

Big Prime Field FFT on Multi-core Processors

Svyatoslav Covanov¹ Davood Mohajerani²
Marc Moreno Maza^{2,3} Linxiao Wang^{2,3}

¹University of Lorraine, France

²ORCCA, University of Western Ontario, Canada

³IBM Center for Advanced Studies, Markham, Canada

ISSAC 2019, Beihang University, Beijing, China
July 16, 2019

Outline

- 1 Introduction
- 2 Generalized Fermat prime field arithmetic
- 3 Implementation of the multiplication in $\mathbb{Z}/p\mathbb{Z}$
- 4 Implementation of the FFT
- 5 Experimental results
- 6 Conclusions

Outline

- 1 Introduction
- 2 Generalized Fermat prime field arithmetic
- 3 Implementation of the multiplication in $\mathbb{Z}/p\mathbb{Z}$
- 4 Implementation of the FFT
- 5 Experimental results
- 6 Conclusions

Background

Big prime field FFT on GPUs (Chen, Covanov, Mohajerani, and Moreno Maza, ISSAC 2017)

- CUDA implementation for arithmetic operations and FFT in $\mathbb{Z}/p\mathbb{Z}$, where p is a Generalized Fermat prime of size 8 or 16 machine words.
- GPUs are suitable for **fine-grained parallelism** as GPU architectures offer a **fine control of hardware resources**.

Big prime fields are attractive

- Modular methods (e.g. in polynomial system solving) can take advantage of primes larger than the machine-word size.
- The work reported in the above mentioned ISSAC 2017 paper suggests that computing over a single **big** prime field can outperform computing over several **small** prime fields.

Big prime field FFT on Multicores

- GPU implementation techniques can not be easily ported and applied to the context of multi-core processors.
- On multi-cores: much higher overhead for thread management, memory levels managed by hardware and OS instead of programmer.

Challenges and contributions

Challenges

- Can a serial CPU implementation take advantage of the properties of the Generalized Fermat Prime Fields (GFPF) towards an efficient implementation of FFT over those fields?
- Can we implement an efficient multi-threaded FFT over such big prime fields on multi-core processors?

Contributions

- Fast arithmetic in GFPFs, including fast multiplication of two arbitrary elements.
- Parallel implementation of FFT over GFPFs for multi-core processors.
- Our implementation is part of the BPAS library which is publicly available at <http://www.bpaslib.org/>

Motivation of our project: an idea of Martin Fürer

Assumptions

- Let p be a k -machine-word prime and $N > 0$ an integer dividing $p - 1$.
- Consider the FFT of a vector of size N over the prime field $\mathbb{Z}/p\mathbb{Z}$.
- Assume $N = K^e$ for some “small” K (say $K = 32$) and an integer $e \geq 2$.
- Let ω be a N -th primitive root of unity in $\mathbb{Z}/p\mathbb{Z}$ and let $\eta = \omega^{N/K}$.
- Assume that multiplying an arbitrary element of $\mathbb{Z}/p\mathbb{Z}$ by η^i ($0 \leq i \leq K$) can be done within $O(k)$ word ops. This assumption can hold when p is a GFP.

Consequences

- Then, every arithmetic operation involved in DFT_K , that is, an FFT on K points of $\mathbb{Z}/p\mathbb{Z}$, can be done within $O(k)$ word ops.
- Therefore, DFT_K can be performed within $O(K \log(K) k)$ word ops instead of the “a priori” $O(K \log(K) M(k))$ word ops.
- Under our hypotheses, unrolling Cooley-Tukey formula, it follows that DFT_N can be performed within $O(N \log_2(N) k + N \log_k(N) k \log_2(k))$ word ops instead of $O(N \log_2(N) k + N k \log_2^2(k))$ word ops for small-primes+CRT.

