Efficient Lightweight Compression Alongside Fast Scans

Orestis Polychroniou

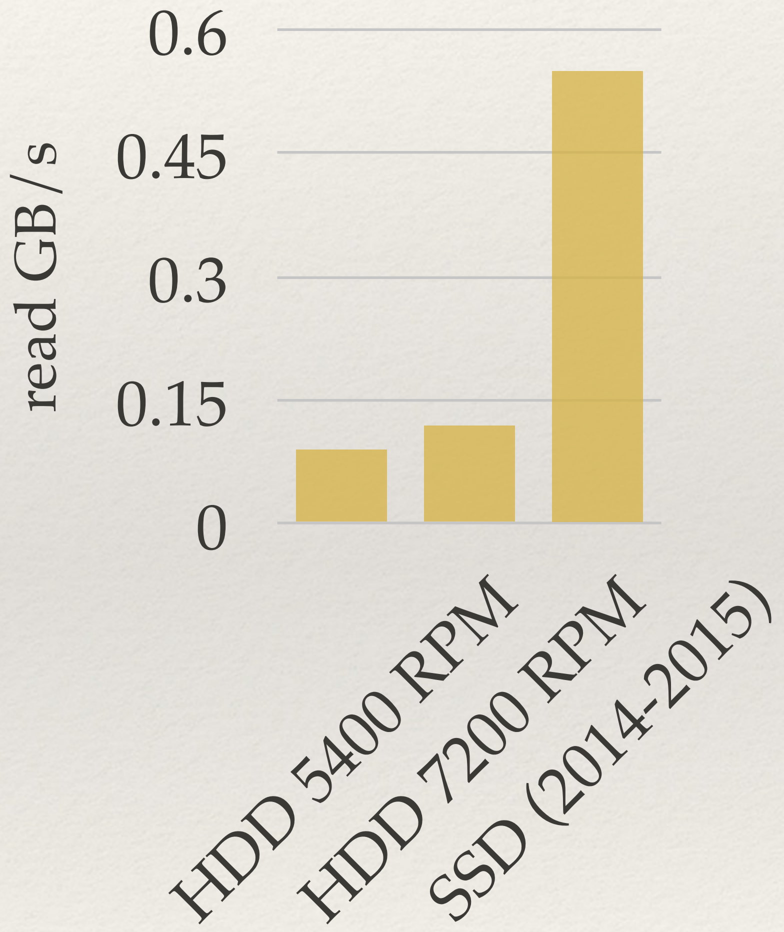
Kenneth A. Ross

Columbia University
in the City of New York

DaMoN 2015, Melbourne, Victoria, Australia

Databases & Compression

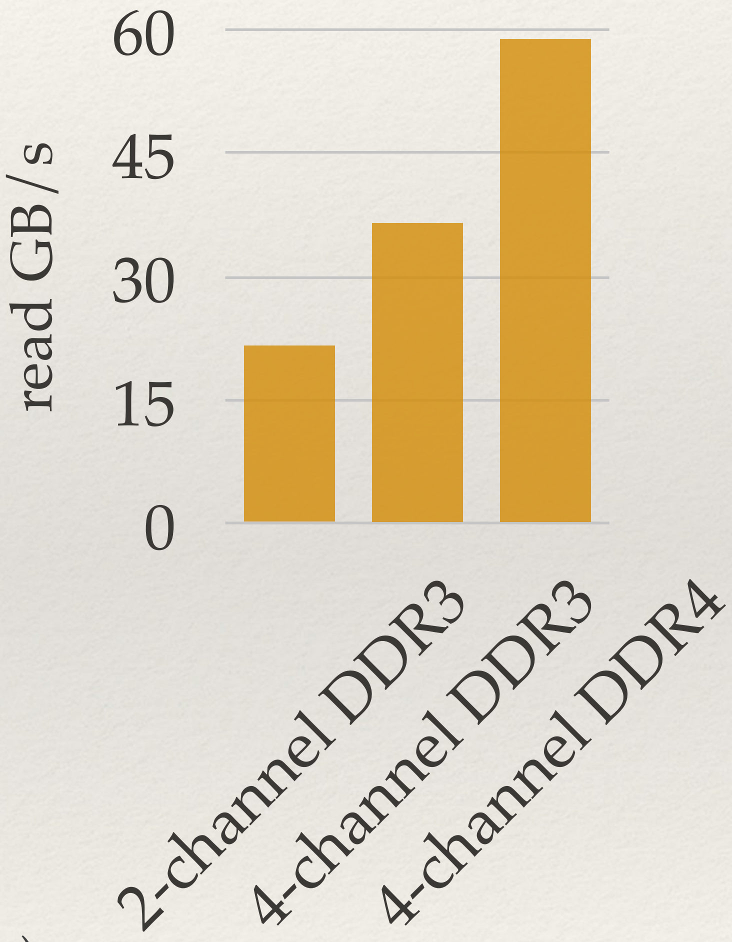
- Process data on ***disk***
  - Nearly ***unlimited*** capacity
  - Affects query ***optimization***
    - Minimize # of blocks fetched
    - Minimize # of random block accesses
  - Compress to improve disk speed
    - Focused on ***compression rate*** since disks are "slow"



Databases & Compression

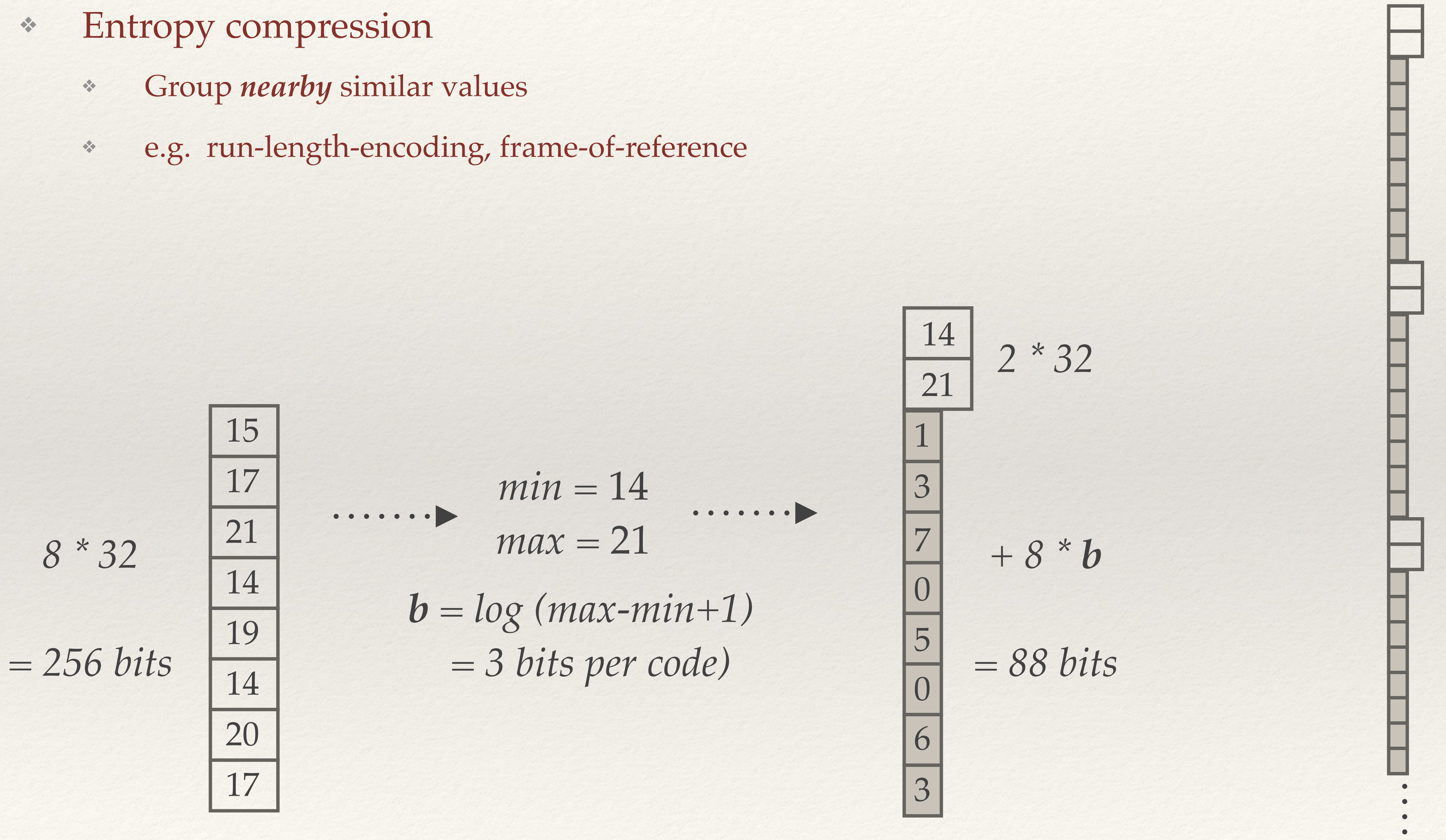
- Process data on ***disk***
  - Nearly ***unlimited*** capacity
  - Affects query ***optimization***
    - Minimize # of blocks fetched
    - Minimize # of random block accesses
  - Compress to improve disk speed
    - Focused on ***compression rate*** since disks are "slow"

- Process data on ***RAM***
  - Always ***limited*** capacity
  - Affects query ***optimization*** & query ***execution***
    - Minimize # of accesses (e.g. column stores & late materialization)
    - Minimize # of ***random*** (out of CPU cache) accesses (e.g. partitioned join)
  - Compress to improve RAM speed & ***avoid*** disk
    - Focused on ***(de-) compression efficiency*** as RAM is "fast"



Lightweight Compression

- Compression schemes
  - Entropy compression
    - Group ***nearby*** similar values
    - e.g. run-length-encoding, frame-of-reference



*8 * 32*

= 256 bits

15
17
21
14
19
14
20
17

min = 14
max = 21

b *= log (max-min+1)*
= 3 bits per code)

14
21

*2 * 32*

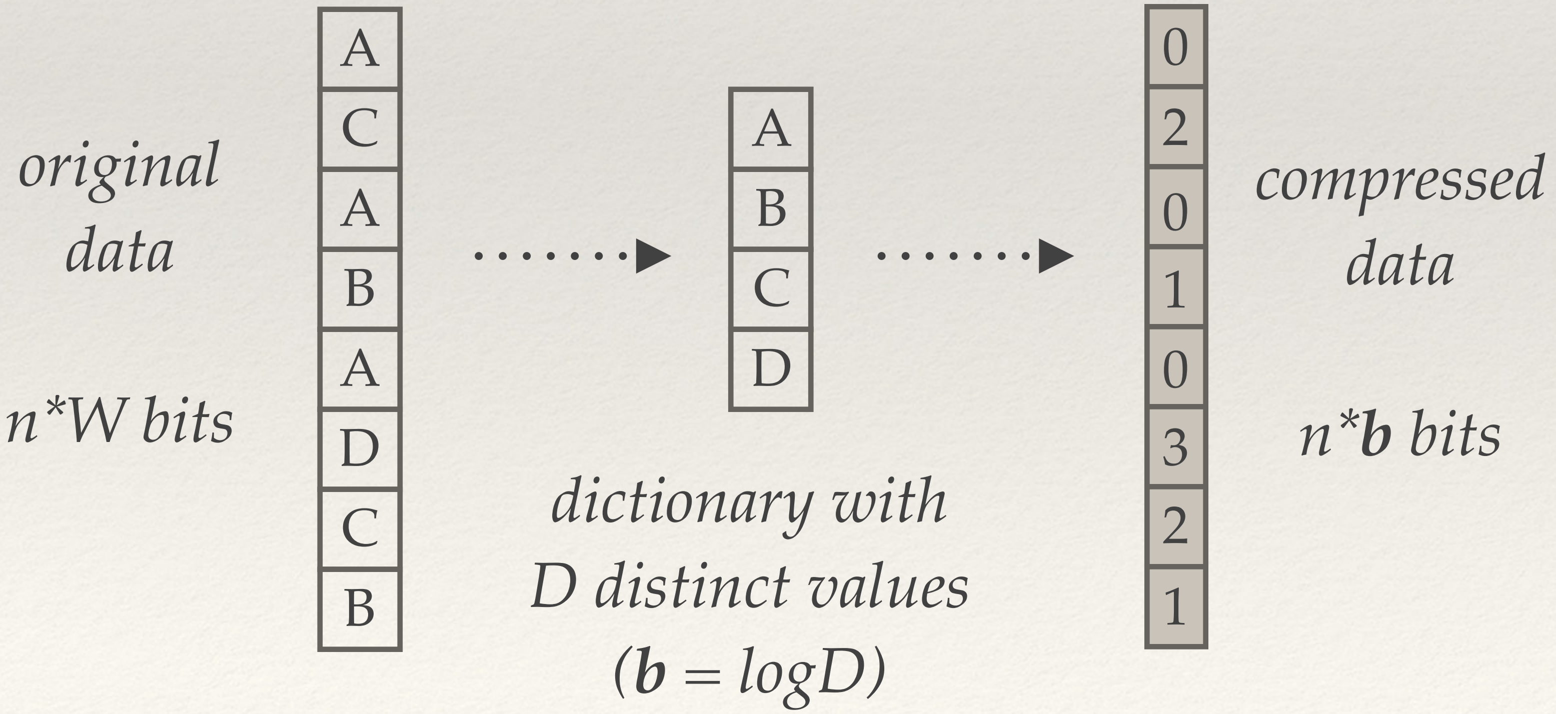
1
3
7
0
5
0
6
3

*+ 8 * **b***

= 88 bits

Lightweight Compression

- Compression schemes
  - Entropy compression
    - Group ***nearby*** similar values
    - e.g. run-length-encoding, frame-of-reference
  - Symbol compression
    - Assign a ***symbol*** to each distinct value
    - e.g. dictionary compression



original data

*n*W bits*

A
C
A
B
A
D
C
B

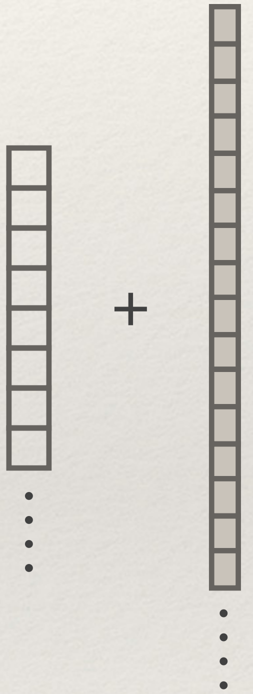
A
B
C
D

dictionary with D distinct values ($\mathbf{b}$ = logD)

0
2
0
1
0
3
2
1

compressed data

*n****b*** *bits*



+

Lightweight Compression

- Compression schemes
  - Entropy compression
    - Group ***nearby*** similar values
    - e.g. run-length-encoding, frame-of-reference
  - Symbol compression
    - Assign a ***symbol*** to each distinct value
    - e.g. dictionary compression
  - Frequency (symbol) compression
    - Compress ***frequent*** symbols with less bits
    - e.g. Huffman coding (slow), multiple dictionaries (fast)

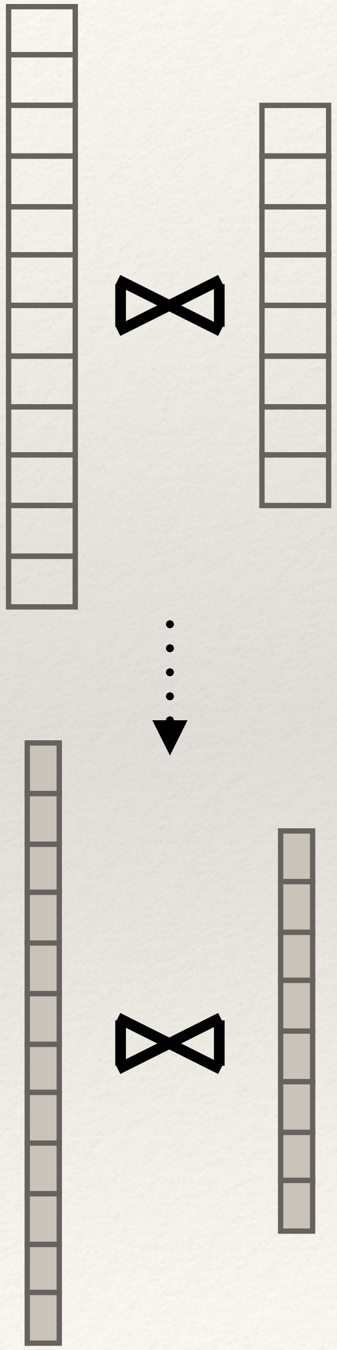
Lightweight Compression

- Compression schemes
  - Entropy compression
    - Group ***nearby*** similar values
    - e.g. run-length-encoding, frame-of-reference
  - Symbol compression
    - Assign a ***symbol*** to each distinct value
    - e.g. dictionary compression
  - Frequency (symbol) compression
    - Compress ***frequent*** symbols with less bits
    - e.g. Huffman coding (slow), multiple dictionaries (fast)

