Traversing Large Compressed Graphs on GPUs

Prasun Gera*
Cerebras Systems
prasun@cerebras.net

Hyesoon Kim
Georgia Tech
hyesoon@cc.gatech.edu

***Abstract*—GPUs can be used effectively for accelerating graph analytics, provided the datasets fit in GPU memory. This is often not the case for large real-world datasets such as social, web, or biological graphs. We propose a graph compression format for static unweighted graphs based on Elias-Fano encoding that is amenable to run-time decompression on massively parallel architectures such as GPUs. We show that we can compress a variety of large graphs by a factor of 1.55x over the commonly used compressed sparse row (CSR) representation. The scheme is particularly beneficial for cases where conventional CSR based approaches do not work at all due to memory capacity constraints, or incur a significant penalty for out-of-core processing. We implement GPU accelerated breadth first search for this graph representation and show that the runtime performance for in-memory compressed graphs is 3.8x-6.5x better than out-of-core implementations for CSR graphs. Further, our implementation is also 1.45x-2x faster than the current state of the art in GPU based compressed graph traversals while maintaining a competitive compression ratio. We also extend our work to other analytics applications such as single source shortest paths and PageRank. Finally, we explore the interplay between graph reordering, graph compression, and performance.**

***Index Terms*—GPU, Graph Compression, Graph Analytics**

I. Introduction

Graphics Processing Units (GPUs) are a class of processors characterised by massive parallelism and a memory hierarchy that offers high performance but is limited in capacity. General purpose computing on GPUs is a popular form of acceleration for workloads such as machine learning, scientific computing, and data analytics. In this work, we focus on applications that work with large sparse graphs such as those arising from web, social, and biological networks. These graphs can have billions of edges and range up to tens or hundreds of gigabytes in size even in a sparse representation. However, GPU memory capacity is limited to a few gigabytes in the common case. A possible solution for large graphs is to distribute them over multiple GPUs and/or CPUs [1]–[3]. There is also recent work on using out-of-core processing [4]–[6], where the data is streamed from the host over the interconnect. These solutions come with trade-offs, namely higher implementation complexity and hardware costs, and bottlenecks due to the slow interconnect. In this work, we propose a complementary solution, namely graph compression, so that one can accommodate larger graphs in GPU memory. The application works with compressed data and uses constant local memory to decompress only the needed portions at runtime. We implement breadth first search (BFS), single source shortest paths (SSSP), and PageRank for compressed graphs in this work. The graph traversal pattern captures many fundamental challenges that arise in graph analytics on GPUs and has broad utility. Other analytics such as betweenness centrality and connected components can also be implemented using a similar approach.

While graph compression has been studied extensively in general [7], prior work has mostly focused on CPUs. This is mainly because the decompression stage is often sequential, irregular, and branch-intensive. Compression schemes reduce the storage requirements of repeating or predictable patterns, and one needs to follow dependent chains during decompression to recover the values. These characteristics do not map well to the GPU architecture, in which it is desirable that threads do the same *type* and *amount* of work without diverging too much. Skews in the degree distribution make the mapping of work to threads in the single instruction multiple threads (SIMT) model difficult. Compression adds another layer of imbalance as blocks of compressed data of the same size do not represent the same number of edges in the graph. Despite these challenges, graph compression is attractive since a GPU's internal memory bandwidth is an order of magnitude higher than the interconnect's bandwidth. In typical scenarios, graph analytics kernels are memory bound and the compute resources are under-utilised, which presents an opportunity to trade off compute resources for memory capacity. Our goal is to decompress the graphs at runtime as a part of the analytics kernel without severely affecting the performance. To that end, this paper makes the following contributions:

- We present a compressed graph representation, Elias-Fano graph (EFG) format, based on the Elias-Fano encoding scheme. Our contribution here is the GPU implementation[1] for run-time decompression in graph analytics. We implement BFS, SSSP, and PageRank for compressed graphs.
- The key challenge in efficient run-time decompression on GPUs is maintaining a high degree of parallelism and load balance between threads. We achieve this by decomposing the problem into smaller high-performance primitives such as parallel scans and searches.
- The proposal satisfies several desirable criteria such as high decompression throughput, competitive compression ratio, a priori determination of compression ratio, and independence from the graph's ordering.

* Contributed while at Georgia Tech

[1] https://github.com/pgera/efg

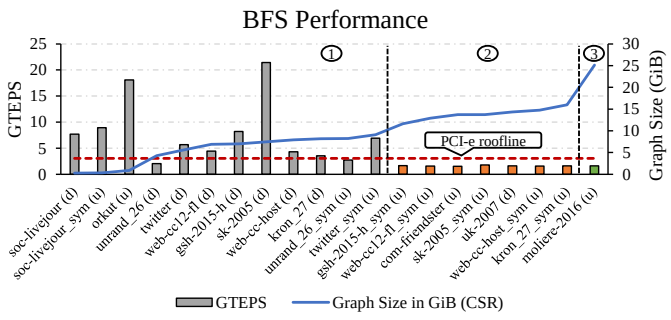


Fig. 1. BFS performance measured in billions of Traversed Edges per Second (GTEPS) on a Titan Xp GPU with 12 GiB memory. Regions: (1) Graphs that fit in memory. (2) Graphs that exceed memory, but would fit after compression. (3) Graphs that will exceed memory even after compression.

TABLE I
GPU BANDWIDTH CHARACTERISTICS

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

- EFG achieves an average compression ratio of 1.55x over the compressed sparse row (CSR) format, and breadth first traversals show a speedup of 3.8x - 6.5x over the equivalent out-of-core CSR implementation.
- EFG achieves a speed-up of 1.45x - 2x over CGR [8], the current state of the art in GPU based compressed graph traversals. Our solution also continues to work in the out of core regime, whereas CGR does not.
- EFG is tolerant to pathological graph orderings. While preprocessing is not needed for EFG to achieve a good compression ratio, graph reordering can improve the decompression performance further.

II. MOTIVATION

GPUs have been used successfully for accelerating dense and sparse matrix/graph applications. Much of the prior work [1], [9] in the area covers various optimisation for improving load-balance, utilisation, and the memory accesses in graph analytics kernels. These solutions work well when the graphs fit in GPU memory. However, real-world graphs can have billions of edges, which exceeds a typical GPU's memory capacity. When the graphs exceed memory capacity, there are different approaches [4]–[6] such as unified virtual memory (UVM) and *zero-copy* memory transfers for out-of-core processing. In the *zero-copy* approach described in EMOGI [4], the graph resides in CPU memory, and data is streamed at cacheline granularity. We adopt this approach in the following experiment.

In Fig. 1, we plot the performance of breath first search (BFS) in billions of traversed edges per second (GTEPS) on a Titan Xp GPU with 12 GiB memory. The graphs are represented in the compressed sparse row (CSR) format and arranged in increasing order of their sizes. We use NVIDIA's high performance graph analytics framework, cugraph [10], for this experiment with light modifications for *zero-copy* memory transfers. We see that the performance drops sharply when the graphs exceed the GPU's memory capacity. This is attributed to the fact that the GPU's internal memory bandwidth is $\sim$ 35x higher than the host-to-device interconnect's bandwidth (Table I). We have marked three regions in Fig. 1:

- In the first region, CSR encoded graphs fit in memory. Here, cugraph's performance is quite good, and we do not need compression unless additional space (e.g., working data or outputs) is needed for the analytics kernel.
- The second region corresponds to graphs that exceed memory capacity. This is the region that stands to benefit the most from graph compression. Graphs in this region would fit in memory after compression and take advantage of the higher internal memory bandwidth.
- The last region is for massive graphs that would not fit in memory even after compression. However, due to the reduction in data transferred over the slow interconnect, compression is still useful in this region.

With 32-bit types, we get a theoretical peak of 3.03 GTEPS for the out-of-memory region as the interconnect's bandwidth is 12.11 GB/s. The primary motivation for graph compression is to overcome this barrier imposed by the interconnect.

