# Accelerate Reed-Solomon Coding for Fault-Tolerance in RAID-like system

Shuai YUAN

## 1 Background

### 1.1 Why Reed-Solomon codes?

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

One common approach for fault-tolerance is replication. However, when data size scales up, replication will result in large storage overhead.

Here comes another solution of erasure codes, which can reduce the redundancy ratio tremendously. And among various types of erasure codes, Reed-Solomon code is one of the popular that frequently used in research.

Reed-Solomon code is a kind of optimal erasure codes which has the so-called Maximum Distance Separable (MDS) property. $(n, k)$ MDS property is defined as: Given original data, we divide it into $k$ equal-size native fragments, and encode them into $n$ fragments. Then we can select any $k$ out of $n$ fragments to reconstruct the original data. It provides good flexibility for reconstruction, and the storage overhead is low: let the size of the original data is $M$, then the storage overhead is only $(n-k)\frac{M}{k}$ since we have $(n-k)$ more fragments.

### 1.2 Reed-Solomon coding mechanism

Let $D = [d_1, d_2, \cdots, d_k]^T$ be the $k$ native data fragments, Reed-Solomon code has mainly two processes:

- encoding process: a encoding matrix $V = [v(i,j)]_{k\times(n-k)}$ is used to produce $(n-k)$ code fragments $C = [c_1, c_2, \cdots, c_{(n-k)}]^T$:

  $$c_j = \sum_{i=1}^{k} v(i,j) \cdot d_i$$

  To achieve MDS property, Reed-Solomon code uses a particular encoding matrix, which is a Vandermonde's Matrix $v(i,j) = i^{(j-1)}$, and limit all arithmetic operations in Galois Field.

  After completing code fragments generation, we store $(n-k)$ code fragments together with $k$ native fragments.

- 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 $V'$:

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

### 1.3 Optimization motivation

The reason that discourages using Reed-Solomon coding to replace replication is its high computational complexity: Galois Field arithmetic is complex, and matrix operations consume a lot of time.

Our goal is to use GPU to accelerate Reed-Solomon encoding and decoding. Our code is written in CUDA.

## 2 Data type and size

Instead of implementing the Reed-Solomon coding above the bare storage system, I implement it upon the file system. leaving some issues like how to stripe the file into several devices to the file system and controller of storage system.

For the input data, it must be a file. There is no special limitation for the type and size of the file.

## 3 How to use CUDA to accelerate

### 3.1 Accelerate operations in Galois Field (GF)

Operations in Galois Field is one of the bottlenecks in Reed-Solomon coding performance, thus deserving to find an appropriate way for GPU to accelerate.

As stated in [5], 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 $i$th bit of $b$ to the coefficient of $x^i$ in $r(x)$.

After mapping, addition and subtraction of binary elements of GF($2^w$) can be performed by bitwise exclusive-or, which is fast.

However, multiplication is still costly. One must convert the binary numbers to their polynomial elements, multiply the polynomials modulo $q(x)$, and then convert the answer back to binary. To compute the result of binary word $b$ multiplied by binary word $a$ in GF($2^w$), we have to use bitwise operations to emulate polynomial of $b$ multiplying the polynomial of $a$ modulo $q(x)$. Here shows how to perform byte multiplication in GF($2^8$):

---

```
uint8_t gf256_mul_by_bit(uint8_t a, uint8_t b)
{
  uint8_t result;
  while(b)
  {
    if(b & 1)
    {
      //emulate polynomial addition
      result ^= a;
    }
    //primitive polynomial of GF(2^8):
    //x^8+x^4+x^3+x^2+1
    //which is mapped into 0x1d
    //(a & 0x80? 0x1d: 0): emulate polynomial modulo
    a = (a << 1) ^ (a & 0x80? 0x1d: 0);
    b >>= 1;
  }
  return result;
}
```

Such a loop takes at most 8 iterations to complete. In Reed-Solomon coding, multiplication in GF($2^w$) is used so frequently that such cost is unbearable.

Another approach is to store all the result in a multiplication table. which is adopted in CPU-based implementation like NCCloud [2].

The multiplication table may use the bitwise method mentioned above to setup at the beginning, or simply defined as constant.