- DBMS integration
  - Decompress ***during*** execution
    - In CPU cache (non-integrated) or in registers (integrated)

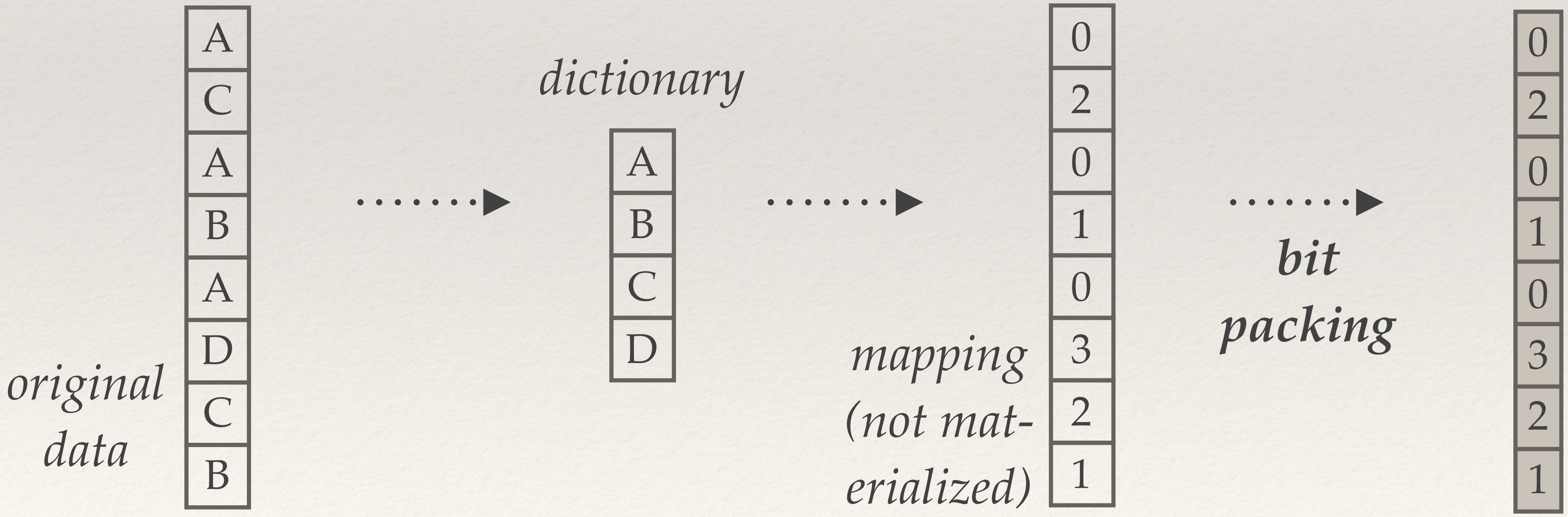
Lightweight Compression

- Compression schemes
  - Entropy compression
    - Group ***nearby*** similar values
    - e.g. run-length-encoding, frame-of-reference
  - Symbol compression
    - Assign a ***symbol*** to each distinct value
    - e.g. dictionary compression
  - Frequency (symbol) compression
    - Compress ***frequent*** symbols with less bits
    - e.g. Huffman coding (slow), multiple dictionaries (fast)
- DBMS integration
  - Decompress ***during*** execution
    - In CPU cache (non-integrated) or in registers (integrated)
  - Process ***compressed*** data without decompressing



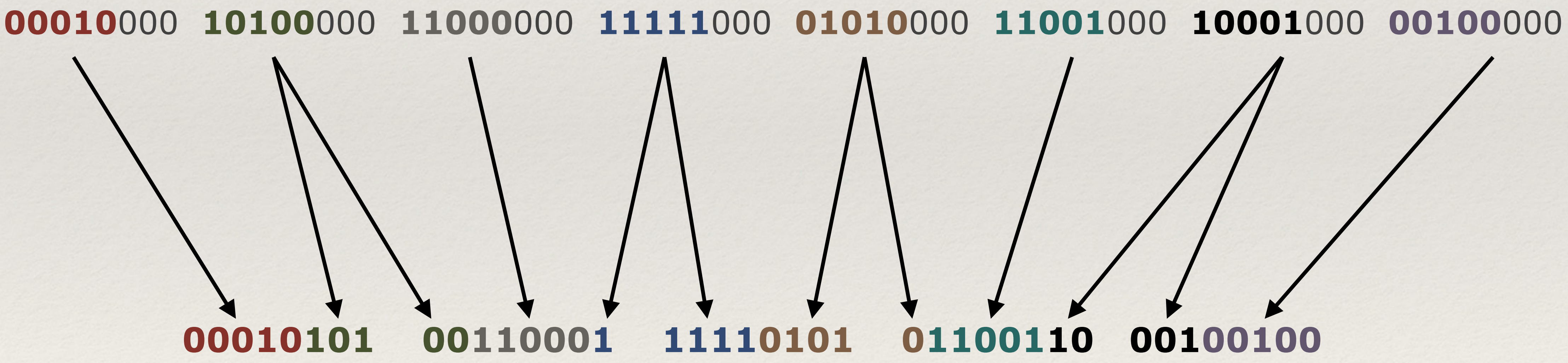
Bit Packing

- Definition
  - Input code width is hardware-supported
    - 8-bit, 16-bit, 32-bit, 64-bit
  - Output code width ***b*** must be (almost) constant
    - Either constant across the ***entire*** input
    - Or constant for the next ***group*** of items (e.g. frame-of-reference)



Bit Packing

- Layouts
  - ***Horizontal*** bit packing
    - Bits per code are ***contiguous***

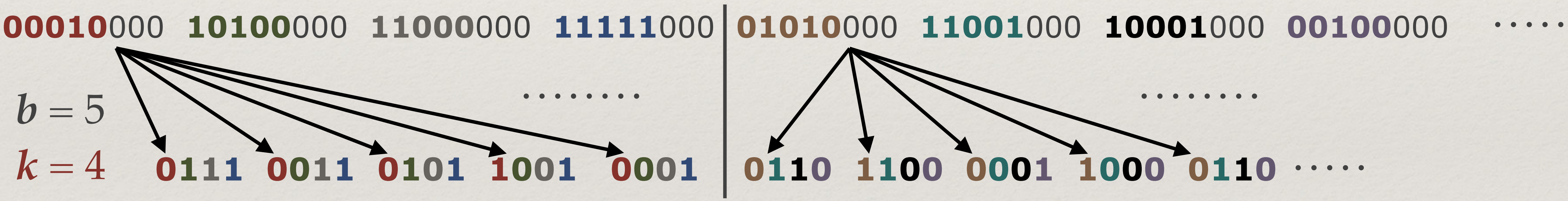


00010000 10100000 11000000 11111000 01010000 11001000 10001000 00100000

00010101 00110001 11110101 01100110 00100100

Bit Packing

- Layouts
  - ***Horizontal*** bit packing
    - Bits per code are ***contiguous***
  - ***Vertical*** bit packing
    - Bits of codes are ***interleaved***



00010000 10100000 11000000 11111000 ·····

01010000 11001000 10001000 00100000 ·····

·········

·········

$b = 5$

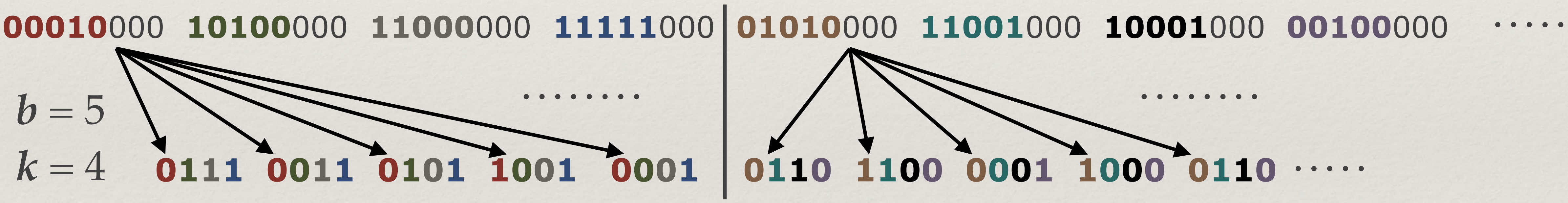
$k = 4$

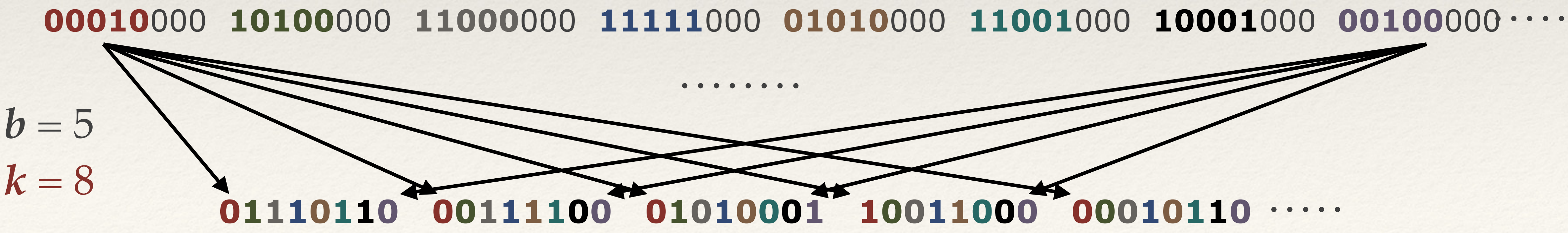
0111 0011 0101 1001 0001

0110 1100 0001 1000 0110 ·····

Bit Packing

- Layouts
  - *Horizontal* bit packing
    - Bits per code are *contiguous*
  - *Vertical* bit packing
    - Bits of codes are *interleaved*





Outline

- Operations
  - Packing
  - Unpacking
  - Scanning

Outline

- Operations
  - Packing
  - Unpacking
  - Scanning

- Horizontal layouts
  - Fully packed
    - Fast unpacking & scanning
  - Word aligned
    - Faster scanning

Outline

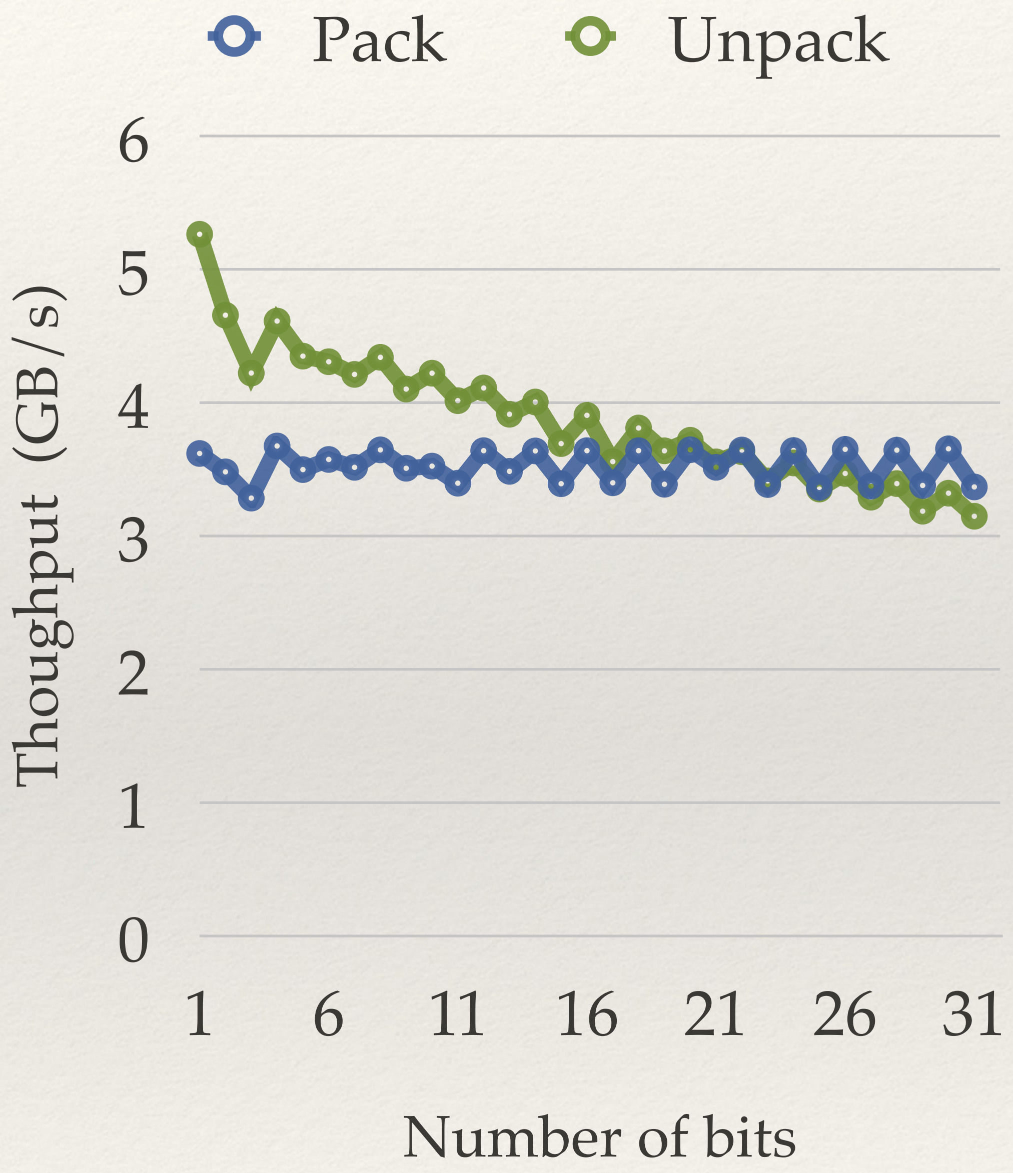
- Operations
  - Packing
  - Unpacking
  - Scanning
- Horizontal layouts
  - Fully packed
    - Fast unpacking & scanning
  - Word aligned
    - Faster scanning
- Vertical layout
  - Known traits
    - Fastest scanning
  - **New traits**
    - **Fast packing & unpacking**

Horizontal Layout

- *Fully* packed
    - No space wasted
        - Codes can *span* across 2 packed words

