Elias Revisited: Group Elias SIMD Coding

Andrew Trotman
University of Otago
Dunedin, New Zealand
andrew@cs.otago.ac.nz

Kat Lilly
University of Otago
Dunedin, New Zealand
kat.lilly@gmail.com

ABSTRACT

The prior belief that the Elias gamma and delta coding are slow because of the bit-wise manipulations is examined in the light of new CPU instructions that perform those manipulations. It is shown that despite using those instructions, Elias gamma and Elias delta remain slow compared to SIMD codecs such as QMX. We provide a theoretical basis on which to bit-wise encode data, and show that it is equivalent to SIMD extensions to Elias gamma that others have already introduced. Extending this we introduce a new SIMD Elias delta variant. Experiments comparing these two codecs to QMX on public data, and in the JASSv2 search engine show that although the index is slightly larger than QMX, search throughput is increased and latency is decreased.

CCS CONCEPTS

• **Information systems → Search index compression**;

KEYWORDS

Integer Compression, Elias Compression, Index Compression

1 INTRODUCTION

Index compression has been an area of investigation in Information Retrieval for many decades. By increasing the compression ratio (effectiveness) the index of more documents can be stored in the same space – resulting in a cost saving. Increased decompression speed (efficiency)[1] decreases the CPU costs, and also has a positive effect on the user experience as a consequence of reduced latency.

Early index compression codecs focused primarily on index size as there was a belief that a smaller index would be faster. Since the index was stored on moving parts disks, the index read time would dominate any decoding time. At that time the Elias gamma code [8], Elias delta code [8], Golomb code [9] and Rice code [17] were popular. These codecs are still discussed today in Information Retrieval textbooks [4], and implementation of these Elias codes is included in the Terrier [14] and ATIRE [24] search engines.

Trotman [22] later demonstrated that variable byte encoding was substantially faster at decoding than either the Elias gamma code or the Elias delta code. He modelled a moving parts hard drive as well as decoding latency, and showed that even when disk latency was accounted for, variable byte encoding was the clear winner. Scholer *et al.* [19] demonstrated that within the context of a search engine, use of the less effective variable byte encoding resulted in a latency decrease due to decoding efficiency.

The work of Trotman and of Scholer *et al.* lead to a fundamental change in compression of postings lists. Beforehand the most important factor was space, afterhand the most important factor was decoding speed (but effectiveness remained important). Subsequent to their work others introduced a multitude of codecs including Simple-9 [1], PForDelta [28], and so on.

The most recent work on postings list compression is far more integrated than prior work. In the context of a Document-at-a-Time (DaaT) search engine, Partitioned Elias-Fano [15] codes are used as they allow the compressed list to be searched without being decoded – something essential for use with the WAND [3] or BlockMaxWAND [7] algorithms commonly employed in such search engines. For Score-at-a-Time search engines (SaaT), linear decompression speed is important as postings are decoded in blocks. Partitioned Elias-Fano has proved inefficient at linear decoding [16], but QMX [23] has proven to be well adapted to this purpose because of the use of SIMD instructions while decoding. Lin & Trotman [13] experimented with not using compression for SaaT search and show a small increase in performance at the cost of a large increase in index size.

It is now a commonly held belief that the bit-wise codecs of yesteryear, for example the Elias codes, are "moderately expensive" to decode because there is "lots of bit twiddling" [21].[2] In this investigation we directly challenge this belief. Others have already shown that Elias gamma does not require program-level bit twiddling due to the introduction of CPU instructions that directly perform the bit manipulations [27], and we show that *neither does Elias delta*. Others have previously introduced group Elias gamma [18] and we introduce *Group Elias Delta SIMD*, and show that both codecs are highly efficient at decoding.

2 POSTINGS LISTS

The inverted index data structure commonly used in search engines builds, for each unique token in the document collection, a list of which documents contain that token along with the number of times that token in seen in that document (the term frequency, tf). Each document is given a unique numerical identifier (the *docid*), which starts with 1 for the first document and increments with each document being indexed. These $< docid, tf >$ integer pairs

[1] Throughout this contribution we use 'effective' to discuss compression ratio and 'efficient' to discuss decoding latency.

ADCS '18, December 11–12, 2018, Dunedin, New Zealand

ACM ISBN 978-1-4503-6549-9/18/12…$15.00
https://doi.org/10.1145/3291992.3292001

[2] See slide 29 of https://www.slideshare.net/CraigMacdonald/efficient-query-processing-infrastructures

are called *postings* and the lists are called *postings lists*, the docids for a given token forming a strictly increasing sequence.

It is common to store the difference between consecutive docids rather than the ids themselves as the differences (known as *d-gaps*) are smaller than the docids and further compress very well.[3] When SIMD instructions are used for decoding, various different d-gapping techniques have been used, with a preference being for computing the difference between docids that are machine word distance apart (such as 4 when SSE 128-bit registers are used) because decoding multiple d-gaps can be done with a single instruction. However there is an effectiveness loss due to using other than consecutive integers (D1 d-gaps).

Postings lists stored in increasing docid order are known as docid-ordered and there are several Document-at-a-Time (DaaT) processing strategies such as WAND and BlockMaxWAND that employ skipping to efficiently merge these lists. Codecs such as Partitioned Elias-Fano are well suited to DaaT processing because they allow skipping without decoding.

Many first-round ranking functions (such as BM25 and Language Models with Dirichlet smoothing) can be described as the sum, over each token, t, in the query, q, of some function of the token statistics, the document, d, statistics, and the collection, c, statistics: $\sum_{t \in q} f(t, d, c)$.

All except the sum can be computed at indexing time – that is, the *impact* of the term in the document given the collection is known in advance of any searching. The principal of impacted indexes is to compute this impact at indexing time and to store it in the index in place of the term frequency. However, these impact scores are typically not integers, a problem that is overcome if the impact scores are bucketed. Prior work suggests that a small number of buckets (for example, 255 for .gov2, but larger for larger collections) is often enough to avoid any significant loss of results quality [5].

