Excavating the Potential of GPU for Accelerating Graph Traversal

Pengyu Wang, Lu Zhang, Chao Li, Minyi Guo
Department of Computer Science and Engineering
Shanghai Jiao Tong University, Shanghai, China
Email: {wpybtw, luzhang}@sjtu.edu.cn, {lichao, guo-my}@cs.sjtu.edu.cn

***Abstract*—Graph traversal is an essential procedure for a growing amount of applications today. This type of algorithms typically iterate input graph datasets until convergence and the logic of each iteration is quite simple. GPUs are used extensively as graph traversal accelerators due to the capability of massive parallelism and high-bandwidth memory access. However, existing methods are inefficient in two ways. First, streaming multiprocessors (SMs) are still underutilized due to the unbalanced load allocation and uncoalesced memory access. Second, they use space-inefficient data structures or need auxiliary data to assist traversal. It is undesirable, considering the limited GPU memory capacity. Moreover, existing designs commonly focus on optimizing kernel execution time. Data-transfer time is also notable in the whole procedure. Thus, space-efficient data structure and data-transfer policy should be concerned.**

In this paper, we propose *EtaGraph*, a novel GPU graph traversal framework optimized for GPU memory system and execution parallelism. EtaGraph has several features: 1). It uses a frontier-like kernel execution model, featuring a lightweight graph transformation procedure, named *Unified Degree Cut*, allowing GPU threads to process skewed graph efficiently without modification of raw data or introducing extra space overhead; 2). It uses on-demand data-transfer to overlap computation so that it optimizes the total time of data-transfer and execution; 3). It adopts an explicit utilization of Shared Memory to enhance memory coalescing and to improve effective memory bandwidth. Evaluation of EtaGraph shows significant and consistent speedups over the state-of-the-art GPU-based graph processing frameworks on both real-world and synthetic graphs.

***Index Terms*—GPU, graph traversal, prefetch**

I. Introduction

GPU has arisen as an attractive graph processing platform as modern GPUs support thousands of concurrent threads and high memory bandwidth. However, directly mapping graph traversal applications on GPUs suffers from low streaming multiprocessor (SM) utilization and unsatisfactory memory bandwidth utilization due to the skewed degree distribution of vertices and the fine-grained, random memory access of graph data. Prior works [1] [2] [3] show that carefully designed GPU-based frameworks can achieve comparable or even orders of magnitude better performance than shared-memory or distributed systems, such as Graphlab [4] and Ligra [5].

Despite the contributions to graph processing, prior works have certain limitations. Memory capacity of GPU limits the scalability of graph traversal. Single GPU graph processing systems [2] [6] [3] need to load all graph data in device GPU memory, thus the GPU memory capacity severely constrains their processing ability. GPU memory, hardly more than 16GB (for even high-end computing cards [7]), is scarce compared with main memory up to TBs capacity. Some studies make good attempts, trying to scale out GPU processing capability by utilizing the resources of multiple GPUs such as [8] [9]. However, communication bandwidth through the PCI-e interface is relatively low and the overhead significantly limits the scalability of multi-GPUs systems (often no more than 8 GPUs [8] [9]).

Moreover, existing methods cannot fully utilize the GPUs' memory sub-system. First, uncoalesced memory access is still a major bottleneck due to the fine-grained memory access of graph traversal. It leads to low effective memory bandwidth. Some works [2] [3] [10] try to improve memory coalescing. However, they rely on customized data structure and can hardly apply to other graph frameworks. Such customized data structures often need time-consuming pre-processing. Second, redundant graph data structures or auxiliary data assisting the execution are required, thus are inefficient in space. The space overhead makes GPU memory capacity even scarcer.

Furthermore, researchers often focus on optimizing kernel execution time. Nevertheless, data-transfer is also an important part of the whole procedure and often dominate the total time. Thus, space-efficient data structure and wise data-transfer policy should be concerned. There are works [11] [12] trying to overlap data-transfer and kernel execution by using several *CUDA Streams*. They both use fixed-sized data chunks (partitions) to stream. This could cause waste of work if there is only a small part of data actually used in one chunk. Flexible overlapping could be potentially efficient.

Beyond that, prefetching is a well-explored technique on CPUs to hide memory access latency. By fetching instructions or data to cache before used, one can reduce memory access latency. Several works [13] [14] adopt prefetching on GPUs. Due to the irregularity of graph data, it is not straightforward to do prefetching for graph processing. There has been prior work [15] showing that prefetching graph data into unused register improves performance for graph algorithms, but it needs to modify GPU hardware or compiler. Easy-to-use prefetching on GPUs for graph applications are still absent.

In this paper, we investigate the inefficiency or underutilization of GPU processing graph traversal in data management and kernel execution. We make several attempts: We try to use a lightweight degree-optimized transformation policy to relieve time-consuming pre-processing and save memory usage. We attempt to use a frontier-like kernel-invoking model to improve SMs' execution efficiency and load balancing. We intend to reduce the total time of data-transfer and kernel execution by using Unified Memory (UM). We exploit the potential of GPU shared memory, which is often overlooked in prior works, to improve memory sub-system efficiency.

This work makes follow contributions:

- we propose a lightweight graph transformation policy named Unified Degree Cut (UDC). UDC allows GPU process skewed graph data efficiently without pre-processing of graph data. Combined with a frontier-like execution method, it improve utilization of SMs.
- We propose a new GPU graph traversal programming pattern. This programming pattern allows overlapping between transfer and processing of data requested of each iteration in a fine-grained granularity.
- We introduce a software prefetching mechanism to improve effective memory bandwidth, named *shared memory prefetch (SMP)*, to explicitly exploit the GPU shared memory. It improves IPC (instructions per cycle), cache hit rate, cache throughput and reduces request memory transactions dramatically.
- Finally, we introduce EtaGraph, a GPU graph processing framework efficient in memory usage and kernel execution for graph traversal. We compare it with several state-of-the-art systems.

The rest of this paper organizes as follows. Section II briefly introduces background and motivation of this paper. In section III, we introduce the graph transformation procedure. In section IV, we present the kernel execution method. In section V, we introduce how to exploit shared memory for graph applications. In section VI, we present the implementation, evaluation and analysis of results. Section VII discusses related work. Finally, section VIII summarizes this paper.

II. Background and Motivation

In this section, we briefly introduce the execution and memory hierarchy of GPUs. Then we introduce the data structures to store graph data. Finally, we introduce the notion of graph traversal problems.

A. GPU Execution and Memory Hierarchy

The basic processing units of GPUs are GPU cores organized as SMs. SMs have a fixed number of computing units (depending on specific architecture). Computing units on the same SM are processing in SIMT (single instruction, multiple threads) manner controlled by the Control Unit. GPU programs (kernels) are launched as a grid of thread blocks (TBs). 32 threads are grouped as a thread warp (for NVIDIA GPUs), while thread warps are organized in a TB. TBs are assigned to SMs. These concurrent TBs residing on one multiprocessor share resources of SM.

