$O(m^{1+o(1)} \log m)$ Polynomial Modular Multiplication using Mersenne Transform and Subfields

Kittiphon Phalakarn

University of Waterloo, Canada
kphalakarn@uwaterloo.ca

Abstract. Polynomial modular multiplication in GF(p^m) can be speeded up by performing the multiplication in frequency domain. Two well-known transformation techniques are NTT and Mersenne transform. The first uses $O(m \log m)$ multiplications and additions over GF(p), while the second uses only $O(m)$ multiplications with $O(m^2)$ additions and bit-shifts. In this work, we present a new alternative approach which uses an idea of subfields with the Mersenne transform. The resulting algorithm uses $O(m^{1+o(1)})$ multiplications with $O(m^{1+o(1)} \log m)$ additions and bit-shifts. One advantage of this algorithm is that it is applicable to all finite fields, including ones to which NTT and Mersenne transform cannot be applied. We also give an example of a case when our proposed algorithm uses smaller number of operations comparing to other techniques.

1 Introduction

Polynomial modular multiplication, $c(x) = a(x) \cdot b(x) \bmod f(x)$, is an important operation of finite fields GF(p^m) and is required in various applications, such as cryptography [5] and error correcting codes [9]. The operation consists of two main steps: polynomial multiplication and polynomial modular reduction. When considering polynomials of degree less than m, a multiplication of two polynomials can be performed by the schoolbook method using $O(m^2)$ arithmetic operations, and a modular reduction can be performed using $O(m)$ arithmetic operations when the field generating polynomial $f(x)$ is fixed. However, doing polynomial modular multiplication using $O(m^2)$ operations is not optimal, and many works were proposed to improve this result. For example, the Karatsuba algorithm [3] performs the multiplication using $O(m^{\log_2 3})$ operations, and the Toom-Cook algorithm [4] reduces the number of operations to $O(m^{\log_3 5})$.

The number of operations can be reduced further when multiplications are performed in frequency domain. In order to do the multiplication in frequency domain, $d \geq 2m-1$ points of each polynomial $a(x)$ and $b(x)$ are required so that the result does not overflow, and a primitive d-th root of unity in GF(p) is needed. By using number theoretic transform (NTT) [6], the conversions between the time and frequency domains can be performed using $O(d \log d)$ operations, and the polynomial multiplication in frequency domain can be performed using only $O(d)$ operations. Thus, the total number of operations is $O(d \log d)$. The NTT performs best when d is a power of two, but it can still be applied when d is highly composite.

The other direction to speed up polynomial multiplication is to use Mersenne transform [7] where p is a Mersenne prime, i.e. $p = 2^n - 1$. It is known that $p = 2^n - 1$ is prime only if n is prime. The advantages of using a Mersenne prime as a modulus is that, for an integer $b \in \mathrm{GF}(p)$, the value $2b$ can be computed easily by one left cyclic shift of b, and $-b$ is b with all bits flipped. To take these advantages, it is preferred that 2 or -2 is a primitive d-th root of unity in $\mathrm{GF}(p)$, and all costly multiplications in $\mathrm{GF}(p)$ in NTT can then be converted to cyclic shifts, which can be considered free in hardware. However, the requirements that $p = 2^n - 1$, and 2 or -2 is a primitive d-th root of unity in $\mathrm{GF}(p)$ limit the choices of p and d. Some examples of p, d, and m which comply with the requirements are shown in Table 1. It can be proved that $d = n$ when using 2 as a primitive d-th root of unity, and $d = 2n$ when using -2.

Table 1. Some examples of p, d, and m used in Mersenne transform

n	$p = 2^n - 1$	order of 2 and -2	d	$m \leq \frac{d+1}{2}$
2	3	$2^2 \equiv 1 \pmod 3$	2	1
3	7	$2^3 \equiv 1 \pmod 7$	3	2
		$(-2)^6 \equiv 1 \pmod 7$	6	3
5	31	$2^5 \equiv 1 \pmod{31}$	5	3
		$(-2)^{10} \equiv 1 \pmod{31}$	10	5
7	127	$2^7 \equiv 1 \pmod{127}$	7	4
		$(-2)^{14} \equiv 1 \pmod{127}$	14	7

Since $d = n$ or $d = 2n$ and n is prime, d is not highly composite. Hence, the NTT does not perform well and results in $O(d^2)$ operations. Nevertheless, all $O(d^2)$ operations in Mersenne transform are additions and bit-shifts, while $O(d \log d)$ operations in NTT are a combination of additions and costly multiplications.

Another possibility of modular multiplication in frequency domain is to do both multiplication and modular reduction in frequency domain. Thus, no transformation between domains is required when the input polynomials are given in frequency domain. The DFT modular multiplication presented in [1] performs the Montgomery reduction in frequency domain considering Mersenne prime using $O(d^2)$ additions and bit-shifts, but with smaller coefficient comparing to the above Mersenne transform.

To the best of our knowledge, in order to do the polynomial modular multiplication in frequency domain with $d = O(m)$, one has to choose either NTT using $O(m \log m)$ multiplications over $\mathrm{GF}(p)$, or Mersenne transform using $O(m^2)$ operations with $O(m)$ multiplications. In this work, we propose a new polynomial modular multiplication algorithm in frequency domain considering Mersenne prime and subfields. It uses $O(m^{1+o(1)} \log m)$ additions and bit-shifts and $O(m^{1+o(1)})$ multiplications over $\mathrm{GF}(p)$. One advantage of this algorithm is that it is applicable to all finite fields $\mathrm{GF}(p^m)$, including ones to which NTT and Mersenne transform cannot be applied. We also give an example of a case when our proposed algorithm uses smaller number of operations comparing to other techniques.

2 Preliminaries

2.1 Number Theoretic Transform

The NTT computes the values of a polynomial $a(x) \in \mathrm{GF}(p^m)$ at d points. One requirement of this algorithm is that a primitive d-th root of unity in $\mathrm{GF}(p)$ must exist, hence $d \mid (p-1)$. This implies $2m-1 \le d \le p-1$ and $m \le \lfloor \frac{p}{2} \rfloor$. Let $a(x) = \sum_{i=0}^{d-1} a_i x^i$ where $a_i \in \mathrm{GF}(p)$, $d = d_1 \cdot d_2 \cdot \ldots \cdot d_k$, and r be a primitive d-th root of unity in $\mathrm{GF}(p)$, i.e. $r^d \equiv 1 \pmod p$. The NTT computes $a(r^\ell)$ for each $0 \le \ell \le d-1$ recursively as follows:

1. Let $d' = d/d_k$. We can write $a(x) = \sum_{i=0}^{d'-1} \sum_{j=0}^{d_k-1} (a_{d_k i+j})(x^{d_k i+j}) = \sum_{j=0}^{d_k-1} \left(\sum_{i=0}^{d'-1} (a_{d_k i+j})(x^{d_k})^i \right) x^j$.
2. Let $a^{(j)}(x) = \sum_{i=0}^{d'-1} (a_{d_k i+j})(x^{d_k})^i$. Since $r^{d' \cdot d_k} \equiv 1$, then $(r^{sd'+t})^{d_k} \equiv (r^{s'd'+t})^{d_k}$ for all $0 \le t \le d'-1$ and $0 \le s, s' \le d_k - 1$, i.e. $a^{(j)}(r^{sd'+t}) = a^{(j)}(r^{s'd'+t})$. Thus, to compute $a^{(j)}(r^\ell)$ for all $0 \le \ell \le d-1$, it is sufficient to compute only $a^{(j)}(r^{\ell'})$ for $0 \le \ell' \le d'-1$, and $a^{(j)}(r^\ell) = a^{(j)}(r^{(\ell \bmod d')})$.
3. Let $\widehat{a}^{(j)}(y) = \sum_{i=0}^{d'-1} (a_{d_k i+j}) y^i$ and $\widehat{r} = r^{d_k}$. Then, $a^{(j)}(r^{\ell'}) = \widehat{a}^{(j)}(\widehat{r}^{\ell'})$. It can be proved that $\widehat{r}$ is a primitive d'-root of unity in $\mathrm{GF}(p)$. Hence, $\widehat{a}^{(j)}(\widehat{r}^{\ell'})$ for $0 \le \ell' \le d'-1$ can be computed using NTT.
4. Compute $a(r^\ell) = \sum_{j=0}^{d_k-1} \widehat{a}^{(j)}(\widehat{r}^{(\ell \bmod d')}) \cdot (r^\ell)^j$ for all $0 \le \ell \le d-1$. Note that $(r^\ell)^j$ can be precomputed.

Analysis: Let $\mathcal{M}(d)$ and $\mathcal{A}(d)$ be the numbers of multiplications and additions for computing d points of a degree-$(d-1)$ polynomial. Step 3-4 give $\mathcal{M}(d) = d_k \cdot \mathcal{M}(d/d_k) + d \cdot d_k$ and $\mathcal{A}(d) = d_k \cdot \mathcal{A}(d/d_k) + d \cdot (d_k - 1)$. Suppose d is a q-smooth number, i.e. $d_i \le q$ for all $1 \le i \le k$, where q is a small integer. Thus,

$$\begin{aligned}
\mathcal{M}(d) &\le d_k \cdot \mathcal{M}(d/d_k) + dq \\
&\le d_k(d_{k-1} \cdot \mathcal{M}(d/(d_k d_{k-1})) + (d/d_k)q) + dq \quad = d_k d_{k-1} \cdot \mathcal{M}(d/(d_k d_{k-1})) + 2dq \\
&\le \cdots \\
&\le d_k d_{k-1} \cdots d_1 \cdot \mathcal{M}(1) + kdq \quad = kdq \quad \le (\log_2 d)dq \quad = O(d \log d).
\end{aligned}$$

Similarly, we have $\mathcal{A}(d) = O(d \log d)$. Note that $\mathcal{M}(d) = O(d^2)$ if $q = d$, e.g. d is a prime number.

Example 1. Consider a polynomial $a(x) = \sum_{i=0}^{d-1} a_i x^i$ over $\mathrm{GF}(31)$ where $d = 15 = 3 \cdot 5$. We choose $r = 9$ as a primitve 15th root of unity. The NTT can be applied to compute $a(r^\ell)$ for $0 \le \ell \le 14$ as follows:

1. Write $a(x) = \sum_{j=0}^{4} (a_j + a_{j+5}x^5 + a_{j+10}x^{10}) x^j$.
2. Let $a^{(j)}(x) = a_j + a_{j+5}x^5 + a_{j+10}x^{10}$. We have $a^{(j)}(r^{3s+\ell'}) = a^{(j)}(r^{\ell'})$ for $0 \le s \le 4$ and $0 \le \ell' \le 2$.
3. Let $\widehat{a}^{(j)}(y) = a_j + a_{j+5}y + a_{j+10}y^2$ and $\widehat{r} = r^5 \equiv 25$. Then, $a^{(j)}(r^{\ell'}) = \widehat{a}^{(j)}(\widehat{r}^{\ell'})$. We can see that $\widehat{r} = 25$ is a 3rd root of unity. So, we can use NTT to compute $\widehat{a}^{(j)}(\widehat{r}^{\ell'})$ for $0 \le \ell' \le 2$ and $0 \le j \le 4$.
4. Compute $a(r^\ell) = \sum_{j=0}^{4} \widehat{a}^{(j)}(\widehat{r}^{(\ell \bmod 3)}) \cdot (r^\ell)^j$ for all $0 \le \ell \le 14$. Note that $(r^\ell)^j \equiv r^{(\ell j \bmod 15)}$.

2.2 Mersenne Transform

The Mersenne tranform is another technique that transforms a polynomial between time and frequency domain. It is applicable when the characteristic p of the finite field $\mathrm{GF}(p^m)$ is a Mersenne prime, i.e. $p = 2^n - 1$. It is known that $p = 2^n - 1$ is prime only if n is prime, since $2^{uv} - 1$ is divisible by both $2^u - 1$ and $2^v - 1$. However, there exists a prime n such that $2^n - 1$ is not prime, e.g. $2^{11} - 1 = 23 \cdot 89$.

We first consider some properties of the field $\mathrm{GF}(p)$. Let $0 \leq b \leq 2^n - 2$ be a member of $\mathrm{GF}(p)$. Then, we can represent b as an n-bit integer $(b_{n-1}b_{n-2}\dots b_0)_2$. To compute $2b$, since $2^n \equiv 1 \pmod{p}$, we have

$$2b = 2 \cdot (b_{n-1}b_{n-2}\dots b_0)_2 = b_{n-1}2^n + (b_{n-2}b_{n-3}\dots b_0 0)_2 \equiv (b_{n-2}b_{n-3}\dots b_0 b_{n-1})_2.$$

And, to compute $-b$, since $0 \equiv p \pmod{p}$ and $p = 2^n - 1 = \sum_{i=0}^{n-1} 2^i$, we have

$$-b \equiv p - b = \sum_{i=0}^{n-1} 2^i - \sum_{i=0}^{n-1} b_i 2^i = \sum_{i=0}^{n-1} (1 - b_i)2^i = (\overline{b}_{n-1}\overline{b}_{n-2}\dots\overline{b}_0)_2$$

where $\overline{b}_i$ denotes a bit negation of b_i. In hardware implementation, the cyclic shifts and bit negations can be performed efficiently and hence considered as inexpensive operations. The Mersenne transform uses these properties of Mersenne primes to compute the values of $a(x)$ at d points as follows.

The Mersenne transform requires that $d = n$ or $d = 2n$. When $d = n$, we have 2 is a primitive d-th root of unity since $2^d \equiv 2^n \equiv 1 \pmod{p}$ and $d = n$ is prime, so it is not possible that the order of 2 is smaller than d. When $d = 2n$ and n is an odd prime, we have -2 is a primitive d-th root of unity since $(-2)^d \equiv (-2)^{2n} \equiv (-1)^{2n}(2^n)^2 \equiv 1 \pmod{p}$, $(-2)^n \equiv -1 \not\equiv 1 \pmod{p}$, and $(-2)^2 \equiv 2^2 \not\equiv 1 \pmod{p}$. For $n = 2$ and $p = 2^2 - 1 = 3$, a primitive $2n$-th root of unity does not exist. By the constraint $d \geq 2m - 1$, the range of m is limited to $m \leq \lfloor \frac{n+1}{2} \rfloor$ when $d = n$, and $m \leq \lfloor \frac{2n+1}{2} \rfloor = n$ when $d = 2n$.

