Out-of-core GPU Memory Management for MapReduce-based Large-scale Graph Processing

Koichi Shirahata*†, Hitoshi Sato*†, Satoshi Matsuoka*†
*Tokyo Institute of Technology, Tokyo, Japan
†CREST, Japan Science and Technology Agency, Tokyo, Japan

***Abstract*—GPUs can accelerate edge scan performance of graph processing applications; however, the capacity of device memory on GPUs limits the size of graph to process. Efficient techniques to handle GPU memory overflows, including overflow detection and performance analysis in large-scale systems, are not well investigated. To address the problem, we propose a MapReduce-based out-of-core GPU memory management technique for processing large-scale graph applications on heterogeneous GPU-based supercomputers. Our proposed technique automatically handles memory overflows from GPUs by dynamically dividing graph data into multiple chunks and overlaps CPU-GPU data transfer and computation on GPUs as much as possible. Our experimental results on TSUBAME2.5 using 1024 nodes (12288 CPU cores, 3072 GPUs) exhibit that our GPU-based implementation performs 2.10x faster than running on CPU when graph data size does not fit on GPUs. We also study the performance characteristics of our proposed out-of-core GPU memory management technique, including application's performance and power efficiency of scale-up and scale-out approaches in terms of the number of GPUs.**

Keywords*—*Large-scale Graph Processing; GPGPU; MapReduce; Out-of-core Algorithms; Big Data Applications

I. Introduction

Recently extremely large-scale graphs emerge in various application fields, such as health care, social networks, system biology, and electric power grids, etc., which typically consist of millions to trillions of vertices and edges. These large-scale graphs require fast and scalable analysis using HPC technologies, by fully exploiting performance of recent supercomputers. Moreover, large-scale graphs attract attention to the Graph500 benchmark [1], which ranks supercomputers by executing a large-scale graph search problem as an instance of data-intensive supercomputing applications. On the other hand, modern supercomputers employ commodity graphics processing unit (GPU) in addition to general purpose CPU, since GPU-based heterogeneous supercomputers continue to attract attention due to their high peak performance and high power efficiency. In practice, several existing GPU-based graph processing techniques have also shown that the GPUs accelerate the performance on several graph applications, such as Breadth-First Search (BFS) [2], PageRank [3], etc. In our earlier work [4], we proposed a distributed multi-GPU implementation of a MapReduce-based graph processing, where we found that our multi-GPU-based PageRank implementation scales well compared with the multi-core CPU-based implementation on the TSUBAME2.0 supercomputer [5] using 256 nodes and 768 GPUs.

Although GPU-based heterogeneous supercomputers are suitable for graph applications, the capacity of device memory on GPUs limits scalable large-scale graph processing, since the GPUs typically have smaller memory capacity than the CPU host. For example, the TSUBAME2.5 supercomputer [6] employs 1408 compute nodes, each of which equips 3 GPU devices and 2 CPU sockets, where the capacity of device memory on each GPU has 6GB, while the CPU host memory has 54GB. Thus, in order to process larger-scale graphs whose size exceeds the capacity of GPU memory, data management techniques for handling GPU memory overflows are required. However, such out-of-core GPU data management techniques with detailed performance studies for large-scale graph processing are not well investigated. Furthermore, even if we apply the out-of-core GPU data management techniques, execution approaches whether we use only the device memory on GPUs (scale-out) or offload partial graph data to the secondary CPU memory (scale-up) on a multi-node environment are considered another important issue in terms of graph application's performance and its power efficiency.

In order to address these problems, we propose an out-of-core GPU data management technique for GPU-MapReduce-based graph applications. Our proposed technique automatically handles GPU memory overflows by dividing graph data into multiple chunks and hides CPU-GPU data transfer overheads by overlapping computations on GPUs and CPU-GPU data transfers. We also investigate the balance of the scale-up and the scale-out approaches, in terms of the density of GPUs and processing graph data size per node, by comparing application's performance and power efficiency.

We conduct experiments on TSUBAME2.5 using up to 1024 nodes (12288 CPU cores, 3072 GPU devices). The results exhibit that our GPU-based implementation performs 2.81 GE/s (billion edges per second) on a large-scale graph with 2^{34} (17.18 billion) vertices and 2^{38} (274.9 billion) edges. These results indicate that our GPU-based implementation performs 2.10x faster than the multi-core CPU-based implementation even when the graph data size exceeds the capacity on multiple GPUs. We also show that the scale-up approach outperforms the scale-out approach by 1.71x in power efficiency on the TSUBAME-KFC supercomputer.

Here we describe a summary of contributions of our work:

- We propose an out-of-core GPU data management technique for GPU-based-MapReduce-based large-scale graph processing.
- We demonstrate the scalability of our proposed technique on heterogeneous large-scale GPU-based supercomputers by utilizing several optimization techniques.

- We investigate the balance of scale-up and scale-out strategies, i.e., the density of GPUs and processing graph data per node, whose results suggest that the scale-up approach may help power-efficient graph processing rather than the simple scale-out approach.

II. Graph Processing on GPUs

Modern supercomputers employ commodity graphics processing unit (GPU) in addition to general purpose CPU, since GPU-based heterogeneous supercomputers attract attention due to their high peak performance and high power efficiency. GPU-based techniques are also applied to various graph applications to accelerate edge scan performance; however, the size of graphs in most of the existing GPU-based graph processing techniques is limited by the capacity of GPU memory. In this section, we describe the existing graph processing techniques on GPUs and point out the issues for processing large-scale graphs on GPUs.

A. Existing Graph Processing Techniques on GPUs

Several existing GPU-based graph processing techniques have shown that GPU accelerates performance on several graph algorithms, such as BFS [2], [7], [8], [9], PageRank [3], etc. Several graph processing frameworks are also accelerated by using a single GPU [10], [11], [12]; however, most of these efforts focus on algorithms for a single GPU. Thus, the size of processing graphs in these algorithms reaches around 10 million vertices and 60 million edges.

Several efforts focus on the use of multiple GPUs on a single node for BFS [13], [14], [15], PageRank [15], [16] etc., in which the size of graphs to process reaches around 50 million vertices and 100 million edges. However, these work do not consider communication between multiple nodes nor show the scalability when the graph data exceeds on the CPU memory capacity on a single node.

As for the efforts on multi-node multi-GPU environments, GPU-based implementations of sparse matrix vector multiplication for PageRank [17], [18] and BFS [19] have been proposed. We have proposed a multi-GPU implementation of a MapReduce-based PageRank algorithm for large-scale computing environments [4], which processes a graph with 1.1 billion vertices and 17.2 billion edges using 768 GPUs. However, these implementations cannot handle GPU memory overflows due to heavy CPU-GPU data transfer overheads.

