Traversing Large Graphs on GPUs with Unified Memory

Prasun Gera
Georgia Tech
prasun.gera@gatech.edu

Hyojong Kim
Georgia Tech
hyojong.kim@gatech.edu

Piyush Sao
Oak Ridge National Lab
saopk@ornl.gov

Hyesoon Kim
Georgia Tech
hyesoon@cc.gatech.edu

David Bader*
New Jersey Institute of Tech
david.bader@njit.edu

ABSTRACT

Due to the limited capacity of GPU memory, the majority of prior work on graph applications on GPUs has been restricted to graphs of modest sizes that fit in memory. Recent hardware and software advances make it possible to address much larger host memory transparently as a part of a feature known as unified virtual memory. While accessing host memory over an interconnect is understandably slower, the problem space has not been sufficiently explored in the context of a challenging workload with low computational intensity and an irregular data access pattern such as graph traversal. We analyse the performance of breadth first search (BFS) for several large graphs in the context of unified memory and identify the key factors that contribute to slowdowns. Next, we propose a lightweight offline graph reordering algorithm, HALO (Harmonic Locality Ordering), that can be used as a pre-processing step for static graphs. HALO yields speedups of 1.5x-1.9x over baseline in subsequent traversals. Our method specifically aims to cover large directed real world graphs in addition to undirected graphs whereas prior methods only account for the latter. Additionally, we demonstrate ties between the locality ordering problem and graph compression and show that prior methods from graph compression such as recursive graph bisection can be suitably adapted to this problem.

PVLDB Reference Format:
Prasun Gera, Hyojong Kim, Piyush Sao, Hyesoon Kim, and David Bader. Traversing Large Graphs on GPUs with Unified Memory. *PVLDB*, 13(7): 1119-1133, 2020.
DOI: https://doi.org/10.14778/3384345.3384358

1. INTRODUCTION

Graphics Processing Units (GPUs) have been used successfully for accelerating graph based applications. These applications are both interesting and challenging for GPUs in that there is often ample parallelism in the algorithm, but the data access pattern tends to be highly irregular. Graph applications leverage thread-level parallelism (TLP) and the high bandwidth afforded by the GPU's internal memory to hide long latency operations. Real world graphs such as social networks, web graphs, and biological networks tend to have sizes in the order of tens to hundreds of gigabytes. However, GPUs have traditionally only had accesses to local memory that is in the range of a few gigabytes. The majority of prior work on GPUs and graph processing deals only with graphs that fit in GPU memory. While there is some work on distributed multi-GPU graph processing, the working assumption is still that the graphs fit in the collective memory of the GPUs.

Recent GPUs support unified virtual memory (UVM) between CPUs and GPUs. UVM simplifies a lot of programming abstractions by presenting a unified memory space to the programmer, and it also supports oversubscription of GPU memory. GPU memory can be oversubscribed as long as there is sufficient host memory to back the allocated memory. The driver and the runtime system handle data movement between CPUs and GPUs transparently (i.e., without the programmer's involvement). UVM allows us to run GPU applications that may otherwise be infeasible due to the large size of datasets. The performance impact of UVM, specifically with regard to oversubscription, has not been studied extensively. We found that using unified memory naïvely for graph applications leads to a severe performance penalty. With the increasing popularity of accelerators in solving domain specific problems, we believe that the problem of managing data movement efficiently between devices will become an important one in the future. Prior works that improve the locality or other memory access characteristics in graph applications span a few different approaches. On the theoretical side, there are external memory algorithms [4] that broadly model such systems. Some of these ideas have also been incorporated in disk based implementations [46]. A common theme in graph computing is to organise the underlying graph data in a particular way to improve performance. For example, graph partitioning is an important consideration in the distributed setting where the goal is to improve load balance and reduce communication costs. The data structures used for representing graphs also impact performance, and different graph structures [20, 53] as well as compression schemes [13] have been proposed. Graph ordering is another area with performance implications for graph applications and sparse

*Contributed while at Georgia Tech

Proceedings of the VLDB Endowment, Vol. 13, No. 7
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3384345.3384358

linear algebra. The focus of this paper is graph reordering for static graphs. We use breadth first search (BFS) as a representative graph traversal kernel. BFS has been studied extensively and is also equivalent to sparse matrix vector multiplication (SpMV) [42], which lets us compare relevant works from sparse linear algebra. It is also used as a building block in other applications such as betweenness centrality and strongly connected components. It captures the key properties of irregularity and unpredictability of future memory accesses, which are our main areas of interest in the UVM context. We found that prior reordering solutions such as RCM [26, 34], which generally perform well for sparse symmetric matrices, do not perform well for large directed graphs. Other prior methods such as Gorder [71] are too expensive to be feasible and also do not outperform RCM significantly for BFS. This paper makes the following contributions:

- We propose a new graph reordering method, HALO (Harmonic Locality Ordering) (Sec. 7), that targets improvements to the data locality and the data transfer volume for traversals on GPUs in a semi-external memory model such as the one with UVM.
- Real world graphs are often directed and disconnected, which makes locality optimisations difficult. HALO improves the locality of arbitrary BFS traversals over a wide range of real and synthetic graphs. To our knowledge, this is the first work that looks at graph reordering in the context of GPUs and unified memory.
- HALO performs better than prior methods such as RCM and can be parallelised and approximated efficiently.
- Traversals on reordered graphs show speedups in the range of 1.52x-1.9x (Sec. 10). We also identify some additional optimisations that reduce unnecessary coherence traffic and lead to an additional speedup of 1.85x (Sec. 10.4).
- We show that the problem's formulation (Sec. 5) leads to a natural overlap with the graph compression problem. We create an additional ordering method, BFS-BP (Sec. 8), that adapts techniques from graph compression such as recursive bisection to this problem.

2. MOTIVATION

In recent years, several works have proposed methods for doing efficient graph processing on GPUs. Breadth first search (BFS) in particular is often studied as a representative kernel and is also included in benchmarks such as Parboil [67], Rodinia [24], and Graph500 [59]. The core BFS kernel is also used as a building block in analytics applications such as betweenness centrality [19], cycle detection, maximum flow, strongly connected components [32], and others. Several prior works [38, 39, 56, 52, 11, 51] and GPU graph frameworks [70, 36, 60] cover different optimisations that deal with issues arising from load imbalance, uncoalesced memory accesses, or redundant work. The majority of these proposals assume that the graphs fit in GPU memory. For larger graphs, there are some works that distribute the graph over multiple GPUs [61, 55] or between the CPU and the GPU [35].

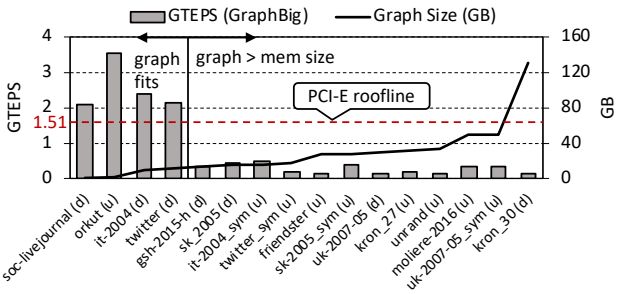


Figure 1: BFS performance measured in billions of Traversed Edges per Second (GTEPS) on a Titan Xp GPU

Table 1: GPU Bandwidth Characteristics

GPU	Mem.	HtoD Link	DtoD BW	HtoD BW
Titan Xp	12 GB	PCI-e 3.0	417.4 GB/s	12.1 GB/s

Recent GPUs support unified virtual memory (UVM) between multiple GPUs and CPUs. The feature is now supported by CUDA [66], OpenCL [2], and kernel drivers [1]. UVM presents a unified abstraction for memory management between several devices, and it supports oversubscription of device memory. The driver manages on-demand migration of pages transparently between devices. UVM is attractive for development since the changes required are generally not very invasive for applications, and it makes working with datasets larger than a single GPU's memory capacity feasible. While applications such as deep learning can deal with datasets larger than GPU memory, they are explicitly designed to work in a pipelined fashion wherein small batches of data are sent to the GPU at a time, which in turn can be overlapped with computation. This approach does not map naturally to irregular graph applications since the data needed in the future is often not known beforehand. Further, an application like BFS has low computational intensity which makes it difficult to overlap computation with data transfers. Hence, most of the prior work on GPUs and graph computing deals only with graph sizes that fit in GPU memory. UVM makes it possible to use existing GPU graph frameworks for large graphs without extensive changes.

While UVM makes it easy to overprovision memory, the performance is quite poor if we take an application like BFS and change its `cudaMalloc` calls to `cudaMallocManaged` calls. In Fig. 1, we show BFS's performance results, measured in billions of traversed edges per second (GTEPS), across different graph sizes after minimally modifying GraphBig [60] to take advantage of UVM. The results are accumulated over fifty traversals from random sources for each graph. We see that performance drops off sharply when the graph sizes exceed the GPU's memory capacity of 12 GB. The Titan Xp GPU supports a peak transfer bandwidth of 12.11 GB/s over PCI-e 3.0. We use 64-bit data types, which gives us a theoretical peak GTEPS rate of 1.51. This is a theoretical rate for just transferring the data linearly whereas an application like BFS does highly irregular work as well. We see that the average GTEPS rate for BFS in the UVM region is 0.24, which is much lower than the theoretical peak, and we would like to improve its performance.

It is also worth noting that this is a memory bound problem where current optimised parallel CPU implementations (e.g., [28]) will likely outperform UVM due to higher DRAM bandwidth. However, faster interconnects like NVLINK and OpenCAPI, and newer versions of PCI-e have much higher bandwidth and are making their way to mainstream products. Additionally, the traversal itself may be a part of a larger GPU accelerated pipeline with higher computational complexity. Newer architectures with non-volatile memory or storage attached directly to CPUs or GPUs will face similar challenges. We view this as a forward looking problem that motivates and prepares for advances in memory systems, interconnects, and heterogeneous computing. Understanding a single GPU with UVM paves the way for more complex heterogeneous systems with multiple GPUs and/or CPUs and a common pool of addressable memory. To that end, we would like to answer the following questions:

- What are the primary factors that impact the performance in the UVM model?
- How is the UVM memory hierarchy different from other memory hierarchies?
- How can we improve the performance of graph traversals in the context of UVM?

