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

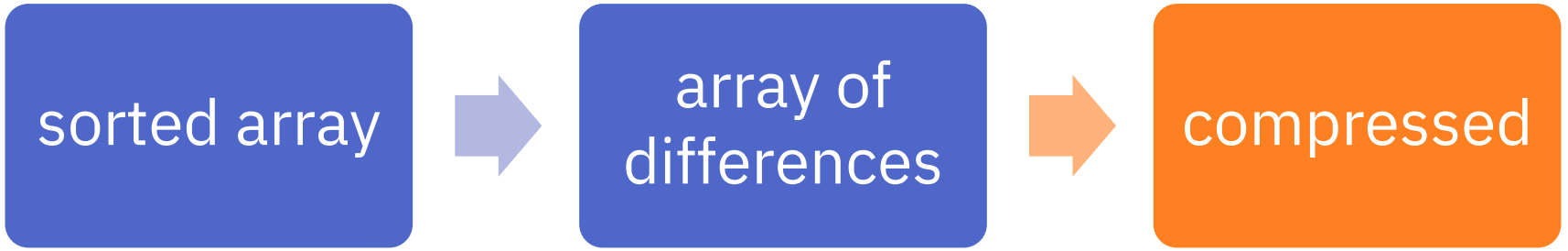
Results and summary

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**

???

Compressing and decompressing billions of sorted 32-bit integers per second through vectorization (2012)