B. Issues for Processing Large-scale Graphs on GPUs

One of the significant issues for processing large graphs on GPUs is considered that how to manage graph data whose size exceeds the capacity of GPU memory with minimal performance overheads. As explained in the previous section, the GPU memory generally has the smaller capacity than the CPU memory, and computation on GPUs requires to transfer data between the CPU memory and the GPU memory. Thus, when we naively apply graph algorithms to GPUs, data transfers dominantly disturb efficient graph processing. In particular, when the size of the graphs exceeds the capacity of the device memory on GPUs, the number of data transfers drastically increases for executing dependent graph kernels.

We can certainly overcome the GPU capacity limitation problem by using multi-node multi-GPU environments. Indeed, several existing efforts have shown good weak-scaling performance of graph processing on GPU-based large-scale environments; however, these techniques still have the limit on the size of the graphs below the device memory capacity on GPUs. Although out-of-core GPU memory management techniques may help the problem by utilizing secondary host memory volumes, best approaches with detailed performance studies whether we use only device memory on the GPUs (scale-out) or offload partial graph data to the secondary CPU memory (scale-up) on a multi-node environment are not well investigated in terms of graph application's performance and power efficiency. Moreover, optimization techniques for out-of-core GPU memory management techniques to achieve good weak scalability on large-scale environments should be investigated, since graph algorithms generally include irregular data accesses to sparse data sets, whose situations introduce significant performance overheads and disturb scalable large-scale graph processing.

III. Related Work

A. Out-of-core CPU processing

There are several work on out-of-core graph processing on CPU. As for CPU-based graph processing on a single node, several techniques, such as a sequential I/O optimization [20], a data placement optimization [21], and a data prefetch optimization [22], have been proposed. These work focus on the utilization of a single node. Thus, distributed computing environments are not supported. Pearce et al. [23] have proposed an out-of-core CPU large-scale graph processing technique for distributed computing environments. Their technique introduces a graph partitioning strategy and applies to their multithreaded algorithm using distributed external memory; however, this algorithm cannot be straightforwardly applicable to GPUs, since this algorithm is highly designed for utilizing multi-core CPUs. The MapReduce [24] programming model has been proposed for processing big data applications with automatic memory/storage hierarchy encapsulation, and Hadoop [25] is one of the widely used MapReduce implementation. MR-MPI [26] is a MPI-based MapReduce implementation on CPU, which employs an out-of-core processing technique including in the sort phase after inter-node data exchanges. These MapReduce implementations are designed for CPU-based distributed environments, while our work focuses on GPU-based environments.

B. Out-of-core GPU Processing

There are several work on out-of-core GPU processing algorithms in a wide range of application fields, such as BFS [9], stencil [27], rendering [28], etc. These work have shown GPU accelerations by using out-of-core techniques; however, the scope of these applications are limited on specific algorithms. Out-of-core GPU sorting algorithms, such as a sample-based sorting [29] and a merge-based sorting [30], have been also studied; however, these algorithms are designed for a single node execution. These work also have not well investigated on load balancing issues for highly skewed data such as real world graphs. There also exists work on I/O issues from a GPU to filesystems [31]; however, they have not conducted experiments on realistic large-scale applications such as graph processing.

C. MapReduce on GPUs

The MapReduce model can provide out-of-core processing with simple application interfaces. There exists a generalized graph processing algorithm for the MapReduce model called GIM-V (we explain the details in Section IV) and its Hadoop-based implementation [32]. However, the implementation does not show good performance due to heavy overheads derived from the Hadoop framework. In our earlier work [4], we proposed a distributed multi-GPU-MapReduce-based graph processing implementation, and found that our multi-GPU-based PageRank implementation outperforms the Hadoop-based implementation considerably on TSUBAME2.0. GPMR [33] is a multi-GPU MapReduce library supporting out-of-core GPU execution on distributed computing environments. However, the sort phase in GPMR is executed on CPUs when the size of input data exceeds the capacity of the GPU memory, instead of executing on GPUs. Besides, the performance studies on CPU vs. GPU comparison have not been sufficiently conducted, especially in the out-of-core situation.

IV. MapReduce-based Graph Processing: GIM-V

GIM-V (Generalized Iterative Matrix-Vector multiplication) [32] is a general expression of matrix-vector multiplication with iterative operations for MapReduce-based large-scale graph processing. Let $M = (m_{i,j})$ be a matrix of size $n \times n$, and $v = (v_i)$ be a vector of size n, where $i, j \in \{1, ..., n\}$. Matrix-vector multiplication is described as follows:

$$M \times v = v' \text{ where } v'_i = \sum_{j=1}^{n} m_{i,j} v_j$$

Here the above expression is described by using three operators: *combine2*, *combineAll*, and *assign*:

combine2:	Multiply $m_{i,j}$ and v_j.
combineAll:	Sum the results of *combine2* for vertex i.
assign:	Update v_i to the new result v'_i.

By introducing the operator $\times_G$, we can define the GIM-V algorithm as follows:

$$\begin{aligned} &v' = M \times_G v \\ &\text{where } v'_i = \mathit{assign}(v_i, \mathit{combineAll}_i(\{x_j \mid j = 1..n, \\ &\qquad\qquad \text{and } x_j = \mathit{combine2}(m_{i,j}, v_j)\})) \end{aligned}$$

We iterate the above operation until satisfying a convergence condition defined by graph algorithms such as PageRank, Random Walk, and Connected Component, etc. We can describe these graph algorithms by defining the above three operators.

As an example, here we describe the PageRank algorithm [34], which is a well-known algorithm for scoring relative importance in web pages, Let p be a PageRank eigenvector of n web pages; the PageRank algorithm satisfies the following characteristic equation:

$$p = (cE^T + (1 - c)U)p$$

where c denotes a dumping factor, set to 0.85 in typical configuration, E denotes a row-normalized adjacency matrix, and U denotes a matrix with all elements set to $1/n$. In order to acquire the next PageRank eigenvector p^{next}, we initialize p^{cur} and set all the elements to $1/n$, then we calculate $p^{next} = (cE^T + (1-c)U)p^{cur}$. We continue the multiplication until p converges. The three operations are defined as follows:

$$\mathit{combine2}(m_{i,j}, v_j) = c \times m_{i,j} \times v_j$$

$$\mathit{combineAll}_i(x_1, \ldots, x_n) = \frac{(1-c)}{n} + \sum_{j=1}^{n} x_j$$

