Big Prime Field FFT on the GPU

Liangyu Chen
East China Normal University
lychen@sei.ecnu.edu.cn

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

Davood Mohajerani
University of Western Ontario
dmohajer@uwo.ca

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

ABSTRACT

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, we show 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 efficient implementation of fast Fourier transforms on graphics processing units.

CCS CONCEPTS

• **Mathematics of computing → Mathematical software performance**;

KEYWORDS

Fast Fourier transforms, Finite fields, Generalized Fermat prime numbers, Graphics processing units, CUDA.

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 small characteristic, that is, based on prime numbers that fit in a machine word. 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 yields major issues 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]. This makes using larger primes desirable.

In this paper, 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, we show 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 efficient implementation of fast Fourier transforms on graphics processing units (GPUs).

We present algorithms for arithmetic operations in a "big" prime field $\mathbb{Z}/p\mathbb{Z}$, where p is a generalized Fermat number of the form $p = r^k + 1$ where r fits a machine-word and k is a power of 2. We report on a GPU implementation of those algorithms as well as a GPU implementation of a Fast Fourier Transform (FFT) over such a big prime field. Our experimental results show that

(1) computing an FFT of size N, over a big prime field for p fitting on k 64-bit machine-words, and
(2) computing $2k$ FFTs of size N, over a small prime field (that is, where the prime fits a 32-bit half-machine-word) followed by a combination (i.e. CRT-like) of those FFTs

are two competitive approaches in terms of running time. Since the former approach has the advantage of reducing the occurrence of unlucky primes when applying modular methods (in particular in the area of polynomial system solving), we view this experimental observation as a promising result.

The reasons for a GPU implementation are as follows. First, the model of computations and the hardware performance provide interesting opportunities for big prime field arithmetic, in particular in terms of vectorization of the program code. Secondly, highly optimized FFTs over small prime fields have been implemented on GPUs by Wei Pan [18, 19] in the CUMODP library, see www.cumodp.org, and we use them in our experimental comparison.

Section 7 reports on various comparative experimentations. First, a comparison of the above two approaches implemented on GPU, exhibiting an advantage for the FFT over a big prime field. Second, a comparison between the two same approaches implemented on a single-core CPU, exhibiting an advantage for the CRT-based FFT over small prime fields. Third, from the two previous comparisons, one deduces a comparison of the FFT over a big prime field (resp. the CRT-based FFT over small prime fields) implemented on GPU and CPU, exhibiting a clear advantage for the GPU implementations. Overall, the big prime field FFT on the GPU is the best approach.

A discrete Fourier transform (DFT) over $\mathbb{Z}/p\mathbb{Z}$, when p is a generalized Fermat prime, can be seen as a generalization of the FNT (Fermat number transform), which is a specific case of the NTT (number theoretic transform). However, the computation of a DFT over $\mathbb{Z}/p\mathbb{Z}$ implies additional considerations, which are not taken into account in the literature on NTT or FNT computations [1, 9].

The computation of a NTT can be done via various methods used for a DFT, among them is the radix-2 Cooley-Tukey, for example.

However, the final complexity depends on the way a given DFT is computed. It appears that, in the context of generalized Fermat primes, there is a better choice than the radix-2 Cooley-Tukey. The method used in the present paper is related to the article [7], which is derived from Fürer's algorithm [11] for the multiplication of large integers. The practicality of this latter algorithm is an open question. And, in fact, the work reported in our paper is a practical contribution responding to this open question.

The paper [1] discusses the idea of using Fermat number transform for computing convolutions, thus working modulo numbers of the form $F = 2^b + 1$, where b is a power of 2. This is an effective way to avoid round-off error caused by twiddle factor multiplication in computing DFT over the field of complex numbers. The paper [9] considers *generalized Fermat Mersenne* (GFM) prime numbers that prime of the form $(q^{pn} - 1)/(q - 1)$ where, typically, q is 2 and both p and n are small. These numbers are different from the primes used in our paper, which have the form $r^k + 1$ where r is typically machine-word long and k is a power of 2 so that r is a $2k$-th primitive root of unity, see Section 3.

2 COMPLEXITY ANALYSIS

Consider a prime field $\mathbb{Z}/p\mathbb{Z}$ and N, a power of 2, dividing $p - 1$. Then, the finite field $\mathbb{Z}/p\mathbb{Z}$ admits an N-th primitive root of unity[1] Denote by ω such an element. Let $f \in \mathbb{Z}/p\mathbb{Z}[x]$ be of degree at most $N - 1$. Then, computing the DFT of f at ω via an FFT amounts to:

(1) $N \log(N)$ additions in $\mathbb{Z}/p\mathbb{Z}$,
(2) $(N/2) \log(N)$ multiplications by a power of ω in $\mathbb{Z}/p\mathbb{Z}$.

If the bit-size of p is k machine words, then

(1) each addition in $\mathbb{Z}/p\mathbb{Z}$ costs $O(k)$ machine-word operations,
(2) each multiplication by a power of ω costs $O(\mathsf{M}(k))$ machine-word operations,

where $n \longmapsto \mathsf{M}(n)$ is a multiplication time as defined in [23]. Therefore, multiplication by a power of ω becomes a bottleneck as k grows. To overcome this difficulty, we consider the following trick proposed by Martin Fürer in [11, 12]. We assume that $N = K^e$ holds for some "small" K, say $K = 32$ and an integer $e \geq 2$. Further, we define $\eta = \omega^{N/K}$, with $J = K^{e-1}$ and assume that multiplying an arbitrary element of $\mathbb{Z}/p\mathbb{Z}$ by η^i, for any $i = 0, \ldots, K - 1$, can be done within $O(k)$ machine-word operations. Consequently, every arithmetic operation (addition, multiplication) involved in a DFT of size K, using η as a primitive root, amounts to $O(k)$ machine-word operations. Therefore, such DFT of size K can be performed with $O(K \log(K) k)$ machine-word operations. As we shall see in Section 3, this latter result holds whenever p is a so called *generalized Fermat number*.

Returning to the DFT of size N at ω and using the factorization formula of Cooley and Tukey, we have

$$\mathrm{DFT}_{JK} = (\mathrm{DFT}_J \otimes I_K) D_{J,K} (I_J \otimes \mathrm{DFT}_K) L_J^{JK}, \qquad (1)$$

see Section 4. Hence, the DFT of f at ω is essentially performed by:

(1) K^{e-1} DFT's of size K (that is, DFT's on polynomials of degree at most $K - 1$),
(2) N multiplications by a power of ω (coming from the diagonal matrix $D_{J,K}$) and

[1]See Section 4 for this notion.

(3) K DFT's of size K^{e-1}.

Unrolling Formula (1) so as to replace DFT_J by DFT_K and the other linear operators involved (the diagonal matrix D and the permutation matrix L) one can see that a DFT of size $N = K^e$ reduces to:

(1) $e K^{e-1}$ DFT's of size K, and
(2) $(e - 1) N$ multiplication by a power of ω.