GPUs are equipped with GBs of DRAMs, noted as device memory or global memory, to transfer data between main memory and store data. Each GPU core has its own register file while all GPU Core in one SM share L1 Cache. SMs also have shared memory. Shared memory is as fast as L1 cache but is programmable. More precisely, shared memory (or scratch-pad memory) can be managed by users, while L1 Cache is hardware-controlled. L1 Cache and shared memory are in the same area on prior generations of GPUs (Fermi, Kepler) and the ratio can be manually configured. Shared memory on newer generations (Maxwell, Pascal) has preset size. SMs also have texture memory and constant memory, but they can be hardly utilized in graph processing. There is also local memory for SM, but it is used only when there are no sufficient registers and it has the same transfer latency as global memory.

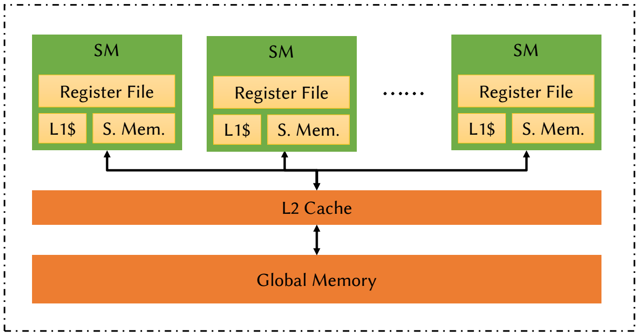


Fig. 1: GPU memory architecture

B. Graph Data Structure

Graph datasets often have millions or even billions of vertices and are very sparse. The adjacency matrix is not suitable to store large graph datasets. Several sparse graph representations are introduced to improve space efficiency.

Compressed Sparse Row (CSR) is a commonly used graph representation. It uses two arrays, Column Index Array and Row Offset Array, to store the adjacency list and the index of each node. Compressed Sparse Column (CSC) is similar.

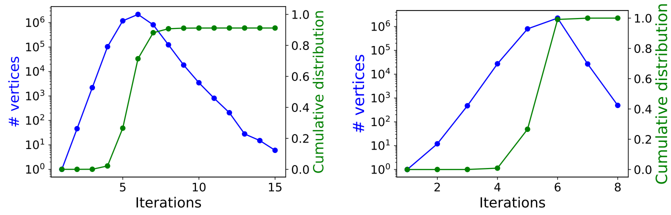
Edge-centric frameworks, like X-Stream [16], often use edge-list (tuples of Source Node and Destination Node) to store graph data. Cusha [2] proposes a data structure named G-shard. It is basically the same as edge-list but ordered and divided into small parts.

C. Notion of Graph Traversal

Graph traversal is to calculate labels of all nodes reachable from a given root node. Visited nodes can reach other nodes by their out-going edges. Breadth-First Search (BFS), Single Source Shortest Path (SSSP) and Single Source Widest Path (SSWP) are well-used graph traversal problems to find certain attribute label during iterations.

Definition 1: A vertex is *active* at one iteration if it is updated by its neighbor in the previous iteration.

Vertices can be visited several times. For BFS, nodes can be active only once because later visit can not update their labels with a smaller value. For SSSP and SSWP, it is not the case if edges are not uniformly weighted.



(a) Vertex activation of Livegjournal (b) Vertex activation of com-orkut

Fig. 2: Number and cumulative distribution of active vertices at each iteration for BFS algorithm on Livejournal [17] and com-Orkut [18]. The vertices become active gradually during iterations.

Definition 2: Activatable subgraph (AS) is the induced subgraph that has all its vertices reachable from the source node together with all edges whose endpoints are such vertices.

Generally, AS is the superset of the strongly connected component which source node is located in.

Graph traversal algorithms are different from PageRank-like algorithms. Original PageRank updates all vertices' value at every iteration until value converges. For traversal algorithms, one vertex is assigned as the source at the first iteration. Then, the vertices recently visited in the previous iteration become active and their label value will propagate to neighbor vertices in the current iteration. For BFS, all vertices within the *activatable subgraph* of a directed graph will be visited only once. Vertices outside the AS will not be active at all. For other traversal algorithms, vertices within AS could be active multiple times if the edges have different values. Otherwise, vertices within AS will be active at most once.

As Fig. 2 shows, the number of active vertices starts from 1 and grows exponentially at first few iterations, and then decreases exponentially. Cumulative distribution of active vertices remains relatively low in the first few iterations. Afterward, it increases dramatically until most of the vertices active, and then stay stable. Note that vertices of Livejournal is not strongly connected and not all vertices will be visited.

III. Unified Degree Cut

In this section, we first introduce the general idea of the graph partition method called *unified degree cut (UDC)*. Then we verify the correctness for UDC in graph traversal.

A. Degree Cut Transformation

Realistic graphs are often highly skewed. The degree of vertices in social networks follows power-law-like distributions. Labels are propagated through the out-going edges in push-based vertex-centric algorithms. Directly using vertex-centric programming model could cause a long-tail distribution of thread processing time, thus most of the number of threads have to wait until threads of large out-degree nodes finish. The basic idea to solve this is to set a limit for vertices' outdegree so that the upper bound of one thread's work is determined.

	Theory Space Overhead	Normalized Usage for LJ
G-Shard [2]	$2\|E\|$	1.87
Edge List [3]	$2\|E\|$	1.87
VST [6]	$\|E\| + 2\|N\| + 2\|V\|$	1.32
CSR	$\|E\| + \|V\|$	1

TABLE I: Theoretical space overhead and normalized usage for LiveJournal [17]

We introduce a number K called *Degree Limit*. This number serves as an upper bound of outdegrees. Any vertices with degree larger K will be treated as multiple nodes. We use *Unified Degree Cut* to denote such transformation.

Definition 3: **Unified Degree Cut** transformation is a mapping for vertex *v* along with its edge set E_v:

$$\mathcal{T} : (v, E_v) \rightarrow (\{v'\}, \{E_{v'}\}).$$

where $\{v'\}$ is a set of 'virtual vertices' of vertex *v* having same ID, named *shadow vertices*; degrees of shadow vertices are less or equal than K; $\{E_{v'}\}$ is the set of edge sets of shadow vertices; $E_v = \cup_{u \in \{v'\}} E_u$ and $\cap_{u \in \{v'\}} E_u = \emptyset$.

Frankly speaking, shadow vertices of the same original vertex share original Vertex ID and a disjoint portion of outgoing edges so that outdegree of each shadow vertices are less than or equal to K. A vertex, whose outdegree is no more than K, itself can be seen as a shadow vertex.

When traveling graphs, vertex-centric kernels are invoked to process shadow vertices with outdegree on more than K. Thus, it restricts the execution time of single threads and it improves lower bound of threads efficiency of warps.