III. BACKGROUND

A. GPU Architecture

The NVIDIA GPU architecture consists of a number streaming multiprocessors (SMs). All the SMs have access to a common pool of global memory (e.g., 12 GiB for Titan Xp), and each SM has additional local fast memory known as *shared memory*. Each SM has a number of SIMD lanes, and a collection of threads, known as a *warp*, execute in lock-step fashion. The global *grid* of threads is logically divided into *thread blocks*, where threads within a block have access to shared memory and synchronisation primitives. The GPU connects to the host via an interconnect such as PCI-e.

B. Graph Analytics on a GPU

Algorithm 1 BFS_Level_Advance

```
1: for all u ∈ current_frontier in parallel do
2:     for all (u, v) ∈ E in parallel do
3:         if (visited[v] == false) then
4:             old = atomic_or(&visited[v], true)
5:             if (old == false) then
6:                 add_to_next_frontier(v)
```

A number of graph analytics applications use the idiom of frontier expansion, filtering, and compaction. In Alg. 1, we show the basic structure of expanding a frontier in breadth first search (BFS). Given a frontier of active vertices, the GPU threads explore their neighbours in parallel and add unvisited vertices to the next frontier. The complexity of the GPU implementation is abstracted in the first two lines which are responsible for visiting the edges in parallel. Since

the degree distribution of nodes can be skewed, the primary challenge is mapping this expansion in a load balanced way onto the GPU's threads, which we cover in more detail in Sec. VI. This pattern also extends to applications such as single source shortest path (SSSP), PageRank (PR), betweenness centrality, and others. Traversals on compressed graphs involve decompressing the edges. Once we have the destination of an edge, the rest of the algorithm is similar to the uncompressed case. We use a collection of large web, social, biological, and synthetic graphs [11]–[16]. Directed graphs are denoted with (d) whereas undirected ones are noted with (u). Symmetrised versions of directed graphs are marked with a *_sym* suffix.

C. Scans and Searches

We make extensive use of two operations in this work: i) parallel scans, and ii) binary searches. For a list of n elements [a_0, a_1,..., a_{n-1}], an identity element I, and a binary associative operator $\oplus$, the exclusive scan returns the array [I, $(I \oplus a_0)$, $(I \oplus a_0 \oplus a_1)$,..., $(I \oplus a_0 \oplus a_1 ... \oplus a_{n-1})$]. This can be computed in parallel with $\mathcal{O}(n)$ work and $\mathcal{O}(\log(n))$ depth. A variation of the regular scan is a segmented scan [17] where an additional boolean array is used to mark segments, and the goal is to compute the scans within segments. The binary searches we use in this work are parallel *bounded* searches, where each search finds the index of the largest value less than or equal to the search value. When combined, scans and searches serve an important role in load-balancing.

D. Compressed Sparse Row (CSR) Format

The compressed sparse row (CSR) format is a popular representation for static graphs that stores only the non-zero entries in an adjacency matrix. In the CSR format, a graph $G = (V, E)$ is stored as two arrays: (i) `vlist` of length $|V| + 1$ that consists of row offsets, and (ii) `elist` of length $|E|$ that consists of column indices. For a given vertex id v, its neighbours can be accessed in the range `elist[vlist[v] : vlist[v + 1]]`[2]. Additional arrays can be used for storing properties such as edge weights. In Figs. 3(a)-(b), we show a sample graph and its CSR representation. The majority of prior work on GPUs and graph analytics uses the CSR representation, and it serves as a baseline for comparison here.

IV. Elias-Fano Encoding

Our graph compression format is based on Elias-Fano [18] (EF) encoding for monotone sequences which dates back to 1970s. More recently, Vigna [19] described this encoding for compressing inverted indices, and he calls them *quasi-succinct* indices. Succinctness here refers to the classification of data structures that take close to the information theoretic lower bound in storage while still supporting efficient queries. We describe the main aspects of the encoding here and refer the reader to Vigna [19] for more details. Consider a monotonically increasing sequence of $n > 0$ natural numbers $0 \leq x_0 \leq x_1 ... \leq x_{n-1} \leq u$ where $u > 0$ is any upper bound on the last value. In standard binary encoding, such a sequence takes $n \lceil \log(u + 1) \rceil$ bits of storage. The EF representation encodes it as follows:

- The lower $l = max\{0, \lfloor \log u/n \rfloor\}$ bits of each x_i are stored contiguously in the *lower-bits* array.
- The remaining upper bits of each x_i are stored in the *upper-bits* array as unary coded gaps.

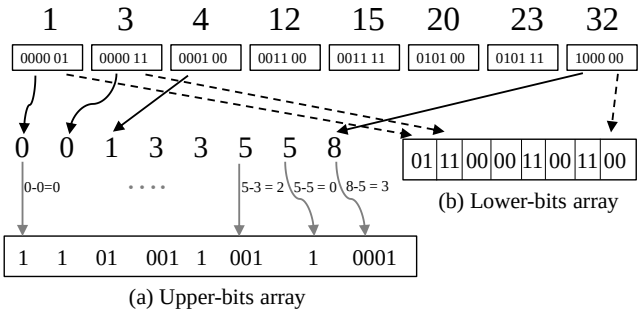


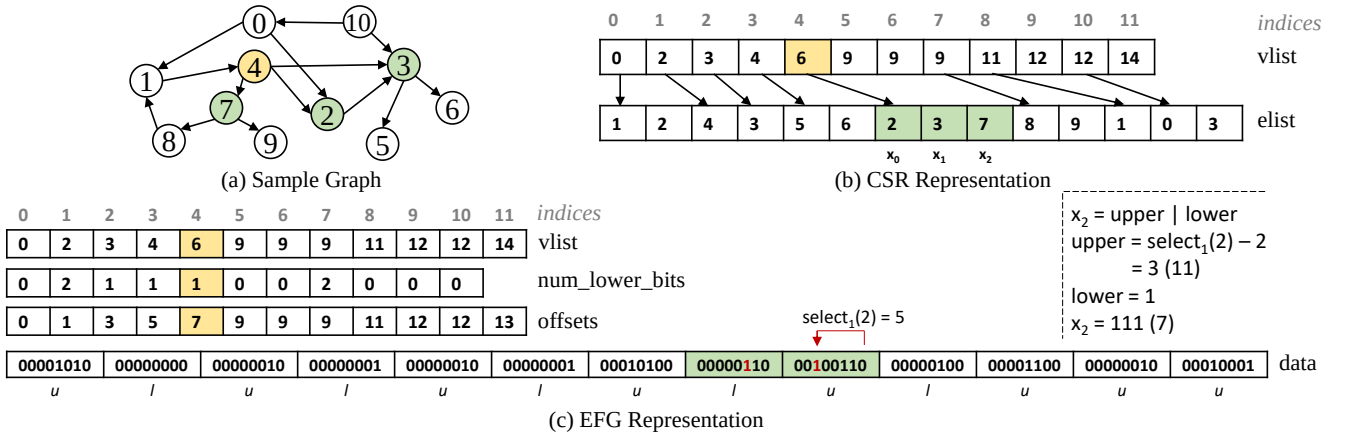
Fig. 2. A monotone sequence {1,3,...,32} coded with Elias-Fano encoding. For $n = 8$ and an upper bound $u = 32$, we need $\lfloor \log 32/8 \rfloor = 2$ lower bits. Successive gaps between the upper bits are encoded in unary with 1 as the stop-bit to form the upper-bits array. The lower bits are concatenated directly.

See Fig. 2 for an example. The lower-bits array stores the 2 lower bits from each element. The remaining upper-bits are treated as integers (e.g., $1000_b \rightarrow 8$), and successive differences between them are encoded in unary (e.g., 8 - 5 = 3 which is 000) with a stop bit (1) and concatenated to form the upper-bits array. The interesting property of this representation is that it takes at most $n(2+\lceil \log u/n \rceil)$ bits in total to represent the sequence, which is quite close to the lower bound for monotone sequences [19]. In Fig. 2, the binary representation needs $6 * 8 = 48$ bits, whereas the EF representation needs 32 bits (16 each for the lower and upper halves).