Impact-ordered indexes store the postings lists sorted on decreasing impact rather than increasing docid. More precisely, store $< i_h, < d_{h,1}, ...d_{h,h} >> \ldots < i_l, < d_{l,1} ... d_{l,h} >>$ where i_h is the highest impact, $d_{h,l}$ is the lowest docid for the highest impact, i_l is the lowest impact, $d_{l,h}$ is the highest docid for the lowest impact, and likewise for $d_{h,h}$ and $d_{l,l}$. One $< i, < d_l, ...d_h >>$ chunk is known as a *segment*.

The advantage of storing docids in increasing order is that d-gap encoding can be used. The advantage of storing from highest to lowest impact is that highest impact docids can be processed first. The advantage to bringing all documents with the same impact together is a reduction in the number of integers stored in the index (and hence a smaller index).

When processing queries Score-at-a-Time (SaaT), all the segments for all the terms in the query are ordered from highest to lowest impact score. The highest impact segment is pulled from this sorted list, decoded, and processed in full. The process is continued until all segments have been processed or an early termination criterion has been met.

SaaT processing takes advantage of the CPUs ability to pre-fetch encoded postings from memory to cache, and of fast linear decoding of compressed docids. Encoding schemes such as Partitioned Elias-Fano are not efficient at linear decoding, but SIMD codecs are because they can decode many consecutive integers in a small number of instructions.

We are interested in fast linear decoding of compressed d-gaps within the context of an impact-ordered search engine using the SaaT processing strategy known as *anytime* [12]. This investigation examines the use of Elias codes for this purpose, and compares to variable byte encoding, because it is popular; and to QMX because it has proven to be efficient in SaaT search [13].

3 PRIOR WORK

3.1 Variable Byte Encoding

Variable byte encoding is not a single encoding, but a class of encodings that all follow the principal that an integer should be stored in the minimum possible number of whole bytes [23].

The particular variant used herein takes a single binary integer and stores it in 7-bits with the high bit of each byte being used as a stop bit (a 0 indicates continuation and a 1 indicates stop). The bytes are written to memory from most significant to least significant.

$$1905_{10} = 11101110001_2 \rightarrow [0]0001110[1]1110001$$

Figure 1: Variable byte encoding of 1905_{10} (11101110001_2) when stored big-endian and using the high bit as a stop-bit.

Figure 1 illustrates this process. It shows the integer 1905_{10} (11101110001_2) being broken into 2×7-bit chunks (0001110_2 and 1110001_2) and then the high bit of the first byte being set to 0 to indicate continuation and the high bit of the second byte being set to 1 to indicate stop.

Decoding is straightforward and fast. Initialise a sum to 0, read a byte, if the high bit is 0 then shift the sum left by 7 and add the byte to it, moving on to the next byte. If the high bit is 1 then turn that bit off, add it to the sum, then return the sum. In fast implementations, such as that of Zhang *et al.* [26], the decoding is done with an unwound loop. We employ this optimisation.

The number of bits needed to store an integer, n, using variable byte encoding, b_v, is: $\lceil \lceil \log_2(n+1) \rceil / 7 \rceil \times 8$.

3.2 Unary

The unary encoding encodes integers one at a time. First an arbitrary symbol is chosen, then that symbol is repeated n times for the integer n. When encoding multiple integers in a stream it is convenient to encode the last (or first) symbol differently from the others. One such convention is to use $n-1$ zeros followed by a 1 (the stop bit). In this way, 6_{10} would be represented as 000001_1.

Decoding a stream of integers from a unary coded bit-stream is performed by counting zeros until the stop bit is found, then returning the number of symbols read (including the stop bit).

The number of bits necessary to store an integer, n, using unary, b_u, is: n.

3.3 Elias gamma code

The Elias gamma code encodes integers one at a time. Each integer is encoded in two parts (called snips): the base-2 magnitude of the integer, and the integer itself. The magnitude is stored in unary while the integer is stored in binary. To encode an integer, first compute the magnitude, $M = \lfloor \log_2(x) \rfloor$, write out M zeros, then

[3] This technique is not used in Partitioned Elias-Fano encoding.

write out the integer in binary. For example, the magnitude of the integer 6_{10}, is $M = 2$, the binary is 110_2, and so the Elias gamma encoding of the magnitude snip is $[00_1]$, the binary snip is $[110_2]$, giving a gamma encoding of 00110_2.

To decode, count the number of leading 0s, add 1, then read that number of binary digits. So for 00110_2, read two zeros then reconstitute the integer by reading three binary digits.

The number of bits necessary to store an integer, n, using the gamma code, b_g, is: $2 \times \lfloor \log_2(n) \rfloor + 1$.

3.4 Elias delta code

The delta code is designed to be space efficient with larger integers. This is achieved by storing the magnitude (strictly, one more than the magnitude) of the integer using the gamma code rather than unary, then storing the integer in binary (with the high bit suppressed). For example, the magnitude of the integer 6_{10}, is $M = 2$, the binary is 110_2. So first write out $M + 1 = 3$ using the gamma code $[011_2]$, followed by the binary of 6_{10}, but with the high bit suppressed $[10_2]$ giving 01110_2.

To decode delta encoded integers first gamma decode the magnitude then read that number of bits, then turn on the suppressed high bit. But more directly, read the number of leading zeros, then one more than that binary bits (call this h), then read $h - 1$ binary digits to get the number with the high bit suppressed, then add 2^{h-1} to get the decoded integer. So for 01110_2, read $M_m = 1$ zeros, then $M_m + 1 = 2$ binary digits (giving 3), then $M_m - 1 = 2$ binary digits (giving 10_2), then add $2^{M_m - 1}$ (giving 100_2), resulting in $110_2 = 6_{10}$.

