Accelerate Reed-solomon coding for Fault-Tolerance in RAID-like system

Shuai YUAN

NTHU
LSA lab

Outline

- Background
- How to use CUDA to accelerate
- Experiment
- Q & A

Why fault-tolerance?

In a RAID-like system, storage is distributed among several devices, the probability of one of these devices failing becomes significant.

Why fault-tolerance?

In a RAID-like system, storage is distributed among several devices, the probability of one of these devices failing becomes significant.
MTTF (mean time to failure) of one device is P,
MTTF of a system of n devices is $\dfrac{P}{n}$.

Why fault-tolerance?

In a RAID-like system, storage is distributed among several devices, the probability of one of these devices failing becomes significant.
MTTF (mean time to failure) of one device is P,
MTTF of a system of n devices is $\frac{P}{n}$.
Thus in such systems, fault-tolerance must be taken into account.

Why erasure codes?

Replication is expensive!
e.g. To achieve double-fault tolerance, we need 2 replicas.

Erasure Codes

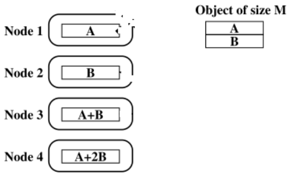
Given a message/file of k equal-size blocks/chunks/fragments, encode it into n blocks/chunks/fragments ($n > k$)

- Optimal erasure codes: any k out of the n fragments are sufficient to recover the original message/file.
  i.e. can tolerate arbitrary $(n - k)$ failures.
  We call this (n,k) MDS (maximum distance separable) property.
  e.g. Reed Solomon Codes
- Near-optimal erasure codes: require $(1 + \varepsilon)k$ code blocks/chunks to recover ($\varepsilon > 0$).
  i.e. cannot tolerate arbitrary $(n - k)$ failures.
  e.g. Local Reconstruction Codes (used in WAS)

Replication vs. Erasure Codes

Given a file of size M

Using Reed Solomon Codes(n=4, k=2)



Achieve double fault tolerance

- R-S Codes: storage overhead $= M$
- Relication: storage overhead $= 2M$ (2 replicas)

Reed-Solomon encoding process

Given file F: divide it into k equal-size native chunks:
$F = [F_i]_{i=1,2,\ldots,k}$. Encode them into $(n-k)$ code chunks:
$C = [C_i]_{i=1,2,\ldots,n-k}$.
Use Encoding Matrix $EM_{(n-k)\times k}$ to produce code chunks:

$$C^T = EM \times F^T$$

C_i is the linear combination of $F_1, F_2, \ldots, F_k$.

Reed-Solomon encoding process

In our case,

$$
\begin{aligned}
F &= \begin{pmatrix} A & B \end{pmatrix} \\
EM &= \begin{pmatrix} 1 & 1 \\ 1 & 2 \end{pmatrix} \\
C^T &= \begin{pmatrix} 1 & 1 \\ 1 & 2 \end{pmatrix} \times \begin{pmatrix} A \\ B \end{pmatrix} \\
&= \begin{pmatrix} A + B \\ A + 2B \end{pmatrix}
\end{aligned}
$$

Reed-Solomon encoding process

Let $P = [P_i]_{i=1,2,\ldots,n} = [F_1, F_2, \ldots, F_k, C_1, C_2, \ldots, C_{n-k}]$ be the n chunks in storage, $EM' = \begin{pmatrix} I \\ EM \end{pmatrix}$, Here

$$I = \begin{pmatrix} 1 & 0 & \ldots & 0 \\ 0 & 1 & \ldots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \ldots & 1 \end{pmatrix}$$

then

$$\begin{aligned} P^T &= EM' \times F^T \\ &= \begin{pmatrix} F^T \\ C^T \end{pmatrix} \end{aligned}$$

Reed-Solomon encoding process

In our case,

$$
\begin{aligned}
EM' &= \begin{pmatrix} I \\ EM \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 1 \\ 1 & 2 \end{pmatrix} \\
P^T &= EM' \times F^T \\
&= \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 1 \\ 1 & 2 \end{pmatrix} \times \begin{pmatrix} A \\ B \end{pmatrix} \\
&= \begin{pmatrix} A \\ B \\ A + B \\ A + 2B \end{pmatrix}
\end{aligned}
$$

Reed-Solomon encoding process

EM' is the key of MDS property!

