MultiGraph: Efficient Graph Processing on GPUs

Changwan Hong, Aravind Sukumaran-Rajam, Jinsung Kim, P. Sadayappan
Department of Computer Science and Engineering
The Ohio State University
Columbus, OH 43201.
{hong.589, sukumaranrajam.1, kim.4232, sadayappan.1}@osu.edu

Abstract

High-level GPU graph processing frameworks are an attractive alternative for achieving both high productivity and high performance. Hence, several high-level frameworks for graph processing on GPUs have been developed. In this paper, we develop an approach to graph processing on GPUs that seeks to overcome some of the performance limitations of existing frameworks. It uses multiple data representation and execution strategies for dense versus sparse vertex frontiers, dependent on the fraction of active graph vertices. A two-phase edge processing approach trades off extra data movement for improved load balancing across GPU threads, by using a 2D blocked representation for edge data. Experimental results demonstrate performance improvement over current state-of-the-art GPU graph processing frameworks for many benchmark programs and data sets.

1. Introduction

GPUs offer the potential for higher performance and energy efficiency than multi-core processors. However, actually achieving high performance with GPUs is non-trivial. It generally requires significant programmer expertise and understanding of details on low-level execution mechanisms in GPUs, and attention to a number of considerations essential to achieving high performance. Such development of high-performance GPU algorithms is very time consuming for GPU experts and is not very feasible for the vast number of developers of new data/graph analytics methods.

Therefore, there is considerable interest in developing domain-specific graph processing frameworks that offer application developers a convenient high-level abstraction for developing their algorithms, and also deliver high performance on GPUs. Several such GPU graph processing frameworks have been recently developed, including VWC [9], MapGraph [7], Medusa [22], CuSha [11], WS [10], Frog [19], GreenMarl [8], Falcon [4], Groute [2], and Gunrock [21]. While these frameworks achieve much higher performance than popular vertex-centric graph processing frameworks like Pregel [14], Giraph [20], GraphLab [13] etc., as elaborated in the next section, current GPU graph processing frameworks still have some performance limitations. In this paper, we identify sources of performance limitation for current GPU graph processing frameworks and then present the MultiGraph[1] system for graph processing on GPUs.

[1] (https://github.com/hochawa/multigraph)

The approaches we present in this paper could be incorporated into existing GPU graph processing frameworks like Gunrock, Groute, WS, etc. A key hypothesis behind this work is that graphs and graph traversals exhibit significant non-uniformity and therefore a single data representation or execution strategy is unlikely to be effective across the board. By identifying important use cases and execution scenarios, we develop a GPU graph processing framework that uses multiple internal representations of the graph as well as different execution strategies for different use cases of graph traversal. The key ideas behind the proposed approach are as follows:

- Multiple data representation and execution strategies are used for dense versus sparse vertex frontiers, dependent on the fraction of the graph vertices that are active in a given iteration. Topology-driven algorithms [18] process every graph vertex and edge during each iteration. A different graph representation and a different vertex/edge processing strategy is used for this use case, in contrast to the scenario where only a small fraction of vertices is active in a given iteration.
- A two-phase edge processing approach using a 2D blocked distribution facilitates improved load balancing across GPU threads, and improved global-memory access efficiency. While 2D partitioning strategies have been used earlier for coarse-grained multi-GPU parallelization [3], we are unaware of its prior use for improving fine-grained parallelization in a GPU.
- Different representations of edge data for high-degree vertices versus low-degree vertices enables avoidance of expensive global-memory atomic operations typically used by GPU graph frameworks.
- Efficient dynamic work distribution is achieved for sparse frontier processing by grouping a vertex's edges into bins of three granularities: thread-block, warp, and thread.

We present experimental data comparing performance with the proposed approach and state-of-the-art GPU graph-processing frameworks, using graph datasets that have been used in other recent studies. We show that the new approach is able to achieve very good performance for a range of benchmarks.

2. Background and Motivation

Several customized frameworks have been developed for implementation of graph algorithms on GPUs [9, 11, 10, 21, 2]

978-1-5090-6764-0/17 $31.00 © 2017 IEEE
DOI 10.1109/PACT.2017.48

The motivation for the developments described in this paper is that existing frameworks for graph processing on GPUs have performance limitations stemming from one or more of the challenges in achieving high performance on GPUs. In this section, we first provide general background on graph processing frameworks, followed by a discussion on specific sources of overhead with current state-of-the-art GPU graph processing frameworks.

Graph Processing Frameworks Graph processing frameworks facilitate the convenient development of portable high-performance algorithms that iteratively traverse "active" graph vertices and/or edges, performing some operation to update vertex/edge attributes. For example, let us consider Breadth-First Search (BFS). An attribute is associated with each vertex to designate its level in the BFS. This attribute is initialized to infinity (a number larger than the number of vertices) for all but the root vertex, whose level is set to zero. At each iteration, an active set of vertices is processed, with the active vertex at the first step being the root vertex. The processing of an active vertex involves traversing each of its outgoing edges to determine the current level of the destination vertices. If a neighbor vertex has a current level of infinity, its level is set to one more than that of the active source vertex. All such vertices that have their level set in the current iteration are placed in the active front for the next iteration. This process is repeated until the levels of all vertices get finalized and the active front for the next iteration is empty.

GPU Performance Challenges There is parallelism at two levels for each iteration of the BFS execution described above: i) each vertex in the current active front can be processed in parallel, and ii) each outgoing edge of an active vertex can be processed in parallel. When edges are processed, the identification of the active vertices for the next front can be done in parallel. However, achieving efficient parallelization on GPUs poses challenges.

- **Load Balancing:** One option for work distribution is to assign each active vertex to a thread, but different threads may have very different amounts of work since the degrees of the active vertices can differ significantly. Another option is to assign each outgoing edge of an active vertex to a thread, but this is challenging to do efficiently: each thread may spend more time identifying its edge attributes than in processing it.
- **Concurrent Edge Processing:** Often, the processing of an edge entails some modification of an attribute at the destination vertex. Since two concurrently processed edges may point to a common destination vertex, atomic operations may need to be used, which can be quite expensive, especially if performed on global memory locations.
- **Formation of Next Front:** The insertion of vertices/edges to form the next active front requires coordination among concurrently executing threads.
- **Uncoalesced Global Memory Data Access:** Access of attributes from the set of destination vertices of a set of active edges will generally require inefficient uncoalesced access since they will generally be scattered and not contiguously located in global memory.

Analysis of Existing Frameworks We now contrast different GPU graph processing frameworks in terms of their merits and challenges.

CuSha [11]: The CuSha framework was the first to address the limitation of uncoalesced global memory data accesses for GPU graph processing, by performing updates in shared memory. Instead of the standard CSR data structure, it uses an alternative G-Shard representation. CuSha is a framework for topology-driven algorithms, making it inefficient for data-driven algorithms like BFS. Although results are updated in shared memory, CuSha can suffer from serialization overhead from atomics when processing highly skewed datasets.

WS [10]: WS (Warp Segmentation) provides an edge-distributing load-balancing strategy using binary search to locate the vertex ID corresponding to an edge in CSR format. This kind of load balancing idea, first proposed by Andrew et al. [5], is now widely used in many GPU frameworks such as Gunrock. WS is also a framework for topology-driven algorithms and therefore not well suited for data-driven algorithms. In addition, it also incurs uncoalesced global memory accesses for vertex values, due to use of the CSR representation.

Gunrock [21]: Gunrock uses a data-centric abstraction focused on operations on frontiers (vertex or edge). These primitives help programmers develop new graph algorithms without much effort. Gunrock is currently regarded as one of the state-of-the-art GPU graph processing systems. However, Gunrock incurs overhead due to uncoalesced global memory accesses and global memory atomic operations. When the frontier size is large, Gunrock's performance is limited by these factors.

IrGL [17]: Pai et al. [17] identify three factors that limited GPU performance for processing graph algorithms. To resolve these limitations, they introduce three optimizations: iteration outlining, hierarchical aggregation, and nested parallelism. The IRGL compiler produces CUDA code from an intermediate-level program representation. They demonstrate excellent performance when these limitations are resolved. However, as with Gunrock, uncoalesced global memory accesses and global-memory atomic operations can limit performance.

Groute [2]: Contrary to synchronous GPU frameworks, Groute implements a scalable asynchronous model for graph processing. Performance on mesh-like networks is considerably better than synchronous frameworks like Gunrock. However, the issues with uncoalesced global memory accesses and global-memory atomics can limit performance.