Horizontal Layout

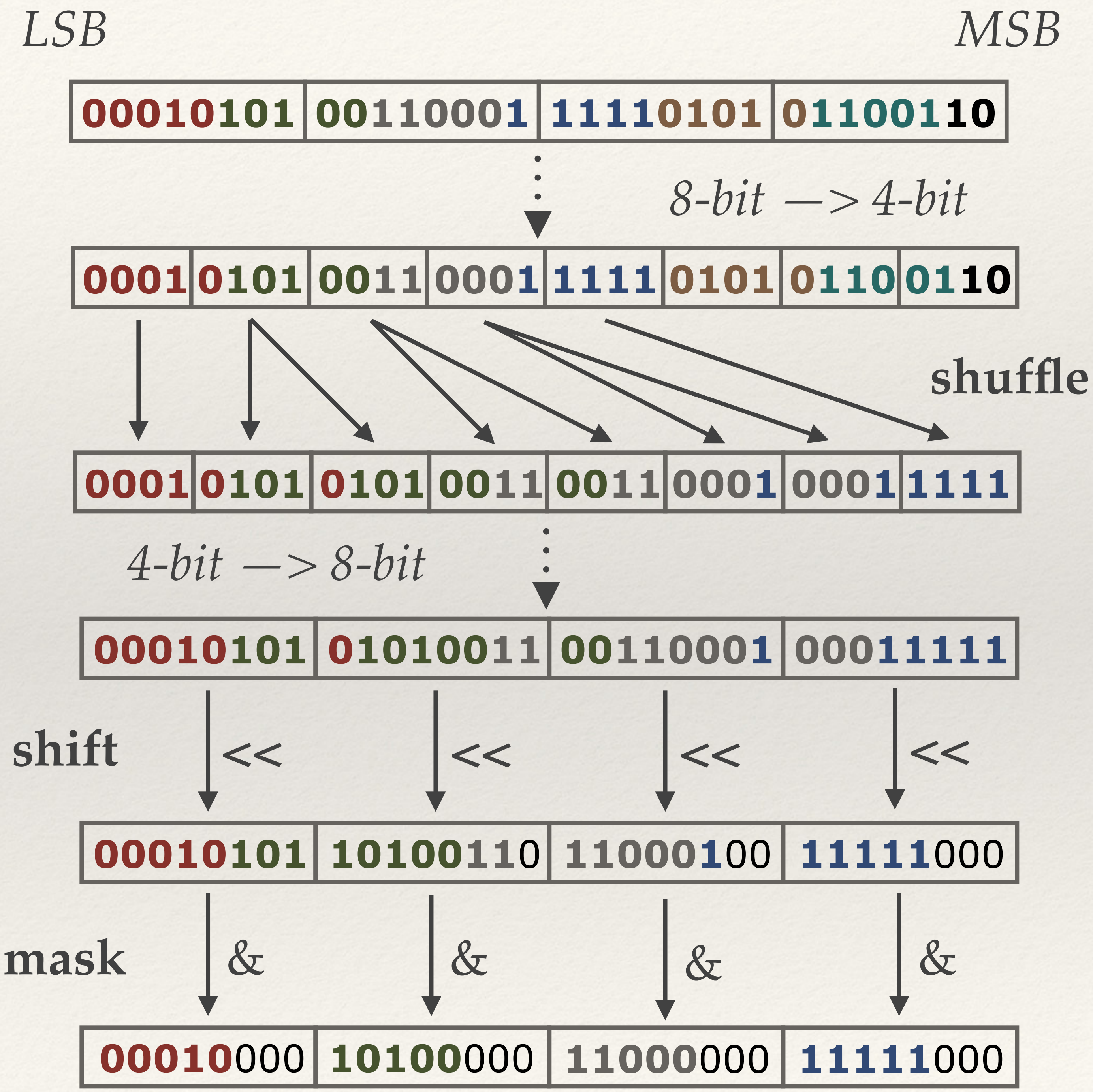
- ***Fully*** packed
  - No space wasted
    - Codes can ***span*** across 2 packed words
  - Packing
    - Process 1 unpacked ***code*** per iteration
    - Branch to store output packed ***word***
  - Unpacking
    - Process 1 output ***code*** per iteration
    - Branch to load input packed ***word***



Horizontal Layout

- *Fully* packed
  - No space wasted
    - Codes can *span* across 2 packed words
  - Packing
    - Process 1 unpacked ***code*** per iteration
    - Branch to store output packed ***word***
  - Unpacking
    - Process 1 output ***code*** per iteration
    - Branch to load input packed ***word***
    - Can be written in ***SIMD*** !

Based on paper by
T. Willhalm et al.
@ VLDB 2009
(& improved using
latest SIMD ISA)



| 00010101 | 00110001 | 11110101 | 01100110 |

| 0001 | 0101 | 0011 | 0001 | 1111 | 0101 | 0110 | 0110 |

| 0001 | 0101 | 0101 | 0011 | 0011 | 0001 | 0001 | 1111 |

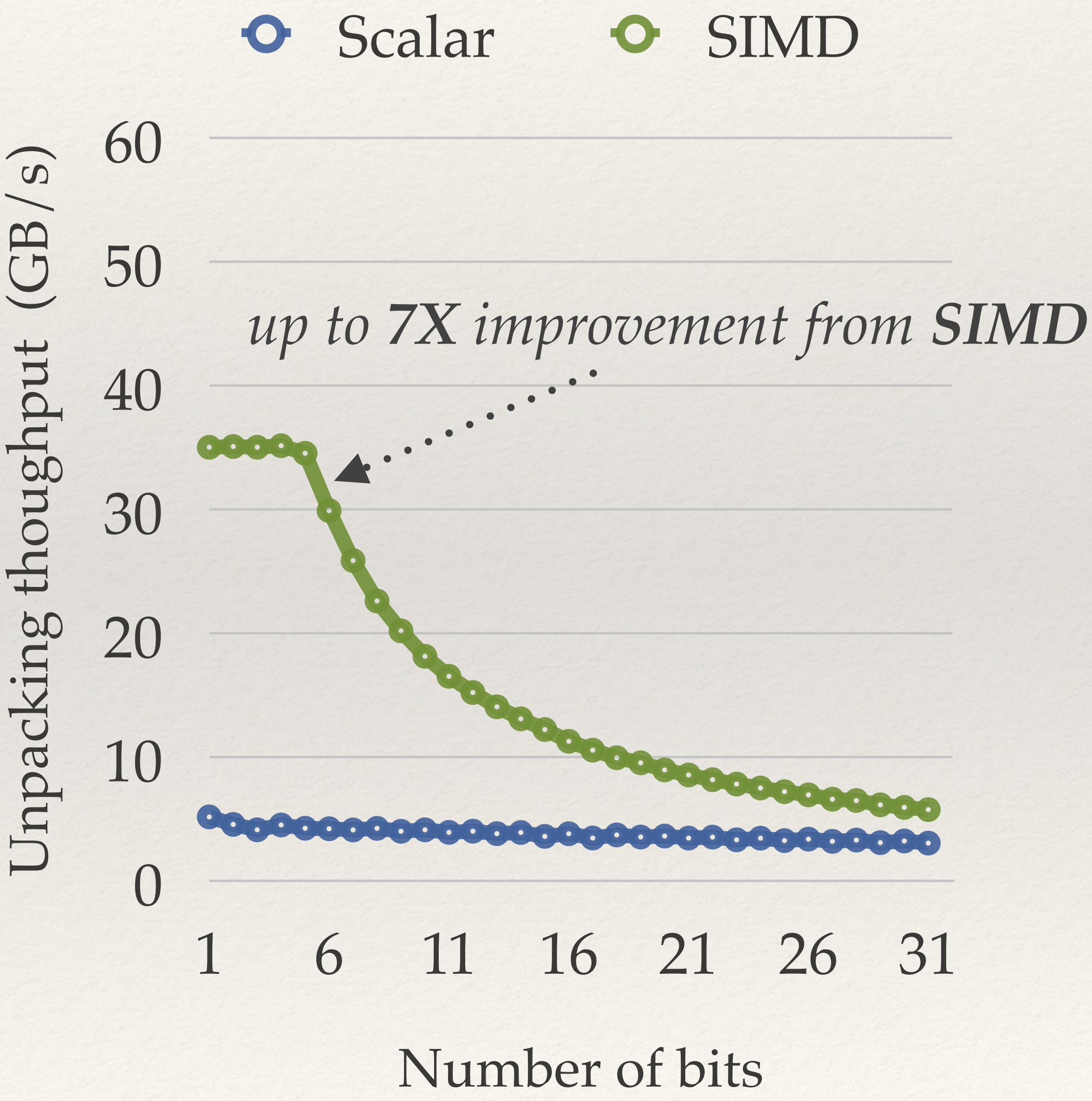
| 00010101 | 01010011 | 00110001 | 00011111 |

| 00010101 | 10100110 | 11000100 | 11111000 |

| 00010000 | 10100000 | 11000000 | 11111000 |

Horizontal Layout

- ***Fully*** packed
  - No space wasted
    - Codes can ***span*** across 2 packed words
  - Packing
    - Process 1 unpacked ***code*** per iteration
    - Branch to store output packed ***word***
  - Unpacking
    - Process 1 output ***code*** per iteration
    - Branch to load input packed ***word***
    - Can be written in ***SIMD*** **!**

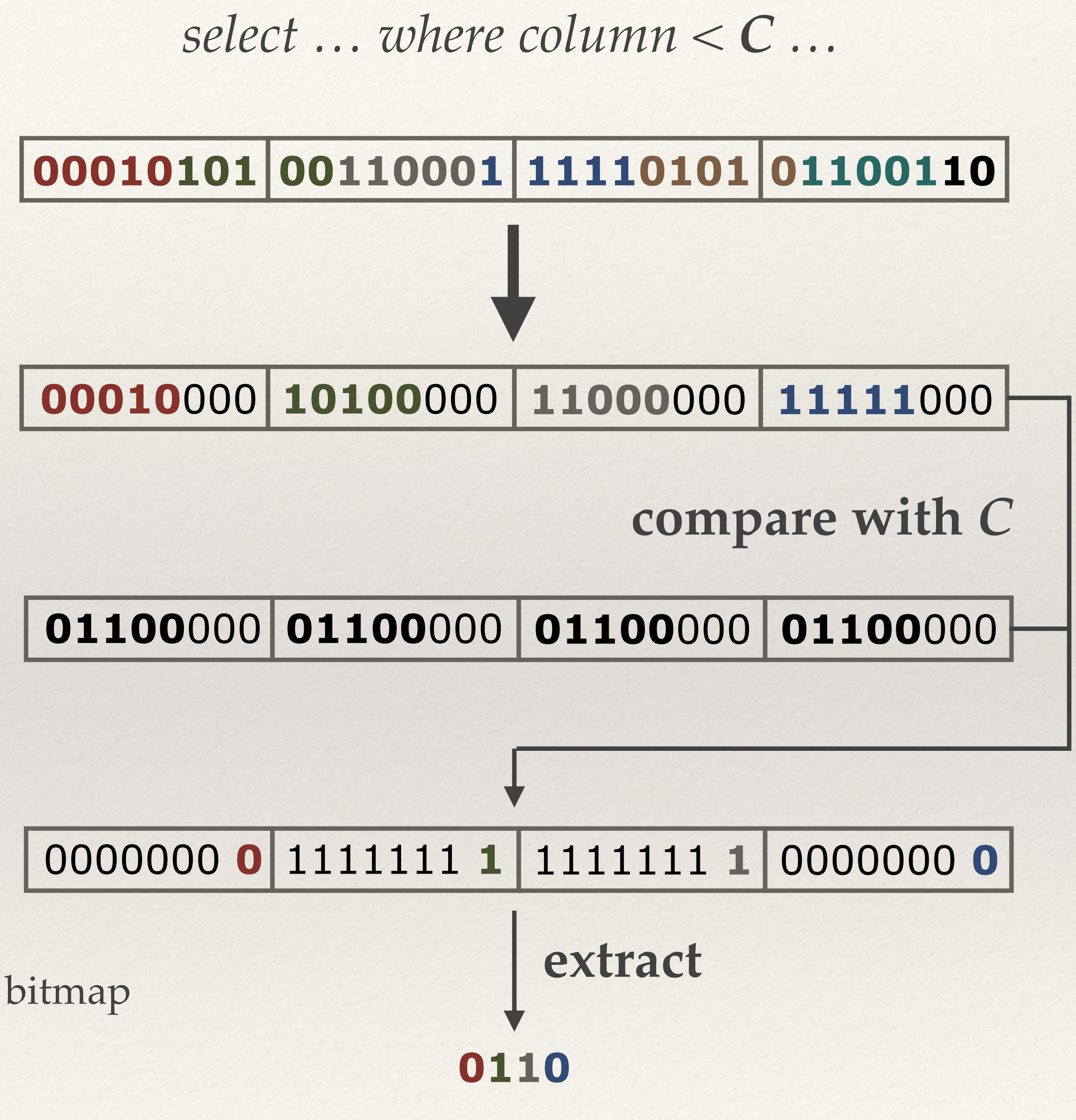


Horizontal Layout

- ***Fully*** packed
    - No space wasted
        - Codes can ***span*** across 2 packed words
    - Packing
        - Process 1 unpacked ***code*** per iteration
        - Branch to store output packed ***word***
    - Unpacking
        - Process 1 output ***code*** per iteration
        - Branch to load input packed ***word***
        - Can be written in ***SIMD*** **!**
    - Scanning
        - ***Unpack*** the codes in CPU registers
        - ***Evaluate*** selective predicates and append to bitmap
        - Must unpack first thus bounded by ***O(n)***

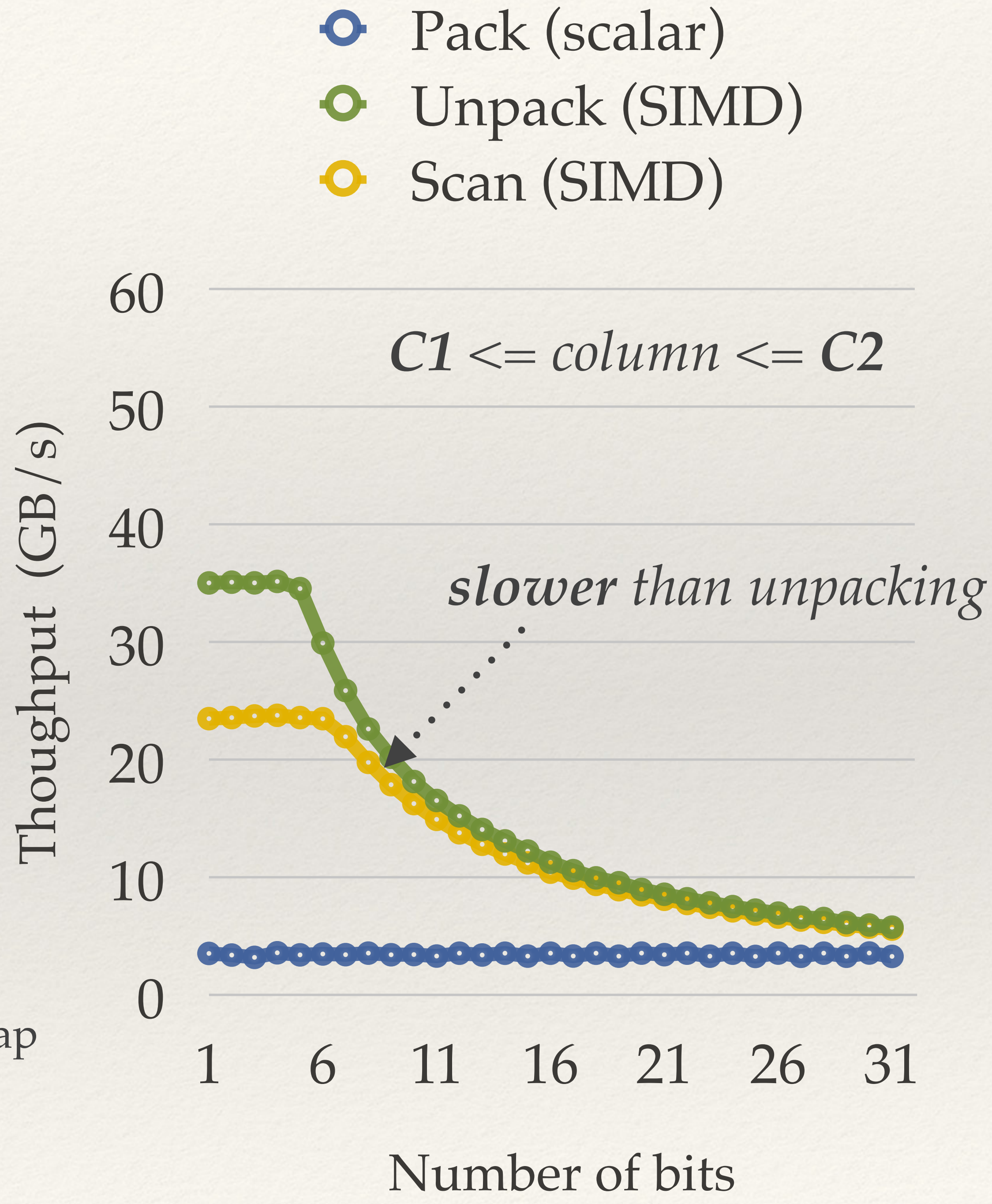
Horizontal Layout

- ***Fully*** **packed**
  - No space wasted
    - Codes can ***span*** across 2 packed words
  - Packing
    - Process 1 unpacked ***code*** per iteration
    - Branch to store output packed ***word***
  - Unpacking
    - Process 1 output ***code*** per iteration
    - Branch to load input packed ***word***
    - Can be written in ***SIMD*** **!**
  - Scanning
    - ***Unpack*** the codes in CPU registers
    - ***Evaluate*** selective predicates and append to bitmap
    - Must unpack first thus bounded by ***O(n)***
    - Can also be written in ***SIMD*** via SIMD unpacking



