Big Prime Field FFT on Multi-core Processors

Svyatoslav Covanov, Davood Mohajerani, Marc Moreno Maza, Linxiao Wang

Big Prime Field FFT on Multi-core Processors

Svyatoslav Covanov
University of Lorraine, France
svyatoslav.covanov@loria.fr

Davood Mohajerani
University of Western Ontario, Canada
dmohajer@uwo.ca

Marc Moreno Maza
University of Western Ontario, Canada
moreno@csd.uwo.ca

Linxiao Wang
University of Western Ontario, Canada
lwang739@uwo.ca

ABSTRACT

We report on a multi-threaded implementation of Fast Fourier Transforms over generalized Fermat prime fields. This work extends a previous study realized on graphics processing units to multi-core processors. In this new context, we overcome the less fine control of hardware resources by successively using FFT in support of the multiplication in those fields. We obtain favorable speedup factors (up to 6.9x on a 6-core, 12 threads node, and 4.3x on a 4-core, 8 threads node) of our parallel implementation compared to the serial implementation for the overall application thanks to the low memory footprint and the sharp control of arithmetic instructions of our implementation of generalized Fermat prime fields.

CCS CONCEPTS

• **Computing methodologies** → **Symbolic and algebraic manipulation**; **Algebraic algorithms**; **Parallel algorithms**.

KEYWORDS

Fast Fourier transforms; Finite fields; Generalized Fermat numbers; Specialized arithmetic; Multi-core processors; Intel CilkPlus;

1 INTRODUCTION

Prime field arithmetic plays a central role in computer algebra and supports computation in Galois fields which are essential to coding theory and cryptography algorithms. The prime fields that are used in computer algebra systems, in particular in the implementation of modular methods, are often of single precision. Increasing precision beyond the machine word size can be done via the Chinese Remainder Theorem (CRT) or Hensel Lemma. However, using machine-word size, thus small, prime numbers has serious inconveniences in certain modular methods, in particular for solving systems of non-linear equations. Indeed, in such circumstances, the so-called unlucky primes are to be avoided, see for instance [2, 8].

We consider prime fields of large characteristic, typically fitting on k machine words, where k is a power of 2. When the characteristic of these fields is restricted to a subclass of the generalized Fermat numbers, the authors of [5] have shown, in an ISSAC 2017 paper, that arithmetic operations in such fields offer attractive performance, both in terms of algebraic complexity and parallelism. In particular, these operations can be vectorized, leading to an efficient implementation of fast Fourier transforms on graphics processing units (GPUs), reported in that same paper.

In the present work, we turn our attention to the most commonly used processors of today's laptops and desktops, namely multi-core processors. These architectures are, in principle, not suitable for fine grained parallelism, in contrast with GPUs. GPUs and multi-core processors differ in memory hierachies as well as communication and synchronization mechanisms between threads. Moreover, GPU architectures offer programmers a finer control of hardware resources than multi-core processors and thus more opportunities to reach high performance. These features of GPU architectures have been essential in the implementation of arithmetic operations of generalized Fermat prime fields. Hence, the implementation techniques developed in [5] can not be easily ported and applied to the context of multi-core processors.

This leads us to a first question: can a serial implementation (written in C programming language) take advantage of the properties of those finite fields towards an implementation of fast Fourier transform (FFT) over those fields? The answer is yes, however, the route that we took is, of course, quite different than in the GPU case. Instead of performing many batches of arithmetic operations (a natural way of doing things in a GPU implementation) we have focused our effort in optimizing the multiplication between two arbitrary elements of our generalized Fermat prime fields. Consider a generalized Fermat prime number of the form $p = r^k + 1$, where k is a power of 2 and r is of machine-word size. As mentioned in [5], multiplying by a power of r modulo p can be done in $O(k)$ machine-word operations. However, multiplying two arbitrary elements of $\mathbb{Z}/p\mathbb{Z}$ is a non-trivial operation. Note that we encode elements of $\mathbb{Z}/p\mathbb{Z}$ in radix r expansion. Thus, multiplying two arbitrary elements of $\mathbb{Z}/p\mathbb{Z}$ requires computation of the product of two univariate polynomials in $\mathbb{Z}[X]$, of degree less than k, modulo $X^k + 1$. In [5], this is done by using plain multiplication, thus $\Theta(k^2)$ machine-word operations. In Section 3, we explain how to multiply two arbitrary elements x, y of $\mathbb{Z}/p\mathbb{Z}$ via FFT.

A second natural question is whether a multi-threaded implementation of big prime field FFT can deliver interesting speedup factors. While obtaining efficient multi-threaded implementation of FFTs with coefficients in single or double precision is a standard research topic [1, 12, 18, 19], the case of higher precision has received little attention so far. With coefficients in the generalized Fermat

prime field $\mathbb{Z}/p\mathbb{Z}$, our FFT is in the spirit of the algorithms of Schönhage and Strassen [21] and Fürer [13], where fast multiplication is achieved by "composing" FFTs operating on different vector sizes.

The practicality of Fürer's algorithm is still an open question, a question that we touch in this paper, without fully addressing it. Several algorithms, similar to Fürer's, have been proposed since. For example, in [9, 10] De et al. gave a similar algorithm which relies on *finite field arithmetic* and achieves the same running time as Fürer's algorithm. Later, Harvey, Van der Hœven and Lecerf proposed, for the integer multiplication, a theoretical improvement to Fürer's algorithm in [16] based on Bluestein's chirp transform. In [15], they also propose a similar algorithm for the multiplication over finite fields, achieving a Fürer-like complexity. This work led to an efficient implementation in [17], using multiplication of polynomials over the special field $\mathbb{F}_{2^{60}}$. In [7], Covanov and Thomé proposed an algorithm based on generalized Fermat primes and the same scheme as Fürer's algorithm, to multiply integers with a Fürer-like complexity.

Returning to our second question, addressing the parallel execution of FFT over big prime fields on multi-cores, the answer is yes. On a 4-core processor and on a 6-core processor, both equipped with hyper-threading technology, we reached nearly linear speedup for the largest input data that we tried.

To measure the benefits of our optimized implementation of the generalized Fermat prime field $\mathbb{Z}/p\mathbb{Z}$, we have realized a naive implementation of the same field, where the radix representation is not used. In this second implementation, the sum $a+b \mod p$ and the product $a \times b \mod p$ are simply computed by calling the modular sum and modular product functions from the GNU Multiple Precision Arithmetic Library (GMP) [14]. The performance of our big prime field FFT degrades substantially with this second implementation of $\mathbb{Z}/p\mathbb{Z}$. The difference in the performance of the optimized implementation can be attributed to, by our measurements, the sharp management of computing resources (i.e. specialized arithmetic and minimal usage of memory).

The experimental results reported in Section 5 support the positive answers to our two questions. Our code is part of the *Basic Polynomial Algebra Subprograms*, also known as the BPAS library [3] and is publicly available at http://www.bpaslib.org/.

