Fast polynomial multiplication on a GPU

Marc Moreno Maza and Wei Pan
University of Western Ontario, London ON, Canada
E-mail: moreno@csd.uwo.ca; wpan9@csd.uwo.ca

Abstract. We present CUDA implementations of Fast Fourier Transforms over finite fields. This allows us to develop GPU support for dense univariate polynomial multiplication leading to speedup factors in the range $21 - 37$ with respect to the best serial C-code available to us, for our largest input data sets. Since dense univariate polynomial multiplication is a core routine in symbolic computation, this is promising result for the integration of GPU support into computer algebra systems.

1. Introduction

Polynomials and matrices are the fundamental objects on which most computer algebra algorithms operate. In the past 15 years, significant efforts have been deployed by different groups of researchers for delivering highly efficient software packages for computing symbolically with polynomials and matrices, like LINBOX, MAGMA, and NTL [16, 1, 25]. However, most of these works are dedicated to serial implementation, in particular in the case of polynomials. Only a few studies [18, 21, 22] report on parallel implementation (targeting multicores) of polynomial arithmetic. None of the computer algebra software packages available today takes advantage of graphics processing units (GPUs) in support of libraries for polynomial arithmetic. The work reported in this paper aims at filling this gap.

This contrasts sharply with the state of affairs in numerical linear algebra and in digital signal processing. For instance, the commercialized software system MATLAB with its Parallel Computing Toolbox [17] and GPU Toolbox [13] provides programming support for different parallelism paradigms (data-parallelism, MPI, multithreading) and parallel architectures (GPUs, multicores, clusters) together with many library functions taking advantage of this support. In digital signal processing, in particular for the computation of Fast Fourier Transforms (FFTs), the use of hardware acceleration technologies, notably GPUs, has been investigated in several works [10, 20, 12, 27].

In this paper, we present a GPU implementation of fast polynomial multiplication. We focus on dense univariate polynomials over prime fields for the following reasons. First, many algorithms in symbolic computation tend to densify intermediate data, even if the input and output are sparse. Second, multivariate polynomial multiplication can be reduced to univariate multiplication through the so-called Kronecker's substitution. Third, computation with polynomials over non-prime fields can be reduced to the prime field case by means of modular techniques. We refer to the landmark book *Modern Computer Algebra* [11] for an extensive presentation of these ideas.

This reduction to dense univariate polynomial over coefficient fields $\mathbb{Z}/p\mathbb{Z}$, where p is a prime, allows us to rely on FFT techniques, which is the basis of fast polynomial arithmetic [4]. However, as detailed in Section 2.3, FFT computations over finite fields present specific challenges. For this reason, techniques for FFTs with floating point number coefficients are not sufficient for supporting polynomial multiplication over finite fields. This motivates the work reported in this paper.

Most serial implementations of FFT over finite fields, see [8] and the references therein, rely on the radix-2 Cooley-Tukey Formula [6]. On multicores, the row-column FFT algorithm is used successfully, see [21, 22]. In the case of GPUs, to which this paper is devoted, it is natural to revisit the popular FFT formulas of Cooley-Tukey and Stockham [26] in the context of finite fields. We review these formulas in Section 2.2. As in [23, 5] we take advantage of the formalism of tensorial calculus to generate code and identify our GPU kernels. The Cooley-Tukey and Stockham formulas differ only in the way that intermediate computations are stored. We concentrated our efforts on these two formulas, despite of the existence of other formulas for computing FFTs, for the following reasons. First, the radix-2 Cooley-Tukey formula is well understood in the context of finite fields. Second, in numerical computing, the Stockham formula seems to be well-suited for GPU implementation [10].

In this work, we present our detailed implementations of the Cooley-Tukey and Stockham FFT formulas, aiming at utilizing the horsepower of Graphics Processing Units (GPUs). The organization of the paper is as follows. In Section 2, we first formalize FFTs in terms of Kronecker (tensor) product, then we discuss an efficiency-critical operation in a finite field, namely modular multiplication. Sections 3 and Section 4 focus on our CUDA [2] implementation of the Cooley-Tukey and Stockham FFTs. We present experimental results in Section 5 and draw conclusions in the end.

