Compressed In-memory Graphs for Accelerating GPU-based Analytics

Noushin Azami
Texas State University
noushin.azami@txstate.edu

Martin Burtscher
Texas State University
burtscher@txstate.edu

***Abstract*—Processing large graphs has become an important irregular workload. We present Massively Parallel Log Graphs (MPLG) to accelerate GPU graph codes, including highly optimized codes. MPLG combines a compressed in-memory representation with low-overhead parallel decompression. This yields a speedup if the boost in memory performance due to the reduced footprint outweighs the overhead of the extra instructions to decompress the graph on the fly. However, achieving a sufficiently low overhead is difficult, especially on GPUs with their high-bandwidth memory. Prior work has only successfully employed similar ideas on CPUs, but those approaches exhibit limited parallelism, making them unsuitable for GPUs. On large real-world inputs, MPLG speeds up graph analytics by up to 67% on a Titan V GPU. Averaged over 15 graphs from several domains, it improves the performance of Rodinia's breadth-first search by 11.9%, Gardenia's connected components by 5.8%, and ECL's graph coloring by 5.0%.**

***Index Terms*—Compressed in-memory representation, graph analytics, massive parallelism, GPU acceleration**

I. Introduction

Graph analytics algorithms such as connected components [1] and pattern mining [2] are widely used in various fields. When processing real-world inputs, many of these irregular codes are memory-bound and exhibit poor locality [3]. Hence, memory accesses are often the performance bottleneck and determine the overall execution time. Using a compressed in-memory representation can speed up the memory accesses at the cost of extra computation to decompress the data dynamically. However, achieving a low-enough decompression overhead in combination with a high-enough compression ratio to yield a net benefit is challenging, especially on GPUs with high-bandwidth memory.

Due to their high performance and energy efficiency, GPUs are now widely used as accelerators. However, they tend to have much smaller memories than CPUs. Operating directly on compressed data allows to fit more information into the memory but necessitates the need to decompress every element as it is being accessed. This idea has previously been demonstrated for CPUs but is much harder to realize on GPUs due to their high parallelism and memory bandwidth.

This paper demonstrates how to speed up graph algorithms running on GPUs by operating on a log-compressed graph [4] and decompressing it faster than directly operating on the uncompressed data. Note that we decompress the accessed graph elements on the fly while the graph is being processed and not ahead of time in a separate pre-processing step. Consequently, our approach requires no storage to hold the uncompressed graph. Using the compressed representation reduces the number of main-memory accesses and cache misses, which, combined with our massively-parallel low-latency decompression approach, often yields significant speedups. This is our primary goal, but our compressed representation also makes it possible to fit larger graphs on the GPU. LIGRA+ [5] similarly uses a compressed in-memory format that is decompressed dynamically. LIGRA+ only targets CPUs and is primarily designed to fit larger graphs in memory without loss of processing performance. It exhibits relatively low parallelism as it does not use parallelization within short and medium-sized adjacency lists, making it unsuitable for GPUs.

Compression can reduce the number of main-memory transactions because compressed graphs have a smaller footprint, exhibit more spatial locality, and allow a fixed-sized cache to hold a larger portion of the graph, leading to higher in-cache presence. However, these benefits tend to be less pronounced on GPUs, which have faster main memories and smaller caches than CPUs, making it more difficult to exploit compression. This paper makes the following contributions.

- It presents MPLG, the first approach that makes it possible for GPUs to run graph analytics programs directly on compressed graphs without loss of performance.
- It allows to fit 20% to 70% more data into memory and to process correspondingly larger inputs.
- It introduces a decompression technique that is so quick that it often results in faster operation (about 5% on average) than accessing the uncompressed data, thus making some of the fastest graph codes run even faster.
- It introduces 3 levels of massively-parallel, low-latency, high-throughput decompression of MPLG graphs.

Some of our MPLG-accelerated codes are publicly available at https://cs.txstate.edu/~burtscher/research/MPLG/.

II. Background

A. GPU Memory Architecture

GPUs employ various types of fast memory, including GDDR6, GDDR6X, and HBM2. The high bandwidth makes it difficult to perform real-time decompression because only very few computations can be executed before any benefit is lost. This may be why the related work has only focused on CPU implementations so far.

B. Warp-based Execution

GPUs are vector processors where a warp is the basic unit of execution. In CUDA, a warp is represented as a collection of 32 threads that either execute the same instruction or are disabled. We exploit warp-based execution to speed up our warp, block, and hybrid implementations of decompression.

C. Coalesced Memory Accesses

Due to warp-based execution, the threads of a warp perform loads and stores together. The GPU's memory interface is optimized for this scenario. In particular, the hardware combines the up to 32 memory accesses into groups such that each group only contains requests that go to the same naturally aligned block in main memory. Only one memory transaction is performed per group, which is called a coalesced access. In the best case, all 32 accesses of a warp fit into a single group, meaning just one coalesced transaction is needed to satisfy them. In the worst case, each thread's access will form a group by itself, resulting in 32 individual memory transactions. Compressing a graph's adjacency list will place more list elements into the same block, thus increasing the coalescing opportunity. This helps with thread-level operations on low-degree vertices as well as with warp- and block-level operations on medium- and high-degree vertices.

III. Related Work

Many papers describe optimizations to more efficiently process and store large graphs on GPUs. For example, the CuSha graph processing framework [6] speeds up graph algorithms by using sets of ordered edges, called shards, as the graph representation to boost coalescing. This is in contrast to the CSR format (cf. Section IV-A), which tends to suffer from uncoalesced memory accesses for warp-based algorithms. SIMD-x [7] is another GPU framework for accelerating data-intensive irregular algorithms. It utilizes system-level optimizations and just-in-time task management to achieve speedups over Gunrock [8] and Galois [9]. There have also been many papers on accelerating specific graph algorithms. For example, Merrill et al. [10] use fine-grained task management and parallel prefix sums as an alternative to atomics to speed up breadth-first search (BFS) on large sparse graphs. Alabandi et al. [11] propose shortcut computations to increase the parallelism and the performance of graph coloring (GC). These and many other optimizations are orthogonal to our approach, meaning MPLG can be employed on top of them to boost the performance.

Most prior work on graph compression aims at maximally compressing graphs without considering the decompression speed or the performance of the algorithms that process them [12]–[15].

LIGRA+ [5], an upgrade to LIGRA [16] (lightweight graph processing framework for shared memory) trades off a smaller memory footprint for extra overhead to fit larger graphs on the CPU while maintaining and sometimes improving performance of graph algorithms. We compare our approach to LIGRA+ in Section VI-D.

