Decoding billions of integers per second through vectorization

Paper by D. Lemire and L. Boytsov
Presentation by Xander Morgan

Problem Statement: Optimize Space and Speed

- We would like to encode and decode large arrays of integers efficiently in terms of space and time, i.e., achieve good compression and high rate of processing
- Focus on 32-bit integer sequences, often sorted
- Main memory, rather than disk, often limits computation speed in modern algorithms
- Applications in search engines and relational databases

Aside to illustrate compression ideas (Space optimization is a solved problem)

Suppose we have a 4-sided die. The probabilities of the sides are listed below. We would like to roll this die many times and communicate results to a friend using a sequence of bits. How to efficiently do this?

Side	Probability
0	1/4
1	1/4
2	1/4
3	1/4

Aside to illustrate compression ideas (Space optimization is a solved problem)

Suppose we have a 4-sided die. The probabilities of the sides are listed below. We would like to roll this die many times and communicate results to a friend using a sequence of bits. How to efficiently do this?

Side	Probability	Bit representation
0	1/4	00
1	1/4	01
2	1/4	10
3	1/4	11

Aside to illustrate compression ideas (Space optimization is a solved problem)

Suppose we have a 4-sided die. The probabilities of the sides are listed below. We would like to roll this die many times and communicate results to a friend using a sequence of bits. How to efficiently do this?

Side	Probability
0	1/2
1	1/4
2	1/8
3	1/8

Aside to illustrate compression ideas (Space optimization is a solved problem)

Suppose we have a 4-sided die. The probabilities of the sides are listed below. We would like to roll this die many times and communicate results to a friend using a sequence of bits. How to efficiently do this?

Side	Probability	Bit representation
0	1/2	0
1	1/4	10
2	1/8	110
3	1/8	111

Aside to illustrate compression ideas (Space optimization is a solved problem)

- Given a probability distribution $p_X(\cdot)$ over a set $\{x_1, \ldots, x_n\}$, the average number of bits required to encode i.i.d. symbols from this distribution is at least

$$H[X] = \sum_x -p_X(x) \log_2(p_X(x))$$

- Furthermore, there are coding algorithms that can (with some nuance) achieve this bound

Binary and Unary

Binary:

1 -> 1

2 -> 10

3 -> 11

Unary:

1 -> 1

2 -> 01

3 -> 001

Elias gamma and delta coding

- Elias gamma: encode number of bits in unary, followed by binary representation of the number (without the MSB)

- Elias delta: encode number of bits using Elias gamma, followed by binary representation of the number (without the MSB)

Variable byte encoding

- Break integer into 7 bit chunks. Each 7 bit chunk is stored in a byte, with 0 as the 8th bit denoting “continue” and 1 denoting “end”

- E.g., 11001000 gets encoded as 10000001 01001000

Varint-GB and Varint-G8IU

Varint-GB

- Use one byte broken into four chunks of 2 bits each. Each 2 bit chunk encodes number of bytes used to describe an integer {1, 2, 3, 4}
- The integer encodings follow the descriptor byte

Varint-G8IU

- Use one byte descriptor that describes layout of 8 data bytes. A “0” indicates the end of an integer
- The integer encodings follow the descriptor byte
- Can be efficiently decoded using SIMD “pshufb” instruction

Idea 1: Differential Coding (Space)

- For sorted arrays, first pre-process elements into deltas
- $\delta_j = x_j - x_{j-1}$
- Recover original elements using prefix sum
- Compute differences in-place working from end of the array backwards toward the start

Idea 2: Utilize SIMD Operations (Speed)

- Many modern CPUs provide SIMD operations

- SIMD operations have been used to speedup varint-G8IU by 50% (decoding) and 300% (encoding) previously

- Use SIMD operations for encoding/decoding process AND prefix-sum

Idea 2: Utilize SIMD Operations (Speed)

- In particular, partition array into consecutive blocks of 4 elements each, take element-wise differences between blocks.

- This increases speed from 2 billion integers per second to 5 billion integers per second.

- Causes differences to be, on average, four times larger (costs 2 bits of storage)

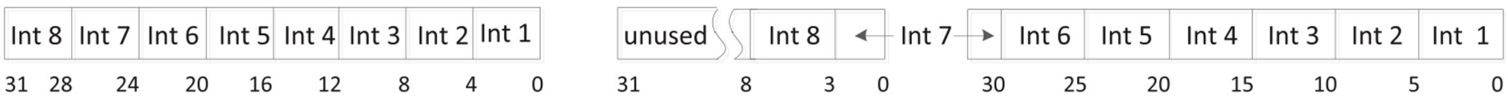
Idea 3: Break large arrays into small arrays during processing (Speed)

- For arrays with more than 256 KB worth of data, break them into 256 KB chunks and process them independently.