Addressing Performance Limiters In this paper, we develop an approach to GPU graph processing that alleviates

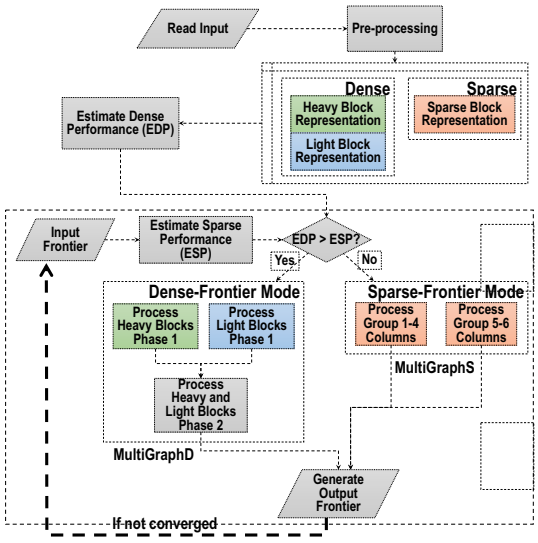


Figure 1: Overview of MultiGraph System

some of the performance limiters of current frameworks through use of a dual data representation and different vertex/edge processing strategies that depend on the context. We provide an overview of the approach in the next section.

3. Overview

This section provides an overview of the system for iterative graph processing that is developed in this paper. A guiding hypothesis behind this work is that no single standard graph representation (such as CSR or CSC) offers sufficient flexibility to achieve high processing efficiency for the variety of traversal patterns and graph characteristics encountered in graph applications.

Fig. 1 provides a high-level view of the graph processing system. A primary distinction is made between two scenarios: i) all (or most) vertices are processed in each iteration (**dense input frontier**), ii) only a small subset of vertices is processed in an iteration (**sparse input frontier**) and that active set of vertices is not known until the end of the previous iteration. Different data representations and execution strategies are used for these two cases, as described in detail in Sec. 4 (dense-frontier) and Sec. 5 (sparse-frontier). For dense-frontiers, the same processing happens every iteration and hence the data structures can be set up to optimize execution. In contrast, with sparse-frontiers, the active set can change significantly across iterations, making it more challenging to achieve high performance. The input graph is first read in and a pre-processing step generates the dual representations of the graph, so that it can be processed in either dense or sparse frontier mode, as appropriate.

The dense-frontier mode can also be used (via "masking") for input frontiers where only a subset of vertices is active. Although the sparse-frontier mode only traverses edges corresponding to the active vertices in the front, whereas the dense-frontier mode essentially traverses all edges, the dense-frontier mode can achieve higher performance than the sparse-frontier mode when the fraction of active vertices is sufficiently high. The cross-over front-density threshold depends both on the graph as well as the graph algorithm. Therefore a sampling based approach is used to implement a hybrid algorithm (Sec. 6) that selects either the dense-frontier or sparse-frontier mode of execution for each iteration of an iterative graph algorithm. The choice is made by comparing estimate performance for sparse-front processing (ESP) with estimated performance for dense-front processing (EDP). EDP is independent of the fraction of active vertices and is determined by execution with a small fraction of vertices before the first iteration of execution of the full initial front. ESP is estimated for each iteration by sampling a small fraction of the active vertices and processing them in sparse-front mode. Depending on which of EDP or ESP is larger, the dense-frontier mode or sparse-frontier mode is chosen.

Table 1: Grouping criteria

Group	Criteria
1	in-degree >= 2048
2	1024 <= in-degree <2048
3	512 <= in-degree <1024
4	256 <= in-degree <512
5	128 <= in-degree <256
6	in-degree <128

In describing the algorithmic details of the developed graph processing framework, we use an edge-matrix abstraction to represent the graph. Each element in this matrix represents an edge in the graph. During the pre-processing step, the vertices of the rows of the edge-matrix are reordered by placing vertices into one of six groups, based on number of non-zeros in a row (which is the corresponding vertex's 'in-degree'), as specified in Table 1. Row (vertex) reordering is performed so that vertices in the same group get consecutively numbered. The grouping together of vertices of similar in-degree facilitates load-balanced work distribution across threads, and different edge processing strategies based on the amount of work per vertex. In sparse-frontier mode, different strategies are used for groups 1-4 versus 5-6. In dense-frontier mode, graph edges are grouped by 2D block-partitioning of the index space, with different representations and processing strategies for heavily populated versus lightly covered blocks. After processing the active vertices using the selected mode, the output frontier is created, and the iterative process repeated until the iteration termination condition is reached.

4. MultiGraphD (Dense Frontier)

This section describes MultiGraphD, the two-phase streaming approach for dense-frontier processing. It is aimed at achieving good load balance and low divergence across threads, high warp occupancy, and efficient coalesced data access to/from global memory. In addition, compared to frameworks like Gun-

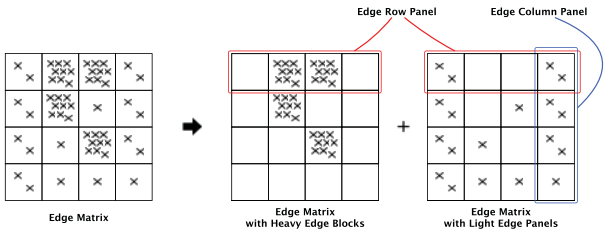
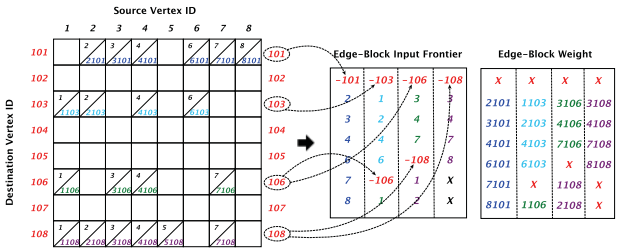


Figure 2: MultiGraphD: Edge matrix partitioning into heavy and light blocks/panels



Number of threads =4; x : denotes invalid values

Figure 3: MultiGraphD: Processing of heavy edge-block

rock, MultiGraphD's two phase scheme is designed to reduce the total number of atomic operations on global memory and global memory transactions (as is quantitatively demonstrated using hardware counter measurements when experimental data is presented in Sec. 7). Processing is performed in two phases:

- In Phase-1, graph edges are streamed in, structured in batches over limited contiguous ranges in the column-index space of the edge matrix, i.e., over a limited range of source vertex IDs for the processed edges. Prior to streaming in a batch of edges, the vertex attributes over that index-range are loaded into shared memory. Multiple contributions for any destination vertex are first locally combined and (destination_index,contribution) tuples are written out into pre-determined locations in an intermediate stream-buffer in GPU global memory.
- In Phase-2, key-value pairs for contributions to destination vertices are streamed in, pre-structured in batches that correspond to limited contiguous ranges in the row-index space of the edge matrix. This enables multiple accumulations to a destination vertex to be performed using cheaper shared-memory atomic operations instead of global-memory atomics, as typically used in other graph processing frameworks like Gunrock. Final accumulated result values are written out to destination vertex attributes in global-memory.

4.1. Phase-1

In phase-1, both edge and vertex data are streamed in from global memory in a coalesced manner, and partial contributions are accumulated in registers or shared memory. Different strategies are used for edge processing, depending on the amount of work per edge-block. This differential processing is motivated by the fact that different edge-blocks can differ greatly in the amount of work to be performed. If the number of edges in an edge-block is greater than 18k (an empirically determined threshold that was found to be best), then it is classified as *heavy*, otherwise it is classified as *light* in work.

As shown in Figure 2 the edge-matrix is split into two parts: a part containing heavy edge-blocks and another with only light blocks. This split is performed only once, in the pre-processing phase. All the light edge-blocks in a single block column (column-panel) are processed by a single thread block. In contrast, each heavy edge-block is processed by a thread block. The destination vertex attributes are kept in global memory and are set to the identity of the associative accumulation operator (zero for addition).