Reed-Solomon encoding process

Theorem (necessary and sufficient condition)

Every possible $k \times k$ submatrix obtained by removing $(n-k)$ rows from EM' has full rank.
equivalent expression of full rank:

- *rank* $= k$
- *non-singular*

$\vdots$

Alternative view:
Consider the linear space of $P = [P_i]_{i=1,2,\ldots,n} = [F_1, F_2, \ldots, F_k, C_1, C_2, \ldots, C_{n-k}]$, its dimension is k, and any k out of n vectors form a basis of the linear space.

Reed-Solomon encoding process

Reed-Solomon Codes uses Vandermonde matrix V as EM

$$V = \begin{bmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & 2 & 3 & \dots & k \\ 1^2 & 2^2 & 3^2 & \dots & k^2 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1^{(n-k)} & 2^{(n-k)} & 3^{(n-k)} & \dots & k^{(n-k)} \end{bmatrix}$$

Reed-Solomon encoding process

$$
EM' = \begin{bmatrix}
1 & 0 & 0 & \dots & 0 \\
0 & 1 & 0 & \dots & 0 \\
0 & 0 & 1 & \dots & 0 \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
0 & 0 & 0 & \dots & 1 \\
1 & 1 & 1 & \dots & 1 \\
1 & 2 & 3 & \dots & k \\
1^2 & 2^2 & 3^2 & \dots & k^2 \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
1^{(n-k)} & 2^{(n-k)} & 3^{(n-k)} & \dots & k^{(n-k)}
\end{bmatrix}
$$

Reed-Solomon encoding process

Remark:

- All arithmetic operations in Galois Field GF(2^x). Then every number is less than 2^x.
- EM' satisfies MDS property.

Reed-Solomon decoding process

We randomly select k out of n fragments, say $X = [x_1, x_2, \cdots, x_k]^T$.
Then we use the coefficients of each fragments x_i to construct a $k \times k$ matrix V'.
Original data can be regenerated by multiplying matrix X with the inverse of matrix V':

$$F = V'^{-1}X$$

Optimization motivation

The reason that discourages using Reed-Solomon coding to replace replication is its high computational complexity:

- Galois Field arithmetic
- matrix operations

Accelerate operations in Galois Field (GF)

GF(2^w) field is constructed by finding a primitive polynomial $q(x)$ of degree w over GF(2), and then enumerating the elements (which are polynomials) with the generator x. Addition in this field is performed using polynomial addition, and multiplication is performed using polynomial multiplication and taking the result modulo $q(x)$.

Accelerate operations in Galois Field (GF)

GF(2^w) field is constructed by finding a primitive polynomial $q(x)$ of degree w over GF(2), and then enumerating the elements (which are polynomials) with the generator x. Addition in this field is performed using polynomial addition, and multiplication is performed using polynomial multiplication and taking the result modulo $q(x)$.

In implementation, we map the elements of GF(2^w) to binary words of size w, so that computation in GF(2^w) becomes bitwise operations. Let $r(x)$ be a polynomial in GF(2^w). Then we can map $r(x)$ to a binary word b of size w by setting the ith bit of b to the coefficient of x^i in $r(x)$.

Accelerate operations in Galois Field (GF)

GF(2^w) field is constructed by finding a primitive polynomial $q(x)$ of degree w over GF(2), and then enumerating the elements (which are polynomials) with the generator x. Addition in this field is performed using polynomial addition, and multiplication is performed using polynomial multiplication and taking the result modulo $q(x)$.

In implementation, we map the elements of GF(2^w) to binary words of size w, so that computation in GF(2^w) becomes bitwise operations. Let $r(x)$ be a polynomial in GF(2^w). Then we can map $r(x)$ to a binary word b of size w by setting the ith bit of b to the coefficient of x^i in $r(x)$.

