Improving High-Performance GPU Graph Traversal with Compression[⋆]

Krzysztof Kaczmarski[1], Piotr Przymus[2], and Paweł Rzążewski[1]

[1] Warsaw University of Technology, Poland
{p.rzazewski,k.kaczmarski}@mini.pw.edu.pl
[2] Nicolaus Copernicus University, Poland
eror@mat.umk.pl

Abstract. Traversing huge graphs is a crucial part of many real-world problems, including graph databases. We show how to apply Fixed Length lightweight compression method for traversing graphs stored in the GPU global memory. This approach allows for a significant saving of memory space, improves data alignment, cache utilization and, in many cases, also processing speed. We tested our solution against the state-of-the-art implementation of BFS for GPU and obtained very promising results.

Keywords: graph searching, BFS, graph compression, data-intensive computation, GPU, CUDA, graph databases.

1 Introduction

Graph algorithms are a foundation of many fields of computer science, including graph databases. Since the graphs appearing in applications tend to be bigger and bigger, both science and industry conduct research to find some more efficient and powerful methods allowing to process them.

Recently, implementations of graph algorithms for the Graphic Processing Units (GPUs) have received considerable attention. A prominent speed-up has been expected due to a massive parallelism offered by the GPU technology. Although the parallel threads programming is now much simplified in this programming model, most of the algorithms (except the ones for embarrassingly parallel problems) need to be redesigned and rewritten. For this reason, new GPU implementations of already known graph algorithms are extensively studied. An example of such an algorithm is the *Breadth-first search (BFS)*, being a building block of many more complicated algorithms and data mining techniques. There have been many studies addressing the implementation of this algorithm on a GPU, followed by a novel work of Merrill, Garland and Grimshaw [11], which outperformed all previous achievements. As the GPU cards often have severe memory limitations, Merril *et al.* also cover the usage of multiple GPU cards, which allows to scale the problem, when the size of the data increases.

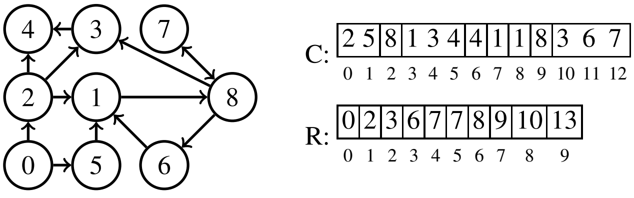
In this work we propose an extension and improvement to the work by Merrill *et al.* [11], by combining BFS with a lightweight compression algorithm. As a result, it

[⋆] The project was funded by National Science Centre, decision DEC-2012/07/D/ST6/02483.

is possible to decrease an overall communication cost between the CPU and the GPU and fit significantly larger graphs in a single GPU. Surprisingly, for large graphs it is also possible to improve the processing time of the algorithm on a single GPU device.

1.1 Preliminaries

In this paper, we consider directed graphs $G = (V,E)$, being a pair of a vertex set V and a directed edge (arc) set $E \subseteq V \times V$. Neither multiple edges of loops are allowed. We use a well-known compressed sparse row (CSR) format to represent an adjacency matrix of our graph. The vertices are indexed with successive non-negative integers. We store a graph $G = (V,E)$ using two arrays C and R. The array C is a concatenation of adjacency list of successive vertices. Therefore its length is exactly $|E|$. The array R has $|V|+1$ elements. The value of $R[i]$ (for $i \in \{0,1,..,|V|-1\}$) points to the index in C of the first element of adjacency list of i. In $R[|V|]$ we keep the total number of edges $|E|$. See Figure 1 for an example. Let C_v be a subarray (segment) of C containing nodes pointed by edges of vertex v, i.e. an adjacency list of v. Clearly $C_v = C[R[v]],\ldots,C[R[v+1]-1]$.



Iter.	Vertex frontier	Edge frontier
1	{0}	{2,5}
2	{2,5}	{1,1,3,4}
3	{1,3,4}	{4,8}
4	{8}	{3,6,7}
5	{6,7}	{1,8}
6	∅	

Fig. 1. A graph, its CSR representation and vertex- and edge-frontiers in iterations of BFS started with node 0

BFS (*Breadth-first search*), one of the best known graph algorithms, starts from a given vertex v and traverses the set of all vertices in a breadth-first manner. Every vertex is labelled with its distance from v (measured in the number of edges). Sometimes we also store the immediate predecessor of each vertex on the shortest path from v. This allows us to re-create all such paths. The complexity of a sequential BFS is $O(|V|+|E|)$.

Although BFS is simple and not very interesting on itself, it is a crucial part of many more complicated and useful graph algorithms, e.g. detecting (strongly) connected components, detecting cycles, checking for bipartiteness or finding a maximum flow. We refer the reader to a classical book by Cormen *et al.* [3] for more information about possible uses of BFS.

