Big Prime Field FFT on the GPU
Liangyu Chen, Svyatoslav Covanov, Davood Mohajerani, Marc Moreno Maza

To cite this version:
Liangyu Chen, Svyatoslav Covanov, Davood Mohajerani, Marc Moreno Maza. Big Prime Field FFT on the GPU. ISSAC 2017, Jul 2017, Kaiserslautern, Germany. pp.85-92, 10.1145/3087604.3087657. hal-01518830

HAL Id: hal-01518830
https://hal.archives-ouvertes.fr/hal-01518830

Submitted on 5 May 2017

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
Big Prime Field FFT on the GPU

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

Davood Mohajerani
University of Western Ontario
dmohajer@uwo.ca

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

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.

Categories and Subject Descriptors
I.1.2 [Symbolic and Algebraic Manipulation]: Algorithms—Algebraic algorithms, Analysis of algorithms

General Terms
Algorithms, Experimentation

Keywords
Fast Fourier transforms; Finite fields; Generalized Fermat 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 have also 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, 3].

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.

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 (Graphics Processing Units) implementation of those algorithms as well as a GPU implementation of a Fast Fourier Transform (FFT) over such 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 2$k$ 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 to implement 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 www.cumodp.org we use them in our experimental comparison.

Section 4 reports on various comparative experiments. 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 NTT (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 describing the computation of a NTT, or a FNT [1, 2].
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 [11], which is derived from Füredi’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 [11] discusses the idea of using Fermat number transform for computing convolutions, thus working modulo numbers of the form \( p = 2^b + 1 \), 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 by twiddle factor multiplication in computing DFT over the field of complex numbers. The paper [9] considers by twiddle factor multiplication in computing DFT over the field of complex numbers.

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 \( \eta \). We denote by \( \omega \) such an element. Let \( f \in \mathbb{Z}/p\mathbb{Z}[x] \) be a polynomial of degree at most \( N - 1 \). Then, computing the DFT of \( f \) at \( \omega \) 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 \( \omega \) 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 \( \omega \) costs \( O(M(k)) \) machine-word operations,

where \( n \rightarrow M(n) \) is a multiplicative time as defined in [23]. Therefore, multiplication by a power of \( \omega \) becomes a bottleneck as \( k \) grows. To overcome this difficulty, we consider the following trick proposed by Martin Füredi in [11] [14]. We assume that \( N = K^r \) holds for some ‘small’ \( K \), say \( K = 32 \) and an integer \( e \geq 2 \). Further, we define \( \eta = \omega^{N/K} \) with \( J = K^{-1} \) and assume that multiplying an arbitrary element of \( \mathbb{Z}/p\mathbb{Z} \) by \( \eta^i \), 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 \( n \) 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 \( \omega \) and using the factorization formula of Cooley and Tukey, we have

\[
DFT_{JK} = (DFT_J \otimes K)D_JK(I_J \otimes DFT_K)L_JK^{-1},
\]

(1)

see Section 2. Hence, the DFT of \( f \) at \( \omega \) is essentially performed by:

1. \( K^{-1} \) DFT’s of size \( K \) (that is, DFT’s on polynomials of degree at most \( K - 1 \)),
2. \( N \) multiplications by a power of \( \omega \) (coming from the diagonal matrix \( D_{I,K} \)) and
3. \( K \) DFT’s of size \( K^{-1} \).

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 deduce that a DFT of size \( N = K^r \) reduces to:

1. \( e K^{-1} \) DFT’s of size \( K \), and
2. \( (e - 1)N \) multiplication by a power of \( \omega \).

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

\[
O(N \log_2(N) k + N \log_2(N) M(k)) \quad (2)
\]

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

\[
O(N \log_2(N) k + N \log_2(N) M(k)) \quad (3)
\]

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 \( O(N \log_2(N) M(k)) \) 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 \( p_1, \ldots, p_k \) 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 \( \mathbb{Z}/p_i\mathbb{Z} \) is an \( N \)-th primitive root of unity, \( \omega, \ldots, \omega_k \). Then, \( \omega = (\omega_1, \ldots, \omega_k) \) is an \( N \)-th primitive root of \( \mathbb{Z}/m\mathbb{Z} \). Indeed, the ring \( \mathbb{Z}/p_1\mathbb{Z} \oplus \cdots \oplus \mathbb{Z}/p_k\mathbb{Z} \) 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 \( \omega \) 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 \( \omega_i \) 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 \( \omega \).

