Achieving Efficient Polynomial Multiplication in Fermat Fields Using the Fast Fourier Transform

Selçuk Baktır
Worcester Polytechnic Institute
Worcester, MA 01609, USA
selcuk@wpi.edu

Berk Sunar
Worcester Polytechnic Institute
Worcester, MA 01609, USA
sunar@wpi.edu

ABSTRACT

We introduce an efficient way of performing polynomial multiplication in a class of finite fields $GF(p^m)$ in the frequency domain. The Fast Fourier Transform (FFT) based frequency domain multiplication technique, originally proposed for integer multiplication, provides an extremely efficient method for multiplication with the best known asymptotic complexity, i.e. $O(n \log n \log \log n)$. Unfortunately, the original FFT method bears significant overhead due to the conversions between the time and the frequency domains, which makes it impractical to perform multiplication of relatively short $(160 - 1024$ bits$)$ integer operands as used in many applications. In this work, we introduce an efficient way of performing polynomial multiplication in finite fields using the FFT. We show that, with careful selection of parameters, all the multiplications required for the FFT computations can be avoided and polynomial multiplication in finite fields can be achieved with only $O(m)$ multiplications in addition to $O(m \log m)$ simple shift, addition and subtraction operations. We show that, especially in constrained devices where multiplication is expensive, polynomial multiplication in the suggested finite fields using the FFT outperforms both the schoolbook and Karatsuba methods for practically small finite fields, e.g., relevant to elliptic curve cryptography.

Keywords

Finite fields, polynomial multiplication, Fast Fourier Transform (FFT), Fermat numbers, Fermat transform, elliptic curve cryptography, coding theory

1. INTRODUCTION

Finite fields have many applications in coding theory [4, 3] and cryptography [9, 5, 11]. Hence efficient implementation of finite field arithmetic operations is desired. The classical polynomial multiplication method has quadratic complexity, i.e. $O(m^2)$, given in terms of ground field multiplications and additions. The complexity may be improved to $O(m^{\log_2 3})$ using the Karatsuba method [8]. Despite the significant improvement gained by the Karatsuba method, the complexity is still not optimal. Furthermore, the implementation of the Karatsuba method is more burdensome due to its recursive nature. The known fastest multiplication algorithm, introduced by Schönhage and Strassen [15], performs multiplication in the frequency domain using the Fast Fourier Transform (FFT). Application of Fast Fourier Transform (FFT) algorithms, first introduced by Cooley and Tukey [6], speeds up the Fourier transform computations immensely leading to fast FFT multiplication algorithms [15] with asymptotic complexity $O(n \log n \log \log n)$ [7]. However the size of the operands for which FFT multiplication outperforms regular multiplication is very large, which makes FFT multiplication impractical for most applications. The burden in the FFT based multiplication algorithms are due to the costly inverse and forward FFT operations which are usually performed with floating point numbers using complex number arithmetic.

Although FFT integer multiplication algorithms have been under close scrutiny, not much work has been done for application of the FFT for polynomial multiplication in finite fields. In this work, we propose the use of the FFT for efficient computation of polynomial multiplication in finite fields for practically small operand sizes, e.g., relevant to elliptic curve cryptography. We show that by selecting special parameters, FFT computations can be performed efficiently by using finite field arithmetic merely with integers, rather than complex number arithmetic with floating point numbers. We show that with the drastic improvements gained in the FFT computations, the proposed FFT polynomial multiplication in finite fields becomes efficient for practically small fields, and even outperforms the Karatsuba multiplication algorithm in constrained environments where multiplication operation is expensive compared to simpler operations such as addition, subtraction and bitwise shift.

2. BACKGROUND

2.1 Number Theoretic Transform (NTT)

The number theoretic transform was introduced by Pollard [12]. For a finite field $G(p)$ and a sequence (a) of length d whose entries are from $G(p)$, the forward NTT of (a) over $G(p)$, denoted by (A), can be computed by utilizing a d-th primitive root of unity, denoted by r, from $G(p)$ or a finite extension of $G(p)$ as

$$A_j = \sum_{i=0}^{d-1} a_i r^{ij} \ , \ 0 \leq j \leq d-1 \ , \tag{1}$$

$$\tag{2}$$

where a_i and A_i denote the elements of (a) and (A), respectively for $0 \leq i \leq d-1$. Similarly, the inverse NTT of (A) over $GF(p)$ can be computed as

$$a_i = \frac{1}{d} \cdot \sum_{j=0}^{d-1} A_j r^{-ij} \ , \ 0 \leq i \leq d-1 \ . \tag{3}$$

In this setting, (a) and (A) are referred to as the time and frequency domain representations, respectively, of the same sequence. The above NTT operations over $GF(p)$ are performed by utilizing a d-th primitive root of unity r defined as follows.