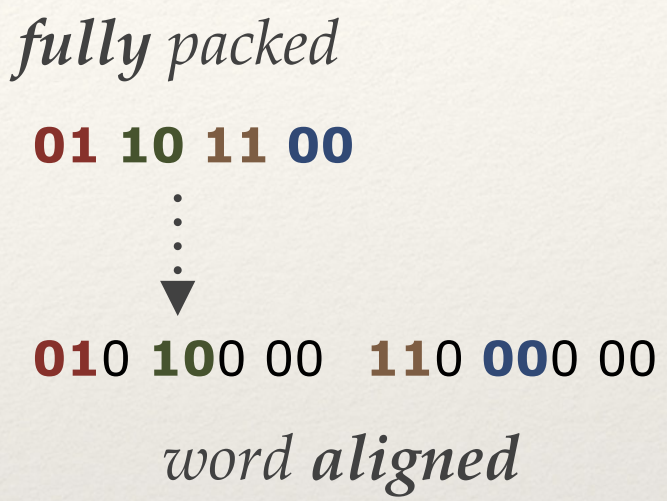
Horizontal Layout

- ***Fully*** packed
  - No space wasted
    - Codes can ***span*** across 2 packed words
  - Packing
    - Process 1 unpacked ***code*** per iteration
    - Branch to store output packed ***word***
  - Unpacking
    - Process 1 output ***code*** per iteration
    - Branch to load input packed ***word***
    - Can be written in ***SIMD*** **!**
  - Scanning
    - ***Unpack*** the codes in CPU registers
    - ***Evaluate*** selective predicates and append to bitmap
    - Must unpack first thus bounded by ***O(n)***
    - Can also be written in ***SIMD*** via SIMD unpacking



Horizontal Layout

- Word ***aligned***
  - Waste space to get alignment
    - Pack $b' = w / (b+1)$ codes per processor word
    - Extra bit per word used for ***scanning***

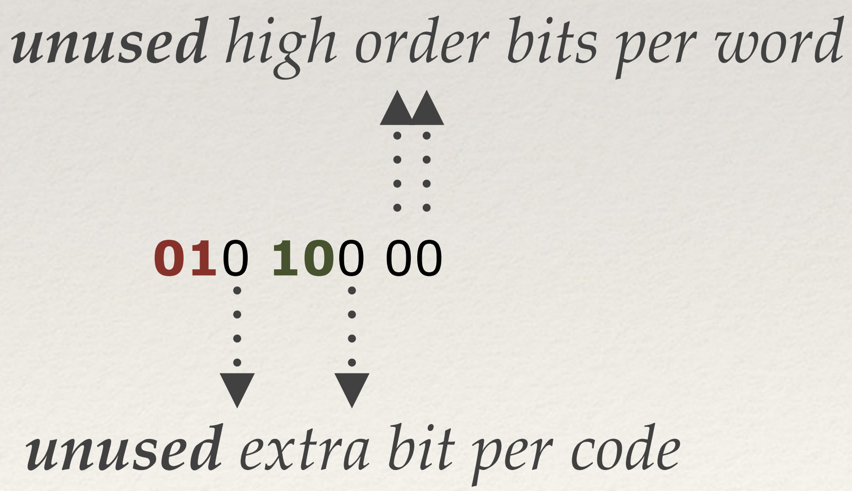


fully *packed*

01 10 11 00

010 100 00 110 000 00

word ***aligned***



unused *high order bits per word*

010 100 00

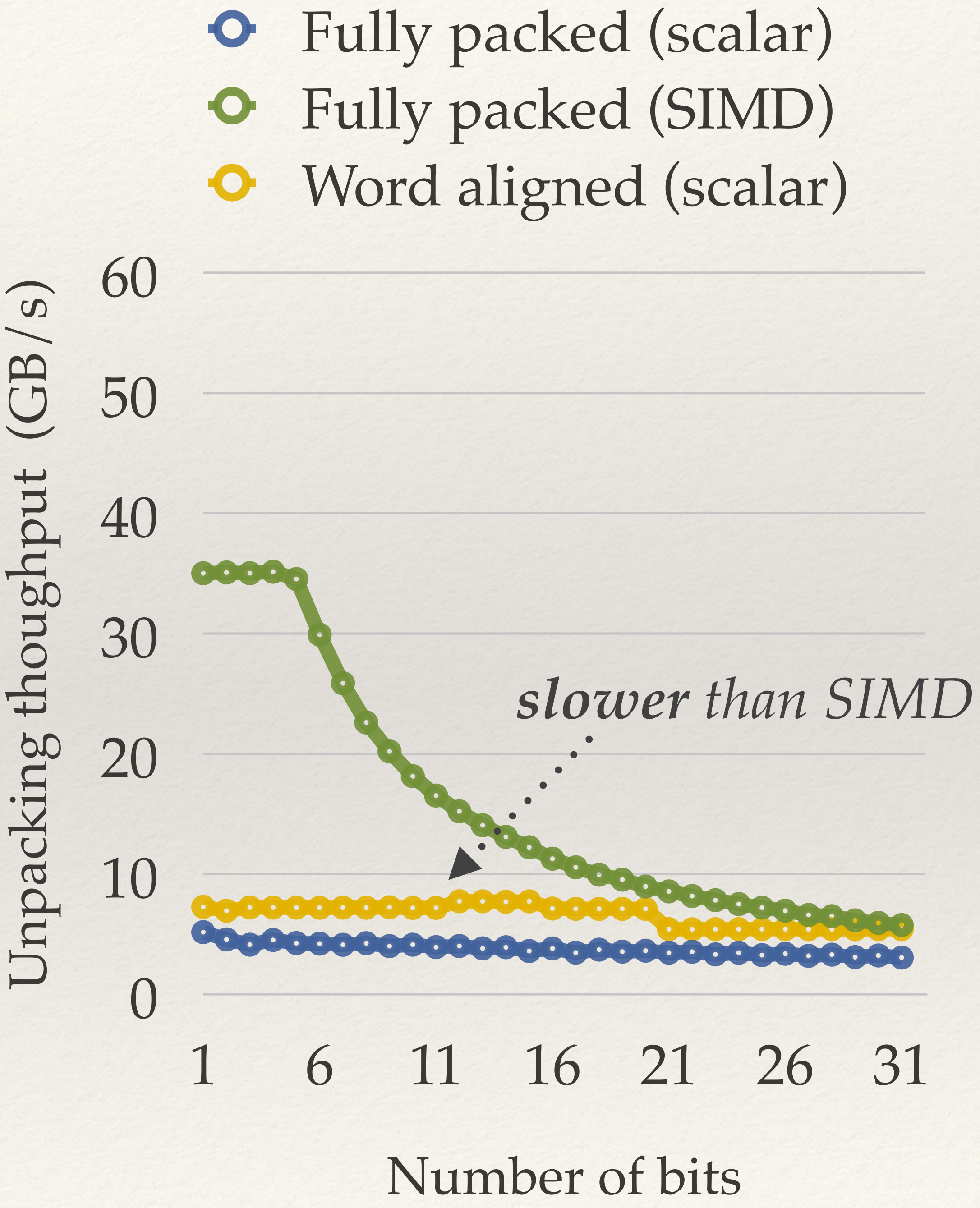
unused *extra bit per code*

Horizontal Layout

- Word ***aligned***
  - Waste space to get alignment
    - Pack $\boldsymbol{b' = w / (b+1)}$ codes per processor word
    - Extra bit per word used for ***scanning***
  - Packing
    - 1 packed word at a time
    - Nested loop to pack $\boldsymbol{b'}$ codes

Horizontal Layout

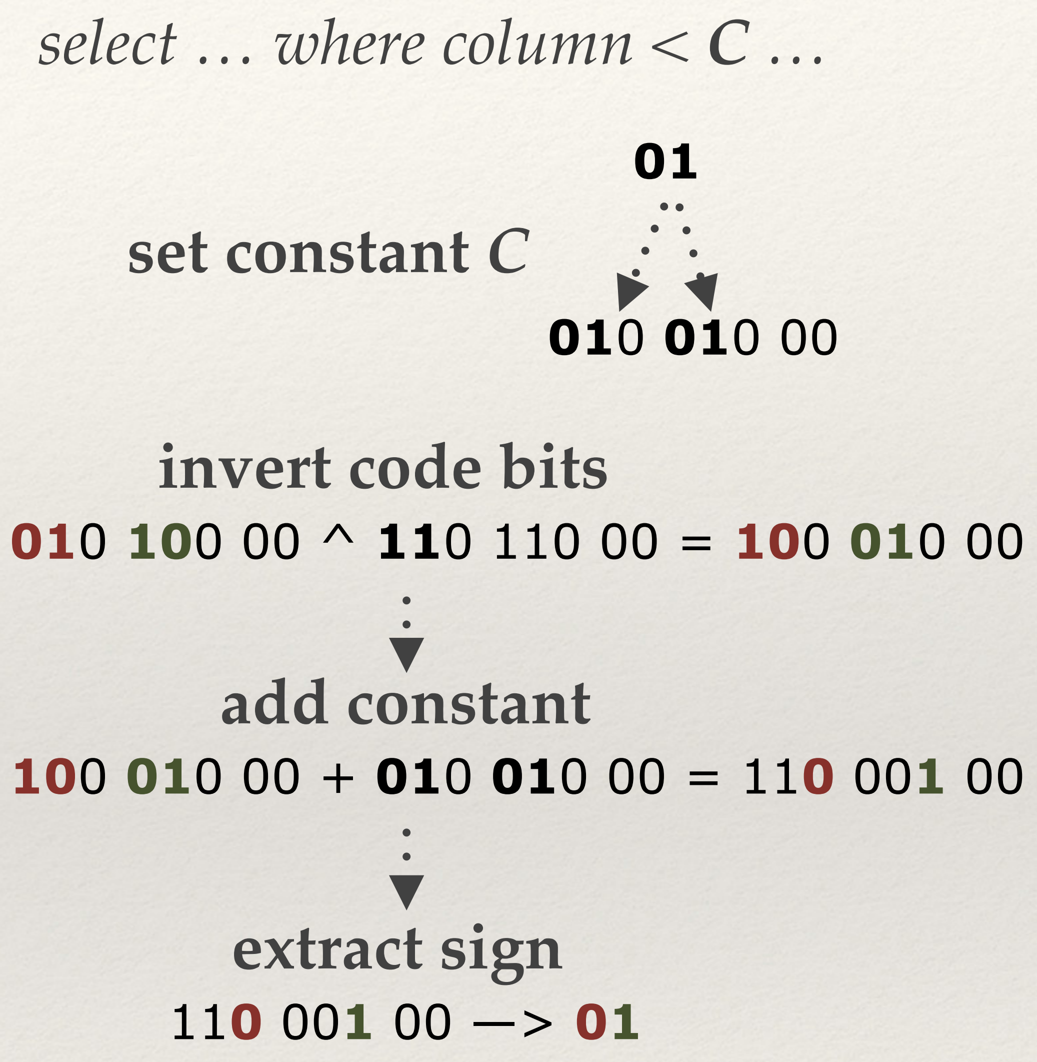
- Word ***aligned***
  - Waste space to get alignment
    - Pack ***b′** = **w** / (**b**+1)* codes per processor word
    - Extra bit per word used for ***scanning***
  - Packing
    - 1 packed word at a time
    - Nested loop to pack ***b′*** codes
  - Unpacking
    - 1 packed word at a time
    - Nested loop to unpack ***b′*** codes



slower *than SIMD*

Horizontal Layout

- Word ***aligned***
  - Waste space to get alignment
    - Pack $b' = w / (b+1)$ codes per processor word
    - Extra bit per word used for ***scanning***
  - Packing
    - 1 packed word at a time
    - Nested loop to pack b' codes
  - Unpacking
    - 1 packed word at a time
    - Nested loop to unpack b' codes
  - Scanning
    - Evaluate predicates ***without*** unpacking
    - Works with ***simple*** order predicates: <,=,>
    - Boolean result in ***overflow*** bit of ***b***-bit arithmetic
    - Executing < $O(n)$ operations



*select … where column < **C** …*

set constant *C*

01

010 010 00

invert code bits

010 100 00 ^ 110 110 00 = 100 010 00

add constant

100 010 00 + 010 010 00 = 110 001 00

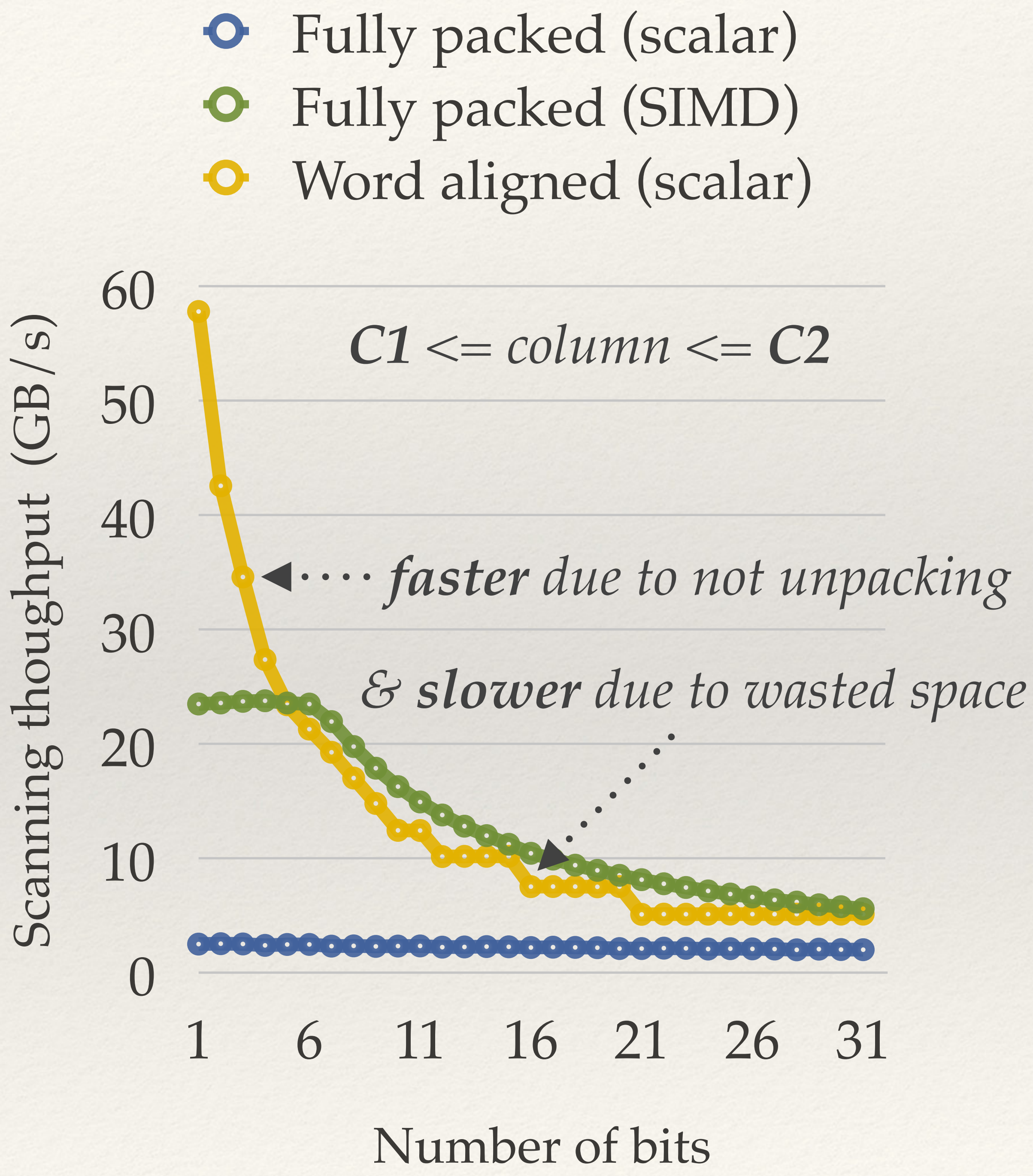
extract sign

110 001 00 —> 01

Based on paper by
Leslie Lamport
@ CACM 1975

Horizontal Layout

- Word ***aligned***
  - Waste space to get alignment
    - Pack ***b′ = w / (b+1)*** codes per processor word
    - Extra bit per word used for ***scanning***
  - Packing
    - 1 packed word at a time
    - Nested loop to pack ***b′*** codes
  - Unpacking
    - 1 packed word at a time
    - Nested loop to unpack ***b′*** codes
  - Scanning
    - Evaluate predicates ***without*** unpacking
    - Works with ***simple*** order predicates: <,=,>
    - Boolean result in ***overflow*** bit of ***b***-bit arithmetic
    - Executing **< *O(n)*** operations