After mapping, Addition/subtraction of binary elements of GF(2^w) can be performed by bitwise exclusive-or.

Accelerate operations in Galois Field (GF)

However, multiplication is still costly.
Use bitwise operations to emulate polynomial multiplication in GF(2^w): a loop of w iterations.
e.g. In GF(2^8), multiplying 2 bytes takes 8 iterations.

Accelerate operations in Galois Field (GF)

CPU approach

store all the result in a multiplication table

e.g. In GF(2^8), multiplying 2 bytes takes $256 \times 256 = 64KB$
consumes a lot of space $\rightsquigarrow$ not suitable for GPU

Accelerate operations in Galois Field (GF)

trade-off between computation and memory consumption ([1])
We build up two smaller tables:

- exponential table: maps from a binary element b to power j such that x^j is equivalent to b.
- logarithm table: maps from a power j to its binary element b such that x^j is equivalent to b.

Multiplication in GF(2^w):

- looking each binary number in the logarithm table for its logarithm (this is equivalent to finding the polynomial)
- adding the logarithms modulo $2^w - 1$ (this is equivalent to multiplying the polynomial modulo $q(x)$)
- looking up exponential table to convert the result back to a binary number

Totally three table lookups and a modular addition.

Accelerate encoding process

- Generating Vandermonde's Matrix
- Matrix multiplication

Accelerate decoding process

- Computing inverse matrix
- Matrix multiplication

Accelerate matrix inversion

Guassian/Guass-Jordon elimination in GF(2^8)

$$[V'I] = \begin{bmatrix} 1 & 0 & 0 & 0 & | & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & | & 0 & 1 & 0 & 0 \\ 1 & 2 & 3 & 4 & | & 0 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 & | & 0 & 0 & 0 & 1 \end{bmatrix}$$

Accelerate matrix inversion

Guassian/Guass-Jordon elimination in GF(2^8)

$$[V'I] = \left[\begin{array}{cccc|cccc} 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 2 & 3 & 4 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 1 & 0 & 0 & 1 \end{array}\right]$$

Accelerate matrix inversion

Guassian/Guass-Jordon elimination in GF(2^8)

$$[V'I] = \left[\begin{array}{cccc|cccc} 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 3 & 4 & 1 & 2 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 0 & 1 \end{array}\right]$$

Accelerate matrix inversion

Guassian/Guass-Jordon elimination in GF(2^8)

$$[V'I] = \left[\begin{array}{cccc|cccc} 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 247 & 244 & 245 & 244 & 0 \\ 0 & 0 & 0 & 246 & 245 & 244 & 244 & 1 \end{array}\right]$$

Accelerate matrix inversion

Guassian/Guass-Jordon elimination in GF(2^8)

$$[V'I] = \left[\begin{array}{cccc|cccc} 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 104 & 187 & 186 & 210 \\ 0 & 0 & 0 & 1 & 105 & 186 & 186 & 211 \end{array}\right]$$

Accelerate matrix inversion

Guassian/Guass-Jordon elimination in GF(2^8)

GPU

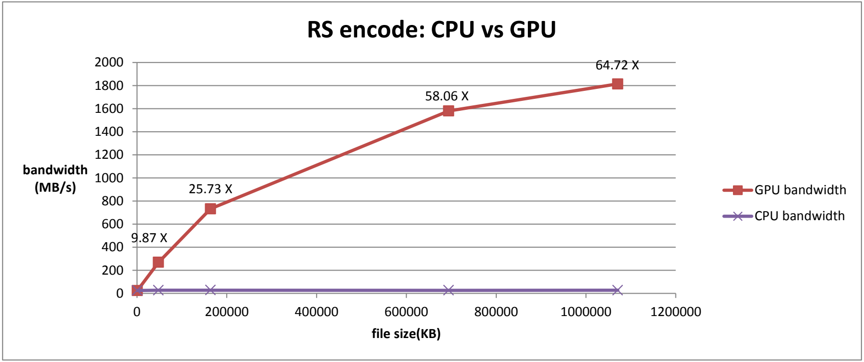
- normalize row
- make column into reduced echelon form

