Faster across the PCIe bus: A GPU library for lightweight decompression

including support for patched compression schemes

Eyal Rozenberg
CWI, Amsterdam
E.Rozenberg@cwi.nl

Peter Boncz
CWI, Amsterdam
P.Boncz@cwi.nl

ABSTRACT

This short paper present a collection of GPU lightweight decompression algorithms implementations within a FOSS library, **Giddy** – the first to be published to offer such functionality. As the use of compression is important in ameliorating PCIe data transfer bottlenecks, we believe this library and its constituent implementations can serve as useful building blocks in GPU-accelerated DBMSes — as well as other data-intensive systems.

The paper also includes an initial exploration of GPU-oriented *patched* compression schemes. Patching makes compression ratio robust against outliers, and is important with real-life data, which (in contrast to many synthetic benchmark datasets) exhibits non-uniform data distributions and noise.

An experimental evaluation of both the unpatched and the patched schemes in `Giddy` is included.

1. INTRODUCTION

The computational power of GPUs and the rapid pace of its increase, has made GPUs an interesting platform for a broad class of General-Purpose tasks (GPGPU), among which query processing [4]. In practice, though, actually accelerating database workloads by GPU processing is hindered by the required data transfers over the PCIe bus. In a previous GPU-accelerated query processor one of the authors had participated in building [2], on TPC-H benchmark query workloads, 70% of time was typically spent purely on such transfer. Some published research circumvents this problem, by assuming the data is GPU-memory-resident; but this is unrealistic, since the fast on-GPU memory is 10x-25x smaller than main system memory, and minuscule when considering secondary storage. It is thus imperative to tackle this bottleneck for GPUs to make a difference where acceleration is important, i.e. when processing very large amounts of data.

While heavily data distribution dependent, compression in data warehouses reduce volumes by a factor of 3–4 in typical use cases, with this translating directly into shorter PCIe transfer times. And while compressed-form data incurs a penalty in computation and GPU memory write time for decompression, overall performance is still likely to improve by a factor close to the compression ratio, times the fraction of time spent on data transfer. Moreover, compressed forms of data may be operated on directly, speeding up various execution plan operations [1] and reducing the amount of data to require actual decompression eventually.

DaMoN'17, May 15, 2017, Chicago, IL, USA

ISBN 978-1-4503-5025-9/17/05…15.00

DOI: http://dx.doi.org/10.1145/3076113.3076122

Prior work on de/compression with GPUs mostly regards use of the GPU as the *means* (e.g. [8]) and de/compression as the *ends* — a computational task offloaded to a GPU. The only works where lightweight compression schemes are the means, and computational work (such as query processing) on the GPU the ends, are [7] and [15]. The latter contributes useful auto-tuning compression methods and parameters, but does not describe GPU decompression methods, which are the focus here. The former paper surveys a number of lightweight compression schemes usable for sending columnar data across PCIe; discusses the relative desirability of composing GPU compression schemes; and presents experimental results, evaluating the performance of decompression proper and of some analytic queries on compressed data. The data layout and decompression algorithm for each scheme is described in broad strokes mostly. Unfortunately, by now (6 years later), the authors of [7] report the source code for that work has been lost.

This paper contributes a free, open-source GPU compression library with a modern implementation of the most widely-used lightweight database compression schemes. It is the first paper to explore the use of GPU-targetted *patched* decompression schemes [23]. Patching makes compression ratios robust to outliers, which are prevalent in (noisy) real-life data — as opposed to synthetic benchmark-generated datasets, such as the TPC benchmarks. We think this library provides a useful building block for today's and tomorrow's GPU-accelerated data management systems.

2. THE DECOMPRESSORS LIBRARY

Our library implementing the decompressors can be found in an online public repository [16]. It uses C++14 for host-side code and C++11-flavored CUDA for device-side code, and is GPLv3-licensed. There are three modes of using it: (i) Directly including GPU kernel sources in a program — as they are kept separate for any wrapper code (ii) Using thin per-kernel low-level GPU kernel wrapper classes (iii) Obtaining opaque kernel wrappers, accessed via a base-class pointer, from a string-keyed factory initialized at program load time. A facility is provided for selecting appropriate kernel launch parameters, either automatically (with options (ii) and (iii)) or manually (all options).