For $a(x) = \sum_{i=0}^{d-1} a_i x^i$, the Mersenne transform computes $a(2^\ell) = \sum_{i=0}^{d-1} a_i 2^{\ell i}$ when $d = n$, or $a((-2)^\ell) = \sum_{i=0}^{d-1} a_i (-2)^{\ell i}$ when $d = 2n$, for $0 \leq \ell \leq d - 1$. The values of $a_i 2^{\ell i}$ and $a_i(-2)^{\ell i}$ can be easily computed using bit-shifts and negations. Thus, no multiplication over $\mathrm{GF}(p)$ is required. We have the number of bit-shifts is $(d-1)^2 = O(d^2)$, and the number of additions is $d(d-1) = O(d^2)$.

The Mersenne transform can be written as a matrix-vector multiplication as follows (for $d = n$):

$$\begin{pmatrix} a(1) \\ a(2) \\ a(2^2) \\ \vdots \\ a(2^{d-1}) \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & 2 & 2^2 & \cdots & 2^{d-1} \\ 1 & 2^2 & 2^4 & \cdots & 2^{2(d-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & 2^{d-1} & 2^{2(d-1)} & \cdots & 2^{(d-1)(d-1)} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_{d-1} \end{pmatrix}.$$

The transformation from frequency to time domain can be performed by multiplying the inverse matrix:

$$\begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_{d-1} \end{pmatrix} = (d^{-1}) \begin{pmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & 2^{-1} & 2^{-2} & \cdots & 2^{-(d-1)} \\ 1 & 2^{-2} & 2^{-4} & \cdots & 2^{-2(d-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & 2^{-(d-1)} & 2^{-2(d-1)} & \cdots & 2^{-(d-1)(d-1)} \end{pmatrix} \begin{pmatrix} a(1) \\ a(2) \\ a(2^2) \\ \vdots \\ a(2^{d-1}) \end{pmatrix}.$$

To compute $2^{-1}b$, we can replace $2^{-1} \equiv 2^{n-1} \pmod{p}$, or perform a right cyclic shift to b since

$$2^{-1}b = 2^{-1} \cdot (b_{n-1}b_{n-2} \dots b_0)_2 = (0\, b_{n-1} \dots b_2 b_1)_2 + 2^{-1}b_0 \equiv (b_0 b_{n-1} \dots b_2 b_1)_2.$$

Note that the reverse transformation requires d additional multiplications with the constant d^{-1}. Furthermore, the NTT is not suitable for finite fields over a Mersenne prime since $d = n$ or $d = 2n$ and n is prime, thus we have $q = d$ or $q = d/2$ and $\mathcal{M}(d) = O(d^2)$ when applying NTT.

2.3 Subfields

In the case that m of $\mathrm{GF}(p^m)$ is composite, i.e. $m = m_1 \cdot m_2$, we can consider the field $\mathrm{GF}(p^m)$ as $\mathrm{GF}((p^{m_1})^{m_2})$. We call $\mathrm{GF}(p^{m_1})$ as a subfield of $\mathrm{GF}(p^m)$. An element a in $\mathrm{GF}((p^{m_1})^{m_2})$ can be represented as $a = \sum_{i=0}^{m_2-1} a_i y^i$ where $a_i \in \mathrm{GF}(p^{m_1})$. Each element a_i can then be represented as $a_i = \sum_{j=0}^{m_1-1} a_{ij} x^j$ where $a_{ij} \in \mathrm{GF}(p)$. Hence, we can write an element in $\mathrm{GF}((p^{m_1})^{m_2})$ as $a(x, y) = \sum_{i=0}^{m_2-1} \sum_{j=0}^{m_1-1} a_{ij} x^j y^i$. We require an irreducible polynomial $f_1(x)$ of degree m_1 over $\mathrm{GF}(p)$ to construct $\mathrm{GF}(p^{m_1})$ and an irreducible polynomial $f_2(y)$ of degree m_2 over $\mathrm{GF}(p^{m_1})$ to construct $\mathrm{GF}((p^{m_1})^{m_2})$. This can be generalized to subfields of any dimension, e.g. $\mathrm{GF}(((p^{m_1})^{m_2})^{m_3})$ where $a(x, y, z) = \sum_{i=0}^{m_3-1} \sum_{j=0}^{m_2-1} \sum_{k=0}^{m_1-1} a_{ijk} x^k y^j z^i$.

Example 2. Consider a field $\mathrm{GF}(31^{12})$ as $\mathrm{GF}((31^3)^4)$. An element $a \in \mathrm{GF}((31^3)^4)$ can be represented as

$$a(x, y) = (a_{00}+a_{01}x+a_{02}x^2) + (a_{10}+a_{11}x+a_{12}x^2)y + (a_{20}+a_{21}x+a_{22}x^2)y^2 + (a_{30}+a_{31}x+a_{32}x^2)y^3.$$

We can also consider $\mathrm{GF}(31^{12})$ as $\mathrm{GF}(((31^2)^3)^2)$. An element $a \in \mathrm{GF}(((31^2)^3)^2)$ can be represented as

$$\begin{aligned} a(x, y, z) = {} & ((a_{000}+a_{001}x) + (a_{010}+a_{011}x)y + (a_{020}+a_{021}x)y^2) + \\ & ((a_{100}+a_{101}x) + (a_{110}+a_{111}x)y + (a_{120}+a_{121}x)y^2)z. \end{aligned}$$

Note that $\mathrm{GF}(31^{12})$, $\mathrm{GF}((31^3)^4)$, $\mathrm{GF}(((31^2)^3)^2)$ are isomorphic since they have the same number of elements.

3 Proposed Algorithm

We see in the previous section that the choices of m is limited. We have $m \leq \lfloor \frac{p}{2} \rfloor$ for NTT and $m \leq n \approx \log_2 p$ for Mersenne transform. In this section, we propose a new transformation that is applicable to all finite fields $\mathrm{GF}(p^m)$ by using an idea of subfields. We first focus on the case that $p = 2^n - 1$ is a Mersenne prime.

For a given m, we can find m' such that $m \leq m' < 2m$ and m' is an n-smooth number, i.e. all factors of m' are at most n. This m' always exists since there exists e such that $2^{e-1} < m \leq 2^e$ and $m \leq 2^e < 2m$. Thus, one choice of m' is 2^e, which is an n-smooth number. Let $m' = m_1 \cdot m_2 \cdot \ldots \cdot m_k$ where $m_i \leq n$ for $1 \leq i \leq k$. Using subfields $\mathrm{GF}(p^{m'}) = \mathrm{GF}((((p^{m_1})^{m_2}) \cdots)^{m_k})$, we can write $a(x) = \sum_{i=0}^{m-1} a_i x^i \in \mathrm{GF}(p^m)$ as

$$a(x_1, x_2, \ldots, x_k) = \sum_{i_k=0}^{m_k-1} \cdots \sum_{i_2=0}^{m_2-1} \sum_{i_1=0}^{m_1-1} a_{i_k,\ldots,i_2,i_1} x_1^{i_1} x_2^{i_2} \cdots x_k^{i_k}$$

by replacing $x_1 = x$, $x_2 = x^{m_1}$, $x_3 = x^{m_1 m_2}$, $\ldots$, $x_k = x^{m_1 m_2 \ldots m_{k-1}}$ in $a(x)$. We apply the same to $b(x)$.

