Fast Integer Compression using SIMD Instructions

Benjamin Schlegel
Technische Universität Dresden
Dresden, Germany
benjamin.schlegel@tu-dresden.de

Rainer Gemulla
IBM Almaden Research Center
San Jose, CA, USA
rgemull@us.ibm.com

Wolfgang Lehner
Technische Universität Dresden
Dresden, Germany
wolfgang.lehner@tu-dresden.de

ABSTRACT

We study algorithms for efficient compression and decompression of a sequence of integers on modern hardware. Our focus is on universal codes in which the codeword length is a monotonically non-decreasing function of the uncompressed integer value; such codes are widely used for compressing "small integers". In contrast to traditional integer compression, our algorithms make use of the SIMD capabilities of modern processors by encoding multiple integer values at once. More specifically, we provide SIMD versions of both null suppression and Elias gamma encoding. Our experiments show that these versions provide a speedup from 1.5x up to 6.7x for decompression, while maintaining a similar compression performance.

1. INTRODUCTION

The need for efficient integer compression arises in a variety of data processing tasks. Traditional database management systems use compression to reduce the I/O cost of reading data from disk [1, 9, 11, 12]. Some of the more recent database systems store almost all of the data in main memory [6, 10] and heavily rely on compression techniques that provide a good compression ratio and efficient decompression. Moreover, many of the popular serialization frameworks—including Google's protocol buffers and Apache's Avro—make use of integer compression for both data and metadata.

In all those applications, compression serves two purposes: It decreases the space required to store the data and it decreases the cost required to transfer the data (e.g., across the network, into the main memory, into the CPU cache). Thus, within a fixed amount of time, more data can be read or written when compression is used than when it were not used. In order to translate these savings in data transfer cost into faster data processing, the (de)compression algorithm (1) has to be able to keep up with the amount of data it receives, and (2) has to leave enough resources for the actual computation. For this reason, efficient (de)compression is key to good overall performance.

Proceedings of the Sixth International Workshop on Data Management on New Hardware (DaMoN 2010), June 7, 2010, Indianapolis, Indiana.
Copyright 2010 ACM 978-1-4503-0189-3 ...$10.00.

There are two different approaches to speed up compression algorithms. The first approach is to utilize pipelining in order to increase the instructions per cycle (IPC) of compression algorithms. For example, Zukowski et. al [15] propose a lightweight compression technique that avoids branches in performance-critical computations. As a consequence, the pipelining capabilities of modern processors are used more efficiently. The second approach is to use the SIMD capabilities—for Single Instruction, Multiple Data—of modern CPUs and to *parallelize* compression algorithms. Willhalm et. al [13] show how to improve the performance of bit unpacking—i.e., expanding fixed-length bit strings into machine word length—using SIMD instructions. Unfortunately, SIMD opportunities often cannot be automatically identified by compilers because, in most cases, the compression algorithms have to be modified to avoid issues with data organization and data alignment that plague a naïve parallelization.

In this paper, we describe how to exploit SIMD instructions to derive parallel versions of two well-known integer compression techniques: null suppression [11, 12] and Elias gamma encoding [3]. Both algorithms work by assigning short codewords to small integer values and long codewords to large integer values; no knowledge about the actual data distribution is required. In applications, such techniques are used to compress datasets that consist of mostly small integers. For example, null suppression is frequently used in database systems, both row stores [12] and column stores [1]. Elias gamma encoding is a popular choice for compressing sparse bitmaps such as bloom filters [2] and bitmap indexes [8], as well as integers in inverted indexes [14].

A major drawback of both null suppression and Elias gamma encoding, however, is that even highly-tuned sequential implementations are expensive and lead to a significant performance penalty. We show that by using SIMD instructions to compress or decompress multiple integers at once, these penalties can be reduced or completely avoided. Our algorithms have been designed explicitly for SIMD processing; they differ from the sequential algorithms in terms of in-memory data structures and layout of the compressed data. We refer to our algorithms as *k-wise null suppression* and *k-gamma encoding*, where k refers to the degree of parallelism.

We evaluated our compression techniques on three different processors: the Intel Core i7, the AMD Phenom, and the PPE of an IBM Cell Broadband Engine. Our experiments indicate that our SIMD algorithms have a compression ratio that is on par with or close to the sequential algorithms but allow for significantly faster decompression. In fact, our algorithms can have up to 6.7x the throughput of the corresponding sequential versions.