Experiment Settings

- Intel(R) Xeon(R) CPU E5620 @ 2.40GHz
- nVidia Corporation GF100 [Tesla C2050]
  - 448 CUDA cores @ 1.15GHz
  - 3GB GDDR5 (1.44GB/s)
  - 64KB L1 cache + shared memory / 32 cores, 768KB L2 cache

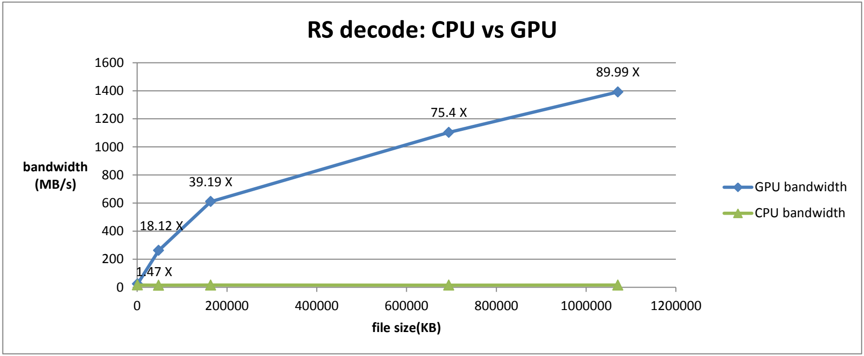
Experiment Result (fixed n, k)

$k = 4, n = 6$



Experiment Result (fixed n, k)

$k = 4, n = 6$



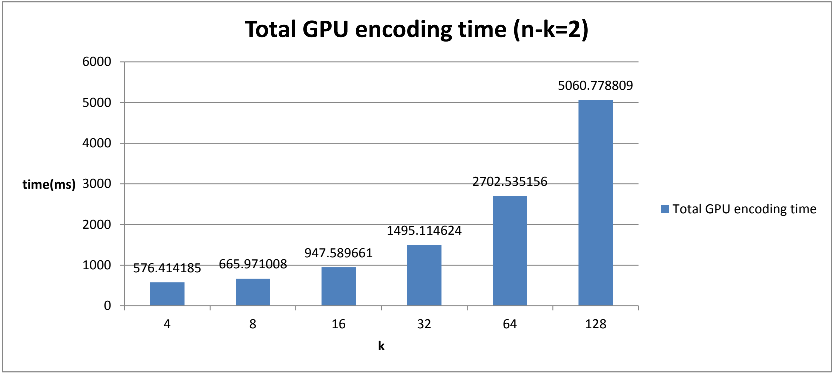
Experiment Result (fixed n, k)

Conclusion

1. CPU bandwidth is nearly constant.
2. GPU achieves better speed-up when the file size scales up.
3. GPU performance speed-up gets slower.

Experiment Result (fixed file size)

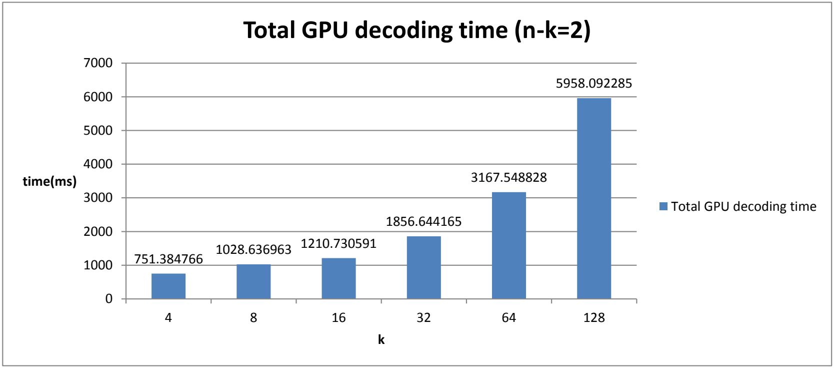
File size: 1.1GB
double fault tolerance



Experiment Result (fixed file size)

