# Screaming Fast Galois Field Arithmetic Using Intel SIMD Instructions

James S. Plank
University of Tennessee

*USENIX FAST*
San Jose, CA
February 15, 2013.

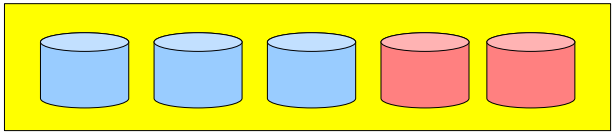
# Authors

Jim Plank
Tennessee

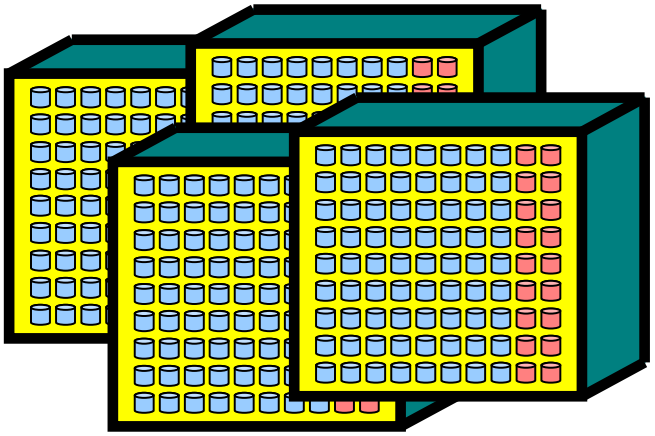
Kevin Greenan
EMC/Data Domain

Ethan Miller
UC Santa Cruz

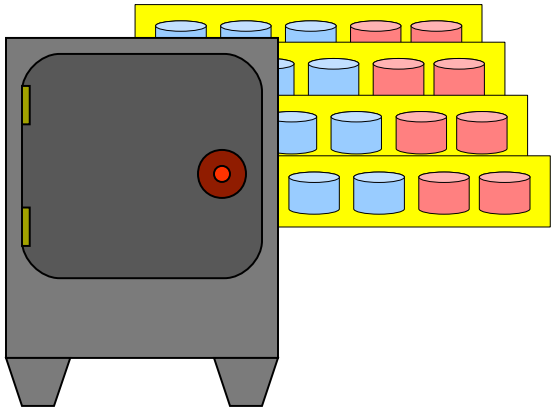
# Erasure Codes are Everywhere



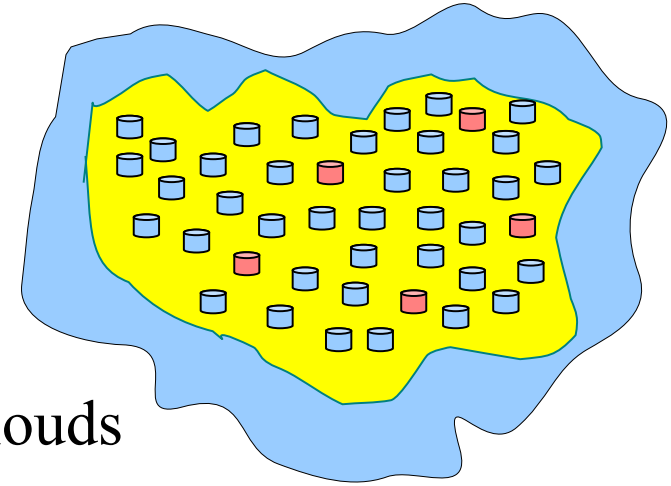
RAID Systems



Data Centers

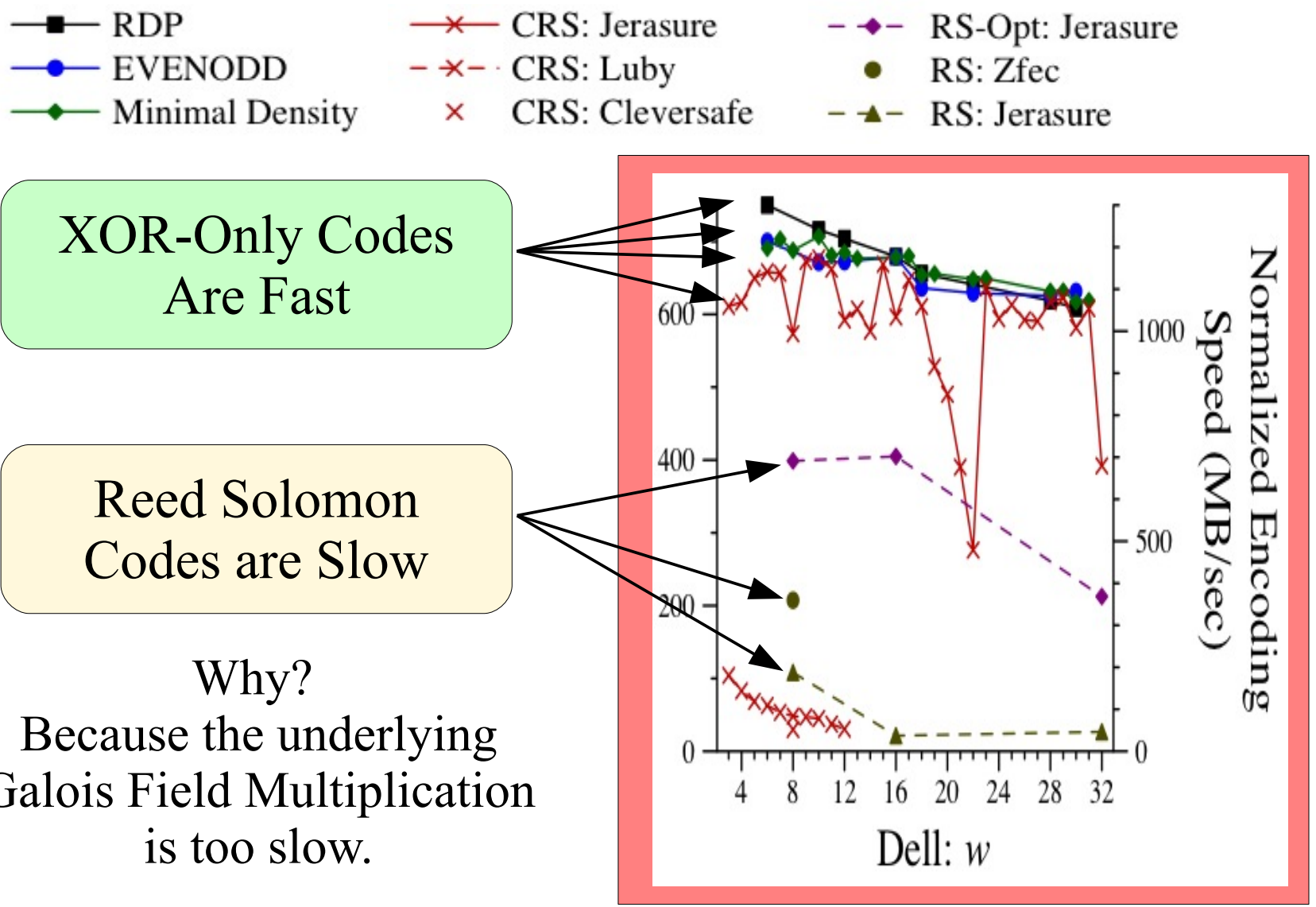


Archival Systems



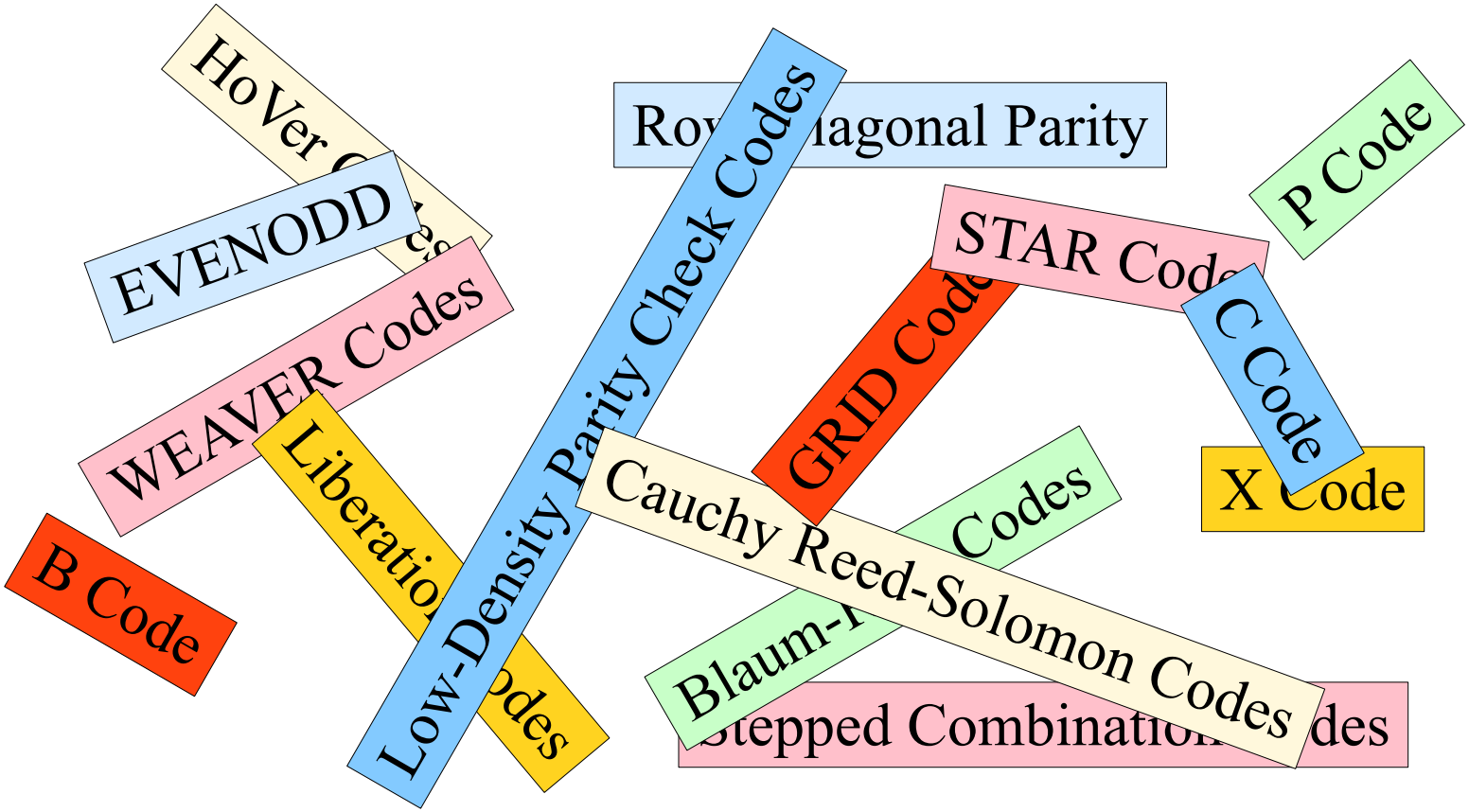
Clouds

# Conventional Wisdom (FAST 2009)



# Conventional Wisdom

- This is inconvenient, because Reed-Solomon codes are powerful, general and flexible.
- Has led to a proliferation of XOR Codes