2. PREREQUISITES

The main observation exploited by universal compression of small integers is as follows: For all small integers, most of the space is used for storing leading zero bits. For example, consider the binary representation of the number 100 as a 32-bit integer value:

00000000 00000000 00000000 01100100.

There are 25 leading zero bits and only 7 bits that carry data.

In this paper, we are interested in compressing a *sequence* of small integers. If the maximum value in the sequence is known to never exceed 255, the sequence can be compressed by only storing the least-significant byte of each element. Knowledge of the exact distribution of the integer values leads to even better codes. If, however, neither an upper bound nor the distribution of integers can be provided, universal codes come to the rescue. They encode integers with variable-length codewords: Small values have short codewords, large values have long codewords. In the remainder of this section, we review two prominent examples of such codes: null suppression and Elias gamma. We focus on compression of 32-bit integers; our discussion naturally extends to longer integers.

2.1 Null Suppression

The idea behind null suppression is to substitute null values or successive zeros with a description of how many zeros are omitted. There are many variants of null suppression [11]; we focus on a variant proposed by Westmann [12]. Given an integer x, we can partition the binary representation of x into a (potentially empty) sequence of *leading zero bytes* and a (non-empty) sequence of *effective bytes*. For $x = 100$ as above, we have three leading zero bytes and one effective byte. Define the *effective byte length* of x as the number $l(x) = \lceil \log_{256}(x + 1)\rceil$ of its effective bytes.

The idea behind null suppression is to encode only the number of leading zero bytes and the effective bytes; leading zero bytes are omitted. For a 32-bit integer, the number of leading zero bytes is given by $4 - l(x)$ and lies in the interval $[0, 3]$. It can thus can be encoded using a 2-bit *compression mask*. For example, the encoded value of 100 is

$\underline{11}$ 01100100,

where $\underline{11}$ denotes the compression mask and 01100100 denotes the effective byte. The compression ratio of null suppression is quite low when compared to entropy encoding. Integers of effective byte length 1/2/3/4 are stored using 10/18/26/34 bits, respectively. At best, the compression ratio is therefore $\frac{10}{32} \approx 31\%$.

2.2 Elias Gamma Encoding

Elias gamma encoding [3] also partitions the binary representation of the integer of interest, but the partitioning is based on bits instead of bytes. Denote by $b(x) = \lceil \log_2(x + 1)\rceil$ the *effective bit length* of integer $x > 0$.[1] The compressed code consists of a *prefix* of $b(x) - 1$ zero bits followed by the sequence of *effective bits* of x. For example, 100 is encoded as

$\underline{000000}$ 1100100.

The length of the codeword is $2b(x)-1$ bits. When compared to null suppression, Elias gamma encoding provides a better compression ratio for small numbers (up to $\frac{1}{32} \approx 3\%$) but a worse compression ratio for large numbers.

2.3 SIMD Instructions Sets

In the following, we distinguish between *sequential algorithms* and *SIMD algorithms*. Sequential algorithms use only scalar instructions; SIMD algorithms also exploit the parallelism provided by the SIMD instructions sets found in modern processors. These instructions allow concurrent processing of k data values per instruction, where k depends on the processor and word length. Current processors provide 128-bit SIMD registers so that four 32-bit values (integers or floats) can be processed simultaneously. We expect future processors to provide larger SIMD registers. For example, Intel's upcoming AVX instruction set supports 256-bit SIMD registers [4].

The SIMD instructions exploited in our algorithms each belong to one of the following three classes: (1) load and store instructions, (2) element-wise instructions, and (3) horizontal instructions.

Load and store instructions copy data from main memory into a SIMD register and vice versa. Loading and storing is restricted to continuous chunks of main memory. Therefore, up to k load instructions and mask instructions[2] are required to load k values from non-continuous memory locations into a single SIMD register. All SIMD-capable processors provide load and store instructions for aligned memory access. Some processors also support unaligned access, although it is typically slower. If unsupported, unaligned load/store instruction can be emulated using two aligned load/store instructions.

Element-wise instructions perform operations on the elements of one or more SIMD registers. Instructions of interest include bit-shifting the elements in one SIMD register as well as combining the elements in multiple SIMD registers by applying a logical function.

Finally, *horizontal instructions* act across the elements of a SIMD register. The key horizontal instruction exploited in one of our algorithms is the byte permutation instruction (also called shuffle), which permutes the bytes of one (SSSE3 [5]) or two (Altivec [7]) input vectors according to a given permutation vector.