Recall that the assumption on the cost of a multiplication by η^i, for $0 \leq i < K$, makes the cost for one DFT of size K to $O(K \log_2(K) k)$ machine-word operations. Hence, all the DFT's of size K together amount to $O(e N \log_2(K) k)$ machine-word operations. That is, $O(N \log_2(N) k)$ machine-word operations. Meanwhile, the total cost of the multiplication by a power of ω is $O(e N \mathsf{M}(k))$ machine-word operations, that is, $O(N \log_K(N) \mathsf{M}(k))$ machine-word operations. Indeed, multiplying an arbitrary element of $\mathbb{Z}/p\mathbb{Z}$ by an arbitrary power of ω requires $O(\mathsf{M}(k))$ machine-word operations. Therefore, under our assumption, a DFT of size N at ω amounts to

$$O(N \log_2(N) k + N \log_K(N) \mathsf{M}(k)) \qquad (2)$$

machine-word operations. When using generalized Fermat primes, we have $K = 2k$ and the above estimate becomes

$$O(N \log_2(N) k + N \log_k(N) \mathsf{M}(k)) \qquad (3)$$

The second term in the big-O notation dominates the first one. However, we keep both terms for reasons that will appear shortly.

Without our assumption, as discussed earlier, the same DFT would run in $O(N \log_2(N) \mathsf{M}(k))$ machine-word operations. Therefore, using generalized Fermat primes brings a speedup factor of $\log(K)$ w.r.t. the direct approach using arbitrary prime numbers.

At this point, it is natural to ask what would be the cost of a comparable computation using small primes and the CRT. To be precise, let us consider the following problem. Let $p_1, \ldots, p_k$ be pairwise different prime numbers of machine-word size and let m be their product. Assume that N divides each of $p_1 - 1, \ldots, p_k - 1$ such that each of fields $\mathbb{Z}/p_1\mathbb{Z}, \ldots, \mathbb{Z}/p_k\mathbb{Z}$ admits an N-th primitive roots of unity, $\omega_1, \ldots, \omega_k$. Then, $\omega = (\omega_1, \ldots, \omega_k)$ is an N-th primitive root of $\mathbb{Z}/m\mathbb{Z}$. Indeed, the ring $\mathbb{Z}/p_1\mathbb{Z} \otimes \cdots \otimes \mathbb{Z}/p_k\mathbb{Z}$ is a direct product of fields. Let $f \in \mathbb{Z}/m\mathbb{Z}[x]$ be a polynomial of degree $N - 1$. One can compute the DFT of f at ω in three steps:

(1) Compute the images $f_1 \in \mathbb{Z}/p_1\mathbb{Z}[x], \ldots, f_k \in \mathbb{Z}/p_k\mathbb{Z}[x]$ of f.
(2) Compute the DFT of f_i at ω_i in $\mathbb{Z}/p_i\mathbb{Z}[x]$, for $i = 1, \ldots, k$,
(3) Combine the results using CRT so as to obtain a DFT of f at ω.

The first and the third above steps will run within $O(N \times \mathsf{M}(k) \log_2(k))$ machine-word operations meanwhile the second one amount to $O(k \times N \log(N))$ machine-word operations, yielding a total of

$$O(N \log_2(N) k + N \mathsf{M}(k) \log_2(k)) \qquad (4)$$

These estimates yield a running-time ratio between the two approaches of $\log(N)/\log_2^2(k)$, which suggests that for k large enough the big prime field approach may outperform the CRT-based approach. We believe that this analysis is part of the explanation for the observation that the two approaches are, in fact, competitive in practice, as we shall see in Section 7.

3 SPARSE RADIX GENERALIZED FERMAT NUMBERS

The n-th Fermat number, denoted by F_n, is given by $F_n = 2^{2^n} + 1$. This sequence plays an important role in number theory and, as mentioned in the introduction, in the development of asymptotically fast algorithms for integer multiplication [12, 21].

Arithmetic operations modulo a Fermat number are simpler than modulo an arbitrary positive integer. In particular 2 is a 2^{n+1}-th primitive root of unity modulo F_n. Unfortunately, F_4 is the largest Fermat number which is known to be prime. Hence, when computations require the coefficient ring be a field, Fermat numbers are no longer interesting. This motivates the introduction of other family of Fermat-like numbers, see, for instance, Chapter 2 in the text book *Guide to elliptic curve cryptography* [13].

Numbers of the form $a^{2^n} + b^{2^n}$ where $a > 1$, $b \geq 0$ and $n \geq 0$ are called *generalized Fermat numbers*. An odd prime p is a generalized Fermat number if and only if p is congruent to 1 modulo 4. The case $b = 1$ is of particular interest and, by analogy with the ordinary Fermat numbers, it is common to denote the generalized Fermat number $a^{2^n} + 1$ by $F_n(a)$. So 3 is $F_0(2)$. We call a the *radix* of $F_n(a)$. Note that, Landau's fourth problem asks if there are infinitely many generalized Fermat primes $F_n(a)$ with $n > 0$.

In the finite ring $\mathbb{Z}/F_n(a)\mathbb{Z}$, the element a is a 2^{n+1}-th primitive root of unity. However, when using binary representation for integers on a computer, arithmetic operations in $\mathbb{Z}/F_n(a)\mathbb{Z}$ may not be as easy to perform as in $\mathbb{Z}/F_n\mathbb{Z}$. This motivates the following.

DEFINITION 1. *We call* sparse radix generalized Fermat number, *any integer of the form $F_n(r)$ where r is either $2^w + 2^u$ or $2^w - 2^u$, for some integers $w > u \geq 0$. In the former case, we denote $F_n(r)$ by $F_n^+(w, u)$ and in the latter by $F_n^-(w, u)$.*

Table 1 lists sparse radix generalized Fermat numbers (SRGFNs) that are prime. For each such number p, we give the largest power of 2 dividing $p - 1$, that is, the maximum length N of a vector to which a radix-K FFT algorithm where K is an appropriate power of 2.

Table 1: SRGFNs of practical interest.

p	$\max\{2^e \text{ s.t. } 2^e \mid p - 1\}$
$(2^{63} + 2^{53})^2 + 1$	2^{106}
$(2^{64} - 2^{50})^4 + 1$	2^{200}
$(2^{63} + 2^{34})^8 + 1$	2^{272}
$(2^{62} + 2^{36})^{16} + 1$	2^{576}
$(2^{62} + 2^{56})^{32} + 1$	2^{1792}
$(2^{63} - 2^{40})^{64} + 1$	2^{2560}
$(2^{64} - 2^{28})^{128} + 1$	2^{3584}

NOTATION 1. *In the sequel, we consider $p = F_n(r)$, a fixed SRGFN. We denote by 2^e the largest power of 2 dividing $p - 1$ and we define $k = 2^n$, so that $p = r^k + 1$ holds.*

As we shall see in the sequel of this section, for any positive integer N which is a power of 2 such that N divides $p - 1$, one can find an N-th primitive root of unity $\omega \in \mathbb{Z}/p\mathbb{Z}$ such that multiplying an element $x \in \mathbb{Z}/p\mathbb{Z}$ by $\omega^{i(N/2k)}$ for $0 \leq i < 2k$ can be done in

linear time w.r.t. the bit size of x. Combining this observation with an appropriate factorization of the DFT transform on N points over $\mathbb{Z}/p\mathbb{Z}$, we obtain an efficient FFT algorithm over $\mathbb{Z}/p\mathbb{Z}$.