A. Decoding

To decode x_i, we need to recover and combine the lower and upper bits. The lower part can be readily recovered with a random access in the *lower-bits* array. The upper part of x_i can be recovered with a $select_1(i) - i$ operation on the *upper-bits* array, where $select_1(i)$ is defined as the operation that returns the position of the i^{th} (0-indexed) set bit from the start position. For example, to recover the upper bits of x_4, we compute $select_1(4) - 4 = 7 - 4 = 3 \equiv 11_b$, since the position of the 4^{th} (0-indexed) set bit is 7. The lower bits of x_4 are `lower[4]` = 11_b, and thus `upper|lower` is $1111_b = 15$, which is indeed x_4. The *select* operation is a foundational piece for information-retrieval systems. Vigna [19] describes *forward pointers* as a straightforward way to get on-average constant time *select* operations. Forward pointers store precomputed values of $select_1$ at multiples of some chosen quantum size. Now, performing $select_1(i)$ reduces to a *select* starting at the nearest forward pointer boundary. We discuss forward pointers in more detail in Sec. VI-C. Techniques such as broadword programming [20] or special hardware instructions such as Intel's bit manipulation instructions (BMI) can be used to

[2][start:stop] refers to the right open interval [start, stop)



(a) Sample Graph (b) CSR Representation (c) EFG Representation

Fig. 3. (a) A sample graph, (b) its CSR representation, and (c) its EFG representation. Node 4 and its neighbours are highlighted with yellow and green respectively. u and l denote upper and lower bits, respectively. This is a small example for illustration which does not benefit from compression.

accelerate *select* operations. This works for CPU applications since an application can replace its memory accesses with a call to the decoder, and the rest of the application remains unchanged. Our problem is different in two important ways:

1) **Multiple Values**: In the GPU setting, we do not decode a single value in a list. Rather, a set of threads needs to decode a set of values. If each thread were to call *select* independently, that would be wasteful since $select_1(i+1)$ would not use the work done during $select_1(i)$.
2) **Multiple Lists**: The values are not from the same list. We have a set of nodes whose neighbours we wish to explore. Each node's neighbour list is compressed with EF. The goal is to map this expansion of multiple non-contiguous compressed lists of different sizes in a load balanced way on the GPU's grid.

V. Elias-Fano Graph (EFG) Format

We propose the Elias-Fano Graph (EFG) format as a GPU friendly compressed graph representation based on the EF encoding scheme. Each neighbour list is individually encoded with EF, and the overall graph is laid out in a format that is similar to CSR. The only requirement for encoding the neighbour lists with EF is that the original sequences should be in sorted order.

In Fig. 3(c), we show the sample graph encoded in the EFG format. The representation consists of four arrays: `vlist`, `num_lower_bits`, `offsets`, and `data`. The first three arrays are indexed with the vertex id. The `vlist` array is similar to the one used in CSR. This array gives us constant time access to the degree of a node (i.e., $deg_i = vlist[i+1] - vlist[i]$), which is the number of elements in the compressed neighbour list. Unlike CSR, this array is not used directly for indexing the data. Instead, a separate `offsets` array stores the exclusive prefix sum of raw offsets into the compressed `data` array. The `num_lower_bits` array stores the number of lower bits used while encoding the vertex's neighbour list with EF. We use the folly [21] library for encoding the lists. The data is stored as `forward pointers`, `lower bits`, `upper bits`, in that order, in our implementation, and each section is byte aligned. There are no forward pointers in this example. To decode the neighbours of a node, we first recover its degree, the number of lower bits, and offsets in the `data` array. For example, node 4 in Fig. 3 has a degree of 3, uses 1 lower bit per value, and its raw data is stored in 2 bytes. The three neighbours of node 4 are {2,3,7}, and the list is encoded the same way as the previous example in Fig. 2. To recover x_2 (i.e., the third element), we compute $select_1(2) - 2 = 5 - 2 = 3 \equiv 11_b$ for the upper half and 1_b for the lower half for a total of $111_b = 7$, which is indeed the third neighbour of node 4[3].

The EFG format compresses the structure of the graph. A graph may have additional data such as edge weights, which can be arbitrary floating point values. Compressing such auxiliary data is beyond the scope of this work.

VI. Compressed Graph Traversal

We describe our implementation for GPU based traversal for EFG graphs in stages. We look at the top-down decomposition of the problem first since it shares similarities with a typical CSR based implementation. This is followed by our solution for list decompression on GPUs. Finally, we discuss extensions to other applications such as SSSP and PageRank.

A. Load Balanced Partitioning

The main operation of interest in a GPU based traversal is the load-balanced expansion of the frontier of vertices (first two lines of Alg. 1). Consider the example in Fig. 4. The frontier consists of four vertices whose out-degrees are {2, 3, 2, 1} for a total of 8 edges that emanate from this frontier. We need to visit these 8 edges to recover the destination vertices and eventually check a property such as the visited flag or a distance value. In an ideal mapping, 8 threads would explore the 8 edges. To achieve such a mapping [1], the following steps are taken: First, we compute the exclusive prefix sum

[3]In Fig. 3, since we show the actual layout of bits in memory, the LSB is at the right end. Hence, *select* goes from right to left. Elsewhere in the paper, we use the prior convention of scanning bits from left to right for readability.

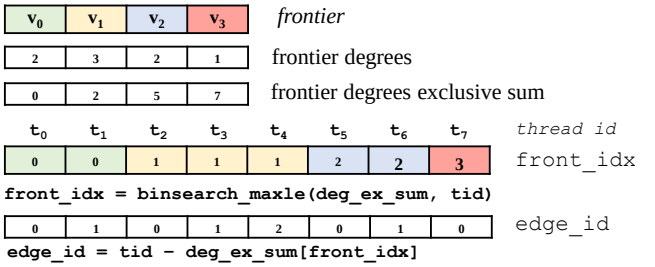


Fig. 4. Mapping of edges to threads in frontier expansion. `binsearch_maxle` returns the index of the largest value less than or equal to the search value.

(Sec. III-C) of the degrees of these vertices. Next, each thread does a binary search in the exclusive sum array to find the index of the largest value less than or equal to the thread id. The returned index is an index in the frontier array, which is the vertex whose edge this thread needs to visit. For example, thread t_4 searches for 4 in the exclusive sum array, which returns the index 1 since 2 is the largest value ≤ 4. Hence, thread t_4 needs to visit a neighbour of v_1. Finally, to know which edge in particular to visit, the thread subtracts the exclusive sum from the thread id. Thread t_4 needs to visit 4 - 2 = 2 (i.e., 3rd edge) of node v_1. Our implementation uses the thrust library's vectorised search functions for this phase. Notice that once a thread knows which edge it needs to visit, it can readily recover it in constant time in the CSR representation. Recall that the destination of the n^{th} edge of the i^{th} vertex in CSR is at `elist[vlist[i] + n]`. This does not hold true in the EFG representation. We use this partitioning at the top level so that each thread block is responsible for roughly equal number of edges. However, within a thread block, we need a new implementation for decompressing the data.

In summary, the top level decomposition partitions the edges equally between thread blocks. The remainder of the problem can be formulated as that of decompressing multiple neighbour lists, including partial lists, within a thread block. We describe the solution as a progressive generalisation of specific cases.

B. Decompressing A Single List

Consider the specific case of decompressing a single list within a thread block. This is, in fact, sufficient for implementing the traversal, albeit in an inefficient way due to skews in the degree distribution. Instead of a load-balanced split of edges, as described earlier, we can assign a thread block to each node in the frontier, and the thread block decompresses that node's edge list. Recall from Sec. IV-A that we are mainly interested in computing $select_1(i) - i$ for each of the set bits in the upper-bits array, where $select_1(i)$ returns the position of the i^{th} set bit in a bitstream. Since the lower-bits array can be accessed in constant time, we focus only on the upper-bits array. The pseudo-code for decompressing a single list is shown in Alg. 2 with a corresponding example in Fig. 5.