3. BACKGROUND

3.1 Graph Storage

We use the compressed sparse row (CSR) format for representing graphs. Since real world graphs tend to be sparse, the CSR format, which stores the non-zero elements instead of the entire adjacency matrix, is a suitable choice. The CSR format is also popular for static graphs as one can leverage high performance sparse linear algebra libraries. Dynamic graphs, which are outside the scope for this work, would benefit from a different representation such as the one used by Hornet [22]. In the CSR data format, a graph $G = (V, E)$ is stored as two arrays. Hereafter, we denote $|V|$ as n and $|E|$ as m. The first array, `vlist` of length $(n+1)$, consists of row offsets, and the second array, `elist` of length m, consists of column indices. For a given vertex id v, its neighbours can be accessed in the range `elist`[`vlist`[v] : `vlist`[$v+1$]]. Other optional arrays can be used for storing properties such as edge weights. An application may also use additional structures for storing the output or intermediate results. In our implementation, BFS uses a `level` array of size n for storing the level numbers and optionally a `pred` array for storing the predecessors. We use 64 bit types for all the structures to account for the most general massive graphs. In practice, it is possible to use either 32 bit types or a combination of 32 bit and 64 bit types for smaller graphs. Fig. 2 shows a sample graph and its corresponding CSR representation.

3.2 BFS

Breadth first search (BFS) is a fundamental graph traversal algorithm that is used as a building block in other graph algorithms. The algorithm starts graph traversal from a given source and visits other vertices in the graph in level order. The algorithm has a serial complexity of $\mathcal{O}(n + m)$. BFS has been studied extensively, and there are optimised implementations for GPUs [38, 39, 56, 52, 11, 51]. Further, the problem can also be expressed as iterative sparse matrix vector multiplication (SpMV) [42, 21]. A simple GPU kernel that traverses one level in a top-down BFS is presented in Alg. 1. Levels are initialised to ∞ for all vertices except the source, which is initialised to 0. A central part of the BFS algorithm is the notion of a frontier. The vertices in the current frontier explore their neighbours and add them to the next frontier. This step at line 5 in the algorithm is responsible for bringing in data from the CPU to the GPU in the UVM setting. Note that the level array is serving as an implicit frontier here, which is also the vector when you see BFS as SpMV. It is possible to use different data structures for the frontier along with different load balancing strategies. These considerations are orthogonal to this work and can be applied independently. We are mainly concerned with memory accesses in the `elist` array at line 5, which remain unchanged as they are a function of the graph's topology and ordering.

Algorithm 1 BFS_Advance (G, level, pred, curr, stop)

```
1: for all v ∈ V in parallel do
2:   if level[v] == curr then
3:     start_idx = vlist [v], end_idx = vlist [v+1]
4:     for all idx ∈ [start_idx, end_idx) in parallel do
5:       nbr = elist[idx]
6:       old_val = atomic_min(&level[nbr], curr + 1)
7:       if old_val == ∞ then
8:         pred[nbr] = v
9:         stop = false
```

Table 2: Graphs used in this work

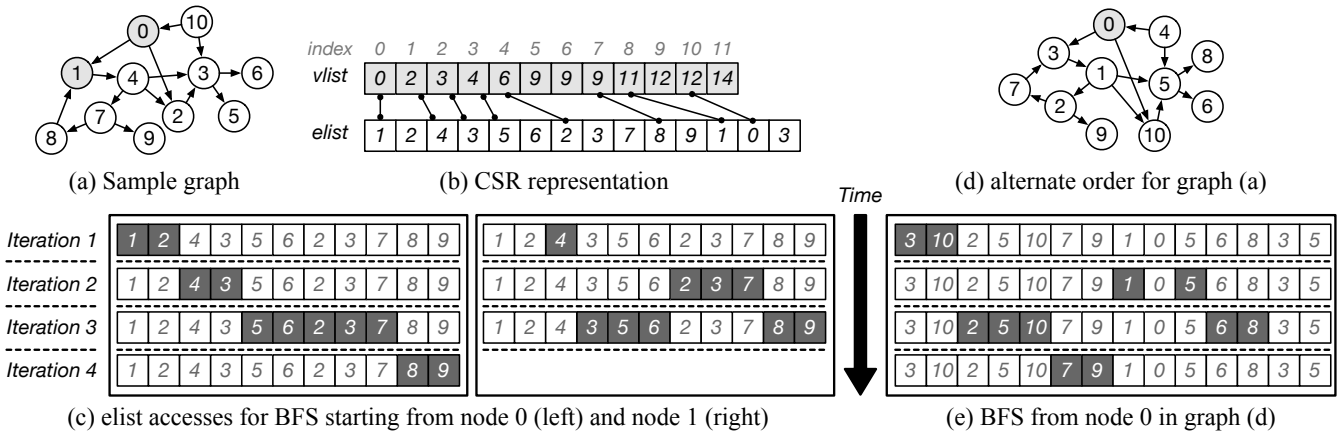
Graph	Category	Vertices	Edges
soc-livejournal (d)	Social Net	4.85 M	68.99 M
orkut (u)	Social Net	3.07 M	234.37 M
friendster (u)	Social Net	65.61 M	3.61 B
twitter (d)	Social Net	41.65 M	1.47 B
it-2004 (d)	Web Crawl	41.2 M	1.15 B
gsh-2015-h (d)	Web Host	68.66 M	1.80 B
sk-2005 (d)	Web Crawl	65.61 M	1.95 B
uk-2007 (d)	Web Crawl	105.22 M	3.74 B
moliere-2016 (u)	Biomedical	30.22 M	6.68 B
kron_(27,28,29,30)	Kronecker	2^{scale}	$\|V\|$ x 16
unrand (u)	Erdős-Rényi	134.22 M	4.29 B

3.3 Model Assumptions

We assume a sem-external memory model in this work. We cover graphs whose $\mathcal{O}(n)$ structures such as `vlist` and the `level` array can fit in GPU memory whereas $\mathcal{O}(m)$ structures such as `elist` do not. While unified memory does not preclude much larger graphs where even $\mathcal{O}(n)$ structures do not fit in the GPU's memory, such problems will likely need additional work to achieve efficient solutions. For general external memory BFS, we refer the reader to theoretical work in the area [4]. With current GPU memory capacity trends, the semi-external model still scales to billion node graphs. All the graphs are treated as unweighted since we are primarily concerned with BFS.

3.4 Graph Selection

We use several large real and synthetic graphs spanning web crawls [17, 15, 14], social networks [48], and a biological hypothesis network [68]. These graphs generally have a



(a) Sample graph (b) CSR representation (d) alternate order for graph (a)

(c) elist accesses for BFS starting from node 0 (left) and node 1 (right) (e) BFS from node 0 in graph (d)

Figure 2: (a) A sample graph, (b) its CSR representation, (c) memory access pattern in `elist` for BFS from two different sources; (d) an alternate order for the graph, and (e) access pattern in the alternate order for traversal from node 0

small diameter, high clustering, and scale-free degree distributions. As a counterpoint, we also include one uniformly random Erdős-Rényi graph [63], where our goal is to show that locality based optimisations do not extend to all graphs. Our graphs include a mix of directed and indirected graphs. For the directed graphs, we also show results for symmetrised version of the same graph. These graphs are noted with the *sym* suffix. Directed graphs are denoted with (d) whereas undirected ones are noted with (u).

4. DATA ACCESS PATTERN

Since the graphs that we want to work with are larger than the GPU's memory capacity, data is transferred between the host and device on demand at runtime. The bandwidth between the host and the GPU is much lower than the GPU's internal bandwidth. In Table 1, we can see that the external bandwidth is 34x lower than the internal bandwidth. Further, the effective bandwidth is inversely proportional to the chunk size of transfers. Since unified memory is a page fault handling mechanism, transfers happen at page granularity (4 KB) or multiples thereof. The runtime uses prefetching and batch processing of page faults to mitigate the high latency of transfers [66, 65, 73, 7]. However, this has side effects such as read/write amplification, thrashing, and evictions, which are exacerbated in the case of an irregular application like graph traversal. We look at some simple examples in our sample graph from Fig. 2 to highlight the factors that impact performance.

Dependence on the Source Node: Consider a BFS from two different source nodes, 0 and 1, in the sample graph from Fig. 2(a). We show the progress of the traversal in Fig. 2(c). The shaded blocks in `elist` are the ones that are accessed at each level in line 5 of Alg. 1. The first BFS from node 0 is the ideal case for UVM as all the accesses in `elist` move contiguously in one direction with time. This leads to fewer page transfers, good page replacement decisions, and predictable prefetching. However, the second traversal from node 1 accesses non-contiguous locations. This is a small example, but such accesses can span several pages in a large graph and result in multiple pages being transferred with a low ratio of useful data, thrashing, and poor accuracy for the prefetcher. We make the following observation: The source node in a BFS affects the locality of accesses, but we have no control over the source. Our goal is to optimise traversals from arbitrary source nodes.

Dependence on Graph Ordering: Consider a different ordering of the same sample graph as show in Fig. 2(d) and a BFS traversal from source node 0. The access pattern of words in `elist`, as seen in Fig. 2(e), is very different from the contiguous pattern in the original order although it is the same traversal. We make the following observation: A graph's ordering affects the locality of accesses in a BFS traversal. If a frontier's nodes are close to each other in their labels, the exploration of their neighbourhoods would have good locality in `elist` in the next iteration.

Dependence on Directedness: Directed edges and the structure of the graph affect locality of accesses in `elist`. For example, a traversal from node 5 in the original graph would not explore any additional nodes, but would traverse the entire graph in the undirected version since there is only one connected component. This is an important distinction because an ordering that optimises undirected graphs does not necessarily optimise directed ones. We found that degree based metrics alone are inadequate for directed graphs. In directed graphs, not only do we have separate in and out degrees, neither is a good indicator of broader reachability since the degree only looks at connectivity one hop away.