3.1 Representation of $\mathbb{Z}/p\mathbb{Z}$

We represent each element $x \in \mathbb{Z}/p\mathbb{Z}$ as a vector $\vec{x} = (x_{k-1}, x_{k-2}, \ldots, x_0)$ of length k and with non-negative integer coefficients such that we have

$$x \equiv x_{k-1} r^{k-1} + x_{k-2} r^{k-2} + \cdots + x_0 \mod p. \quad (5)$$

This representation is made unique by imposing the following constraints
(1) either $x_{k-1} = r$ and $x_{k-2} = \cdots = x_1 = 0$,
(2) or $0 \leq x_i < r$ for all $i = 0, \ldots, (k-1)$.

We also map x to a univariate integer polynomial $f_x \in \mathbb{Z}[T]$ defined by $f_x = \sum_{i=0}^{k-1} x_i t^i$ such that $x \equiv f_x(r) \mod p$.

Now, given a non-negative integer $x < p$, we explain how the representation $\vec{x}$ can be computed. The case $x = r^k$ is trivially handled, hence we assume $x < r^k$. For a non-negative integer z such that $z < r^{2^i}$ holds for some positive integer $i \leq n = \log_2(k)$, we denote by $\text{vec}(z, i)$ the unique sequence of 2^i non-negative integers $(z_{2^i-1}, \ldots, z_0)$ such that we have $0 \leq z_j < r$ and $z = z_{2^i-1} r^{2^i-1} + \cdots + z_0$. The sequence $\text{vec}(z, i)$ is obtained as follows:
(1) if $i = 1$, we have $\text{vec}(z, i) = (q, s)$,
(2) if $i > 1$, then $\text{vec}(z, i)$ is the concatenation of $\text{vec}(q, i-1)$ followed by $\text{vec}(s, i-1)$,
where q and s are the quotient and the remainder in the Euclidean division of z by $r^{2^{i-1}}$. Clearly, $\text{vec}(x, n) = \vec{x}$ holds.

We observe that the sparse binary representation of r facilitates the Euclidean division of a non-negative integer z by r, when performed on a computer. Referring to the notations in Definition 1, let us assume that r is $2^w + 2^u$, for some integers $w > u \geq 0$. (The case $2^w - 2^u$ would be handled in a similar way.) Let z_{high} and z_{low} be the quotient and the remainder in the Euclidean division of z by 2^w. Then, we have

$$z = 2^w z_{\text{high}} + z_{\text{low}} = r z_{\text{high}} + z_{\text{low}} - 2^u z_{\text{high}}. \quad (6)$$

Let $s = z_{\text{low}} - 2^u z_{\text{high}}$ and $q = z_{\text{high}}$. Three cases arise:
(S1) if $0 \leq s < r$, then q and s are the quotient and remainder of z by r,
(S2) if $r \leq s$, then we perform the Euclidean division of s by r and deduce the desired quotient and remainder,
(S3) if $s < 0$, then (q, s) is replaced by $(q + 1, s + r)$ and we go back to Step (S1).

Since the binary representations of r^2 can still be regarded as sparse, a similar procedure can be done for the Euclidean division of a non-negative integer z by r^2. For higher powers of r, we believe that Montgomery algorithm is the way to go, though this remains to be explored.

3.2 Finding primitive roots of unity in $\mathbb{Z}/p\mathbb{Z}$

NOTATION 2. *Let N be a power of 2, say 2^ℓ, dividing $p - 1$ and let $g \in \mathbb{Z}/p\mathbb{Z}$ be an N-th primitive root of unity.*

Recall that such an N-th primitive root of unity can be obtained by a simple probabilistic procedure. Write $p = qN + 1$. Pick a random

$\alpha \in \mathbb{Z}/p\mathbb{Z}$ and let $\omega = \alpha^q$. Little Fermat theorem implies that either $\omega^{N/2} = 1$ or $\omega^{N/2} = -1$ holds. In the latter case, ω is an N-th primitive root of unity. In the former, another random $\alpha \in \mathbb{Z}/p\mathbb{Z}$ should be considered. In our various software implementation of finite field arithmetic [3, 14, 16], this procedure finds an N-th primitive root of unity after a few tries and has never been a performance bottleneck.

In the following, we consider the problem of finding an N-th primitive root of unity ω such that $\omega^{N/2k} = r$ holds. The intention is to speed up the portion of FFT computation that requires to multiply elements of $\mathbb{Z}/p\mathbb{Z}$ by powers of ω.

PROPOSITION 1. *In $\mathbb{Z}/p\mathbb{Z}$, the element r is a $2k$-th primitive root of unity. Moreover, the following algorithm computes an N-th primitive root of unity $\omega \in \mathbb{Z}/p\mathbb{Z}$ such that we have $\omega^{N/2k} = r$ in $\mathbb{Z}/p\mathbb{Z}$.*

Algorithm 1 Primitive N-th root $\omega \in \mathbb{Z}/p\mathbb{Z}$ s.t. $\omega^{N/2k} = r$

---

  procedure PRIMITIVEROOTASROOTOF(N, r, k, g)
     $\alpha := g^{N/2k}$
     $\beta := \alpha$
     $j := 1$
     while $\beta \neq r$ **do**
       $\beta := \alpha\beta$
       $j := j + 1$
     end while
     $\omega := g^j$
     return (ω)
  end procedure

Proof Since $g^{N/2k}$ is a $2k$-th root of unity, it is equal to r^{i_0} (modulo p) for some $0 \leq i_0 < 2k$ where i_0 is odd. Let j be a non-negative integer. Observe that we have

$$g^{j 2^\ell/2k} = (g^i g^{2kq})^{2^\ell/2k} = g^{i 2^\ell/2k} = r^{i\, i_0}, \tag{7}$$

where q and i are quotient and the remainder of j in the Euclidean division by $2k$. By definition of g, the powers $g^{i 2^\ell/2k}$, for $0 \leq i < 2k$, are pairwise different. It follows from Formula (7) that the elements $r^{i\, i_0}$ are pairwise different as well, for $0 \leq i < 2k$. Therefore, one of those latter elements is r itself. Hence, we have j_1 with $0 \leq j_1 < 2k$ such that $g^{j_1 N/2k} = r$. Then, $\omega = g^{j_1}$ is as desired and Algorithm 1 computes it. □

3.3 Addition and subtraction in $\mathbb{Z}/p\mathbb{Z}$

Let $x, y \in \mathbb{Z}/p\mathbb{Z}$ represented by $\vec{x}, \vec{y}$, see Section 3.1 for this latter notation. Algorithm 2 computes the representation $\overrightarrow{x + y}$ of the element $(x + y) \mod p$.