Definition 1. r is a primitive d^{th} root of unity modulo p if

$$r^d = 1 \pmod p$$

and

$$r^{d/t} - 1 \neq 0 \pmod p$$

for any prime divisor t of d.

2.2 Convolution Theorem and Polynomial Multiplication in $GF(p^m)$ Using the NTT

According to the convolution theorem, computing the convolution of two sequences in the time domain is equivalent to computing the componentwise parallel multiplication of their frequency domain representations. The convolution of two d-element sequences (a) and (b) results in another d-element sequence (c) and can be shown as follows:

$$c_i = \sum_{j=0}^{d-1} a_j b_{i-j \bmod d} \ , \ \ 0 \leq i \leq d-1 \ . \tag{4}$$

Let (A), (B) and (C) denote the NTTs of (a), (b) and (c), respectively. The above convolution operation in the time domain is equivalent to the following computation in the frequency domain:

$$C_i = A_i \cdot B_i \ , \ \ 0 \leq i \leq d-1 \ . \tag{5}$$

Hence, convolution of two d-element sequences in the time domain, with complexity $O(d^2)$, is equivalent to simple pairwise multiplication of the DFTs of these sequences and has a surprisingly low $O(d)$ complexity. Note in (4) that, the convolution is computed by making the sequences (a) and (b) periodic with d. This is equivalent to the *cyclic convolution* of the two sequences.

2.2.1 NTT based Polynomial Multiplication in $GF(p^m)$

The finite field $GF(p^m)$ is generated by using an m^{th} degree irreducible polynomial over $GF(p)$ and comprises the residue classes modulo the irreducible field generating polynomial. Hence, in polynomial basis representation, the elements of $GF(p^m)$ are represented by polynomials of degree $m-1$ with coefficients in $GF(p)$[9]. Multiplication of two polynomials is basically the same as the *acyclic (linear) convolution* of the polynomial coefficients. We have seen that cyclic convolution can be performed very efficiently in the frequency domain by pairwise coefficient multiplications. Hence, it will be wise to represent the elements of $GF(p^m)$, which are $(m-1)^{st}$ degree polynomials, with at least $d = (2m-1)$ element sequences by using the m polynomial coefficients and appending zeros at the higher ordered $m-1$ positions, so that the cyclic convolution of two such sequences will be equivalent to their acyclic convolution and give us their polynomial multiplication. We can form sequences by taking the ordered coefficients of polynomials. For instance,

$$a(x) = a_0 + a_1 x + a_2 x^2 + \ldots + a_{m-1} x^{m-1} \ ,$$

a polynomial over $GF(p)$ and an element of $GF(p^m)$ with $a_i \in GF(p)$ for $0 \leq i \leq m-1$, can be interpreted as the following sequence after appending $d-m$ zeros to the right:

$$(a) = (a_0, a_1, a_2, \ldots, a_{m-1}, 0, 0, \ldots, 0) \ . \tag{6}$$

For $a(x), b(x) \in GF(p^m)$, and for $d \geq 2m-1$, the cyclic convolution of (a) and (b) yields a sequence (c) whose entries can be interpreted as the coefficients of a polynomial $c(x)$ such that $c(x) = a(x) \cdot b(x)$. The computation of this cyclic convolution can be performed by simple pairwise coefficient multiplications in the frequency domain. The following steps realize the polynomial multiplication $c(x) = a(x) \cdot b(x)$:

1. Interpret the coefficients of $a(x)$ and $b(x)$ as elements of the sequences (a) and (b), respectively, and append zeros to the right to make their lengths exactly d.
2. Convert (a) and (b) into their respective frequency domain representations (A) and (B) using the NTT operation of (1).
3. Multiply (A) and (B) together to compute (C) as in (5).
4. Convert (C) back to the time domain representation (c) using the inverse NTT operation of (3).
5. Interpret the first $2m-1$ coefficients of (c) as the coefficients of the product polynomial $c(x)$.