Parallel versions of BFS have also been investigated. In this approach, we also start with some initial vertex, forming a one-element set V_1, called a *vertex-frontier*. Then, in every iteration i, the current vertex frontier V_i is expanded, forming a multiset V'_{i+1} of neighbours of vertices from vertex frontier. This multiset is called an *edge-frontier*. To obtain the next vertex frontier V_{i+1}, we need to remove from V'_{i+1} all duplicates and all vertices that have already been visited. If so obtained vertex frontier is empty, the algorithms stops. For an example, consult Figure 1.

1.2 Short History of BFS Implementations for the GPU

In 2006 NVIDIA published the first version of the CUDA platform which enabled programmers to write arbitrary programs executed by vectorised parallel threads with simplified random memory access. Simplified programming paradigm and spectacular benefits in many applications have led CUDA to become a de-facto standard in the general purpose graphic processor unit programming (GPGPU). In this paper, we assume that a reader is familiar with general purpose GPU computing problems and NVIDIA CUDA architecture. Due to the strong page limit we shall not describe notions like SIMD computing, threads, warps, blocks and various memory hierarchy levels. We kindly suggest reading the documentations provided by NVIDIA [12,13], if necessary.

Breadth-first search, being a building block of many graph algorithms and data mining techniques, appeared to be an important task for GPU devices. In 2007 Harish and Narayanan [7] presented the first CUDA implementation. Their approach processes a graph in levels, starting from a given source vertex. In each iteration all vertices which have to be visited in the next step are marked by parallel threads. No global queue of vertices is created due to possible conflicts in memory access by parallel threads and necessity of duplicates removal. This becomes a problem for graphs of high average degree, where the same vertex may be pointed to by many edges. Parallel post-processing and removal of duplicates in the vertex frontier may become a complex and expensive task itself. An obvious strategy is then not to create the vertex queue at all and visit all vertices in each iteration, checking if each vertex has to be visited or not. Unfortunately this solution leads to quadratic processing time (compared to $O(|V|+|E|)$ for the sequential algorithm).

Deng *et al.* [5] achieved the same quadratic complexity for sparse graphs represented by adjacency matrices. In each iteration a frontier propagation is computed by multiplication of a matrix and a vertex vector. Since the entire graph traversal will require $O(|V|)$ multiplications in the worst case and each multiplication has a complexity of $O(|E|)$, we again get at least quadratic processing time.

The only way to achieve an efficiency comparable to a sequential algorithm is to organize a set of vertices to be visited in the next step optimally, without duplicates. Luo, Wong and Hwu [9] were the first who presented such a solution for the GPU. Their hierarchical queue structure produces a vertex frontier array processing incoming vertices. It is first initiated in the shared memory, using the warp-level threads cooperation. Then, on the block-level, the next step of a queue processing is performed. The final shape of the array is formed in the global memory by a proper copying of block level frontiers. It is worth to mention that an efficient implementation is guaranteed by memory coalesced writes and reads.

In 2011 Hong *et al.* [8] noticed that performance may be significantly improved, if the edge frontier array can be processed by warps instead of single threads. They described a *warp-based* task allocation model and extended it further to virtual warps (smaller than normal 32-thread warps), which allowed for better utilization of threads in multiprocessors. In this approach each thread in a virtual warp processes the same vertex and then it processes a single edge coming out of this vertex. Although authors were successful in general description of a better task allocation method, their BFS implementation is based on the Harish and Narayanan solution and achieves only quadratic performance.

In 2011 Merrill, Garland and Grimshaw [11] presented a new implementation of the BFS, which significantly outperformed all previous works. Since this solution is a starting point for our research, a more detailed description follows in the next section.

1.3 Highly Optimized BFS Implementation

The most important part of various BFS implementations is the generation of a vertex frontier for iteration i from a vertex-frontier in iteration $i-1$ (and possibly some other data). Merrill *et al.* [11] described and evaluated a few possible strategies listed below.

Expand-contract. In this approach a single kernel takes care of the current vertex-frontier, expands it into an edge-frontier and then contracts it into the next vertex-frontier. First, threads try to detect duplicates within the warp using some heuristic methods. Then the majority of duplicates and already visited vertices is discarded on the block level. Finally, the whole block is assigned to gather the neighbours from unexpanded adjacency lists of the vertices from the vertex-frontier. This assignment is fine-grained and uses a prefix-sum operation.

Contract-expand. This approach is very similar to the previous one. A single kernel first contracts a given edge-frontier by deleting already visited vertices and identifying most of the duplicates. Then it expands this vertex frontier to an edge frontier, again using prefix-sum operation.

Two-phase. This approach provides separate kernels for the expansion and contraction steps. The expansion kernel uses a clever synchronization strategy, using block-level and warp-level synchronization. Then the duplicates and previously visited vertices are deleted by the contraction kernel, thus producing the next vertex frontier.

The authors finally decide to use the *hybrid strategy*. They perform the *two-phase* approach for large iterations and the *contract-expand* approach in small iterations (i.e. when the edge frontier is small).

It is important to mention that using refined strategies for transforming a vertex-frontier to an edge-frontier and then again to a next vertex-frontier, requires a random access to arrays in which we store the graph. This was not the case in previous, less complicated implementations.

