Subway: Minimizing Data Transfer during Out-of-GPU-Memory Graph Processing

Amir Hossein Nodehi Sabet
anode001@ucr.edu
University of California, Riverside

Zhijia Zhao
zhijia@cs.ucr.edu
University of California, Riverside

Rajiv Gupta
gupta@cs.ucr.edu
University of California, Riverside

Abstract

In many graph-based applications, the graphs tend to grow, imposing a great challenge for GPU-based graph processing. When the graph size exceeds the device memory capacity (i.e., GPU memory oversubscription), the performance of graph processing often degrades dramatically, due to the sheer amount of data transfer between CPU and GPU.

To reduce the volume of data transfer, existing approaches track the activeness of graph partitions and only load the ones that need to be processed. In fact, the recent advances of unified memory implements this optimization implicitly by loading memory pages on demand. However, either way, the benefits are limited by the coarse-granularity activeness tracking – each loaded partition or memory page may still carry a large ratio of inactive edges.

In this work, we present, to the best of our knowledge, the first solution that only loads active edges of the graph to the GPU memory. To achieve this, we design a fast subgraph generation algorithm with a simple yet efficient subgraph representation and a GPU-accelerated implementation. They allow the subgraph generation to be applied in almost every iteration of the vertex-centric graph processing. Furthermore, we bring asynchrony to the subgraph processing, delaying the synchronization between a subgraph in the GPU memory and the rest of the graph in the CPU memory. This can safely reduce the needs of generating and loading subgraphs for many common graph algorithms. Our prototyped system, **Subway** (subgraph processing with asynchrony) yields over 4X speedup on average comparing with existing out-of-GPU-memory solutions and the unified memory-based approach, based on an evaluation with six common graph algorithms.

ACM Reference Format:
Amir Hossein Nodehi Sabet, Zhijia Zhao, and Rajiv Gupta. 2020. Subway: Minimizing Data Transfer during Out-of-GPU-Memory Graph Processing. In *Fifteenth European Conference on Computer Systems (EuroSys '20), April 27–30, 2020, Heraklion, Greece.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3342195.3387537

1 Introduction

In many graph-based applications, graphs naturally grow over time. Considering web analytics [10, 47], the sizes of web graphs quickly increase as more webpages are crawled. In social networks [42, 54] and online stores [51], graphs related to user interactions (e.g., following and liking) and product purchases (e.g., who bought what item) also grow consistently. These growing graphs pose special challenges to GPU-based graph processing systems. When the size of an input graph exceeds the GPU memory capacity, known as *memory oversubscription*, existing GPU-based graph systems either fail to process (such as CuSha [32], Gunrock [67], and Tigr [45]), or process it with dramatic slowdown [21, 55].

State of The Art. Existing efforts [21, 33, 55] in addressing the GPU memory oversubscription for graph applications mainly follow ideas in out-of-core graph processing, such as GraphChi [34], X-Stream [52], and GridGraph [74]. Basically, the oversized graph is first partitioned, then explicitly loaded to the GPU memory in each iteration of the graph processing. Hereinafter, we refer to this explicit memory management as the *partitioning-based approach*. A major challenge for this approach is the low computation to data transfer ratio due to the nature of iterative graph processing. As a result, the data movement cost usually dominates the execution time, as shown in Figure 1. For this reason, one focus in optimizing this approach is *asynchronously* streaming the graph data to the GPU memory to overlap the data transfer with the GPU kernel executions [21, 33, 55]. Unfortunately, as Figure 1 indicates, for many graph applications, the computation cost is substantially smaller than the data transfer cost. Moreover, it varies significantly over iterations. Both factors limit the benefits of this overlapping. A more promising direction is to directly reduce the data movements between CPU and GPU. To achieve this, GraphReduce [55] and Graphie [21] track the partitions with to-be-processed (active) vertices/edges and only load those to the GPU memory, which have shown promises in reducing data movements. However, the benefits of this activeness checking are limited to the partition level. For example, a partition with few active edges would still be loaded entirely. Moreover, for real-world power-law graphs,

EuroSys '20, April 27–30, 2020, Heraklion, Greece
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-6882-7/20/04…$15.00
https://doi.org/10.1145/3342195.3387537

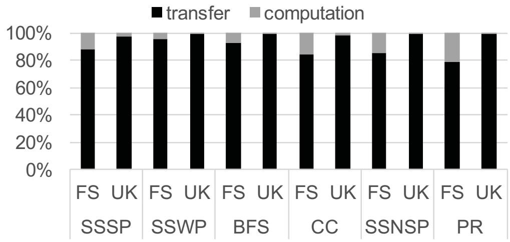


Figure 1. Time Breakdown of Partitioning-based Approach (six analytics on two real-world graphs from Section 4).

most partitions may stay active due to their connections to some high-degree vertices, further limiting the benefits.

As a more general solution, *unified memory* recently has become available with the release of CUDA 8.0 and the Pascal architecture (2016) [1]. It allows GPU applications to access the host memory transparently with memory pages migrated on demand. By adopting unified memory, graph systems do not have to track the activeness of graph partitions, instead, the memory pages containing active edges/vertices will be automatically loaded to the GPU memory triggered by the page faults. Despite the ease of programming and on-demand "partition" loading, unified memory-based graph systems suffer from two limitations: First, on-demand page faulting is not free. As shown later, there are significant overheads with page fault handling (such as TLB invalidations and page table updates); Second, similar to the explicit graph partition activeness tracking, the loaded memory pages may also contain a large ratio of inactive edges/vertices, wasting the CPU-GPU data transfer bandwidth.

Unlike prior efforts and unified memory-based approach, this work aims to load only the active edges (and vertices), which are essentially a subgraph, to the GPU memory. The challenges lie in the costs. Note that the subgraph changes in every iteration of the graph processing. The conventional wisdom is that repetitively generating subgraphs at runtime is often too expensive to be beneficial [63]. Contrary to the wisdom, we show that, with the introduction of (i) a concise yet efficient subgraph representation, called SubCSR; (ii) a highly parallel subgraph generation algorithm; and (iii) a GPU-accelerated implementation, the cost of the subgraph generation can be low enough that it would be beneficial to apply in (almost) every iteration of the graph processing.

On top of subgraph generation, we bring asynchrony to the in-GPU-memory subgraph processing scheme. The basic idea is to delay the synchronization between a subgraph (in GPU memory) and the rest of the graph (in host memory). After the subgraph is loaded to GPU memory, its vertex values will be propagated asynchronously (to the rest of the graph) until they have reached a local convergence before the next subgraph is generated and loaded. In general, this changes the behavior of value convergences and may not be applicable to every graph algorithm. However, as we will discuss later, it can be safely applied to a wide range of common graph algorithms. In practice, the asynchrony tends to reduce the number of iterations and consequently the number of times a subgraph must be generated and loaded.

Together, the proposed techniques can reduce both the number of times the input graph is loaded and the size of loaded graph each time. We prototyped these techniques into a runtime system called **Subway**. By design, Subway can be naturally integrated into vertex-centric graph processing systems with the standard CSR graph representation. Our evaluation with six commonly used graph applications on a set of real-world and synthesized graphs shows that Subway can significantly improve the efficiency of GPU-based graph processing under memory oversubscription, yielding 5.6X and 4.3X speedups comparing to the unified memory-based approach and the existing techniques, respectively.

Contributions. In summary, this work makes three major contributions to GPU-based graph processing:

- First, this work presents a highly efficient subgraph generation technique, which can quickly and (often) significantly "shrink" the size of the loaded graph in each graph processing iteration.
- Second, it brings asynchrony to the in-GPU-memory subgraph processing, making it possible to reduce the times of subgraph generations and loading.
- Third, it compares the proposed techniques (Subway) with existing out-of-GPU-memory graph processing solutions, unified memory-based graph processing, as well as some of the state-of -the-art CPU-based graph systems. The source code of Subway is available at: https://github.com/AutomataLab/Subway.

Next, we first provide the background of this work.

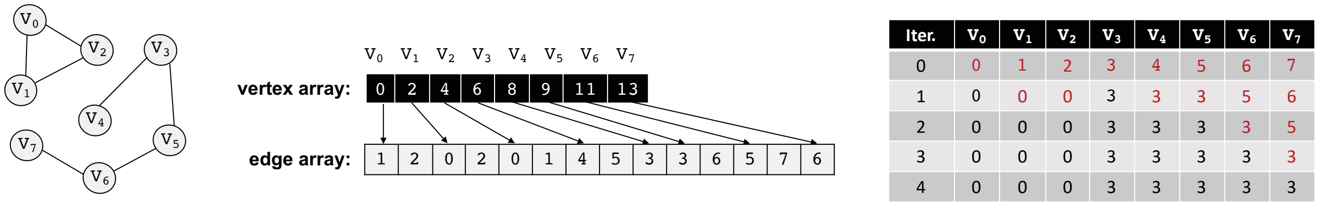
2 Background

This section first introduces the basics of graph applications and their programming model, including a discussion of the major issues in GPU-based graph processing.

2.1 Graph Applications and Programming Model

As a basic yet versatile data structure, graphs are commonly used in a wide range of applications to capture their linked data and to reveal knowledge at a deeper level, ranging from influence analysis in social networks [43] and fraud detection in bank transactions [53] to supply chain distribution [64] and product recommendations [26]. As a sequence, there have been consistent interests in developing graph processing systems, covering shared-memory graph processing (e.g., Galois [44] and Ligra [56]), distributed graph processing (e.g., Pregel [38] and Distributed GraphLab [36]), out-of-core graph processing (such as GraphChi [34] and X-Stream [52]),

[1] CUDA 6.0 supports automatic data migration between CPU and GPU memories, but does not support GPU memory oversubscription.



	V0	V1	V2	V3	V4	V5	V6	V7
vertex array:	0	2	4	6	8	9	11	13

edge array:	1	2	0	2	0	1	4	5	3	3	6	5	7	6

Iter.	V0	V1	V2	V3	V4	V5	V6	V7
0	0	1	2	3	4	5	6	7
1	0	0	0	3	3	3	5	6
2	0	0	0	3	3	3	3	5
3	0	0	0	3	3	3	3	3
4	0	0	0	3	3	3	3	3

(a) Graph (b) CSR Representation (c) Iterations

Figure 2. Graph Representation and Vertex-Centric Graph Processing (Connected Components).

and GPU-accelerated graph processing (such as CuSha [32] and Gunrock [67]). More details about these graph systems and others will be shown later in Section 5.

