GraphReduce: Processing Large-Scale Graphs on Accelerator-Based Systems

Dipanjan Sengupta
Georgia Institute of Technology
dsengupta6@gatech.edu

Shuaiwen Leon Song
Pacific Northwest National Lab (PNNL)
Shuaiwen.Song@pnnl.gov

Kapil Agarwal
Georgia Institute of Technology
kapila@gatech.edu

Karsten Schwan
Georgia Institute of Technology
schwan@cc.gatech.edu

ABSTRACT

Recent work on real-world graph analytics has sought to leverage the massive amount of parallelism offered by GPU devices, but challenges remain due to the inherent irregularity of graph algorithms and limitations in GPU-resident memory for storing large graphs. We present GraphReduce, a highly efficient and scalable GPU-based framework that operates on graphs that exceed the device's internal memory capacity. GraphReduce adopts a combination of edge- and vertex-centric implementations of the Gather-Apply-Scatter programming model and operates on multiple asynchronous GPU streams to fully exploit the high degrees of parallelism in GPUs with efficient graph data movement between the host and device. GraphReduce-based programming is performed via device functions that include gatherMap, gatherReduce, apply, and scatter, implemented by programmers for the graph algorithms they wish to realize. Extensive experimental evaluations for a wide variety of graph inputs and algorithms demonstrate that GraphReduce significantly outperforms other competing out-of-memory approaches.

Categories and Subject Descriptors

C.1.2 [**Multiple Data Stream Architecture**]: Single-Instruction, Multiple-Data Processors (SIMD); D.1.3 [**Programming Techniques**]: Concurrent Programming-Parallel Programming

General Terms

Design, Experimentation, Performance, Big Data

Keywords

Graph Analytics, Big Data, GPGPU, Performance Optimization, Data Movement Optimization

1. INTRODUCTION

The need to rapidly process large graph-structured data, in both scientific and commercial applications, has engendered recent efforts to leverage cost-efficient GPUs [34, 35] for efficient graph analytics. Doing so, however, requires addressing substantial technical challenges, including (1) dealing with the dynamic nature of graph parallelism [43, 17, 22, 15], (2) coping with constrained on-GPU memory capacity, i.e., to process graphs with memory footprints that exceed that capacity [23, 32], and (3) addressing programmability issues for developers with limited insights into how to best exploit the resources of evolving and varied GPU architectures [27, 21, 24].

Previous work on parallel graph processing has sought to exploit scale-out methods, by distributing large graph data across the different nodes of computational clusters [26]. Recognizing the low computation to communication ratios of typical graph processing algorithms [23, 32], the 'GraphReduce' (GR) programming framework presented in this paper uses the alternative 'scale up' approach in which large graphs processed by memory-limited GPUs can take advantage of the potentially considerable memory capacities of the host machines to which they are attached. The implementation of GR for NVIDIA GPUs evaluated in this paper efficiently runs irregular graph algorithms on datasets considerably larger than GPU memory sizes, by (i) partitioning graphs into fixed size chunks – shards – asynchronously moved between GPU and host, (ii) adopting a combination of edge-(X-Stream [32]) and vertex-(GraphChi [23]) centric implementations of graph representations, (iii) overlapping GPU computation with data transfer via concurrent GPU operations, using CUDA Streams, and (iv) using 'spray' operations to further divide shards and obtain fine-grain parallelism that exploits the Hyper-Q feature of Kepler GPUs [7]. Specifically, spray operations are used to further divide each shard into multiple sub-buffers transferred over dynamically created CUDA streams. The purpose is to efficiently use GPU hardware features like Hyper-Qs [7].

GraphReduce runs graph algorithms on GPUs without unduly burdening graph algorithm developers. Programmers write the appropriate sequential codes for their algorithms, e.g., for data mining, machine learning, etc., and then use its simple APIs to express their use for processing various graphs. The GR runtime partitions the graph into different shards, each single one of which entirely fits into GPU memory. Graph processing, then, overlaps shard movement with GPU-level graph processing, the latter using multiple levels of GPU-level parallelism, as indicated above. With such automation, GR can deal with graph sizes much exceeding GPU memory sizes. This is important because even a common Yahoo web-graph comprised of 1.4 billion vertices [11] requires approximately 6.6 GB of memory to store just its vertex values (not even including the edges and their corresponding states).

In summary, with GraphReduce, GPUs can be used to accelerate analytics performed on graphs with billions of edges, operating at speeds much exceeding that of similar operations run on CPUs, and programmed in ways accessible to programmers who are not experts in GPU programming. To the best of our knowledge, GraphReduce is the first to support in-GPU-memory and out-of-GPU-memory graph processing, thus aiming for scale-up graph processing on HPC systems with discrete GPUs and high end (i.e., memory-rich) hosts.

The GraphReduce framework uses a Gather-Apply-Scatter (GAS) model to efficiently process graphs of sizes larger than GPU memory. Its technical contributions include the following:

- High performance is obtained in part from its use of a combination of edge- and vertex-centric graph programming, to match the different types of parallelism present in different phases of the GAS execution model.
- Efficiency in graph processing via improved asynchrony in computation and communication, gained by GR's runtime via dynamic characterization of data buffers based on data access pattern and access locality. Additional hardware parallelism is extracted via spray streams for deep copy operations on separate CUDA streams.
- Use of computational frontier information for efficient GPU hardware thread scheduling and data movement between host and GPU. Specifically, GR moves data into GPU memory only when a subset of the graph has at least one active vertex or edge. Further, when possible, GR uses dynamic phase fusion/elimination to merge/eliminate multiple GAS phases, to avoid unnecessary kernel launches and associated data movement.

Results show that GraphReduce can significantly outperform the existing state-of-the-art graph analytics frameworks, across a wide variety of algorithms and for large-scale graphs that do not fit into GPU-resident memory. Specifically, it achieves up to **79x and 21x** and an average of **13.4x and 5x** speedup over CPU-based frameworks like GraphChi [23] and X-Stream [32], respectively, for several real-world large-scale graphs with various algorithms. At the same time, GraphReduce also achieves comparable performance with the existing highly optimized in-GPU-memory solutions like MapGraph [17] and CuSha [22], for smaller in-memory graph inputs.

The remainder of the paper is organized as follows: Section 2 discusses the background and motivation for GraphReduce. Section 3 dissects the design choices. Section 4 introduces our GraphReduce framework. Section 5 highlights the major optimizations used in GraphReduce. Section 6 presents the experimental setup and result analysis. Section 7 discusses the related work and Section 8 concludes with future work.

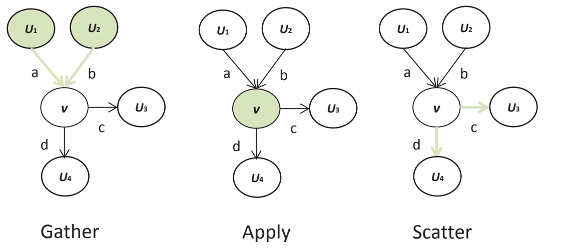


Figure 1: An example of GAS abstraction.

2. BACKGROUND AND MOTIVATION

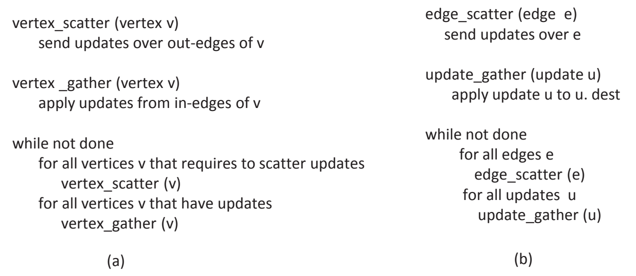
This section introduces the computational model used in GraphReduce. It also motivates the GR approach by describing some of the challenges faced by the existing state-of-the-art graph processing approaches.

2.1 Computational Model: GAS Abstraction

GraphReduce exposes the Gather-Apply-Scatter (GAS) computational model used by Pregel [28], Powergraph [19], and GraphLab [26]. With GAS, a problem is described as a directed (sparse) graph, $G = (V, E)$, where V denotes the vertex set and E denotes the directed edge set. A value is associated with each vertex $v \in V$, and each directed edge e_{ij} is associated with a source vertex u and a target vertex v: $e_{ij} = (u, v) \in E$. Given a directed edge $e_{ij} = (u, v)$, we refer to e_{ij} as vertex v's in-edge, and as vertex u's out-edge. A typical GAS computation, then, has three stages [10]: (1) Initialization, (2) Iterations, and (3) Output. Initialization deals with initializing vertex/edge values and a starting *computation frontier*, which is defined as the set of active vertices for a given iteration. In each Iteration stage, a sequence of iterations is run, each gathering the values seen on the incoming edges, updating the values of elements, and then defining a new frontier for the next iteration. Figure 1 illustrates these three phases, assuming vertex v to be the central vertex.