3. POLYNOMIAL MULTIPLICATION IN FERMAT FIELDS $GF(P^M)$

In this section, we utilize the FFT for speeding up the forward and inverse NTT operations in the computation of polynomial multiplication in a special class of *Fermat fields* $GF(p^m)$ where p is a Fermat prime of the form $p = 2^{2^n} + 1$ and $m = 2^n$. Here we call a finite field with a Fermat prime characteristic a *Fermat field*. We briefly explain how the FFT works, and refer the reader to [10] for further information. We derive the complexities of the forward and inverse FFT operations. Finally, we present the complexity of polynomial multiplication in $GF(p^m)$ using the FFT.

3.1 NTT Modulo A Fermat Prime

Definition 2. A Fermat number, i.e. $F_n = 2^{2^n} + 1$ for a positive integer n, that is also prime is called a Fermat prime.

Fermat primes are popular choices as finite field characteristics due to their computational advantage in the modular reduction operation. Modular reduction by a Fermat prime can be performed by simple addition/subtraction and shift operations.

There exist further advantages of Fermat primes in the computation of polynomial multiplication in $GF(p^m)$ using the NTT. If p is chosen as a Fermat prime, i.e. when $p = 2^{2^n} + 1$, then $2^{2^{n+1}} \equiv 1 \pmod{p}$ and $r = 2$ is a d^{th} primitive root of unity where $d = 2^{n+1}$. In this case, since d is a power of 2, the FFT can be applied very efficiently for computation of the NTT of a sequence of length d, significantly reducing the complexity of NTT polynomial multiplication in finite fields. In Table 1, we give the list of the Fermat primes and values for d and m that allow for application of the FFT and thus possibly efficient polynomial multiplication in $GF(p^m)$ in the frequency domain. In fact, after coming up with the idea of using Fermat numbers as the finite field characteristic for efficient NTT computations, we realized that the idea of computing finite transforms in rings of integers modulo Fermat numbers were already proposed by Rader in [14, 13]. Such transforms were defined as *Fermat transforms* and proposed for fast convolution and digital filtering by Agarwal and Burrus [1, 2]. Hence, with this paper, it happens that we do not invent or propose a totally new transform technique, yet we propose the use of Fermat transform in *finite fields* for practically efficient *finite field polynomial multiplication* which may find applications in coding theory and cryptography.

Table 1: Fermat primes $p = 2^{2^n} + 1$ and d, m values for FFT polynomial multiplication in $GF(p^m)$.

n	$p = 2^{2^n} + 1$	d	m	field size (in bits)
0	3	2	1	2
1	5	4	2	6
2	17	8	3, 4	15, 20
3	257	16	6, 7, 8	54, 63, 72
4	65537	32	13, 14, 15, 16	221, 238, 255, 272

3.2 Fast Fourier Transform

The symmetry of the NTT computation and the periodicity of the d^{th} primitive root of unity, i.e. r, can be exploited significantly using the FFT algorithms. In the FFT computations, the NTT of a large sequence is computed in terms of the NTTs of smaller subsequences. When the sequence length d is a power of 2, computation of a d-element NTT can be performed by recursively computing the NTTs of the two subsequences with half the length of the original one, drastically reducing the complexity of the NTT of the original sequence.

The NTT of a d-element sequence (a), where $d = 2^k$ for some positive integer k, can be expressed as follows:

$$\begin{aligned}
A_j &= \sum_{i=0}^{d-1} a_i r^{ij} \\
&= \sum_{0 \le i \le d-1}^{i\ even} a_i r^{ij} + \sum_{0 \le i \le d-1}^{i\ odd} a_i r^{ij} \qquad (7) \\
&= \sum_{i=0}^{\frac{d}{2}-1} a_{2i} r^{2ij} + \sum_{i=0}^{\frac{d}{2}-1} a_{2i+1} r^{(2i+1)j} \\
&= \sum_{i=0}^{\frac{d}{2}-1} a_{2i} (r^2)^{ij} + r^j \sum_{i=0}^{\frac{d}{2}-1} a_{2i+1} (r^2)^{ij}, \quad 0 \le j \le d-1 \ .
\end{aligned}$$