To support graph application developments, *vertex-centric programming* [38] has been widely adopted as a popular graph programming model, thanks to its simplicity, high scalability, and strong expressiveness. It defines a generic vertex function $f(\cdot)$ based on the values of neighbor vertices. During the processing, the vertex function is evaluated on all (active) vertices iteratively until all vertex values stop changing or the iterations have reached a limit. Depending on the value propagation direction, the vertex function can be either *pull*-based (gathering values along in-coming edges) or *push*-based (scattering values along out-going edges) [44].

Algorithm 1 illustrates the push-based vertex function for connected components (CC), where the graph is in CSR (Compressed Sparse Row) format, a commonly used graph representation that captures the graph topology with two arrays: *vertex array* and *edge array*. As shown in Figure 2-(b), the vertex array is made up of indexes to the edge array for locating the edges of the corresponding vertices (Line 8-9 in Algorithm 1). In addition, the vertex function also accesses *activeness labeling array* to check vertex activeness (Line 4) and update its neighbors' (Line 12), as well as the *value array* for reading and updating the connected component IDs (Line 7 and 10-11). Its iterations on the example graph are shown in Figure 2-(c). Initially, all vertices are active with connected component IDs the same as their vertex IDs. After three iterations, the vertices fall into two connected components, labelled with the smallest vertex IDs.

2.2 GPU-based Graph Processing

With the abundant parallelism exposed by the vertex-centric programming, GPUs built with up to thousands of cores have great potential in accelerating graph applications [9, 22, 25, 32, 41, 45, 67, 71]. Despite this promise, two main challenges arise in GPU-based graph processing: the *highly irregular graph structures* and the *ever-increasing graph sizes*.

The graph irregularity causes non-coalesced accesses to the GPU memory and load imbalances among GPU threads. Existing research on GPU-based graph processing, including CuSha [32], Gunrock [67], and Tigr [45], mainly focus on addressing the graph irregularity problem and have brought significant efficiency improvements. For example, CuSha brings new data structures (i.e., G-Shards and Concatenated Windows) to enable fully coalesced memory accesses. In compaison, Gunrock designs a frontier-based abstraction which allows easy integrations of multiple optimization strategies. More directly, Tigr proposes to transform the irregular graphs into more regular ones.

Algorithm 1 Vertex Function (CC) on CSR.

```
1:  /* Connected Components */
2:  procedure CC
3:      tid = getThreadID()
4:      if isActive[tid] == 0 then
5:          return
6:      end if
7:      sourceValue = value[tid]
8:      for i = vertex[tid] : vertex[tid+1] do
9:          nbr = edge[i]
10:         if sourceValue < value[nbr] then
11:             atomicMin(value[nbr], sourceValue)
12:             isActiveNew[nbr] = 1      /* active in next iter. */
13:         end if
14:     end for
15: end procedure
```

However, big gap remains in efforts towards the second challenge - processing oversized graphs. Despite that GPU memory capacity has been increasing, it is still too limited to accommodate many real-world graphs [21, 55]. There are two basic strategies to handle such cases: (i) *partitioning-based approach* and (ii) *unified memory-based approach.*

***Partitioning-based Approach*.** This approach follows the ideas in out-of-core graph processing [34, 52, 74] – it first partitions the oversized graph such that each partition can fit into the GPU memory, then loads the graph partitions into the GPU memory in a round-robin fashion during the iterative graph processing. Most existing solutions adopt this strategy, including GraphReduce [55], GTS [33], and Graphie [21]. To improve the processing efficiency under GPU memory oversubscription, two main optimizations have been proposed: the first one tries to hide some data transfer cost by asynchronously streaming the graph partitions to the GPU memory while the kernels are executing [21, 33, 55]; The second optimization tracks the activeness of each graph partition and only loads the ones with active edges (i.e., active partitions) to the GPU memory [21, 55]. Unfortunately, as discussed in Section 1, the benefits of both optimizations are limited by the sparsity nature of iterative graph processing – in most iterations, only a small subset of vertices tend to be active (a.k.a the frontier). First, the sparsity results in low computation-to-transfer ratio, capping the benefits of overlapping optimization. Second, in each loaded active partition, there might still be a large portion of inactive edges (i.e., their vertices are inactive) due to the sparsity.

Unified Memory-based Approach. Rather than explicitly managing the data movements, a more general solution is adopting unified memory [2], a technique has been fully realized in recent Nvidia GPUs. The main idea of unified memory is defining a managed memory space in which both CPU and GPU can observe a single address space with a coherent memory image [2]. This allows GPU programs to access data in the CPU memory without explicit memory copying. A related concept is *zero-copy memory* [1], which maps pinned (i.e., non-pageable) host memory to the GPU address space, also allowing GPU programs to directly access the host memory. However, a key difference is that, in unified memory, the memory pages containing the requested data are automatically *migrated* to the memory of the requesting processor (either CPU or GPU), known as *on-demand data migration*, enabling faster accesses for the future requests.

Adopting the unified memory for graph applications is straightforward: when allocating memory for the input graph, use a new API `cudaMallocManaged()`, instead of the default `malloc()`. In this way, when the graph is accessed by GPU threads and the data is not in the GPU memory, a page fault is triggered and the memory page containing the data (active edges) is migrated to the GPU memory. On one hand, this implicitly avoids loading memory pages with only inactive edges, sharing similar benefits with the partition activeness-tracking optimization in the partitioning-based approach. On the other hand, it also suffers from a similar limitation – loaded memory pages may contain a large ratio of inactive edges. Moreover, as we will show later, the page fault handling introduces substantial overhead, compromising the benefits of on-demand data migration.

In addition, unified memory can be tuned via APIs such as `cudaMemAdvise()` and `cudaMemPrefetchAsync()` [2] for better data placements and more efficient data transfer (more details will be given in Section 4). In addition, a recent work, ETC [35], also proposes some general optimization strategies, such as proactive eviction, thread throttling, and capacity compression. However, as we will show in Section 4, such general optimizations either fail to improve the performance or bring limited benefits, due to their inability in eliminating the expensive data transfer of inactive edges in each iteration of the graph processing.

Table 1. Ratio of Active Vertices and Edges

V-avg (max): average ratio of active vertices (maximum ratio);
E-avg (max): average ratio of active edges (maximum ratio).

	friendster-snap [6]		uk-2007 [4]	
Algo.	V-avg (max)	E-avg (max)	V-avg (max)	E-avg (max)
SSSP	4.4% (43.3%)	9.1% (85.1%)	4.6% (60.4%)	5.1% (67.7%)
SSWP	2.1% (38.4%)	5.2% (78.3%)	0.6% (12.6%)	0.6% (12.4%)
BFS	2.1% (32.3%)	4.1% (75.8%)	0.6% (12.6%)	0.6% (12.4%)
CC	8.1% (100%)	9.8% (100%)	3.2% (100%)	3.2% (100%)
SSNSP	2.1% (32.3%)	4.1% (75.8%)	0.6% (12.6%)	0.6% (12.4%)
PR	6.6% (100%)	24.1% (100%)	1.1% (100%)	1.7% (100%)

In summary, both the existing partitioning and unified memory-based methods load the graph based on coarse-grained activeness tracking, fundamentally limiting their performance benefits. Next, we present Subway, a low-cost CPU-GPU graph data transfer solution based on fine-grained activeness tracking.

3 Subway

The core to Subway is a *fast subgraph generation* technique which can quickly extract a subgraph from a standard CSR formatted graph based on the specified vertices. When fed with the active vertices (i.e., the frontier), this technique can "shrink" the original graph such that only a subgraph needs to be loaded to the GPU memory. In addition, Subway offers an *in-GPU-memory asynchronous subgraph processing* scheme, which can reduce the needs of subgraph generation and loading. Together, these techniques can significantly bring down the cost of CPU-GPU data transfer, enabling efficient out-of-memory graph processing on GPUs. Next, we present these techniques in detail.

3.1 Fast Subgraph Generation

For many common graph algorithms, it is often that a subset of vertices are active (need to be processed) in an iteration of the graph processing. In most iterations, the ratio of active vertices is often very low, so as to the ratio of active edges. Table 1 reports the average and maximum ratios of active vertices and edges across all iterations, collected from six graph algorithms on two real-world graphs. In these tested cases, the average ratios of active vertices and active edges are always below 10%. Motivated by this fact, we explore the possibility of separating the active parts of the graph from the rest and only load those to the GPU memory. This can greatly improve the state of the art [21, 55] which is only able to separate the inactive partitions. Despite the promise, the key challenge with this fine-grained separation is the cost. Recent work [63] shows that dynamically restructuring a graph could be very expensive in the out-of-core graph processing. Fortunately, in GPU-based graph processing, we can always leverage the power of GPU to accelerate this subgraph generation. Along with a concise design of the

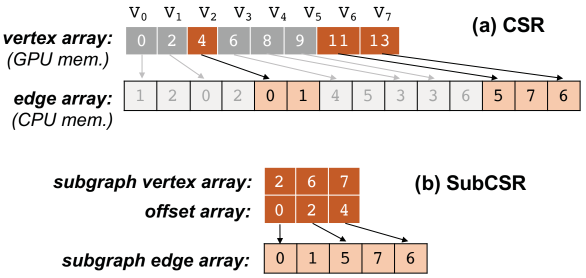


Figure 3. SubCSR Representation.

subgraph representation and a highly parallel generation algorithm, we find that the cost of subgraph generation can be quite affordable in comparison to the benefits that it brings. Next, we first introduce the subgraph representation, then present its generation algorithm.

Subgraph Representation. Subway assumes that the input graph is in CSR format [2], the commonly used graph format in GPU-based graph processing (see Section 2). Also, following a prior assumption [21], Subway allocates the vertex array (and its values) in the GPU memory and the edge array (usually dominating the graph size) in the host memory, as shown in Figure 3-(a). This design completely avoids writing data back to host memory at the expense of less GPU memory for storing the graph edges. Recall that the task is to separate the active vertices and edges (i.e., a subgraph) from the rest of the graph, while satisfying two objectives:

- First, the representation of the separated subgraph should remain concise, just like the CSR format;
- Second, the vertices and edges of the subgraph can be efficiently accessed during the graph processing.

To achieve these goals, we introduce SubCSR – a format for representing a subset of vertices along with their edges in a CSR-represented graph. Figure 3-(b) shows the SubCSR for the active vertices and edges of the CSR in Figure 3-(a). At the high level, it looks close to the CSR representation. The main difference is that the vertex array in the CSR is replaced with two arrays: *subgraph vertex array* and *offset array*. The former indicates the positions of active vertices in the original vertex array, while the latter points to the starting positions of their edges in the *subgraph edge array*. In the example, the active vertices are V_2, V_6, and V_7, hence their indices (2, 6, and 7) are placed in the subgraph vertex array. The subgraph edge array in the SubCSR consists only the edges of selected vertices. Obviously, the size of SubCSR is linear to the number of selected vertices plus their edges.