Heavy edge-block: The representation used for heavy edge-blocks is depicted in figure 3. The conceptual view of a single heavy edge-block is represented on the left. The data representation is comprised of two matrices: i) a frontier matrix, holding source/destination vertex IDs (middle); and ii) a matrix with edge weights. The dimensionality and total number elements in both the matrices are the same. The total number of elements is computed as the sum of the number of threads in a thread block, number of edges and number of destinations which have at least one contribution from the current edge-block. The number of columns is equal to number of threads in a thread block. The number of rows is the ceiling of the number of elements divided by the number of threads (ceil(#elements/#threads)). The edge-block frontier is filled in column major order. The first element is the negated value of destination id which receives a contribution from the current edge-block. This is followed by placing all the edges which are connected to the latter destination node. Then the negated value of next destination node is placed and the corresponding incoming edges and so on. While filling the table, if the rows limit is reached, the destination id is placed at the beginning of next column, which is followed by edges. For example, destination node 106 in Figure 3 has 4 edges. After placing the first source vertex, the row limit is reached. Hence the destination id is repeated again in the beginning on next column. In this matrix, the negated values represent the destination IDs and positive values represent source IDs. Corresponding to each source vertex in the edge-block frontier matrix, the edge-block is populated with edge weights. All other entries in the edge-block frontier matrix are marked as invalid. This representation is created during pre-processing.

Pseudocode for processing of heavy edge-blocks is shown in Algorithm 1. All the threads in a thread block collectively bring a portion of the input frontier corresponding to an edge-block to shared memory. All vertices in the front may not be active. The information regarding whether a frontier vertex is active or not is also bought to shared memory.

The work is uniformly distributed across threads. Each thread processes a single column in the edge-block frontier matrix using the corresponding element in the edge-block

```
kernel MultiGraphD__heavy_edge_block()
    edge_col = edge_block_list[tb_id].col
    edge_blk_id = tb_id
    // Bring input frontier to shared memory
    for (i = edge_col + t_id to edge_col +
    EDGE_BLK_SIZE − 1 step tb.size()) do
        sm_frontier_val[i-edge_col] = frontier_val[i]
        sm_frontier_active[i-edge_col] = frontier_active[i]
        sm_dest_value[i -edge_col] = init()
    end
    __syncthreads
    start = start_edge_position[edge_blk_id]
    end = start_edge_position[edge_blk_id + 1]
    num_rows = floor((end-start + tb.size() -1 - t_id)/tb.size())
    dest_id = abs(edge_block[edge_blk_id][0][t_id] + 1)
    reg_val = init()
    for i = 1 to num_rows − 1 do
        cur_val = edge_block[edge_blk_id][i][t_id]
        if cur_val <0 then
            if reg_val != init() then
                // Update dest val & actv flag
                atomic_update(&sm_dest_value[dest_id],
                reg_value)
                reg_val = init()
            end
            dest_id = abs(cur_val + 1)
        end
        else if sm_frontier_active[cur_value] == True then
            // Update register value
            reg_val = comp_contrib(sm_frontier_val[cur_val] ,
            edge_block_wt[edge_blk_id][i][t_id], reg_val)
        end
    end
    if reg_val != init() then
        atomic_update(&sm_dest_value[dest_id], reg_value)
    end
    stream_buf_base = loc_dest_buffer[tb_id]
    for (i = t_id to EDGE_BLK_SIZE − 1 step tb.size()) do
        stream_buf_base[i] = sm_dest_value[i]
    end
```

Algorithm 1: Algorithm for processing heavy edge-blocks in dense frontiers

weight matrix. The partial contributions are accumulated to a thread local register as long as the destination vertex does not change. If the destination vertex changes, as indicated by a negative entry in the column, the accumulated partial contribution is written to shared memory using an atomic operation. The use of shared-memory atomics is necessary since two threads could concurrently attempt to update the same destination value (for example, with destination 106 in Fig. 3). After a heavy edge-block is processed, accumulated contributions to all destination vertices of the edge-block are written out.

The data representation used for the heavy edge-blocks combines both the coalesced data access benefit from the transposed data representation, as used by Liu and Vinter with CSR5 [12], as well as the load balancing across threads

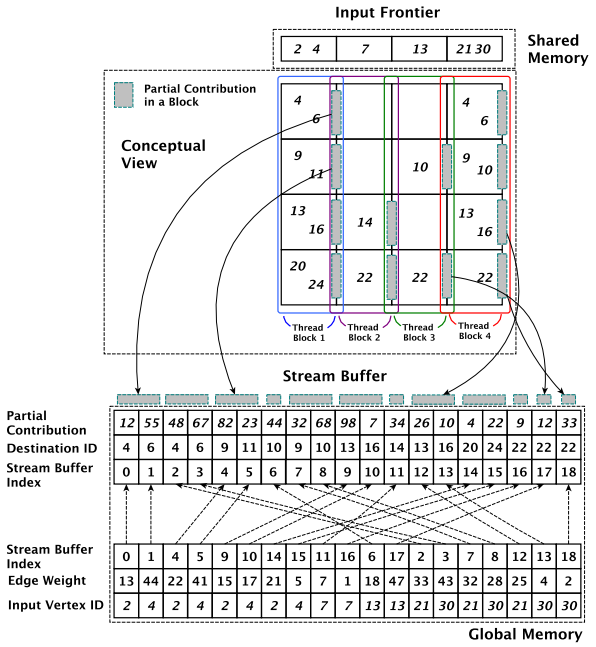


Figure 4: MultiGraphD: Processing light edge-column-panel

achieved with the merge-bases CSR scheme of Merrill and Garland [15].

Each thread has to efficiently find a location in global memory to write its partial contribution. During pre-processing, the maximum space required to save the partial results (by assuming that the output frontier is dense) is computed. The space is allocated in such a way that all the edge-blocks in an edge-row-panel occupy contiguous memory locations.

The thread block size was chosen to be 1024, and edge block size was chosen as 6*1024 to fully utilize shared memory, as well as achieve maximum occupancy, and minimize the number of edge-blocks.

Light edge-blocks: The representation used for light edge-blocks is shown in Figure 4. The top matrix shows the conceptual view of the edge-matrix. Unlike heavy edge-blocks, all light edge-blocks in an edge-column-panel are processed by a single thread block. The edges are streamed in from global memory. This is represented by the bottom matrix. The last row contains the input frontier vertex id and the corresponding edge-weights are shown in the second row. The first row contains an index/pointer to a stream-buffer location where the partial contributions is be written; the actual location is represented by the first row of middle matrix. The second row of middle matrix contains the ID of the destination node.

Algorithm 2 describes the processing of the light column-panels. Each column-panel is processed by a single thread block. All the threads collectively bring the input frontier values to shared memory. The entire work is then divided cyclically across threads. Each thread first loads a vertex and if it is active it directly writes the partial product to global

```
kernel MultiGraphD__light_edge_block()
    edge_col = tb_id * edge_block.size()
    // Bring input frontier to shared memory
    for (i = edge_col + t_id to edge_col +
    EDGE_BLK_SIZE − 1 step tb.size()) do
        sm_frontier_val[i-edge_col] = frontier_val[i]
        sm_frontier_active[i-edge_col] = frontier_active[i]
    end
    __syncthreads
    start = start_edge_position[tb_id]
    end = start_edge_position[tb_id+1]
    // Stream out contributions to global memory
    for i = start + t_id to end − 1 step tb.size() do
        if sm_frontier_active[input_vertex_id[i]] == True then
            pos_to_write = stream_buf_index[i]
            stream_buf[pos_to_write] =
            compute_contrib(sm_frontier_val[
            input_vertex_id[i]], edge_weight[i])
        end
    end
```

Algorithm 2: Algorithm for processing light edge-blocks in dense frontiers

memory. Since the memory is pre-allocated, atomic operations are not required. However, unlike heavy edge-blocks, contributions from the light edge-blocks are not locally compressed.

This work division strategy between heavy and light edge-blocks helps in achieving good performance with a small number of atomic operations. Since the number of partial contributions from a heavy edge-block is high, assigning an entire thread block to it helps in efficient processing. In contrast, since the number of partial contributions from a light edge-block is generally low, assigning a thread block to an entire edge-column-panel helps in attaining higher performance.

4.2. Phase-2

A single destination vertex can have contributions from multiple edge-blocks, i.e., there could be contributions to the same destination vertex from different heavy edge-blocks and different light edge-blocks. Each such partial product should be combined to produce the final result. Phase-2 combines the partial results produced in phase-1 to form the final output.

In phase 2, each edge-row-panel is processed by a thread block. Note that the global memory allocated for storing the partial products from heavy edge-blocks and light edge-blocks are contiguous in memory. Each thread block loads the corresponding output vertices to shared memory, which helps in reducing the total number of global transactions. Then it loads the corresponding global memory blocks containing the partial results, cyclically distributing work among threads to achieve coalesced global-memory access. Updates to output vertices are performed using shared-memory atomic operations.