Again taking GF($2^8$) for instance, we setup the following table:

```
uint8_t  gf256mul[256][256];
```

Here gf256mul[a][b] stores the result of binary word $b$ multiplied by binary word $a$. Therefore, when computing multiplication, we simply need to find the corresponding item in the multiplication table. Such an approach can speed up computation while sacrificing more memory space.

For a large-width finite field, this method might not be suitable for GPU. GPU has greater computation power than CPU, but memory access can be another bottleneck. One common optimization technique is to allocate shared memory space for the multiplication table. But in GF($2^8$), the multiplication table has $256 \times 256$ items, which consumes so large space that shared memory may not be able to provide. In addition, note that $a \times b = b \times a$, there are redundancies in the multiplication table (e.g. gf256mul[a][b] is the same as gf256mul[b][a]), which is a waste of space.

However, this approach can still be applied in a field with smaller width, and we have implemented it in GF($2^4$). We use the multiplication table with $16 \times 16$ items and two helper tables(exponential and logarithm tables, will be introduced later) with 16 items correspondingly for the convenience of div and pow operations. These tables are stored in the Constant Memory of GPU, which will be cached. Since a byte is the smallest addressable unit, we need to transfer byte operations into multiple nibble (4-bit) operations. (Note that simply extending 4bits to 8bits is not appropriate, since it is equivalent to doubling the file size and will increase the execution time.) Here shows how we perform byte multiplication in GF($2^4$):

```
__host__ __device__ uint8_t gf_mul(uint8_t a, uint8_t b)
{
        uint8_t a_high_nibble = ( a >> 4 ) & 0x0f;
        uint8_t b_high_nibble = ( b >> 4 ) & 0x0f;
        uint8_t result_high_nibble = gfmul_16[a_high_nibble][b_high_nibble];
        uint8_t a_low_nibble = a & 0x0f;
        uint8_t b_low_nibble = b & 0x0f;
        uint8_t result_low_nibble = gfmul_16[a_low_nibble][b_low_nibble];

        return ( (result_high_nibble << 4) | result_low_nibble);
}
```

For a large-width finite field, we need to make a trade-off between computation and memory: we still need to use shared memory to accelerate, without produce much memory overhead like the multiplication table method. Here comes the third approach [4]:

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($2^w$) then consists of looking each binary number in the logarithm table for its logarithm (this is equivalent to finding the polynomial), then adding the logarithms modulo $2^w - 1$ (this is equivalent to multiplying the polynomial modulo $q(x)$) and looking up exponential table to convert the result back to a binary number. Here is the pseudo code for multiplication in GF($2^8$):

```
//number of elements in GF(2^8)
int NW = 1 << 8;
uint8_t gf_mul(uint8_t a, uint8_t b)
{
  int result;
  if (a == 0 || b == 0)
  {
    return 0;
  }
  result = (gflog[a] + gflog[b]) % (NW-1);
  return gfexp[result];
}
```

Division is performed in the same manner, except the logarithms are subtracted instead of added.

In this method, multiplication and division of two binary elements takes three table lookups and a modular addition, which achieves better compromise for GPU.

## 3.2 Accelerate encoding process

Firstly, we need to generate a MDS encoding matrix, which is a Vandermonde's Matrix in Reed-Solomon coding algorithm. This can be paralleled in GPU by letting every thread compute few items of Vandermonde's Matrix. Since the power operations in GF($2^w$) is mapped to table entry lookup and memory access, the workload of all the threads is balanced.

Matrix multiplication to generate code fragments is the most suitable part for CUDA to accelerate. Some common techniques like tile algorithm are used to optimize the computation/communication ratio. Tile algorithm is used to improve the temporary locality and reduce cache misses when an array is accessed in a row major order and in a column major order at the same time (Matrix multiplication is such a good example). This technique is introduced in the classical computer architecture textbook [1] as **blocking**. For further details of the technique, please refer to [3].

## 3.3 Accelerate decoding process

In the decoding process of Reed-Solomon coding, we read arbitrary $k$ out of $n$ fragments to reconstruct and the total time of read access is as fast as the response time of the slowest storage node.

After knowing which $k$ fragments are used for decoding, we know the corresponding rows of coefficients in the encoding matrix $V$. These rows form the $k \times k$ matrix $V'$, and we need to compute its inverse. Unfortunately, we found no standard APIs for inversing a matrix in Galois Field, so we implement Gauss elimination to compute inverse matrix: augment the square matrix $V'$ with the identity matrix $I$ of the same dimensions to obtain $[V'|I]$ and then apply matrix operations to transfer $[V'|I]$ into its reduced row echelon form: $V'^{-1}[V'|I] = [I|V'^{-1}]$.