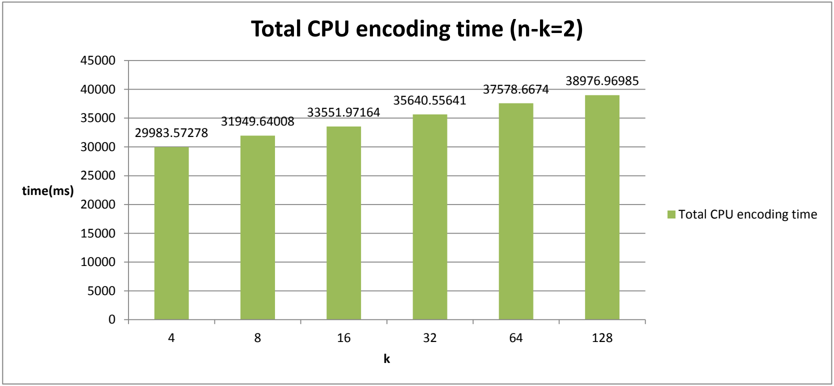
File size: 1.1GB

double fault tolerance



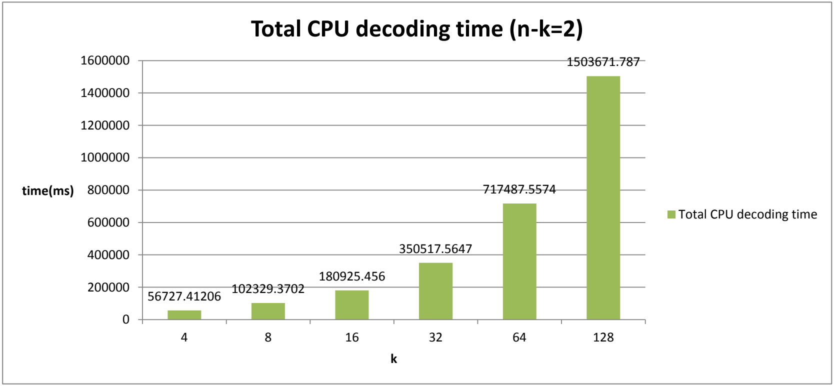
Experiment Result (fixed file size)

File size: 1.1GB
double fault tolerance



Experiment Result (fixed file size)

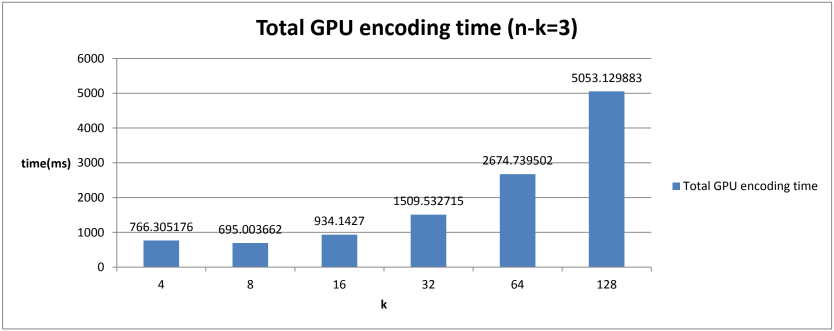
File size: 1.1GB
double fault tolerance



Experiment Result (fixed file size)

