Graphie: Large-Scale Asynchronous Graph Traversals on Just a GPU

Wei Han, Daniel Mawhirter, and Bo Wu
Department of Computer Science
Colorado School of Mines
Golden, CO, USA
e-mail: {whan, dmawhirt}@mymail.mines.edu, bwu@mines.edu

Matthew Buland[‡]
Salesforce
Louisville, CO, USA
e-mail: gik0geck0@gmail.com

***Abstract*—Most GPU-based graph systems cannot handle large-scale graphs that do not fit in the GPU memory. The ever-increasing graph size demands a scale-up graph system, which can run on a single GPU with optimized memory access efficiency and well-controlled data transfer overhead. However, existing systems either incur redundant data transfers or fail to use shared memory. In this paper we present Graphie, a system to efficiently traverse large-scale graphs on a single GPU. Graphie stores the vertex attribute data in the GPU memory and streams edge data asynchronously to the GPU for processing. Graphie's high performance relies on two renaming algorithms. The first algorithm renames the vertices so that the source vertices can be easily loaded to the shared memory to reduce global memory accesses. The second algorithm inserts virtual vertices into the vertex set to rename real vertices, which enables the use of a small boolean array to track active partitions. The boolean array also resides in shared memory and can be updated in constant time. The renaming algorithms do not introduce any extra overhead in the GPU memory or graph storage on disk. Graphie's runtime overlaps data transfer with kernel execution and reuses transferred data in the GPU memory. The evaluation of Graphie on 7 real-world graphs with up to 1.8 billion edges demonstrates substantial speedups over X-Stream, a state-of-the-art edge-centric graph processing framework on the CPU, and GraphReduce, an out-of-memory graph processing systems on GPUs.**

***Index Terms*—Graph Traversal; GPUs; Out-Of-Memory Processing; Data Transformation**

I. Introduction

Graphs are used in various domains, such as machine learning, social networking, and bioinformatics, thanks to their flexible modeling capability. With ever-increasing graph sizes, it becomes critical to improve the performance of graph processing, because a Breadth-First Search (BFS) run on a real-world graph in a high-end system may take more than 10 minutes [1]. Scaling up the performance of graph processing is however challenging due to the well-known random access problem[1]–[3] and dramatic frontier change across phases of the same application and across inputs.

To accelerate large-scale graph analytics, researchers have proposed many scale-out and scale-up graph processing systems on CPUs [4]–[6]. PowerGraph [7] considers the power-law distribution of vertex degrees and implements a vertex-cut partitioning method to reduce inter-machine communication and improve load balance. PowerLyra [8] further improves the performance by selectively applying vertex-cut and edge-cut approaches that match the characteristics of different parts of the graph. Although those distributed graph systems provide impressive performance, users may still prefer a single-machine based graph system, which is easy to manage and understand [9]. GraphChi [9] is the first graph system that can process large-scale graphs with decent performance on a single machine. X-Stream [1] proposes the edge-centric processing model which sequentializes accesses to edge data. Galois [10] implements a high-performance data-centric infrastructure to support existing graph processing domain-specific languages.

With the increasing popularity of GPU computing, scaling up graph processing on a single GPU also attracted substantial attention [11]–[14]. CuSha [2] implements G-Shard, a similar data structure as used in GraphChi, which optimizes memory coalescing. Gunrock [15] provides a set of high-level primitives, which demonstrate an order of magnitude speedup over PowerGraph. Unfortunately, neither CuSha nor Gunrock can process graphs that do not fit in the GPU memory. However, many real-world graphs have billions of edges, and the size of the edge data alone (e.g., 11 GB for the Twitter graph used in this work) can be easily larger than the limited GPU memory size (e.g., 6GB for the Nvidia Titan GPU).

In this paper, we focus on large-scale graph traversals, such as BFS and Connected Components (CC), which most existing GPU-based graph systems cannot handle. We face three major challenges. First, a traversal touches a large amount of data but performs little computation. For example, prior work shows that the ratio between data transfer time and kernel execution time on real-world graphs can be up to 2 [16], indicating that data transfer may dominate the execution. Second, the random access problem leads to poor GPU memory efficiency, and meanwhile makes it hard to leverage shared memory. Third, the frontier (the set of active vertices) of a graph traversal changes throughout the execution depending on the topology of the graph.

GraphReduce [17] and GTS [16] are two existing GPU-based graph systems that claim to be able to process out-of-memory graphs, which do not fit into the GPU memory. But neither of them well addresses all three challenges. For

[‡]The work was conducted as a graduate student at Colorado School of Mines.

example, GraphReduce heavily optimizes for GPU memory access efficiency. It uses the Compressed Sparse Column (CSC) format for the gather phase and Compressed Sparse Row (CSR) format for the scatter phase. Transferring both CSC and CSR data contains substantial redundancy, which worsens the GPU memory pressure and lengthens data transfer time. GTS can adapt to the dynamic frontiers and avoid redundant data transfers. In addition, its slotted page format helps improve load balance and memory coalescing. But GTS fails to exploit shared memory, and its graph representation is rarely seen in the graph processing field.

In this paper, we present Graphie, the first GPU-based graph system that addresses all the three challenges of large-scale graph traversal. It overcomes the GPU memory capacity limitation and can efficiently process graphs with billions of edges. Graphie uses one of the most popular graph formats, edge list, and divides it into partitions. It keeps the vertex attribute data in the GPU memory, and streams the edge partitions to the GPU. Unlike current systems (e.g., GraphReduce), Graphie does not introduce any redundancy besides the edge data. Its optimized performance comes from one key idea: vertex renaming. The renaming has two rounds powered by two algorithms. Once the first-round renaming is done, Graphie allows efficient use of shared memory to accelerate vertex attribute data accesses, as well as improving memory coalescing. After the second-round renaming, Graphie can use a small boolean array to keep track of the partitions that contain active vertices as source vertices and hence should be transferred to the GPU. Graphie stores the boolean array in shared memory, and updates its elements in constant time, which is infeasible without renaming. Graphie hides data transfer overhead through asynchronous streaming and avoids redundant data transfers by reusing edge partitions already resident in the GPU memory. These techniques combined together make Graphie substantially outperform X-Stream (up to 98X performance improvement), a state-of-the-art edge-centric graph processing framework on the CPU. We cannot directly compare the performance with GraphReduce [17], which is a similar system but is not released to public. However, although our used GPU is just slightly more powerful than the GPU used in the GraphReduce work (details in Section VI), on the same set of non-trivial graphs used by GraphReduce, the results of Graphie demonstrate up to 179X speedup over the results reported for GraphReduce.

We make the following contributions in this paper:

- We propose two renaming algorithms to improve large-scale graph traversal's performance on GPUs. The first algorithm enables efficient use of shared memory for accessing vertex attribute data. The second enables the use of a small boolean array in shared memory to track the active partitions that should be transferred to the GPU. Neither algorithm introduces any space overhead in the GPU memory or in the graph storage on disk.
- We propose an asynchronous edge streaming runtime, which hides data transfer overhead and efficiently reuses transferred data across super steps.

```
for v in vertex_partition
  if v.active == true
    for e in v.in_edges
      process_in_edge(e)
    for e in v.out_edges
      process_out_edge(e)
```

(a) Vertex-centric

```
for e in edge_partition
  if e.src.active == true
    //read e.src
    send_update_over_edge(e)
    //write e.dst
    apply_update(e.dst)
```

(b) Edge-centric

Fig. 1: Vertex-centric vs. edge-centric graph processing.

- We integrate the renaming algorithms and the runtime into a GPU-based graph system named Graphie, which supports expressive graph algorithm programming and the traversal of graphs with billions of edges.
- We evaluate Graphie on 7 real-world and synthetic graphs used in various studies. The results show that Graphie produces up to 98X speedup over X-Stream. When processing small graphs, Graphie's performance is comparable to CuSha, a high-performance GPU-based system to process in-memory graphs.

II. Background and Motivation

This section first provides the background of the vertex-centric and edge-centric graph processing models, and explains the reason for Graphie to choose the edge-centric model. It then presents the high-level framework to process out-of-memory graphs on GPUs. It motivates the work by describing the performance issues an optimizing graph system must address.

