Decoding billions of integers per second through vectorization (2012)

D. Lemire, L. Boytsov

*Softw. Pract. Exper.* 2015; **45**:1–29

presented by Krit Boonsiriseth

MIT 6.506 Spring 2023
Overview
Vectorization
Integer compression
Results and summary
Decoding billions of integers per second through vectorization (2012)

D. Lemire, L. Boytsov
What are we **decoding**?

- Well, billions of integers... stored in a compressed format
- Specific problem: **compressing and decompressing sorted arrays of 32-bit integers**

```
1, 4, 10, 12
+3 +6 +2
```

```
1, 3, 6, 2
```

```markdown
sorted array
array of differences
compressed
```
Compressing and decompressing billions of sorted 32-bit integers per second through vectorization (2012)

D. Lemire, L. Boytsov
Compressing and decompressing billions of sorted 32-bit integers per second through vectorization (2012)
D. Lemire, L. Boytsov
Why are we **compressing**?

- Memory hierarchy: compressed data fits into **faster storage**
Why is **compression** possible?

- We can’t compress a truly random list of integers.
- However, in practice most integers we encounter are far smaller than $2^{32}$.
- We can generally use much less than 32 bits per integer.
How are we compressing?

• We’ll talk about this later!
Compressing and decompressing billions of sorted 32-bit integers per second through vectorization (2012)
D. Lemire, L. Boytsov
Why **sorted** 32-bit integers?

- Example use case: **database indexes**
- Suppose we want to index occurrences of “Alice”.
- The row numbers form a **sorted array of integers**!

<table>
<thead>
<tr>
<th>0</th>
<th>Bob</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Alice</td>
</tr>
<tr>
<td>2</td>
<td>Bob</td>
</tr>
<tr>
<td>3</td>
<td>Bob</td>
</tr>
<tr>
<td>4</td>
<td>Alice</td>
</tr>
<tr>
<td>5</td>
<td>Alice</td>
</tr>
<tr>
<td>6</td>
<td>Bob</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

1, 4, 5, ...
Why *sorted* 32-bit integers?

- In this implementation, does not lose generality: the actual *compression* step works on any array of integers.
Why sorted 32-bit integers?

• A guess: this paper was written in 2012, when 128-bit vector registers were still the norm

• We’ll talk more about vectorization later!

<table>
<thead>
<tr>
<th>Vector instruction set</th>
<th>Register width</th>
<th>Proposed</th>
<th>Shipped</th>
</tr>
</thead>
<tbody>
<tr>
<td>SSE</td>
<td>128 bits</td>
<td>1999</td>
<td>1999</td>
</tr>
<tr>
<td>SSE2</td>
<td>128 bits</td>
<td>2001</td>
<td>2001</td>
</tr>
<tr>
<td>SSE3</td>
<td>128 bits</td>
<td>2004</td>
<td>2004</td>
</tr>
<tr>
<td>AVX</td>
<td>256 bits</td>
<td>2008</td>
<td>2011</td>
</tr>
</tbody>
</table>
Compressing and decompressing billions of sorted 32-bit integers per second through vectorization (2012)
D. Lemire, L. Boytsov
How fast is **billions per second**?

• Processor clocks: a few billion cycles per second
• Fastest algorithm described by this paper takes 1.5 cycles per integer to decode and 2.1 cycles per integer to encode
• This is around **1.5x faster** than existing algorithms with comparable compression ratios
  • More evaluation results at the end!
• Possible because we can operate on multiple integers per instruction, through vectorization!
Compressing and decompressing billions of sorted 32-bit integers per second through vectorization (2012)
D. Lemire, L. Boytsov
Overview

Vectorization

• Introduction
• History
• Vectorizing bit packing
• Vectorizing differential coding

Integer compression

Results and summary
What is **vectorization**?

- Vectorization is the use of **vector instructions**, which operate on multiple data at once.

```
1 + 2 = 3
add
```

```
1  2  3  4
+   +
2  3  5  7
=   =
3  5  8 11
vectorized add
```
Brief history of vectorization

1966  First vectorized computer (ILLIAC IV)
1970s  Vectorized supercomputers are commonplace
1990s  Supercomputers move away from vectorization, vectorization starts being commonplace in PCs
1996  Intel MMX instructions (64-bit vector registers)
1999  Intel SSE instructions (128-bit vector registers)
2004  End of clock speed scaling, parallelism becomes necessary for optimal performance
2011  Intel AVX instructions (256-bit vector registers)
Benefits of vectorization

• "In a sense, the speed gains we have achieved are a **direct application of advanced hardware instructions** to the problem of integer coding (specifically SSE2 introduced in 2001)"
Vectorizing **bit packing**

- Sorted array: 1, 4, 10, 12
  - Differences: +3, +6, +2
- Array of differences: 1, 3, 6, 2
- Compressed: ???