3. PARALLEL INTEGER COMPRESSION

In this section, we present SIMD versions of null suppression and Elias gamma encoding. We first discuss alternatives for data layout and then describe the algorithms in detail.

[1]Elias gamma cannot encode 0. If the data contains zeros, one may encode $x + 1$ instead of x.

[2]Some instructions sets (e.g., SSE4A, SSE4.1) provide insert and extract instructions for loading and storing single values of a SIMD register, in which case there is no need for mask instructions.

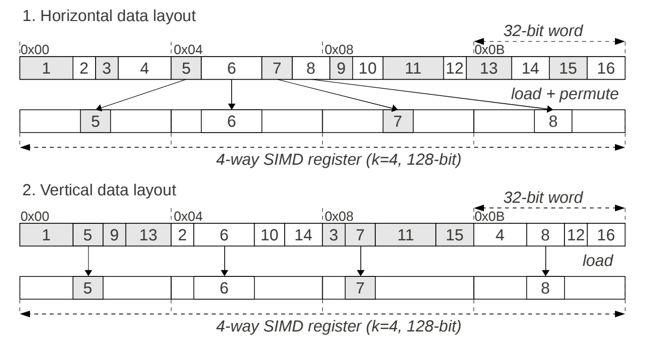


Figure 1: Horizontal and vertical data layout ($k = 4$)

3.1 Layout of Compressed Data

The layout of the compressed data plays an important role in our compression techniques. In contrast to uncompressed data, codewords have a variable length and thus require storage of length information. This poses challenges for parallel decompression (of k codewords). First, we have to make sure that all k codewords can be loaded efficiently into a single SIMD register. Second, the length information of the k codewords has to be organized in a way amenable to parallelization.

There are two alternative ways to address the first issue: a horizontal data layout and a vertical data layout, see Figure 1. For both layouts, we describe how to load k codewords from memory and distribute them across the elements of a single SIMD register for further processing. In a *horizontal* data layout, the k codewords are stored successively in memory. This facilitates loading (and storing) compressed data using a single unaligned load/store instruction, but distributing the k codewords into the k elements of the SIMD register requires data permutation. In contrast, in a *vertical* data layout, each of the k codewords is stored in a different memory word. Thus, after loading, the data is already distributed across the k elements of the SIMD register and data permutation is avoided. However, successive codewords may reside in non-successive memory locations so that up to k load/store instructions are required to access all k compressed codewords.[3]

Parallel decompression requires knowledge of the length information (effective byte length or effective bit length) of all k codewords. In a horizontal data layout, the length information of k codewords should be combined and stored before the k codewords themselves. Otherwise, parallel decompression is not possible because the location of the $i{+}1$th codeword is known only after the ith codeword has been decompressed. For a vertical data layout, the length information could be stored in front of each codeword. In general, however, it is advantageous to separate length information and codewords.

3.2 k-Wise Null Suppression

Recall that in null suppression, a 32-bit integer is compressed by storing both a 2-bit compression mask and the effective bytes of the binary representation of the integer. To parallelize this procedure, we are working on k integers

[3]Scatter/gather instructions may solve this problem, but none of the available processors supports these instructions.

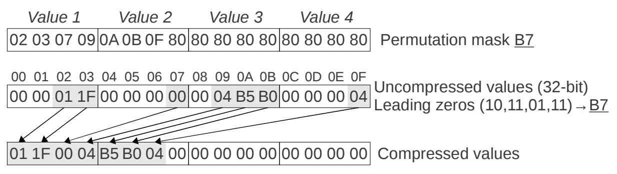


Figure 2: Compression using the byte-shuffle instruction ($k = 4$)

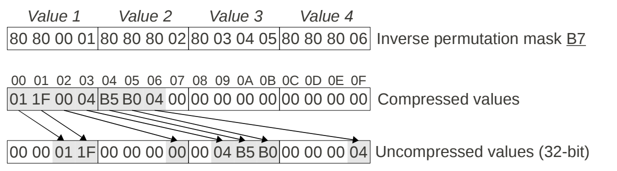


Figure 3: Decompression using the byte-shuffle instruction ($k = 4$)

at once. Suppose that the k integers are stored in a single SIMD register. Treating the register content as a single sequence of bytes, compression amounts to removing leading zero bytes, while decompression amounts to reinserting leading zero bytes. Our algorithm is based on the observation that both operations can be implemented with a horizontal byte permutation instruction.