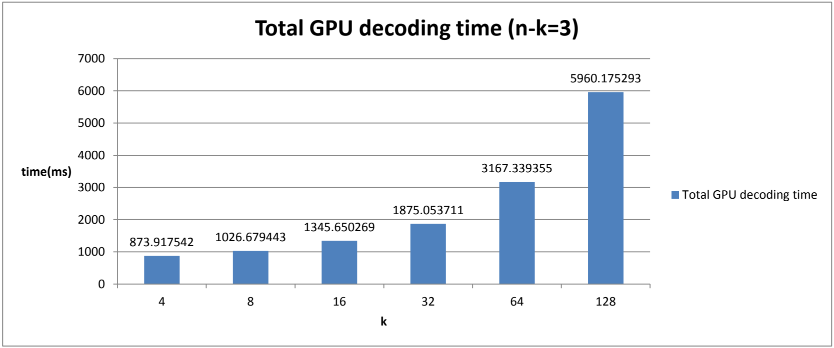
File size: 1.1GB

triple fault tolerance



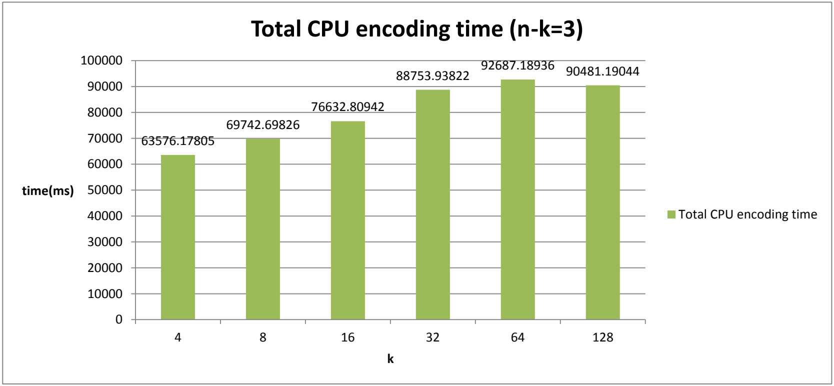
Experiment Result (fixed file size)

File size: 1.1GB
triple fault tolerance



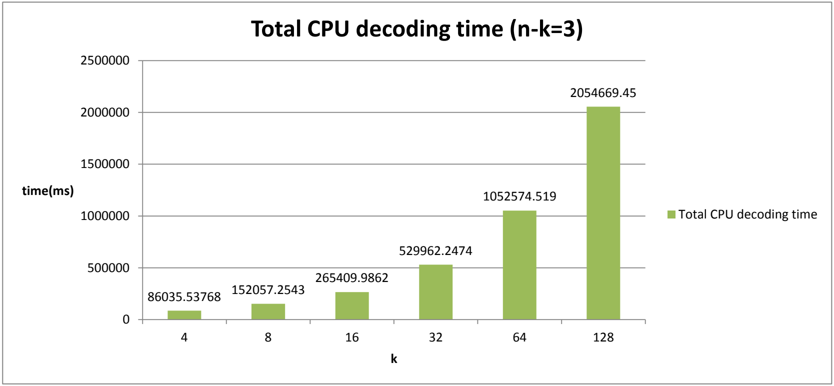
Experiment Result (fixed file size)

File size: 1.1GB
triple fault tolerance



Experiment Result (fixed file size)

File size: 1.1GB
triple fault tolerance



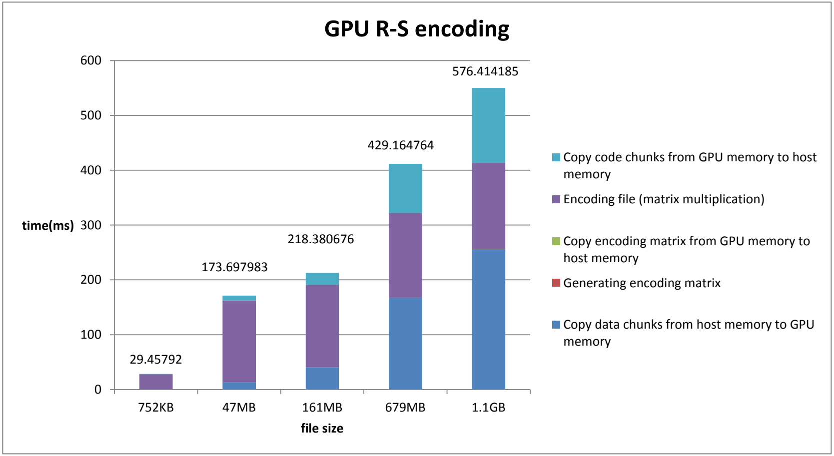
Experiment Result (fixed file size)

Conclusion

1. If we divide the original data into more fragments (i.e. increasing k), we can achieve less storage overhead, meanwhile both CPU and GPU performance will become worse.
2. In reality, we seldom cut the files into more than 30 fragments, so the overall performance of GPU would still be acceptable.