LIGRA+ first sorts and delta encodes the adjacency list of each vertex. It then compresses the lists using variable-length codes. Rather than encoding each value separately, it groups consecutive values that require the same number of bytes (or nibbles) together and precedes each group with a one-byte header indicating both the number of bytes (or nibbles) each value requires and the number of values in the group. These header bytes lower the compression ratio but simplify and speed up the decoding.

LIGRA+'s decoding is primarily parallelized over the vertices, meaning each adjacency list is processed by a single thread. Only high-degree vertices are decoded in parallel. To avoid the costly prefix-sum computation that would be needed to perform the delta decoding in parallel, LIGRA+ splits long adjacency lists into independent 1000-value chunks. This works well for CPUs with dozens of cores but does not yield enough parallelism for GPUs running 100,000s of threads, which is why we adopted a different approach.

Due to its encoding scheme, LIGRA+ needs to separately store each vertex's degree, which increases the memory footprint. Like the widely-used CSR graph format, our approach requires no such extra storage as the degrees can trivially be derived from the starting locations of the adjacency lists. Moreover, our approach does not need the adjacency lists to be sorted and does not employ delta encoding. This means that there is no need for a prefix sum in the MPLG decoder as opposed to LIGRA+, making random access to the decompressed data possible in our approach. Finally, LIGRA+ is specifically designed for LIGRA whereas our approach is designed for easy integration into third-party codes.

The recent Log(Graph) work [4] describes a compressed graph representation that uses logarithmic lower bounds to compress graph information (by eliminating a fixed number of leading zero bits from all values) while decreasing the overhead of decompression to a point where performance is comparable to non-compressed algorithms. This work also only targets CPUs. We cannot compare to Log(Graph) as the code is not public. However, we adopt and extend its log-encoding idea to boost the graph-processing performance on GPUs. In particular, we designed the MPLG format to enable massively parallel decoding with minimal overhead and implemented three such decoders that operate at thread, warp, and block granularity as well as hybrids thereof.

IV. Approach

This section describes our approach in detail, which combines the CSR graph representation with log-based encoding. We call our approach Massively Parallel Log Graphs (MPLG).

A. Compressed Spare Row Format

The Compressed Sparse Row (CSR) format is probably the most widely used representation for large sparse graphs [17]. It includes the word "compressed" because it does not store the zero elements of the corresponding adjacency matrix. Instead, CSR is based on two dense arrays: an array of indices and an array of edges. The edge array holds the concatenated adjacency lists of all vertices. The index array holds the starting position (index) of each adjacency list. It has an extra element at the end specifying the size of the edge array. Figure 1 shows an example graph and its CSR representation. Note that Vertex 1 has no outgoing edges.

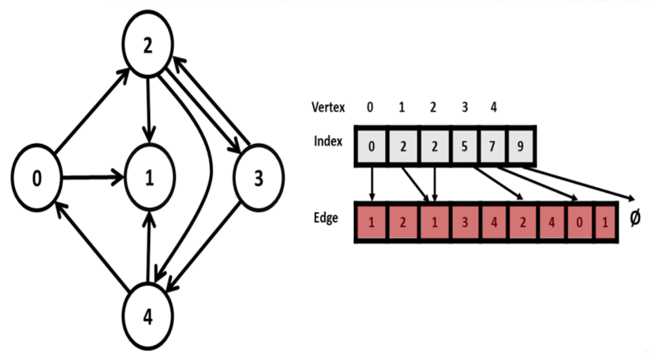


Fig. 1. Example graph (left) and corresponding CSR representation (right)

Our MPLG approach employs a compressed form of CSR as its in-memory graph representation. To achieve a sufficiently low decompression overhead to obtain a net performance benefit, we augmented log encoding in the following three ways. First, we designed a GPU-friendly massively parallel decompression technique for it. Second, we implemented this technique at multiple levels of granularity to match the thread, warp, block, and hybrid implementations of existing graph codes. Third, we only compress the edge array, which is highlighted in Figure 1, as it is typically the larger of the two arrays and accessed more often.

B. Log Encoding

The MPLG format is a compressed representation of graphs that is based on log encoding of the CSR format.

Log encoding reduces the storage requirement of each value in the edge array by eliminating leading zero bits [4]. The largest value in the edge array of many graphs is much smaller than the maximum value that can be expressed by a word. In this case, all stored values start with some number of leading zeros that provide no useful information but take up space.

MPLG eliminates these unnecessary zero bits from each value in the edge array, concatenates the remaining bits, and places them into a shorter array. To determine the number of zero bits that can be eliminated, we need to know the largest value in the array. Since each array element refers to a vertex ID, the largest value cannot be greater than the number of vertices in the graph. Figure 2 illustrates MPLG's encoding scheme on an array of three integers. Integer c is the largest of the three values and has 18 leading zeros, so we eliminate the 18 zeros highlighted in red from all three integers and place the remaining 14 bits into the encoded array, one after another. The resulting encoded array requires only two integers or about 67% of the size of the original array.

We experimented with many other encoding schemes that yield higher compression ratios, such as using blocks of 8-

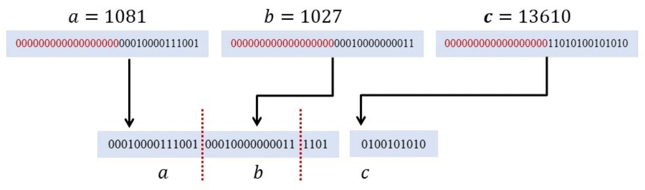


Fig. 2. Log encoding example

and 16-bit values with continuation bits to store variable-length values in an array of integers. We also combined these approaches with delta coding to reduce the number of bits per value. However, none of these approaches yielded a speedup because the decompressors ended up being too slow. In contrast, log encoding does not produce a particularly high compression ratio but makes decompression of each element of the edge list independent and fast. We tried increasing the compression ratio by compressing not only the adjacency lists but also the index list, but doing so resulted in an overall decrease in performance. The simple log encoding of the adjacency lists enables massively-parallel decompression because each thread can independently extract any edge, eliminating the need for synchronization. MPLG's log-encoding technique also makes massively-parallel compression possible (with the use of atomic operations). However, graphs are typically read many more times than they are written, which is why we focus on decompression in this paper.

C. Log Decoding

To maximize the performance of graph algorithms through the use of compression, we must minimize the overhead of the on-the-fly decompression, which is invoked every time an adjacency-list element is read. As mentioned, MPLG achieves this by using an encoding scheme that can be decoded both quickly and independently.

