Fast Software Implementations of Finite Field Operations *
(Extended Abstract)

Cheng Huang Lihao Xu
Department of Computer Science & Engineering
Washington University in St. Louis, MO 63130
{cheng, lihao}@cse.wustl.edu

1 Introduction

Software implementation of *finite field operations* is becoming more and more widely used in communication, cryptography and multimedia applications. It is more flexible and easier to deploy, compared to dedicated hardware component. However, finite field operations often become the bottle-neck for high throughput applications and are impacting the life time of power constrained devices. It is thus desirable to keep pushing the efficiency of their implementation. In this paper, we describe several improvement techniques to increase the throughput of finite field operations without incurring extra complexity.

Maintaining pre-computed tables to simplify operation and improve performance is a common practice [3]. In finite field operations, modular operation and conditional branch (**if**$\cdots$**then**) are frequently used. These operations are very expensive in computation compared to table look-up. In this paper, we explore the idea of precomputation to a new depth and replace these operations with simpler ones, using moderately larger memory, which is usually not an issue in most platforms. We propose to use *augmented* table to eliminate modular operation and conditional branch completely. Simulation results show that the best improvement approach reduces execution time of finite field multiplication and division by about 67% and increases the encoding throughput of *Reed-Solomon* (RS) [4] code about 3 times. We also show that there is limitation in trading memory usage for efficiency: when complete precomputation and *extra* large table are used, the throughput of finite field operations degrades instead.

2 Improvement Approaches of GF(2^n) Operation

We consider the most frequently used operations, *addition*, *multiplication* and *division* in Galois field over 2^n in this paper.

In GF(2^n), addition operation is trivial and equals to the *XOR* sum of two operators. In the rest sections, we focus mainly on multiplication and division operations and investigate approaches to improve both.

Every element in a finite field can be represented uniquely by a power to a primitive element α [1], except for 0. Let $x = \alpha^i, x \in GF(2^n)$, then i is the *discrete logarithm* of x, with respect to α. Once the construction of the field is determined, the primitive element α is fixed. Then, an exponentiation function $expf$ can be defined to represent elements with the power value. For example, x is represented as:

$$x = \text{expf}(i) \qquad x \neq 0 \tag{1}$$

*This work was in part supported by NSF Grants CCR-TC-0208975 and ANI-0322615.

Similarly, a discrete logarithm function $logf$ can be defined as:

$$i = \text{logf}(x) \qquad x \neq 0 \tag{2}$$

Let y be another element in GF(2^n), $y = \alpha^j$. Then the multiplication $mul(x, y)$ of x and y can be represented as:

$$\text{mul}(x, y) = \alpha^i \alpha^j = \alpha^{i+j}$$

From the property of primitive element, $\alpha^{2^n-1} = 1$. Let $Q = 2^n - 1$, then $\alpha^Q = 1$. Thus, we have

$$\text{mul}(x, y) = \alpha^{i+j} = \alpha^{(i+j)\%Q} \tag{3}$$

where, % means *modular operation*.

With the definition of the exponentiation and the discrete logarithm functions, we can derive the multiplication function as

$$\text{mul}(x, y) = \text{expf}\Big(\big(\text{logf}(x) + \text{logf}(y)\big)\%Q\Big) \tag{4}$$

The precomputation approach is to compute all possible values of the $expf$ and $logf$ functions based on the primitive element and store them in a table. The exponentiation and discrete logarithm operations are just table look-ups thereafter. The size of the table equals to $Q - 1$, which is also the number of non-zero elements in the field. For example, GF(256) has table size of 255.

Therefore, the multiplication operation involves 3 table look-ups, 1 addition and 1 modular operation. Taking into account that an operator could be 0, the complete multiplication procedure is

```
procedure mul(x, y)
{
        if(x==0||y==0) return 0;
        return expf[(logf[x]+logf[y])%Q];
}
```

We refer this as the *basic* multiplication approach and try to improve it through the rest of this section.