Gauss elimination contains the following steps:

1. Check whether the diagonal item in the current row of $V'$ is nonzero. If if is zero, find a nonzero item and switch the two columns. The same column switching should also apply to $I$. Then the diagonal item in the current row of $V'$ is our pivot.

2. Normalize the current row by the pivot.

3. Eliminate other rows so that the reverse column of the the current row becomes reduced echelon form.

Step 2 and Step 3 are suitable for GPU to parallely execute, but since we have to set barriers between steps, Gauss elimination is still not a fully-paralleled algorithm.

Finally, we multiply matrix $V'^{-1}$ with the matrix of $k$ fragments, and the original data is reconstructed. This part is the same as the matrix multiplication in the encoding process.

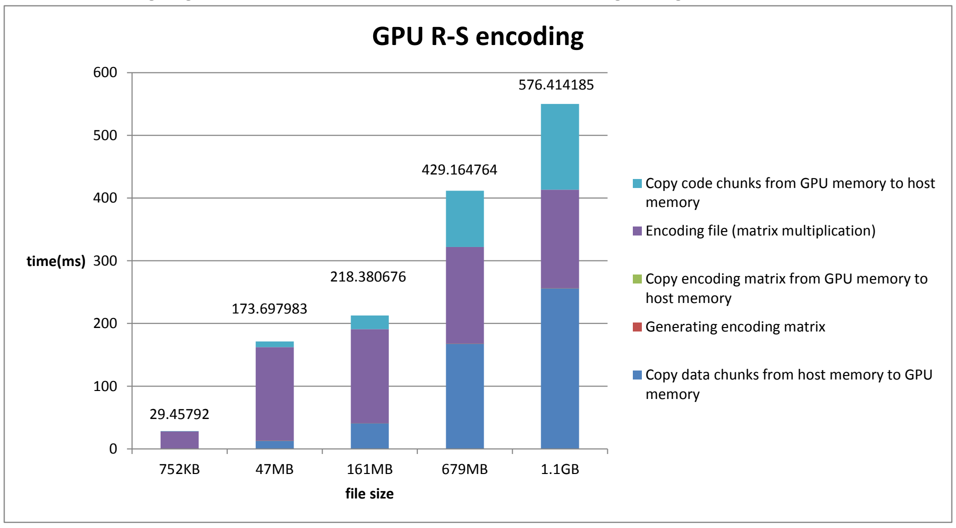
## 4 Experiment Result

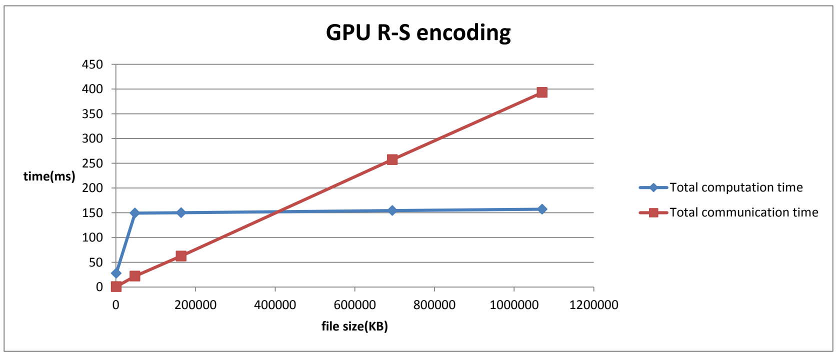
Experiment Setting:

- CPU: Intel(R) Xeon(R) CPU E5620 @ 2.40GHz
- GPU: nVidia Corporation GF100 [Tesla C2050]
- 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

