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)
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).
Erasure codes are everywhere

RAID Systems

Archival Systems

Data Centers

Clouds
Conventional wisdom (FAST 2009)

XOR-Only Codes Are Fast

Reed Solomon Codes are Slow

Why?

Because the underlying Galois field multiplication is too slow!
Conventional wisdom

- Inconvenient: Reed-Solomon codes are powerful, general and flexible
- Led to a proliferation of XOR-based codes
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!
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
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
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.
How do we pick $w$?

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

Larger, more complex coding systems
More expensive to implement

$w = 4$  $w = 8$  $w = 16$  $w = 32$  $w = 64$  $w = 128$

Microsoft Azure

Most RAID systems (including Linux RAID-6)

RSA’s HAIL

Bigger $w$ has (historically) been slower
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($2^w$)
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
Performing fast multiplication

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

Result should look like 1024 individual multiplications…

… but doing 1024 individual multiplications can be slow!
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
Bitwise SIMD instructions

- Bitwise operations
  - XOR: \( v = \_\_m\_m\_x\_o\_r\_s\_i128 \ (a, b) \)
  - AND: \( v = \_\_m\_m\_a\_n\_d\_s\_i128 \ (a, b) \)

- Other bitwise operations also supported
Word-oriented instructions

- **Shift left (operates on 64-bit words):**
  
  \[ v = _{\text{mm}}_{\text{slli}}_{\text{epi64}} \ (a, \ x) \]

- **“Load one” (put same value into all 8-bit elements):**
  
  \[ v = _{\text{mm}}_{\text{set1}}_{\text{epi8}} \ (b) \]

- **Not a single instruction—compiler expands it**

\[ v: \boxed{\text{b b b b b b b b b b b b b b b}} \]
Killer instruction: shuffle

- Shuffle instruction: \[ v = \text{mm_shuffle_epi8} (a, x) \]
- Performs 16 simultaneous table lookups using:
  - a: 16 element table
  - b: 16 indices, each 4 bits long
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 $GF(2^4)$

### Calculate \(l\_tbl\)

- \(h\_tbl = \_mm\_slli\_epi64(l\_tbl, 4)\)
- \(l\_mask = \_mm\_set1\_epi8(0xf)\)
- \(h\_mask = \_m\_slli\_epi64(l\_mask, 4)\)

**Setup**

- Since \(5 \times 7 = 6\), the low order bits should be 6
- Since \(a \times 7 = 3\), the high order bits should be 3
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
Buffer-constant multiply in $\text{GF}(2^4)$

Example: multiply 16 bytes $A$ by 7 in $\text{GF}(2^4)$

Byte position: F E D C B A 9 8 7 6 5 4 3 2 1 0

- $l_{\text{tbl}}$: 0b 0c 05 02 04 03 0a 0d 06 01 08 0f 09 0e 07 00
- $h_{\text{tbl}}$: b0 c0 50 20 40 30 a0 d0 60 10 80 f0 90 e0 70 00
- $l_{\text{mask}}$: 0f 0f 0f 0f 0f 0f 0f 0f 0f 0f 0f 0f 0f 0f 0f 0f
- $h_{\text{mask}}$: 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

$L = \_\text{mm\_and\_si128}(A, l_{\text{mask}})$: 05 01 05 02 01 08 0a 0e 04 02 09 01 02 0c 0e 0f

$L = \_\text{mm\_shuffle\_epi8}(L, l_{\text{tbl}})$: 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
Example: multiply 16 bytes $A$ by 7 in $\text{GF}(2^4)$

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

$L = \_\_mm\_and\_si128(A, l\_mask)$
$L = \_\_mm\_shuffle\_epi8(L, l\_tbl)$
$H = \_\_mm\_andi\_si128(A, h\_mask)$
$H = \_\_mm\_srli\_epi64(H, 4)$
$H = \_\_mm\_shuffle\_epi8(H, h\_tbl)$

Create indices from the high-order bits
Perform the table lookup using a shuffle
Buffer-constant multiply in GF(2^4)

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

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

\[
L = _mm_and_si128(A, l\_mask) \\
L = _mm_shuffle_epi8(L, l\_tbl) \\
H = _mm_andi_si128(A, h\_mask) \\
H = _mm_slli_epi64(H, 4) \\
H = _mm_shuffle_epi8(H, h\_tbl) \\
R = _mm_xor_si128 (H, L)
\]

XOR the two products, and you’re done!