5. PROBLEM FORMULATION

Given that BFS has low computational complexity, we are primarily memory bound, and the inefficiencies in the UVM model indicate that, from an algorithmic standpoint, the main avenues for improvement lie in reducing the volume of data transfer and improving the efficiency of transfers. There are broadly three ways to improve the performance:

1. Change the algorithm to do less work
2. Compress the graph
3. Reorder the graph

Direction optimising BFS [11] is a solution in the first category that reduces the number of edges visited. Such solutions can be applied independently. Graph compression has also been used successfully for large scale web and social graphs [17]. It is a promising direction, but decompressing graphs on the GPU is a challenging problem in its own right,

which we leave for future work. Finally, graph reordering as a means to improve the locality is the focus of this work. Based on the observations in Sec. 4, there are two desirable properties that we seek in an ordering:

- The labels of nodes *within* a frontier are close to each other
- The labels of nodes *across* successive frontiers are close to each other

When combined, these two properties lead to a pattern similar to the one we saw in Fig. 2(c). For the first property, we formulate the problem as follows: For a general directed unweighted graph, we define the MinIntraBFS ordering as the ordering that minimises the total cost

$$C = \sum_{\text{sources}} \sum_{\text{levels}} \sum_{i=1}^{|\text{level}|-1} c(\pi(u_{i+1}) - \pi(u_i)) \quad (1)$$

where π is the permutation function, $u_1..u_{|level|}$ are the nodes in each level of the traversal such that $\pi(u_i) < \pi(u_{i+1})$, and c is a cost function. That is, we seek to lower the gaps between node labels *within* any frontier from any source node in the graph. This formulation is quite close to other graph ordering problems such as the minimum linear (or log) arrangement problem (MLA/MinLogA) [33, 25] and the minimum Log Gap Arrangement (MinLogGapA) [25, 29] problem, which are NP-hard problems. The MinLogGapA problem seeks to create compression friendly graph orderings by way of reducing the gaps between labels of neighbours of nodes since smaller gaps can be encoded more efficiently. Recently, Dhulipala et al. [29] proposed a new model called the bipartite minimum logarithmic arrangement (BiMLogA) problem that generalises both MinLogA and MinLogGapA. If the cost function is identity or log, we can show that the MinIntraBFS problem is also NP-hard.

THEOREM 1. *MinIntraBFS (log cost) is NP-hard*

PROOF. We prove this by reducing an instance of the BiMLogA problem, which is NP-hard, to the MinIntraBFS problem. The BiMLogA [29] problem is defined as follows: Let $G = (Q \cup D, E)$ be an undirected unweighted bipartite graph with disjoint sets of vertices Q and D representing query and data vertices. The goal is to find a permutation, π, of data vertices, D, so that the following objective is minimised:

$$\sum_{q \in Q} \sum_{i=1}^{deg_q - 1} \log(\pi(u_{i+1}) - \pi(u_i)) \quad (2)$$

where deg_q is the degree of query vertex $q \in Q$, and q's neighbors are $u_1, \ldots, u_{deg_q}$ with $\pi(u_1) < ... < \pi(u_{deg_q})$. Given an instance of BiMLogA, create a new graph $G' = (Q \cup D, E')$, where each undirected edge in E becomes a one-way directed edge in E' from Q to D. Notice that solving MinIntraBFS on G' solves BiMLogA on G. This is because when a node in D is the source for a traversal, its out-degree is zero and does not contribute to the cost in C. When a node in Q is the source, each of the traversals is just one level deep with the source at the root level and its neighbours in D at the next. Since the cost looks at gaps between adjacent nodes in a level, the nodes from Q do not contribute to the cost as they are isolated in their respective root levels. Thus, the final cost is the same as the cost of BiMLogA for G. Since the nodes from Q do not appear in the cost, they can be moved away to one side in the permutation without increasing the total cost. Thus, we have solved BiMLogA for G, which proves the claim of the theorem. □

The identity cost version can be proved similarly. We can also reduce in the other direction. That is, we can go from MinIntraBFS to BiMLogA. We pursue this approach in more detail in Sec. 8, where we create an ordering method, BFS-BP, that optimises the MinIntraBFS cost function. BFS-BP is based on the recursive bisection algorithm, BP, proposed by Dhulipala et al. [29].

The second desirable property of reducing gaps *across* BFS levels is more challenging to formulate succinctly. For two successive levels of sizes $|p|$ and $|q|$, we get $|p||q|$ cross terms for pairwise gaps between nodes in these levels. Further, we may need to account for gaps beyond just successive levels. Accounting for inter-level gaps would lead to an explosion of terms. We do not pursue inter-level gaps further in this work. The actual cost is also different from a log or identity cost in practice. The cost function depends on the out-degree distribution in the graph as well as the memory hierachy's characteristics. This is also where the UVM hierarchy differs from, for example, cache and DRAM. The unit of data transfers in UVM is a page (4 KB). If two nodes are relatively close in their labels, their neighbourhoods in `elist` may fall on the same page. On the other hand, they may not fall on the same cache line. The cost does not grow proportionately with the gap. Rather, it behaves like a step function. Once we exceed the page boundary, two neighbourhood accesses will fall on two pages, no matter how far the labels end up being. Despite these simplifications, we show in Sec. 10.1 that the MinIntraBFS (log cost) model correlates well with overall performance and data transfer volume. Given the large granularity of transfers, we believe that our ordering methods would also benefit other similar architectures (e.g., non-volatile or solid state storage).

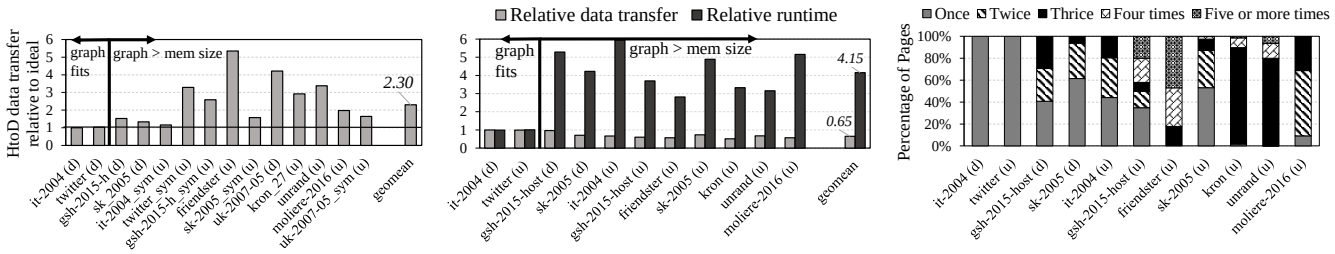
The main reordering algorithm proposed in this work, HALO (Harmonic Locality Ordering) (Sec. 7) uses a geometric measure for ordering instead of optimising the cost function directly. Nevertheless, it reduces the MinIntraBFS cost as well as application runtime (Sec. 10). The other proposed method, BFS-BP (Sec. 8), is based on optimising the cost function directly. All the reordering methods considered in this work are offline methods. Some prior works treat reordering as an online step [6], but their primary workloads have higher complexity than BFS. Any reordering scheme that scans the graph ends up being as expensive as BFS asymptotically making any proposal for online reordering rather difficult. An ideal reordering scheme in the context of this problem should strive to achieve similar costs as BFS to be practical.

Before we describe the proposed solutions, we look at a few performance metrics to better understand the inefficiencies in the UVM model.

6. PERFORMANCE

6.1 Read Amplification

We compare the volume of data transferred from the host to the GPU to the ideal volume that needs to be transferred for a traversal. The ideal volume is the number of traversed edges times the unit edge size. The ideal transfer size is



(a) Read Amplification **(b)** Performance with prefetch disabled **(c)** Evictions

Figure 3: BFS performance characteristics on a Titan Xp GPU with UVM. (a) Average read amplification of 2.3x; (b) When the prefetcher is disabled, performance suffers severely despite a reduction in data transfer volume; (c) Distribution of the number of times the same pages are fetched from the host to device due to evictions

bounded by the graph size, but it can be lower if the traversal does not touch the entire graph. We show the results in Fig. 3(a), where we see that on average we transfer 2.3x the ideal volume of data when we do traversals in the UVM model.

6.2 Prefetches

A portion of the total data transfer volume can be ascribed to prefetches. Since the interconnect has very high latency, GPU page faults are handled in batches, which are called page fault groups in NVIDIA's terminology [66]. The prefetcher uses heuristics to transfer more pages than requested based on the faulting addresses in the group. In Fig. 3(b), we see that disabling the prefetcher in an attempt to reduce the data transfer volume hurts performance severely. Disabling the prefetcher reduces the transfer volume by 35%, but it increases the total runtime by 4.15x.

6.3 Evictions

The poor locality of accesses in graph applications also makes page replacement decisions challenging. In NVIDIA's implementation, evictions happen at a 2 MB granularity. When a new page needs to be mapped in, a 2 MB victim region is selected for eviction, and any 4 KB pages mapped in the region are evicted. The eviction follows an age based LRU policy. It is different from traditional LRU in that it tracks the timing of page allocations rather than page accesses by the GPU. We confirmed this from the UVM driver's source code. In Fig. 3(c), we plot the distribution of 4 KB pages that are fetched from the host to device. We see that it is extremely common for a large fraction of pages to be fetched multiple times. These pages are originally fetched as either demand loads or prefetches, but are evicted due to capacity constraints, only to be fetched again in the future.

7. HARMONIC LOCALITY ORDERING

There are two primary challenges with reordering a graph for efficient accesses in an arbitrary BFS traversal: i) The source node for the BFS is not known beforehand, and ii) directed graphs make reordering challenging. If the source vertex for the BFS traversal were known beforehand, one could label the vertices in the graph in the order in which they are actually visited during the traversal starting with the source as vertex 0. The sample graph in Fig. 2 is, in fact, labelled in BFS order with node 0 as the source. Hence, it is not surprising that it has good locality of accesses in `elist` for that particular traversal. As we discussed in Sec. 4, this does not extend to other traversals. Our main contribution is that we devise effective and efficient heuristics based on centrality scores to create an ordering. We describe our solution in stages as we refine it towards the final ordering algorithm.