2 GENERALIZED FERMAT PRIME FIELDS

The residue classes modulo p, where p is a prime number, form a field (unique up to isomorphism) called the *prime field* with p elements, denoted by GF(p) or $\mathbb{Z}/p\mathbb{Z}$. Single-precision and multi-precision primes are referred to as *small primes* and *big primes*.

Since modular methods for polynomial systems rely on polynomial arithmetic, these large prime numbers must support FFT-based algorithms, such as FFT-based polynomial multiplication. Therefore, we consider the so-called generalized Fermat prime numbers. The detailed introduction of generalized Fermat prime numbers can be found in the previous work of our research group [5].

In this paper, we denote a generalized Fermat prime number p as $p = r^k + 1$, and $\mathbb{Z}/p\mathbb{Z}$ to represent the finite field GF(p). In particular, in the field $\mathbb{Z}/p\mathbb{Z}$, r is a $2\,k$-th primitive root of unity. Each element $x \in \mathbb{Z}/p\mathbb{Z}$ is represented by a vector $\vec{x} = (x_{k-1}, \ldots, x_0)$ of length k. We can also use a univariate polynomial $f_x \in \mathbb{Z}[R]$ to represent x: we write $f_x = \sum_{i=0}^{k-1} x_i R^i$, such that $x \equiv f_x(r) \mod p$. The basic arithmetic algorithms in $\mathbb{Z}/p\mathbb{Z}$ are also introduced in [5] Section 3.

As we have mentioned above, for $p = r^k + 1$, r is a $2\,k$-th primitive root unity in $\mathbb{Z}/p\mathbb{Z}$, Section 3.3 of [5] has provided a very efficient algorithm for multiplication between elements $x, y \in \mathbb{Z}/p\mathbb{Z}$, where one of them is a power of r. We assume that $y = r^i$ for some $0 \le i \le 2k$. The cases $i = 0$ and $i = 2k$ are trivial, since r is a $2k$-th primitive root of unity in $\mathbb{Z}/p\mathbb{Z}$, we have $r^0 = r^{2k} = 1$. Also we have $r^k = -1$ in $\mathbb{Z}/p\mathbb{Z}$, so that for $i = k$, we have $x = -x$ and for $k < i < 2k$, $r^i = -r^{i-k}$ holds. Now let us only consider the case $0 < i < k$, where we have the following equation:

$$\begin{aligned} x r^i &\equiv (x_{k-1}\, r^{k-1+i} + \cdots + x_0\, r^i) \mod p \\ &\equiv \sum_{j=0}^{j=k-1} x_j r^{j+i} \mod p \equiv \sum_{h=i}^{h=k-1+i} x_{h-i} r^h \mod p \\ &\equiv (\sum_{h=i}^{h=k-1} x_{h-i} r^h - \sum_{h=k}^{h=k-1+i} x_{h-i} r^{h-k}) \mod p \end{aligned}$$

We see that for all $0 \le i \le 2k$, the product $x \cdot r^i$ is reduced to a shift and a subtraction. We call this process *cyclic shift*.

The C implementation can be found in the BPAS library [3], we refer to this function as MulPowR in this paper. Our main motivation for using generalized Fermat primes is that, thanks to cyclic shifts, multiplications of elements of $\mathbb{Z}/p\mathbb{Z}$ by a power of r are computationally cheap; this offers the opportunity to reduce the average time spent in multiplication operations during the execution of FFT algorithm over such finte fields. Multiplication between two arbitrary elements in $\mathbb{Z}/p\mathbb{Z}$ can be very complicated and expensive, our previous work [5] gave a theoretical algorithm of computing the product $x\,y \in \mathbb{Z}/p\mathbb{Z}$ using polynomial multiplication (See Algorithm 3 in [5]). In the following section, we will discuss the multiplication between arbitrary elements in more detail, and explain the C implementation.

3 OPTIMIZING MULTIPLICATION IN GENERALIZED FERMAT PRIME FIELDS

In this section, we discuss how we can efficiently multiply two arbitrary elements in $\mathbb{Z}/p\mathbb{Z}$ (when p is a generalized Fermat prime) using FFT. In Section 3.1, we outline an algorithm based on polynomial multiplication via FFT. In Section 3.2 we present an implementation of the FFT-based multiplication, then, proceed by explaining each sub-routine.

3.1 Algorithms

For a generalized Fermat prime p, our approach follows the concepts from Section 2, which treats any two elements x and y of $\mathbb{Z}/p\mathbb{Z}$ as polynomials f_x and f_y, then, uses polynomial multiplication algorithms to obtain the product xy. In practice, there are more details to be considered in order to reach high-performance. For instance, how do we efficiently convert a positive integer in the range $(0, r^3)$ into radix-r representation.

Consider $u = x\,y \mod p$ with $x, y, u \in \mathbb{Z}/p\mathbb{Z}$. We use the polynomial representation of the elements in the field, that is, $f_x(R) = x_{k-1}\,R^{k-1} + \cdots + x_1\,R + x_0$ and $f_y(R) = y_{k-1}\,R^{k-1} + \cdots + y_1\,R + y_0$. The first step is to multiply the two polynomials f_x and f_y. Computing $f_u(R) = f_x(R) \cdot f_y(R) \mod (R^k + 1)$ can be interpreted as a *negacyclic convolution*. A cyclic convolution computes $f(x) \cdot g(x) \mod (x^n - 1)$ for two polynomials f and g with degree less than n. Fast algorithms for computing cyclic convolutions via discrete Fourier transform (DFT) are presented, for instance, in [22]. Similar approaches can be used for computing negacyclic convolutions.

Let q be a prime, ω be an n-th primitive root of unity in $\mathbb{Z}/q\mathbb{Z}$, and θ be a $2n$-th primitive root of unity in $\mathbb{Z}/q\mathbb{Z}$. Also, we have two polynomials $f(x)$ and $g(x)$ with degree less than n, we use $\vec{a}$ and $\vec{b}$ to represent the coefficient vector of the f and g. The negacyclic convolution of f and g can be computed as follows:

$$\vec{A'} \cdot \mathsf{InverseDFT}(\mathsf{DFT}(\vec{A} \cdot \vec{a}) \cdot \mathsf{DFT}(\vec{A} \cdot \vec{b})) \tag{1}$$

where $\vec{A} = (1, \theta, \ldots, \theta^{n-1})$ and $\vec{A'} = (1, \theta^{-1}, \ldots, \theta^{1-n})$. All the dots between vectors are point-wise multiplications. The InverseDFT and DFTs are all computed at k points. In our implementation, we use unrolled DFTs (similar to the base-case DFTs given in Section 4.3 but relying on prime field arithmetic for a single machine word).

Notice that for f_x and f_y in $\mathbb{Z}/p\mathbb{Z}$, the size of each coefficient must be at most 63 bits wide. This implies that when we compute $f_u(R) = f_x(R) \cdot f_y(R) \mod (R^k + 1)$, the size of the coefficients of f_u will be at most $\log k + (2 \cdot 63) = 126 + \log k$, which is more than one machine word. We overcome this situation by means of a scheme based on the Chinese Remainder Theorem (CRT).