When multiplying a and b, the degree of each x_i can be up to $2m_i - 2$. Thus, to prevent an overflow, the representations of a for the transformation have to be extended to

$$a(x_1, x_2, \ldots, x_k) = \sum_{i_k=0}^{d_k-1} \cdots \sum_{i_2=0}^{d_2-1} \sum_{i_1=0}^{d_1-1} a_{i_k,\ldots,i_2,i_1} x_1^{i_1} x_2^{i_2} \cdots x_k^{i_k}$$

where $d_i \geq 2m_i - 1$. The algorithm then transforms both a and b to frequency domain by computing the values at $d_1 d_2 \cdots d_k$ points. Since $m_i \leq n$, we can use $d_i = 2n$ and hence we can use $(-2)^{\ell_i}$ for $0 \leq \ell_i \leq d_i - 1$ as values for x_i. If $m_i \leq \lfloor \frac{n+1}{2} \rfloor$, we can also use $d_i = n$ and 2^{ℓ_i} as values for x_i. For simplicity, we assume that $d_i = n$ for all $1 \leq i \leq k$. So, all values at $d_1 d_2 \cdots d_k$ points are

$$a(2^{\ell_1}, 2^{\ell_2}, \ldots, 2^{\ell_k}) = \sum_{i_k=0}^{d_k-1} \cdots \sum_{i_2=0}^{d_2-1} \sum_{i_1=0}^{d_1-1} a_{i_k,\ldots,i_2,i_1} 2^{\ell_1 i_1 + \ell_2 i_2 + \ldots + \ell_k i_k}, \qquad 0 \leq \ell_i \leq d_i - 1. \tag{$*$}$$

Next, in frequency domain, we compute pairwise-multiplications of a and b, called c, of $d_1 d_2 \cdots d_k$ points. The reverse transformation is then applied to obtain all coefficients $c_{i_k,\ldots,i_2,i_1}$ of c, and we can obtain $c(x) = \sum_{i=0}^{2m-2} c_i x^i$ by replacing $x_1 = x$, $x_2 = x^{m_1}$, $x_3 = x^{m_1 m_2}$, $\ldots$, $x_k = x^{m_1 m_2 \ldots m_{k-1}}$ and adding the coefficients of each x^i. Finally, the modular reduction by the field generating irreducible polynomial $f(x)$ is applied to obtain the final result $c(x) = a(x) \cdot b(x) \bmod f(x)$.

The equation $(*)$ can also be written as a matrix multiplication. For $k = 2$, we can write a_{i_2,i_1} as

$$\begin{pmatrix} a_{0,0} & a_{0,1} & \cdots & a_{0,d_1-1} \\ a_{1,0} & a_{1,1} & \cdots & a_{1,d_1-1} \\ \vdots & \vdots & \ddots & \vdots \\ a_{d_2-1,0} & a_{d_2-1,1} & \cdots & a_{d_2-1,d_1-1} \end{pmatrix}.$$

From $a(x_1, x_2) = \sum_{i_2=0}^{d_2-1} \sum_{i_1=0}^{d_1-1} a_{i_2,i_1} x_1^{i_1} x_2^{i_2} = \sum_{i_2=0}^{d_2-1} \left(\sum_{i_1=0}^{d_1-1} a_{i_2,i_1} x_1^{i_1} \right) x_2^{i_2}$, let $A_{i_2}(x_1) = \sum_{i_1=0}^{d_1-1} a_{i_2,i_1} x_1^{i_1}$. The values at $d_1 d_2$ points can be computed by

$$A = \begin{pmatrix} A_0(1) & A_0(2) & \cdots & A_0(2^{d_1-1}) \\ A_1(1) & A_1(2) & \cdots & A_1(2^{d_1-1}) \\ \vdots & \vdots & \ddots & \vdots \\ A_{d_2-1}(1) & A_{d_2-1}(2) & \cdots & A_{d_2-1}(2^{d_1-1}) \end{pmatrix} = \begin{pmatrix} a_{0,0} & a_{0,1} & \cdots & a_{0,d_1-1} \\ a_{1,0} & a_{1,1} & \cdots & a_{1,d_1-1} \\ \vdots & \vdots & \ddots & \vdots \\ a_{d_2-1,0} & a_{d_2-1,1} & \cdots & a_{d_2-1,d_1-1} \end{pmatrix} \begin{pmatrix} 1 & 1 & \cdots & 1 \\ 1 & 2 & \cdots & 2^{d_1-1} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & 2^{d_1-1} & \cdots & 2^{(d_1-1)(d_1-1)} \end{pmatrix}$$

$$\begin{pmatrix} a(1,1) & a(2,1) & \cdots & a(2^{d_1-1},1) \\ a(1,2) & a(2,2) & \cdots & a(2^{d_1-1},2) \\ \vdots & \vdots & \ddots & \vdots \\ a(1,2^{d_2-1}) & a(2,2^{d_2-1}) & \cdots & a(2^{d_1-1},2^{d_2-1}) \end{pmatrix} = \begin{pmatrix} 1 & 1 & \cdots & 1 \\ 1 & 2 & \cdots & 2^{d_2-1} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & 2^{d_2-1} & \cdots & 2^{(d_2-1)(d_2-1)} \end{pmatrix} \begin{pmatrix} A_0(1) & A_0(2) & \cdots & A_0(2^{d_1-1}) \\ A_1(1) & A_1(2) & \cdots & A_1(2^{d_1-1}) \\ \vdots & \vdots & \ddots & \vdots \\ A_{d_2-1}(1) & A_{d_2-1}(2) & \cdots & A_{d_2-1}(2^{d_1-1}) \end{pmatrix}.$$

Note that each column of the matrix A corresponds to each value of $x_1 \in \{1, 2, \ldots, 2^{d_1-1}\}$.

For a more complicated case of $k = 3$, we can write a_{i_3,i_2,i_1} with d_3 matrices as

$$\begin{pmatrix} a_{0,0,0} & a_{0,0,1} & \cdots & a_{0,0,d_1-1} \\ a_{0,1,0} & a_{0,1,1} & \cdots & a_{0,1,d_1-1} \\ \vdots & \vdots & \ddots & \vdots \\ a_{0,d_2-1,0} & a_{0,d_2-1,1} & \cdots & a_{0,d_2-1,d_1-1} \end{pmatrix}, \quad \cdots, \quad \begin{pmatrix} a_{d_3-1,0,0} & a_{d_3-1,0,1} & \cdots & a_{d_3-1,0,d_1-1} \\ a_{d_3-1,1,0} & a_{d_3-1,1,1} & \cdots & a_{d_3-1,1,d_1-1} \\ \vdots & \vdots & \ddots & \vdots \\ a_{d_3-1,d_2-1,0} & a_{d_3-1,d_2-1,1} & \cdots & a_{d_3-1,d_2-1,d_1-1} \end{pmatrix}.$$