7.1 Step 1: Connected Undirected Graphs

As a first step, let us consider undirected graphs with a single connected component. We would like to place the nodes in BFS frontiers close to each other in their ordering, but we do not know the source beforehand. Instead of a particular source, let us consider all possible sources as the MinIntraBFS cost suggests. If we do a BFS from every source, we would know when a node is visited in each of these traversals. Some nodes are likely to be discovered sooner than the others on average. If we were to create this distribution, we could use this heuristic to order the graph starting with the node that is most likely to be discovered first. Formally, for each node, we can compute

$$c(x) = \frac{1}{\sum_{y \neq x} d(y,x)} \tag{3}$$

where $d(y,x)$ is the shortest distance from a different node y to x. This is, in fact, a centrality metric known as closeness centrality [10, 64] that is used in network and social analysis. A node with high closeness centrality is close to all other nodes in the graph since it has a lower average distance from other nodes. In other words, it is likely to be discovered soon in an arbitrary BFS. The nodes of the graph can be labelled in decreasing order of their $c(x)$ values. The interesting relation is that we can go from a spatial metric in the graph (i.e, distance of a vertex from others) to temporal aspects of the traversal algorithm (i.e., when a vertex is likely to be in a frontier) back to spatial aspects of the graph's storage (i.e., how to lay out the node labels).

7.2 Step 2: Efficiency Considerations

Computing the closeness centrality in unweighted graphs reduces to performing a BFS from each node, which costs $\mathcal{O}(n(n+m))$ and is impractical. Instead, we can sample the starting nodes. If we choose k nodes uniformly at random as starting nodes, Eppstein et al. [30] show that for a graph with diameter d and an error term ϵ, the inverse centrality can be approximated to within an additive error of ϵd if k is $\Theta(\log n/\epsilon^2)$. Since real world graphs such as social networks have a small diameter [58], the approximation works well for

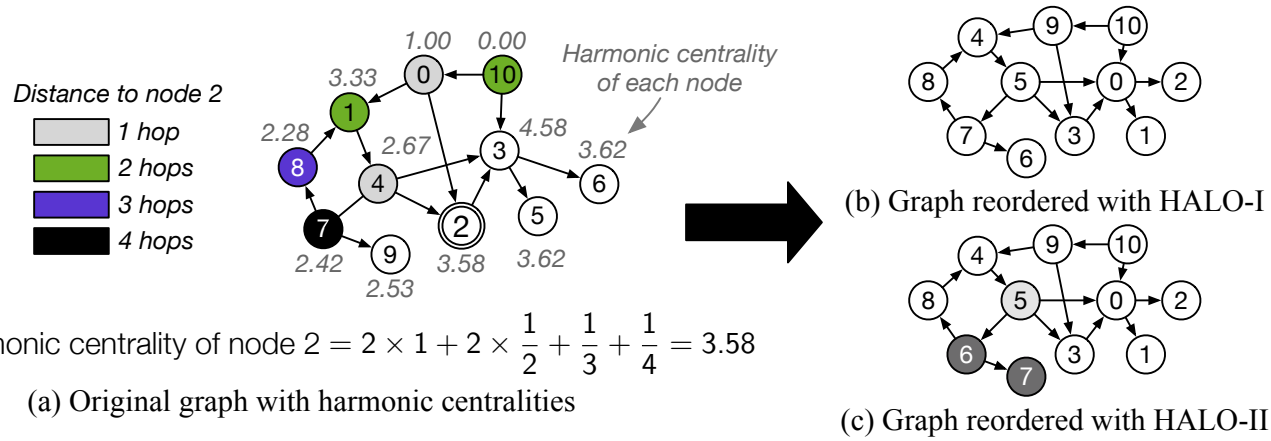


Figure 4: Graph reordering with HALO. HALO-I labels the nodes in decreasing order of centralities. HALO-II additionally labels the neighbours of nodes before moving to the next centrality score.

such cases. Hence, the total complexity with this approximation is $\mathcal{O}((n+m)^{\log n/\epsilon^2})$. Further, each of these sampling BFSes can be done in parallel since they are independent, followed by a parallel reduction.

Algorithm 2 HALO-I (Approximate Harmonic Centrality)

Input: $G = (V, E)$, a sample parameter $\mathbf{k} > 1$
Output: harm[ ] array of size $|V|$ with centrality scores

```
1: for i = 0 to k do
2:    source[i] = get_random_vertex()
3:    levels[i,:] = BFS(G, source[i])
4: for j = 0 to |V| do
5:    harm[j] = 0
6:    for i = 0 to k  do
7:       // Skip zero level values (source nodes)
8:       if (levels[i][j] != 0) then
9:          harm[j] = harm[j] + 1/(levels[i][j])
10: for i = 0 to k do
11:    // Scale values for source nodes that were skipped
12:    harm[source[i]] = (k * harm[source[i]] ) / (k-1)
```

7.3 Step 3: Directed and Disconnected Graphs

Notice that closeness centrality is only defined for undirected and connected graphs. Since distances are summed in closeness centrality, directionality as well as disconnectedness introduces infinities in the summation. In general, the distance of a node from other nodes may be infinite, and computing closeness from Eq. 3 would collapse the entire term. Instead of summing distances, we sum the reciprocal of distances. Formally, we compute

$$H(x) = \sum_{y \neq x} \frac{1}{d(y,x)} \tag{4}$$

While this looks similar to Eq. 3, taking the reciprocal of distances before summing them makes it a harmonic sum rather than an arithmetic one. This deals with the problem of infinite distances elegantly as it does not collapse the term. This centrality metric has been termed as harmonic centrality [62] and is relatively new. As an aside, Boldi et al. [18] compare several common centrality measures in a principled way and find harmonic centrality to satisfy the essential criteria that they term as "axioms" for centrality. Futher, Eppstein et al.'s bounds for sampling based approximation still apply. The first variant of our ordering scheme, called **HALO-I** hereafter, orders the nodes of a graph in decreasing order of their harmonic centrality scores. The algorithm for calculating approximate harmonic centralities is presented in Alg. 2. In Fig. 4(b), we show how the original sample graph from Fig. 2 is permuted by this algorithm. The centrality scores are denoted outside the nodes with explicit calculation shown for node 2. Node 3 is the one with the highest harmonic centrality, followed by nodes 5 and 6. Hence, they get labelled as nodes 0-2 in the reordered graph. The rationale is that node 3 in the original graph is central for an arbitrary BFS and likely to be discovered soon.

7.4 Step 4: Biasing with Neighbourhood

While HALO-I gives us centrality scores, these scores can be the same or very close for nodes that may be in different parts of the graph. Since our primary problem is to optimise the locality for traversals, we make a modification wherein we bias the labelling to favour (out-)neighbours of nodes that we think are likely to be discovered soon. The modification is that instead of going in strict decreasing order of centralities for labelling, at each step we also try to label the immediate neighbourhood of nodes wherever possible. This variant is described in Alg. 3 and called **HALO-II** hereafter. The ordering becomes a nested loop where the outer one (line 2) goes in centrality order, and the inner one (line 5) labels the unlabelled neighbours of nodes. In this scheme, some such neighbours may already have been labelled earlier, either because they have higher centralities themselves, or because they are neighbours of nodes with higher centralities. Such nodes would be skipped. We also show how this changes the ordering in our running example in Fig. 4(c). When node 4 in the original graph is labelled in the outer loop, HALO-II labels its neighbour (node 7 in the original graph) next. Since this additional pass visits each edge only once, it does not increase the cost asymptotically.

7.5 Degree Based Ordering as a Special Case

We can look at harmonic centrality of a node in Eq. 4 as a sum of terms with a decay function. The higher order terms have their weights discounted by a factor equal to the

Algorithm 3 HALO-II (Biasing with Neighbourhood)

Input: $G = (V, E)$, harm[n] calculated in Alg. 2
Output: A bijection ϕ: $V \rightarrow V$
1: count = 0
2: **for all** $u \in V$ in decreasing order of centralities **do**
3: **if** ϕ[u] not valid **then**
4: ϕ[u] = count++
5: **for all** $v \in V$ such that (u,v) $\in$ E **do**
6: **if** ϕ[v] not valid **then**
7: ϕ[v] = count++

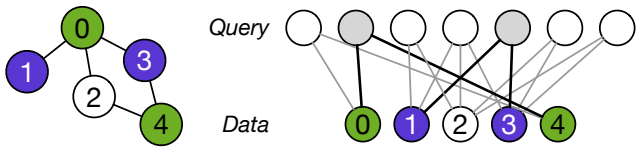
distance they are away from the node [18]. The first term in the harmonic centrality of a node is the number of nodes at distance one, which is its (in-)degree. The second term is the number of nodes at distance two, but this term is scaled by the factor 2, and so on. Hence, we can get a first order approximation of the centrality from its (in-)degree. We evaluate these simplifications for both the variants of HALO in Sec. 10.

8. RECURSIVE BISECTION (BFS-BP)

While HALO uses a geometric measure for reordering the graph, we also create a separate method that optimises the MinIntraBFS cost directly. The idea is to reduce MinIntraBFS to the BiMLogA problem, and leverage the recursive bisection algorithm, BP [29], for BiMLogA. Recall that the BiMLogA formulation seeks to reduce gaps between the data nodes (D) in a bipartite graph consisting of query (Q) and data nodes (D). To use this formulation, we add a query node for *each* level of a BFS from *each* source, and we add edges from the query node to the nodes in the respective BFS level. In Fig. 5, we show a sample graph and the corresponding BiMLogA graph that we need to create in order to optimise the objective in MinIntraBFS. For each possible level from each possible source, we add a query node in Q and add edges to the nodes in that BFS level in D.