The compression schemes currently supported are:

Unpatched: BITMAP, FOR, DELTA, DICT, NS, NSV (variable length), MODEL (see below), RLE.

Patched: DICT, NSF, NSV, FOR, BITMAP, MODEL.

These are supported for all relevant combinations of sizes of 1, 2, 4 and 8 bytes. The patched schemes are available in two variants, "naive" and with compressed patch position indices; these are discussed in Subsection 3.1 below. Note that among the many variants of NS schemes, we follow — at this stage of implementation — the choice in [7] of two NS variants, in which the compressed elements themselves are *byte-aligned*.

The library is under active development, particularly with the aim of further improving performance, and supporting additional compression schemes (e.g. from among those surveyed [5], and the patched schemes PFOR-DELTA [23] and Fast-PFOR [5]). This paper only covers what has already implemented and tested.

3. COMPRESSION SCHEMES

For most of the unpatched schemes, we use a different layout of compressed data, and/or a different algorithm for decompression, than established CPU-oriented variants of these schemes; often these are also different also from the GPU-targeted work in [7]. Going into the details is beyond the scope of this short-form paper; instead, we'll list recurring aspects of these variations.

Columnnar layout of compressed data. While several DBMSes using compression are columnar, i.e. physically hold the data of a single column of all records toghether, separate from the data for other columns — this 'columnarity' of the layout is not typically maintained after compression. The common practice is creating tightly-knit compressed blocks, with headers and in-block data of various types; and that is very useful on a CPU, keeping all relevant data necessary for decompressing a short sequence of elements closeby in cache. On a GPU, however, such of the cache for a lightweight decompression workload is essentially impossible: So many threads would try to use is simultaneously, or nearly-simultaneously, and it would simply get thrashed. For this reason (among others) we chose keep the different kinds of data as separate, uniformly-typed arrays. Figure 1 illustrates, in particular, this difference in data layout.

Anchoring. In some schemes, a small number of single values, which may be derived from a large part the compressed form with some computational work, are necessary for completing the decompression. An example: The sum of compressed elements upto some point, in the DELTA scheme. Instead of actually computing these, we augment the compressed form with their pre-computated values, regularly over large compressed segments; this speeds up the decompression algorithm at the price of a minute degradation in compression ratio. Anchors may involve *positions* in the input or output, or other parameters which aren't values in the uncompressed domain.

Scheme-specific implementation optimization. The work in [7] made use of well-known GPU primitives such as Gather (referring to a Map with the map function being an array lookup). and Prefix Sum (a.k.a. Prefix Scan; see [10, §3.1.1]). We utilize them less often, and when we do, it is sometimes with some modification or a part cut-out. Instead, more schemes involve specific code for their decompression.

3.1 Patched schemes

The original patched lightweight compression schemes in [23] (PFOR, PFOR-DELTA and PDICT) were designed in the fit-everything-into-the-block-in-cache approach, mentioned above; and the patch values for each block are, in fact, crammed into it. The consequence of this choice is temporal locality and cache-locality of the unpatched decompression and the application patches. In [12] (and [13]), this construction can be said to be partially unpacked in favor of vertical layout of unpatched data and patch information. On a GPU, we opt for *complete* unpacking and fully vertical layouts — following the same similarly to the unpatched schemes. In fact, we decouple the unpatched decompression from the application of patches entirely: We schedule a complete decompression using an unpatched scheme — any of those listed in Section 2 (except for DELTA); and an *aposteriori patching* operation on the decompressed column.

The nature of this aposteriori patching phase depends on how the patches are laid out in memory. The *Naive Patching* layout constitutes two corresponding arrays, of patch positions and values of the uncompressed type; its application is simply a Scatter operation [9] — in which case they need not even be sorted. An example appears in 1.