Merrill *et al.* [11] test their solution against the previous solutions for the GPU by Hong *et al.* [8] and Luo *et al.* [9] and also some CPU implementations (both serial and parallel). They were able to obtain a significant improvement in all test cases. The performance they achieve reaches $3.3 \cdot 10^9$ traversed edges per second.

1.4 Motivation

Graphs that need to be processed in real-life applications tend to be really huge. For example, Facebook has about $1.23 \cdot 10^9$ monthly active users ($1.26 \cdot 10^9$ total). The number of friendships (i.e. edges in the social graph) is $125 \cdot 10^9$. The average degrees may vary depending on the region. For example there are $128 \cdot 10^6$ Facebook users in the US and an average number of Facebook friends for them is 350. For a detailed analysis of the Facebook social graph, we refer the reader to Ugander *et al.* [19].

Above mentioned examples shows that the time needed to process an input graph is not the only constraint – we also need to be able to store it in the memory (in our case, memory of the GPU device) somehow. A common solution here is to distribute the algorithm to multiple GPUs, each of which keeps a part of the graph.

In this paper we show how to compress the graph, so that we can store more data in a memory of a single GPU device. This, combined with the distribution of work to multiple GPUs, would allow us to process some extremely large graphs. Our compression proposal has the following properties:

– a smooth integration with existing algorithms,
– a minimal instruction overhead,
– a decreased overall memory footprint (so that bigger graphs can be stored on single GPU device),
– a localized warp-centric and thread-centric decompression, minimizing the number of instructions necessary for the decompression.

Our experience from other applications of a compression shows that in many data-intensive algorithms, the decompression procedure may increase instruction throughput, since threads often have to wait for memory reads to complete. We expect that our approach will improve graph processing algorithms in graph databases.

1.5 Lightweight Compression Methods

The GPU compression topic was raised in several studies. A considerable has been paid to the so-called *lightweight compression* algorithms, which are primarily intended for real-time applications and favor compression/decompression speed over the compression ratio. Their main purpose is to increase a data throughput by a reduction of a data volume. A detailed description of presented compression algorithms may be found in [6,22,4,15].

Interesting results on the GPU compression were presented by Andrzejewski *et al.* [1], where Word Aligned Hybrid compression algorithm for the GPU was presented. Wu *et al.* [20] discussed an implementation of Lempel-Ziv 77 (LZ77) algorithm on CUDA framework and showed that the performance of this algorithm was poor on the GPU processor when compared to the classical CPU implementation, due to many branches and threads divergence problem. Interesting results in the area of lossless compression on the GPU were presented by Fang *et al.* [6]. Using a compression planner it was possible to achieve a significant improvement in overall query processing on the GPU by reducing a data transfer time from RAM to the global device's memory space.

In our previous work we studied possibilities of composing several lightweight compression methods to improve the compression ratio. We have shown that finding the optimal compression plan by a dynamic data analysis may significantly improve results without sacrificing the decompression speed [16].

The **Fixed Length (FL)** compression method works by removing leading zeros at the most significant bits and thus truncating each value to a fixed length, which is the same for all input elements. The main advantage of the FL algorithm (and its variants) is the fact that both compression and decompression are highly effective on the GPU, because these routines contain minimal number of branching-conditions, which decrease parallelism of SIMD operations. For the best efficiency, dedicated compression and decompression routines are prepared for every bit encoding length with unrolled loops and using only shift and mask operations. In our case this method may be used for both arrays R and C. Additionally for C, the number of bits may be different for C_v for each vertex v. This may lead to a better compression ratio, but also a more complicated decompression.

Another method, which may be used, is the **Frame Of Reference (FOR)**. It works in a similar way to FL, but before a compression it transforms each value into an offset from the reference value (for example the smallest value in the set) in a compression block. The reference value is then stored in the compression header. In this situation, we need exactly $\lfloor \log_2(\max - \min) \rfloor + 1$ bits to encode each value in the frame and $\lfloor \log_2 \min \rfloor + 1$ to store the reference value. The best efficiency can be reached if the ordering of vertices reflects the structure of a graph in such a way that the vertices that are close to each other (in the graph) have relatively close indexes. This can be achieved for example by clustering the graph and ordering the vertices of each cluster separately.

Let us now analyse the applicability of FOR compression of C and R arrays (having in mind that we require a random access to its elements). As explained above, this method divides data into segments named frames. All values in one frame are compressed together but it is possible to decompress any of them by reading a header (the reference value) and a compressed value itself. Reading two values (instead of one) in each read operation is a waste of bandwidth and processing time.

The **Differential Representation** approach stores only the differences between successive data points. Since we need a random access to all elements, a decompression of a value would require to scan the whole array, which takes a linear time. Also the **Dictionary** method (in all variants including the Tunstall encoding) is not suitable for CSR graph representation, since there are too many different values to encode. Creating a dictionary of them would make no sense. The **Run length encoding** stores an array of repeating values as an array of pairs: value and run length (the number of successive appearances of an element). This method is not suitable for us, since we do not expect values to repeat often on consecutive positions in the arrays R (they would correspond to subsequent vertices with no out-edges) or C (they would correspond to subsequent vertices of out-degree 1, having a common neighbour).