In this example, a thread block of 4 threads (`DIMX`) collectively decompresses a list by computing $select_1(i) - i$ for

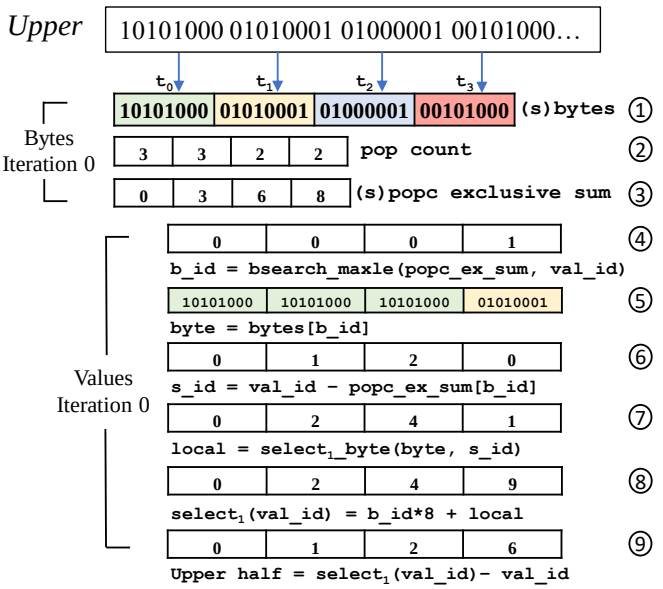


Fig. 5. Decompressing a single list within a thread block. The threads collectively compute $select_1(i) - i$. Shared data structure are marked with (s). `val_id` is the same as `thread_id` in the first iteration.

each set bit. The upper bits array (*Upper*) resides in global memory. Each thread ① loads a byte to a shared bytes array. Shared data resides in a fast user managed cache accessible by all threads within the block. Next, each thread ② computes a population count of the set bits in its respective local byte with the `popcount` instruction. The counts are {3, 3, 2, 2} here. The number of set bits in a byte is also the number of values that the byte will eventually produce. Since these values are not uniform, we need a load balanced split similar to the approach described earlier for splitting edges based on unequal degrees. Hence, the threads collectively compute the ③ exclusive prefix sum (Sec. III-C) over their respective pop-counts. The results are stored in a shared array. The exclusive scan also returns the total pop-count (10 here), which is the total number of values to be decoded. This starts the inner loop where each thread decodes one value in each iteration. We show the first iteration in the example.

In the first iteration, the index of the value to be decoded is the same as the thread id (line 14, Alg. 2). Thread *i* needs to find the position of the i^{th} set bit. Since we have the exclusive prefix sum of pop-counts, each thread does a ④ binary search in the array to narrow the search to a ⑤ target byte. In the example, we see that the first three threads get the first byte whereas the fourth thread gets the second byte. The difference between the search parameter and the exclusive sum gives the new search parameter for ⑥ selecting a bit within a byte, called $select_1\_byte$ hereafter (e.g., thread t_2 finds the position of the 2nd set bit in `10101000`). A byte has 256 possible values, and for each value, there can be at most 8 positions. Hence, $select_1\_byte(i)$ is implemented with a ⑦ lookup table of 2 KiB in constant memory. The number of preceding bits in the list is ⑧ added to the result of $select_1\_byte$ to produce the

Algorithm 2 decompress_single_list (upper, n_bytes)

```
1:  __shared__ s_exsum[DIMX];
2:  __shared__ s_bytes[DIMX];
3:  prev_vals = 0;
4:  b_iters = ceil(n_bytes / DIMX);
5:  for (i=0; i <b_iters; i++) do
6:      byte_id = i * DIMX + thread_id;
7:      byte = (byte_id <n_bytes) ? upper[byte_id] : 0;
8:      s_bytes[tid] = byte;                                   (1)
9:      popc = __popc(byte);                                  (2)
10:     total_vals = do_ex_sum(popc, s_exsum);                (3)
11:     CTA_sync();
12:     val_iters = ceil(total_vals / DIMX);
13:     for (j = 0 ; j <val_iters; j++) do
14:         val_id = j * DIMX + thread_id;
15:         if (val_id <total_vals) then
16:             tb_id = bsearch_max_le(s_exsum, val_id);      (4)
17:             target = s_bytes[tb_id];                      (5)
18:             s_id = val_id - s_exsum[tb_id];               (6)
19:             select_result = select1_byte(target, s_id);   (7)
20:             bits_before_me = (i*DIMX + t_byte_id)*8;
21:             select_result += bits_before_me;              (8)
22:             global_val_id = prev_vals + val_id;
23:             upper_half = select_result - global_val_id;   (9)
24:             lower_half = get_lower_half (global_val_id);
25:             decoded_val = combine(upper_half, lower_half);
26:             // Use value in analytics
27:     prev_vals += total_vals;
28:     CTA_sync();
```

global $select_1$ value, and (9) subtracting the index of the value from it gives the upper-half of the decompressed value. This is combined with the lower half and used in the application.

C. Decompressing A Partial List

Assigning a list to a block has obvious limitations since some lists can be much longer than others, particularly in real-world graphs with a power-law degree distribution. The next logical step is to split long lists across thread blocks in order to avoid over-subscription of thread blocks. Since a list can span multiple blocks, the problem can be formulated as that of decoding values in some range $[a, b]$ within a list, where $0 \leq a \leq b < n$ for a list with n elements. This is achieved with the use of forward pointers. The conventions herein are based on the folly [21] library. For a list of size n and a quantum parameter $k > 0$, we store $\lfloor n/k \rfloor$ forward pointers that enable fast $select_1(i)$ operations for $i = \{k-1, 2k-1, ..., \lfloor n/k \rfloor k-1\}$. For instance, $k = 8$ stores values for $select_1(8-1=7)$, $select_1(15)$, and so on. The pointers actually store $select_1(i)-i$ rather than $select_1(i)$ since it takes fewer bits, and i can be added later if needed. Consider the example in Fig. 6, where forward pointers are stored for $k = 8$, and a thread block of 4 threads needs to decode values $[x_{12}, x_{19}]$ in the list. The closest preceding pointer for x_{12} is at $\texttt{forward}[\lfloor (12+1)/8 \rfloor - 1]$, which is the first pointer, and it corresponds to x_7. We get a

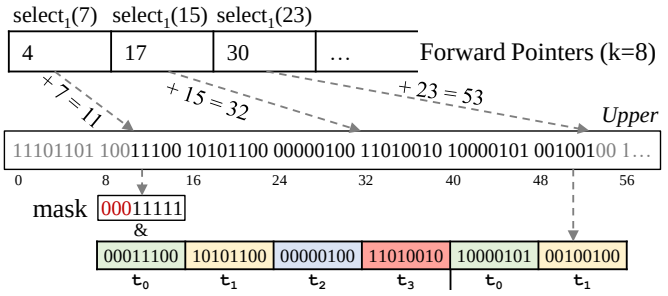


Fig. 6. Decompressing a partial list within a thread block. Forward pointers store the value of $select_1(i) - i$ at regular intervals ($k = 8$ here). The thread block loads the bytes between two boundary pointers.

bit position of $4 + 7 = 11$, as shown in the figure. Similarly, the last pointer of interest corresponds to x_{23}, which gives a position of 53. Thus, this thread block only needs to scan between bits 11 and 53.

Splitting large lists across blocks helps with over-subscription, but not with under-subscription since small lists are still assigned to individual blocks. The last missing piece in the general solution is the ability to decode multiple lists within a block, which we describe next.

D. Decompressing Multiple Lists

The load-balanced partitioning described in Sec. VI-A assigns an equal number of edges to different thread blocks. Hence, each thread block is responsible for decoding a number of lists. The single-list solution in Alg. 2 changes in a few ways to account for multiple lists. First, we now need an additional outer loop for lists. That is, we need three levels of nesting that go from lists to bytes to decoded values, and the bytes come from multiple lists instead of a single list. Second, notice that at the innermost level, a thread's view is extremely local. Each thread needs to compute $select_1(i) - i$ for some i, and in order to go from $select_1\_byte$ on a local byte to the global decoded value, it needs two pieces of information: i) What the position of the byte in the list is, and ii) What the global index (i) of the value in the list is. This requires a segmented prefix sum (Sec. III-C), where the segments represent list boundaries. We look at an example next to explain this.