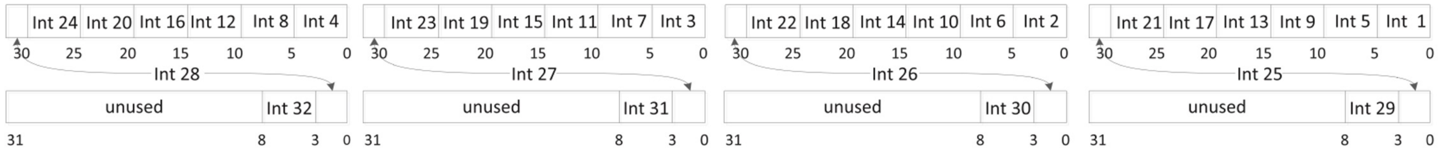
- Improves cache efficiency by reducing the number of cache misses

Idea 4: Bit packing (Space)

```
struct Fields4_8  {                    struct Fields5_8 {
    unsigned    Int1: 4;                   unsigned    Int1: 5;
    unsigned    Int2: 4;                   unsigned    Int2: 5;
    unsigned    Int3: 4;                   unsigned    Int3: 5;
    unsigned    Int4: 4;                   unsigned    Int4: 5;
    unsigned    Int5: 4;                   unsigned    Int5: 5;
    unsigned    Int6: 4;                   unsigned    Int6: 5;
    unsigned    Int7: 4;                   unsigned    Int7: 5;
    unsigned    Int8: 4;                   unsigned    Int8: 5;
};                                     };
```



(a) 4-bit integers (b) 5-bit integers



Terminology

- Page: Group of thousands/millions of integers
- Block: Group of 128 integers

In particular, an array of integers comprises many pages, and each page comprises many blocks.

Idea 6: Store Exceptions at the page level (Space)
Idea 7: Choose different bit widths for each block (Space)

- An exception is an unusually large integer within a block

- The bit width of a block is chosen to match the “typical” integer bit width within the block. Any integers that exceed this bit width are stored as an exception.

Example

- Can choose b = 6 bits, but would like b to be smaller
- Heuristic of cost for each exception: 8 + (6 - b) = 14 - b
- Choose b to minimize 128*b + (14 - b) * c   (in this example, replace 128 with 16)
- Here, choose b = 2 to minimize cost

10, 10, 1, 10, 100110, 10, 1, 11, 10, 100000, 10, 110100, 10, 11, 11, 1

10, 10, 1, 10, 10, 10, 1, 11, 10, 00, 10, 00, 10, 11, 11, 1.

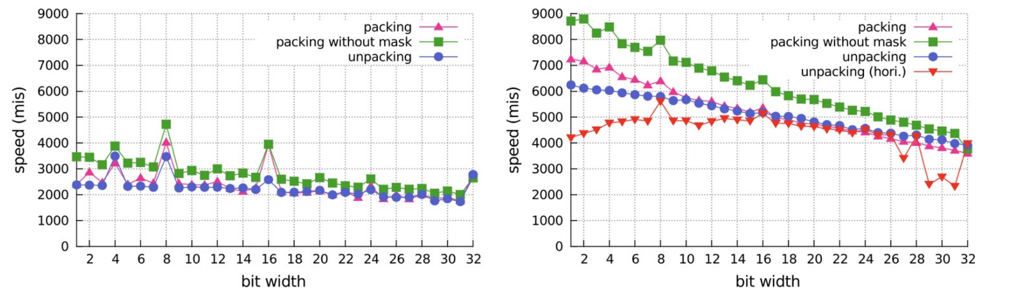
Example

- Compressed page starts with 32 bit integer describing the total size of the truncated sequence
- Skip the truncated sequence to reach the byte array
- Byte array contains b, maximal bit width, number of exceptions, locations of exceptions, all using one byte each.
- Exceptions follow the byte array and are further compressed
  - SimplePFOR compresses using Simple-8b
  - FastPFOR has 32 arrays, one for each possible exception bit-width
- Exceptions are the first component decoded and are decoded in bulk

Data to be compressed:	…10, 10, 1, 10, 100110, 10, 1, 11, 10, 100000, 10, 110100, 10, 11, 11, 1…
Truncated data: ($16 \times 2 = 32$ bits)	…10, 10, 01, 10, 10, 10, 01, 11, 10, 00, 10, 00, 10, 11, 11, 01 …
Byte array: ($6 \times 8 = 48$ bits)	…2, 6, 3, 4, 9, 11 …
Exception data: (to be compressed)	…1001, 1000, 1101 …

Experiments

- "Linux server equipped with Intel Core i7 2600 (3.40 GHz, 8192 KB of L3 CPU cache) and 16 GB of RAM. The DDR3-1333 RAM with dual channel has a transfer rate of 20,000 MB/s or 5300 mis."