Proof At Step (1), $\vec{x}$ and $\vec{y}$, regarded as vectors over $\mathbb{Z}$, are added component-wise. At Steps (2) and (3), the carry, if any, is propagated. At Step (4), there is no carry beyond the leading digit z_{k-1}, hence $(z_{k-1}, \ldots, z_0)$ represents $x + y$. Step (5) handles the special case where $x + y = p - 1$ holds. Step (6) is the *overflow* case which is handled by subtracting $1 \mod p$ to $(z_{k-1}, \ldots, z_0)$, finally producing $\overrightarrow{x + y}$. □

Algorithm 2 Computing $x + y \in \mathbb{Z}/p\mathbb{Z}$ for $x, y \in \mathbb{Z}/p\mathbb{Z}$

---

  procedure BIGPRIMEFIELDADDITION$(\vec{x}, \vec{y}, r, k)$
  1: compute $z_i = x_i + y_i$ in $\mathbb{Z}$, for $i = 0, \ldots, k - 1$,
  2: let $z_k = 0$,
  3: for $i = 0, \ldots, k - 1$, compute the quotient q_i and the remainder s_i in the Euclidean division of z_i by r, then replace (z_{i+1}, z_i) by $(z_{i+1} + q_i, s_i)$,
  4: if $z_k = 0$ then return $(z_{k-1}, \ldots, z_0)$,
  5: if $z_k = 1$ and $z_{k-1} = \cdots = z_0 = 0$, then let $z_{k-1} = r$ and return $(z_{k-1}, \ldots, z_0)$,
  6: let i_0 be the smallest index, $0 \leq i_0 \leq k$, such that $z_{i_0} \neq 0$, then let $z_{i_0} = z_{i_0} - 1$, let $z_0 = \cdots = z_{i_0-1} = r - 1$ and return $(z_{k-1}, \ldots, z_0)$.
  end procedure

A similar procedure computes the vector $\overrightarrow{x - y}$ representing the element $(x - y) \in \mathbb{Z}/p\mathbb{Z}$. Recall that we explained in Section 3.1 how to perform the Euclidean divisions at Step (S3) in a way that exploits the sparsity of the binary representation of r.

In practice, the binary representation of the radix r fits a machine word, see Table 1. Consequently, so does each of the "digit" in the representation $\vec{x}$ of every element $x \in \mathbb{Z}/p\mathbb{Z}$. This allows us to exploit machine arithmetic in a sharper way. In particular, the Euclidean divisions at Step (S3) can be further optimized.

3.4 Multiplication by a power of r in $\mathbb{Z}/p\mathbb{Z}$

Before considering the multiplication of two arbitrary elements $x, y \in \mathbb{Z}/p\mathbb{Z}$, we assume that one of them, say y, is a power of r, say $y = r^i$ for some $0 < i < 2k$. Note that the cases $i = 0 = 2k$ are trivial. Indeed, recall that r is a $2k$-th primitive root of unity in $\mathbb{Z}/p\mathbb{Z}$. In particular, $r^k = -1$ in $\mathbb{Z}/p\mathbb{Z}$. Hence, for $0 < i < k$, we have $r^{k+i} = -r^i$ in $\mathbb{Z}/p\mathbb{Z}$. Thus, let us consider first the case where $0 < i < k$ holds. We also assume $0 \leq x < r^k$ holds in $\mathbb{Z}$, since the case $x = r^k$ is easy to handle. From Equation (5) we have:

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

The case $k < i < 2k$ can be handled similarly. Also, in the case $i = k$ we have $xr^i = -x$ in $\mathbb{Z}/p\mathbb{Z}$. It follows, that for all $0 < i < 2k$, computing the product $x r^i$ simply reduces to computing a subtraction. This fact, combined with Proposition 1, motivates the development of FFT algorithms over $\mathbb{Z}/p\mathbb{Z}$.

3.5 Multiplication in $\mathbb{Z}/p\mathbb{Z}$

Let again $x, y \in \mathbb{Z}/p\mathbb{Z}$ represented by $\vec{x}, \vec{y}$ and consider the univariate polynomials $f_x, f_y \in \mathbb{Z}[T]$ associated with x, y; see Section 3.1 for this notation. To compute the product $x y$ in $\mathbb{Z}/p\mathbb{Z}$, we proceed as follows.

Algorithm 3 Computing $xy \in \mathbb{Z}/p\mathbb{Z}$ for $x, y \in \mathbb{Z}/p\mathbb{Z}$

---

    procedure BigPrimeFieldMultiplication(f_x, f_y, r, k)

1: We compute the polynomial product $f_u = f_x f_y$ in $\mathbb{Z}[T]$ modulo $T^k + 1$.

2: Writing $f_u = \sum\limits_{i=0}^{k-1} u_i T^i$, we observe that for all $0 \le i \le k-1$ we have $0 \le u_i \le kr^2$ and compute a representation $\vec{u_i}$ of u_i in $\mathbb{Z}/p\mathbb{Z}$ using the method explained in Section 3.1.

3: We compute $u_i r^i$ in $\mathbb{Z}/p\mathbb{Z}$ using the method of Section 3.4.

4: Finally, we compute the sum $\sum\limits_{i=0}^{k-1} u_i r^i$ in $\mathbb{Z}/p\mathbb{Z}$ using Algorithm 2.

    end procedure

For large values of k, $f_x f_y \mod T^k + 1$ in $\mathbb{Z}[T]$ can be computed by asymptotically fast algorithms (see the paper [4, 7]). However, for small values of k (say $k \le 8$), using plain multiplication is reasonable.

4 FFT BASICS

We review the Discrete Fourier Transform over a finite field, and its related concepts. See [12] for details.

Primitive and principal roots of unity. Let $\mathcal{R}$ be a commutative ring with units. Let $N > 1$ be an integer. An element $\omega \in \mathcal{R}$ is a *primitive* N-th root of unity if for $1 < k \le N$ we have $\omega^k = 1 \iff k = N$. The element $\omega \in \mathcal{R}$ is a *principal* N-th root of unity if $\omega^N = 1$ and for all $1 \le k < N$ we have

$$\sum_{j=0}^{N-1} \omega^{jk} = 0. \tag{8}$$

In particular, if N is a power of 2 and $\omega^{N/2} = -1$, then ω is a principal N-th root of unity. The two notions coincide in fields of characteristic 0. For integral domains every primitive root of unity is also a principal root of unity. For non-integral domains, a principal N-th root of unity is also a primitive N-th root of unity unless the characteristic of the ring $\mathcal{R}$ is a divisor of N.

The discrete Fourier transform (DFT). Let $\omega \in \mathcal{R}$ be a principal N-th root of unity. The *N-point DFT at ω* is the linear function, mapping the vector $\vec{a} = (a_0, \ldots, a_{N-1})^T$ to $\vec{b} = (b_0, \ldots, b_{N-1})^T$ by $\vec{b} = \Omega\vec{a}$, where $\Omega = (\omega^{jk})_{0 \le j,k \le N-1}$. If N is invertible in $\mathcal{R}$, then the N-point DFT at ω has an inverse which is $1/N$ times the N-point DFT at ω^{-1}.