Note that r^2 is a $(\frac{d}{2})^{th}$ primitive root of unity in $GF(p)$. Hence, the above d-element NTT computation of A_j, for $0 \le j \le d-1$, can be performed with two $(\frac{d}{2})$-element NTTs which are the NTTs of the $(\frac{d}{2})$-element sequences consisting of the even indexed elements and the odd indexed elements of (a). In (7), the first and the second summations correspond to the $(\frac{d}{2})$-element NTTs of even and odd indexed elements of (a), respectively. Here, A_j needs to be computed for $0 \le j \le d-1$, not for $0 \le j \le \frac{d}{2}-1$. However, $(r^2)^j$ is periodic with $\frac{d}{2}$ for a d^{th} primitive root of unity r and d even and therefore $r^{j+\frac{d}{2}} = -r^j$. Thus, the equalities

$$\sum_{i=0}^{\frac{d}{2}-1} a_{2i} (r^2)^{i(j+\frac{d}{2})} = \sum_{i=0}^{\frac{d}{2}-1} a_{2i} (r^2)^{ij}$$

and

$$r^{j+\frac{d}{2}} \sum_{i=0}^{\frac{d}{2}-1} a_{2i+1} (r^2)^{i(j+\frac{d}{2})} = -r^j \sum_{i=0}^{\frac{d}{2}-1} a_{2i+1} (r^2)^{ij}$$

hold. Therefore, once A_j is computed for $0 \le j \le \frac{d}{2}-1$ as in (7) by performing two $(\frac{d}{2})$-element NTTs, $\frac{d}{2}-1$ multiplications for multiplications of the second summations by r^j (for $j = 0$ no multiplication is necessary for a multiplication by r^j) and $\frac{d}{2}$ additions for merging the two summations together, we can compute A_j for $\frac{d}{2} \le j \le d-1$ immediately by using the same already computed summations and with only additional $\frac{d}{2}$ subtractions for merging the two summations.

The inverse FFT of a d-element sequence can be computed in a similar manner as the forward FFT. The inverse NTT of a d-element sequence (A), where $d = 2^k$ for k a positive integer, can be expressed as follows:

$$a_i = \frac{1}{d} \sum_{j=0}^{d-1} A_j r^{-ij}, \quad 0 \le i \le d-1 \ .$$

Computation of the above inverse NTT operation can be performed by the inverse FFT in the same manner as the forward NTT is performed using the forward FFT. The only difference in the inverse FFT computation is that there are minus signs in front of the powers of r and d additional constant multiplications are performed due to the multiplications by d^{-1}.

The forward and the inverse FFTs of d-element sequences, for $d = 2^k$ where k is a positive integer, can be performed by recursively computing the $d = 2^k$ element (inverse)NTTs in terms of two $d' = 2^{k-1}$ element NTTs as shown in (7) for $2 \le i \le k$. Hence, with the FFT algorithm, computation of a 2^k-element (inverse) NTT is reduced to the computation of 2^{k-1} 2-element (inverse) NTTs in addition to some multiplications (due to the multiplications by r^j or r^{-j}), constant multiplications by d^{-1} (for the inverse FFT computation) and additions/subtractions for merging the summations as shown in (7).

3.3 Complexity of the FFT Computations in Fermat Fields

In this section, we derive the complexities of the forward and the inverse FFT computations of sequences of length $d = 2^{n+1}$ over Fermat fields $GF(p)$ where $p = 2^{2^n} + 1$ is a Fermat prime. We present the complexities for $r = 2$ which is always a d-th primitive root of unity in this scenario. Since the forward and inverse FFT operations consist mainly of multiplications by powers of r, i.e. r^{ij} and r^{-ij} for $0 \leq i,j \leq d-1$, when $r=2$ all these multiplications can be achieved by simple shift operations (by at most $d-1$ bits since r is a d^{th} primitive root of unity and $r^i = r^{i \bmod d}$ for any integer i), thereby reducing the complexity of FFT operations significantly. We utilize the following notation for denoting the complexities of computations in our complexity derivations:

- $\mathcal{F}^k$**:** Complexity of the forward FFT of a d-element sequence for $d = 2^k$,
- $\mathcal{M}$: Complexity of a multiplication in $GF(p)$,
- $\mathcal{S}'$: Complexity of a shift operation in $GF(p)$,
- $\mathcal{A}$: Complexity of an addition in $GF(p)$,
- $\mathcal{S}$: Complexity of a subtraction in $GF(p)$.
- $\mathcal{C}$: Complexity of a multiplication by a constant number in $GF(p)$, e.g. complexity of a multiplication by d^{-1} in the inverse FFT computation.