All integer encoding methods of variable length based on prefix-codes (e.g. Elias, Shannon-Fano, Huffman, Golomb – consult a comprehensive book by Salomon [18] for more information) do not support a random access and thus are not applicable in case of graph algorithms.

The main drawback of many lightweight compression schemes is that they are prone to outliers in the data frame. For example, consider the following data frame $\{1,2,3,5,7,128\}$ and the FL compression scheme. One could use a 3-bit fixed-length compression to encode almost all values in the frame, but due to the outlier (the value

128) we have to use 7-bit fixed-length compression. A solution of this problem is to use the so-called **Patched Lightweight Compression**. An example of this approach has been proposed by Zukowski *et al.* [22] as a modification of three lightweight compression algorithms. Their main idea is to store outliers as exceptions in an additional array. However, variable number of exceptions lead to many branches in code and decrease efficiency of parallel threads. Various solutions have been proposed to cope with this problem, such as reducing the frame size [22], avoiding too many exceptions [4,21] or separating decoding and patching processes [14,17].

2 Compression of the CSR Graph Representation

To choose a compression method which is suitable for graph processing using a GPU device, we need to analyse the behavior of the BFS algorithm in the aspect of memory access and the graph representation in memory.

To our best knowledge, in parallel implementations of BFS there are two possibilities of parallel threads behavior, when reading the edges to be visited in the next algorithm iteration:

- A single thread reads a vertex to be visited and then performs a series of sequential read operations in the array C_v, looking for the corresponding edges.
- A group of threads (a warp or a block) reads the same vertex to be visited and then, in parallel, reads all its edges from the array C_v. If the number of edges is bigger than the number of threads, then this process iterates until all edges are visited. Similarly, if number of edges is smaller, then some threads may be idle.

In both options, both arrays C and R are accessed randomly, but in the second solution bigger fragments of array C can be read together.

In the case of our example in Figure 1, the first approach would lead to one thread reading vertex number 0 and then its neighbours $\{2,5\}$ followed by two threads processing two vertices 2 and 5 in parallel and producing two edge frontiers $\{1,3,4\}$ and $\{1\}$, respectively. The number of read operations is three in the case of the the first thread and only one for the second thread.

The second approach would use a group of threads (most effectively a warp) for reading vertex 0 from the vertex frontier. Next two threads would concurrently read two edges from the C_0 array and then produce one edge frontier containing $\{2,5\}$. In the second iteration two groups (warps) would access two distant places in array C reading corresponding edge frontiers. Each thread in a group would read its corresponding value: 1,3 and 4 in the first group and 1 in the second group. Two resulting frontiers will be created simultaneously in one step. If a coalesced memory access is possible, then the process would end in two memory read operations. This fact of a better thread utilization in the case of a warp-level edge-frontier access was already noted by other authors [9,11] and is related to the parallel processing model of the GPU device.

Considering the memory space needed to store the graph, we observe that the array R is sorted and stores indices of the much bigger array of edges C. Each segment C_v can also be sorted. Both arrays contain only non-negative integers. We also need to note that R contains much bigger values (its last element is the sum of degrees of all vertices, which is equal to the total number of edges).

The threads behaviour and memory representation leads to important conclusions:
1. the array R after compression must allow for a random access to any of its elements;
2. the same holds for the array C, but this array may be divided into blocks C_v;
3. a group of threads may cooperate in decompression of edge frontiers read from C.

Note that BFS is a data intensive algorithm. It performs very few computations and can significantly slow down if a decompression method is too expensive or creates some unwanted threads divergence by branching.

The above statements focus our attention on lightweight compression methods, which are local, do not use patching mechanisms and allow a value to be decompressed solely upon information from the data read in a constant time. According to the analysis from Section 1.5, the FL compression method seems to be the most flexible and promising. Therefore we chose to use it in our approach.

2.1 Fixed-Length Compression of Large In-Memory Arrays

In this section we discuss in detail the consequences of choosing the FL compression scheme for a large array, which require a random access.

Memory Organization Consequences of the FL Compression Method. Consider an array compressed with the FL algorithm, with each value written on ℓ bits. We store them in an array of k-element memory cells (in most cases we shall use $k = 32$, as it is best supported by current GPU devices). Observe that some values will be stored in two consecutive memory cells. In those cases, to retrieve the value, we need to read two cells, which significantly increases the cost of the read operation. Therefore we want to keep the number of such values as small as possible. In a perfect situation, when ℓ divides k, there are no values spanning over multiple cells.

Consider a block A of our array, whose length is equal to $\mathrm{lcm}(k,\ell)$, being the least common multiple of k and ℓ (as the whole array consists of such blocks and some remainder, which has constant length and therefore can be omitted in our analysis). Let x and y be integers such that $\mathrm{lcm}(k,\ell) = x \cdot \ell = y \cdot k$. The number of values spanning over two consecutive cells is exactly $y - 1 = \frac{\ell}{\gcd(k,\ell)} - 1$ $(\star)$.