# Conventional Wisdom Says...

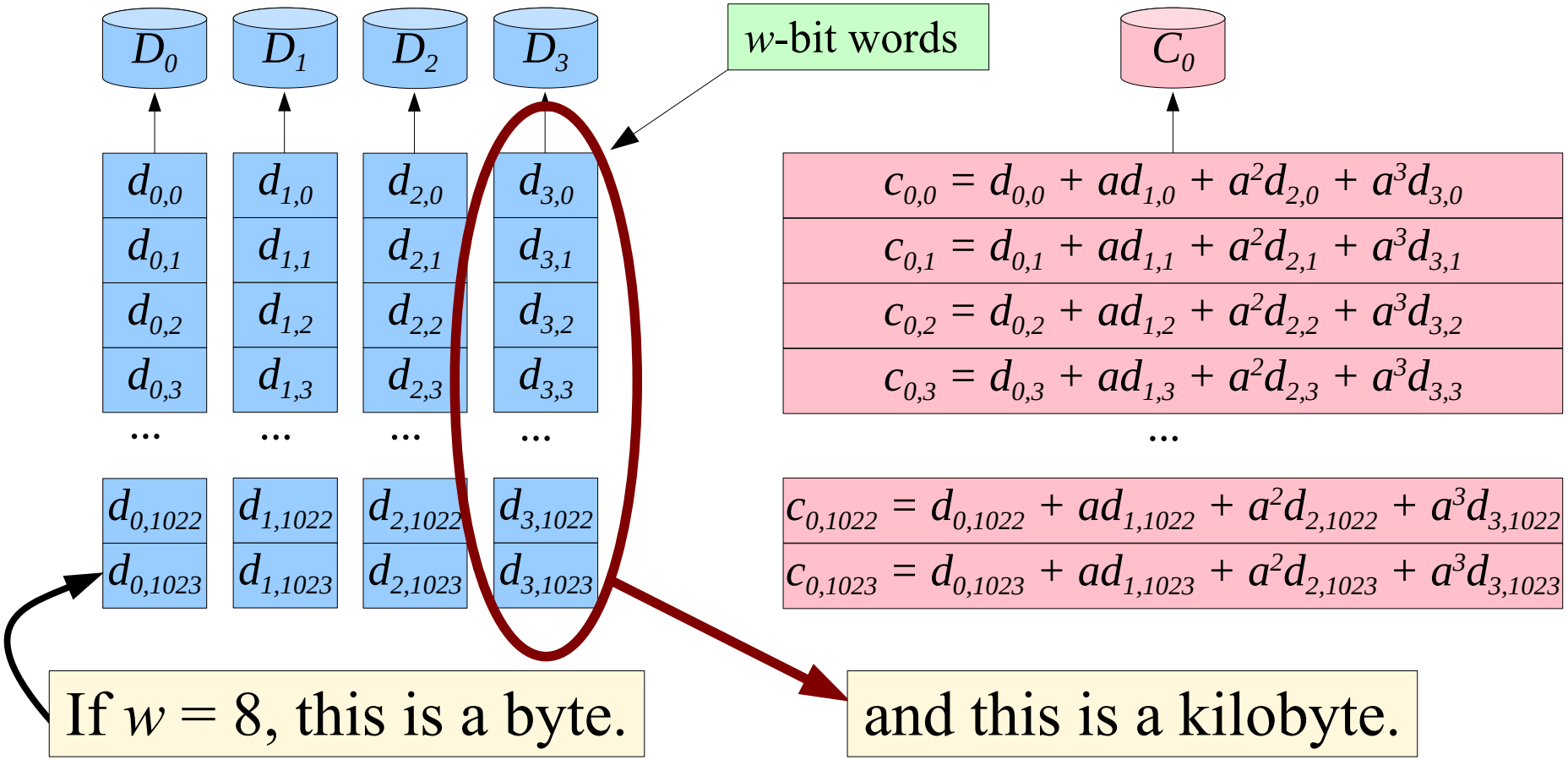
- However, in recent years....
  - Eerily smug reportings of doing Reed-Solomon coding at “cache line speeds.”
  - Renders all of that XOR mess moot.
  - But no one reveals the secret handshake.
- In this talk, we're gonna reveal the secret handshake.
  - And you won't need to know anything special about Galois Field arithmetic.

# Some Bottom Lines

- Using Intel's SSE3 “SIMD” Instructions
  - Perform Galois Field fast enough that its performance is limited by the L2/L3 caches.
  - *Factors* of 2.7 to 12 times faster than previous implementations.
- “GF-Complete”
  - Open Source C library (BSD License)
  - Gives you the handshake.

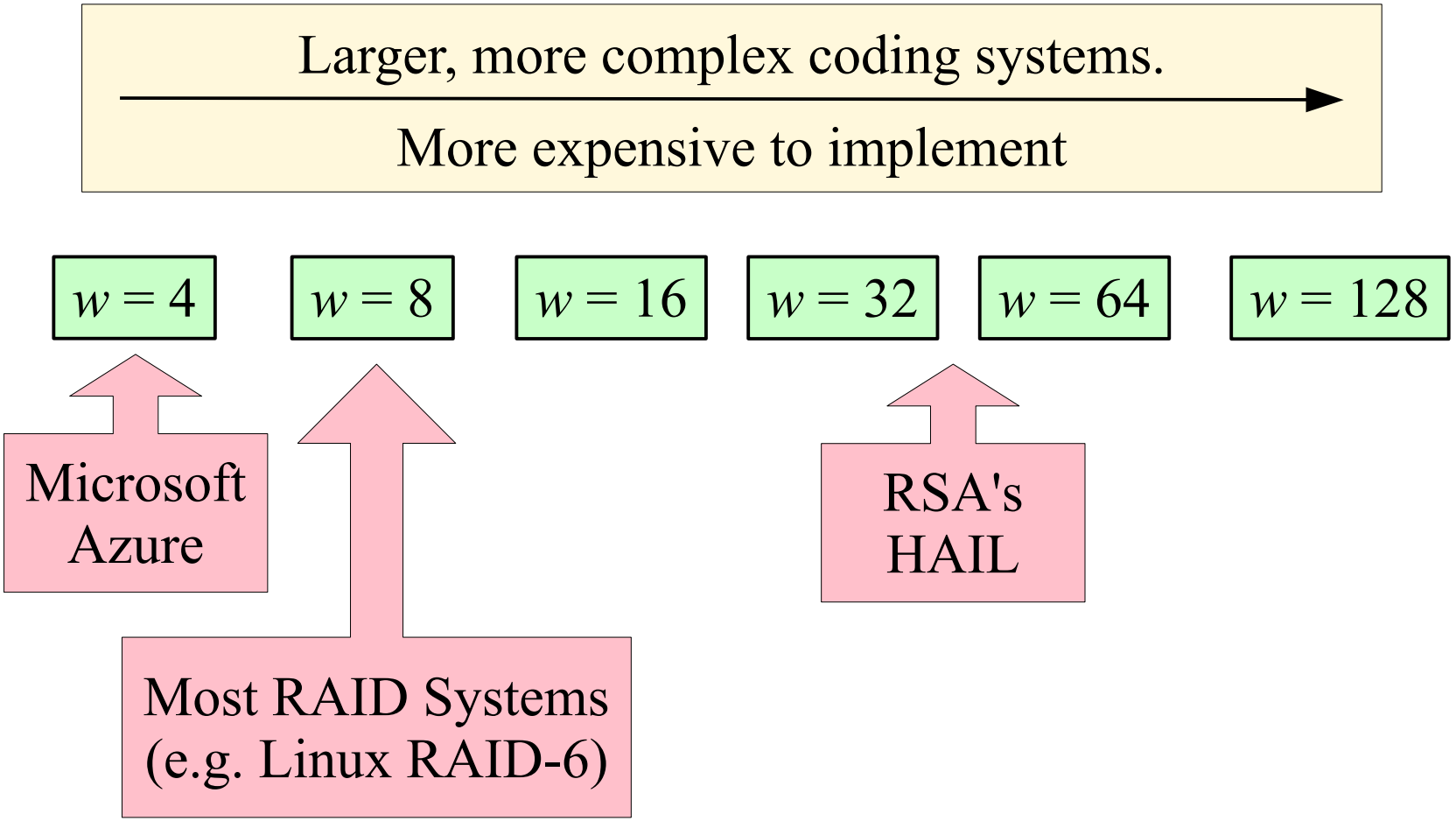
# How do Storage Systems use GF Arithmetic?

Erasure codes are structured as linear combinations of $w$-bit data words in a Galois Field – termed $GF(2^w)$.



# What is *w*?

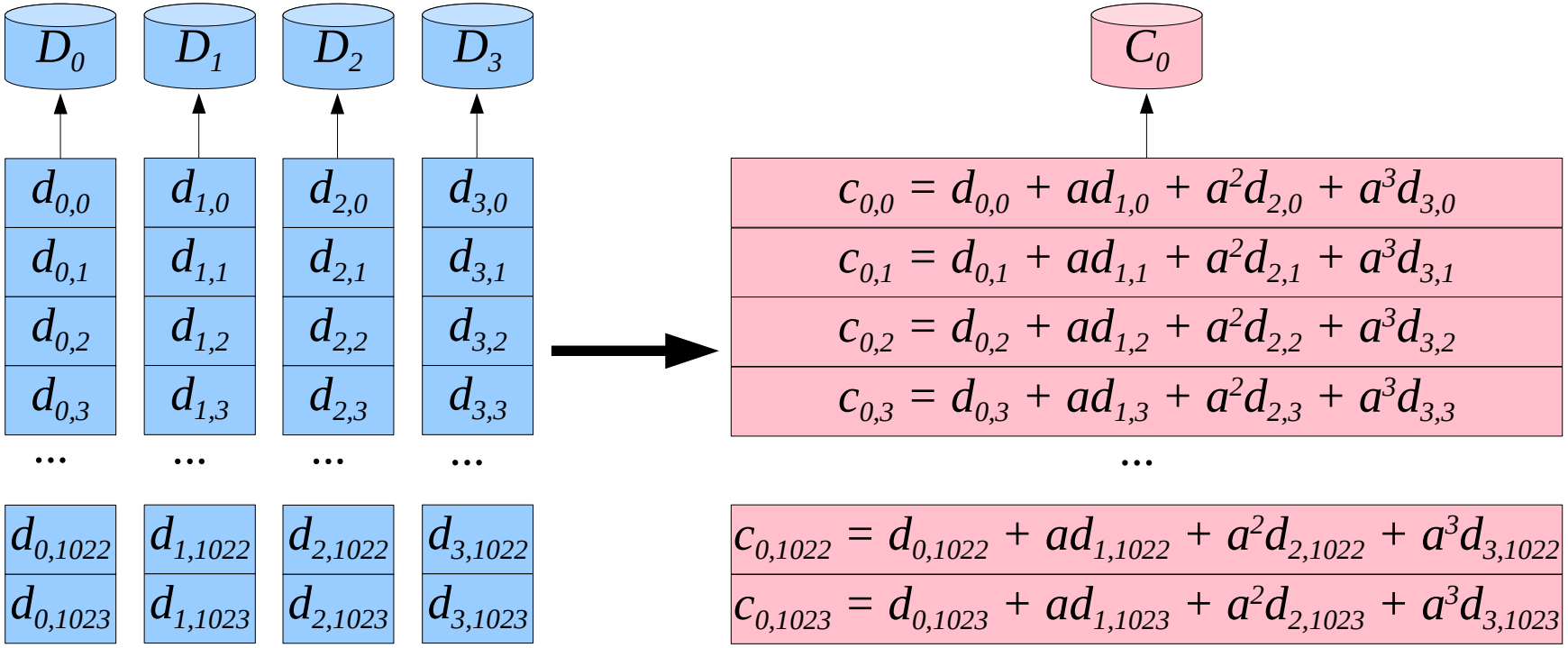
- The number of bits in the erasure-coding "word".