A. Graph processing models and data organization

There exist many models for single-machine large-scale graph processing, such as vertex-centric [9], edge-centric [1], data-centric [10], path-centric [18], and matrix-based [19] models. We limit the discussion to the vertex-centric model, represented by GraphChi [9], and the edge-centric model, represented by X-Stream [1], because they are extensively studied and implemented in many systems. Figure 1 (a) shows the high-level workflow of the vertex-centric model, which divides the vertices into vertex partitions. During the processing of each vertex partition, the accesses to the vertices have good spatial locality, while the accesses to the in_edge and out_edge are random. Alternatively, the edge-centric model, shown in Figure 1 (b), divides the edges into partitions and enables sequential accesses to edges. However, the accesses to vertices are random as a downside. Because the number of edges is typically much larger than the number of vertices, the edge-centric model, by sequentializing the accesses to edges, outperforms its counterpart as demonstrated by multiple systems [1], [16], [20].

B. Out-of-memory graph traversals on GPUs

GPUs have been successfully used for in-memory graph traversals [12], [15], [21]. The graph data only need to be copied at the beginning of the processing, whose overhead is amortized to the many phases of traversals. Once the whole graph data are readily available in the GPU memory, systems

TABLE I: Datasets Used in The Experiments

Name	Vertices	Edges
cage15[22]	5.1M	99.1M
kron_g500_logn21[23]	2.1M	182.1M
nlpkkt160[24]	8.3M	221.1M
orkut[25]	3.1M	117.2M
uk-2002[26]	18.5M	298.1M
friendster[25]	124.8M	1,806.1M
twitter[27]	61.6M	1,468.4M

such as CuSha [2] or GunRock [15], can provide up to two orders of magnitude performance improvement over state-of-the-art CPU-based graph systems. However, GPUs have limited main memory. A modern GPU, such as Nvidia Titan Z, is only equipped with 6GB memory, while many real-world graphs have billions of edges and are hence too large to fit in the GPU memory. Table I shows 7 graphs used in this paper, which are used by other studies [1], [16], [17]. Suppose an edge needs 8 bytes, 4 bytes for the source vertex ID and 4 bytes of the destination vertex ID. The graph friendster's topology data (i.e., edges) alone need 14GB memory space. Since the execution also needs to store the vertex attributes and possibly edge weights, the actual memory requirement can be significantly larger.

Algorithm 1 shows the basic workflow to process out-of-memory graphs on a GPU. The function $ProcessGraphOnGPU$ runs on the CPU and takes a graph G stored in the CPU memory as the input. It initializes the vertex attribute array VA_CPU and copies it to the GPU memory. The assumption is that the GPU memory is large enough to hold the vertex attribute array, which is true for most real-world graphs [2], [16], [17]. The graph's edge data are divided into partitions (i.e., $G.edge\_partitions$). The size of the partitions is chosen such that a partition can reside in the GPU memory together with the vertex attribute array VA_GPU. Each iteration of the while loop represents a super step, whose finishing point implicitly indicates a global synchronization. The loop body transfers the edge partitions one by one and invokes a kernel to process the transferred edge partitions to update the attribute array. Note that the edge partitions are read-only and can be safely overwritten after being processed. The kernel function $PartitionKernel$ launches as many threads as the number of edges in the partition. One thread corresponds to one edge and calls the update device function if the edge's source vertex is active. The update function may update the attribute data of the destination vertex. If an update happens, we say the destination vertex is activated. Both GTS and GraphReduce implement a similar workflow.

a) Example: Figure 2 shows an example graph and its edge partitions. The graph has 8 vertices and 16 edges. Each edge partition has 4 edges. We do not assume the edges are sorted, because the input graphs may not have been pre-processed. Suppose vertex 5 is the root node for a BFS traversal. When the execution starts, it is the only active vertex. We further suppose each kernel invocation launches

Algorithm 1: Basic workflow to process out-of-memory graphs on the GPU.

```
1  //G is the input Graph
2  Function ProcessGraphOnGPU(G)
3      $VA\_CPU \leftarrow init\_vertex\_attr(G)$
4      while not finished do
5          $trans\_data(VA\_GPU, VA\_CPU, CPUToGPU)$
6          foreach $EP\_CPU$ in $G.edge\_partitions$ do
7              $trans\_data(EP\_GPU, EP\_CPU, CPUToGPU)$
8              $PartitionKernel <<< ... >>>$
                 $(EP\_GPU, VA\_GPU, ...)$
9      $trans\_data(VA\_CPU, VA\_GPU, GPUToCPU)$
10     return $VA\_CPU$
11 Function PartitionKernel(edge_partition)
12     $tid \leftarrow get\_thread\_id()$
13     $e \leftarrow edge\_partition[tid]$
14     if $e.src$ is active then
15         $Up(VA[e.dst], VA[e.src])$
```

one single thread block of 4 threads to process the transferred edge partition. When processing the third edge partition, the first thread would activate vertex 1. The GPU can successfully process this graph if the memory is large enough to hold the vertex attribute data (8 variables) and one edge partition (4 edges). In the remainder of the paper, whenever this example is used, we have the same assumption about the thread block size and the partition size. We also assume the thread block can only load 4 vertices to the shared memory, which can be viewed as software-controlled cache.

Algorithm 1 shows that to improve the performance we should reduce the data transfer overhead (line 7) and/or improve the kernel's performance (line 8). We next present the challenges of achieving these goals using the example shown in Figure 2.

b) Issue 1, dynamic frontiers: Graph traversals typically have complicated behaviors depending on the algorithm and graph topology. Specifically, the number of active vertices (i.e., the frontier) and their distribution in the vertex set may vary dramatically across super steps. For example, BFS starts with one active vertex (i.e., the root vertex) and in each super step activates a new set of vertices which are just discovered in this step. Connected component, on the other hand, has all vertices as active vertices at the beginning of the execution. The number of active vertices decreases towards the end of the execution. In the context of graph traversals on the GPU, the transfer of an edge partition is redundant if none of its edges are out-edges of active vertices. For instance, Figure 2 shows that in the second super step of a BFS run with vertex 6 as the root, the second edge partition contains all the out-edges of the activated vertex (i.e., vertex 1), and hence is the only one that should be transferred to the GPU for optimized performance.

Current graph systems that support out-of-memory graph processing on GPUs solve this problem by keeping track of

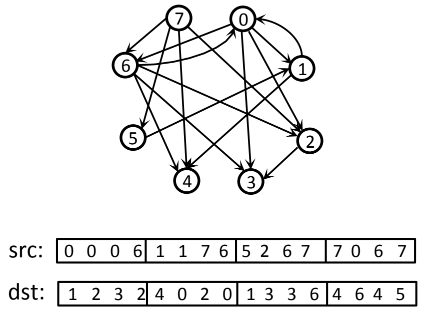

7
0
6
1
5
2
4
3
src: 0 0 0 6 | 1 1 7 6 | 5 2 6 7 | 7 0 6 7
dst: 1 2 3 2 | 4 0 2 0 | 1 3 3 6 | 4 6 4 5

Fig. 2: An example graph and its partitioned edge list.

the active vertices via a boolean array. The size of the array is the number of vertices in the input graph, because any vertex may be active for the next super step. This approach has three drawbacks. First, despite contributing nothing to the real computation, this meta array stays in the GPU memory and incurs non-trivial space overhead (e.g., 124.8MB for friendster). Second, the array needs to be copied back to the CPU at the end of each super step, causing time overhead on the critical path. Third, the array is too large to fit in GPU's shared memory. As such, every update causes one extra GPU main memory access to update the corresponding boolean variable.

c) Issue 2, memory access inefficiency: GPU kernel's performance highly relies on memory access efficiency. There are two major ways to improve the efficiency. First, if the threads running on the same SIMD unit access nearby memory locations, the memory accesses may be coalesced to reduce the number of memory transactions. Second, if the threads of the same thread block repeatedly access the same memory location, the data element at that location should be fetched to shared memory for those threads to quickly access. A naive implementation of Algorithm 1 fails to exploit either memory coalescing or shared memory. Random vertex accesses, which occur during processing each partition, leads to excessive uncoalesced memory transactions. The randomness also complicates the use of shared memory, which needs heavyweight pre-processing to figure out the set of accessed vertices, easily offsetting the benefit.

In the next two sections, we propose several techniques to address these two issues, followed by the presentation of the Graphie framework that integrates these techniques for efficient large-scale graph traversal.

