Fast Erasure Coding for Data Storage: A Comprehensive Study of the Acceleration Techniques

Tianli Zhou & Chao Tian
Texas A&M University
Contents

• Motivation
• Background and Review

• Evaluating Individual Techniques
• Find the Best Strategy under Optimized Bitmatrix
• Proposed Coding Procedure and Evaluation

• Conclusion
Contents

• Motivation
• Background and Review

• Evaluating Individual Techniques
• Find the Best Strategy under Optimized Bitmatrix
• Proposed Coding Procedure and Evaluation

• Conclusion
Data Reliability
Data Reliability

• Disaster happens
Data Reliability

• Disaster happens

• Disk failure
• Server malfunction
• and more....

data
Replication - Erasure Code

Computational Overhead vs. Replication

Erasure Code vs. Space Overhead
Replication - Erasure Code

- Minimum space overhead
- Much more computational load
- Erasure code relies on Galois Field (finite field) ops
Replication - Erasure Code

- Minimum space overhead
- Much more computational load
- Erasure code relies on Galois Field (finite field) ops

How to reduce it

![Graph showing the relationship between Replication, Erasure Code, Computational Overhead, and Space Overhead.](image)
Acceleration Techniques

- Covert to XOR operations
- Specially Designed Code
- Optimize Coding Matrix
- Schedule Computing Order
- Apply SIMD
- Optimize for CPU Cache
Acceleration Techniques

- **Fast GF Coding**
  - Covert to XOR operations
  - Optimize Coding Matrix
  - Schedule Computing Order

- **Coding Theory**
  - Specially Designed Code

- **Computation Resource**
  - Apply SIMD
  - Optimize for CPU Cache
Acceleration Techniques

Fast GF Coding
- Covert to XOR operations
- Optimize Coding Matrix
- Schedule Computing Order

Coding Theory
- Specially Designed Code

Computation Resource
- Apply SIMD
- Optimize for CPU Cache
Question to Answer

• Most effective technique(s)?
• Utilize together?
• Components to be optimized?

• Problem is still important
  – Reduce energy consumption
  – Virtualized environment
  – Mobile / embedded system
  – Future I/O technology
Contents

• Motivation
• Background and Review

• Evaluating Individual Techniques
• Find the Best Strategy under Optimized Bitmatrix
• Proposed Coding Procedure and Evaluation

• Conclusion
Erasure Code

Store coded data to multiple nodes
Erasure Code

Recover even nodes failures happen
Erasure Code

$k=3$
segments

Encode

$n=5$
coded nodes

storage nodes
Erasure Code

$k=3$ segments

Encode

-storage nodes

$n=5$ coded nodes
Erasure Code

$k=3$
segments

MDS code: recover data from any $k$ surviving nodes
Erasure Code

$k=3$ segments

MDS code: recover data from any $k$ surviving nodes
Erasure Code

$k=3$ segments

$k=3$ systematic nodes
$m=2$ parity nodes
Erasure Code

$k=3$ segments

$k=3$ systematic nodes
$m=2$ parity nodes

storage nodes
Encoding

- Matrix multiplication on finite field
Encoding

- Matrix multiplication on finite field

\[
\begin{pmatrix}
1 & 0 & 0 \\
0 & 1 & 0 \\
0 & 0 & 1
\end{pmatrix}
\times
\begin{pmatrix}
p_{0,0} & p_{0,1} \\
p_{1,0} & p_{1,1} \\
p_{2,0} & p_{2,1}
\end{pmatrix}
= \begin{pmatrix}
p & Q & R
\end{pmatrix}
\]

All elements and ops in $GF(2^w)$

Galois Field (GF)
Finite field
Decoding
Decoding

\[
\begin{bmatrix}
1 & 0 & 0 \\
0 & 1 & 0 \\
0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
\begin{bmatrix}
p_{0,0} & p_{0,1} \\
\end{bmatrix} \\
\begin{bmatrix}
p_{1,0} & p_{1,1} \\
\end{bmatrix} \\
\begin{bmatrix}
p_{2,0} & p_{2,1} \\
\end{bmatrix}
\end{bmatrix}
\]
Decoding

- Submatrix corresponding to surviving data
Decoding

- Submatrix corresponding to surviving data
- Invert

\[
\begin{bmatrix}
1 & 0 & 0 \\
0 & 1 & 0 \\
0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
\mathbf{I}
\end{bmatrix}
\begin{bmatrix}
p_{0,0} & p_{0,1} \\
p_{1,0} & p_{1,1} \\
p_{2,0} & p_{2,1}
\end{bmatrix}
\begin{bmatrix}
p_{0,0} & p_{0,1} \\
p_{1,0} & p_{1,1} \\
p_{2,0} & p_{2,1}
\end{bmatrix}^{-1}
\]
Decoding

- Submatrix corresponding to surviving data
- Invert
- Multiplication

\[
\begin{bmatrix}
1 & 0 & 0 \\
0 & 1 & 0 \\
0 & 0 & 1
\end{bmatrix} \times
\begin{bmatrix}
p_{0,0} & p_{0,1} \\
p_{1,0} & p_{1,1} \\
p_{2,0} & p_{2,1}
\end{bmatrix}^{-1} =
\begin{bmatrix}
0 & p_{0,0} & p_{0,1} \\
0 & p_{1,0} & p_{1,1} \\
1 & p_{2,0} & p_{2,1}
\end{bmatrix}
\]
Cauchy Reed-Solomon Codes

- Direct assign $P$ to Cauchy matrix
Cauchy Reed-Solomon Codes

\[ X = (x_1, \ldots, x_k) \quad Y = (y_1, \ldots, y_m) \]

\[ C = \begin{bmatrix}
\frac{1}{x_1 + y_1} & \frac{1}{x_1 + y_2} & \cdots & \frac{1}{x_1 + y_m} \\
\frac{1}{x_2 + y_1} & \frac{1}{x_2 + y_2} & \cdots & \frac{1}{x_2 + y_m} \\
\vdots & \vdots & \ddots & \vdots \\
\frac{1}{x_k + y_1} & \frac{1}{x_k + y_2} & \cdots & \frac{1}{x_k + y_m}
\end{bmatrix} \]
Cauchy Reed-Solomon Codes

\[ X = (x_1, \ldots, x_k) \quad Y = (y_1, \ldots, y_m) \]

\[
C = \begin{bmatrix}
\frac{1}{x_1 + y_1} & \frac{1}{x_1 + y_2} & \cdots & \frac{1}{x_1 + y_m} \\
\frac{1}{x_2 + y_1} & \frac{1}{x_2 + y_2} & \cdots & \frac{1}{x_2 + y_m} \\
\vdots & \vdots & \ddots & \vdots \\
\frac{1}{x_k + y_1} & \frac{1}{x_k + y_2} & \cdots & \frac{1}{x_k + y_m}
\end{bmatrix}
\]

All elements and ops in \( GF(2^w) \)
Cauchy Reed-Solomon Codes

• Direct assign $P$ to Cauchy matrix

\[
\begin{bmatrix}
1 & 0 & \cdots & 0 \\
0 & 1 & \cdots & 0 \\
\vdots & \vdots & \ddots & \vdots \\
0 & 0 & \cdots & 1
\end{bmatrix}
\begin{bmatrix}
1 \\
\frac{1}{x_1 + y_1} \\
\frac{1}{x_2 + y_1} \\
\vdots \\
\frac{1}{x_k + y_1}
\end{bmatrix}
\begin{bmatrix}
1 \\
\frac{1}{x_1 + y_2} \\
\frac{1}{x_2 + y_2} \\
\vdots \\
\frac{1}{x_k + y_2}
\end{bmatrix}
\begin{bmatrix}
1 \\
\frac{1}{x_1 + y_m} \\
\frac{1}{x_2 + y_m} \\
\vdots \\
\frac{1}{x_k + y_m}
\end{bmatrix}
\]
Fast GF Ops – Binary Representation

• Binary representation of GF elements

All coefficients and messages are in \( GF(2^w) \)

Cauchy Parity Coding Matrix over \( GF(2^3) \)

\[ \begin{array}{c|c}
1 & 5 \\
5 & 1 \\
2 & 3 \\
7 & 4 \\
4 & 7 \\
\end{array} \]

\[ u_0 \quad u_1 \quad u_2 \quad u_3 \quad u_4 \]

\[ p_0 \quad p_1 \]
Fast GF Ops – Binary Representation

• Message bit-vector representation

Cauchy Parity Coding Matrix over GF(2^3)