Log decoding is quite fast as it only entails reading two adjacent integers, concatenating their bits, shifting them, and masking out extraneous bits. These operations are mostly single machine instructions. In fact, CUDA supports funnel shifts that concatenate the bits of two integers and shift them in a single operation.

Massively parallel log decoding is possible because there are no dependencies between the decoding operations. After all, a thread that needs the k^{th} edge can simply extract the bits in the range $[k \times b, (k+1) \times b - 1]$ from the compressed edge array, where b is the number of bits used to represent each value. This bit extraction is independent of whatever edges other threads may be extracting, thus enabling random accesses.

GPU hardware exposes three levels of parallelism, namely thread, warp, and block parallelism, and many high performance codes explicitly take advantage of them. Hence, we implemented MPLG decompression at all three levels as well as hybrids thereof. Listings 1, 3, and 5 show thread-, warp-, and block-level implementations of a generic graph traversal algorithm, and Listings 2, 4, and 6 show the MPLG counterparts. The adjacency list of a vertex is accessed and processed by either a thread, a warp, or a block. The $ProcessNeighbor$ function in each listing refers to the algorithm-specific code that is independent of whether compression is used or not. The highlighted sections in the MPLG versions mark the extra code needed for the dynamic decompression. $logn$ denotes the number of bits used to represent each value. Since 32 is a power of two, the division and modulus operations are compiled into fast shift and bitwise AND instructions. Moreover, the expression $(1 << logn) - 1$ is loop invariant and only computed once before the loop starts. As a result, the highlighted statements in the loop bodies map to just a few fast machine instructions.

```
tid = threadIdx.x + blockIdx.x * blockDim.x;
if (tid < numv) {
  beg = nindex[tid];
  end = nindex[tid + 1];
  for (j = beg; j < end; j++) {
    nei = nlist[j];
    ProcessNeighbor(nei);
  }
}
```

Listing 1. thread-level graph processing

```
tid = threadIdx.x + blockIdx.x * blockDim.x;
if (tid < numv) {
  beg = nindex[tid];
  end = nindex[tid + 1];
  cur = beg * logn;
  shift = cur % 32;
  for (j = beg; j < end; j++) {
    pos = cur / 32;
    res = __funnelshift_rc(encoded[pos], encoded[pos
    + 1], shift);
    cur += logn;
    shift = (shift + logn) % 32;
    nei = res & ((1 << logn) - 1);
    ProcessNeighbor(nei);
  }
}
```

Listing 2. MPLG-based thread-level graph processing

```
wid = (threadIdx.x + blockIdx.x * blockDim.x) /
    WarpSize;
if (wid < numv) {
  beg = nindex[wid];
  end = nindex[wid + 1];
  lane = threadIdx.x % WarpSize;
  for (j = beg + lane; j < end; j += WarpSize) {
    nei = nlist[j];
    ProcessNeighbor(nei);
  }
}
```

Listing 3. warp-level graph processing

We inserted these MPLG decoding operations into the CUDA implementations of various graph algorithms from three different suites. In high-performance GPU graph codes that process the edge lists of low-degree vertices with individual threads but the edge lists of high-degree vertices with entire warps [11] or even thread blocks [18], we inserted multiple versions of MPLG.

```
wid = (threadIdx.x + blockIdx.x * blockDim.x) /
    WarpSize;
if (wid < numv) {
  beg = nindex[wid];
  end = nindex[wid + 1];
  lane = threadIdx.x % WarpSize;
  wsln = WarpSize * logn;
  lln = lane * logn;
  cur = beg * logn;
  curlo = cur % 32;
  curhi = cur / 32;
  shift = (curlo + lln) % 32;
  pos = curhi + (curlo + lln) / 32;
  for (j = beg + lane; j < end; j += WarpSize) {
    res = __funnelshift_rc(encoded[pos], encoded[pos
    + 1], shift);
    pos += logn;
    shift = (shift + wsln) % 32;
    nei = res & ((1 << logn) - 1);
    ProcessNeighbor(nei);
  }
}
```

Listing 4. MPLG-based warp-level graph processing

```
bid = blockIdx.x;
if (bid < numv) {
  beg = nindex[bid];
  end = nindex[bid + 1];
  for (j = beg + threadIdx.x; j < end; j += blockDim
    .x) {
    nei = nlist[j];
    ProcessNeighbor(nei);
  }
}
```

Listing 5. block-level graph processing

```
bid = blockIdx.x;
if (bid < numv) {
  beg = nindex[bid];
  end = nindex[bid + 1];
  bsln = blockDim.x * logn;
  tln = threadIdx.x * logn;
  cur = beg * logn;
  curlo = cur % 32;
  curhi = cur / 32;
  shift = (curlo + tln) % 32;
  pos = curhi + (curlo + tln) / 32;
  for (j = beg + threadIdx.x; j < end; j += blockDim
    .x) {
    res = __funnelshift_rc(encoded[pos], encoded[pos
    + 1], shift);
    pos += logn * WarpsPerBlock;
    shift = (shift + bsln) % 32;
    nei = res & ((1 << logn) - 1);
    ProcessNeighbor(nei);
  }
}
```

Listing 6. MPLG-based block-level graph processing

V. Methodology

We present performance results from two systems. The first system is based on a Ryzen Threadripper 2950X CPU with 16 hyperthreading cores and a TITAN V GPU. This GPU has 5120 processing elements distributed over 80 multiprocessors (SMs). Each SM has 96 kB of L1 data cache. The 80 SMs share a 4.5 MB L2 cache as well as 12 GB of global memory with a peak bandwidth of 652 GB/s and a clock frequency of 1455 MHz. The second system is based on a Xeon Gold 6226R

CPU with 32 hyperthreading cores and an RTX 3090 GPU. This GPU has 10,496 processing elements distributed over 82 SMs. Each SM has 128 kB of L1 data cache. The 82 SMs share a 6 MB L2 cache as well as 24 GB of global memory with a peak bandwidth of 936 GB/s and a clock frequency of 1395 Mhz. These differences between the two systems may cause different timing behaviors, which is why we test our programs on more than one system.

We compiled the CPU codes using g++ version 11.2.1 with the $-O3$ flag. For the GPU codes, we used nvcc version 11.4 with the $-arch = sm\_70$ flag for the TITAN V and the $-arch = sm\_86$ flag for the RTX 3090. We ran each experiment 9 times on each input and use the median runtime for the results shown in this paper.

A. Codes

We added MPLG to the following 8 GPU codes from the Rodinia [19] and Gardenia [20] benchmark suites and from ECL [11], [21]. Some of the algorithms are represented multiple times but implemented differently in each case.