III. Optimizing Kernel Execution through Vertex Renaming

This section discusses the inadequacy of existing solutions to the performance issues described in last section. It presents two vertex renaming algorithms to improve memory access efficiency and to efficiently determine the active partitions that should be transferred. The section explains why the renaming process does not introduce any space overhead in the GPU memory.

A. Improving GPU memory access efficiency

A naive solution to improving the memory access efficiency problem is to sort the edges by source vertex ID. Figure 3 (a) shows the sorted edge list of the example in Figure 2. We note that the sorting improves the performance of the accesses to the source vertices because of enhanced locality but with the accesses to the destination vertices remaining random.

It may seem after the sorting, the distinct source vertices can be loaded into the shared memory to reduce main memory accesses. However, although the number of distinct vertices is up to the number of edges in the partition, the gap between the first source vertex ID and the last source vertex ID can be larger than the number of vertices that can be loaded to shared memory. The reason is that many vertices whose IDs are in between do not have out-going edges. As Figure 3 (a) shows, the second partition has three distinct vertices, but the gap is 4. Recall that we assume the shared memory used by one thread block can hold up to 4 vertices, so we cannot load 5 vertices (2–6) to the shared memory. Therefore, even with the sorted edges, it still requires a non-trivial pre-processing phase to figure out the distinct source vertices (i.e., vertices 1, 2 and 5 in the example).

To address these problems, we propose a renaming technique which not only improves memory coalescing but also makes using shared memory straightforward. The renaming process happens after the edges are sorted by source vertex. Algorithm 2 shows how the technique works through two functions: $RenameForMemory$ to rename the vertices and $PartitionKernelV2$ to demonstrate the convenient use of shared memory. The idea of $RenameForMemory$ is to pack the vertices into contiguous values where the vertices with nonzero out degree occupy the lower indices. It first scans the edges to compute the out-going degree for each vertex (line 4). It then uses an array new_to_old to compute the new IDs for the vertices. After the first for loop, the vertex of the ID given by $new\_to\_old[i]$ should have the new ID i. To quickly access the new ID given the old ID, we use an array old_to_new, whose ith element is the new ID of the ith vertex in the original graph (lines 10). Finally, the out-edges of each vertex are sorted by destination (line 13–15), which improves the spatial locality of accessing destination vertices, thus improving memory coalescing.

After the edges are renamed and reordered, $PartitionKernelV2$ shows the convenient use of shared memory in the kernel. Since the IDs of the source vertices of the edges are contiguous, we easily calculate the number of distinct source vertices in each partition based on the source vertex IDs of the first edge and the last edge (line 21). We only load the attribute variables of the distinct source vertices to shared memory (lines 24 and 25) followed by a thread block-level barrier to avoid data races. The update function Up accesses the attribute variables of the source vertices in the shared memory (lines 28), which may significantly reduce the number of global memory accesses because those source vertices may have many out-going edges.

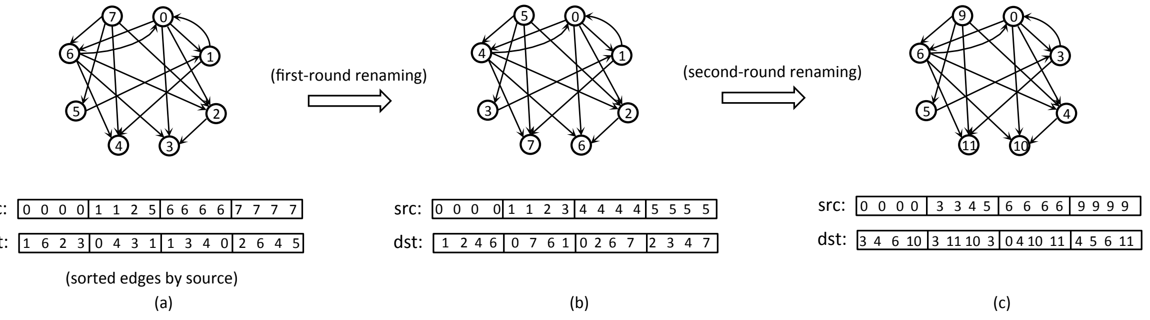

(first-round renaming)
(second-round renaming)
src: 0 0 0 0 | 1 1 2 5 | 6 6 6 6 | 7 7 7 7
dst: 1 6 2 3 | 0 4 3 1 | 1 3 4 0 | 2 6 4 5
(sorted edges by source)
(a)
src: 0 0 0 0 | 1 1 2 3 | 4 4 4 4 | 5 5 5 5
dst: 1 2 4 6 | 0 7 6 1 | 0 2 6 7 | 2 3 4 7
(b)
src: 0 0 0 0 | 3 3 4 5 | 6 6 6 6 | 9 9 9 9
dst: 3 4 6 10 | 3 11 10 3 | 0 4 10 11 | 4 5 6 11
(c)

Fig. 3: Illustration of the renaming process.

Once the processing on the GPU finishes, we transfer the vertex attribute array (VA_GPU) back to the CPU to store in the array VA_CPU. However, because the vertices are renamed, we need to map the updated attribute data to the corresponding vertices. The problem can be easily solved, because we maintain the mapping in the array new_to_old obtained in $RenameForMemory$. The ith attribute variable $VA\_CPU[i]$ should belong to the vertex of ID $new\_to\_old[i]$ in the original graph.

a) Example: Figure 3 (b) shows the renamed and re-ordered graph data of the example graph after being processed by Algorithm 2 (first-round renaming). Vertex 3 and 4 in the original graph have new IDs 6 and 7 (i.e., the largest IDs), respectively, because they do not have out-going edges. Correspondingly, the IDs of vertex 5–7 in the original graph are reduced by 2. Observe that the source vertices of the edges are contiguous and that the out-going edges of the same source vertex are sorted by destination vertex ID. Unlike Figure 3 (a), we can now easily compute the number of distinct source vertices of the second partition thanks to the source ID contiguity.

b) Analysis: $RenameForMemory$ runs online or offline on the CPU to pre-process the edge data. It requires the edges to be sorted by source ID, but most systems demand a similar sorting process (e.g., GraphChi and GTS). The rationale of the pre-processing is that it is done just once and the cost can be amortized over many runs with different inputs such as different roots for BFS or with different algorithms on the same graph. In $RenameForMemory$, each state before the final sorting (line 13) takes linear time in terms of the number of vertices or the number of edges. The sorting stage takes $O(N \times \Delta log(\Delta))$, where Δ is the maximum outdegree. Therefore, $RenameForMemory$'s worst complexity is $O(M + N \times \Delta log(\Delta))$, where M is the number of edges. If we assume Δ to be the average degree multiplied by a constant, the complexity can be estimated as $O(M + N \times (M/N) \times log(M/N)) = O(Mlog(M/N))$, which is less than sorting the edge list, which takes (i.e., $O(Mlog(M))$). Moreover, Algorithm 2 does not incur any extra space overhead in the GPU or extra storage overhead in the disk.

Algorithm 2: Renaming vertices to improve memory access efficiency.

```
1  //$G$ is the input graph
2  //$N$ is the number of vertices
3  **Function** *RenameForMemory($G$)*
4      $out\_degrees \leftarrow ComputeOutDegrees(G)$
5      $new\_to\_old \leftarrow \{0, 1, ..., N-1\}$
6      **foreach** $id$ *in* $new\_to\_old$ **do**
7          **if** $out\_degrees[id] = 0$ **then**
8              move $id$ to the end of $new\_to\_old$
9      **for** $i \leftarrow 0$ ***to*** $N-1$ **do**
10         $old\_to\_new[new\_to\_old[i]] \leftarrow i$
11     **foreach** $e$ *in* $G.edges$ **do**
12         $e.src \leftarrow old\_to\_new[e.src]$;
             $e.dst \leftarrow old\_to\_new[e.dst]$;
13     **foreach** $v$ *in* $G.vertices$ **do**
14         **if** $out\_degrees[v.id] \neq 0$ **then**
15             Sort $v$'s outgoing edges by destination
16 //$BS$ is the thread block size
17 //$SVA$ is an array in the shared memory
18 **Function** *PartitionKernelV2($EP$, $VA$)*
19     $tid \leftarrow get\_thread\_id()$
20     $start\_vertex \leftarrow EP[0].src$
21     $num\_distinct\_srcs \leftarrow$
           $EP[BS-1].src - EP[0].src$
22     $e \leftarrow edge\_partition[tid]$
23     //Load attributes of distinct vertices to shared
         memory
24     **if** $tid < num\_distinct\_srcs$ **then**
25         $SVA[tid] \leftarrow VA[start\_vertex + tid]$
26     $Barrier()$ //synchronize threads to avoid data race
27     **if** $e.src$ *is active* **then**
28         $Up(VA[e.dst], SVA[e.src - start\_vertex])$
```