In Fig. 7, a thread block of 6 threads decompresses multiple lists. Steps marked with $\star$ are different from the single-list case. Like the single-list case, each thread needs to load a byte to the shared bytes array. However, since the bytes now come from multiple non-contiguous lists of different sizes, we use the familiar idiom of a prefix sum over the number of bytes per list followed by a binary search to map the bytes to threads. Like the single-list case, these bytes would produce a number of values, and we need an exclusive prefix sum over the local popcounts for use in the innermost loop. In addition, we need to mark the list boundaries. Threads that start a new list mark a flag in the `is_list_start` array. For e.g., thread t_2 starts $list_1$, and marks its bit as 1. This flag array is used in conjunction with the popcounts to create an additional segmented prefix sum array, where each prefix sum runs only

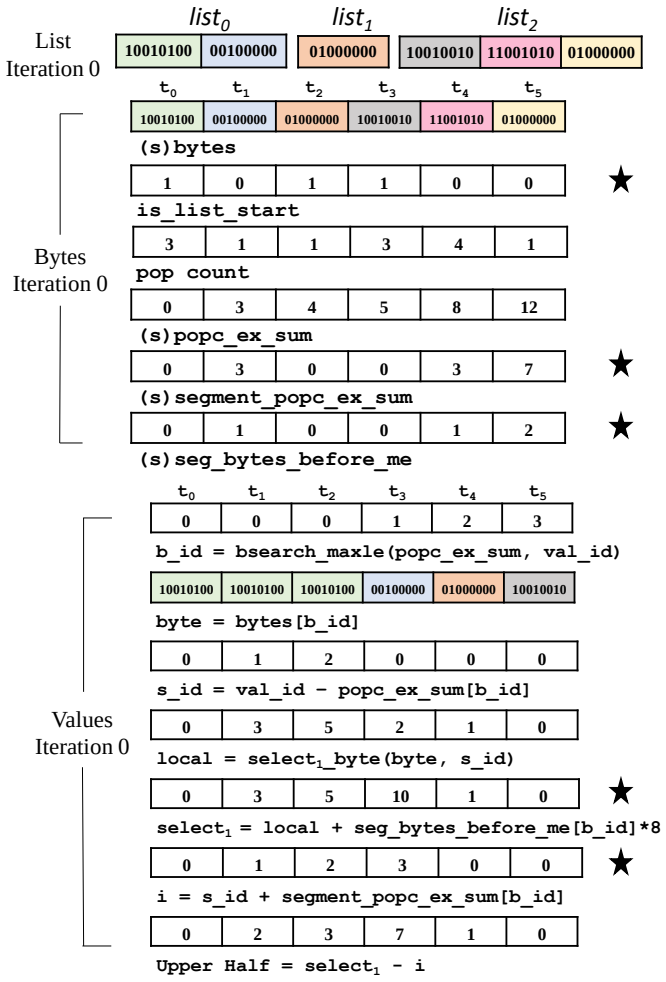


Fig. 7. Decompressing multiple lists within a thread block. Steps marked with $\star$ are different from the single-list case.

within the list. While thread t_4's block-wide exclusive sum is 8, its exclusive sum within the list is 3. Similarly, the `seg_bytes_before_me` array keeps track of the number of preceding bytes *within* the list. In the innermost loop for decoding the values, the threads identify their target bytes by searching the block-wide exclusive array like the single-list case. After computing the local $select_1\_byte(i)$ value, each thread looks up the number of preceding bits within the list and adds it to get the global $select_1$ value. For eg., since t_3's target byte id is 1, which has one preceding byte in the list, 8 is added to its local select value of 2 for a total of 10. On the other hand, t_4's target byte is byte 2, which is the start of a list. Hence, nothing is added to its local select value. Similarly, the segmented exclusive sum of the target byte is added to the id within the byte to get the global id for the value. Finally, the upper half is computed as $select_1(i) - i$ and combined with the lower half to recover the value. The multi-list solution combined with partial lists gives us a fully general solution.

E. Partially Sorting the Frontier

In a parallel BFS, nodes are added to the frontier in arbitrary order. Hence, the lists that a thread block decompresses during frontier expansion may be scattered across memory. One way to improve the locality of accesses is to sort the nodes in a frontier, so that thread blocks touch non-overlapping regions of memory and threads within a block touch increasing memory addresses. However, sorting the frontier at each level is expensive. Fortunately, we do not need an exact sort since this is an optimisation and does not affect correctness. We use the CUB [22] library's GPU implementation for radix-sort to sort only the higher order bits. We sort 65% of the bits (i.e., we pretend as though the lower 35% bits do not exist) while sorting the frontier. We see an average improvement of 9% (max 33%) in runtime from this optimisation.

F. SSSP and PageRank

The general traversal pattern extends to many applications, and we extend it to single source shortest paths (SSSP) and PageRank here. The SSSP implementation is similar to BFS except that instead of a boolean visited property, we need to relax a floating point distance value. We mark the relaxed nodes atomically in a bitmap of size $\mathcal{O}(|V|)$ and use a parallel scatter to create the frontier from the bitmap. Note that the edge weights in the input graph also require $\mathcal{O}(|E|)$ in storage since we compress the graph structure but not the weights. Hence, SSSP gets in the out-of-core regime much before BFS. Compressing weights is outside the scope of this work. In PageRank, all the nodes are active in each iteration, and we do not need a frontier (i.e., the frontier comprises all the nodes). The PageRank value of a destination is updated atomically once the edge is decoded.

VII. Related Work

The literature on graph compression methods is vast [7]. Perhaps the most widely-used method for compressing large web-graphs is BV [23]. BV exploits locality within lists and similarity across lists. The gaps between neighbours are coded with a variable length code and reference chains are used for capturing the similarity between different lists. Due to sequential dependencies, it is diffult to adapt BV for GPUs. Graph reordering methods such as LLP [24], Shingle [25], and BP [26], reorder the graph to reduce gaps between node labels as smaller gaps take fewer bits to encode. LogGraph [27] and Ligra+ [28] focus on the decompression performance in the parallel CPU setting. Ligra+ uses run length encoded byte/nibble codes for compression. The current state of the art for GPU based traversals on compressed graphs is CGR by Mo Sha et al. [8]. The compression scheme breaks lists into intervals and residuals, followed by gap transformation, and gaps are encoded with a variable length code.

We compare our proposed EFG representation with CGR and Ligra+. Ligra+ uses a direction optimising implementation by default, which can switch between top-down and bottom-up flows in the application. However, this requires storing in-edges in addition to out-edges, which doubles the storage

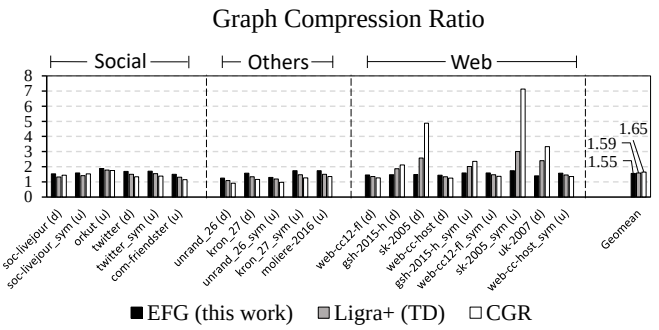


Fig. 8. Compression ratio for EFG (this work), Ligra+(TD) [28], and CGR [8]

requirements for directed graphs. To maintain parity across implementations, we use Ligra+ in the top-down (TD) mode.

VIII. Results

We evaluate CSR, EFG, Ligra+(TD), and CGR graph representations in a number of experiments. We use cugraph [1], [10] for CSR graphs on the GPU and the reference implementations by the authors for for CGR [8] and Ligra+ [28]. The majority of experiments are performed on a Titan Xp GPU (250W TDP) with 12 GiB memory while we also scale some experiments to a V100 GPU with 32 GiB memory. CPU measurements for Ligra+(TD) are performed on a sytem with 2x E5-2696 processors (44 cores, 88 threads, 2x145W TDP). We choose 100 random starting nodes for BFS and SSSP and average the results. Edge weights are initialised to random floating point values between 0 and 1 for SSSP. PageRank runs are capped at 50 iterations. We modified cugraph to support out-of-core processing in BFS and SSSP. The forward pointer quantum parameter k is set to $k = 512$ in EFG.