RAW TEXT

Vectorizing **bit packing**

sorted array array of differences compressed

1, 4, 10, 12
+3 +6 +2

1, 3, 6, 2

???
Vectorizing bit packing

**Idea:** just convert everything to vector instructions

```c
void unpack5_8(const uint32_t* in, uint32_t* out) {
    *out++ = (*((in)) & 31;
    *out++ = (*((in) >> 5 ) & 31;
    *out++ = (*((in) >> 10) & 31;
    *out++ = (*((in) >> 15) & 31;
    *out++ = (*((in) >> 20) & 31;
    *out++ = (*((in) >> 25) & 31;
    *out = (*((in) >> 30);
    ++in;
    *out++ |= (*((in) & 7) << 2;
    *out = (*((in) >> 3) & 31;
}
```

```c
const static __m128i m7 = _mm_set1_epi32(7U);
const static __m128i m31 = _mm_set1_epi32(31U);

void SIMDunpack5_8(const __m128i* in, __m128i* out) {
    __m128i i = _mm_load_si128(in);
    _mm_store_si128(out++, _mm_and_si128( i , m31));
    _mm_store_si128(out++, _mm_and_si128( _mm_srl_epi32(i, 5) , m31));
    _mm_store_si128(out++, _mm_and_si128( _mm_srl_epi32(i,10) , m31));
    _mm_store_si128(out++, _mm_and_si128( _mm_srl_epi32(i,15) , m31));
    _mm_store_si128(out++, _mm_and_si128( _mm_srl_epi32(i,20) , m31));
    _mm_store_si128(out++, _mm_and_si128( _mm_srl_epi32(i,25) , m31));
    __m128i o = _mm_srl_epi32(i,30);
    i = _mm_load_si128(++in);
    o = _mm_or_si128(o, _mm_slli_epi32(_mm_and_si128(i, m7), 2));
    _mm_store_si128(out++, o);
    _mm_store_si128(out++, _mm_and_si128( _mm_srl_epi32(i,3) , m31));
}
```

bit unpacking
vectorized bit unpacking
Vectorizing differential coding

- Sorted array: 1, 4, 10, 12
  +3 +6 +2
- Array of differences: 1, 3, 6, 2
- Compressed: ???
Vectorizing **differential coding**

**Idea:** compute differences of array elements that are 4 elements apart instead of consecutive elements

<table>
<thead>
<tr>
<th>1</th>
<th>4</th>
<th>10</th>
<th>12</th>
<th>13</th>
<th>18</th>
<th>19</th>
<th>20</th>
<th>22</th>
<th>25</th>
<th>28</th>
<th>29</th>
</tr>
</thead>
</table>

\[
\begin{align*}
- \\
1 & 4 & 10 & 12
\end{align*}
\]

\[
\begin{align*}
= \\
1 & 4 & 10 & 12
\end{align*}
\]

\[
\begin{align*}
- \\
13 & 18 & 19 & 20
\end{align*}
\]

\[
\begin{align*}
= \\
9 & 7 & 9 & 9
\end{align*}
\]

This is faster, but results in ~4x larger differences, which require around **2 more bits per integer**
Overview
Vectorization

**Integer compression**
- Introduction
- Compression metrics
- Examples of encodings

Results and summary
Integer compression

1, 4, 10, 12
+3 +6 +2

1, 3, 6, 2

???
Integer compression

• Most integers ‘should’ use much less than 32 bits.

00000000 00000000 00000000 00011001

• We’d like to just store this as 11001
• Issue: need to define an encoding to make it clear where each integer starts and ends!
What makes for a good encoding?

- integers decoded / encoded per second
- decode speed is usually more important

**speed**

- bits used per integer

**compression**
Integer encodings

The paper includes *many* examples of encodings; we’ll focus on the ones that build up to the encodings that were actually used.