From this we can see that there are two ways to minimize the number of read operations – by making ℓ small or by making $\gcd(k,\ell)$ large.

Expected Cost of Random Array Access. The above statements lead to an important conclusion that the additional cost of memory operations (when compared to an array without any compression) depends on the values of ℓ and k. Suppose we have an array of X values, consisting of blocks of size $\frac{k}{\gcd(k,\ell)}$ (as mentioned before, we do not care about some remainder, as its length is constant). Therefore, from $(\star)$, the total number of values which occupy two consecutive cells is $\frac{X}{x} \cdot \left(\frac{\ell}{\gcd(k,\ell)} - 1\right) = X \cdot \frac{\ell - \gcd(k,\ell)}{k}$. Let $\alpha := \frac{\ell - \gcd(k,\ell)}{k}$. Now αX is the number of values spanning over two consecutive cells. If we choose a random value (with a uniform probability), we get a value in two cells with probability $\frac{\alpha X}{X} = \alpha$ and a value in just one cell with probability $\frac{1-\alpha X}{X} = 1 - \alpha$. Suppose we want to read m random values, chosen with uniform probability. The expected value of read operations is:

$$\mathbb{EX}(\text{number of read operations when reading } m \text{ values}) =$$
$$m \cdot \mathbb{EX}(\text{number of read operations when reading one value}) =$$
$$= m \cdot (1 \cdot (1-\alpha) + 2 \cdot \alpha) = m \cdot (1+\alpha) = m \cdot (1 + \frac{\ell - \gcd(k,\ell)}{k}).$$

This is compared with m read operations needed to retrieve m values from a non-compressed array. Obviously if ℓ divides k then additional cost is 0.

Moreover, observe that so far we only considered the simplest case when each value was immediately followed by the next one and we had no unused bits. However, this may not be an optimal approach. Consider for example cells of size $k = 32$ bits and $\ell = 5$. We may consider storing 6 values in a single cell and leaving two remaining bits unused (so the next value starts in the next cell). With this approach we increase the size of the data (and thus reduce the compression ratio), but we never have to read more than one cell to retrieve a single value, which improves the efficiency of processing (we need m read operations to read m values).

Actually, in our experiment we use such a modification. Instead of storing each value on $\ell = \max\{\lfloor \log_2 z \rfloor + 1 \colon z \text{ is a value to be stored}\}$ bits, we chose some $\ell' \geq \ell$, which allows us to reduce the number of values spanning over two cells. Table 1 shows the optimal values ℓ and chosen values ℓ' for benchmark graph. If $\ell' = 21$, then we just stored three values in two 32-bit memory cells and left one last bit unused. For the graphs with $\ell = \ell' = 16$, we just stored two values in a single 32-bit memory cell. Observe that a small loss in the compression ratio is justified by fewer read operations.

3 Benchmark Graphs and Results of Experiments

In order to confirm the effectiveness of our approach we test it against the fastest known BFS implementation, which was already discussed in Section 1.3. Unfortunately most of the data sets mentioned by Merrill *et al.* were not available when we performed the tests. We only managed to download several Citeseer and DBLP graphs. However, we were able to use the same graph generator: R-MAT (see Chakrabarti *et al.* [2] for details). Such graphs reflect specific properties of large graphs appearing in real-world applications.

We run the experiments on graphs having from $65.5 \cdot 10^3$ to $2 \cdot 10^6$ vertices and up to $300 \cdot 10^6$ edges. Table 1 lists the parameters of benchmark graphs.

The code of the solution by Merrill *et al.* [11] is available for public as a part of the *back40computing (b40c)* project [10]. Therefore we were able to apply our improvements directly into their fine-tuned implementation. Although the changes were not straight-forward, eventually we modified the original code in two aspects (altogether highly touching many places in the code):

– creating graph representation in the memory (by adding the FL compression),
– the function call controlling an access to the elements of C (decoding vertices).

The graph compression depends on the selected method's parameters ℓ, k and was explained in Section 2.1.

The internal architecture of the *b40c* implementation is using almost all available on-chip shared memory. We were not able to utilize it in the decompression process. Therefore the vertex decoding had to be done in a very simple way just using threads' private registers and without any additional intercommunication between threads. This solution required more memory operations and processing when compared to the ideal one. The differences between the thread behavior in the original approach and our approach is presented in Figure 2.

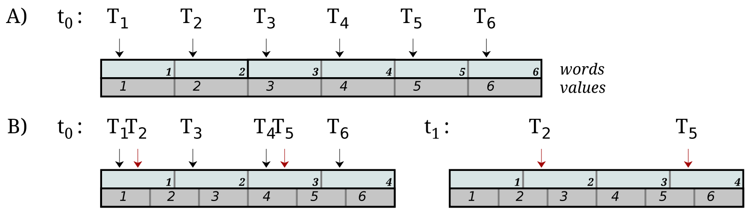

A)
t_0:
T_1 T_2 T_3 T_4 T_5 T_6
words
values
B)
t_0:
T_1T_2 T_3 T_4T_5 T_6
t_1:
T_2 T_5

Fig. 2. A simple strategy of parallel values decoding. A) no compression. A single memory cell stores a single value, which is read by a single thread. B) The FL compression. One memory cell stores 1.5 values. Some threads need to read two cells with 100% cache hits. Overall cache usage is decreased. No threads intercommunication.