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

\[
O(N \log_2(N) k + N M(k) \log_2(k)) \quad (4)
\]

These estimates yield a running-time ratio between the two approaches of \( \log(N) / \log_2(k) \), 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 4.

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 [21] [12].

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, $F_0$ 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 $a^2 + b^2$ where $a > 1$, $b > 0$ and $n > 0$ are called generalized Fermat numbers. An odd prime $p$ is a generalized Fermat number if and only if $p$ is congruent to 3 modulo 4. In particular, we denote by $\nu(p)$ the largest power of 2 dividing $p - 1$. For some integers $2^n u$ and $v$, we have $\nu(p) = 2^n u v - u v - 1$, so that $p = 2^n u v + 1$ holds.

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.

<table>
<thead>
<tr>
<th>$p$</th>
<th>$\nu(2^n u v - 1)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(2^{2^r} + 2^{2^s})^2 + 1$</td>
<td>$2^{106}$</td>
</tr>
<tr>
<td>$(2^{2^r} + 2^{2^s})^3 + 1$</td>
<td>$2^{100}$</td>
</tr>
<tr>
<td>$(2^{2^r} + 2^{2^s})^4 + 1$</td>
<td>$2^{114}$</td>
</tr>
<tr>
<td>$(2^{2^r} + 2^{2^s})^5 + 1$</td>
<td>$2^{116}$</td>
</tr>
<tr>
<td>$(2^{2^r} + 2^{2^s})^{10} + 1$</td>
<td>$2^{176}$</td>
</tr>
<tr>
<td>$(2^{2^r} + 2^{2^s})^{12} + 1$</td>
<td>$2^{1992}$</td>
</tr>
<tr>
<td>$(2^{2^r} - 2^{2^s})^{10} + 1$</td>
<td>$2^{1992}$</td>
</tr>
<tr>
<td>$(2^{2^r} - 2^{2^s})^{12} + 1$</td>
<td>$2^{1994}$</td>
</tr>
</tbody>
</table>

Note 1. In the sequel, we consider $p = F_n(r)$, a fixed SRGN. We denote by $2^n$ the largest power of 2 dividing $p - 1$ and we define $k = 2^n$, so that $p = k^n + 1$ 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 $\omega(N/2k)$ for $0 \leq i < k$ 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

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

This representation is made unique by imposing the following constraints

1. either $x_{k-1} = r$ and $x_{k-2} = \ldots = x_1 = 0$,  
2. or $0 < x < r$ for all $i = 0, \ldots, (k - 1)$.

We also map 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 $x \equiv f(x) \mod p$.

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 $z < r^k$ holds for some positive integer $i \leq n = \log_2(p)$, we denote by vec($z, i$) the unique sequence of $2^n$ non-negative integers $(z_{2^{i-1}}, \ldots, z_0)$ such that we have $0 \leq z_i < r$ and $z = \sum_{i=0}^{k-1} r^{k-1} \ldots + z_0$. The sequence vec($z, i$) is obtained as follows:

1. if $i = 1$, we have vec($z, i$) = $(q, s)$,
2. if $i > 1$, then vec($z, i$) is the concatenation of vec($q, i - 1$) followed by vec($s, i - 1$),

where $q$ and $s$ are the quotient and the remainder in the Euclidean division of $z$ by $r^{i-1}$. Clearly, vec($x, n$) = $\vec{x}$ holds.

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

$$z = 2^n z_{\text{high}} + z_{\text{low}} = r z_{\text{high}} + z_{\text{low}} - 2^n z_{\text{high}}. \hspace{1cm} (6)$$

Let $s = z_{\text{low}} - 2^n z_{\text{high}}$ 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 $\mathbb{Z}/p\mathbb{Z}$

Note 2. Let $N$ be a power of 2, say $2^k$, 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 $\omega = \alpha^N$. Little Fermat theorem implies that either $\omega^{N/2} = 1$ or $\omega^{N/2} = -1$ holds. In the latter case, $\omega$ 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, if 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 $\omega$ 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 $\omega$.