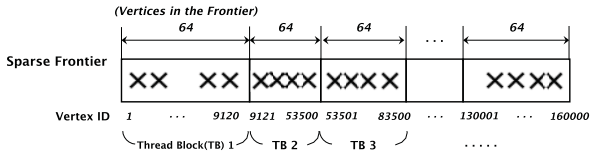


Figure 5: MultiGraphS: Work distribution

5. MultiGraphS (Sparse Frontier)

When the input vertex frontier represents only a small fraction of the graph's vertices, the data representation and processing approach of MultiGraphD is very inefficient, since essentially every edge must be traversed. For such a scenario, we use a completely different graph representation and edge processing approach. The subsystem for processing sparse frontiers, MultiGraphS, is described in this section. A major challenge to be addressed is that of balanced work distribution. Intra-warp, inter-warp and inter thread-block load balance is a key factor affecting performance.

MultiGraphS partitions the vertices of the input frontier into groups 64 active vertices. Each such group is processed by a single thread block of 256 threads. This work distribution is shown in Figure 5.

The out-degrees of frontier vertices can differ very significantly. If a warp is assigned to process a frontier node, there could be severe load imbalance as some warps may have very little work while others may have a lot of work. Further, for vertices with out-degree much less than 32, many threads within a warp will be inactive and cause idleness of hardware resources in the GPU's SIMD functional units. Since the out-degree of each active vertex is known, it is of course feasible to form a prefix sum of Below, we describe the MultiGraphS approach to load balancing.

The processing of outgoing edges of active frontier vertices is performed in three stages:

- **Stage-1** involves processing of a small number of edges from each vertex, so that the remaining unprocessed edge-count for each vertex is a perfect multiple of 32. If $vertex_k$ has E_k outgoing edges, the last E_k % 32 edges are selected for processing in this stage. For this stage, the thread block of 256 threads is partitioned into 64 virtual warps of 4 threads each, and virtual warp k handles edges from vertex k in the current group.
- **Stage-2** involves processing more outgoing edges (if any remain) from each vertex, so that the remaining unprocessed edge-count of each vertex is a perfect multiple of 256. For $vertex_k$ with E_k initial edges, of which E_k % 32 got processed in stage-1, (E_k - E_k % 32) % 256 will be selected for processing in stage-2. For this stage, the thread block of 256 threads is organized as 8 warps, with each warp processing all stage-2 edges from 8 vertices. For this stage, no intra-warp load imbalance or thread divergence can occur, with the only negative effect of inter-warp load imbalance being loss of effective warp

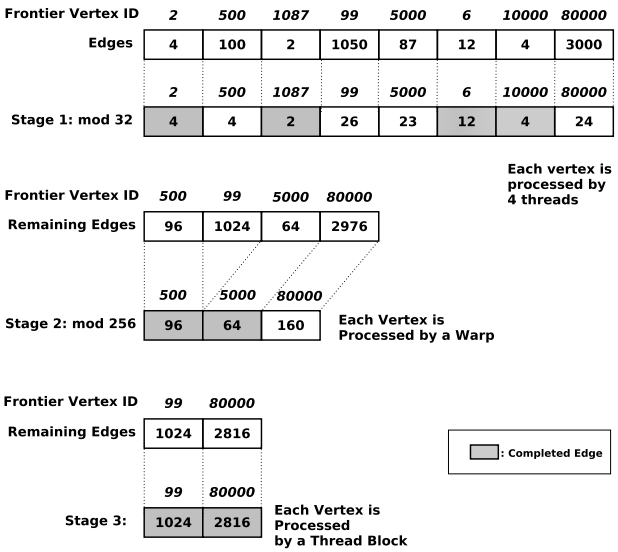


Figure 6: MultiGraphS: Example illustrating 3-stage work distribution

occupancy since some active warps may be idle.

- **Stage-3** processes all remaining edges for all vertices. This is done sequentially across frontier vertices that still have edges to be processed. Since that count is a multiple of 256, perfect load-balancing across all warps of the thread-block is achieved.

Fig. 6 shows an example illustrating the three stages of processing. Frontier vertex-2 has 4 out-going edges and all its edges are selected for processing in stage-1 ($4 \% 32 == 4$). Frontier vertex-500 has 100 out-going edges, from which 4 edges are selected for processing in stage-1 ($100 \% 32 == 4$). In stage-1, each vertex is processed by four threads.

The edges that were not processed in stage-1 are passed to stage-2. In Figure 6, for frontier vertex-500, all the 96 remaining edges are selected for processing in stage-2 ($96 \% 256 == 96$). Similarly, for frontier vertex-80000, 160 edges ($2976 \% 256 == 160$) are selected for processing in stage-2. Each vertex (corresponding to the selected edges) is processed by the threads of a warp using a cyclic work distribution. The remaining set of edges after stage-2 are processed in stage 3 where an entire thread-block is assigned to each vertex.

The active frontier is loaded to shared memory and its computation is efficiently done using warp aggregation [6]. After processing edges in stage-1, vertices that have fewer than 32 outgoing edges will be fully processed. Hence, they should be removed from the frontier. This is done using warp aggregation. Similarly, after stage-2, all vertices which have fewer than 256 outgoing edges should be removed.

Updates from each edge are accumulated in global memory using atomic operations. An output node with 'n' incoming edges will require exactly 'n' atomic operations on the same memory location. This implies that nodes with heavy in-edge degree will have limited parallelism, since the atomic operations on the same memory locations will have a serializing effect. This will have a lesser impact on performance if the atomic operations were done in shared memory instead of global memory. One way to resolve this is to bring a set of output edges in shared memory, process them (using atomics in shared memory) and then stream out the results to global memory. However, using a lot of shared memory may degrade performance as warp occupancy will be affected. In addition, output nodes with few incoming edges will have minimal benefits when using shared memory for atomics. Hence we use shared memory only for output nodes with high in-degree. The grouping information computed during pre-processing (mentioned in Section 3) is used for this classification. All the output nodes belonging to group-1 to group-4 are classified as 'high in-degree' nodes and the rest are classified as 'low in-degree' nodes. The high degree nodes are partitioned into segments.

Each such segment is further sub-divided into sub segments such that each sub segment contains 1024 active input frontier vertices. This helps in achieving good inter-block load balance. The segmentation strategy is depicted in Figure 7. Each heavy sub-segment is processed by a single thread block. The thread block loads the output vertices to shared memory, accumulates the partial contributions in shared memory using atomics and writes out the result to global memory.

The entire set of edges for low in-degree nodes are treated as single segment. Each such segment is then sub-divided into sub-segments with 64 active input frontier vertices. Each sub-segment is processed by a thread block. For efficient access, each segment is represented in CSC format. The CSC representation for each segment is created during pre-processing.

For group 1-4, the largest possible thread block size (1024) was selected and frontier size was also set to 1024 to maximize available shared memory size per thread block and minimize the number of segments. The key idea behind load balancing of MultiGraphS comes from Merrill et al [16]. The same thread block size of 256 was chosen as it was empirically verified to outperform other thread block sizes. The frontier size per thread block was chosen to be 64 because each vertex is processed by 4 threads in stage-1 (256/4 = 64).