2. The Kronecker product and FFT over finite fields

This section reviews the Fast Fourier Transform (FFT) in the language of tensorial calculus, see [28] for an extensive presentation. This formalism facilitates code generation as explained in [5, 9], and in particular it helps identifying GPU kernel specifications. We also highlight the specific features of FFTs over finite fields and refer to [22] for details. Throughout this paper, we denote by $\mathbb{K}$ a field. In practice, this field is often a prime field $\mathbb{Z}/p\mathbb{Z}$ where p is a prime number greater than 2.

2.1. Basic operations on matrices

Let n, m, q, s be positive integers and et A, B be two matrices over $\mathbb{K}$ with respective formats $m \times n$ and $q \times s$. The tensor (or Kronecker) product of A by B is an $mq \times ns$ matrix over $\mathbb{K}$ denoted by $A \otimes B$ and defined by

$$A \otimes B = [a_{k\ell}B]_{k,\ell} \quad \text{with} \quad A = [a_{k\ell}]_{k,\ell} \tag{1}$$

The direct sum of A and B is an $(m + q) \times (n + s)$ matrix over $\mathbb{K}$ denoted by $A \oplus B$ and defined by

$$A \oplus B = \begin{bmatrix} A & 0 \\ 0 & B \end{bmatrix}. \tag{2}$$

More generally, for n matrices $A_0, \ldots, A_{n-1}$ over $\mathbb{K}$, the direct sum of $A_0, \ldots, A_{n-1}$ is defined as $\oplus_{i=0}^{n-1} A_i = A_0 \oplus (A_1 \oplus (\cdots \oplus A_{n-1}) \cdots)$. The stride permutation matrix L_m^{mn} permutes an input vector $\mathbf{x}$ of length mn as follows

$$\mathbf{x}[im + j] \mapsto \mathbf{x}[jn + i], \tag{3}$$

for all $0 \leq j < m$, $0 \leq i < n$. If $\mathbf{x}$ is viewed as an $n \times m$ matrix, then L_m^{mn} performs a transposition of this matrix.

2.2. Discrete Fourier Transform

We fix an integer $n \geq 2$ and an n-th primitive root of unity $\omega \in \mathbb{K}$. The n-point Discrete Fourier Transform (DFT) at ω is a linear map from the $\mathbb{K}$-vector space $\mathbb{K}^n$ to itself, defined by $\mathbf{x} \longmapsto \mathrm{DFT}_n \mathbf{x}$ with the n-th DFT matrix

$$\mathrm{DFT}_n = [\omega^{k\ell}]_{0 \leq k, \ell < n}. \tag{4}$$

In particular, the DFT of size 2 corresponds to the butterfly matrix

$$\mathrm{DFT}_2 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}. \tag{5}$$

The well-known Cooley-Tukey Fast Fourier Transform (FFT) [6] in its recursive form is a procedure for computing $\mathrm{DFT}_n\,\mathbf{x}$ based on the following factorization of the matrix DFT_n, for any integers q, s such that $n = qs$ holds:

$$\mathrm{DFT}_{qs} = (\mathrm{DFT}_q \otimes I_s) D_{q,s} (I_q \otimes \mathrm{DFT}_s) L^{qs}_q, \tag{6}$$

where $D_{q,s}$ is the diagonal twiddle matrix defined as

$$D_{q,s} = \bigoplus_{j=0}^{q-1} \mathrm{diag}(1, \omega^j, \ldots, \omega^{j(s-1)}), \tag{7}$$

Formula (8) illustrates Formula (6) with DFT_4:

$$\begin{aligned} \mathrm{DFT}_4 &= (\mathrm{DFT}_2 \otimes I_2) D_{2,2} (I_2 \otimes \mathrm{DFT}_2) L^2_2 \\ &= \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & \omega \end{bmatrix} \begin{bmatrix} 1 & 1 & 0 & 0 \\ 1 & -1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & -1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \\ &= \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & \omega & -1 & -\omega \\ 1 & -1 & 1 & -1 \\ 1 & -\omega & -1 & \omega \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & \omega^1 & \omega^2 & \omega^3 \\ 1 & \omega^2 & \omega^4 & \omega^6 \\ 1 & \omega^3 & \omega^6 & \omega^9 \end{bmatrix}. \end{aligned} \tag{8}$$