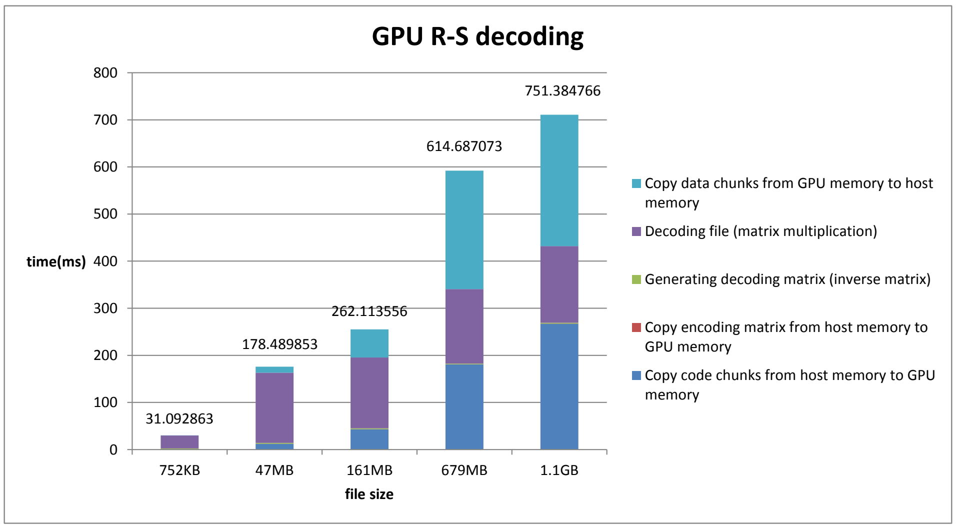
### 4.1 GPU Cost Breakdown

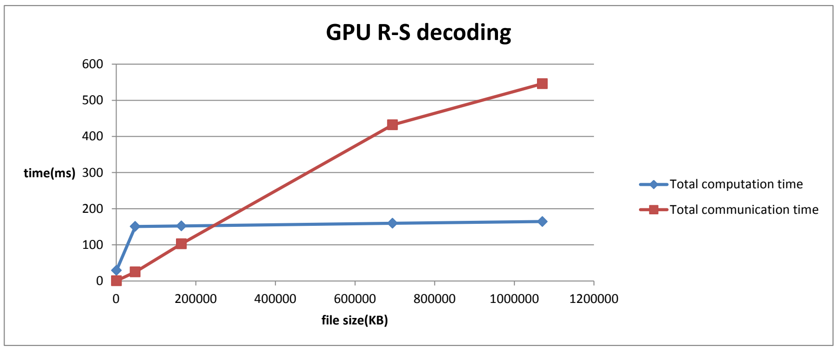
In this experiment, we set $k = 4, n = 6$.
The following figure shows the time of GPU encoding progress steps.





The following figure shows the time of GPU decoding progress steps.





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.

## 4.2 CPU vs GPU

We set $k = 4, n = 6$, the same as the above experiment.

The following figure shows the bandwidth comparison between CPU and GPU in the Reed-Solomon encoding progress.

Table 1: GPU speed-up in the Reed-Solomon encoding progress

| size(Byte) | 752 | 47968 | 163864 | 694276 | 1070616 |
|---|---|---|---|---|---|
| speed-up | - | 9.87 | 25.73 | 58.06 | 64.72 |

Table 2: GPU speed-up in the Reed-Solomon decoding progress

| size(Byte) | 752 | 47968 | 163864 | 694276 | 1070616 |
|---|---|---|---|---|---|
| speed-up | 1.47 | 18.12 | 39.19 | 75.4 | 89.99 |

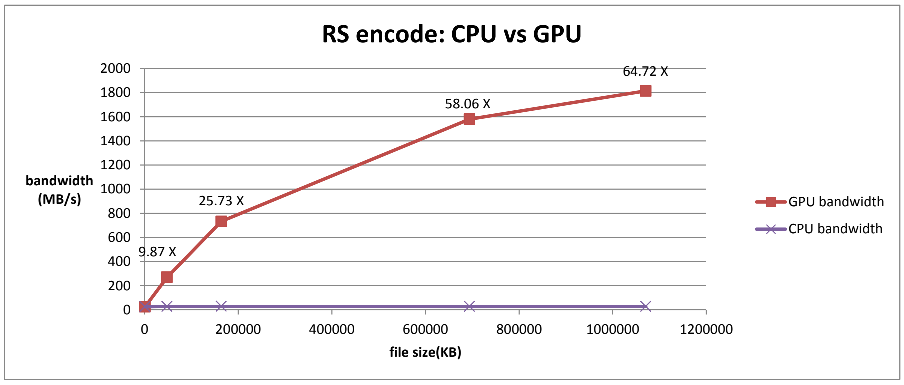


Table 1 is the performance speed-up of GPU in the Reed-Solomon encoding progress:

The following figure shows the bandwidth comparison between CPU and GPU in the Reed-Solomon decoding progress.