- **BFS**: Breadth First Search is a graph traversal algorithm that computes the distance of each vertex (in number of edges) from a given source vertex. We selected the thread-level BFS implementation from Rodinia [22], in which each adjacency list is processed by a separate thread.
- **CC**: Connected Components is an algorithm that computes maximal subgraphs of an undirected graph such that there is a path between any pair of vertices within a subgraph but there is no path between any pair of vertices from different subgraphs. We selected three CC codes, a thread- and a warp-level implementation from Gardenia [20] and a hybrid implementation from ECL [18].
- **GC**: Graph Coloring assigns a color to each vertex such that no two adjacent vertices have the same color. The goal is to do this with as few colors as possible. As this problem is NP-hard, heuristics are employed in practice that do not necessarily result in the smallest number of colors. We selected the hybrid GC code from ECL [11].
- **MIS**: Maximal Independent Set is an algorithm that computes a subset of the vertices such that none of the vertices from the set are adjacent (independence) and no other vertex from the graph can be added to the set without violating the independence (maximality). We selected the thread-level MIS code from ECL [21].
- **MST**: Minimum Spanning Tree is an algorithm that computes a minimum-weight subset of the edges that connects all vertices of the graph. We selected the thread-level MST code from Gardenia [20].
- **SSSP**: Single Source Shortest Paths is an algorithm that computes the distance (the sum of the edge weights) of the shortest path from a given source vertex to all other vertices. We selected the thread-level SSSP code from Gardenia [20].

Table I shows the granularity at which we applied MPLG to each of these codes.

TABLE I
CODES AUGMENTED WITH MPLG DECOMPRESSION (E = ECL, G = GARDENIA, R = RODINIA)

Algorithm	Thread-level			Warp-level			Hybrid		
	E	G	R	E	G	R	E	G	R
BFS			✓						
CC		✓			✓		✓		
GC							✓		
MIS							✓		
MST		✓							
SSSP		✓							

B. Input Graphs

We used the 15 graphs listed in Table II as inputs. They have between about two million and half a billion edges and represent various types, including social-network, road-map, random, Internet, and RMAT graphs. They were obtained from the Center for Discrete Mathematics and Theoretical Computer Science at the University of Rome (Dimacs) [23], the Galois framework (Galois) [9], the Stanford Network Analysis Platform (SNAP) [24], and the SuiteSparse Matrix Collection (SSMC) [25]. Where necessary, we made the graphs undirected, removed self-edges, and added weights. Each undirected edge is represented by two directed edges. In all cases that use MPLG, the edge array is log encoded. The resulting compression ratios (CR) are shown in the rightmost column of Table II, that is, the MPLG size divided by the original CSR graph size (lower percentages are better).

TABLE II
TESTED GRAPHS AND MPLG COMPRESSION RATIOS

Graph Name	Edges	Vertices	Type	Source	CR
2d-2e20.sym	4,190,208	1,048,576	grid	Galois	62.5%
amazon0601	4,886,816	403,394	product co-purchases	SNAP	59.3%
as-skitter	22,190,596	1,696,415	Internet topology	SNAP	65.6%
cit-Patents	33,037,894	3,774,768	patent citations	SSMC	68.7%
citationCiteseer	2,313,294	268,495	publication citations	SSMC	59.3%
coPapersDBLP	30,491,458	540,486	publication citations	SSMC	62.5%
delaunay_n24	100,663,202	16,777,216	triangulation	SSMC	78.1%
europe_osm	108,109,320	50,912,018	road map	SSMC	81.2%
in-2004	27,182,946	1,382,908	web links	SSMC	65.6%
kron_g500-logn21	182,081,864	2,097,152	kronecker	SSMC	65.6%
r4-2e23.sym	67,108,846	8,388,608	random	Galois	71.8%
rmat22.sym	65,660,814	4,194,304	RMAT	Galois	71.8%
soc-LiveJournal1	85,702,474	4,847,571	RMAT	SNAP	71.8%
uk-2002	523,574,516	18,520,486	web links	SSMC	78.1%
USA-road-d.USA	57,708,624	23,947,347	road map	Dimacs	78.1%

VI. RESULTS

In this section, we first present the performance benefit and the reduction in cache misses due to MPLG on the eight GPU codes. Then, we compare the compression ratio of MPLG to that of LIGRA+. Finally, we show the performance of LIGRA+ on its own set of CPU graph codes for reference.

A. MPLG Performance

Tables III and V list the median running times in milliseconds for each baseline code and input pair. Tables IV and VI show the speedups achieved by adding MPLG. The bold-printed values highlight the cases where MPLG yields a performance improvement. Tables III and IV are for the newer

TABLE III
RTX 3090 RUNTIMES (MS) OF ORIGINAL CODES

Input graph	*ecl-CC*	*ecl-GC*	*ecl-MIS*	*gr-CC-t*	*gr-CC-w*	*gr-MST*	*gr-SSSP*	*rd-BFS*
2d-2e20.sym	0.41	0.30	0.17	0.11	0.26	6.30	29.08	23.15
amazon0601	0.29	0.59	0.26	0.75	0.15	5.95	5.64	23.11
as-skitter	0.96	8.30	2.63	8.72	1.24	48.28	25.98	23.13
citationCiteseer	0.22	0.29	0.09	0.33	0.09	2.66	2.75	22.91
cit-Patents	4.16	7.05	1.84	2.79	2.51	74.54	16.12	47.90
coPapersDBLP	0.58	22.74	1.12	1.49	0.34	25.62	9.14	23.26
delaunay_n24	17.80	8.33	12.50	1.18	3.72	100.37	50.82	138.86
europe_osm	19.86	10.22	23.27	2.01	11.34	174.95	318.22	393.41
in-2004	0.73	23.13	1.15	4.25	0.78	31.14	50.35	22.99
kron_g500-logn21	3.86	89.88	2.15	63.96	8.61	404.30	158.42	31.13
r4-2e23.sym	9.30	18.84	8.85	6.85	6.75	146.01	37.11	81.30
rmat22.sym	5.32	41.95	3.52	6.45	6.76	1.16	29.53	48.82
Soc-LiveJournal1	4.98	29.76	5.92	8.37	4.72	141.12	40.33	56.56
uk-2002	18.86	156.35	37.54	34.66	14.00	148.34	132.20	157.23
USA-road-d.USA	15.63	5.18	11.10	0.99	5.08	473.59	113.61	194.91