We outline the main aspects of the algorithm here, and refer the reader to BP [29, 54] for more details. The algorithm follows the classic Kernighan-Lin [43] heuristic for recursive graph bisection. The set D of data nodes is split into into two sets, $V1$ and $V2$, and a computational cost of the partition is defined. Next, the algorithm exchanges pairs of vertices in $V1$ and $V2$ in order to improve the cost. For every vertex $v \in D$, a move gain, which is the difference of the cost after moving v from its current set to the other set, is computed. The vertices of $V1$ and $V2$ are sorted in decreasing order of gains, and pairs of vertices are exchanged if the sum of their move gains is positive. This describes a single iteration, and the same step continues for a fixed number of iterations (20 in our case) or until convergence. The algorithm continues recursively on the partitions. The cost of the partition, which guides the move gains in BP, is defined as follows: For every vertex $q \in Q$, let $deg_1(q) = |(q, v) : v \in V1|$, that is, the number of adjacent vertices in set $V1$; define $deg_2(q)$ similarly. Then the cost of the partition is:

$$\sum_{q \in Q} \left(deg_1(q) \log(\frac{n_1}{deg_1(q)+1}) + deg_2(q) \log(\frac{n_2}{deg_2(q)+1}) \right) \quad (5)$$



(a) Sample graph (b) Reduction to BIMLogA graph

Figure 5: Reduction from MinIntraBFS to BiMLogA. When node 2 is the source, {0,4} and {1,3} are the level sets. We add one query node for each set along with the corresponding edges to the data nodes. The same process is followed for a traversal from each source.

Table 3: Related Work

Reordering Method	Use Case
RCM [26, 34]	Bandwidth Minimisation
Shingle [25], Slashburn [50], BP [29], LLP [16], BFS [5]	Graph Compression
Gorder [71], Norder [47], RabbitOrder [6], GRO [37], Degree Based [72, 9, 31], EmptyHeaded [3]	Locality Optimisation
METIS [41]	Graph Partitioning

where n_1 is $|V_1|$ and n_2 is $|V_2|$. The cost estimates the number of bits needed for coding the gaps between vertex labels in binary. This follows from the fact that if the neighbours of $q \in Q$ are uniformly distributed in the final arrangement of $V1$ and $V2$, then the average gap between consecutive numbers in q's adjacency list is $gap_1 := n_1/(deg_1(q) + 1)$ and $gap_2 := n_2/(deg_2(q) + 1)$ for $V1$ and $V2$, respectively.

The cost of running BP is $\mathcal{O}(m \log n + n \log^2 n)$. However, in order to create the BiMLogA graph, we need to do a BFS from every source, which is $\mathcal{O}(n(n + m))$, and the resultant BiMLogA graph would have $\mathcal{O}(n^2)$ edges since we add $\mathcal{O}(n)$ edges for each BFS. This is again not practical as a preprocessing step. Similar to the sampling strategy in HALO, we only do BFSes from the sampled nodes instead of all nodes. This reordering method is called **BFS-BP** hereafter.

9. RELATED WORK

There is a large body of work on graph reordering that spans sparse matrix optimisations, locality optimisations, graph partitioning, and graph compression. We list some of these in Table 3. Many sparse matrix computations benefit from reordering. The goal of reordering in these cases is to reduce fill-in in direct solvers or to improve the locality in iterative solvers. Since BFS can be expressed as SpMV, it seems natural to apply similar techniques. The Cuthill-McKee (CM) algorithm [26] and the related reverse Cuthill-McKee (RCM) algorithm [34] are commonly used for reducing the bandwidth of symmetric sparse matrices. RCM is also interesting because the ordering method itself does a variation of BFS traversal. Wei et al. [71] make the observation that sibling relationships between vertices are crucial to locality, and they propose Gorder to reduce CPU cache misses. Gorder's performance for BFS was comparable to RCM in their evaluations. We found that Gorder has a high

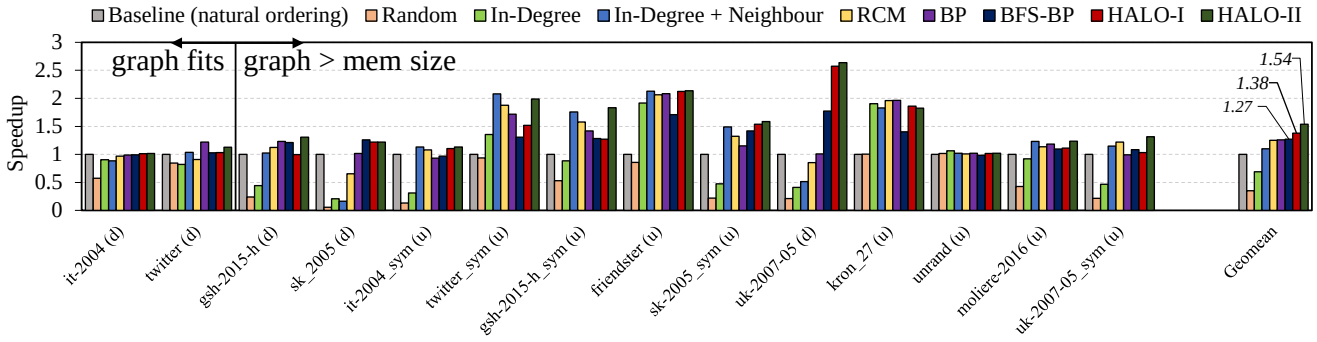


Figure 6: Single-source BFS performance for different graph orderings relative to natural ordering on a Titan Xp GPU. Results are accumulated over 50 traversals from random sources. Geomean reported for graphs larger than memory capacity.

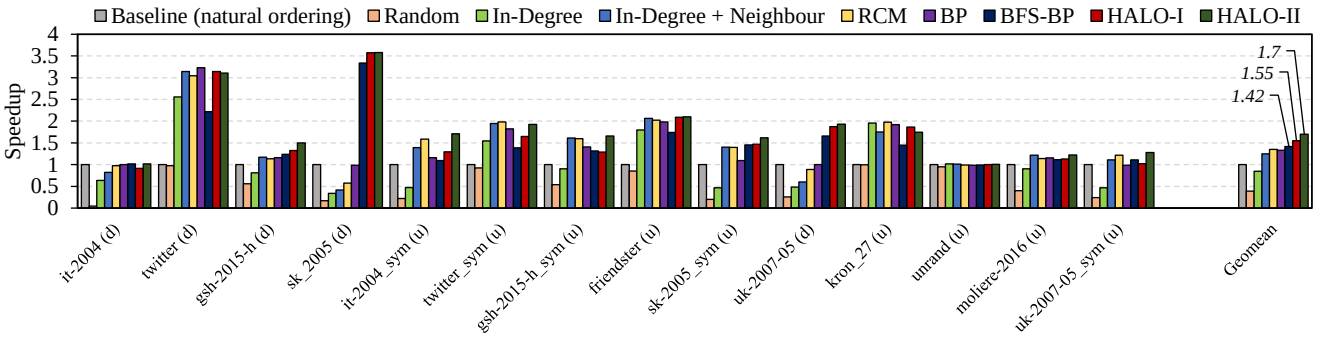


Figure 7: Multi-source BFS performance for different graph orderings relative to natural ordering on a Titan Xp GPU. Each kernel does a BFS from 10 random sources at the same time.

runtime cost, and we were not able to use it for graphs at our scale. Norder [47] uses (in-)degree + neighbourhood for ordering. As we showed in Sec. 7.5, degree can be treated as a first order approximation of harmonic centrality. Norder also looks at neighbours two hops away rather than just immediate neighbours. However, any depth greater than one increases the reordering time asymptotically. A few other schemes also use variations of degree based sorting and clustering [72, 9, 37, 31]. Empty-Headed [3] uses a hybrid BFS and degree based sorting that is similar to RCM. Rabbit Order [6] is a reordering scheme based on community detection for page-rank like applications. It works on undirected graphs, and the performance for BFS is comparable to RCM in the authors' results. We found that a lot of reordering methods target the locality of vertex property structures (i.e., the vector in SpMV like formulation). These methods help in improving the temporal locality of properties such as page-rank or distance values of nodes. They do not benefit accesses in `elist` in our semi-external memory model. There is no temporal reuse of edges in a BFS. Various graph compression methods [25, 50, 29, 16, 5] help with locality in general, although the goals are somewhat different. Graph partitioning methods such as the ones used by METIS [41] are also employed in distributed computing scenarios. The goal in partitioning is to reduce the communication costs and to improve load imbalance. This does not map to our model directly as there is only a single compute node that cannot keep a partition in memory across iterations. There are also disk based methods like GraphChi [46], which uses a sharding technique for improving locality. These methods also work better for iterative convergent algorithms like page-rank that need to access the entire edge list in each iteration.

From the ordering methods in prior works, we use RCM and BP in our evaluations. RCM is close to the best performing ordering for BFS in prior works [71, 6], and BP is highly competitive for graph compression metrics. RCM is used for reducing the bandwidth of sparse symmetric matrices (i.e., undirected graphs). For directed graphs, we use RCM on the symmetrised version (i.e., $A+A^T$) of the graph.