For k small enough, we use two machine word size primes p_1 and p_2 satisfying the relation of $R \leq \frac{p_1\,p_2 - 1}{2}$ where $R = k\,r^2$ is greater or equal than each of $|u_0|, \ldots, |u_{k-1}|$. Let m_1 and m_2 be two integers such that $p_1\,m_1 + p_2\,m_2 = 1$. Then, each coefficient u_i of f_u can be computed using the Chinese Remaindering Theorem.

Now each coefficient u_i of f_u is the combination of k terms, so the absolute value of each u_i is bounded over by $k \cdot r^2$ which implies that it needs at most $\lfloor \log k + 2\,\log r \rfloor + 1$ bits to be encoded. Since k is usually between 4 to 256, a radix r representation of u_i of length 3 is sufficient to encode u_i. Hence, we denote by $[c_i, h_i, l_i]$ the 3 integers uniquely given by $u_i = c_i r^2 + h_i r + l_i$, where $0 \leq h_i, l_i < r$ and $c_i \in [-(k-1), k]$.

Then, we can rewrite:

$$f_u(R) = f_x(R) \cdot f_y(R) \mod (R^k + 1) = \sum_{i=0}^{k-1} (c_i\,R^{2+i} + h_i\,R^{1+i} + l_i\,R^i).$$

Now, we have all the coefficients of f_u in the form of $[l, h, c]$. Rearranging the k $[l, h, c]$ vectors gives us three vectors $\vec{l} = [l_0, \ldots, l_{k-1}], \vec{h} = [h_0, \ldots, h_{k-1}]$ and $\vec{c} = [c_0, \ldots, c_{k-1}]$.

Finally, we compute $\vec{l} + \vec{h}.r + \vec{c}.r^2$ to get the final result of $x\,y \in \mathbb{Z}/p\mathbb{Z}$. We refer to this approach of multiplying two arbitrary elements in $\mathbb{Z}/p\mathbb{Z}$ as the FFT-based multiplication in the generalized Fermat prime field. The complete solution is presented in Algorithm 1.

Algorithm 1 FFT-based multiplication for two arbitrary elements in $\mathbb{Z}/p\mathbb{Z}$.

```
1: procedure FFT-basedMultiplication(x⃗, y⃗, r, k)
2:     z⃗1 := NegacyclicConvolution(x⃗, y⃗, p1, k)
3:     z⃗2 := NegacyclicConvolution(x⃗, y⃗, p2, k)
4:     for 0 ≤ i < k do
5:         [s0i, s1i] := CRT(p1, p2, m1, m2, z1i, z2i)
6:         [li, hi, ci] := LHC(s0i, s1i, r)
7:     end for
8:     c⃗ := MulPowR(c⃗, 2, k, r)
9:     h⃗ := MulPowR(h⃗, 1, k, r)
10:    u⃗ := BigPrimeFieldAddition(l⃗, h⃗, k, r)
11:    u⃗ := BigPrimeFieldAddition(u⃗, c⃗, k, r)
12:    return u⃗
13: end procedure
```

3.2 Implementation in C

In this section we describe our implementation of the FFT-based multiplication for two arbitrary elements of $\mathbb{Z}/p\mathbb{Z}$. We follow the ideas of Algorithm 1 and take care of implementation details.

Note that Algorithm 1 heavily relies on single-precision modular multiplications, especially in the convolution step. To maximize practical performance, we use Montgomery's tricks from [20] for performing operations in $\mathbb{Z}/p\mathbb{Z}$, in particular multiplication. We use the improved Montgomery multiplication (similar to an algorithm from [6]) which we have implemented using *inline assembly* in C. The code can be found in the BPAS library.

Note that in Algorithm 1, both the convolution and CRT steps require a large number of modular multiplication operations. With that in mind, before performing either of the convolutions, we convert the two vectors $\vec{x}$ and $\vec{y}$ into Montgomery representation, once for p_1 and once for p_2. After that, we compute the negacyclic convolutions. Once the convolution is carried out, we need to retrieve the result from the Montgomery representation. This step is performed as part of the CRT computation:

$$a'_2 = (a_2\,m_1) \mod p_2$$
$$a'_1 = (a_1\,m_2) \mod p_1$$

In the next step, we compute the second part of the CRT algorithm:

$$a'_2\,p_1 + a'_1\,p_2$$

Note that here we need to perform two 64-bit multiplications (thus using two 128-bit numbers), then, add the results via 128-bit arithmetic. Once again for the sake of efficiency, we turn to inline assembly in C (the implementation code can be found in the BPAS Library [3]). Finally, for u_i as a coefficient of $f_u = f_x \cdot f_y \mod (R^k + 1) \in \mathbb{Z}$, the result is stored as a pair of 64-bit numbers $[s_0, s_1]$ so that we have $u_i = s_1\,2^{64} + s_0$.

At this point, as we discussed in Section 3.1, we need to convert the coefficients of f_u into radix-based representation (l, h, c). Provided that the following relations are satisfied:

$$s_0 = q_0\,r + m_0 \text{ with } q_0, m_0 < r, \tag{2}$$
$$s_1 = q_1\,r + m_1 \text{ with } q_1, m_1 < r, \tag{3}$$
$$2^{64} = q_2\,r + m_2 \text{ with } q_2, m_2 < r, \tag{4}$$

we proceed by computing the triple $[l', h', c']$ as follows:

$$\begin{aligned}[l', h', c'] &= (q_0\,r + m_0) + (q_1\,r + m_1)\,(q_2\,r + m_2) \\ &= q_1\,q_2\,r^2 + (m_1\,q_2 + m_2\,q_1 + q_0)\,r + (m_0 + m_1\,m_2) \\ &= c'\,r^2 + h'\,r + l'\end{aligned}$$

Notice that the triple $[l', h', c']$ is still not the final result since either of h' or l' can be greater than r. For that matter, we need to compute the quotient and the remainder of h' (resp. l') by r. As the value of r remains constant during the whole computation, we use an adaptation of Barret reduction [4] using 128-bit arithmetic for computing the division by r (for more details, see function div_by_const_R_ptr in [3]). Then, we have

$$l' = h1.r + l1 \quad \text{and} \quad h' = h2.r + l2$$

The final result is computed by the following additions:

$$l + h.r + c.r^2 = [l1, h1, 0] + [l2, h2, 0].r + [0, 0, c']$$

To this end, we have explained the full implementation of the FFT-based multiplication for multiplying two arbitrary elements in $\mathbb{Z}/p\mathbb{Z}$. In Section 5, we present experimental results for comparing our implementation against that of the GMP library [14].

4 A GENERIC IMPLEMENTATION OF FFT OVER PRIME FIELDS

In Section 4.1, we first review the tensor algebra formulation of FFT, following the presentation of [11]. In Section 4.2, we explain how one can use the recursive formulation of the six-step DFT to derive an iterative algorithm in which all DFT computations are performed via a fixed size *base-case*. In the context of Generalized Fermat prime fields, this reduction allows us to take advantage of the "cheap" multiplication by powers of the radix r introduced in Section 2. Finally, in Section 4.3, we discuss the implementation of efficient routines for computing the base-case DFT_K.