To demonstrate the access efficiency of SubCSR, we next illustrate how SubCSR is used in the (sub)graph processing. Algorithm 2 shows how a GPU thread processes the subgraph after it is loaded to the GPU memory. Comparing it with

Algorithm 2 Vertex Function (CC) on SubCSR.

```
1:  /* Connected Components */
2:  procedure CC
3:      tid = getThreadID()
4:      vid = subVertex[tid] /* difference: an extra array access */
5:      if isActive[vid] == 0 then
6:          return
7:      end if
8:      sourceValue = value[vid]
9:      for i = offset[tid] : offset[tid+1] do
10:         nbr = subEdge[i]
11:         if sourceValue < value[nbr] then
12:             atomicMin(value[nbr], sourceValue)
13:             isActiveNew[nbr] = 1
14:         end if
15:     end for
16: end procedure
```

Algorithm 1, except that `vertex[]` and `edge[]` are replaced with `offset[]` and `subEdge[]`, the only difference is at Line 4, an extra access to the subgraph vertex array `subVertex[]`. This extra array access may slightly increase the cost of vertex evaluation. However, as shown in the evaluation, this minimum increase of computation can be easily outweighed by the significant benefits of subgraph generation.

Next, we describe how to generate the SubCSR efficiently for a given set of active vertices with the help of GPU.

Generation Algorithm. Algorithm 3 and Figure 4 illustrate the basic ideas of GPU-accelerated SubCSR generation. The inputs to the algorithm include the CSR (i.e., `vertex[]` and `edge[]`), the vertex activeness labeling array `isActive[]` [3], and the degree array `degree[]` (can also be generated from CSR). The output of the algorithm is the SubCSR for active vertices (i.e., `subVertex[]`, `offset[]`, and `subEdge[]`). At the high level, the generation follows six steps.

Step-1: Find the indices of active vertices `subIndex[]` using an exclusive prefix sum over the vertex activeness labeling array `isActive[]` (Line 2). Assuming the active vertices are V_2, V_6, and V_7, this step puts the IDs of active vertices into corresponding positions of `subIndex[]` (see Figure 4).

Step-2: Create the subgraph vertex array `subVertex[]` with a stream compaction based on `subIndex[]`, `isActive[]`, and the array index `tid` (Line 3, 11-18). If a vertex is active, put its ID (such as 2, 6, or 7 in the example) into `subVertex[]`.

Step-3: Based on the degree array of CSR `degree[]` (assume it is available or has been generated), create a degree array `subDegree[]` for the active vertices, where the degrees of inactive vertices are reset to zeros (Line 4, 20-29).

Step-4: Compute the offsets of active vertices `subOffset[]` with an exclusive prefix sum over `subDegree[]` (Line 5). In

[2] Note that using edge list, another common graph format, does not simplify the subgraph generation, but increases the memory consumption.

[3] Note that using a labeling array of the same length as `edge[]` may simplify the subgraph generation, but this large labeling array may not fit into the GPU memory, thus making its maintenance very expensive.

Algorithm 3 SubCSR Generation.

```
1:  procedure genSubCSR(vertex[], edge[], isActive[], degree[])
2:      subIndex[] = gpuExclusivePrefixSum(isActive[])
3:      subVertex[] = gpuSC1(isActive[], subIndex[])
4:      subDegree[] = gpuResetInactive(isActive[], degree[])
5:      subOffset[] = gpuExclusivePrefixSum(subDegree[])
6:      offset[] = gpuSC2(isActive[], subIndex[], subOffset[])
7:      subEdge = cpuSC(vertex[], edge[], subVertex[], offset[])
8:      return subVertex[], offset[], subEdge[]
9:  end procedure
10:
11: /* GPU stream compact vertex indices */
12: procedure gpuSC1(isActive[], subIndex[])
13:     tid = getThreadID()
14:     if isActive[tid] == 1 then
15:         subVertex[subIndex[tid]] = tid
16:     end if
17:     return subVertex[]
18: end procedure
19: /* GPU reset degrees of inactive vertices */
20: procedure gpuResetInactive(isActive[], degree[])
21:     tid = getThreadID()
22:     if isActive[tid] == 0 then
23:         subDegree[tid] = 0
24:     else
25:         subDegree[tid] = degree[tid]
26:     end if
27:     return subDegree[]
28: end procedure
29: /* GPU stream compact offset array */
30: procedure gpuSC2(isActive[], subIndex[], subOffset[])
31:     tid = getThreadID()
32:     if isActive[tid] == 1 then
33:         offset[subIndex[tid]] = subOffset[tid]
34:     end if
35:     return offset[]
36: end procedure
37: /* CPU stream compact edge array */
38: procedure cpuSC(vertex[], edge[], subVertex[], offset[])
39:     parallel for $i$ = 0 to numActiveVertices do
40:         v = subVertex[i]
41:         subEdge[offset[i]:offset[i+1]]
42:         = edge[vertex[v]:vertex[v+1]]
43:     end parallel for
44:     return subEdge[]
45: end procedure
```

the example, the offset of the first active vertex (V_2) is always zero, and since V_2 has two edges (see `degree[]`), the second active vertex (V_6) has offset two. Similarly, V_7 has an offset of four. Note that this step depends on the reset in Step-3.

Step-5: Compact `subOffset[]` into `offset[]` by removing the elements of inactive vertices (Line 6, 31-38). Note that this step needs not only `isActive[]`, but also `subIndex[]`.

Step-6: Finally, compact the edge array `edge[]` to `subEdge[]` by removing all inactive edges (Line 7, 40-48). This requires to access the CSR as well as `subVertex[]` and `offset[]`. Basically, edges of active vertices are copied from `edge[]` to `subEdge[]`. Note that `offset[]` is critical here in deciding the target positions of the copying.

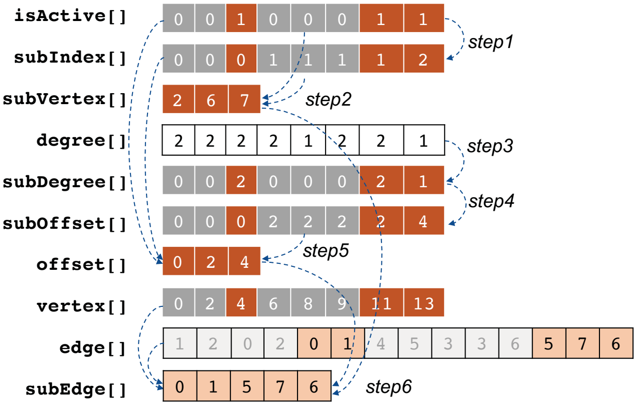

isActive[] 0 0 1 0 0 0 1 1
subIndex[] 0 0 0 1 1 1 1 2
subVertex[] 2 6 7
degree[] 2 2 2 2 1 2 2 1
subDegree[] 0 0 2 0 0 0 2 1
subOffset[] 0 0 0 2 2 2 2 4
offset[] 0 2 4
vertex[] 0 2 4 6 8 9 11 13
edge[] 1 2 0 2 0 1 4 5 3 3 6 5 7 6
subEdge[] 0 1 5 7 6
step1
step2
step3
step4
step5
step6

Figure 4. SubCSR Generation for Example in Figure 3.

Though there are six steps, each step only involves a few simple operations. More importantly, all the six steps are highly parallel, making it straightforward to take advantage of the massive parallelism of GPU. More specifically, as the comments in Algorithm 3 indicate, the first five steps are offloaded to the GPU, where the activeness labeling array `isActive[]` is maintained. After the fifth step, two arrays (`subVertex[]` and `offset[]`) are transferred back to the host memory, where the sixth step is performed. At the end of the generation, the SubCSR for the active vertices are formed in the host memory and are ready to be loaded to the GPU memory.

Cost-Benefit Analysis. In Algorithm 3, the first five steps have a time complexity of O(|V|), where V is the vertex set; while the last step takes O($|E_{active}|$), where E_{active} is the set of active edges. Hence, the overall time complexity of SubCSR generation is O(|V| + $|E_{active}|$). Since set V is fixed, the cost of SubCSR generation varies depending on the amount of active edges. On the other hand, the benefit of SubCSR generation comes from the reduced data transfer: instead of loading the whole edge array (with a size of |E|, where E is the edge set) to the GPU memory, only the SubCSR of active vertices is loaded, with a size of $|E_{active}| + 2 * |V_{active}|$, where V_{active} is the set of active vertices. Note that, there is also data transfer during the SubCSR generation (between Step-5 and Step-6), with a size of $2 * |V_{active}|$. Therefore, the total saving of data transfer is $|E| - |E_{active}| - 4 * |V_{active}|$. Assuming the CPU-GPU data transfer rate is r_{trans}, then the time saving $S_{trans} = r_{trans} * (|E| - |E_{active}| - 4 * |V_{active}|)$.

Assuming the concrete cost of SubCSR generation is C_{gen}, that is, C_{gen} = O(|V| + $|E_{active}|$), then, theoretically speaking, if $C_{gen} < S_{trans}$, applying SubCSR would bring net benefit to the out-of-memory graph processing on GPUs. In practice, we can set a threshold for enabling SubCSR generation based on a simpler metric – the *ratio of active edges*, denoted as P_{active}. As P_{active} increases, the cost of SubCSR generation increases, but the benefit decreases. Hence, there is a sweet spot beyond which the SubCSR generation will not bring any net benefit (i.e., the threshold). Based on our evaluation (Section 4), we found P_{active} = 80% is a threshold that works well in general for tested applications and graphs. In fact, for most tested cases, we found P_{active} is below (often well below) 80% for almost all iterations of the graph processing, making the SubCSR generation applicable across (almost) all iterations. When P_{active} is beyond 80%, SubCSR generation would be disabled and the conventional partitioning-based approach would be employed as a substitution.

Oversized SubCSR Handling. Though P_{active} is usually low enough that the generated SubCSR can easily fit into the GPU memory, there are situations where the SubCSR remains oversized. To handle such cases, the conventional partitioning-based approach can be adopted. Basically, an oversized SubCSR can be partitioned such that each partition can fit into the GPU memory, then the SubCSR partitions are loaded into GPU memory and processed one after another. The partitioning of SubCSR is similar to the partitioning of the original graph (CSR): logically partition the subgraph vertex array `subVertex[]` such that each vertex array chunk, along with its offset array `offset[]` and subgraph edge array `subEdge[]`, are close to but smaller than the available GPU memory size. Since logical partitioning is free and the total cost of SubCSR loading remains the same, handling oversized SubCSR keeps the benefits of SubCSR generation.