There are two ways to perform such transformation: *in-core* or *out-of-core*. In-core means transformation is processed on GPU when vertices are about to be processed. Differently, out-of-core is to transform all vertices ahead at the main memory. Thus, out-of-core will consume extra memory.

We directly copy CSR data from main memory to GPU device memory and perform UDC on the fly. We list the amount of memory transferred from CPU to GPU of several data structures for comparison. Without loss of generality, we only consider topology data of graph, i.e. the connectivity of vertices. Table I shows the memory overhead and normalized usage compared with several data structures used in the state-of-the-art GPU-based graph processing systems. $|E|$, $|V|$ are the numbers of edges and vertices, respectively. $|N|$ is the number of shadow vertices with K equals to 10.

Inspired by the Virtual Split Transformation (VST) of Tigr [6], UDC is superior to VST in two ways. First, VST introduces extra $2n_k$ space overhead (n_k is the number of virtual parts generated with degree bound k) shown in Table I. Second, Tigr performs VST during the pre-processing procedure and need to generate a copy of raw data to insert outdegree informations. Even though pre-processing time is normally not considered in prior research papers, UDC does save time at the data-loading procedure and generate shadow vertices on the fly with negligible overhead.

B. Correctness for Graph Traversal

Without loss of generality, we consider graphs with no duplicate edges. Then we have

Theorem 1: If a vertex has an edge to its neighbor, then one of its shadow vertices has the edge to that neighbor.

This is obvious according to the definition of shadow vertices. Thus, if a vertex is reachable from source vertex originally, then this vertex is reachable from shadow vertices of source vertex. We use *virtual path* to denote the path connecting shadow vertices.

Theorem 2: A vertex has a path *p* from source through shadow vertices has a path *p'* through one of shadow vertices of each vertex in *p*.

When calculating the attribute of the virtual path using same edge or vertex attributes, we can get the same results as the actual path. Thus, traversal using the virtual path is identical as using the actual path.

IV. Selective Kernel Execution

In this section, we first introduce *active set* to show graph traversal procedure. Then we present our modified version of frontier. Finally, we propose a novel graph traversal programming pattern.

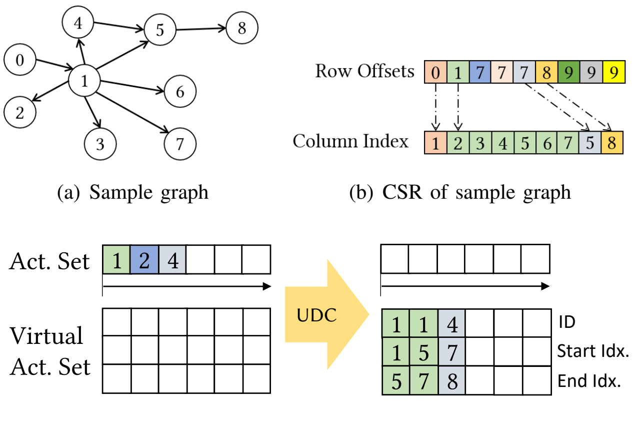
A. Active Set

We use *active set* to denote the set of active vertices of one certain iteration. Clearly, vertices in the active set will visit their neighbor vertices and update neighbors' label in this iteration. During iterations, kernels are invoked for vertices in the active set of the last iteration. Afterward, the updated vertices are added to the new active set. After each iteration, vertices in the active set are transformed using UDC and generated shadow vertices are added to a *virtual active set*.

The virtual active set record the *ID*, *Start Index* and *End Index* of each shadow vertices with an array of 3-tuple. Start index and End Index show the start and end index of shadow vertex's out-going edges, respectively. For example, Fig. 3(b) is the CSR expression of the example graph in Fig. 3(a). As Fig. 3(c) shows, if vertex 1, 2 and 4 are visited at one iteration, they are pushed in active set and should be active in the next iteration. Vertex 1 is partitioned as two shadow vertices. Vertex 2 will not be partitioned into any shadow vertices because its outdegree is 0 and won't propagate its label to any vertices. If we have K equals 4, Vertex 4 is treated as one shadow vertex since its outdegree is smaller than K.

When UDC is finished at each iteration, the active set is reset to reuse the memory allocation. And each shadow vertex in *virtual active set* will be assigned to one GPU thread to process in a new iteration. *Virtual active set* is reset when shadow vertices are processed.

With this procedure, the efficiency of GPU threads improves in two ways. First, only active vertices are assigned with GPU threads and it naturally filter active vertices with outdegree equals to 0. Thus, this makes sure all the invoked GPU threads are doing useful work. Second, it achieve load balancing because the workload of each threads are determined for varying graph datasets.



(a) Sample graph

(b) CSR of sample graph

(c) Active set to virtual active set

Fig. 3: CSR expression and transformation with K=4

With active set, kernels only need to be invoked on necessary vertices to be processed. For BFS and unweighted SSSP, all vertices can be active at most one time. Thus, only $|N|$, i.e. the number of shadow vertices, kernels need to be processed. As for memory request, then there is no need to check whether vertices are active. In this case, the amount of memory request is $|E| + |V|$. If a graph needs a large number of iterations to traversal, it will save significant memory request in theory.

Active set in our terminology is often called as *frontier* in [19] [3]. We use *active set* and *virtual active set* to address the UDC transformation procedure, which allow to generate load-balanced work items directly using CSR.

B. Fine-grained Overlapping

The SIMT execution pattern of GPU relies on data remaining local at GPU device to keep all threads active. In this case, kernels get major performance due to massive parallelism. Otherwise, threads warp will halt until requested memory arrives. The programming pattern of existing (in-GPU-memory) GPU graph processing systems as follows:

- Data load and pre-processing.
- Move data to GPU device.
- Parallel execution of GPU kernels.
- Results transfer back to main memory.

This programming pattern is straightforward to implement and is superior to distributed CPU systems with GPUs as accelerator [11]. However, graph data are originally stored at the main memory. Transferring large graph data from main memory to device memory will consume lots of time. This pattern may be unable to obtain the best performance when considering the total time of data-transfer and execution.

As described before, only part of the vertices are active at each iteration for graph traversal algorithms. If we transfer data of newly generated active vertices needed in the next iteration, overlapping of data-transfer and kernel execution can be potentially achieved.

We propose a novel GPU graph traversal programming pattern to allow fine-grained overlapping. This programming pattern is as follows:

- Data load and pre-processing.
- Move data of vertices in the active set to GPU device for each iteration and execute until coverage.
- Results transfer back to main memory.

We argue that graph traversal on GPU with this programming pattern is potential efficient in two ways. 1). Kernel execution and memory transfer procedure could be deeply overlapped. 2). Part of the graph data do not have to be transferred if only part of vertices will be active.

Note that maintaining high performance with the above programming pattern can be challenging. First, it is not straightforward to select data of active vertices and aggregate them in an efficient and low-latency manner. Second, the latency of the interconnect (PCIe, etc.) interface could be a bottleneck for performance.