Related work

FFT over finite fields

- For more than two decades, NTL has been a reference library for univariate polynomials over \mathbb{Z} and finite fields; in the big prime field case (thus for multi-precision arithmetic) NTL falls back to GMP.
- Other computer algebra systems devote effort to polynomial arithmetic over finite fields. Among them: FLINT, Magma, Mathemagix, etc.
- Up to our knowledge, there are no specific implementations of FFT over big prime fields.

Implementation techniques for FFT in scientific computing

- The FFTW and SPIRAL projects have extensively investigated techniques for code generation of FFT kernels. They have inspired us in this work.

Fast algorithms for polynomial multiplication

- The quest for faster polynomial/integer multiplication initiated by Martin Fürer and extended by others including Harvey, van der Hoeven and Lecerf [2016,...,2019] and Covanov and Thomé [2019] has inspired this work too.
- More references can be found in the article.

Outline

- 1 Introduction
- 2 Generalized Fermat prime field arithmetic
- 3 Implementation of the multiplication in $\mathbb{Z}/p\mathbb{Z}$
- 4 Implementation of the FFT
- 5 Experimental results
- 6 Conclusions

Generalized Fermat prime

Generalized Fermat prime

- Prime in the form of $p = r^k + 1$, where k is some power of 2 and the radix r is a machine-word size integer. Also, r is a $2k$ -th primitive root of unity modulo p .
- From now on, p is a Generalized Fermat prime and we refer to $\mathbb{Z}/p\mathbb{Z}$ as a GFPP.

Representing elements of $\mathbb{Z}/p\mathbb{Z}$

An element $x \in \mathbb{Z}/p\mathbb{Z}$ is represented in one of the following equivalent ways:

- by a vector $\vec{x} = (x_{k-1}, \dots, x_0)$ of length k

$$x \equiv x_{k-1} r^{k-1} + x_{k-2} r^{k-2} + \dots + x_1 r + x_0 \pmod{p}$$

- by a polynomial $f_x \in \mathbb{Z}[R]$,

$$f_x = \sum_{i=0}^{k-1} x_i R^i$$

such that $x \equiv f_x(r) \pmod{p}$.

- Either way, we have two cases for the value of x :

- ① When $x \equiv p - 1 \pmod{p}$ holds, we have $x_{k-1} = r$ and $x_{k-2} = \dots = x_0 = 0$.
- ② When $0 \leq x < p - 1$ holds, we have $0 \leq x_i < r$ for $i = 0, \dots, k - 1$.

Using the radix representation

- Addition and multiplication can be computed like grade school arithmetic.
- p is too small in practice for considering multi-threaded addition or multiplication.

$$\begin{array}{r} 28 \\ +45 \\ \hline 3 \end{array} \quad \begin{array}{c} 1 \\ \Rightarrow \\ \hline 73 \end{array} \qquad \begin{array}{r} 432 \\ \times 211 \\ \hline 432 \\ 864 \\ \hline 91152 \end{array}$$

Multiplication of two arbitrary elements

- For $x, y \in \mathbb{Z}/p\mathbb{Z}$, multiplying $x \cdot y$ in $\mathbb{Z}/p\mathbb{Z}$, is done computing $f_x(R) \cdot f_y(R)$ in $\mathbb{Z}[R]/\langle R^k + 1 \rangle$ followed by a conversion into the radix r representation.
- **Challenge:** How to multiply the two polynomials $f_x(R) \cdot f_y(R)$ efficiently, when the size of the intermediate products $x_i y_j$ can be larger than one machine word?

Cheap multiplication: multiplying by a power of radix r

- For an arbitrary element $x \in \mathbb{Z}/p\mathbb{Z}$, we want to compute $x \cdot r^i \pmod p$, for $0 \leq i < k$.
- Computing modulo $p = r^k + 1$, we can replace every r^k by -1 .