The number of bits necessary to store an integer, n, using the delta code, b_d, is: $\lfloor \log_2(n) \rfloor + 2 \times \lfloor \log_2(\lfloor \log_2(n) \rfloor + 1) \rfloor + 1$.

3.5 Notes on the Gamma and Delta Codes

Table 1 gives the variable byte, unary, gamma, and delta encodings of the first 32 integers. For integers less than 16, the gamma code is the most effective. For numbers larger than 16, variable byte encoding is most effective.

It should also be noted that there is no unary, gamma, or delta encoding of zero and no encoding of negative numbers for these codecs (as described). Within the context of a search engine this is inconsequential because docids can be assigned from 1 rather than 0, and so 0 or less cannot occur if d-gaps are being used.

3.6 QMX

A generational change in decoding efficiency was seen with the introduction of SIMD-based codecs. Several have been proposed including varint-G8IU [20], SIMD-BP128 [11], TurboPFor [25], TurboPackV [25], and QMX [23]. Prior work has shown that these last two do not substantially differ in decoding time, but that they are both more efficient that the others, which in turn show a substantial improvement over not using SIMD instructions [23]. SIMD codecs are efficient decoders because they utilise the parallel nature of SIMD instructions to extract more than one integer at a time.

Figure 2 illustrates QMX. A 4-bit selector and a 4-bit selector run-length are fused to form a single 8-bit word which is stored at the end of the encoding. Payloads are stored starting at the beginning of the encoding. Each payload stores some number of integers all of which are stored using the same bit width specified by the selector. There are heuristic rules that sometimes double the size of the payload and sometimes a payload isn't needed, however the general principal is fixed-width bin-packing of integers into SIMD words and striping those integers across each of the 32-bit elements in an Intel SSE4 SIMD 128-bit word.

Decoding involves ANDing with a machine word in order to extract 4 integers (1 instruction), then right-shifting the word to prepare for the next extraction (1 instruction). The number of bits necessary to store an integer, n, using QMX, b_q, is dependant on other integers in the stream being compressed. The worst case for a 32-bit integer is: $b_q = 32$, but this is unlikely to occur in a search engine index.

With the introduction of SIMD decoders, it has become possible to count integers decoded per instruction rather than instructions to decode an integer – which is why it is a generational improvement on prior work. Prior work has examined the extension of "light weight" SIMD codecs (i.e. SIMD-BP) from 128 to 512 bit registers showing improvements [10]. It is not obvious how to extend QMX in such a way.

4 ELIAS IMPLEMENTATIONS

4.1 Bitwise Implementations

Implementation of the two Elias codecs are seen in several open source search engines. Terrier [14] (written in the Java) includes an implementation of both the gamma code and the delta code. The gamma code reads the unary magnitude byte-at-a-time and if that byte is non-zero it performs a table lookup to find which bit is set. The binary decoding is also performed byte at a time with masking operations. The delta code calls the gamma code implementation to extract the magnitude then calls the binary extractor to get the integer.

ATIRE [24] (written in C++) also includes an implementation of both the gamma code and the delta code. Both are reliant on a long bit-stream library that keeps track of how many bits have been consumed. To unary decode, bits are consumed one by one until a 1 bit is found. To binary decode, bits are extracted (one by one) to give the decoded value. The delta code calls the gamma code implementation to extract the magnitude then bit by bit extracts the integer.

It is reasonable to assume that the approach taken by Terrier is more efficient that that taken by ATIRE, but it is not clear that this is the case as there is a greater overhead involved. We do not investigate this further as neither are SIMD and so both can be assumed to be slow relative to an SIMD codec.

4.2 Non-Bitwise Implementation

In 2013 Intel introduced Haswell CPUs that included the (then) new BMI1 and BMI2 bit manipulation instruction subsets. Three instructions from BMI1 are of particular interest for the implementation of decoders for the gamma and delta codes: `TZCNT`, `LZCNT`, and `BEXTR`. The first instruction, `TZCNT`, counts the number of trailing zeros in an integer. `LZCNT` counts the number of leading zeros in an integer. `BEXTR` extracts the given number of bits from the given location in an integer and right aligns them.

A gamma decoder can be implemented with these instructions. First count the number of leading zeros (`LZCNT`), then extract that

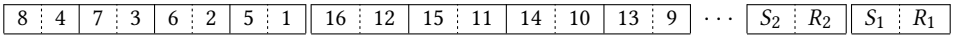

Figure 2: The QMX encoding first stores the 4 × 32-bit payloads then the selector / run-length pairs afterwards and in reverse. The image shows 8 × 16-bit integers per payload striped across the 32-bit integers with the placement being in the order given. The second 128-bit word storing integers 9 through 16. The dotted line shows the separation between parts of a word (16-bits of a 32-bit word of a 128-bit payload, and 4-bits of an 8-bit selector run-length pair).

Table 1: Variable byte, unary, Elias gamma, and Elias delta encoding of the first 32 integers, along with the size (in bits) of the encoding. QMX is not included because the encoding is dependant on other integers in the sequence. Zero cannot be encoded in Unary or the Elias codes. Numbers less than 16 are best encoded using Elias gamma, numbers larger than 16 are best encoded using Variable Byte.