# How do Storage Systems use GF Arithmetic?

The two major erasure coding operations are:

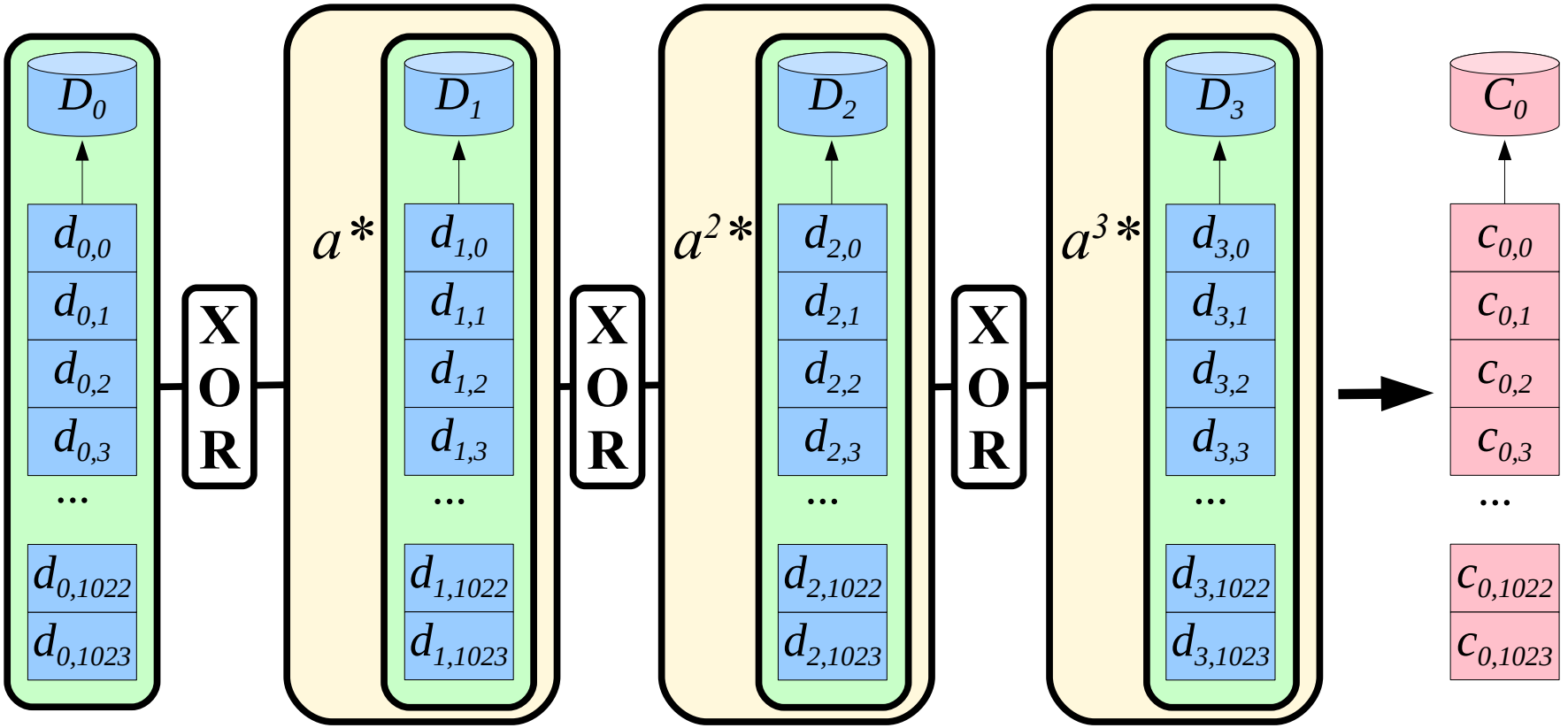
- XOR-ing two regions of memory
- Multiplying a region of memory by a constant in $GF(2^w)$.



# How do Storage Systems use GF Arithmetic?

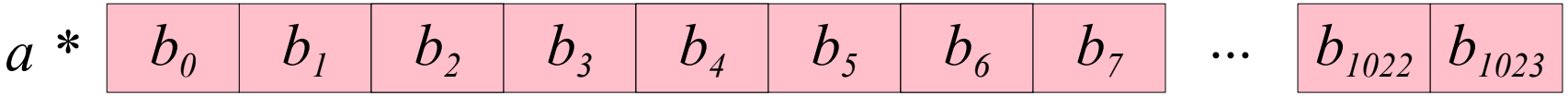
The two major erasure coding operations are:

- XOR-ing two regions of memory
- Multiplying a region of memory by a constant in $GF(2^w)$.

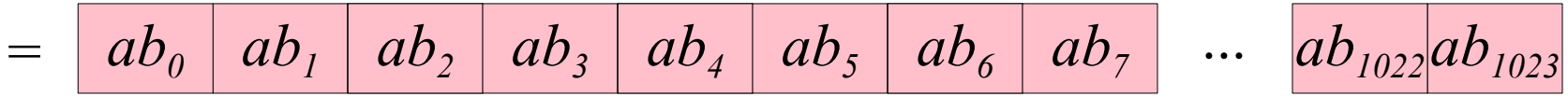


# How do Storage Systems use GF Arithmetic?

Viewed another way, when we want to multiply a 1K region of words in $GF(2^8)$ by a constant $a$:




We want the result to look like 1024 individual multiplications,

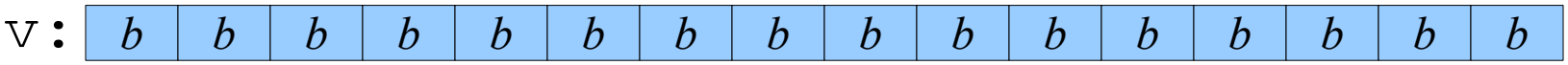



But we don't want to actually do 1024 individual multiplications.

# Intel's "Streaming SIMD" Instructions

Works on 128-bit "vectors:

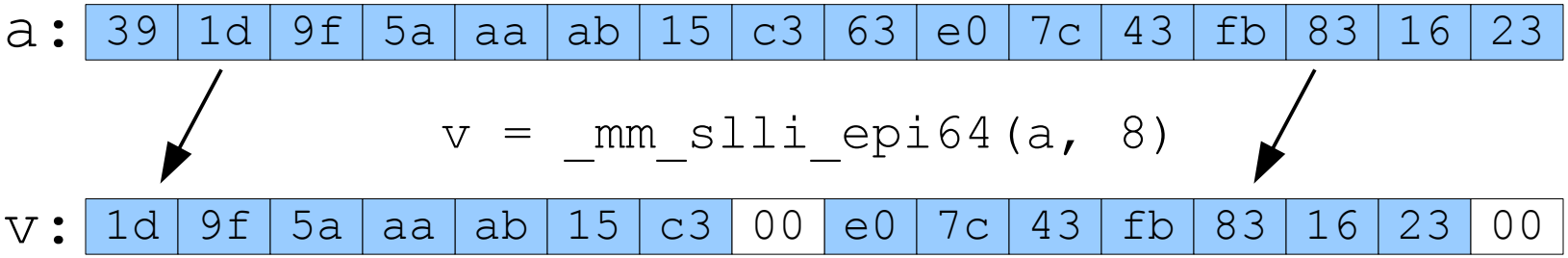
`v = _mm_set_epi8(b)` - Replicate *b* 16 times:




`v = _mm_xor_si128(a, b)` - 128-bit XOR

`v = _mm_and_si128(a, b)` - 128-bit AND

`v = _mm_slli_epi64(a, x)` - 2 64-bit left shifts by *x*.

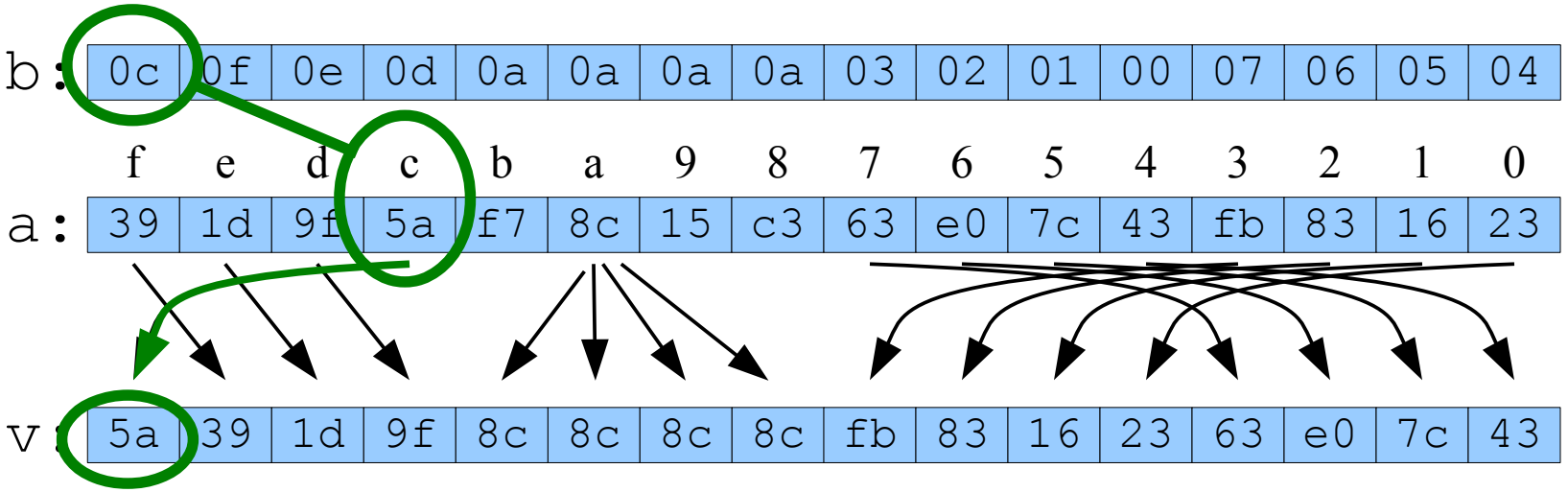


# Intel's "Streaming SIMD" Instructions

The killer instruction is *_mm_shuffle_epi8()*:

```
v = _mm_shuffle_epi8(a, b)
```

- Performs 16 simultaneous table lookups using:
  - *a* as a 16-element table.
  - *b* as 16 4-bit indices.

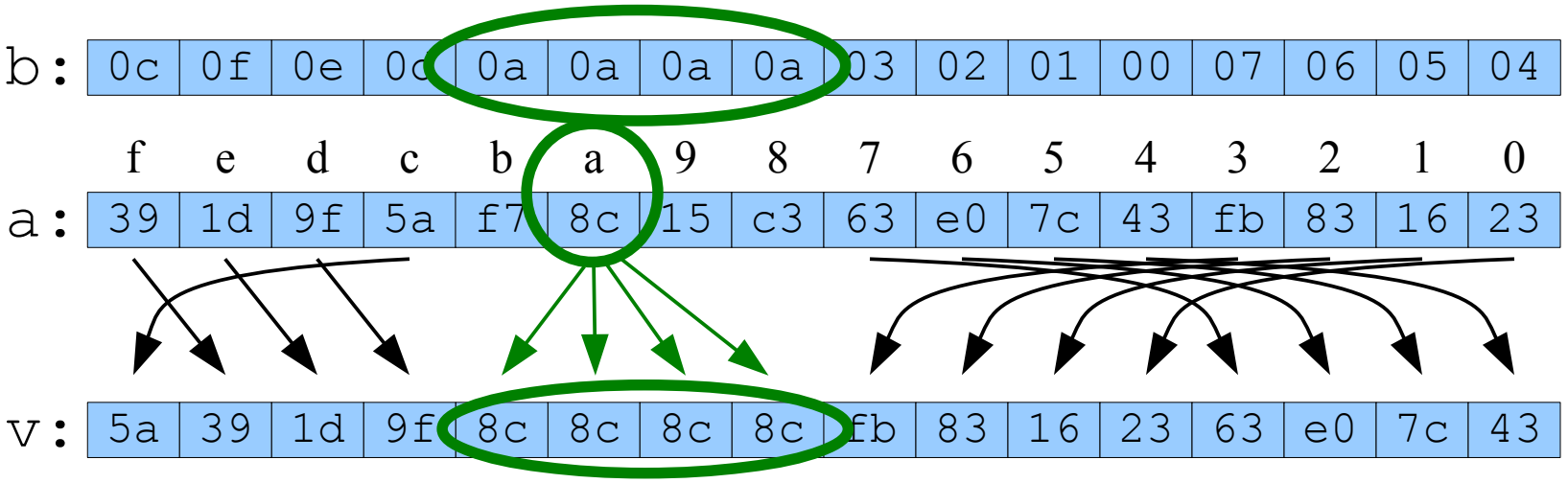


# Intel's "Streaming SIMD" Instructions

The killer instruction is *_mm_shuffle_epi8()*:

```
v = _mm_shuffle_epi8(a, b)
```

- Performs 16 simultaneous table lookups using:
  - *a* as a 16-element table.
  - *b* as 16 4-bit indices.