3.2 Asynchronous Subgraph Processing

Traditionally, there are two basic approaches to evaluate the vertex function: the *synchronous approach* [38] and the *asynchronous approach* [34, 36]. The former only allows the vertices to synchronize at the end of each iteration, that is, all (active) vertices read values computed from the last iteration. This design strictly follows the bulk-synchronous parallel (BSP) model [62]. By contrast, the asynchronous approach allows the vertex function to use the *latest values*, which may be generated in the current iteration (intra-iteration asynchrony). In both schemes, a vertex is only *evaluated once per iteration*. This simplifies the development of graph algorithms, but results in a low *computation to data transfer ratio*, making the data transfer a serious bottleneck under GPU memory oversubscription. To overcome this obstacle, Subway offers a more flexible vertex function evaluation strategy – *asynchronous subgraph processing*.

Asynchronous Model. Under this model, after a subgraph (or a subgraph partition) is loaded into the GPU memory, it will be processed asynchronously with respect to the rest of the graph in the host memory. Algorithm 4 illustrates its basic idea. Each time when a subgraph partition, say P_i, is loaded to the GPU memory (Line 6), the vertices in P_i are iteratively evaluated until there is no active ones in P_i (Line 7-11). This adds a second level of iteration inside the whole-graph level iteration (Line 1-13). The outer level iteration ensures that the algorithm will finally reach a global convergence and terminate, while the inner level iteration maximizes the value propagations in each loaded subgraph partition. Since the inner iteration makes the vertex values "more stable" – closer to their eventual values, the outer level iterations tend to converge faster. Thus, there will be less need for generating and loading subgraphs (Line 3 and 6). Next, we illustrate this idea with the example in Figure 5.

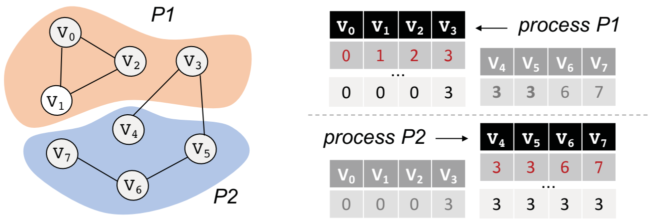
Algorithm 4 Asynchronous Subgraph Processing

```
1:  do /* whole-graph-level iteration */
2:      $V_{active}$ = getActiveVertices(G)
3:      $G_{sub}$ = genSubCSR(G, $V_{active}$)
4:      /* the subgraph may be oversized, thus partitioned */
5:      for $P_i$ in partitions of subgraph $G_{sub}$ do
6:          load $P_i$ to GPU memory
7:          do /* partition-of-subgraph-level iteration */
8:              for $v_i$ in vertices of $P_i$ do
9:                  $f(v_i)$ /* evaluate vertex function */
10:             end for
11:         while anyActiveVertices($P_i$) == 1
12:     end for
13: while anyActiveVertices(G) == 1
```

Example. Initially, all the vertices are active (also refer to Figure 2-(c)). Assume that their edges cannot fit into the GPU memory, hence the (sub)graph is partitioned into two parts: $P1$ and $P2$, as shown in Figure 5-(a). First, $P1$ is loaded to the GPU memory and processed iteratively until their values are (locally) converged. Note that, at this moment, the values of V_4 and V_5 have been updated (under a push-based scheme). After that, $P2$ is loaded to the GPU memory and processed in the same way. Since vertices in $P2$ observe the latest values (3 for V_4 and V_5), they can converge to "more stable" values, in this case, their final values. The whole processing takes only one outer iteration to finish, comparing to three iterations in the conventional processing (see Figure 2-(c)).

Note that choosing the above asynchronous model does not necessarily reduce the total amount (GPU) computations, which include both the inner and outer iterations. According to our experiments, the change to the total computations fluctuates slightly case by case (see Section 4). On the other hand, the saving from reduced subgraph generation and loading is significant and more consistent. As shown in the above example, it only needs to load the (sub)graph into the GPU memory once, rather than three times.

Related Ideas. One closely related processing scheme is the TLAG model ("*think like a graph*") proposed in distributed graph processing [61]. In this case, a large input graph is partitioned and stored on different computers connected by the network. During the iterative processing, different



(a) Partitioned Graph

(b) Iterations (one outer iteration)

Figure 5. Example under Asynchronous Model.

computers send messages to each other to synchronize their values. To reduce the amount of messages generated in the distributed system, TLAG lifts the programming abstraction from vertex level to partition level. In the new abstraction, graph programming becomes partition-aware, permitting vertex values to be freely propagated within each partition. As the vertex values may become "more stable" when they are propagated to the other partitions, the communications across partitions tend to be reduced, so as to the messages generated in the distributed systems.

Both TLAG and asynchronous subgraph processing extend the conventional intra-iteration asynchrony to some kind of inter-partition asynchrony. However, the key difference is that, in TLAG, all partitions are processed simultaneously, while in asynchronous subgraph processing, the partitions are processed in serial, one partition at a time. The difference makes the latter "more asynchronous" than the former: in a partition-by-partition processing, a later loaded partition can directly observe the latest values computed from earlier partitions, while in TLAG, the latest values from the other partitions are not exposed until all partitions have reached their local convergences. Due to this reason, asynchronous subgraph processing may converge even faster than TLAG. For example, it takes one more outer iteration for the example in Figure 5 to converge when implemented in TLAG.

Correctness. The use of asynchronous subgraph processing may alter the value propagation priority (i.e., preferring to evaluate vertices in the "current" partition), similar to TLAG. Therefore, it may not be suitable for every graph algorithm, especially those that are sensitive to the value propagation order. For example, prior work [61] has shown that TLAG is applicable to three major categories of graph algorithms: *graph traversal*, *random walk*, and *graph aggregation*. But, for some other algorithms (e.g., PageRank), it requires some changes to the algorithm design. As a general sufficient (but not necessary) condition, as long as the final vertex values do not depend on the vertex evaluation order, it is ensured that the asynchronous subgraph processing will preserve the convergence and the converged values. It is not hard to manually verify that the commonly used graph traversal algorithms, such as SSSP (single-source shortest path), BFS (breath-first search), SSWP (single-source widest path), and CC (connected components), all satisfy the above condition. In addition, after adopting the accumulative update-based algorithm [70], PageRank can also satisfy this condition, thus runs safely under asynchronous subgraph processing. The correctness of these graph algorithms has also been verified by our extensive experiments (see Section 4).

Note that, automatically reasoning about the necessary correctness conditions for a specific graph algorithm under asynchronous subgraph processing is a challenging research problem not covered by this work. For this reason, Subway provides asynchronous subgraph processing as an optional feature, enabled only when the correctness has been verified.

3.3 Implementation

We prototyped Subway as a runtime system with both the fast subgraph generation technique (Section 3.1) and the asynchronous subgraph processing (Section 3.2). In order to demonstrate the effectiveness of Subway, we integrated it into an existing open-source GPU-based graph processing framework, called Tigr [4] [45]. In fact, Subway uses Tigr for in-memory graph processing when the input graph fits into GPU memory. Another reason we choose Tigr is for its use of the standard vertex-centric programming and the CSR graph representation, which make it immediately ready to adopt Subway. For simplicity, we refer to this integrated graph system as Subway when the reference context is clear. By default, the fast subgraph generation is enabled when the ratio of active vertices in the current iteration is beyond the threshold 80%. By contrast, the asynchronous subgraph processing scheme is disabled by default for the correctness reason mentioned earlier. In the implementation of fast subgraph generation, we used the `Thrust` library [7] for the exclusive prefix sum in the first and fourth steps of the SubCSR generation, and the `Pthread` library for the parallel edge array compaction in the sixth step (see Section 3.1).

In-Memory Processing. Subway automatically detects the size of the input graph, when the graph fits into the GPU memory, it switches to the in-memory processing mode (i.e., Tigr). The performance of Subway would be the same as Tigr, which has been compared to other well-known GPU-based frameworks such as Gunrock [67] and CuSha [32], showing promising results [45]. On the other hand, when the graph cannot fit into the GPU memory, optimizations from Tigr would be disabled and the system mainly relies on Subway runtime (subgraph generation and asynchronous processing if applicable) for performance optimizations.

4 Evaluation

In this section, we evaluate the prototyped system Subway under the scenarios of GPU memory oversubscription, with an emphasis on the cost of CPU-GPU data movements and the overall graph processing efficiency.

[4] https://github.com/AutomataLab/Tigr

4.1 Methodology

Our evaluation includes the following approaches:

- *Basic Partitioning-based Approach* (`PT`): This one follows the basic ideas of partitioning-based memory management without further optimizations. It logically partitions the graph based on the vertex array, then loads the partitions into the GPU memory one by one during each iteration of the graph processing. We include this approach to make the benefits reasoning of other approaches easier.
- *Optimized Partitioning-based Approach* (`PT-Opt`): On top of `PT`, we incorporated several optimizations from existing solutions [21, 55], including asynchronous data streaming (with 32 streams), active partition tracking (see Section 1), and reusing loaded active partitions [21]. This approach roughly approximates the state-of-the-art in GPU-based graph processing under memory oversubscription.
- *Optimized Unified Memory-based Approach* (`UM-Opt`): To adopt unified memory, we allocated the graph (i.e., edge array) with `cudaMallocManaged()` [5], so active edges can be automatically migrated to the GPU memory as needed. After that, we tried to optimize this approach based on CUDA programming guidance [2] and ideas from a recent work on unified memory optimizations, ETC [35]. First, we provided a *data usage hint* via `cudaMemAdvise()` API to make the edge array `cudaMemAdviseSetReadMostly`. As the edge array does not change, this hint will avoid unnecessary writes back to the host memory. Based on our experiments, this optimization reduces the data transfer by 47% and total processing time by 23% on average. Second, we applied *thread throttling* [35] by reducing the number of active threads in a warp. However, we did not observe any performance improvements. A further examination revealed two reasons: (i) the maximum number of threads executing concurrently on a GPU is much smaller than the number of vertices (i.e., 2048×30 on the tested GPU versus tens of millions of vertices), so the actual working set is much smaller than the graph (assuming that each thread processes one vertex), easily fitting into the GPU memory; (ii) the accesses to the edge array exhibit good spatial locality thanks to the CSR representation. For these reasons, thread throttling turned out to be not effective. Besides the above two optimizations, another optimization we considered but found not applicable is *prefetching* [2]. The challenge is that the active vertices in next processing iteration are unpredictable, so to the edges needed to load. Finally, some lower-level optimizations such as memory page compression [35] and page size tuning [6] might be applicable, but they are not the focus of this work; we leave systematic low-level optimizations to the future work. To the best of our knowledge, this is the first time that unified memory is systematically evaluated for GPU-based graph processing, since it is fully realized recently.
- *Synchronous Subgraph Processing* (`Subway-sync`): In this approach, the asynchronous processing scheme in Subway is always disabled. Therefore, its measurements will solely reflect the benefits of subgraph generation technique.
- *Asynchronous Subgraph Processing* (`Subway-async`): In the last approach, the asynchronous processing scheme in Subway is enabled, but may be disabled temporarily as needed, depending on the ratio of active edges.