- **Gather Phase**: each vertex aggregates values associated with its incoming edges and their source vertices. We define the gather function as $G(u, v, e_{ij})$, and we use binary operator $\uplus$ to aggregate the outputs from multiple Gs into one value R. In Figure 1 (a), the result (R) from the Gather Phase for vertex v can be represented as $R = G(u_1, v, a) \uplus G(u_2, v, b)$.
- **Apply Phase**: the value of each vertex in the current frontier is updated through the gather result. We define the update function as $U(v, R)$, where R is the result from the Gather Phase. Shown in the Figure 1 (b), we have the updated vertex v as: $v' = U(v, R)$.
- **Scatter Phase**: the new vertex state is propagated to neighbors, by updating the state of its out-edges (e.g., c and d in Figure 1). We define the Scatter function for updating the out-edges of v as $S(v', e_{out})$, where v' is the updated vertex v and e_{out} represents v's out-edges. Shown in Figure 1 (c), two updated edges c' and d' are denoted as: $c' = S(v', c)$ and $d' = S(v', d)$.

As shown in much prior work [19, 28, 42], the GAS model is not only simple to use, but it is also sufficiently general to express a broad set of graph algorithms, ranging from PageRank to Connected Components, and from Heat Simulation to Sparse Linear Algebra. For example, the PageRank algorithm [30] can be expressed as follows. In the **Gather**



```
vertex_scatter (vertex v)
    send updates over out-edges of v

vertex _gather (vertex v)
    apply updates from in-edges of v

while not done
    for all vertices v that requires to scatter updates
        vertex_scatter (v)
    for all vertices v that have updates
        vertex_gather (v)

(a)
```

```
edge_scatter (edge  e)
    send updates over e

update_gather (update u)
    apply update u to u. dest

while not done
    for all edges e
        edge_scatter (e)
    for all updates  u
        update_gather (u)

(b)
```

Figure 2: (a) Vertex-centric Scatter-Gather. (b) Edge-centric Scatter-Gathe [32].

Phase, each vertex v_i in the current frontier accumulates $G_i = \sum \frac{R_j}{n_j}$ from all in-edges from source vertex v_j, where R_j is the rank of v_j and n_j is the number of out-edges $(v_j \rightarrow v_i)$ of v_j. Then, in the **Apply Phase**, vertex v_i updates its value using some common PageRank formula like $R_i = 0.85 + 0.15 \times G_i$. Since in PageRank, the values of out-edges of v_i will not change in the **Scatter Phase**, there are no operations for this phase.

Figure 2 shows two common ways to implement graph algorithms with GAS: edge-centric vs. vertex-centric execution, which differs in whether the Scatter and Gather phases iterate over and update edges or vertices (their pseudo codes are shown in Figure 2). Implementation can also vary in terms of Update functions, to be implemented as either Bulk-Synchronous Parallel (BSP) [39] for simplicity or via asynchronous execution, for faster convergence. In either case, the graph algorithm terminates when some application-specific condition is met, e.g., when no more changes in vertex and edge states beyond a certain threshold.

2.2 Motivation and Challenges

Graph Name	Vertices	Edges	In-memory Size
	GPU In-Memory		
ak2010[3]	45,292	108,549	7.9MB
coAuthorsDBLP[5]	299,067	977,676	69.5MB
kron_g500-logn20[31]	1,048,576	44,620,272	2.4GB
webbase-1M[40]	1,000,005	3,105,536	211.6MB
belgium_osm[4]	1,441,295	1,549,970	5.4MB
	GPU Out-of-Memory		
kron_g500-logn21[31]	2,097,152	91,042,010	4.84GB
nlpkkt160[33]	8,345,600	221,172,512	11.9GB
uk-2002[8]	18,520,486	298,113,762	16.4GB
orkut[41]	3,072,441	117,185,083	6.2GB
cage15[1]	5,154,859	99,199,551	5.4GB

Table 1: Datasets used in this paper. 'Out-of-memory' means that the input graphs cannot fit into the limited GPU memory. A commercial K20c GPU with a 4.8 GB global memory is used as an example to illustrate in-memory and out-of-memory cases.

Graphs	X-Stream (ms)	CuSha(ms)	Speedup
ak2010	215.155	7.75	28x
belgium_osm	2695.88	791.299	3x
coAuthorsDBLP	1275	11.553	110x
delaunay_n13	80.89	5.184	16x
kron_g500-logn20	46550.7	119.824	389x
webbase-1M	3909.12	13.515	290x

Table 2: Performance comparision between two state-of-the-art graph processing approaches. X-Stream runs on a 16 core Xeon E5-2670 CPU with 32GB memory. CuSha runs on a NVIDIA K20c Kepler GPU with 4.8 GB memory.

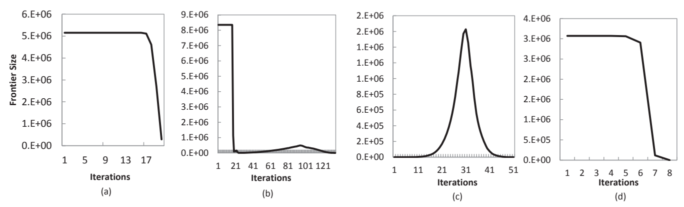


Figure 3: Frontier size changes across iterations using the GAS model on GPUs. This phenomenon highly depends on the input graph and algorithm, showcasing the inherent graph irregularity. Four cases from left to right: (a) Cage15 - PageRank; (b) nlpkkt160 - PageRank; (c) Cage15 - BFS; and (d) orkut - Connected Component (CC).

The high compute power and multi-level parallelism provided by the SIMT (Single Instruction Multiple Threads) architectures of GPGPUs [1] present opportunities for accelerating many graph algorithms [43, 22, 10, 17]. Table 2 shows the performance comparison between two state-of-the-art graph analytics processing the BFS algorithm: X-Stream [32] for CPUs and CuSha [22] for in-memory GPU processing. Significant performance speedups are observed from using the GPU. For instance, graph kron_g500-logn20 [31] processed by CuSha on a commercial K20c Kepler GPU (4.8GB memory) achieves 390x speedup over X-Stream on a 16 core Xeon E5-2670 CPU (32 GB memory).

Acceleration of graph analytics via GPUs is limited, however, by the fact that many real-world graphs cannot fit into GPUs' limited memories. As mentioned earlier, a common Yahoo-web graph [11] with 1.4 billion vertices requires 6.6 GB memory just to store its vertex values. Additional examples of graphs exceeding GPU memory sizes appear in Table 1. Previous work on GPU-based graph processing has not addressed this issue. CuSha [22], MapGraph [17], VertexAPI [10] and Medusa [43] all assume graphs to reside in GPU memory. GraphChi [23] and X-Stream [32] are designed for CPU-based systems, unable to benefit from the multi-level massive parallelism offered by GPUs (shown in Table 2). Hybrid approaches (CPU+GPU) like Totem [18] are only able to process a fixed sub-graph that can fit into GPU memory after statically partitioning the graph between CPU and GPU, which results in underutilization of GPU's fullest processing power and parallelism.

There are several challenges to efficiently process larger-than-memory graphs on GPUs. They involve the need to provide end users with convenient programming constructs for their graph algorithms, but without unduly burdening them with (i) graph partitioning to fit sub-graphs into GPU memory, (ii) how and when such partitions are moved between GPU and host memories, and (iii) how to best extract multi-level parallelism from their GPU-resident execution. The GraphReduce framework presented in this paper addresses these challenges.

Graph partitioning or chunking for fitting into GPU memory must deal with the irregular nature of graph algorithms and how they access the input data. More specifically, to obtain high on-GPU performance, chunking must be done to minimize GPU-host data movement. For the GAS model, this requires ensuring GPU memory residence of the vertices and edges that actively take part in the computation itera-

[1]Without specified mention, NVIDIA terminology will be used throughout the paper to illustrate our work. However, the proposed methodology can be easily applied to a wide range of massively parallel architectures.

tions being performed. This despite the fact that due to the inherent irregularity in graph algorithms, in every computation iteration, the number of edges and vertices that actively take part in computation (computation frontier size) is not constant, and it varies with graph algorithms and datasets, as shown in Figure 3. Across all of these cases, the frontier sizes [2] incur significant changes (either dropping or climbing). For instance, in Figure 3(b) for graph nlpkkt160 processed by PageRank, the frontier size drops sharply after a few iterations and remains low for the rest of the execution. Given these results, ideally, the GR runtime should *move sub-graphs to the GPU only if they contain active vertices and edges*. Otherwise, such movements simply cause unnecessary overhead. For the same nlpkkt160 case in Figure 3(b), after several iterations, most of the sub-graphs do not have active vertices/edges, so there is no need to move those chunks to the GPU. We have found this phenomena to be very common across most of the graphs shown in Table 1, for various algorithms. The GR methods presented in this paper address this issue, along with (ii) and (iii) above.