# Buffer-constant Multiplication in *GF(2⁴)*

Example: Multiplying 16 bytes *A* by 7 in *GF(2⁴)*.

| byte | f | e | d | c | b | a | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| *table1:* | 0b | 0c | 05 | 02 | 04 | 03 | 0a | 0d | 06 | 01 | 08 | 0f | 09 | 0e | 07 | 00 |
| *table2 = mm_slli_epi64(table1, 4):* | b0 | c0 | 50 | 20 | 40 | 30 | a0 | d0 | 60 | 10 | 80 | f0 | 90 | e0 | 70 | 00 |
| *mask1 = mm_set1_epi8(0xf):* | 0f | 0f | 0f | 0f | 0f | 0f | 0f | 0f | 0f | 0f | 0f | 0f | 0f | 0f | 0f | 0f |
| *mask2 = mm_set1_epi8(0xf0):* | f0 | f0 | f0 | f0 | f0 | f0 | f0 | f0 | f0 | f0 | f0 | f0 | f0 | f0 | f0 | f0 |
| *A:* | 39 | 1d | 9f | 5a | aa | ab | 15 | c3 | 63 | e0 | 7c | 43 | fb | 83 | 16 | 23 |

Start by setting up two 16-byte tables, and a few masks.

For example, 3*7 = 9,
so the high four bits of the product should equal nine.

# Buffer-constant Multiplication in $GF(2^4)$

Example: Multiplying 16 bytes *A* by 7 in $GF(2^4)$.

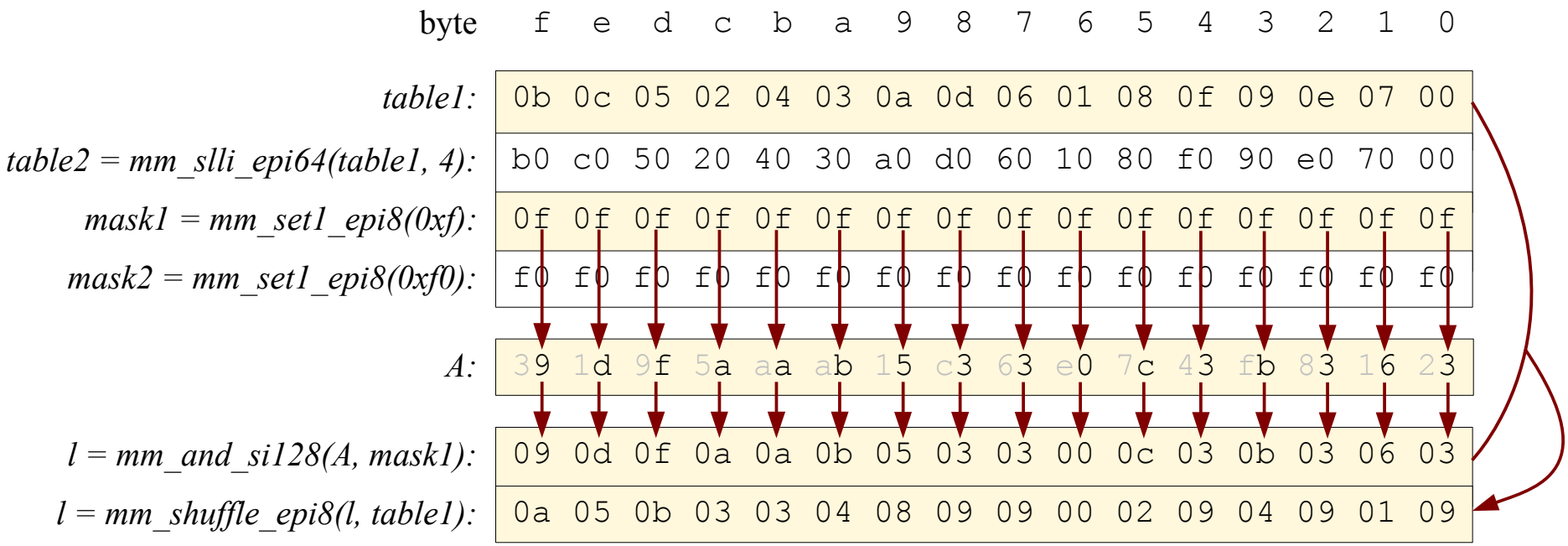
| byte | f | e | d | c | b | a | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| *table1:* | 0b | 0c | 05 | 02 | 04 | 03 | 0a | 0d | 06 | 01 | 08 | 0f | 09 | 0e | 07 | 00 |
| *table2 = mm_slli_epi64(table1, 4):* | b0 | c0 | 50 | 20 | 40 | 30 | a0 | d0 | 60 | 10 | 80 | f0 | 90 | e0 | 70 | 00 |
| *mask1 = mm_set1_epi8(0xf):* | 0f | 0f | 0f | 0f | 0f | 0f | 0f | 0f | 0f | 0f | 0f | 0f | 0f | 0f | 0f | 0f |
| *mask2 = mm_set1_epi8(0xf0):* | f0 | f0 | f0 | f0 | f0 | f0 | f0 | f0 | f0 | f0 | f0 | f0 | f0 | f0 | f0 | f0 |
| *A:* | 39 | 1d | 9f | 5a | aa | ab | 15 | c3 | 63 | e0 | 7c | 43 | fb | 83 | 16 | 23 |

Start by setting up two 16-byte tables, and a few masks.

For example, 3*7 = 9,
so the high four bits of the product should equal nine,
and the next four bit should equal ten (0xa).

# Buffer-constant Multiplication in *GF(2⁴)*

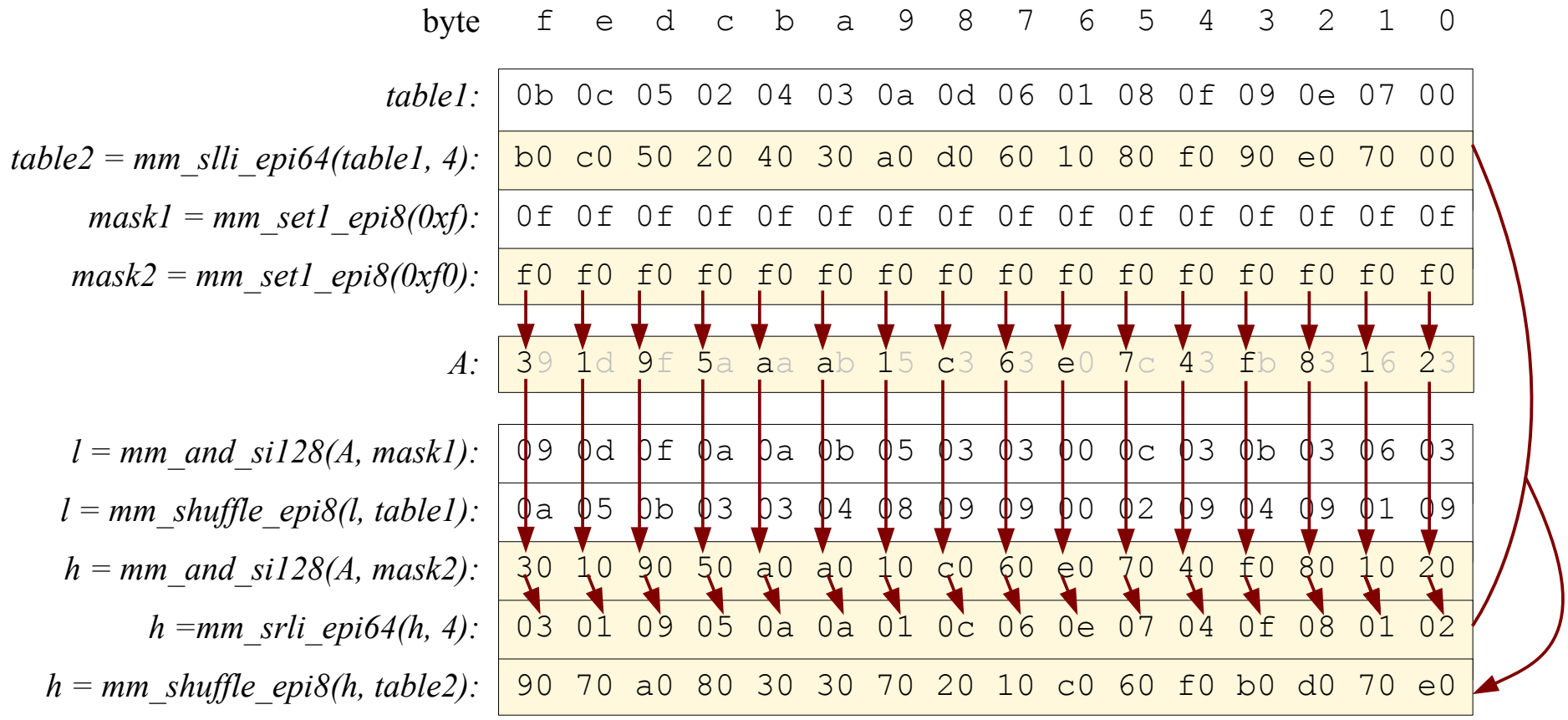
Example: Multiplying 16 bytes *A* by 7 in *GF(2⁴)*.




Create indices from the low 4 bits of each byte and perform the table lookups.

# Buffer-constant Multiplication in *GF(2⁴)*

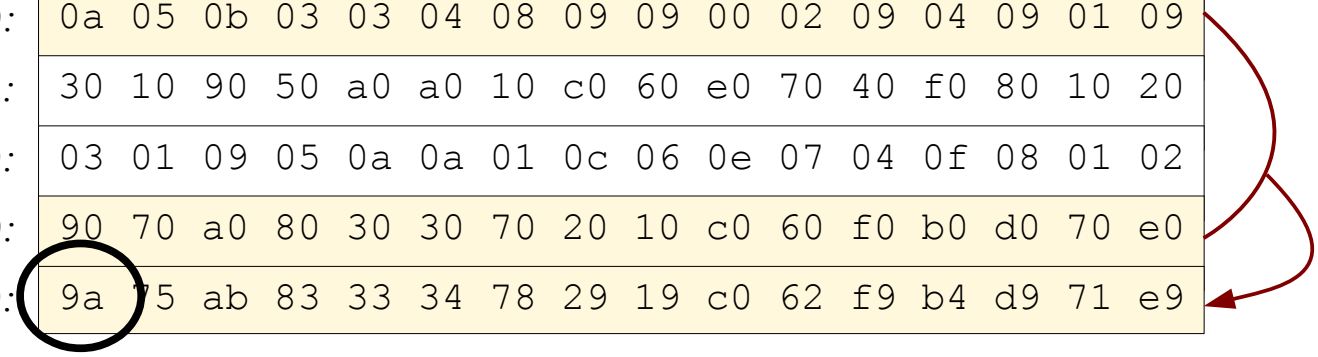
Create indices from the high 4 bits of each byte and perform the table lookups with the second table.