Data Symbols

Binary Parity Coding Matrix

Parity Symbols
Fast GF Ops – Binary Representation

• Message bit-vector representation

Cauchy Parity Coding Matrix over GF($2^3$)

Data Symbols

Binary Parity Coding Matrix

Parity Symbols

\[ \begin{array}{c}
1 \\
5 \\
5 \\
2 \\
7 \\
4 \\
4 \\
7 \\
\end{array} \]

\[ P \]

\[ \begin{array}{c}
u_0 \\
u_1 \\
u_2 \\
u_3 \\
u_4 \\
\end{array} \times \]

\[ \begin{array}{c}
p_0 \\
p_1 \\
\end{array} \]
Fast GF Ops – Binary Representation

• Message bit-vector representation

Cauchy Parity Coding Matrix over GF(2^3)

1 by w bit-vector
Fast GF Ops – Binary Representation

- Bitmatrix representation

Cauchy Parity Coding Matrix over GF($2^3$)
Fast GF Ops – Binary Representation

• Bitmatrix representation
Fast GF Ops – Binary Representation

• Bitmatrix representation

Cauchy Parity Coding Matrix over GF(2^3)

\[ P \]

\[ \begin{array}{cccc}
1 & 5 & \\
5 & 1 & 3 \\
2 & 3 & 7 \\
4 & 7 & 4 \\
\end{array} \]

\[ w \text{ by } w \text{ bitmatrix} \]

\[ u_0 \quad u_1 \quad u_2 \quad u_3 \quad u_4 \]

Data Symbols

Binary Parity Coding Matrix

Parity Symbols

\[ p_0 \quad p_1 \]
Fast GF Ops – Binary Representation

• Matrix multiplication on GF -> XORs

Cauchy Parity Coding Matrix over GF(2^3)

Data Symols × Binary Parity Coding Matrix = Parity Symols

p_0 p_1
Fast GF Ops – Binary Representation

• Matrix multiplication on GF -> XORs
Fast GF Ops – Binary Representation

- Matrix multiplication on GF -> XORs

Cauchy Parity Coding Matrix over GF(2^3)

\[
\begin{pmatrix}
1 & 5 \\
5 & 1 \\
2 & 3 \\
7 & 4 \\
4 & 7 \\
\end{pmatrix}
\]

\[
\begin{pmatrix}
? \\
? \\
? \\
? \\
? \\
\end{pmatrix}
\]

Data Symbols

Binary Parity Coding Matrix

Parity Symbols

\[
\begin{pmatrix}
\color{red}P \\
\end{pmatrix}
\]

\[
\begin{pmatrix}
p_0 \\
p_1 \\
\end{pmatrix}
\]
Fast GF Ops – Binary Representation

- Matrix multiplication on GF -> XORs

Cauchy Parity Coding Matrix over GF($2^3$)
Contents

• Motivation
• Background and Review

• Evaluating Individual Techniques
• Find the Best Strategy under Optimized Bitmatrix
• Proposed Coding Procedure and Evaluation

• Conclusion
Techniques Tiers

- **Bitmatrix Tier**
  - Normalization (BN)

- **Scheduling Tier**
  - Smart Scheduling (SS)
  - Matching (UM, WM)
  - Scheduling for Caching Optimization (S-CO)

- **Hardware Tier**
  - Vectorization (V-XOR)
Bitmatrix Normalization (BN)

- Normalize by each element, find less bit 1s

Bitmatrix Normalization (BN)

- Normalize by each element, find less bit 1s
Bitmatrix Normalization (BN)

- Normalize by each element, find less bit 1s

Bitmatrix Normalization (BN)

• Normalize by each element, find less bit 1s
Bitmatrix Normalization (BN)

- Normalize by each element, find less bit 1s

Bitmatrix Normalization (BN)

- Normalize by each element, find less bit 1s
Bitmatrix Normalization (BN)

- Normalize by each element, find less bit 1s

Bitmatrix Normalization (BN)

- Normalize by each element, find less bit 1s

Bitmatrix Normalization (BN)

- Normalize by each element, find less bit 1s

---

Smart Scheduling (SS)

• Utilize both message and computed parities

Smart Scheduling (SS)

- Utilize both message and computed parities

\[ p_0 = u_0 \oplus u_2 \oplus u_3 \oplus u_4 \]
\[ p_1 = u_0 \oplus u_2 \oplus u_3 \oplus u_5 \]

3 XORs

3 XORs

---

Smart Scheduling (SS)

- Utilize both message and computed parities

\[ p_0 = u_0 \oplus u_2 \oplus u_3 \oplus u_4 \]

3 XORs

3 XORs

---

Bitmatrix Tier

- Normalization (BN)

Scheduling Tier

- Smart Scheduling (SS)
- Matching (UM, WM)
- Scheduling for Caching Optimization (S-CO)

Hardware Tier

- Vectorization (V-XOR)

---

Smart Scheduling (SS)

- Utilize both message and computed parities

\[ p_0 = u_0 \oplus u_2 \oplus u_3 \oplus u_4 \]

\[ p_1 = p_0 \oplus u_4 \oplus u_5 \]

Smart Scheduling (SS)

- Utilize both message and computed parities

\[ p_0 = u_0 \oplus u_2 \oplus u_3 \oplus u_4 \]

\[ p_1 = p_0 \oplus u_4 \oplus u_5 \]

Smart Scheduling (SS)

- Utilize both message and computed parities

\[
p_0 = u_0 \oplus u_2 \oplus u_3 \oplus u_4
\]

3 XORs

\[
p_1 = p_0 \oplus u_4 \oplus u_5
\]

3 XORs

---

Smart Scheduling (SS)

- Utilize both message and computed parities

\[ p_0 = u_0 \oplus u_2 \oplus u_3 \oplus u_4 \]

\[ p_1 = p_0 \oplus u_4 \oplus u_5 \]

Matching (UM, WM)

- Common XORs of pair message bits
- Unweighted/weighted greedy algorithm

Matching (UM, WM)

- Common XORs of pair message bits
- Unweighted/weighted greedy algorithm

Matching (UM, WM)

- Common XORs of pair message bits
- Unweighted/weighted greedy algorithm

---

Matching (UM, WM)

- Common XORs of pair message bits
- Unweighted/weighted greedy algorithm

Scheduling - Cache Optimization (S-CO)

• Reduce cache miss penalty

\[ p_0 = u_0 \oplus u_2 \oplus u_3 \oplus u_4 \]
\[ p_1 = u_0 \oplus u_2 \oplus u_3 \oplus u_5 \]

Scheduling - Cache Optimization (S-CO)

- Reduce cache miss penalty

\[ p_0 = u_0 \oplus u_2 \oplus u_3 \oplus u_4 \]
\[ p_1 = u_0 \oplus u_2 \oplus u_3 \oplus u_5 \]

Scheduling - Cache Optimization (S-CO)

- Reduce cache miss penalty

\[ p_0 = u_0 \oplus u_2 \oplus u_3 \oplus u_4 \]
\[ p_1 = u_0 \oplus u_2 \oplus u_3 \oplus u_5 \]

Scheduling - Cache Optimization (S-CO)

- Reduce cache miss penalty

\[ p_0 = u_0 \oplus u_2 \oplus u_3 \oplus u_4 \]
\[ p_1 = u_0 \oplus u_2 \oplus u_3 \oplus u_5 \]
Vectorization

- Using SIMD ISA perform multiple bits operation
  - SSE: 128 bits
  - AVX2: 256 bits
  - AVX512: 512 bits

<table>
<thead>
<tr>
<th>Bitmatrix Tier</th>
<th>Normalization (BN)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scheduling Tier</td>
<td>Smart Scheduling (SS)</td>
</tr>
<tr>
<td>Hardware Tier</td>
<td>Scheduling for Caching Optimization (S-CO)</td>
</tr>
</tbody>
</table>
Vectorization

• Using SIMD ISA perform multiple bits operation
  – SSE: 128 bits
  – AVX2: 256 bits
  – AVX512: 512 bits
Vectorization

- Using SIMD ISA perform multiple bits operation
  - SSE: 128 bits
  - AVX2: 256 bits
  - AVX512: 512 bits
Vectorization