- For $0 < i < k$

$$\begin{aligned}xr^i &\equiv (x_{k-1} r^{k-1+i} + \dots + x_0 r^i) \pmod p \\ &\equiv \left(\sum_{h=i}^{h=k-1} x_{h-i} r^h - \sum_{h=k}^{h=k-1+i} x_{h-i} r^{h-k} \right) \pmod p\end{aligned}$$

- We reduce a multiplication to a subtraction.

Outline

- 1 Introduction
- 2 Generalized Fermat prime field arithmetic
- 3 Implementation of the multiplication in $\mathbb{Z}/p\mathbb{Z}$
- 4 Implementation of the FFT
- 5 Experimental results
- 6 Conclusions

Computing $f_x(R) \cdot f_y(R)$

- Treat elements in $\mathbb{Z}/p\mathbb{Z}$ as polynomials over \mathbb{Z} .
- Using FFT over small prime fields to multiply two polynomials.
- Normalizing the result polynomial into an element in $\mathbb{Z}/p\mathbb{Z}$.
- Convolution computes $f_x \cdot f_y \pmod{R^k - 1}$.
- However, we need to compute $f_x \cdot f_y \pmod{R^k + 1}$.

Negacyclic convolution computes $f_u = f_x \cdot f_y \pmod{R^k + 1}$ in a field:

- Assume that θ is a $2k$ -th primitive root of unity.
- With $\vec{A} = (1, \theta, \dots, \theta^{k-1})$ and $\vec{A}' = (1, \theta^{-1}, \dots, \theta^{1-k})$, **negacyclic convolution** computes:

$$\vec{u} = \vec{A}' \cdot \text{InverseDFT}(\text{DFT}(\vec{A} \cdot \vec{x}) \cdot \text{DFT}(\vec{A} \cdot \vec{y}))$$

Each $|u_i|$ is at most kr^2 ; it can exceed the size of a machine word!

$$\begin{aligned}
 f_u(R) &= f_x(R) \cdot f_y(R) \pmod{(R^k + 1)} \\
 &= \sum_{m=0}^{2k-2} \sum_{0 \leq i, j < k} x_i y_j R^m \pmod{(R^k + 1)} \\
 &= (x_{k-1} y_0 + x_{k-2} y_1 + x_{k-3} y_2 + \cdots + x_1 y_{k-2} + x_0 y_{k-1}) R^{k-1} \\
 &\quad + (x_{k-2} y_0 + x_{k-3} y_1 + \cdots + x_1 y_{k-3} + x_0 y_{k-2} - x_{k-1} y_{k-2}) R^{k-2} \\
 &\quad \dots \\
 &\quad + (x_0 y_0 - x_{k-1} y_1 - \cdots - x_1 y_{k-1}) \\
 &= \sum_{m=0}^{k-1} \left(\sum_{0 \leq i, j < k} x_i y_j - \sum_{0 \leq i, j < k} x_i y_j \right) R^m
 \end{aligned}$$

Implementation of the FFT-based multiplication in BPAS(2/4)

CRT step

- Choose $\mathbb{Z}/q_1\mathbb{Z}$ and $\mathbb{Z}/q_2\mathbb{Z}$, where q_1 and q_2 are machine word size primes.
- Compute FFT over $\mathbb{Z}/q_i\mathbb{Z}$ and deduce FFT over $\mathbb{Z}/(q_1 q_2)\mathbb{Z}$ via CRT.

LHC step

- After CRT, coefficients of $f_u = f_x \cdot f_y \pmod{(R^k + 1)} \in \mathbb{Z}$ are 128-bit numbers.
- Each coefficient u_i of f_u is re-written as $u_i = c_i r^2 + h_i r + \ell_i$ where $0 \leq \ell_i, h_i < r$ and $c_i \in [-k, k]$. Doing so, we have:

$$\begin{aligned} f_u(R) &= f_x(R) \cdot f_y(R) \pmod{(R^k + 1)} \\ &= (c_0 R^2 + h_0 R + \ell_0) + (c_1 R^2 + h_1 R + \ell_1)R + (c_2 R^2 + h_2 R + \ell_2)R^2 \\ &\quad + \dots \\ &\quad + (c_{k-2} R^2 + h_{k-2} R + \ell_{k-2})R^{k-2} + (c_{k-1} R^2 + h_{k-1} R + \ell_{k-1})R^{k-1} \\ &= \sum_{i=0}^{k-1} (c_i R^{2+i} + h_i R^{1+i} + \ell_i R^i) \\ &= R^2 \sum_{i=0}^{k-1} (c_i R^i) + R \sum_{i=0}^{k-1} (h_i R^i) + \sum_{i=0}^{k-1} (\ell_i R^i) \end{aligned}$$

FFT-based multiplication algorithm

```

1: procedure FFT-BASEDMULTIPLICATION( $\vec{x}, \vec{y}, r, k$ )
2:    $\vec{z}_1 :=$  NegacyclicConvolution( $\vec{x}, \vec{y}, p_1, k$ )
3:    $\vec{z}_2 :=$  NegacyclicConvolution( $\vec{x}, \vec{y}, p_2, k$ )
4:   for  $0 \leq i < k$  do
5:      $[s_{0i}, s_{1i}] :=$  CRT( $p_1, p_2, m_1, m_2, z_{1i}, z_{2i}$ )
6:   end for
7:   for  $0 \leq i < k$  do
8:      $[\ell_i, h_i, c_i] :=$  LHC( $s_{0i}, s_{1i}, r$ )
9:   end for
10:   $\vec{c} :=$  MulPowR( $\vec{c}, 2, k, r$ )
11:   $\vec{h} :=$  MulPowR( $\vec{h}, 1, k, r$ )
12:   $\vec{u} :=$  BigPrimeFieldAddition( $\vec{\ell}, \vec{h}, k, r$ )
13:   $\vec{u} :=$  BigPrimeFieldAddition( $\vec{u}, \vec{c}, k, r$ )
14:  return  $\vec{u}$ 
15: end procedure

```

Problem: Lots of modular multiplications in the negacyclic convolutions

Solution: We use Montgomery multiplication inside convolutions!

Problem: CRT and LHC parts need multi-precision arithmetic!

- gcc provides 128-bit arithmetic, however, it is not the most efficient way!
- **Solution:** Using assembly code for CRT and LHC computation.

Example: inline assembly for multiplying two 64-bit integers

```
void mult_u64_u64
(const usfixn64 *a, const usfixn64 *b,
 usfixn64 *s0_out, usfixn64 *s1_out)
{
    usfixn64 s0=0, s1=0;
    __asm__ __volatile__(
        "movq  %2, %%rax;\n\t" // rax = a
        "movq  %3, %%rdx;\n\t" // rdx = b
        "mulq  %%rdx;\n\t"     // rdx:rax = a * b
        "movq  %%rax, %0;\n\t" // s0 = rax (low part)
        "movq  %%rdx, %1;\n\t" // s1 = rdx (high part)
        : "=&q" (s0), "=&q" (s1)
        : "q" (*a), "q" (*b)
        : "%rax", "%rdx", "memory");
    *s0_out = s0;
    *s1_out = s1;
}
```

Outline

- 1 Introduction
- 2 Generalized Fermat prime field arithmetic
- 3 Implementation of the multiplication in $\mathbb{Z}/p\mathbb{Z}$
- 4 Implementation of the FFT
- 5 Experimental results
- 6 Conclusions

The big prime field FFT in the BPAS library

- Our implementation is based on the **six-step FFT algorithm**:

$$\text{DFT}_N = L_K^N (I_J \otimes \text{DFT}_K) L_J^N D_{K,J} (I_K \otimes \text{DFT}_J) L_K^N \text{ with } N = JK.$$

- Here, \otimes denotes the tensor product of two matrices A and B :

$$A \otimes B = \begin{bmatrix} a_{11}B & \cdots & a_{1n}B \\ \vdots & \ddots & \vdots \\ a_{m1}B & \cdots & a_{mn}B \end{bmatrix}$$

- The stride permutation L_m^{mn} permutes an input vector \vec{x} of length mn :
 $\vec{x}[in + j] \mapsto \vec{x}[jm + i].$
- The twiddle factor $D_{K,J}$ is a diagonal matrix of the powers of ω .