3. DESIGN CHOICES

3.1 Hybrid Programming Model

Existing systems choose some specific programming model for graph execution. GraphLab [26], Pregel [28] and GraphChi [23] use the vertex-centric model, while X-Stream [32] uses the edge-centric model. In comparison, GraphReduce employs a hybrid programming model [36] using both edge- and vertex-centric operations. This is because in the GAS model, different processing phases have different types of parallelism and consequently, offer different parallelism opportunities, coupled with different memory access characteristics. For instance, an edge-centric model should be used in the **Gather Phase** (shown in Figure 1), because a GPU hardware thread will then be assigned to work on behalf of an edge in the graph. This is preferable to the vertex-centric model, because first, real-world graphs commonly have more edges than vertices (shown in Table 1), thus giving rise to higher degrees of parallelism and decreased GPU core idling. Second, in the vertex-centric approach, each vertex receives information from multiple in-edges, resulting in a consequent need for synchronization or atomics to order the receive operations from each of the in-edges. This could potentially degrade the overall performance. The same observations hold for using the edge-centric model in the **Scatter Phase**. In contrast, in the **Apply Phase**, there are parallelism opportunities only over the vertex set, thus favoring a vertex-centric model.

3.2 Characterization of Buffers in Play

Graph data chunked to fit into GPU memory and to be moved from host to GPU, is comprised of edges that have either a destination or a source vertex in some well-defined graph partition (see Section 4.2 for details). Henceforth termed *shards*, such chunks reside in memory buffers that experience different access patterns. GraphReduce characterizes such access patterns in order to appropriately map corresponding memory buffers to the memory abstractions exposed by current GPUs. In terms of data movement, buffers can be classified as static vs. streaming buffers.

[2]The term “frontier size” is synonymous with the number of active vertices in a given iteration. The variation of the frontier size during execution is sensitive to the starting-point for graph processing.

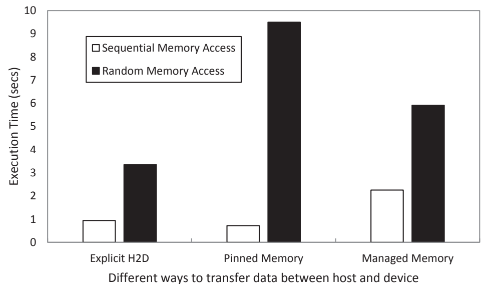


Figure 4: Performance of transferring 100,000,000 double elements, using three techniques for data exchange between CPU and GPU.

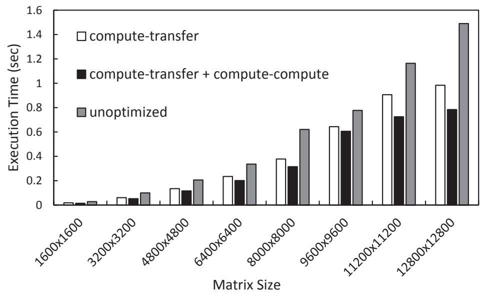


Figure 5: Performance benefits of using a combination of compute-transfer and compute-compute schemes for processing matrix multiplication with different input sizes. Stripe size=50, which refers to the contiguous number of rows of the matrix being fetched into the GPU memory as a chunk.

Static buffers are copied only once to GPU memory, typically in the Initialization phase. They remain there for the lifetime of the graph execution. An example is a vertex set of a graph that fits into GPU memory. Streaming buffers, on the other hand, are moved in and out of GPU memory as processing progresses, and at any point in time, a particular instance of some streaming buffer resides in GPU memory, e.g., a subset of a graph's edge set. In the GAS programming model, static buffers are accessed in all three phases, while streaming buffers only appear in a single phase. Another way to characterize buffers is by their access rules, such as read-only or read/write access. For example, vertex and edge data buffers (containing mutable states) have both read and write access patterns, while the vertex set (containing immutable vertex IDs) is read-only. Based on these attributes, the GR runtime makes decisions on whether or not to transfer certain buffers back to the host (see Section 4.3). Finally, buffers can be classified in terms of the spatial locality of their accesses, e.g., random or sequential access. For example, in the edge-centric approach shown in Figure 2, there are random accesses to the vertex set.

Once characterized, buffers are mapped to the different memory abstractions exposed by GPU, which at minimum, contain slow and fast memory (e.g., host memory and GPU memory). In the different phases of the GAS model, there are a mix of random and sequential accesses to the input buffers (e.g., edge/vertex sets). For this mix, we posit that random access to slow memory is much more expensive than random access to faster memory, whereas for sequential access, memory-level parallelism and prefetching can help mask slower memory access speed. Therefore, due to the limited fast memory size (GPU), we choose to map all sequential accesses in a GAS phase to the slower CPU memory and all the random accesses to the faster GPU memory. We next validate these assumptions.

Figure 4 depicts the performance of three techniques for data exchange between host and GPU (through CUDA runtime APIs): (a) explicit data transfer using cudaMemcpy() or *Explicit H2D*; (b) *Pinned Memory* using Unified Virtual Addressing (UVA) [9], in which data is transferred implicitly by the CUDA runtime but the memory is allocated as locked memory on the host side; and (c) *Managed Memory* (introduced in CUDA 6 [9] as Unified memory), where data is transferred between host and device on demand. The measurements shown in the figure illustrate that in the case of sequential memory access, *Pinned Memory* performs the best, because the accesses directly translate to memory loads/stores operations over the PCIe in which (i) sequential accesses benefit from memory level parallelism (MLP) and (ii) software-level prefetching can hide communication overheads. In the case of random access, *Explicit H2D* performs the best and *Pinned Memory* performs the worst. In other words, random access performs best when data resides in faster GPU memory, and the performance of the *Pinned Memory* degrades as the load/store memory operations over the PCIe fail to benefit from prefetching (after all, accesses are random!). Since *Pinned Memory* performs the best for sequential accesses, one straightforward approach is to organize graphs such that all memory accesses are sequential. However, because of the significant number of random accesses to either the edges or vertices of a graph in at least one phase of the GAS model [23, 32], this is not a viable solution for GR as the benefits of sequential accesses are overshadowed by the huge overhead of the random accesses to the slow memory. In response, GR uses explicit data transfer as the mechanism for transferring data between host and device, in way that aim to leverage GPU memory coalescing and software prefetching for the sequential accesses. Although certain performance benefits may exist through intellegent runtime buffer-type selecting. we leave this exploration for the future work.

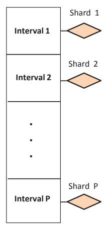
3.3 Coordinated Computation and Data Movement

The spatial choice of where in memory to locate data requires an associated temporal choice in when to perform data movement between host and GPU memories. GR uses two methods to attain high performance: (1) hide communication costs by overlapping GPU computation with necessary data transfers, and (2) utilize the GPU's inherent high degree of potential internal parallelism. (1) is obtained via software-based prefetching to move shards into GPU memory while GPU kernel(s) are being executed. (2) is realized by leveraging underutilized GPU resources (idle threads) caused by the irregular nature of graph processing (shown in Figure 3). It involves (i) detecting such idle threads, using the computation frontier information available to the GR runtime, and (ii) initiating the execution of new shards when idleness is present (note that shards within a single GAS phase do not have data dependencies, so they can be processed in parallel). GR accomplishes this by automatically launching multiple kernels (within the same context), according to the resources available in each GAS phase. Denoting (1) as *compute-transfer* scheme and (2) as a *compute-compute* scheme, Figure 5 shows the performance benefits obtained from using these approaches vs. an unoptimized scenario when processing a large matrix that doesn't fit into

```
Connected Component (CC)

        __host__ __device__
        static int gatherReduce(const int& left, const int& right)
        {
            return min(left, right);
        }
        __host__ __device__
        static int gatherMap(const VertexData* dstLabel, const VertexData
                                *srcLabel, const EdgeData* edge)
        {
            return *srcLabel;
        }
        __host__ __device__
        static bool apply(VertexData* curLabel, GatherResult label)
        {   bool changed = label < *curLabel;
            *curLabel = min(*curLabel, label);
            return changed;
        }
        __host__ __device__  static void scatter(const VertexData* src, const
                                                  VertexData *dst, EdgeData* edge)
        {
            //no scatter operations for CC algorithm
        }
```

Figure 6: Writing sequential code using GAS model for Connected Component (CC) algorithm in GraphReduce.



```
Data Structures for Shard and Interval

struct interval
{
  int start, end;
};
struct edge
{
  int src, dest;
  float val;
};
struct vertex
{
  int num_of_in_edges;
  int num_of_out_edges;
  float val;
};

struct shard
{
  int start_vertex, end_vertex;
  union
  {
    int in_edge_v_index[MAX_V_PER_SHARD];
    int out_edge_v_index[MAX_V_PER_SHARD];
  }
  vertex in_edge_array[MAX_INEDGE_PER_SHARD];
  vertex out_edge_array[MAX_OUTEDGE_PER_SHARD];
  edge edge_update_array[MAX_INEDGE_PER_SHARD];
  vertex vertex_update_array[MAX_VERTEX_PER_SHARD];
};
```

Figure 7: Illustration of *shard* and its data structure.