$$\mathit{assign}(v_i, v_{new}) = v_{new}$$

Algorithm 1 GIM-V Stage 1.

```
Require: Matrix $M = \{(id_{src}, (id_{dst}, mval))\}$,
         Vector $V = \{(id, vval)\}$
Ensure: Partial vector $V' = \{(id_{src}, \texttt{combine2}(mval, vval))\}$
 1: Stage1-Map(Key k, Value v);
 2: if $(k, v)$ is of type $V$ then
 3:     Output($k$,$v$);                //(k: $id$, v: $vval$)
 4: else if $(k, v)$ is of type $M$ then
 5:     $(id_{dst}, mval) \leftarrow v$;
 6:     Output($id_{dst}, (k, mval)$);      //(k: $id_{src}$)
 7: end if
 8: Stage1-Reduce(Key k, Value v[1..m]);
 9: $saved\_kv \leftarrow [\,]$;
10: $saved\_v \leftarrow [\,]$;
11: for all $v \in v[1..m]$ do
12:     if $(k, v)$ is of type $V$ then
13:         $saved\_v \leftarrow v$;
14:         Output($k$, (“$self$”, $saved\_v$));
15:     else if $(k, v)$ is of type $M$ then
16:         Add $v$ to $saved\_kv$        //(v:($id_{src}, mval$))
17:     end if
18: end for
19: for all $(id'_{src}, mval' \in saved\_kv)$ do
20:     Output($id'_{src}$, (“$others$”, combine2($mval', saved\_v$)))
21: end for
```

Algorithm 2 GIM-V Stage 2.

```
Require: Partial vector $V' = \{(id_{src}, vval')\}$
Ensure: Result vector $V = \{(id_{src}, vval)\}$
 1: Stage2-Map(Key k, Value v);
 2: Output($k$, $v$);
 3: Stage2-Reduce(Key k, Value v[1..m]);
 4: $others\_v \leftarrow [\,]$;
 5: $self\_v \leftarrow [\,]$;
 6: for all $v \in v[1..m]$ do
 7:     $(tag, v') \leftarrow v$;
 8:     if $tag ==$ “$self$” then
 9:         $self\_v \leftarrow v'$;
10:     else if $tag ==$ “$others$” then
11:         Add $v'$ to $others\_v$;
12:     end if
13: end for
14: Output($k$, assign($self\_v$, combineAll$_k$($others\_v$)));
```

The GIM-V algorithm can be implemented using two MapReduce stages: GIM-V Stage1 and Stage2, whose pseudo codes are shown in Algorithm 1 and 2. The GIM-V Stage1 per-

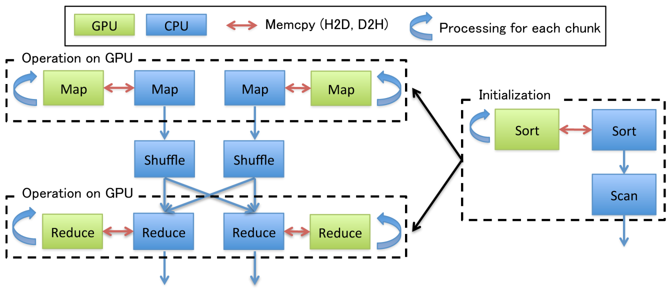

Fig. 1. Overview of our out-of-core multi-GPU MapReduce framework. Blue boxes represent operations called on CPU, and green boxes represent operations running on GPUs. The dashed boxes on the left side represent operations initialized by the dashed box on the right side.

forms the *combine2* operation by combining m_{ij} of the matrix M and v_j of the vector v, and outputs key-value pairs, where the key denotes the source vertex id i and the value denotes the partially combined result $x_j = combine2(m_{ij}, v_j)$. Then the output of the GIM-V Stage1 is forwarded to the input of the GIM-V Stage2. The GIM-V Stage2 combines all partial results from the GIM-V Stage1 by applying $combineAll_i(x_j \mid j = 1 \ldots n)$, and assigns the new vector v_{new} to the old vector v_i by applying $assign(v_i, combineAll_i(x_j \mid j = 1 \ldots n))$. These two MapReduce operations are iterated until the application-specific convergence criterion is met.

V. Out-of-core GPU Memory Management for GPU-MapReduce-based Graph Processing

A. Basic Idea

Our out-of-core GPU memory management technique is designed on top of the MapReduce model, since the MapReduce model can transparently encapsulate memory hierarchies by providing automatic memory management from the system. Before describing our proposed data management technique, we introduce the target multi-GPU MapReduce framework.

Figure 1 shows an overview of the framework. The basic architecture of the framework remains the same as our previous proposal [4], but we use a different implementation here. We firstly read key-value pairs as input data from a distributed file system to CPU memory on multiple nodes and keep the data on CPU memory. Next we sort and reorder the input key-value pairs by key to obtain a set of values for a key. Note that we may skip the sorting process for *Map* operations. Then users call *Map*, *Shuffle*, or *Reduce* operations based on user-specific application workflow. When the *Map* or the *Reduce* operations are called, the input data are processed on GPUs inside the framework with user-provided operations. When the *Shuffle* operations are called, the input data are exchanged between multiple nodes based on system-provided or user-provided splitters by using MPI all-to-all communications. Finally, output data are transferred onto the CPU memory on each node. Our framework is flexibly designed so that the users can define multiple *Map* and *Reduce* operations and call the *Map*, *Shuffle*, and *Reduce* operations in an arbitrary order. The users can also write applications with iterative computations by writing loop syntax with user-provided convergence criteria.

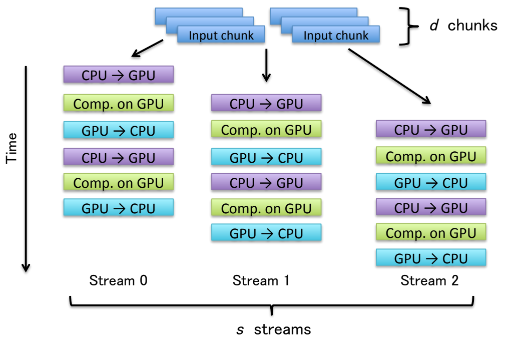

Fig. 2. Overview of our stream-based out-of-core GPU memory management. Upper blue bars represent input chunks on CPU memory to be processed. Purple bars represent data transfers from CPU memory to GPU memory, green bars represent computations on GPUs, and light blue bars represent data transfers from GPU memory to CPU memory, respectively.

The above technique includes a significant limitation that the framework cannot handle GPU memory overflows. Thus we simply extend the above framework based on two straightforward ideas, streaming processing on GPUs and GPU-based external sorting. By dividing input data into multiple chunks and by processing each chunk one by one in a stream, we apply overlapping techniques between computation and data transfer for hiding data transfer overheads as much as possible.

