Screaming Fast Galois Field Arithmetic Using Intel SIMD Instructions

Ethan L. Miller
Center for Research in Storage Systems
University of California, Santa Cruz
(and Pure Storage)

10 YEARS SDC STORAGE DEVELOPER CONFERENCE SNIA SANTA CLARA, 2013

Authors

Jim Plank
Univ. of Tennessee

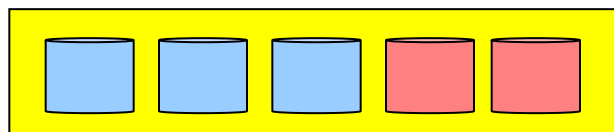
Kevin Greenan
EMC/Data Domain
(now at Box.com)

Ethan Miller
UC Santa Cruz
(and Pure Storage)

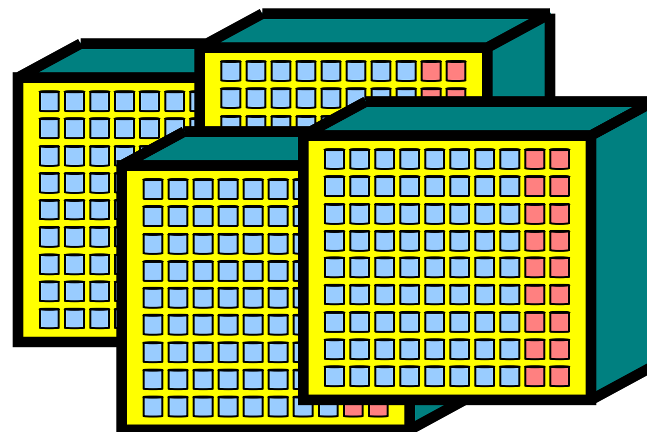
These slides are derived from the presentation Jim gave at FAST 2013 (used with permission).

2013 Storage Developer Conference.
© Ethan L. Miller & James S. Plank   All Rights Reserved.

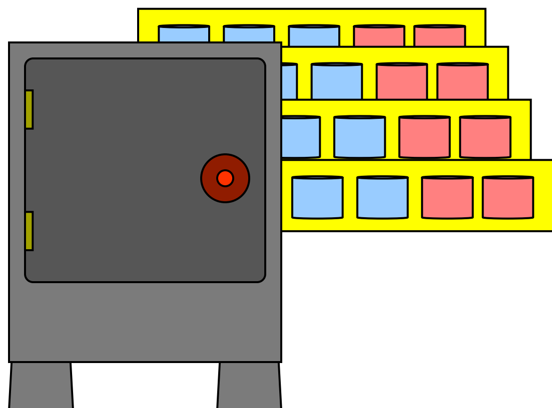
Erasure codes are everywhere



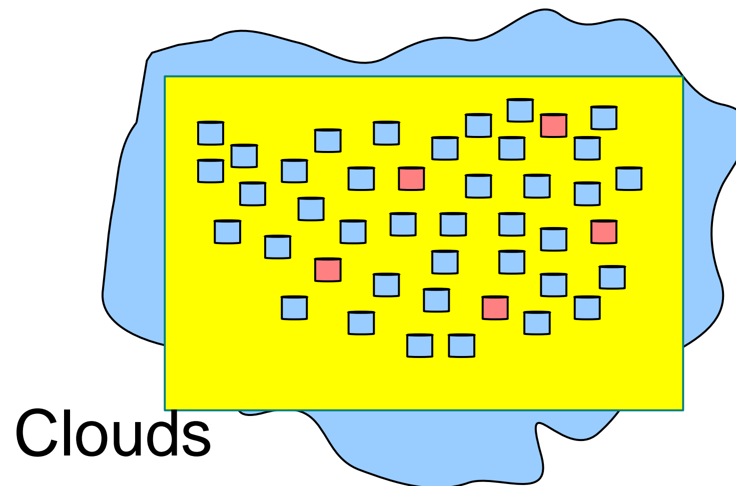
RAID Systems



Data Centers



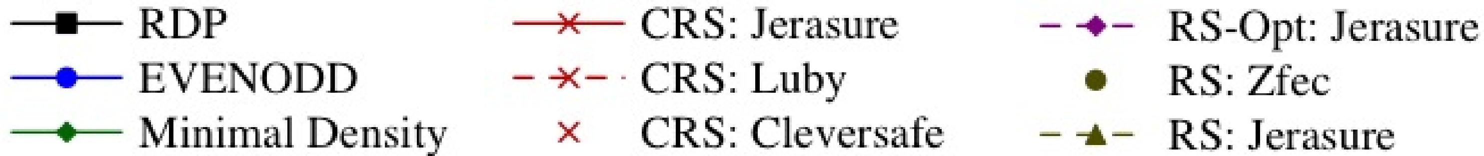
Archival Systems



Clouds

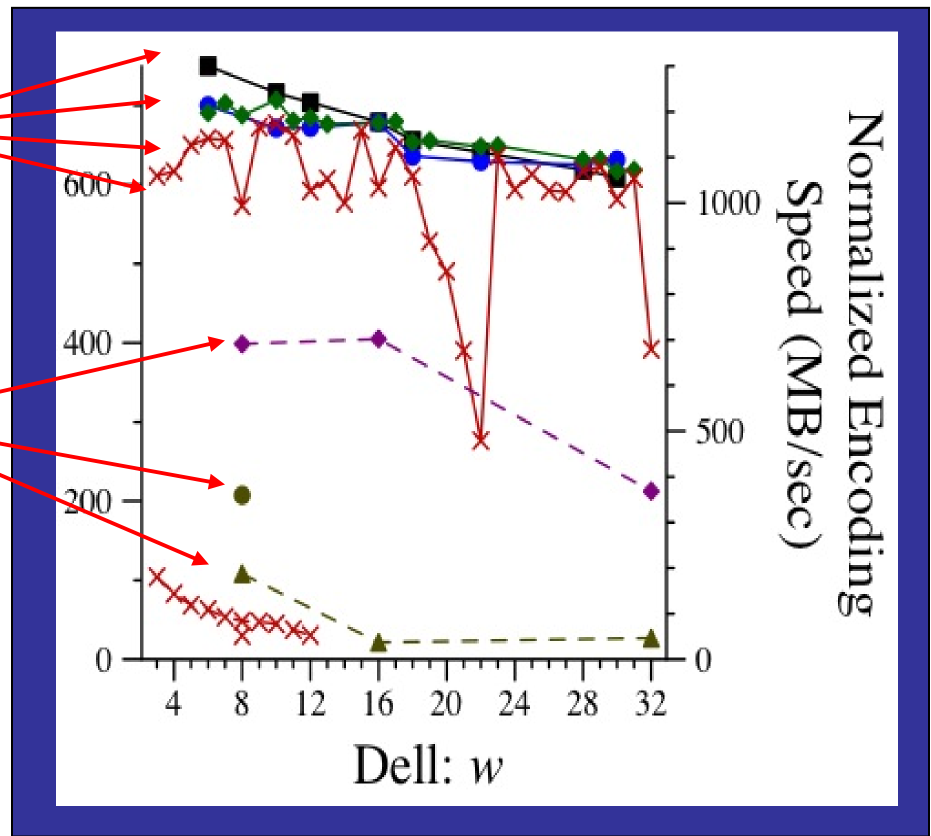
2013 Storage Developer Conference.
© Ethan L. Miller & James S. Plank All Rights Reserved.

Conventional wisdom (FAST 2009)



XOR-Only Codes Are Fast

Reed Solomon Codes are Slow



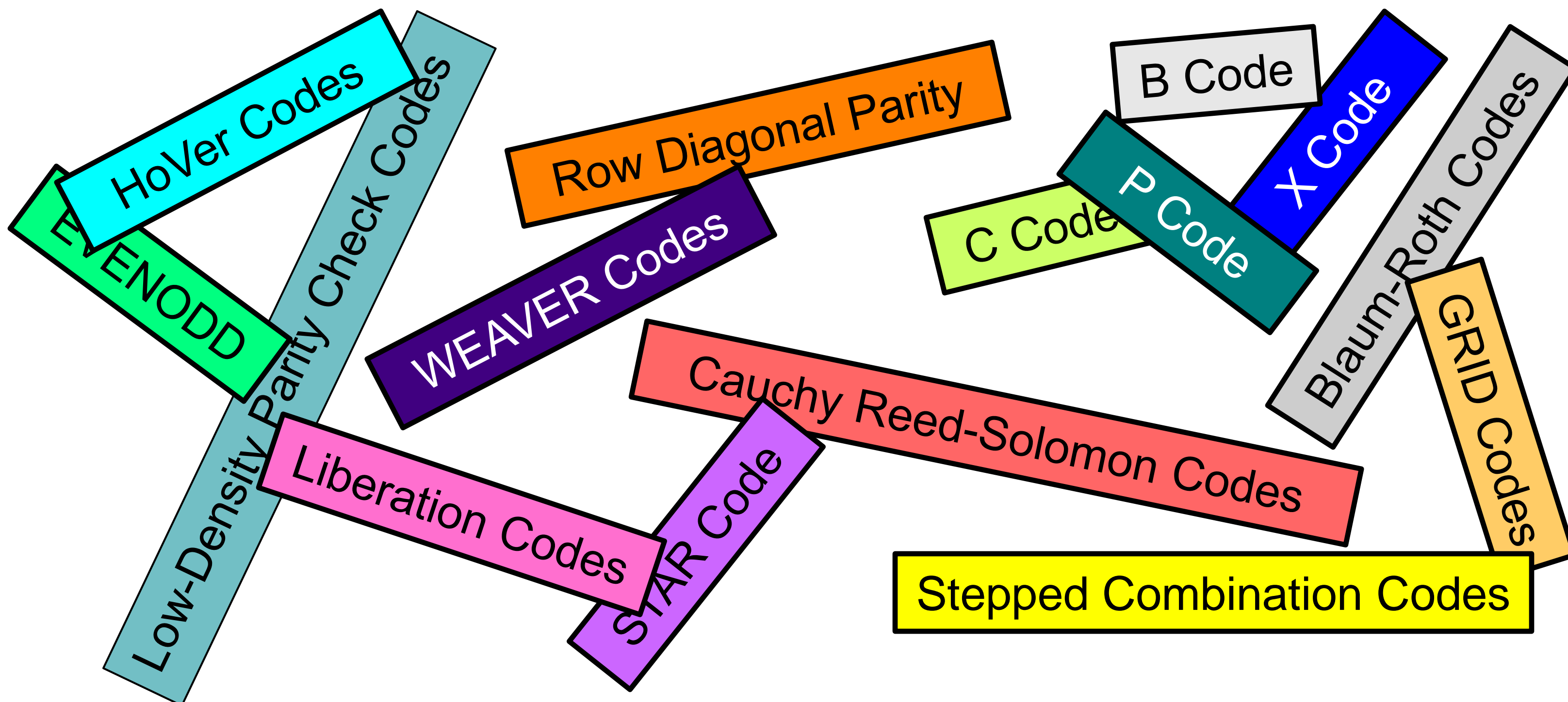
Why?