Datasets and Algorithms. Table 2 lists the graphs used in our evaluation, including five real-world graphs and one synthesized graph. Among them, `friendster-konect` and `twitter-mpi` are from the Koblenz Network Collection [3], `friendster-snap` is from the Stanford Network Analysis Project [6], `uk-2007` and `sk-2005` are two web graphs from the Laboratory for Web Algorithmics [4], and `RMAT` is a widely used graph generator [11]. In addition, Table 2 reports the numbers of vertices and edges, the range of estimated diameters, and the in-memory graph sizes with and without the edge weights, respectively.

Table 2. Graph Datasets

$|V|$: number of vertices; $|E|$: number of edges; Est. Dia.: estimated diameter range; Size$_w$: in-memory graph size (CSR) with edge weights; Size$_{nw}$: in-memory graph size (CSR) without edge weights.

Abbr.	Dataset	$\vert V\vert$	$\vert E\vert$	Est. Dia.	Size$_w$	Size$_{nw}$
SK	sk-2005 [4]	51M	1.95B	22-44	16GB	8GB
TW	twitter-mpi [3]	53M	1.96B	14-28	16GB	8GB
FK	friendster-konect [3]	68M	2.59B	21-42	21GB	11GB
FS	friendster-snap [6]	125M	3.61B	24-48	29GB	15GB
UK	uk-2007 [4]	110M	3.94B	133-266	32GB	16GB
RM	RMAT [11]	100M	10.0B	5-10	81GB	41GB

There are six widely used graph analytics evaluated. They include breath-first search (BFS), connected components (CC), single-source shortest path (SSSP), single-source widest path (SSWP), single-source number of shortest path (SSNSP), and PageRank (PR). Note that SSSP and SSWP work on weighted graphs, thus, the sizes of their input graphs are almost doubled comparing to other algorithms. To support asynchronous subgraph processing, PageRank and SSNSP are implemented using accumulative updates [70].

Evaluation Platform. We evaluated Subway mainly on a server that consists of an NVIDIA Titan XP GPU of 12 GB memory and a 64-core Intel Xeon Phi 7210 processor with 128 GB of RAM. The server runs Linux 3.10.0 with CUDA 9.0 installed. All GPU programs are compiled with `nvcc` using the highest optimization level.

Out-of-GPU-memory Cases. With edge weights (required by SSSP and SSWP), none of the six graphs in Table 2 fit into the GPU memory. In fact, besides the first two graphs

[5] The version of CUDA driver is v9.0.

[6] The default page size, 4KB, is used in our evaluation.

(SK and TW), the sizes of the other four graphs, as shown in the second last column of Table 2, are well beyond the GPU memory capacity (12GB). Without weights, three graphs (SK, TW, and FK) fit into the GPU memory. In the following, we only report results of the out-of-GPU-memory cases.

Next, we first compare the overall performance of different approaches, then focus on evaluating UM-Opt and the two versions of Subway: Subway-sync and Subway-async.

4.2 Overall Performance

Table 3 reports the overall performance results, where the PT column reports the raw execution time, while the following ones show the speedups of other methods over PT.

First, PT-Opt shows consistent speedup over PT, 2.0X on average, which confirms the effectiveness of optimizations from existing work [21, 55]. By contrast, UM-Opt does not always outperform PT, depending on the algorithms and input graphs. The numbers in italics correspond to the cases UM-Opt runs slower than PT. We will analyze the benefits and costs of UM-Opt in detail shortly in Section 4.3. On average, UM-Opt still brings in 1.5X speedup over PT.

Next, Subway-sync shows consistent improvements over not only PT (6X on average), but also existing optimizations PT-Opt (3X on average) and unified memory-based approach UM-Opt (4X on average). The significant speedups confirm the overall benefits of the proposed subgraph generation technique, despite its runtime costs. Later, in Section 4.4, we will breakdown its costs and benefits.

Finally, Subway-async shows the best performance among all, except for six algorithm-graph cases, where Subway-sync performs slightly better. More specifically, it yields up to 41.6X speedup over PT (8.5X on average), 15.4X speedup over PT-Opt (4.3X on average), and 12.2X speedup over UM-Opt (5.7X on average). In addition, it outperforms synchronous Subway (Subway-sync) by 1.4X on average. These results indicate that, when asynchronous subgraph processing is applicable (ensured by developers), it is worthwhile to adopt it under the out-of-GPU-memory scenarios. We will evaluate Subway-async in depth in Section 4.5.

4.3 Unified Memory: Benefits and Costs

To better understand the inconsistent benefits from unified memory, we further analyze its benefits and costs with more detailed measurements. First, as mentioned earlier, recent releases of unified memory (CUDA 8.0 and onwards) come with on-demand data migration, which essentially resembles the partition activeness-tracking optimization [21, 55]. With this new feature, CUDA runtime only loads the memory pages containing the data that GPU threads need, skipping the others. In the context of vertex-centric graph processing, on-demand data migration avoids loading the memory pages consisting only inactive edges. To confirm this benefit, we measured the volume of data transfer between CPU and GPU in UM-Opt and compared it with that in PT. The data was

Table 3. Performance Results

Numbers (speedups) in bold text are the highest among the five methods; Numbers (speedups) in italics are actually slowdown comparing to PT.

		PT	PT-Opt	UM-Opt	Subway-sync	Subway-async
SSSP	SK	118.3s	1.5X	3.5X	5.8X	**9.5X**
	TW	20.4s	1.7X	*0.8X*	2.9X	**6.0X**
	FK	53.0s	1.7X	*0.6X*	4.2X	**8.0X**
	FS	68.5s	1.6X	*0.7X*	4.2X	**6.7X**
	UK	492.7s	2.9X	1.9X	6.5X	**15.6X**
	RM	66.6s	1.3X	*0.6X*	2.0X	**3.1X**
SSWP	SK	174.7s	1.8X	5.2X	13.2X	**23.1X**
	TW	19.7s	2.2X	1.2X	4.4X	**7.0X**
	FK	50.3s	2.1X	1.2X	7.4X	**13.1X**
	FS	71.3s	1.8X	1.1X	8.0X	**12.5X**
	UK	350.8s	3.7X	4.9X	**38.8X**	36.3X
	RM	58.3s	1.1X	*0.5X*	2.2X	**3.7X**
BFS	FS	30.9s	1.9X	*0.9X*	6.7X	**9.6X**
	UK	176.3s	3.2X	10.3X	**28.8X**	21.8X
	RM	32.7s	1.5X	*0.7X*	3.2X	**3.7X**
CC	FS	22.9s	2.1X	1.1X	4.2X	**5.7X**
	UK	388.1s	5.7X	4.0X	10.9X	**26.0X**
	RM	25.5s	1.3X	*0.5X*	1.9X	**2.3X**
SSNSP	FS	59.1s	1.5X	*0.9X*	4.4X	**5.6X**
	UK	349.4s	4.0X	8.9X	**25.6X**	25.2X
	RM	61.7s	1.1X	*0.8X*	**4.6X**	3.9X
PR	FS	278.4s	2.5X	1.9X	**2.8X**	2.2X
	UK	577.9s	2.7X	3.4X	16.5X	**41.6X**
	RM	319.5s	1.2X	*0.9X*	**3.2X**	3.0X
GEOMEAN			2.0X	1.5X	6.0X	**8.5X**

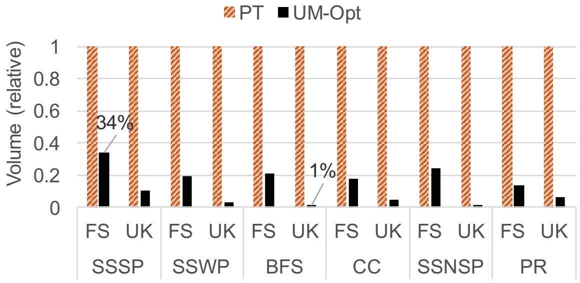


Figure 6. CPU-GPU Data Transfer (by volume).

collected with the help of Nvidia Visual Profiler 9.0 [5]. As reported in Figure 6, comparing to PT, the data transfer in UM-Opt is greatly reduced, up to 99% (occurred to BFS-UK). Moreover, unified memory also simplifies the programming by implicitly handling out-of-GPU-memory cases.

Despite the promises, the benefits of using unified memory for graph processing are limited by two major factors. First, the on-demand data migration is not free. When a requested memory page is not in the GPU memory, a page fault is triggered. The handling of page faults involves not only data transfer, but also TLB invalidations and page table updates, which could take tens of microseconds [2]. To estimate the significance of this extra overhead, we first collected the data transfer cost and the cost of graph computations, then

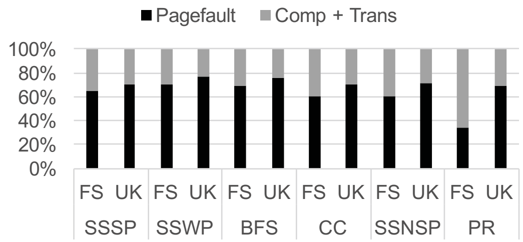


Figure 7. Page Fault Overhead in Unified Memory.

subtracted them from the total runtime of UM-Opt [7]. The remaining time is used as the estimation of page fault-related overhead. Figure 7 reports the breakdown, where overhead related page fault takes 23% to 69% of the total time. These substantial overhead may outweigh the benefits of reduced data transfer. As shown in Table 3 (UM-Opt column), in 5 out of 18 tested cases, UM-Opt runs slower than PT.

The second major factor limiting the benefits of unified memory-based graph processing lies in the *granularity* of data migration – memory pages: a migrated page may still carry inactive edges due to the sparse distributions of active vertices. We will report the volume of unnecessary data migration in the next section. While reducing memory page size may mitigate this issue, it increases the costs of page fault handling as more pages would need to be migrated.