# Buffer-constant Multiplication in $GF(2^4)$

XOR the two products, and you're done.
Six instructions for 32 multiplications!

| byte | f | e | d | c | b | a | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| *table1:* | 0b | 0c | 05 | 02 | 04 | 03 | 0a | 0d | 06 | 01 | 08 | 0f | 09 | 0e | 07 | 00 |
| *table2 = mm_slli_epi64(table1, 4):* | b0 | c0 | 50 | 20 | 40 | 30 | a0 | d0 | 60 | 10 | 80 | f0 | 90 | e0 | 70 | 00 |
| *mask1 = mm_set1_epi8(0xf):* | 0f | 0f | 0f | 0f | 0f | 0f | 0f | 0f | 0f | 0f | 0f | 0f | 0f | 0f | 0f | 0f |
| *mask2 = mm_set1_epi8(0xf0):* | f0 | f0 | f0 | f0 | f0 | f0 | f0 | f0 | f0 | f0 | f0 | f0 | f0 | f0 | f0 | f0 |
| *A:* | 39 | 1d | 9f | 5a | aa | ab | 15 | c3 | 63 | e0 | 7c | 43 | fb | 83 | 16 | 23 |
| *l = mm_and_si128(A, mask1):* | 09 | 0d | 0f | 0a | 0a | 0b | 05 | 03 | 03 | 00 | 0c | 03 | 0b | 03 | 06 | 03 |
| *l = mm_shuffle_epi8(l, table1):* | 0a | 05 | 0b | 03 | 03 | 04 | 08 | 09 | 09 | 00 | 02 | 09 | 04 | 09 | 01 | 09 |
| *h = mm_and_si128(A, mask2):* | 30 | 10 | 90 | 50 | a0 | a0 | 10 | c0 | 60 | e0 | 70 | 40 | f0 | 80 | 10 | 20 |
| *h =mm_srli_epi64(h, 4):* | 03 | 01 | 09 | 05 | 0a | 0a | 01 | 0c | 06 | 0e | 07 | 04 | 0f | 08 | 01 | 02 |
| *h = mm_shuffle_epi8(h, table2):* | 90 | 70 | a0 | 80 | 30 | 30 | 70 | 20 | 10 | c0 | 60 | f0 | b0 | d0 | 70 | e0 |
| *yA = mm_xor_si128(h, l):* | 9a | 75 | ab | 83 | 33 | 34 | 78 | 29 | 19 | c0 | 62 | f9 | b4 | d9 | 71 | e9 |



# In $GF(2^8)$

- Split each 8-bit word into two 4-bit components, and use the distributive law of multiplication.

$$a = (a_{high} << 4) \oplus a_{low} \longrightarrow ab = (a_{high} << 4)b \oplus a_{low}b$$

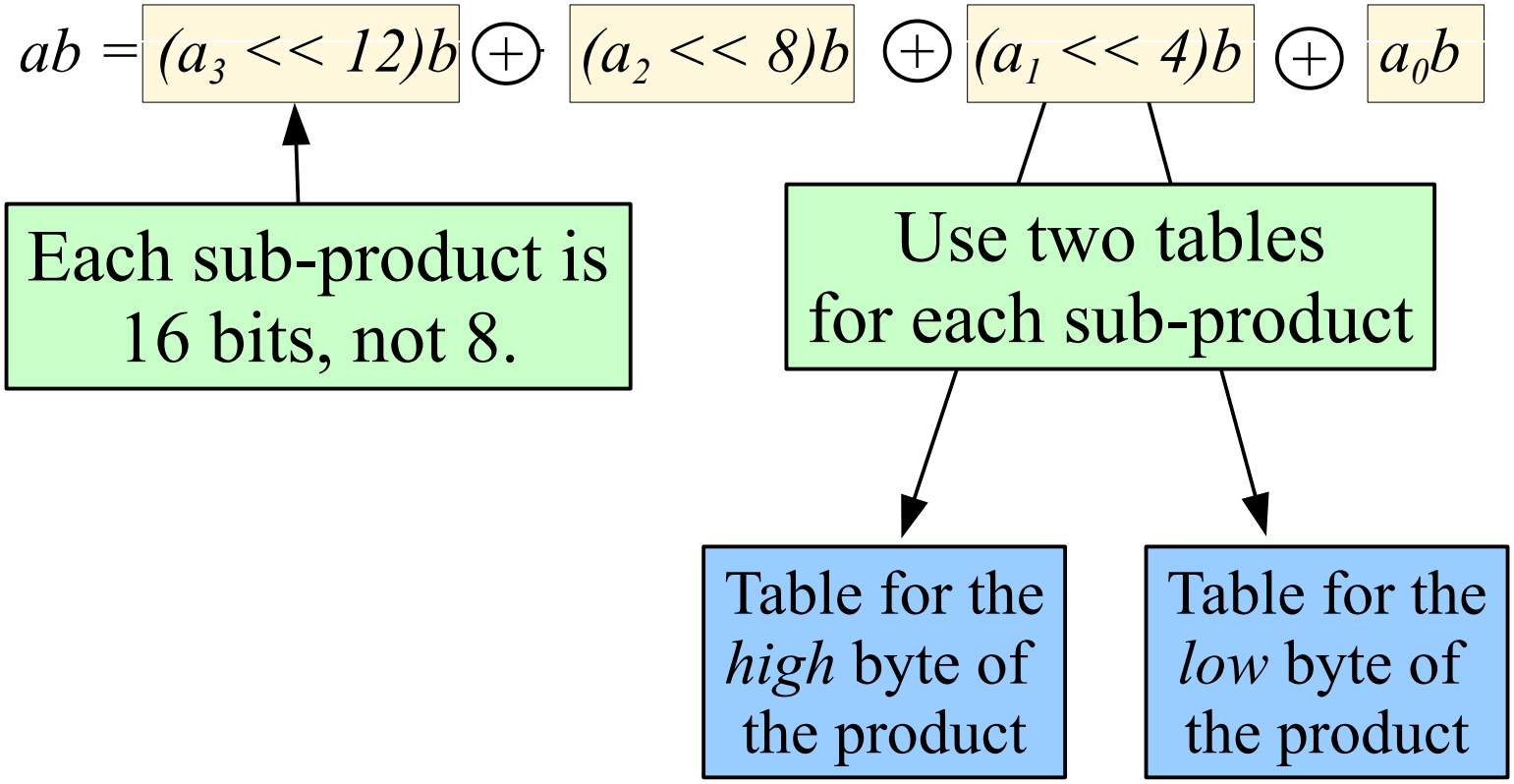
Two different tables

| byte | f | e | d | c | b | a | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| *table1:* | 2d | 2a | 23 | 24 | 31 | 36 | 3f | 38 | 15 | 12 | 1b | 1c | 09 | 0e | 07 | 00 |
| *table2:* | ea | 9a | 0a | 7a | 37 | 47 | d7 | a7 | 4d | 3d | ad | dd | 90 | e0 | 70 | 00 |

Otherwise the code is the same as the last slide.

# In $GF(2^{16})$

- Again use the distributive law on 4-bit words:

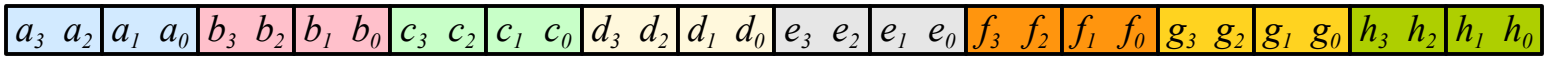



That's a total of 8 tables, two for each sub-product.

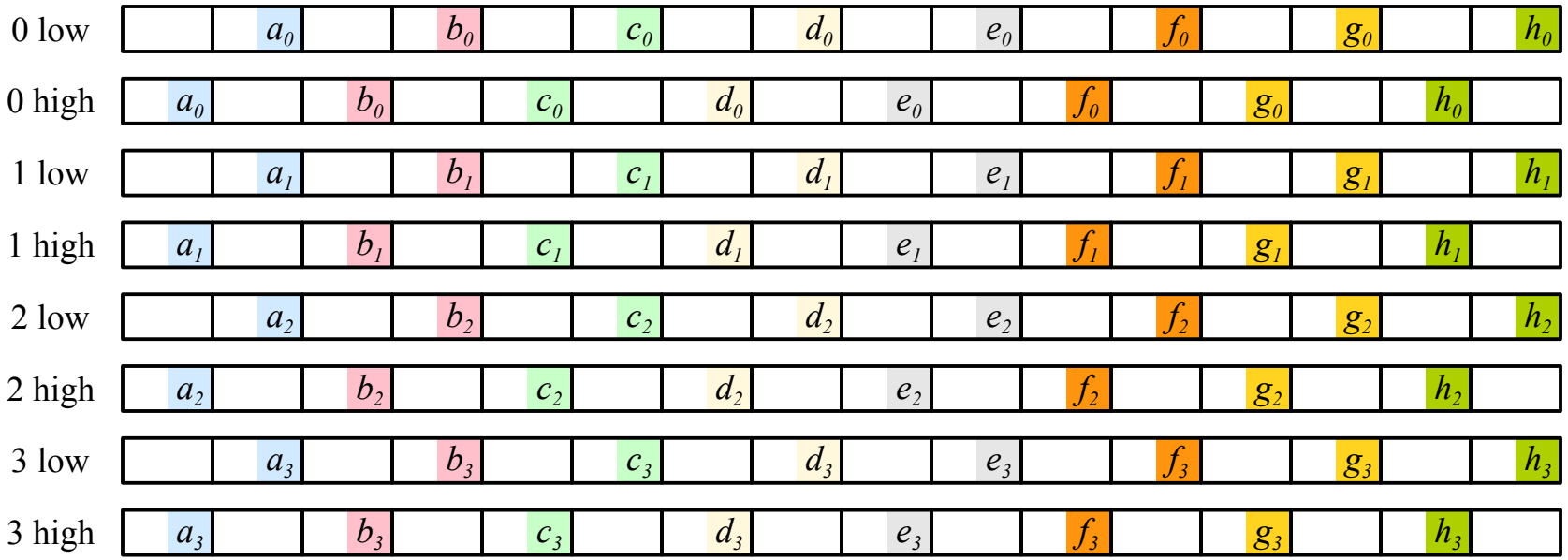
# In $GF(2^{16})$

- Mapping words to memory makes a difference.