In this setting, the recursive complexity of a $d=2^k$ element forward FFT computation can be stated as

$$\mathcal{F}^k = 2\mathcal{F}^{k-1} + (2^{k-1}-1)\mathcal{M} + 2^{k-1}\mathcal{A} + 2^{k-1}\mathcal{S} \ .$$

Note that, rather than 2^{k-1}, $2^{k-1}-1$ multiplications are required due to multiplications by powers of r. This is because of the fact that one of these multiplications in the FFT computation is the multiplication by the zeroth power of r, i.e. r^0, which equals one, and hence can be avoided. The non-recursive complexity of the above forward FFT computation is found as

$$\begin{aligned}
\mathcal{F}^k &= 2^k\mathcal{F}^0 + \sum_{j=1}^{k} 2^{k-j}((2^{j-1}-1)\mathcal{M} + 2^{j-1}\mathcal{A} + 2^{j-1}\mathcal{S}) \\
&= \sum_{j=1}^{k} 2^{k-j}((2^{j-1}-1)\mathcal{M} + 2^{j-1}\mathcal{A} + 2^{j-1}\mathcal{S}) \\
&= (k2^{k-1} - 2^k + 1)\mathcal{M} + k2^{k-1}\mathcal{A} + k2^{k-1}\mathcal{S} \\
&= (\frac{d\log_2 d}{2} - d + 1)\mathcal{M} + \frac{d\log_2 d}{2}\mathcal{A} + \frac{d\log_2 d}{2}\mathcal{S} \ .
\end{aligned}$$

Note that $\mathcal{F}^0$ equals 0, since the FFT of a 1-element sequence equals itself. For $r=2$, no multiplications are required at all for the computation of the forward FFT operation since all of the required multiplications are due to multiplications by powers of r and hence can be performed by simple shifts. Hence, for $r=2$ the new complexity for a $d=2^k$ element forward FFT computation becomes

$$\mathcal{F}^k_{(r=2)} = (\frac{d\log_2 d}{2} - d + 1)\mathcal{S}' + \frac{d\log_2 d}{2}\mathcal{A} + \frac{d\log_2 d}{2}\mathcal{S} \ .$$

Complexity of the inverse FFT computation is the same as the complexity of the forward FFT computation, except for the additional $d=2^k$ constant multiplications by d^{-1}. Hence, we may write

$$\begin{aligned}
\mathcal{F}^k_{inverse} &= (k2^{k-1} - 2^k + 1)\mathcal{M} + 2^k\mathcal{C} + k2^{k-1}\mathcal{A} + k2^{k-1}\mathcal{S} \\
&= (\frac{d\log_2 d}{2} - d + 1)\mathcal{M} + d\mathcal{C} + \frac{d\log_2 d}{2}\mathcal{A} + \frac{d\log_2 d}{2}\mathcal{S} \ .
\end{aligned}$$

Again, for $r=2$, all multiplications can be achieved by simple shift operations. Furthermore, all of the d constant multiplications by $d^{-1} = 2^{-k} = 2^{d-k} \pmod p$ can also be achieved by simple shifts. Hence, the above complexity of the inverse FFT computation becomes

$$\mathcal{F}^k_{inverse(r=2)} = (\frac{d\log_2 d}{2} + 1)\mathcal{S}' + \frac{d\log_2 d}{2}\mathcal{A} + \frac{d\log_2 d}{2}\mathcal{S} \ .$$

In Figure 1 (Appendix), for illustration we present a flow diagram showing the computation of a 32-element FFT. On the left hand side of the diagram is the input sequence (a) with elements $a_0, a_1, a_2, \ldots, a_{31}$ and on the right hand side are the elements $A_0, A_1, A_2, \ldots, A_{31}$ of (A), the FFT of (a). In the flow diagram, the values of every branch entering into a circled node from the left are added up to produce the value of the node. Every branch coming out from the right hand side of a node has the value of that node. Even though not all of the branches are drawn as directed, all branches move from left to the right. If a branch does not have a directed arrow at the end of it, then its value equals the value of the node which it originated from. Otherwise, the branch value equals the node value weighted by a certain power of r shown next to the arrow. The flow diagram for the inverse FFT computation would be similar to the one in Figure 1, except the exponents of r would have minus signs in front of them and at the end all of the 32 sequence elements would go through a final multiplication by the constant d^{-1}. One may find, both in the above complexity formulas and in Figure 1, that 49 multiplications, 80 additions and 80 subtractions are required for the computation of a 32-element FFT.