**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)
    α := g^N/2k
    β := α
    j := 1
    while β + r do
        β := αβ
        j := j + 1
    end while
    ω := g^j
    return (ω)
end procedure
```

**Proof** Since $g^{N/2k}$ is a $2k$-th root of unity, it is equal to $r^{i0}$ (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

$$g^{j2^l/2k} = (g^{j2^l/2k})^{2^l/2k} = g^{j2^l/2k} = r^{j i_0},$$

where $q$ and $i$ are quotient and the remainder of $j$ in the Euclidean division by $2k$. By definition of $g$, the powers $g^{j2^l/2k}$, for $0 \leq j < 2k$, are pairwise different. It follows from Formula (7) that the elements $r^{j i_0}$ are pairwise different as well, for $0 \leq j < 2k$. Therefore, one of those latter elements is $r$ itself. Hence, we have $j_i$ with $0 \leq j_i < 2k$ such that $g^{j_i N/2k} = r$. Then, $\omega = g^{j_0}$ 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 $x + y$ 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 $x_{k-1}$. Since $z_{k-1} \ldots z_0$ 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 $\vec{x} + \vec{y}$.

A similar procedure computes the vector $\vec{x} - \vec{y}$ 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 $z$ 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 a root, $y = r^i$ for some $0 \leq i < 2k$. Note that the cases $i = 0 = 2k$ are trivial. Indeed, recall that $r$ is a $2k$-th primitive root of unity in $\mathbb{Z}/p\mathbb{Z}$. Hence, for $0 \leq i < k$, we have $r^{i+1} = -r^i$ 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 (6) we have:

$$xr^i \equiv (x_{k-1} r^{h_{k-1}+1} + \ldots + x_0 r^1) \mod p \equiv \sum_{j=0}^{j=k-1} x_{j} r^{j+1} \mod p \equiv \sum_{h=\text{h-1}}^{h=1} x_{h} r^{h+1} \mod p \equiv \left( \sum_{h=\text{h}}^{h=0} x_{h} r^{h} - \sum_{h=\text{h}}^{h=0} x_{h} r^{h-k} \right) \mod p$$

The case $k < i < 2k$ can be handled similarly. Also, in the case $i = k$ we have $x r^i = -x$ 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. 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 [12] and [21]). However, for small values of $k$ (say $k \leq 8$), using plain multiplication is reasonable.
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 $f_u = \sum_{i=0}^{k-1} u_i T^i$, we observe that for all $0 \leq i \leq k - 1$ we have $0 \leq u_i \leq kr^2$ and compute a representation $u_i$ 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 3.3.
4. Finally, we compute the sum $\sum_{i=0}^{k-1} u_i r^i$ in $\mathbb{Z}/p\mathbb{Z}$ using Algorithm 2.
end procedure

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 $1 < k \leq N$ 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 $1 \leq k < N$ we have

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

In particular, if $N$ is a power of 2 and $\omega^{N/2} = -1$, then $\omega$ 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 $\omega$ is the linear function, mapping the vector $\tilde{a} = (a_0, \ldots, a_{N-1})^T$ to $\tilde{b} = (b_0, \ldots, b_{N-1})^T$ by $\tilde{b} = \tilde{a} \tilde{\omega}$, where $\tilde{\omega} = (\omega^k)_{0 \leq k < N}$. If $N$ is invertible in $\mathcal{R}$, then the $N$-point DFT at $\omega$ has an inverse which is $1/N$ times the $N$-point DFT at $\omega^{-1}$.

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

$$DFT_{J \cdot K} = (DFT_J \otimes I_K)(I_J \otimes DFT_K)L_{J \cdot K}^{IK}, \quad (9)$$

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 $m q \times n s$ matrix over $\mathcal{R}$ called the tensor product of $A$ by $B$ and defined by

$$A \otimes B = [a_{k, l}]_{k, l} \text{ with } A = [a_{k, l}]_{k, l}. \quad (10)$$

In the above formula, $DFT_{J \cdot K}$, $DFT_J$ and $DFT_K$ are respectively the $N$-point DFT at $\omega$, the $J$-point DFT at $\omega^K$ and the $K$-point DFT at $\omega^J$. The stride permutation matrix $L_{J \cdot K}^{IK}$ permutes an input vector $x$ of length $J$ as follows

$$x[iJ + j] \mapsto x[jJ + i]. \quad (11)$$

for all $0 \leq j < J$, $0 \leq i < K$. If $x$ is viewed as a $K \times J$ matrix, then $L_{J \cdot K}^{IK}$ performs a transposition of this matrix.