B. Stream-based GPU MapReduce processing

Figure 2 shows an overview of our streaming processing technique for GPU-based *Map* and *Reduce* operations. In order to optimize data transfer between CPU and GPU, we overlap three operations: data transfer from CPU memory to GPU memory, the *Map* and *Reduce* operations on GPU, and data transfer from GPU memory to CPU memory, otherwise we suffer additional CPU-GPU data transfer overheads for each *Map* or *Reduce* operation. Note that our stream-based memory management provides additional benefits that hide CPU-GPU data transfer from the *Map* and *Reduce* operations on the GPU even if the size of input data fits the capacity of the GPU memory. The detailed instructions of our stream-based CPU-GPU memory management technique are shown as follows:

STEP1: Divide input key-value data into d chunks evenly, where d denotes the number of chunks. We determine the number of chunks dynamically so that each chunk fits on the GPU memory.

STEP2: Create s CUDA streams, where s denotes the number of streams, and allocate s buffers on GPU for the chunks of the input key-value data; a single buffer is linked to a single CUDA stream.

STEP3: Repeat streaming processing d times; transferring a chunk of the input key-value data from CPU to a buffer on GPU, running the *Map* and *Reduce* operations on GPU, and transferring output from the buffer on GPU to CPU. These three operations are overlapped using asynchronous function calls (i.e. *cudaMemcpyAsync* function with pinned memory).

We set the s parameter to three by default in order to overlap the above three operations. We dynamically update the d

parameter to fit the size of input data chunks on the capacity of GPU device memory.

C. Out-of-core GPU Sorting

We introduce a GPU sorting implementation to the framework for handling GPU memory overflows. The implementation consists of a combination of existing GPU-based out-of-core and in-core algorithms. As for out-of-core GPU sorting, we employ an existing sample-based out-of-core sorting algorithm for GPUs proposed by Ye et al. [29], while as for in-core GPU sorting, we employ the radix sort algorithm based on Thrust library [35]. Out-of-core GPU sorting is conducted when the size of input data exceeds the GPU memory capacity. Otherwise, in-core GPU sorting is conducted.

Figure 3 shows an overview of the out-of-core GPU sorting algorithm. Sample-based parallel sorting uses $t-1$ samples as splitters to partition the input data set into several data chunks, where t denotes the number of sample points. The chunks can be put on GPU memory by considering the size of chunks and the capacity of GPU memory. We determine the number of chunks dynamically by checking the available amount of memory and the input data size at the beginning. If the input data is too large to fit on the GPU, our framework divides the input data into d chunks based on the available GPU memory capacity and the input data size. The detailed instructions of the out-of-core GPU sorting algorithm are shown as follows:

STEP1: Randomly select c keys as sample candidates from input keys on CPU host memory.
STEP2: Sort the c sample candidates. Then, pick the $(k+1)\cdot c/d^{\,th}$ sample points and set the points to $t[k]$, where $k \in [0, d-1]$. Here d denotes the number of chunks. We set $t[d-1]$ to the maximum limit value on the host memory.
STEP3: Divide the input data set into d chunks on the host, each of which contains n/d elements evenly, where n denotes the number of input keys.
STEP4: Copy each chunk onto GPU memory, sort each chunk using the in-core sorting algorithm, and split each chunk into d buckets using splitters based on the sample points on the GPU.
STEP5: Swap the buckets among chunks on the host, so that elements in the $(i+1)^{th}$ chunk are no smaller than those in the i^{th} chunk.
STEP6: Copy each chunk onto the GPU and sort one by one using the in-core sorting algorithm on the GPU.

We straightforwardly use the GPU-based radix sorting as the in-core GPU sorting algorithm. Our out-of-core sorting algorithm differs from the existing out-of-core GPU sorting proposed by Ye al. [29] in that we present less CPU-GPU data transfer overheads by simplifying the data dividing strategy than the existing algorithm, since we observe good load balance when we set c to larger numbers than one thousand.

We implement a stream-based overlapping feature for GPU sorting and CPU-GPU data transfers, whose instructions are shown in Step 4 and 6. The Thrust library uses default CUDA stream and does not presently have a mechanism for controlling execution steams. In order to overlap with

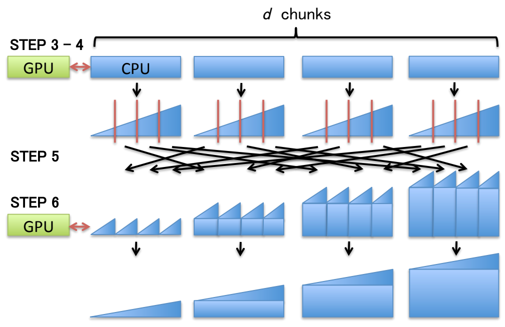

Fig. 3. Overview of the out-of-core GPU sorting algorithm. Blue boxes represent operations called on CPU and green boxes represent operations running on GPUs. Red vertical bars represent splitters based on sample points.

the default stream, we create multiple streams by *cudaStreamCreateWithFlags* with *cudaStreamNonBlocking* flag, a new feature enabled from CUDA 5.0. *cudaStreamNonBlocking* flag enables overlapping with default stream. In Step 5, we also implement pointer-based swapping algorithm with low-overheads. Pointers of buckets are swapped instead of the payload.

D. Optimizations

In order to achieve good weak-scaling performance on large-scale GPU-based heterogeneous supercomputers, we apply several optimization techniques to the framework with our out-of-core GPU memory management technique. Here we describe the details of the optimization techniques.

1) Data Structure: We employ a compact data structure similar to CSR (Compressed Sparse Row) for sparse matrix formats, which consists of an array of unique keys, values, and indices of first values for unique keys, for compressing redundant data and for achieving efficient *Map* and *Reduce* processing. For instance, if a Kronecker graph in the Graph500 benchmark is given as input data, we can compress duplicate keys to around $1/16$, since the graph includes 16 edges per vertex on average. We firstly reorder input key-value pairs by using the out-of-core GPU sorting algorithm. Then we apply the scan (prefix sum) operation to the sorted keys in order to calculate indices of first values for unique keys. Finally, we compact the duplicated keys by using the unique operation.

2) Shuffle: We implement *Shuffle* operation for redistributing intermediate data onto each node based on system-provided or user-defined splitter function. We implement range-based and hash-based splitters as system-provided splitters. We provide a default splitter as the range-based splitter, where each node takes charge of a range of the number of data. Instead users can also implement customized splitters. We observe the range-based splitter performs good load balance for skewed graphs generated in the same way as the Graph500 benchmark by randomizing vertex indices. Although load balance depends on the input graph structure, our *Shuffle* operation can extend to other splitters by customization according to the input graph structure.