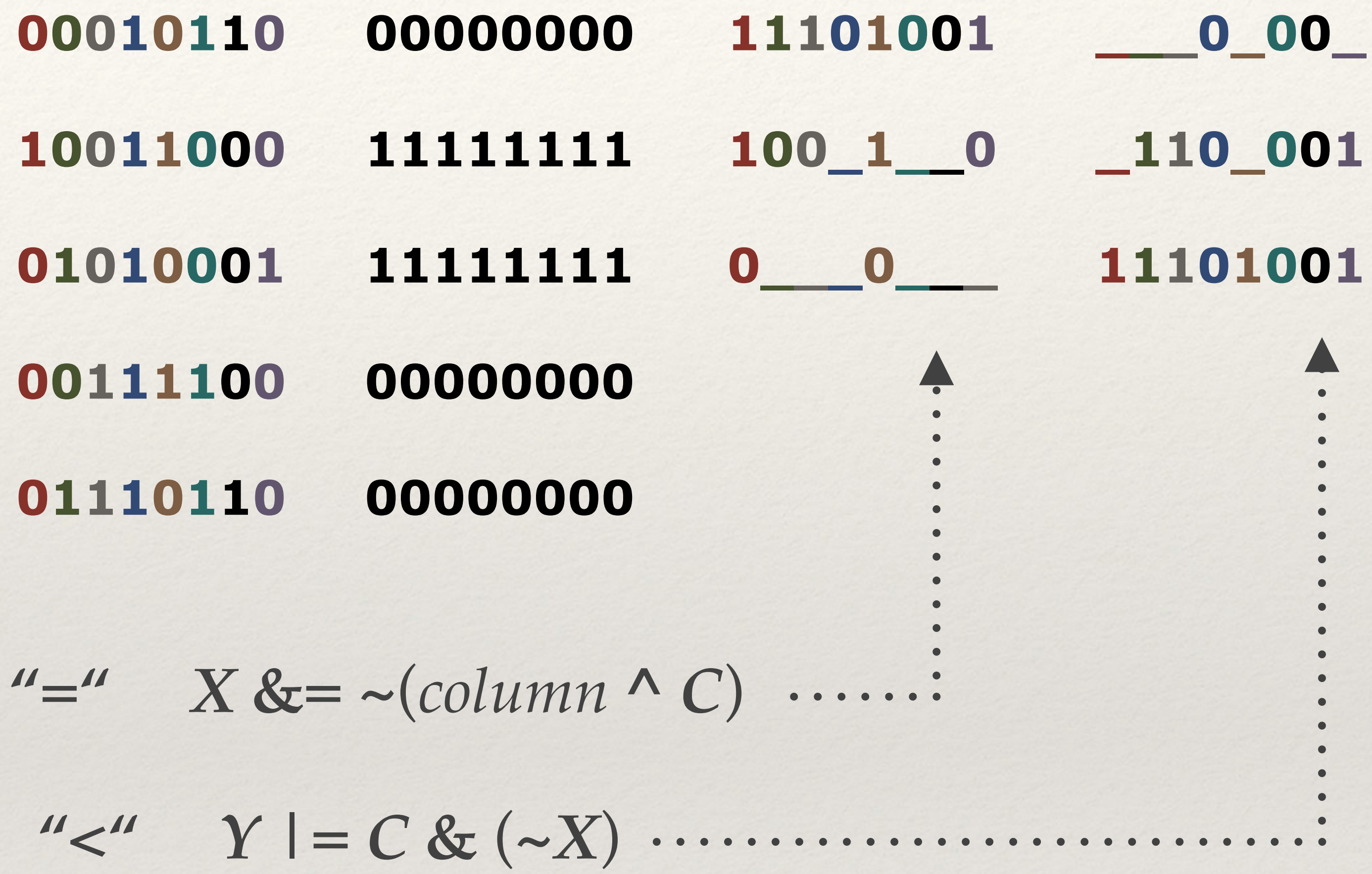


Vertical Layout

- ***Fully*** packed & word ***aligned***
  - Interleave bits of ***k*** codes
    - *k* divides the processor word

Vertical Layout

- *Fully* packed & word ***aligned***
  - Interleave bits of ***k*** codes
    - *k* divides the processor word
  - Scanning
    - Evaluate ***without*** unpacking
    - Can ***skip*** words early



```
00010110    00000000    11101001    ___0_00_
10011000    11111111    100_1__0    _110_001
01010001    11111111    0___0___    11101001
00111100    00000000
01110110    00000000
```

“=“ $X \mathrel{\&}= \sim(\mathit{column} \;\hat{}\; C)$

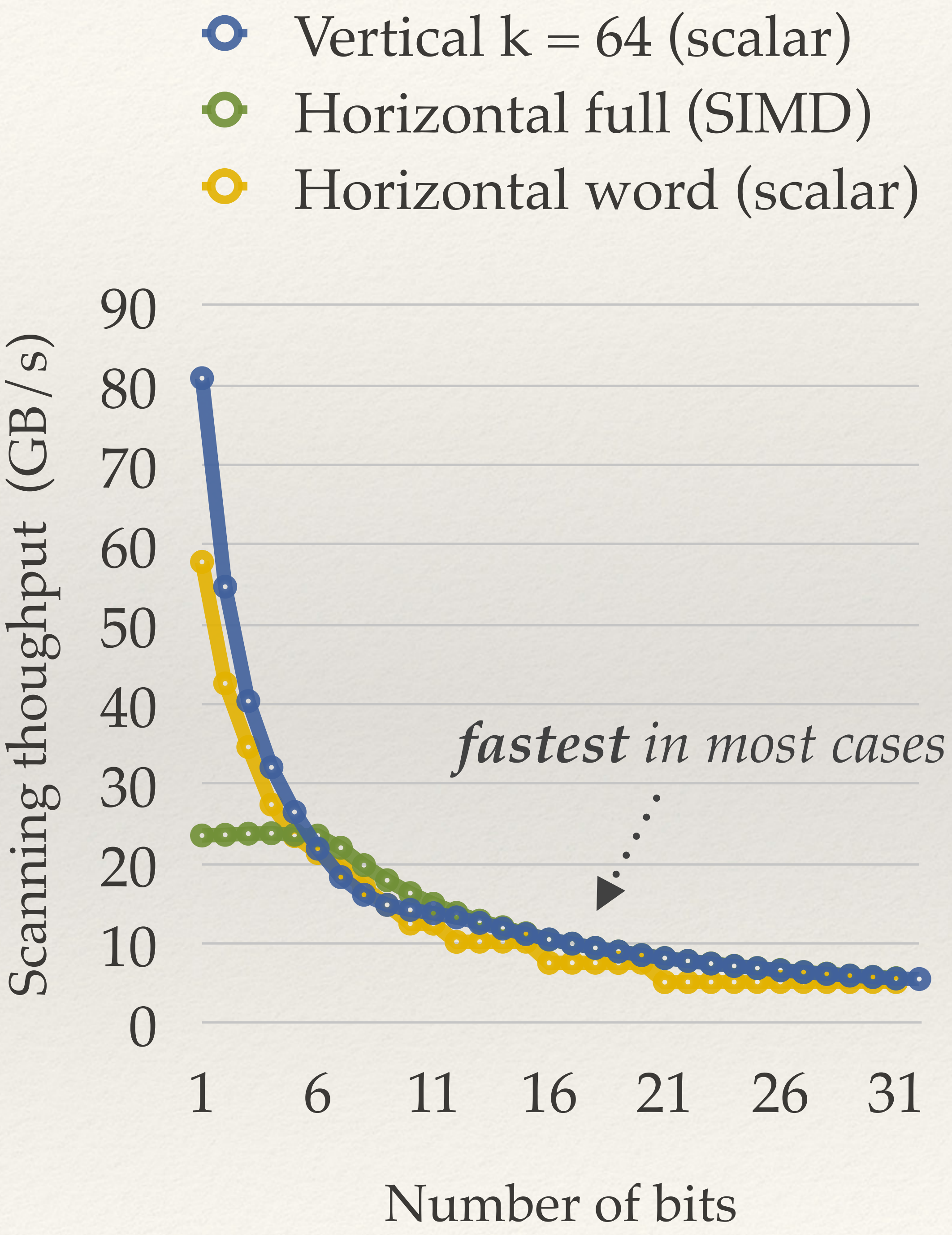
“<“ $Y \mathrel{|}= C \mathbin{\&} (\sim X)$

stop if $X = 0$

Based on paper by
Y. Li et al.
@ SIGMOD 2013

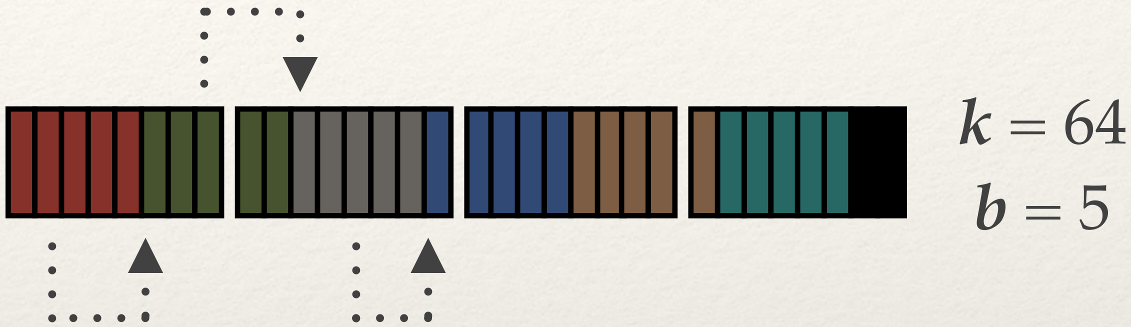
Vertical Layout

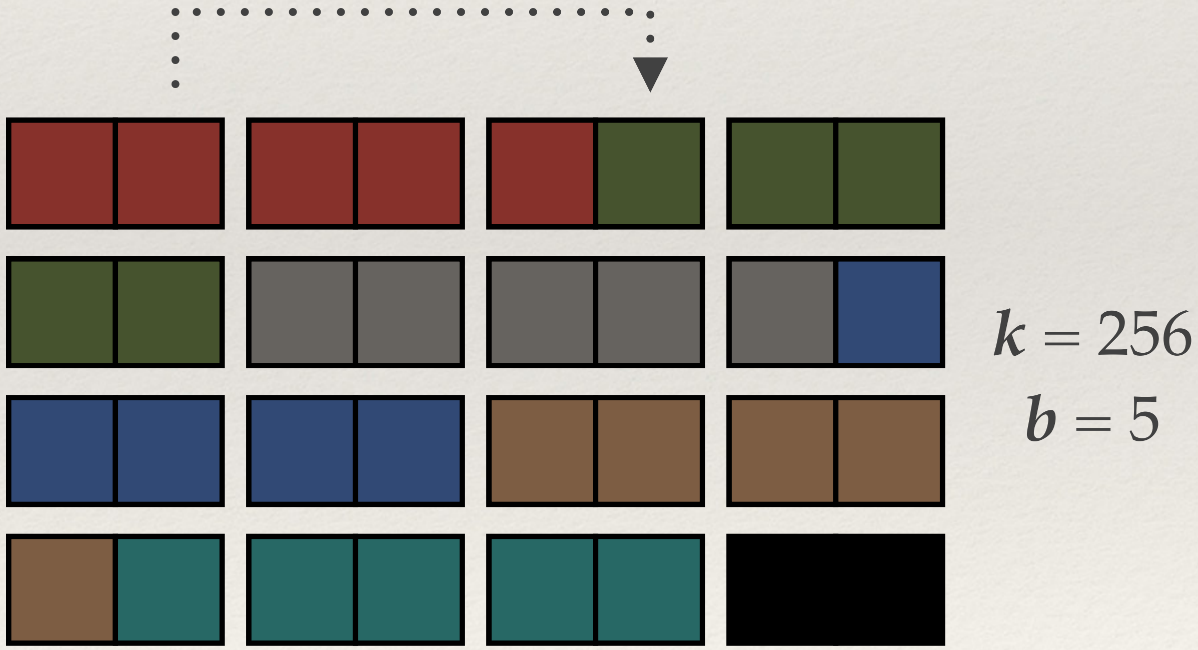
- ***Fully*** packed & word ***aligned***
  - Interleave bits of ***k*** codes
    - ***k*** divides the processor word
  - Scanning
    - Evaluate ***without*** unpacking
    - Can ***skip*** words early



Vertical Layout

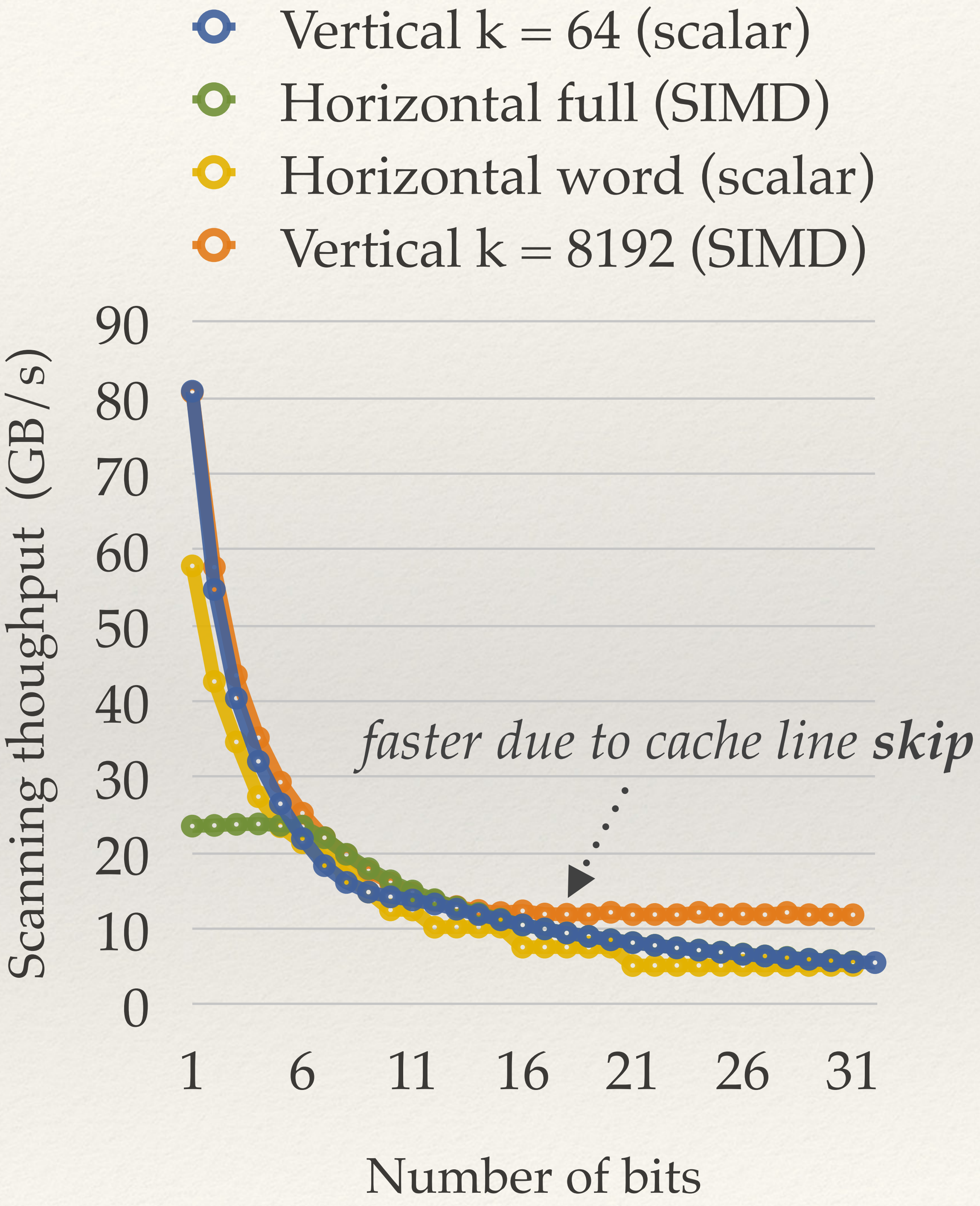
- ***Fully*** packed & word ***aligned***
  - Interleave bits of k codes
    - k divides the processor word
  - Scanning
    - Evaluate ***without*** unpacking
    - Can ***skip*** words early
    - Increase k to minimize false (pre)fetches