4.4 Subgraph Generation: Benefits and Costs

Similar to the unified memory-based approach, using our subgraph generation brings both benefits and costs. The benefits come from reduced data transfer – only active edges are transferred to the GPU memory. On the other hand, the costs of subgraph generation occur on the critical path of the iterative graph processing – the next iteration waits until its subgraph (SubCSR) is generated and loaded. In addition, there is a minor cost in the subgraph processing due to the use of SubCSR, as opposed to CSR (see Section 3.1). Before comparing its costs and benefits, we report the frequencies of SubCSR generation and partitioning first.

Table 4 reports how frequently the SubCSR is generated and how many times the SubCSR/CSR is partitioned across iterations. Here, we focus on Subway-sync, we will discuss Subway-async shortly in Section 4.5. Note that the criterion for enabling SubCSR generation is that the ratio of active edges is over 80%. Among 24 algorithm-graph combinations, in 11 cases, this ratio is always under 80%; in 10 cases, the ratio exceeds 80% only in one iteration; and in the remaining three cases (CC-RM, PR-FS, and PR-RM), the ratio exceeds 80% more often: in 2 iterations, 11 iterations, and 7 iterations, respectively. This is because RM is the largest graph among the tested ones and algorithms PR and CC often activate more edges due to their nature of computation – all vertices (and edges) are active (100%) initially according to the algorithms.

Table 4. SubCSR Generation and Partitioning Statistics

Itr: total number of iterations; $Itr_{>80\%}$: number of iterations with more than 80% active edges (iteration IDs); $Itr_{>GPU}$: number of iterations with SubCSR greater than GPU memory capacity (iteration IDs).

		Subway-sync			Subway-async		
		Itr	$Itr_{>80\%}$	$Itr_{>GPU}$	Itr	$Itr_{>80\%}$	$Itr_{>GPU}$
SSSP	SK	90	0	0	86	1(1)	1(1)
	TW	15	1(5)	1(5)	10	1(1)	1(1)
	FK	30	1(7)	3(6-8)	22	1(1)	1(1)
	FS	27	1(7)	3(6-8)	21	1(1)	1(1)
	UK	187	0	16(17-32)	179	1(1)	1(1)
	RM	8	1(3)	2(3-4)	8	1(1)	3(1-3)
SSWP	SK	134	0	0	115	1(1)	1(1)
	TW	15	0	0	6	1(1)	1(1)
	FK	30	1(7)	2(6-7)	18	1(1)	1(1)
	FS	30	0	2(6-7)	25	1(1)	1(1)
	UK	134	0	0	122	1(1)	1(1)
	RM	10	1(3)	2(3-4)	9	1(1)	1(1)
BFS	FS	24	0	1(6)	15	1(1)	1(1)
	UK	134	0	0	122	1(1)	1(1)
	RM	6	1(3)	1(3)	4	1(1)	1(1)
CC	FS	15	1(1)	2(1-2)	8	1(1)	1(1)
	UK	291	1(1)	6(1-6)	122	1(1)	1(1)
	RM	4	2(1-2)	2(1-2)	4	1(1)	1(1)
SSNSP	FS	24	0	1(6)	18	1(1)	1(1)
	UK	134	0	0	127	1(1)	1(1)
	RM	6	0	1(3)	5	1(1)	2(1-2)
PR	FS	75	11(1-11)	17(1-17)	44	1(1)	12(1-12)
	UK	359	1(1)	3(1-3)	310	1(1)	1(1)
	RM	45	7(1-7)	21(1-21)	33	5(1-5)	14(1-14)

When SubCSR is not generated (i.e., the activeness ratio is below 80%), the graph (CSR) has to be partitioned; Otherwise, the graph (SubCSR) only needs to be partitioned if it remains oversized for the GPU. Both partitioning cases in general happen infrequently, except for algorithms PR and CC and graph RM, due to the same reasons just mentioned earlier.

Next, we report the benefits of reduced data transfer. As Figure 8 shows, the volume of data transfer in Subway-sync is dramatically reduced comparing to PT, with a reduction ranging from 89.1% to 99%. It is worthwhile to note that the reduction in Subway-sync is more significant than that in UM-Opt (Figure 8 vs. Figure 6). On average, the data transfer volume in UM-Opt is 3.3X of that in Subway-sync. The extra 2.3X data transfer is due to the saving of loading inactive edges carried by the migrated memory pages.

As to the cost of subgraph generation, instead of reporting its cost ratios, we combine the costs of subgraph generation and the subgraph transfer together (a breakdown between the two can be found in Figure 11), then compare the total cost to the transfer cost without subgraph generation. As reported in Figure 9, even adding the two costs together, the total remains significantly less than the data transfer cost without subgraph generation, with a reduction of time ranging from 83% to 98%. These results confirm the subgraph

[7] We also tried to measure the page fault cost with Nvidia Visual Profilers 9.0 and 10.0, which unfortunately do not produce reasonable results.

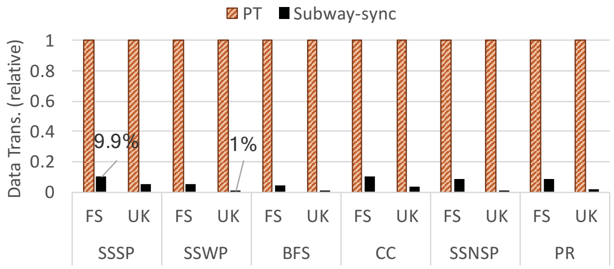


Figure 8. CPU-GPU Data Transfer (by volume).

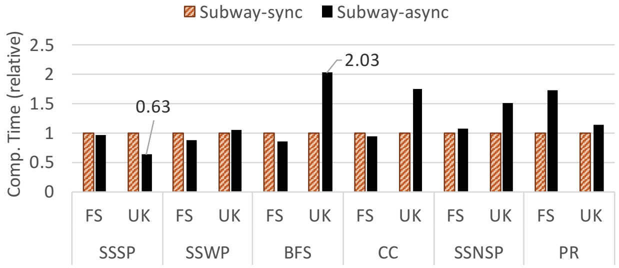


Figure 12. Impacts on Graph Computation Time.

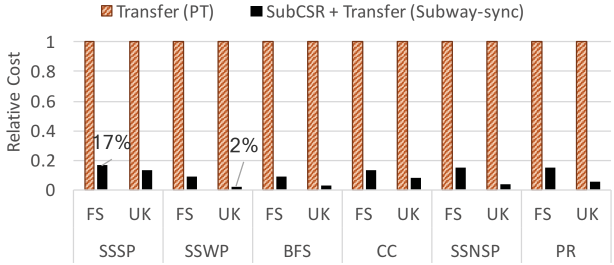


Figure 9. Time Costs of SubCSR Generation + Data Transfer.

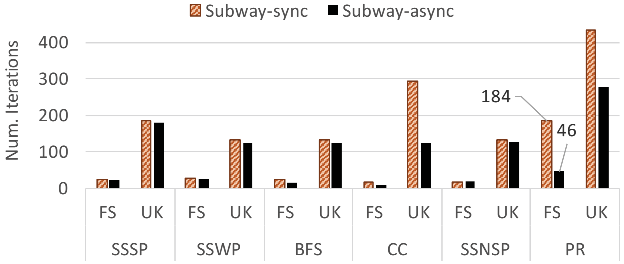


Figure 10. Impacts on Numbers of (Outer) Iterations.

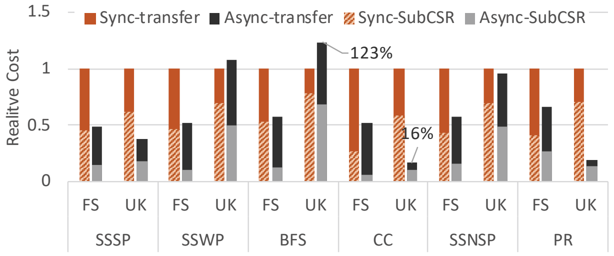


Figure 11. Impacts on SubCSR Generation + Transfer.

generation as a cost-effective way to reduce the data transfer. For this reason, `Subway-sync` exhibits significantly higher speedup than `UM-Opt` on average (see Table 3).

4.5 Asynchronous Processing: Benefits and Costs

Next, we examine the benefits and costs of asynchronous subgraph processing. As discussed in Section 3.2, adopting the asynchronous model tends to reduce the (outer) iterations of graph processing, thus saving the needs for subgraph generations and loading. To confirm this benefit, we profiled the total number of iterations in the outer loop of graph processing (also refer to Algorithm 4). Figure 10 compares the numbers of iterations with and without the asynchronous model. In general, the numbers of iterations are consistently reduced across all tested cases, except for `SSNSP-FS`, where the number of iterations remains unchanged. On average, the number of iterations is reduced by 31%. Correspondingly, the number of times for generating a subgraph and loading it to GPU is also reduced, as shown on the right of Table 4. Note that in asynchronous processing mode, to maximize the value propagation within a subgraph (i.e., more inner iterations), `Subway-async` always partitions and loads the entire graph in the first outer iteration.

However, it is interesting to note that the costs of subgraph generation and data transfer may not be reduced in the same ratios as the number of iterations, as indicated by Figure 11. For example, in the case of `PR-UK`, the number of iterations is reduced from 434 to 278 (about 36% reduction). However, its costs of subgraph generation and data transfer are reduced more significantly, by 71%. The opposite situations may also happen (e.g., in the case of `PR-FS`). The reason is that the asynchronous model affects not only the number of (outer) iterations, but also the amounts of active vertices and edges in the next (outer) iteration, thus altering the cost of subgraph generation and the volume of data transfer. In general, the overall impacts on the costs of subgraph generation and data transfer are positive, leading to a 52% reduction on average. Among the 12 examined cases, only 2 cases (`SSWP-UK` and `BFS-UK`) show cost increases, by up to 23%.

At last, by altering the way that values are propagated, the asynchronous model may also change the overall amount of graph computations, as reported in Figure 12. In general, the changes to the graph computation time vary across tested cases, ranging from 0.63X to 2.03X.

Adding the above impacts together (Figures 11 and 12), adopting the asynchronous model remains beneficial overall, yielding speedups in 18 out of 24 tested cases (see Table 3). For the others, the performance loss is within 10% on average.