- Using SIMD ISA perform multiple bits operation
  - SSE: 128 bits
  - AVX2: 256 bits
  - AVX512: 512 bits

V-XOR

V-GF

Jerasure 2.0
Vectorization

- Using SIMD ISA perform multiple bits operation
  - SSE: 128 bits
  - AVX2: 256 bits
  - AVX512: 512 bits

<table>
<thead>
<tr>
<th>Bitmatrix Tier</th>
<th>Normalization (BN)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scheduling Tier</td>
<td>Smart Scheduling (SS)</td>
</tr>
<tr>
<td>Hardware Tier</td>
<td>Vectorization (V-XOR)</td>
</tr>
</tbody>
</table>

V-XOR

\[
\begin{array}{c}
0 \\
1 \\
1
\end{array} \quad \begin{array}{c}
0 \\
1 \\
1
\end{array} \quad \begin{array}{c}
0 \\
1 \\
1
\end{array}
\]

V-GF

\[
\begin{array}{c}
5 \\
3 \\
6
\end{array} \quad \begin{array}{c}
5 \\
2 \\
3
\end{array} \quad \begin{array}{c}
5 \\
2 \\
3
\end{array}
\]

V.S.

\[
\begin{array}{c}
0 \\
1 \\
1
\end{array} \quad \begin{array}{c}
0 \\
1 \\
1
\end{array} \quad \begin{array}{c}
0 \\
1 \\
1
\end{array}
\]

Jerasure 2.0

\[
\begin{array}{c}
5 \\
2 \\
4
\end{array} \quad \begin{array}{c}
5 \\
1 \\
2
\end{array} \quad \begin{array}{c}
5 \\
1 \\
2
\end{array}
\]

\[
\begin{array}{c}
6 \\
4 \\
5
\end{array} \quad \begin{array}{c}
6 \\
4 \\
5
\end{array} \quad \begin{array}{c}
6 \\
4 \\
5
\end{array}
\]
Vectorization

- Using SIMD ISA perform multiple bits operation
  - SSE: 128 bits
  - AVX2: 256 bits
  - AVX512: 512 bits

V-XOR

V-GF

V.S.

Jerasure 2.0
Scheduling - Cache Optimization (S-CO)

- Reduce cache miss penalty
  \[ p_0 = u_0 \oplus u_2 \oplus u_3 \oplus u_4 \]
  \[ p_1 = u_0 \oplus u_2 \oplus u_3 \oplus u_5 \]

<table>
<thead>
<tr>
<th>Bitmatrix Tier</th>
<th>Normalization (BN)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scheduling Tier</td>
<td>Smart Scheduling (SS)</td>
</tr>
<tr>
<td>Hardware Tier</td>
<td>Scheduling for Caching Optimization (S-CO)</td>
</tr>
</tbody>
</table>

Cache

Memory

Scheduling - Cache Optimization (S-CO)

- Reduce cache miss penalty

\[ p_0 = u_0 \oplus u_2 \oplus u_3 \oplus u_4 \]
\[ p_1 = u_0 \oplus u_2 \oplus u_3 \oplus u_5 \]
\[ u_0 \rightarrow p_0, p_1, p_2... \]
\[ u_1 \rightarrow p_2, p_3, p_5... \]
\[ u_2 \rightarrow p_0, p_1, p_4... \]
\[ u_3 \rightarrow p_2, p_6, p_8... \]
Question to Answer

• Most effective technique(s)?
• Utilize together?
• Components to be optimized?
# Individual Techniques

Table 1: Performance improvements by individual techniques

<table>
<thead>
<tr>
<th>(n, k, w)</th>
<th>The number of XOR’s in the baseline code</th>
<th>Reduction in the number of XOR’s</th>
<th>Throughput increase</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>BN</td>
<td>SS</td>
</tr>
<tr>
<td>(8,6,4)</td>
<td>104</td>
<td>42.31%</td>
<td>17.31%</td>
</tr>
<tr>
<td>(8,6,8)</td>
<td>362</td>
<td>53.31%</td>
<td>33.70%</td>
</tr>
<tr>
<td>(9,6,4)</td>
<td>152</td>
<td>32.89%</td>
<td>19.74%</td>
</tr>
<tr>
<td>(9,6,8)</td>
<td>549</td>
<td>44.63%</td>
<td>29.14%</td>
</tr>
<tr>
<td>(10,6,4)</td>
<td>200</td>
<td>27.50%</td>
<td>22.00%</td>
</tr>
<tr>
<td>(10,6,8)</td>
<td>736</td>
<td>40.90%</td>
<td>28.80%</td>
</tr>
<tr>
<td>(12,8,4)</td>
<td>256</td>
<td>23.44%</td>
<td>23.44%</td>
</tr>
<tr>
<td>(12,8,8)</td>
<td>1028</td>
<td>36.38%</td>
<td>24.81%</td>
</tr>
<tr>
<td>(16,10,4)</td>
<td>496</td>
<td>18.95%</td>
<td>21.77%</td>
</tr>
<tr>
<td>(16,10,8)</td>
<td>1920</td>
<td>30.16%</td>
<td>21.98%</td>
</tr>
<tr>
<td>Average</td>
<td></td>
<td>35.05%</td>
<td>24.27%</td>
</tr>
</tbody>
</table>
## Individual Techniques

Table 1: Performance improvements by individual techniques

<table>
<thead>
<tr>
<th>( (n,k,w) )</th>
<th>The number of XOR’s in the baseline code</th>
<th>Reduction in the number of XOR’s</th>
<th>Throughput increase</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>( (n,k,w) )</td>
<td>BN</td>
<td>SS</td>
</tr>
<tr>
<td>(8,6,4)</td>
<td>104</td>
<td>42.31%</td>
<td>17.31%</td>
</tr>
<tr>
<td>(8,6,8)</td>
<td>362</td>
<td>53.31%</td>
<td>33.70%</td>
</tr>
<tr>
<td>(9,6,4)</td>
<td>152</td>
<td>32.89%</td>
<td>19.74%</td>
</tr>
<tr>
<td>(9,6,8)</td>
<td>549</td>
<td>44.63%</td>
<td>29.14%</td>
</tr>
<tr>
<td>(10,6,4)</td>
<td>200</td>
<td>27.50%</td>
<td>22.00%</td>
</tr>
<tr>
<td>(10,6,8)</td>
<td>736</td>
<td>40.90%</td>
<td>28.80%</td>
</tr>
<tr>
<td>(12,8,4)</td>
<td>256</td>
<td>23.44%</td>
<td>23.44%</td>
</tr>
<tr>
<td>(12,8,8)</td>
<td>1028</td>
<td>36.38%</td>
<td>24.81%</td>
</tr>
<tr>
<td>(16,10,4)</td>
<td>496</td>
<td>18.95%</td>
<td>21.77%</td>
</tr>
<tr>
<td>(16,10,8)</td>
<td>1920</td>
<td>30.16%</td>
<td>21.98%</td>
</tr>
<tr>
<td>Average over all tested cases</td>
<td></td>
<td>35.05%</td>
<td>24.27%</td>
</tr>
</tbody>
</table>
## Individual Techniques

Table 1: Performance improvements by individual techniques

<table>
<thead>
<tr>
<th>$(n,k,w)$</th>
<th>The number of XOR’s in the baseline code</th>
<th>Reduction in the number of XOR’s</th>
<th>Throughput increase</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>BN</td>
<td>SS</td>
</tr>
<tr>
<td>(8,6,4)</td>
<td>104</td>
<td>42.31%</td>
<td>17.31%</td>
</tr>
<tr>
<td>(8,6,8)</td>
<td>362</td>
<td>53.31%</td>
<td>33.70%</td>
</tr>
<tr>
<td>(9,6,4)</td>
<td>152</td>
<td>32.89%</td>
<td>19.74%</td>
</tr>
<tr>
<td>(9,6,8)</td>
<td>549</td>
<td>44.63%</td>
<td>29.14%</td>
</tr>
<tr>
<td>(10,6,4)</td>
<td>200</td>
<td>27.50%</td>
<td>22.00%</td>
</tr>
<tr>
<td>(10,6,8)</td>
<td>736</td>
<td>40.90%</td>
<td>28.80%</td>
</tr>
<tr>
<td>(12,8,4)</td>
<td>256</td>
<td>23.44%</td>
<td>23.44%</td>
</tr>
<tr>
<td>(12,8,8)</td>
<td>1028</td>
<td>36.38%</td>
<td>24.81%</td>
</tr>
<tr>
<td>(16,10,4)</td>
<td>496</td>
<td>18.95%</td>
<td>21.77%</td>
</tr>
<tr>
<td>(16,10,8)</td>
<td>1920</td>
<td>30.16%</td>
<td>21.98%</td>
</tr>
<tr>
<td>Average over all tested cases</td>
<td>35.05%</td>
<td>24.27%</td>
<td>36.63%</td>
</tr>
</tbody>
</table>