Assume that n is a power of 2, say $n = 2^k$. Formula (6) can be unrolled so as to reduce DFT_n to DFT_2 (or a base case DFT_m, where m divides n) together with the appropriate diagonal twiddle matrices and stride permutation matrices. This unrolling can be done in various ways. Before presenting one of them, we introduce a notation. For integers $i, j, h \geq 1$, we define

$$\Delta(i, j, h) = (I_i \otimes \mathrm{DFT}_j \otimes I_h) \tag{9}$$

which is a square matrix of size ijh. For $m = 2^\ell$ with $1 \leq \ell < k$, the following formula holds:

$$\mathrm{DFT}_{2^k} = \left(\prod_{i=1}^{k-\ell} \Delta\left(2^{i-1}, 2, 2^{k-i}\right) \left(I_{2^{i-1}} \otimes D_{2,2^{k-i}}\right) \right) \Delta\left(2^{k-\ell}, m, 1\right) \left(\prod_{i=k-\ell}^{1} (I_{2^{i-1}} \otimes L^{2^{k-i+1}}_2) \right). \tag{10}$$

Therefore, Formula (10) reduces the computation of DFT_{2^k} to composing DFT_2, DFT_{2^ℓ}, diagonal twiddle endomorphisms and stride permutations. This is the basis of the implementation presented in Section 3. Another recursive factorization of the matrix DFT_{2^k} is

$$\mathrm{DFT}_{2^k} = (\mathrm{DFT}_2 \otimes I_{2^{k-1}}) D_{2,2^{k-1}} L^{2^k}_2 (\mathrm{DFT}_{2^{k-1}} \otimes I_2), \tag{11}$$

from which one can derive the Stockham FFT [26] as follows

$$\mathrm{DFT}_{2^k} = \prod_{i=0}^{k-1} (\mathrm{DFT}_2 \otimes I_{2^{k-1}}) (D_{2,2^{k-i-1}} \otimes I_{2^i}) (L^{2^{k-i}}_2 \otimes I_{2^i}). \tag{12}$$

This is the basis of the implementation presented in Section 4.

2.3. FFTs over finite fields

As mentioned in the introduction, for FFT computations, the case where $\mathbb{K}$ is a finite field offers specific challenges in comparison to the case where $\mathbb{K}$ is the field $\mathbb{C}$ of complex numbers. This explains why FFT techniques used for numerical computation do not adapt straightforwardly to symbolic computation.

On the algebraic side, the most obvious difference is that the prime field $\mathbb{Z}/p\mathbb{Z}$ admits an n-th primitive root of unity if and only if n divides $p-1$. In addition, radix 2 FFTs are often preferred to others in symbolic computation, since many algorithms, such as those for polynomial factorization, require the prime p to be small, say $p = 3, 5, 7, \ldots$. Since the radix must be invertible in $\mathbb{Z}/p\mathbb{Z}$, this essentially imposes the restriction to radix 2 FFTs. See [11] for details on these algebraic considerations.

On the implementation side, multiplying two elements a, b of $\mathbb{Z}/p\mathbb{Z}$ is obviously a key routine. Unlike the case of single and double precision floating point arithmetic, the operation $(a, b, p) \longmapsto (ab) \mod p$, for $a, b, p \in \mathbb{Z}$, is not provided directly by hardware. This operation is thus an efficiency-critical low-level software routine that the programmer has to take care of. When p is a machine word size prime, which is the assumption in this paper, two techniques are popular in the symbolic computation community.

The first one takes advantage of hardware floating point arithmetic, see [7]. We call `double_mul_mod` our implementation of this technique, for which our CUDA code is shown below. The fourth argument `pinv` is the inverse of `p` which is precomputed in floating point.

```
__device__ int double_mul_mod(int a, int b, int p, double pinv) {
   int q  = (int) ((((double) a) * ((double) b)) * pinv);
   int res = a * b - q * p;
   return (res < 0) ? (-res) : res;
}
```

In our implementation, double precision floating point numbers are encoded on 64 bits and make this technique work correctly for primes p up to 30 bits.