The fast Fourier transform. Let $\omega \in \mathcal{R}$ be a principal N-th root of unity. Assume that N can be factorized to JK with $J, K > 1$. Recall Cooley-Tukey factorization formula [5]

$$\mathrm{DFT}_{JK} = (\mathrm{DFT}_J \otimes I_K)D_{J,K}(I_J \otimes \mathrm{DFT}_K)L_J^{JK}, \tag{9}$$

where, for two matrices A, B over $\mathcal{R}$ with respective formats $m \times n$ and $q \times s$, we denote by $A \otimes B$ an $mq \times ns$ matrix over $\mathcal{R}$ called the tensor product of A by B and defined by

$$A \otimes B = [a_{k\ell}B]_{k,\ell} \quad \text{with} \quad A = [a_{k\ell}]_{k,\ell}. \tag{10}$$

In the above formula, DFT_{JK}, DFT_J and DFT_K are respectively the N-point DFT at ω, the J-point DFT at ω^K and the K-point DFT at

ω^J. The *stride permutation matrix* L_J^{JK} permutes an input vector $\mathbf{x}$ of length JK as follows

$$\mathbf{x}[iJ + j] \mapsto \mathbf{x}[jJ + i], \tag{11}$$

for all $0 \le j < J$, $0 \le i < K$. If $\mathbf{x}$ is viewed as a $K \times J$ matrix, then L_J^{JK} performs a transposition of this matrix. The *diagonal twiddle matrix* $D_{J,K}$ is defined as

$$D_{J,K} = \bigoplus_{j=0}^{J-1} \mathrm{diag}(1, \omega^j, \ldots, \omega^{j(K-1)}), \tag{12}$$

Formula (9) implies various divide-and-conquer algorithms for computing DFTs efficiently, often referred to as fast Fourier transforms (FFTs). See the seminal papers [20] and [10] by the authors of the SPIRAL and FFTW projects, respectively. This formula also implies that, if K divides J, then all involved multiplications are by powers of ω^K.

5 BLOCKED FFT ON THE GPU

In the sequel of this section, let $\omega \in \mathcal{R}$ be a principal N-th root of unity. In the factorization of the matrix DFT_{JK}, viewing the size K as a base case and assuming that J is a power of K, Formula (9) translates into a recursive algorithm. This recursive formulation is, however, not appropriate for generating code targeting many-core GPU-like architectures for which, formulating algorithms iteratively facilitates the division of the work into kernel calls and thread-blocks. To this end, we shall unroll Formula (9).

NOTATION 3. *Assuming $c = 0$, that is, $N = K^e$, we define the following linear operators, for $i = 0, \ldots, e-1$:*

$$
\begin{aligned}
U_i(\omega) &= \left(I_{K^i} \otimes \mathrm{DFT}_K(\omega^{K^{e-1}}) \otimes I_{K^{e-i-1}}\right) \cdot \\
&\quad \left(I_{K^i} \otimes D_{K, K^{e-i-1}}(\omega^{K^i})\right), \\
V_i(\omega) &= I_{K^i} \otimes L_K^{K^{e-i}}, \\
W_i(\omega) &= I_{K^i} \otimes \left(L_{K^{e-i-1}}^{K^{e-i}} \cdot D_{K, K^{e-i-1}}(\omega^{K^i})\right).
\end{aligned}
\tag{13}
$$

REMARK 1. *We recall two classical formulas for tensor products of matrices. If A and B are square matrices over $\mathcal{R}$ with respective orders a and b, then we have*

$$A \otimes B = L_a^{ab} \cdot (B \otimes A) L_b^{ab}. \tag{14}$$

If C and D are two other square matrices over $\mathcal{R}$ with respective orders a and b, then we have

$$(A \otimes B) \cdot (C \otimes D) = (A \cdot C) \otimes (B \cdot D). \tag{15}$$

Our GPU implementation reported in Section 6 is based on the following two results. We omit the proofs, which can easily be derived from Remark 1 and the Cooley-Tukey factorization formula; see [18]. Computer program code can be generated from Proposition 3 using the techniques of [20].

PROPOSITION 2. *For $i = 0, \ldots, e-1$, we have*

$$U_i(\omega) = V_i(\omega)\left(I_{K^{e-1}} \otimes \mathrm{DFT}_K(\omega^{K^{e-1}})\right)W_i(\omega) \tag{16}$$

The following formula reduces the computation of a DFT on K^e points to computing e DFT's on K points.

PROPOSITION 3. *The following factorization of* $\mathrm{DFT}_{K^e}(\omega)$ *holds:*

$$\mathrm{DFT}_{K^e}(\omega) = U_0(\omega)\cdots U_{e-1}(\omega)V_{e-1}(\omega)\cdots V_0(\omega). \quad (17)$$

6 IMPLEMENTATION

In this section, we discuss implementation techniques. Our experimental results are reported in Section 7. We have realized a GPU implementation in the CUDA language of the algorithms presented in Sections 3 and 5. We have used the third and the fourth Generalized Fermat primes from Table 1, namely $P_3 := (2^{63} + 2^{34})^8 + 1$ and $P_4 := (2^{62} + 2^{36})^{16} + 1$. We have tested our code and collected the experimental data on three different types of GPU cards.

Parallelization. Performing arithmetic operations on vectors of elements of $\mathbb{Z}/p\mathbb{Z}$ has inherent data parallelism, which is ideal for implementation on GPUs. In our implementation, each arithmetic operation is computed by one thread. An alternative approach would be to use multiple threads for computing one operation. However, this would not improve performance mostly due to overhead of handling carry propagation (in the case of addition and subtraction), or increased latency because of frequent accesses to global memory (in the case of twiddle factor multiplications).

Memory-bound kernels. Performance of our GPU kernels are limited by frequent accesses to memory. Therefore, we have considered solutions for minimizing memory latency, maximizing occupancy (i.e. number of active warps on each streaming multiprocessor) to hide latency, and maximizing IPC (instructions per clock cycle).

Location of data. At execution time, each thread needs to perform computation on at least one element of $\mathbb{Z}/p\mathbb{Z}$, meaning that it will read/write at least k digits of machine-word size. Often, in such a scenario, shared memory is utilized as an auxiliary memory, but this approach has two shortcomings. First, on a GPU, each streaming multiprocessor has a limited amount of shared memory which might not be large enough for allowing each thread to keep at least one element of $\mathbb{Z}/p\mathbb{Z}$ (since the value of k can be quite large). Second, using a huge amount of shared memory will reduce occupancy. At the same time, there is no opportunity for using texture memory or constant memory when computing over $\mathbb{Z}/p\mathbb{Z}$. Conclusively, the only remaining solution is to keep all data on global memory.

Maximizing global memory efficiency. Assume that for a vector of N elements of $\mathbb{Z}/p\mathbb{Z}$, consecutive digits of each element of $\mathbb{Z}/p\mathbb{Z}$ are stored in adjacent memory addresses. Therefore, such a vector can be considered as the row-major layout of a matrix with N rows and k columns. In practice, this data structure will hurt performance due to increased memory overhead, caused by non-coalesced accesses to global memory. In this case, an effective solution is to apply a stride permutation L_k^{kN} on all input vectors (if data is stored in a row-major layout, this permutation is equivalent to transposing the input to a matrix of k rows and N columns). Therefore, all kernels are written with the assumption that consecutive digits of the same element are N steps away from each other in the memory. As a result, accesses to global memory will be coalesced, increasing memory load and store efficiency, and lowering the memory overhead.