Because the underlying Galois field multiplication is too slow!

2013 Storage Developer Conference.
© Ethan L. Miller & James S. Plank All Rights Reserved.

Conventional wisdom

- Inconvenient: Reed-Solomon codes are powerful, general and flexible
- Led to a proliferation of XOR-based codes



2013 Storage Developer Conference.
© Ethan L. Miller & James S. Plank All Rights Reserved.

Conventional wisdom says…

- However, in recent years….
  - Eerily smug reports of doing Reed-Solomon coding at “cache line speeds”
  - No need for messy XOR codes!
  - But what’s the secret handshake?
- In this talk, we reveal the secret handshake
  - No prior experience with Galois field arithmetic necessary!

2013 Storage Developer Conference.
© Ethan L. Miller & James S. Plank All Rights Reserved.

Core takeaways

- Using Intel's SSE3 SIMD instructions gets you
  - Galois field arithmetic fast enough that performance is limited by L2/L3 cache
  - Factor of **2.7x** to **12x** faster than previous implementations
  - All on a *single* general-purpose CPU core!
- Open source library: GF-Complete
  - Gives you the secret handshake in a neat package
  - Flexible BSD license

2013 Storage Developer Conference.
© Ethan L. Miller & James S. Plank All Rights Reserved.

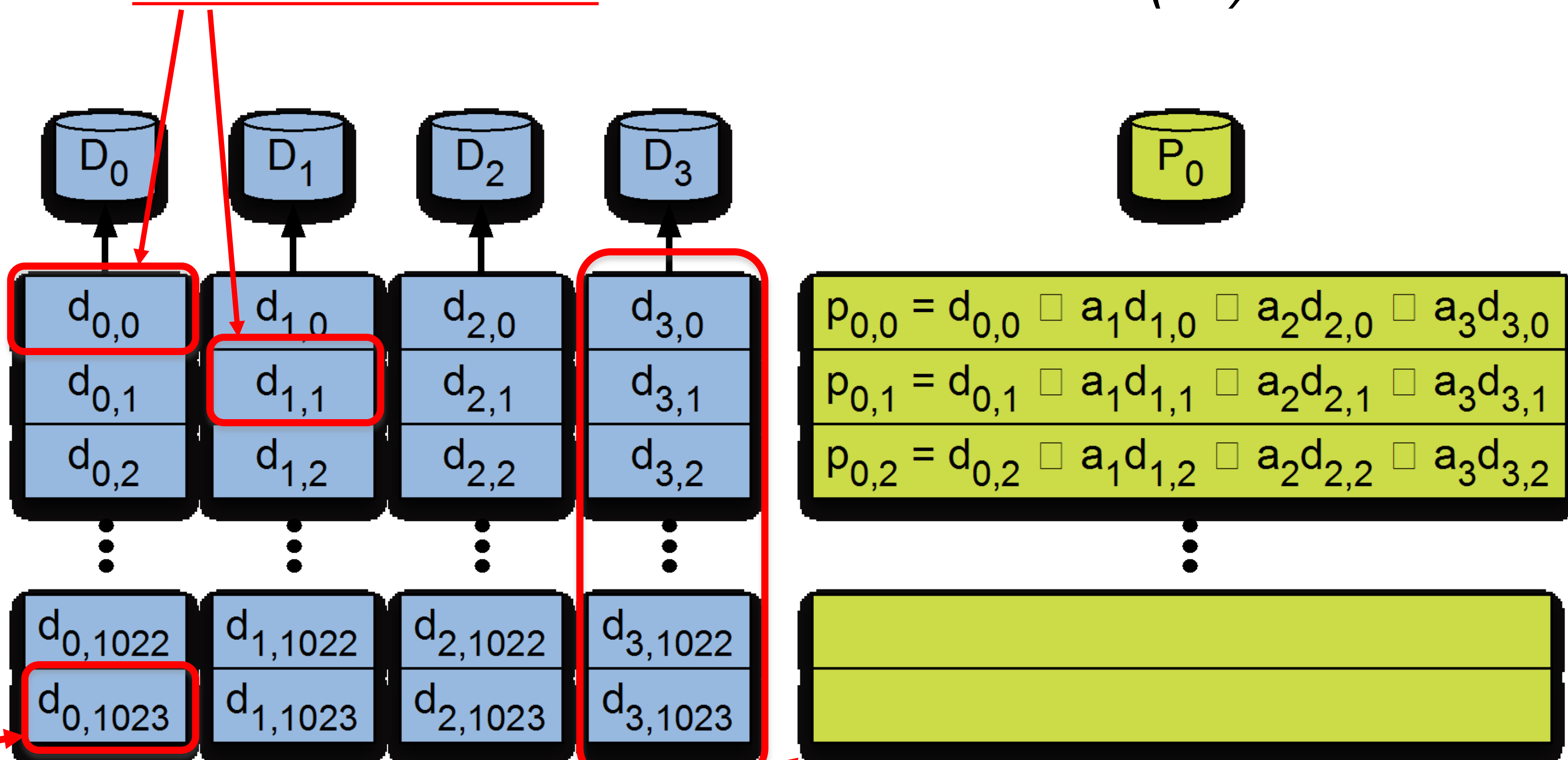
What *is* a Galois field?

- Galois field is also known as a *finite field*
- Contains a finite set of elements
  - Field with *k* elements is called GF(*k*)
  - Often, *k* is a power of 2: $GF(2^w)$
- Supports two operations: add & multiply
  - All results must be elements in the field
  - Additive inverse and multiplicative inverse
  - Usual rules apply (associative, distributive, etc.)
  - Add is done by XOR
  - Multiplication is … more difficult

2013 Storage Developer Conference.
© Ethan L. Miller & James S. Plank All Rights Reserved.

How do storage systems use Galois field arithmetic?

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



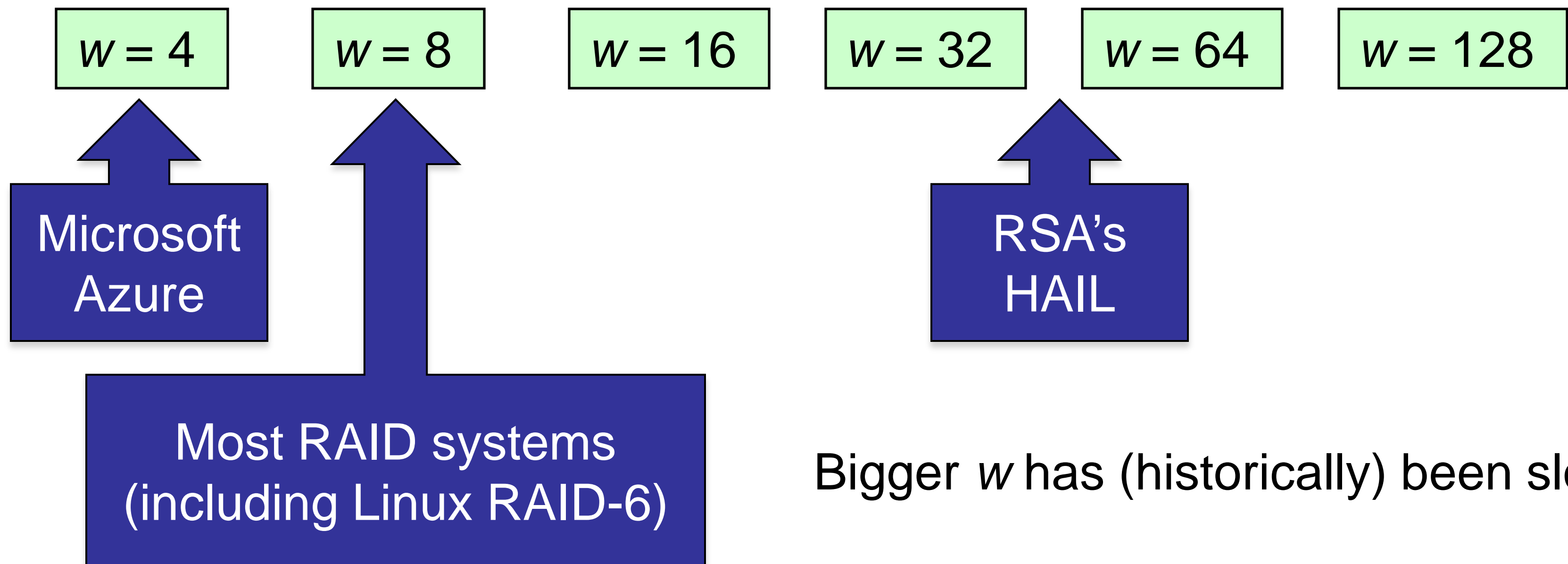
If $w = 8$, this is a byte

and this is a kilobyte

2013 Storage Developer Conference.
© Ethan L. Miller & James S. Plank All Rights Reserved.

How do we pick *w*?