Denote by $z_1, z_2, \ldots, z_k$ the number of leading zero bytes of each of the k integers, respectively, where $0 \le z_i \le 3$ for 32-bit integers and 128-bit SIMD registers.. There are four possible values for each z_i and thus 4^k possible cases for the location of the leading zero bytes in the entire SIMD register. The removal of these zero bytes can be seen as a permutation that moves the effective bytes to the front and the leading zero bytes to the end.

To efficiently perform this permutation, we make use of a SIMD byte permutation instruction. The instruction takes as input two SIMD registers: one containing the bytes to be permuted and one containing a permutation mask. Computation of the permutation mask is expensive, but since there are only 4^k different permutation masks of interest (256 for $k = 4$), we can precompute and store them in a *permutation lookup table.* The table is indexed by the *combined compression mask*, which is formed by concatenating the 2-bit binary representations of the z_i. To determine the total length of the compressed values, we make use of a separate *length lookup* table, which is indexed in the same way. The compressed data is stored using a horizontal data layout so that a single store instruction suffices for writing the compressed codewords to memory.

The compression process is illustrated in Figure 2 for four 32-bit integers (shown in a hexadecimal representation). Effective bytes are shaded gray. Note that the least-significant byte of the second integer is considered an effective byte—even though it is zero—because at most three leading zero bytes are removed. The combined compression mask B7 (hexadecimal) is given by concatenating the binary representations of $z_1 = 2$, $z_2 = 3$, $z_3 = 1$, and $z_4 = 3$. The corresponding permutation mask is shown at the top of Figure 2, and the result of the permutation is shown below the uncompressed values. Note that permutation of the zero bytes is not necessary because the permutation instruction (of almost all instruction sets) writes zero bytes into the result vector if the corresponding entry in the permutation mask is set to 80 (hexadecimal).

The entire compression algorithm consists of the following three steps: (1) Use k count-leading-zeros instructions on the k uncompressed values, then shift and combine the results to determine the combined compression mask. (2) Perform the corresponding permutation. (3) Store the combined compression mask followed by the compressed integers by using an unaligned store instruction. Decompression is performed in a similar fashion, but the permutation lookup table contains permutations that "reinsert" zero bytes. Figure 3 illustrates this process.

3.3 k-Gamma Encoding

Elias gamma requires a more substantial change of the algorithm in order to make parallelization effective. This is because neither a horizontal nor a vertical data layout can efficiently handle variable-length codewords preceded by their *individual* length information; see the discussion in Section 3.1. To fix this, we modify Elias gamma encoding by using *shared* length information for each block of k integers. Within a block, each value is therefore encoded using the same number of bits. We refer to this encoding as k-gamma encoding.

The shared codeword length, called *shared prefix*, can lead to either an increase or decrease of the compression ratio. To see this, set $k = 2$ and consider the following two blocks of integers: $(7, 1)$ and $(7, 6)$. The effective bit lengths of the integers are $(3, 1)$ and $(3, 3)$, respectively. Elias gamma encoding uses 6 bits for the first block and 10 bits for the second block. In contrast, 2-gamma encoding uses the same codeword length for each integer within a block; this codeword length is given by the maximum effective bit length of the integers in the block. In our example, both blocks have a codeword length of three bits. The shared prefix (2 bits) needs to be stored only once so that each block is encoded using 8 bits. The relative compression ratio of Elias gamma and k-gamma encoding thus depends on the data; see also Section 4.1.

Our algorithm is divided into two parts: (1) Obtain the shared length information and (2) construct the k codewords. The implementation of both parts can be done without any branches or loops; only parallel shift, load, and store instructions are necessary. The shared codeword length is determined using a scalar count-leading-zero instruction on the result of a logical OR of the k integers. The codewords are stored in a vertical layout, but the shared prefixes are stored in a separate memory area. This has the advantage that all memory accesses are aligned.

Each shared prefix consists of a (possibly empty) sequence of zero bits followed by a one bit; this bit serves as separator from the next shared prefix. In contrast to Elias gamma, usage of a separator bit allows us to encode 0. There are two different ways to do this. The first way is used by k-gamma encoding: The length of the shared prefix denotes the maximum number of effective bits of the k values. When k 0/1 values are encoded, the shared prefix and the k codewords together require $k + 1$ bits (recall that the 0 value has one effective bit). An alternative way, which we refer to as k-gamma$_0$ encoding, is to let the length of the zero bit sequence of the shared prefix denote the maximum number of effective bits of only the *non-zero* values. If all values are 0, only the separator bit has to be stored and the zero bit sequence of the shared prefix has length zero. Example codewords for k-gamma as well as k-gamma$_0$ encoding can be found in the appendix.