An example of a decompression function is shown in Figure 3. It was prepared for $\ell' = 21$ bits. Choosing another length would require slight changes in the function. The key point in this code are modulo operations which are done by bit operations only. They are necessary to find beginnings of compressed values. In some cases, if the value is split across two memory cells, a thread needs to perform another read operation (see line 13). This is necessary if we assume no communication between threads.

We believe that this approach will be successful also in other applications since it allows for a random array access and imposes no additional restrictions.

```
1  efine NBITSTOMASK(n) ((1<<(n)) - 1)
2  efine GETNBITS(a,n)  ((a) & NBITSTOMASK(n)) // returns a0a1..an-1
3  efine GETNPBITS(a, n, p) GETNBITS((a>>p), (n)) // returns apap+1..ap+n-1
4
5  device__ __forceinline__ static
6  id Ld_21_64(T &val, T *ptr, long n) {
7  unsigned int a = (unsigned) n;
8  a = (0x55555555*a+(a>>1)-(a>>3))>>30;
9  unsigned int pos = ((unsigned)n-a)*0xAAAAAAAB;
10 pos = pos * 2 + (a>>1);
11 val = GETNPBITS(ptr[pos],21-10*(a&1),((a*21)&31));
12 if(a&1)
13   val = val|GETNPBITS(ptr[pos+(a&1)],10*(a&1),0)<<11;
```

Fig. 3. An example of a data retrieving function. **Ld_21_32** function with FL decompression for $\ell' = 21$ and $k = 32$ (three values are encoded in two subsequent integers) n – an index of an element to be decoded.

Table 1. Experimental data sets. The first group of columns shows the number of vertices, edges and an average degree of each graph. The second group shows an optimal (ℓ) and a chosen by us (ℓ') length of an encoding of a single value and k, being the size of a single memory cell. Third group shows the size of C before compression, after compression with each value encoded on ℓ or ℓ' bits and corresponding compression ratios.

Graph	vert. $\cdot 10^3$	edges $\cdot 10^6$	avg. degree	ℓ bits	ℓ' bits	k bits	C [MB]	$\text{FL}_\ell(C)$ [MB]	compr. ratio (ℓ)	$\text{FL}_{\ell'}(C)$ [MB]	compr. ratio (ℓ')
citationCiteseer	268	1.15	4.3	20	21	32	4.39	2.74	0.63	2.92	0.67
coAuthorsCiteseer	227	0.81	3.58	20	21	32	3.09	1.93	0.63	2.06	0.67
coAuthorsDBLP	299	0.98	3.26	20	21	32	3.74	2.34	0.63	2.49	0.67
coPapersCiteseer	434	16.03	37.55	20	21	32	61.15	38.22	0.63	40.77	0.67
coPapersDBLP	540	15.24	28	20	21	32	58.14	36.33	0.63	38.76	0.67
RM131Kv19Me	131	19.65	150	17	21	32	74.96	39.82	0.53	49.97	0.67
RM131Kv39Me	131	39.30	300	17	21	32	149.92	79.64	0.53	99.95	0.67
RM131Kv78Me	131	78.60	600	17	21	32	299.84	159.29	0.53	199.89	0.67
RM2Mv150Me	2000	150	150	21	21	32	572.20	375.51	0.66	381.47	0.67
RM2Mv301Me	2000	301	301	21	21	32	1148.22	753.52	0.66	765.48	0.67
RM2Mv350Me	2000	350	350	21	21	32	1335.14	876.19	0.66	890.10	0.67
RM2Mv400Me	2000	400	400	21	21	32	1525.88	1001.36	0.66	1017.25	0.67
RM65.5Kv10Me	65.5	10	152.67	16	16	32	38.15	19.07	0.50	19.07	0.50
RM65.5Kv67Me	65.5	67	1022.90	16	16	32	255.58	127.79	0.50	127.79	0.50
RM65.5Kv104Me	65.5	104	1587.78	16	16	32	396.73	198.36	0.50	198.36	0.50
RM65.5Kv268Me	65.5	268	4091.60	16	16	32	1022.34	511.17	0.50	511.17	0.50

All the experiments were executed on the same model of GPU processor as the experiments by Merrill *et al.* [11]. Detailed hardware configuration: two six-core processors *Intel® Xeon® E5649 2.53GHz*, 8GB RAM and *Nvidia® Tesla M2070* card.

3.1 Discussion on Results

The results of our experiments are shown in Table 2 (average value of 10 executions).