This programming pattern can be implemented in several ways, manually or with CUDA's memory support. Zero-Copy Memory and Unified Memory [20] of CUDA can support our programming pattern. Zero-Copy Memory is memory allocation pinned in CPU system memory, that can be transferred on demand. Pinned in CPU system memory means that memory is page-locked to improve performance. On the other hand, Unified Memory is a single coherent memory image with a common address space. Data can be transparently migrated towards the processor that requests it. GPUs use page faulting mechanism to support this feature. If a kernel running on the GPU accesses a page that is not resident in its memory, it triggers page faults, allowing the page to be automatically migrated to the GPU memory on-demand. The essential difference is that Zero-Copy Memory is always in main memory thus GPU access it slower. We won't manipulate graph data during GPU processing. Thus, Unified Memory is a better choice to store graph topology data.

Unified Memory can be tuned by memory usage hints such as *cudaMemPrefetchAsync*. The *cudaMemPrefetchAsync* is a CUDA API that could guide CUDA runtime transfer data to the desired location immediately. Otherwise, UM will migrate on demand. It is essentially the same as using common GPU memory allocation when *cudaMemPrefetchAsync* is set.

As for the performance issue, GPUs use multi-threading as a latency-hiding technique. Each SM processes several thread warps simultaneously and warps are scheduled when stall. We will characterize UM performance for graph traversal at Subsection VI-C.

V. Shared Memory Prefetch

In this section, we first illustrate the inefficiency of GPU memory sub-system when processing graph algorithms. Then we introduce how SMP works.

A. Underutilization in Memory Sub-system

It is well-known that graph processing has very poor space locality. The architecture of GPU makes this even worse. Vertex-centric graph frameworks often follow this pattern: load and process neighbor vertices one by one. Neighbor vertices' IDs are often stored consecutively in CSR. The IDs of adjacent neighbor can be cached when accessing prior vertices. However, every SM has a dedicated but small L1 cache (24, 48, or 96 KB depending on GPU architecture). Similarly, a single GPU has about a few KB of byte L2 cache (2800∼KB for GTX 1080Ti GPU) shared among all SMs. Unlike CPU, there is only a few hundred bytes of L2 cache for every thread warps when GPU achieve relatively high occupation. The L1 and L2 cache can be quickly evicted before used. Thus, graph processing cannot fully utilize its GPU cache in theory. In our experiments, L2 read hit rate is around 19% for Tigr [6].

Graph traversal application performs fine-grained memory access when reading neighbor vertex data (usually stored in 4-byte format). However, memory requests from a warp are transformed into cache line requests with a size of 32B to GPU memory. Thus, a 4-byte cache miss will occur 32-byte memory access from device memory and results in huge memory bandwidth waste.

B. Improving Memory Efficiency

In this paper we ask this question: *now that loading and processing vertices one by one is not cache-friendly, what if we load all neighbor vertices at first?*

Shared memory, which is often overlooked in graph frameworks, provides high-throughput and low-latency memory access and can be used to share data among threads within the same thread block (TB). It is a good candidate for one to store vertices data temporarily. The size of shared memory is quite limited and computing memory index could introduce overhead due to the degree diversity of vertices. Fortunately, we can transform vertices into shadow vertices with a certain degree limit K as we mentioned earlier. Thus, the maximum number of memory access is known in advance. Each thread processes a virtual vertex with maximum outdegree of K. In this case, we could just assign all threads to a partition of shared memory and rely on the kernel to fetch data from device memory to shared memory from the beginning. In addition, at the compiler side we can let all kernels aggressively load K neighbor vertices and unroll loops of memory requests.

The above strategy has two benefits. First, the cache hit rate could improve and global memory transactions are reduced. One neighbor vertex is fetched right after the prior one being fetched, potentially using cached data before eviction and requiring no more global memory transactions. Second, better Instruction Level Parallelism (ILP) is achieved due to unrolled memory access because the amount of memory requested is known beforehand.

We name this procedure *shared memory prefetch (SMP)*. In practice, we use two virtual active set to store shadow vertices. One virtual active set stores shadow vertices having the degree equal to K, while another one stores shadow vertices whose degree is less than K. We prefetch K vertices for first virtual active set and prefetch *K-1* for second virtual active set. By doing this, more data requests are issued for those shadow

Procedure 1

```
Data: Graph data in CSR format
Result: Vertex labels $label$
Process Main():
    Load data into UM allocation $CSR$;
    Init $label$ and transfer to GPU;
    Allocate $actSet$ at GPU;
    Allocate $virtActSet$ at GPU;
    Init $actSet$;
    cudaMemPrefetchAsync($CSR$) ;
    while $actSet.isNotEmpty()$ do
        $actSet2virtActSet()$;
        $invokeKernel(BFS, virtActSet.size())$;
    end
Function actSet2virtActSet():
    for $v_k inactSet$ do
        $virtActSet.append(v_k.toVirtVtx())$;
    end
Kernel BFS():
    $getVirtVtx(thread.id, v)$;
    Fetch $v.neighbor$ to $sharedMem$;
    for $v'$ $in$ $v.neighbor$ do
        if $label[v'] > label[v] + 1$ then
            $label[v'] = label[v] + 1$;
            $actSet.append(v')$;
        end
    end
```

vertices whose degree is less than *K-1*. However, performance actually improves. Detailed evaluations of SMP in Subsection VI-C show its efficiency in cache hit rate, throughput, and overall memory transactions.

VI. Evaluation

In this section, we discuss our implementation of EtaGraph and the experiment methodology. Then we present performance results and in-depth analysis.

A. Implementation

We implement EtaGraph in standard C++ and CUDA. We utilize the Unified Memory (UM) introduced in CUDA 6.0 supported by post-Kepler generation GPUs. UM allows one to use a single pointer to data and data will automatically migrate towards the processor requesting it. We implement two versions of EtaGraph with and without *cudaMemPrefetchAsync*, denoted as EtaGraph and EtaGraph w/o UMP. We use a simple device array to track active nodes and active shadow vertices. The device array uses atomic operations to add elements. We use Galois [21] CSR binary format to store graph data for loading from disk. Procedure 1 shows the underlying execution flow of EtaGraph processing BFS.

B. Methodology

We conduct all experiments on a Linux server with two 2.50 GHz Intel 6-core, hyperthreaded Exon E5-2620 CPUs. The main memory is 128GB. An NVIDIA GTX 1080Ti GPU with 11GB GDDR5X memory is connected to this system.