4.1 The tensor algebra formulation of FFT

In this section, we review the tensor formulation of FFT. Recall that over a commutative ring R, an n-point DFT_n is a linear map from R^n to R^n. For $N = JK$, we use the six-step FFT factorization presented in [11]:

$$\text{DFT}_N = L^N_K\,(I_J \otimes \text{DFT}_K) L^N_J\, D_{K,J}\,(I_K \otimes \text{DFT}_J)\, L^N_K \quad with \quad N = J\,K \tag{5}$$

Definition 1. *The stride permutation L^{KJ}_K permutes an input vector $\vec{x}$ of length KJ as follows, with $0 \le i < K$ and $0 \le j < J$:*

$$\vec{x}[iJ + j] \mapsto \vec{x}[jK + i] \tag{6}$$

For an input vector $\vec{x}$ of length KJ, if we look at the vector as a row-major $J \times K$ matrix M, then, the stride permutation L^{KJ}_K is equivalent to performing a transposition on M:

$$L^{KJ}_K(M_{J\times K}) = (M_{J\times K})^T \tag{7}$$

For example, let $\vec{x}_8 = [0, 1, 2, 3, 4, 5, 6, 7]$, we compute $L^{2\times 4}_2(\vec{x})$. We can rearrange $\vec{x}$ as a row-major 4×2 matrix M, then, perform a transpose:

$$M^T_{4\times 2} = \begin{bmatrix} 0 & 1 \\ 2 & 3 \\ 4 & 5 \\ 6 & 7 \end{bmatrix}^T = \begin{bmatrix} 0 & 2 & 4 & 6 \\ 1 & 3 & 5 & 7 \end{bmatrix}$$

We retrieve the result by reading the consequent rows of M. Therefore, we have $L^{2\times 4}_2(\vec{x}) = [0, 2, 4, 6, 1, 3, 5, 7]$.

Definition 2. *The twiddle factor $D_{K,J}$ is a matrix of the powers of ω:*

$$D_{K,J} = \bigoplus_{j=0}^{K-1} diag(1, \omega_i^j, \ldots, \omega_i^{j(J-1)}) \tag{8}$$

4.2 The BPAS implementation of the FFT

The dominant cost during computation of FFT over $\mathbb{Z}/p\mathbb{Z}$ is the time spent in the multiplication by twiddle factors (powers of root of unity). Even though we can compute all the twiddle factor multiplications with Algorithm 1, however, inspired by the ideas discussed in Fürer's paper [13], our goal is to efficiently compute FFT on a vector of size $N = K^e$ through base-case DFT_K's. We face three main challenges. First, we need an algorithm to reduce the computation of DFT_N to base-case DFT_K's. Second, we need an efficient implementation of the base-case DFT_K which relies on cheap multiplications by K-th primitive roof of unity (as it is explained in Section 2). Finally, we need to have an FFT implementation which can be parallelized on a multi-core CPU, therefore the choice of the FFT algorithm is critical to achieve high performance.

In the BPAS library, and with respect to the above challenges, we decided to implement DFT over $\mathbb{Z}/p\mathbb{Z}$ based on the six-step FFT factorization of [11] (see Equation (5) in Section 4.1). The six-step FFT factorization provides an easy solution to the first challenge: we simply unroll Equation (5) until all DFT computations are performed through a sequence of DFT_K's. The process of reduction to the base-case is as follows. For computing the product $I_K \otimes \text{DFT}_J$, we can further expand it until we reach the base-case DFT_K. The derived solution is presented in Algorithm 2.

Regarding the parallelization, Algorithm 2 is iterative and it has no recursive calls, it only includes a number of nested for-loops. This makes the whole implementation suitable for a parallel implementation on a multi-core CPU. In fact, the inner for-loop nests at Lines L5, L10, L16, L21, L25 can be executed in parallel. On that basis, we have parallelized our implementations of FFT over $\mathbb{Z}/p\mathbb{Z}$ using `Intel CilkPlus`. Experimental results for comparing parallel and serial implementations are reported in Section 5.

4.3 Efficient implementation of DFT_K

Once again, we benefit from reduction to a base-case. This time, for computing DFT_K, we reduce the whole computation to a sequence of base-case DFT_2's which are defined in the following way:

$$\text{DFT}_2(x_0, x_1) = (x_0 + x_1, x_0 - x_1) \tag{9}$$

Then, for $K = 2^n$, we recursively apply the following factorization until all DFT computations are in DFT_2:

$$\text{DFT}_{2^n} = L^{2^n}_2\,(I_{2^{n-1}} \otimes \text{DFT}_2)\, L^{2^n}_{2^{n-1}}\, D_{2,2^{n-1}}\,(I_2 \otimes \text{DFT}_{2^{n-1}})\, L^{2^n}_2 \tag{10}$$

Now, let us consider the example of base-case DFT_8 in $\mathbb{Z}/p\mathbb{Z}$ when $p = r^4 + 1$. Let us assume that ω_0 is an 8-th primitive root of unity (thus ${\omega_0}^8 = 1$). Also, let $\omega_1 = {\omega_0}^2$, thus a 4-th primitive root of unity (then, ${\omega_1}^4 = {\omega_0}^8 = 1$).

$$\text{DFT}_8(\omega_0) = L^8_2(I_4 \otimes \text{DFT}_2)\, L^8_4\, D_{2,4}\,(I_2 \otimes \text{DFT}_4)\, L^8_2 \tag{11}$$

$$\text{DFT}_4(\omega_1) = L^4_2\,(I_2 \otimes \text{DFT}_2)\, L^4_2\, D_{2,2}\,(I_2 \otimes \text{DFT}_2)\, L^4_2. \tag{12}$$

Substituting Equation (12) in Equation (11), we have:

$$\begin{aligned}\text{DFT}_8(\omega_0) =& L^8_2\,(I_4 \otimes \text{DFT}_2)\, L^8_4\, D_{2,4}\,(I_2 \otimes L^4_2)(I_4 \otimes \text{DFT}_2) \\ & (I_2 \otimes L^4_2)(I_2 \otimes D_{2,2})(I_4 \otimes \text{DFT}_2)(I_2 \otimes L^4_2)(L^8_2).\end{aligned} \tag{13}$$

The unrolled Equation (13) follows from a sequence of basic operations, which helps us in the following ways. First, we avoid performing the permutation and actually moving data around. Instead,

Algorithm 2 Computing DFT on K^e points in $\mathbb{Z}/p\mathbb{Z}$.

input:
- size of the base-case K (8, 16, 32, 64, 128, or 256),
- a positive integer e,
- a vector $\vec{x}$ of size K^e,
- ω which is a K^e-th primitive root of unity in $\mathbb{Z}/p\mathbb{Z}$.