- Bitmatrix Normalization
- Unweighted Matching
- Weighted Matching
- XOR-based Vectorization
# Individual Techniques

Table 1: Performance improvements by individual techniques

<table>
<thead>
<tr>
<th>$(n, k, w)$</th>
<th>The number of XOR’s in the baseline code</th>
<th>Reduction in the number of XOR’s</th>
<th>Throughput increase</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>BN</td>
<td>SS</td>
</tr>
<tr>
<td>(8,6,4)</td>
<td>104</td>
<td>42.31%</td>
<td>17.31%</td>
</tr>
<tr>
<td>(8,6,8)</td>
<td>362</td>
<td>53.31%</td>
<td>33.70%</td>
</tr>
<tr>
<td>(9,6,4)</td>
<td>152</td>
<td>32.89%</td>
<td>19.74%</td>
</tr>
<tr>
<td>(9,6,8)</td>
<td>549</td>
<td>44.63%</td>
<td>29.14%</td>
</tr>
<tr>
<td>(10,6,4)</td>
<td>200</td>
<td>27.50%</td>
<td>22.00%</td>
</tr>
<tr>
<td>(10,6,8)</td>
<td>736</td>
<td>40.90%</td>
<td>28.80%</td>
</tr>
<tr>
<td>(12,8,4)</td>
<td>256</td>
<td>23.44%</td>
<td>23.44%</td>
</tr>
<tr>
<td>(12,8,8)</td>
<td>1028</td>
<td>36.38%</td>
<td>24.81%</td>
</tr>
<tr>
<td>(16,10,4)</td>
<td>496</td>
<td>18.95%</td>
<td>21.77%</td>
</tr>
<tr>
<td>(16,10,8)</td>
<td>1920</td>
<td>30.16%</td>
<td>21.98%</td>
</tr>
<tr>
<td><strong>Average over all tested cases</strong></td>
<td></td>
<td>35.05%</td>
<td>24.27%</td>
</tr>
</tbody>
</table>
# Individual Techniques

Table 1: Performance improvements by individual techniques

<table>
<thead>
<tr>
<th>$(n, k, w)$</th>
<th>The number of XOR’s in the baseline code</th>
<th>Reduction in the number of XOR’s</th>
<th>Throughput increase</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>BN</td>
<td>SS</td>
</tr>
<tr>
<td>(8,6,4)</td>
<td>104</td>
<td>42.31%</td>
<td>17.31%</td>
</tr>
<tr>
<td>(8,6,8)</td>
<td>362</td>
<td>53.31%</td>
<td>33.70%</td>
</tr>
<tr>
<td>(9,6,4)</td>
<td>152</td>
<td>32.89%</td>
<td>19.74%</td>
</tr>
<tr>
<td>(9,6,8)</td>
<td>549</td>
<td>44.63%</td>
<td>29.14%</td>
</tr>
<tr>
<td>(10,6,4)</td>
<td>200</td>
<td>27.50%</td>
<td>22.00%</td>
</tr>
<tr>
<td>(10,6,8)</td>
<td>736</td>
<td>40.90%</td>
<td>28.80%</td>
</tr>
<tr>
<td>(12,8,4)</td>
<td>256</td>
<td>23.44%</td>
<td>23.44%</td>
</tr>
<tr>
<td>(12,8,8)</td>
<td>1028</td>
<td>36.38%</td>
<td>24.81%</td>
</tr>
<tr>
<td>(16,10,4)</td>
<td>496</td>
<td>18.95%</td>
<td>21.77%</td>
</tr>
<tr>
<td>(16,10,8)</td>
<td>1920</td>
<td>30.16%</td>
<td>21.98%</td>
</tr>
<tr>
<td>Average over all tested cases</td>
<td></td>
<td>35.05%</td>
<td>24.27%</td>
</tr>
</tbody>
</table>

- Bitmatrix Normalization
- Smart Scheduling
- Unweighted Matching
- Weighted Matching
- Cache Optimization
- XOR-based Vectorization
Optimization Tiers

- Bitmatrix Tier
  - Normalization (BN)
  - Smart Scheduling (SS)
  - Matching (UM, WM)

- Scheduling Tier
  - Scheduling for Caching Optimization (S-CO)

- Hardware Tier
  - Vectorization (V-XOR)
Optimization Tiers

Bitmatrix Tier
- Normalization (BN)

Scheduling Tier
- Smart Scheduling (SS)
- Matching (UM, WM)
- Scheduling for Caching Optimization (S-CO)

Hardware Tier
- Vectorization (V-XOR)
Optimization Tiers

- **Bitmatrix Tier**
  - Normalization (BN)

- **Scheduling Tier**
  - Smart Scheduling (SS)
  - Matching (UM, WM)
  - Scheduling for Caching Optimization (S-CO)

- **Hardware Tier**
  - Vectorization (V-XOR)

- Generate Coding Schedule
- Run Coding Schedule
Optimization Tiers

Bitmatrix Tier

Normalization (BN)

Smart Scheduling (SS)

Matching (UM, WM)

Scheduling Tier

Generate Coding Schedule
Optimization Tiers

- Bitmatrix Tier
  - Normalization (BN)
  - By-pass

- Scheduling Tier
  - Smart Scheduling (SS)
  - Unweighted Matching (UM)
  - Weighted Matching (WM)
  - By-pass

Generate Coding Schedule
Question to Answer

• Most effective technique(s)?
• Utilize together?
• Components to be optimized?
Combinations (i,j)-strategy

Bitmatrix Tier
- By-pass
- Normalization (BN)

Scheduling Tier
- By-pass
- Smart Scheduling (SS)
- Unweighted Matching (UM)
- Weighted Matching (WM)
Combinations (i,j)-strategy

<table>
<thead>
<tr>
<th>Bitmatrix Tier</th>
<th>Scheduling Tier</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>By-pass</td>
<td>By-pass</td>
</tr>
<tr>
<td>Normalization (BN)</td>
<td>Smart Scheduling (SS)</td>
</tr>
</tbody>
</table>

\[ i \in [0,1] \]

\[ j \in [0,3] \]
Combinations $(i,j)$-strategy

Example: strategy-(1,1)

\[ i \in [0,1] \]

\[ j \in [0,3] \]
Combinations (i,j)-strategy

Example: strategy-(1,1)

- Bitmatrix Tier
  - By-pass
  - Normalization (BN)

- Scheduling Tier
  - By-pass
  - Smart Scheduling (SS)
  - Unweighted Matching (UM)
  - Weighted Matching (WM)

\( i \in [0,1] \)
\( j \in [0,3] \)
Contents

• Motivation
• Background and Review

• Evaluating Individual Techniques
  • Find the Best Strategy under Optimized Bitmatrix
  • Proposed Coding Procedure and Evaluation

• Conclusion
Question to Answer

• Most effective technique(s)?
• Utilize together?
• Components to be optimized?
Choice of Cauchy Matrix

• Different $X, Y$ array yields different Cauchy Matrix
Choice of Cauchy Matrix

• Different $X, Y$ array yields different Cauchy Matrix
Choice of Cauchy Matrix

• Different $X,Y$ array yields different Cauchy Matrix

\[
\begin{array}{cccc}
173 & 142 & 150 & 157 \\
152 & 1 & 61 & 221 \\
157 & 244 & 93 & 173 \\
167 & 150 & 142 & 71 \\
71 & 93 & 244 & 167 \\
122 & 61 & 1 & 186 \\
142 & 173 & 167 & 244 \\
244 & 157 & 71 & 142 \\
\end{array}
\]

$X = [1, 1, 1, 6, 0]$  
$Y = [9, 1, 0, 8, 4, 5, 7, 3, 2]$  
# of 1s = 913
Choice of Cauchy Matrix

- Different $X, Y$ array yields different Cauchy Matrix
Choice of Cauchy Matrix

- Different Cauchy Matrix affect coding chain

X,Y array

Coding schedule, (memcpy, XOR)