The diagonal twiddle matrix $D_{J, K}$ is defined as

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

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 $\omega^K$.

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 $DFT_{J \cdot K}$, 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, $N = K^c$, we define the following linear operators, for $i = 0, \ldots, e - 1$:

$$U_i(\omega) = \left( I_{K^i} \otimes DFT_K(\omega^{K^{c-i}}) \otimes I_{K^{e-i-1}} \right) \left( I_{K^i} \otimes DFT_K, K^{c-i-1}, \omega^{K^i} \right), \quad (13)$$

$$V_i(\omega) = I_{K^i} \otimes L_{K^{c-i}}^{K^{c-i}}, \quad (14)$$

$$W_i(\omega) = I_{K^i} \otimes \left( L_{K^{c-i-1}}^{K^{c-i-1}} \cdot DFT_{K^{c-i-1}}(\omega^{K^i}) \right). \quad (15)$$

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

$$A \otimes B = I_a^b \cdot (B \otimes A) L_b^a. \quad (16)$$

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

$$(A \otimes B) \cdot (C \otimes D) = (A \cdot C) \otimes (B \cdot D). \quad (17)$$

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 [13].

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

$$U_i(\omega) = V_i(\omega) \left( I_{K^{e-i-1}} \otimes DFT_K(\omega^{K^{c-i}}) \right) W_i(\omega) \quad \text{(16)}$$

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

PROPOSITION 3. The following factorization of $DFT_{K^c}(\omega)$ holds:

$$DFT_{K^c}(\omega) = U_0(\omega) \cdots U_{e-1}(\omega) V_{e-1}(\omega) \cdots V_0(\omega). \quad (17)$$
6. IMPLEMENTATION

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 $P_3 := (2^{63} + 2^{21} + 1)$ and $P_4 := (2^{64} + 2^{32} + 1)$. We have tested our code and collected the experimental data on three different GPU cards.

In this section, we discuss implementation techniques. Our experimental results are reported in Section 7.

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 would be to use multiple threads for computing one operation. However, it would not improve the performance mostly due to overhead of handling propagation of carry (in case of addition and subtraction), or increased latency because of frequent accesses to global memory (in 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 $\mathbb{Z}/p\mathbb{Z}$ (which depending on the value of $k$, can be quite large). Second, using a huge amount of shared memory will reduce the occupancy. At the same time, there is no use for texture memory and constant memory for 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 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 that is caused by non-coalesced accesses to global memory. In this case, an effective solution is to apply a stride permutation $L_k^{N}$ 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 the occupancy, or even worse, can lead to register spilling. In order to prevent 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 $P_3 := (2^{63} + 2^{21} + 1)$ and $P_4 := (2^{64} + 2^{32} + 1)$. Therefore, we rely on 64-bit instructions on GPUs. As it is explained in [8], 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 $\mathbb{Z}/p_i\mathbb{Z}[x], \ldots, \mathbb{Z}/p_s\mathbb{Z}[x]$ for $i = 1, \ldots, k$,
2. **images**: compute the DFT of $f_i$ at $\omega_i$ in $\mathbb{Z}/p_i\mathbb{Z}[x]$, for $i = 1, \ldots, k$ (using the CUMODP library [13]),
3. **combination**: combine the results using CRT so as to obtain a DFT of $f$ at $\omega$.

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 7.2.

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 7.3.

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 $s = 2k$ FFTTs 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 [13] [19] and [14]): the Cooley-Tukey FFT algorithm (CT), the Cooley-Tukey FFT algorithm with pre-computed powers of the primitive root (CT-pow), and the Stockham FFT algorithm. The above codes compute DFTTs for input vectors of $2^n$ elements, where $20 \leq n \leq 26$ is typical.

Our CUDA implementation of the big prime field approach computes DFT over $\mathbb{Z}/p\mathbb{Z}$, for $P_3 := (2^{63} + 2^{21} + 1)$.