GPU memory, thus clearly demonstrating the importance of coordinating computation with data movement. We will use these two schemes for processing graph algorithms across phases in the GAS model.

4. GRAPHREDUCE FRAMEWORK

The GraphReduce framework can efficiently process graphs with large inputs and mutable edge values that cannot fit into the limited memories of discrete accelerators. GraphReduce simplifies graph analytics programming by supporting a multi-level, asynchronous model of computation. Figure 8 shows the general software architecture of GraphReduce which consists of three major components: Partition Engine, Data Movement Engine, and Compute Engine, all supporting an easily used GAS-based API.

4.1 User Interface

As shown in Figure 6, programmers can write a sequential algorithm by simply defining the graph's state data types (for vertices and edges) and four functions for the different phases in the GAS programming model. GraphReduce will then seamlessly generate parallel codes to run on the GPU. User-defined functions include $gatherMap()$, $gatherReduce()$, $apply()$ and $scatter()$, corresponding to the functions defined in Section 2.1, i.e., to $G()$, $\biguplus$, $U()$, and $S()$. Along with the vertex and edge data types, these functions are stored in a tuple called UserInfoTuple: $<gather(), apply(), scatter(), VertexDataType, EdgeDataType>$.

4.2 Partition Engine

The *Partition Engine* shown as ❶ in Figure 9 is responsible for (1) load-balanced shard creation and (2) providing graph partitioning logics and associated orderings of vertices/edges. Partitioning is performed by dividing the ver-

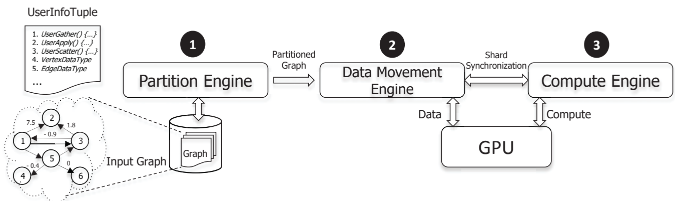


Figure 8: Architecture of GraphReduce framework.

tex set V of graph $G = (V, E)$ into disjoint intervals (i.e., sets of vertices) and for each interval, maintaining a *shard* data structure (shown in Figure 7), where each shard stores all of the edges that have either a destination or a source vertex in that interval.

GraphReduce answers the following questions about such sharding: (1) choice of interval, (2) number and sizes of shards, and, (3) how to order the edges in each shard. For (1), shown in Figure 7, intervals are chosen by the Shard Creator of the Partition Engine in a load-balanced fashion. Specifically, each shard contains an approximately equal number of edges (in- plus out-edges). For (2), the number of intervals P is chosen such that at least one shard (maybe multiple) can be loaded completely into GPU memory. Therefore, if more than one shard is allocated in GPU memory, the total number of shards simultaneously participating in graph computation can be calculated as a function of the total number of concurrent memory copy operations to and from the GPU. Finally, for (3), the graph dataset is a set of source and destination vertex pairs (edges) with the associated value for each edge. This set of tuples is generally unordered. The Graph Layout Engine inside of the Partition Engine defines the layout of the data by sorting the in-edges in the order of their destinations and the out-edges in the order of their sources. For such sorted data, we use the compressed Sparse Column (CSC) and compressed Sparse Row (CSR) formats [12] to store graphs to be used in the Gather and Scatter phases, respectively. Therefore, there is no overhead for runtime data-format transposition between CSC and CSR formats.

Edges are stored in some specific sorted order for three reasons. First, with ordered edges, the data moved across the PCIe link from host to GPU is contiguous, which can improve system throughput. Second, when updates are collected for each vertex in the gather phase, they can be stored in consecutive memory locations for each vertex, which avoids random memory accesses. Similarly, the out-edges are stored in the order of source vertices, so that the neighbors of a vertex whose states have been updated can be accessed sequentially. Third, sequential accesses to GPU memory can improve performance due to memory coalescing. Note that although we are sorting the source and destination vertices when partitioning graphs, GraphReduce is able to take any user-provided partitioning logic as a plug-in to the *Partition Logic Table* inside the Partition Engine (Figure 9).

4.3 Data Movement Engine

To address the cost of data movement caused by accesses to and updates of edge/vertex states, the Data Movement Engine (shown in Figure 10 as ❷) in GraphReduce seeks to accelerate data movement via asynchronous memory-copy operations for concurrent GPU kernel execution. For instance, for NVIDIA GPUs, the programming environment

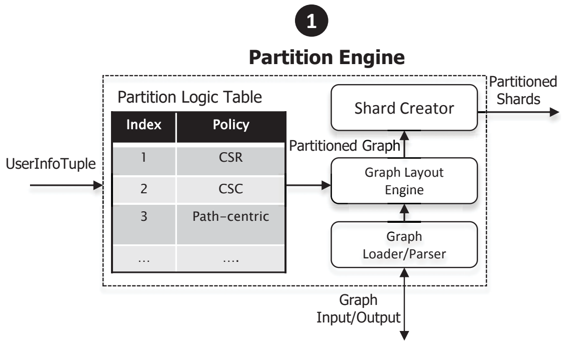


Figure 9: The structure of the Partition Engine.

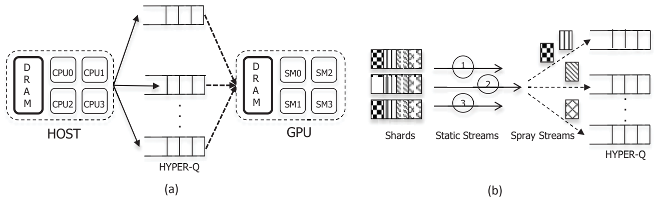


Figure 11: (a) Data Movement from host to GPU in GraphReduce through Hyper-Q. (b) Illustration of Spray Streams for better throughput.

allows the concurrent execution of operations from the same GPU protection domain or context. A sequence of operations that execute in issue-order on the GPU is defined as a *Stream Object* [2]. Operations from multiple *Streams* can be executed concurrently and interleaved, leveraging the parallelism provided by multiple hardware queues (e.g., Hyper-Qs [7] provided by NVIDIA Kepler architectures shown in Figure 11(a); they permit host to launch multiple concurrent kernels onto a single GPU). In GraphReduce, multiple intervals (and their associated shards from the Partition Engine) can also be concurrently processed by different *Streams*, to obtain a high degree of parallelism. For different shards, each *Stream* spawned by the *Static Stream Creator* inside the Data Movement Engine in Figure 10 typically issues multiple MemcpyAsync() operations and graph computation kernels asynchronously, overlapping data transfer with computation time. In the Data Movement Engine, *Streams* are scheduled and ordered, with the goal to maximize concurrency in data transfer and computation across different shards of the graph.

We now show how to derive the optimal number of shards being transferred concurrently, to maximize the use of PCIe bandwidth. Assuming that shard size is sufficiently large to saturate PCIe bandwidth (since we are dealing with large graphs), we determine the optimal number of shards transferred concurrently as a function of concurrency (number of concurrent operations). Specifically, we define P as the total number of shards, G as the size of the entire graph including vertices and edges, V as the size of the vertex set, E as the size of the edge set, K as the total number of concurrent *Streams*; M as the GPU memory size; and B as the bytes needed to achieve maximum PCIe bandwidth. We will have:

$$K * (V/P) + K * B \leq M \tag{1}$$

$$B = (\alpha \times |E| + \beta \times |V|) \tag{2}$$

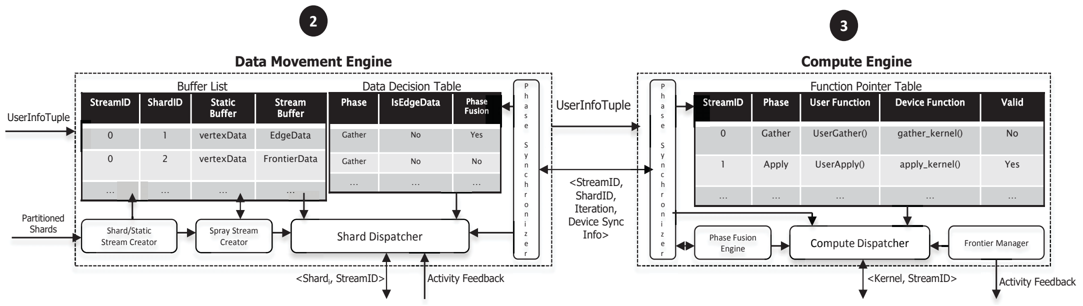
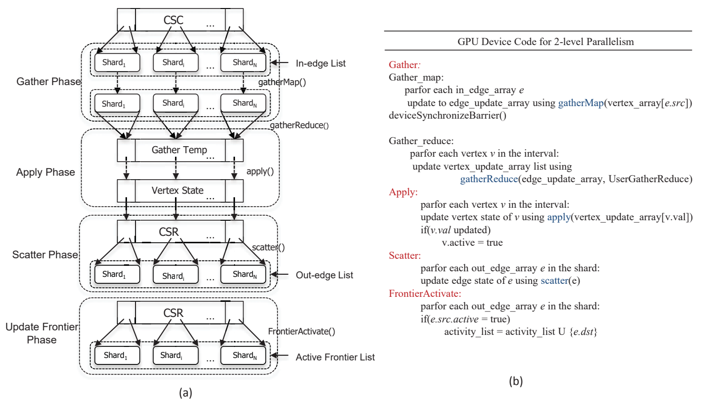