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

Computing DFT_{K^e} through base-case DFT_K

- Since $p = r^k + 1$, we know that $r^k = -1 \pmod p$, r is a $2k$ -th primitive root of unity in $\mathbb{Z}/p\mathbb{Z}$.
- We know that multiplying by powers of the radix r is cheap.

Recall Fürer's trick

- Define $\eta = \omega^{N/K}$, let $J = K^{e-1}$, and assume that multiplying an arbitrary element of $\mathbb{Z}/p\mathbb{Z}$ by η^i ($0 \leq i \leq K$) can be done within $O(k)$ word ops.
- Every arithmetic operation involved in DFT_K at η costs $O(k)$ word ops.
- Therefore, such DFT_K can be performed within $O(K \log(K) k)$ word ops.

Reducing to base-case

- Let $N = K^e$, then we can compute DFT_{K^e} by DFT_K .
- By choosing $K = 2k$, the multiplication inside DFT_{2k} is cheap.

Computing base-case DFT_K (1/3)

Reducing DFT_K to DFT_2 for $K = 2^n$

- Definition of DFT_2 :

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

- Six-step factorization of DFT_{2^n} :

$$\text{DFT}_{2^n} = L_2^{2^n} (I_{2^{n-1}} \otimes \text{DFT}_2) L_{2^{n-1}}^{2^n} D_{2,2^{n-1}} (I_2 \otimes \text{DFT}_{2^{n-1}}) L_2^{2^n}$$

- Example of DFT_8 through DFT_2 :

$$\text{DFT}_8 = L_2^8 (I_4 \otimes \text{DFT}_2) L_4^8 D_{2,4} (I_2 \otimes \text{DFT}_4) L_2^8 \quad (1)$$

$$\text{DFT}_4 = L_2^4 (I_2 \otimes \text{DFT}_2) L_2^4 D_{2,2} (I_2 \otimes \text{DFT}_2) L_2^4 \quad (2)$$

$$\begin{aligned} \text{DFT}_8 = & L_2^8 (I_4 \otimes \text{DFT}_2) L_4^8 D_{2,4} (I_2 \otimes L_2^4) (I_4 \otimes \text{DFT}_2) \\ & (I_2 \otimes L_2^4) (I_2 \otimes D_{2,2}) (I_4 \otimes \text{DFT}_2) (I_2 \otimes L_2^4) (L_2^8). \end{aligned} \quad (3)$$

Computing base-case DFT_K (2/3)

Avoiding permutation

- Avoiding the permutation and actually data movement.
- Pre-compute the position of elements after each permutation and hard-code those values in the algorithm for computing the base-case.

- Index of input data: $\vec{M} = (0, 1, 2, 3, 4, 5, 6, 7)$
 $\vec{M}_1 = L_2^8 \vec{M} = (0, 2, 4, 6, 1, 3, 5, 7)$ (4)

$$\vec{M}_2 = (I_2 \otimes L_2^4) \vec{M}_1 = (0, 4, 2, 6)(1, 5, 3, 7)$$
 (5)

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

Twiddle multiplications

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

$$D_{2,2} = \text{diag}(1, 1, \omega_1^0, \omega_1^1), \quad D_{2,4} = \text{diag}(1, 1, 1, 1, \omega_0^0, \omega_0^1, \omega_0^2, \omega_0^3)$$
 (7)

- We have $\omega_0 = r$ and $\omega_1 = r^2$ ($r^8 \equiv 1 \pmod{p}$, for $p = r^4 + 1$).