Algorithm 3: Renaming vertices to efficiently activate partitions.

```
1  //G is the graph produced by Algorithm 2
2  //N is the number of vertices
3  // $P$ is the number of partitions
4  //$num\_virtual\_vertices$ is an array of size $P$
     initialized to all 0's
5  //$new\_ids$ is initialized as $\{0, 1, ..., N-1\}$
6  Function RenameForActivePartitions(G)
7      //$SZ$ sotres the number of distinct source vertices
         for all partitions
8      traverse partitions to compute $SZ$
9      compute $SZ\_max$ //the maximum number in $SZ$
10     for $i \leftarrow 0$ to $P-1$ do
11         $num\_virtual\_vertices \leftarrow SZ\_max - SZ[i]$
12     $pre\_sum \leftarrow$
         $InclusiveScan(num\_virtual\_vertices)$
13     append 0 at the front of $pre\_sum$
14     for $i \leftarrow 0$ to $P-1$ do
15         foreach $v \in EP[i].distinct\_source\_vertices$
             do
16             $new\_ids[v.ID] \leftarrow$
                 $new\_ids[v.ID] + pre\_sum[i]$
17     foreach vertex $v$ without out-going edges do
18         $new\_ids[v.ID] \leftarrow$
             $new\_ids[v.ID] + pre\_sum[P]$
19     foreach $e$ in $G.edges$ do
20         $e.src \leftarrow new\_ids[e.src]$;
             $e.dst \leftarrow new\_ids[e.dst]$;
21 //$activated$ is an array of size $P$ in the shared memory
     initialized to all $False$
22 //$OV$ is the number of vertices that have out-going
     edges Function PartitionKernelV3(EP, $VA$)
23     $tid \leftarrow get\_thread\_id()$
24     $start\_vertex \leftarrow EP[0].src$
25     $num\_distinct\_srcs \leftarrow$
         $EP[BS-1].src - EP[0].src$
26     $e \leftarrow edge\_partition[tid]$
27     if $e.src < OV$ then
28         $src\_offset \leftarrow pre\_sum[e.src/SZ\_max]$
29     if $e.dst < OV$ then
30         $dst\_offset \leftarrow pre\_sum[e.dst/SZ\_max]$
31     else
32         $dst\_offset \leftarrow pre\_sum[P]$
33     if $tid < num\_distinct\_srcs$ then
34         $SVA[tid] =$
             $VA[start\_vertex + tid - src\_offset]$
35     $Barrier()$ //synchronize threads to avoid data race
36     if $e.src$ is active then
37         if $Up(VA[e.dst - dst\_offset], SVA[e.src -$
             $start\_vertex])$ and $e.dst < OV$ then
38             $activated[e.dst/SZ\_max] \leftarrow True$
```

B. Efficiently Activating Partitions

Recall that graph traversals have dynamic frontiers, and only the edge partitions that contain one or more vertices in the dynamic frontier should be transferred to the GPU to save data transfer time. As discussed in Section II-B, both GTS and GraphReduce uses an array of size N (i.e., the number of vertices) to record which vertices are in the frontier for the next super step, which burdens the GPU memory and increases data transfer cost. Worse, the array is too large to benefit from shared memory. To address this problem, the ideal solution is to have a small boolean array (also called a tag array), of size P the number of partitions, on the GPU to track the partitions that contain active vertices as source vertices. For example, for the graph shown in Figure 3 (b), we only need a boolean array of size 4 (instead of 16 in existing systems). If only vertex 2 is activated in the current super step, the boolean array should be $\{False, True, False, False\}$. After processing this array, the CPU knows that only partition 2 should be transferred to the GPU in the next super step, as it contains all the out-going edges of the vertex 2. However, computing which partition contains the activated vertex involves searching and needs $O(logP)$ time. Since the overhead occurs every time a vertex is updated, this approach may perform worse than the existing approach of maintaining a large boolean array.

We propose a technique to further rename the vertices based on the renamed graph produced by Algorithm 2. Algorithm 3 shows the process of reducing the cost of figuring out the activated partition to one single division operation. The essential idea is to insert virtual vertices, which do not need storage, to the source vertex sets of partitions, so that the IDs of distinct source vertices of each partition fall into ranges of the same size. Lines 8 and 9 calculate for each partition the difference between its number of distinct source vertices and the maximum number of distinct vertices across partitions. $Num\_virtual\_vertices$ of size P maintains the numbers of virtual vertices that should be inserted. Pre_sum, computed via an inclusive scan on $num\_virtual\_vertices$, stores the total number of inserted virtual vertices before each edge partition. With 0 appended at the front, $pre\_sum[P]$ is now the total number of inserted virtual vertices. The vertex IDs of the source vertices of each partition i should be increased by $pre\_sum[i]$ to reflect the number of inserted virtual vertices before them. Similarly, the IDs of the vertices without out-going edges should be increased by $pre\_sum[P]$. The renaming of the IDs in edge data is the same as in Algorithm 2.

We next show how the renaming enables constant time update of the boolean array $activated$ in the new kernel function $PartitionKernelV3$. Recall that the size of $activated$ is equal to P, the number of partitions, and hence the array is usually small enough to be stored in shared memory. One key difference from $PartitionKernelV2$ in Algorithm 2 is that when the attribute array is accessed, the index should be decreased by an offset (lines 34 and 37). The offset equals the number of inserted virtual vertices before the partition that contains the vertex as a source vertex. Lastly, if the update function returns true, meaning that the destination vertex ($e.dst$) is updated, and the updated vertex has out-going edges, the partition that contains it as a source vertex should be activated. The ID of that partition can be easily calculated as $e.dst/SZ\_MAX$, a tremendous improvement over naive searching.

a) Example: Figure 3 (c) shows the renamed graph by Algorithm 3 based on Figure 3 (b). Partition 2 has 3 distinct source vertices, the largest among all the partitions. All the other partitions only have 1 distinct source vertex. Hence, we insert 2 virtual vertices in each of the source vertex set of partition 1, 3, and 4. Suppose a thread processes edge $(5, 3)$ and needs to update vertex 3. It writes vertex 3's new value to $VA[3-2]$ (i.e., $VA[1]$). The ID of the partition that contains 3 is easily computed as $3/3 = 1$. The boolean variable $activated[1]$ is then assigned to $True$.

b) Analysis: We stress that although Algorithm 3 may insert a large number of virtual vertices, it, like Algorithm 2, does not introduce extra space overhead in the GPU memory, because we only need N elements in the attribute array. All the steps of $RenameForActivePartitions$, including the nested loop (lines 14–16), can be implemented in linear time. Thus, the time complexity is $O(M)$.

IV. Minimizing Data Transfer Overhead through Asynchronous Edge Streaming

This section presents the techniques used by Graphie to reduce data transfer overhead through asynchronous edge streaming.

Graph traversals are memory-intensive with very low arithmetic intensity. As such, the data transfer of a partition may take longer than its processing on the GPU. Fortunately, modern GPUs support parallel command queues (e.g. Hyper-Q in Nvidia GPUs [28]), which allow overlapping between kernel execution and data transfer. Graphie leverages this capability to hide edge partition transfer overhead as shown in Figure 4. After initializing the vertex attribute array (VA), Graphie divides the remaining GPU memory into K partition buffers, where K is the number of streams, unless the user explicitly specifies the partition size. Graphie takes turns to use the streams to transfer the edge partitions to the GPU. In each stream, a kernel invocation command always follows a data transfer command to process the transferred partition. The commands sent to the same command queue are executed

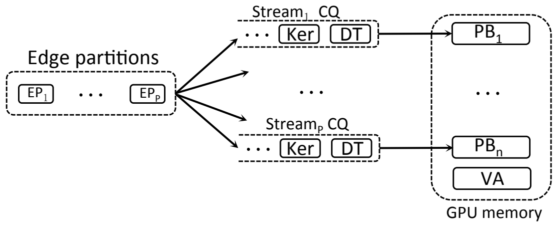


Fig. 4: Asynchronous edge streaming through parallel command queues. DT, CQ, and PB represent data transfer, command queue and partition buffer, respectively.

sequentially. Current Nvidia GPUs support up to 32 command queues. If more streams are used, some streams will be serialized to use the same command queue. Hence, Graphie uses 32 streams by default unless specified otherwise.