One can go beyond naive patching by compressing the patch values and/or the patch positions. Both may involve cascading other lightweight compression schemes (an option not implemented in **Giddy**), but for the patch positions one can tailor a custom representation, based on knowledge regarding their distribution. Indeed, we may assume that patch positions are monotone increasing; and in this case, either there are very few patches (and then the compression ratio of the patches doesn't matter), or they are not too far apart on the average, and can form multiple runs with small offsets from a baseline position. We are currently working on a *Compressed Indices aposteriori patching* scheme using these principles — which has an initial implementation in-place, not yet instrumented for performance testing.

3.2 Less-common unpatched schemes

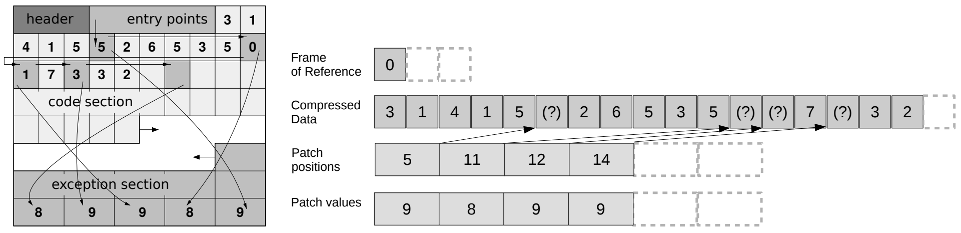
Generalized FOR. In the FOR scheme, the same constant reference value is added to every compressed element (or rather the same value throughout each segment of the column). It is rather straightforward to generalize this to the addition of a value depending on the position of the uncompressed element; in other words, decompression is the elementwise column addition of model-function-generated column and the compressed form of our column: Instead of $d(i) = c(i) + v$ we have $d(i) = c(i) + f(i)$, where d is our decompressed column, c is its compressed form, and v and f are a constant and a function respectively. One naturally conceives of some finite-dimensional linear space of potential model functions f; say, the space of polynomial functions of degree up to d. Thus a model function is specified to the kernel by passing its $d+1$ coefficients (or rather, a set of such coefficients for each segment). The original FOR scheme is the special case of this generalization with $f(i) = v$ uniformly.

MODEL. Consider the generalized FOR just described, without the compressed elements; this is the MODEL compression scheme. The decompressed data is merely the evaluation of a (per-segment) model function. This is a 'degenerate' scheme, which cannot apply to most columns; but its implementation and use is not without motivation. For example, "Record ID"-type columns in schemata of append-only databases (such as `c_custkey` in Section 4) do often perfectly fit an affine function ($f(i) = a_0 + i$); and there is also its relation to generalized FOR, and its possible augmentation with *patches* for outlier data (see Subsection 3.1).

4. EXPERIMENTAL EVALUATION

4.1 Experimental setup

The library was evaluated using a GPU kernel test harness available online [17], along with instructions on reproducing the results. Timing includes only the kernel execution proper; the PCIe transfer times are stated based on the rule of thumb



(a) in Vectorwise (b) in Giddy (naive patching)

Figure 1: Patched FOR compressed form layout, with digits of π as the data: 3.1415926535897932

of 12 GB/sec achievable bandwidth with PCIe 3.0. Experiments were conducted on an ASUS card with nVIDIA GTX Titan X GPU (GM200 chip, 336 GB/sec memory bandwidth), with factory settings. CUDA v8.0.44 was used, on a Fedora 24 system with nVIDIA driver v375.26.

Data sets used. The implementations were evaluated using two kinds of data: synthetic and real-life. Our *synthetic* data are the generated tables for the widely-used **TPC-H** benchmark [21]. They have a uniform distributions for non-deterministic data: No local skews, noise or outliers. Thus when they are compressible, they are only amenable to uniform, non-adaptive compression — as they exhibits no changing behavior, outliers, or skewed frequency distribution. We used Scale Factor 10, with 59986052≅60M records in `lineitem`.