- Then, the twiddle matrices are updated as follows:

$$D_{2,4} = \text{diag}(1, 1, 1, 1, 1, r, r^2, r^3)$$
 (8)

$$D_{2,2} = \text{diag}(1, 1, 1, r^2)$$
 (9)

Example: Computing DFT_8 for vector \vec{a}

```
1: DFT2(a0, a4); DFT2(a1, a5);
2: DFT2(a2, a6); DFT2(a3, a7);
3: a6 := a6 ω2;
4: a7 := a7 ω2;
5: DFT2(a0, a2); DFT2(a1, a3);
6: DFT2(a4, a6); DFT2(a5, a7);
7: a5 := a5 ω1;
8: a3 := a3 ω2;
9: a7 := a7 ω2;
10: DFT2(a0, a1); DFT2(a2, a3);
11: DFT2(a4, a5); DFT2(a6, a7);
12: swap(a1, a4);
13: swap(a3, a6);
14: return  $\vec{a}$ ;
```

Parallelization of the FFT

Choice of the FFT algorithm

- The six-step FFT can be implemented in an iterative fashion.
- Unroll $I_K \otimes \text{DFT}_J$ until there's only DFT on K points.
- There is no data dependency between the iterations of each inner-loop.

Programming considerations

- CILK: work-stealing scheme, light-weight threads (re-use of the threads), etc.
- The FFT over the crafted BPAS implementation of GPPF incurs less memory accesses than the FFT based on GMP arithmetics; see experimental results.

Implementation of the six-step FFT

```
1: procedure DFT_GENERAL( $\vec{x}$ ,  $K$ ,  $e$ ,  $\omega$ )
2:   for  $0 \leq i < e - 1$  do
3:     for  $0 \leq j < K^i$  do
4:       stride_permutation( $x_{jK^{e-i}}$ ,  $K$ ,  $K^{e-i-1}$ )
5:     end for
6:   end for
7:    $\omega_a := \omega^{K^{e-1}}$ 
8:   for  $0 \leq j < K^{e-1}$  do
9:     idx :=  $jK$ 
10:    DFT_K( $x_{idx}$ ,  $\omega_a$ )
11:   end for
12:   for  $e - 2 \geq i \geq 0$  do
13:      $\omega_i := \omega^{K^i}$ 
14:     for  $0 \leq j < K^i$  do
15:       idx :=  $jK^{e-i}$ 
16:       twiddle( $x_{idx}$ ,  $K^{e-i-1}$ ,  $K$ ,  $\omega_i$ )
17:       stride_permutation( $x_{idx}$ ,  $K^{e-i-1}$ ,  $K$ )
18:     end for
19:     for  $0 \leq j < K^{e-1}$  do
20:       idx :=  $jK$ 
21:       DFT_K( $x_{idx}$ ,  $\omega_a$ )
22:     end for
23:     for  $0 \leq j < K^i$  do
24:       idx :=  $jK^{e-i}$ 
25:       stride_permutation( $x_{idx}$ ,  $K$ ,  $K^{e-i-1}$ )
26:     end for
27:   end for
28: end procedure
```

▷ Can be replaced with Parallel-For.
▷ Step 1

▷ Can be replaced with Parallel-For.
▷ Step 2

▷ Can be replaced with Parallel-For.
▷ Step 3
▷ Step 4

▷ Can be replaced with Parallel-For.
▷ Step 5

▷ Can be replaced with Parallel-For.
▷ Step 6

Outline

- 1 Introduction
- 2 Generalized Fermat prime field arithmetic
- 3 Implementation of the multiplication in $\mathbb{Z}/p\mathbb{Z}$
- 4 Implementation of the FFT
- 5 Experimental results
- 6 Conclusions

Benchmarks

Two approaches: GPPF vs. GMP

- Generalized Fermat prime field arithmetic relying on FFT-based multiplication
- GMP arithmetic.