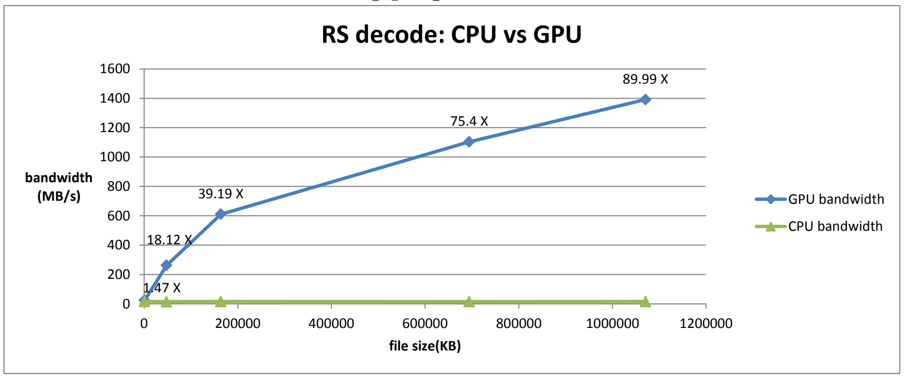


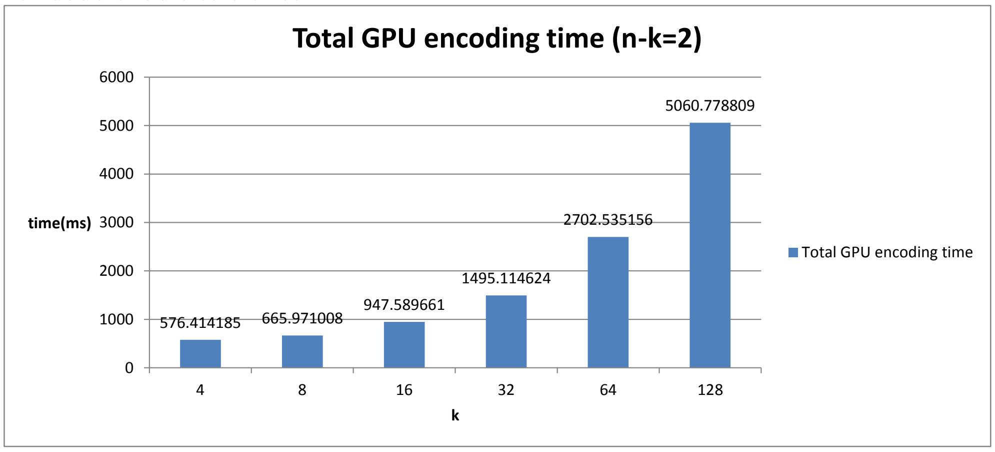
Table 2 is the performance speed-up of GPU in the Reed-Solomon encoding progress:

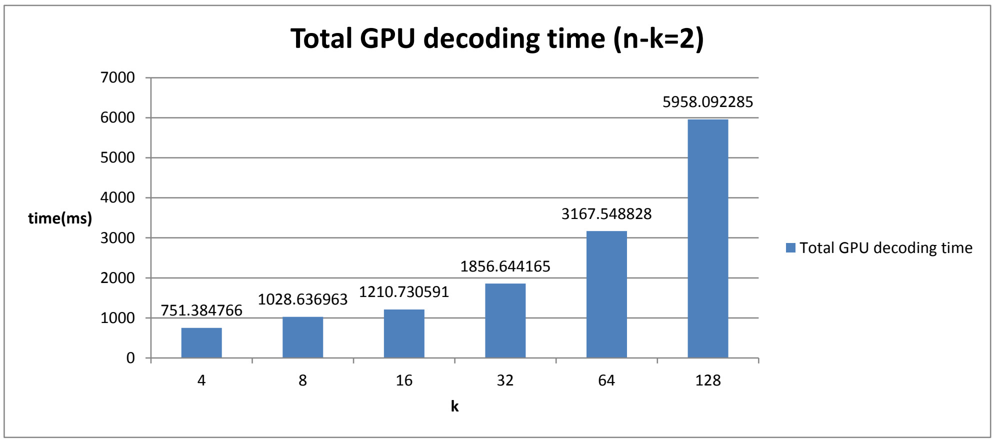
Conclusion:

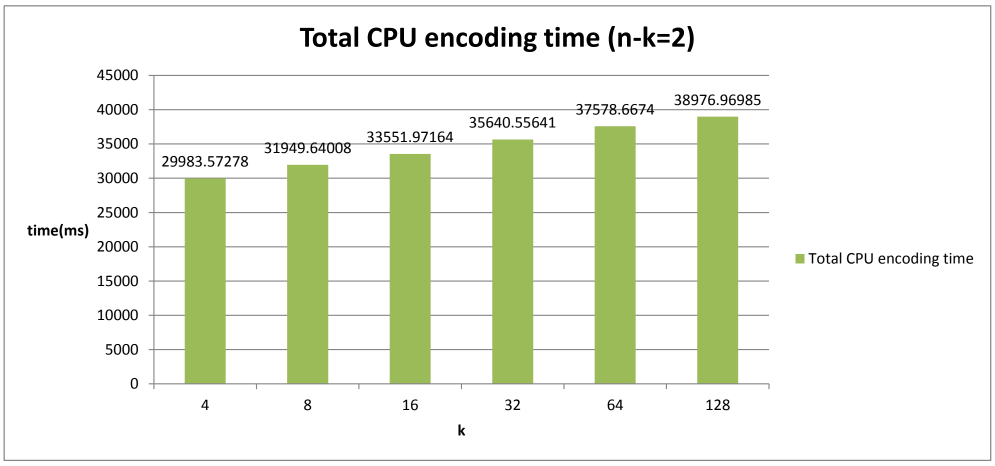
1. CPU bandwidth is nearly constant.
2. GPU achieves better speed-up when the file size scales up.
3. GPU performance speed-up gets slower.

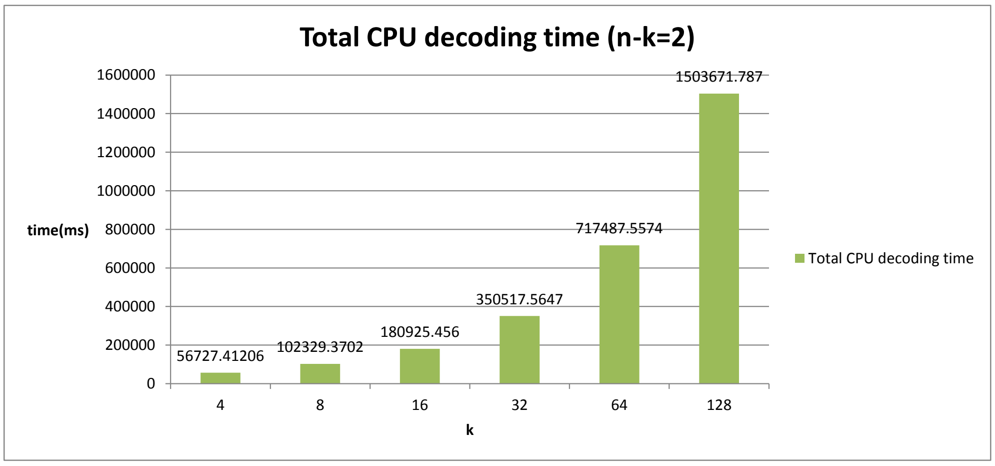
Since double and triple fault tolerance is the most commonly used, we treat $k$ as a variable with fixed file size of 1.1GB and run several experiments in these two cases.

For double fault tolerance:

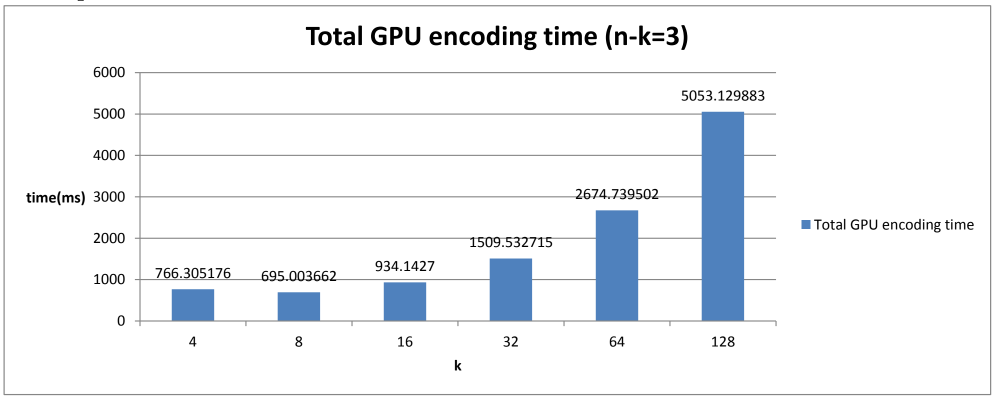


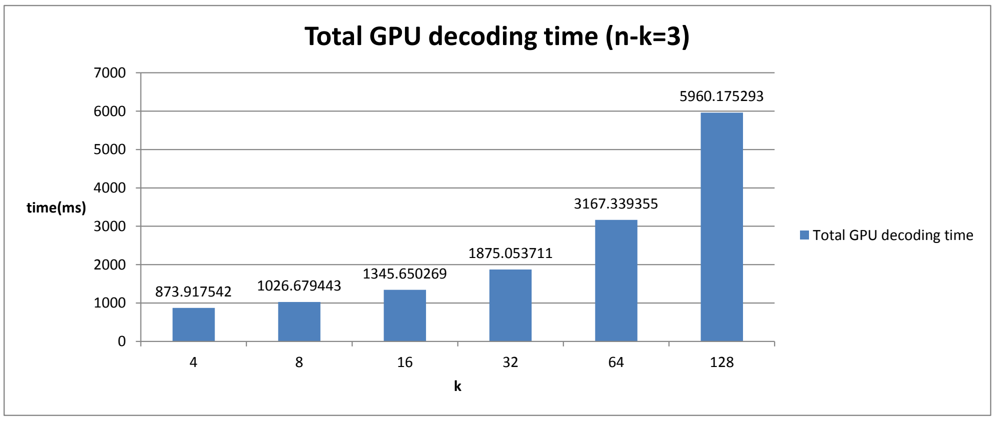


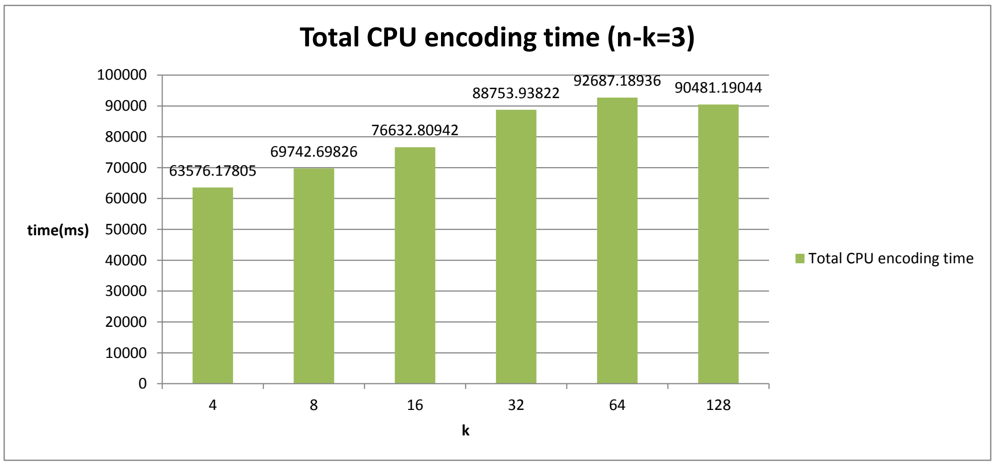


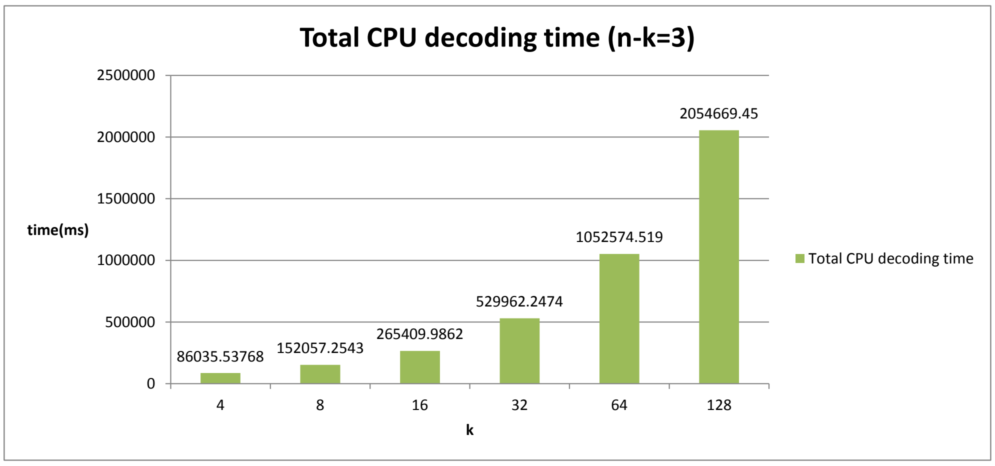


For triple fault tolerance:









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.

## 4.3 GF(16) method vs GF(256) method

In this experiment, we set $k = 4, n = 6$, and use 1.1GB as file size.

Table 3 is the performance speed-up of GPU in the Reed-Solomon encoding progress:

Analysis:

For a Galois Field multiplication operation, GF(256) method costs three table-lookups in shared memory (programmable cache in GPU) and one addition and modular in the rational field, while GF(16) costs two table-lookups in constant memory (normally is cached, will costs one additional memory access only on a cache miss) and eight bit-operations (>>, & and |). Thus, the GF(256) method has more communication costs and less computation costs

Table 3: GPU cost breakdown comparison between GF(16) and GF(256) methods

| GPU cost breakdown | GF(16) | GF(256) |
|---|---|---|
| Copy data from CPU to GPU(ms) | 277.482697 | 265.243866 |
| Generating encoding matrix(ms) | 0.047328 | 0.252352 |
| Copy encoding matrix from GPU to CPU(ms) | 0.023104 | 0.02352 |
| Encoding file(ms) | 9.11728 | 160.520676 |
| copy code from GPU to CPU(ms) | 193.164154 | 139.608734 |
| Total computation time(ms) | 9.164608 | 160.773026 |
| Total communication time(ms) | 470.669952 | 404.876129 |
| Total GPU encoding time(ms) | 505.69104 | 770.560547 |
| GPU bandwidth(MB/s) | 2067.514 | 1356.835 |
| Copy code from CPU to GPU(ms) | 276.235229 | 257.23764 |
| Copy encoding matrix from CPU to GPU(ms) | 0.013536 | 0.030016 |
| Generating decoding matrix (inverse matrix)(ms) |  | 1.93504 |
| Decoding file(ms) | 16.037184 | 166.371994 |