Section III-B described the renaming technique to efficiently identify the activated partitions to process in the next super step. Graphie's runtime makes sure that only activated partitions are transferred. While this optimization greatly reduces the data transfer overhead when the number of activated partitions is small, redundant transfer may still occur if the activated partition is already in the GPU memory. Suppose the average number of activated partitions in each super step is A. On average, the percentage of redundant partition transfers can be estimated as $\frac{A \times K}{P}$, which can be non-trivial if the graph size is not dramatically larger than the GPU memory size.

Because of the FIFO property of the command queue, it is obvious that the last processed partition by each queue can be reused in the next super step. However, to reuse those partitions, it is critical to not overwrite them before processing them. Graphie solves this problem by first processing the partitions that are activated and also resident in the GPU memory. For each of such partitions, Graphie inserts only the kernel invocation command to the queue which handled that partition in the last super step.

V. Graphie System

A. Graphie workflow

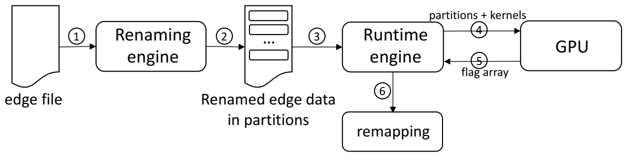


Fig. 5: The workflow of Graphie.

We integrate all the proposed techniques in the Graphie system, whose workflow is shown in Figure 5. The circled numbers represent different steps. Graphie reads edge data in text or binary format (step 1). It stores the edge data in an edge list, which is processed by the renaming engine for renaming and partitioning (step 2). Note that the renaming engine can work online or offline. For each edge partition, Graphie uses one array to store the source vertex IDs and one array to store the destination vertex IDs. Graphie uses one more array to store the weight data if there is any. Such a design choice is to improve memory coalescing, which is used in many other GPU-based systems. The runtime engine reads all edge partitions in the CPU memory (step 3). In each super step, it transfers edge partitions to the GPU and invokes kernels to process the partitions (step 4). At the end of each super step, it copies back the flag array (i.e., $activated$ in Algorithm 3). If any partition is activated, it starts another super step that consists of steps 4 and 5. Once it detects no active partitions, the output vertex attribute array is remapped to the original vertex ids to cope with renaming.

B. Programming interface

Graphie provides a generic kernel, which implements the $PartitionKernelV3$ function in Algorithm 3. It invokes two device functions that the user must implement. The first device function is $Initialize\_VA$, which should initialize the vertex attribute array. The implementation is application dependent. For example, for BFS $VA[root]$ should be initialized to 0, while all other elements should be initialized as positive infinity. The second device function is Up, which processes an edge if the source vertex is active, and returns true if the destination vertex of the edge is updated (i.e., activated). For BFS, the destination vertex is updated if its distance (i.e., $VA\_GPU[e.dst]$) is larger than the distance of the source vertex plus one (i.e., $VA\_GPU[e.src] + 1 < VA\_GPU[e.dst]$). On the CPU side, the user needs to specify which partitions are active and hence should be processed on the GPU in the first super step. For example, the partition that contains the root vertex should be active for BFS, while all the partitions should be active for CC.

C. Selecting partition size

As discussed in Section IV, Nvidia GPUs support up to 32 parallel command queues. We use GS, VS, AS to denote the size of the GPU memory, the size of the vertex data, and size of the flag array, respectively. Given 32 streams, the partition size can be computed as $(GS - VS - AS)/32$. However, this partition size does not address the various properties of graphs. For small graphs, the whole graph may fit in one partition. As such, it only uses one stream and does not leverage the concurrency of the command queues. The kernel computation starts after the whole graph is transferred, wasting the opportunity to overlap transfer and compute. It is also possible that a traversal only accesses a subset of the edges. In this case, transferring the whole graph is not necessary. Due to these reasons, Graphie chooses partition size as $ES/32$, where ES is the edge data size, to fully utilize all the available command queues and avoid unnecessary data transfer. For large graphs, it may be infeasible to support $(GS - VS - AS)/32$ as the partition size. Recall that during partitioning, Graphie inserts virtual vertices to the source vertex sets for renaming, which increases the largest vertex ID. Because Graphie uses 32 bits to store the vertex ID, a small partition size may cause an integer overflow problem. Therefore, Graphie chooses the smallest partition that does not overflow the 32-bit integer. Note that we can simply address the problem by using 64-bit representation for vertex IDs. However, we then increase the edge data size by 2 times, leading to increased data transfer overhead. It is our future work to understand the trade-off between the vertex representation and partition size.

VI. Experimental Evaluation

This section evaluates the performance of Graphie by comparing it with existing systems and quantifies the effectiveness of the proposed optimization techniques. Before presenting the results, we introduce the experiment settings and the methodology for the experiments.

TABLE II: GPU Specifications

	Titan Z (half)	Tesla K20c
GPU architecture	Kepler (GK110B)	Kepler (GK110)
Num. of SMX	14	13
Memory	6GB GDDR5	5GB GDDR5
Memory bandwidth	288 GB/S	208 GB/S
Num. of CUDA cores	2,688	2,496
Theoretical throughput	4,494 GFLOPS	3,524 GFLOPS

A. Experiment setting

a) Environment: Our system has an Intel Xeon (E7-4830v3, 2.1GHz) 12-core CPU with Hyperthreading disabled. The main memory of the system is 256GB (16x16GB DDR3 modules at 1866MHz). We use the NVCC compiler version 7.5.17 (g++ version 4.8.4) with O3 to compile all the programs. The operating system is Ubuntu Linux 14.04 with Linux kernel version 3.13. The GPU is a NVIDIA Titan Z containing 2 GPU dies each with 6GB of memory. In all the experiments, we only use one GPU, and hence the device memory is limited to 6GB. Table II shows the specification of a single die of the GPU.

b) Datasets and applications: We evaluate the performance of Graphie using three graph traversal algorithms:

- Breadth-First Search (BFS)
- Connected Components (CC)
- Single-Source Shortest Path (SSSP)

The BFS algorithm traverses the vertices of the graph in order to compute unweighted distances of all vertices from a root vertex. In SSSP the weights are considered and the cost of the cheapest path (in terms of the sum of the weights of its constituent edges) from a root to every vertex is returned. For BFS and SSSP, we always select a vertex in the largest connected component as the root. The CC algorithm finds connected subgraphs of maximal size and returns the component id for each vertex.

We experiment with 7 real-world and synthetic graphs as shown in Table I. Cage15 is an undirected graph describing DNA electrophoresis, 15 monomers in polymer. Kron_g500-logn21 (Kron) is a synthetic graph used in the DIMACS competition. Nlpkkt160 (Nlpktt) is a graph generated by a symmetric indefinite KKT matrix when solving a 3D PDE-constrained optimization problem. Orkut and Friendster are graphs from online gaming and social networks. The graph uk-2002 was obtained from a crawl of the .uk domain in 2002. Twitter is a subgraph of the Twitter follower graph.

c) Compared systems and methodology: We compare Graphie with three graph systems: X-Stream [1], CuSha [2], and GraphReduce [17]. X-Stream is a state-of-the-art edge-centric graph system. Its superior performance over GraphiChi [9], a vertex-centric graph system, is reported in several studies [1], [17]. To make fair comparisons, we allocate enough main memory for X-Stream to load the entire graph. We exclude the IO time, and only measure the graph processing time. CuSha is a high performance GPU-based graph system to process in-memory graphs. For both CuSha and Graphie, we measure the elapsed time between the point the first data transfer from the CPU to the GPU starts and the

TABLE III: Execution times of Graphie and the compared systems.