Integer	Variable Byte	b_v	Unary	b_u	Elias Gamma	b_g	Elias Delta	b_d
1	00000001	8	1	1	1	1	1	1
2	00000010	8	01	2	010	3	0100	4
3	00000011	8	001	3	011	3	0101	4
4	00000100	8	0001	4	00100	5	01100	5
5	00000101	8	00001	5	00101	5	01101	5
6	00000110	8	000001	6	00110	5	01110	5
7	00000111	8	0000001	7	00111	5	01111	5
8	00001000	8	00000001	8	0001000	7	00100000	8
9	00001001	8	000000001	9	0001001	7	00100001	8
10	00001010	8	0000000001	10	0001010	7	00100010	8
11	00001011	8	00000000001	11	0001011	7	00100011	8
12	00001100	8	000000000001	12	0001100	7	00100100	8
13	00001101	8	0000000000001	13	0001101	7	00100101	8
14	00001110	8	00000000000001	14	0001110	7	00100110	8
15	00001111	8	000000000000001	15	0001111	7	00100111	8
16	00010000	8	0000000000000001	16	000010000	9	001010000	9
17	00010001	8	00000000000000001	17	000010001	9	001010001	9
18	00010010	8	000000000000000001	18	000010010	9	001010010	9
19	00010011	8	0000000000000000001	19	000010011	9	001010011	9
20	00010100	8	00000000000000000001	20	000010100	9	001010100	9
21	00010101	8	000000000000000000001	21	000010101	9	001010101	9
22	00010110	8	0000000000000000000001	22	000010110	9	001010110	9
23	00010111	8	00000000000000000000001	23	000010111	9	001010111	9
24	00011000	8	000000000000000000000001	24	000011000	9	001011000	9
25	00011001	8	0000000000000000000000001	25	000011001	9	001011001	9
26	00011010	8	00000000000000000000000001	26	000011010	9	001011010	9
27	00011011	8	000000000000000000000000001	27	000011011	9	001011011	9
28	00011100	8	0000000000000000000000000001	28	000011100	9	001011100	9
29	00011101	8	00000000000000000000000000001	29	000011101	9	001011101	9
30	00011110	8	000000000000000000000000000001	30	000011110	9	001011110	9
31	00011111	8	0000000000000000000000000000001	31	000011111	9	001011111	9
32	00100000	8	00000000000000000000000000000001	32	00000100000	11	0011000000	10

number plus one bits from that position in the integer (`BEXTR`). If integers are restricted to 32-bits and byte-aligned then one 64-bit read is guaranteed to contain an entire gamma encoded 32-bit integer, which can be decoded using `LZCNT` then `BEXTR`.

Byte aligning integers is ineffective as there are many wasted bits between integers – especially if those integers are small: whereas the gamma code can encode a 1 in one bit, byte aligning would result in 1 bit being used and 7 bits being wasted. In practice a long bit-string is needed and a pointer into that bit-string must be kept. In order to avoid wastage at the end of the long bit-string, it is best to pack from the low end to the high end of a machine word (because words are stored in memory little-endian). However, this introduces a problem.

The gamma code relies on the unary being stored in the high bits followed by the integer in binary because the unary stop-bit is shared with the high bit of the binary integer. By moving the unary to the low end of the encoding, the stop-bit is no longer shared. For example, encoding 6_{10} using the gamma code results in $[00_2][110_2]$, but by storing it as the low bits results in $[110_2][00_2]$ which incorrectly unary decodes because the low bit of the binary encoding is zero.

Shifting the high bit to the low bit is a standard technique known as zigzag encoding, and is used in Google Protocol Buffers to encode negative numbers.[4] It is a form of left shift known as a circular shift (or rotate without carry). For Elias coding, the difference is that the rotation is not through the entire length of the machine word, it is up-to and including the highest set bit.

Figure 3 illustrates this process. The value 10000110_2 when shifted left gives 00001100_2 as the bottom bit is shifted in from the carry, when rotated left gives 00001101_2 as the top bit is shifted to the low bit. The difference we employ is that we rotate around the highest set bit rather than the highest bit. For example, when 110_2 is zigzagged it results in 101_2 as leading zeros are ignored. Decoding of a zigzagged integer whose magnitude is known can be done with a shift right and a bit set.

Zigzag encoding guarantees the low bit will be set, and so the unary decoding will work correctly (6_{10} is gamma encoded as $[101_2][00_2]$. Decoding a single gamma encoded integer from a 64-bit word involves counting trailing zeros with `TZCNT`, then extraction of the integer with `BEXTR`, then decoding the zigzag with a shift-right and a bit set. To prepare for the next extraction a bit-shift to

[4] https://developers.google.com/protocol-buffers/docs/encoding

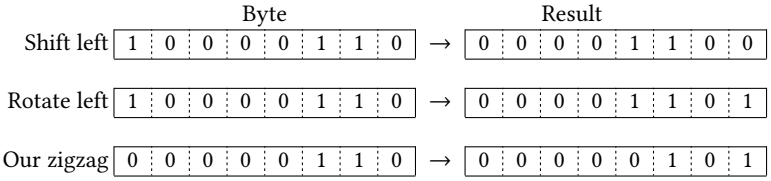

Figure 3: The difference between shift left, rotate left, and our zigzag left (assuming a 0 carry). Our zig-zag rotates around the highest set bit. This moves the most significant set bit to the least significant bit and moves all other bits one to the left.

Table 2: The partitions of 5 and the permutations of those partitions. There are 7 partitions and 16 compositions of 5.

Partition	Permutations of Partition (Compositions)
5	5
4+1	4+1, 1+4
3+2	3+2, 2+3
3+1+1	3+1+1, 1+3+1, 1+1+3
2+2+1	2+2+1, 2+1+2, 1+2+2
2+1+1+1	2+1+1+1, 1+2+1+1, 1+1+2+1, 1+1+1+2
1+1+1+1+1	1+1+1+1+1

re-align to the next integer is need, then some management for the next read from memory.

5 GROUP ELIAS SIMD

5.1 Combinatorics

Starting from first principles, we ask how many ways there are to non-uniformly pack integers into a 32-bit machine word (without loss of generality, and ignoring decoding). This question is exactly equivalent to asking how many ways there are to make 32 by adding natural numbers. The 32 being the machine word size that we wish to pack into, and the natural numbers being the different bit-widths (magnitudes) of integers being packed. This is a well known problem in combinatorics and is known as the number of *partitions* of an integer. Imagining a 5-bit word, Table 2 column 1 gives the 7 partitions.