TABLE IV
RTX 3090 SPEEDUPS DUE TO MPLG

Input graph	*ecl-CC*	*ecl-GC*	*ecl-MIS*	*gr-CC-t*	*gr-CC-w*	*gr-MST*	*gr-SSSP*	*rd-BFS*
2d-2e20.sym	**1.15**	**1.16**	0.89	**1.19**	0.94	**1.04**	**1.01**	**1.22**
amazon0601	**1.13**	**1.03**	0.93	**1.01**	0.99	0.91	0.97	**1.20**
as-skitter	**1.14**	0.99	**1.01**	**1.02**	**1.01**	**1.05**	0.93	**1.22**
citationCiteseer	**1.02**	0.96	0.83	0.98	0.92	0.87	**1.00**	**1.18**
cit-Patents	**1.07**	**1.11**	**1.10**	**1.18**	**1.02**	**1.08**	**1.01**	**1.06**
coPapersDBLP	**1.02**	**1.02**	**1.05**	**1.32**	**1.09**	**1.02**	0.98	**1.22**
delaunay_n24	**1.04**	**1.12**	**1.15**	**1.14**	0.88	0.98	**1.04**	**1.02**
europe_osm	**1.03**	**1.05**	0.87	**1.03**	0.89	0.97	**1.01**	**1.01**
in-2004	**1.09**	**1.02**	**1.00**	0.96	0.99	**1.01**	**1.00**	**1.21**
kron_g500-logn21	**1.06**	**1.03**	**1.02**	0.98	**1.00**	**1.01**	**1.01**	**1.15**
r4-2e23.sym	**1.09**	**1.06**	**1.10**	**1.10**	**1.03**	0.92	**1.03**	**1.04**
rmat22.sym	**1.06**	**1.03**	**1.04**	**1.05**	**1.03**	**1.00**	**1.01**	**1.10**
Soc-LiveJournal1	**1.07**	**1.02**	**1.01**	**1.10**	**1.03**	0.84	**1.04**	**1.08**
uk-2002	**1.06**	**1.03**	**1.01**	0.98	**1.01**	**1.08**	0.95	**1.03**
USA-road-d.USA	**1.03**	**1.11**	**1.22**	**1.03**	0.88	0.98	0.97	**1.01**
GEOMEAN	**1.07**	**1.05**	**1.01**	**1.07**	0.98	0.98	**1.00**	**1.11**

Ampere-based RTX 3090 GPU and Tables V and VI for the Volta-based Titan V GPU.

To better visualize and analyze the results, Figure 3 presents the performance of the GPU codes (along the x axis) with MPLG over the performance of the original codes that do not use compression. Ratios (along the y axis) above 1.0 mean that MPLG provides a speedup. Each box and whisker plot summarizes the 15 ratios (one per input graph). The bottom whisker reflects the lowest, the bottom of the blue box the first quartile, the transition from blue to orange the median, the top of the orange box the third quartile, and the top whisker the highest of the 15 ratios.

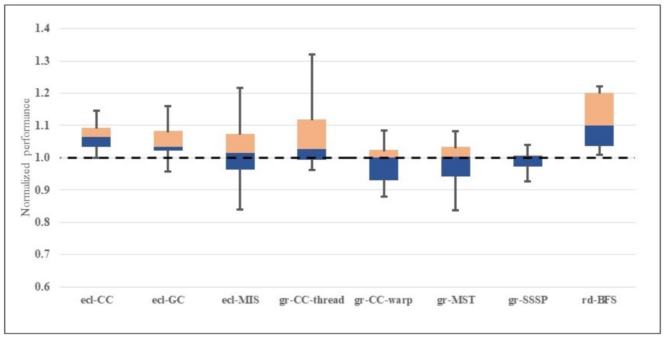
The general trend is the same for both GPUs, so we discuss them together unless otherwise noted. For each code, we see a range of behaviors. This is typical for irregular codes, which tend to be highly input dependent [3]. Except for Gardenia's gr-CC-warp code on the RTX 3090 and ECL's ecl-MIS code on the Titan V, the median is above 1.0, meaning that using

TABLE V
TITAN V RUNTIMES (MS) OF ORIGINAL CODES

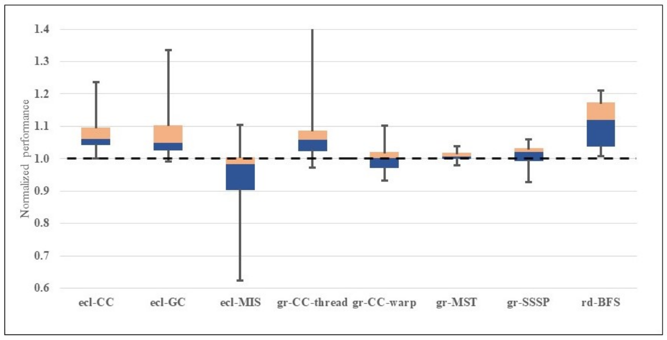
Input graph	*ecl-CC*	*ecl-GC*	*ecl-MIS*	*gr-CC-t*	*gr-CC-w*	*gr-MST*	*gr-SSSP*	*rd-BFS*
2d-2e20.sym	0.642	0.713	0.243	0.230	0.327	8.699	34.467	25.232
amazon0601	0.588	1.109	0.342	1.004	0.186	8.306	7.328	25.470
as-skitter	1.881	12.342	4.136	10.088	1.658	64.679	35.402	34.293
citationCiteseer	0.319	0.409	0.124	0.434	0.118	3.707	3.479	25.768
cit-Patents	7.435	13.959	3.244	4.401	3.858	103.315	28.968	46.399
coPapersDBLP	0.809	37.071	2.096	2.968	0.513	37.924	15.584	25.489
delaunay_n24	28.795	12.667	16.362	1.842	4.332	144.593	70.145	146.114
europe_osm	29.662	15.655	17.325	3.084	15.566	244.202	383.849	401.615
in-2004	1.162	32.408	1.968	4.776	0.926	43.056	60.722	25.325
kron_g500-logn21	7.068	140.984	4.071	75.637	13.905	513.938	222.861	34.620
r4-2e23.sym	20.305	33.590	13.800	11.277	10.989	214.770	67.209	86.059
rmat22.sym	10.418	87.065	5.872	10.751	11.189	201.722	54.581	54.949
Soc-LiveJournal1	9.823	57.791	10.763	12.711	7.528	209.045	65.571	56.246
uk-2002	36.158	243.466	70.036	42.036	20.712	693.578	218.983	160.472
USA-road-d.USA	25.021	8.615	9.839	1.364	5.766	121.591	154.689	201.353