• Variable byte family
• Simple family
• **Binary packing family**
• **Patched binary packing family**

Variants implemented by this paper
Variable byte encoding

Use 7 bits in each byte for data, one bit for metadata (1 to mark starting point of each integer)

10000110 00010001 10000110 10010001
↓ decodes as
1100010001, 110, 10001
**varint-G81U\(^1\) encoding**

**New ingredient**: store metadata in separate bytes

<table>
<thead>
<tr>
<th>metadata</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>1 ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>data</td>
<td>00000011 00010001 00000110 00010001</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

↓ decodes as

1100010001, 110, 10001

This is faster because it can use a *shuffle intrinsic*, but uses slightly more bits per integer

\(^1\) actual encoding uses different byte order and flips metadata bits
From **Variable byte** to **Simple**

**Inefficiency #1:** variable byte requires padding integers to bytes even when most integers are less than a byte!

**Fix #1:** partition integers into blocks, and use *different integer sizes* for each block
Simple-8b encoding

Encode into 64-bit blocks. Each block has 4 bits of metadata, which determines the integer width for the remaining 60 bits of data.

metadata 0110 → mode 6: width is 5 bits per integer

data 00111 00010 10101 00000 11100 ...

↓ decodes as

111, 10, 10101, 0, 11100, ...

This is slightly slower, but uses fewer bits per integer when most integers are less than a byte.
From *Simple* to *Binary packing*

**Inefficiency #2:** what if instead of (base 10) 7, 2, 13, 0, 22 we have

1000007, 1000002, 1000013, 1000000, 1000022

**Fix #2:** include *offset* in the metadata for each block
Binary packing encoding

Fix #2: include offset in the metadata for each block

metadata [bit width = 5], [offset = 10^6]
data 00111 00010 10101 00000 11100 ...
decodes as 1000007, 1000002, 1000013, 1000000, 1000022, ...
Binary packing + patching

Inefficiency #3: what if instead of (base 10)
7, 2, 13, 0, 22
we have
7, 2, 13, 1000000, 22

Fix #3: use small bit width, and store exceptions separately
("patching")
Binary packing + patching

- Use small bit width, and store exceptions outside of blocks

blocks (~1000 bits each)
- metadata [bit width = 5], [offset = 0], ...
- data 7, 3, 13, *, 22, ...
- exceptions 1000000, ...
Binary packing + patching

- Organize blocks into **pages** that fit into LLC, and store the exceptions in each page in an exception array

**Pages (~32 MB each)**

- Page metadata
- Exception array

**Blocks (~1000 bits each)**

- Metadata data
- Metadata data
- Metadata data
- ...
More optimizations

• Use variable-length blocks
• Compress exception arrays!
• Use sampling heuristic to determine bit width for each block
• Store low bits of exception values as normal data
Options, options, options

There are many design choices

Table III. Overview of the patched coding schemes: Only PFOR and PFOR2008 generate compulsory exceptions and use a single bit width $b$ per page. Only NewPFD and OptPFD store exceptions on a per block basis. We implemented all schemes with 128 integers per block and a page size of at least $2^{16}$ integers.

<table>
<thead>
<tr>
<th>Scheme</th>
<th>Compulsory</th>
<th>Bit width</th>
<th>Exceptions</th>
<th>Compressed exceptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>PFOR [26]</td>
<td>Yes</td>
<td>Per page</td>
<td>Per page</td>
<td>No</td>
</tr>
<tr>
<td>PFOR2008 [25]</td>
<td>Yes</td>
<td>Per page</td>
<td>Per page</td>
<td>8, 16, 32 bits</td>
</tr>
<tr>
<td>NewPFD/OptPFD [10]</td>
<td>No</td>
<td>Per block</td>
<td>Per block</td>
<td>Simple-16</td>
</tr>
<tr>
<td>FastPFOR (Section 5)</td>
<td>No</td>
<td>Per block</td>
<td>Per page</td>
<td>Binary packing</td>
</tr>
<tr>
<td>SIMD-FastPFOR (Section 5)</td>
<td>No</td>
<td>Per block</td>
<td>Per page</td>
<td>Vectorized bin. Pack.</td>
</tr>
<tr>
<td>SimplePFOR (Section 5)</td>
<td>No</td>
<td>Per block</td>
<td>Per page</td>
<td>Simple-8b</td>
</tr>
</tbody>
</table>
Overview
Vectorization
Integer compression