The second technique, called the Montgomery reduction [19], relies only on hardware integer arithmetic. We summarize this elegant trick. Consider a positive integer $R \geq p$ such that $\gcd(R, p) = 1$. Hence there exists integers R^{-1}, p' such that we have:

$$RR^{-1} - pp' = 1 \text{ and } 0 < p' < R. \tag{13}$$

Consider an integer x, satisfying $0 \leq x < p^2$, and for which we want to compute $x/R \mod p$. Let c and d (resp. e and f) be the quotient and remainder in the Euclidean division of x by R (resp. dp' and R). Then, it is easy to prove that there exists an integer q such that $x + fp = qR$ holds, that is, satisfying $q \equiv x/R \mod p$. If $p > 2$ then R can be chosen to be a power of 2. Therefore, with this choice, computing $x/R \mod p$ amounts to 2 multiplications, 2 additions and 3 shifts. Now, in order to compute products in $\mathbb{Z}/p\mathbb{Z}$, one "represents" any residue class $a \mod p$ by $aR \mod p$. Then, applying the previous trick to $x = (aR)(bR) \mod p$ one obtains efficiently $(ab)/R \mod p$, that is, the representative of $(ab) \mod p$. An improved version of this trick was proposed in [15].

3. Implementation of the Cooley-Tukey FFT

We stick to the notations and hypotheses introduced in Section 2.2. Our purpose is to describe our CUDA implementation of Formula (10). The idea behind this formula is that the base case DFT_m can be implemented efficiently, for m small enough, typically $m = 16$. This formula can be interpreted as the composition of three computational steps:

$\mathbf{S_1}$: $\mathbf{x} \longmapsto \prod_{i=k-\ell}^{1} (I_{2^{i-1}} \otimes L_2^{2^{k-i+1}})\mathbf{x}$,

$\mathbf{S_2}$: $\mathbf{x} \longmapsto \Delta\left(2^{k-\ell}, m, 1\right)\mathbf{x}$,

$\mathbf{S_3}$: $\mathbf{x} \longmapsto \prod_{i=1}^{k-\ell} \Delta\left(2^{i-1}, 2, 2^{k-i}\right)\left(I_{2^{i-1}} \otimes D_{2,2^{k-i}}\right)\mathbf{x}$.

According to the definition $\Delta(2^{k-\ell}, m, 1) = I_{2^{k-\ell}} \otimes \mathrm{DFT}_m$, the step $\mathbf{S}_2$ essentially reduces to execute a sequence of base DFT_m, each of which operates on a subarray of $\mathbf{x}$, independently. Therefore, we focus hereafter on $\mathbf{S}_1$ and $\mathbf{S}_3$. Note that in steps $\mathbf{S}_1$ and $\mathbf{S}_3$ we need to double-buffer the array to avoid synchronizations among different CUDA thread blocks, see [3] for details. That is, at the same time, we have two arrays X and Y of length n, one of which is the input and the other is the output, and they switch their role after a kernel call on the input.

3.1. Implementation of step $\mathbf{S}_1$

Step $\mathbf{S}_1$ consists of a sequence of calls to the following GPU kernel, with s ranging from 1 to $\frac{n}{2m}$. Its specification is

```
/**
 * Compute Y = (I_s x L_2^{n/s})X
 *
 * @X, input array of length n
 * @Y, output array of length n
 */
void list_transpose_kernel(int *Y, int *X, int n, int s);
```

Performing the product of $I_s \otimes L_2^{n/s}$ by a vector $\mathbf{x}$ of length n is equivalent to

(i) dividing x evenly into s subarrays,

(ii) regarding each subarray as a $\frac{n}{2s} \times 2$ matrix and transposing it.

Therefore, step $\mathbf{S}_1$ essentially consists of s matrix transpositions of size $\frac{n}{2s} \times 2$. Following the spirit of [24] for matrix transposition, we realized an efficient subroutine to transpose a list of matrices. Note that we could not directly adapt their code since each matrix has only two columns. Without padding the input data with zeros, our implementation is still able to utilize the shared memory of CUDA devices effectively. For simplicity, we present our implementation with the following example.