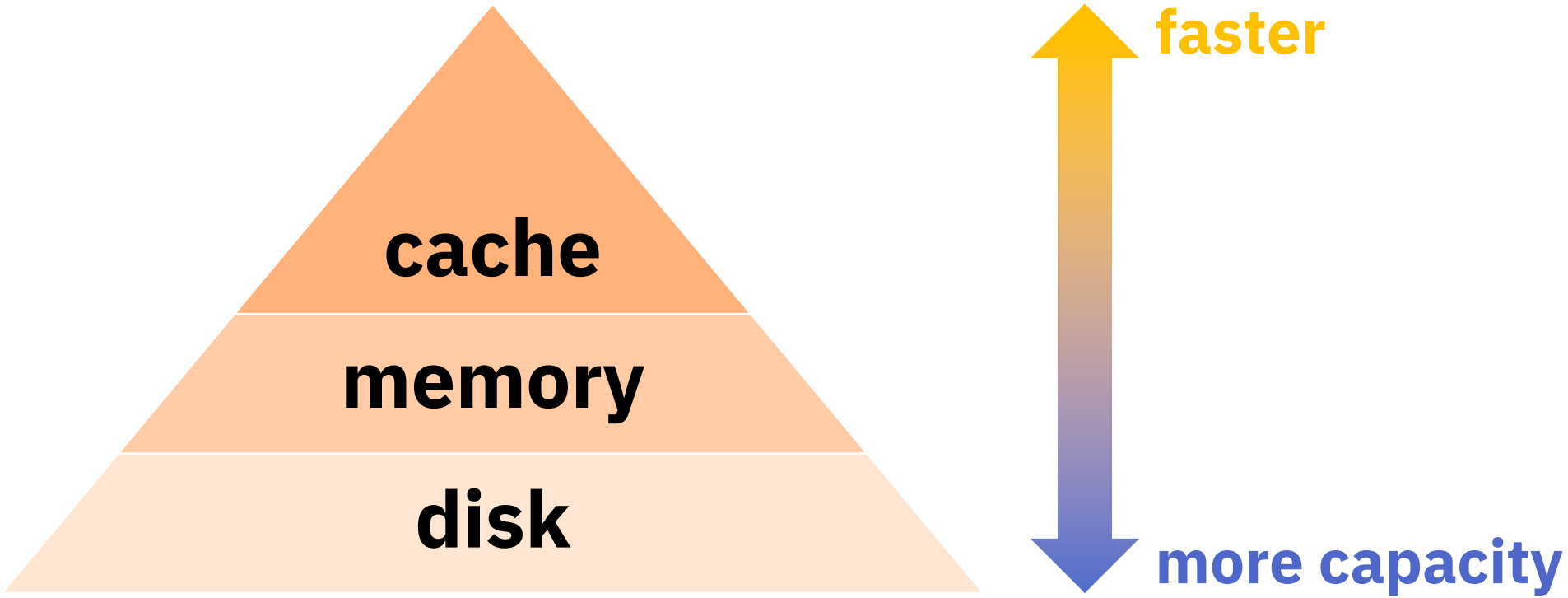
D. Lemire, L. Boytsov

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!

Overview

Vectorization

Integer compression ← here

Results

Compressing and decompressing billions of sorted 32-bit integers per second through vectorization (2012)

D. Lemire, L. Boytsov

Why **sorted** 32-bit integers?

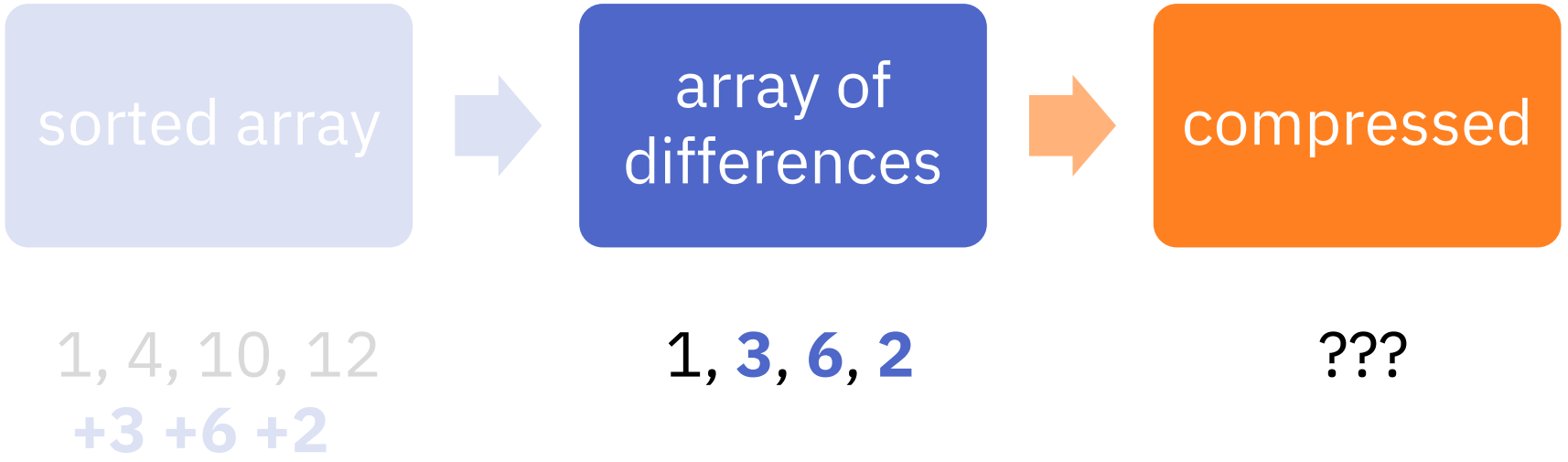
- Example use case: **database indexes**
- Suppose we want to index occurrences of “Alice”.
- The row numbers form a sorted array of integers!

0	Bob
1	**Alice**
2	Bob
3	Bob
4	**Alice**
5	**Alice**
6	Bob
…	…

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!