3) Thread Assignment Policy on GPUs: We apply a thread assignment optimization on GPUs for handling the skew of vertex degrees on large-scale graph processing. We

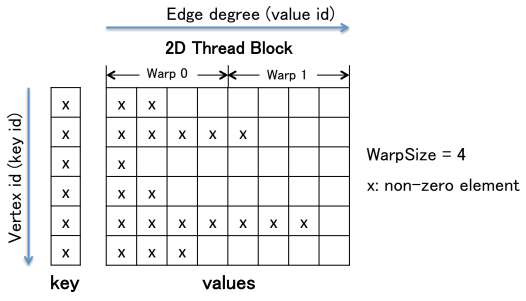

Fig. 4. Warp-based thread assignment onto 2D thread block on GPU. The mesh in the left side represents keys, and the mesh in the right side represents values corresponding to each key. Each warp is assigned to a portion of values corresponding to a key. Multiple warps are assigned to values whose length are larger than the warp size.

consider the following three strategies for assigning threads onto vertices and edges.

1) **Thread-based Assignment:** Assign one thread per vertex.
2) **Warp-based Assignment:** Assign one warp per vertex. The warp size on recent GPUs is set to 32.
3) **Thread Block-based Assignment:** Assign one thread block per vertex. The thread block size on recent GPUs (e.g. NVIDIA Tesla K20X) is set to 1024.

The strategies 2) and 3) are expected to achieve good performance on GPUs by utilizing massive amounts of threads; however, these strategies require to write CUDA-specific descriptions, such as `threadIdx`, `blockDim` etc., in the user-defined *Map* and *Reduce* operations. On the other hand, the strategy 1) can work on both CPUs and GPUs without any special descriptions. We employ the strategy 2) since the warp size is expected to be close to the average number of edges per vertex for wide range of graphs. For example, in graphs used in the Graph500 benchmark, the average number of edges per vertex is set to 16. As another example in real world graphs, the average number of edges per vertex in the Facebook friend network reaches around 130. For graphs with a large average number of edges, the strategy 3) is expected to achieve good performance. We set the thread block size as $(max\_tbs/ws, ws, 1)$ and the grid size as $(nv/blockDim.x, 1, 1)$ for the strategy 2), where max_tbs denotes the maximum number of threads per thread block, ws denotes the warp size, and nv denotes the number of vertices per GPU.

E. Implementation of GIM-V-based Graph Algorithm on GPU

We demonstrate an implementation of the PageRank algorithm on top of the GPU-based MapReduce framework with our proposed out-of-core GPU memory management technique. We implement two stages of MapReduce (*Map1-Reduce1* and *Map2-Reduce2* phases) based on the GIM-V algorithm explained in Section IV. First, *Map1* phase simply passes input key-value pairs to *Reduce1* phase. Next, the *Reduce1* phase conducts the *combine2* operation. Then, *Map2* phase simply passes the results of key-value pairs to *Reduce2* phase. Finally, the *Reduce2* phase conducts the *combineAll* and *assign* operations. In the *Reduce1* and *Reduce2* phases, we apply the warp-based thread assignments onto key-value scans: lines 11 to 18 and lines 19 to 21 in Algorithm 1 for the *Reduce1*, and lines 6 to 13 in Algorithm 2 for the *Reduce2*. We use shared memory for efficient warp-based key-value scans. We also apply warp shuffle operations to the *combineAll* operations for fast reduction. The warp shuffle operation is a new feature of the NVIDIA Kepler compute architecture.

VI. Experiments

In order to understand the efficiency of our out-of-core GPU memory management technique for GPU-MapReduce-based graph processing, we run a PageRank application based on the GIM-V algorithm on the TSUBAME2.5 supercomputer [6]. TSUBAME2.5 mainly consists of 1408 compute nodes, each of which has 2 sockets of Intel Xeon X5670 (Westmere EP, 2.93GHz, 6 cores) CPU, 54GB of DDR3 main memory, 3 devices of NVIDIA Tesla K20X GPU with 6GB of discrete GDDR5 memory connected to PCI-Express 2.0 $\times$ 16 buses, and 2 cards of QDR InfiniBand HBA (40Gbps) connected to the dual rail interconnect network with full bisection fat tree, and runs on SUSE Linux Enterprise 11 SP1. We use up to 1024 compute nodes of TSUBAME2.5 in the experiments. We use Kronecker graphs generated in the same way as Graph500 benchmark, using the recursive matrix (R-MAT) procedure with the following initiator parameters: (A, B, C, D) = (0.57, 0.19, 0.19, 0.05) and an average vertex degree of 16. We describe the size of the graphs as SCALE, the logarithm base two of their number of vertices. We use Open MPI 1.4.2 with GNU GCC 4.3.4 for the MPI implementation, and CUDA driver 5.0, CUDA runtime 5.0, and thrust 1.7.0 for the GPU implementation.

A. Comparison with CPU-based implementation

We compare our proposed GPU-based implementation with a CPU-based implementation in order to investigate the efficiency of GPU acceleration when the size of graph exceeds the capacity of device memory on GPUs. In order to make fair comparisons, we extend our GPU-based implementation to support multi-node multi-CPU environments as well. Our implementation employs a hybrid parallelization technique using MPI and OpenMP. MPI is used for parallelization between compute nodes (or processes) in the same way as our GPU-based implementation, whereas OpenMP is used for parallelization of *Map*, *Reduce*, and *Sort* operations inside a single node (or process). Our implementation parallelizes the *Map* and *Reduce* operations in a straightforward manner by using a simple fork-join model. In the *Sort* operation, we use OpenMP-based parallel sorting in the Thrust library. We use Thrust's OpenMP sorting instead of parallel STL sorting, since parallel STL sorting is not compatible with the CUDA compiler which we use in the CPU-based implementation.

Figure 5 shows the results of the weak-scaling performance of our CPU- and GPU-based implementations on TSUBAME2.5, where the x axis denotes the number of compute nodes and the y axis denotes the performance in ME/s (million edges per second) in each stage. Each node has the constant problem size: SCALE 23 for running on 1 CPU and 1 GPU and SCALE 24 for running on 2 CPUs, 2 GPUs, and 3 GPUs. Note that the size of graphs in the configurations exceeds the capacity of device memory on GPUs. For example, the size of a SCALE 23 graph exceeds the capacity of device memory

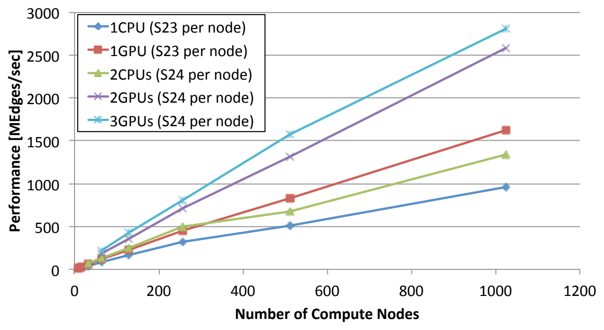

Fig. 5. Results of weak scaling performance, where SCALE 23 for running on 1 CPU and 1 GPU per node and SCALE 24 for running on 2 CPUs, 2 GPUs, and 3 GPUs per node.

on a GPU, and the size of a SCALE 24 graph also exceeds the aggregate capacity of device memory on 2 and 3 GPUs. Here we describe the GPU-based implementation as *nGPU(s)*, where *n* denotes the number of GPU devices per node, and the CPU-based implementation as *mCPU(s)*, where m denotes the number of CPU sockets per node. We use up to 1024 nodes in both *nGPU(s)* and *mCPU(s)* experiments; we vary the number of GPUs per node from 1 to 3, while we use 12 threads per node using 1 or 2 socket(s) in *mCPU(s)*. We see that our implementation on *3GPUs* achieves 2.81 GE/s (billion edges per second) on SCALE 34 on 1024 nodes (12288 CPU cores and 3072 GPUs). The results also exhibit 2.10x performance improvement compared with *2CPUs* on SCALE 34 on 1024 nodes.