A. Compression Ratio

We show the compression ratio of EFG, Ligra+(TD), and CGR relative to CSR in Fig. 8, and absolute numbers are provided in Table II. The graphs are grouped by categories in Fig. 8 as social, web, and other graphs. EFG achieves a compression ratio of 1.55x over CSR, and the compression is quite consistent across graphs. CGR shows a significant skew in the compression ratio where it excels at web-graphs, but in other graphs, its ratio is lower than EFG. CGR's average compression ratio across all graphs is 1.65x. Ligra+(TD) achieves an overall compression ratio of 1.59x with similar trends as CGR in that it is better at web-graphs than other types of graphs. Overall, EFG achieves the best compression in social and other graphs whereas CGR is best with web-graphs. The reason for the disparity in web-graphs is two fold: i) Web-graphs have strong locality where long runs of contiguous values are common. The interval and residual representation in CGR and the run length encoded version of gaps in Ligra+ lends itself well to this structure. ii) EF encoding has limitations in compressing such sequences, and solutions [29] that address it exist, although they were not incorporated here. We discuss them in Sec. IX. Lower compression in web-graphs notwithstanding, EFG has other benefits. EFG's runtime for traversals is lower than CGR and Ligra+(TD) in all cases. It is also inexpensive to estimate EFG's storage requirements since the upper bound on storage in EF only depends on the number of elements and the largest value (Sec. IV) in each list. That is, we do not need to compress the graph to know how well it will compress with EFG.

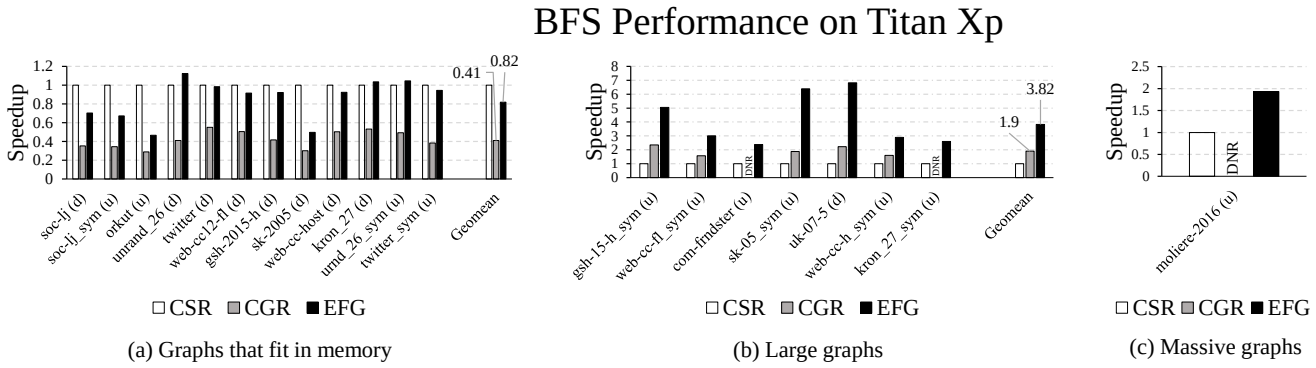
TABLE II
BFS performance on Titan Xp and 2x E5-2696 v4

Graph	$\|V\|(\|E\|)$	Size in GiB (Runtime in ms)			
		GPU			CPU
		CSR [10]	CGR [8]	EFG	Lg+TD [28]
soc-lj (d)	4.85 M (68.9 M)	0.28 (8)	0.19 (22)	0.18 (11)	0.21 (77)
soc-lj_sym (u)	4.85 M (86.22 M)	0.34 (10)	0.22 (28)	0.21 (14)	0.24 (90)
orkut (u)	3.07 M (234.3 M)	0.88 (13)	0.5 (45)	0.47 (28)	0.5 (140)
urnd_26 (d)	67.1 M (1.07 B)	4.25 (525)	4.72 (1277)	3.4 (467)	3.92 (1523)
twitter (d)	41.6 M (1.47 B)	5.63 (234)	4.23 (425)	3.33 (238)	3.77 (1589)
web-cc-fl (d)	80.76 M (1.77 B)	6.92 (249)	5.48 (493)	4.76 (272)	5.13 (2193)
gsh-15-h (d)	68.66 M (1.8 B)	6.97 (160)	3.3 (385)	4.73 (174)	3.74 (1007)
sk-05 (d)	65.61 M (1.95 B)	7.45 (57)	1.53 (190)	5.02 (115)	2.89 (533)
web-cc-host (d)	89.11 M (2.03 B)	7.93 (303)	6.36 (603)	5.52 (328)	5.92 (2530)
kron_27 (d)	63.07 M (2.12 B)	8.15 (511)	7.01 (962)	5.18 (494)	6.07 (1900)
urnd_26_sym (u)	67.1 M (2.14 B)	8.25 (793)	8.59 (1610)	6.39 (758)	6.93 (2445)
twitter_sym (u)	41.6 M (2.40 B)	9.11 (348)	6.61 (906)	5.34 (368)	5.89 (3379)
gsh-15-h_sym (u)	68.66 M (3.05 B)	11.62 (1824)	4.94 (776)	7.33 (361)	5.77 (2198)
web-cc-fl_sym (u)	80.76 M (3.39 B)	12.92 (2140)	9.48 (1360)	8.17 (713)	8.84 (7589)
com-frndster (u)	65.61 M (3.61 B)	13.7 (2387)	11.98 (DNR)	9.15 (1006)	10.54 (4082)
sk-05_sym (u)	65.61 M (3.64 B)	13.75 (2062)	1.93 (1098)	7.9 (323)	4.58 (1326)
uk-07-05 (d)	105.22 M (3.74 B)	14.32 (1444)	4.3 (648)	10.31 (212)	5.97 (1009)
web-cc-h_sym (u)	89.11 M (3.87 B)	14.76 (2441)	10.89 (1519)	9.37 (842)	10.11 (7306)
kron_27_sym (u)	63.07 M (4.22 B)	15.97 (2600)	12.61 (DNR)	9.23 (997)	10.87 (4128)
moliere-16 (u)	30.22 M (6.68 B)	25.1 (4149)	18.65 (DNR)	14.5 (2148)	16.82 (5138)

B. BFS Performance

We first look at the relative performance of BFS for CSR, CGR and EFG representations on a Titan Xp GPU in Fig. 9 and Table II. As noted in Sec. II, the set of graphs break down into three categories. The first set of small graphs fit in memory even in the CSR representation, and here cugraph's CSR implementation performs the best on average. EFG achieves 0.82x of this performance. Note that this is still 2.1x faster than CGR. Unless one wishes to save memory for other uses, there is little reason to use compressed representations when the graphs fit in memory. The next set of graphs exceeds the memory capacity in the CSR representation. We can see that EFG performs the best here as all the graphs fit in memory after compression. EFG has an average speedup of 3.8x (max 6.8x) over the out-of-core CSR implementation and a 2x speedup over CGR in this region. Notice that EFG's runtime is lower that CGR in all cases, including web-graphs. For example, EFG's runtime is 3.4x lower than CGR (323 ms v/s 1098 ms) for `sk-2005_sym`. The last graph in the set takes more than 12 GiB in storage even after compression with EFG. However, compression is still beneficial here since it reduces the volume of data transferred over the interconnect. EFG sees a speedup of 1.8x over CSR here. Since CGR does not support out-of-core processing, experiments where CGR is unable to process graphs are marked as 'did not run' (DNR).

Next, we look at Ligra+(TD)'s performance on the CPU in Table II. For the first set of graphs that fit in GPU memory, cugraph's CSR implementation is faster by 6.7x on average than Ligra+(TD) and our EFG implementation is faster by



(a) Graphs that fit in memory (b) Large graphs (c) Massive graphs

Fig. 9. BFS performance relative to CSR (higher is better) on a Titan Xp GPU with 12 GiB memory

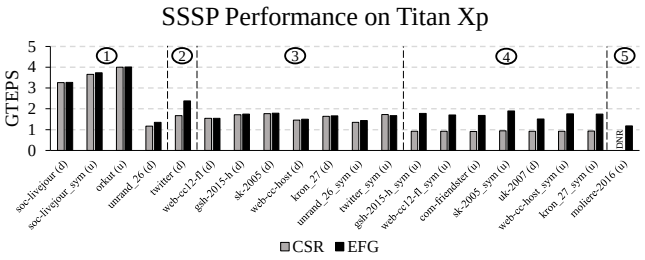
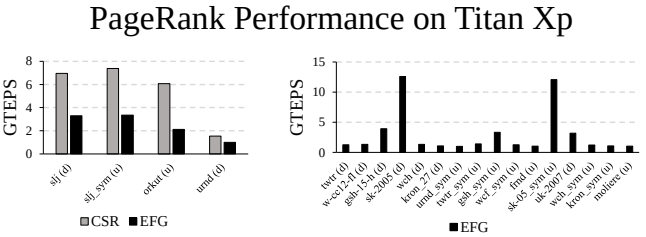


Fig. 10. SSSP performance measured in GTEPS. Regions: (1) CSR & EFG graphs fit in memory. (2) EFG fits entirely but CSR's weights do not fit. (3) Weights do not fit in either format. (4) Neither edges nor weights fit for CSR; weights do not fit for EFG. (4) EFG's edges and weights do not fit.