and $P_k := (2^{62} + 2^{36})^{16} + 1$, 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 $2 \leq e \leq 5$, while for $P_4$ (due to the limited size of global memory on a GPU card), we have $2 \leq e \leq 4$.

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.

Eventually, for $P_3$, and for a vector of size $N = K^2$, 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 $P_3 := (2^{62} + 2^{36})^{16} + 1$, this range is $[2^{12}, 2^{16}]$ while $P_4 := (2^{62} + 2^{36})^{16} + 1$, 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).

<table>
<thead>
<tr>
<th>$P_k$</th>
<th>$N = K^e$</th>
<th>$K = 16$</th>
<th>$K = 32$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_3$</td>
<td>$2^{62} + 2^{36}$</td>
<td>100</td>
<td>240</td>
</tr>
<tr>
<td>$P_4$</td>
<td>$2^{62} + 2^{36}$</td>
<td>100</td>
<td>240</td>
</tr>
</tbody>
</table>

Table 3: Running time of computing the benchmark for $N = K^e$ using NTL library on CPU (timings in milliseconds).

<table>
<thead>
<tr>
<th>$P_k$</th>
<th>$N = K^e$</th>
<th>$K = 16$</th>
<th>$K = 32$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_3$</td>
<td>$2^{62} + 2^{36}$</td>
<td>100</td>
<td>240</td>
</tr>
<tr>
<td>$P_4$</td>
<td>$2^{62} + 2^{36}$</td>
<td>100</td>
<td>240</td>
</tr>
</tbody>
</table>

Table 4: Speedup ratio ($\frac{\text{Speedup}}{\text{GPU}}$) for computing the benchmark for $N = K^e$ for $P_3$ and $P_4$ (timings in milliseconds).

<table>
<thead>
<tr>
<th>$P_k$</th>
<th>$N = K^e$</th>
<th>$K = 16$</th>
<th>$K = 32$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_3$</td>
<td>$2^{62} + 2^{36}$</td>
<td>100</td>
<td>240</td>
</tr>
<tr>
<td>$P_4$</td>
<td>$2^{62} + 2^{36}$</td>
<td>100</td>
<td>240</td>
</tr>
</tbody>
</table>

Figure 1 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 it illustrated, for both primes, computation follows a similar pattern, with multiplication by twiddle factors as the main bottleneck. Finally, Table 5 presents the profiling data for computing the base-case DFT$_K$ on a GTX 760M GPU.

Table 5: Profiling results for computing base-case DFT$_K$ on a GTX 760M GPU (collected using NVIDIA nvprof).

<table>
<thead>
<tr>
<th></th>
<th>Measured on a GTX760M GPU</th>
<th>$P_2 = (2^6 + 2 + 1)$</th>
<th>$K = 16$</th>
<th>$P_3 = (2^9 + 2^4 + 1)$</th>
<th>$K = 32$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Metric Description</td>
<td>Shrink by</td>
<td>Add/Sub</td>
<td>Mult by</td>
<td>Add/Sub</td>
<td>Mult by</td>
</tr>
<tr>
<td>Static Occupancy</td>
<td>84%</td>
<td>84%</td>
<td>73%</td>
<td>73%</td>
<td></td>
</tr>
<tr>
<td>Device Memory Read Throughput (GB/s)</td>
<td>22.09</td>
<td>22.09</td>
<td>22.09</td>
<td>22.09</td>
<td></td>
</tr>
<tr>
<td>Device Memory Write Throughput (GB/s)</td>
<td>22.09</td>
<td>22.09</td>
<td>22.09</td>
<td>22.09</td>
<td></td>
</tr>
<tr>
<td>Global Memory Read Throughput (GB/s)</td>
<td>9.78</td>
<td>9.78</td>
<td>9.78</td>
<td>9.78</td>
<td></td>
</tr>
<tr>
<td>Global Memory Write Throughput (GB/s)</td>
<td>9.78</td>
<td>9.78</td>
<td>9.78</td>
<td>9.78</td>
<td></td>
</tr>
<tr>
<td>Global Memory Read Efficiency</td>
<td>80.41%</td>
<td>80.41%</td>
<td>80.41%</td>
<td>80.41%</td>
<td></td>
</tr>
<tr>
<td>Global Memory Write Efficiency</td>
<td>80.41%</td>
<td>80.41%</td>
<td>80.41%</td>
<td>80.41%</td>
<td></td>
</tr>
</tbody>
</table>

7.2 CPU vs GPU implementations

Table 6 presents the sequential running time of computing the benchmark for both primes on three different CPUs (measured in milliseconds). In addition, Table 4 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.

8. CONCLUSION

Our results show the advantage of the big prime field approach. To be precise, for a size range of vectors, one can find a suitable large prime module 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 with this article is part of the CUMODP library freely available at www.cumodp.org.

Acknowledgements

This work is supported by IBM Canada Ltd (CAS project 880) and NSERC of Canada (CRD grant CRDPJ500717-16).