렬 처리 방법인 GStream 을 제안한다. GStream 은 GPU 의 장치메모리보다 큰 규모의 그래프(예, 10 억개의 정점)를 중첩 루프 조인과 비동기적인 GPU 스트림을 이용함으로써 매우 효율적으로 처리한다. GStream 은 GPU 간에 균일한 작업 부하를 분산하는 방법으로 다수의 GPU 를 활용한다. 또한 스트리밍으로 복사되는 그래프 데이터를 캐시하는 방법으로 GPU 장치 메모리의 가용공간을 활용한다. 우리가 아는 범위에서 GStream 은 첫 번째 데이터 규모 및 GPU 의 개수에 있어서 확장성을 가지는 방법이다. 대규모의 실험 결과는 GStream 이 TOTEM 보다 합성 데이터와 실제 데이터에 대해서 절대적인 성능, 확장성 그리고 그래프 규모에 있어서 일관적이며 상당히 더 나은 결과를 내는 것을 보여준다.

핵심어: 그래프 처리, 대규모, GPU, 스트림

Acknowledgement

I would like to express my gratitude to all those who gave me the motivation to complete this thesis. Above all, I am deeply indebted to my supervisor Prof. Min-Soo, Kim, whose help, is advice and encouragement helped me in most of the research time and the writing of this thesis.

Also, I would like to express thanks for my colleagues from the Department of Information and Communication Engineering supported me in my research work. I want to thank them for all their help, support, interest and valuable hints. Especially, I would like to give my special thanks to my family and SHS whose patient love enabled me to complete this work.

CURRICULUM VITAE

Hyunseok Seo

05.01.1990

Education

- **Master of Science in Information & Communication Engineering, Mar. 2013 – Feb. 2015**
  DGIST (Daegu Gyeongbuk Institute of Science and Technology), Daegu, Korea

- **Bachelor of Engineering in Computer Engineering, Mar. 2009 – Feb. 2013**
  Hanbat National University, Daejeon, Korea

Work Experience

- **External Developer in Electronics and Telecommunications Research Institute (ETRI), Mar. 2012 – Jan. 2013**

  Future Internet Service Research Team

  Research and development of composite context based adaptive service path configuration technology

  Composite Context Management Function (CCMF) module development using Java

Professional Activities

- **High Performance Computing Summer School (HPCSS) @ UNIST, July 2013**
  Advanced course, UNIST, Ulsan, Korea

- **Computer Research Association (CRA), July 2011 – present**

  Large scale graph processing on GPUs using CUDA, June 2013 – present

  RFID barcode recognition android application, Mar. 2012 – Nov. 2012

  Image processing to recognize the object by using humanoid robot, July 2011 – Dec. 2011

  Medicine database design using JDBC and MyBatis, Oct. 2011 – Dec. 2011

- **Technology Business Academy, May 2011 – Sept. 2011**
  Hanbat National University Business Incubation Center, Daejeon, Korea

Honors and Awards

- **The Second Prize Award in Portfolio Contest, Dec. 2012**

  Hanbat National University, Daejeon, Korea

- **The Grand Prize Award in Academic Contest, Jan. 2012**

  Hanbat National University, Daejeon, Korea

- **The Second Prize Award in Portfolio Contest, Dec. 2011**

  Hanbat National University, Daejeon, Korea

- **The Participation Award in Intelligent SoC Robot War, Nov. 2011**

  HURO-Competition, SDIA, KAIST, Daejeon, Korea

Publication

- **Principles and Practice of Parallel Programming (PPoPP), Feb. 2015 - Poster session**

  "GStream: A Graph Streaming Processing Method for Large-scale Graphs on GPUs", Airport Marriott Waterfront, San Francisco, USA

- **Ubiquitous Computing and Web Information Technology (UCWIT), Dec. 2013**

  "A performance comparison between Matrix format and adjacency list format for GPU based PageRank", Kyungpook National University, Daegu, Korea