Vector instruction set	Register width	Proposed	Shipped
SSE	128 bits	1999	1999
SSE2	128 bits	2001	2001
SSE3	128 bits	2004	2004
AVX	256 bits	2008	**2011**

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 ←

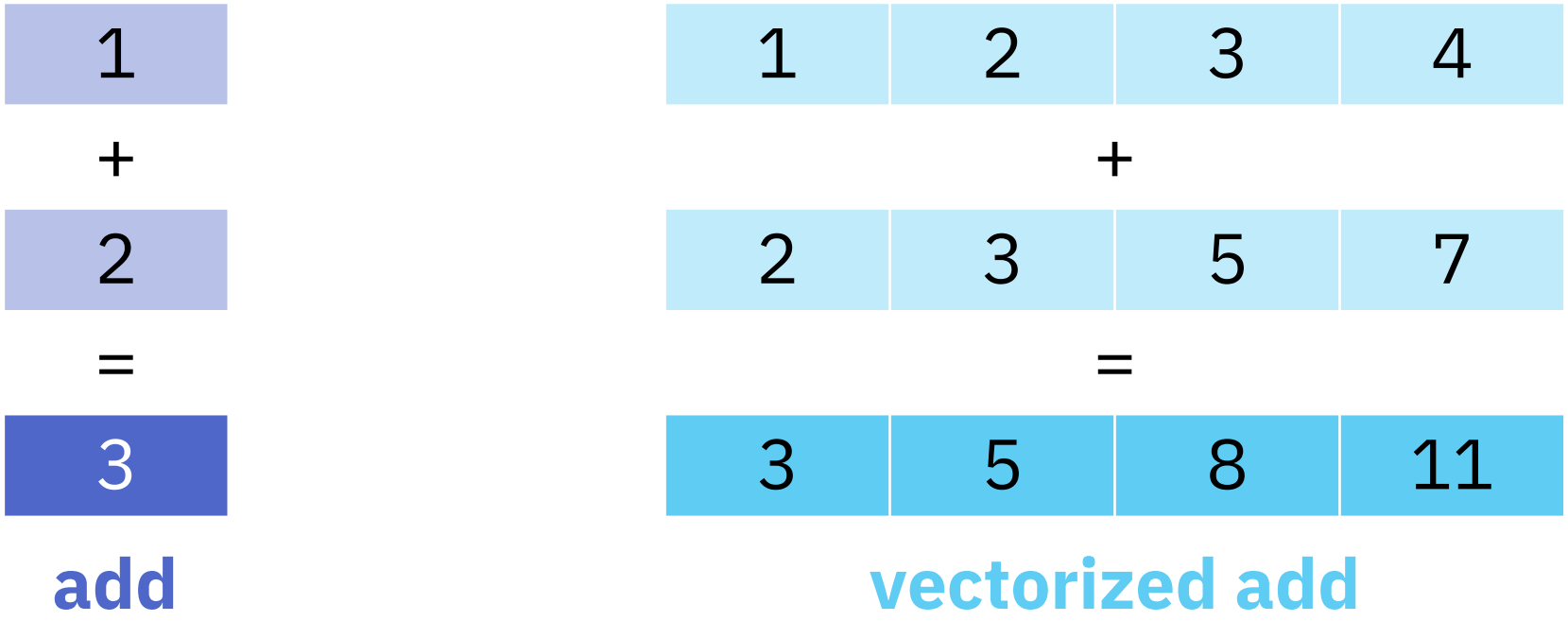
- Introduction
- History
- Vectorizing bit packing
- Vectorizing differential coding

Integer compression

Results and summary

What is **vectorization**?