output:
- the final result stored in $\vec{x}$

procedure DFT_GENERAL($\vec{x}$, K, e, ω,)
  for $0 \le i < e-1$ **do**
    for $0 \le j < K^i$ **do** ▷ Can be replaced with Parallel-For.
      stride_permutation($x_{jK^{e-i}}$, K, K^{e-i-1}) ▷ Step 1
    end for
  end for
  $\omega_a := \omega^{K^{e-1}}$
  for $0 \le j < K^{e-1}$ **do** ▷ Can be replaced with Parallel-For.
    idx := jK
    DFT_K(x_{idx}, ω_a) ▷ Step 2
  end for
  for $e-2 \ge i \ge 0$ **do**
    $\omega_i := \omega^{K^i}$
    for $0 \le j < K^i$ **do** ▷ Can be replaced with Parallel-For.
      idx := jK^{e-i}
      twiddle(x_{idx}, K^{e-i-1}, K, ω_i) ▷ Step 3
      stride_permutation(x_{idx}, K^{e-i-1}, K) ▷ Step 4
    end for
    for $0 \le j < K^{e-1}$ **do** ▷ Can be replaced with Parallel-For.
      idx := jK
      DFT_K(x_{idx}, ω_a) ▷ Step 5
    end for
    for $0 \le j < K^i$ **do** ▷ Can be replaced with Parallel-For.
      idx := jK^{e-i}
      stride_permutation(x_{idx}, K, K^{e-i-1}) ▷ Step 6
    end for
  end for
end procedure

we precompute the position of elements after each permutation and hard-code those values in the algorithm for computing the base-case. Also, we reduce the number of multiplications in the base-case. Moreover, each multiplication in the base-case can be reduced to a cyclic shift (as explained in Section 2).

4.3.1 Avoiding stride permutations in DFT_K. In our example for DFT_8, there are 4 permutation steps in Equation (13). We begin by the two right-most ones, $(I_2 \otimes L^4_2)(L^8_2)$. Rather than moving the data, we precompute the position of permuted elements. Let $\vec{M} = (0, 1, 2, 3, 4, 5, 6, 7)$ be the vector containing the initial position of the elements of $\vec{x}$. Then,

$$\vec{M}_1 = L^8_2 \vec{M} = (0, 2, 4, 6, 1, 3, 5, 7) \tag{14}$$

$$\vec{M}_2 = (I_2 \otimes L^4_2)\vec{M}_1 = (0, 4, 2, 6)(1, 5, 3, 7) \tag{15}$$

Moving from right to left in Equation (13), when we reach $I_4 \otimes \text{DFT}_2$ (the third statement in Equation (13)), we apply four DFT_2's on elements of $\vec{x}$, while we retrieve the order of elements as recorded in M_2:

$$\text{DFT}_2(0, 4) \rightarrow \text{DFT}_2(2, 6) \rightarrow \text{DFT}_2(1, 5) \rightarrow \text{DFT}_2(3, 7) \tag{16}$$

Following this trend, we reach L^8_4 and L^8_2 on the left-most side of Equation (13):

$$\vec{M}_3 = (L^8_4)\vec{M}_2 = (0, 1, 4, 5, 2, 3, 6, 7) \tag{17}$$

$$\vec{M}_4 = (L^8_2)\vec{M}_3 = (0, 4, 2, 6, 1, 5, 3, 7) \tag{18}$$

At the very end, we need to swap some elements of $\vec{x}$ in order to correct their position in the result vector. That means the position of elements in the result vector must be updated from what they are in $\vec{M}_4$ to the values in M_{out} in the following way:

$$\begin{array}{rcccccccc} \vec{M}_4 & = & (0, & 4, & 2, & 6, & 1, & 5, & 3, & 7) \\ & & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow \\ \vec{M_{out}} & = & (0, & 1, & 2, & 3, & 4, & 5, & 6, & 7) \end{array}$$

Here, rather than permuting the whole vector, we only need to swap the elements that are shown in the same color. For case of DFT_8, we end up swapping only 4 out of 8 elements.

4.3.2 Twiddle multiplications in the DFT_K. Remember Equation (8):

$$D_{K,J} = \bigoplus_{j=0}^{K-1} diag(1, \omega_i^j, \ldots, \omega_i^{j(J-1)})$$

Then, we have the following twiddle matrices as part of DFT_8:

$$D_{2,2} = (1, 1, \omega_1^0, \omega_1^1) \tag{19}$$

$$D_{2,4} = (1, 1, 1, 1, \omega_0^0, \omega_0^1, \omega_0^2, \omega_0^3) \tag{20}$$

As we are computing over $\mathbb{Z}/p\mathbb{Z}$ where the prime is $p = r^4 + 1$, then, the radix r is the 8-th root of unity, therefore, can be used for computation of DFT_8. Let $\omega_0 = r$ and $\omega_1 = r^2$, then, the twiddle matrices are updated as follows:

$$D_{2,4} = (1, 1, 1, 1, 1, r, r^2, r^3) \tag{21}$$

$$D_{2,2} = (1, 1, 1, r^2) \tag{22}$$

We see that more than half of the multiplications in the DFT_8 are by 1 and do not require any actual computation.

More importantly, the multiplications by the powers of the radix are done by cyclic shift from Section 2. In a similar way, this argument is valid for any DFT_K as long as we are computing modulo a generalized Fermat prime of the form $p = r^k + 1$.

At the end, putting all the optimizations together, and following Equation (13) from right to left, we get an unrolled algorithm presented in Algorithm 3 for computing DFT_8. The algorithm computes the DFT of a vector of size 8 over a generalized Fermat prime in the form of $p = r^4 + 1$, note that r is an 8-th primitive root of unity of p. Following the above process, we have implemented base-cases for K equal to 8, 16, 32, 64, 128, and 256 in the BPAS library. We believe that the currently implemented base-case sizes are large enough for real world applications. Thus, we have skipped prime sizes larger than 128 machine-words in our current implementation.

Algorithm 3 Unrolled base-case DFT_8 over $\mathbb{Z}/p\mathbb{Z}$ for $p = r^4 + 1$.

procedure DFT8($\vec{x}$,r)
  DFT2(x_0, x_4); DFT2(x_2, x_6); ▷ DFT on permuted indexes.
  DFT2(x_1, x_5); DFT2(x_3, x_7); ▷ DFT on permuted indexes.
  $x_6 := x_6\, r^2$; ▷ Twiddle multiplication $x_6\, r^2$.
  $x_7 := x_7\, r^2$; ▷ Twiddle multiplication $x_7\, r^2$.
  DFT2(x_0, x_2); DFT2(x_4, x_6); ▷ DFT on permuted indexes.
  DFT2(x_1, x_3); DFT2(x_5, x_7); ▷ DFT on permuted indexes.
  $x_5 := x_5\, r^1$; ▷ Twiddle multiplication $x_5\, r^1$.
  $x_3 := x_3\, r^2$; ▷ Twiddle multiplication $x_3\, r^2$.
  $x_7 := x_7\, r^3$; ▷ Twiddle multiplication $x_7\, r^3$.
  DFT2(x_0, x_1); DFT2(x_4, x_5); ▷ DFT on permuted indexes.
  DFT2(x_2, x_3); DFT2(x_6, x_7); ▷ DFT on permuted indexes.
  Swap(x_1, x_4); Swap(x_3, x_6); ▷ Final permutation.
  return $\vec{x}$;
end procedure

5 EXPERIMENTATION

In this section, first, we briefly describe the setup used in our experimentation. Then, in Section 5.2, we present the comparison of the two implementations of the multiplication in $\mathbb{Z}/p\mathbb{Z}$ introduced in Section 3. Section 5.3 reports on the results for computing FFT over

the big prime fields with the BPAS library. Finally, in Section 5.4, we analyze speedup that we gain for parallelizing each approach. All the experimental results have been verified using equivalent code written in GMP [14].