From $a(x_1, x_2, x_3) = \sum_{i_3=0}^{d_3-1} \sum_{i_2=0}^{d_2-1} \sum_{i_1=0}^{d_1-1} a_{i_3,i_2,i_1} x_1^{i_1} x_2^{i_2} x_3^{i_3} = \sum_{i_3=0}^{d_3-1} \left(\sum_{i_2=0}^{d_2-1} \sum_{i_1=0}^{d_1-1} a_{i_3,i_2,i_1} x_1^{i_1} x_2^{i_2} \right) x_3^{i_3}$, let $A_{i_3}(x_1, x_2) = \sum_{i_2=0}^{d_2-1} \sum_{i_1=0}^{d_1-1} a_{i_3,i_2,i_1} x_1^{i_1} x_2^{i_2}$. The matrices of $A_{i_3}(x_1, x_2)$ for $0 \le i_3 \le d_3 - 1$ can be computed using the same approach as computing $a(x_1, x_2)$ above. For example,

$$\begin{pmatrix} a_{0,0,0} & a_{0,0,1} & \cdots & a_{0,0,d_1-1} \\ a_{0,1,0} & a_{0,1,1} & \cdots & a_{0,1,d_1-1} \\ \vdots & \vdots & \ddots & \vdots \\ a_{0,d_2-1,0} & a_{0,d_2-1,1} & \cdots & a_{0,d_2-1,d_1-1} \end{pmatrix} \Longrightarrow \begin{pmatrix} A_0(1,1) & A_0(2,1) & \cdots & A_0(2^{d_1-1},1) \\ A_0(1,2) & A_0(2,2) & \cdots & A_0(2^{d_1-1},2) \\ \vdots & \vdots & \ddots & \vdots \\ A_0(1,2^{d_2-1}) & A_0(2,2^{d_2-1}) & \cdots & A_0(2^{d_1-1},2^{d_2-1}) \end{pmatrix}$$

$$\begin{pmatrix} a_{1,0,0} & a_{1,0,1} & \cdots & a_{1,0,d_1-1} \\ a_{1,1,0} & a_{1,1,1} & \cdots & a_{1,1,d_1-1} \\ \vdots & \vdots & \ddots & \vdots \\ a_{1,d_2-1,0} & a_{1,d_2-1,1} & \cdots & a_{1,d_2-1,d_1-1} \end{pmatrix} \Longrightarrow \begin{pmatrix} A_1(1,1) & A_1(2,1) & \cdots & A_1(2^{d_1-1},1) \\ A_1(1,2) & A_1(2,2) & \cdots & A_1(2^{d_1-1},2) \\ \vdots & \vdots & \ddots & \vdots \\ A_1(1,2^{d_2-1}) & A_1(2,2^{d_2-1}) & \cdots & A_1(2^{d_1-1},2^{d_2-1}) \end{pmatrix}$$

To compute $a(x_1, x_2, x_3) = \sum_{i_3=0}^{d_3-1} A_{i_3}(x_1, x_2) \cdot x_3^{i_3}$ using matrix multiplication, we require $A_0(x_1, x_2)$, $A_1(x_1, x_2)$, $\ldots$, $A_{d_3-1}(x_1, x_2)$ to be in the same matrix. This can be done by combining the corresponding columns of the $A_{i_3}(x_1, x_2)$ matrices. For example, the result of combining the first columns is

$$\begin{pmatrix} A_0(1,1) & A_1(1,1) & \cdots & A_{d_3-1}(1,1) \\ A_0(1,2) & A_1(1,2) & \cdots & A_{d_3-1}(1,2) \\ \vdots & \vdots & \ddots & \vdots \\ A_0(1,2^{d_2-1}) & A_1(1,2^{d_2-1}) & \cdots & A_{d_3-1}(1,2^{d_2-1}) \end{pmatrix}.$$

Note that each row of the above matrix now corresponds to the same value of (x_1, x_2). The above matrix is for $x_1 = 1$. There are d_1 such matrices for each value of x_1.

Finally, the values at $d_1 d_2 d_3$ points of $a(x_1, x_2, x_3)$ can be computed as (following is for $x_1 = 1$)

$$\begin{pmatrix} a(1,1,1) & a(1,1,2) & \cdots & a(1,1,2^{d_3-1}) \\ a(1,2,1) & a(1,2,2) & \cdots & a(1,2,2^{d_3-1}) \\ \vdots & \vdots & \ddots & \vdots \\ a(1,2^{d_2-1},1) & a(1,2^{d_2-1},2) & \cdots & a(1,2^{d_2-1},2^{d_3-1}) \end{pmatrix}$$

$$= \begin{pmatrix} A_0(1,1) & A_1(1,1) & \cdots & A_{d_3-1}(1,1) \\ A_0(1,2) & A_1(1,2) & \cdots & A_{d_3-1}(1,2) \\ \vdots & \vdots & \ddots & \vdots \\ A_0(1,2^{d_2-1}) & A_1(1,2^{d_2-1}) & \cdots & A_{d_3-1}(1,2^{d_2-1}) \end{pmatrix} \begin{pmatrix} 1 & 1 & \cdots & 1 \\ 1 & 2 & \cdots & 2^{d_3-1} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & 2^{d_3-1} & \cdots & 2^{(d_3-1)(d_3-1)} \end{pmatrix}.$$

This process can be generalized to subfields of any dimension: first compute the values of subfields with $k-1$ dimensions using matrix multiplication, then rearrange the matrices so that each row corresponds to the same value of $(x_1, x_2, \ldots, x_{k-1})$, and do one more matrix multiplication for each matrix. Note also that this process is invertible since each matrix multiplication is invertible. For the reverse transformation, we can multiply the constant $(d^{-1})^k$ to each entry once after all matrix multiplication.

We end this section by mentioning the case when p is not a Mersenne prime. Instead of using Mersenne transform in each step, we can apply NTT using d_i-th root of unity in $\mathrm{GF}(p)$ for each $1 \leq i \leq k$. It is always possible to find m' such that there exist d_i for each m_i satisfying $d_i \geq 2m_i - 1$ and $d_i \mid (p-1)$, e.g. $m' = 2^e$, $m_i = 2$, and $d_i = p - 1 \geq 2 \cdot 2 - 1$ which is true for $p > 3$.

4 Example

We give one example of the whole polynomial modular multiplication using our proposed algorithm.