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



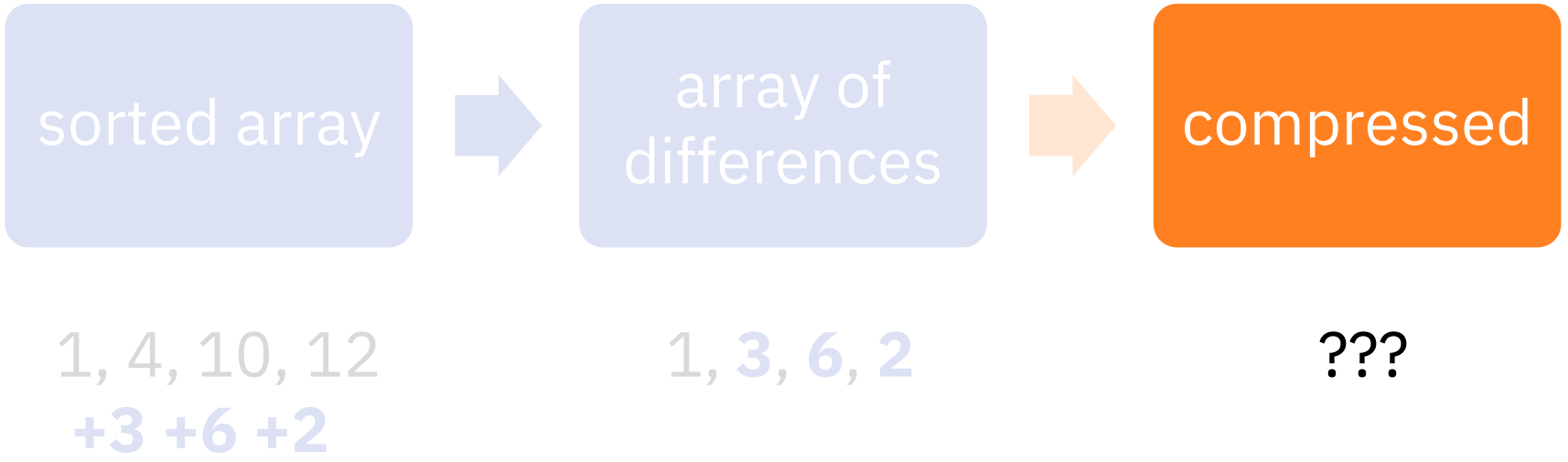
Brief history of **vectorization**

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**



Vectorizing **bit packing**

Idea: just convert everything to vector instructions

```
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;
}
```

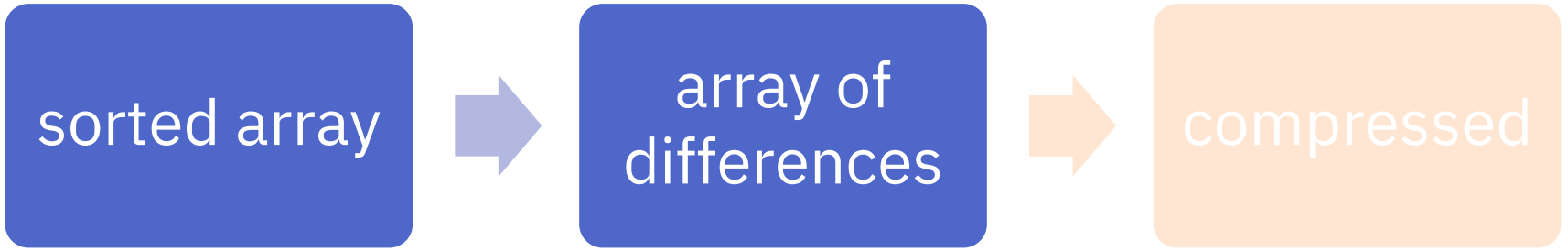
bit unpacking

```
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_srli_epi32(i,5)  , m31));
  _mm_store_si128(out++, _mm_and_si128(  _mm_srli_epi32(i,10) , m31));
  _mm_store_si128(out++, _mm_and_si128(  _mm_srli_epi32(i,15) , m31));
  _mm_store_si128(out++, _mm_and_si128(  _mm_srli_epi32(i,20) , m31));
  _mm_store_si128(out++, _mm_and_si128(  _mm_srli_epi32(i,25) , m31));
  __m128i  o = _mm_srli_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_srli_epi32(i,3)  , m31));
}
```