Decomposing computation into multiple kernels. Inside a kernel, consuming too many registers per thread can lower occupancy, or even worse, lead to register spilling. In order to prevent from register spilling, register-intensive kernels are broken into multiple smaller kernels.

Size of thread blocks. Our GPU kernels do not depend on the size of a thread block. So, we choose a configuration for a thread block that will maximize the percentage of occupancy, the value of IPC (instruction per clock cycle), and bandwidth-related performance metrics such as the load and store throughput. We have achieved the best experimental results for thread blocks of 128 threads, or 256 threads.

Effect of GPU instructions on performance. Our current implementation is optimized for the primes $P_3 := (2^{63} + 2^{34})^8 + 1$ and $P_4 := (2^{62} + 2^{36})^{16} + 1$. Therefore, we rely on 64-bit instructions on GPUs. As it is explained in [6], even though 64-bit integer instructions are supported on NVIDIA GPUs, at compile time, all arithmetic and memory instructions will first be converted to a sequence of 32-bit equivalents. This might have a negative impact on the overall performance of our implementation. Specially, compared to addition and subtraction, 64-bit multiplication is computed through a longer sequence of 32-bit instructions. Finally, using 32-bit arithmetic provides more opportunities for optimization such as instruction level parallelism.

7 EXPERIMENTATION

We compare our implementation of FFT over a big prime field against a comparable approach based on FFTs over small prime fields. To be precise, we implement the two approaches discussed in Section 2. Recall that the first approach computes an FFT of size N over a big prime field of the form $\mathbb{Z}/p\mathbb{Z}$ where p is a SRGFN of size k machine words. The second approach uses $s = 2k$ half-machine word primes $p_1, \ldots, p_s$ and proceeds as follows:

(1) **projection:** compute the image f_i of f in $\mathbb{Z}/p_1\mathbb{Z}[x], \ldots,$ $\mathbb{Z}/p_k\mathbb{Z}[x]$, for $i = 1, \ldots, k$,

(2) **images:** compute the DFT of f_i at ω_i in $\mathbb{Z}/p_i\mathbb{Z}[x]$, for $i = 1, \ldots, k$ (using the CUMODP library [18]),

(3) **combination:** combine the results using CRT so as to obtain a DFT of f at ω.

We use half-machine word primes (instead of machine-word primes as discussed in Section 2) because the small prime field FFTs of the CUMODP library impose this choice. Experimental results are gathered in Section 7.1.

We also have implemented and tested a sequential, CPU version of both approaches. For the small prime field approach, we use the NTL library [22], supporting FFT modulo machine-word size primes of 60 bits. However, for the big prime field approach, we have implemented our own arithmetic in a sequential C program. Experimental results are gathered in Section 7.2.

7.1 Big prime vs small prime on the GPU

The output of the two approaches is the DFT of a vector of size N over a ring R which is either a prime field or a direct product of prime fields, and for which each element spans k machine-words. Hence these two approaches are equivalent building blocks in a modular method. For realizing the benchmark, first, we perform

the reduction step, followed by computing $s = 2k$ FFTs of size N over small prime fields. In the small field case, we use the highly optimized implementation of the following FFT algorithms from the CUMODP library (see [18, 19] and [14]): the Cooley-Tukey FFT algorithm (CT), the Cooley-Tukey FFT algorithm with precomputed powers of the primitive root (CT-pow), and the Stockham FFT algorithm. The above codes compute DFTs for input vectors of 2^n elements, where $20 \le n \le 26$ is typical.

Our CUDA implementation of the big prime field approach computes DFT over $\mathbb{Z}/p\mathbb{Z}$, for $P_3 := (2^{63} + 2^{34})^8 + 1$ and $P_4 := (2^{62} + 2^{36})^{16} + 1$, and input vectors of size $N = K^e$ where $K = 16$ for P_3, and $K = 32$ for P_4. Furthermore, for P_3, we have $2 \le e \le 5$, while for P_4 (due to the limited size of global memory on a GPU card), we have $2 \le e \le 4$.

The benchmark is computed on an NVIDIA Geforce GTX 760M (CC 3.0), an NVIDIA Tesla C2075 (CC 2.0), and an NVIDIA Tesla M2050 (CC 2.0). The first card has effective bandwidth of 48 GB/s, with 4 streaming multiprocessor, and the total number of 768 CUDA cores.
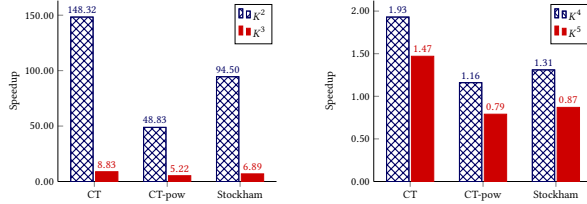

Figure 1: Speedup diagram for computing the benchmark for a vector of size $N = K^e$ ($K = 16$) for $P_3 := (2^{63} + 2^{34})^8 + 1$.
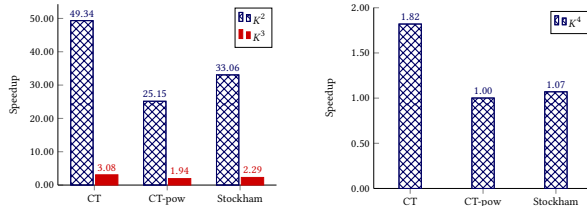

Figure 2: Speedup diagram for computing the benchmark for a vector of size $N = K^e$ ($K = 32$) for $P_4 := (2^{62} + 2^{36})^{16} + 1$.

Figures 1 and 2 show the speedup of the big prime field FFT compared to the small prime field approach, measured on the first GPU card. Moreover, Table 2 presents the running times of computing the benchmark on the mentioned GPU cards. In each table, the first three columns give the running times for computing the small prime field FFT based on the Cooley-Tukey algorithm, the Cooley-Tukey FFT algorithm with precomputed powers of the primitive root, and the Stockham algorithm, respectively. Meanwhile, the last column presents the running time for computing the big prime field FFT.

As it is reported in [18], the FFT algorithms of the CUMODP library gain speedup factors for vectors of the size 2^{16} and larger, therefore, the input vector should be large enough to keep the GPU

device busy, and thus, provide a high percentage of occupancy. This explains the results displayed on Figures 1 and 2; for both primes P_3 and P_4, when $N = K^2$ and $N = K^3$, our big prime field FFT approach significantly outperforms the small prime field FFT approach.

More importantly, for both primes P_3 and P_4, and with vectors of size $N = K^4$, our experimental results demonstrate that computing the big prime field FFT is competitive with the small prime field approach in terms of running time. For both primes P_3 and P_4, we can compute FFT for an input vector of size $N = K^4$, which is equivalent of 2^{16} and 2^{20} elements, respectively, and is large enough to cover many practical applications.

Eventually, for P_3, and for a vector of size $N = K^5$, the Cooley-Tukey (with precomputation) and Stockham FFT codes are slightly faster than the big prime field FFT. Nevertheless, for each of the tested big primes, there is a bit size range of input vectors over which the big prime field approach outperforms the small prime approach, which is coherent with the analysis of Section 2. For $P_3 := (2^{63}+2^{34})^8+1$, this range is $[2^{12}, 2^{16}]$ while for $P_4 := (2^{62}+2^{36})^{16}+1$, this range is $[2^{15}, 2^{20}]$. Our GPU implementation of the big prime field arithmetic is generic and thus can support larger SRGFNs, see Table 1.