5.1 Experimental setup

Table 1 provides the set of prime numbers we use for different base-cases. The k is between 4 and 128 (i.e. up to 128 machine-words).

We have used two node configurations for our benchmarking purposes. The first configuration which we refer to as `Intel-i7-7700K`, has an Intel-i7-7700K 4-core processor (with 8 threads when hyper-threading is enabled), clocking at 4.50 GHz, and equipped with 16 GB of memory (clocking at 2133 MHz). The second configuration which we refer to as `Xeon-X5650` has an Intel Xeon-X5650 processor with 6 physical cores (and 12 threads when hyper-threading is enabled) clocking at 2.66 GHz, and is equipped with 48 GB of memory (clocking at 1133 MHz).

Table 1: The set of big primes of different sizes which are used for experimentations.

prime	$K(=2k)$	k	r
P_4	8	4	$2^{59}+2^{58}+2^{11}$
P_8	16	8	$2^{59}+2^{57}+2^{39}$
P_{16}	32	16	$2^{58}+2^{55}+2^{45}$
P_{32}	64	32	$2^{58}+2^{55}+2^{17}$
P_{64}	128	64	$2^{57}+2^{56}+2^{11}$
P_{128}	256	128	$2^{57}+2^{52}+2^{20}$

5.2 Multiplication in generalized Fermat prime fields

As discussed in Section 3, we provide an algorithm for multiplying two arbitrary elements of the generalized Fermat prime field $\mathbb{Z}/p\mathbb{Z}$ (referred to as GFPF) which relies on negacyclic convolution using DFTs over small prime fields. Our goal is to compare the running-time of our approach with that of the integer arithmetic provided by GMP [14]. To this end, we provide the same input data to both multiplication functions (randomly generated data, but the same data passed to all experiments), the multiplication is carried out, and at the end, the results are verified.

Table 2 shows the time (in milliseconds) spent in computation of 10^6 multiplications using each of the two implementations (the number 10^6 is chosen as an input size which is large enough to reduce the errors in time measurement). Also, Table 2 shows the running-time ratio of GFPF versus the GMP multiplications. The experimentation has been conducted on `Intel-i7-7700K`. We observe that the GFPF implementation is slower than GMP multiplication, however, the GFPF multiplication becomes faster as the value of k increases.

Recall that the GFPF multiplication has four steps (see Section 3.2):

I. negacyclic convolution (includes converting the vector into Montgomery representation),
II. Chinese remainder algorithm (includes converting the vector out from Montgomery representation),
III. LHC algorithm (fast division of a three machine-word number by radix r), and

Table 2: The running-time of computing 10^6 modular multiplications in $\mathbb{Z}/p\mathbb{Z}$ for P_8, P_{16}, P_{32}, and P_{64} (measured on `Intel-i7-7700K`).

prime	k	GFPF	GMP	Ratio ($\frac{t_{GFPF}}{t_{GMP}}$)
P_8	8	645 (ms)	171(ms)	3.77x
P_{16}	16	1318 (ms)	417 (ms)	3.16x
P_{32}	32	2852 (ms)	1179 (ms)	2.41x
P_{64}	64	6101 (ms)	3452 (ms)	1.76x

IV. cyclic shift, addition, and normalization (carry-handling).

Table 3 shows the percentage of time spent in each step of the GFPF multiplication during multiplication of 10^6 arbitrary elements of $\mathbb{Z}/p\mathbb{Z}$, for primes P_8, P_{16}, P_{32}, and P_{64}, collected on `Intel-i7-7700K`. It also presents the actual running-time (shown in milliseconds); clearly, computing the convolution is the dominant cost.

Table 3: Time (in milliseconds) and percentage (%) of the total time spent in different steps of computing 10^6 GFPF multiplications of arbitrary elements in $\mathbb{Z}/p\mathbb{Z}$ for primes P_8, P_{16}, P_{32}, and P_{64} (measured on `Intel-i7-7700K`).

prime	k	Convolution		CRT		LHC		Normalization	
		Time	%	Time	%	Time	%	Time	%
P_8	8	323	45	150	21	208	29	35	5
P_{16}	16	851	52	288	18	425	26	64	4
P_{32}	32	2083	57	563	15	847	23	177	5
P_{64}	64	4751	61	1115	14	1497	19	434	6

5.3 FFT over big prime fields

In this section, we provide experimental data for computing FFTs over big prime fields. As we have explained in Section 4, our FFT implementations which compute DFT on a vector of size $N = K^e$ over $\mathbb{Z}/p\mathbb{Z}$ (with $p = r^k + 1$) are based on Algorithm 2. We compare the running-time of our GFPF implementation versus the GMP implementation, both executed in serial. Once more, we compare the running-time of the two implementations, this time both executed in parallel.

Table 4 provides the running-time and running-time ratio for our generalized Fermat prime fields (GFPF) based implementation versus the GMP implementation of computing FFT of size $N = K^e$ over $\mathbb{Z}/p\mathbb{Z}$ (for primes P_4, P_8, P_{16}, P_{32}, P_{64}, and P_{128}) in sequential and parallel mode. We skip the case of $N = K^3$ for P_{128} ($K = 256$) as it is too large to fit in the memory of either of our compute nodes. All measurements are completed on `Intel-i7-7700K`. Table 5 provides similar comparisons measured on `Xeon-X5650`. In the case of `Xeon-X5650`, we observe that with more cores and threads, our parallel GFPF implementation gains more speedup compared to the parallel GMP implementation. For both the serial and parallel cases, we find our implementation using GFPF multiplication is faster than GMP in most cases.

5.4 Performance analysis of FFT implementations

In this section we compare the parallel speedup factors for each of GFPF and GMP approaches compared to their corresponding serial implementations. From the previous section and Table 2, we know that the GFPF multiplication of two arbitrary elements in

Table 4: The running-time (in milliseconds) and ratio ($t_{\text{GFPF}}/t_{\text{GMP}}$) of serial and parallel computation of FFT on vectors of size $N = K^e$ over $\mathbb{Z}/p\mathbb{Z}$ for P_4, P_8, P_{16}, P_{32}, P_{64}, and P_{128} (measured on `Intel-i7-7700K`).