Example 3.1 Let M be a 16×2 matrix. We set the thread block size to 16×2 with indices (i, j) for $0 \le i < 16$ and $j = 0, 1$. Then we first read M into an array M_s of size 32 residing in the shared memory space as follows

```
int i = threadIdx.y * 16 + threadIdx.x;
M_s[i] = M[i];
```

That is, the above segment of code transforms M into the shared array M_s via two coalesced reads, without changing the data layout. Still, we look at the shared array M_s as a 16×2 matrix, then we achieve the transposition by writing the data back to the global memory column-wise as follows

```
int i = threadIdx.y * 16 + threadIdx.x;
M[i] = M_s[threadIdx.x * 2 + threadIdx.y];
```

The first 16 threads (a half warp) $\{(i, 0) \mid 0 \le i < 16\}$ read in $M_s[0], M_s[2], \ldots, M_s[30]$, and write to $M[0], M[1], \ldots, M[15]$. On the other hand, the second half warp of threads $\{(i, 1) \mid 0 \le i < 16\}$ read in $M_s[1], M_s[3], \ldots, M_s[31]$ and write to $M[16], M[17], \ldots, M[31]$. Again all the writes to global memory are coalesced.

The above example can be generalized to transpose a list of $m \times 2$ matrices with only coalesced reads and writes for any $m \ge 16$, which satisfies the specification of the kernel *list_transpose_kernel*.

3.2. Implementation of step $\mathbf{S}_3$

We are going to map the formula

$$(I_{2^{i-1}} \otimes \mathrm{DFT}_2 \otimes I_{2^{k-i}})(I_{2^{i-1}} \otimes D_{2,2^{k-i}}), \quad 1 \leq i \leq k-\ell \tag{14}$$

to a GPU kernel function. The following relation is easy to prove:

$$(I_{2^{i-1}} \otimes \mathrm{DFT}_2 \otimes I_{2^{k-i}})(I_{2^{i-1}} \otimes D_{2,2^{k-i}}) = I_{2^{i-1}} \otimes \big((\mathrm{DFT}_2 \otimes I_{2^{k-i}})D_{2,2^{k-i}}\big) \tag{15}$$

Hence step $\mathbf{S}_3$ consists of a sequence of calls to the following GPU kernel, with q ranging from $\frac{n}{2}$ to m. It specification is

```
/**
 * Compute Y = (I_{n/2q} x (DFT_2 x I_q) D_{2, q}) X
 *
 * @X, input array of length n
 * @Y, output array of length n
 */
void list_butterfly_kernel(int *Y, int *X, int n, int q);
```

We notice that $(\mathrm{DFT}_2 \otimes I_q)D_{2,q}$ is, in fact, the *classical butterfly* operation, which can be realized as,

```
for (i = 0; i < q; ++i) {
    Y[i]   = X[i] + X[q+i] * W[i];
    Y[q+i] = X[i] - X[q+i] * W[i];
}
```

with $W[i] = \theta^i$ and θ is a $(2q)$-th primitive root of unity. The formula $(\mathrm{DFT}_2 \otimes I_q)D_{2,q}$ will be applied to a segment of data of length $2q$. Hence, with $n/2$ threads, one can realize *list_butterfly_kernel* which implements the formula $I_{2^{i-1}} \otimes \big((\mathrm{DFT}_2 \otimes I_{2^{k-i}})D_{2,2^{k-i}}\big)$ for each i.

4. Implementation of the Stockham FFT

Recall the Stockham FFT formula (12):

$$\mathrm{DFT}_{2^k} = \prod_{i=0}^{k-1} (\mathrm{DFT}_2 \otimes I_{2^{k-1}})(D_{2,2^{k-i-1}} \otimes I_{2^i})(L_2^{2^{k-i}} \otimes I_{2^i}).$$

For each fixed $0 \leq i < k$, it consists of three computational steps:

$\mathbf{A}_1$**:** $\mathbf{x} \longmapsto (L_2^{2^{k-i}} \otimes I_{2^i})\mathbf{x}$,
$\mathbf{A}_2$**:** $\mathbf{x} \longmapsto (D_{2,2^{k-i-1}} \otimes I_{2^i})\mathbf{x}$,
$\mathbf{A}_3$**:** $\mathbf{x} \longmapsto (\mathrm{DFT}_2 \otimes I_{2^{k-1}})\mathbf{x}$.