Benchmarks: Comparing performance of the following

- Multiplication of two arbitrary elements of the big prime field.
- Serial vs. parallel implementation of each approach for computing FFT of large vectors over different big prime fields.
- Each step of an FFT computation.

Experimentation Setup

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

prime	$K(= 2k)$	k	r
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	$2^{57} + 2^{56} + 2^{11}$

- Intel-i7-7700K: 4-cores @4.50 GHz (8 threads when hyper-threading is enabled), 16 GB of memory (@2133 MHz).
- Xeon-X5650: 6-cores @2.66 GHz (12 threads when hyper-threading is enabled), 48 GB of memory (@1133 MHz).

Multiplication between arbitrary elements in big prime fields

Table: 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

Table: 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

FFT over big prime fields: measured on Intel-i7-7700K

Table: 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			Parallel Speedups	
				GFPF	GMP	$\frac{t_{\text{GFPF}}}{t_{\text{GMP}}}$	GFPF	GMP	$\frac{t_{\text{GFPF}}}{t_{\text{GMP}}}$	GFPF	GMP
P_4	4	8	2	0.019	0.030	0.63x	0.057	0.118	0.48x	0.33	0.25
P_4	4	8	3	0.314	0.363	0.86x	0.215	0.276	0.77x	1.46	1.32
P_8	8	16	2	0.181	0.202	0.89x	0.117	0.143	0.81x	1.55	1.41
P_8	8	16	3	5.771	5.486	1.05x	1.603	2.247	0.71x	3.60	2.44
P_{16}	16	32	2	1.644	1.730	0.95x	0.513	0.693	0.74x	3.20	2.50
P_{16}	16	32	3	103.423	104.620	0.98x	24.052	35.017	0.68x	4.30	2.99
P_{32}	32	64	2	14.815	20.341	0.72x	3.507	5.411	0.64x	4.22	3.76
P_{32}	32	64	3	1922.373	2431.867	0.79x	462.746	702.163	0.65x	4.15	3.46
P_{64}	64	128	2	140.995	278.188	0.50x	33.507	69.879	0.47x	4.21	3.98
P_{128}	128	256	2	580.961	3745.353	0.15x	154.064	905.799	0.17x	3.77	4.13

FFT over big prime fields: measured on Xeon-X5650

Table: 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 Xeon-X5650).

prime	k	K	e	Serial			Parallel			Parallel Speedups	
				GFPF	GMP	$\frac{t_{\text{GFPF}}}{t_{\text{GMP}}}$	GFPF	GMP	$\frac{t_{\text{GFPF}}}{t_{\text{GMP}}}$	GFPF	GMP
P_4	4	8	2	0.051	0.071	0.71x	0.155	0.114	1.35x	0.33	0.62
P_4	4	8	3	0.843	0.917	0.91x	0.452	0.577	0.78x	1.87	1.59
P_8	8	16	2	0.472	0.546	0.86x	0.217	0.320	0.67x	2.18	1.71
P_8	8	16	3	16.661	15.231	1.09x	2.837	4.806	0.59x	5.87	3.17
P_{16}	16	32	2	4.444	5.085	0.87x	0.877	1.371	0.63x	5.07	3.71
P_{16}	16	32	3	284.080	297.904	0.95x	41.012	66.635	0.61x	6.93	4.47
P_{32}	32	64	2	39.809	64.307	0.61x	5.701	11.640	0.48x	6.98	5.52
P_{32}	32	64	3	4674.079	6501.669	0.71x	696.311	1289.061	0.54x	6.71	5.04
P_{64}	64	128	2	376.450	909.041	0.41x	53.578	140.610	0.38x	7.03	6.46
P_{128}	128	256	2	1395.310	13371.369	0.10x	240.362	1811.282	0.13x	5.81	7.38

Time spent in each step of FFT

Time spent in each step of FFT

Table: 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)

Average multiplication time in FFT

K	16				32				64		
	1	2	3	4	1	2	3	4	1	2	3
FFT-based	0.32	2.96	3.53	3.77	0.65	5.72	6.84	7.31	1.44	10.87	12.98
GMP	4.17	4.17	4.17	4.17	11.79	11.79	11.79	11.79	34.53	34.53	34.53

Table: Average time (in milliseconds) spent in one modular multiplication during computation of FFT over big prime fields, presented for the three implementations measured on Intel-i7-7700K.