3.4 Complexity Analysis of Polynomial Multiplication in $GF(p^m)$ Using the FFT for $p = 2^{2^n}+1$ and $m \leq 2^n$

Using Fermat primes $p = 2^{2^n}+1$ as the field characteristic, polynomial multiplication in $GF(p^m)$, where $m \leq 2^n$, can be efficiently achieved in the frequency domain using the FFT with $d=2$, since in this case $r=2$ is a d^{th} primitive root of unity where $d = 2^{2^{n+1}}$.

Remember in Section 2.2.1 that for performing polynomial multiplication in the frequency domain, we first need to compute the NTTs of the input sequences. Then, we do pairwise multiplications in the frequency domain. Finally, we find the time domain sequence for the resulting polynomial with a final inverse NTT computation.

In Section 3.3, we presented the complexities of the FFT and the inverse FFT computations for arbitrary sequences of length $d=2^k$ with $r=2$. However, in polynomial multiplication in $GF(p^m)$ we know that all but the first m elements of the input sequences are zeros. Since $d \geq 2m-1$, for $d=2^k$ at least the higher $d-m = 2^k - \lfloor\frac{2^k+1}{2}\rfloor = 2^{k-1}$ coefficients of the input sequences are zero. Hence, while performing the forward FFT computations of the input sequences, at least 2^{k-1} additions and 2^{k-1} subtractions (related to the zero elements) at the bottom of the FFT recursion need not be computed, e.g. see Figure 1 for $d=2^5$ where $a_{16}, a_{17}, \ldots, a_{31}$ are all zeros, $r^0 = 1$ and $r^{16} = -1$ (Appendix). Hence, the complexity of the forward FFT computation for polynomial multiplication becomes

$$\mathcal{F}^k_{(r=2)} = (\frac{d\log_2 d}{2} - d + 1)\mathcal{S}' + \frac{d\log_2 d - d}{2}\mathcal{A} + \frac{d\log_2 d - d}{2}\mathcal{S} \ .$$

Similarly, we know that multiplication of two polynomials, both with degree $m-1$, results in a degree $2m-2$ polynomial. Therefore, the higher ordered $d-2m+1$ elements of the d-element sequence for the resulting product of the polynomial multiplication are zero and need not be computed in the final inverse FFT computation. For $m \leq \frac{d}{2}$ the highest ordered element of the sequence for the product is always zero and need not be computed. For instance, albeit not significant, for $m = d/2$ we can save a subtraction and a constant multiplication by d^{-1} (or a shift for $r = 2$) by avoiding the computation of the highest ordered element of the product sequence in the inverse FFT computation. Hence, the complexity of the inverse FFT computation for polynomial multiplication becomes

$$\mathcal{F}^{k}_{inverse(r=2)} = (\frac{d\log_2 d}{2})\mathcal{S}' + \frac{d\log_2 d}{2}\mathcal{A} + (\frac{d\log_2 d}{2} - 1)\mathcal{S} \ .$$

Table 2 gives a complexity analysis of polynomial multiplication in $GF(p^m)$ using the FFT for $r = 2$, $d = 2^k$, and $m = 2^{k-1}$ for a positive integer k. For the same parameters, Table 3 compares the complexities of the FFT polynomial multiplication, the schoolbook multiplication and the recursive Karatsuba algorithm. For the derivation of the complexity of Karatsuba algorithm presented here, the interested reader is referred to [11]. Finally, Table 4 gives the complexities of the three methods for the specific case of $m = 2^4$, $d = 2^5$ and $p = 2^{2^4} + 1$ for polynomial multiplication in the 256-bit field $GF((2^{2^4} + 1)^{16})$.

Table 3: Complexity upper bounds of schoolbook, Karatsuba and the FFT methods for multiplication in $GF(p^m)$ in terms of $GF(p)$ operations for $p = 2^{2^n}+1$ and $m \leq 2^n$.

	#Mult.	#Shift	#Add.	#Subt.
Schoolbook	m^2	–	$(m-1)^2$	–
Karatsuba	$m^{\log_2 3}$	– –	$6m^{\log_2 3}$ $-8m+2$	– –
FFT	$2m$	$3m\log_2 m$ $-m+2$	$3m\log_2 m$ $+m$	$3m\log_2 m$ $+m-1$

Table 4: Complexities of polynomial multiplication in $GF(p^{16})$ for $p = 2^{2^4}+1$.