So a 5-bit word could fit a 4-bit integer and a 1-bit integer, or a 3-bit integer and a 2-bit integer, and so on. There is no simple formula for computing the number of partitions of an integer, but there are 8,349 partitions of 32.

The integers being packed into the machine word can come in any order, so the number of sequences that might be seen is the sum of the number of *permutations* of each partition of the width of the machine word. For example, with a 5-bit word we might see any or all of $3+1+1$, $1+3+1$ or $1+1+3$. For the 5-bit word there are 16 possible sequences not just the 7 partitions (see Table 2).

The number of permutations of all partitions of an integer, n, is known as the number of *compositions* of the integer n. There are 2^{n-1} compositions of n. The proof is well known[5], but informally reproduced here as it relates to Elias's encodings:

Write the number being composed, n, as a sequence of n 1s (in unary). For example for 5, write

11111

between each 1, either write a ',' or a '+', for example,

$1+1, 1, 1+1$

re-write this equation as a composition by computing the results of the sums, in this case giving:

2, 1, 2.

Since there are $n-1$ binary decisions to be made ('+' or ',' between each digit) there are 2^{n-1} compositions. For $n = 32$, there are $2^{31} = 2,147,483,648$.

There is an obvious numbering of each composition. By replacing '+' with 0 and ',' with 1, each of the compositions of $n = 32$ can be uniquely identified by a 31-bit integer. For convenience, we store a 1 in the high bit and use a 32-bit integer. As an example, using 5-bit integers, the bit sequence 10110_2 would number the composition $1+1, 1, 1+1$, or 2, 1, 2.

We observe that the bit sequence given from the composition number can be used as the unary encoding of the magnitude of integers being packed into words. That is, the 5-bit selector 10110_2 would describe 3 integers being packed into a 5-bit word where the width of the first integer is 2, the second integer is 1, and the third integer is 2. For example, the integer sequence 2, 1, 3 could be encoded with the composition number first $[10110_2]$ then the integers second $[10_2][1_2][11_2]$, giving 1011010111_2. To decode, extract the three bit widths first, then extract the three integers.

Our derivation from first principles has resulted in an encoding that is equivalent to the Elias gamma coding, except that magnitudes have been pulled together and stored first (in unary) then a payload containing the integers (in binary) is stored second. For this reason we call this encoding group Elias gamma (after group-varint [6]) and observe that it is referred to as k-Gamma by Schlegel [18] and generalised to Group-Scheme with a 1-bit granularity by Zhao et al. [27]. Our proof is that in this encoding the selector is the same size as the payload, and there are no wasted bits between the codewords as each is encoded in the smallest possible number of bits.

We now introduce two improvements. In the first, likewise to others before us [18, 27], we use SIMD (in our case, AVX-512 SIMD 16×32-bit) payloads. In the second the unary composition numbers are replaced by the Elias gamma coding of the unary.

5.2 Group Elias Gamma SIMD

Figure 4 shows the layout used in Group Elias Gamma SIMD. First the 512-bit payload is broken into 16×32-bit integers that we call *rows*. Then 16 integers are read from the input stream and the base-2 magnitude of the largest is computed. This magnitude, M, is written in unary to the selector. A *column* of M bits is allocated from each row, and the integers are striped across those columns. The process is continued until the payload is full. First the 32-bit selector is written then the 512-bit payload is written. In the case where there are 'spare' bits in the payload, the integer sequence is

[5] See the Wikipedia article: "Composition (combinatorics)"

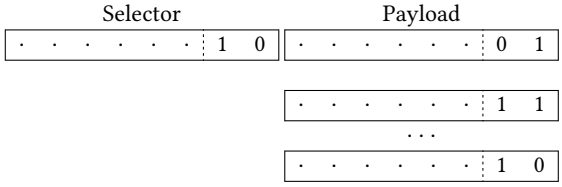

Figure 4: Group Elias Gamma SIMD breaks the 512-bit payload into 16×32-bit words called *rows* and then allocates fixed-width *columns* from those rows for storing the integers (in this case 1_{10}, 3_{10}, ... 2_{10}). The width is stored in the selector which is encoded in unary from least significant bit to most significant bit (in this case 10_2).

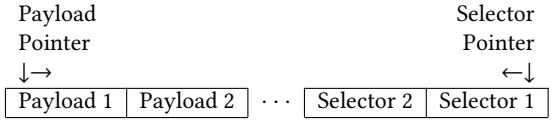

Figure 5: Group Elias delta SIMD encoding stores all the 512-bit payloads before the selectors and uses two pointers to keep track of decoding progress.

split with the high bits being stored in the first payload and the low bits being carried over to the next payload (and the low bits are stored in the selector, with the high bits being carried over to the next selector). It is always possible to store selectors and payloads interlaced because each payload is one composition of 32 and its composition number fits in one 32-bit integer.

When decoding, the width of the columns is extracted from the composition number (the selector) using `TZCNT`, then that number of bits is extracted from each row of the payload using the `VPANDD` instruction. Both are then shifted right so that the decoder can move on to the next column.

An integer sequence encoded using group Elias gamma SIMD is expected to be larger than the gamma encoding as there can be wasted bits between encoded integers (because the column width is the same for each row). Decoding, however, is expected to be vastly more efficient.

5.3 Group Elias Delta SIMD

Zhao et al. [27] experimented with using binary or unary selector lengths. We differ from previous work in that in group Elias delta SIMD the length selector is gamma encoded.

Figure 5 shows the layout used in group Elias delta SIMD. By gamma encoding the selector the property that each selector encodes exactly and completely one payload no longer exists. For this reason we first store the payloads then we store the selectors in reverse order (a technique used in QMX).

Two pointers are used for decoding: a payload pointer which starts at the beginning of the encoded sequence, and a selector pointer which starts at the end. Each time a new selector is needed the selector pointer is decremented by one. Each time a new payload is needed a pointer is incremented by one.