(a) Small Graphs (b) Larger Graphs

Fig. 11. PageRank performance measured in GTEPS

5.5x. However, as we get to larger graphs, the gap between the GPU based CSR implementation and CPU based Ligra+(TD) shrinks to 1.4x in the second set of graphs. This is due to the limited bandwidth of the PCIe interconnect for the out-of-core GPU implementation. At the same time, EFG maintains a speedup of 5.6x over Ligra+(TD) in this region since the graphs fit in GPU memory after compression. Since the last graph exceeds 12 GiB even with EFG compression, the speedup over Ligra+(TD) reduces to 2.3x.

C. SSSP and PageRank Performance

We show the performance of SSSP and PageRank on the Titan Xp GPU in Figs. 10 and 11, respectively. CGR is not evaluated here as an implementation was not available. Since SSSP requires an additional edge weights array, only small graphs fit entirely in memory. Nevertheless, compression is beneficial as one can fit the graph structure and stream the weights. CSR graphs fit entirely in memory in region 1 in Fig. 10, and EFG graphs fit entirely in regions 1 and 2. In region 3, the weights are streamed in for both the formats. In region 4, EFG continues to stream the weights, whereas CSR has to stream both the weights and edges. In region 5, we exceed the memory capacity even for edges in the EFG format. In regions 2 and 4, where EFG has the advantage over CSR due to more data fitting in memory, we see speedups of 1.41x and 1.85x respectively. The performance is quite similar in region 3 as both CSR and EFG need to stream weights, which is the bottleneck. PageRank's performance (Fig. 11) shows a similar trend as BFS in that cugraph's in-memory implementation for CSR performs better than EFG. PageRank is not evaluated for CSR for large graphs since an out-of-core version was not available.

D. Graph Reordering

Graph reordering is a common preprocessing technique that is used for different purposes. In the context of graph compression, reordering methods relabel the nodes to reduce gaps between successive neighbours since smaller gaps can be encoded more efficiently. Graph reordering can also be used to improve locality and performance. We use two graph reordering methods - BP [26] that optimises for lower gaps, and HALO [5] that optimises for locality, and evaluate their impact on different compressed graph representations. We also evaluate random ordering as a pathological case since random ordering destroys all locality. The impact of different orderings on compression ratio and BFS' runtime performance on the Titan Xp GPU is shown in Fig. 12. The main observations are as follows:

Compression Ratio: EFG does not use a gap-based encoding. Hence, it is not affected by the distribution of gaps and its compression ratio is virtually unchanged for all the ordering methods (Fig. 12(a)). Note that random ordering does not negatively impact the compression ratio. The storage bounds for EF only depend on the largest value in a list and the number of elements. On the other hand, CGR and Ligra+(TD) benefit from lower gaps (9-15% improvement), and their compression ratio deteriorates in random ordering (18-32% deterioration)

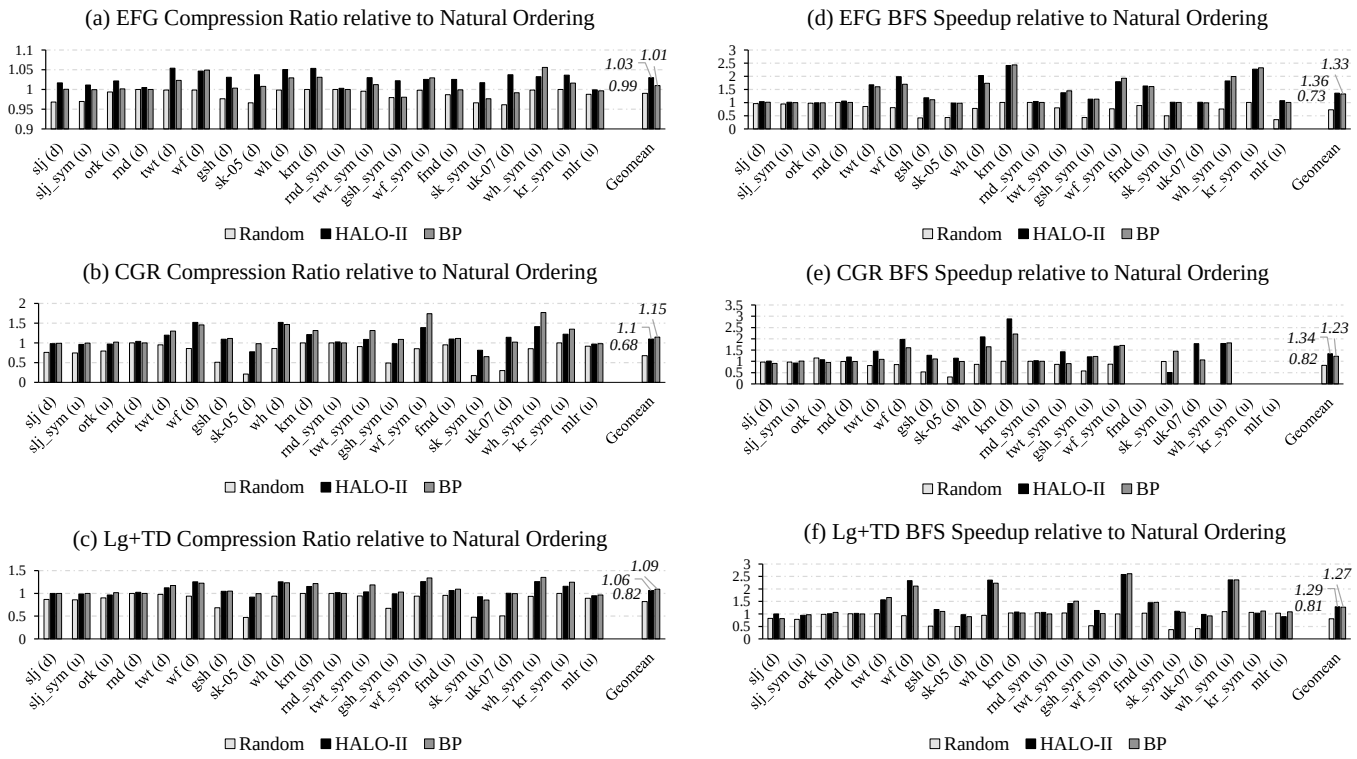


Fig. 12. Impact of graph reordering on compression ratio and BFS performance

(Figs. 12(b,c)). For example, `sk-05`'s compression drops to 0.2x-0.5x with random ordering, whereas `wh_sym`'s compression improves by 1.3x-1.7x with BP for these encoding schemes.

BFS Performance: All the representations benefit from improved locality and are impacted negatively by random ordering (Figs. 12(d,e,f)). This is expected as the ordering affects the memory access pattern irrespective of whether a graph is compressed or not. The runtime performance of BFS on a GPU depends on factors such as memory coalescing, read amplification, the prefetcher's effectiveness, etc., and locality friendly orderings improve these characteristics while random ordering hurts them. HALO improves performance by 1.26x-1.32x on average, and random ordering reduces the performance to 0.65x-0.8x across different graph representations.