Figure 10: The structures of the Data Movement Engine and Compute Engine. Tables/buffer_list are data structures (passive elements of the engine) while rectangles are modules (active elements of the engine).



```
GPU Device Code for 2-level Parallelism

Gather:
Gather_map:
    parfor each in_edge_array e
     update to edge_update_array using gatherMap(vertex_array[e.src])
deviceSynchronizeBarrier()

Gather_reduce:
     parfor each vertex v in the interval:
      update vertex_update_array list using
                 gatherReduce(edge_update_array, UserGatherReduce)
Apply:
        parfor each vertex v in the interval:
        update vertex state of v using apply(vertex_update_array[v.val])
        if(v.val updated)
            v.active = true
Scatter:
        parfor each out_edge_array e in the shard:
        update edge state of e using scatter(e)
FrontierActivate:
        parfor each out_edge_array e in the shard:
        if(e.src.active = true)
             activity_list = activity_list U {e.dst}
```

Figure 12: (a) Sub-phases of the computation stage. (b) GPU device pseudo code for exploiting two-level parallelism in different phases.

where the upper bound of K depends on the GPU architecture (e.g., $K \leq 32$ in K20 NVIDIA GPUs), and α and β are the number of edge- and vertex-set streaming buffers (discussed in Section 3.2) used in each Stream. In Equation (1), V/P is the size of the interval of the vertex set for one partition. B is the minimum buffer size to saturate PCIe bandwidth (we assume each shard is big enough to saturate that bandwidth). Unknown parameters are K and P, of which P can be derived from fixing the shard size to maximum PCIe bandwidth (Equation (2)). With the limited GPU memory size M, the maximum number of concurrent transfers is bounded by the size of vertex interval plus the sizes of concurrent shards. For instance, based on (1) and (2), we can estimate the optimal number of shards being transferred concurrently to be 2 for our NVIDIA K20 Kepler GPU with 4.8 GB memory.

4.4 Computation Engine

The *Computation Engine* shown as 3 in Figure 10 is mainly responsible for GPU in-memory computation (i.e., parallelize each phase of the GAS model) and to send feedback information to the *Data Movement Engine* about the computation frontier used for the next iteration (discussed in Section 2.2) .

Recall the shard data structure illustrated in Figure 7, where edge_update_array and vertex_update_array are the update lists of the vertices and their in-edges, respectively, in the corresponding interval (or shard). They store the updates from the *Gather* and *Scatter* phases of the programming model, shown in Figure 12(a). With these data structures at the very top level, GraphReduce implements a variation of the GAS programming model [28, 19, 26], shown in Figure 12(b). It includes the following five phases, where every iteration is over all shards instead of all edges:

- *gatherMap*: this function fetches all the updates and messages along the in-edges, getting each edge the state of the source vertex and updating that in the edge_update_array or GatherTemp data structure.
- *gatherReduce*: reduces all the collected updates for each vertex with the reduction function defined by programmers.
- *apply*: applies reduced updates to each vertex to obtain new states for the vertices.
- *scatter*: distributes the updated states of the vertices along the out-edges, i.e., update the edge states of the out-edges of the vertices in each shard. In this phase, only the edge states are updated (if the algorithm allows mutable edge states).
- FrontierActivate: marks the set of edges and vertices that would be active in the next iteration. *This phase is not user-defined but auto-generated by the GR framework.*

For each phase in Figure 12(b), GraphReduce requires memcpy-in and memcpy-out operations so as to process all shards of the entire graph. The next phase will not start until the previous phase has been completed (following the model of Bulk Synchronous Parallelism). This can be optimized through dynamic phase fusion and elimination through the *Phase Fusion Engine* inside the Compute Engine, discussed in Section 5.3.

Parallelism in different phases exists at two levels. First, operations are run concurrently within each shard in a given phase. Second, in a given phase, computation across different shards can also be executed concurrently (i.e., multiple shards residing in GPU memory at the same time), because there are no data dependencies between shards in the same phase. The device code in Figure 12(b) shows how GraphReduce exploits this two-level parallelism. This also motivates the use of a hybrid programming model explained in Section 3.1, as we use an edge-centric implementation for gatherMap, scatter, and FrontierActivate phases, but use a vertex-centric implementation for the gatherReduce and apply phases.

Note that the *Function Pointer Table* inside the Compute Engine can take user-defined load-balancing strategies as plug-ins. In the current version of GraphReduce, we apply *CTA (Cooperative Thread Array) load balancing* from the Modern GPU library [6]. Scan operations and merge-sorts are also implemented using the Modern GPU library.

5. OPTIMIZATIONS

5.1 Asynchronous Execution and the Spray Operation

GraphReduce asynchronously performs computation and communication. Specifically, it leverages CUDA Streams, double buffering, and hardware support like Hyper-Qs provided by architectures like NVIDIA's Kepler, to enable data streaming and computation in parallel. As shown in Figure 10, the *Static Stream Creator* of the Data Movement Engine spawns separate CUDA Streams to launch multiple kernels and to transfer data asynchronously, overlapping memory copies within and across phases of computation. Novelty in GraphReduce is its use of separate CUDA Streams for **deep copy operations**, in order to take advantage of the large number of hardware queues offered by modern GPU architectures. This is motivated by the fact that a shard in GraphReduce is not a single contiguous byte-array, but consists of many sub-arrays containing edge, vertex, and frontier data. Each of these sub-arrays requires a separate deep copy to move data between GPU and host. GraphReduce exploits this fact, as shown in Figure 11(b), by not moving the entire shard in one copy performed by a single CUDA stream, but instead, the *Spray Stream Creator* in ❷ dynamically spawns multiple CUDA Streams to move these sub-arrays to the GPU. The outcome is concurrent use of the GPU's many hardware queues, which consequently improves overall throughput.

5.2 Dynamic Frontier Management

As discussed in Section 2.2, with irregularity in graph processing, in every computation iteration, the frontier size is not constant, varying with graph algorithms and datasets. For instance, as shown in Figure 3(c), for dataset Cage-15 processed through BFS, only one vertex is active for the first iteration. Inactive vertices or edges in each iteration can result in significant performance degradation due to GPU thread idling. To address this problem, we integrate the *Frontier Manager module* into the Computation Engine to maintain the list of active vertices whose states have changed in the current iteration and uses it to determine the set of vertices in one hop neighborhood that will be active in the next iteration (based on out-edges information in a shard). GraphReduce then uses this frontier information of the *next* iteration to avoid unnecessary memcpys and kernel launching. It also uses the active vertex information of the *current* iteration for CTA load balancing to avoid GPU core idling.

5.3 Dynamic Phase Fusion/Elimination

A graph algorithm implemented with GraphReduce need not implement user-defined functions for all three GAS phases. For example, BFS need not implement the Scatter phase. In response, when a phase is elided by the user, GraphReduce eliminates the repeated and unnecessary movement of shards into GPU memory, before and after that phase, in each iteration. This is termed phase elimination. For instance, if the graph algorithm does not have a defined gather function, GraphReduce will avoid bringing in the entire shard (in-edges+out-edges), only moving the out-edges to the GPU memory (because in-edges are used only in the Gather phase). Out-edges are moved regardless, because the FrontierActivate phase operates even when there is no scatter phase defined. The resulting dynamic phase elimination reduces unnecessary kernel launching and data movement.

In certain scenarios, merging two or more GAS phases is possible, again to avoid unnecessary extra data movement. For example, if a graph algorithm only defines Apply and FrontierActivate phases, GraphReduce will automatically merge these two phases, thus avoiding the *memcopy* operations that would have been required for executing the two phases separately. We term this action dynamic phase fusion. An example of a graph algorithm for which this method is used is again BFS. It only requires users to define the apply phase, in which the BFS tree depth for every vertex is marked to be the iteration number. GraphReduce will automatically merge the Apply and FrontierActivate phases for BFS. Note that Dynamic Fusion/Elimination functionalities are enabled through the *Phase Fusion Engine* inside the Compute Engine.