10. RESULTS

We compare the performance impact of different ordering schemes in two different experiments. In the first experiment, as shown in Fig. 6, we look at the performance of 50 different traversals performed one at a time from random sources. This is the same setup that we have used throughout the paper. In Fig. 7, we also look at the performance of multi-source BFS traversals where each kernel does 10 different traversals from random sources at the same time. This is meant to simulate patterns seen in applications like betweenness centrality [19, 8], all pairs shortest paths, or BFS in large diameter graphs [69], where instead of a single frontier, multiple frontiers progress at the same time. The reordering methods compared are natural ordering (baseline), random ordering, the two HALO variants and their first order approximations to (in-)degree, BP [29], RCM [26], and BFS-BP. For the sampling based approximation in HALO

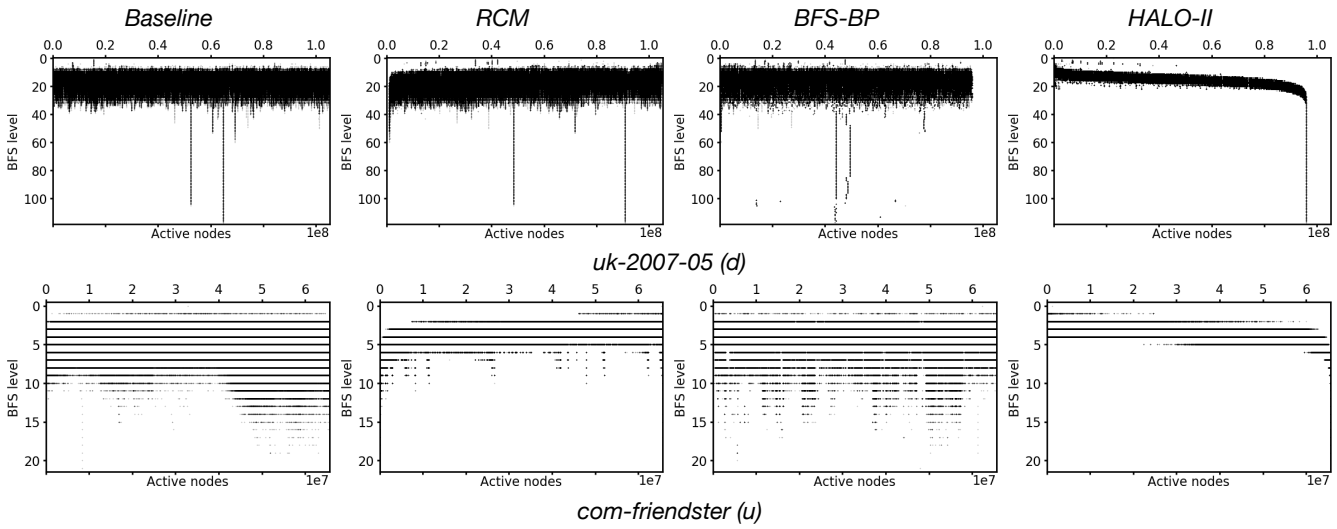


Figure 8: Distribution of active nodes across different frontiers of the traversal

and BFS-BP, we used 20 BFS traversals from different and unrelated random sources.

We first discuss the results in Fig. 6. The ordering methods are arranged in increasing order of average performance. The baseline's ordering is natural ordering. There is often some locality in natural ordering, which depends on the process used for generating the graph. For example, the web graphs in this collection are ordered based on the lexicographical ordering of URLs. This is important because the potential for improving locality is highly dependent on the original ordering. The two synthetic graphs, `kron_27` and `unrand`, don't have a natural ordering, and their baseline is the same as random ordering. Since `unrand` is a uniformly random graph, it does not benefit from any ordering scheme. The random ordering destroys any existing locality in other graphs and leads to an average slowdown of 65%. The simple in-degree based ordering scheme also leads to an average 31% slowdown. However, with the addition of immediate neighbour heuristic to the in-degree scheme, the performance improves, but only for undirected graphs. It performs poorly for directed graphs, and the overall average speedup is 1.1x. Next, RCM and BP perform better than in-degree + neighbour, but the performance for directed graphs is again variable with better results for undirected graphs. Overall, RCM and BP show speedups of 1.25x and 1.26x, respectively. BFS-BP performs more consistently across both directed and undirected graphs, but does not outperform RCM for undirected graphs. This is consistent with prior works which find RCM to perform well for undirected graphs. BFS-BP's overall speedup is 1.27x. Finally, HALO-I and HALO-II show an overall speedup of 1.38x and 1.54x, respectively. BP, BFS-BP, and HALO are consistent across the datasets in that they perform at least as well as the baseline for every case. Since the original graph may already have good locality, the reordering method should not destroy it.

The performance trends for the multi-source BFS experiment in Fig. 7 are similar with BFS-BP, HALO-I, and HALO-II achieving speedups of 1.42x, 1.55x, and 1.7x, respectively. There are two graphs at the left end that exhibit larger swings in performance. The `twitter_d` graph fits in memory on its own, but with the additional metadata needed for storing BFS levels for multiple sources, it starts to exceed the GPU's memory. Reordering the graph manages to keep the accesses largely in-memory. This leads to a more pronounced speedup. This effect diminishes once the graphs become larger, and the overall trend becomes similar to Fig. 6.

To demonstrate the results visually, we show the distribution of active nodes with the passage of time in Fig. 8. The vertical axis plots increasing level numbers from top to bottom, and the horizontal axis plots the vertex ids active at each level. For the directed uk web graph, we can clearly see that the active nodes *within* the levels as well as *across* the levels follow the desirable "staircase" pattern when the graph is ordered with HALO-II. BFS-BP on the other hand manages to reduce the overall spread of active vertices within levels, as evidenced by the gap at the right end, but does not outperform HALO-II. RCM fails to improve over baseline for the directed graph. For the undirected friendster graph, both RCM and HALO-II achieve good results whereas BFS-BP falls behind the others.

10.1 Data Transfer Volume and Log Gap Cost

The results show that the performance improvements due to reordering are strongly correlated with the reduction in data transfer volume. Earlier in Sec. 4, we observed that the baseline ordering causes read amplification of 2.31x over the expected ideal transfer volume. In Fig. 9, we see that HALO-II reduces this amplification from 2.31x to 1.42x. BFS-BP and RCM manage to reduce it to 1.72x and 1.8x, respectively. We also compare the average log gap between nodes within a BFS level in Fig. 10. This is the MinItraBFS cost from Eq. 1 for the evaluated traversals. Once again, we see that HALO-II achieves the lowest average log gap cost, which is consistent with the volume of data transfer in Fig. 9 as well as performance trends in Fig. 6. The degree of improvement in the log gap cost is also a good indicator of improvement in performance. We see that graphs like the `uk-2007-05(d)` show a significant reduction in the log gap

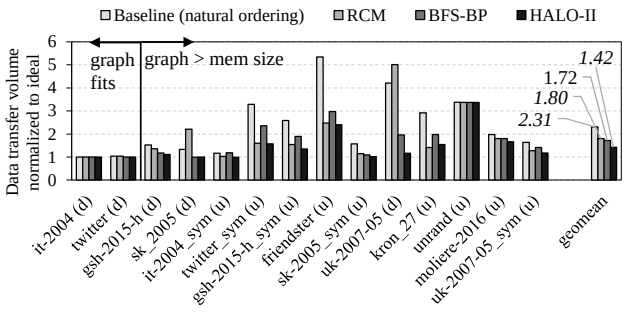


Figure 9: Host to device data transfer volume (lower is better) relative to ideal transfer volume

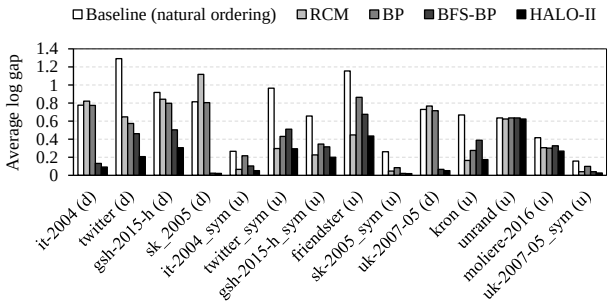


Figure 10: Average log gap (i.e., MinIntraBFS cost; lower is better) between nodes within a BFS level for 50 traversals

cost, but the cost is relatively unchanged in `unrand`. Although BFS-BP is designed to minimise the log gap cost explicitly, it is still an approximate solution with simplifying assumptions. For example, the cost of the partition in Eq. 5 assumes that neighbours of $q \in Q$ are uniformly distributed in the final arrangement. We also using sampling as an additional simplification to limit the runtime.

10.2 Sample Size Sensitivity and Reordering Overhead

The practicality of using HALO hinges on the number of sample BFS traversals needed for approximating harmonic centrality. We used 20 random samples for the experiments so far. We now vary the sample parameter systematically from 2 to 256 and measure the performance for single-source BFS. The results are shown in Fig. 11. We see that the performance saturates around 32 samples with negligible improvements beyond that. These graphs have roughly 100M nodes (Sec. 3.4), and the results indicate that the performance saturates sooner than Eppstein et al.'s bounds [30] (i.e., more like $\log n$ rather than $\Theta(\log n/\epsilon^2)$). There are a few likely reasons for this. One is that we are using centralities indirectly for ordering. Specifically, we are using a relative ordering of centralities; the absolute error in centrality values is less relevant for our case. Second, since UVM is a page fault handling mechanism, we benefit from locality improvements at a coarse granularity and are more tolerant to errors at a finer level.

Sequential Costs: HALO's asymptotic cost for both the variants is $\mathcal{O}(\log n/\epsilon^2(n+m))$. RCM's runtime cost is $\mathcal{O}(m)$ [23]. However, this does not include the cost of finding pseudo-peripheral nodes, which is an essential additional step in RCM. We are not aware of bounds on finding pseudo-peripheral nodes, but various methods [45] have been proposed for pruning the number of additional BFSes needed. BFS-BP costs $\mathcal{O}(kn \log n + n \log^2 n)$ for k sample BFSes since each BFS adds $\mathcal{O}(n)$ edges.

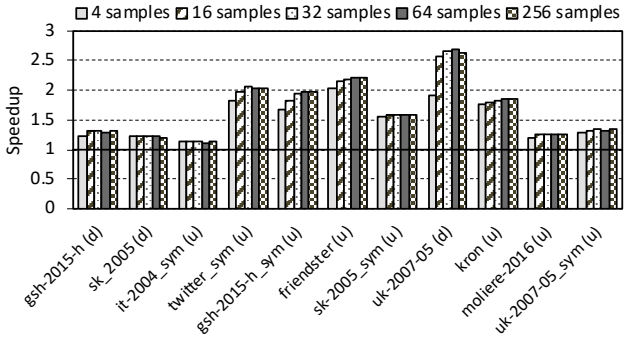


Figure 11: Sensitivity to the number of samples used in HALO-II. Performance saturates around 32 samples.

Parallelisation: The computation of approximate harmonic centralities is trivially parallel since the sample BFSes are completely independent and each BFS can also be parallelised. In contrast, the additional BFSes needed for finding pseudo-peripheral nodes in RCM are dependent in nature. Parallelising RCM itself is also challenging [40]. We note that the additional pass over the graph in HALO-II (Alg. 3) is sequential in our current implementation. BFS-BP follows recursive parallelism where each partition becomes an independent subtask, but processing within a partition is still serial.