Example 3. Consider $c(x) = a(x) \cdot b(x) \bmod f(x)$ in $\mathrm{GF}(7^5)$ where $f(x) = x^5 + x + 3$ and

$$\begin{aligned} a(x) &= 4 + 5x + x^2 + 3x^4 \\ b(x) &= 2 + 6x^2 + 2x^3 + 5x^4. \end{aligned}$$

This gives $p = 7$, $n = 3$, and $m = 5$. First, we choose $m' = 6 = 3 \cdot 2$ so that $m \leq m' < 2m$ and m' is 3-smooth. We have $m_1 = 3$ and $m_2 = 2$. By replacing $y = x^3$, we can represent a and b as

$$\begin{aligned} a(x,y) &= (4 + 5x + 1x^2) + (0 + 3x + 0x^2)y \\ b(x,y) &= (2 + 0x + 6x^2) + (2 + 5x + 0x^2)y. \end{aligned}$$

In order to apply Mersenne transform to each dimension, we select $d_1 = 2n = 6$ and $d_2 = n = 3$. Thus, the coefficients of a and b can be represented in a matrix form as follows.

$$a: \begin{pmatrix} 4 & 5 & 1 & 0 & 0 & 0 \\ 0 & 3 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \qquad b: \begin{pmatrix} 2 & 0 & 6 & 0 & 0 & 0 \\ 2 & 5 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Next, we transform a and b to frequency domain by computing values at 18 points:

$$\begin{pmatrix} a(1,1) & a(-2,1) & \cdots & a((-2)^5,1) \\ a(1,2) & a(-2,2) & \cdots & a((-2)^5,2) \\ a(1,2^2) & a(-2,2^2) & \cdots & a((-2)^5,2^2) \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 \\ 1 & 2 & 2^2 \\ 1 & 2^2 & 2^4 \end{pmatrix} \begin{pmatrix} 4 & 5 & 1 & 0 & 0 & 0 \\ 0 & 3 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & -2 & (-2)^2 & \cdots & (-2)^5 \\ 1 & (-2)^2 & (-2)^4 & \cdots & (-2)^{10} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & (-2)^5 & (-2)^{10} & \cdots & (-2)^{25} \end{pmatrix}$$

$$= \begin{pmatrix} 6 & 6 & 3 & 4 & 3 & 2 \\ 2 & 0 & 1 & 1 & 2 & 4 \\ 1 & 2 & 4 & 2 & 0 & 1 \end{pmatrix}$$

$$\begin{pmatrix} b(1,1) & b(-2,1) & \cdots & b((-2)^5,1) \\ b(1,2) & b(-2,2) & \cdots & b((-2)^5,2) \\ b(1,2^2) & b(-2,2^2) & \cdots & b((-2)^5,2^2) \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 \\ 1 & 2 & 2^2 \\ 1 & 2^2 & 2^4 \end{pmatrix} \begin{pmatrix} 2 & 0 & 6 & 0 & 0 & 0 \\ 2 & 5 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & -2 & (-2)^2 & \cdots & (-2)^5 \\ 1 & (-2)^2 & (-2)^4 & \cdots & (-2)^{10} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & (-2)^5 & (-2)^{10} & \cdots & (-2)^{25} \end{pmatrix}$$

$$= \begin{pmatrix} 1 & 4 & 1 & 5 & 3 & 3 \\ 1 & 3 & 2 & 2 & 1 & 6 \\ 1 & 1 & 4 & 3 & 4 & 5 \end{pmatrix}.$$

Then, $c = a \cdot b$ in frequency domain can be obtained by pairwise-multiplications

$$\begin{pmatrix} c(1,1) & c(-2,1) & \cdots & c((-2)^5,1) \\ c(1,2) & c(-2,2) & \cdots & c((-2)^5,2) \\ c(1,2^2) & c(-2,2^2) & \cdots & c((-2)^5,2^2) \end{pmatrix} = \begin{pmatrix} 6{\cdot}1 & 6{\cdot}4 & 3{\cdot}1 & 4{\cdot}5 & 3{\cdot}3 & 2{\cdot}3 \\ 2{\cdot}1 & 0{\cdot}3 & 1{\cdot}2 & 1{\cdot}2 & 2{\cdot}1 & 4{\cdot}6 \\ 1{\cdot}1 & 2{\cdot}1 & 4{\cdot}4 & 2{\cdot}3 & 0{\cdot}4 & 1{\cdot}5 \end{pmatrix} = \begin{pmatrix} 6 & 3 & 3 & 6 & 2 & 6 \\ 2 & 0 & 2 & 2 & 2 & 3 \\ 1 & 2 & 2 & 6 & 0 & 5 \end{pmatrix}$$

and the coefficients of c can be computed by the inverse transformation

$$\begin{pmatrix} c_{00} & c_{01} & \cdots & c_{05} \\ c_{10} & c_{11} & \cdots & c_{15} \\ c_{20} & c_{21} & \cdots & c_{25} \end{pmatrix} = (3^{-1}) \begin{pmatrix} 1 & 1 & 1 \\ 1 & 2^{-1} & 2^{-2} \\ 1 & 2^{-2} & 2^{-4} \end{pmatrix} \begin{pmatrix} 6 & 3 & 3 & 6 & 2 & 6 \\ 2 & 0 & 2 & 2 & 2 & 3 \\ 1 & 2 & 2 & 6 & 0 & 5 \end{pmatrix} (6^{-1}) \begin{pmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & (-2)^{-1} & (-2)^{-2} & \cdots & (-2)^{-5} \\ 1 & (-2)^{-2} & (-2)^{-4} & \cdots & (-2)^{-10} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & (-2)^{-5} & (-2)^{-10} & \cdots & (-2)^{-25} \end{pmatrix}$$

$$= \begin{pmatrix} 1 & 3 & 5 & 2 & 6 & 0 \\ 1 & 1 & 6 & 2 & 0 & 0 \\ 0 & 6 & 1 & 0 & 0 & 0 \end{pmatrix}$$

$$c(x,y) = (1{+}3x{+}5x^2{+}2x^3{+}6x^4) + (1{+}x{+}6x^2{+}2x^3)y + (6x{+}x^2)y^2.$$

By replacing $y = x^3$, we obtain $c(x) = 1 + 3x + 5x^2 + 3x^3 + 6x^5 + 2x^6 + 6x^7 + x^8 = a(x) \cdot b(x)$.

Finally, since $x^5 \equiv 6x{+}4 \pmod{f(x)}$, thus $x^6 \equiv 6x^2{+}4x$, $x^7 \equiv 6x^3{+}4x^2$, and $x^8 \equiv 6x^4{+}4x^3 \pmod{f(x)}$. Therefore, we have

$$\begin{aligned} c(x) \bmod f(x) &= 1 + 3x + 5x^2 + 3x^3 + 6(6x+4) + 2(6x^2+4x) + 6(6x^3+4x^2) + (6x^4+4x^3) \\ &= 4 + 5x + 6x^2 + x^3 + 6x^4. \end{aligned}$$

Note that $\mathrm{GF}(7^5)$ is not applicable to the standard NTT and Mersenne transform since $m > \lfloor \frac{p}{2} \rfloor$ and $m > n$.

5 Analysis

In this section, we analyse the number of multiplications, additions, and bit-shifts in GF(p) of implementations of our proposed algorithm. We express these numbers asymptotically using bit-O notation.

5.1 Analysis of the Transformation

We first consider the straightforward implementation following the equation $(*)$. Let $D = d_1 d_2 \cdots d_k$. The equation $(*)$ computes values at D points, and the value at each point is the sum of D values. Thus, the number of additions is $D(D-1) = O(D^2)$ and the number of bit shifts is $D^2 = O(D^2)$. This implementation is not efficient as some computations are performed several times. For example, the innermost summation $\sum_{i_1=0}^{d_1-1} a_{i_k,\ldots,i_2,i_1} 2^{\ell_1 i_1}$ depends only on the value of $x_1 = 2^{\ell_1}$, so it can be computed just once for each x_1. This issue can be solved by using the matrix multiplication.