Application	Framework	Runtime for Graph (in seconds)						
		cage15	friendster	kron_g500-logn21	nlpkkt160	orkut	twitter	uk-2002
bfs	Graphie	0.63	16.44	0.59	6.11	0.21	5.42	4.3
	X-Stream	3.2	927.78	3.33	15.24	2.27	48	10.64
	GraphReduce	18	N/A	4	60	6	N/A	49
	CuSha	0.45	O.O.M.	0.98	2.57	0.38	O.O.M.	O.O.M.
cc	Graphie	0.23	12.46	0.48	1.02	0.26	4.21	5.04
	X-Stream	5.43	1224.52	6.5	21.84	6.23	64.45	28.73
	GraphReduce	41	N/A	9	183	16	N/A	162
	CuSha	0.6	O.O.M.	1.13	1.26	0.6	O.O.M.	O.O.M.
sssp	Graphie	0.24	29.24	1.67	7.03	0.6	14.67	11.73
	X-Stream	4.52	2601.75	9.36	35.98	7.78	930.52	63.04
	GraphReduce	25	N/A	7	92	10	N/A	80
	CuSha	0.57	O.O.M.	1.07	1.53	0.67	O.O.M.	O.O.M.

point the final result data transfer from the GPU to the CPU finishes. GraphReduce is an out-of-memory graph processing system on GPUs, which is also based on edge streaming like Graphie. Unfortunately, GraphReduce is not released to public. We can hence only reference to the reported execution times (data transfer + kernel execution) in [17] on the same set of graphs to make rough comparisons. Note that the reported results in [17] are obtained on an Nvidia Tesla K20c GPU. Table II shows the specification comparison between K20c and the Titan GPU used for Graphie. Both GPUs are based on the Kepler architecture. The Titan GPU has slightly larger main memory (6GB vs. 5GB) and a larger number of CUDA cores (2,688 vs. 2,496).

B. Overall results

Table III summarizes the execution times of Graphie and the three compared systems. N/A for GraphReduce means the corresponding graph is not used in [17]. O.O.M for CuSha means the graph is too large to fit in the GPU memory. We notice that CuSha can only process 4 of the 7 graphs because of the in-memory processing design. It cannot process uk-2002, though the graph's size (3.3GB) is less than the GPU memory. The reason is that CuSha needs to transform the original graph to the G-Shard representation, which incurs non-trivial space overhead. The transformed graph does not fit in the GPU.

Figure 6 shows the substantial speedups of Graphie over X-Stream. The most significant improvements are for Friendster, which is the largest graph we experiment with. Graphie accelerates CC on the graph by 98X, bringing down the execution time from 1224.5 seconds from X-Stream to only 12.5 seconds. On average, Graphie achieves 7.2X, 15.5X, and 20.3X speedups for BFS, SSSP, and CC, respectively. The results demonstrate the power of using GPUs to process large-scale graph traversals and Graphie's lightweight but efficient design to match the GPU programming model and architecture.

For Orkut, Cage15, and Kron, Graphie outperforms CuSha for 7 out of the 9 runs. The results are impressive because Graphie is designed for large-scale graph traversal, while CuSha is heavily optimized for in-memory graph processing. Graphie benefits from asynchronous edge streaming and its concise graph representation, while CuSha's G-Shard format

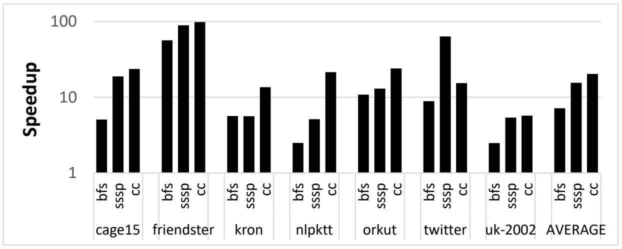


Fig. 6: Speedup over X-Stream.

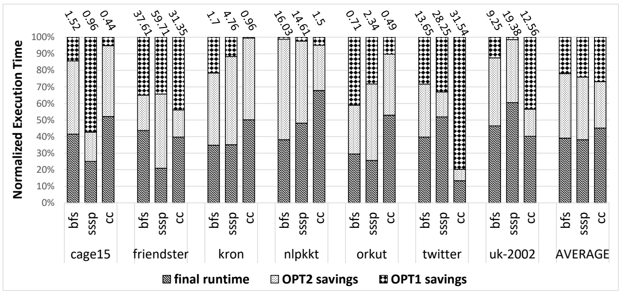


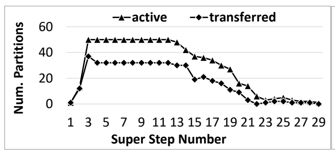
Fig. 7: Benefits breakdown.

introduces space overhead and makes it hard to use streams. For Nlpkkt, CuSha produces superior performance, because the graph has a very large diameter, which has been shown to cause performance problems for the edge-centric model.

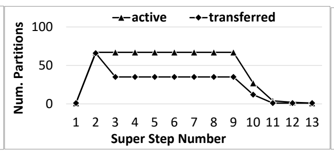
It is worth stressing that the execution times for GraphReduce are reported in [17]. Observe that Graphie reduces the execution times of GraphReduce by up to 99.4% (CC on nlpkkt160) by using a GPU whose theoretical throughput is only 28% higher than that of the GPU used by GraphReduce. Moreover, GraphReduce categorizes the graphs as out-of-memory graphs, because it needs both the CSR and CSC representations of the graph. Thanks to the concise edge partition representation, Graphie can easily fit all these graphs in the GPU memory using less than 4GB GPU memory.

C. Breakdown of the optimization benefits

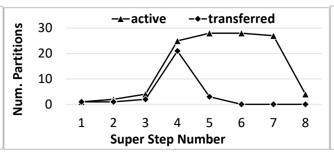
To understand the performance contributions from the proposed techniques, we use OPT1 to represent the optimizations (i.e., shared memory use + only transferring active partitions) enabled by Algorithm 3 and OPT2 to represent the optimization to reuse partitions in the GPU memory. Figure 7 demonstrates the execution time savings from OPT1 and OPT2



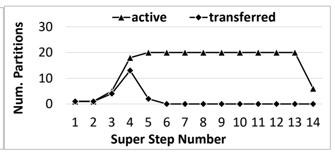
(a) Friendster-BFS



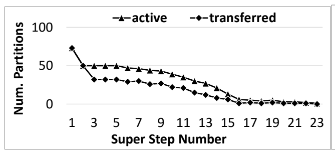
(b) Twitter-BFS



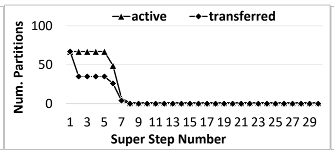
(c) Orkut-BFS



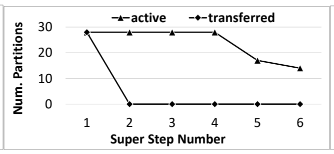
(d) Kron-BFS



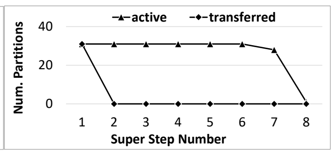
(e) Friendster-CC



(f) Twitter-CC



(g) Orkut-CC



(h) Kron-CC

Fig. 8: The number of active partitions vs. the number of transferred partitions.

with the naive implementation as the baseline, which does not use shared memory and transfers all partitions in each super step. In all the runs, asynchronous streaming is enabled, and we will analyze its benefit in Section VI-D. The results show 22%–27% average execution time reductions from OPT1 for the three algorithms. The largest performance gain is from the CC execution on Twitter, showing 80% reduction of the execution time (i.e., a 5X speedup). But some executions, such as the three runs on Nlpkkt, just show trivial performance improvement, because most of the time all the partitions are active. OPT2 also dramatically improves the performance with average execution time reductions between 28% and 38% across the three algorithms. We observe non-trivial reductions for all runs, confirming the importance of reusing partitions in the GPU memory. Working together, OPT1 and OPT2 reduce the execution time of the naive implementation by an average of 61% for BFS, 62% for SSSP, and 55% for CC across the inputs.

To explain the results we just discussed, we show in Figure 8 the number of active partitions and the number of transferred partitions across super steps for 2 out-of-memory graphs and 2 in-memory graphs. The numbers of active partitions may change dramatically because of the dynamic frontier property of graph traversal algorithms. The patterns for BFS and CC are however very different. BFS starts with 1 active partition, which contains the root vertex. The number of active partitions increases because more vertices and hence partitions are activated. The number decreases at the end of the execution because most of the vertices have been processed. Note that the number of active partitions may remain the same across super steps. It does not mean the corresponding frontiers have the same size. A partition is active even if it only contains one single active source vertex. Therefore, the minimum and the maximum sizes of the frontier to make all partitions active are P and N, respectively, which demonstrate a huge gap. The CC algorithm starts with all partitions being active, and the number of active partitions decreases along the execution. The decrease can be fast (e.g., for Twitter) or slow (e.g., for Kron) depending on the topology of the graph.