Figure 6 shows the performance breakdown on SCALE 31 on 256 nodes, where the y axis denotes the elapsed time in milliseconds. We divide a single GIM-V iteration into five phases; *Map*, *Shuffle*, *Reduce*, *Sort*, and *Others*. *Map* and *Reduce* phases include the time for *Map* and *Reduce* kernel executions and CPU-GPU data transfer. *Shuffle* phase includes the time for inter-node data transfer and its preparation. *Sort* phase includes the time for sorting in each *Map*, *Shuffle*, and Reduce phase. *Others* includes the time for the rest of the *Map*, *Shuffle*, *Reduce*, and *Sort* phases. The results exhibit that the elapsed times for *Map*, *Reduce*, and *Sort* phases in *3GPUs* achieve 1.41x, 1.49x, and 4.95x faster than those on *2CPUs* respectively. The reason for this performance improvement is considered that our implementation hides CPU-GPU data transfer overheads efficiently in *Map*, *Reduce*, and *Sort* phases by stream-based asynchronous computation.

We analyze further breakdown of *Map* and *Reduce* phases in a single GIM-V iteration. Figure 7 shows the results, where *Map n* and *Reduce n* denotes each *Map* or *Reduce* phase in GIM-V Stage *n*. As we see in Algorithm 1 and 2 in Section IV, *Map1* and *Map2* operations only pass input vertices or edges data to the next phase, *Reduce1* operation combines a vertex and connecting edges for all vertices and passes to the next phase, and *Reduce2* operation combines all edges connecting to a vertex into an updated vertex for all vertices. Thus, we expect *Reduce1* and *Reduce2* operations to be accelerated compared to *Map1* or *Map2* operations by using GPUs, since *Reduce1* and *Reduce2* operations include actual computation kernels as opposed to *Map1* and *Map2*

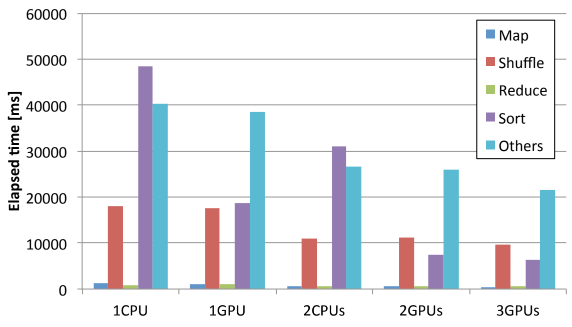

Fig. 6. Results of performance breakdown on SCALE 31 using 256 nodes.

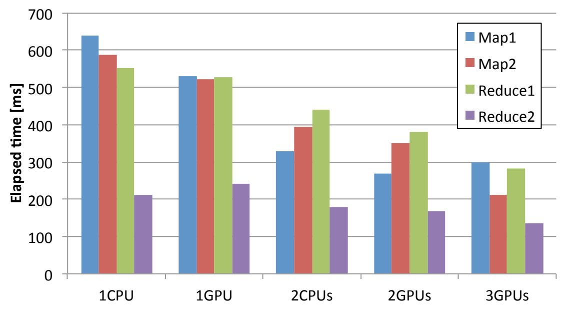

Fig. 7. Results of *Map* and *Reduce* phases on SCALE 31 using 256 nodes.

operations. The results in Figure 7 indicates that *Map1* and *Map2* operations are accelerated 1.41x, *Reduce1* operation is accelerated 1.56x, and *Reduce2* operation are accelerated 1.33x respectively by using 3 GPUs per node compared with 2 CPUs per node. As expected, *Reduce1* operation is more accelerated than *Map1* and *Map2* phases; however, we also see that *Reduce2* operation is not accelerated as much as the other phases. We consider the result comes from not fully overlapping CPU-GPU communication, since the computation time in *Reduce2* operation is not sufficiently large.

B. *Results of Out-of-core GPU Sorting*

In order to investigate the efficiency of the out-of-core GPU sorting technique explained in Section V-C, we compare the performance of our our-of-core GPU sorting implementation with STL sort and Thrust OpenMP sort on TSUBAME2.5 using a single node. The objective of this experiment is to understand the effectiveness of the use of GPUs in the *Sort* operation when the input key-value data exceeds the capacity of GPU memory. Figure 8 shows the results, where the x axis denotes the input number of key-value pairs in millions and the y axis denotes the sorting rate on key-value pairs in millions per second. Note that the blue vertical bar between 100 and 150 on the x axis denotes the border that the size of input data exceeds the capacity of device memory on a GPU when the input data increase. The results exhibit that our implementation performs 2.53x faster than STL sort at 285 million of the input key-value pairs. These results indicate that GPU can accelerate sorting performance even though the size of input data exceeds the memory capacity on a GPU; however, we also see the performance degradation in our implementation when the size of input data becomes large. This degradation is largely caused by the nature of the out-of-core GPU sorting

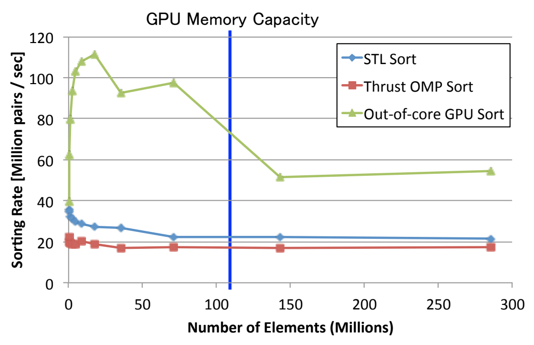

Fig. 8. Results of out-of-core GPU sorting.