6. EXPERIMENTAL EVALUATION

6.1 Experimental Setup

Evaluation Platform: GraphReduce is evaluated on a typical heterogeneous HPC node equipped with 16-core Intel Xeon E5-2670 processors running at 2.6 GHz with 32 GB of DDR3 RAM, and one attached NVIDIA Tesla K20c GPU with 13 SMX multiprocessors and 4.8 GB GDDR5 RAM. The Kepler GPU is enabled with CUDA 6.5 runtime and the version 340.29 driver, while the host CPU side is running Fedora version 20 with kernel v.3.11.10-301 x86. All the runs are compiled with the highest optimization level flag.

Graph Dataset. Shown in Table 1, we evaluate the performance and efficiency of GraphReduce using two types of graph inputs: small size graphs that will fit into GPU memory (named In-memory graphs) and large graphs that do not fit (named Out-of-memory graphs). Here, we define the size of a graph as the amount of memory required to store the edges, vertices, and edge/vertex data states in terms of the user-defined datatypes and a few of the temporary buffers. All experiments use datatype *float*. Note that the size of a graph can expand after loading it to in-memory buffers, because the size of the datatypes for edge and vertex states is in general larger than their representations in the raw graph format (e.g., *char*).

In-memory graphs are used to evaluate the effectiveness of GraphReduce's in-memory optimizations against other state-of-the-art in-memory approaches (e.g., MapGraph and CuSha), while Out-of-memory graphs are used to evaluate it against frameworks that can process large graph sets (e.g., GraphChi and X-Stream).

The ten real-world graphs listed in Table 1 are publicly available and cover a wide range of sparsity and sizes. For example, *orkut* is an undirected social network, in which vertices and edges represent the friendship between users. *uk-2002* is a large crawl of the *.uk* domains, in which vertices are the pages and edges are the links. *nlpkkt160* is from the 3D PDE-constrained optimization problem with vertices as state variables and edges as control variables.

Evaluated Algorithms. Four widely used algorithms are evaluated, including Breadth First search (BFS), Page Rank (PR), Single-Source Shortest Paths (SSSP), and Connected Components (CC). Algorithms requiring undirected graphs as inputs, e.g., connected components, are stored as pairs of directed edges.

6.2 Evaluation and Analysis

6.2.1 Comparison with Out-of-Memory Frameworks

Graph		BFS	SSSP	Pagerank	CC
kron-logn21	GraphChi	365	442	328	236
	Xstream	95	97	98	97
	GR	4	7	93	9
nlpkkt160	GraphChi	503	510	447	1560
	Xstream	128	136	144	133
	GR	60	92	140	183
uk-2002	GraphChi	1100	1283	1091	1073
	Xstream	330	374	335	348
	GR	49	80	153	162
orkut	GraphChi	311	320	285	268
	Xstream	124	131	127	127
	GR	6	10	84	16
cage15	GraphChi	262	265	240	389
	Xstream	114	119	115	143
	GR	18	25	19	41

Table 3: Execution times of out-of-memory graph processing frameworks on different algorithms and graph inputs. Reported times are wall time and in **seconds**.

Graph		BFS	SSSP	Pagerank	CC
ak2010	MG	7.94	79.01	23.86	19.03
	CuSha	7.75	31.99	12.08	10.16
	GR	9.26	3.81	14.61	17.78
coAuthorsDBLP	MG	5.28	8.75	68.92	30.26
	CuSha	11.55	12.75	79.84	13.99
	GR	5.31	5.42	53.14	16.43
kron-logn20	MG	51.81	139.43	6789	308.91
	CuSha	119.82	269.88	1852	138.7
	GR	27.88	28.34	4365	266.86
webbase-1M	MG	8.71	13.56	72.86	50.97
	CuSha	13.52	12.65	270.83	317.41
	GR	1.4	6.07	57.76	37.45
belgium_osm	MG	195.79	261.32	102.64	2219
	CuSha	791.3	897.03	45.8	920.7
	GR	279.8	281.39	71.33	40.63

Table 4: Performance results of in-memory (small) graph processing frameworks on different algorithms and graph inputs. Reported times are in **milliseconds**. **MG stands for MapGraph.**

Since the state-of-the-art GPU-based graph processing approaches [17, 10, 43, 22] assume that input graphs fit in GPU memory, we compare GR's out-of-memory performance with Graphchi [23] and X-Stream [32], both state-of-the-art, out-of-memory, CPU-based frameworks targeting large real-world graphs. For fairness in comparison, the datasets chosen (in-memory sizes shown in Table 1) fit in host memory but do not fit into GPU memory. This is to avoid I/O (SSD access) overheads in systems like GraphChi and X-Stream. GR, however, incurs the unavoidable costs of moving *shards* in and out of GPU memory.

Shown in Table 3, Figure 13, and Figure 14, GR achieves an average speedup of **13.4x and 5x** over GraphChi and X-Stream (running with 16 threads), respectively, despite its need to move data between GPU and CPU via PCIe; while Graphchi and X-Stream benefit from local (host) memory access. GR achieves some significant speedups, e.g., up to 79x over GraphChi and 21x over X-Stream, for kron_g500-logn21 processed by BFS. These performance improvements are due to its (i) asynchronous mode and spray operation (leveraging CUDA Streams, Hyper-Qs, and deep memory copy operations); (ii) dynamic frontier management, to avoid unnecessary kernel launching and GPU core idling; and (iii) dynamic phase fusion/elimination to remove unnecessary data movement. The hybrid programming model also contributes to the performance improvements over GraphChi and X-Stream, by extracting access pattern-based parallelism opportunities across different phases. GraphChi (vertex-centric) and X-Stream (edge-centric), on the other

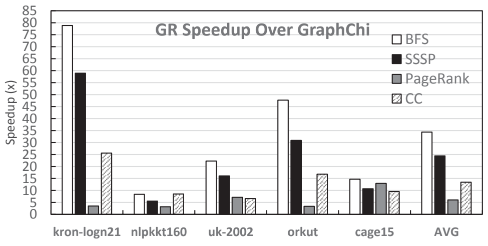


Figure 13: GR's speedup over GraphChi for various algorithms and out-of-memory graph inputs.

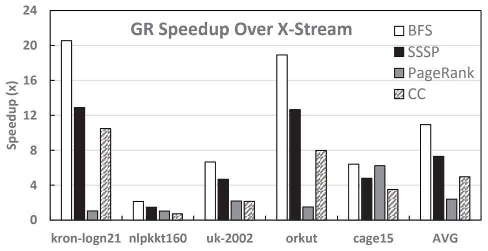


Figure 14: GR's speedup over X-Stream for various algorithms and out-of-memory graph inputs.

hand, suffer from significant random accesses to either their edge or vertex sets, due to their use of a unified model. There is only one case that X-Stream performs slightly better than GR, which is the the nlpkkt160 input processed by CC. This is due to the fact that GR experiences substantial overheads from the large data movement over PCIe, and these overheads are not sufficiently compensated by the massive parallelism offered by GPU. GrapChi and X-Stream, in comparison, have all data accessible locally in the host memory and are therefore, not subject to such overheads.

6.2.2 Comparison with GPU In-Memory Frameworks

The results above establish GR's ability to process large graphs that do not fit into GPU memory, at levels of performance higher than that seen for CPU-based solutions. In other words, additional costs arising from GPU-host data movement are typically dwarfed by the performance advantages offered by fast GPUs. At the same time, GR also performs as well as the existing in-GPU-memory solutions for smaller graph inputs. Table 4 shows GR's in-memory performance for smaller graphs to be comparable to the state-of-the-art in-memory processing frameworks like MapGraph (MG) and CuSha, which apply multi-level fine-tuned optimizations for in-GPU workloads. In many cases, GR outperforms MG and CuSha significantly, e.g., kron_g500-logn20 with SSSP and webbase-1M with BFS. For processing these smaller graphs, one major contributing factor for high performance in GR is its use of active vertex information of the same iteration for the CTA load balancing. One interesting observation is that not all of the GPU in-memory graph processing approaches work well for every graph input and algorithm. This prompts us to (also see Sections 4.2 and 4.4) to add flexibility to GR's Partition Engine – the Partition Logic Table can be easily modified to incorporate desired user-defined specific optimizations, e.g., to use the partition and graph layout algorithms employed in CuSha.