2.1 Improvement Approach 1

In the basic multiplication approach, the modular operation takes significant amount of time more than table look-up and addition. Thus, improvement approach 1 is to replace the modular operation with more efficient operations.

For non-zero x and y, $logf[x]$ and $logf[y]$ are within range $[0, Q-1]$, here numbers 0, 1 $\cdots$, Q are handy representation of elements in GF(2^n). So $logf[x]+logf[y]$ is within range $[0, 2Q-2]$. We can divide this range into three parts with respect to "modulo Q".

$$(\text{logf}[x] + \text{logf}[y])\%Q = \begin{cases} (\text{logf}[x] + \text{logf}[y]) & (\text{logf}[x] + \text{logf}[y]) \in [0, Q-1] \\ 0 & (\text{logf}[x] + \text{logf}[y]) = Q \\ (\text{logf}[x] + \text{logf}[y])\&Q + 1 & (\text{logf}[x] + \text{logf}[y]) \in [Q+1, 2Q-2] \end{cases}$$

Combining case 1 and 3, $(logf[x] + logf[y])\%Q$ can be replaced by

$$(\text{logf}[x] + \text{logf}[y])\%Q = (\text{logf}[x] + \text{logf}[y])\&Q + (\text{logf}[x] + \text{logf}[y]) >> n \tag{5}$$

with one exception as $(logf[x] + logf[y])\% = Q$. Here, $>>$ is right shift operation. The exception can be easily handled by augmenting the exponentiation table $expf$, which original has Q values, with one more value. Let $expf[Q] = expf[0]$ and the modular operation can be eliminated.

$$\text{expf}\Big[\big(\text{logf}[x] + \text{logf}[y]\big)\%Q\Big] = \text{expf}\Big[(\text{logf}[x] + \text{logf}[y])\&Q + (\text{logf}[x] + \text{logf}[y]) >> n\Big] \tag{6}$$

And the complete multiplication procedure is:

```
procedure mul(x, y)
{
        if(x==0||y==0) return 0;
        return expf[(logf[x]+logf[y])&Q+(logf[x]+logf[y])>>n];
}
```

This approach uses one binary AND, one right shift and one addition operation to replace the original modular operation.

2.2 Improvement Approach 2

In the previous approach, one element is used to augment the exponentiation table. The augmentation idea is further explored in improvement approach 2, where another $Q-2$ elements are added into the exponentiation table. The lower part of the table in the range of $[0, Q]$ is already computed and the new part in the range of $[Q+1, 2Q-2]$ can be pre-computed as

$$\text{expf}[i] = \text{expf}[i\%Q] \qquad i \in [Q+1, 2Q-2] \tag{7}$$

With this augmented exponentiation table, the multiplication procedure is:

```
procedure mul(x, y)
{
        if(x==0||y==0) return 0;
        return expf[(logf[x]+logf[y])];
}
```

This approach eliminates modular operation at the cost of larger table size (almost twice), which, however could be pre-computed, and extra operations of approach 1 are also eliminated.

2.3 Improvement Approach 3

The **if** $\cdots$ **then** conditional branch in previous approaches has impact on the throughput of multiplication procedure. It is easy to understand that a lot of the time the operators x and y are non-zero, then hardware architecture employs branch prediction will have fairly close performance compared to pure operation without the branch. However, if x or y was zero, then there would be miss-prediction, which degrades the speed of multiplication operation. It is desirable to eliminate the conditional branch whenever possible. Further exploring the idea of augmenting table turns out to achieve this goal.