Results and summary ←
## Results

<table>
<thead>
<tr>
<th></th>
<th>(a) ClueWeb09</th>
<th></th>
<th></th>
<th></th>
<th>(b) GOV2</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Coding</td>
<td>Decoding</td>
<td>Bits/int</td>
<td>Coding</td>
<td>Decoding</td>
<td>Bits/int</td>
<td>Coding</td>
<td>Decoding</td>
</tr>
<tr>
<td>SIMD-BP128*</td>
<td>1600</td>
<td>2300</td>
<td>11</td>
<td>1600</td>
<td>2500</td>
<td>7.6</td>
<td>350</td>
<td>1900</td>
</tr>
<tr>
<td>SIMD-FastPFOR*</td>
<td>330</td>
<td>1700</td>
<td>9.9</td>
<td>350</td>
<td>1900</td>
<td>7.2</td>
<td>1000</td>
<td>1700</td>
</tr>
<tr>
<td>SIMD-BP128</td>
<td>1000</td>
<td>1600</td>
<td>9.5</td>
<td>1000</td>
<td>1700</td>
<td>6.3</td>
<td>240</td>
<td>1500</td>
</tr>
<tr>
<td>varint-G8IU*</td>
<td>220</td>
<td>1400</td>
<td>12</td>
<td>240</td>
<td>1500</td>
<td>10</td>
<td>290</td>
<td>1400</td>
</tr>
<tr>
<td>SIMD-FastPFOR</td>
<td>250</td>
<td>1200</td>
<td>8.1</td>
<td>290</td>
<td>1400</td>
<td>5.3</td>
<td>250</td>
<td>1300</td>
</tr>
<tr>
<td>PFOR2008</td>
<td>260</td>
<td>1200</td>
<td>10</td>
<td>250</td>
<td>1300</td>
<td>7.9</td>
<td>310</td>
<td>1300</td>
</tr>
<tr>
<td>PFOR</td>
<td>330</td>
<td>1200</td>
<td>11</td>
<td>310</td>
<td>1300</td>
<td>7.9</td>
<td>310</td>
<td>1300</td>
</tr>
<tr>
<td>varint-G8IU</td>
<td>210</td>
<td>1200</td>
<td>11</td>
<td>230</td>
<td>1300</td>
<td>9.6</td>
<td>230</td>
<td>1300</td>
</tr>
<tr>
<td>BP32</td>
<td>760</td>
<td>1100</td>
<td>8.3</td>
<td>790</td>
<td>1200</td>
<td>5.5</td>
<td>790</td>
<td>1200</td>
</tr>
<tr>
<td>SimplePFOR</td>
<td>240</td>
<td>980</td>
<td>7.7</td>
<td>270</td>
<td>1100</td>
<td>4.8</td>
<td>270</td>
<td>1100</td>
</tr>
<tr>
<td>FastPFOR</td>
<td>240</td>
<td>980</td>
<td>7.8</td>
<td>270</td>
<td>1100</td>
<td>4.9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NewPFD</td>
<td>100</td>
<td>890</td>
<td>8.3</td>
<td>150</td>
<td>1000</td>
<td>5.2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VSEncoding</td>
<td>11</td>
<td>740</td>
<td>7.6</td>
<td>11</td>
<td>810</td>
<td>5.4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Simple-8b</td>
<td>280</td>
<td>730</td>
<td>7.5</td>
<td>340</td>
<td>780</td>
<td>4.8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OptPFD</td>
<td>14</td>
<td>500</td>
<td>7.1</td>
<td>23</td>
<td>710</td>
<td>4.5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable Byte</td>
<td>570</td>
<td>540</td>
<td>9.6</td>
<td>730</td>
<td>680</td>
<td>8.7</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Results

Algorithms in this paper are on the speed/compression ratio frontier.

better compression

faster
Summary

• This paper presents several integer encodings that are on the speed/compression ratio frontier.

• This is achieved by vectorization and by optimizing some design choices in a patched binary packing encoding.
Summary and discussion

• This paper presents several integer encodings that are on the speed/compression ratio frontier.

• This is achieved by vectorization and by optimizing some design choices in a patched binary packing encoding.

• It feels to me that the main idea for this paper is mostly “vectorization works!”, but this paper was written in 2012, which is around a decade after vectorization became popular.

• Natural directions for future work includes using newer vector instruction sets (AVX, AVX-512) and further optimizing in the design space of existing integer encoding families.