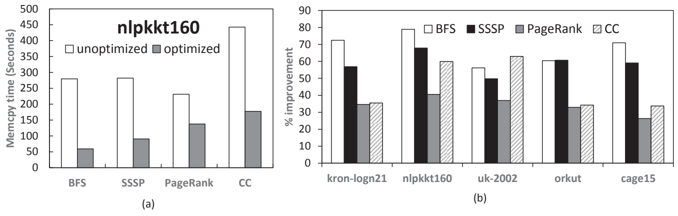


Figure 15: Performance gained from memcpy optimization. (a) Actual memcpy time comparison between optimized and unoptimized GR for nlpkktt160. (b) Percentage improvement of memcpy performance from optimized GR against unoptimized GR.

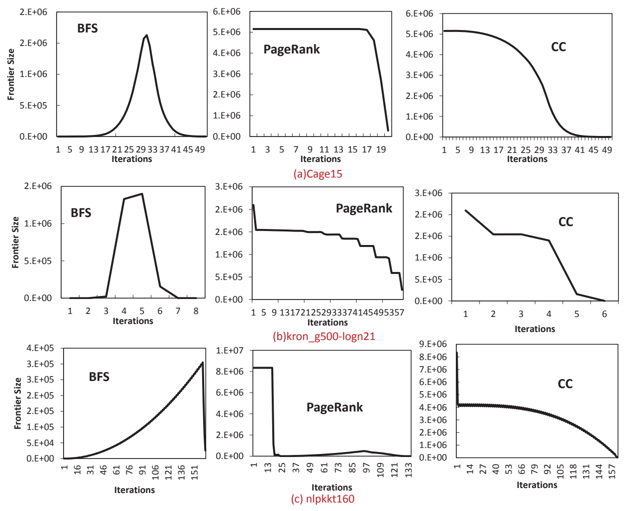


Figure 16: Frontier size changes across iterations shown for several large out-of-memory graphs with three algorithms.

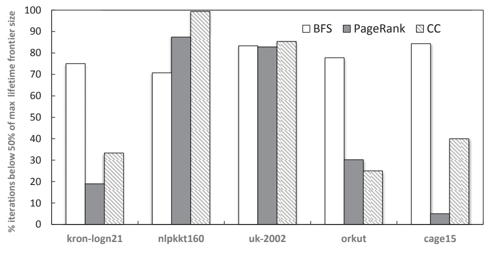


Figure 17: For out-of-memory graphs, percentage of iterations that are below 50% of the max lifetime frontier size.

6.2.3 *Performance Effects of GraphReduce Optimizations*

Experimental results show memcpy time to be a dominant factor for performance, occupying on the average above 95% of the total execution time for the five large out-of-memory graph inputs studied above. This makes it the primary target for GR optimizations. Figure 15 shows the performance improvements gained from the three optimizations discussed in Section 5, including asynchronous execution/spray operation, dynamic frontier management, and dynamic phase fusion/elimination. For example, without these optimizations, Figure 15(a) shows that the performance of nlpkkt160 suffers significantly from memcpy, e.g., up to 443 seconds for the CC with the total execution time only being 451 seconds. With the optimizations, memcpy time drops from 443 seconds to 178 seconds, improved by 60%. Figure 15(b) shows the percentage improvement of memcpy time from the optimized GR over the baseline unoptimized scenario, with an average of 51.5% and up to 78.8 % across all large datasets and algorithms.

A simple example shows how dynamic frontier management affects memcpy time. Figure 16 shows that frontier sizes vary with the iterations for three large graph inputs and three algorithms (Note: SSSP is not included here because the frontier patterns for BFS and SSSP are very similar as BFS is essentially SSSP with equal edge weights). This indicates that the basic pattern (or shape) of the frontier graphs is algorithm-dependent, e.g. for BFS it starts with 1 active vertex, then the frontier size climbs up, attains a peak and eventually falls. On the other hand, for PageRank and CC, the number of active vertices starts with the total number of vertices in the graph and then falls as the algorithm progresses. Another key insight that can be drawn from Figure 16 is that the rate at which the frontier size changes with iterations is input-dependent, e.g. for PageRank with Cage15, the number of active vertices remains high for the most part of the algorithm but for nlpkkt it drops very quickly at the beginning. Figure 17 quantifies this dataset-specific frontier size behavior by showing the percentage of iterations with active vertices that are below 50% of the max lifetime frontier size (the peak value shown in Figure 16) across five large data graphs and three algorithms.Combining this figure with Figure 15(b), we can observe that graph inputs with higher percentage of iterations with small frontier sizes benefit the most from the dynamic frontier management. For example, BFS with maximum percentage of iterations with very low activity consistently show substantial improvement in memcpy time across all datasets. For PageRank and CC, several datasets (e.g., nlpkk and uk2002) that have very low activity across iterations, also benefit the most in memcpy-time reduction, with 60% and 63% overhead reduction in CC and 40% and 37% in PageRank respectively.

6.2.4 *Discussion*

Experiments demonstrate that (1) GraphReduce can process graphs of sizes larger than GPU memory, achieving up to 79x and 21x, and an average of 13.4x and 5x, speedup over the competing CPU-based methods implemented in GraphChi and X-Stream respectively, for several real-world large-scale graphs with various algorithms. (2) GR performance is comparable to that of existing in-GPU- memory solutions like MapGraph and CuSha, for smaller input graphs (i.e., those that fit into GPU memory). (3) Memcpy time is the dominant factor in GR's graph processing, occupying on the average above 95% of total execution time. Because there is a strong correlation between the change in active vertices per iteration vs. the amount of unnecessary data movement, the more inactive vertices there are per iteration, the more opportunities exist to avoid such unnecessary data copies. This is evident from performance results showing the effects of GR's dynamic frontier management. Finally, (4) GR employs additional optimizations that include concurrent copy operations, overlapping computation and communication operations; deep copies via spray streams, leveraging the multiple hardware queues (Hyper-Qs) in GPUs; and performs dynamic phase merging and elimination to avoid unnecessary data copying. With these optimizations, GR achieves an average of 51.5% and up to 78.8% reduction in memcpy time across the large datasets and algorithms used in our evaluation.

7. RELATED WORK

In-Memory Graph Processing. Merrill *et al.*[29] present a parallelization of BFS tailored to the GPU's requirement for large amounts of fine-grained BSP; they achieve an asymptotically optimal $O(|V| + |E|)$ work complexity. Duong *et al.* [14] conduct detailed GPU-based optimizations for PageRank and achieves significant speedup over a multi-core CPU implementation. Chapuis *et al.* [13] provide an algorithmic optimization solution to speedup all-pairs shortestpath (APSP) for planar graphs that exploits the massive on-chip parallelism available on GPUs. GraphReduce can be extended to implement the algorithm-specific optimizations above and in contrast to such work, it offers user-level APIs for programming graph algorithms and provides a general framework addressing a wide range of parallel graph algorithms and hiding architecture-level optimizations from users.

Concerning frameworks for GPU-based graph processing, earlier work like *Medusha* [43] introduces some basic graph-centric optimizations for GPUs, offering a small set of user-defined APIs, but its performance is not comparable to the state-of-the-art low-level GPU optimizations. To address this issue, *MapGraph* [17] and *VertexAPI*[10] implement runtime-based programming frameworks with levels of performance that match those seen for low-level specific algorithm optimizations. *MapGraph* chooses among different scheduling strategies, depending on the size of the frontier and the adjacency lists for the vertices in the frontier. *VertexAPI* provides a GAS model-based GPU library, gaining high performance primarily from using the ModernGPU [6] library for load balancing and memory coalescing. *CuSha* [22] identifies the shortcomings of the state-of-the-art CSR-based virtual warp-centric method for processing graphs on GPUs and in response, proposes G-Shards and Concatenated Windows to address its performance inefficiency. All of the approaches above make the fundamental assumption that large graphs fit into GPU memory, a restriction that is not present for GraphReduce. As discussed in Section 6, GraphReduce not only addresses the processing of out-of-memory graphs, but also matches the in-memory performance seen with these state-of-the-art approaches, in many cases outperforming them significantly.

Out-of-Memory Graph Processing. Out-of-Memory graph processing has been concerned with CPU-based hosts processing graphs that do no fit into host memory. *GraphChi* [23], for instance, is based on a vertex-centric implementation of graph algorithms where graphs are sharded onto the SSD drives attached to the host. Its SSD-targeting sharding methods motivate GraphReduce's approach to how GPUs view and interact with host memory. We also borrow from X-Stream [32] the edge-centric way to organize data for our GAS model. *Totem* [18] offers a high-level abstraction for graph processing on GPU-based systems, by statically partitioning graphs into GPU and host memories, placing low-degree vertices on the host and high-degree vertices on the GPU. The approach improves performance if the graphs follow a power-law vertex degree distribution, and as graph size increases, only a fixed sub-graph able to fit in GPU memory will be processed, resulting in GPU underutilization and eventual CPU-based bottlenecks for graph processing. *Green-Marl* [20] is a Domain Specific Language (DSL) for efficient graph analysis on CPUs; its implementation is not amenable to many-core architectures.