- *w*: the number of bits in each element
  - Small *w* limits the width of each stripe

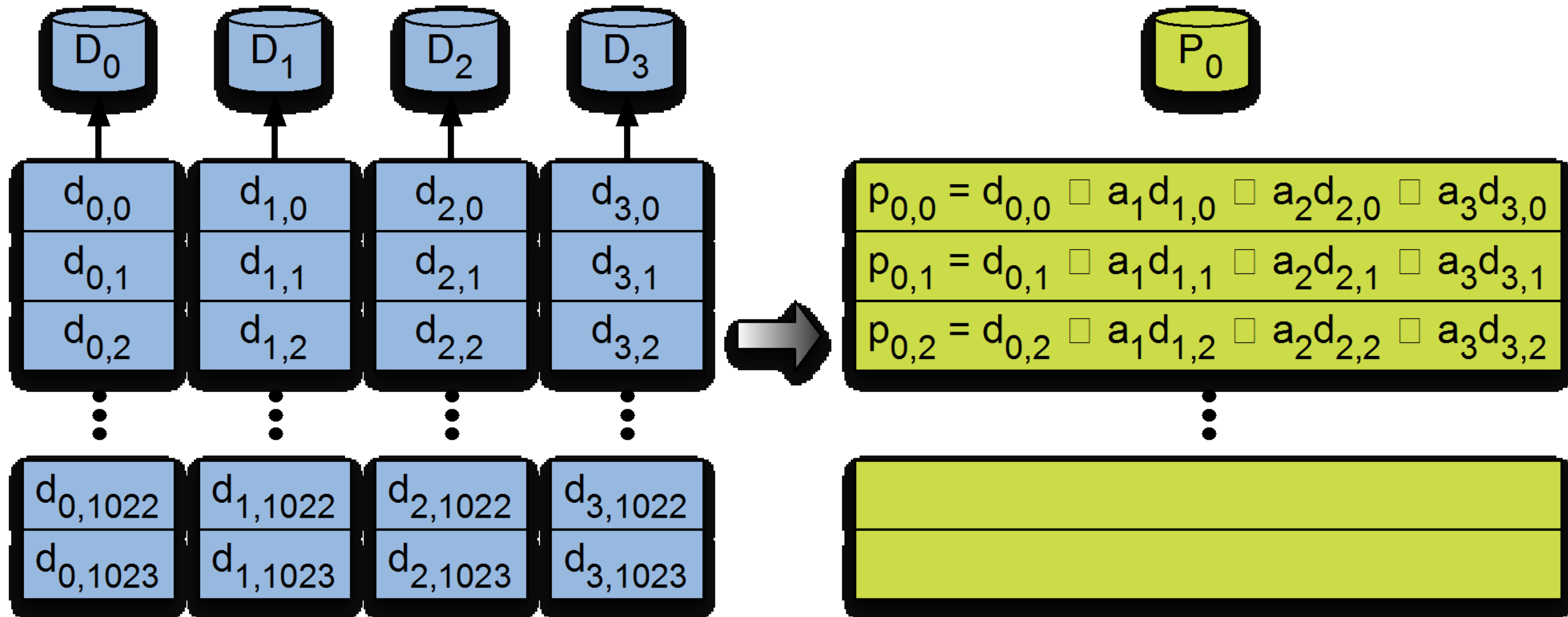


Bigger *w* has (historically) been slower

2013 Storage Developer Conference.
© Ethan L. Miller & James S. Plank All Rights Reserved.

What operations do we need?

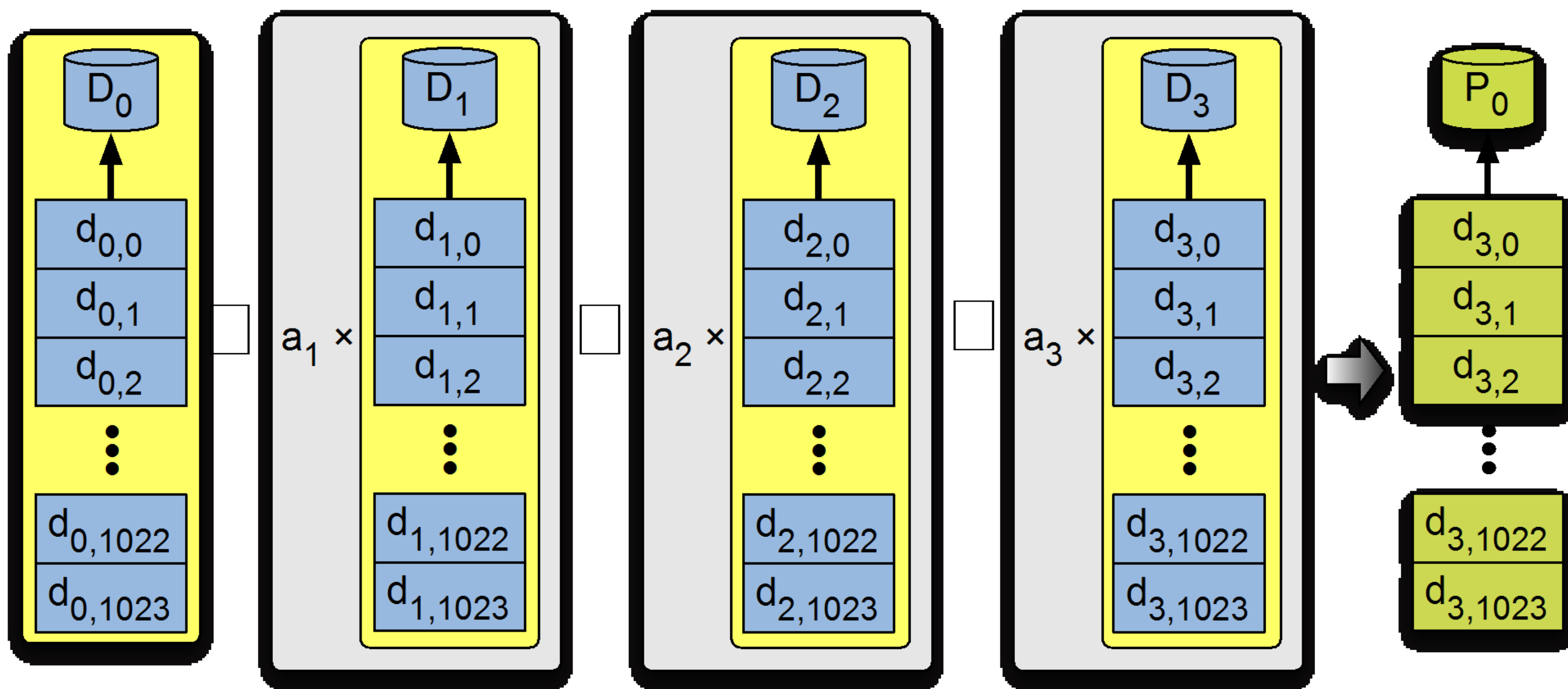
- Required operations are
  - XOR two regions of memory together (addition)
  - Multiply a region of memory by a constant in GF(2^w)



2013 Storage Developer Conference.
© Ethan L. Miller & James S. Plank All Rights Reserved.

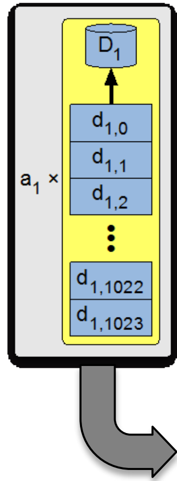
Using multiplication and XOR to generate a code symbol

- Requires *n*-1 XORs and *n*-1 multiplications
  - Need to multiply each data symbol by a (usually different) constant

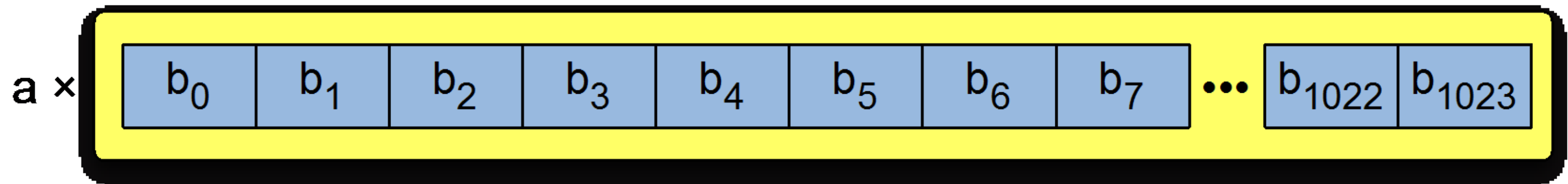


2013 Storage Developer Conference.
© Ethan L. Miller & James S. Plank All Rights Reserved.

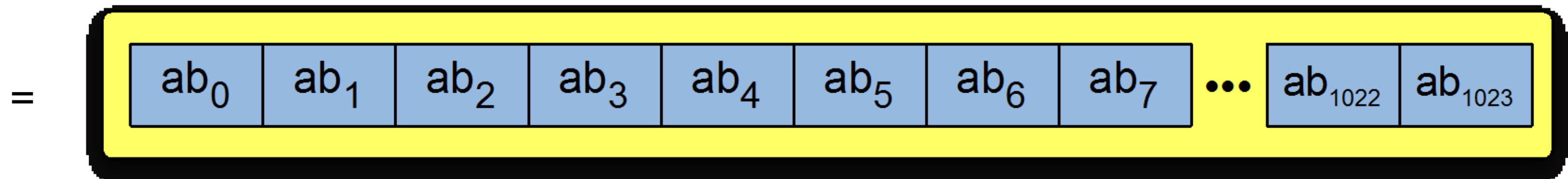
Performing fast multiplication



Common (non-trivial) operation:
multiply a 1K (large) vector of words (b_i) in GF(2^8) by a constant *a*

a ×


Result should look like 1024 individual multiplications…

=


… but doing 1024 individual multiplications can be slow!

2013 Storage Developer Conference.
© Ethan L. Miller & James S. Plank All Rights Reserved.

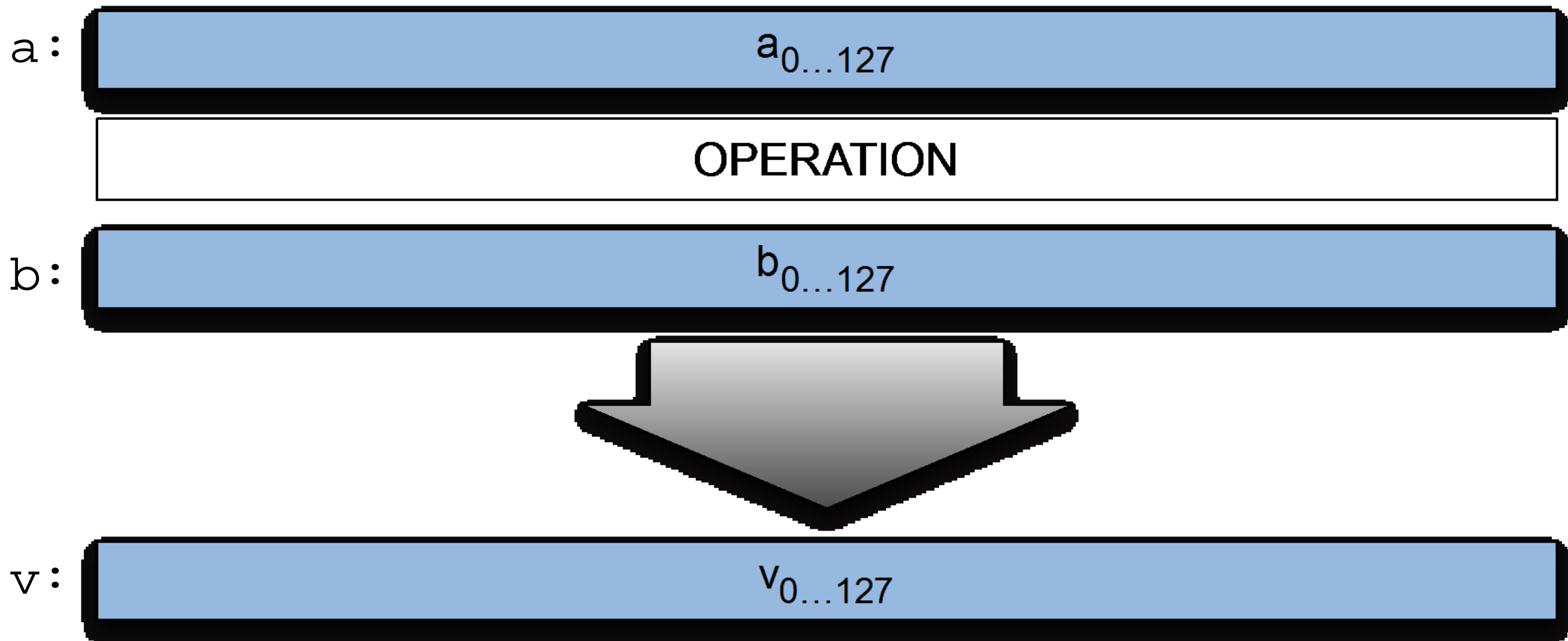
Solution: use vector instructions

- Modern Intel processors support vector operations: Intel "Streaming SIMD" instructions
  - 128 bits per vector
    - 256 for some instructions in newest CPUs
  - Operations done on all elements in parallel
    - Some instructions operate bitwise (*e.g.*, XOR)
    - Others operate on *k*-bit words (*k*=8, 16, 32, 64)
- Other architectures support similar instructions
  - ARM
  - Power