E. Scaling to Larger GPUs

We also confirmed the main experimental trends on a newer V100 GPU with 32 GiB HBM memory that supports internal bandwidth of 731.3 GiB/s. The V100 GPU is still connected to the host with a PCIe 3.0 link that supports peak bandwidth of 12.1 GiB/s. The difference between the internal and external bandwidth is even greater here, and we expect compression to be more beneficial in such cases. In Table III, we show the BFS performance metrics for some of the larger graphs for CSR, EFG, and CGR representations. EFG achieves 0.67x of CSR's performance when CSR graphs fit in memory, and it shows a speedup of 6.55x when CSR graphs do not fit. The speedup is higher due to the greater disparity between internal and external bandwidth (∼60x in V100 v/s ∼35x in TitanXp). EFG also shows a speedup of 1.48x over CGR.

TABLE III
BFS RUNTIME ON A V100 GPU (32 GIB MEMORY)

Graph	$\|V\|(\|E\|)$	Size in GiB (Runtime in ms)		
		CSR [10]	CGR [8]	EFG
com-frndster (u)	65.61 M (3.61 B)	13.7 (316)	11.98 (389)	9.15 (349)
sk-05_sym (u)	65.61 M (3.64 B)	13.75 (77)	1.93 (735)	7.9 (153)
uk-07-05 (d)	105.22 M (3.74 B)	14.32 (68)	4.3 (169)	10.31 (117)
web-cc-h_sym (u)	89.11 M (3.87 B)	14.76 (273)	10.89 (445)	9.37 (340)
kron_27_sym (u)	63.07 M (4.22 B)	15.97 (325)	12.61 (426)	9.23 (370)
moliere-16 (u)	30.22 M (6.68 B)	25.1 (189)	18.65 (341)	14.5 (296)
kron_28_sym (u)	121.23 M (8.47 B)	32.46 (7319)	26.43 (1170)	19.64 (1012)
kron_29 (d)	232.99 M (8.53 B)	33.52 (6178)	30.46 (DNR)	22.95 (1043)

F. Compression Time

Since compression is an offline step for static graphs, optimising the compression performance was not a priority in this work. Nevertheless, compressing graphs with EF is quite efficient, and we were able to compress all the graphs in this work with EFG in a few minutes on the 2x E5-2696 v4 server. Ligra+ also took about the same time for compression whereas CGR took more than 30-45 minutes for several graphs.

IX. LIMITATIONS AND DISCUSSION

The EF encoding scheme works well when the distribution of values being compressed is random. However, in cases like

web-graphs, we typically have long runs of contiguous values. This is a good case for gap based encoding schemes, but EF does not benefit from smaller gaps. To use a motivating example [29], consider a sequence $S = [0, 1, 2, ..., n-2, u-1]$ of length n where the first $n-1$ values are contiguous. This is a highly compressible sequence where the length of the run and the last value are sufficient to describe S. On the other hand, EF still uses $2 + \lceil \log u/n \rceil$ bits to encode each element, which is the same as any random sequence. The general approach to deal with this problem is to partition the sequence so that some partitions can be encoded more efficiently (e.g., with run-length coding). In PEF [29], the authors propose a method for selecting partition sizes in this model. We did not incorporate this here, but extensions to the EFG format are possible. The work described here can also be used in conjunction with methods such as virtual node compression [30] and other distributed or out-of-core solutions.

X. Conclusion

Since real-world graphs are typically larger than a GPU's memory capacity, a variety of solutions can be used to address the challenges involved in graph analytics on GPUs. In this work, we looked at graph compression as a means to accommodate large graphs in GPU memory. Traditional graph compression schemes, while effective in compressing the graphs, cannot be used easily on GPUs due to the sequential and dependent nature of the decompression phase. We proposed the Elias Fano Graph (EFG) representation as a GPU-friendly compressed graph representation that encompasses several desirable properties in terms of compression ratio and the decompression performance. The seemingly irregular problem of decompression can be broken down into common high performance primitives such as parallel scans and searches, which results in an efficient and load-balanced implementation for graph traversals. We showed that we can compress several large graphs by a factor of 1.5x and outperform out-of-core approaches in traversal runtime by a factor of 3.8x-6.5x. Our implementation is also 1.45x-2x faster than the current state of the art in GPU based compressed graph traversals. The code for this work is published at https://github.com/pgera/efg.

References

[1] M. Bisson, M. Bernaschi, and E. Mastrostefano, "Parallel distributed breadth first search on the kepler architecture," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 7, pp. 2091–2102, 2015.

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

[3] 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.

[4] S. W. Min, V. S. Mailthody, Z. Qureshi, J. Xiong, E. Ebrahimi, and W.-m. Hwu, "Emogi: Efficient memory-access for out-of-memory graph-traversal in gpus," *Proceedings of the VLDB Endowment*, vol. 14, no. 2, pp. 114–127, 2021.

[5] P. Gera, H. Kim, P. Sao, H. Kim, and D. Bader, "Traversing large graphs on gpus with unified memory," *Proceedings of the VLDB Endowment*, vol. 13, no. 7, pp. 1119–1133, 2020.

[6] A. H. N. Sabet, Z. Zhao, and R. Gupta, "Subway: minimizing data transfer during out-of-gpu-memory graph processing," in *Proceedings of the Fifteenth European Conference on Computer Systems*, 2020.

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

[8] M. Sha, Y. Li, and K.-L. Tan, "Gpu-based graph traversal on compressed graphs," in *Proceedings of the 2019 International Conference on Management of Data*, 2019, pp. 775–792.

[9] D. Merrill, M. Garland, and A. Grimshaw, "High-performance and scalable gpu graph traversal," *ACM Transactions on Parallel Computing (TOPC)*, vol. 1, no. 2, pp. 1–30, 2015.

[10] "cugraph - rapids graph analytics library," https://github.com/rapidsai/cugraph, (Accessed on 10/02/2020).

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

[12] P. Boldi, A. Marino, M. Santini, and S. Vigna, "Bubing: Massive crawling for the masses," *ACM Transactions on the Web (TWEB)*, vol. 12, no. 2, pp. 1–26, 2018.

[13] R. Meusel, S. Vigna, O. Lehmberg, and C. Bizer, "Graph structure in the web—revisited: a trick of the heavy tail," in *WWW Companion*, 2014, pp. 427–432.

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

[15] 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*, ser. KDD '17, 2017.

[16] S. Beamer, K. Asanović, and D. Patterson, "The gap benchmark suite," *arXiv preprint arXiv:1508.03619*, 2015.

[17] D. Culler, J. P. Singh, and A. Gupta, *Parallel computer architecture: a hardware/software approach*. Gulf Professional Publishing, 1999.

[18] P. Elias, "Efficient storage and retrieval by content and address of static files," *Journal of the ACM (JACM)*, vol. 21, no. 2, pp. 246–260, 1974.

[19] S. Vigna, "Quasi-succinct indices," in *Proceedings of the sixth ACM international conference on Web search and data mining*. ACM, 2013, pp. 83–92.

[20] ——, "Broadword implementation of rank/select queries," in *International Workshop on Experimental and Efficient Algorithms*. Springer, 2008, pp. 154–168.

[21] "Folly: An open-source c++ library developed and used at facebook." https://github.com/facebook/folly, (Accessed on 08/04/2020).

[22] D. Merrill, "Cub," *NVIDIA Research*, 2015.

[23] P. Boldi and S. Vigna, "The webgraph framework i: compression techniques," in *Proceedings of the 13th international conference on World Wide Web*, 2004, pp. 595–602.

[24] 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*, 2011, pp. 587–596.

[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*, 2009, pp. 219–228.

[26] 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*, 2016, pp. 1535–1544.

[27] M. Besta, D. Stanojevic, T. Zivic, J. Singh, M. Hoerold, and T. Hoefler, "Log (graph) a near-optimal high-performance graph representation," in *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques*, 2018, pp. 1–13.

[28] J. Shun, L. Dhulipala, and G. E. Blelloch, "Smaller and faster: Parallel processing of compressed graphs with ligra+," in *2015 Data Compression Conference*. IEEE, 2015, pp. 403–412.

[29] G. Ottaviano and R. Venturini, "Partitioned elias-fano indexes," in *Proceedings of the 37th international ACM SIGIR conference on Research & development in information retrieval*, 2014, pp. 273–282.

[30] G. Buehrer and K. Chellapilla, "A scalable pattern mining approach to web graph compression with communities," in *Proceedings of the 2008 International Conference on Web Search and Data Mining*, 2008, pp. 95–106.