TABLE VI
TITAN V SPEEDUPS DUE TO MPLG

Input graph	*ecl-CC*	*ecl-GC*	*ecl-MIS*	*gr-CC-t*	*gr-CC-w*	*gr-MST*	*gr-SSSP*	*rd-BFS*
amazon0601.egr	**1.24**	**1.34**	0.96	**1.08**	0.97	**1.04**	**1.03**	**1.18**
as-skitter.egr	**1.16**	0.99	0.96	**1.07**	**1.01**	**1.03**	0.99	**1.12**
citationCiteseer.egr	**1.04**	**1.05**	0.80	**1.02**	**1.00**	**1.02**	0.99	**1.21**
cit-Patents.egr	**1.06**	**1.13**	0.98	**1.16**	**1.03**	**1.00**	**1.05**	**1.09**
coPapersDBLP.egr	**1.04**	**1.03**	**1.11**	**1.14**	**1.10**	**1.01**	**1.02**	**1.18**
delaunay_n24.egr	**1.04**	**1.13**	0.90	**1.06**	0.94	**1.01**	**1.03**	**1.03**
europe_osm.egr	**1.03**	**1.05**	0.80	**1.04**	0.95	**1.00**	**1.00**	**1.01**
in-2004.egr	**1.12**	**1.05**	**1.10**	0.99	**1.00**	0.98	0.94	**1.18**
kron_g500-logn21.egr	**1.07**	**1.02**	**1.06**	0.97	**1.04**	**1.00**	**1.02**	**1.13**
r4-2e23.sym.egr	**1.06**	**1.08**	0.99	**1.10**	**1.00**	**1.01**	**1.02**	**1.05**
rmat22.sym.egr	**1.06**	**1.01**	0.98	**1.06**	**1.00**	**1.01**	**1.04**	**1.13**
soc-LiveJournal1.egr	**1.06**	**1.03**	0.99	**1.05**	**1.03**	**1.02**	**1.06**	**1.06**
uk-2002.egr	**1.05**	**1.01**	**1.02**	0.97	**1.00**	**1.00**	0.93	**1.02**
USA-road-d.USA.egr	**1.04**	**1.06**	0.62	**1.03**	0.93	**1.00**	**1.01**	**1.02**
GEOMEAN	**1.08**	**1.07**	0.94	**1.05**	**1.00**	**1.01**	**1.01**	**1.10**



(a) RTX 3090 GPU



(b) Titan V GPU

Fig. 3. Performance with MPLG normalized to performance w/o compression

MPLG results in better performance on the majority of the input graphs for 7 of the 8 tested codes. On about half of the codes, over three quarters of the inputs derive a benefit from MPLG. On ecl-CC and Rodinia's rd-BFS codes, MPLG yields a speedup on all tested inputs. On Gardenia's thread-level CC code, MPLG yields speedups of up to 32% on the RTX 3090 and up to 67% on the Titan V. The four best median speedups on the RTX 3090 are 6.4% for ecl-CC, 3.3% for ecl-GC, 2.7% for gr-CC-thread, and 10.0% for rd-BFS. On the Titan V, the four best median speedups are 6.1% for ecl-CC, 5.0% for ecl-GC, 5.8% for gr-CC-thread, and 11.9% for rd-BFS.

B. Cache Misses

Since the performance of MPLG is a tradeoff between faster memory accesses and decompression overhead, this subsection separately investigates just the memory behavior. In particular, we used *nvprof* to count the L2 cache misses (i.e., the main memory accesses) on the Titan V.

Figure 4 presents the number of L2 misses in the MPLG-enhanced codes divided by the number of L2 misses in the original codes that do not use compression. Note that this is a lower-is-better metric. A value below 1.0 indicates that there is a decrease in cache misses, which should improve performance on memory-bound codes. We employ the same type of box and whiskers plots as before to show the minimum, first quartile, median, third quartile, and maximum of the normalized L2-cache misses of the 8 GPU codes on the 15 inputs.

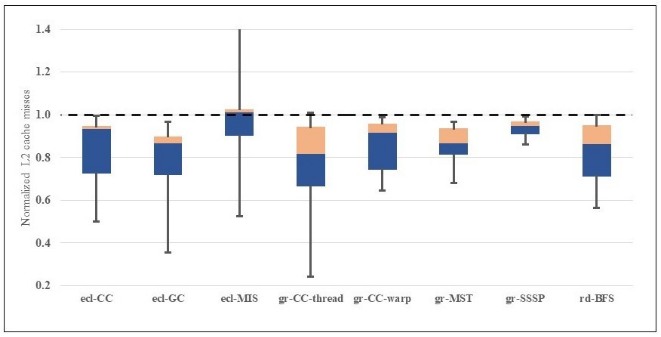


Fig. 4. L2 cache misses with MPLG normalized to L2 misses without compression on the Titan V

Except for about half of the inputs on ecl-MIS, we observe a reduction in the number of cache misses on essentially all inputs and codes. This is, of course, the goal of the compression and expected as MPLG yields a significant reduction of the memory footprint for all inputs (cf. Table II).

Comparing the cache results from Figure 4 to the performance results from Figure 3, we find a clear correlation between the performance increase and the cache miss decrease. For example, ecl-MIS exhibits no performance improvement on average because it yields no reduction in the number of cache misses on average. In contrast, the code with the highest performance improvement (rd-BFS) yields one of the highest cache-miss reductions. Since the decompression overhead hurts performance, the cache miss reduction is generally higher than the resulting increase in performance. For instance, MPLG is able to decrease the number of cache misses on all 15 inputs for Gardenia's warp-level CC code, but this only yields a performance gain on half the inputs (cf. Figure 3). On gr-CC-thread, MPLG reduces the L2 cache misses by over a factor of 4 on one input. The four best median reductions are 15.5% on ecl-GC, 22.6% on gr-CC-thread, 15.2% on gr-MST, and 15.9% on rd-BFS.

C. Decompression Throughput

Table VII shows the decompression throughput in number of edges processed per second by MPLG on the Titan V. On Gardenia's warp-level CC code, MPLG decompresses up to 65.6 billion edges per second. This amounts to 262 gigabytes of decompressed data generated per second, which exceeds the main-memory bandwidth of many high-end CPUs. On ECL's GC code, MPLG achieves a geomean decompression throughput of 15 billion edges per second (i.e., over 60 GB/s).