2013 Storage Developer Conference.
© Ethan L. Miller & James S. Plank All Rights Reserved.

Bitwise SIMD instructions

- Bitwise operations
  - XOR: `v = _mm_xor_si128 (a, b)`
  - AND: `v = _mm_and_si128 (a, b)`
- Other bitwise operations also supported

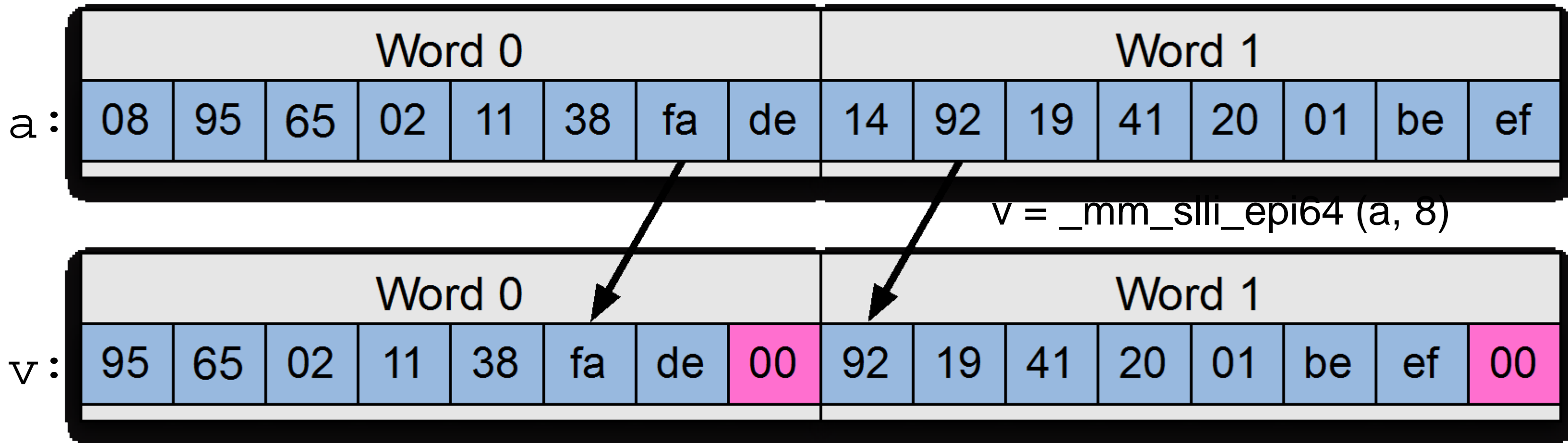


© Ethan L. Miller & James S. Plank All Rights Reserved.

Word-oriented instructions

- Shift left (operates on 64-bit words):

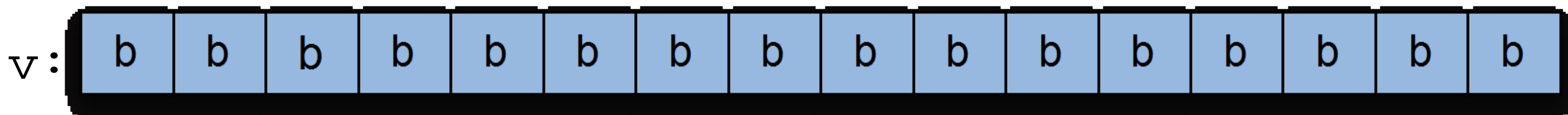
```
v = _mm_slli_epi64 (a, x)
```



- “Load one” (put same value into all 8-bit elements):

```
v = _mm_set1_epi8 (b)
```

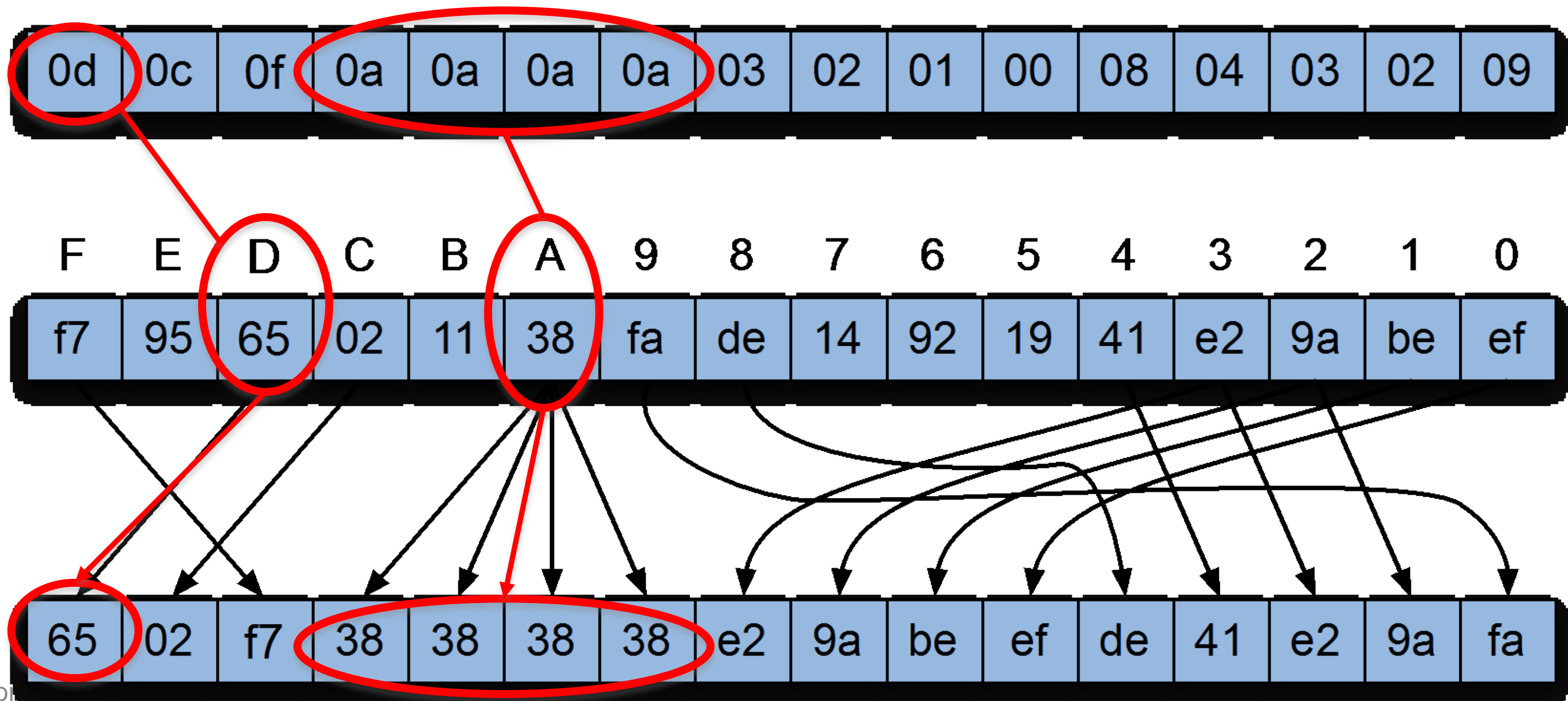
- Not a single instruction—compiler expands it



2013 Storage Developer Conference.
© Ethan L. Miller & James S. Plank All Rights Reserved.

Killer instruction: shuffle

- Shuffle instruction: `v = _mm_shuffle_epi8 (a, x)`
- Performs 16 simultaneous table lookups using
  - a: 16 element table
  - b: 16 indices, each 4 bits long



2013 Sto
© Ethan L. Miller & James S. Plank All Rights Reserved.

Buffer-constant multiply in GF(2^4)

- We can use a single lookup to multiply in GF(2^4)
- Example: multiply 16 bytes *A* by 7 in GF(2^4)

Byte position	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
l_tbl	0b	0c	05	02	04	03	0a	0d	06	01	08	0f	09	0e	07	00
h_tbl	b0	c0	50	20	40	30	a0	d0	60	10	80	f0	90	e0	70	00
l_mask	0f	0f	0f	0f	0f	0f	0f	0f	0f	0f	0f	0f	0f	0f	0f	0f
h_mask	f0	f0	f0	f0	f0	f0	f0	f0	f0	f0	f0	f0	f0	f0	f0	f0
A:	a5	81	65	02	11	38	fa	de	14	92	19	41	e2	9c	be	ef

Setup

```
Calculate l_tbl
h_tbl = _mm_slli_epi64(l_tbl, 4)
l_mask = _mm_set1_epi8(0xf)
h_mask = _m_slli_epi64(l_mask, 4)
```

Since 5 × 7 = 6,
the low order bits should be 6

Since a × 7 = 3,
the high order bits should be 3

2013 Storage Developer Conference.
© Ethan L. Miller & James S. Plank All Rights Reserved.

Where does that table come from?