Compression. Due to the limitation of the BFS implementation we worked with, we could only use a very simple FL compression scheme with a random access to the array of edges (C). Obviously in such a case the compression ratio depends on the number of bits which are used to store a vertex identifier. In most of the reference sample data sets only 21 bits were used, which was enough to pack three nodes into two integers (i.e. a 64 bit segment). In such cases, achieved compression ratio varied from 0.53 to 0.62 (of the original size).

Let us now analyse how big graphs may be stored in a GPU device with 6 GB of memory (this is the theoretical storage space of *Nvidia® Tesla M2070*), assuming that the average degree of a node is 40 and the values are stored as 32-bit integers. Using the CSR representation, a single vertex v requires 4 bytes for a corresponding cell in R array and $40 \cdot 4$ bytes on average for the C_v array. Therefore, in theory, a professional GPU device with a memory of 6 442 450 944 bytes lets us to store a graph of up to

$n := 39\ 283\ 237$ vertices (of course this would require to use all the memory just for the graph representation, leaving no space for e.g. some additional structures used by the algorithm, so it is just a theoretical upper bound). To store the indices of these nodes we need 26 bits. By compressing the array C with FL method and using $\ell = 26$ and $k = 32$, we could pack 6 values in 5 integers (by wasting 4 bits). A single vertex with its out-edges needs now $4(1 + \lceil \frac{5 \cdot 40}{6} \rceil) = 140$ bytes on average. Therefore our graph with n vertices would occupy only 5 499 653 180 bytes together gives 5.12GB. Memory we saved in such a way would let us to store 6 734 629 additional vertices and their compressed edges (note that the vertex indices still can be stored on 26 bits). In this configuration we managed to increase the practical device capacity by over 17%. Notice that this can be significantly improved for graphs with larger average degree (as the number of bits needed to represent a vertex remains low and the size of C grows).

This may be crucial is some cases – e.g. a graph with $2 \cdot 10^9$ vertices and $400 \cdot 10^9$ was too big to fit into the memory of the GPU device without a compression (Table 2).

We also observe that it would need a device with memory storage of 328 GB to use all 32 bits in an integer encoding the vertex indices. Therefore, in the case of current GPU devices, savings using the FL compression are always possible.

Another benefit of this compression method is that it can significantly decrease the memory bandwidth when executing multi-GPU algorithms and whenever memory transfer of a graph or its parts is used.

BFS Algorithm Time. At the beginning we have to observe that the time of processing of compressed graphs depends on two factors: a ratio between ℓ (or ℓ') and k, i.e. the efficiency of a compression (being the number of values we can pack into a single memory cell) and an average degree of a vertex.

Moreover, we observe that the additional compression/decompression cost is compensated for medium-sized graphs. For large graphs we are even able to speed up the computation.

Table 2. The time of BFS processing for benchmark graphs [ms] (smaller is better). The last column shows the improvement over the original solution (greater is better). The graph RM2Mv400Me with $400 \cdot 10^9$ edges could not be processed without the compression.

	Processing time [ms]		**Speed up**		**Processing time [ms]**		**Speed up**
Graph	**b40c**	**b40c+FL**	**[%]**	**Graph**	**b40c**	**b40c+FL**	**[%]**
citationCiteseer	2.9948	3.4951	-14.31	RM2Mv150Me	61.3072	59.4512	3.12
coAuthorsCiteseer	2.0487	2.6138	-21.62	RM2Mv301Me	115.656	111.2375	3.97
coAuthorsDBLP	2.4007	2.9336	-18.17	RM2Mv350Me	132.7916	127.9881	3.75
coPapersCiteseer	12.7294	14.2871	-10.90	RM2Mv400Me	X	144.6979	X
coPapersDBLP	11.5055	12.5845	-8.57	RM65.5Kv10Me	4.6509	4.3214	7.62
RM131Kv19.6Me	7.0177	7.0178	0.00	RM65.5Kv67Me	22.9852	20.9861	9.53
RM131Kv39.3Me	12.6992	12.5758	0.98	RM65.5Kv104Me	35.6296	32.466	9.74
RM131Kv78.6Me	23.897	23.5884	1.31	RM65.5Kv268Me	97.943	86.212	13.61

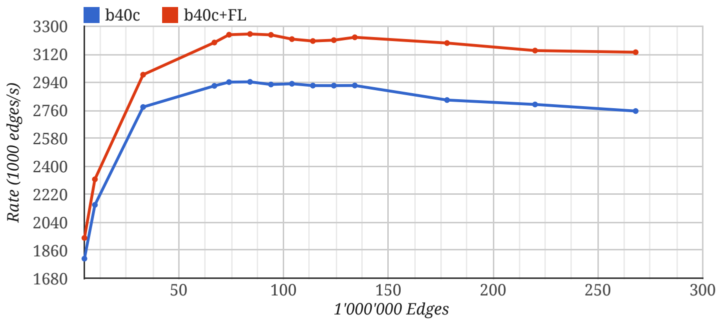


Fig. 4. Rate (edges per millisecond, greater is better) of the BFS algorihtm for graphs with $65.5 \cdot 10^3$ vertices and different number of edges

4 Conclusions and Future Work

We have presented a method of compressing graphs stored in the CSR format and processed in GPU devices. Our solution is characterized by an ultra-fast decompression time, a simplicity of integration with already existing algorithms and an optimization of parallel threads computation.