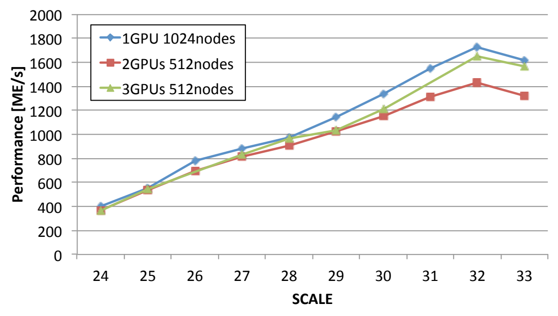

Fig. 9. Results of performance of scale-up and scale-out strategies on TSUBAME2.5.

algorithm; We conduct the in-core GPU sorting when the size of input data fits the capacity of the device memory on a GPU, while conducting the out-of-core GPU sorting only when the input data exceeds the GPU memory capacity. The out-of-core GPU sorting algorithm introduces several additional instructions, such as multiple repetitions of chunk-based in-core GPU sorting and data transfers between CPU and GPU compared with the in-core GPU sorting; however, GPU sorting still has performance benefits even for large data sets that exceed the capacity of GPU device memory.

C. Balance between Scale-up and Scale-out approaches

In order to investigate execution approaches whether we should use only device memory on GPUs (scale-out) or offload partial graph data to secondary CPU memory (scale-up) on a multi-node environment, we conduct performance studies on the balance of the number of compute nodes and GPUs per node. We vary the number of GPUs per node from 1 to 3 and use two patterns of the number of nodes: 512 and 1024. Then we set the three configurations: a) 1 GPU per node on 1024 nodes (1024 GPUs in total), b) 2 GPUs per node on 512 nodes (1024 GPUs in total), and c) 3 GPUs per node on 512 nodes (1536 GPUs in total), and compare the edge scan performance of each configuration. Figure 9 shows the results of the experiment, where the x axis denotes the size of graphs in SCALE and the y axis denotes the performance in ME/s (million edges per second). We see that the configurations b) and c) exhibit 0.81x and 0.97x of the performance compared with the configuration a). The results indicate that we can obtain competitive performance results when we use a large number of GPUs per node in a small number of nodes.

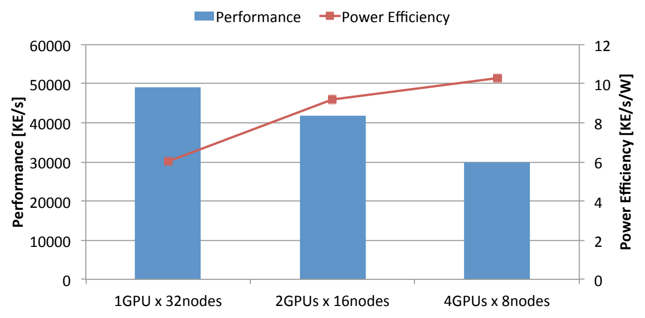

Fig. 10. Results of the performance and the power efficiency on scale-up and scale-out strategies on TSUBAME-KFC.

Furthermore, we also investigate the power efficiency on scale-up and scale-out approaches on TSUBAME-KFC, each of which has 2 sockets of Intel Xeon E5-2620 v2 (Ivy Bridge EP, 2.10GHz, 6 cores) CPU, 64GB of DDR3 main memory, 4 devices of NVIDIA Tesla K20X GPU with 6GB of discrete GDDR5 memory connected to PCI-Express 2.0 $\times$ 16 buses, and 1 card of FDR InfiniBand HBA (56Gbps) connected to a single rail interconnect network, and runs on CentOS release 6.4. We use Open MPI 1.7.2 with GNU GCC 4.4.7 for the MPI implementation, and CUDA driver 5.5 and CUDA runtime 5.5 for the GPU implementation. We use a SCALE 27 graph and measure the elapsed time and the mean power consumption using GPUs. Figure 10 shows the results of the performance and the power efficiency using three configurations: d). 32 nodes with 1 GPU per node, e). 16 nodes with 2 GPUs per node, and f). 8 nodes with 4 GPUs per node. Note that the three configurations use the same number of GPUs (i.e. 32 GPUs) in total. The results show that the simple scale-out approach d) performs the best in the three configurations in edge scan performance. On the other hand, the scale-up approaches e) and f) perform better power efficiency than the scale-out strategy, by 1.53x and 1.71x respectively. These results suggest that the scale-up approach should be considered as an option for the architectures of next generation supercomputers, since the power efficiency is considered as one of the most important problems for future large-scale computing environments.

VII. Conclusions

We propose an out-of-core GPU memory management technique for large-scale MapReduce-based graph applications. The proposed technique handles memory overflows from GPUs by automatically dividing graph data into multiple chunks and overlaps CPU-GPU data transfer overheads as much as possible. Our experimental results on TSUBAME 2.5 using 1024 nodes (12288 CPU cores, 3072 GPUs) exhibit that our GPU-based implementation performs 2.10x faster than the CPU-based implementation on a graph with 17.18 billion vertices and 274.9 billion edges. We reveal that our GPU-based approach with out-of-core GPU data management can accelerate the *Map* and *Reduce* phases by fully overlapping CPU-GPU data transfer and by applying several optimizations. We also show that the scale-up approach performs better power efficiency than the simple scale-out approach.

Our future work includes the use of Non-Volatile Memory such as flash for handling the larger size of graph data than the CPU memory capacity. We plan to investigate efficient hierarchical memory management techniques that utilize three-level memory layers including GPU device memory, CPU host memory, and Non-Volatile Memory.

ACKNOWLEDGEMENT

This work was partially supported by the JST-CREST Projects, and JSPS KAKENHI Grant Number 26011503, 26540050.

REFERENCES

[1] J. A. Ang, B. W. Barrett, K. B. Wheeler, and R. C. Murphy, "Introducing the Graph 500," May 2010.

[2] P. Harish and P. J. Narayanan, "Accelerating large graph algorithms on the GPU using CUDA," in *Proceedings of the 14th International Conference on High Performance Computing*, ser. HiPC'07. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 197–208.

[3] T. Wu, B. Wang, Y. Shan, F. Yan, Y. Wang, and N. Xu, "Efficient PageRank and SpMV Computation on AMD GPUs," in *39th International Conference on Parallel Processing*, ser. ICPP '10, Sept 2010, pp. 81–89.

[4] K. Shirahata, H. Sato, T. Suzumura, and S. Matsuoka, "A Scalable Implementation of a MapReduce-based Graph Processing Algorithm for Large-Scale Heterogeneous Supercomputers," in *Proceedings of the 2013 IEEE/ACM 13th International Symposium on Cluster, Cloud and Grid Computing*, ser. CCGrid '13, May 2013, pp. 277–284.

[5] S. Matsuoka, T. Endo, N. Maruyama, H. Sato, and S. Takizawa, "The Total Picture of TSUBAME2.0," *Tsubame e-Science Journal*, vol. 1, pp. 2 – 4, 2010.