Dataset	#vertices	#edges	Avg.Degree	Size(GB)	%LCC
Realistic Graph					
Slashdot [17]	77K	0.9M	11.7	0.011	98
LiveJournal [17]	5M	69M	14.2	1.1	99
com-Orkut [17]	3M	117M	38.1	1.7	99
uk-2005 [18]	39M	936M	23.7	16	65.2
sk-2005 [18]	50M	1, 949M	38.5	32	70.8
uk-2006 [18]	80M	2, 481M	30.7	42	71
Synthetic Graph					
RMAT25	32M	512M	32	8.3	81

TABLE II: Graph datasets used in evaluation. LCC stands for largest connected conpoment's percentage size of whole graph.

Our operating system is Ubuntu 16.04 with Linux kernel 4.15.0. We use the NVCC compiler version 9.0.176 (g++ version 5.4.0) to compile all the programs with flag O3.

Table II shows the datasets we use for all experiments. These datasets consist of both real-world graphs and a synthetic graph. The realistic graphs are 6 well-used social networks with edge numbers ranging from 0.9 million to 2.5 billion (the maximum outdegree ranging from 5.2K to 33K). To show performance generality of EtaGraph, we generate synthetic graph using PaRMAT [22] with parameter a=0.45, b=0.22, c=0.22. The size of datasets varies from 11MB to 42GB in human-readable edge lists format.

We compare EtaGraph with 3 state-of-the-art GPU processing systems: Cusha [2], Gunrock [3] and Tigr [6]. Cusha is an edge-centric framework optimized for memory coalescing. Gunrock is a frontier-based framework using high-level primitives. Tigr is a vertex-centric framework utilizing a similar degree-optimized transformation. Thus, we compare EtaGraph with these three systems without loss of generality.

We measure the total time of memory transfer (from main memory to GPU device memory) and kernel execution if no further notifications. We repeat experiment 5 times and the average value of the obtained results is reported.

We implement 3 well-used graph traversal algorithms: *breadth-first search*, *single-source shortest path*, *single-source widest path* to compare. For all algorithms, we start with the first source node of each dataset for fair comparison and make sure the queried traversal is untrivial. For all systems, graph datasets are transformed into their required data format in advance. For Cusha, we experiment its all three processing methods (G-Shards, Concatenated Windows, Virtual Warp-Centric) and report the best results for comparison.

C. Results and Analysis

In Table III, we present detailed performance results obtained from EtaGraph and other baseline frameworks. Table IV shows the percentage of active vertices and iteration numbers of EtaGraph.

Performance Comparison As Table III shows, EtaGraph achieves the best performance for most algorithms on a variety of datasets except for BFS on Slashdot. Beyond that, EtaGraph

Alg.	Frameworks	Runtime (ms) Slashdot	Livejournal	com-Orkut	RMAT25	uk-2005	sk-2005	uk-2006
BFS	Cusha	2.4/3.7	33/123	35/186	O.O.M	O.O.M	O.O.M	O.O.M
	Gunrock	3.4/3.9	56/109	78/162	607.9/1113	470/1127	O.O.M	O.O.M
	Tigr	**0.4/1.7**	18.5/87.8	20.9/126.6	294.4/814.9	810.8/1710	510/2578	O.O.M
	EtaGraph	2.5	**60**	**95**	**575**	**618**	**987**	1661
	EtaGraph w/o UMP	2.8	93	128	771	1412	1865	**1.3**
SSSP	Cusha	2.9/3.0	42/154	43/232	O.O.M	O.O.M	O.O.M	O.O.M
	Gunrock	7.4/8.3	306/409	418/584	2421/2997	1879/3101	O.O.M	O.O.M
	Tigr	–	22/100	25/213	335/1284	827/2284	O.O.M	O.O.M
	EtaGraph	**2.6**	**63**	**98**	**603**	**623**	**998**	1296
	EtaGraph w/o UMP	3.1	98	127	786	1409	1877	**1.3**
SSWP	Tigr	–	22/144	24.9/216	333/1274	824/1762	O.O.M	O.O.M
	EtaGraph	**2.6**	**63**	**98**	**600**	**626**	**1002**	1662
	EtaGraph w/o UMP	3.1	99	153	793	1431	1882	**1.3**

TABLE III: Performance Comparison. t_{kernel}/t_{total} are provided for other systems, '−' indicate that system crushed or do not get right results. O.O.M denotes system run out of GPU memory.

	Slashdot	Livejournal	Orkut	RMAT25	UK2005	sk-2005	uk-2006
Act. %	100	91	99	81	99	99	1.15E-04
Itr. #	8	15	8	9	200	57	4

TABLE IV: Activation and iteration details of EtaGraph

	Avg. Size (KB)	Min Size (KB)	Max Size (KB)
LJ w/o UMP	43.8	4	996
Orkut w/o UMP	44.3	4	924
rmat25 w/o UMP	44.3	4	964
uk2005 w/o UMP	48.9	4	996
LJ	1974	504	2048
Orkut	1993	1024	2048
rmat25	2048	2048	2048
uk2005	1998	544	2048

TABLE V: Size of migrated pages

simply outperforms Cusha, Tigr, Gunrock in all three graph traversal algorithms. In particular, EtaGraph can yield up to 6.5× speedup on SSSP over Gunrock, upto 2.4× speedup on SSSP over Cusha and upto 3.6× speedup on SSSP over Tigr. Note that the total time of EtaGraph is even smaller than the kernel execution time of Tigr for SSSP on uk-2005 dataset and Gunrock on several datasets. It is evident that EtaGraph has performance advantage over Gunrock and Tigr, even if higher-bandwidth CPU-GPU interconnect (NVLink [23], etc.) is equipped.

On small Slashdot, EtaGraph is slightly slower than Tigr. This happens because vertex number of Slashdot is so small that the overhead of invoking on-the-fly transformation of active set to virtual active set outweighs the efficiency gain of kernel execution. For Livejournal, Orkut, RMAT25 and sk-2005, EtaGraph achieve 1.4-2.5× speedup than the best of others. EtaGraph achieves up to 3.6× speedup over the best of others on SSSP for uk-2005. This is due to the numerous iterations that magnify the advantage of kernel execution efficiency with frontier-like active set. Performance of EtaGraph w/o UMP is no better than EtaGraph in almost all datasets expect for uk-2006. This is because traversal starting from queried vertex can only reach a small part of the whole graph for uk-2006 and only a small part of data need to be transferred to GPU while most of the vertices are visited in other datasets.

Memory Usage Analysis As shown in Table I, CSR is the most space-efficient among several familiar data structures. By directly using CSR, EtaGraph requires the least main memory and GPU device memory for storing graph topology data. EtaGraph also does not require auxiliary data to help traversal processing expect for storing active set. When the size of graph datasets grows, Cusha, Gunrock and Tigr successively throw an out-of-memory error. EtaGraph, however, can store all data in GPU device memory. Note that the total size of raw data and label data of uk-2006 is larger than GPU memory capacity. Thanks to migration and *oversubscription* of UM supported by Pascal generation architecture GPU, EtaGraph can process graph traversal on graphs larger than GPU memory capacity.