Similar to the implementation of the Cooley-Tukey FFT, we double-buffer these steps.

4.1. Implementation of step $\mathbf{A}_1$

We describe how to map the formula $L_2^{n/s} \otimes I_s$ to a GPU kernel, where n is the FFT size and s is the stride size. Let M be an $(n/s-1) \times 2s$ matrix stored in the row-major layout. The effect of this stride permutation on M is to perform the following reordering:

$$M = \begin{bmatrix} \mathbf{S}_0 & \mathbf{S}_1 \\ \mathbf{S}_2 & \mathbf{S}_3 \\ \mathbf{S}_4 & \mathbf{S}_5 \\ \vdots & \vdots \\ \mathbf{S}_{(n/s-2)} & \mathbf{S}_{(n/s-1)} \end{bmatrix} \Longrightarrow T = \begin{bmatrix} \mathbf{S}_0 & \mathbf{S}_2 & \mathbf{S}_4 & \cdots & \mathbf{S}_{(n/s-2)} \\ \mathbf{S}_1 & \mathbf{S}_3 & \mathbf{S}_5 & \cdots & \mathbf{S}_{(n/s-1)} \end{bmatrix}$$

where $\mathbf{S}_i$ denotes the elements of M with indices $is \cdots (is+s-1)$. When we regard M as an $(n/s-1)\times 2$ matrix (each stride $\mathbf{S}_i$ is a single element of M), the output matrix T, of size $2 \times (n/s-1)$, is the matrix transposition of M.

We describe how to realize this stride permutation in a CUDA kernel. Let τ be the number of threads in a thread block, typically $\tau = 128$. To use the shared memory space efficiently, τ should be a multiple of 16. Under the above setting, the number of threads blocks required is given by $\lambda = \frac{n}{\tau}$. We need to distinguish the following two cases

(1) $s \geq \tau$, that is, $\delta = \frac{s}{\tau}$ blocks are needed to move a stride of length s,

(2) $s < \tau$, that is, a thread block moves $\delta = \frac{\tau}{s}$ strides of data.

The reason to have such a case discussion is that the relation between τ and s determines the behavior of each thread block, specified in detail as follows.

Case $s \geq \tau$: Given a thread block index $0 \leq i < \lambda$, we define

$$i_q = \mathrm{quo}(i, \delta) \quad \text{and} \quad i_r = \mathrm{rem}(i, \delta).$$

Here i_q determines the stride index which thread block i is working on and i_r determines the offset inside this stride. If i_q is a multiple of 2 then $\mathbf{S}_{i_q}$ appears in the first row of the output matrix, otherwise it appears in the second row. Hence the offset for the thread block i is given by the following formula:

$$\mathrm{rem}(i_q, 2) * \frac{n}{2} + \mathrm{quo}(i_q, 2) * s + i_r * \tau. \tag{16}$$

In this case, each thread block does a direct copy, that is, no data shuffle is needed.

Case $s < \tau$: Given a thread block index $0 \leq i < \lambda$, there are two offsets:

$$i * \mathrm{quo}(\tau, 2) \quad \text{and} \quad i * \mathrm{quo}(\tau, 2) + \frac{n}{2}. \tag{17}$$

In this case, the behavior inside each thread block is not just a direct copy. Since there are δ strides inside each thread block, those strides with even index use the first offset to move data, while the other strides use the second offset to move data (this may be viewed as an in-block data shuffle).

4.2. Implementation of steps $\mathbf{A}_2$ and $\mathbf{A}_3$

According to its definition, $D_{2,2^{k-i-1}}$ is a diagonal matrix of size 2^{k-i} and thus $D_{2,2^{k-i-1}} \otimes I_{2^i}$ is again a diagonal matrix of size n, with each diagonal element repeated 2^i times. Hence step $\mathbf{A}_2$ simply scales $\mathbf{x}$ with powers of the primitive root of unity ω. On the other hand, step $\mathbf{A}_3$ is a list of basic butterflies with stride size $n/2$. This step is accesses data in a very uniform manner. In the following section, we discuss the performance implications of steps $\mathbf{A}_2$ and $\mathbf{A}_3$.