Standard mapping of 16-bit words *a-h* to 128-bit vectors:



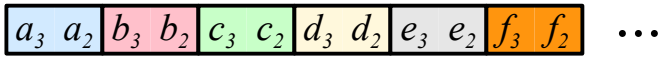

Have to do 8 table lookups for the 8 products:



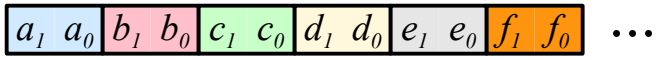
# In $GF(2^{16})$

- Instead, split each 16-bit word over two 128-bit vectors:

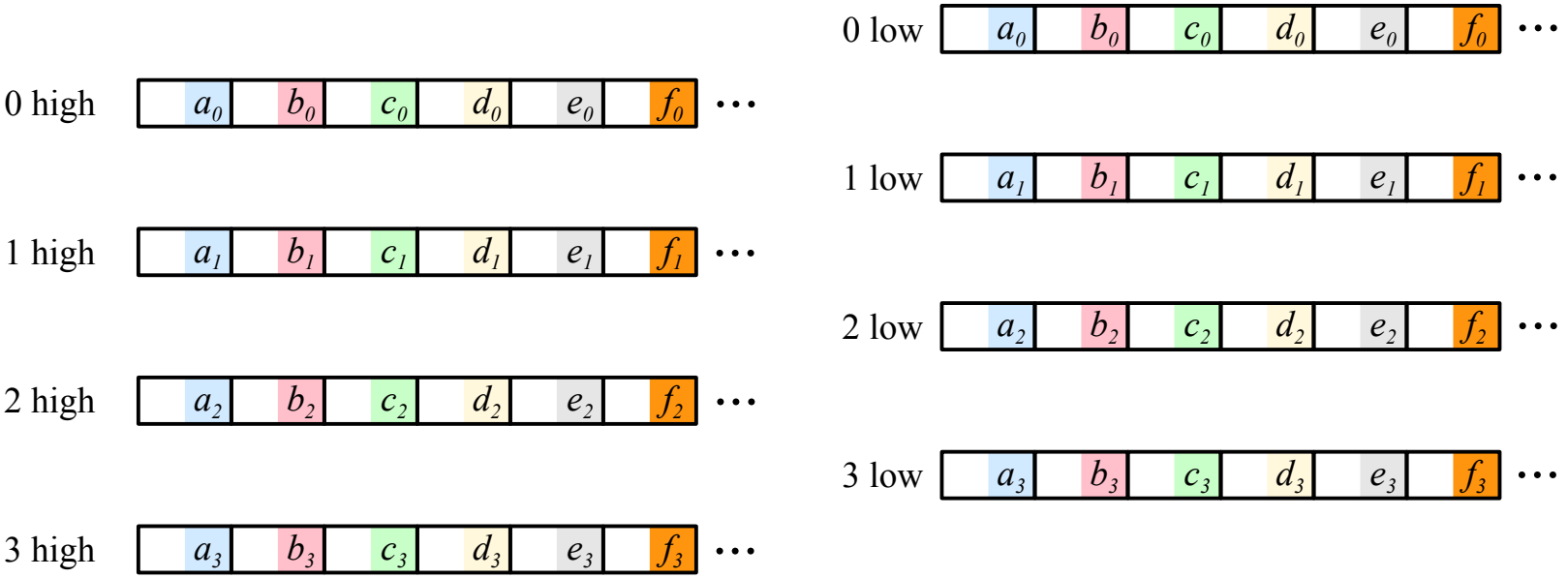
Vector of high bytes




Vector of low bytes




Still have 8 table lookups, but for 256 bits of words:



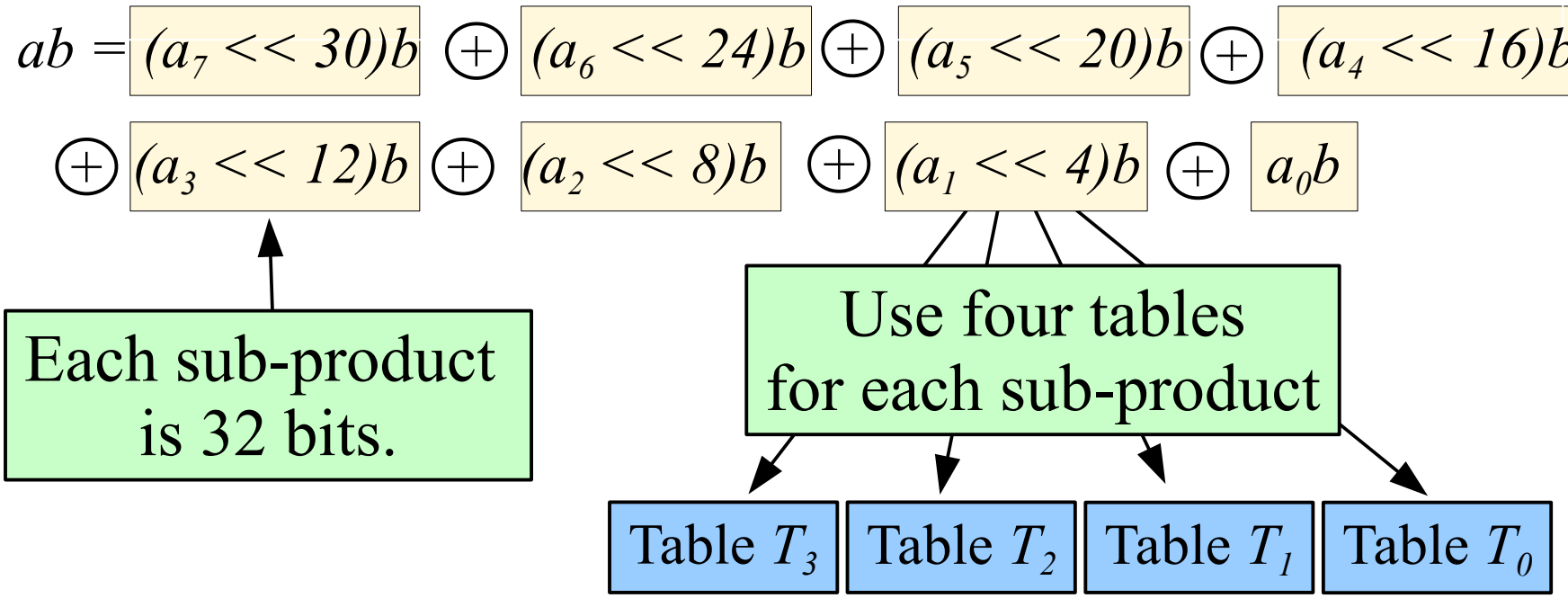
# In $GF(2^{16})$

- Call this *alternate mapping.*
- Has all the properties you need for Reed-Solomon coding.
- Is a little confusing, since it's harder to "read" memory.
- Can convert standard mapping to alternate mapping with 7 SIMD instructions, and back again with 2 SIMD instructions.

BUT YOU DON'T HAVE TO
IF YOU DON'T WANT TO!

# In $GF(2^{32})$

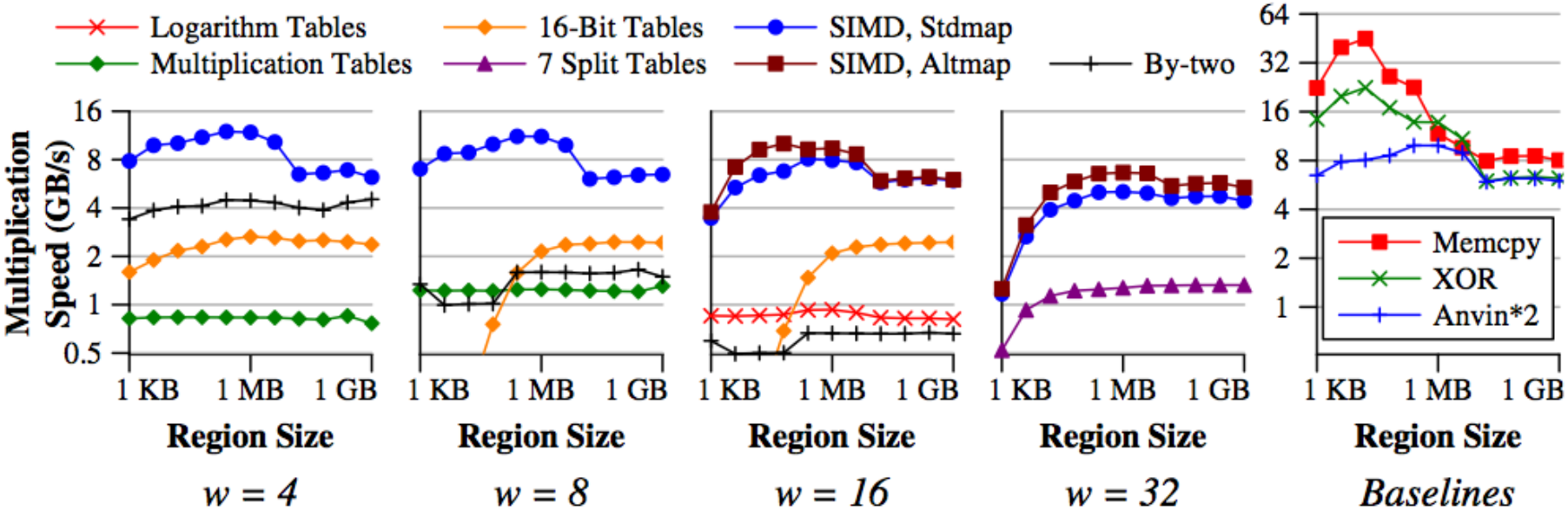
- Again use the distributive law on 4-bit words:




That's a total of 32 tables.

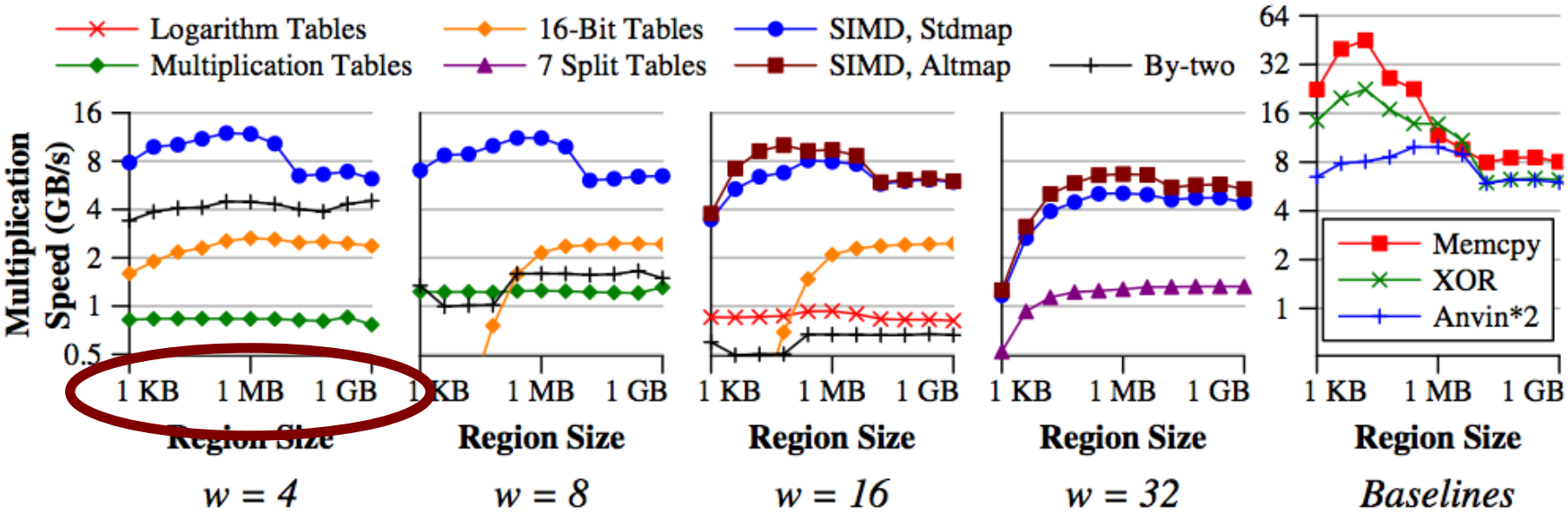
Can use the same alternate mapping trick.