vectorized bit unpacking

Vectorizing **differential coding**



1, 4, 10, 12

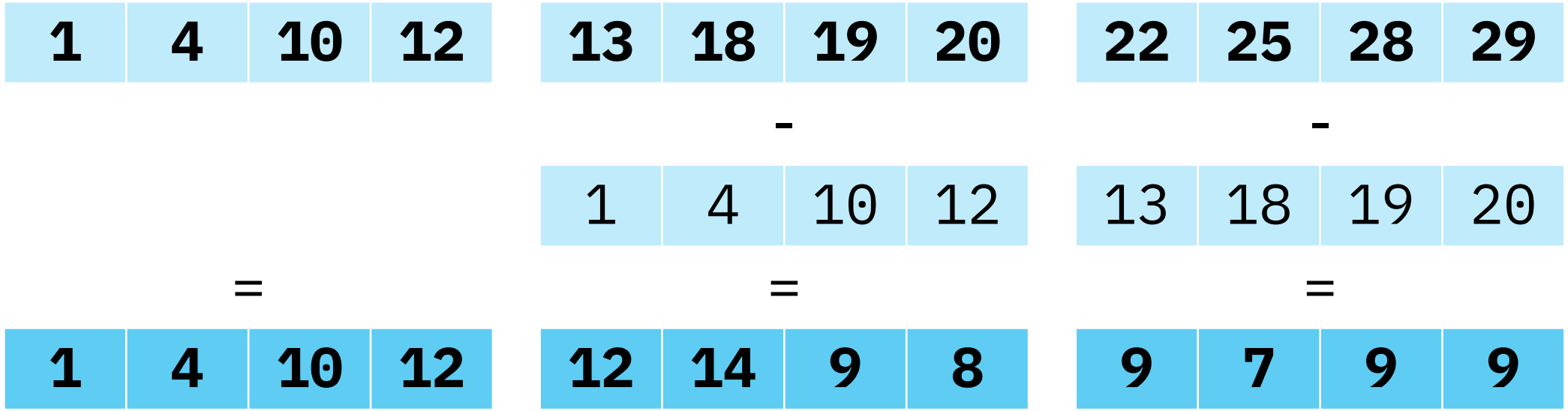
+3 +6 +2

1, **3**, **6**, **2**

???

Vectorizing **differential coding**

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



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

Overview

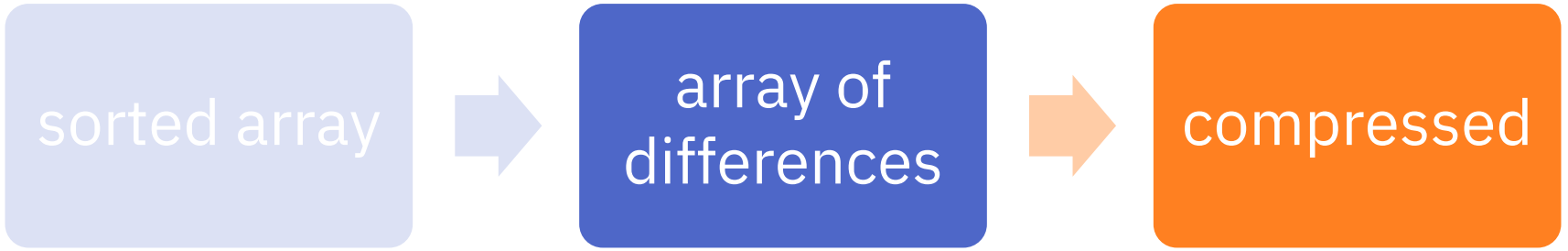
Vectorization

Integer compression ←

- Introduction
- Compression metrics
- Examples of encodings

Results and summary

Integer compression



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

- bits used per integer

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 **0**0010001 **1**0000110 **1**0010001