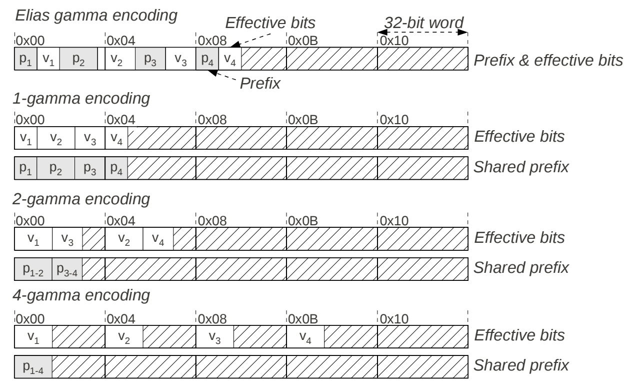


Figure 4: Memory representation of Elias gamma and k-gamma encoding of four example codewords

Figure 4 illustrates the memory layout of an example with 4 codewords for Elias gamma, 1-gamma, 2-gamma, and 4-gamma encoding. Elias gamma encoding—shown at the top of Figure 4—stores prefix and value of each codeword together. k-gamma encoding stores shared prefix and the values separately. While each value has its own prefix for 1-gamma encoding, two or four values share a prefix when 2-gamma or 4-gamma encoding, respectively, is used. Furthermore, each of the k values of a block starts at the same relative bit address in their memory word (e.g., v_3 and v_4 in words 0x00 and 0x04). In our actual implementation of k-gamma encoding, we split the memory into smaller chunks. The codewords grow forward from the start of each chunk and the shared prefixes grow backwards from the end of the chunk.[4]

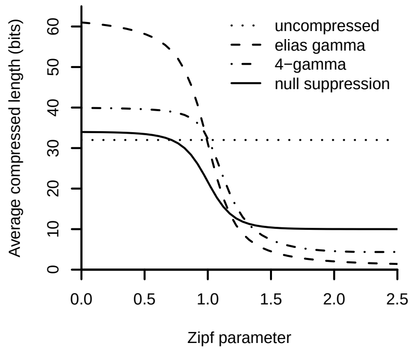
4. EXPERIMENTAL EVALUATION

We conducted a variety of experiments to gauge the performance and compression ratio of both the sequential and the SIMD versions of integer compression. We found that parallel processing is between 1.5x and 6.7x faster than sequential processing.

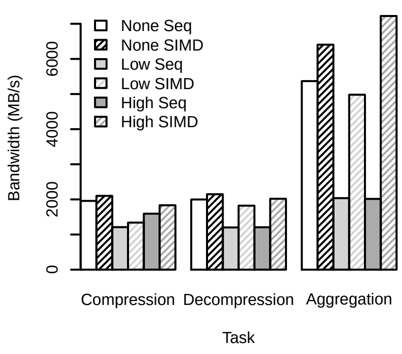
4.1 Test Setup

Experiments were conducted on a variety of different processors: the Intel Core i7-920 processor (2.67GHz), the AMD Phenom (2.8GHz), and the PPE of an IBM Cell Broadband Engine (3.2GHz). Linux was used as operating system. We implemented both sequential (Seq) and parallel (SIMD) versions of null suppression and gamma encoding. The algorithms were implemented in C and compiled using gcc, and intrinsics were used for SIMD instructions. To ensure fair comparison, all implementations are hand-tuned: Both the sequential and the SIMD versions make extensive use of the available instruction set for efficient processing. For null suppression (both versions), we manually unrolled loops to avoid data dependencies by storing 4 compression masks followed by 16 compressed integers.

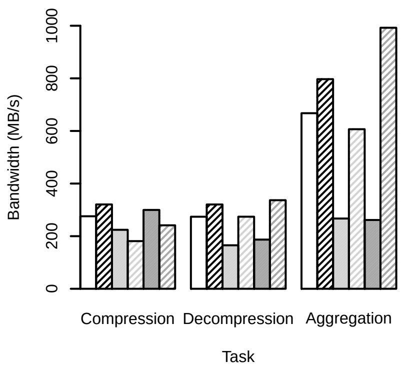
[4]Growth is reversed for processors using the big-endian format.