	#Mult.	#Shift	#Add.	#Subt.
Schoolbook	256	–	225	–
Karatsuba	81	–	360	–
FFT	32	178	208	207

As demonstrated in Tables 3 and 4, finite field polynomial multiplication may be achieved with dramatically less number of coefficient multiplications than the schoolbook and Karatsuba methods of multiplication even for practically small field sizes, e.g. relevant to elliptic curve cryptography. For computationally constrained platforms such as cell phones and wireless sensor nodes implementation of computationally excessive cryptographic algorithms is still a requirement for securing confidential information. In such constrained platforms where multiplication instruction may not be readily available, or is too costly compared to simpler operations such as additions, subtractions or bitwise shifts, a significant performance improvement can be gained by utilizing the FFT polynomial multiplication algorithm.

4. FUTURE RESEARCH DIRECTIONS

4.1 Time and Energy Efficiency

Battery powered portable devices such as cell phones, personal digital advisors and wireless sensor nodes are becoming more widespread, and in many applications they need to run cryptographic algorithms which require performing costly arithmetic operations. Hence, energy efficiency becomes a critical issue.

Multiplication is inherently more complex than simple arithmetic operations such as addition, subtraction and shift. Therefore, a multiplier circuit is designed significantly larger in area to make it perform as fast, e.g. in digital signal processors, or otherwise it takes a longer time to perform multiplication with a simple and small multiplier circuit. In any case, either due to large circuit area and more switching activities or due to longer processing time, the total energy consumed in performing a multiplication would be significantly larger than that for addition, subtraction or shift operations. Since the algorithm proposed in this paper requires a much smaller number of coefficient multiplications for performing polynomial multiplication, we believe that it will be more energy efficient. It may be interesting to investigate the time and energy efficiency of the proposed approach over several computational platforms.

4.2 Generalization

The FFT polynomial multiplication algorithm proposed in this work applies to a class of Fermat fields where the field characteristic is a Fermat prime of the form $p = 2^{2^n} + 1$ and the field extension degree is 2^n. It may be interesting to investigate if the proposed approach may be modified or generalized to apply efficiently to a wider class of finite fields.

5. CONCLUSIONS

We proposed an efficient way of performing polynomial multiplication in a class of Fermat fields using the FFT. We showed that, unlike the originally introduced FFT multiplication algorithms proposed for integer multiplication, FFT based polynomial multiplication in Fermat fields can be achieved efficiently for practically small operand sizes. Furthermore, we showed that all of the multiplications required for the FFT computations can be avoided and polynomial multiplication in a Fermat field can be achieved with only $O(m)$ multiplications in addition to $O(m \log m)$ simple shift, addition and subtraction operations. Finally, we claim that in computationally constrained environments where multiplication is expensive, polynomial multiplication in such Fermat fields outperforms both the schoolbook method and the Karatsuba algorithm for practically small operand sizes.

6. ACKNOWLEDGMENTS

This work was supported by the US National Science Foundation under Grant No. ANI-0133297.

7. REFERENCES

[1] R. C. Agarwal and C. S. Burrus. Fast Digital Convolution Using Fermat Transforms. In *Southwest IEEE Conf. Rec.*, pages 538–543, Houston, Texas, USA, April 1973.

[2] R. C. Agarwal and C. S. Burrus. Fast Convolutions Using Fermat Number Transforms with Applications

Table 2: Complexity analysis of polynomial multiplication in Fermat fields $GF(p^m)$, where $p = 2^{2^n} + 1$ and $m = 2^n$, using the FFT .

$(\mathbf{c}) = (\mathbf{a}) * (\mathbf{b})$	**#Mult.**	**#Shift**	**#Add.**	**#Subt.**
1. $(A) = FFT((a))$	–	$m \log_2 m - m + 1$	$m \log_2 m$	$m \log_2 m$
2. $(B) = FFT((b))$	–	$m \log_2 m - m + 1$	$m \log_2 m$	$m \log_2 m$
3. $(C) = (A) \cdot (B)$	$2m$	–	–	–
4. $(c) = FFT^{-1}((C))$	–	$m \log_2 m + m$	$m \log_2 m + m$	$m \log_2 m + m - 1$
Total Cost:	$2m$	$3m \log_2 m - m + 2$	$3m \log_2 m + m$	$3m \log_2 m + m - 1$

to Digital Filtering. *IEEE Transactions on Computers*, ASSP-22(2):87–97, April 1974.