The *"Real-life"* data used is the US Department of transport's on-time flight information, **Ontime** for short, available online [22]. It is a single de-normalized table, with ~100 columns. Each record describes a flight: When it left and arrived, where to and where from, what diversions and delays it experienced and so on. We limit our data to the years 2000-2008: 59287698≅60M records overall, similar to TPC-H at SF 10. While this data set is not part of a well-established database benchmark, it has been to compare analytic DBMS performance in the past [20]. The raw data files are 'dirty': mis-typed values, missing fields which should not be nulls, duplicate entries for the same flight etc.; we therefore perform some minimal scrubbing and unification of in-file order before actually loading the data.

Readers are referred to the repositories [18, 19], with scripts to obtain and load the two data sets. These scripts may be useful independently of the reproduction of our results.

Experiment description. For every column selected for the experiments, we chose a single compression scheme when it clearly outdoes all others, or several schemes otherwise. If the column benefits from patching, the best patched and unpatched schemes were chosen. Each column was decompressed 6 times with each chosen scheme; the first run was ignored and the reported result is the median of the last 5.

Direct comparison with other work on lightweight compression, or reproduction of experiments therein, was not carried out. First, w.r.t.on-CPU decompression — most compression schemes are somewhat modified, as described in Section 3, thus it would be an apples-to-oranges comparison; also, the bandwidths achievable on a CPU are typically an order of magnitude lower, or more (see e.g. [12, 5]). Unfortunately, immediate comparison was also impossible with the work in [7]: The paper itself includes few exact experiment results; the GPU used there was of a much older architecture; and the source code used in that paper was misplaced by the authors [6] and cannot be brought up-to-date.

Direct reproducing of experiments in other work was not carried out. With respect to on-CPU decompression, usually the data layout is somewhat different, so they are not supported as such, and it would be an apples-to-oranges comparison; also, on-CPU decompression speeds are typically an order-of-magnitude slower than on the GPU).

4.2 Results and observations

Column	byte wi-dth	size (MB)	compres-sion scheme	co-mpr. ratio	TX time (ms)	decomp. b/w (GB/s)	decomp. time (ms)
`l_returnflag`	1	60	BITMAP	2.67	1.874	39.287	1.526
`l_linestatus`	1	60	BITMAP	4.00	1.249	34.232	1.752
`c_custkey`	4	6	MODEL	∞	0	325.520	0.018
`p_partkey`	4	8	MODEL	∞	0	312.903	0.025
`c_custkey`	4	6	DELTA	3.99	0.125	76.065	0.078
`o_orderkey`	4	60	DELTA	3.99	1.252	98.102	0.611
`p_partkey`	4	8	DELTA	3.99	0.166	78.052	0.102
`s_suppkey`	4	0.4	DELTA	3.99	0.008	40.064	0.009
`l_quantity`	8	479.9	DICT	8.00	4.998	209.261	2.293
`l_discount`	8	479.9	DICT	8.00	4.998	208.310	2.303
`p_size`	4	8	DICT	4.00	0.166	159.137	0.050
`s_nationkey`	4	0.4	DICT	4.00	0.008	51.440	0.007
`c_nationkey`	4	6	DICT	4.00	0.125	168.619	0.035
`l_linenumber`	4	239.9	DICT	4.00	4.998	178.493	1.344
`p_retailprice`	8	16	DICT	3.76	0.354	154.896	0.103
`l_linenumber`	4	239.9	NS	4.00	4.998	149.360	1.606
`l_discount`	8	479.9	NS	8.00	4.998	179.193	2.678
`l_quantity`	8	479.9	NS	4.00	9.997	166.839	2.876
`l_extendedprice`	8	479.9	NS	2.00	19.995	160.358	2.992
`ps_availqty`	4	32	NS	2.00	1.333	135.612	0.235
`l_shipdate`	4	239.9	FOR	2.00	10.017	160.653	1.493
`o_orderdate`	4	60	FOR	2.00	2.504	160.795	0.373
`l_orderkey`	4	239.9	FOR	2.00	10.017	144.370	10.017

Table 1: Timing results with TPC-H columns.