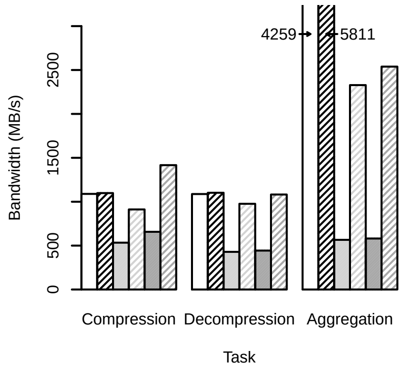
(a) Compression potential



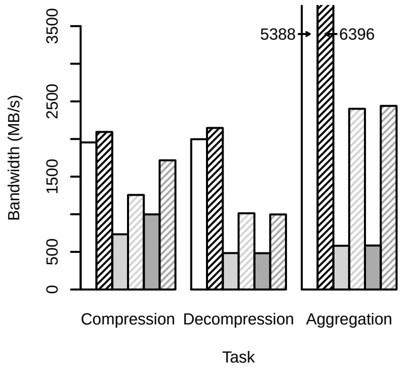
(b) Null suppression/Core i7



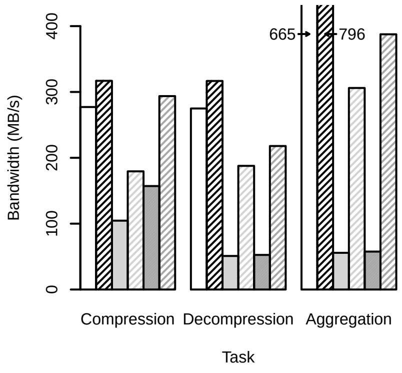
(c) Null suppression/Cell PPE



(d) Gamma encoding/Phenom



(e) Gamma encoding/Core i7



(f) Gamma encoding/Cell PPE

Figure 5: Experimental results

We generated synthetic datasets consisting of 32-bit integers in the interval $[1, 2^{32} - 1]$. The integers follow a Zipf distribution with Zipf parameter z. A parameter of $z = 0$ corresponds to a uniform distribution (no compression possible), while parameter values of $z > 1$ represent distributions that are heavily skewed towards small integers. Figure 5(a) illustrates the expected codeword length for the different compression algorithms and varying values of z. As can be seen, compression is beneficial for datasets that consist of sufficiently many small integers ($z > 0.75$). In this region, parallel Elias gamma has a slightly lower compression ratio than the sequential version. Our experiments indicate that this small loss in compression ratio is offset by large gains in decompression performance.

We generated two datasets of 32MB for each compression algorithm: a dataset with *low* compression potential and a dataset with *high* compression potential. We then measured the bandwidth—i.e., the amount of uncompressed data that can be processed per second—of three different tasks: (1) compression, (2) decompression, and (3) decompression with aggregation. The first and the second task involve loading uncompressed/compressed values from main memory and storing of compressed/uncompressed values back into main memory, respectively. The third task reads compressed values from main memory and sums them up, i.e., it does not put the uncompressed data back into main memory.

4.2 k-wise Null Suppression

As mentioned previously, we manually unrolled the loops of both versions of null suppression to avoid data dependencies. This is done as follows: the compression masks of b blocks of 4 integers are combined and stored consecutively up front. We measured no noticeable effect on the performance of the sequential algorithm, but the parallel algorithm performs best for $b = 4$ consecutive compression masks followed by 16 compressed integers. Thus we used $b = 4$ for all our experiments.

As a baseline for comparison, we measured the bandwidth of all three tasks when working on uncompressed integers. The compression and decompression tasks then reduce to a simple `memcpy` operation. We implemented a sequential `memcpy` operation, which was a few percent faster than the corresponding library function, as well as a parallel `memcpy` function using SIMD instructions. Similarly, the decompression and aggregation task reduces to an array sum. The sequential version of this task was the only function that was automatically parallelized by the gcc compiler; we thus explicitly deactivated the SIMD instruction sets to enforce sequential processing.

For null suppression, we set $z_{\text{low}} = 0.8$ and $z_{\text{high}} = 1.25$ for the datasets with low and high compression potential, respectively (cf. Figure 5(a)). The corresponding compression ratios are 95% and 38%.