6. MultiGraph (Hybrid)

In previous sections, the data structures and algorithms for dense-frontier (MultiGraphD) and sparse-frontier (MultiGraphS) processing were presented When processing graphs, the sizes of frontiers do not remain constant across iterations. The profile of variation in size of frontier depends on the algorithm/benchmark and the dataset. Consider Table 2, which shows how the frontier size varies over the iterations of BFS for dataset 'soc-orkut', as well as the execution time taken by MultiGraphD and MultiGraphS. The frontier size initially grows and then decreases. For small frontier sizes MultiGraphS is much faster, while MultiGraphD can be up to 3 times as fast for the densest front (at iteration 6). Most datasets

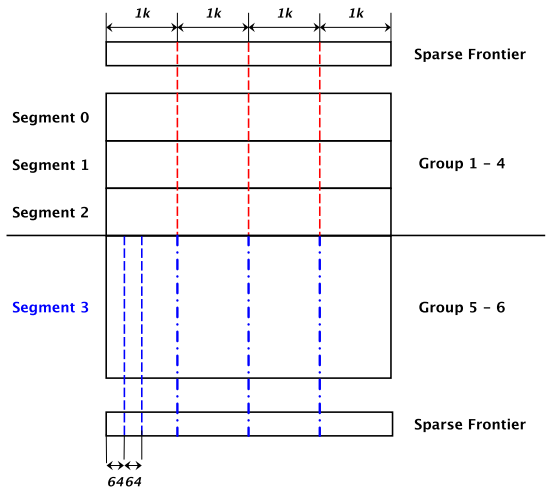


Figure 7: MultiGraphS: Edge block segmentation

Table 2: BFS frontier size variation: soc-orkut

Iter	Frontier size # vertices	# edges	Sparse time	Dense time	Best possible time
1	1	12	0.07	21.81	0.07
2	12	580	0.05	21.81	0.05
3	460	38590	0.13	21.85	0.13
4	25400	3294619	1.40	21.95	1.40
5	728870	73887541	33.93	22.48	22.48
6	2210215	135105894	66.53	22.22	22.22
7	32216	371130	0.26	21.84	0.26
8	32	52	0.07	21.82	0.07
	total time		102.45	175.78	46.69

exhibit a similar trend, with neither MultiGraphD nor MultiGraphS being consistently better across all iterations of an algorithm like BFS. This motivates the need for a hybrid algorithm that judiciously selects between the MultiGraphD and MultiGraphS at the beginning of each iteration, based on the size of the frontier. If such a choice could be achieved with perfect precision and zero overhead, the achievable completion time is shown in the last column of Table 2 to be 46.69 ms, compared to 102.45 ms for MultiGraphS and 175.78 ms for MultiGraphD.

The major challenges associated with designing a hybrid algorithm are: i) Performance modeling - for a given benchmark-dataset pair and frontier size, estimating the relative performance of MultiGraphS and MultiGraphD; ii) Efficient mode-switching - if predicted performance is better for a different mode than the current one, efficient switching to the other mode. One approach to estimate performance is to build a linear regression model against the number of active vertices and edges in the current and next frontier. This was attempted, and although it worked reasonably well for some cases, it was not consistently effective across datasets and graph algorithms. Therefore, a dynamic sampling-based approach was developed for a hybrid implementation, called MultiGraph.

```
void MultiGraph_Hybrid()
    EDP = MultiGraphD_sample()
    mode = init_processing_mode()
    if mode == sparse then
        frontier = init_frontier()
    end
    else
        frontier_active = init_frontier_active()
    end
    while not_converged do
        if mode == dense then
            // sample_frontier is just a small
            part of frontier
            sample_frontier =
            generate_sample_frontier(frontier_active)
        end
        else
            sample_frontier = select_part(frontier)
        end
        ESP = MultiGraphS_sample(sample_frontier)
        if EDP ≥ ESP then
            if mode == sparse then
                frontier_active =
                generate_frontier_active(frontier)
            end
            mode = dense
            frontier_active = MultiGraphD()
        end
        else
            if mode == dense then
                frontier =
                generate_frontier(frontier_active)
            end
            mode = sparse
            frontier = MultiGraphS()
        end
    end
```

Algorithm 3: Algorithm for MultiGraph_Hybrid

Pseudocode for MultiGraph (the hybrid algorithm) is shown in Algorithm 3. At the beginning of each iteration, a random fraction of the current frontier is sampled and the corresponding subgraph processed using MultiGraphS. The performance of MultiGraphD is also estimated using a small sample of vertices. However, unlike MultiGraphS, for a given dataset and algorithm, the performance of MultiGraphD does not vary much as a function of the number of active vertices in the frontier. This can be observed in Table 2. Hence, MultiGraphD execution on a sample is only done once, before the first iteration.

In each iteration, MultiGraphS is executed on a small sampled fraction of the frontier. If the current mode is dense (i.e. MultiGraphD), then a small subset of the frontier is generated with the flag "frontier_active". Warp aggregation is

Table 3: Dataset description

Dataset						
Type	**Name**	**|V|**	**|E|**	**Max Degree**	**Dia.**	**Type**
Scale-free	rmat_s24_e16	16.8M	520.0M	434813	6	s
	rmat_s23_e32	8.4M	506.4M	438471	6	s
	rmat_s22_e64	4.2M	484.0M	428234	6	s
	indochina-04	7.4M	302.0M	256425	26	r
	soc-orkut	3M	212.7M	27466	9	r
	kron_g500-logn21	2.1M	182.1M	213904	6	s
	hollywood-09	1.1M	112.8M	11467	11	r
	soc-Livejournal1	4.8M	85.7M	20333	16	r
Mesh-like	rgg_n_24	16.8M	265.1M	40	2622	r
	road_USA	23.9M	57.7M	40	6809	s
	roadnet-CA	2M	5.5M	12	849	r

s: synthetic; r: real-world

Table 4: Machine configuration

Resource	Details
Host CPU	Intel core i7-2600 (4 cores, 3.40 GHz, 8MB L3cache), 16GB DDR3-1333)
GPU	Tesla K40c (15 Kepler SMs, 192 cores/MP, 12 GB Global Memory, 745 MHz, 1.5MB L2 cache, ECC off)

used to accumulate sample_frontier. The overhead for generating sample_frontier is low since only a small fraction of the whole frontier is used. If the current mode is sparse (i.e. MultiGraphS), then a sample can be directly obtained from the whole frontier.

When switching from MultiGraphD to MultiGraphS, the sparse frontier has to be computed. This is done using warp aggregation [6]. Each warp scans the frontier_active flag and collects the list of active vertices. The warp leader then uses a single atomic operation to allocate space for storing the sparse front. Each thread then writes the active vertex (if any) to the allocated space. Switching from MultiGraphS to MultiGraphD is done by first resetting the frontier_active flag, and then marking the flags corresponding to the sparse front as active.

7. Experimental Evaluation

In this section, experimental results are presented, comparing MultiGraph with other GPU graph processing frameworks: CuSha, Groute, Gunrock, and Warp-Segmentation (WS). All experiments were performed on an Nvidia Tesla K40c GPU with 12GB memory, hosted by an Intel Xeon system. Table 4 provides details about the system. ECC was turned off for the tests[2]. For all evaluations, Nvidia's nvcc compiler (version 7.5) was used, with the -O3 flag. We report performance for MultiGraph and other frameworks with all data already in GPU global memory. In reporting achieved performance (throughput), we do not include time for input of graphs from file, any pre-processing costs for the graphs, or the time for data transfer from CPU to GPU. Later in this section, we separately report pre-processing overhead for each test dataset.

[2]Experiments were also conducted with ECC turned on. There was very little difference in performance, well under 5%.

All tests were run 10 times and the average was used for the reported results. The Gunrock team actively maintains a web-site (http://gunrock.github.io) with performance data on a range of graph datasets for various graph processing frameworks. The dataset includes 8 scale-free datasets and 3 mesh-like datasets We used the same datasets, downloaded from their web-site (except the synthetic datasets RMAT22, RMAT23, and RMAT24, which were generated using the RMAT generator [10]). Characteristics of the datasets are shown in Table 3.

7.1. Performance comparison with other graph frameworks

The performance achieved with MultiGraph was compared with several available GPU graph processing frameworks, including both topology-driven frameworks (CuSha, WS (Warp Segmentation)) and data-driven frameworks (Groute, and Gunrock version 0.4). The following standard graph algorithms were evaluated:

Breadth-First Search (BFS): For each iteration, the BFS level of all vertices connected to a vertex in the current frontier are updated. We note that Gunrock provides a "DO(Direction-Optimized)-BFS" which is very powerful for scale-free graphs, but we only implemented standard BFS. In order to enable comparison of the various frameworks on the same algorithm, we present results for Gunrock with DOBFS disabled as well as with DOBFS enabled.

Single-Source Shortest Path (SSSP): For each iteration, relaxation is performed for all vertices connected to any vertex in the active frontier. We did not use any other optimization techniques like priority queues.

Connected Components (CC): We implemented Gunrock's CC algorithm [1].

Page Rank (PR): We developed two variants: i) topology-driven PR (PR-topology) - CuSha's topology-driven page-rank algorithm, and ii) data-driven PR (PR-data) - Gunrock's data-driven page-rank algorithm.

Betweenness Centrality (BC): We implemented Gunrock's BC algorithm [1].

In order to optimize performance using Groute for BFS and SSSP, we auto-tuned two parameters: "prio_delta_fused" and "prio_delta_softprio". For Gunrock, we used tuned parameters used by the authors and reported in their publications.