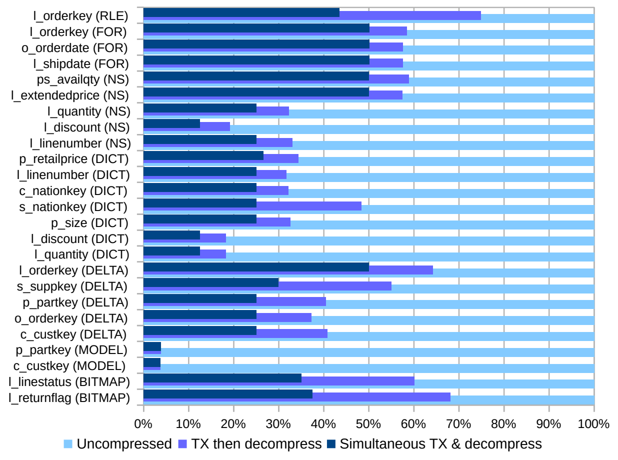


Figure 2: Benefits of using compression for copying a column to GPU memory (TPCH data)

Ontime column results – observations. As this work regards lightweight schemes, the yardstick for evaluating performance should be writing at the full memory bandwidth (336 GB/sec for the GM200 GPU used); normalize by this value when considering the bandwidth column in the results tables 1 and 2. One should also bear this in mind when comparing against On-CPU decompression, where the per-socket memory bandwidth is typically under 30 GB/sec per socket today.

Two of the columns fit a model function perfectly, and are thus compressed using the MODEL scheme, with an affine model function: $i \mapsto 1 \cdot i + 1$. In an actual DBMS, one might expect such a column never to be materialized at all, with accesses to it replaced by invocations of the function.

As predicted, the TPC-H data does not benefit from patched compression schemes (except perhaps for `l_orderkey`), nor from using NSV (as opposed to NS).

Decompression speed for shorter columns degrades — below several million elements performance gets lower. This is partly due to suboptimal launch configuration choice for small columns, but partly since a massively-parallel device needs more threads to mask the latency of memory reads.

The compression ratios achieved are sometimes lower than the potential optima, since some columns are best compressed to a number of bits which isn't a full number of bytes (as one can conclude from [21, §5]).

Column	byte wi-dth	size (MB)	compres-sion scheme	co-mpr. ratio	TX time (ms)	decomp. b/w (GB/s)	decomp. time (ms)
depdel15	1	59.3	PBITMAP	2.80	1.763	38.7	1.532
depdel15	1	59.3	BITMAP	2.67	1.853	36.2	1.636
originairportid	4	237.2	PDICT(1B)	3.93	5.029	174.5	1.359
originairportid	4	237.2	DICT(2B)	2.00	9.881	161.0	1.473
originairportseqid	4	237.2	PDICT(1B)	3.88	5.098	163.0	1.455
originairportseqid	4	237.2	DICT(2B)	2.00	9.881	160.8	1.475
origincitymarketid	4	237.2	PDICT(1B)	3.98	4.965	175.1	1.354
origincitymarketid	4	237.2	DICT(2B)	2.00	9.881	160.7	1.475
depdelay	2	118.6	PDICT(1B)	1.97	5.017	138.1	0.858
depdelayminutes	2	118.6	DICT(2B)	1.98	5.017	128.6	0.921
originstate	2	118.6	DICT(1B)	2.00	10.875	135.3	0.876
originstatefips	2	118.6	DICT(1B)	2.00	11.875	135.3	0.876
airlineid	4	237.2	RLE	354.48	14.875	91.4	2.594
year_	2	118.6	RLE	337.80	15.875	62.1	1.911
uniquecarrier	2	118.6	RLE	211.05	16.875	45.5	2.603
carrier	2	118.6	RLE	211.05	17.875	45.5	2.605
quarter	1	59.3	RLE	169.32	18.875	32.1	1.846
month_	1	59.3	RLE	169.26	19.875	32.1	1.848
dayofmonth	1	59.3	RLE	165.95	20.875	31.3	1.895
dayofweek	1	59.3	RLE	165.95	21.875	31.3	1.892
flightdate	4	237.2	RLE	162.42	22.875	120.2	1.974