- The multiplication table is calculated using slower arithmetic
  - Not that slow…
- Similar to calculating multiplication tables for base-10 arithmetic
  - Done by repeated multiply-by-two and reduction
- Details aren't important for now: just treat the table like a lookup table

2013 Storage Developer Conference.
© Ethan L. Miller & James S. Plank All Rights Reserved.

Buffer-constant multiply in GF(2^4)

- Example: multiply 16 bytes *A* by 7 in GF(2^4)

Byte position	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
l_tbl	0b	0c	05	02	04	03	0a	0d	06	01	08	0f	09	0e	07	00
h_tbl	b0	c0	50	20	40	30	a0	d0	60	10	80	f0	90	e0	70	00
l_mask	0f	0f	0f	0f	0f	0f	0f	0f	0f	0f	0f	0f	0f	0f	0f	0f
h_mask	f0	f0	f0	f0	f0	f0	f0	f0	f0	f0	f0	f0	f0	f0	f0	f0
A:	a5	81	65	02	11	38	fa	de	14	92	19	41	e2	9c	be	ef
L = _mm_and_si128(A, l_mask)	05	01	05	02	01	08	0a	0e	04	02	09	01	02	0c	0e	0f
L = _mm_shuffle_epi8(L, l_tbl)	08	07	08	0e	07	0d	03	0c	0f	0e	0a	07	0e	02	0c	0b

Create indices from the low-order bits of each byte in the vector

Perform the table lookup using a shuffle

2013 Storage Developer Conference.
© Ethan L. Miller & James S. Plank All Rights Reserved.

Buffer-constant multiply in GF(2^4)

- Example: multiply 16 bytes *A* by 7 in GF(2^4)

Byte position	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
l_tbl	0b	0c	05	02	04	03	0a	0d	06	01	08	0f	09	0e	07	00
h_tbl	b0	c0	50	20	40	30	a0	d0	60	10	80	f0	90	e0	70	00
l_mask	0f	0f	0f	0f	0f	0f	0f	0f	0f	0f	0f	0f	0f	0f	0f	0f
h_mask	f0	f0	f0	f0	f0	f0	f0	f0	f0	f0	f0	f0	f0	f0	f0	f0
A:	a5	81	65	02	11	38	fa	de	14	92	19	41	e2	9c	be	ef
L = _mm_and_si128(A, l_mask)	05	01	05	02	01	08	0a	0e	04	02	09	01	02	0c	0e	0f
L = _mm_shuffle_epi8(L, l_tbl)	08	07	08	0e	07	0d	03	0c	0f	0e	0a	07	0e	02	0c	0b
H = _mm_andi_si128(A, h_mask)	a0	80	60	00	10	30	f0	d0	10	90	10	40	e0	90	b0	e0
H = _mm_srli_epi64(H, 4)	0a	08	06	00	01	03	0f	0d	01	09	01	04	0e	09	0b	0e
H = _mm_shuffle_epi8(H, h_tbl)	30	d0	10	00	70	90	b0	50	70	a0	70	f0	c0	a0	40	c0

Create indices from the high-order bits

Perform the table lookup using a shuffle

2013 Storage Developer Conference.
© Ethan L. Miller & James S. Plank All Rights Reserved.

Buffer-constant multiply in GF(2^4)

- Example: multiply 16 bytes *A* by 7 in GF(2^4)

Byte position	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
l_tbl	0b	0c	05	02	04	03	0a	0d	06	01	08	0f	09	0e	07	00
h_tbl	b0	c0	50	20	40	30	a0	d0	60	10	80	f0	90	e0	70	00
l_mask	0f	0f	0f	0f	0f	0f	0f	0f	0f	0f	0f	0f	0f	0f	0f	0f
h_mask	f0	f0	f0	f0	f0	f0	f0	f0	f0	f0	f0	f0	f0	f0	f0	f0
A:	a5	81	65	02	11	38	fa	de	14	92	19	41	e2	9c	be	ef
L = _mm_and_si128(A, l_mask)	05	01	05	02	01	08	0a	0e	04	02	09	01	02	0c	0e	0f
L = _mm_shuffle_epi8(L, l_tbl)	08	07	08	0e	07	0d	03	0c	0f	0e	0a	07	0e	02	0c	0b
H = _mm_andi_si128(A, h_mask)	a0	80	60	00	10	30	f0	d0	10	90	10	40	e0	90	b0	e0
H = _mm_srli_epi64(H, 4)	0a	08	06	00	01	03	0f	0d	01	09	01	04	0e	09	0b	0e
H = _mm_shuffle_epi8(H, h_tbl)	30	d0	10	00	70	90	b0	50	70	a0	70	f0	c0	a0	40	c0
R = _mm_xor_si128 (H, L)	38	d7	18	0e	77	9d	b3	5c	7f	ae	7a	f7	ce	a2	4c	cb

6 instructions ➡ 32 multiplications!

XOR the two products, and you're done!

© Ethan L. Miller & James S. Plank All Rights Reserved.

How about GF(2^8)?

- Split each symbol into two 4-bit pieces
- Use the distributive law of multiplication
- All operands and results are 8 bits

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

Two different tables!

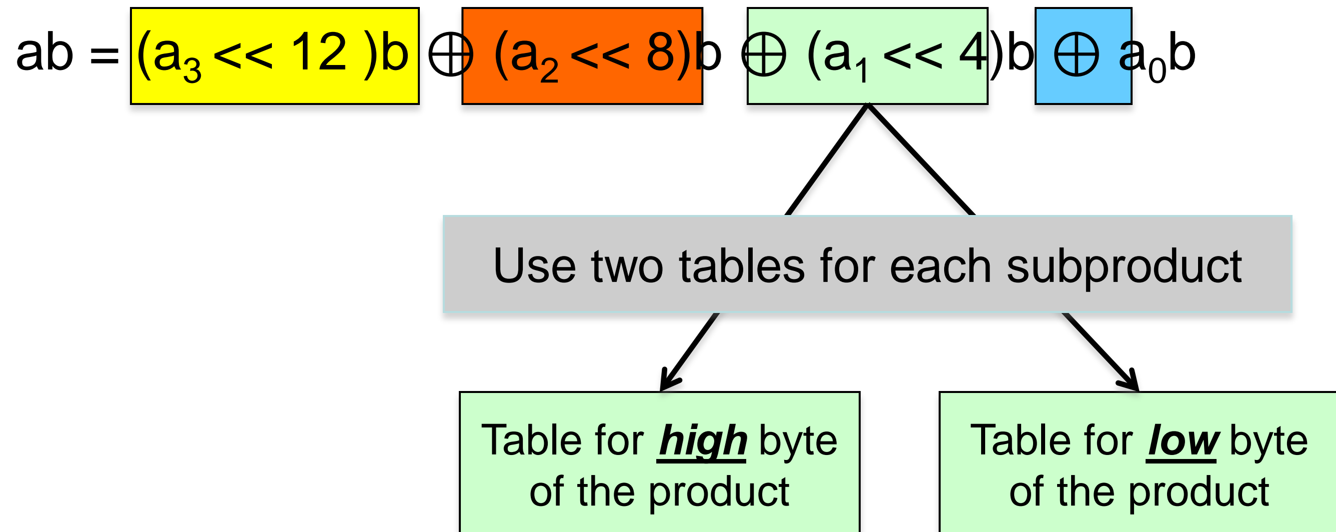
Example: $0xe4 = 0xe0 \oplus 0x04$ ⟶ $0x85 \times 0xe4 = 0x85 \times 0xe0 \oplus 0x85 \times 0x$

- Need to calculate h_tbl in a similar way to how we calculated l_tbl
- Otherwise, code is identical to GF(2^4)!

2013 Storage Developer Conference.
© Ethan L. Miller & James S. Plank All Rights Reserved.

Does this work for GF(2^{16})?

- We can still use the distributive law, but…
- Operands and results are 16 bits
  - Tables can only handle 8 bits at a time!



4 pieces in each 16-bit word, and 2 tables per piece = **8 total tables**

2013 Storage Developer Conference.
© Ethan L. Miller & James S. Plank All Rights Reserved.

Mapping of words to memory matters

- Standard mapping of 16-bit words *a-h* to 128 bit vector (each box is 4 bits)

- Requires 8 table lookups for 8 products

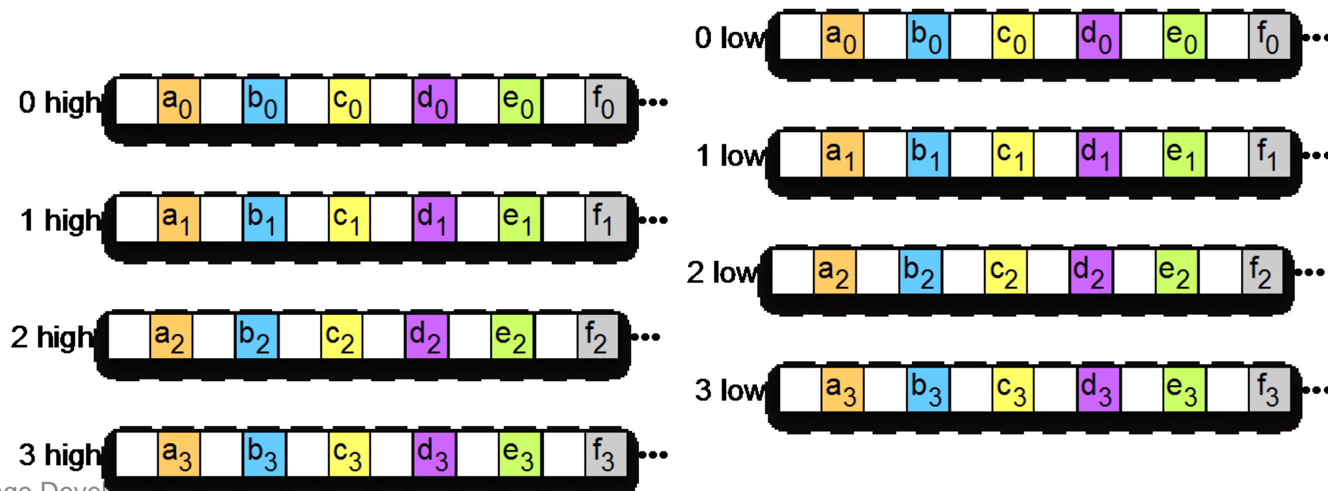
0 low				a_0				b_0				c_0				d_0				e_0				f_0				g_0				h_0
0 high		a_0				b_0				c_0				d_0				e_0				f_0				g_0				h_0		
1 low				a_1				b_1				c_1				d_1				e_1				f_1				g_1				h_1
1 high		a_1				b_1				c_1				d_1				e_1				f_1				g_1				h_1		
2 low				a_2				b_2				c_2				d_2				e_2				f_2				g_2				h_2
2 high		a_2				b_2				c_2				d_2				e_2				f_2				g_2				h_2		
3 low				a_3				b_3				c_3				d_3				e_3				f_3				g_3				h_3
3 high		a_3				b_3				c_3				d_3				e_3				f_3				g_3				h_3		

© Ethan L. Miller & James S. Plank All Rights Reserved.