We also investigate the characteristics of Unified Memory. Fig. 4 shows that EtaGraph overlaps data-transfer and computation for the first 60%-80% of the time when *cudaMemPrefetchAsync* is disabled. Processing UK-2005 is different in that data-transfer occurs several times. This is because the first part of data-transfer does not transfer all data as no active vertices in some part of data yet.

The Unified Memory driver processes page faults and merge smaller page requests into larger page requests on the GPU. Table V shows the size of migrated pages. When Unified Memory Prefetch is disabled, the sizes are down to 4KB (the system memory page size) and have an average value around 44KB. If we enable *cudaMemPrefetchAsync*, the page sizes mostly are 2MB. Thus, EtaGraph w/o UMP needs more frequent migrations to transfer all data used in the traversal. This could be for this reason that EtaGraph is much faster when UMP is disabled, traversing a large part of graphs (though it is still comparable to state-of-the-art frameworks).

Performance Stability Fig. 5 shows the number of visited nodes over time. Notice that EtaGraph traversal vertices on nearly linear growth expect for Slashdot because its tiny size

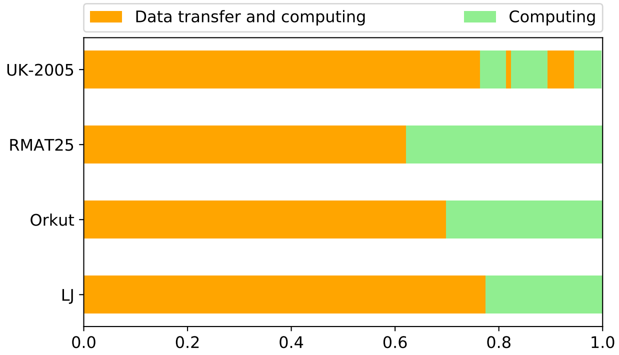
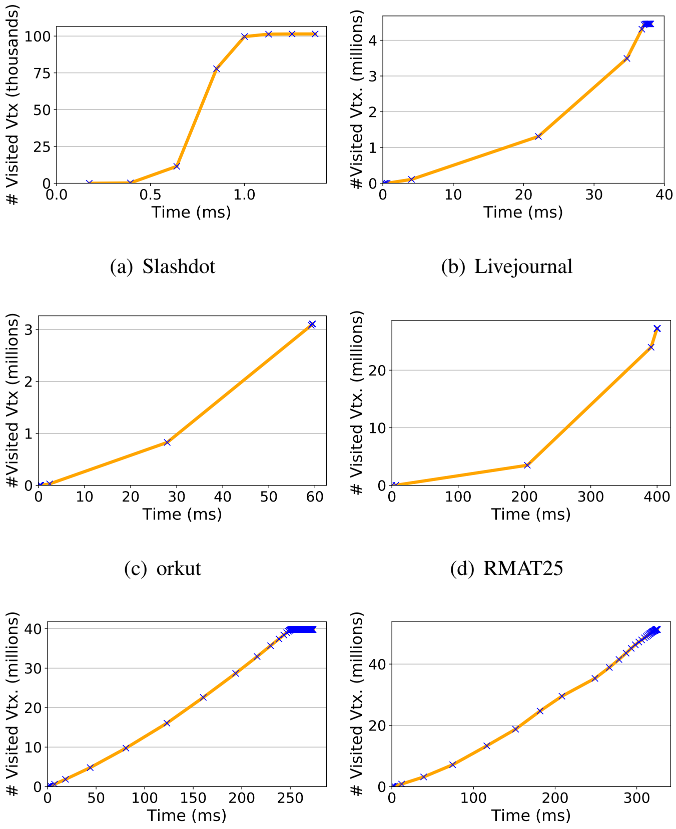


Fig. 4: Execution status during EtaGraph w/o UMP running SSSP. Data-transfer and computing are overlapped for about 60%-80% of time



(a) Slashdot

(b) Livejournal

(c) orkut

(d) RMAT25

(e) UK2005

(f) sk-2005

Fig. 5: Number of visited vertices over time. The number grows nearly linearly over time regardless the number of active vertices of each iteration.

only needs a few iterations. This shows that EtaGraph can efficiently process various graph datasets, regardless of the stage of traversal (i.e. number of active vertices). Thus, the performance of EtaGraph is consistent.

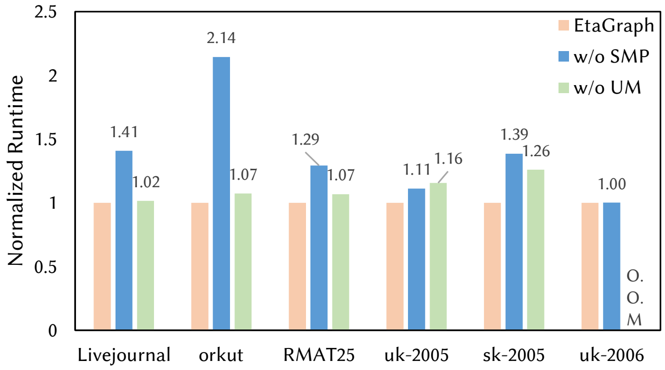


Fig. 6: Normalized runtimes of several EtaGraph setups

Performance Breakdown To illustrate the source of performance gain, we experiment with several versions of EtaGraph and record their total runtimes. The 'w/o SMP' is the one with Shared Memory Prefetch disabled. The 'w/o UM' is the one using normal *cudaMalloc* rather than Unified Memory. Fig. 6 reports their runtime on different datasets.

The runtime of 'w/o SMP' is 1.11-2.14× more than EtaGraph's except on the first 5 datasets. It is almost identical for uk-2006 due to little traversal time compared to data-transfer time. As we can see, SMP contributes to 1.11-2.14× overall speedup in EtaGraph. Similarly, adopting UM contributes 1.02-1.26× overall speedup and allow to process uk-2006, which is larger than GPU memory capacity. Speedup over other frameworks also comes from our space-efficient data layout and execution efficiency inside EtaGraph.

To better show the effect of SMP, we measure several metrics (IPC, Unified Cache hit rate, L2 cache hit rate, several memory throughputs, and global memory transactions) of BFS on LiveJournal with and without SMP. Unified Cache is the combination of L1 and Texture cache for Maxwell and Pascal architecture GPU. All results are generated using *nvprof* [24]. As Fig. 7 shows, adopting SMP alone achieves 1.42× IPC, 1.02× Unified Cache hit rate, 1.19× L2 cache hit rate and achieve 2.2× throughput on L2, Unified Cache and global memory. Adopting SMP also let the traversal kernel issue only 0.48× global memory transactions.