The group Elias delta SIMD encoding is expected to be larger than the gamma SIMD encoding for long postings lists, but smaller for shorter postings lists. In the gamma version the composition number is stored in unary, but in the delta version it is stored gamma encoded. Examining Table 1, the draw even point on storing an integer using unary versus gamma is 5 – so any d-gap smaller than $2^6 = 64$ will be more effectively stored using unary than using gamma.

6 EXPERIMENTS

6.1 Experimental Conditions

Experiments were conducted on an 18-core (36-thread) Dell Precision 5820 with an Intel Xeon W-2195 at 2.30GHz with 256GB DDR4-2666 RAM running 64-bit Windows 10. Programs were written in C++ and were compiled with Microsoft Visual Studio 2017 with C++ compiler version 19.14.26430 and maximum optimisation.

6.2 *In Vacuo* Experiment

To examine space (effectiveness) and decompression speed (efficiency), the open indexes of Lemire[6] were used. These datasets are postings lists of the TREC .gov2 collection of 25,205,179 web pages crawled from the .gov domain in 2004. There are two variants, the first is the collection in collection order, the second is the collection sorted in URL order, a technique well known to improve effectiveness and efficiency on this collection [2]. Terms that appear in fewer than 100 documents are not in the data. These datasets were altered only in so far as each docid was incremented by one as Lemire counts from 0 – recall that 0 cannot be encoded with the Elias codecs. In all cases d-gaps of 1 were used (which is not the most efficient for SIMD).

We implemented variable byte encoding, we used the ATIRE implementations of Elias gamma and Elias delta, we implemented Elias gamma and Elias delta using BMI1 instructions, and we implemented the group Elias gamma SIMD and group Elias delta SIMD. We did not implement the non-SIMD versions of the group Elias codecs because we had no reason to believe that they would be as efficient at decoding as the SIMD versions. The version of QMX we used came from the source code of JASS. All implementations were verified on the URL sorted data by compressing, then decompressing, then comparing the decompressed to the original of each and every postings list.

In our experiments we, one by one and for all postings lists, loaded a postings list from file, encoded it, measured the space taken, then measured the wall clock decoding time using using the C++ `std::chrono::steady_clock` – all in a single thread of the otherwise idle machine. This was repeated 5 times and the numbers reported here are the medians of those 5 runs. We choose median as we believe it is a better indication of *expected* performance than fastest, mean, or slowest.

6.3 *In Vacuo* Results

Figure 6 presents the space requirements of each of the codecs on the .gov2 collection in collection order. Not surprisingly, variable byte encoding takes the most space, also not surprisingly, QMX takes more space than the Elias codes, the group Elias SIMD codecs place in between. Of the group Elias SIMD codecs, the gamma performs better when postings lists are long.

Group Elias SIMD encodes each column of integers in the space required to store the largest number in the column – but this width

[6] http://lemire.me/data/integercompression2014.html

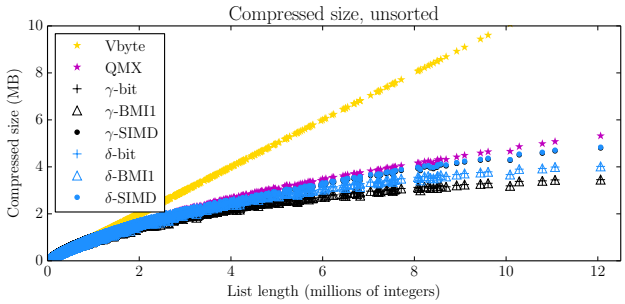

Figure 6: Space taken for encoded lists of .gov2 in collection order.

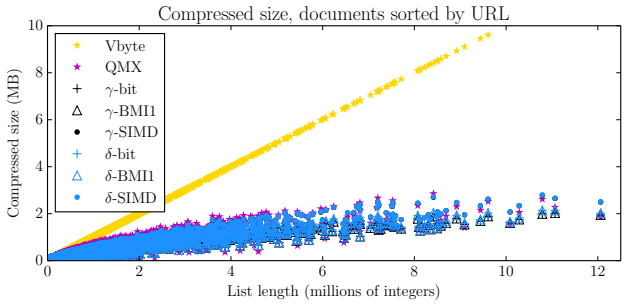

Figure 7: Space taken for encoded lists of .gov2 in URL order.

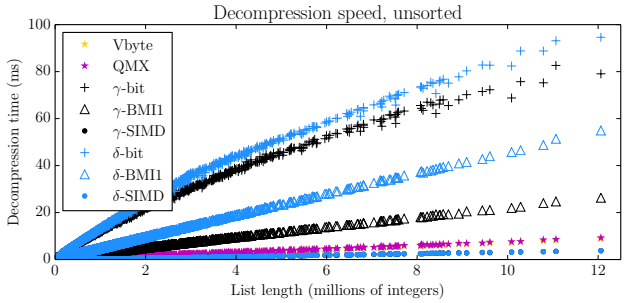

Figure 8: Time to decode lists from .gov2 in collection order. Vbyte is partially occluded by QMX.

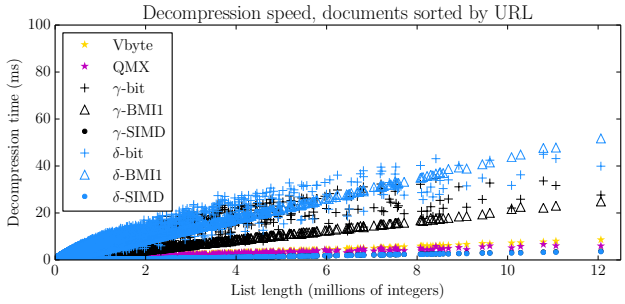

Figure 9: Time to decode lists from .gov2 in URL order.

is likely to be larger than the space required to encode the smallest integer in the column. When this happens there is wastage that does not happen in the non SIMD versions and hence it takes more space. QMX packs as many integers as it can into a 128-bit integer, all integers taking the width of the largest integer. In the case of highly frequent terms with a typical d-gap of 1, it only requires a single 2 to drop the effectiveness from 128 integers per word to 64 integers per word (to halve the effectiveness). In the case of the group Elias SIMD this 'damage' is limited to a single column of integers – the effect of an outlier is substantially reduced. The worst case of the selector for QMX is one byte per 128-bit word, for group Elias gamma SIMD uses 4 bytes per 512-bit word – in other words, the worst case for QMX is the normal case for group Elias gamma SIMD. This QMX effectiveness gain does not appear to compensate for the loss due to an outlier.