8. CONCLUSIONS AND FUTURE WORK

In contrast to the previous work, GraphReduce (GAS model based) is able to process graphs of sizes much exceeding that of GPU memory, by sharding graph data and asynchronously moving shards between GPU and host memories. Technical advances offered by GraphReduce include its usage of a hybrid programming model of edge- and vertex-centric processing, asynchronous execution/spray operation, dynamic phase fusion/elimination, and dynamic frontier management. With these optimizations, GraphReduce achieves levels of performance similar to those of prior in-GPU-memory and significant speedup over out-of-memory implementations of graph processing like GraphChi and X-Stream respectively, for several real-world large-scale graphs processed by various algorithms. Further, as a framework, GraphReduce permits the usage of alternative data partitioning schemes and associated data layout methods, thereby enabling extensions that can take advantage of the state-of-the-art schemes for graph processing developed elsewhere.

There are several interesting future directions of our work, including: (1) extending GraphReduce to support multiple on-node GPUs, (2) addressing the limited on-node memory size through the usage of SSD and other storage devices; (3) processing dynamically evolving graphs; (4) understanding how dynamic profiling and processor choice (i.e., GPU vs. CPU execution) [16, 38, 37] could be integrated into GraphReduce; and (5) adapting architectural- and runtime-level optimizations to further improve performance and energy efficiency of the highly irregular graph algorithm [25].

9. ACKNOWLEDGMENTS

We would like to thank James Lewis and Brad Bebee from SYSTAP for providing us the newest version of MapGraph, and Amitabha Roy from EPFL for his help with X-Stream setup. The authors also thank the anonymous reviewers for the feedback that helped improve the paper. This work was partially supported by DOE ASCR Beyond Standard Modeling (BSM) Project and a gift grant from Intel. The Pacific Northwest National Laboratory is operated by Battelle for the U.S. Department of Energy under contract DE-AC05-76RL01830.

10. REFERENCES

[1] Cage15: www.cise.ufl.edu/research/sparse/matrices/vanheukelum.
[2] CUDA 7.0: https://developer.nvidia.com/cuda-downloads/.
[3] DIMACS10 ak2010: www.cise.ufl.edu/research/sparse/matrices/dimacs10.
[4] DIMACS10 belgium_osm: www.cise.ufl.edu/research/sparse/matrices/dimacs10.
[5] DIMACS10 coAuthorsDBLP: www.cc.gatech.edu/dimacs10/data/coauthor/.
[6] Modern GPU library:http://nvlabs.github.io/moderngpu/.
[7] NVIDIA Kepler GK110 white paper. 2012.
[8] uk-2002: http://law.di.unimi.it/webdata/uk-2002/.
[9] Unified Virtual Addressing: http://devblogs.nvidia.com/parallelforall/unified-memory-in-cuda-6/.
[10] VertexAPI2: http://www.royal-caliber.com/main.html. 2015.
[11] Yahoo WebScope: G7 webscope.sandbox.yahoo.com/catalog.php?datatype=g.
[12] N. Bell and M. Garland. Efficient sparse matrix-vector multiplication on CUDA. NVIDIA Technical Report NVR-2008-004, NVIDIA Corporation, Dec. 2008.

[13] H. Djidjev, S. Thulasidasan, G. Chapuis, R. Andonov, and D. Lavenier. Efficient multi-gpu computation of all-pairs shortest paths. In *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, pages 360–369, May 2014.

[14] 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*, SoICT '12, pages 223–230, New York, NY, USA, 2012. ACM.

[15] N. Farooqui, C. J. Rossbach, Y. Yu, and K. Schwan. Leo: A profile-driven dynamic optimization framework for gpu applications. In *2014 Conference on Timely Results in Operating Systems (TRIOS 14)*, Broomfield, CO, Oct. 2014. USENIX Association.

[16] N. Farooqui, K. Schwan, and S. Yalamanchili. Efficient instrumentation of gpgpu applications using information flow analysis and symbolic execution. In *Proceedings of Workshop on General Purpose Processing Using GPUs*, GPGPU-7, pages 19:19–19:27, New York, NY, USA, 2014. ACM.

[17] Z. Fu, M. Personick, and B. Thompson. Mapgraph: A high level api for fast development of high performance graph analytics on gpus. In *Proceedings of Workshop on GRAph Data Management Experiences and Systems*, GRADES'14, pages 2:1–2:6, New York, NY, USA, 2014. ACM.

[18] A. Gharaibeh, L. Beltrão Costa, E. Santos-Neto, and M. Ripeanu. A yoke of oxen and a thousand chickens for heavy lifting graph processing. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, PACT '12, pages 345–354, New York, NY, USA, 2012. ACM.

[19] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 17–30, Hollywood, CA, 2012. USENIX.

[20] S. Hong, H. Chafi, E. Sedlar, and K. Olukotun. Green-marl: A dsl for easy and efficient graph analysis. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 349–362, New York, NY, USA, 2012. ACM.

[21] S. Hong, T. Oguntebi, and K. Olukotun. Efficient parallel graph exploration on multi-core cpu and gpu. In *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*, pages 78–88, Oct 2011.

[22] F. Khorasani, K. Vora, R. Gupta, and L. N. Bhuyan. Cusha: Vertex-centric graph processing on gpus. In *Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing*, HPDC '14, pages 239–252, New York, NY, USA, 2014. ACM.

[23] 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*, OSDI'12, pages 31–46, Berkeley, CA, USA, 2012. USENIX Association.

[24] J. Leskovec, K. J. Lang, A. Dasgupta, and M. W. Mahoney. Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters, 2008.

[25] C. Li, S. L. Song, H. Dai, A. Sidelnik, S. K. S. Hari, and H. Zhou. Locality-driven dynamic gpu cache bypassing. In *Proceedings of the 29th ACM on International Conference on Supercomputing*, ICS '15, pages 67–77, New York, NY, USA, 2015. ACM.

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

[27] A. LUMSDAINE, D. GREGOR, B. HENDRICKSON, and J. BERRY. Challenges in parallel graph processing. *Parallel Processing Letters*, 17(01):5–20, 2007.

[28] 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 *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 135–146, New York, NY, USA, 2010. ACM.

[29] 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*, PPoPP '12, pages 117–128, New York, NY, USA, 2012. ACM.

[30] L. Page, S. Brin, R. Motwani, and T. Winograd. The pagerank citation ranking: Bringing order to the web. 1999.

[31] R. A. Rossi and N. K. Ahmed. The network data repository with interactive graph analytics and visualization. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*, 2015.

[32] A. Roy, I. Mihailovic, and W. Zwaenepoel. X-stream: Edge-centric graph processing using streaming partitions. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 472–488, New York, NY, USA, 2013. ACM.

[33] O. Schenk, A. WÃd'chter, and M. Weiser. Inertia-revealing preconditioning for large-scale nonconvex constrained optimization. *SIAM Journal on Scientific Computing*, 31(2):939–960, 2009.

[34] D. Sengupta, R. Belapure, and K. Schwan. Multi-tenancy on gpgpu-based servers. In *Proceedings of the 7th International Workshop on Virtualization Technologies in Distributed Computing*, VTDC '13, pages 3–10, New York, NY, USA, 2013. ACM.

[35] D. Sengupta, A. Goswami, K. Schwan, and K. Pallavi. Scheduling multi-tenant cloud workloads on accelerator-based systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '14, pages 513–524, Piscataway, NJ, USA, 2014. IEEE Press.

[36] D. Sengupta, A. Kapil, S. Song, and K. Schwan. Graphreduce: Large-scale graph analytics on accelerator-based hpc systems. In *Parallel and Distributed Processing Symposium Workshop (IPDPSW), 2015 IEEE 28th International*, May 2015.

[37] S. Song and K. Cameron. System-level power-performance efficiency modeling for emergent gpu architectures. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, PACT '12, pages 473–474, New York, NY, USA, 2012. ACM.

[38] S. Song, C. Su, B. Rountree, and K. Cameron. A simplified and accurate model of power-performance efficiency on emergent gpu architectures. In *Parallel Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, pages 673–686, May 2013.

[39] L. G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, Aug. 1990.

[40] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. In *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, SC '07, pages 38:1–38:12, New York, NY, USA, 2007. ACM.

[41] J. Yang and J. Leskovec. Defining and evaluating network communities based on ground-truth. In *Proceedings of the ACM SIGKDD Workshop on Mining Data Semantics*, MDS '12, pages 3:1–3:8, New York, NY, USA, 2012. ACM.

[42] A. Yoo, E. Chow, K. Henderson, W. McLendon, B. Hendrickson, and U. Catalyurek. A scalable distributed parallel breadth-first search algorithm on bluegene/l. In *Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*, SC '05, pages 25–, Washington, DC, USA, 2005. IEEE Computer Society.

[43] J. Zhong and B. He. Medusa: A parallel graph processing system on graphics processors. 2013.