VII. Related Work

A. CPU-based Graph Processing

Graph processing has been extensively studied on the CPU-based system. Google first introduced the vertex-centric graph processing model and Pregal system [25]. From then on, a lot of graph processing systems appear, such as Giraph [26], GraphLab [4], and PowerGraph [27]. These platforms run on distributed system to take advantage of many CPU cores to process parallelly and large distributed memory to store graph data. Lots of works focus on efficient partition method over distributed machines to reduce communication overhead and improve load-balancing. GraphX [28] is a graph processing framework built on top of Apache Spark [29]. With optimization of distributed join and data-flow execution model, GraphX achieves fault tolerance.

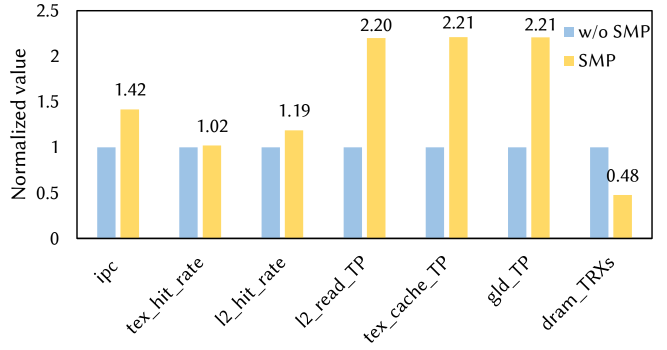


Fig. 7: Normalized ipc, Unified Cache hit rate, L2 cache hit rate and throughput of L2 read, Unified Cache, global memory read (higher is better) and global memory read transactions (lower is better).

There are also numerous frameworks based on shared memory machines (e.g., NUMA). Ligra [5] implements vertex and edge mapping routines to process a certain subset of graphs. Galois [30] uses domain-specific languages (DSLs) for high-level programming. Gemini uses a chunk-based partitioning scheme to facilitate locality of graph on both shared-memory and distributed systems. A few works like Graphchi [31], X-Stream [16] focus on using low-cost PCs to process large graphs storing in hard disks or flash memory. Pan et al. investigated to schedule multiple graph processing queries on shared-memory machines for better system throughput [32].

Graph traversal system has been studied in prior work [33]. It uses low-overhead data compression to reduce data-transfer volumes among machines along with latency-hiding optimization. There are also designs explicitly optimized for specific graph algorithms, such as BFS [34] and SSSP [35].

B. GPU Graph Processing Frameworks

Harish et al. treated GPU as an SIMD processor array to process graph traversal [36]. Merrill et al. used prefix sum to construct fine-grained tasks and leveraged vertex/edge frontier to store vertices (or edges) to be processed [19]. Totem [8] system is a GPU-CPU hybrid platform for graph processing. It uses CPU to handle high-degree nodes for fast sequential processing and allocate numerous low-degree nodes on GPU for massive parallelism processing. MapGraph [37] combines three different scheduling strategies together and dynamically choose the most suitable one for each vertex based on its degree. Cusha [2] uses two data structures, named G-Shards and Concatenated Windows (CW), to avoid non-coalesced memory access and improve memory bandwidth. Using the similar idea of *frontier*, Gunrock [3] performs operations on the frontier with data-centric abstraction. It uses two workload mapping strategies, e.g. per-thread fine-grained, per-warp and per-CTA coarse-grained. Tigr [6] proposes a virtual transformation to transform skewed graphs into *virtual vertices* for efficient processing.

There are designs explicitly optimized for multi-GPU graph processing. For example, GTS [11] stores attribute data at device and streams topology data among GPUs, allowing GTS process large graphs. Lux [38], a distributed system accelerated by GPUs, uses a dynamic graph repartitioning mechanism to achieve load-balancing among GPUs. Groute [9] allows multi-GPU asynchronous execution with ring-topology communication among GPUs.

Several GPU graph processing frameworks [11] [12] try to overlap data-transfer and kernel execution. They use several CUDA streams to transfer and process data chunks. Size of data chunks are preset. They need to transfer intact data chunks regardless of how much data are actually needed.

Some studies explore prefetch mechanism on GPU for graph application. Lakshminarayana et al. proposed to use modified GPU hardware and compiler to prefetch graph data into unused register [15]. Koo et al. proposed to use a hardwired prefetcher and scheduler to assist prefetching [39] for several applications including BFS.

VIII. Conclusion

In this paper, we analyze the inefficiency of kernel execution and memory usage of graph traversal on GPU. We present a graph transformation policy to transform vertices in skewed graphs without the need for pre-processing or modification of raw data. This procedure is space-efficient and lightweight even on the fly. We combine the graph transformation with frontier-like workload mapping mechanism. We also propose shared memory prefetch, a lightweight yet efficient mechanism to explicitly utilizing shared memory of GPU and improve effective GPU memory bandwidth. SMP can be easily applied to other vertex-centric frameworks. We synergistically combine these optimizations in a graph traversal framework named EtaGraph. Exhaustive evaluations show that EtaGraph can achieve significant speedup over several state-of-the-art frameworks on representative graph datasets.

Acknowledgment

This work is supported by the National Key Research and Development Program of China (NO. 2018YFB1003503). Corresponding author is Chao Li from Shanghai Jiao Tong University.

References

[1] 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. New York, NY, USA: ACM, 2012, pp. 117–128. [Online]. Available: http://doi.acm.org/10.1145/2145816.2145832

[2] F. Khorasani, K. Vora, R. Gupta, and L. N. Bhuyan, “Cusha: vertex-centric graph processing on gpus,” in *The 23rd International Symposium on High-Performance Parallel and Distributed Computing, HPDC’14, Vancouver, BC, Canada - June 23 - 27, 2014*, 2014, pp. 239–252. [Online]. Available: http://doi.acm.org/10.1145/2600212.2600227

[3] Y. Wang, A. A. Davidson, Y. Pan, Y. Wu, A. Riffel, and J. D. Owens, “Gunrock: a high-performance graph processing library on the gpu,” in *PPOPP*, 2015.

[4] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein, “Graphlab: A new framework for parallel machine learning,” in *UAI 2010, Proceedings of the Twenty-Sixth Conference on Uncertainty in Artificial Intelligence, Catalina Island, CA, USA, July 8-11, 2010*, 2010, pp. 340–349.

[5] J. Shun and G. E. Blelloch, "Ligra: a lightweight graph processing framework for shared memory," in *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '13, Shenzhen, China, February 23-27, 2013*, 2013, pp. 135–146. [Online]. Available: http://doi.acm.org/10.1145/2442516.2442530

[6] A. H. N. Sabet, J. Qiu, and Z. Zhao, "Tigr: Transforming irregular graphs for gpu-friendly graph processing," in *ASPLOS*, 2018.