We report performance in pseudo MTEPS (Millions of Traversed Edges Per Second), where the MTEPS metric is derived as follows: For BFS, SSSP, PR, and CC, MTEPS is computed as $|E|$/t where $|E|$ is the number of edges in the graph and t the execution time in $\mu secs$. For BC, MTEPS is defined as $2|E|$/t since each edge is visited in the forward pass and again in the backward pass. Since the execution times vary significantly across the graphs, this normalized measure is more convenient. Even if different frameworks actually process different number of edges (an edge may be relaxed multiple times, and the number of times could differ from one implementation to another),

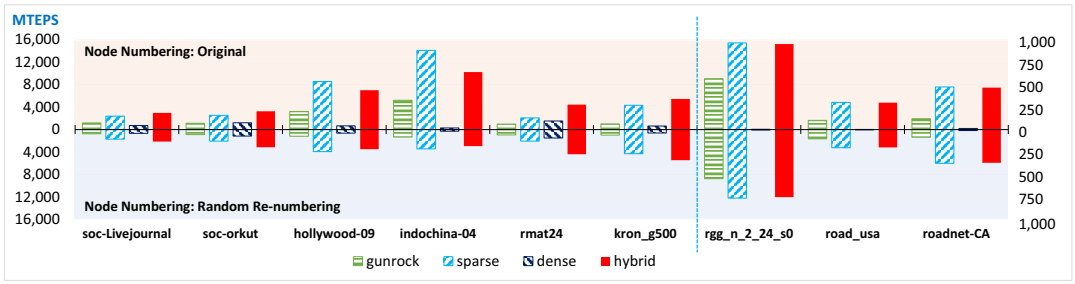


Figure 8: Performance: Betweenness Centrality

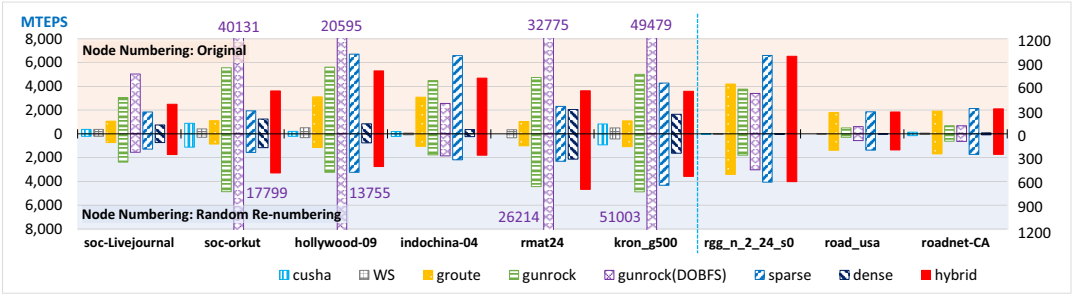


Figure 9: Performance: Breadth First Search

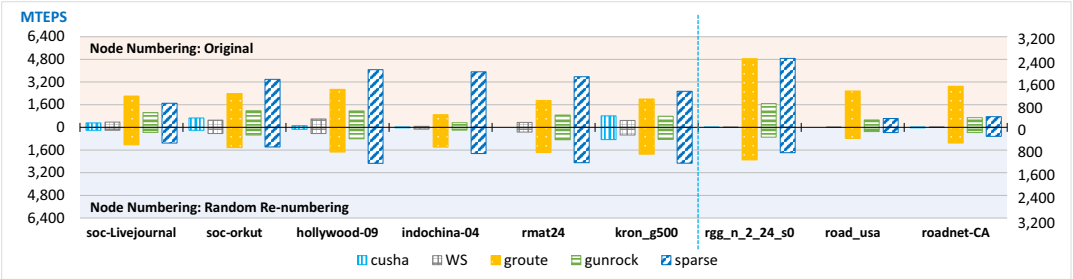


Figure 10: Performance: Connected Components

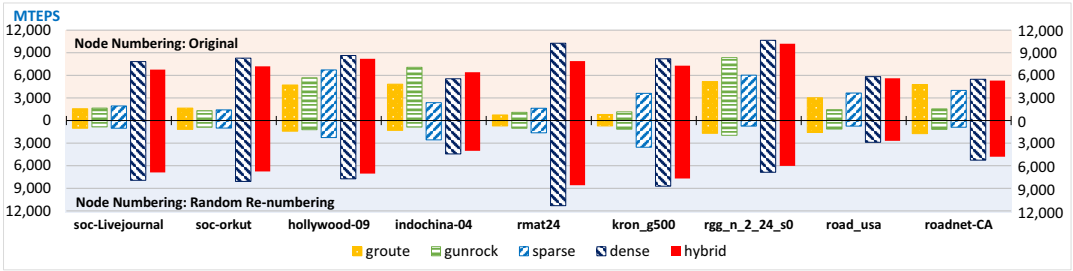


Figure 11: Performance: Page Rank (Data-Driven)

the MTEPS measure is inversely proportional to t and allows a fair comparison of the ratios of completion times of different frameworks. The data structure for CuSha requires more space than the other frameworks. On the test GPU, there was insufficient global memory to hold RMAT22,RMAT23,RMAT24, and roadnet_USA Hence performance data is not available for these datasets with CuSha.

For all benchmarks, the relative performance trends for RMAT22 and RMAT23 were found to be very similar to that of RMAT24, with RMAT23 having slightly higher performance, and RMAT22 a bit higher still. So we only display performance for RMAT24. For all benchmarks, we also tested the different frameworks on performance sensitivity to a random renumbering of graph vertices. We observed that many datasets use contiguous numbering for many neighbor vertices. This is very favorable for a CSR representation, where access to needed vertex values to process a set of edges achieves high degree of coalescing. In the bar charts, the performance with

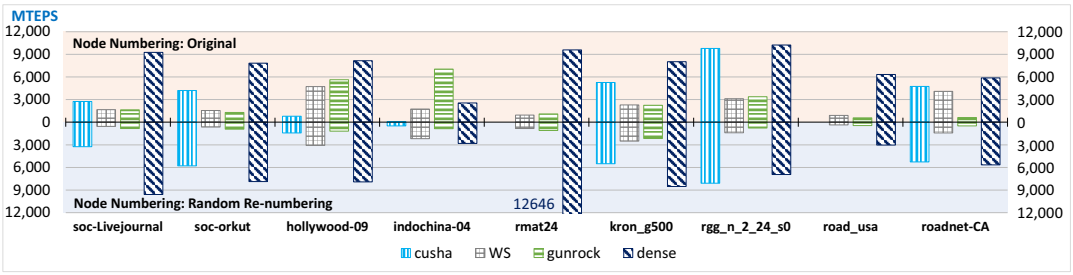


Figure 12: Performance: Page Rank (Topology-Driven)

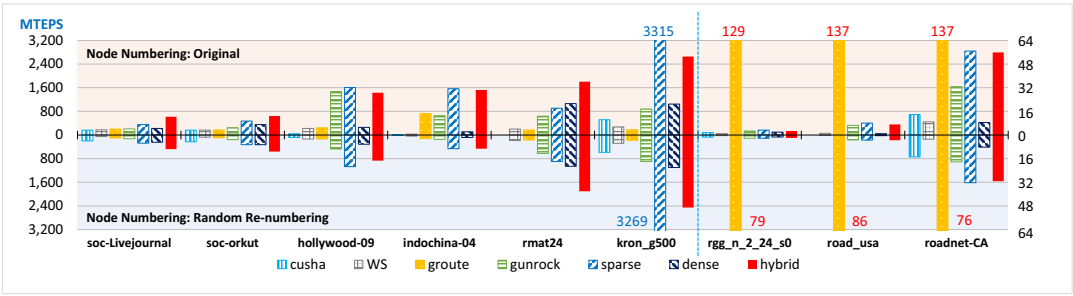


Figure 13: Performance: Single Source Shortest Path

the given dataset order faces upwards, while the corresponding data with randomly renumbered vertices faces downwards. A general conclusion is that except CuSha and MultiGraphD, other frameworks are quite sensitive to vertex numbering, often suffering around 30% loss of performance.