⬇ decodes as

1100010001, 110, 10001

varint-G81U[1] encoding

New ingredient: store metadata in separate bytes

metadata	**1**	**0**	**1**	**1 …**
data	00000011	00010001	00000110	00010001

⬇ 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**

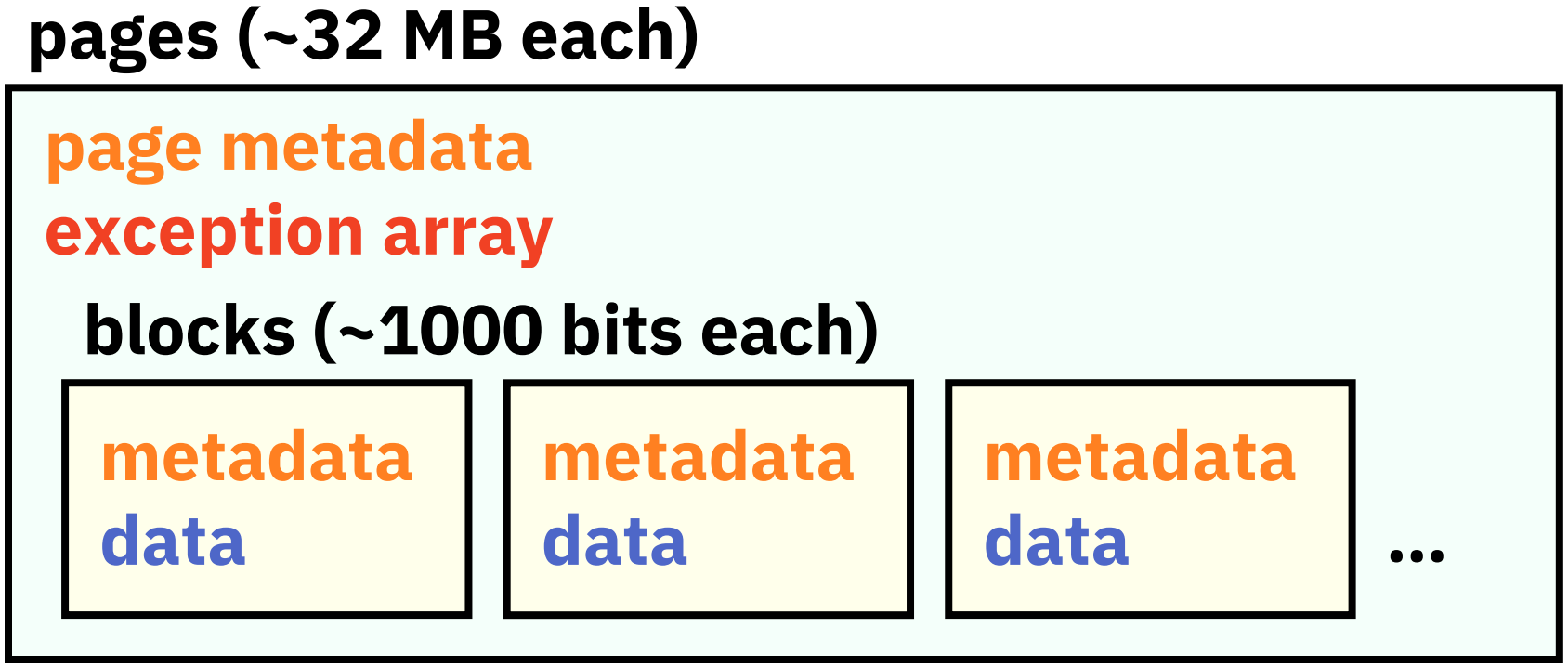
blocks (~1000 bits each)

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



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.

	Compulsory	Bit width	Exceptions	Compressed exceptions
PFOR [26]	Yes	Per page	Per page	No
PFOR2008 [25]	Yes	Per page	Per page	8, 16, 32 bits
NewPFD/OptPFD [10]	No	Per block	Per block	Simple-16
FastPFOR (Section 5)	No	Per block	Per page	Binary packing
SIMD-FastPFOR (Section 5)	No	Per block	Per page	Vectorized bin. Pack.
SimplePFOR (Section 5)	No	Per block	Per page	Simple-8b