[3] E. R. Berlekamp. *Algebraic Coding Theory.* McGraw-Hill, New York, New York, USA, 1968.

[4] R. E. Blahut. *Theory and Practice of Error Control Codes.* Addison-Wesley, Reading, Massachusetts, USA, 1983.

[5] I. Blake, X. Gao, R. Mullin, S. Vanstone, and T. Yaghgoobin. *Applications of Finite Fields.* Kluwer Academic, 1999.

[6] J. Cooley and J. Tukey. An Algorithm for the Machine Calculation of Complex Fourier Series. *Mathematics of Computation*, 19:297–301, 1965.

[7] R. Crandall and C. Pomerance. *Prime Numbers.* Springer-Verlag, New York, NY, USA, 2001.

[8] A. Karatsuba and Y. Ofman. Multiplication of Multidigit Numbers on Automata. *Sov. Phys. Dokl. (English translation)*, 7(7):595–596, 1963.

[9] R. J. McEliece. *Finite Fields for Computer Scientists and Engineers.* Kluwer Academic Publishers, 2nd edition, 1989.

[10] A. V. Oppenheim, R. W. Schafer, and J. R. Buck. *Discrete-Time Signal Processing.* Prentice-Hall, Upper Saddle River, New Jersey, USA, 2nd edition, 1999.

[11] C. Paar. *Efficient VLSI Architectures for Bit-Parallel Computation in Galois Fields.* PhD thesis, (Engl. transl.), Institute for Experimental Mathematics, University of Essen, Essen, Germany, June 1994. ISBN 3–18–332810–0.

[12] J. M. Pollard. The Fast Fourier Transform in a Finite Field. *Mathematics of Computation*, 25:365–374, 1971.

[13] C. M. Rader. Discrete Convolutions via Mersenne Transforms. *IEEE Transactions on Computers*, C-21(12):1269–1273, December 1972.

[14] C. M. Rader. The Number Theoretic DFT and Exact Discrete Convolution. In *IEEE Arden House Workshop on Digital Signal Processing*, Harriman, NewYork, January 1972.

[15] A. Schonhage and V. Strassen. Schnelle Multiplication grosser Zahlen. *Computing*, 7:281–292, 1971.

APPENDIX

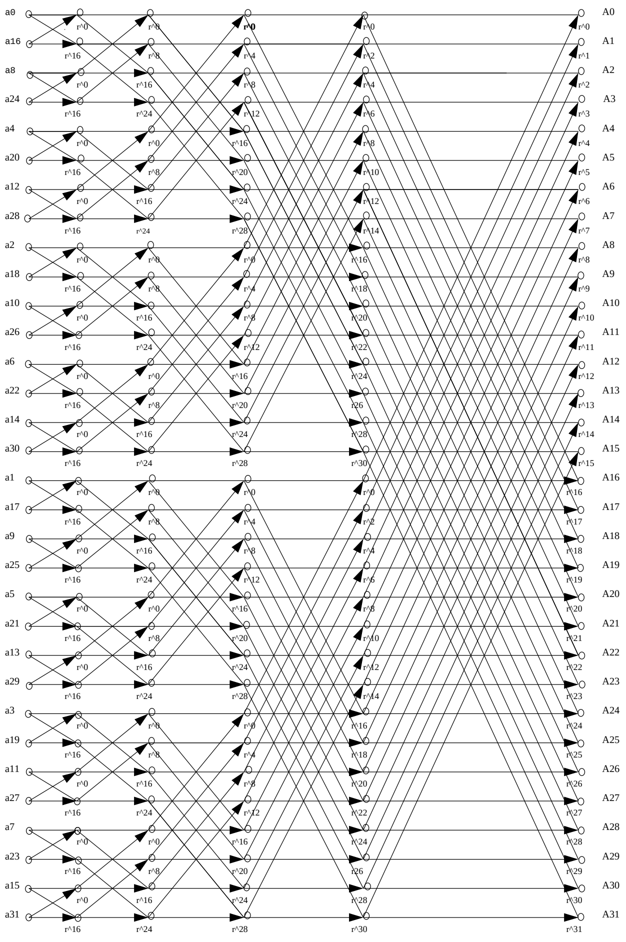

Figure 1. Flow diagram for 32−element FFT computation