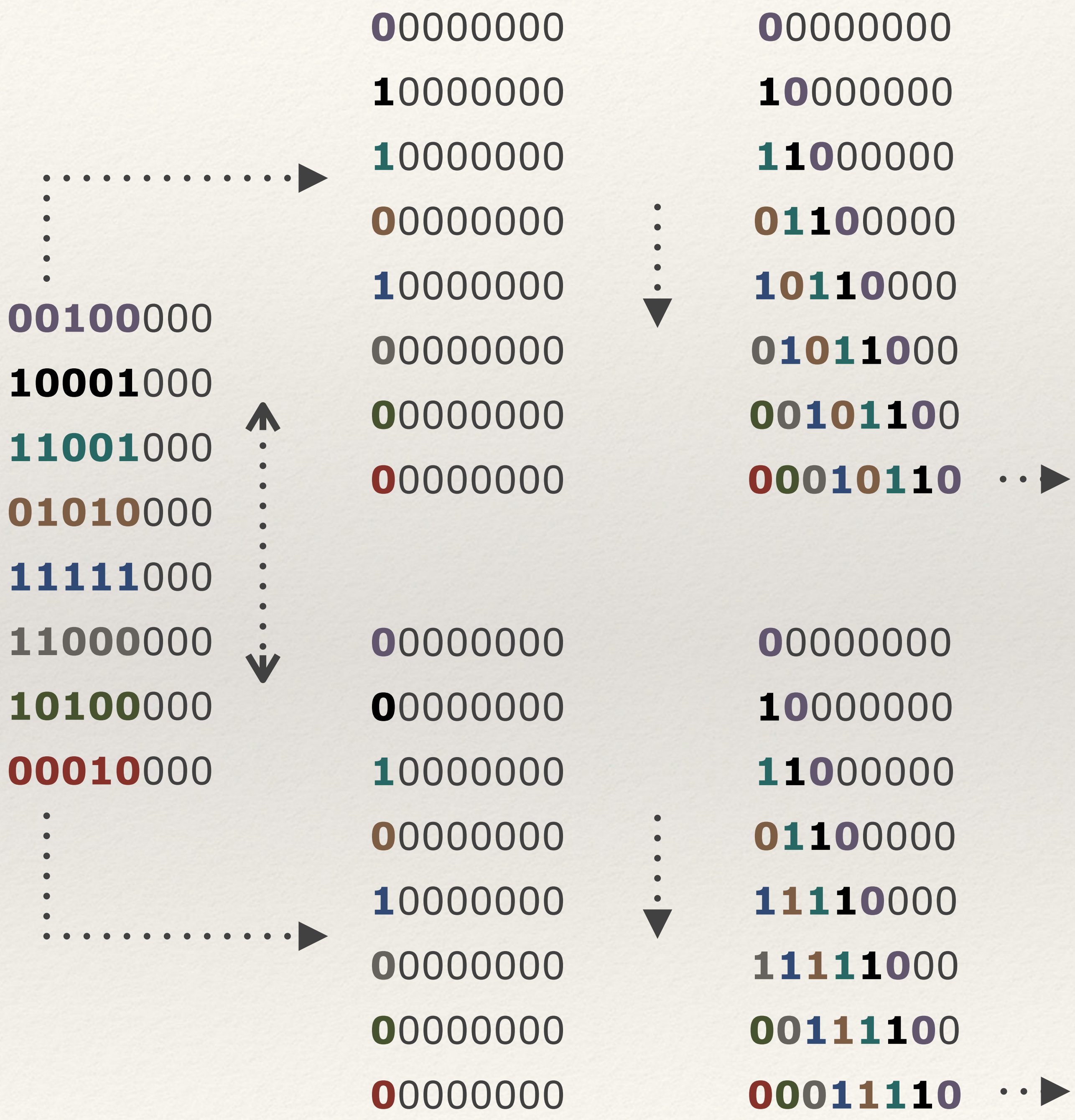
Vertical Layout

- ***Fully*** packed & word ***aligned***
  - Interleave bits of ***k*** codes
    - ***k*** divides the processor word
  - Scanning
    - Evaluate ***without*** unpacking
    - Can ***skip*** words early
    - Increase ***k*** to minimize false (pre)fetches



Vertical Layout

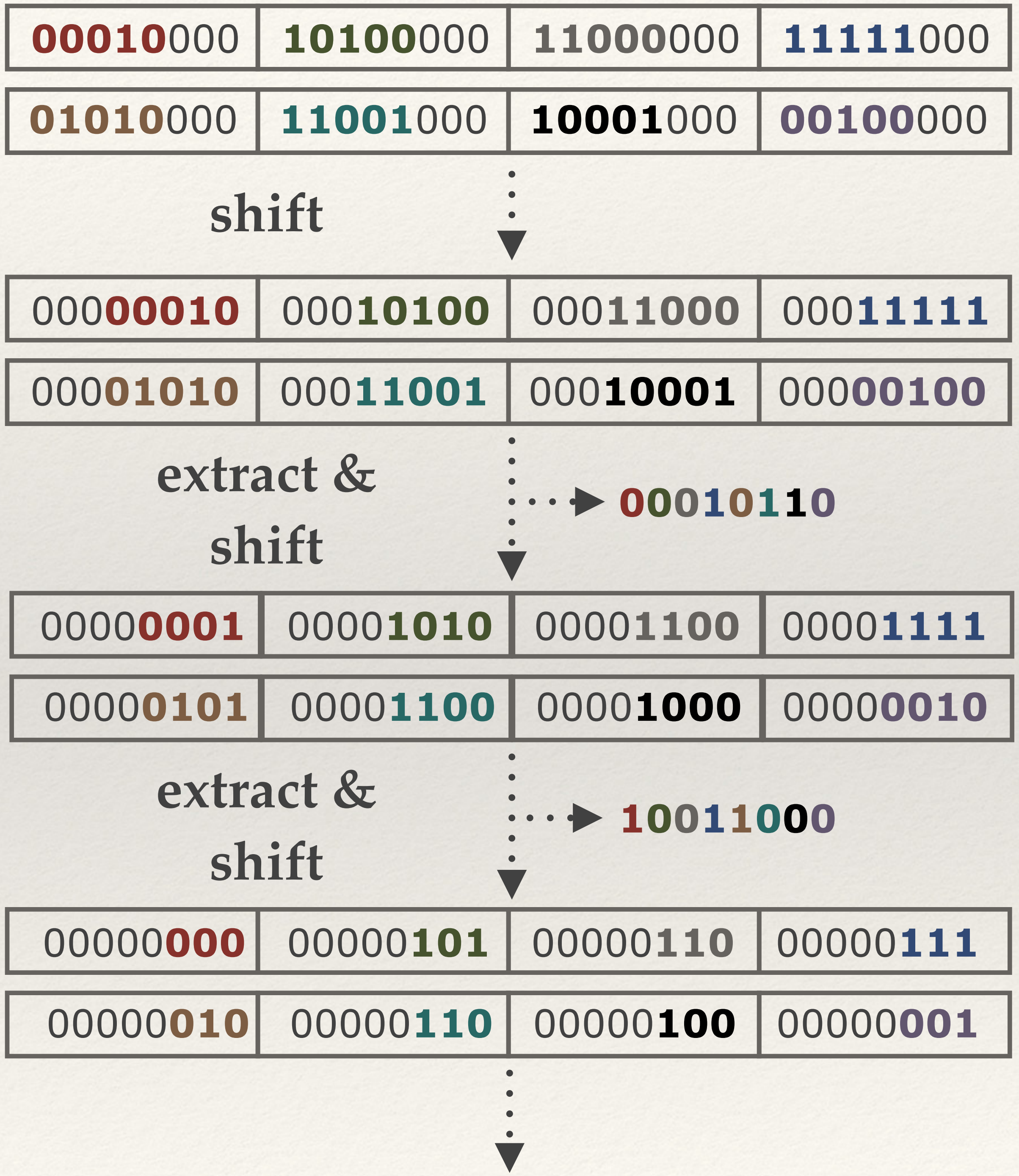
- ***Fully*** packed & word ***aligned***
  - Interleave bits of ***k*** codes
    - ***k*** divides the processor word
  - Scanning
    - Evaluate ***without*** unpacking
    - Can ***skip*** words early
    - Increase ***k*** to minimize false (pre)fetches
  - Packing
    - Transfer ***nb*** bits across registers



```
                   00000000      00000000
                   10000000      10000000
                   10000000      11000000
                   00000000      01100000
                   10000000      10110000
00100000           00000000      01011000
10001000           00000000      00101100
11001000           00000000      00010110
01010000
11111000
11000000           00000000      00000000
10100000           00000000      10000000
00010000           10000000      11000000
                   00000000      01100000
                   10000000      11110000
                   00000000      11111000
                   00000000      00111100
                   00000000      00011110
```

Vertical Layout

- ***Fully*** packed & word ***aligned***
  - Interleave bits of ***k*** codes
    - ***k*** divides the processor word
  - Scanning
    - Evaluate ***without*** unpacking
    - Can ***skip*** words early
    - Increase ***k*** to minimize false (pre)fetches
  - Packing
    - Transfer ***nb*** bits across registers
    - Can be written in ***SIMD*** !



00010000	10100000	11000000	11111000
01010000	11001000	10001000	00100000

shift

00000010	00010100	00011000	00011111
00001010	00011001	00010001	00000100

extract & shift

00010110

00000001	00001010	00001100	00001111
00000101	00001100	00001000	00000010

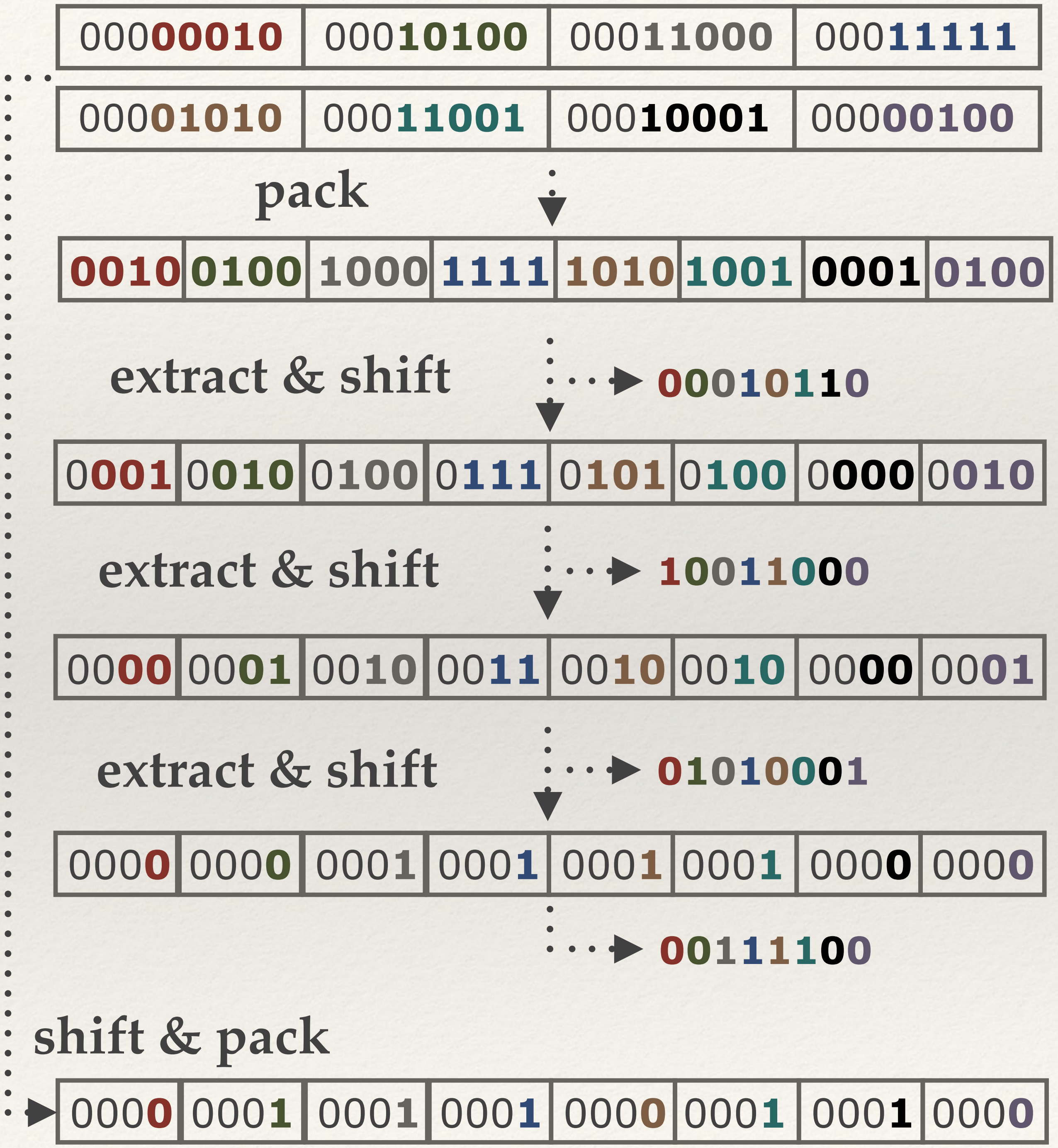
extract & shift

10011000

00000000	00000101	00000110	00000111
00000010	00000110	00000100	00000001

Vertical Layout

- ***Fully*** packed & word ***aligned***
  - Interleave bits of ***k*** codes
    - ***k*** divides the processor word
  - Scanning
    - Evaluate ***without*** unpacking
    - Can ***skip*** words early
    - Increase ***k*** to minimize false (pre)fetches
  - Packing
    - Transfer ***nb*** bits across registers
    - Can be written in *SIMD* !
    - Extract bits ***per byte*** not per int