This paper

Overview

Vectorization

Integer compression

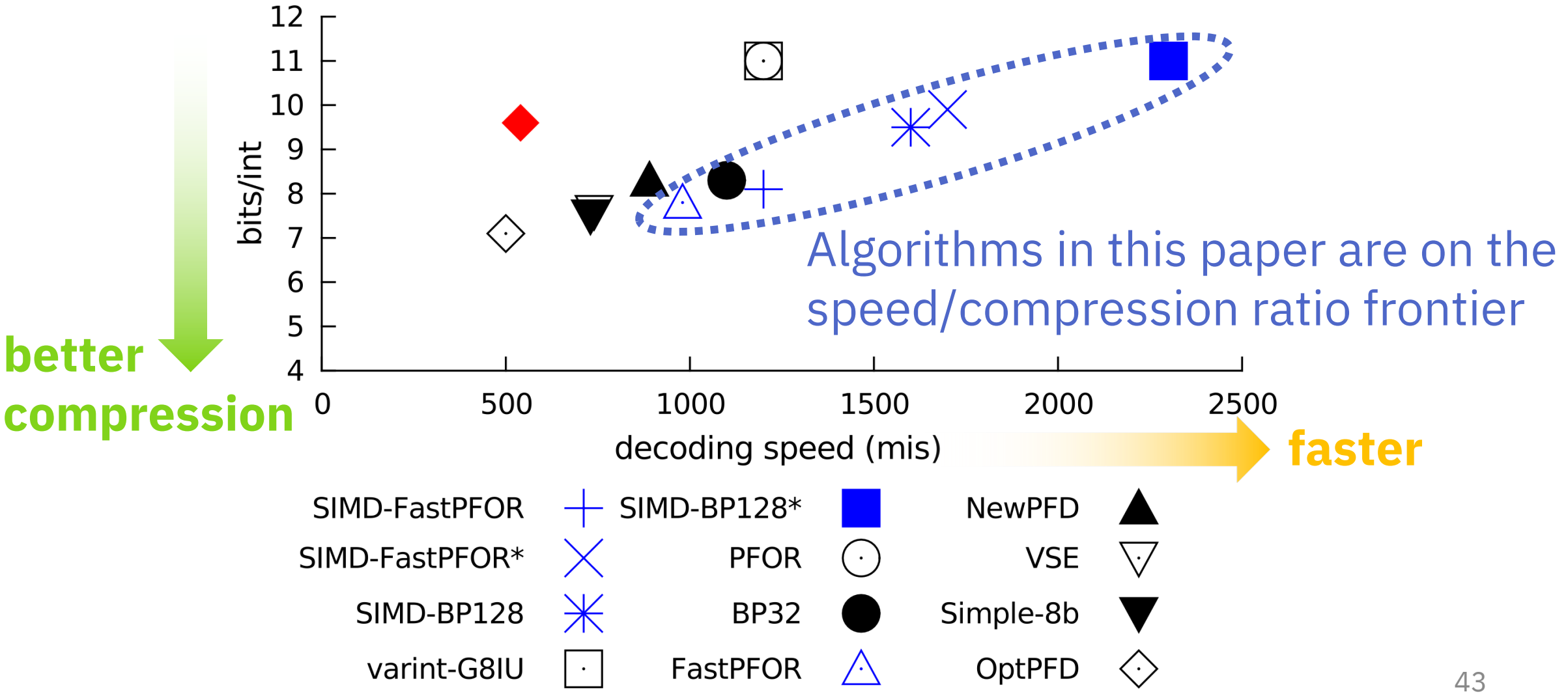
Results and summary ←

Results

This paper

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

Results


bits/int
12
11
10
9
8
7
6
5
4
0
500
1000
1500
2000
2500
decoding speed (mis)
better compression
faster
Algorithms in this paper are on the speed/compression ratio frontier
SIMD-FastPFOR
SIMD-FastPFOR*
SIMD-BP128
varint-G8IU
SIMD-BP128*
PFOR
BP32
FastPFOR
NewPFD
VSE
Simple-8b
OptPFD

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