The matrix multiplication approach recursively computes the values of subfields with $k-1$ dimensions, rearranges the matrices, and do one more matrix multiplication for each matrix. Let $\mathcal{A}(k)$ and $\mathcal{S}(k)$ be the number of additions and bit-shifts for computing the values of subfields with k dimensions. The algorithm first uses $\mathcal{A}(k-1)$ additions and $\mathcal{S}(k-1)$ bit-shifts. Then, the matrices are rearranged. Without loss of generality, suppose the values of each matrices correspond to the same $(x_1, x_2, \ldots, x_{k-2})$, each row of the matrix corresponds to the same x_{k-1}, and there are d_k columns. Thus, there are $d_1 d_2 \cdots d_{k-2}$ matrices of dimension $d_{k-1} \times d_k$. The last matrix multiplication is performed on two matrices of dimension $d_{k-1} \times d_k$ and $d_k \times d_k$, where the latter consists of powers of 2 or -2.

The naive schoolbook multiplication of two matrices of dimension $m \times n$ and $n \times n$ takes mn^2 bit-shifts and $mn(n-1)$ additions. In the case that both matrices are $n \times n$, there exist more efficient algorithms which use less than $O(n^3)$ operations. For example, the Strassen algorithm [8] uses $O(n^{log_2 7}) \approx O(n^{2.807})$ operations, and the Coppersmith–Winograd algorithm [2] uses $O(n^{2.376})$ operations. In our setting, it is possible to have $d_1 = d_2 = \ldots = d_k$, hence all matrix multiplications are performed on square matrices. However, these algorithms manipulate matrix entries which result in numbers that are not a power of 2 or -2 when multiplications over GF(p) take place. Thus, these sub-cubic algorithms cannot be applied if we wish to take advantages of Mersenne primes. Nevertheless, an analysis on the case that $d_1 = d_2 = \ldots = d_k$ using sub-cubic matrix multiplication algorithms will be presented.

Therefore, from the above description we have

$$\begin{aligned} \mathcal{S}(k) &= \mathcal{S}(k-1) + (d_1 d_2 \cdots d_{k-2})(d_{k-1} d_k^2) \quad &= \mathcal{S}(k-1) + D \cdot d_k \\ &= \mathcal{S}(k-2) + D \cdot d_{k-1} + D \cdot d_k \quad = \quad \ldots \quad &= D \left(\sum_{i=1}^{k} d_i \right). \end{aligned}$$

Example 4. Referring to the case of $k = 2$ in Section 3, the first matrix multiplication is performed on the matrices of dimension $d_2 \times d_1$ and $d_1 \times d_1$ and the second multiplication is on the matrices of dimension $d_2 \times d_2$ and $d_2 \times d_1$. Thus, $\mathcal{S}(2) = d_2 d_1 d_1 + d_2 d_2 d_1 = D(d_1 + d_2)$.

For $k = 3$, we first compute the values of subfields with 2 dimensions, i.e. the multiplications of $d_2 \times d_1$ and $d_1 \times d_1$ matrices, and $d_2 \times d_2$ and $d_2 \times d_1$ matrices, for all d_3 matrices. This takes $d_3(d_2 d_1 d_1 + d_2 d_2 d_1)$ operations. After the rearrange, we perform multiplications of $d_2 \times d_3$ and $d_3 \times d_3$ matrices, for all d_1 matrices. This takes $d_1(d_2 d_3 d_3)$ operations. Hence, $\mathcal{S}(3) = d_3 d_2 d_1 d_1 + d_3 d_2 d_2 d_1 + d_1 d_2 d_3 d_3 = D(d_1 + d_2 + d_3)$.

Assume that $m' \approx 2m$, $m_i \approx n$, $d_i \approx 2m_i$, and $n > 2$ is a constant. Thus, $k \approx \log_n m'$ and we have

$$\begin{aligned} D &= d_1 d_2 \cdots d_k \approx (2m_1)(2m_2) \cdots (2m_k) = 2^k m' \\ &\approx 2^{\log_n m'} m' = m'^{\log_n 2} m' = m'^{(1+\log_n 2)} \approx (2m)^{(1+\log_n 2)} \\ &= O(m^{1+o(1)}) \end{aligned}$$

$$\begin{aligned} \mathcal{S}(k) &= D(d_1 + \ldots + d_k) \approx D(2m_1 + \ldots + 2m_k) \approx D(2n + \ldots + 2n) = 2nDk \\ &\approx 2n(2m)^{(1+\log_n 2)}(\log_n m') \approx 2n(2m)^{(1+\log_n 2)}(\log_n 2m) \\ &= 2^{(2+\log_n 2)} \cdot n \cdot m^{(1+\log_n 2)} \cdot (\log_n m + \log_n 2) \\ &= O(m^{1+o(1)} \log m). \end{aligned}$$

Similarly, $\mathcal{A}(k) = O(m^{1+o(1)} \log m)$. Note that the number of points computed, i.e. D, is not linear in m because the size of each subfield is doubled: $d_i \approx 2m_i$.

We now suppose that the sub-cubic matrix multiplication algorithms can be applied. Let $d_1 = d_2 = \ldots = d_k$, $D = d_1^k$, and the number of operations used by the algorithm be $O(n^{3-\epsilon})$ where $0 \leq \epsilon < 1$. For instance, $\epsilon = 0$ for schoolbook method, $\epsilon \approx 0.193$ for the Strassen algorithm, and $\epsilon \approx 0.624$ for the Coppersmith–Winograd algorithm. Since $d_i \approx 2m_i$ and $d_1 = \ldots = d_k$, then $m_1 \approx \ldots \approx m_k \approx (m')^{1/k} \approx (2m)^{1/k}$, and

$$\begin{aligned} \mathcal{S}(k) &= \mathcal{S}(k-1) + (d_1 d_2 \cdots d_{k-2})(d_1^{3-\epsilon}) = \mathcal{S}(k-1) + D d_1^{1-\epsilon} = kD d_1^{1-\epsilon} \\ &\approx (\log_n m')(2m)^{(1+\log_n 2)}(2m_1)^{1-\epsilon} \\ &\approx (\log_n 2m)(2m)^{(1+\log_n 2)}(2(2m)^{1/k})^{1-\epsilon} \\ &= 2^{(2+\log_n 2-\epsilon+(1-\epsilon)/k)} \cdot m^{(1+\log_n 2+(1-\epsilon)/k)} \cdot (\log_n m + \log_n 2) \\ &= O(m^{1+o(1)} \log m). \end{aligned}$$

Hence, sub-cubic algorithms do not asymptotically improve the number of operations.

We mention at the end of Section 3 that our proposed algorithms can be used with NTT. After the rearrange, we can perform NTT on each row of the matrix instead of doing matrix multiplications. In this case, the number of multiplications is

$$\begin{aligned}\mathcal{M}(k) &\approx \mathcal{M}(k-1) + (d_1 d_2 \cdots d_{k-2})(d_{k-1} \cdot d_k \log d_k) \\ &= \mathcal{M}(k-1) + D \log d_k \\ &= D(\log d_1 + \ldots + \log d_k) = D \log D \\ &= O(m^{1+o(1)} \log m).\end{aligned}$$

Similarly, $\mathcal{A}(k) = O(m^{1+o(1)} \log m)$. Note that even the numbers of operations are no better than the standard NTT, this proposed algorithm works with any finite field GF(p^m).