00000010	00010100	00011000	00011111
00001010	00011001	00010001	00000100

pack

0010	0100	1000	1111	1010	1001	0001	0100

extract & shift → 00010110

0001	0010	0100	0111	0101	0100	0000	0010

extract & shift → 10011000

0000	0001	0010	0011	0010	0010	0000	0001

extract & shift → 01010001

0000	0000	0001	0001	0001	0001	0000	0000

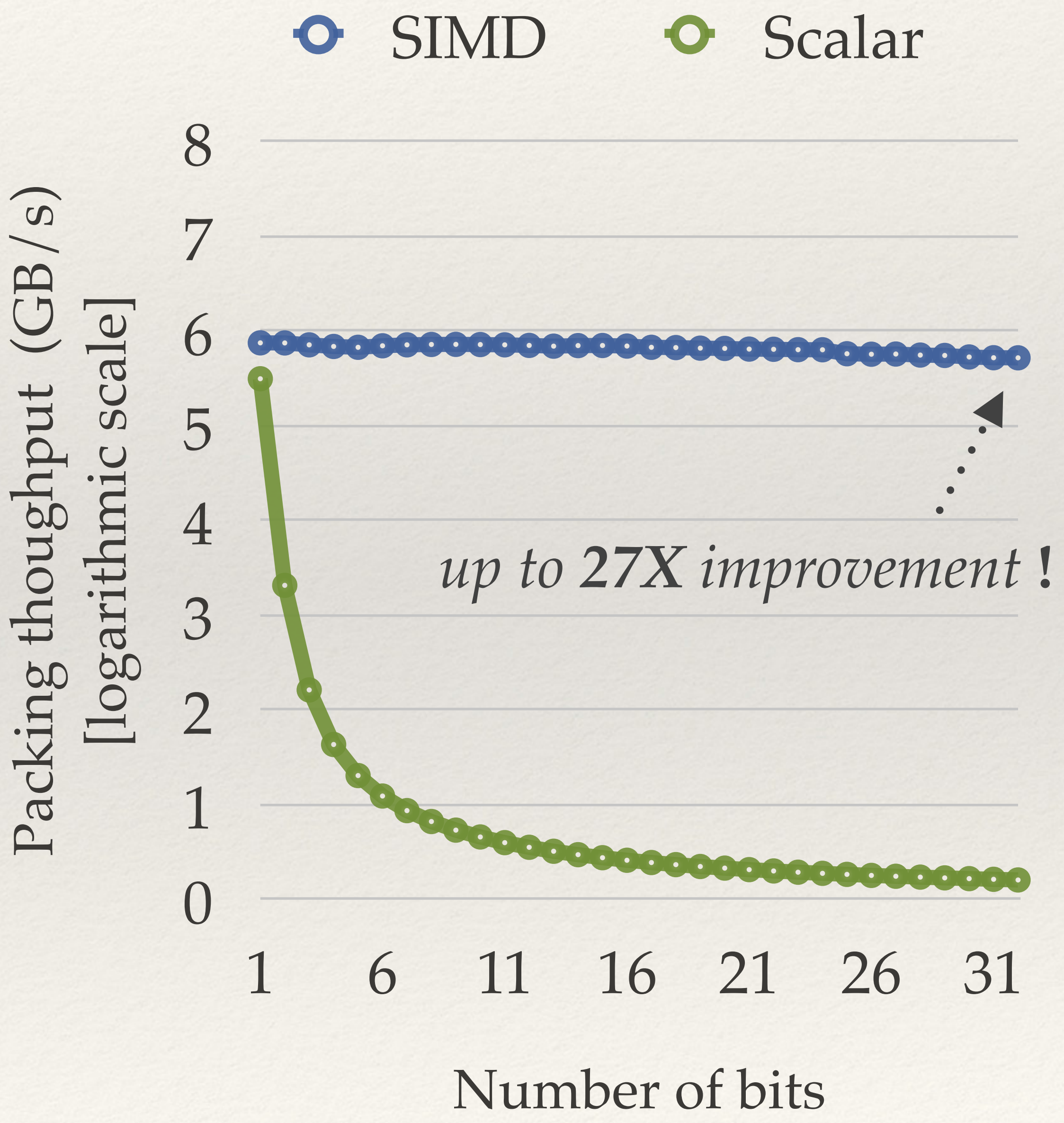
→ 00111100

shift & pack

0000	0001	0001	0001	0000	0001	0001	0000

Vertical Layout

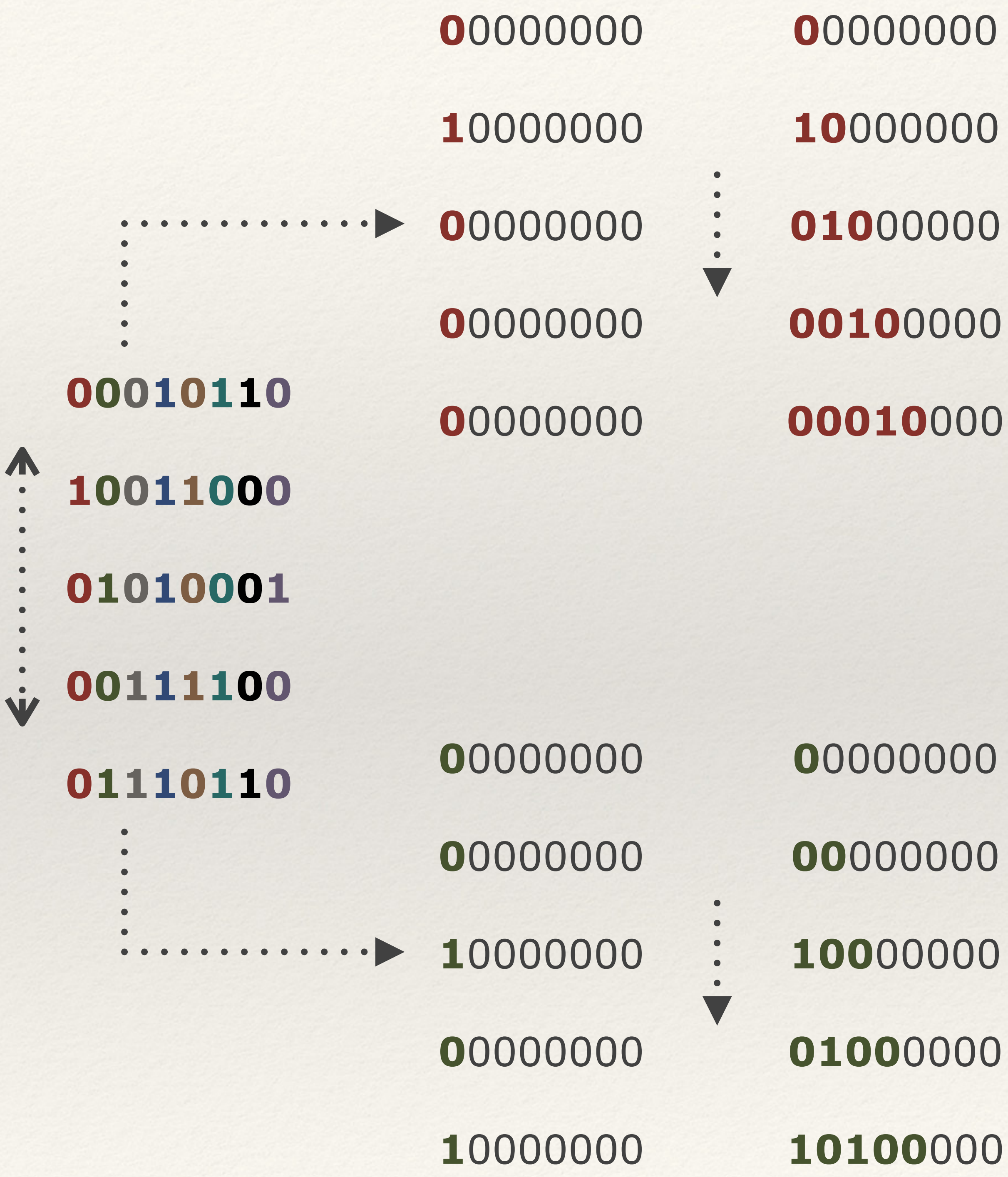
- ***Fully*** packed & word ***aligned***
    - Interleave bits of ***k*** codes
        - ***k*** divides the processor word
    - Scanning
        - Evaluate ***without*** unpacking
        - Can ***skip*** words early
        - Increase ***k*** to minimize false (pre)fetches
    - Packing
        - Transfer ***nb*** bits across registers
        - Can be written in ***SIMD*** **!**
        - Extract bits ***per byte*** not per int



*up to **27X** improvement* **!**

Vertical Layout

- ***Fully*** packed & word ***aligned***
  - Interleave bits of ***k*** codes
    - ***k*** divides the processor word
  - Scanning
    - Evaluate ***without*** unpacking
    - Can ***skip*** words early
    - Increase ***k*** to minimize false (pre)fetches
  - Packing
    - Transfer ***nb*** bits across registers
    - Can be written in ***SIMD*** !
    - Extract bits ***per byte*** not per int
  - Unpacking
    - Transfer ***nb*** bits across registers



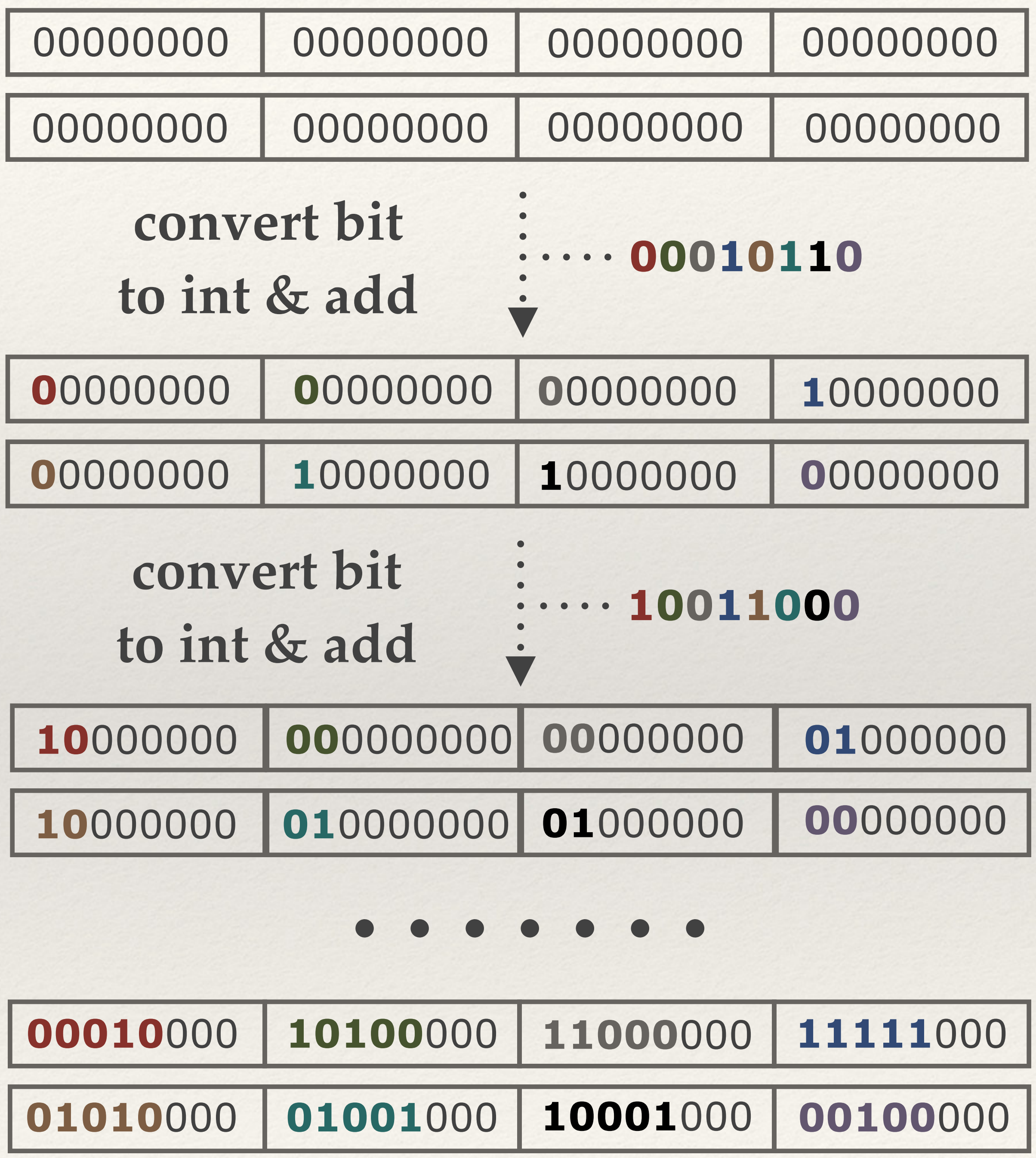
00000000 00000000
10000000 10000000
00000000 01000000
00000000 00100000
00000000 00010000

00010110
10011000
01010001
00111100
01110110

00000000 00000000
00000000 00000000
10000000 10000000
00000000 01000000
10000000 10100000

Vertical Layout

- ***Fully*** packed & word ***aligned***
  - Interleave bits of ***k*** codes
    - ***k*** divides the processor word
  - Scanning
    - Evaluate ***without*** unpacking
    - Can ***skip*** words early
    - Increase ***k*** to minimize false (pre)fetches
  - Packing
    - Transfer ***nb*** bits across registers
    - Can be written in ***SIMD*** **!**
    - Extract bits ***per byte*** not per int
  - Unpacking
    - Transfer ***nb*** bits across registers
    - Can be written in ***SIMD*** **!**



00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000

convert bit
to int & add

00010110

00000000 00000000 00000000 10000000
00000000 10000000 10000000 00000000

convert bit
to int & add

10011000

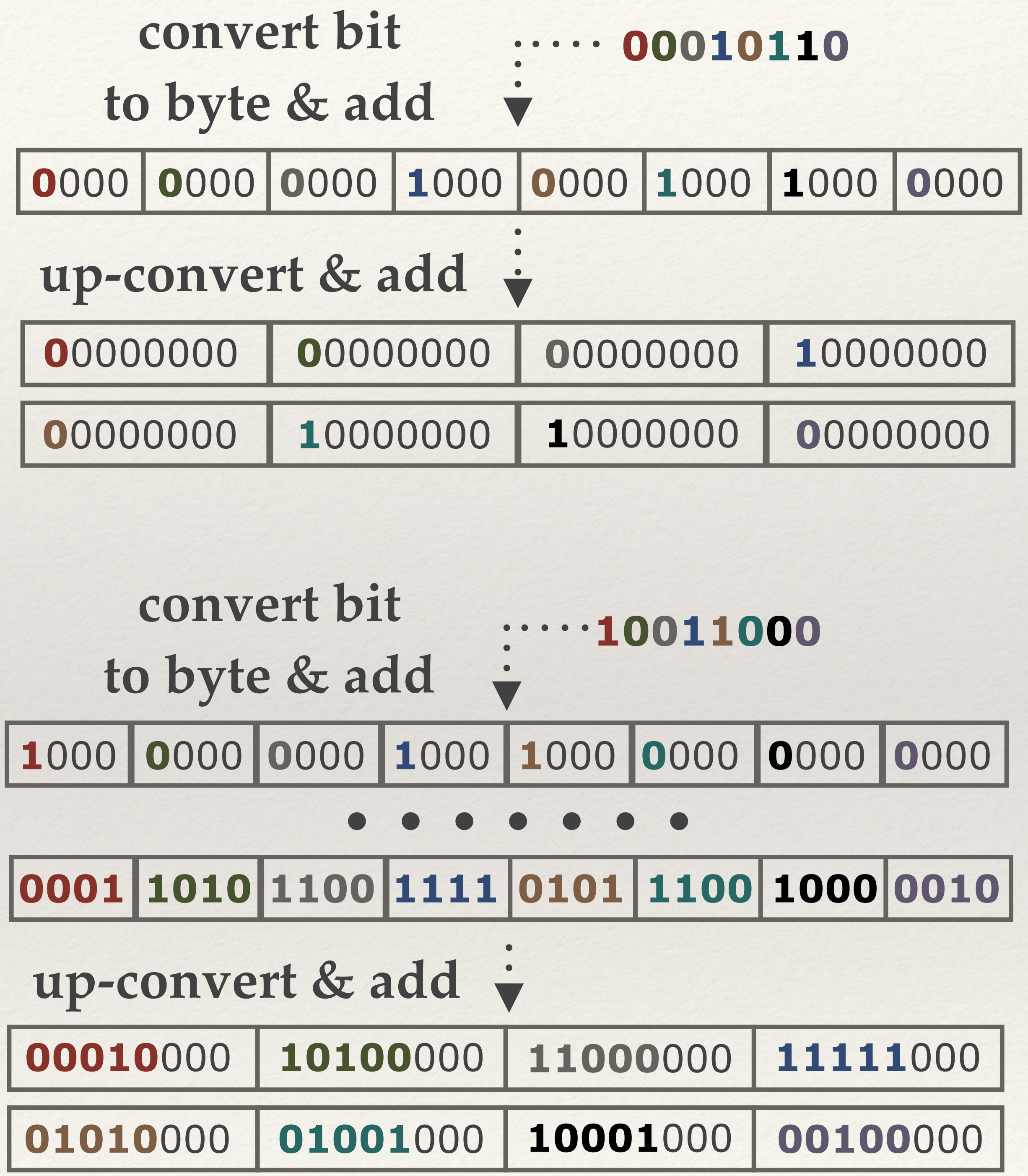
10000000 000000000 00000000 01000000
10000000 010000000 01000000 00000000