Table 3: Size of JASS .gov2 postings file encoded using QMX and the group Elias SIMD codecs.

Codec	Size (bytes)	Relative to QMX
QMX	12,855,247,344	100%
Group Elias Gamma SIMD	14,659,573,813	114%
Group Elias Delta SIMD	18,028,897,041	140%

Figure 7 presents the space requirements of each of the codecs on the .gov2 collection in URL order. Again, variable byte encoding takes the most space, the others are highly variable on the effectiveness, with QMX showing a wider variability than the others, and with the Elias codecs performing better than the SIMD versions. There is no clear "always best" performer.

Figure 8 presents the decoding time requirements of each of the codecs on the .gov2 collection in collection order. As expected, the bitwise implementations all take longer than the BMI1 implementations. Variable byte encoding is faster than the Elias codecs, and on this architecture, there is little difference in the decoding time of it and QMX. The group Elias SIMD codecs outperform the others with little difference between the gamma and delta versions.

Figure 9 presents the decoding time requirements of each of the codecs on the .gov2 collection in URL order. The Elias codecs are variable in performance, but take longer than variable byte and QMX. The group Elias SIMD decoders are the most efficient, and again there is little difference between them.

6.4 *In Situ* Experiment

We indexed .gov2 using ATIRE's impact ordered indexer, 8-bit quantized with BM25 ($k1 = 0.9$, $b = 0.4$) and then used an extended `atire_to_jass_index` to convert the index into JASS format. Three indexes were generated, one for each of: QMX, group Elias gamma SIMD, and group Elias delta SIMD. We then compared the sizes of the index. Finally, using JASSv2 we loaded the entire index into memory and used the C++ `std::chrono::steady_clock` to measure time to search to completion (no early termination, but with top-$k = 100$) for all 20,000 queries from the TREC Million Query Track (from 2007 and 2008). We ran the timing experiment 5 times and report medians. We repeated the experiment as the number of queries processed in parallel increased from 1 to 32 (recall, the computer has 18 cores giving 36 hyperthreads).

6.5 *In Situ* Results

JASS stores the postings lists in a file separate from the remainder of the index (the vocabulary, *etc.*), and Table 3 shows the size of the postings file for each of the three codecs along with the size relative to QMX. QMX takes the least space, most probably because it can encode small postings lists (for example, with one posting)

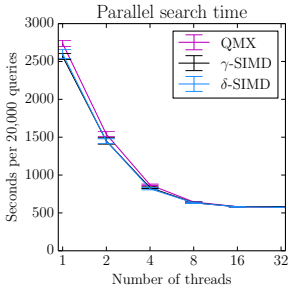

Figure 10: Wall time to execute all 20,000 TREC Million Query Track queries on .gov2 as the level of parallelism increases.

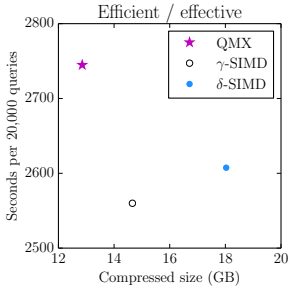

Figure 11: Postings list size and time to sequentially process all 20,000 TREC Million Query Track queries on .gov2.

in less than 128 bits, but the group Elias SIMD codecs cannot store an encoding in less than a single 512-byte payload and a 32-bit selector. We leave for further work the obvious extension of only using group Elias SIMD when size warrants.[7] Group Elias gamma SIMD takes less space than the delta variant because the majority of d-gaps in the index are small and the gamma code is better at storing smaller integers than the delta code.

Examining the throughput, Figure 10 shows the wall time required to complete all 20,000 queries as the level of parallelism is increased from 1 to 32. The figure shows that group Elias gamma SIMD is more efficient than the delta version which is more efficient than QMX. When the number of threads is small this is likely to be simply the number of integers that can be decoded per SIMD instruction. The levelling-out at approximately 8 threads suggests a hardware bottleneck. Given the small number of instructions necessary to decode the integers, this is likely to be the CPU / memory bandwidth being flooded (i.e. the memory wall). At 32-threads, there is little difference between the codecs, this is most likely because once the encoded data is in the cache the CPU can process it quickly and then stalls waiting for more data – and that stall is approximately the same regardless of the codec.

Figure 11 presents the space / time tradeoff of the three codecs (index size vs search time) with a single thread (as might be seen in a desktop search environment). It shows group Elias exhibiting an effectiveness loss for an efficiency improvement over QMX.

7 DISCUSSION AND CONCLUSIONS

One of the advantages of the group Elias SIMD codecs over QMX and other SIMD codecs is that it is obvious how to extend to larger (or reduce to smaller) word sizes. An AVX-2 version using 256-bit SIMD registers would differ only in the number of rows being used for encoding (from 16×32-bit to 8×32-bit).[8] When wider SIMD registers become available the number of rows seen in Figure 4 could simply be increased to match the number of 32-bit words in that SIMD register. Indeed, it could be parameterised in an implementation. It is not obvious how to extend QMX as the width of a machine word increases as it is not grounded on a solid theoretical base. Although the group Elias gamma SIMD .gov2 index is larger than the QMX index, no account has been made for short postings lists or the ends of long postings lists (partly-full payloads). This we leave for further investigation, however since it is obvious how to manage 256-bit, 128-bit, or even 64-bit or 32-bit payloads, there is an obvious channel of investigation.