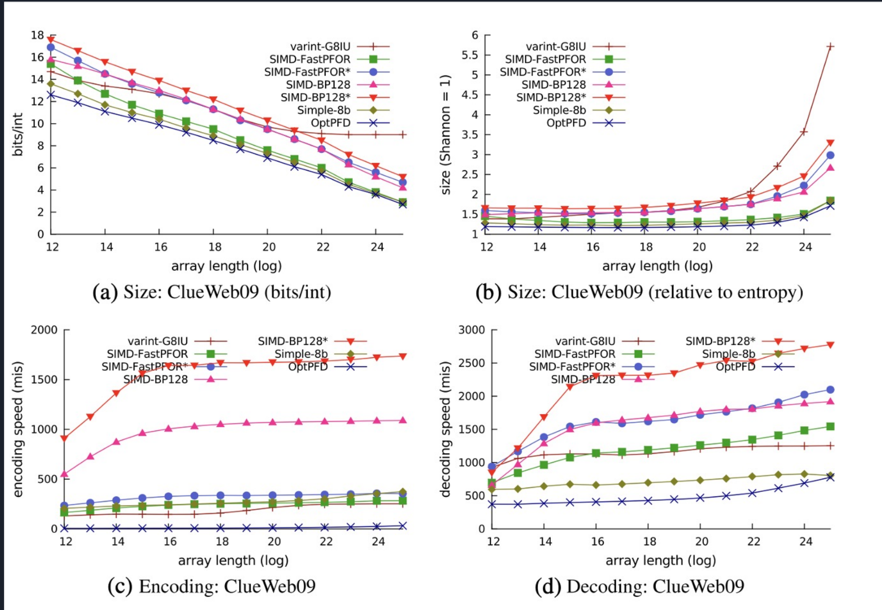


(a) Optimized but portable C++ (b) Vectorized with SSE2 instructions

Experiments

- Test on ClusterData and Uniform model synthetic datasets
- Test on real datasets: ClueWeb09 (Category B) data set and GOV2 data set
- Test against other state-of-the art algorithms like PFOR and varint-G8IU
- End up with the fastest coding and decoding speeds, with competitive compression ratios
- SIMD-BP128 works very well across test cases

Experiments

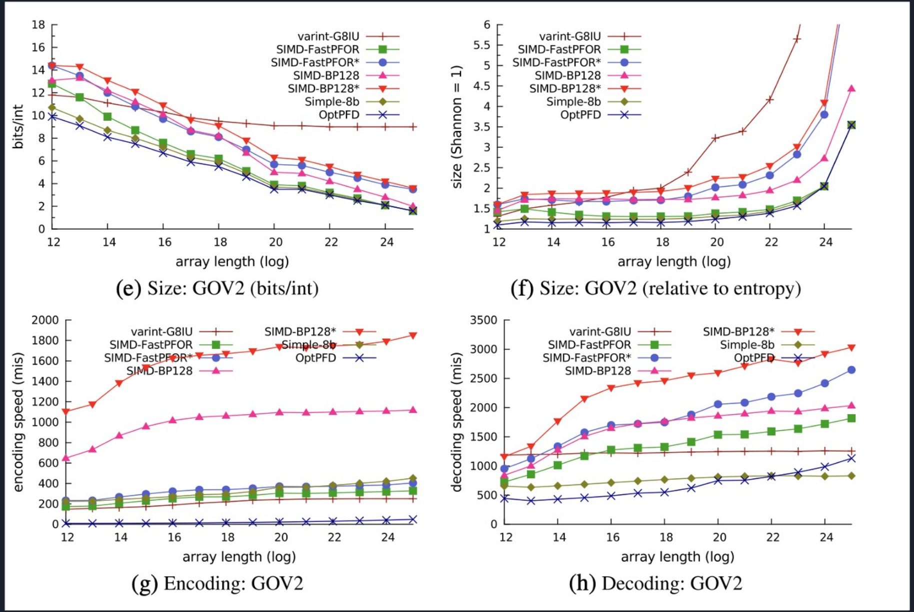


(a) Size: ClueWeb09 (bits/int)

(b) Size: ClueWeb09 (relative to entropy)

(c) Encoding: ClueWeb09

(d) Decoding: ClueWeb09



(e) Size: GOV2 (bits/int)

(f) Size: GOV2 (relative to entropy)

(g) Encoding: GOV2

(h) Decoding: GOV2

Evaluation of paper and comparison to previous work

- Strong speed increase over previous methods
- Utilizes binary packing and vectorizes it (not done previously, at least not nearly as effectively)
- Strength: Comprehensive analysis, many other compression schemes are introduced and the authors did extensive testing to compare their ideas to previous work
- Strength: Good examples.
- Weakness: Many acronyms to keep track of, and many variants of the compression schemes. This makes it harder (at least for me) to develop general “take away” ideas from the paper.
- Future work: data-based adaptive compression schemes, and probabilistic analysis of the algorithms proposed here