Table 5. Subway (Out-of-GPU-memory) vs. Galois (CPU)

GPU: Titan XP (3840 cores, 12GB);
CPU: Xeon Phi 7210 (64 cores); RAM: 128GB

		`Subway-sync`	`Subway-async`	`Galois`
SSSP	SK	20.28s	12.51s	**5.42s**
	TW	6.94s	**3.41s**	7.94s
	FK	12.54s	**6.65s**	22.129s
	FS	16.17s	**10.26s**	29.28s
	UK	75.47s	31.54s	**13.44s**
	RM	33.1s	**21.49s**	29.63s
BFS	FS	4.65s	**3.21s**	8.35s
	UK	6.12s	8.1s	**5.07s**
	RM	10.2s	**8.72s**	17.32s
CC	FS	5.49s	**4.03s**	5.23s
	UK	35.76s	14.93s	**4.88s**
	RM	13.7s	11.11s	**10.47s**

4.6 Out-of-GPU-Memory vs. CPU-based Processing

Instead of providing out-of-GPU-memory graph processing support, another option is switching to the CPU-based graph processing, though this will put more pressure on the CPU, which may not be preferred if CPU is already overloaded. Nonetheless, we compare the performance of the two options for reference purposes. Note that the performance comparison depends on the models of CPU and GPU. In our setting, the GPU is Nvidia Titan XP with 3840 cores and 12GB memory while the CPU is Intel Xeon Phi 7210 processor with 64 cores. Both processors are hosted on the same machine with a RAM of 128GB. Note that the cost of the CPU is 4 to 5X more expensive than the GPU (according to our purchasing price). The CPU-based graph processing system we chose for comparison is a state-of-the-art system, Galois [44]. We also tried Ligra [56], another popular shared-memory graph system, however, due to its high memory demand, we found none of our tested graphs can be successfully processed on our machine. Table 5 reports the running time of both graph systems for the graph algorithms that both natively support. Note that Galois provides alternative implementations for each graph analytics. We used the best implementation in our setting: `default` for BFS and CC, and `topo` for SSSP. Overall, we found the performance of Subway is comparable to Galois in our experimental setup. In fact, Subway (async version) outperforms Galois in 7 out of 12 tested algorithm-graph combinations. Also note that, as an out-of-GPU-memory solution, the time of Subway includes not only all the data transfer time from CPU to GPU, but also the SubCSR generation time.

5 Related Work

Graph Processing Systems. There have been great interests in designing graph processing systems. Early works include Boost Graph Library [57] and parallel BGL [19]. Since the introduction of Pregal [38], vertex-centric programming has been adopted by many graph engines, such as Giraph [39], GraphLab [36], and PowerGraph [17]. More details about vertex-centric graph processing systems can be found in a survey [40]. A number of graph processing systems are built on distributed platforms to achieve high scalability [12–14, 17, 18, 36, 38, 59, 73]. They partition graphs and store the partitions across machines, based on edges [30, 50], vertices [17], or value vectors [68]. Some of the distributed graph processing systems adopt the idea of asynchronous processing among machines to improve the convergence rate and/or reduce the communication costs [14, 36, 61].

On shared-memory platforms, Ligra [56] and Galois [44] are widely recognized graph processing systems for their high efficiencies. Charm++ [29] and STAPL [60] are two more general parallel programming systems, with intensive supports for irregular computations, like graph processing.

A more relevant body of research is the out-of-core graph processing systems. Some representative systems include GraphChi [34], X-Stream [52], GraphQ [66], GridGraph [74], CLIP [8], Wanderland [69], Graspan [65], and among others. Some of their ideas have been adopted for handling GPU memory oversubscription, as discussed earlier. In addition, ideas in some prior work [8, 63] are also closely related to the techniques proposed in this work, but different in both contexts and technical details. In prior work [63], a dynamic graph partitioning method is proposed for disk-based graph processing, which operates on shards [34], a data structure optimized for disk-based graph processing. In comparison, our subgraph generation is based on CSR, a more popular representation in GPU-based graph processing. Furthermore, our technique features a new subgraph representation and a GPU-accelerated generation. In another prior work, CLIP [8], local iterations are applied to a graph partition loaded from the disk, which resembles our asynchronous model, but in a different context. Moreover, they only applied it to two graph analytics. In comparison, we have discussed the correctness of the asynchronous model and successfully applied it to a much broader range of graph analytics.

GPU-based Graph Processing. On GPU-based platforms, research has been growing to leverage the computation power of GPUs to accelerate graph processing. Early works in this area include the one from Harish and others [23], Maximum warp [25], CuSha [32], Medusa [72], and many algorithm-specific GPU implementations [15, 20, 27, 41, 46, 58]. More recently, Gunrock [67] introduced a frontier-based model for GPU graph processing, IrGL [48] presents a set of optimizations for throughput, and Tigr [45] proposed a graph transformation to reduce the graph irregularity.

Most of the above systems assume that the input graph can fit into the GPU memory, thus they are unable to handle GPU memory oversubscription scenarios. To address this limitation, GraphReduce [55] proposed a partitioning-based approach to explicitly manage the oversized graphs, with the capability to detect and skip inactive partitions. More recently, Graphie [21] further improved the design of the partitioning-based approach, with an adoption of X-Stream style graph processing and a pair of renaming techniques to reduce the cost of explicit GPU memory management.

In general, Subway is along the same direction as the above systems, with two critical advancements. First, it introduces a GPU-accelerated subgraph generation technique, which pushes the state-of-the-art partition-level activeness tracking down to the vertex level. Second, it brings asynchrony to in-GPU-memory subgraph processing to reduce the needs for subgraph generations and reloading. As demonstrated in Section 4, both techniques can significantly boost the graph processing efficiency under memory oversubscription.

Besides single-GPU graph processing scenarios, prior work also built graph systems on multi-GPU platforms [9, 28, 31, 49, 72] and proposed hybrid CPU-GPU processing scheme [16, 24, 37]. In these cases, the idea of asynchronous processing can be adopted among different graph partitions to reduce inter-GPU and CPU-GPU communications, similar to that in distributed graph processing (such as TLAG [61]).

6 Conclusion

For GPU-based graph processing, managing GPU memory oversubscription is a fundamental yet challenging issue to address. This work provides a highly cost-effective solution to extracting a subgraph that only consists of the edges of active vertices. As a consequence, the volume of data transfer between CPU and GPU is dramatically reduced. The benefits from data transfer reduction outweigh the costs of subgraph generation in (almost) all iterations of graph processing, bringing in substantial overall performance improvements. In addition, this work introduces an asynchronous model for in-GPU-memory subgraph processing, which can be safely applied to a wide range of graph algorithms to further boost the processing efficiency. In the evaluation, the proposed system (Subway) is compared with not only the existing techniques, but also an optimized unified memory-based implementation. The results confirm both the effectiveness and efficiency of the proposed techniques.

Acknowledgments

The authors would like to thank the anonymous reviewers for their time and comments and Dr. Phillip Stanley-Marbell for shepherding this work. This material is based upon the work supported in part by National Science Foundation (NSF) Grants No. 1813173 and 1524852. We also thank Nvidia for denoting the GPU, which was used for our experiments.

References

[1] CUDA C best practices guide: Chapter 9.1.3. zero copy. https://docs.nvidia.com/cuda/cuda-c-best-practices-guide. Accessed: 2019-06-30.

[2] CUDA C programming guide. https://docs.nvidia.com/cuda/archive/8.0/cuda-c-programming-guide/index.html. Accessed: 2019-06-30.

[3] The koblenz network collection. http://konect.uni-koblenz.de/. Accessed: 2019-06-30.

[4] Laboratory for web algorithmics. http://law.di.unimi.it/. Accessed: 2019-06-30.

[5] NVIDIA CUDA toolkit v8.0. http://developer.download.nvidia.com/compute/cuda/8$_0$/rel/docs/CUDA$_T$oolkit$_R$elease$_N$otes.pdf. Accessed: 2019-06-30.

[6] Stanford network analysis project. https://snap.stanford.edu/. Accessed: 2019-06-30.

[7] Thrust - parallel algorithms library. https://thrust.github.io/. Accessed: 2019-06-30.

[8] Zhiyuan Ai, Mingxing Zhang, Yongwei Wu, Xuehai Qian, Kang Chen, and Weimin Zheng. Squeezing out all the value of loaded data: An out-of-core graph processing system with reduced disk I/O. In *2017 USENIX Annual Technical Conference (USENIX ATC)*, pages 125–137, 2017.

[9] Tal Ben-Nun, Michael Sutton, Sreepathi Pai, and Keshav 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*, pages 235–248, 2017.

[10] Anthony Bonato. A survey of models of the web graph. In *Workshop on Combinatorial and Algorithmic Aspects of Networking*, pages 159–172. Springer, 2004.

[11] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. R-mat: A recursive model for graph mining. In *Proceedings of the 2004 SIAM International Conference on Data Mining*, pages 442–446. SIAM, 2004.

[12] Rong Chen, Jiaxin Shi, Yanzhe Chen, and Haibo Chen. Powerlyra: differentiated graph computation and partitioning on skewed graphs. In *Proceedings of the Tenth European Conference on Computer Systems*, pages 1–15, 2015.

[13] Roshan Dathathri, Gurbinder Gill, Loc Hoang, Hoang-Vu Dang, Alex Brooks, Nikoli Dryden, Marc Snir, and Keshav Pingali. Gluon: A communication-optimizing substrate for distributed heterogeneous graph analytics. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 752–768, 2018.

[14] Roshan Dathathri, Gurbinder Gill, Loc Hoang, Vishwesh Jatala, Keshav Pingali, V Krishna Nandivada, Hoang-Vu Dang, and Marc Snir. Gluon-async: A bulk-asynchronous system for distributed and heterogeneous graph analytics. In *2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 15–28. IEEE, 2019.

[15] Anil Gaihre, Zhenlin Wu, Fan Yao, and Hang Liu. Xbfs: exploring runtime optimizations for breadth-first search on GPUs. In *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing*, pages 121–131, 2019.

[16] Abdullah Gharaibeh, Lauro Beltrão Costa, Elizeu Santos-Neto, and Matei Ripeanu. A yoke of oxen and a thousand chickens for heavy lifting graph processing. In *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, pages 345–354, 2012.

[17] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 17–30, 2012.

[18] Joseph E Gonzalez, Reynold S Xin, Ankur Dave, Daniel Crankshaw, Michael J Franklin, and Ion Stoica. Graphx: Graph processing in a distributed dataflow framework. In *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*, pages 599–613, 2014.

[19] Douglas Gregor and Andrew Lumsdaine. The parallel BGL: A generic library for distributed graph computations. *Parallel Object-Oriented Scientific Computing (POOSC)*, 2:1–18, 2005.

[20] John Greiner. A comparison of parallel algorithms for connected components. In *Proceedings of the sixth annual ACM symposium on Parallel algorithms and architectures*, pages 16–25. ACM, 1994.