Table 2: Running time of computing the benchmark for $N = K^e$ on GPU (timings in milliseconds).

Computing the benchmark for $N = K^e$ for $P_3 := (2^{63} + 2^{34})^8 + 1$ ($K = 16$)					Computing the benchmark for $N = K^e$ for $P_4 := (2^{62} + 2^{36})^{16} + 1$ ($K = 32$)				
Measured on a NVIDIA GTX-760M GPU					Measured on NVIDIA GTX-760M GPU				
e	CT	CT-pow	Stockham	Big FFT	e	CT	CT-pow	Stockham	Big FFT
2	8.30	2.73	5.29	0.05	2	18.30	9.33	12.26	0.37
3	10.96	6.49	8.55	1.24	3	62.98	39.74	46.72	20.40
4	50.49	30.29	34.37	26.06	4	1772.9	974.01	1042.62	971.28
5	820.82	444.07	490.72	558.22					
Measured on a NVIDIA Tesla C2075 GPU					Measured on NVIDIA Tesla C2075 GPU				
e	CT	CT-pow	Stockham	Big FFT	e	CT	CT-pow	Stockham	Big FFT
2	9.44	2.93	5.16	0.03	2	19.82	9.56	11.56	0.27
3	11.72	6.27	7.54	0.89	3	44.50	23.39	27.98	15.16
4	31.85	15.57	19.07	17.71	4	891.35	437.29	464.69	695.02
5	418.58	191.57	205.13	371.48					
Measured on a NVIDIA Tesla M2050 GPU					Measured on NVIDIA Tesla M2050 GPU				
e	CT	CT-pow	Stockham	Big FFT	e	CT	CT-pow	Stockham	Big FFT
2	12.92	3.12	5.35	0.03	2	27.22	9.91	11.62	0.27
3	15.35	6.66	8.00	0.88	3	51.81	23.93	28.60	14.80
4	35.59	15.93	19.62	17.41	4	902.35	449.53	465.51	678.34
5	424.98	198.46	206.71	364.88					

Figure 3 shows the percentage of time spent in each operation in order to compute the big prime field FFT on a randomly generated input vector of size $N = K^4$ (measured for both primes and on the first mentioned GPU card). As illustrated, for both primes, computation follows a similar pattern, where multiplication by twiddle factors is the main bottleneck. Finally, Table 3 presents the profiling data for computing the base-case DFT_K on a GTX 760M GPU.
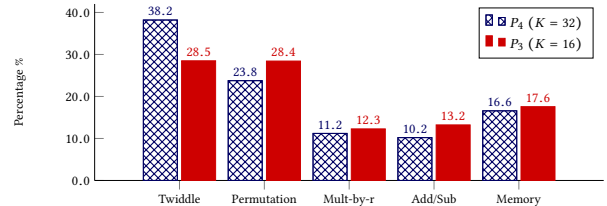

Figure 3: Running time of computing DFT_N with $N = K^4$ on a GTX 760M GPU.

Table 3: Profiling results for computing base-case DFT_K **on a GTX 760M GPU (collected using NVIDIA** `nvprof`**).**

Measured on a GTX760M GPU	$P_3 = (2^{63} + 2^{34})^8$ ($K = 16$)		$P_4 = (2^{62} + 2^{36})^{16}$ ($K = 32$)	
Metric Description	Mult by r	Add/Sub	Mult by r	Add/Sub
Achieved Occupancy	74%	45%	62%	46%
Device Memory Read Throughput	22.0 GB/s	20.7 GB/s	22.50 GB/s	10.00 GB/s
Device Memory Write Throughput	22.0 GB/s	20.7 GB/s	23.19 GB/s	10.06 GB/s
Global Memory Replay Overhead	0.28	0.03	0.25	0.01
Executed IPC	0.72	2.41	0.78	4.56
Instruction Replay Overhead	0.47	0.13	0.53	0.028
Global Store Throughput	22.08 GB/s	20.74 GB/s	44.42 GB/s	9.91 GB/s
Global Load Throughput	24.57 GB/s	20.91 GB/s	46.39 GB/s	10.22 GB/s
Global Memory Load Efficiency	90.44%	43.60%	48.70%	98.86%
Global Memory Store Efficiency	94.95%	43.75%	49.35%	99.99%

7.2 CPU vs. GPU implementations

Table 4 gathers running times for computing FFT sequentially with both small prime and big prime approaches, on three different CPUs (measured in milliseconds). In addition, Table 5 shows the speedup range for computing the small and the big prime field approaches on CPU and GPU. For each prime, the first and the second column show the lowest and the highest running time of the same approach on CPU and GPU, respectively. Also, the last column contains the lowest and the highest speedup ratio of computing the same approach on CPU to its counterpart on GPU .

Table 4: Running time of computing the benchmark for $N = K^e$ **using sequential C code on CPU (timings in milliseconds).**

Measured on Intel Xeon X5650 @ 2.67GHz CPU			Measured on AMD FX(tm)-8350 @ 2.40GHz CPU			Measured on Intel Core i7-4700HQ @ 2.40GHz CPU		
Computing the benchmark for $N = K^e$ for $P_3 := (2^{63} + 2^{34})^8 + 1$ ($K = 16$)			Computing the benchmark for $N = K^e$ for $P_3 := (2^{63} + 2^{34})^8 + 1$ ($K = 16$)			Computing the benchmark for $N = K^e$ for $P_3 := (2^{63} + 2^{34})^8 + 1$ ($K = 16$)		
e	NTL Small FFT	C Big FFT	e	NTL Small FFT	C Big FFT	e	NTL Small FFT	C Big FFT
2	2.51	1.85	2	4.06	4.13	2	3.12	0.73
3	23.19	35.08	3	16.06	20.01	3	14.19	21.06
4	372.19	750.40	4	296.00	528.00	4	232.76	505.96
Computing the benchmark for $N = K^e$ for $P_4 := (2^{62} + 2^{36})^{16} + 1$ ($K = 32$)			Computing the benchmark for $N = K^e$ for $P_4 := (2^{62} + 2^{36})^{16} + 1$ ($K = 32$)			Computing the benchmark for $N = K^e$ for $P_4 := (2^{62} + 2^{36})^{16} + 1$ ($K = 32$)		
e	NTL Small FFT	C Big FFT	e	NTL Small FFT	C Big FFT	e	NTL Small FFT	C Big FFT
2	14.94	12.91	2	12.00	8.00	2	12.48	9.79
3	384.10	692.16	3	296.00	396.00	3	233.26	496.03
4	11303.76	33351.29	4	10128.00	22992.00	4	7573.65	26089.53

Table 5: Speedup ratio $(\frac{T_{\text{CPU}}}{T_{\text{GPU}}})$ **for computing the benchmark for** $N = K^e$ **for** P_3 **and** P_4 **(timings in milliseconds).**