5. Experimentation

We have realized in CUDA 2.2 both Cooley-Tukey FFT and Stockham FFT, and conducted a series of benchmarks using a Geforce GTX 285 graphics card on a desktop with the processor Intel Core 2 Quad CPU Q9400 @ 2.66GHz and 6 GB main memory. This graphics card has the compute capability 1.3, consists of 30 multiprocessors, each of which has 8 cores for integer and single-precision floating-point arithmetic operations. However, each multiprocessor has only 1 double-precision floating-point unit. This is an important characteristic for our `double_mul_mod` routine implementing the map $(a, b, p) \mapsto (ab) \mod p$, as described Section 2.3. We note that for the most recent NVIDIA graphics

cards having the compute capability 2.x, the ability to perform double-precision floating-point operations has been greatly enhanced.

The experimentation is described in the following three subsections. Section 5.1 is dedicated to modular multiplication, and more precisely, to a comparative implementation of the map $(a, b, p) \mapsto (ab) \mod p$ on both CPU and GPU. Section 5.2 presents the results for our GPU implementation of the FFT formulas of Cooley-Tukey and Stockham, as described in Sections 3 and 4. Section 5.3 compares the performance of FFT-based univariate polynomial multiplication codes for CPU and GPU.

5.1. Modular multiplication

Figure 1 and Figure 2 are experimental results for modular multiplication on CPU and GPU respectively. In both cases, each slot of an input array of length $n = 2^k$ consisting of machine word size integers is multiplied by a given machine word size integer ω. This type of calculation is typical for FFT algorithms. For both CPU and GPU we compare our implementations of the Montgomery reduction and `double_mul_mod`. For the GPU kernel, we could choose to have a single thread or multiple threads. In our experimentation, we use the latter and in this case each multiprocessor can only process a double-precision floating-point operation at a time which downgrades the performance.

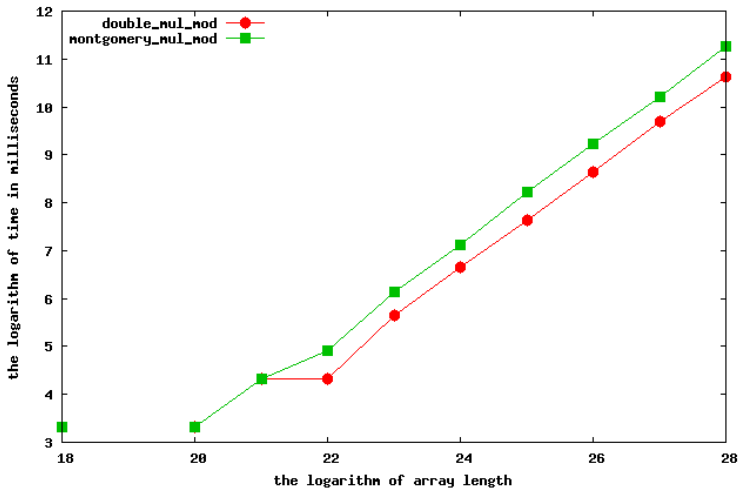

Figure 1: Modular multiplication on CPU. When we increase the number of modular multiplications, the one relying on double-precision floating point computations outperforms the one relying on the Montgomery reduction. Both the array length and the time are scaled by the base 2 logarithm.

Figure 1 shows that `double_mul_mod` is about 1.5 faster than the method based on the Montgomery reduction, when running serial C code on the CPU. On the GPU, Figure 2 shows that `double_mul_mod` is still slightly better than the method based on the Montgomery reduction, which is a surprise to us.