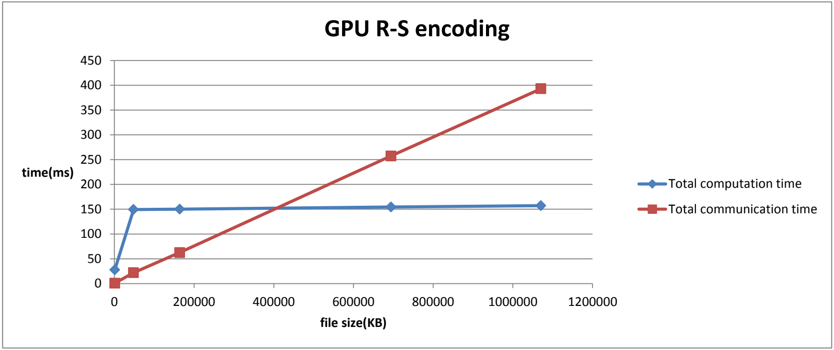
Experiment Result (GPU cost breakdown)

$k = 4, n = 6$



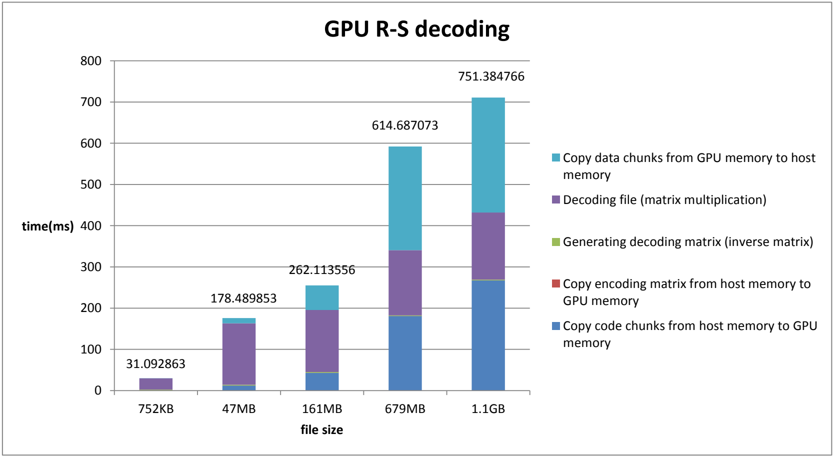
Experiment Result (GPU cost breakdown)

$k = 4, n = 6$



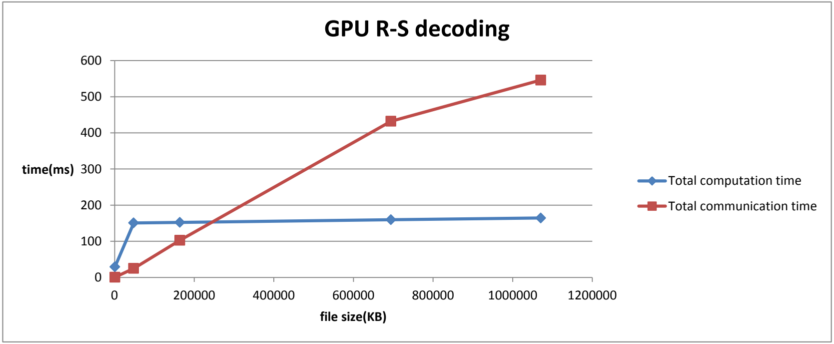
Experiment Result (GPU cost breakdown)

$k = 4, n = 6$



Experiment Result (GPU cost breakdown)

$k = 4, n = 6$



Experiment Result (GPU cost breakdown)

Conclusion

1. GPU computation time becomes almost constant when the file size is large enough.
2. Matrix multiplication time occupies much more percentage than matrix generation time in the breakdown of GPU computation time.
3. GPU spends more time in communication and the communication time is getting more as the file size becomes larger. This phenomenon explains why GPU performance speed-up gets slower.

Q & A

Thank You!

J.S. Plank et al.
A tutorial on reed-solomon coding for fault-tolerance in raid-like systems.
Software Practice and Experience, 27(9):995–1012, 1997.