Mapping of words to memory matters

- Alternate mapping: split each 16-bit word over two 128-bit vectors

- Still requires 8 table lookups for 8 products, but now we get **256** bits for our effort



2013 Storage Developer Conference.
© Ethan L. Miller & James S. Plank All Rights Reserved.

GF(2^{16}) mappings

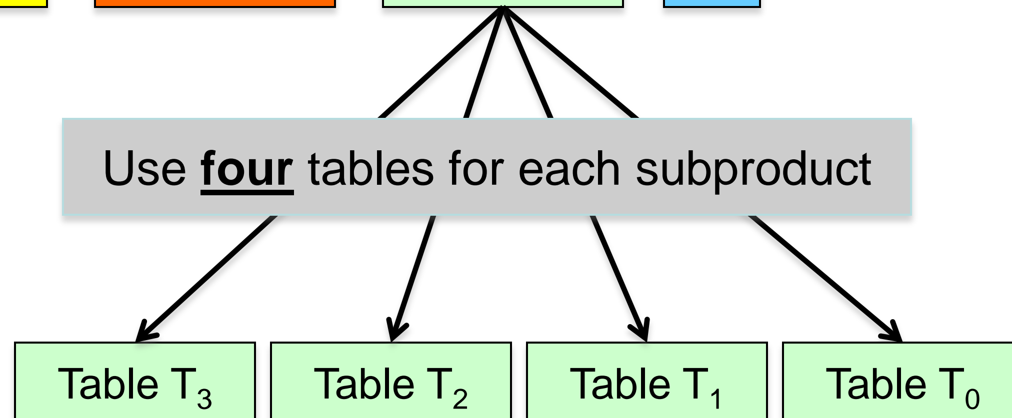
- This is called the *alternate mapping*
  - Has all the properties needed for Reed-Solomon coding
  - May be confusing: it's harder to "read" memory
- Conversions are simple and fast
  - Standard ➡ alternate: 7 SIMD instructions
  - Alternate ➡ standard: 2 SIMD instructions
  - **But you don't need to do this if you don't want to!**

2013 Storage Developer Conference.
© Ethan L. Miller & James S. Plank All Rights Reserved.

Does this work for GF(2^{32})?

- Again, we can use the distributive law, but…
- Operands and results are now 32 bits
  - 8 sub-products × 4 tables each ➡ 32 tables!

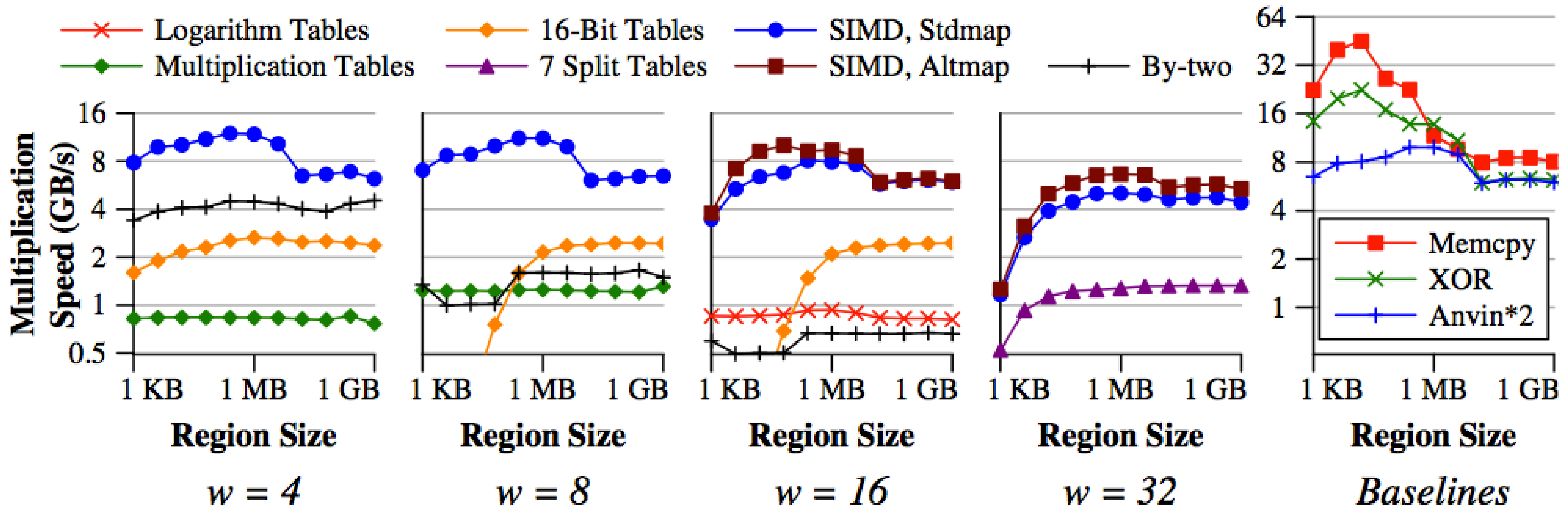
$$ab = (a_7 << 28)b \oplus (a_6 << 24)b \oplus (a_5 << 20)b \oplus (a_4 << 16)b$$
$$(a_3 << 12)b \oplus (a_2 << 8)b \oplus (a_1 << 4)b \oplus a_0 b$$



The same alternate mapping trick can be used here, too.

2013 Storage Developer Conference.
© Ethan L. Miller & James S. Plank All Rights Reserved.

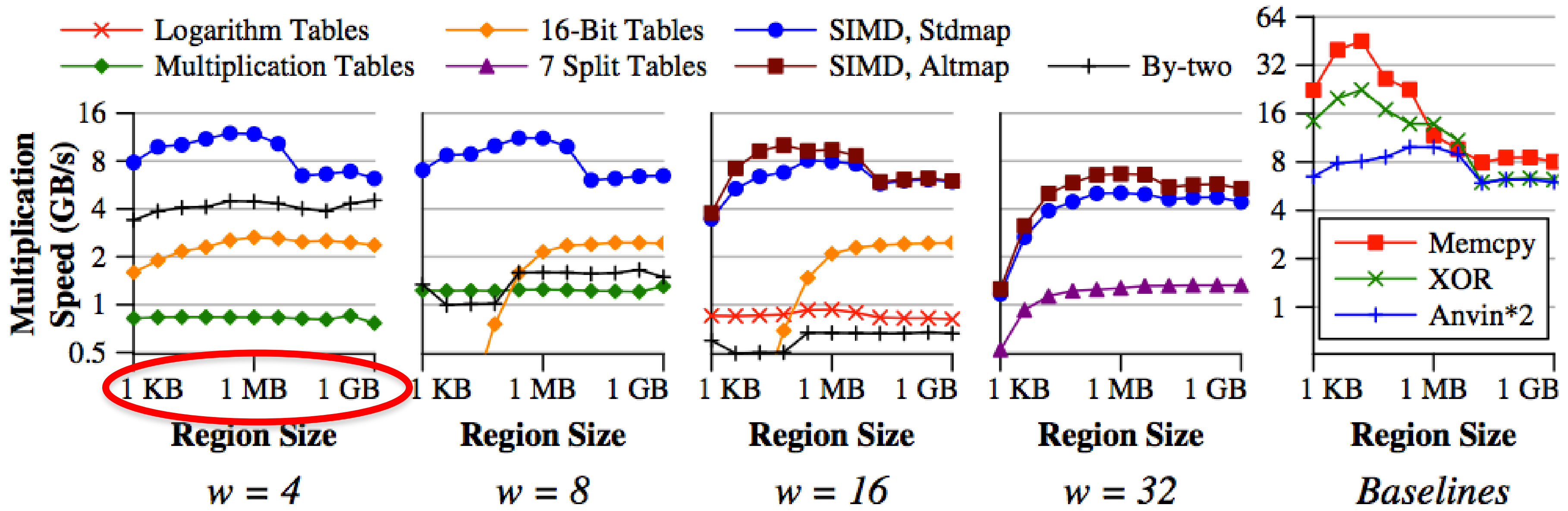
Performance: overview



- 3.4 GHz Intel Core i7-3770
  - 256 KB L2 cache, 8 MB L3 cache
- Running buffer-constant multiply on various buffer sizes
- Lots of comparisons…

© Ethan L. Miller & James S. Plank All Rights Reserved.

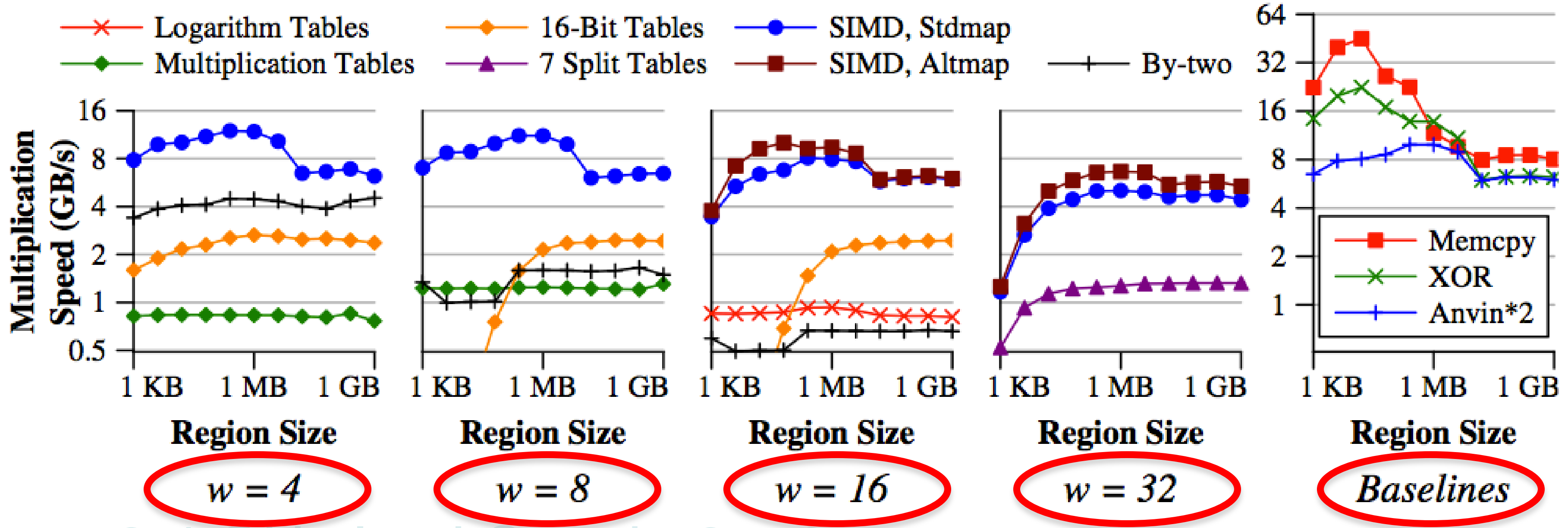
Performance: experiments



- 3.4 GHz Intel Core i7-3770
  - 256 KB L2 cache, 8 MB L3 cache
- Running buffer-constant multiply on various buffer sizes
- Lots of comparisons…