Figure 5(b) shows the bandwidth in MB/s achieved on the Intel i7 processor. Each group of bars corresponds to one task, and each individual bar corresponds to a specific version of the algorithm ("Seq" or "SIMD") and a specific dataset ("low", "high", or "none" for the baseline). As can be seen, parallel compression is only slightly faster than sequential compression; this is due to the sequential count-

Table 1: Compression ratios

	z_{low}		z_{high}	
	unsorted	sorted	unsorted	sorted
Elias gamma	**61.6%**	61.6%	**8.7%**	8.7%
4-gamma	73.2%	**37.1%**	11.0%	6.1%
4-gamma$_0$	74.0%	37.3%	11.1%	**4.3%**

leading-zero instructions in the code. Parallel decompression is between 1.5x to 1.7x faster than sequential decompression. In this task, a lot of time is spend on storing the decompressed results into main memory. This overhead is avoided in the aggregation task, in which the parallel version reaches a speedup of up to 3.6x and is almost as fast (low compression ratio) or faster (high compression ratio) as working on uncompressed data.

The results for the Cell PPE are shown in Figure 5(c). Here, parallel compression is slightly less efficient than sequential compression so that, in practice, one should always use the sequential version. For decompression, the parallel version is 1.7x to 1.8x faster than the sequential version. Higher speed up is achieved for the aggregation task, where the parallel version is between 2.3x and 3.8x faster than the sequential version.

No experiments have been performed on the Phenom processor because it does not provide a SIMD byte permutation instruction.

4.3 Gamma Encoding

In the second set of experiments, we compared Elias gamma and 4-gamma encoding (both versions). Since gamma encoding has a different compression ratio than null suppression, we used different data distributions and set $z_{\text{low}} = 1.1$ and $z_{\text{high}} = 1.75$. For Elias gamma, which cannot encode zero values, we increase every value by one before encoding (not included in execution time measurement). This increase is not necessary for k-gamma encoding.

Table 1 lists the compression ratios of all three algorithms (the best compression ratio for each dataset is highlighted). Elias gamma achieves better compression ratios than 4-gamma encoding for the original datasets. If the datasets are sorted, however, 4-gamma encoding achieves better compression ratios than Elias gamma encoding. Clustering of similar values improves the compression ratio of k-gamma encoding because the shared prefix more closely matches the actual effective bit lengths. Comparing 4-gamma and 4-gamma$_0$, the former has slightly better compression ratios for unsorted datasets and datasets with low compression potential, while the latter achieves better compression ratios for sorted, highly compressible datasets.

Our bandwidth measurements have been performed on the unsorted datasets using the Elias gamma algorithm and the 4-gamma algorithm. The results for the Intel i7 processor are shown in Figure 5(e). Bitwise processing has a high impact on bandwidth. Depending on the dataset, sequential compression reaches at most half the `memcpy` baseline, while sequential decompression reaches at most a quarter of the `memcpy` bandwidth. Compression is faster because it requires only one instead of two store operations and less shift instructions. Compression with 4-gamma encoding is between 1.75x and 2.35x faster than the sequential version. Parallel decompression is about 2x faster. As before, the best bandwidth improvements are achieved in the aggregation task, where the parallel version reaches a speedup of 4.1x.

Figure 5(f) shows the results on the Cell PPE. Similar to the Intel i7 results, compression is almost always faster than decompression for the same reasons. In more detail, parallel compression is about 1.7x to 2.4x faster than sequential compression. Interestingly, for the high potential dataset, the compression task of 4-gamma encoding is about 1.2x faster than the compression task of null suppression (cf. Figure 5(c)) and is roughly as fast as the `memcpy` baseline. This is mainly caused by the reduced sequential fraction of the 4-gamma compression function; only one instead of four count-leading zero instructions are required. Nevertheless, decompression using 4-gamma encoding is slower than decompression using k-wise null suppression, but it is still 3.6x to 3.8x faster than decompression using sequential Elias gamma. Finally, the highest speedup is achieved for the aggregation task; 4-gamma encoding is 5.5x to 6.7x faster than Elias gamma.

The results for the Phenom processor are similar, see Figure 5(d). We achieved the following speedups with 4-gamma encoding: for the compression task 1.7x to 2.2x, for the decompression task 2.2x to 2.4x, and for the aggregation task 4.1x to 4.4x.