The number of transferred partitions is always equal to or less than the number of active partitions. For the latter, Graphie further improves performance by avoiding the transfer of partitions already in the GPU memory. For the two in-memory graphs, Orkut and Kron, the gap between the two curves is large, because the transferred partitions are never overwritten and hence can be reused if needed. For the BFS runs, Graphie does not transfer any more partitions after the 5th super step, because the transferred partitions already contain all the source vertices reachable from the root vertex. The CC runs behave very differently. All the partitions are transferred in the first super step, and thus the whole graph is in the GPU memory for later execution. For the out-of-memory graphs, Friendster and Twitter, the gap between the two curves is smaller, because the GPU memory is not large enough to hold all the partitions and some partitions may be transferred multiple times.

D. *Results on asynchronous streaming*

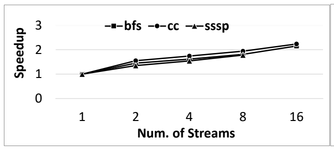
Figure 9 shows the performance benefit from asynchronous edge streaming. We observe that for all the 4 graphs and 3 algorithms, using a larger number of streams improves performance in most cases. Friendster can only leverage 16 streams, because Graphie has to use a large enough partition size to avoid the integer overflow problem discussed in Section V-C. For the other three graphs, Graphie produces more than 2X performance improvement by using 32 streams. The speedup is much worse than linear, because the streams contend to use the PCIe bus and the same set of GPU cores.

E. *Overhead of the renaming processes*

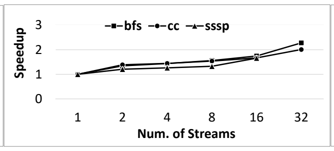
As pointed out in Section III, the renaming process only needs the input graph rather than any runtime parameters (e.g., root vertex ID for BFS) and hence can be performed offline. Table IV shows the overhead of the two rounds of renaming in seconds. For all graphs, the first-round renaming is more expensive than the second-round renaming, which aligns well with the analysis in Section III. We note that the overall overhead from renaming for the two out-of-memory graphs (i.e., Friendster and Twitter) is oftentimes negligible compared to the execution time of X-Stream.

VII. Related Work

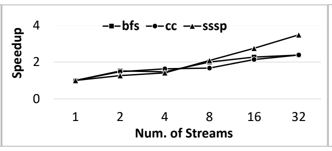
To handle large-scale graphs, researchers have designed many distributed graph processing frameworks [4], [5], [7],



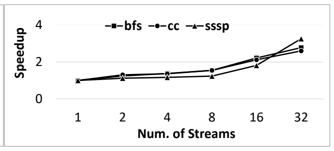
(a) Friendster



(b) Twitter



(c) Orkut



(d) Kron

Fig. 9: Performance improvement when using multiple streams.

TABLE IV: The overhead of renaming in seconds.

Graph	Round 1	Round 2
Cage15	1.28	0.95
Kron	6.51	1.63
Nlpkkt	2.95	2.35
Orkut	2.3	4.26
Uk-2002	6.02	2.67
Friendster	4.48	2.1
Twitter	3.99	1.35

[8], [29], [30] , most requiring the entire graph data, edges and vertex, to reside in main memory during execution. But as several studies have shown [1], [9], [10], [19], [31], [32] a single-machine based system can dramatically reduce management overhead while still providing decent performance. Graphie's design philosophy aligns well with those studies.

Graph processing on GPUs has also been extensively studied from various aspects, including synchronization trade-off [33], data-driven models [12], dynamic graphs [11], graph optimizing compilers [13], [21], and efficient primitives [15]. All those studies assume the input graph fits in the GPU memory, and the research focus is on reducing synchronization overhead or reducing control and memory divergence. Some studies use multiple GPUs to accelerate graph processing. To name a few, Ben-Nun et al. [34] proposed GRoute, which supports efficient asynchronous multi-GPU programming to handle irregularity in graph processing. Liu et al. [35] dramatically improved concurrent BFS on up to 112 GPUs. Khorasani et al. [36] improved inter-GPU communication compared with Medusa [37] and TOTEM [38].

Merrill and others [39] first demonstrated that GPU-based graph traversals can perform substantially better than the CPU-based counterparts. The major idea is to use pre-fix sum to efficiently manage fine-grained tasks. Our work uses pre-fix sum to track the mapping between renamed vertices and their attribute data in the GPU memory. Moreover, their work, like CuSha [2], assumes the graph fits in the GPU. Liu and Huang further improved BFS's performance of in-memory graphs on GPUs through a set of techniques for load balancing and direction optimization. It is unclear whether the proposed techniques work well for out-of-memory graphs on a single GPU.

GraphReduce [17] can process out-of-memory graphs on a single GPU. It optimizes memory coalescing through using two different formats, the benefit of which can be easily cancelled by the redundant data transfers. We show in this paper that Graphie can directly work on edge lists and its renaming and reordering techniques do not introduce any extra space overhead. Further, GraphReduce does not reuse the already transferred data in the GPU when processing large-scale graphs. GTS [16] can also process out-of-memory graphs on GPUs. It leverages slotted page format, which is not popular in the graph processing area. GTS does not use shared memory for accessing vertex data or keeping track of updated vertices. Graphie efficiently tracks the active partitions using shared memory, which involves negligible transfer overhead for the meta flag array.

Several works took advantage of both the CPU and GPU to process graphs. Kaleem et al. [40] proposed a scheduling algorithm to improve load balance between the CPU and GPU. Zhang et al. [41] improved scheduling by matching the irregularity of the tasks and the processor characteristics. Gharaibeh et al. [38] designed a framework to seamlessly use both processors to accelerate graph processing. Our work focuses on only the GPU, but the result shows that Graphie is usually more than 10X faster than the CPU-based system, indicating the small potential of using the CPU besides the GPU.

Researchers have applied different data reorganization techniques to improve the performance of irregular applications for SIMD-based architectures. Wu et al. [42] studied the complexity of data reorganization for optimized GPU memory accesses and proposed several algorithms to strike different trade-offs. Fauzia et al. [43] implemented a tool to automatically characterize uncoalesed memory accesses and transform the data to reduce the degree of divergence. Ren et al. [44] reorganized the tree data structure to improve the performance of CPU vectorization. Jiang et al. [45] studied the reuse of reorganized data for dynamic irregular applications.

VIII. Conclusion

In this paper, we presented Graphie, a GPU-based graph system to perform large-scale graph traversals. Graphie leverages asynchronous edge streaming to stream edge partitions to the GPU to hide data transfer overhead. Different from existing systems with a similar architecture, Graphie improves performance of graph traversal through a novel renaming technique. The renaming process consists of two rounds to enable the convenient use of shared memory and efficient activation of edge partitions, which does not introduce any extra overhead in the GPU memory or in disk. We evaluated Graphie on 7 graphs with up to 1.8 billion edges, and showed that Graphie substantially outperforms X-Stream and GraphReduce.

Acknowledgment

We thank the anonymous reviewers for their insightful comments and suggestions. The effort of this project is funded by National Science Foundation Grants 1464216 and 1618912.

References

[1] 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*, ser. SOSP '13, 2013, pp. 472–488.

[2] 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*, ser. HPDC '14, 2014, pp. 239–252.

[3] K. Vora, G. H. Xu, and R. Gupta, "Load the edges you need: A generic I/O optimization for disk-based graph processing," in *2016 USENIX Annual Technical Conference, USENIX ATC 2016, Denver, CO, USA, June 22-24, 2016.*, 2016, pp. 507–522.

[4] 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*, ser. SIGMOD '10, 2010, pp. 135–146.

[5] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica, "Graphx: Graph processing in a distributed dataflow framework," in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. Broomfield, CO: USENIX Association, 2014, pp. 599–613. [Online]. Available: https://www.usenix.org/conference/osdi14/technical-sessions/presentation/gonzalez

[6] Harshvardhan, A. Fidel, N. M. Amato, and L. Rauchwerger, "Kla: A new algorithmic paradigm for parallel graph computations," in *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, ser. PACT '14, 2014, pp. 27–38.