2013 Storage Developer Conference.
© Ethan L. Miller & James S. Plank All Rights Reserved.

Performance: experiments

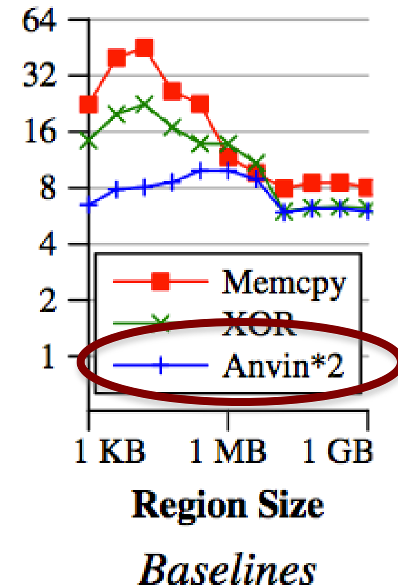


- 3.4 GHz Intel Core i7-3770
  - 256 KB L2 cache, 8 MB L3 cache
- Running buffer-constant multiply on various buffer sizes
- Lots of comparisons…

2013 Storage Developer Conference.
© Ethan L. Miller & James S. Plank All Rights Reserved.

Performance: baselines

- Memcpy & XOR are as you'd think
- Anvin*2 is a technique for multiplying 128 bits by 2 in any Galois field in a few SIMD instructions (from code in the Linux kernel RAID6 driver)



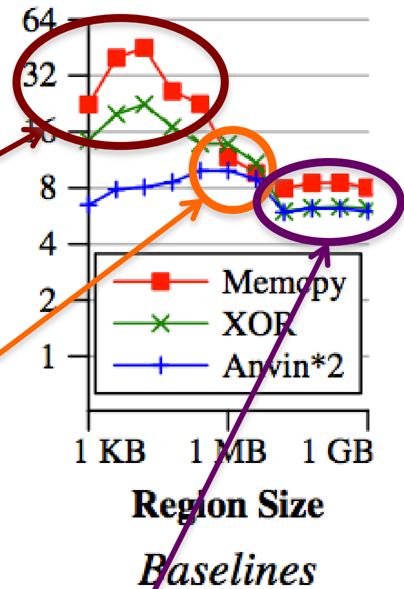
Baselines

- 3.4 GHz Intel Core i7-3770
  - 256 KB L2 cache, 8 MB L3 cache
- Running buffer-constant multiply on various buffer sizes
- Lots of comparisons…

2013 Storage Developer Conference.
© Ethan L. Miller & James S. Plank All Rights Reserved.

Performance: cache saturation

- If your operations are fast enough, you can see cache saturation
  - L2 and L3 caches saturate at different points and speeds

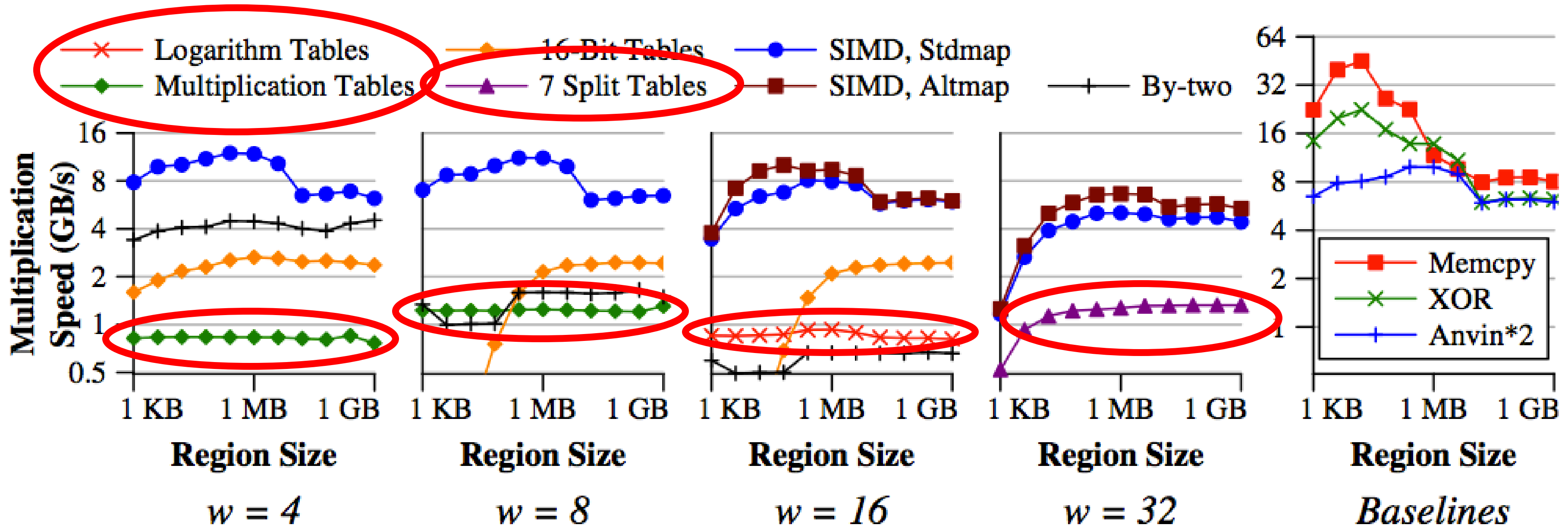


- 3.4 GHz Intel Core i7-3770
  - 256 KB L2 cache, 8 MB L3 cache
- Running buffer-constant multiply on various buffer sizes
- Lots of comparisons…

Too big for cache!

© Ethan L. Miller & James S. Plank All Rights Reserved.

Performance: traditional

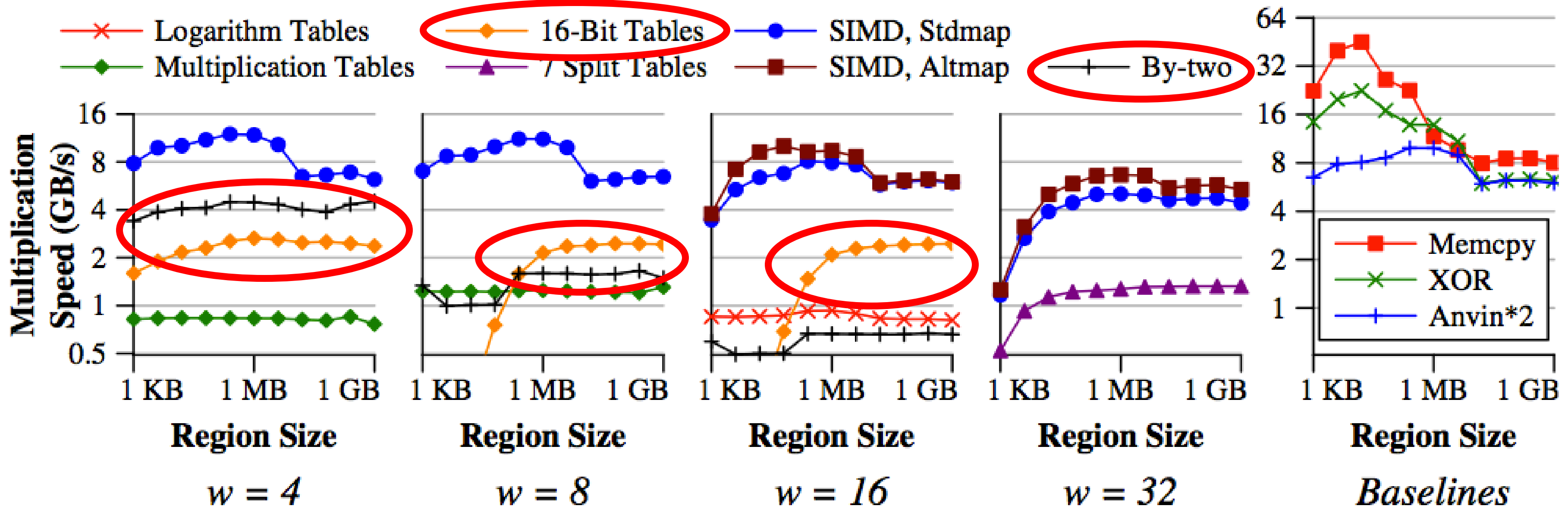


- Traditional techniques don't come close to cache line speeds
  - Rizzo, Jerasure, Onion Networks

NOTE: Both axes use log-scaling

2013 Storage Developer Conference.
© Ethan L. Miller & James S. Plank All Rights Reserved.

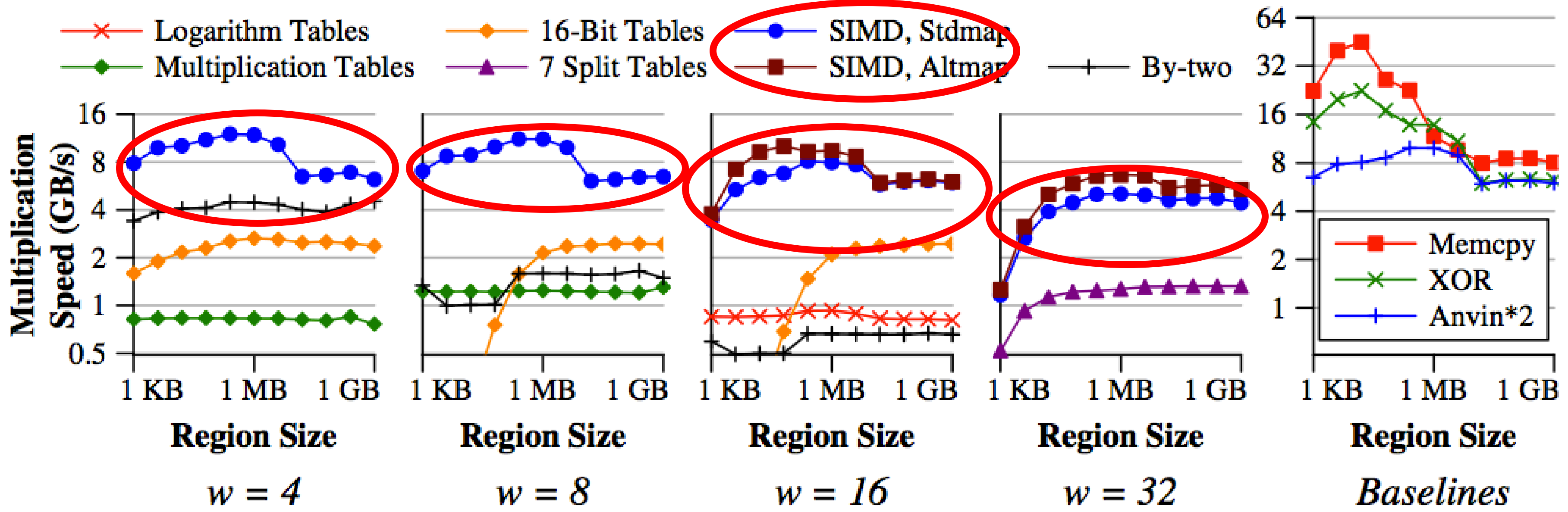
Performance: non-traditional



- Non-traditional techniques do better
  - Require amortization for *w*=8 and *w*=16
  - Not effective for *w*=32
- Still below cache-line speeds

© Ethan L. Miller & James S. Plank All Rights Reserved.