[21] Wei Han, Daniel Mawhirter, Bo Wu, and Matthew Buland. Graphie: Large-scale asynchronous graph traversals on just a GPU. In *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 233–245. IEEE, 2017.

[22] Pawan Harish and PJ Narayanan. Accelerating large graph algorithms on the GPU using CUDA. In *International conference on high-performance computing*, pages 197–208. Springer, 2007.

[23] Pawan Harish and PJ Narayanan. Accelerating large graph algorithms on the GPU using CUDA. In *International conference on high-performance computing*, pages 197–208. Springer, 2007.

[24] Stijn Heldens, Ana Lucia Varbanescu, and Alexandru Iosup. Hygraph: Fast graph processing on hybrid CPU-GPU platforms by dynamic load-balancing. 2016.

[25] Sungpack Hong, Sang Kyun Kim, Tayo Oguntebi, and Kunle Olukotun. Accelerating CUDA graph algorithms at maximum warp. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, pages 267–276, 2011.

[26] Zan Huang, Wingyan Chung, Thian-Huat Ong, and Hsinchun Chen. A graph-based recommender system for digital library. In *Proceedings of the 2nd ACM/IEEE-CS joint conference on Digital libraries*, pages 65–73. ACM, 2002.

[27] Yuntao Jia, Victor Lu, Jared Hoberock, Michael Garland, and John C Hart. Edge v. node parallelism for graph centrality metrics. *GPU Computing Gems*, 2:15–30, 2011.

[28] Zhihao Jia, Yongkee Kwon, Galen Shipman, Pat McCormick, Mattan Erez, and Alex Aiken. A distributed multi-GPU system for fast graph processing. *Proceedings of the VLDB Endowment*, 11(3):297–310, 2017.

[29] Laxmikant V Kale and Sanjeev Krishnan. CHARM++: a portable concurrent object oriented system based on c++. In *OOPSLA*, volume 93, pages 91–108. Citeseer, 1993.

[30] George Karypis and Vipin Kumar. Multilevelk-way partitioning scheme for irregular graphs. *Journal of Parallel and Distributed computing*, 48(1):96–129, 1998.

[31] Farzad Khorasani, Rajiv Gupta, and Laxmi N Bhuyan. Scalable SIMD-efficient graph processing on GPUs. In *2015 International Conference on Parallel Architecture and Compilation (PACT)*, pages 39–50. IEEE, 2015.

[32] Farzad Khorasani, Keval Vora, Rajiv Gupta, and Laxmi N Bhuyan. CuSha: vertex-centric graph processing on GPUs. In *Proceedings of the 23rd international symposium on High-performance parallel and distributed computing*, pages 239–252. ACM, 2014.

[33] Min-Soo Kim, Kyuhyeon An, Himchan Park, Hyunseok Seo, and Jinwook 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*, pages 447–461. ACM, 2016.

[34] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. GraphChi: Large-scale graph computation on just a PC. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 31–46, 2012.

[35] Chen Li, Rachata Ausavarungnirun, Christopher J Rossbach, Youtao Zhang, Onur Mutlu, Yang Guo, and Jun Yang. A framework for memory oversubscription management in graphics processing units. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 49–63. ACM, 2019.

[36] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M Hellerstein. Distributed graphlab: a framework for machine learning and data mining in the cloud. *Proceedings of the VLDB Endowment*, 5(8):716–727, 2012.

[37] Lingxiao Ma, Zhi Yang, Han Chen, Jilong Xue, and Yafei Dai. Garaph: Efficient GPU-accelerated graph processing on a single machine with balanced replication. In *2017 {USENIX} Annual Technical Conference ({USENIX}{ATC} 17)*, pages 195–207, 2017.

[38] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 135–146. ACM, 2010.

[39] Claudio Martella, Roman Shaposhnik, Dionysios Logothetis, and Steve Harenberg. *Practical graph analytics with apache giraph*, volume 1. Springer, 2015.

[40] Robert Ryan McCune, Tim Weninger, and Greg Madey. Thinking like a vertex: a survey of vertex-centric frameworks for large-scale distributed graph processing. *ACM Computing Surveys (CSUR)*, 48(2):1–39, 2015.

[41] Duane Merrill, Michael Garland, and Andrew Grimshaw. Scalable GPU graph traversal. In *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, pages 117–128, 2012.

[42] Alan Mislove, Hema Swetha Koppula, Krishna P Gummadi, Peter Druschel, and Bobby Bhattacharjee. Growth of the flickr social network. In *Proceedings of the first workshop on Online social networks*, pages 25–30. ACM, 2008.

[43] Alan Mislove, Massimiliano Marcon, Krishna P Gummadi, Peter Druschel, and Bobby Bhattacharjee. Measurement and analysis of online social networks. In *Proceedings of the 7th ACM SIGCOMM conference on Internet measurement*, pages 29–42. ACM, 2007.

[44] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. A lightweight infrastructure for graph analytics. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 456–471. ACM, 2013.

[45] Amir Hossein Nodehi Sabet, Junqiao Qiu, and Zhijia Zhao. Tigr: Transforming irregular graphs for GPU-friendly graph processing. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 622–636, 2018.

[46] Hector Ortega-Arranz, Yuri Torres, Diego R Llanos, and Arturo Gonzalez-Escribano. A new GPU-based approach to the shortest path problem. In *High Performance Computing and Simulation (HPCS), 2013 International Conference on*, pages 505–511. IEEE, 2013.

[47] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The PageRank citation ranking: Bringing order to the web. Technical report, Stanford InfoLab, 1999.

[48] Sreepathi Pai and Keshav 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*, pages 1–19, 2016.

[49] Yuechao Pan, Yangzihao Wang, Yuduo Wu, Carl Yang, and John D Owens. Multi-GPU graph analytics. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 479–490. IEEE, 2017.

[50] François Pellegrini and Jean Roman. Scotch: A software package for static mapping by dual recursive bipartitioning of process and architecture graphs. In *High-Performance Computing and Networking*, pages 493–498. Springer, 1996.

[51] Alain Pirotte, Jean-Michel Renders, Marco Saerens, et al. Random-walk computation of similarities between nodes of a graph with application to collaborative recommendation. *IEEE Transactions on Knowledge & Data Engineering*, (3):355–369, 2007.

[52] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. X-stream: Edge-centric graph processing using streaming partitions. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 472–488. ACM, 2013.

[53] Gorka Sadowski and Philip Rathle. Fraud detection: Discovering connections with graph databases. *White Paper-Neo Technology-Graphs are Everywhere*, 2014.

[54] John Scott. Social network analysis. *Sociology*, 22(1):109–127, 1988.

[55] Dipanjan Sengupta, Shuaiwen Leon Song, Kapil Agarwal, and Karsten Schwan. GraphReduce: processing large-scale graphs on accelerator-based systems. In *SC'15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12. IEEE, 2015.

[56] Julian Shun and Guy 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*, pages 135–146, 2013.

[57] Jeremy G Siek, Lie-Quan Lee, and Andrew Lumsdaine. The boost graph library: User guide and reference manual, portable documents, 2001.

[58] Jyothish Soman, Kothapalli Kishore, and PJ Narayanan. A fast GPU algorithm for graph connectivity. In *2010 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW)*, pages 1–8. IEEE, 2010.

[59] Shuang Song, Xu Liu, Qinzhe Wu, Andreas Gerstlauer, Tao Li, and Lizy K John. Start late or finish early: A distributed graph processing system with redundancy reduction. *Proceedings of the VLDB Endowment*, 12(2):154–168, 2018.

[60] Gabriel Tanase, Antal Buss, Adam Fidel, Ioannis Papadopoulos, Olga Pearce, Timmie Smith, Nathan Thomas, Xiabing Xu, Nedal Mourad, Jeremy Vu, et al. *The STAPL parallel container framework*, volume 46. ACM, 2011.

[61] Yuanyuan Tian, Andrey Balmin, Severin Andreas Corsten, Shirish Tatikonda, and John McPherson. From think like a vertex to think like a graph. *Proceedings of the VLDB Endowment*, 7(3):193–204, 2013.

[62] Leslie G Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

[63] Keval Vora, Guoqing Xu, and Rajiv Gupta. Load the edges you need: A generic I/O optimization for disk-based graph processing. In *2016 USENIX Annual Technical Conference (USENIX ATC)*, pages 507–522, 2016.

[64] Stephan M Wagner and Nikrouz Neshat. Assessing the vulnerability of supply chains using graph theory. *International Journal of Production Economics*, 126(1):121–129, 2010.

[65] Kai Wang, Aftab Hussain, Zhiqiang Zuo, Guoqing Xu, and Ardalan Amiri Sani. Graspan: A single-machine disk-based graph system for interprocedural static analyses of large-scale systems code. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 389–404. ACM, 2017.

[66] Kai Wang, Guoqing (Harry) Xu, Zhendong Su, and Yu David Liu. Graphq: Graph query processing with abstraction refinement-scalable and programmable analytics over very large graphs on a single pc. In *USENIX Annual Technical Conference*, pages 387–401, 2015.

[67] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John 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*, pages 1–12, 2016.

[68] Mingxing Zhang, Yongwei Wu, Kang Chen, Xuehai Qian, Xue Li, and Weimin Zheng. Exploring the hidden dimension in graph processing. In *OSDI*, pages 285–300, 2016.

[69] Mingxing Zhang, Yongwei Wu, Youwei Zhuo, Xuehai Qian, Chengying Huan, and Kang Chen. Wonderland: A novel abstraction-based out-of-core graph processing system. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 608–621, 2018.

[70] Yanfeng Zhang, Qixin Gao, Lixin Gao, and Cuirong Wang. Accelerate large-scale iterative computation through asynchronous accumulative updates. In *Proceedings of the 3rd workshop on Scientific Cloud Computing*, pages 13–22. ACM, 2012.

[71] Yu Zhang, Xiaofei Liao, Hai Jin, Bingsheng He, Haikun Liu, and Lin Gu. DiGraph: An efficient path-based iterative directed graph processing system on multiple GPUs. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 601–614. ACM, 2019.

[72] Jianlong Zhong and Bingsheng He. Medusa: Simplified graph processing on GPUs. *IEEE Transactions on Parallel and Distributed Systems*, 25(6):1543–1552, 2013.

[73] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. Gemini: A computation-centric distributed graph processing system. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pages 301–316, 2016.

[74] Xiaowei Zhu, Wentao Han, and Wenguang Chen. GridGraph: Large-scale graph processing on a single machine using 2-level hierarchical partitioning. In *2015 USENIX Annual Technical Conference (USENIX ATC)*, pages 375–386, 2015.