By-pass

Normalization (BN)

Smart Scheduling (SS)

Unweighted Matching (UM)

Weighted Matching (WM)

47
Choice of Cauchy Matrix

• Different Cauchy Matrix affect coding chain
Bitmatrix Optimization

- Individual bitmatrix optimization for all (i,j)-strategies
- Simulated Annealing and Genetic Algorithm
- # of operations as cost
<table>
<thead>
<tr>
<th>(i, j)</th>
<th>(n, k, w) = (8, 6, 4)</th>
<th>(n, k, w) = (9, 6, 4)</th>
<th>(n, k, w) = (10, 6, 4)</th>
<th>(n, k, w) = (12, 8, 4)</th>
<th>(n, k, w) = (16, 10, 4)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0, 0)</td>
<td>112</td>
<td>77</td>
<td>77</td>
<td>216</td>
<td>173</td>
</tr>
<tr>
<td>(0, 1)</td>
<td>94</td>
<td>70</td>
<td>68</td>
<td>134</td>
<td>102</td>
</tr>
<tr>
<td>(0, 2)</td>
<td>90</td>
<td>72</td>
<td>68</td>
<td>127</td>
<td>103</td>
</tr>
<tr>
<td>(0, 3)</td>
<td>90</td>
<td>72</td>
<td>68</td>
<td>127</td>
<td>102</td>
</tr>
<tr>
<td>(1, 0)</td>
<td>68</td>
<td>58</td>
<td>58</td>
<td>114</td>
<td>99</td>
</tr>
<tr>
<td>(1, 1)</td>
<td>64</td>
<td>58</td>
<td>58</td>
<td>99</td>
<td>90</td>
</tr>
<tr>
<td>(1, 2)</td>
<td>64</td>
<td>58</td>
<td>57</td>
<td>98</td>
<td>87</td>
</tr>
<tr>
<td>(1, 3)</td>
<td>64</td>
<td>58</td>
<td>57</td>
<td>98</td>
<td>90</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>(i, j)</th>
<th>(n, k, w) = (8, 6, 8)</th>
<th>(n, k, w) = (9, 6, 8)</th>
<th>(n, k, w) = (10, 6, 8)</th>
<th>(n, k, w) = (12, 8, 8)</th>
<th>(n, k, w) = (16, 10, 8)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0, 0)</td>
<td>378</td>
<td>291</td>
<td>247</td>
<td>573</td>
<td>467</td>
</tr>
<tr>
<td>(0, 1)</td>
<td>256</td>
<td>234</td>
<td>225</td>
<td>413</td>
<td>393</td>
</tr>
<tr>
<td>(0, 2)</td>
<td>286</td>
<td>227</td>
<td>217</td>
<td>408</td>
<td>346</td>
</tr>
<tr>
<td>(0, 3)</td>
<td>286</td>
<td>216</td>
<td>197</td>
<td>408</td>
<td>357</td>
</tr>
<tr>
<td>(1, 0)</td>
<td>185</td>
<td>151</td>
<td>142</td>
<td>328</td>
<td>286</td>
</tr>
<tr>
<td>(1, 1)</td>
<td>164</td>
<td>155</td>
<td>133</td>
<td>285</td>
<td>270</td>
</tr>
<tr>
<td>(1, 2)</td>
<td>167</td>
<td>143</td>
<td>134</td>
<td>272</td>
<td>244</td>
</tr>
<tr>
<td>(1, 3)</td>
<td>167</td>
<td>144</td>
<td>130</td>
<td>273</td>
<td>249</td>
</tr>
<tr>
<td>(i, j)</td>
<td>(n, k, w) = (8, 6, 4)</td>
<td>(n, k, w) = (9, 6, 4)</td>
<td>(n, k, w) = (10, 6, 4)</td>
<td>(n, k, w) = (12, 8, 4)</td>
<td>(n, k, w) = (16, 10, 4)</td>
</tr>
<tr>
<td>--------</td>
<td>----------------------</td>
<td>----------------------</td>
<td>----------------------</td>
<td>----------------------</td>
<td>----------------------</td>
</tr>
<tr>
<td>(0, 0)</td>
<td>112 77 77</td>
<td>164 122 117</td>
<td>216 173 162</td>
<td>272 232 232</td>
<td>520 462 464</td>
</tr>
<tr>
<td>(0, 1)</td>
<td>94 70 68</td>
<td>134 102 100</td>
<td>172 137 134</td>
<td>212 190 188</td>
<td>412 368 368</td>
</tr>
<tr>
<td>(0, 2)</td>
<td>90 72 68</td>
<td>127 103 101</td>
<td>164 134 132</td>
<td>204 186 180</td>
<td>376 344 345</td>
</tr>
<tr>
<td>(0, 3)</td>
<td>90 72 68</td>
<td>127 102 101</td>
<td>164 132 132</td>
<td>204 184 182</td>
<td>376 343 345</td>
</tr>
<tr>
<td>(1, 0)</td>
<td>68 58 58</td>
<td>114 99 98</td>
<td>161 142 141</td>
<td>212 198 198</td>
<td>426 419 419</td>
</tr>
<tr>
<td>(1, 1)</td>
<td>64 58 58</td>
<td>99 90 89</td>
<td>138 120 120</td>
<td>189 174 171</td>
<td>365 352 352</td>
</tr>
<tr>
<td>(1, 2)</td>
<td>64 58 57</td>
<td>98 87 87</td>
<td>133 118 118</td>
<td>176 165 164</td>
<td>326 317 316</td>
</tr>
<tr>
<td>(1, 3)</td>
<td>64 58 57</td>
<td>98 90 87</td>
<td>132 118 118</td>
<td>175 164 164</td>
<td>326 316 316</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>(i, j)</th>
<th>(n, k, w) = (8, 6, 8)</th>
<th>(n, k, w) = (9, 6, 8)</th>
<th>(n, k, w) = (10, 6, 8)</th>
<th>(n, k, w) = (12, 8, 8)</th>
<th>(n, k, w) = (16, 10, 8)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0, 0)</td>
<td>378 291 247</td>
<td>573 467 425</td>
<td>768 611 582</td>
<td>1060 880 841</td>
<td>1968 1685 1681</td>
</tr>
<tr>
<td>(0, 1)</td>
<td>256 234 225</td>
<td>413 393 349</td>
<td>556 518 479</td>
<td>805 740 684</td>
<td>1546 1432 1374</td>
</tr>
<tr>
<td>(0, 2)</td>
<td>286 227 217</td>
<td>408 346 321</td>
<td>532 474 450</td>
<td>726 636 591</td>
<td>1304 1180 1170</td>
</tr>
<tr>
<td>(0, 3)</td>
<td>286 216 197</td>
<td>408 357 329</td>
<td>532 460 450</td>
<td>726 636 627</td>
<td>1304 1180 1170</td>
</tr>
<tr>
<td>(1, 0)</td>
<td>185 151 142</td>
<td>328 286 262</td>
<td>467 434 419</td>
<td>686 613 563</td>
<td>1389 1293 1246</td>
</tr>
<tr>
<td>(1, 1)</td>
<td>164 155 133</td>
<td>285 270 230</td>
<td>411 383 371</td>
<td>593 557 544</td>
<td>1264 1184 1133</td>
</tr>
<tr>
<td>(1, 2)</td>
<td>167 143 134</td>
<td>272 244 225</td>
<td>377 343 335</td>
<td>520 487 462</td>
<td>998 951 922</td>
</tr>
<tr>
<td>(1, 3)</td>
<td>167 144 130</td>
<td>273 249 233</td>
<td>377 337 337</td>
<td>520 473 467</td>
<td>995 941 932</td>
</tr>
</tbody>
</table>
### No Opt by SA vs GA