Breadth-First Search (BFS): Fig. 9 presents performance data in MTEPS for BFS. CuSha and WS use topology-driven execution. For each iteration, all edges and vertices are traversed. Groute and Gunrock use data-driven BFS. For scale-free graphs, 6-26 iterations are required to find all reachable vertices with a bulk synchronous model for the datasets. In contrast, for mesh-like graphs, 555-6626 iterations are needed for BFS. CuSha and WS achieve very low performance due to the data-driven execution. Performance of MultiGraph is generally on par with or better than the other frameworks, except for Gunrock, which is faster than MultiGraph on four of the nine datasets. With DOBFS enabled, Gunrock is up to an order of magnitude faster on several datasets. Also, for scale-free graphs, Gunrock performs better than Groute. On the other hand, for mesh-like graphs, Groute performs better than Gunrock.

For scale-free graphs, the hybrid algorithm is better than either MultiGraphD or MultiGraphS. For mesh-like graphs, the dense-frontier algorithm MultiGraphD is never activated. For each iteration, the sparse algorithm processes the dataset. When we use the hybrid algorithm, a small mode selection overhead is incurred, so that MultiGraphS performs a little better than the hybrid algorithm.

Single-Source Shortest Path (SSSP): Performance data is shown in Fig. 13. The difference between BFS and SSSP performance is because the size of the frontier for SSSP is usually much larger. Several iterations have extremely large frontier size and these iterations dominates the execution time. For these iterations, MultiGraphD significantly outperforms Gunrock, thereby resulting in high speedup relative to BFS. Groute is an asynchronous algorithm, but it may suffer from the limitations described earlier. For mesh-like graphs, the dense algorithm is never activated. MultiGraph performance is better than Gunrock in this case, but the performance of Groute is significantly better than MultiGraph and others. Groute uses a soft priority scheduler and fused workers. These can significantly reduce the amount of redundant (useless) work performed in other frameworks.

Connected Components (CC): For scale-free graphs, MultiGraph performance is consistently better than other frameworks (Fig. 10). For road_USA and roadnet-CA, Groute is much better than other frameworks. The reason is that Groute uses a different connected components algorithm, which is well matched to the asynchronous GPU framework.

Page Rank (PR: topology-driven and data-driven): Performance data is shown in Fig. 11. The topology-driven PR algorithm and data-driven PR algorithm used in Gunrock update values of all vertices until convergence. For MultiGraph, the dense algorithm is always chosen, and MultiGraphD generally outperforms frameworks that are mainly targeted towards data-driven algorithms. CuSha uses a novel structure to process graphs and is also geared towards topology-driven algorithms and performs better than MultiGraph in the case of rgg_n_2_24_s0.

Betweenness Centrality (BC): Performance data is shown in Fig. 8. Only Gunrock had an implementation of BC, and so we compared MultiGraph only with Gunrock. BC has two

Table 5: Speedup: Original node numbering

Benchmark	CuSha	WS	Gunrock	Groute
BFS	16.02	27.30	1.20	1.93
SSSP	7.77	7.04	1.86	1.28
CC	29.81	24.82	2.78	1.01
PR(top)	3.71	3.79	4.14	-
PR(dat)	-	-	4.07	3.95
BC	-	-	2.90	-
average	10.83	11.60	2.59	1.77

Table 6: Speedup: Random re-numbering

Benchmark	CuSha	WS	Gunrock	Groute
BFS	10.96	33.51	1.27	2.05
SSSP	4.05	8.75	2.17	1.53
CC	18.95	22.07	2.81	0.95
PR(top)	2.33	6.13	8.01	-
PR(dat)	-	-	6.96	6.36
BC	-	-	2.97	-
average	6.65	14.11	3.30	2.09

stages: forward processing and backward processing. BFS is used for forward processing. The difference between BFS and BC is that in BC, every edge is visited twice (forward processing and backward processing). Hence, Gunrock cannot take advantage of edge skipping for BC as in BFS. For scale-free datasets, the performance is mainly dependent on a few iterations with very large frontier. Similar to SSSP, for these iterations, MultiGraph chooses the dense algorithm, which outperforms Gunrock. We note that MultiGraphS also achieves consistently higher performance than Gunrock. The reasons are efficient work-list management and use of a combination of shared and global memory atomics, instead of all global memory atomics. For mesh-like graphs, as with previous algorithms, MultiGraph always chooses the sparse algorithm, and achieves higher performance than Gunrock.

Tables 5 and 6 present geometric means of the speedup achieved by MultiGraph over other frameworks, for original node numbering and random renumbering, respectively.

Tables 7 and 8 present data on the pre-processing overheads for MultiGraph – all previously presented performance data excludes any pre-processing time for MultiGraph and any of the other frameworks that require any transformation from a standard CSR representation. Table 7 presents measured pre-processing time for each benchmark algorithm and each graph dataset, both for the original numbering and the randomized renumbering of vertices for the datasets. The measured time accounts for the conversion of the graph data from a standard COO format to the data-structures required for MultiGraphD and MultiGraphS. This data conversion is performed in the GPU, and the reported time does not include the time for reading in data from files to the host CPU and transfer of the data to the GPU global memory. In Table 8, aggregated normalized data is presented as a relative fraction of the actual execution time for each of the tested algorithms (geometric mean across the graph datasets). Optimization of pre-processing overheads has not been a focus so far, and it is expected that significant reduction of this overhead is feasible.

7.2. Hardware performance metrics

In this section, we present some metrics from hardware counter measurements that validate design decisions for MultiGraph. Due to space limitations, we only show data for one of the evaluated graph algorithms: Page Rank. Trends for other algorithms is also broadly similar. Figure 14(a) shows memory storage requirements for the various datasets, normalized to the storage requirements for the standard CSR representation used by Gunrock, Groute, and WS. The total memory requirement is computed as the sum of the memory required for heavy edge-block representation, light edge-block representation in MultiGraphD and the sparse representation in MultiGraphS. The space overhead for MultiGraph over CSR ranges from 1.5x to 2.5x.

Table 7: Preprocessing time: original/renumbered (in ms)

Dataset	BC	BFS/PR	CC	SSSP
soc-Livejournal1	130(200)	128(198)	6.7(6.7)	189(280)
soc-orkut	311(425)	312(419)	13.5(12.7)	473(598)
hollywood-09	191(270)	242(369)	7(6.3)	332(469)
indochina-04	391(557)	453(595)	19.5(16.5)	681(827)
rmat24	1009(1049)	1092(1122)	33.7(26.8)	1655(1681)
kron_g500	675(668)	1077(1065)	9.8(9.9)	1249(1235)
rgg_n_2_24_s0	188(633)	133(588)	15.8(16)	149(828)
road_USA	105(177)	87(144)	8.1(8.1)	111(166)
roadnet-CA	13.2(12.8)	11.8(11.3)	0.7(0.7)	12.5(15.3)

Figure 14(b) presents the global memory efficiency for each framework, computed as follows. For each GPU kernel, the number of global memory load/store transactions and global load/store efficiency were collected using Nvprof. The sum of product of global memory load/store transactions and the corresponding global load/store efficacy was divided by the total number of load/store transactions to obtain an average global memory efficiency for the execution. It can be seen that MultiGraph and CuSha attain consistently high global memory load/store efficiencies of around 80%. WS and Groute exhibit the lowest global memory efficiency, well under 50% for all the datasets. Gunrock achieves efficiencies between 60-70%. The main reason for the higher global memory efficiency for CuSha and MultiGraph is the lower incidence of uncoalesced data accesses, due to the alternate data-structures and edge

Table 8: Normalized pre-processing overhead

Algorithm	Original	Renumbered
BC	1.97	1.97
BFS	3.19	3.40
CC	0.13	0.07
PR(data-driven)	0.55	0.66
PR(topology-driven)	0.15	0.21
SSSP	0.57	0.57