We evaluated our solution against the state-of-the-art in graph algorithms – the highly-optimized BFS implementation for GPU devices by Merrill *et al.* [11]. Our results show that for big graphs the compression not only allows to fit more vertices and edges into a single GPU, but also speeds up the processing by a better utilization of memory caches.

We believe that our improvement can also be used in a case of a distributed computation performed on multiple GPU nodes or in clusters. Using a compression should significantly speed up the most critical operation, which is a data transfer.

References

1. Andrzejewski, W., Wrembel, R.: GPU-WAH: Applying gPUs to compressing bitmap indexes with word aligned hybrid. In: Bringas, P.G., Hameurlain, A., Quirchmayr, G. (eds.) DEXA 2010, Part II. LNCS, vol. 6262, pp. 315–329. Springer, Heidelberg (2010)
2. Chakrabarti, D., Zhan, Y., Faloutsos, C.: R-MAT: A recursive model for graph mining. In: SDM, pp. 442–446 (2004)
3. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, 3rd edn. MIT Press (2009)
4. Delbru, R., Campinas, S., Samp, K., Tummarello, G.: Adaptive frame of reference for compressing inverted lists. Technical report, DERI – Digital Enterprise Research Institute (December 2010)
5. Deng, Y.S., Wang, B.D., Mu, S.: Taming irregular EDA applications on GPUs. In: Proceedings of the 2009 International Conference on Computer-Aided Design, ICCAD 2009, pp. 539–546. ACM, New York (2009)
6. Fang, W., He, B., Luo, Q.: Database compression on graphics processors. Proceedings of the VLDB Endowment 3(1-2), 670–680 (2010)

7. Harish, P., Narayanan, P.J.: Accelerating large graph algorithms on the GPU using CUDA. In: Aluru, S., Parashar, M., Badrinath, R., Prasanna, V.K. (eds.) HiPC 2007. LNCS, vol. 4873, pp. 197–208. Springer, Heidelberg (2007)
8. Hong, S., Kim, S.K., Oguntebi, T., Olukotun, K.: Accelerating CUDA graph algorithms at maximum warp. In: Cascaval, C., Yew, P.-C. (eds.) PPOPP, pp. 267–276. ACM (2011)
9. Luo, L., Wong, M.D.F., Mei, W., Hwu, W.: An effective GPU implementation of breadthfirst search. In: Sapatnekar, S.S. (ed.) DAC, pp. 52–55. ACM (2010)
10. Merrill, D.: Back40computing (2013), `https://code.google.com/p/back40computing/`
11. Merrill, D., Garland, M., Grimshaw, A.S.: Scalable gpu graph traversal. In: Ramanujam, J., Sadayappan, P. (eds.) PPOPP, pp. 117–128. ACM (2012)
12. NVIDIA Corporation. NVIDIA CUDA C programming guide 5.5 (2013)
13. NVIDIA Corporation. CUDA C Toolkit v.5.5 (2014)
14. Przymus, P., Kaczmarski, K.: Improving efficiency of data intensive applications on GPU using lightweight compression. In: Herrero, P., Panetto, H., Meersman, R., Dillon, T. (eds.) OTM 2012 Workshops. LNCS, vol. 7567, pp. 3–12. Springer, Heidelberg (2012)
15. Przymus, P., Kaczmarski, K.: Dynamic compression strategy for time series database using GPU. In: Catania, B., et al. (eds.) New Trends in Databases and Information Systems. AISC, vol. 241, pp. 235–244. Springer, Heidelberg (2014)
16. Przymus, P., Kaczmarski, K.: Dynamic compression strategy for time series database using GPU. In: Catania, B., et al. (eds.) New Trends in Databases and Information Systems. AISC, vol. 241, pp. 235–244. Springer, Heidelberg (2014)
17. Przymus, P., Kaczmarski, K.: Time series queries processing with GPU support. In: Catania, B., et al. (eds.) New Trends in Databases and Information Systems. AISC, vol. 241, pp. 53–60. Springer, Heidelberg (2014)
18. Salomon, D.: Data Compression: The Complete Reference. Springer (1998)
19. Ugander, J., Karrer, B., Backstrom, L., Marlow, C.: The anatomy of the Facebook social graph. CoRR, abs/1111.4503 (2011)
20. Wu, L., Storus, M., Cross, D.: CS315A: Final project CUDA WUDA SHUDA: CUDA compression project (2009)
21. Yan, H., Ding, S., Suel, T.: Inverted index compression and query processing with optimized document ordering. In: Proc. of the 18th Intern. Conf. on World Wide Web, pp. 401–410. ACM (2009)
22. Zukowski, M., Heman, S., Nes, N., Boncz, P.: Super-scalar RAM-CPU cache compression. In: Proc. of the 22nd Intern. Conf. on Data Engineering, ICDE 2006, pp. 59–59. IEEE (2006)