Runtime: Since we did not have uniformly optimised and parallel versions of the different ordering methods, we do not compare the raw runtimes. We used the boost library's implementation for RCM, which was slow and took several hours. We used the PISA framework [54] for implementing BFS-BP. It uses TBB for parallelisation, and running BFS-BP with 20 BFS samples took roughly one hour for the graphs in this work on a 44 core Broadwell server. Reordering with HALO-I and HALO-II generally took minutes to tens of minutes.

10.3 Scaling with Graph500 Graphs

We perform scaling experiments with Kronecker graphs that are generated per Graph500 specifications using the gapbs framework [12]. These graphs have 2^{scale} nodes and $m = 16 \times n$. Single-source BFS performance for 50 random traversals up to scale 30 for directed graphs and scale 29 for undirected ones is shown in Fig. 12. We see a consistent speed-up of around 1.8-1.9x for all the cases.

10.4 Additional Optimisations

In NVIDIA's current implementation, the pages allocated with UVM only exist in a single device's page table at a time. All the memory is assumed to be read-write (RW). The runtime does not support transparent and fine-grained copy-on-write semantics. When RW pages need to be evicted from the GPU, they need to be transferred and mapped in the host's page table. This is a blocking operation that effectively doubles the data transfer volume since everything

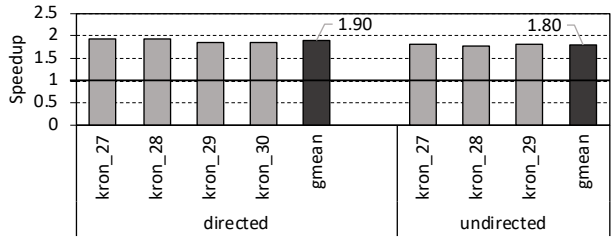


Figure 12: Scaling with Kronecker graphs. Number denotes the scale where $n = 2^{scale}$ and $m = 16 \times n$

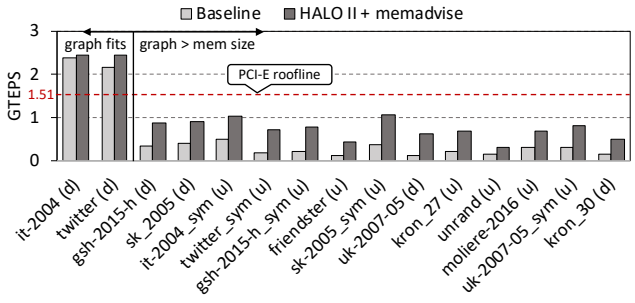


Figure 13: Overall performance after combining HALO-II reordering and the read-only memadvise hint

that is transferred from the host to device is also transferred back for a large enough data set due to evictions. Our application is clearly segregated between read only (RO) structures, `elist` and `vlist`, and RW structures, `level` and `pred`. This allows up to pass a read-only memadvise hint for the RO structures. The GPU creates copies of such pages and drops them freely on eviction instead of waiting for them to be transferred back to the host. This leads to an overall speedup of 1.85x due to the approximate halving of data moved over PCI-e. While this is a simple optimisation, it may not be immediately obvious or possible in all applications. Combined with the average speedup of 1.54x from HALO-II's reordering (Fig. 6), we get an overall average speedup of 2.84x, which we show in Fig. 13.

11. LIMITATIONS AND DISCUSSION

Although we did not encounter pathological cases in our datasets, it is possible to create graphs where harmonic centrality will not create a good ordering for BFS. For example, consider multiple identical copies of a graph as different (weakly) connected components. Since the components are identical in a geometric sense, the centrality values will be identical for every copy of a node in each of the components. This will create an interleaved ordering of nodes if we sort by centrality scores. In this particular case, we can separate the components first before ordering, but in general, one can create similar scenarios even within a single component.

The semi-external memory model, where we assume that only $\mathcal{O}(n)$ structures fit in memory, also places some restrictions on algorithms. For example, some BFS implementations allow duplicates in the frontier and remove duplicates in a subsequent step. This strategy does not work in the UVM model as the frontier would grow to $\mathcal{O}(m)$ with duplicates and cause a severe performance penalty due to random read-write traffic over the slow interconnect. This does not affect our level array implementation of frontiers that fits in memory (Sec. 3), but is a fairly common pattern in graph analytics.

The benefit of graph reordering is also application dependent. In this work, we looked at improving the locality for edge accesses that are hard to predict. In some applications like page-rank, we need to touch all the edges in each iteration. This sort of reordering may not be beneficial, but prior solutions such as GraphChi [46] may apply. In some applications such as single source shortest paths (SSSP), work-optimality considerations are more important that ordering. Common parallel implementations such as Bellman-Ford's algorithm do asymptotically more work than the sequential counterpart, which leads to a severe performance penalty due to remote accesses. Delta-stepping [57, 27] implementations for SSSP have been proposed, but are quite challenging to implement on GPUs. Finally, the UVM architecture is still relatively new and would benefit from a combination of hardware and software solutions. Recent hardware proposals [49, 44] that optimise the GPU's page fault handling mechanism can be used as a complimentary approach.

12. CONCLUSION

In this work, we looked at the problem of improving the locality of data accesses for breadth first search (BFS) in a system architecture where the GPU can overprovision memory by accessing larger, albeit slower, host memory transparently. This poses several challenges as BFS has low computational complexity and an irregular data access pattern. We proposed a new graph reordering algorithm, HALO, that is both lightweight and effective in improving the performance of subsequent traversals on large real world graphs. It is possible to improve the performance of BFS by 1.5x-1.9x in the unified memory setting by reordering the graph. We found that HALO captures the structure of real world directed graphs from the perspective of an arbitrary BFS traversal well whereas popular prior methods such as RCM only do so for undirected graphs. Additionally, the problem ties into other graph ordering problems, and we showed that we can leverage techniques from graph compression such as recursive bisection, which resulted in an additional ordering method (BFS-BP). In general, this opens up the problem space for creating orderings that are both locality and compression friendly. To our knowledge, neither the problem of doing graph traversals in unified memory, nor the use of harmonic centrality for graph reordering have been explored before. The reordering solution is general enough that we expect it would extend to other similar semi-external memory hierarchies with large unit transfer sizes.

Acknowledgements: We thank Srinivas Eswar, Oded Green, and Prof. Richard Peng for helpful discussions. We are grateful to the reviewers for their insightful feedback. This work was supported in part by Defense Advanced Research Projects Agency (DARPA) under contract FA8750-17-C-0086. The content of the information in this document does not necessarily reflect the position or the policy of the U.S. Government, and no official endorsement should be inferred. The U.S. Government is authorised to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation here on.

13. REFERENCES

[1] Heterogeneous memory management (hmm) the linux kernel documentation. https://www.kernel.org/doc/html/latest/vm/hmm.html. (Accessed on 02/24/2020).

[2] Shared virtual memory. https://www.khronos.org/registry/OpenCL/sdk/2.1/docs/man/xhtml/sharedVirtualMemory.html. (Accessed on 02/24/2020).

[3] C. R. Aberger, A. Lamb, S. Tu, A. Nötzli, K. Olukotun, and C. Ré. Emptyheaded: A relational engine for graph processing. *ACM Transactions on Database Systems (TODS)*, 42(4):20, 2017.

[4] D. Ajwani, R. Dementiev, and U. Meyer. A computational study of external-memory bfs algorithms. In *Proceedings of the seventeenth annual ACM-SIAM symposium on Discrete algorithm*, pages 601–610. Society for Industrial and Applied Mathematics, 2006.

[5] A. Apostolico and G. Drovandi. Graph compression by bfs. *Algorithms*, 2(3):1031–1044, 2009.

[6] J. Arai, H. Shiokawa, T. Yamamuro, M. Onizuka, and S. Iwamura. Rabbit order: Just-in-time parallel reordering for fast graph analysis. In *Parallel and Distributed Processing Symposium, 2016 IEEE International*, pages 22–31. IEEE, 2016.

[7] R. Ausavarungnirun, J. Landgraf, V. Miller, S. Ghose, J. Gandhi, C. J. Rossbach, and O. Mutlu. Mosaic: A GPU Memory Manager with Application-transparent Support for Multiple Page Sizes. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2017.

[8] D. A. Bader, S. Kintali, K. Madduri, and M. Mihail. Approximating betweenness centrality. In *International Workshop on Algorithms and Models for the Web-Graph*, pages 124–137. Springer, 2007.

[9] V. Balaji and B. Lucia. When is graph reordering an optimization? studying the effect of lightweight graph reordering across applications and input graphs. In *2018 IEEE International Symposium on Workload Characterization (IISWC)*, pages 203–214, Los Alamitos, CA, USA, oct 2018. IEEE Computer Society.

[10] A. Bavelas. Communication patterns in task-oriented groups. *The Journal of the Acoustical Society of America*, 22(6):725–730, 1950.

[11] S. Beamer, K. Asanović, and D. Patterson. Direction-optimizing breadth-first search. *Scientific Programming*, 21(3-4):137–148, 2013.

[12] S. Beamer, K. Asanovic, and D. Patterson. The GAP Benchmark Suite, 2015.

[13] M. Besta and T. Hoefler. Survey and taxonomy of lossless graph compression and space-efficient graph representations. *arXiv preprint arXiv:1806.01799*, 2018.

[14] P. Boldi, B. Codenotti, M. Santini, and S. Vigna. UbiCrawler: A Scalable Fully Distributed Web Crawler. *Software: Practice & Experience*, 34(8):711–726, 2004.

[15] P. Boldi, A. Marino, M. Santini, and S. Vigna. BUbiNG: Massive Crawling for the Masses. In *Proceedings of the Companion Publication of the 23rd International Conference on World Wide Web*, 2014.

[16] P. Boldi, M. Rosa, M. Santini, and S. Vigna. Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks. In *Proceedings of the 20th international conference on World wide web*, pages 587–596. ACM, 2011.