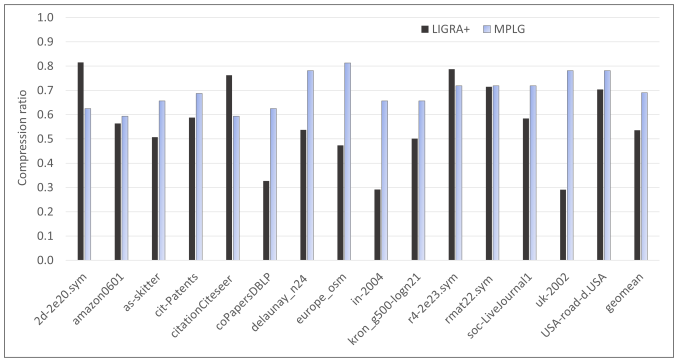


Fig. 5. Compression ratios of LIGRA+ and MPLG

TABLE VII
DECOMPRESSED EDGES (IN MILLIONS) PER SECOND ON THE TITAN V

Input graph	*ecl-CC*	*ecl-GC*	*ecl-MIS*	*gr-CC-thr*	*gr-CC-wrp*	*gr-MST*	*gr-SSSP*	*rd-BFS*
2d-2e20.sym	9,227.1	13,328.0	18,306.2	30,600.0	12,493.5	109.5	120.3	1,966.3
amazon0601	11,116.5	12,961.3	10,911.6	5,249.4	25,642.3	77.0	689.3	2,295.4
as-skitter	14,752.3	14,045.4	4,452.9	2,362.3	13,579.0	39.9	622.2	7,315.0
citationCiteseer	8,262.6	12,997.8	11,644.4	5,467.2	19,691.3	96.5	660.8	1,098.4
cit-Patents	5,029.3	6,417.9	6,118.1	8,748.0	8,835.5	49.7	1,200.9	7,837.7
coPapersDBLP	40,048.1	54,777.9	9,502.7	11,806.9	65,650.4	110.3	1,998.1	14,306.0
delaunay_n24	4,207.7	18,458.6	3,719.2	58,571.7	21,987.0	142.5	1,476.6	7,139.8
europe_osm	5,545.3	12,314.0	6,698.3	36,658.9	6,644.9	89.2	282.0	2,742.2
in-2004	27,562.8	27,463.8	11,364.4	5,701.4	29,876.8	100.3	404.8	12,305.1
kron_g500-logn21	27,712.9	33,280.2	6,060.6	2,352.1	13,620.4	42.9	835.7	59,792.0
r4-2e23.sym	3,933.0	4,903.6	3,341.5	6,548.7	6,138.8	49.1	1,021.0	8,246.5
rmat22.sym	7,016.8	6,498.2	3,557.6	6,480.4	5,880.8	38.2	1,250.8	13,644.3
soc-LiveJournal1	9,807.3	18,240.3	3,760.1	7,093.9	11,740.4	56.9	1,384.9	16,383.8
uk-2002	15,666.0	27,612.5	4,972.5	12,212.0	25,962.7	130.8	2,215.8	33,620.6
USA-road-d.USA	3,319.7	12,236.2	4,269.3	43,775.6	9,395.4	114.5	378.1	2,965.3
GEOMEAN	**9,698.5**	**15,032.5**	**6,301.3**	**9,995.3**	**14,644.0**	**75.8**	**762.3**	**7,392.6**

These throughputs highlight the difficulty of achieving a benefit from on-the-fly decompression and why it only works with a high-speed and massively-parallel approach like MPLG. As mentioned, we have implemented dozens of progressively simpler algorithms and faster decompressors until we finally found a combination that was fast enough to yield a speedup.

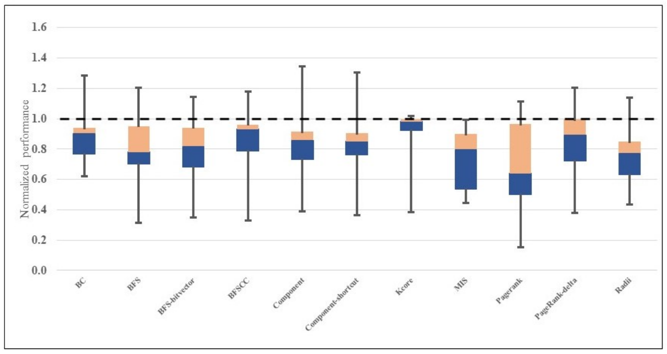
D. Comparison to LIGRA+

In this subsection, we compare MPLG to LIGRA+, which uses a different compression approach that is tailored to CPUs.

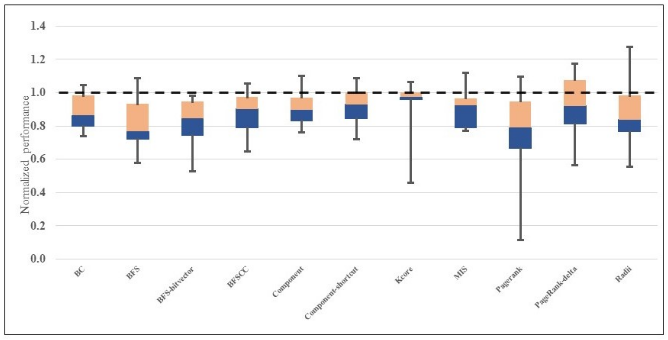
Compression quality We compressed our 15 input graphs using MPLG and LIGRA+'s highest-quality technique. Note that LIGRA+ compresses both the edge and weight arrays (on weighted graphs) whereas MPLG only compresses the edge array. Figure 5 shows the resulting compression ratios.

LIGRA+ achieves better compression on all but 3 of our inputs, which is expected since MPLG's log-based encoding is simple by design to enable fast and massively-parallel decoding. Based on the geometric mean, LIGRA+ shrinks our graphs down to 54% and MPLG shrinks them down to 69% of their original size.

Performance To see whether LIGRA+'s higher compression ratio translates into better performance, we converted the 15 graphs into the LIGRA and LIGRA+ formats and used them as inputs for the codes that are provided with LIGRA/LIGRA+. Figure 6 shows the box and whisker plots of the resulting performance ratios on two types of CPUs.