Table 2: Timing results using USDT-Ontime columns

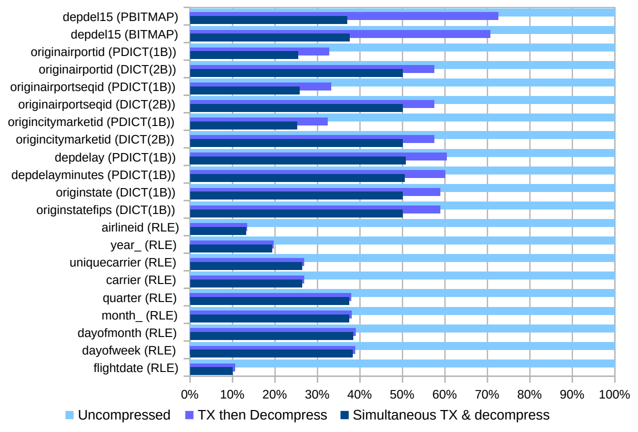


Figure 3: Benefits of using compression for copying a column to GPU memory (USDT-Ontime data)

Ontime column results – observations. The Ontime data has a multi-column key, as opposed to most TPC-H tables; these key columns exhibit long runs, which RLE compresses very well. Unfortunately, the RLE decompressor is not yet performance-optimized, so its use is not maximally beneficial. Most other columns have limited support within their domain — but not the lower valued-elements which would motivate the use of NS or NSV. This explains why many of the columns use DICT compression; and such behavior is typical of (fixed-length) string columns.

Another difference from TPC-H-like data is a non-uniform distribution over this support; for many columns it is concentrated enough, that one can reduce the dictionary size by a full byte and still capture the vast majority of values. For these columns, *patching* becomes relevant.

One of the patched columns, `depdel15`, differs from the others in that its patch data is actually rather large — bringing the compression ratio of down from 4 (BITMAP with 2 bitmaps) to only about 2.8: This is in fact a nullable boolean column, with 2.13% nulls, relatively spread out. With other columns a dictionary on the most frequent values can be constructed, so as to capture all but a tiny fraction of the frequency distribution's weight; in this case we have to swallow a large percentage of patches, not far from the limit of beneficiality: At compression ratio 2.67 it's already worth it to use 3 bitmaps. Glancing at the table, we notice that the PCIe transfer time is almost identical between the two schemes, and Figure 3 shows us the advantage is minimal. Still, the patch information itself is highly compressible, and with cascaded compression this would yield an additional improvement. This is an illustration how real-life data often admits a wide variety of compression scheme features and variations.

Other than `depdel15`, the other columns utilizing patching exhibit a similar behavior both with respect to the compression ratio (2x improvement) and to the decompression bandwidth: A noticeable, but modest, reduction.

5. DISCUSSION AND CONCLUSION

In this paper we have focused on decompressing full columns (or large parts thereof) into GPU memory. However, it is by now well-established that processing analytic queries quickly (using a CPU) involves avoiding column materializations when possible, especially before they have undergone initial filtering in the early stages of an execution plan. On a CPU, the two prominent approaches to doing this are *vectorization* [3] — materializing small chunks into a cache between query plan operations — and just-in-time *compiled* pipelined execution of multiple fused operators, where tuple data is passed through CPU registers [14]. Both techniques can be combined in a system, typically the variation in compression parameters that table scans need to deal with is better dealt with with vectorized decompression in scans, while the subsequent query pipeline can be JIT-compiled [11]. We expect both approaches to be relevant, for on-GPU query processing work, particularly w.r.t. decompression. Thus, in some cases data will only have to be decompressed into registers before it can be filtered or aggregated. In other cases — such as as cascaded compression schemes, perhaps — it may be decompressed into the SM block-shared memory, with blocks performing operations on decompressed chunks. In both these cases, our decompression implementations are likely to serve well enough, sometimes with higher bandwidth (being free of the bottleneck of having to write much more than they read into global GPU memory).