[7] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "Powergraph: Distributed graph-parallel computation on natural graphs," in *10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, Hollywood, CA, USA, October 8-10, 2012*, 2012, pp. 17–30.

[8] R. Chen, J. Shi, Y. Chen, and H. Chen, "Powerlyra: differentiated graph computation and partitioning on skewed graphs," in *Proceedings of the Tenth European Conference on Computer Systems, EuroSys 2015, Bordeaux, France, April 21-24, 2015*, 2015, pp. 1:1–1:15.

[9] A. Kyrola, G. Blelloch, and C. Guestrin, "Graphchi: Large-scale graph computation on just a pc," in *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'12, 2012, pp. 31–46.

[10] D. Nguyen, A. Lenharth, and K. Pingali, "A lightweight infrastructure for graph analytics," in *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*, 2013, pp. 456–471.

[11] R. Nasre, M. Burtscher, and K. Pingali, "Morph algorithms on gpus," in *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '13, 2013, pp. 147–156.

[12] ——, "Data-driven versus topology-driven irregular computations on gpus," in *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, ser. IPDPS '13, 2013, pp. 463–474.

[13] F. Khorasani, R. Gupta, and L. N. Bhuyan, "Efficient warp execution in presence of divergence with collaborative context collection," in *Proceedings of the 48th International Symposium on Microarchitecture*, ser. MICRO-48, 2015, pp. 204–215.

[14] H. Liu and H. H. Huang, "Enterprise: Breadth-first graph traversal on gpus," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '15, 2015, pp. 68:1–68:12.

[15] Y. Wang, A. Davidson, Y. Pan, Y. Wu, A. Riffel, and J. D. Owens, "Gunrock: A high-performance graph processing library on the gpu," in *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '16, 2016, pp. 11:1–11:12.

[16] M.-S. Kim, K. An, H. Park, H. Seo, and J. Kim, "Gts: A fast and scalable graph processing method based on streaming topology to gpus," in *Proceedings of the 2016 International Conference on Management of Data*, ser. SIGMOD '16, 2016, pp. 447–461.

[17] D. Sengupta, S. L. Song, K. Agarwal, and K. Schwan, "Graphreduce: Processing large-scale graphs on accelerator-based systems," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '15, 2015, pp. 28:1–28:12.

[18] P. Yuan, C. Xie, L. Liu, and H. Jin, "Pathgraph: A path centric graph processing system," *IEEE Trans. Parallel Distrib. Syst.*, vol. 27, no. 10, pp. 2998–3012, 2016.

[19] N. Sundaram, N. Satish, M. M. A. Patwary, S. R. Dulloor, M. J. Anderson, S. G. Vadlamudi, D. Das, and P. Dubey, "Graphmat: High performance graph analytics made productive," *Proc. VLDB Endow.*, vol. 8, no. 11, Jul. 2015.

[20] A. Roy, L. Bindschaedler, J. Malicevic, and W. Zwaenepoel, "Chaos: Scale-out graph processing from secondary storage," in *Proceedings of the 25th Symposium on Operating Systems Principles*, ser. SOSP '15, 2015, pp. 410–424.

[21] 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, OOPSLA 2016, part of SPLASH 2016, Amsterdam, The Netherlands, October 30 - November 4, 2016*, 2016, pp. 1–19.

[22] "Cage15," http://www.ufl.edu/research/sparse/matrices/vanhenkelum.

[23] R. Rossi and N. Ahmed, "The network data repository with interactive graph analytics and visualization." 2015.

[24] O. Schenk, A. Wächter, and M. Weiser, "Inertia-revealing preconditioning for large-scale nonconvex constrained optimization," *SIAM Journal on Scientific Computing*, vol. 31, no. 2, pp. 939–960, 2008.

[25] J. Yang and J. Leskovec, "Defining and evaluating network communities based on ground-truth," *Knowledge and Information Systems*, vol. 42, no. 1, pp. 181–213, 2015.

[26] "uk-2002," http://law.di.unimi.it/webdata/uk-2002/.

[27] H. Kwak, C. Lee, H. Park, and S. Moon, "What is twitter, a social network or a news media?" in *Proceedings of the 19th international conference on World wide web*. ACM, 2010, pp. 591–600.

[28] NVIDIA, "Next generation cuda computer architecture kepler gk110," Tech. Rep., 2012.

[29] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein, "Distributed graphlab: A framework for machine learning in the cloud," *PVLDB*, vol. 5, no. 8, pp. 716–727, 2012.

[30] 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. [Online]. Available: http://dx.doi.org/10.14778/2732232.2732238

[31] J. Shun and G. E. Blelloch, "Ligra: A lightweight graph processing framework for shared memory," in *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '13, 2013, pp. 135–146.

[32] K. Zhang, R. Chen, and H. Chen, "Numa-aware graph-structured analytics," in *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2015, San Francisco, CA, USA, February 7-11, 2015*, 2015, pp. 183–193.

[33] R. Kaleem, A. Venkat, S. Pai, M. W. Hall, and K. Pingali, "Synchronization trade-offs in GPU implementations of graph algorithms," in *2016 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2016, Chicago, IL, USA, May 23-27, 2016*, 2016, pp. 514–523.

[34] 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*, ser. PPoPP '17, 2017, pp. 235–248.

[35] H. Liu, H. H. Huang, and Y. Hu, "ibfs: Concurrent breadth-first search on gpus," in *Proceedings of the 2016 International Conference on Management of Data*, ser. SIGMOD '16, 2016, pp. 403–416.

[36] F. Khorasani, R. Gupta, and L. N. Bhuyan, "Scalable simd-efficient graph processing on gpus," in *2015 International Conference on Parallel Architecture and Compilation, PACT 2015, San Francisco, CA, USA, October 18-21, 2015*, 2015, pp. 39–50.

[37] J. Zhong and B. He, "Medusa: A parallel graph processing system on graphics processors," *SIGMOD Record*, vol. 43, no. 2, pp. 35–40, 2014.

[38] A. Gharaibeh, L. B. Costa, E. Santos-Neto, and M. Ripeanu, "A yoke of oxen and a thousand chickens for heavy lifting graph processing," in *International Conference on Parallel Architectures and Compilation Techniques, PACT '12, Minneapolis, MN, USA - September 19 - 23, 2012*, 2012, pp. 345–354.

[39] D. Merrill, M. Garland, and A. Grimshaw, "Scalable gpu graph traversal," in *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '12, 2012, pp. 117–128.

[40] R. Kaleem, R. Barik, T. Shpeisman, B. T. Lewis, C. Hu, and K. Pingali, "Adaptive heterogeneous scheduling for integrated gpus," in *International Conference on Parallel Architectures and Compilation, PACT '14, Edmonton, AB, Canada, August 24-27, 2014*, 2014, pp. 151–162.

[41] F. Zhang, B. Wu, J. Zhai, B. He, and W. Chen, "Finepar: irregularity-aware fine-grained workload partitioning on integrated architectures," in *Proceedings of the 2017 International Symposium on Code Generation and Optimization, CGO 2017, Austin, TX, USA, February 4-8, 2017*, 2017, pp. 27–38.

[42] B. Wu, Z. Zhao, E. Z. Zhang, Y. Jiang, and X. Shen, "Complexity analysis and algorithm design for reorganizing data to minimize non-coalesced memory accesses on GPU," in *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '13, Shenzhen, China, February 23-27, 2013*, 2013, pp. 57–68.

[43] N. Fauzia, L. Pouchet, and P. Sadayappan, "Characterizing and enhancing global memory data coalescing on gpus," in *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2015, San Francisco, CA, USA, February 07 - 11, 2015*, 2015, pp. 12–22.

[44] B. Ren, G. Agrawal, J. R. Larus, T. Mytkowicz, T. Poutanen, and W. Schulte, "SIMD parallelization of applications that traverse irregular data structures," in *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2013, Shenzhen, China, February 23-27, 2013*, 2013, pp. 20:1–20:10.

[45] P. Jiang, L. Chen, and G. Agrawal, "Reusing data reorganization for efficient SIMD parallelization of adaptive irregular applications," in *Proceedings of the 2016 International Conference on Supercomputing, ICS 2016, Istanbul, Turkey, June 1-3, 2016*, 2016, pp. 16:1–16:10.