If we can map 0 to a *large* value in the discrete logarithm table and all *large* values back to 0 in the exponentiation table, then operation on 0 is unified into current table look-up and the conditional branch is completely eliminated. It is clear in approach 2 that $(logf[x] + logf[y]) \in [0, 2Q-2]$. We augment the discrete logarithm table by adding one more element $logf[0] = 2Q$. Notice, this augmenting takes the division operation into account, otherwise, it would be enough to let $logf[0] = 2Q-1$ and save 2 entries in the exponentiation table. Then, $(logf[x] + logf[y])$ is within a new range $[0, 4Q]$. Mapping all elements over $2Q-1$ of the exponentiation table back to 0 as

$$\text{expf}[i] = 0 \qquad i \in [2Q, 4Q] \tag{8}$$

In summary, the discrete logarithm and the exponentiation table are defined as

$$\text{expf}[i] = \begin{cases} x & i \in [0, Q-1], x = \alpha^i \\ \text{expf}[i\%Q] & i \in [Q, 2Q-1] \\ 0 & i \in [2Q, 4Q] \end{cases} \tag{9}$$

$$\text{logf}[x] = \begin{cases} 2Q & x = 0 \\ i & x \in [1, Q], x = \alpha^i \end{cases} \tag{10}$$

And the final multiplication procedure with the augmented discrete logarithm and exponentiation table is as following

```
procedure mul(x, y)
{
        return expf[(logf[x]+logf[y])];
}
```

Also the final division operation can be optimized as

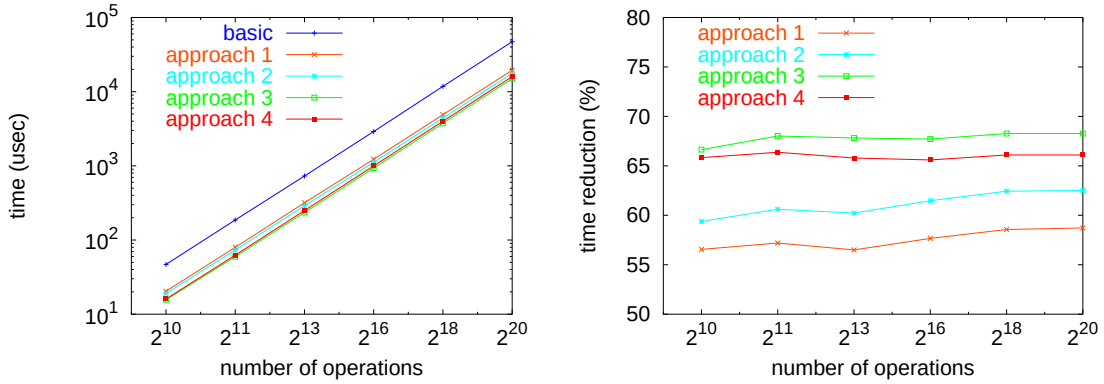
```
procedure div(x, y)
{
        return expf[(logf[x]+Q−logf[y])];
}
```

Notice, in the division operation, y can *not* be 0, which should be guaranteed before the operation is invoked.

2.4 Approach 4: 2-D Table Look-up

A natural attempt to push the augmentation idea to the extreme is to perform complete precomputation and store all results in two 2-dimensional tables, one for multiplication and one for division. Then the only operation is *single* 2-D table look-up.

However, this approach increases table size quadratically and thus uses *extra* larger memory, even for small finite field. For instance, the 2-D table size has 65536 entries for GF(256). Also, index to 2-D table is in general slower than 1-D table look-up. Large amount of repeated values in the table and 2-D index to it eventually degrades throughput, as we see from simulation results presented in Section 3. Therefore, there is limitation in trading memory usage for throughput.



(a) Execution Time

(b) Improvement over Basic Approach

Figure 1: **Effect of Improvement Approaches.** Each round contains one multiplication and one division operation over GF(256).