<table>
<thead>
<tr>
<th>$(i, j)$</th>
<th>$(n, k, w) = (8, 6, 4)$</th>
<th>$(n, k, w) = (9, 6, 4)$</th>
<th>$(n, k, w) = (10, 6, 4)$</th>
<th>$(n, k, w) = (12, 8, 4)$</th>
<th>$(n, k, w) = (16, 10, 4)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0,0)</td>
<td>112</td>
<td>77</td>
<td>164</td>
<td>122</td>
<td>117</td>
</tr>
<tr>
<td>(0,1)</td>
<td>94</td>
<td>70</td>
<td>134</td>
<td>102</td>
<td>100</td>
</tr>
<tr>
<td>(0,2)</td>
<td>90</td>
<td>72</td>
<td>127</td>
<td>103</td>
<td>101</td>
</tr>
<tr>
<td>(0,3)</td>
<td>90</td>
<td>72</td>
<td>127</td>
<td>102</td>
<td>101</td>
</tr>
<tr>
<td>(1,0)</td>
<td>68</td>
<td>58</td>
<td>114</td>
<td>99</td>
<td>98</td>
</tr>
<tr>
<td>(1,1)</td>
<td>64</td>
<td>58</td>
<td>99</td>
<td>90</td>
<td>89</td>
</tr>
<tr>
<td>(1,2)</td>
<td>64</td>
<td>58</td>
<td>98</td>
<td>87</td>
<td>87</td>
</tr>
<tr>
<td>(1,3)</td>
<td>64</td>
<td>58</td>
<td>98</td>
<td>90</td>
<td>87</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$(i, j)$</th>
<th>$(n, k, w) = (8, 6, 8)$</th>
<th>$(n, k, w) = (9, 6, 8)$</th>
<th>$(n, k, w) = (10, 6, 8)$</th>
<th>$(n, k, w) = (12, 8, 8)$</th>
<th>$(n, k, w) = (16, 10, 8)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0,0)</td>
<td>378</td>
<td>291</td>
<td>247</td>
<td>425</td>
<td>582</td>
</tr>
<tr>
<td>(0,1)</td>
<td>256</td>
<td>234</td>
<td>225</td>
<td>349</td>
<td>479</td>
</tr>
<tr>
<td>(0,2)</td>
<td>286</td>
<td>227</td>
<td>217</td>
<td>321</td>
<td>450</td>
</tr>
<tr>
<td>(0,3)</td>
<td>286</td>
<td>216</td>
<td>197</td>
<td>329</td>
<td>450</td>
</tr>
<tr>
<td>(1,0)</td>
<td>185</td>
<td>151</td>
<td>142</td>
<td>262</td>
<td>419</td>
</tr>
<tr>
<td>(1,1)</td>
<td>164</td>
<td>155</td>
<td>133</td>
<td>230</td>
<td>371</td>
</tr>
<tr>
<td>(1,2)</td>
<td>167</td>
<td>143</td>
<td>134</td>
<td>225</td>
<td>335</td>
</tr>
<tr>
<td>(1,3)</td>
<td>167</td>
<td>144</td>
<td>130</td>
<td>233</td>
<td>337</td>
</tr>
</tbody>
</table>
Strategies-\((i,j)\) with Optimized Bitmatrices

- Strategy-\((1,3)\) is the best strategy.
Optimized Bitmatrix Reduced Costs
Cost Function Improvement

- Memcpy is 1.5x faster than XOR
- Cost functions:
  - Total # of XORs in schedule
  - Total # of ops in schedule
  - Weighted sum of ops in schedule

Table 4: Encoding throughput (GB/s) using bitmatrices obtained by the genetic algorithm under different cost functions

<table>
<thead>
<tr>
<th>$(n,k,w)$</th>
<th>Cost Function</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td># of XOR</td>
<td># of XOR</td>
<td>Weighted</td>
<td></td>
</tr>
<tr>
<td></td>
<td>and copying</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(8,6,4)</td>
<td>4.64</td>
<td>4.66</td>
<td>4.68</td>
<td></td>
</tr>
<tr>
<td>(8,6,8)</td>
<td>4.30</td>
<td>4.35</td>
<td>4.32</td>
<td></td>
</tr>
<tr>
<td>(9,6,4)</td>
<td>3.72</td>
<td>3.73</td>
<td>3.80</td>
<td></td>
</tr>
<tr>
<td>(9,6,8)</td>
<td>3.28</td>
<td>3.28</td>
<td>3.44</td>
<td></td>
</tr>
<tr>
<td>(10,6,4)</td>
<td>2.33</td>
<td>2.51</td>
<td>2.52</td>
<td></td>
</tr>
<tr>
<td>(10,6,8)</td>
<td>1.99</td>
<td>1.97</td>
<td>2.11</td>
<td></td>
</tr>
<tr>
<td>(12,8,4)</td>
<td>2.96</td>
<td>3.11</td>
<td>3.16</td>
<td></td>
</tr>
<tr>
<td>(12,8,8)</td>
<td>2.54</td>
<td>2.56</td>
<td>2.58</td>
<td></td>
</tr>
<tr>
<td>(16,10,4)</td>
<td>2.29</td>
<td>2.29</td>
<td>2.32</td>
<td></td>
</tr>
<tr>
<td>(16,10,8)</td>
<td>1.71</td>
<td>1.72</td>
<td>1.74</td>
<td></td>
</tr>
</tbody>
</table>
Contents

• Motivation
• Background and Review

• Evaluating Individual Techniques
• Find the Best Strategy under Optimized Bitmatrix
  • Proposed Coding Procedure and Evaluation

• Conclusion
Proposed Coding Procedure

Pre-optimized Bitmatrix

Bitmatrix Tier
- Normalization (BN)
- Smart Scheduling (SS)
- Matching (UM, WM)
- Scheduling for Caching Optimization (S-CO)

Scheduling Tier

Hardware Tier
- Vectorization (V-XOR)
Proposed Coding Procedure

Coding Schedule

- Bitmatrix Tier
- Scheduling Tier
  - Scheduling for Caching Optimization (S-CO)
- Hardware Tier
  - Vectorization (V-XOR)
Testing Setup

• Ryzen 1700X @ 3.4Ghz (8C/16T)
• 16GB DDR4
• Ubuntu 18.04 64-bit, GCC 7.3.0

• Jerasure 1.2A/ Jerasure 2.0:
  – XOR-based Cauchy RS code (BN, SS applied)
  – GF-based RS code
  – Raid-6

• Other codes:
  – EVENODD
  – RDP
  – STAR
## Encoding v.s. Efficient RS/CRS code

Table 5: Encoding throughput (GB/s) for methods that allow general \((n, k)\) parameters and \(w = 8\)

<table>
<thead>
<tr>
<th>((n, k))</th>
<th>Proposed</th>
<th>Vectorized XOR-based CRS code</th>
<th>Vectorized GF-based RS code [16]</th>
</tr>
</thead>
<tbody>
<tr>
<td>(7,5)</td>
<td>4.64</td>
<td><strong>4.73</strong></td>
<td>2.52</td>
</tr>
<tr>
<td>(8,6)</td>
<td>5.21</td>
<td><strong>5.22</strong></td>
<td>2.70</td>
</tr>
<tr>
<td>(9,7)</td>
<td>5.32</td>
<td><strong>5.45</strong></td>
<td>2.74</td>
</tr>
<tr>
<td>(10,8)</td>
<td>5.36</td>
<td><strong>5.59</strong></td>
<td>2.77</td>
</tr>
<tr>
<td>(12,10)</td>
<td>5.72</td>
<td><strong>5.88</strong></td>
<td>2.81</td>
</tr>
<tr>
<td>(8,5)</td>
<td><strong>3.19</strong></td>
<td>2.75</td>
<td>1.76</td>
</tr>
<tr>
<td>(9,6)</td>
<td>3.49</td>
<td>2.84</td>
<td>1.77</td>
</tr>
<tr>
<td>(10,7)</td>
<td>3.67</td>
<td>2.79</td>
<td>1.80</td>
</tr>
<tr>
<td>(11,8)</td>
<td>3.72</td>
<td>2.92</td>
<td>1.82</td>
</tr>
<tr>
<td>(13,10)</td>
<td>3.82</td>
<td>3.10</td>
<td>1.84</td>
</tr>
<tr>
<td>(10,6)</td>
<td><strong>2.55</strong></td>
<td>2.15</td>
<td>1.31</td>
</tr>
<tr>
<td>(11,7)</td>
<td>2.75</td>
<td>2.17</td>
<td>1.32</td>
</tr>
<tr>
<td>(12,8)</td>
<td>2.86</td>
<td>2.20</td>
<td>1.35</td>
</tr>
<tr>
<td>(14,10)</td>
<td>2.86</td>
<td>2.19</td>
<td>1.40</td>
</tr>
<tr>
<td>(15,10)</td>
<td>2.30</td>
<td>1.79</td>
<td>1.11</td>
</tr>
<tr>
<td>(16,10)</td>
<td><strong>1.96</strong></td>
<td>1.48</td>
<td>0.92</td>
</tr>
</tbody>
</table>