Comparing memory accesses GFPF vs. GMP

The number of memory references measured for each variant (and the ratio $\frac{\#GFPF \text{ D refs}}{\#GMP \text{ D refs}}$) of serial computation of FFT on vectors of size $N = K^2$ over $\mathbb{Z}/p\mathbb{Z}$ for P_{16} , P_{32} , P_{64} , and P_{128} .

Table: Measured on Intel-i7-7700K using Valgrind.

input size $N = K^2$	D refs			D1 miss rate (%)	
	GFPF	GMP	$\frac{\# \text{ GMP refs}}{\# \text{ GFPF refs}}$	GFPF	GMP
K=16	689,220	1,042,440	1.51x	0.2	0.9
K=32	5,704,483	7,810,065	1.36x	0.5	0.7
K=64	50,718,515	82,608,297	1.62x	0.4	0.5
K=128	535,935,616	1,063,157,320	1.98x	0.8	0.5

Table: Measured on Xeon-X5650 using Valgrind.

input size $N = K^2$	D refs			D1 miss rate (%)	
	GFPF	GMP	$\frac{\# \text{ GMP refs}}{\# \text{ GFPF refs}}$	GFPF	GMP
K=16	645,018	1,043,169	1.61x	0.2	0.9
K=32	5,340,965	7,824,678	1.46x	0.5	0.7
K=64	49,143,357	82,748,934	1.68x	0.4	0.5
K=128	556,770,530	1,070,452,476	1.92x	0.7	0.5

Outline

- 1 Introduction
- 2 Generalized Fermat prime field arithmetic
- 3 Implementation of the multiplication in $\mathbb{Z}/p\mathbb{Z}$
- 4 Implementation of the FFT
- 5 Experimental results
- 6 Conclusions

Conclusions

- FFT can be used effectively to improve multiplication time in big prime field.
- Using Generalized Fermat prime fields can lower the average time spent in multiplications in FFT.
- The big prime field FFT can be implemented on CPU efficiently.
- Multiplication of arbitrary elements in $\mathbb{Z}/p\mathbb{Z}$ is still a bottleneck.

Future work

- Improving multiplication of arbitrary elements in $\mathbb{Z}/p\mathbb{Z}$, by making each part more efficient.
- Use different polynomial multiplication algorithms for different sizes of the primes.

Thank You!

Your Questions?

Appendix

System cache specifications

Metric	Intel-i7-7700K	xeonnode02
Line size	64	64
L1d cache	32K	32K
L1i cache	32K	32K
L2 cache	256K	256K
L3 cache	8192K	12288K

Considerations for GMP implementation

- Function `mpz_mul` is not in-place, better to have a separate destination than input arguments.
- Immediate `mpz` functions are cheaper to use (such `mpz_add_ui`).
- Due to memory management overhead, `mpz_mod` is more expensive than `mpz_tdiv_r`.

Precomputation of twiddle factors

Twiddle matrices are in the form of $D_{K,Ke-s}$ where $\omega_i = \omega^{K(s-1)}$ ($1 \leq s < e$)

- We know that $\omega^N \equiv r^{2k} \equiv r^K \equiv 1 \pmod{p}$.
- For $y = x \cdot \omega^{i(N/K)+j}$, we only need to compute:
 - ① $y' = x \cdot \omega^{i(N/K)} = x \cdot r^i$ (a cheap multiplication), and
 - ② $y = y' \cdot \omega^j$ (arbitrary multiplication).
- We can pre-compute ω^j with $0 < j < N/K$, leading to a lower pre-computation expense and less memory usage.