[17] P. Boldi and S. Vigna. The WebGraph Framework I: Compression Techniques. In *Proc. of the Thirteenth International World Wide Web Conference (WWW 2004)*, 2004.

[18] P. Boldi and S. Vigna. Axioms for centrality. *Internet Mathematics*, 10(3-4):222–262, 2014.

[19] U. Brandes. A faster algorithm for betweenness centrality. *Journal of mathematical sociology*, 25(2):163–177, 2001.

[20] A. Buluç, J. T. Fineman, M. Frigo, J. R. Gilbert, and C. E. Leiserson. Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks. In *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*, pages 233–244. ACM, 2009.

[21] A. Buluç and K. Madduri. Parallel breadth-first search on distributed memory systems. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, page 65. ACM, 2011.

[22] F. Busato, O. Green, N. Bombieri, and D. A. Bader. Hornet: An efficient data structure for dynamic sparse graphs and matrices on gpus. In *2018 IEEE High Performance extreme Computing Conference (HPEC)*, pages 1–7. IEEE, 2018.

[23] W.-M. Chan and A. George. A linear time implementation of the reverse cuthill-mckee algorithm. *BIT Numerical Mathematics*, 1980.

[24] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, pages 44–54. Ieee, 2009.

[25] F. Chierichetti, R. Kumar, S. Lattanzi, M. Mitzenmacher, A. Panconesi, and P. Raghavan. On compressing social networks. In *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 219–228. ACM, 2009.

[26] E. Cuthill and J. McKee. Reducing the bandwidth of sparse symmetric matrices. In *Proceedings of the 1969 24th national conference*, pages 157–172. ACM, 1969.

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

[28] L. Dhulipala, G. E. Blelloch, and J. Shun. Theoretically efficient parallel graph algorithms can be fast and scalable. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures*, pages 393–404, 2018.

[29] L. Dhulipala, I. Kabiljo, B. Karrer, G. Ottaviano, S. Pupyrev, and A. Shalita. Compressing graphs and indexes with recursive graph bisection. In *Proceedings*

of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pages 1535–1544, 2016.

[30] D. Eppstein and J. Wang. Fast approximation of centrality. In *Proceedings of the twelfth annual ACM-SIAM symposium on Discrete algorithms*, pages 228–229. Society for Industrial and Applied Mathematics, 2001.

[31] P. Faldu, J. Diamond, and B. Grot. A closer look at lightweight graph reordering. *2019 IEEE International Symposium on Workload Characterization (IISWC)*, 2019.

[32] L. K. Fleischer, B. Hendrickson, and A. Pınar. On identifying strongly connected components in parallel. In *International Parallel and Distributed Processing Symposium*, pages 505–511. Springer, 2000.

[33] M. R. Garey and D. S. Johnson. *Computers and intractability*, volume 29.

[34] J. A. George. Computer implementation of the finite element method. Technical report, STANFORD UNIV CA DEPT OF COMPUTER SCIENCE, 1971.

[35] A. Gharaibeh, T. Reza, E. Santos-Neto, L. B. Costa, S. Sallinen, and M. Ripeanu. Efficient large-scale graph processing on hybrid cpu and gpu systems. *arXiv preprint arXiv:1312.3018*, 2013.

[36] O. Green and D. A. Bader. custinger: Supporting dynamic graph algorithms for gpus. In *High Performance Extreme Computing Conference (HPEC), 2016 IEEE*, pages 1–6. IEEE, 2016.

[37] S. Han, L. Zou, and J. X. Yu. Speeding up set intersections in graph algorithms using simd instructions. In *Proceedings of the 2018 International Conference on Management of Data*, pages 1587–1602. ACM, 2018.

[38] P. Harish and P. Narayanan. Accelerating large graph algorithms on the gpu using cuda. In *International conference on high-performance computing*, pages 197–208. Springer, 2007.

[39] M. Hussein, A. Varshney, and L. Davis. On implementing graph cuts on cuda.

[40] K. I. Karantasis, A. Lenharth, D. Nguyen, M. J. Garzarán, and K. Pingali. Parallelization of reordering algorithms for bandwidth and wavefront reduction. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 921–932. IEEE Press, 2014.

[41] G. Karypis and V. Kumar. A software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices. 1998.

[42] J. Kepner and J. Gilbert. *Graph algorithms in the language of linear algebra*. SIAM, 2011.

[43] B. W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *Bell system technical journal*, 49(2):291–307, 1970.

[44] H. Kim, J. Sim, P. Gera, R. Hadidi, and H. Kim. Batch-aware unified memory management in gpusfor irregular workloads. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2020.

[45] G. K. Kumfert. Object-oriented algorithmic laboratory for ordering sparse matrices. Technical report, Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), 2000.

[46] A. Kyrola, G. E. Blelloch, and C. Guestrin. Graphchi: Large-scale graph computation on just a pc. USENIX, 2012.

[47] E. Lee, J. Kim, K. Lim, S. H. Noh, and J. Seo. Pre-select static caching and neighborhood ordering for bfs-like algorithms on disk-based graph engines. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 459–474, Renton, WA, July 2019. USENIX Association.

[48] J. Leskovec and A. Krevl. SNAP Datasets: Stanford Large Network Dataset Collection. `http://snap.stanford.edu/data`, June 2014.

[49] C. Li, R. Ausavarungnirun, C. J. Rossbach, Y. Zhang, O. Mutlu, Y. Guo, and J. 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.

[50] Y. Lim, U. Kang, and C. Faloutsos. Slashburn: Graph compression and mining beyond caveman communities. *IEEE Transactions on Knowledge and Data Engineering*, 26(12):3077–3089, 2014.

[51] H. Liu and H. H. Huang. Enterprise: Breadth-first graph traversal on gpus. In *High Performance Computing, Networking, Storage and Analysis, 2015 SC-International Conference for*, pages 1–12. IEEE, 2015.

[52] L. Luo, M. Wong, and W.-m. Hwu. An effective gpu implementation of breadth-first search. In *Proceedings of the 47th design automation conference*, pages 52–55. ACM, 2010.

[53] S. Maass, C. Min, S. Kashyap, W. Kang, M. Kumar, and T. Kim. Mosaic: Processing a trillion-edge graph on a single machine. In *Proceedings of the Twelfth European Conference on Computer Systems*, pages 527–543. ACM, 2017.

[54] J. Mackenzie, A. Mallia, M. Petri, J. S. Culpepper, and T. Suel. Compressing inverted indexes with recursive graph bisection: A reproducibility study. In *Proc. ECIR*, pages 339–352, 2019.

[55] E. Mastrostefano and M. Bernaschi. Efficient breadth first search on multi-gpu systems. *Journal of Parallel and Distributed Computing*, 73(9):1292–1305, 2013.

[56] D. Merrill, M. Garland, and A. Grimshaw. High-performance and scalable gpu graph traversal. *ACM Transactions on Parallel Computing*, 1(2):14, 2015.

[57] U. Meyer and P. Sanders. δ-stepping: a parallelizable shortest path algorithm. *Journal of Algorithms*, 49(1):114–152, 2003.

[58] S. Milgram. The small world problem. *Psychology today*, 2(1):60–67, 1967.

[59] R. C. Murphy, K. B. Wheeler, B. W. Barrett, and J. A. Ang. Introducing the graph 500. *Cray Users Group (CUG)*, 19:45–74, 2010.

[60] L. Nai, Y. Xia, I. G. Tanase, H. Kim, and C.-Y. Lin. Graphbig: understanding graph computing in the context of industrial solutions. In *High Performance*

Computing, Networking, Storage and Analysis, 2015 SC-International Conference for, pages 1–12. IEEE, 2015.

[61] Y. Pan, Y. Wang, Y. Wu, C. Yang, and J. D. Owens. Multi-gpu graph analytics. In *Parallel and Distributed Processing Symposium (IPDPS), 2017 IEEE International*, pages 479–490. IEEE, 2017.

[62] Y. Rochat. Closeness centrality extended to unconnected graphs: The harmonic centrality index. Technical report, 2009.

[63] R. A. Rossi and N. K. Ahmed. The Network Data Repository with Interactive Graph Analytics and Visualization. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*, 2015.

[64] G. Sabidussi. The centrality index of a graph. *Psychometrika*, 31(4):581–603, 1966.

[65] N. Sakharnykh. Maximizing unified memory performance in cuda — nvidia developer blog. `https://devblogs.nvidia.com/maximizing-unified-memory-performance-cuda/`. (Accessed on 02/27/2020).

[66] N. Sakharnykh. Unified memory on pascal and volta. `http://on-demand.gputechconf.com/gtc/2017/presentation/s7285-nikolay-sakharnykh-unified-memory-on-pascal-and-volta.pdf`, May 2017.

[67] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. D. Liu, and W.-m. W. Hwu. Parboil: A revised benchmark suite for scientific and commercial throughput computing. *Center for Reliable and High-Performance Computing*, 127, 2012.

[68] J. Sybrandt, M. Shtutman, and I. Safro. MOLIERE: Automatic Biomedical Hypothesis Generation System. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '17, 2017.

[69] J. D. Ullman and M. Yannakakis. High-probability parallel transitive-closure algorithms. *SIAM Journal on Computing*, 20(1):100–125, 1991.

[70] Y. Wang, A. Davidson, Y. Pan, Y. Wu, A. Riffel, and J. D. Owens. Gunrock: A high-performance graph processing library on the gpu. In *ACM SIGPLAN Notices*, volume 51, page 11. ACM, 2016.

[71] H. Wei, J. X. Yu, C. Lu, and X. Lin. Speedup graph processing by graph ordering. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1813–1828. ACM, 2016.

[72] Y. Zhang, V. Kiriansky, C. Mendis, S. Amarasinghe, and M. Zaharia. Making caches work for graph analytics. In *2017 IEEE International Conference on Big Data (Big Data)*, pages 293–302. IEEE, 2017.

[73] T. Zheng, D. Nellans, A. Zulfiqar, M. Stephenson, and S. W. Keckler. Towards High Performance Paged Memory for GPUs. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*, 2016.