6 instructions ➡ 32 multiplications!
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

\[ a = (a_{\text{high}} \ll 4) \oplus a_{\text{low}} \quad ab = (a_{\text{high}} \ll 4)b \oplus a_{\text{low}}b \]

Example: \( 0xe4 = 0xe0 \oplus 0x04 \rightarrow 0x85 \times 0xe4 = 0x85 \times 0xe0 \oplus 0x85 \times 0x \)

- Need to calculate \( h_{\text{tbl}} \) in a similar way to how we calculated \( l_{\text{tbl}} \)
- Otherwise, code is identical to GF(2^4)!
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!

\[ ab = (a_3 \ll 12)b \oplus (a_2 \ll 8)b \oplus (a_1 \ll 4)b \oplus a_0b \]

Use two tables for each subproduct

- Table for **high** byte of the product
- Table for **low** byte of the product

4 pieces in each 16-bit word, and 2 tables per piece = **8 total tables**
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
Mapping of words to memory matters

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

  Vector of high bytes
  \[
  \begin{array}{cccccccc}
  a_3 & a_2 & b_3 & b_2 & c_3 & c_2 & d_3 & d_2 \\
  e_3 & e_2 & f_3 & f_2 & & & & \\
  \end{array}
  \]

  Vector of low bytes
  \[
  \begin{array}{cccccccc}
  a_1 & a_0 & b_1 & b_0 & c_1 & c_0 & d_1 & d_0 \\
  e_1 & e_0 & f_1 & f_0 & & & & \\
  \end{array}
  \]

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

  Vector of high bytes
  \[
  \begin{array}{cccccccc}
  a_0 & b_0 & c_0 & d_0 & e_0 & f_0 & & \\
  \end{array}
  \]

  Vector of low bytes
  \[
  \begin{array}{cccccccc}
  a_1 & b_1 & c_1 & d_1 & e_1 & f_1 & & \\
  \end{array}
  \]

  Vector of high bytes
  \[
  \begin{array}{cccccccc}
  a_2 & b_2 & c_2 & d_2 & e_2 & f_2 & & \\
  \end{array}
  \]

  Vector of low bytes
  \[
  \begin{array}{cccccccc}
  a_3 & b_3 & c_3 & d_3 & e_3 & f_3 & & \\
  \end{array}
  \]
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 $\Rightarrow$ alternate: 7 SIMD instructions
  - Alternate $\Rightarrow$ standard: 2 SIMD instructions
- But you don’t need to do this if you don’t want to!
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!

$ab = (a_7 << 28)b \oplus (a_6 << 24)b \oplus (a_5 << 20)b \oplus (a_4 << 16)b$

- $a_3 << 12)b \oplus (a_2 << 8)b \oplus (a_1 << 4)b \oplus a_0b$

Use **four** tables for each subproduct

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

- 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…
Performance: traditional

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

NOTE: Both axes use log-scaling
Non-traditional techniques do better
- Require amortization for $w=8$ and $w=16$
- Not effective for $w=32$
- Still below cache-line speeds
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
Performance improvement

![Graphs showing performance improvement for different region sizes and window sizes.](image-url)
GF-Complete library now available

- Big open-source GF arithmetic library in C
  - SIMD instructions
  - Logarithm tables
  - Standard tables
  - Lazy tables
  - Split tables
  - Composite fields
  - Anvin’s “by-two” method
  - Cauchy’s XOR conversion
  - Bit grouping
  - Euclid’s inverse determination
  - $w = 4, 8, 16, 32, 64, 128$

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

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!
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
Conclusions

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

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

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