prime	k	K	e	Serial			Parallel		
				GFPF	GMP	$\frac{t_{\text{GFPF}}}{t_{\text{GMP}}}$	GFPF	GMP	$\frac{t_{\text{GFPF}}}{t_{\text{GMP}}}$
P_4	4	8	2	0.019	0.030	0.63x	0.057	0.118	0.48x
P_4	4	8	3	0.314	0.363	0.86x	0.215	0.276	0.77x
P_8	8	16	2	0.181	0.202	0.89x	0.117	0.143	0.81x
P_8	8	16	3	5.771	5.486	1.05x	1.603	2.247	0.71x
P_{16}	16	32	2	1.644	1.730	0.95x	0.513	0.693	0.74x
P_{16}	16	32	3	103.423	104.620	0.98x	24.052	35.017	0.68x
P_{32}	32	64	2	14.815	20.341	0.72x	3.507	5.411	0.64x
P_{32}	32	64	3	1922.373	2431.867	0.79x	462.746	702.163	0.65x
P_{64}	64	128	2	140.995	278.188	0.50x	33.507	69.879	0.47x
P_{128}	128	256	2	580.961	3745.353	0.15x	154.064	905.799	0.17x

Table 5: The running-time (in miliseconds) and ratio ($t_{\text{GFPF}}/t_{\text{GMP}}$) of serial and parallel computation of FFT on vectors of size $N = K^e$ over $\mathbb{Z}/p\mathbb{Z}$ for P_4, P_8, P_{16}, P_{32}, P_{64}, and P_{128} (measured on `Xeon-X5650`).

prime	k	K	e	Serial			Parallel		
				GFPF	GMP	$\frac{t_{\text{GFPF}}}{t_{\text{GMP}}}$	GFPF	GMP	$\frac{t_{\text{GFPF}}}{t_{\text{GMP}}}$
P_4	4	8	2	0.051	0.071	0.71x	0.155	0.114	1.35x
P_4	4	8	3	0.843	0.917	0.91x	0.452	0.577	0.78x
P_8	8	16	2	0.472	0.546	0.86x	0.217	0.320	0.67x
P_8	8	16	3	16.661	15.231	1.09x	2.837	4.806	0.59x
P_{16}	16	32	2	4.444	5.085	0.87x	0.877	1.371	0.63x
P_{16}	16	32	3	284.080	297.904	0.95x	41.012	66.635	0.61x
P_{32}	32	64	2	39.809	64.307	0.61x	5.701	11.640	0.48x
P_{32}	32	64	3	4674.079	6501.669	0.71x	696.311	1289.061	0.54x
P_{64}	64	128	2	376.450	909.041	0.41x	53.578	140.610	0.38x
P_{128}	128	256	2	1395.310	13371.369	0.10x	240.362	1811.282	0.13x

$\mathbb{Z}/p\mathbb{Z}$ is slower than the GMP implementation. At the same time, Table 4 and 5 indicate that computing FFT on large vectors over $\mathbb{Z}/p\mathbb{Z}$ using GFPF multiplication turns out to be faster than GMP arithmetic in most cases, for both serial and parallel modes. This interesting result can be explained as follows.

When we compute DFT using generalized Fermat prime field arithmetic (including GFPF multiplication), the majority of the multiplications are performed in the base-cases, which are carried out in linear time through cyclic shift (a sequence of data movement, subtraction, and carry handling; see Section 2). Meanwhile, in the case of GMP arithmetic, all of the multiplications are done using the same function calls, with no consideration for the cheap multiplications in the base-case DFT$_K$.

Figure 1 presents the ratio of time spent in one modular multiplication operation in FFT over $\mathbb{Z}/p\mathbb{Z}$ on vectors of size $N = K^e$ between the GFPF implementation and the GMP arithmetic. We see that for the GFPF implementation the average time spent in one modular multiplication is much lower than the time spent in the same operation using GMP arithmetic.

This result agrees with our estimation of increased performance due to Fürer's trick [13]. As it is demonstrated, by using cyclic shift for performing cheap multiplications in the base-case, we can lower the average time spent in multiplications, resulting in faster computation of the base-case DFT$_K$'s, and consequently, speed up the computation of the whole FFT over $\mathbb{Z}/p\mathbb{Z}$.

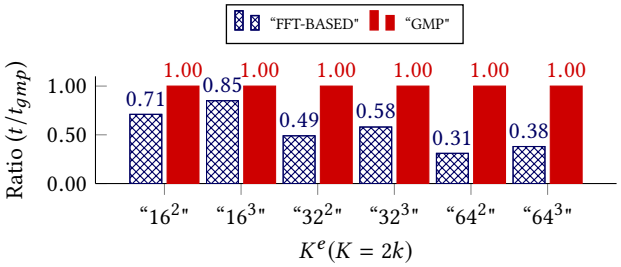

Figure 1: Ratio (t/t_{gmp}) of average time spent in one multiplication operation measured during computation of FFT over $\mathbb{Z}/p\mathbb{Z}$ on vectors of size $N = K^e$.

Now, we take a closer look at the steps involved in the DFT computation. Table 6 provides the running-time data for every step of computing a DFT of size $N = K^3$ ($K = 64$) over $\mathbb{Z}/p\mathbb{Z}$ for prime P_{32}. The timings are measured for both implementations on `Intel-i7-7700K`. As we observe, for both implementations in the serial mode, the time spent in precomputation and stride permutation is negligible compared to the time spent in twiddle multiplications and the base-case DFT$_K$'s. Also, parallelization has little impact on precomputation and stride permutation. In contrast, parallelization significantly improves the time spent in twiddle multiplications and the base-case DFT$_K$'s for both approaches.

Finally, Table 7 comapres the parallel speedup ratios for each implemenentation on both `Intel-i7-7700K` and `Xeon-X5650`. This table indicates that the parallelization of the GFPF implementation appears to be slightly more successful than the parallelization of the GMP implementation. This difference in the performance can be attributed to, by our measurements, the sharp management of computing resources (i.e. specialized arithmetic and minimal usage of memory). We have repeated the same benchmarks with hyper-threading disabled; this is also shown in Table 7. With hyper-threading disabled the speedup drops slightly, nevertheless, both implementations still gain nearly linear parellel speedup.

Table 6: Time spent (milliseconds) in different steps of serial and parallel computation of DFT **of size $N = K^3$ over $\mathbb{Z}/p\mathbb{Z}$, for prime P_{32} ($K = 2k = 64$) measured on `Intel-i7-7700K`.**

Mode	Variant	Precomputation	Permutation	DFT$_K$	Twiddle
Serial	GFPF	14 (ms)	72 (ms)	444 (ms)	1406 (ms)
	GMP	6 (ms)	177 (ms)	1229 (ms)	1026 (ms)
Parallel	GFPF	14 (ms)	51 (ms)	82 (ms)	330 (ms)
	GMP	6 (ms)	181 (ms)	284 (ms)	237 (ms)

Table 7: Ratio ($t_{serial}/t_{parallel}$) for serial vs. parallel execution of each implementation for $N = K^e$ ($K = 2k$, $e = 3$) measured on both `Intel-i7-7700K` and `Xeon-X5650` with and without hyper-threading enabled.

k	`Intel-i7-7700K`				`Xeon-X5650`			
	+ Hyper-threading		- Hyper-threading		+ Hyper-threading		- Hyper-threading	
	GFPF	GMP	GFPF	GMP	GFPF	GMP	GFPF	GMP
4	1.37x	1.25x	1.57x	1.31x	1.87x	1.59x	2.35x	1.95x
8	3.64x	2.44x	3.36x	2.34x	5.52x	3.17x	4.99x	3.40x
16	4.31x	2.96x	3.77x	2.69x	6.93x	4.47x	5.66x	4.16x
32	4.15x	3.48x	3.67x	3.07x	6.71x	5.04x	5.65x	4.47x

6 CONCLUSIONS AND FUTURE WORK

We have presented an implementation of Fast Fourier Transforms over generalized Fermat prime fields on multi-threaded processors. Our parallel implementations using both specialized arithmetic and integer arithmetic from the GMP library achieve nearly linear parallel speedup. We noticed that the parallelization of our specialized implementation is slightly more successful than our GMP implementation. We attribute this higher performance to reduced number of arithmetic instructions due to using specialized arithmetic, minimal memory usage, and unrolling base-case DFT's and hard coding the constants.