| copy data from GPU to CPU(ms) | 386.210175 | 272.606415 |
| Total computation time(ms) | 16.498079 | 168.307037 |
| Total communication time(ms) | 662.458984 | 529.874084 |
| Total GPU decoding time(ms) | 712.471741 | 731.763733 |
| GPU bandwidth(MB/s) | 1467.46 | 1428.772 |

than the GF(16) one. In the experiment, the GF(16) method is faster in performing matrix multiplication. However, there is little gap between the overall performance of the two methods.

## 5 Future Works

- 
- Increase the parallelism of matrix inversion algorithm.
- Better CPU/GPU cooperation:
  - 
- Implementation Issues:
  - While GPU is computing the encoding matrix, CPU can load the input file(s) into memory, thus making the IO and computation processing roughly at the same time.

## 6 Appendix

The source code of this project is available under the GPLv3. And the project's Git repository can be checked out through anonymous access with the following command.

```
git clone https://github.com/yszheda/GPU-RSCode.git
```

The **master** branch is the implementation of GF(256) Reed-Solomon Code, and the **extend** is the implementation of GF(16) Reed-Solomon Code, use the following command to checkout one of the branch:

```
git checkout <branch>
```

## References

[1] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach.* Elsevier, 2012.

[2] Y. Hu, H.C.H. Chen, P.P.C. Lee, and Y. Tang. Nccloud: applying network coding for the storage repair in a cloud-of-clouds. In *USENIX FAST*, 2012.

[3] CUDA NVIDIA. Cuda c best practices guide, 2010.

[4] 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.

[5] B. Sklar. Reed-solomon codes. *Downloaded from URL http://www. informit. com/content/images/art. sub.–sklar7. sub.–reed-solomon/elementLinks/art. sub.–sklar7. sub.–reed-solomon. pdf,(unknown pub date)*, pages 1–33, 2001.