3 Performance Evaluation

3.1 Evaluation Method

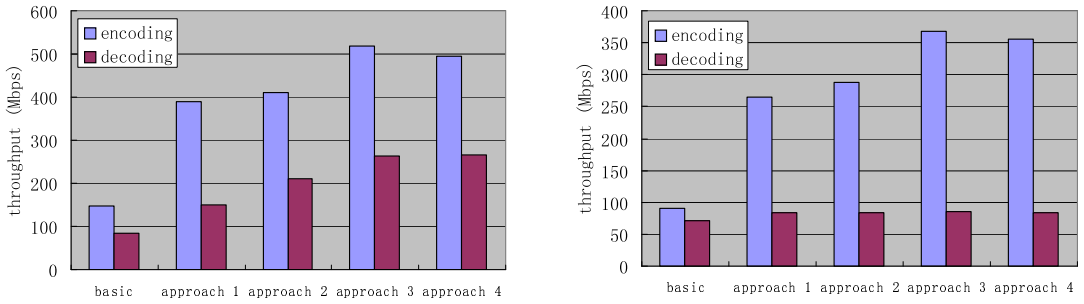
Evaluation is done by comparing the execution time of pure multiplication and the division operations and also the throughput of Reed-Solomon codes using different improvement approaches. These operations

are in general very fast, so we use clock cycle register built in modern computer system to count clock cycles during execution and get very fine granularity measurement. We also use the *K-best Measurement Scheme* to accommodate the disturbance of other activities during our measurement. The basic idea of K-best Measurement Scheme is to perform up to N measurements and see if the fastest K measurements are within relative factor ϵ. Then the fastest measurement is considered as a successful measurement [2]. We use $N = 20$, $K = 3$ and $\epsilon = 0.01$ in our simulation.

3.2 Performance Improvement of Multiplication and Division Operation

By varying the number of rounds from 2^{10} to 2^{20}, the execution time of pure multiplication and division over random value in GF(256) is measured on a P4 1.8GHz machine with 512M memory running Redhat Linux 9.0. Each round consists of one single multiplication and division. The result is shown in Figure 1(a). Improvement approach 1 reduces execution time over basic approach by about 56%, approach 2 reduces by about 60% and approach 3 reduces by about 67%.

Notice that approach 4 is slightly worse than approach 3, this shows the limitation in trading memory usage for throughput, as we discussed in the previous section.



(a) RS(255, 2) Throughput

(b) RS(255, 3) Throughput

Figure 2: Effect of Improvement Approaches on Reed-Solomon Codes

3.3 Performance Improvement of Reed-Solomon Codes

In this subsection, we evaluate the encoding/decoding performance of Reed-Solomon codes over GF(256) with the our improvement approaches of basic finite field operations In particular, RS(255, 2) and RS(255, 3) codes are investigated, which are of particular interests to us, not only because of their easiness in implementation, efficiency in throughput, but also because of their usefulness in *Multicast Key Distribution* [5] practice.

Figure 2 shows the performance improvement of RS(255, 2) and RS(255, 3) with our approaches. It is clear that approach 3 significantly increases the encoding throughput beyond 3 times of the basic approach. Again, approach 4 is slightly worse than approach 3.

4 Summary

In this paper, we explore the idea of precomputation and propose augmented table to facilitate finite field operations. Simulation results show our improvement approach reduces execution time of multiplication and division operation by about 67% and increaes encoding throughput of Reed-Solomon codes about 3 times. We also show that there is limitation in trading memory usage for throughput.

References

[1] R. E. Blahut, "Algebraic Codes for Data Transmission," 1st. ed., *Cambridge University Press*, 2002.

[2] R. E. Bryant, and D. R. O'Hallaron, "Computer Systems: A Programmer's Perspective," *Prentice Hall*, 2002.

[3] C. H. Lim, and P. J. Lee, "More Flexible Exponentiation with Precomputation," *Advances in Cryptology – Crypto'94*, 95-107, 1994.

[4] I. S. Reed and G. Solomon, "Polynomial Codes over Certain Finite Fields," *J. SIAM*, 8(10), 300-304, 1960.

[5] L. Xu, "Computation Efficient Multicast Key Distribution," *ISIT 2003*, Yokohama, Japan, Jun. 2003.