Our results prove that developing specialized arithmetic (e.g. Montgomery multiplication, Barret reduction, cyclic shift introduced in Section 2 and using inline assembly) can be beneficial. Doing so leads to reduced overhead compared to a more generic implementation such as large integer arithmetic functions available in GMP, or other libraries on top of GMP. Unrolling the base-case DFT's improves the performance for two main reasons. First, by removing the majority of permutations (all except the last swap), it minimizes data movement. Second, compared to a naive implementation, a hard coded base-case reduces the number of multiplications by a power of radix to less than half (by simply avoiding the multiplications by 1 in the first place). Designing our implementation based on the iterative six step FFT algorithm was crucial; it allowed for more a finely scheduled parallelization on multi-core CPUs which obtains good speedup.

As part of our future work we should extend our implementation to arbitrary vector sizes, that is, the cases where the size N is not in the form K^e. Also, we must consider how to apply our approach to very large input sizes, for example, when the input vectors are too large to fit into main memory. Finally, we need to address another bottleneck of the current implementation, that is, the arbitrary multiplication in the generalized Fermat prime fields. We need a better solution for the multiplication between two polynomials with 64-bit integer coefficients; indeed, such a multiplication can result in coefficeints up to 192 bits, requiring multi-precision arithmetic.

Acknowledgments

The authors would like to thank IBM Canada Ltd (CAS project 880) and NSERC of Canada (CRD grant CRDPJ500717-16).

REFERENCES

[1] Ayaz Ali, Lennart Johnsson, and Jaspal Subhlok. 2007. Scheduling FFT Computation on SMP and Multicore Systems. In *Proceedings of the 21st Annual International Conference on Supercomputing (ICS '07)*. ACM, New York, NY, USA, 293–301.

[2] Elizabeth A. Arnold. 2003. Modular Algorithms for Computing Gröbner Bases. *Journal of Symbolic Computation* 35, 4 (2003), 403–419.

[3] Mohammadali Asadi, Alexander Brandt, Changbo Chen, Svyatoslav Covanov, Farnam Mansouri, Davood Mohajerani, Robert Moir, Marc Moreno Maza, Linxiao Wang, Ning Xie, and Yuzhen Xie. 2019. Basic Polynomial Algebra Subprograms (BPAS). http://www.bpaslib.org.

[4] Paul Barrett. 1986. Implementing the Rivest Shamir and Adleman public key encryption algorithm on a standard digital signal processor. In *Conference on the Theory and Application of Cryptographic Techniques*. Springer, 311–323.

[5] Liangyu Chen, Svyatoslav Covanov, Davood Mohajerani, and Marc Moreno Maza. 2017. Big Prime Field FFT on the GPU. In *Proceedings of the 2017 ACM on International Symposium on Symbolic and Algebraic Computation, ISSAC*. 85–92.

[6] Svyatoslav Covanov. 2014. *Putting Fürer Algorithm into practice*. Technical Report. ORCCA Lab, London.

[7] Svyatoslav Covanov and Emmanuel Thomé. 2018. Fast integer multiplication using generalized Fermat primes. *Mathematics of Computation* (2018). https://hal.inria.fr/hal-01108166

[8] Xavier Dahan, Marc Moreno Maza, Éric Schost, Wenyuan Wu, and Yuzhen Xie. 2005. Lifting techniques for triangular decompositions. In *ISSAC 2005, Proceedings*, M. Kauers (Ed.). ACM, 108–115.

[9] Anindya De, Piyush P. Kurur, Chandan Saha, and Ramprasad Saptharishi. 2008. Fast integer multiplication using modular arithmetic. In *STOC*. 499–506.

[10] Anindya De, Piyush P Kurur, Chandan Saha, and Ramprasad Saptharishi. 2013. Fast Integer Multiplication Using Modular Arithmetic. *SIAM J. Comput.* 42, 2 (2013), 685–699.

[11] Franz Franchetti and Markus Püschel. 2011. FFT (Fast Fourier Transform). In *Encyclopedia of Parallel Computing*. 658–671.

[12] Franz Franchetti, Yevgen Voronenko, and Markus Püschel. 2006. Tools and techniques for performance - FFT program generation for shared memory: SMP and multicore. In *Proceedings of the ACM/IEEE SC2006 Conference on High Performance Networking and Computing, November 11-17, 2006, Tampa, FL, USA*. 115.

[13] Martin Fürer. 2009. Faster Integer Multiplication. *SIAM J. Comput.* 39, 3 (2009), 979–1005.

[14] Torbjörn Granlund and the GMP development team. 2012. *GNU MP: The GNU Multiple Precision Arithmetic Library* (5.0.5 ed.). http://gmplib.org/.

[15] David Harvey, Joris van der Hoeven, and Grégoire Lecerf. 2017. Faster Polynomial Multiplication over Finite Fields. *J. ACM* 63, 6, Article 52 (Jan. 2017), 23 pages.

[16] David Harvey, Joris van der Hoeven, and Grégoire Lecerf. 2016. Even faster integer multiplication. *Journal of Complexity* 36 (2016), 1–30.

[17] David Harvey, Joris van der Hoeven, and Grégoire Lecerf. 2016. Fast Polynomial Multiplication over F260. In *Proceedings of the ACM on International Symposium on Symbolic and Algebraic Computation (ISSAC '16)*. ACM, New York, NY, USA, 255–262.

[18] Marc Moreno Maza and Yuzhen Xie. 2009. FFT-Based Dense Polynomial Arithmetic on Multi-cores. In *HPCS (Lecture Notes in Computer Science)*, Vol. 5976. Springer, 378–399.

[19] Lingchuan Meng, Yevgen Voronenko, Jeremy R. Johnson, Marc Moreno Maza, Franz Franchetti, and Yuzhen Xie. 2010. Spiral-generated modular FFT algorithms. In *PASCO*. 169–170.

[20] Peter L. Montgomery. 1985. Modular multiplication without trial division. *Mathematics of computation* 44, 170 (1985), 519–521.

[21] Arnold Schönhage and Volker Strassen. 1971. Schnelle Multiplikation großer Zahlen. *Computing* 7, 3-4 (1971), 281–292.

[22] Joachim von zur Gathen and Jürgen Gerhard. 2013. *Modern Computer Algebra (3. ed.)*. Cambridge University Press.