5. SUMMARY

We presented k-wise null suppression and k-gamma encoding, which are parallel integer compression techniques that utilize the SIMD capabilities of modern processors. Our experiments indicate that parallelization is highly effective. Compared to sequential versions, a speedup of 1.5x up to 6.7x is achieved. We expect further performance gains from wider SIMD registers, which will be supported in future processor generations. Today, our parallel versions are on par with uncompressed processing in many cases. This makes our algorithms attractive in practice, particularly for applications that require a large amounts of memory and a high memory bandwidth.

6. REFERENCES

[1] Abadi, D., Madden, S., and Ferreira, M. Integrating compression and execution in column-oriented database systems. In *SIGMOD* (New York, NY, USA, 2006), ACM, pp. 671–682.

[2] Cohen, S., and Matias, Y. Spectral bloom filters. In *SIGMOD* (2003), ACM, p. 252.

[3] Elias, P. Universal codeword sets and representations of the integers. *IEEE Trans. Inform. Theory* (1975), pp. 194 – 203.

[4] Intel Corporation. *Intel AVX: New Frontiers in Performance Improvements and Energy Efficiency*, March 2008.

[5] Intel Inc. *Intel 64 and IA-32 Architectures Software Developer's Manual*, December 2009.

[6] Legler, T., Lehner, W., and Ross, A. Data mining with the sap netweaver bi accelerator. In *VLDB* (2006), pp. 1059–1068.

[7] Motorola. *AltiVec Technology Programming Interface Manual*, 1999.

[8] Pagh, R., and Satti, S. R. Secondary indexing in one dimension: beyond b-trees and bitmap indexes. In *PODS* (2009), pp. 177–186.

[9] Raman, V., and Swart, G. How to wring a table dry: entropy compression of relations and querying of compressed relations. In *VLDB* (2006), VLDB Endowment, pp. 858–869.

[10] Raman, V., Swart, G., Qiao, L., Reiss, F., Dialani, V., Kossmann, D., Narang, I., and Sidle, R. Constant-time query processing. In *ICDE* (Washington, DC, USA, 2008), IEEE Computer Society, pp. 60–69.

[11] Roth, M. A., and Horn, S. J. V. Database compression. *SIGMOD Record 22*, 3 (1993), 31–39.

[12] Westmann, T., Kossmann, D., Helmer, S., and Moerkotte, G. The implementation and performance of compressed databases. *SIGMOD Rec. 29*, 3 (2000), 55–67.

[13] Willhalm, T., Popovici, N., Boshmaf, Y., Plattner, H., Zeier, A., and Schaffner, J. Simd-scan: Ultra fast in-memory table scan using on-chip vector processing units. *PVLDB 2*, 1 (2009), 385–394.

[14] Witten, I., Moffat, A., and Bell, T. *Managing gigabytes: compressing and indexing documents and images.* Morgan Kaufmann, 1999.

[15] Zukowski, M., Héman, S., Nes, N., and Boncz, P. A. Super-scalar ram-cpu cache compression. In *ICDE* (2006), p. 59.

APPENDIX

In this section, we give example codewords for k-gamma encoding and k-gamma$_0$ encoding for k=2. Table 2 shows input values x_1 and x_2, the corresponding codewords v_1 and v_2 and the shared prefix p_{1-2} (including the separator bit). Note the difference in encoding input blocks that consist of only zero-or-one values.

Table 2: Example codewords

(a) k-gamma

x_1	x_2	p_{1-2}	v_1	v_2
0	0	$\underline{1}$	0	0
1	0	$\underline{1}$	1	0
1	1	$\underline{1}$	1	1
2	0	$\underline{01}$	10	00
2	1	$\underline{01}$	10	01
2	2	$\underline{01}$	10	10
3	0	$\underline{01}$	11	00
3	1	$\underline{01}$	11	01
3	2	$\underline{01}$	11	10
3	3	$\underline{01}$	11	11
4	0	$\underline{001}$	100	000

(b) k-gamma$_0$

x_1	x_2	p_{1-2}	v_1	v_2
0	0	$\underline{1}$		
1	0	$\underline{01}$	1	0
1	1	$\underline{01}$	1	1
2	0	$\underline{001}$	10	00
2	1	$\underline{001}$	10	01
2	2	$\underline{001}$	10	10
3	0	$\underline{001}$	11	00
3	1	$\underline{001}$	11	01
3	2	$\underline{001}$	11	10
3	3	$\underline{001}$	11	11
4	0	$\underline{0001}$	100	000