Outperform vectorized XOR-based CRS code for \(m>2\)
### Encoding v.s. Efficient RS/CRS code

Table 5: Encoding throughput (GB/s) for methods that allow general $(n, k)$ parameters and $w = 8$

<table>
<thead>
<tr>
<th>$(n, k)$</th>
<th>Proposed</th>
<th>Vectorized XOR-based CRS code</th>
<th>Vectorized GF-based RS code [16]</th>
</tr>
</thead>
<tbody>
<tr>
<td>(7,5)</td>
<td>4.64</td>
<td>4.73</td>
<td>2.52</td>
</tr>
<tr>
<td>(8,6)</td>
<td>5.21</td>
<td>5.22</td>
<td>2.70</td>
</tr>
<tr>
<td>(9,7)</td>
<td>5.32</td>
<td>5.45</td>
<td>2.74</td>
</tr>
<tr>
<td>(10,8)</td>
<td>5.36</td>
<td>5.59</td>
<td>2.77</td>
</tr>
<tr>
<td>(12,10)</td>
<td>5.72</td>
<td>5.88</td>
<td>2.81</td>
</tr>
<tr>
<td>(8,5)</td>
<td>3.19</td>
<td>2.75</td>
<td>1.76</td>
</tr>
<tr>
<td>(9,6)</td>
<td>3.49</td>
<td>2.84</td>
<td>1.77</td>
</tr>
<tr>
<td>(10,7)</td>
<td>3.67</td>
<td>2.79</td>
<td>1.80</td>
</tr>
<tr>
<td>(11,8)</td>
<td>3.72</td>
<td>2.92</td>
<td>1.82</td>
</tr>
<tr>
<td>(13,10)</td>
<td>3.82</td>
<td>3.10</td>
<td>1.84</td>
</tr>
<tr>
<td>(10,6)</td>
<td>2.55</td>
<td>2.15</td>
<td>1.31</td>
</tr>
<tr>
<td>(11,7)</td>
<td>2.75</td>
<td>2.17</td>
<td>1.32</td>
</tr>
<tr>
<td>(12,8)</td>
<td>2.86</td>
<td>2.20</td>
<td>1.35</td>
</tr>
<tr>
<td>(14,10)</td>
<td>2.86</td>
<td>2.19</td>
<td>1.40</td>
</tr>
<tr>
<td>(15,10)</td>
<td>2.30</td>
<td>1.79</td>
<td>1.11</td>
</tr>
<tr>
<td>(16,10)</td>
<td>1.96</td>
<td>1.48</td>
<td>0.92</td>
</tr>
</tbody>
</table>

**Outperform vectorized XOR-based CRS code for $m>2$**

**Outperform vectorized GF-based RS code with big margin (~ 2x faster)**
# Encoding v.s. Three Parities Codes

Table 6: Encoding throughputs (GB/s): Three parities

<table>
<thead>
<tr>
<th>$(n, k, w)$</th>
<th>Proposed</th>
<th>Vectorized XOR-based CRS code</th>
<th>STAR code [8]</th>
<th>Quantcast QFS [13]</th>
</tr>
</thead>
<tbody>
<tr>
<td>(8,5,4)</td>
<td>3.59</td>
<td>3.25</td>
<td>2.97</td>
<td>2.92</td>
</tr>
<tr>
<td>(8,5,8)</td>
<td>3.19</td>
<td>2.75</td>
<td>2.97</td>
<td>2.92</td>
</tr>
<tr>
<td>(9,6,4)</td>
<td>3.52</td>
<td>3.72</td>
<td>3.42</td>
<td>3.04</td>
</tr>
<tr>
<td>(9,6,8)</td>
<td>3.49</td>
<td>2.84</td>
<td>3.42</td>
<td>3.04</td>
</tr>
<tr>
<td>(10,7,4)</td>
<td>4.15</td>
<td>3.86</td>
<td>3.76</td>
<td>3.25</td>
</tr>
<tr>
<td>(10,7,8)</td>
<td>3.67</td>
<td>2.79</td>
<td><strong>3.76</strong></td>
<td>3.25</td>
</tr>
<tr>
<td>(11,8,4)</td>
<td>4.36</td>
<td>4.13</td>
<td>3.94</td>
<td>3.27</td>
</tr>
<tr>
<td>(11,8,8)</td>
<td>3.72</td>
<td>2.92</td>
<td><strong>3.94</strong></td>
<td>3.27</td>
</tr>
<tr>
<td>(13,10,4)</td>
<td>4.51</td>
<td>4.08</td>
<td>4.37</td>
<td>3.41</td>
</tr>
<tr>
<td>(13,10,8)</td>
<td>3.82</td>
<td>3.10</td>
<td><strong>4.37</strong></td>
<td>3.41</td>
</tr>
</tbody>
</table>

Similar or better performance than specially designed codes
## Encoding v.s. Two Parities Codes

Table 7: Encoding throughputs (GB/s): Two parities

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>(7,5,4)</td>
<td>5.16</td>
<td>4.85</td>
<td>n/a</td>
<td>4.28</td>
<td>4.37</td>
</tr>
<tr>
<td>(7,5,8)</td>
<td>4.64</td>
<td><strong>4.73</strong></td>
<td>2.18</td>
<td>4.28</td>
<td>4.37</td>
</tr>
<tr>
<td>(8,6,4)</td>
<td>4.67</td>
<td><strong>5.22</strong></td>
<td>n/a</td>
<td>4.83</td>
<td>4.95</td>
</tr>
<tr>
<td>(8,6,8)</td>
<td>5.21</td>
<td><strong>5.22</strong></td>
<td>2.15</td>
<td>4.83</td>
<td>4.95</td>
</tr>
<tr>
<td>(9,7,4)</td>
<td><strong>5.77</strong></td>
<td>5.59</td>
<td>n/a</td>
<td>5.20</td>
<td>5.32</td>
</tr>
<tr>
<td>(9,7,8)</td>
<td>5.32</td>
<td><strong>5.45</strong></td>
<td>2.16</td>
<td>5.20</td>
<td>5.32</td>
</tr>
<tr>
<td>(10,8,4)</td>
<td><strong>5.90</strong></td>
<td>5.23</td>
<td>n/a</td>
<td>5.50</td>
<td>5.69</td>
</tr>
<tr>
<td>(10,8,8)</td>
<td>5.36</td>
<td>5.59</td>
<td>2.17</td>
<td>5.50</td>
<td><strong>5.69</strong></td>
</tr>
<tr>
<td>(12,10,4)</td>
<td>6.23</td>
<td>6.00</td>
<td>n/a</td>
<td>6.02</td>
<td><strong>6.24</strong></td>
</tr>
<tr>
<td>(12,10,8)</td>
<td>5.72</td>
<td>5.88</td>
<td>2.17</td>
<td>6.02</td>
<td><strong>6.24</strong></td>
</tr>
</tbody>
</table>

Similar or better performance than specially designed codes
# Overall Encoding Improvement

<table>
<thead>
<tr>
<th>Reference codes or methods</th>
<th>Improvement by proposed code</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>General ((n, k)) Codes</strong></td>
<td></td>
</tr>
<tr>
<td>GF-based RS code w/o vectorization</td>
<td>552.27%</td>
</tr>
<tr>
<td>XOR-based CRS code w/o vectorization</td>
<td>53.65%</td>
</tr>
<tr>
<td>Vectorized GF-based RS code [16]</td>
<td>99.82%</td>
</tr>
<tr>
<td>Vectorized XOR-based CRS code</td>
<td>14.98%</td>
</tr>
<tr>
<td><strong>Three Parities Codes</strong></td>
<td></td>
</tr>
<tr>
<td>STAR [8]</td>
<td>5.59%</td>
</tr>
<tr>
<td>Quancast-QFS [13]</td>
<td>21.68%</td>
</tr>
<tr>
<td><strong>Two Parities Codes</strong></td>
<td></td>
</tr>
<tr>
<td>Raid-6 w/o vectorization</td>
<td>206.88%</td>
</tr>
<tr>
<td>Vectorized Raid-6</td>
<td>142.07%</td>
</tr>
<tr>
<td>RDP [6]</td>
<td>5.85%</td>
</tr>
<tr>
<td>EVENODD [2]</td>
<td>8.79%</td>
</tr>
</tbody>
</table>
Decoding
Decoding