Computing the benchmark for $N = K^e$ for $P_3 := (2^{63} + 2^{34})^8 + 1$ ($K = 16$) (timings in milliseconds)			Computing the benchmark for $N = K^e$ for $P_4 := (2^{62} + 2^{36})^{16} + 1$ ($K = 32$) (timings in milliseconds)		
e	NTL Small FFT	Small FFT GPU	Speed-up		
2	2.51 - 4.06	2.73 - 12.92	0.19X - 1.48X		
3	14.19 - 23.19	6.27 - 15.35	0.92X - 3.69X		
4	232.76 - 372.19	15.57 - 50.49	4.61X - 23.90X		
e	C Big FFT	BigFFT GPU	Speed-up		
2	0.73-4.13	0.03-0.05	14.6X - 137.6X		
3	20.01-35.08	0.88-1.24	16.13X - 39.86X		
4	505.96 - 750.40	17.41-26.06	19.41X - 43.10X		

e	NTL Small FFT	Small FFT GPU	Speed-up
2	12.00 - 14.94	9.33 - 27.22	0.44X - 1.60X
3	233.26 - 384.10	23.39 - 62.98	3.70X - 16.42X
4	7573.65 - 11303.76	437.29 - 1772.92	4.27X - 25.84X
e	C Big FFT	BigFFT GPU	Speed-up
2	8.00 - 12.91	0.27 - 0.37	21.62X - 47.81X
3	396.00 - 692.16	14.80 - 20.80	19.03X - 46.76X
4	22992.00 - 33351.29	695.02 - 971.28	23.67X - 479.62X

8 CONCLUSION

Our results show the advantage of the big prime field approach. To be precise, for a range of vector sizes, one can find a suitable large prime modulo which FFTs outperform the CRT-based approach. Our current implementation is preliminary and much optimization is possible, in particular for the multiplication in the big prime field. We anticipate better performance on more recent GPU cards with larger memory bandwidth and better support for 64-bit arithmetic. The CUDA code presented in this article is part of the CUMODP library freely available at www.cumodp.org.

REFERENCES

[1] RC Agarwal and C Burrus. 1974. Fast convolution using Fermat number transforms with applications to digital filtering. *IEEE Transactions on Acoustics, Speech, and Signal Processing* 22, 2 (1974), 87–97.

[2] E. A. Arnold. 2003. Modular Algorithms for Computing Gröbner Bases. *J. Symb. Comput.* 35, 4 (2003), 403–419. https://doi.org/10.1016/S0747-7171(02)00140-2

[3] C. Chen, S. Covanov, F. Mansouri, M. Moreno Maza, N. Xie, and Y. Xie. 2014. The Basic Polynomial Algebra Subprograms, See [15], 669–676. https://doi.org/10.1007/978-3-662-44199-2_100

[4] C. Chen, S. Covanov, F. Mansouri, M. Moreno Maza, N. Xie, and Y. Xie. 2016. Parallel Integer Polynomial Multiplication. In *SYNASC 2016*. 72–80. https://doi.org/10.1109/SYNASC.2016.024

[5] J. Cooley and J. Tukey. 1965. An Algorithm for the Machine Calculation of Complex Fourier Series. *Math. Comp.* 19, 90 (1965), 297–301.

[6] NVIDIA Corporation. 2016. CUDA C Programming Guide, v8.0. (September 2016).

[7] S. Covanov and E. Thomé. 2015. Fast arithmetic for faster integer multiplication. *CoRR* abs/1502.02800 (2015). http://arxiv.org/abs/1502.02800

[8] X. Dahan, M. Moreno Maza, É. Schost, W. Wu, and Y. Xie. 2005. Lifting techniques for triangular decompositions. In *ISSAC 2005, Proceedings*, M. Kauers (Ed.). ACM, 108–115. https://doi.org/10.1145/1073884.1073901

[9] V. S. Dimitrov, T. V. Cooklev, and B. D. Donevsky. 1994. Generalized Fermat-Mersenne number theoretic transform. *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing* 41, 2 (Feb 1994), 133–139. https://doi.org/10.1109/82.281844

[10] M. Frigo and S. G. Johnson. 2005. The design and implementation of FFTW3. In *Proceedings of the IEEE*, Vol. 93. 216–231.

[11] M. Fürer. 2007. Faster integer multiplication. In *Proceedings of the 39th Annual ACM Symposium on Theory of Computing, San Diego, California, USA, June 11-13, 2007*, D. S. Johnson and U. Feige (Eds.). ACM, 57–66. https://doi.org/10.1145/1250790.1250800

[12] M. Fürer. 2009. Faster Integer Multiplication. *SIAM J. Comput.* 39, 3 (2009), 979–1005. https://doi.org/10.1137/070711761

[13] D. Hankerson, A. J. Menezes, and S. Vanstone. 2004. *Guide to Elliptic Curve Cryptography*. Springer-Verlag New York, Inc. http://opac.inria.fr/record=b1101039

[14] S. A. Haque, X. Li, F. Mansouri, M. Moreno Maza, W. Pan, and N. Xie. 2014. Dense Arithmetic over Finite Fields with the CUMODP Library, See [15], 725–732. https://doi.org/10.1007/978-3-662-44199-2_108

[15] H. Hong and C. Yap (Eds.). 2014. *Mathematical Software - ICMS 2014 - 4th International Congress, Seoul, South Korea, August 5-9, 2014. Proceedings.* Lecture Notes in Computer Science, Vol. 8592. Springer. https://doi.org/10.1007/978-3-662-44199-2

[16] X. Li, M. Moreno Maza, R. Rasheed, and É. Schost. 2011. The modpn library: Bringing fast polynomial arithmetic into Maple. *J. Symb. Comput.* 46, 7 (2011), 841–858. https://doi.org/10.1016/j.jsc.2010.08.016

[17] D. Mohajerani. 2016. *Fast Fourier Transforms over Prime Fields of Large Characteristic and their Implementation on Graphics Processing Units.* Master's thesis. The University of Western Ontario, London, ON, Canada. http://ir.lib.uwo.ca/cgi/viewcontent.cgi?article=6094&context=etd.

[18] M. Moreno Maza and W. Pan. 2010. Fast polynomial arithmetic on a GPU. *J. of Physics: Conference Series* 256 (2010), 10.

[19] M. Moreno Maza and W. Pan. 2011. Solving Bivariate Polynomial Systems on a GPU. *J. of Physics: Conference Series* 341 (2011), 14.

[20] M. Püschel, J. M. F. Moura, J. Johnson, D. Padua, M. Veloso, B. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo. 2005. SPIRAL: Code Generation for DSP Transforms. *Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation"* 93, 2 (2005), 232–275.

[21] A. Schönhage and V. Strassen. 1971. Schnelle Multiplikation großer Zahlen. *Computing* 7, 3-4 (1971), 281–292. https://doi.org/10.1007/BF02242355

[22] V. Shoup. 2016. Number theory library (NTL) for C++. *Available at Shoup's homepage http://www.shoup.net/ntl/* (2016).

[23] J. von zur Gathen and J. Gerhard. 2003. *Modern Computer Algebra* (2 ed.). Cambridge University Press, New York, NY, USA.