• • • • • • •

00010000 10100000 11000000 11111000
01010000 01001000 10001000 00100000

Vertical Layout

- ***Fully*** packed & word ***aligned***
  - Interleave bits of ***k*** codes
    - ***k*** divides the processor word
  - Scanning
    - Evaluate ***without*** unpacking
    - Can ***skip*** words early
    - Increase ***k*** to minimize false (pre)fetches
  - Packing
    - Transfer ***nb*** bits across registers
    - Can be written in ***SIMD*** **!**
    - Extract bits ***per byte*** not per int
  - Unpacking
    - Transfer ***nb*** bits across registers
    - Can be written in ***SIMD*** **!**
    - Insert bits ***per byte*** not per int



convert bit to byte & add

00010110

0000	0000	0000	1000	0000	1000	1000	0000

up-convert & add

00000000	00000000	00000000	10000000
00000000	10000000	10000000	00000000

convert bit to byte & add

10011000

1000	0000	0000	1000	1000	0000	0000	0000

● ● ● ● ● ● ●

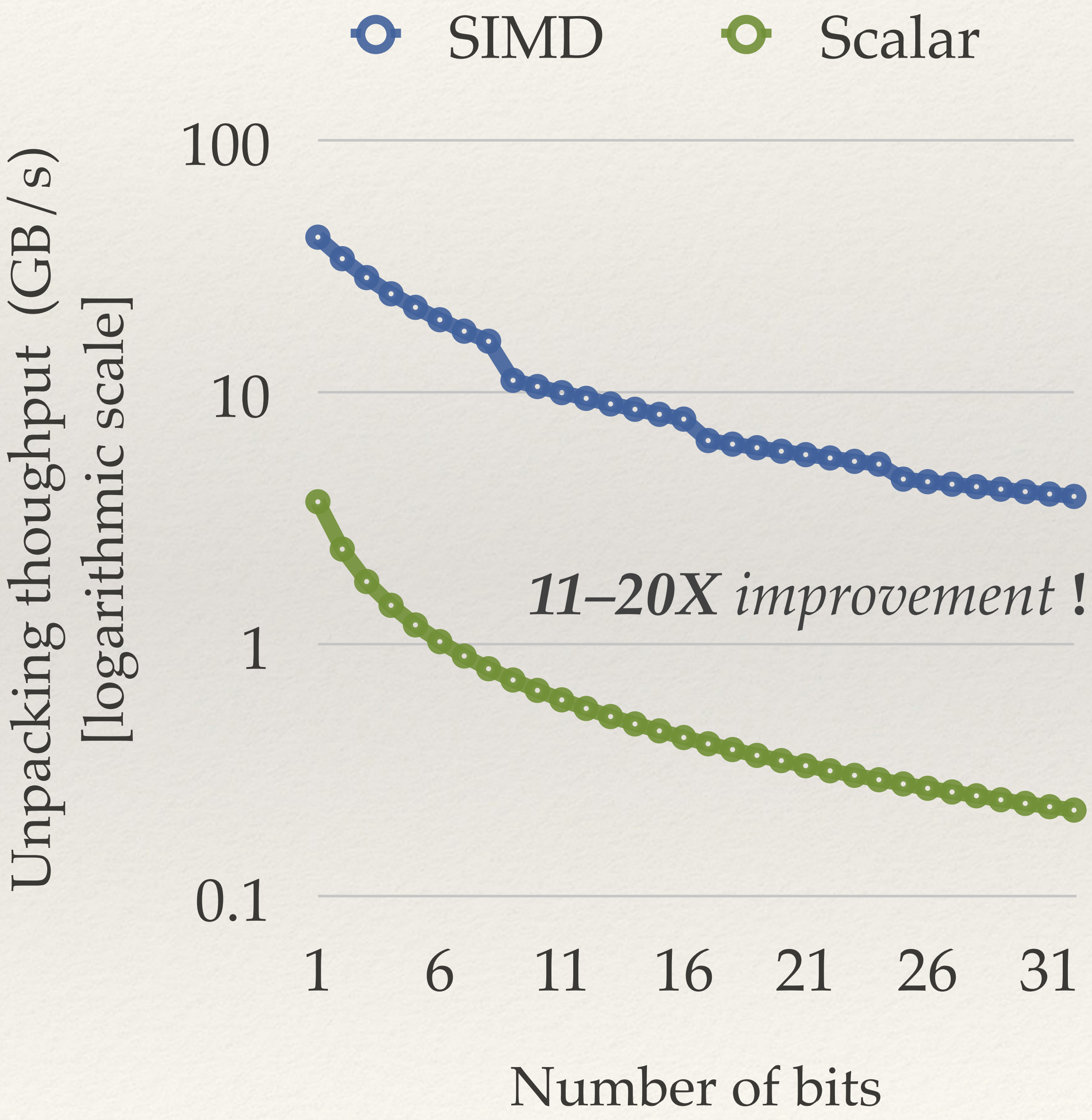
0001	1010	1100	1111	0101	1100	1000	0010

up-convert & add

00010000	10100000	11000000	11111000
01010000	01001000	10001000	00100000

Vertical Layout

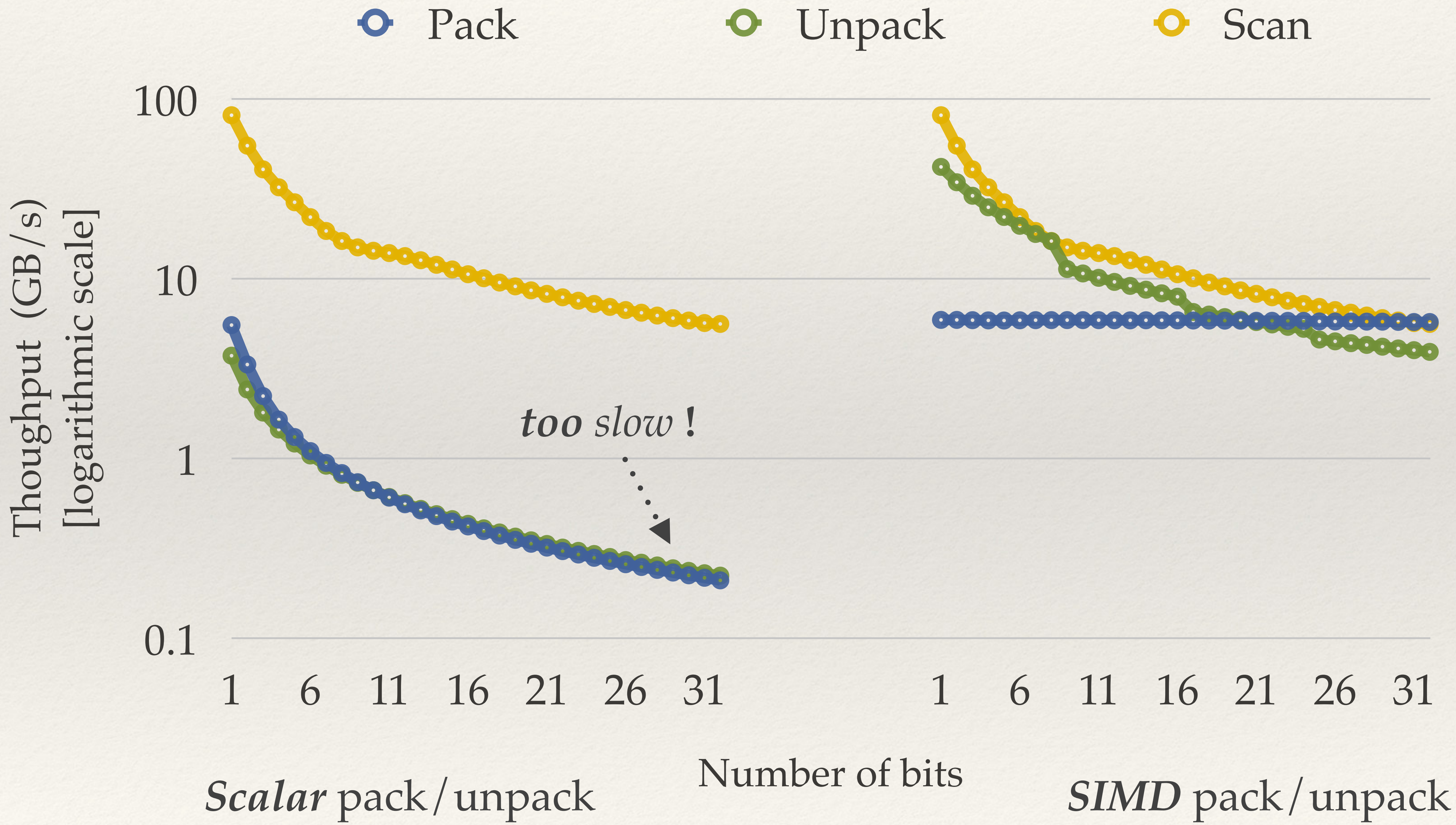
- ***Fully*** packed & word ***aligned***
  - Interleave bits of ***k*** codes
    - ***k*** divides the processor word
  - Scanning
    - Evaluate ***without*** unpacking
    - Can ***skip*** words early
    - Increase ***k*** to minimize false (pre)fetches
  - Packing
    - Transfer ***nb*** bits across registers
    - Can be written in ***SIMD* !**
    - Extract bits ***per byte*** not per int
  - Unpacking
    - Transfer ***nb*** bits across registers
    - Can be written in ***SIMD* !**
    - Insert bits ***per byte*** not per int



***11–20X improvement* !**

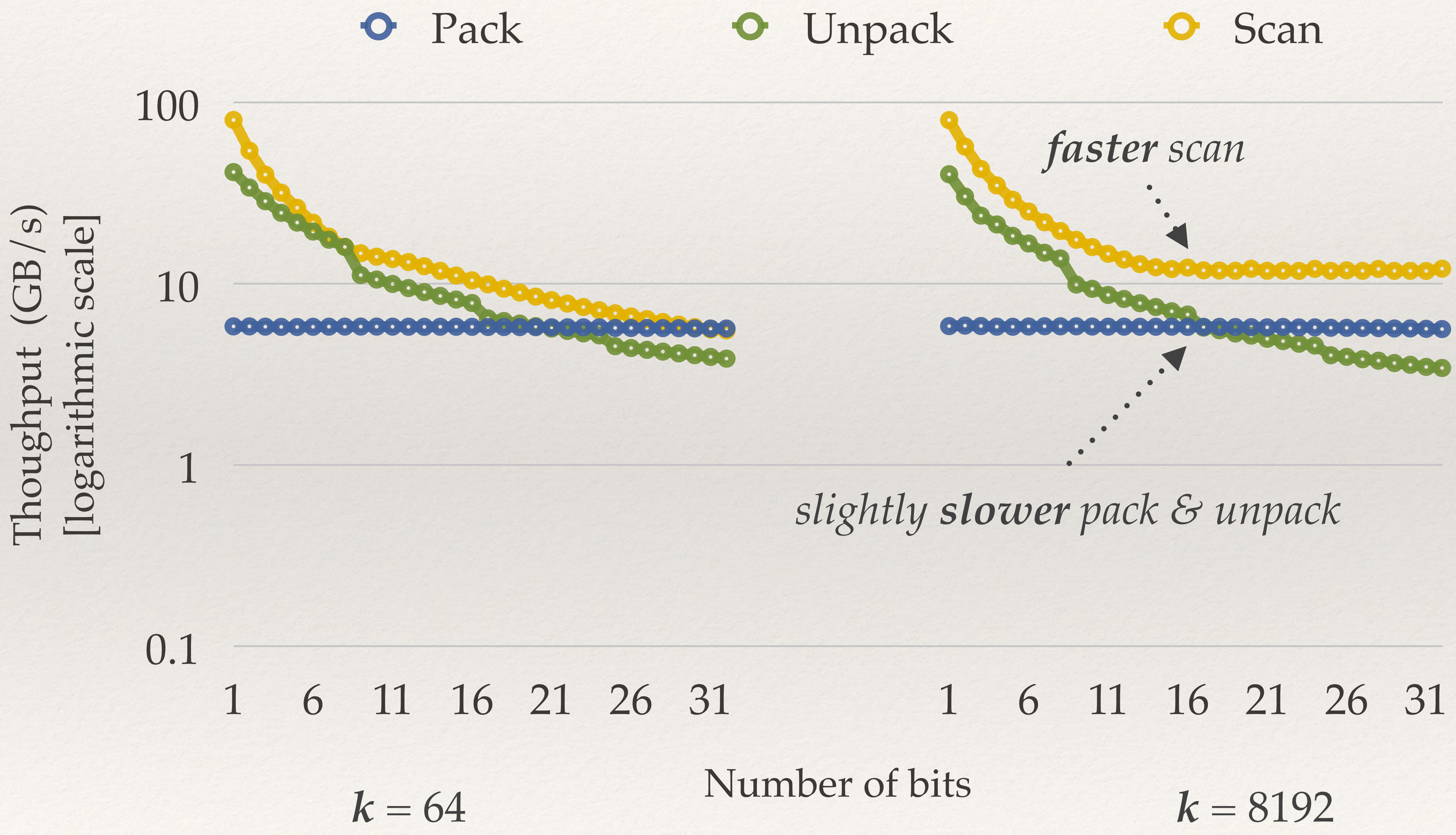
Vertical Layout

- Scalar to ***SIMD*** for packing & unpacking (***k*** = 64)
  - Scalar scan



Vertical Layout

- Increasing ***k*** to the L1 cache size (***k*** = 8192)
  - SIMD scanning

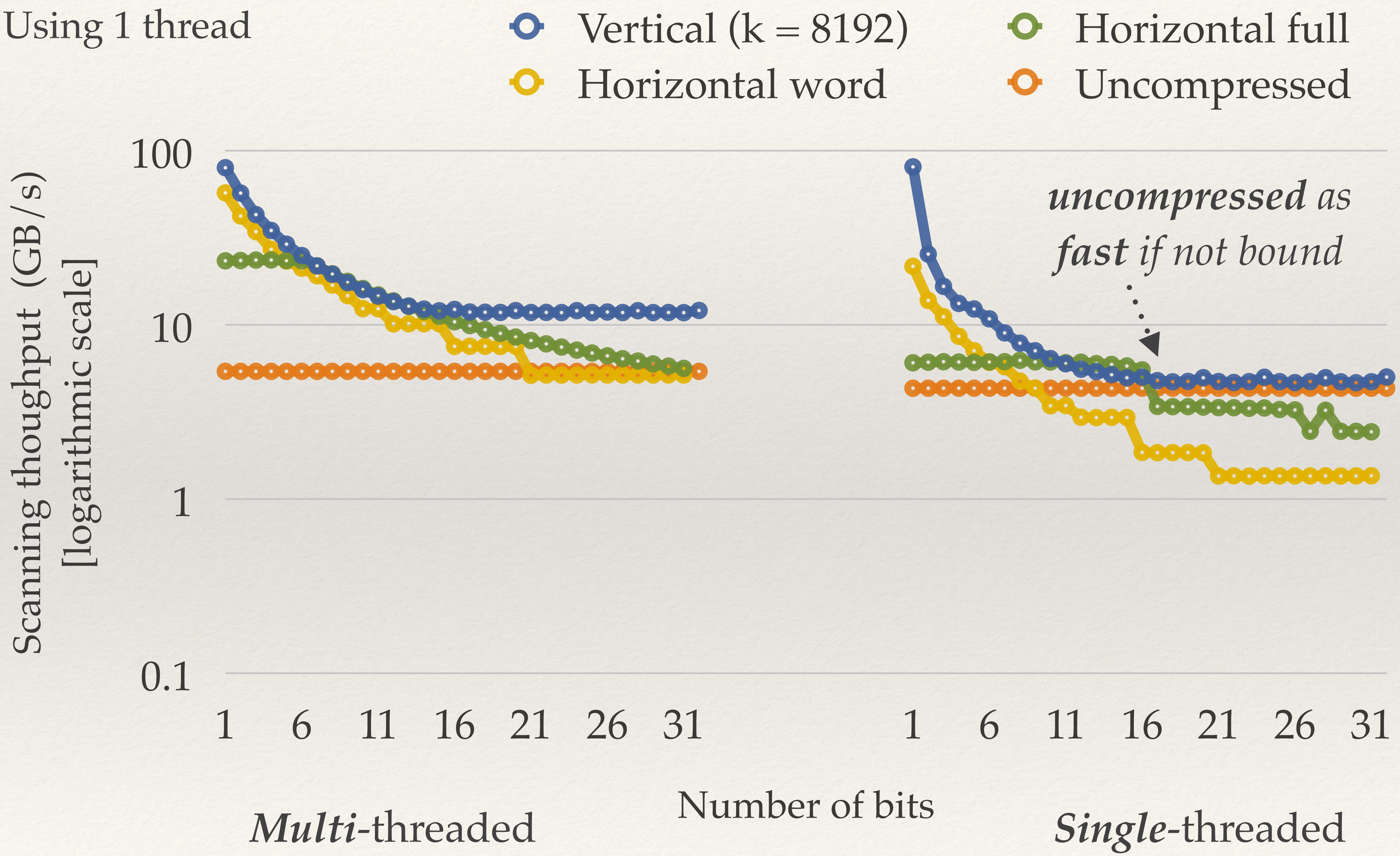


faster scan

*slightly **slower** pack & unpack*

Vertical Layout

- If not memory ***bound***
  - Using 1 thread



***Multi*-threaded**

***Single*-threaded**

***uncompressed** as*
***fast** if not bound*

Conclusions

- Horizontal layouts
  - Fully packed
    - No wasted space but somewhat ***slow***
    - Can optimize unpacking & scanning with ***SIMD***

Conclusions

- Horizontal layouts
  - Fully packed
    - No wasted space but somewhat ***slow***
    - Can optimize unpacking & scanning with ***SIMD***
  - Word aligned
    - Fast ***scalar*** scans but not optimal due to wasted space

Conclusions

- Horizontal layouts
  - Fully packed
    - No wasted space but somewhat ***slow***
    - Can optimize unpacking & scanning with ***SIMD***
  - Word aligned
    - Fast ***scalar*** scans but not optimal due to wasted space
- Vertical layout
  - Known techniques
    - Fast scalar scans ***without*** wasting space
    - Very slow ***scalar*** packing & unpacking

Conclusions

- Horizontal layouts
    - Fully packed
        - No wasted space but somewhat ***slow***
        - Can optimize unpacking & scanning with ***SIMD***
    - Word aligned
        - Fast ***scalar*** scans but not optimal due to wasted space
- Vertical layout
    - Known techniques
        - Fast scalar scans ***without*** wasting space
        - Very slow ***scalar*** packing & unpacking
    - New techniques
        - Fast packing & unpacking using ***SIMD***
        - Maximize bit transfers by using the ***smallest*** SIMD lanes
        - Increase ***k*** to skip cache lines effectively

Questions ?