A third case is patching: A query plan compiler might schedule two different work paths for the underlying-scheme decompressed column, with no patches applied, and for the naive patch data, which can be thought of as a small column in sparse representation. Alternatively, it might opt to schedule a different decompressor, which applies its patches locally (and perhaps less efficiently). This direction points at the possibility of queries operating on the smallest byte-addressable compressed form of data directly. In case data were compressed using sub-byte bit-widths, partial decompression to the smallest whole byte-width could be used, that still falls short of decompressing to the full SQL type the schema demands (if the data distribution allows this – opportunities are detectable using MinMax block-wise statistics).

Overall, we believe compression is a highly relevant theme for database systems work that includes GPUs, and we hope our library can be of use for such research.

6. REFERENCES

[1] D. J. Abadi. *Query execution in column-oriented database systems.* PhD thesis, Massachusetts Institute of Technology, 2008.

[2] A. Agbaria, D. Minor, N. Peterfruend, O. Rosenberg, and E. Rozenberg. Overtaking cpu dbmses with a gpu in whole-query analytic processing. In *Proc. ADMS*, 2016.

[3] P. Boncz, M. Zukowski, and N. Nes. MonetDB/X100: Hyper-pipelining query execution. In *Proc. CIDR*, volume 5, pages 225–237, 2005.

[4] S. Breß, M. Heimel, N. Siegmund, L. Bellatreche, and G. Saake. GPU-accelerated database systems: Survey and open challenges. In *Proc. BigDataScience*. ACM/IEEE, 2014.

[5] P. Damme, D. Habich, J. Hildebrand, and W. Lehner. Lightweight data compression algorithms: An experimental survey. In *Proc. EDBT*, 2017. to appear.

[6] W. Fang and B. He. Personal communication.

[7] W. Fang, B. He, and Q. Luo. Database compression on graphics processors. *Proc. VLDB*, 3(1-2):670–680, 2010.

[8] S. Funasaka, K. Nakano, and Y. Ito. Light loss-less data compression, with GPU implementation. In *Proc. ICA3PP*, pages 281–294, 2016.

[9] B. He, N. K. Govindaraju, Q. Luo, and B. Smith. Efficient gather and scatter operations on graphics processors. In *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, page 46. ACM, 2007.

[10] B. He, M. Lu, K. Yang, R. Fang, N. K. Govindaraju, Q. Luo, and P. V. Sander. Relational query coprocessing on graphics processors. *Trans. DB Sys.*, 34(4):21:1–21:39, Dec. 2009.

[11] H. Lang, T. Mühlbauer, F. Funke, P. Boncz, T. Neumann, and A. Kemper. Data blocks: Hybrid OLTP and OLAP on compressed storage using both vectorization and compilation. In *Proc. SIGMOD*, pages 311–326, 2016.

[12] D. Lemire and L. Boytsov. Decoding billions of integers per second through vectorization. *Software: Practice and Experience*, 45(1):1–29, 2015.

[13] D. Lemire, L. Boytsov, O. Kaser, M. Caron, L. Dionne, M. Lemay, E. Kruus, A. Bedini, M. Petri, and R. B. Araujo. http://github.com/lemire/FastPFOR/.

[14] T. Neumann. Efficiently compiling efficient query plans for modern hardware. *Proc. VLDB*, 4(9):539–550, June 2011.

[15] P. Przymus and K. Kaczmarski. Compression planner for time series database with GPU support. *Trans. Large-Scale Data- and Knowledge-Centered Systems*, 15:36–63, 2014.

[16] E. Rozenberg. https://github.com/eyalroz/libgiddy.

[17] E. Rozenberg. https://bitbucket.org/eyalroz/db-kernel-testbench.

[18] E. Rozenberg. https://github.com/eyalroz/usdt-ontime-tools.

[19] E. Rozenberg. https://github.com/eyalroz/tpch-tools.

[20] B. Schwartz. https://www.percona.com/blog/2009/09/29/quick-comparison-of-myisam-infobright-and-monetdb/.

[21] The TPC Council. *TPC Benchmark H (rev 2.17.1)*, 2014. http://www.tpc.org/tpch.

[22] http://www.rita.dot.gov/bts/.

[23] M. Zukowski, S. Heman, N. Nes, and P. Boncz. Super-scalar RAM–CPU cache compression. In *Proc. ICDE*, pages 59–59. IEEE, 2006.