# Performance



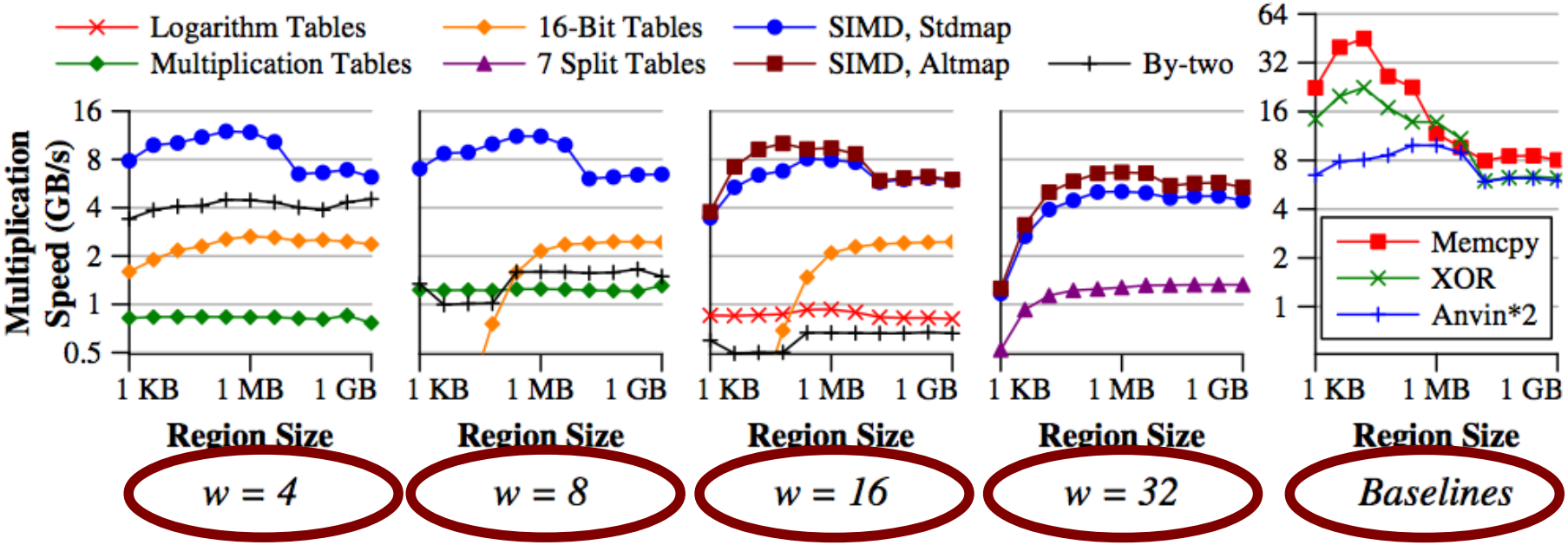

- 3.4 GHz Intel Core i7-3770
- 256 KB L2 Cache, 8 MB L3 Cache
- Performing buffer-constant on various buffer sizes
- Lots of comparisons.

# Performance




- 3.4 GHz Intel Core i7-3770
- 256 KB L2 Cache, 8 MB L3 Cache
- Performing buffer-constant on various buffer sizes
- Lots of comparisons.

# Performance

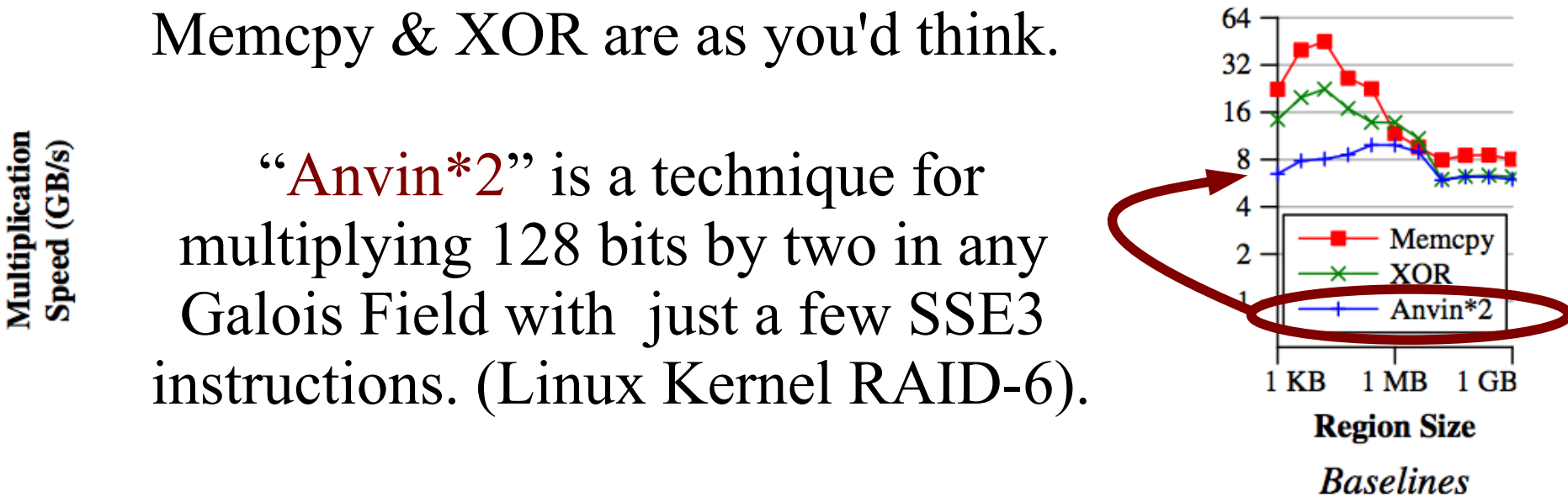



- 3.4 GHz Intel Core i7-3770
- 256 KB L2 Cache, 8 MB L3 Cache
- Performing buffer-constant on various buffer sizes
- Lots of comparisons.

# Performance

Memcpy & XOR are as you'd think.

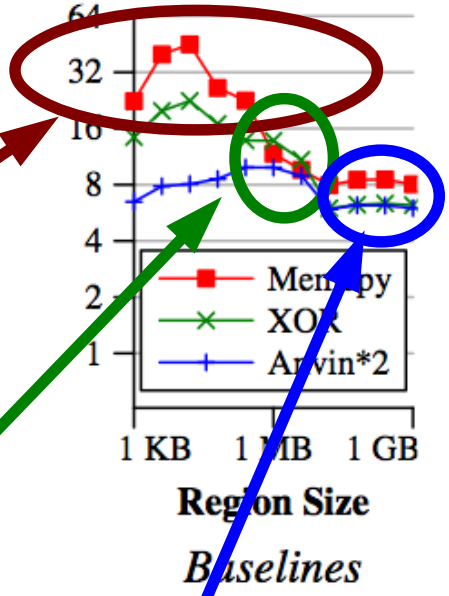
"Anvin*2" is a technique for multiplying 128 bits by two in any Galois Field with just a few SSE3 instructions. (Linux Kernel RAID-6).




- 3.4 GHz Intel Core i7-3770
- 256 KB L2 Cache, 8 MB L3 Cache
- Performing buffer-constant on various buffer sizes
- Lots of comparisons.

# Performance

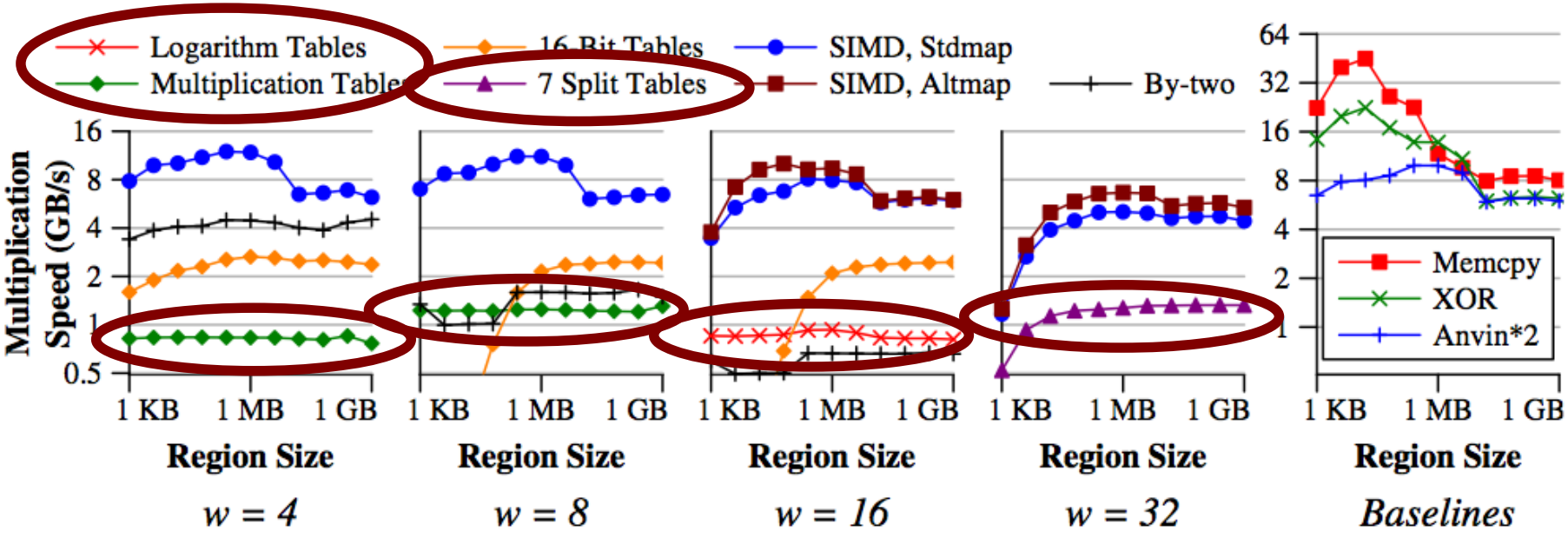
If you're fast enough, you can see effects of saturating the L2 and L3 caches.




- 3.4 GHz Intel Core i7-3770
- 256 KB L2 Cache, 8 MB L3 Cache
- Performing buffer-constant on various buffer sizes
- Lots of comparisons.

Too big

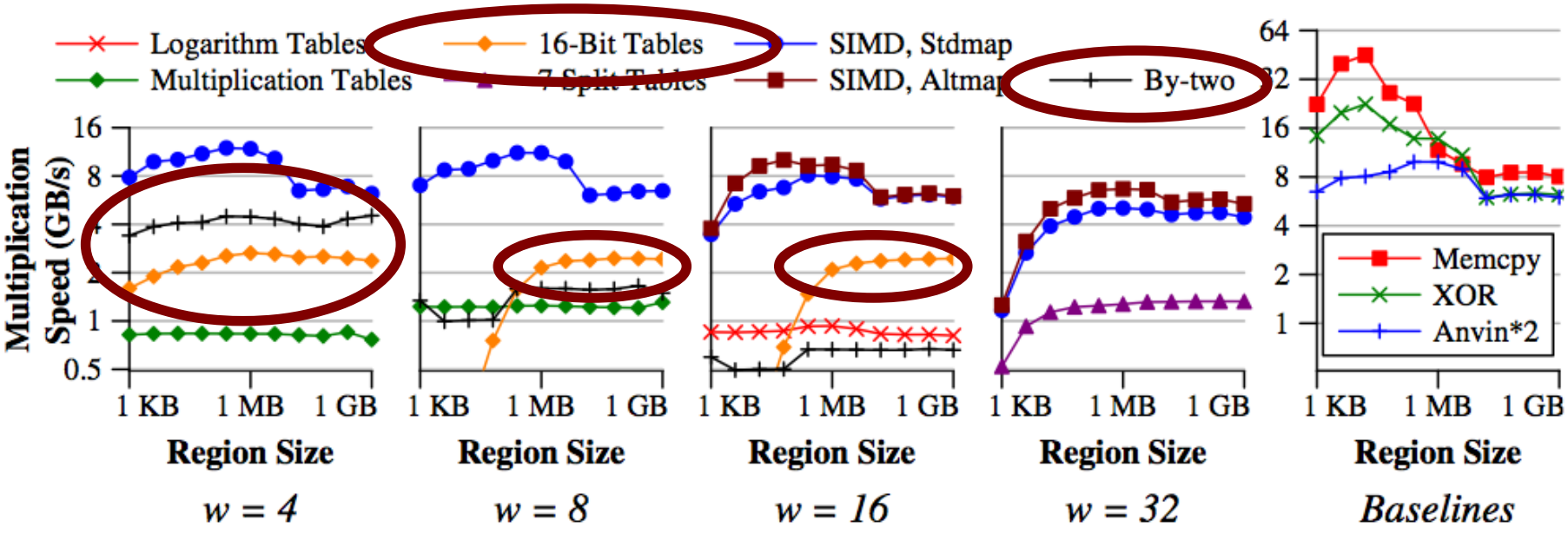
# Performance




Traditional techniques (Rizzo, Jerasure, Onion Networks) don't get close to cache line speeds.