[6] S. Matsuoka, "The TSUBAME2.5 Evolution," *Tsubame e-Science Journal*, vol. 10, pp. 2 – 8, 2013. [Online]. Available: http://www.gsic.titech.ac.jp/en/TSUBAME_ESJ

[7] L. Luo, M. Wong, and W.-m. Hwu, "An Effective GPU Implementation of Breadth-first Search," in *Proceedings of the 47th Design Automation Conference*, ser. DAC '10, 2010, pp. 52–55.

[8] S. Hong, S. K. Kim, T. Oguntebi, and K. Olukotun, "Accelerating CUDA Graph Algorithms at Maximum Warp," in *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '11, 2011, pp. 267–276.

[9] S. Edelkamp and D. Sulewski, "External Memory Breadth-First Search with Delayed Duplicate Detection on the GPU," in *Model Checking and Artificial Intelligence*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2011, vol. 6572, pp. 12–31.

[10] J. Zhong and B. He, "Medusa: Simplified Graph Processing on GPUs," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 6, pp. 1543–1552, June 2014.

[11] E. Elsen, "Speeding Up GraphLab Using CUDA," in *GPU Technology Conference*, 2014.

[12] Y. Wang, "High-Performance Graph Primitives on the GPU: Design and Implementation of Gunrock," in *GPU Technology Conference*, 2014.

[13] D. Merrill, M. Garland, and A. Grimshaw, "Scalable GPU Graph Traversal," in *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '12, 2012, pp. 117–128.

[14] S. Hong, T. Oguntebi, and K. Olukotun, "Efficient Parallel Graph Exploration on Multi-Core CPU and GPU," in *Proceedings of the 20th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '11, Oct 2011, pp. 78–88.

[15] A. Gharaibeh, L. B. Costa, E. Santos-Neto, and M. Ripeanu, "On Graphs, GPUs, and Blind Dating: A Workload to Processor Matchmaking Quest," in *Proceedings of the IEEE 27th International Symposium on Parallel and Distributed Processing*, ser. IPDPS '13, 2013, pp. 851–862.

[16] N. T. Duong, Q. A. P. Nguyen, A. T. Nguyen, and H.-D. Nguyen, "Parallel PageRank Computation Using GPUs," in *Proceedings of the Third Symposium on Information and Communication Technology*, ser. SoICT '12, 2012, pp. 223–230.

[17] X. Yang, S. Parthasarathy, and P. Sadayappan, "Fast Sparse Matrix-vector Multiplication on GPUs: Implications for Graph Mining," *Proceedings of the VLDB Endowment*, vol. 4, no. 4, pp. 231–242, Jan. 2011.

[18] A. Rungsawang and B. Manaskasemsak, "Fast PageRank Computation on a GPU Cluster," in *Proceedings of the 20th Euromicro International Conference onParallel, Distributed and Network-Based Processing*, ser. PDP '12, Feb 2012, pp. 450–456.

[19] K. Ueno and T. Suzumura, "Parallel Distributed Breadth First Search on GPU," in *Proceedings of the 20th IEEE conference on High Performance Computing*, ser. HiPC '13, Dec 2013, pp. 314–323.

[20] A. Kyrola, G. Blelloch, and C. Guestrin, "GraphChi: Large-scale Graph Computation on Just a PC," in *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI '12, 2012, pp. 31–46.

[21] K. Iwabuchi, H. Sato, R. Mizote, Y. Yuichiro, K. Fujisawa, and S. Matsuoka, "Hybrid BFS Approach Using Semi-External Memory," in *IEEE International Symposium on Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW)*, May 2014.

[22] K. Nilakant, V. Dalibard, A. Roy, and E. Yoneki, "PrefEdge: SSD Prefetcher for Large-Scale Graph Traversal," in *7th ACM International Systems and Storage Conference*, 2014.

[23] R. Pearce, M. Gokhale, and N. Amato, "Scaling Techniques for Massive Scale-Free Graphs in Distributed (External) Memory," in *Proceedings of the IEEE 27th International Symposium on Parallel Distributed Processing*, ser. IPDPS '13, May 2013, pp. 825–836.

[24] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," *OSDI '04, Sixth Symposium on Operating System Design and Implementation*, pp. 137–150, 2004.

[25] A. Bialecki, M. Cordova, D. Cutting, and O. O'Malley, "Hadoop: a framework for running applications on large clusters built of commodity hardware," 2005. [Online]. Available: http://lucene.apache.org/hadoop

[26] S. J. Plimpton and K. D. Devine, "MapReduce in MPI for Large-scale Graph Algorithms," *Parallel Computing*, vol. 37, no. 9, pp. 610–632, Sep. 2011.

[27] G. Jin, T. Endo, and S. Matsuoka, "A Parallel Optimization Method for Stencil Computation on the Domain that is Bigger than Memory Capacity of GPUs," in *Proceedings of the IEEE International Conference on Cluster 2013*, ser. CLUSTER '13, Sept 2013, pp. 1–8.

[28] R. Wang, Y. Huo, Y. Yuan, K. Zhou, W. Hua, and H. Bao, "GPU-based Out-of-core Many-lights Rendering," *ACM Transactions on Graphics*, vol. 32, no. 6, pp. 210:1–210:10, Nov. 2013.

[29] Y. Ye, Z. Du, and D. A. Bader, "GPUMemSort: A High Performance Graphic Co-processors Sorting Algorithm for Large Scale In-Memory Data," *GSTF International Journal on Computing*, pp. 1(2):23–28, 2011.

[30] H. Peters, O. Schulz-Hildebrandt, and N. Luttenberger, "Parallel external sorting for CUDA-enabled GPUs with load balancing and low transfer overhead," in *IEEE International Symposium on Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW)*, April 2010, pp. 1–8.

[31] M. Silberstein, B. Ford, I. Keidar, and E. Witchel, "GPUfs: Integrating a File System with GPUs," in *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '13, 2013, pp. 485–498.

[32] U. Kang, C. E. Tsourakakis, and C. Faloutsos, "PEGASUS: A Peta-Scale Graph Mining System Implementation and Observations," in *Proceedings of the 9th IEEE International Conference on Data Mining*, ser. ICDM '09, Washington, DC, USA, 2009, pp. 229–238.

[33] J. A. Stuart and J. D. Owens, "Multi-GPU MapReduce on GPU Clusters," in *Proceedings of the 25th IEEE International Parallel and Distributed Processing Symposium*, ser. IPDPS '11, May 2011.

[34] S. Brin and L. Page, "The Anatomy of a Large-Scale Hypertextual Web Search Engine," in *Proceedings of the 7th International World-Wide Web Conference*, ser. WWW '98, 1998.

[35] N. Bell and J. Hoberock, "Thrust: A Productivity-Oriented Library for CUDA," *GPU Computing Gems*, vol. 7, 2011.