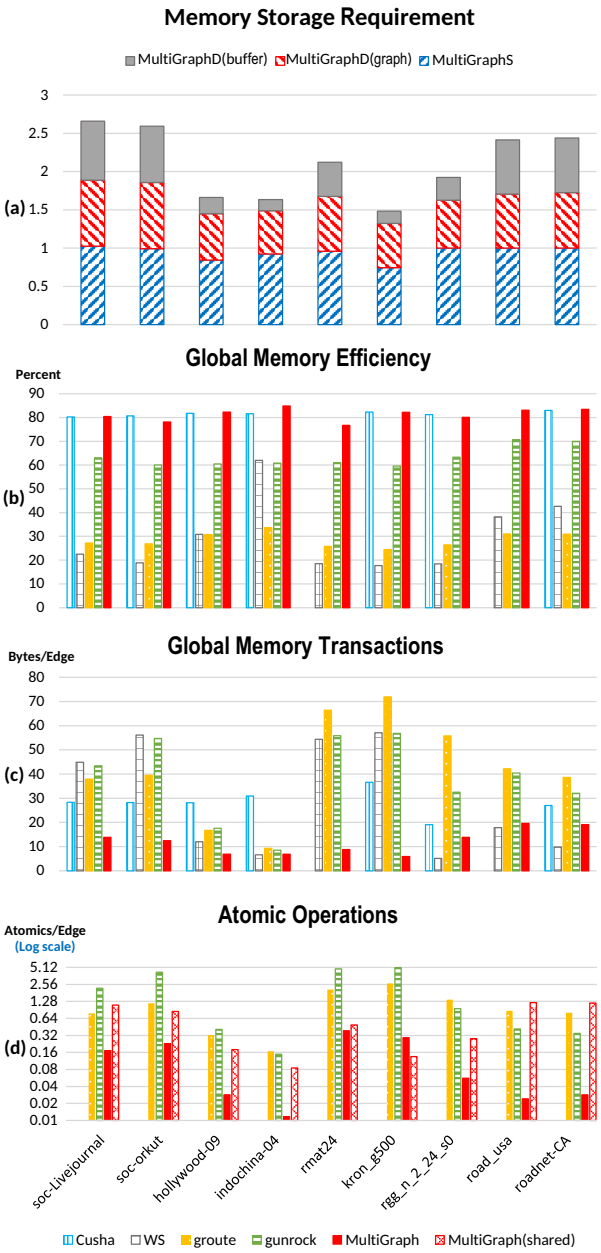


Figure 14: Comparison of memory usage and hardware metrics across systems (PR algorithm)

processing strategies used.

Figure 14(c) displays statistics on measured global memory transactions for the different frameworks, across the different datasets. The metric is presented normalized to the total number of graph edges traversed (graph edges multiplied by number of iterations). It was computed as the sum of DRAM loads/stores (obtained from Nvprof) multiplied by 32 (conversion to bytes) and divided by the product of total number of edges and number of PR iterations. This metric varies across a range from around 8 bytes/edge to 72 bytes/edge across the frameworks and datasets. MultiGraph incurs data movement ranging from 8-20 bytes/edge, while Gunrock incurs consistently higher data movement, ranging from around 10-55 bytes/edge. Groute also requires higher data movement than MultiGraph, ranging from 10-70 bytes/edge. CuSha requires slightly higher global memory data movement than MultiGraph, and has relatively low variance across datasets, ranging from 20-35 bytes/edge. In contrast, WS has the highest variance in data movement volume across datasets, with volumes in some cases being as low as 5 bytes/edge (lower than MultiGraph) and as high as 55 bytes/edge.

Figure 14(d) presents data on the number of atomic operations, again normalized to the number of processed graph edges. The number of global atomic operations were directly obtained using Nvprof. A significant reason for the improved performance of MultiGraph over other frameworks is the reduction in the number of global-memory atomics by use of atomics in shared-memory instead. Since hardware counters are not available on the Nvidia Kepler for shared-memory atomics, it was estimated as follows. For MultiGraphS, it was computed as the sum of number of edges in group 1 to 4 for the active input vertices. The number of shared memory atomic operations for the MultiGraphD was computed as the sum of i) the number of active destination vertices in each heavy edge-block, ii) 1024 * number of heavy edge-blocks iii) sum of the number of active edges in each light edge-block. The total number of global atomic operations for MultiGraph (red solid bars) can be seen to be significantly lower than Gunrock and Groute (we note that the y-axis scale for this chart is logarithmic). For MultiGraph, global atomics have been replaced by shared-memory atomics (red bars with criss-cross pattern), which incur much lower overheads than global atomics.

8. Conclusion

This paper has addressed the development of a high-performance approach to graph processing on GPUs. It uses multiple data representation and execution strategies for dense versus sparse vertex frontiers, dependent on the fraction of active graph vertices. A two-phase edge processing approach trades off extra data movement for improved load balancing across GPU threads, by using a 2D blocked representation for edge data. Experimental results demonstrate performance improvement over current state-of-the-art GPU graph processing frameworks for many benchmark programs and datasets.

Acknowledgment

We are grateful to the reviewers for their feedback and suggestions that greatly helped improve the paper. We thank John Owens for his valuable feedback and numerous suggestions. This work was supported in part by the U.S. National Science Foundation (NSF) through awards 1513120 and 1629548.

References

[1] "Gunrock: High-performance graph primitives on GPUs," gunrock.github.io.

[2] T. Ben-Nun, M. Sutton, S. Pai, and K. Pingali, "Groute: An asynchronous multi-GPU programming model for irregular computations," in *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, 2017, pp. 235–248.

[3] M. Bernaschi, G. Carbone, and F. Vella, "Betweenness centrality on multi-GPU systems," in *Proceedings of the 5th Workshop on Irregular Applications: Architectures and Algorithms*. ACM, 2015, p. 12.

[4] U. Cheramangalath, R. Nasre, and Y. N. Srikant, "Falcon: A graph manipulation language for heterogeneous systems," *ACM Trans. Archit. Code Optim.*, vol. 12, no. 4, pp. 54:1–54:27, Dec. 2015. [Online]. Available: http://doi.acm.org/10.1145/2842618

[5] A. Davidson, S. Baxter, M. Garland, and J. D. Owens, "Work-efficient parallel gpu methods for single-source shortest paths," in *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*. IEEE, 2014, pp. 349–359.

[6] I. J. Egielski, J. Huang, and E. Z. Zhang, "Massive atomics for massive parallelism on GPUs," in *Proceedings of the 2014 International Symposium on Memory Management*. ACM, 2014, pp. 93–103.

[7] 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*. ACM, 2014, pp. 1–6.

[8] 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*. ACM, 2012, pp. 349–362.

[9] 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*. ACM, 2011, pp. 267–276.

[10] F. Khorasani, R. Gupta, and L. N. Bhuyan, "Scalable SIMD-efficient graph processing on GPUs," in *Proceedings of the 2015 International Conference on Parallel Architecture and Compilation (PACT)*. IEEE Computer Society, 2015, pp. 39–50.

[11] 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*. ACM, 2014, pp. 239–252.

[12] W. Liu and B. Vinter, "CSR5: an efficient storage format for cross-platform sparse matrix-vector multiplication," *CoRR*, vol. abs/1503.05032, 2015.

[13] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein, "GraphLab: A new framework for parallel machine learning," *CoRR*, vol. abs/1006.4990, 2010.

[14] 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*. ACM, 2010, pp. 135–146.

[15] D. Merrill and M. Garland, "Merge-based sparse matrix-vector multiplication (Spmv) using the CSR storage format," in *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, 2016, pp. 43:1–43:2.

[16] D. Merrill, M. Garland, and A. Grimshaw, "Scalable GPU graph traversal," in *ACM SIGPLAN Notices*, vol. 47, no. 8. ACM, 2012, pp. 117–128.

[17] S. Pai and K. Pingali, "A compiler for throughput optimization of graph algorithms on GPUs," in *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM, 2016, pp. 1–19.

[18] K. Pingali, D. Nguyen, M. Kulkarni, M. Burtscher, M. A. Hassaan, R. Kaleem, T.-H. Lee, A. Lenharth, R. Manevich, M. Méndez-Lojo, D. Prountzos, and X. Sui, "The Tao of parallelism in algorithms," in *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 2011, pp. 12–25.

[19] X. Shi, J. Liang, X. Luo, S. Di, B. He, L. Lu, and H. Jin, "Frog: Asynchronous graph processing on gpu with hybrid coloring model," *Huazhong University of Science and Technology, Tech. Rep. HUST-CGCL-TR-402*, 2015.

[20] Y. Tian, A. Balmin, S. A. Corsten, S. Tatikonda, and J. McPherson, "From "Think like a vertex" to "Think like a graph"," *Proc. VLDB Endow.*, vol. 7, no. 3, pp. 193–204, Nov. 2013.

[21] Y. Wang, A. Davidson, Y. Pan, Y. Wu, A. Riffel, and J. D. Owens, "Gunrock: A high-performance graph processing library on the GPU," *SIGPLAN Not.*, vol. 51, no. 8, pp. 11:1–11:12, Feb. 2016.

[22] 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, 2014.