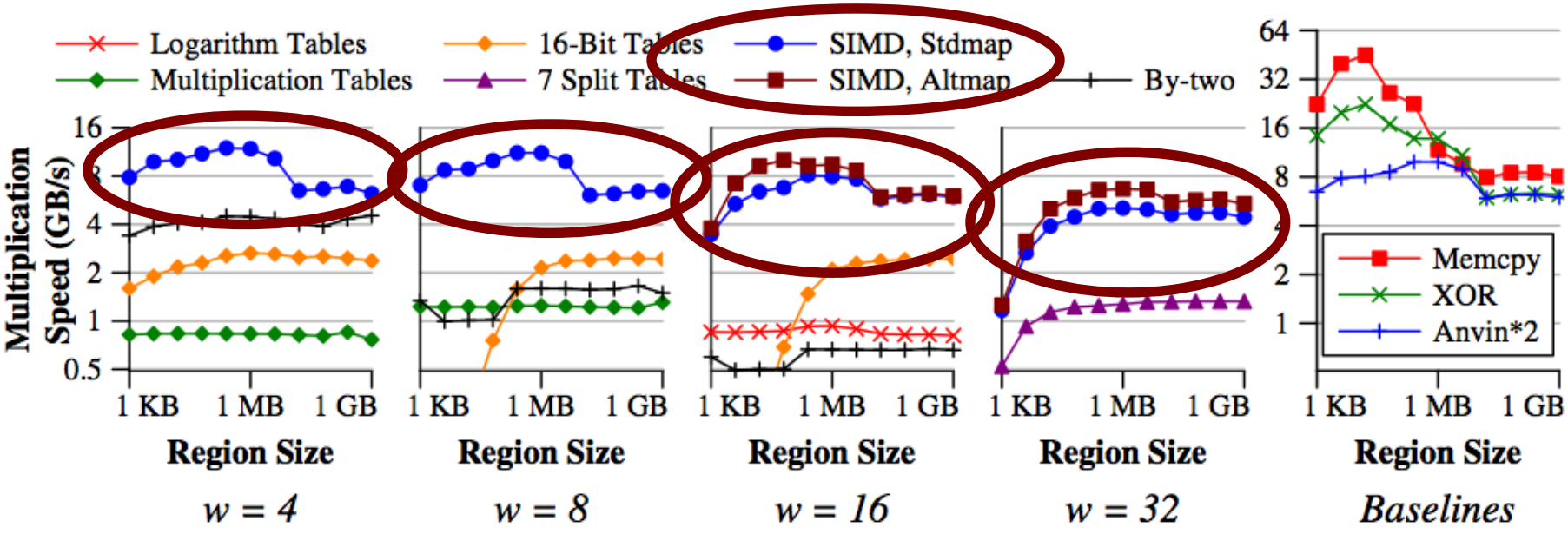
(BTW, both axes are log axes)

# Performance



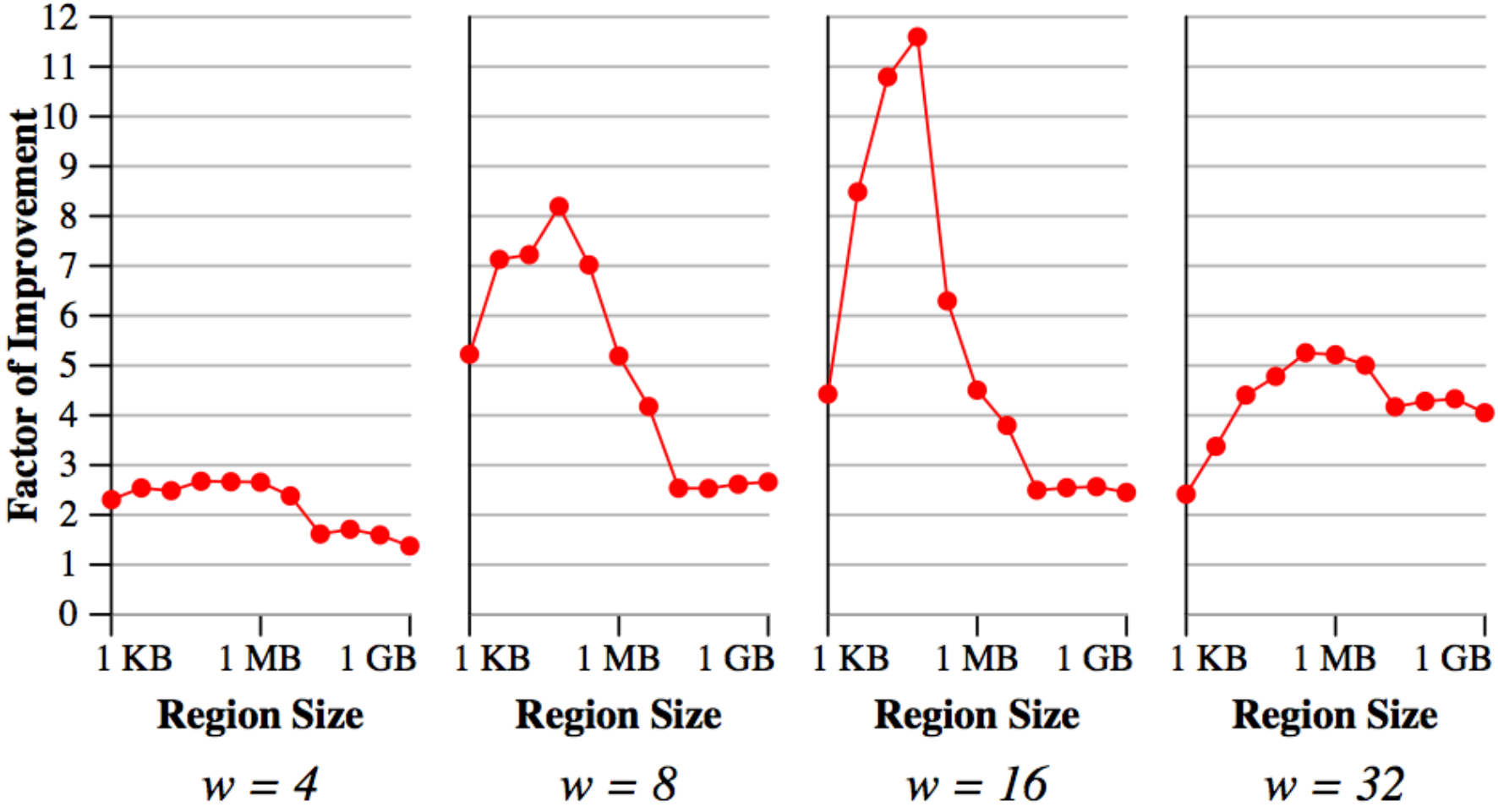

Non-traditional techniques do better, but require amortization for *w=8* and *w=16*.

# Performance



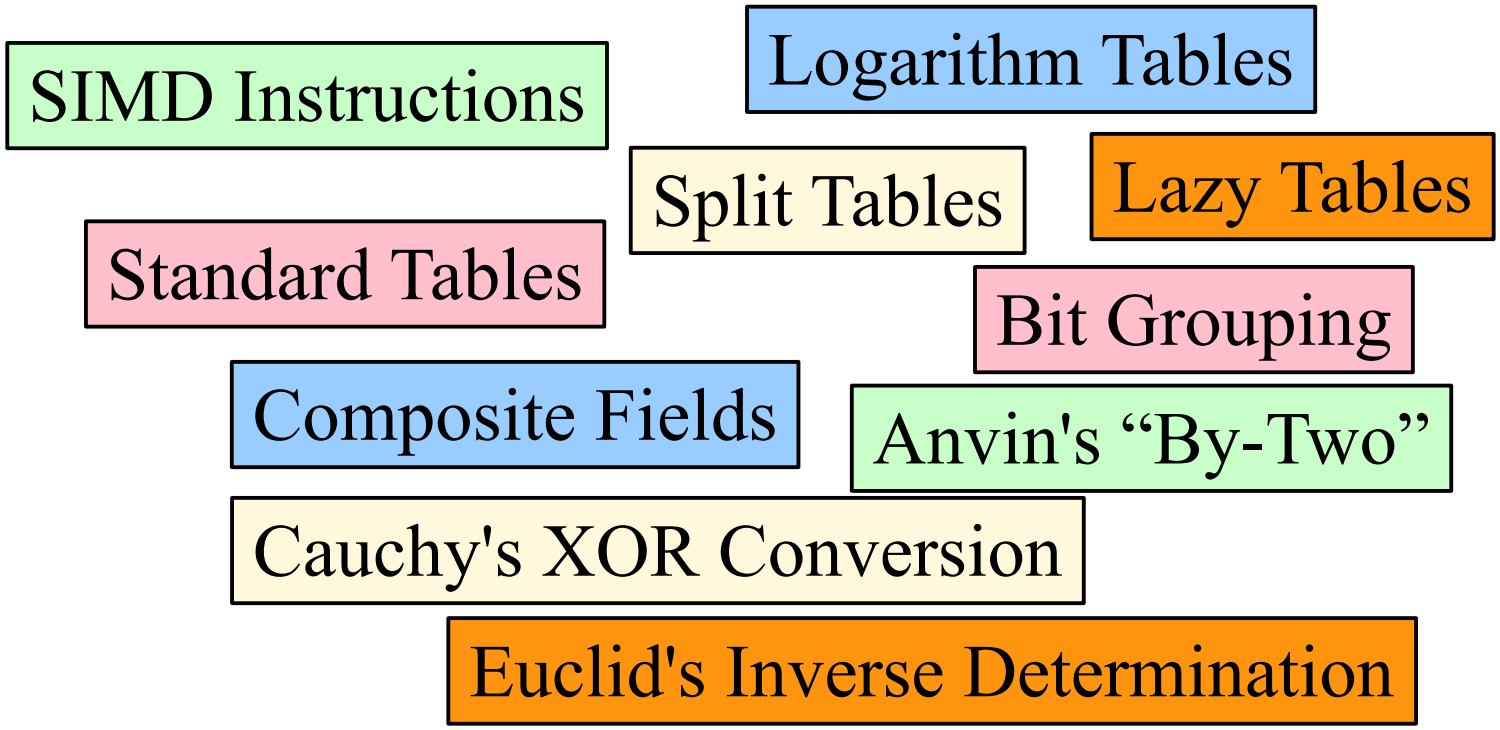

- Our techniques perform identically to "Anvin*2" for *w = 4, 8* and *16*.
  - Cache limited.
- Alternate mapping makes a significant difference.
- *w=16* and *w=32* show some amortization effects.

# Performance



# “GF-Complete” - Released Wednesday

- Big open-source GF arithmetic library in C.




- BSD License. $w = 4, 8, 16, 32, 64, 128.$
- Please use it, and then tell me about it.

# We view this as a Game Changer

- When Galois-Field arithmetic performs as fast as XOR's, it frees up code design.
  - Rotated Reed-Solomon Array Codes
  - Pyramid/LRC Codes (Microsoft)
  - PMDS Codes (IBM)
  - SD Codes
  - All of those regenerating codes

All based on GF Arithmetic

- Code designers no longer handcuffed by XOR's.

# Conclusions

- It's cool
- It's fast
- It's open-source
- It's a game changer

Questions??

# Screaming Fast Galois Field Arithmetic Using Intel SIMD Instructions

James S. Plank
University of Tennessee

*USENIX FAST*
San Jose, CA
February 15, 2013.