[7] This technique is standard practice for codecs such as PForDelta where variable byte encoding is normally used when a block (of typically 128 integers) cannot be filled.

[8] Our implementation chooses, at compile time, between 512-bit registers (if available), or 2×256-bit registers (if not), differing from this suggestion.

REFERENCES

[1] V. N. Anh and A. Moffat. 2005. Inverted Index Compression Using Word-Aligned Binary Codes. *Information Retrieval* 8 (2005), 151–166.

[2] D. Blandford and G. Blelloch. 2002. Index compression through document reordering. In *Proceedings of the Data Compression Conference (DCC 2002)*. 342–351.

[3] A. Z. Broder, D. Carmel, M. Herscovici, A. Soffer, and J. Zien. 2003. Efficient Query Evaluation Using a Two-level Retrieval Process. In *CIKM 2003*. 426–434.

[4] S. Büttcher, C. L. A. Clarke, and G. V. Cormack. 2010. *Information Retrieval: Implementing and Evaluating Search Engines*. MIT Press. xxiv + 606 pages.

[5] M. Crane, A. Trotman, and R. A. O'Keefe. 2013. Maintaining Discriminatory Power in Quantized Indexes. In *CIKM 2013*. 1221–1224.

[6] J. Dean. 2009. Challenges in Building Large-scale Information Retrieval Systems: Invited Talk. In *WSDM 2009*. 1–1.

[7] S. Ding and T. Suel. 2011. Faster Top-k Document Retrieval Using Block-max Indexes. In *SIGIR 2011*. 993–1002.

[8] P. Elias. 1975. Universal Codeword Sets and Representations of the Integers. *IEEE Transactions on Information Theory* 21, 2 (1975), 194–203.

[9] S. W. Golomb. 1966. Run-Length Encodings. *IEEE Transactions on Information Theory* 12, 3 (1966), 399–401.

[10] D. Habich, P. Damme, A. Ungethüm, and W. Lehner. 2018. Make Larger Vector Register Sizes New Challenges?: Lessons Learned from the Area of Vectorized Lightweight Compression Algorithms. In *Proceedings of the Workshop on Testing Database Systems (DBTest 2018)*. Article 8, 6 pages.

[11] D. Lemire and L. Boytsov. 2015. Decoding Billions of Integers Per Second Through Vectorization. *Software: Practice & Experience* 45, 1 (2015), 1–29.

[12] J. Lin and A. Trotman. 2015. Anytime Ranking for Impact-Ordered Indexes. In *ICTIR 2015*. 301–304.

[13] J. Lin and A. Trotman. 2017. The Role of Index Compression in Score-at-a-time Query Evaluation. *Information Retrieval* 20, 3 (2017), 199–220.

[14] C. Macdonald, R. McCreadie, R. L. T. Santos, and I Ounis. 2012. From Puppy to Maturity: Experiences in Developing Terrier. *Proceedings of the SIGIR 2012 Workshop on Open Source Information Retrieval* (2012), 60–63.

[15] G. Ottaviano and R. Venturini. 2014. Partitioned Elias-Fano Indexes. In *SIGIR 2014*. 273–282.

[16] M. Petri and A. Moffat. 2018. Compact Inverted Index Storage Using General-Purpose Compression Libraries. *Software: Practice & Experience* 48, 4 (2018), 974–982.

[17] R. F. Rice. 1979. *Some Practical Universal Noiseless Coding Techniques*. JPL Publication 79-22. NASA Jet Propulsion Laboratory, Pasadena, CA, USA.

[18] B. Schlegel, R. Gemulla, and W. Lehner. 2010. Fast Integer Compression Using SIMD Instructions. In *Proceedings of the 6th International Workshop on Data Management on New Hardware (DaMoN '10)*. 34–40.

[19] F. Scholer, H. E. Williams, J. Yiannis, and J. Zobel. 2002. Compression of Inverted Indexes for Fast Query Evaluation. In *SIGIR 2002*. 222–229.

[20] A. A. Stepanov, A. R. Gangolli, D. E. Rose, R. J. Ernst, and P. S. Oberoi. 2011. SIMD-based Decoding of Posting Lists. In *CIKM 2011*. 317–326.

[21] N. Tonellotto and C. Macdonald. 2018. Efficient Query Processing Infrastructures: A Half-day Tutorial at SIGIR 2018. In *SIGIR 2018*. 1403–1406.

[22] A. Trotman. 2003. Compressing Inverted Files. *Information Retrieval* 6, 1 (2003), 5–19.

[23] A. Trotman. 2014. Compression, SIMD, and Postings Lists. In *ADCS 2014*. Article 50, 8 pages.

[24] A. Trotman, X.-F. Jia, and M. Crane. 2012. Towards an Efficient and Effective Search Engine. In *Proceedings of the SIGIR 2012 Workshop on Open Source Information Retrieval*. 40–47.

[25] A. Trotman and J. Lin. 2016. *In Vacuo* and *In Situ* Evaluation of SIMD Codecs. In *ADCS 2016*. 1–8.

[26] J. Zhang, X. Long, and T. Suel. 2008. Performance of Compressed Inverted List Caching in Search Engines. In *WWW 2008*. 387–396.

[27] W. X. Zhao, X. Zhang, D. Lemire, D. Shan, J.-Y. Nie, H. Yan, and J.-R. Wen. 2015. A General SIMD-Based Approach to Accelerating Compression Algorithms. *TOIS* 33, 3, Article 15 (2015), 28 pages.

[28] M. Zukowski, S. Heman, N. Nes, and P. Boncz. 2006. Super-Scalar RAM-CPU Cache Compression. In *Proceedings of the 22nd International Conference on Data Engineering (ICDE 2006)*. Article 59, 12 pages.