(a) LIGRA+ on a 64-thread 6226R CPU



(b) LIGRA+ on a 32-thread 2950X CPU

Fig. 6. Performance of LIGRA+ normalized to LIGRA

On all tested codes and both CPUs, LIGRA+ results in a slowdown on the majority of our inputs and on average. Its highest median speedup is 0.98. In contrast, MPLG's highest median speedup is 1.12. As before, we observe a large variation in performance ratios across the 15 inputs due to the data-dependent nature of these irregular programs. LIGRA+ delivers an up to 35% performance boost but also an up to 89% performance drop compared to MPLG's maximum 67% boost and 38% drop. On average, MPLG yields substantially more performance than LIGRA+. This is perhaps unexpected since MPLG runs on GPUs, which have a much higher memory bandwidth, making it more difficult to perform the decompression fast enough to obtain a speedup. However, LIGRA+'s main goal is to reduce the in-memory graph sizes without loss in performance whereas MPLG's main goal is to boost performance while reducing the graph sizes.

VII. Summary and Conclusion

We present Massively Parallel Log Graphs (MPLG), a losslessly compressed data representation to increase the performance of graph analytics codes running on GPUs. MPLG trades off a smaller in-memory footprint for extra decompression overhead when reading the graph's adjacency list entries. This accelerates the memory accesses due to increased coalescing, improved spatial locality, and fewer overall main-memory transactions. To maximally exploit this benefit, MPLG employs a log-based encoding scheme that enables low-latency and independent decompression of each adjacency list element, making massively-parallel GPU operation possible. We had to experiment with many different compression algorithms and decompressor implementations until we found a combination that was able to deliver a speedup and operate at the required tens to hundreds of gigabytes per second.

The LIGRA+ and Log(Graph) projects use a similar idea but target CPUs. LIGRA+'s approach is not parallel enough for GPU usage and has too much overhead for the GPU's high-bandwidth memory, which is why we employ an approach that is closer to Log(Graph) in nature. Since the Log(Graph) code is not publicly available, we cannot compare to it. On average, LIGRA+ compresses graphs more but delivers less speedup than MPLG. Moreover, LIGRA+ is an extension of and tied to LIGRA whereas MPLG is independent and designed to be included in third-party codes. While our primary goal is to improve performance, MPLG can also be used to fit larger graphs on a GPU than would otherwise be possible (like LIGRA). For example, our UK graph does not fit on a GPU with 2 GB of memory but the MPLG version does. MPLG shrinks our 15 graphs from various domains down to between 59% and 81% of their original size. Using these inputs on 8 GPU codes that we augmented with MPLG yields significant speedups on average and up to 67% faster execution in the best observed case. Our work demonstrates that real-time decompression is, in many cases, feasible and beneficial even on massively-parallel GPUs with high-bandwidth memory running irregular graph codes. We hope that our work will inspire others to develop real-time decompression approaches for additional data structures and domains.

VIII. Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant No. 1955367 and by an equipment donation from NVIDIA Corporation. We thank the reviewers for their feedback.

References

[1] L. Di Stefano and A. Bulgarelli, "A simple and efficient connected components labeling algorithm," in *Proceedings 10th international conference on image analysis and processing*. IEEE, 1999, pp. 322–327.

[2] V. Dias, C. H. Teixeira, D. Guedes, W. Meira, and S. Parthasarathy, "Fractal: A general-purpose graph pattern mining system," in *Proceedings of the 2019 International Conference on Management of Data*, 2019, pp. 1357–1374.

[3] M. Burtscher, R. Nasre, and K. Pingali, "A quantitative study of irregular programs on gpus," in *2012 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 2012, pp. 141–151.

[4] 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.

[5] 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.

[6] F. Khorasani, K. Vora, R. Gupta, and L. N. Bhuyan, “Cusha: vertex-centric graph processing on gpus,” in *Proceedings of the 23rd international symposium on High-performance parallel and distributed computing*, 2014, pp. 239–252.

[7] H. Liu and H. H. Huang, “{SIMD-X}: Programming and processing of graph algorithms on {GPUs},” in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, 2019, pp. 411–428.

[8] Y. Wang, A. Davidson, Y. Pan, Y. Wu, A. Riffel, and J. D. Owens, “Gunrock: A high-performance graph processing library on the gpu,” in *Proceedings of the 21st ACM SIGPLAN symposium on principles and practice of parallel programming*, 2016, pp. 1–12.

[9] U. of Texas, “Galois framework,” http://iss.ices.utexas.edu/projects/galois/downloads/Galois-2.3.0.tar.bz2, accessed: 2021-11-4.

[10] 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.

[11] G. Alabandi, E. Powers, and M. Burtscher, “Increasing the parallelism of graph coloring via shortcutting,” in *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2020, pp. 262–275.

[12] M. Adler and M. Mitzenmacher, “Towards compressing web graphs,” in *Proceedings DCC 2001. Data Compression Conference*. IEEE, 2001, pp. 203–212.

[13] E. Abbe and S. Verdú, “Compressing data on graphs with clusters,” in *Proc. of the International Symposium on Information Theory (to appear)*, 2017.

[14] Y. Asano, Y. Miyawaki, and T. Nishizeki, “Efficient compression of web graphs,” in *International Computing and Combinatorics Conference*. Springer, 2008, pp. 1–11.

[15] 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.

[16] J. Shun and G. E. Blelloch, “Ligra: a lightweight graph processing framework for shared memory,” in *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*, 2013, pp. 135–146.

[17] J. Dongarra, “Compressed row storage,” http://www.netlib.org/utk/people/JackDongarra/etemplates/node373.html, accessed: 2021-7-3.

[18] J. Jaiganesh and M. Burtscher, “A high-performance connected components implementation for gpus,” in *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing*, 2018, pp. 92–104.

[19] 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 *2009 IEEE international symposium on workload characterization (IISWC)*. Ieee, 2009, pp. 44–54.

[20] Z. Xu, X. Chen, J. Shen, Y. Zhang, C. Chen, and C. Yang, “Gardenia: A domain-specific benchmark suite for next-generation accelerators,” *arXiv preprint arXiv:1708.04567*, 2017.

[21] M. Burtscher, S. Devale, S. Azimi, J. Jaiganesh, and E. Powers, “A high-quality and fast maximal independent set implementation for gpus,” *ACM Transactions on Parallel Computing (TOPC)*, vol. 5, no. 2, pp. 1–27, 2018.

[22] P. Harish and P. J. Narayanan, “Accelerating large graph algorithms on the gpu using cuda,” in *International conference on high-performance computing*. Springer, 2007, pp. 197–208.

[23] U. of Rome, “Center for discrete mathematics and theoretical computer science,” http://www.dis.uniroma1.it/challenge9/download.shtml, accessed: 2021-11-4.

[24] S. University, “Stanford network analysis platform (snap),” https://snap.stanford.edu/data/, accessed: 2021-11-4.

[25] SSMC, “Suitesparse matrix collection,” https://sparse.tamu.edu/, accessed: 2021-11-4.