5.2 Analysis of the Remaining Steps

After both polynomials are transformed to frequency domain, these four following steps are performed for computing modular multiplication:

1. Pairwise-multiplication of D points: This takes $D = O(m^{1+o(1)})$ multiplications over GF(p).
2. Reverse transformation to time domain: The reverse transformation takes $O(m^{1+o(1)} \log m)$ additions and bit-shifts. The constant d_i^{-1} can be accumulated and multiplied once to D entries of the matrix, hence taking $D = O(m^{1+o(1)})$ multiplications.
3. Replacing x_i with $x^{m_1 \cdots m_{i-1}}$: This step converts the representation in subfields $c(x_1, \ldots, x_k)$ to the entire field $c(x)$. As there are D entries, the number of additions required is at most $D = O(m^{1+o(1)})$.
4. Modular reduction: Assuming $f(x)$ is fixed and has low weight, the reduction takes $O(m)$ additions.

Therefore, the proposed polynomial modular multiplication algorithm takes $O(m^{1+o(1)} \log m)$ additions and bitshifts and $O(m^{1+o(1)})$ multiplications over GF(p). This can be summarized as in Table 2.

Table 2. The number of operations over GF(p) used in the proposed polynomial modular multiplication algorithm

Step	**# multiplications**	**# additions**	**# bit-shifts**
1. Forward transformation	-	$O(m^{1+o(1)} \log m)$	$O(m^{1+o(1)} \log m)$
2. Pairwise-multiplication	$O(m^{1+o(1)})$	-	-
3. Reverse transformation	$O(m^{1+o(1)})$	$O(m^{1+o(1)} \log m)$	$O(m^{1+o(1)} \log m)$
4. Convert to the entire field	-	$O(m^{1+o(1)})$	-
5. Modular reduction	-	$O(m)$	-
Total	$O(m^{1+o(1)})$	$O(m^{1+o(1)} \log m)$	$O(m^{1+o(1)} \log m)$

5.3 Comparing to Previous Works

We compare the performances of all algorithms mentioned in Section 1 as shown in Table 3, which summarizes the number of operations of each type over GF(p) of algorithms in literature and the proposed algorithm.

Table 3. The number of operations over GF(p) used in polynomial modular multiplication algorithms

Algorithms	# multiplications	# additions	# bit-shifts
Schoolbook	$O(m^2)$	$O(m^2)$	-
Karatsuba [3]	$O(m^{\log_2 3})$	$O(m^{\log_2 3})$	-
Toom-Cook [4]	$O(m^{\log_3 5})$	$O(m^{\log_3 5})$	-
NTT [6]	$O(m \log m)$	$O(m \log m)$	-
Mersenne transform [7]	$O(m)$	$O(m^2)$	$O(m^2)$
DFT modular multiplication [1]	$O(m)$	$O(m^2)$	$O(m^2)$
Proposed	$O(m^{1+o(1)})$	$O(m^{1+o(1)} \log m)$	$O(m^{1+o(1)} \log m)$

From the table, it can be seen that the proposed algorithm is slightly better than the standard Mersenne transform as the overall complexity is better. This comes with an overhead in a larger number of multiplications used by the algorithm. However, when comparing with NTT, the performance of the proposed algorithm is inferior. Nevertheless, it is possible that our proposed algorithm performs better in some cases. One example is given below.

Example 5. We apply NTT, Mersenne transform, and the proposed algorithm to the finite field GF(31^9).

When using NTT, we require $d \geq 2 \cdot 9 - 1 = 17$ and $d \mid 30$. The only choice is $d = 30 = 2 \cdot 3 \cdot 5$. Hence, the NTT uses approximately $30(2+3+5) = 300$ multiplications and $30(1+2+4) = 210$ additions. The pairwise-multiplication uses 30 multiplications. Hence, the total number of multiplications is $2 \cdot 300 + 30 = 630$ and that of additions is $2 \cdot 210 = 420$.

For the Mersenne transform, it is not applicable because $m = 9 > n = 5$.

When considering the proposed method, we have $m = 3 \cdot 3$ and $d_1 = d_2 = 5$, so the number of points is 25. The forward transformation uses $2(5 \cdot 5 \cdot 4) = 200$ additions. The pairwise-multiplication uses 25 multiplications. The reverse transformation uses another 200 additions with 25 multiplications of the constant $(5^{-1})^2$. Finally, the conversion to the entire field uses 8 additions. Therefore, the total number of multiplications is $2 \cdot 25 = 50$ and that of additions is $2 \cdot 200 + 8 = 408$.

If we wish to apply NTT to our proposed algorithm, we can do so by choosing $d_1 = d_2 = 6$, since 6th root of unity exists. The forward transform uses $2[6 \cdot 6(2+3)] = 360$ multiplications and $2[6 \cdot 6(1+2)] = 216$ additions. In total, the number of multiplications is $2 \cdot 360 + 36 = 756$ and that of additions is $2 \cdot 216 + 14 = 446$.

In this example, the values of p and m are suitable for our proposed algorithm. On the other hand, it requires $d = 30$ for NTT which is much larger than $2m - 1 = 17$.

6 Conclusion

We present a new polynomial modular multiplication in $\mathrm{GF}(p^m)$ using Mersenne transform and subfields, which uses $O(m^{1+o(1)} \log m)$ additions and bit-shifts, and $O(m^{1+o(1)})$ multiplications in $\mathrm{GF}(p)$. When doing the comparison based on the asymptotic number of operations, the proposed algorithm does not perform best. However, there exist some cases which our algorithm has a better performance. Furthermore, our algorithm is applicable to all finite fields whereas the choices of p and m for NTT and Mersenne transform are limited. This advantage is due to the use of subfields. We believe that it might be possible to use the idea of subfields to increase the usability of other existing algorithms and leave this as future work.

References

1. Baktır, S., Sunar, B.: Finite field polynomial multiplication in the frequency domain with application to elliptic curve cryptography. In: International Symposium on Computer and Information Sciences. pp. 991–1001. Springer (2006)
2. Coppersmith, D., Winograd, S.: Matrix multiplication via arithmetic progressions. In: Proceedings of the nineteenth annual ACM symposium on Theory of computing. pp. 1–6 (1987)
3. Karatsuba, A.A., Ofman, Y.P.: Multiplication of many-digital numbers by automatic computers. In: Doklady Akademii Nauk. vol. 145, pp. 293–294. Russian Academy of Sciences (1962)
4. Knuth, D.E.: The Art of Computer Programming, Volume 2 (3rd Ed.): Seminumerical Algorithms. Addison-Wesley Longman Publishing Co., Inc., USA (1997)
5. Menezes, A.J., Van Oorschot, P.C., Vanstone, S.A.: Handbook of applied cryptography. CRC press (1996)
6. Pollard, J.M.: The fast fourier transform in a finite field. Mathematics of computation **25**(114), 365–374 (1971)
7. Rader, C.M.: Discrete convolutions via mersenne transforms. IEEE Transactions on Computers **100**(12), 1269–1273 (1972)
8. Strassen, V.: Gaussian elimination is not optimal. Numerische mathematik **13**(4), 354–356 (1969)
9. Vanstone, S.A., Van Oorschot, P.C.: An introduction to error correcting codes with applications, vol. 71. Springer Science & Business Media (2013)