- Direct read in most cases
Decoding

- Direct read in most cases
- Degraded read (decoding) only if **systematic** nodes fail
Decoding

• Direct read in most cases
• Degraded read (decoding) only if systematic nodes fail
  – Single disk failure rate: 1%
  – In (10,6) erasure code storage system
    • 1 systematic node failure: ~ 0.055
    • 2 systematic node failure: ~ 0.0014
    • 3 systematic node failure: ~ $1.8 \times 10^{-5}$
    • 4 systematic node failure: ~ $1.4 \times 10^{-7}$
Decoding

• Direct read in most cases
• Degraded read (decoding) only if **systematic** nodes fail
  – Single disk failure rate: 1%
  – In (10,6) erasure code storage system
    • 1 systematic node failure: ~ 0.055
    • 2 systematic node failure: ~ 0.0014
    • 3 systematic node failure: ~ 1.8x10^{-5}
    • 4 systematic node failure: ~ 1.4x10^{-7}
• Decoding performance should be viewed as secondary importance
# Decoding Throughput

Table 9: Decoding throughput (GB/s) for methods that allow general \((n,k)\) parameters and \(w=8\)

<table>
<thead>
<tr>
<th>((n,k))</th>
<th>Proposed</th>
<th>Vectorized XOR-based CRS code</th>
<th>Vectorized GF-based RS code [16]</th>
</tr>
</thead>
<tbody>
<tr>
<td>(7,5)</td>
<td>3.87</td>
<td><strong>4.56</strong></td>
<td>2.58</td>
</tr>
<tr>
<td>(8,6)</td>
<td><strong>5.45</strong></td>
<td>4.86</td>
<td>2.67</td>
</tr>
<tr>
<td>(9,7)</td>
<td>4.46</td>
<td><strong>5.06</strong></td>
<td>2.70</td>
</tr>
<tr>
<td>(10,8)</td>
<td>4.89</td>
<td><strong>5.11</strong></td>
<td>2.75</td>
</tr>
<tr>
<td>(12,10)</td>
<td>4.45</td>
<td><strong>5.52</strong></td>
<td>2.79</td>
</tr>
<tr>
<td>(8,5)</td>
<td><strong>3.04</strong></td>
<td>2.11</td>
<td>1.71</td>
</tr>
<tr>
<td>(9,6)</td>
<td>2.94</td>
<td>2.20</td>
<td>1.74</td>
</tr>
<tr>
<td>(10,7)</td>
<td>3.28</td>
<td>2.29</td>
<td>1.76</td>
</tr>
<tr>
<td>(11,8)</td>
<td>3.08</td>
<td>2.31</td>
<td>1.71</td>
</tr>
<tr>
<td>(13,10)</td>
<td>3.21</td>
<td>2.37</td>
<td>1.88</td>
</tr>
<tr>
<td>(10,6)</td>
<td>2.38</td>
<td>1.80</td>
<td>1.31</td>
</tr>
<tr>
<td>(11,7)</td>
<td>2.35</td>
<td>1.85</td>
<td>1.32</td>
</tr>
<tr>
<td>(12,8)</td>
<td>2.54</td>
<td>1.87</td>
<td>1.33</td>
</tr>
<tr>
<td>(14,10)</td>
<td>2.47</td>
<td>1.89</td>
<td>1.38</td>
</tr>
<tr>
<td>(15,10)</td>
<td>2.00</td>
<td>1.48</td>
<td>1.09</td>
</tr>
<tr>
<td>(16,10)</td>
<td><strong>1.77</strong></td>
<td>1.30</td>
<td>0.91</td>
</tr>
</tbody>
</table>

Table 10: Decoding throughputs (GB/s): Three parities

<table>
<thead>
<tr>
<th>((n,k,w))</th>
<th>Proposed</th>
<th>Vectorized XOR-based CRS code</th>
<th>Vectorized STAR code [8]</th>
<th>Quantcast QFS [13]</th>
</tr>
</thead>
<tbody>
<tr>
<td>(8,5,4)</td>
<td>4.28</td>
<td>3.08</td>
<td>3.20</td>
<td>1.77</td>
</tr>
<tr>
<td>(8,5,8)</td>
<td>3.04</td>
<td>2.11</td>
<td>3.20</td>
<td>1.77</td>
</tr>
<tr>
<td>(9,6,4)</td>
<td>4.13</td>
<td>3.41</td>
<td>3.23</td>
<td>1.74</td>
</tr>
<tr>
<td>(9,6,8)</td>
<td>2.94</td>
<td>2.20</td>
<td>3.23</td>
<td>1.74</td>
</tr>
<tr>
<td>(10,7,4)</td>
<td>4.55</td>
<td>3.53</td>
<td>3.52</td>
<td>1.77</td>
</tr>
<tr>
<td>(10,7,8)</td>
<td>3.28</td>
<td>2.29</td>
<td>3.52</td>
<td>1.77</td>
</tr>
<tr>
<td>(11,8,4)</td>
<td>4.70</td>
<td>3.78</td>
<td>3.13</td>
<td>1.68</td>
</tr>
<tr>
<td>(11,8,8)</td>
<td>3.08</td>
<td>2.31</td>
<td>3.13</td>
<td>1.68</td>
</tr>
<tr>
<td>(13,10,4)</td>
<td>4.86</td>
<td>3.71</td>
<td>3.50</td>
<td>1.71</td>
</tr>
<tr>
<td>(13,10,8)</td>
<td>3.21</td>
<td>2.37</td>
<td>3.50</td>
<td>1.71</td>
</tr>
</tbody>
</table>

Table 11: Decoding throughputs (GB/s): Two parities

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>(7,5,4)</td>
<td>5.52</td>
<td>4.85</td>
<td>n/a</td>
<td>6.66</td>
</tr>
<tr>
<td>(7,5,8)</td>
<td>3.87</td>
<td>4.65</td>
<td>n/a</td>
<td>6.66</td>
</tr>
<tr>
<td>(8,6,4)</td>
<td>5.43</td>
<td>5.14</td>
<td>n/a</td>
<td>7.42</td>
</tr>
<tr>
<td>(8,6,8)</td>
<td>5.45</td>
<td>4.86</td>
<td>2.67</td>
<td>7.42</td>
</tr>
<tr>
<td>(9,7,4)</td>
<td>6.03</td>
<td>5.37</td>
<td>n/a</td>
<td>7.65</td>
</tr>
<tr>
<td>(9,7,8)</td>
<td>4.46</td>
<td>5.06</td>
<td>n/a</td>
<td>7.65</td>
</tr>
<tr>
<td>(10,8,4)</td>
<td>5.88</td>
<td>5.73</td>
<td>n/a</td>
<td>7.93</td>
</tr>
<tr>
<td>(10,8,8)</td>
<td>4.89</td>
<td>5.11</td>
<td>2.77</td>
<td>7.93</td>
</tr>
<tr>
<td>(12,10,4)</td>
<td>6.23</td>
<td>5.89</td>
<td>n/a</td>
<td>7.49</td>
</tr>
<tr>
<td>(12,10,8)</td>
<td>4.45</td>
<td>5.52</td>
<td>2.81</td>
<td>7.49</td>
</tr>
</tbody>
</table>
Contents

• Motivation
• Background and Review

• Individual Techniques and Combinations
• Bitmatrix Optimization
• Proposed Coding Procedure and Evaluation

• Conclusion
Conclusion

• Comprehensive study of acceleration techniques
• Combine existing techniques and jointly optimize the bitmatrix
• Proposed approach outperforms most existing approaches in encoding throughput.
Conclusion: Key Finding

- Vectorization at XOR-level is much better choice than vectorization of finite field operations
  - Higher throughput (~ 2x faster)
  - Easy migration to newer SIMD ISA

SSE: 128 bit  
AVX2: 256 bit  
AVX-512: 512 bit

_mm_xor_si128 → _mm256_xor_si256 → _mm512_xor_epi32
Thank you!

Questions?

Tianli Zhou: zhoutl1106@tamu.edu
Chao Tian: chao.tian@tamu.edu

Source code:
https://bitbucket.org/zhoutl1106/zerasure.git
https://github.com/zhoutl1106/zerasure.git