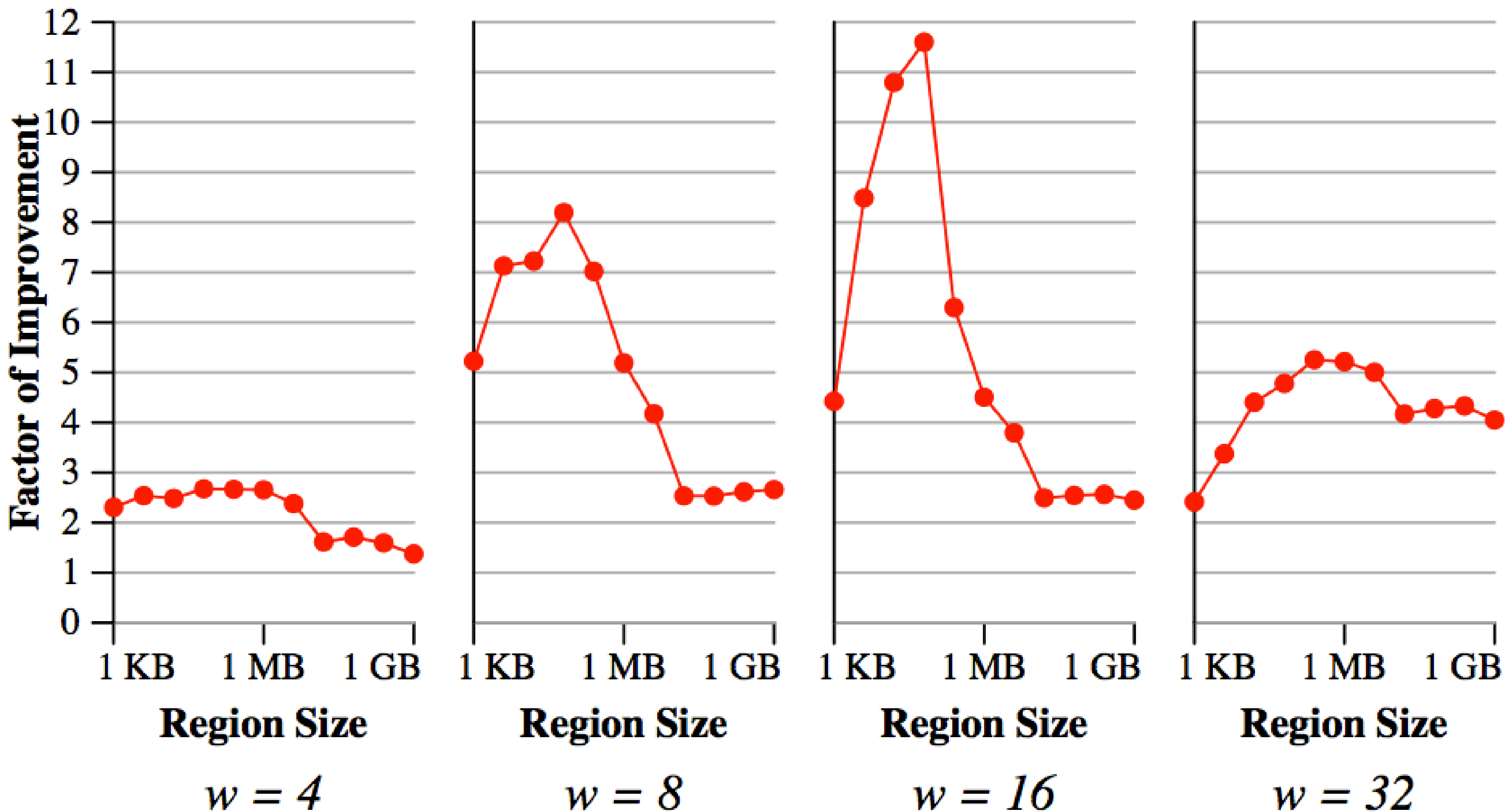
Performance: Intel SIMD



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

© Ethan L. Miller & James S. Plank All Rights Reserved.

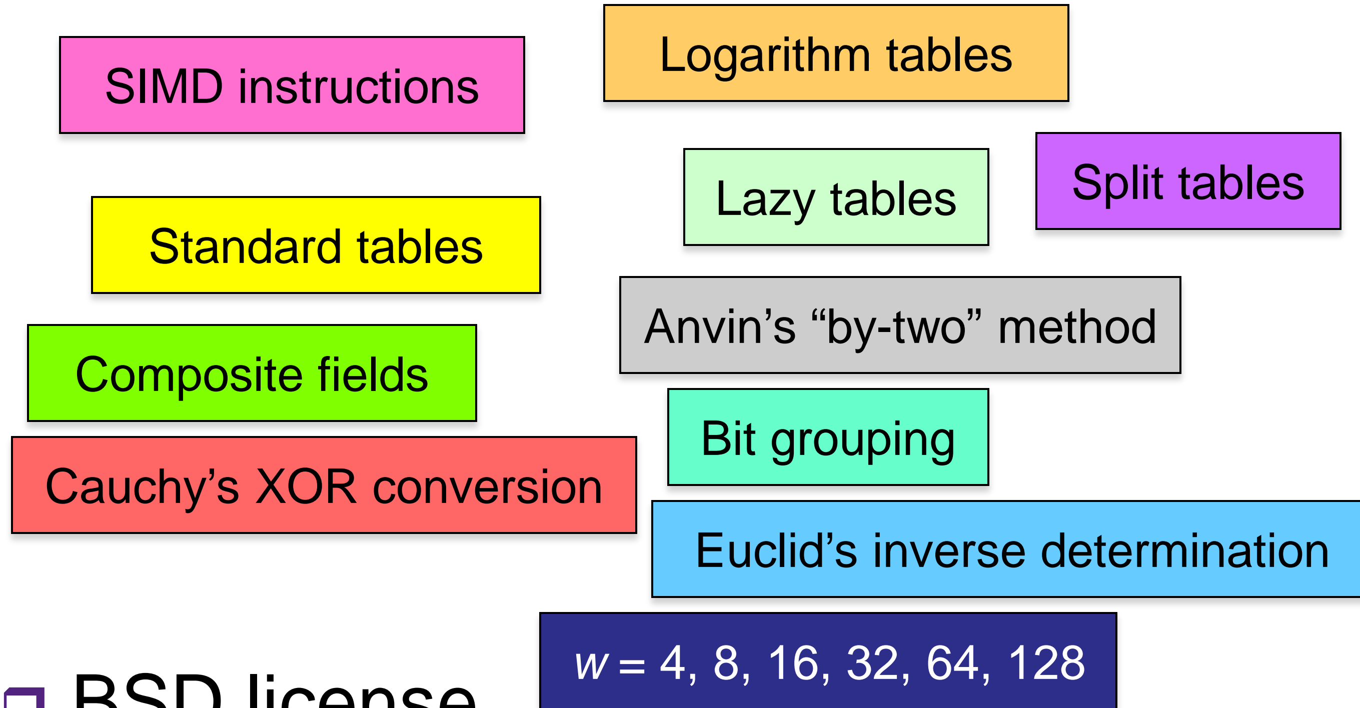
Performance improvement



2013 Storage Developer Conference.
© Ethan L. Miller & James S. Plank All Rights Reserved.

GF-Complete library now available

- Big open-source GF arithmetic library in C



- BSD license
- Please use it, and let us know when you do

2013 Storage Developer Conference.
© Ethan L. Miller & James S. Plank All Rights Reserved.

Where did this work come from?

- 2009: H. Peter Anvin publishes a code sequence for doing fast Galois field arithmetic for RAID6*
- 2010: I implement fast Galois field arithmetic for Pure Storage
- 2012: Jim Plank and I discuss writing a paper at a conference in Asilomar
- 2012–13: We work with Kevin Greenan and some undergrads to write the library
  - Add some new optimizations (you'll learn about)

*http://web.archive.org/web/20090807060018/http://www.kernel.org/pub/linux/kernel/people/hpa/raid6.pdf

2013 Storage Developer Conference.
© Ethan L. Miller & James S. Plank All Rights Reserved.

Where is it going?

- Incorporating new optimizations for the latest Intel instruction sets
  - 256-bit vectors
  - Carry-free multiply for large fields
  - Other optimizations
- Adding erasure code implementations
  - Investigating optimizations for doing the codes themselves
  - Example: calculate codes across or down?
- We're open to suggestions!

© Ethan L. Miller & James S. Plank All Rights Reserved.

This is a game-changer!

- When Galois field arithmetic runs at XOR speed, it frees up code design
  - Rotated Reed-Solomon array codes
  - Pyramid/LRC codes (Microsoft)
  - PMDS codes (IBM)
  - SD codes
  - Regenerating codes
- Erasure code designers are no longer handcuffed to XORs

2013 Storage Developer Conference.
© Ethan L. Miller & James S. Plank All Rights Reserved.

Conclusions

- GF-Complete is
  - Cool
  - Fast
  - Open-source
  - Ready to use!

2013 Storage Developer Conference.
© Ethan L. Miller & James S. Plank All Rights Reserved.

Questions?

http://bitbucket.org/ethanmiller/gf-complete

Thanks to my collaborators:
Jim Plank, Kevin Greenan, and an army of undergrads at UTK

2013 Storage Developer Conference.
© Ethan L. Miller & James S. Plank All Rights Reserved.