[7] Nvidia, "Nvidia Tesla V100 GPU Architecture," *White Paper*, no. v1.1, p. 53, 2017. [Online]. Available: http://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf%0Ahttp://www.nvidia.com/content/gated-pdfs/Volta-Architecture-Whitepaper-v1.1.pdf

[8] 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. [Online]. Available: http://doi.acm.org/10.1145/2370816.2370866

[9] 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, Austin, TX, USA, February 4-8, 2017*, 2017, pp. 235–248. [Online]. Available: http://dl.acm.org/citation.cfm?id=3018756

[10] 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. New York, NY, USA: ACM, 2016, pp. 403–416. [Online]. Available: http://doi.acm.org/10.1145/2882903.2882959

[11] M. 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, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, 2016, pp. 447–461.

[12] W. Han, D. Mawhirter, B. Wu, and M. Buland, "Graphie: Large-scale asynchronous graph traversals on just a GPU," in *26th International Conference on Parallel Architectures and Compilation Techniques, PACT 2017, Portland, OR, USA, September 9-13, 2017*, 2017, pp. 233–245.

[13] S. Ryoo, C. I. Rodrigues, S. S. Stone, S. S. Baghsorkhi, S. Ueng, J. A. Stratton, and W. W. Hwu, "Program optimization space pruning for a multithreaded gpu," in *Sixth International Symposium on Code Generation and Optimization (CGO 2008), April 5-9, 2008, Boston, MA, USA*, 2008, pp. 195–204.

[14] J. Lee, N. B. Lakshminarayana, H. Kim, and R. W. Vuduc, "Many-thread aware prefetching mechanisms for GPGPU applications," in *43rd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2010, 4-8 December 2010, Atlanta, Georgia, USA*, 2010, pp. 213–224.

[15] N. B. Lakshminarayana and H. Kim, "Spare register aware prefetching for graph algorithms on gpus," in *20th IEEE International Symposium on High Performance Computer Architecture, HPCA 2014, Orlando, FL, USA, February 15-19, 2014*, 2014, pp. 614–625.

[16] A. Roy, I. Mihailovic, and W. Zwaenepoel, "X-stream: edge-centric graph processing using streaming partitions," in *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*, 2013, pp. 472–488. [Online]. Available: http://doi.acm.org/10.1145/2517349.2522740

[17] J. Yang and J. Leskovec, "Defining and evaluating network communities based on ground-truth," in *12th IEEE International Conference on Data Mining, ICDM 2012, Brussels, Belgium, December 10-13, 2012*, 2012, pp. 745–754. [Online]. Available: https://doi.org/10.1109/ICDM.2012.138

[18] P. Boldi and S. Vigna, "The WebGraph framework I: Compression techniques," in *Proc. of the Thirteenth International World Wide Web Conference (WWW 2004)*. Manhattan, USA: ACM Press, 2004, pp. 595–601.

[19] D. Merrill, M. Garland, and A. Grimshaw, "Scalable gpu graph traversal," *SIGPLAN Not.*, vol. 47, no. 8, pp. 117–128, Feb. 2012. [Online]. Available: http://doi.acm.org/10.1145/2370036.2145832

[20] C. Nvidia, "Programming guide," 2010.

[21] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. P. Chew, "Optimistic parallelism requires abstractions," in *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007*, 2007, pp. 211–222.

[22] F. Khorasani, R. Gupta, and L. N. Bhuyan, "Scalable simd-efficient graph processing on gpus," in *Proceedings of the 24th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '15, 2015, pp. 39–50.

[23] D. Foley and J. Danskin, "Ultra-performance pascal gpu and nvlink interconnect," *IEEE Micro*, vol. 37, no. 2, pp. 7–17, 2017.

[24] C. NVidia, "Cuda profiler users guide (version 6.5): Nvidia," *Santa Clara, CA, USA*, vol. 87, 2014.

[25] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," in *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, June 6-10, 2010*, 2010, pp. 135–146. [Online]. Available: http://doi.acm.org/10.1145/1807167.1807184

[26] C. Avery, "Giraph: Large-scale graph processing infrastructure on hadoop," *Proceedings of the Hadoop Summit. Santa Clara*, vol. 11, no. 3, pp. 5–9, 2011.

[27] J. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "Powergraph: Distributed graph-parallel computation on natural graphs," in *OSDI*, 2012.

[28] 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, USA, October 6-8, 2014.*, 2014, pp. 599–613.

[29] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *NSDI*, 2012.

[30] D. Nguyen, A. Lenharth, and K. Pingali, "A lightweight infrastructure for graph analytics," in *Proceedings of ACM Symposium on Operating Systems Principles*, ser. SOSP '13, 2013, pp. 456–471. [Online]. Available: http://iss.ices.utexas.edu/Publications/Papers/nguyen13.pdf

[31] A. Kyrola, G. E. Blelloch, and C. Guestrin, "Graphchi: Large-scale graph computation on just a pc," in *OSDI*, 2012.

[32] P. Pan and C. Li, "Congra: Towards efficient processing of concurrent graph queries on shared-memory machines," in *2017 IEEE International Conference on Computer Design, ICCD 2017, Boston, MA, USA, November 5-8, 2017*, 2017, pp. 217–224. [Online]. Available: https://doi.org/10.1109/ICCD.2017.40

[33] N. Satish, C. Kim, J. Chhugani, and P. Dubey, "Large-scale energy-efficient graph traversal: a path to efficient data-intensive supercomputing," in *SC Conference on High Performance Computing Networking, Storage and Analysis, SC '12, Salt Lake City, UT, USA - November 11 - 15, 2012*, 2012, p. 14.

[34] S. Beamer, K. Asanovic, and D. A. Patterson, "Direction-optimizing breadth-first search," in *SC Conference on High Performance Computing Networking, Storage and Analysis, SC '12, Salt Lake City, UT, USA - November 11 - 15, 2012*, 2012, p. 12.

[35] A. Crauser, K. Mehlhorn, U. Meyer, and P. Sanders, "A parallelization of dijkstra's shortest path algorithm," in *International Symposium on Mathematical Foundations of Computer Science*. Springer, 1998, pp. 722–731.

[36] P. Harish and P. J. Narayanan, "Accelerating large graph algorithms on the GPU using CUDA," in *High Performance Computing - HiPC 2007, 14th International Conference, Goa, India, December 18-21, 2007, Proceedings*, 2007, pp. 197–208.

[37] Z. Fu, H. K. Dasari, B. R. Bebee, M. Berzins, and B. B. Thompson, "Mapgraph - graphprocessing at 30 billion edges per second on nvidia gpus," in *International Semantic Web Conference*, 2014.

[38] Z. Jia, Y. Kwon, G. M. Shipman, P. S. McCormick, M. Erez, and A. Aiken, "A distributed multi-gpu system for fast graph processing," *PVLDB*, vol. 11, no. 3, pp. 297–310, 2017. [Online]. Available: http://www.vldb.org/pvldb/vol11/p297-jia.pdf

[39] G. Koo, H. Jeon, Z. Liu, N. S. Kim, and M. Annavaram, "Cta-aware prefetching and scheduling for GPU," in *2018 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2018, Vancouver, BC, Canada, May 21-25, 2018*, 2018, pp. 137–148.