5.2. Cooley-Tukey and Stockham FFT over finite fields on a GPU

It is challenging to figure out what are the best implementation techniques for each of the two formulas. During our experiments, we realized that the pre-computation of the powers $1, \omega, \omega^2, \ldots, \omega^{n/2-1}$ is a necessity, for an n-point FFT. This step is rather time-consuming if we implement it naively, that is, first compute those powers in the host sequentially, and then transfer them to the GPU device. Fortunately, this preprocessing is a special form of the exclusive prefix sum and the pre-computation can be achieved by a

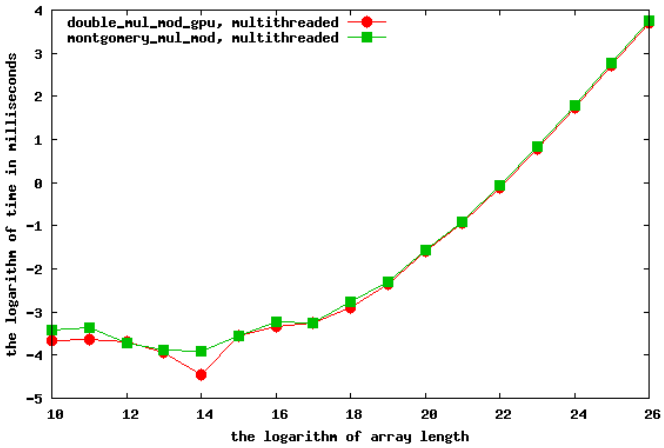

Figure 2: Modular multiplication on GPU. We do the same test as in Figure 1 on GPU, and run the kernels with massive threads. Both the array length and the time are scaled by the base 2 logarithm.

sequence of GPU kernel calls to a subroutine `double_expand`, which takes an array $\{1, \omega, \ldots, \omega^{s-1}\}$ of length s as input and returns $\{\omega^s, \ldots, \omega^{2s-1}\}$ by multiplying each element of the input array with ω^s.

We have noticed that step $\mathbf{S}_3$ of the Cooley-Tukey FFT (as described in Section 3.2) was accessing the powers of ω by performing larger and larger jumps. For example, while the following formula $(\mathrm{DFT}_2 \otimes I_{2^{k-i}})(I_{2^{i-1}} \otimes D_{2,2^{k-i}})$ operates on a subarray of length 2^{k-i+1}, the powers $\{1, \omega^{2^i}, (\omega^{2^i})^2, \ldots, (\omega^{2^i})^{2^{k-i}-1}\}$ get accessed. We call *jumped powers at level* i these latter powers. Therefore, we considered pre-computing not only the powers $\{1, \omega, \ldots, \omega^{n/2-1}\}$ but also all jumped powers at level i for each i.

To visualize the performance of our implementation, we use the NVIDIA's visual profiler `cudaprof` to analyze CUDA kernel calls. It is very helpful to find out the bottlenecks of an implementation. For instance, Figure 3 shows the kernel statistics where the pre-computation of jumped powers has been done on the host. In this figure, the x-axis shows the CUDA kernel call indices in chronological order and the y-axis is proportional to the GPU time for each kernel. We notice that it spends a fairly large amount of time to move the extra $n/2$ pre-computed powers to the GPU.

If the jumped powers were not computed in advance, the accesses to those powers harm the performance heavily as shown by Figure 4, since those memory accesses to the global memory are non-coalesced, in the step $\mathbf{S}_3$ of the Cooley-Tukey FFT. Up to our knowledge, it is hard to achieve coalesced accesses without pre-computing jumped powers while implementing the Cooley-Tukey FFT.

However, the Stockham FFT avoids such a problem. Indeed, all the accesses to a power of ω are packed together, resulting in a broadcasting inside a thread block. Figure 5 shows the kernel statistics of the Stockham FFT of size 2^{26}, which is our best GPU FFT implementation. Our Stockham FFT implementation pre-computes all powers in an extremely fast manner without computing jumped powers. The first and last kernel are for the input and output data transfer, and all the other kernels are run efficiently (the occupancy is 1 for each kernel).

For completeness, in Figure 6 we compare our two GPU implementations for FFT against our C code from `modpn` [14]. This latter library is shipped with the computer algebra system MAPLE and is considered as a reference code for FFT computations over finite fields. Without considering the time spent in host-device data transfer, the speedup we achieve is about 37 for the FFT size 2^{26} (this speedup is about 21 if the data movement time is counted). As shown by Figure 7, our Stockham FFT code is about 2 times faster than our Cooley-Tukey FFT code, mainly due to the jumped powers pre-computation.

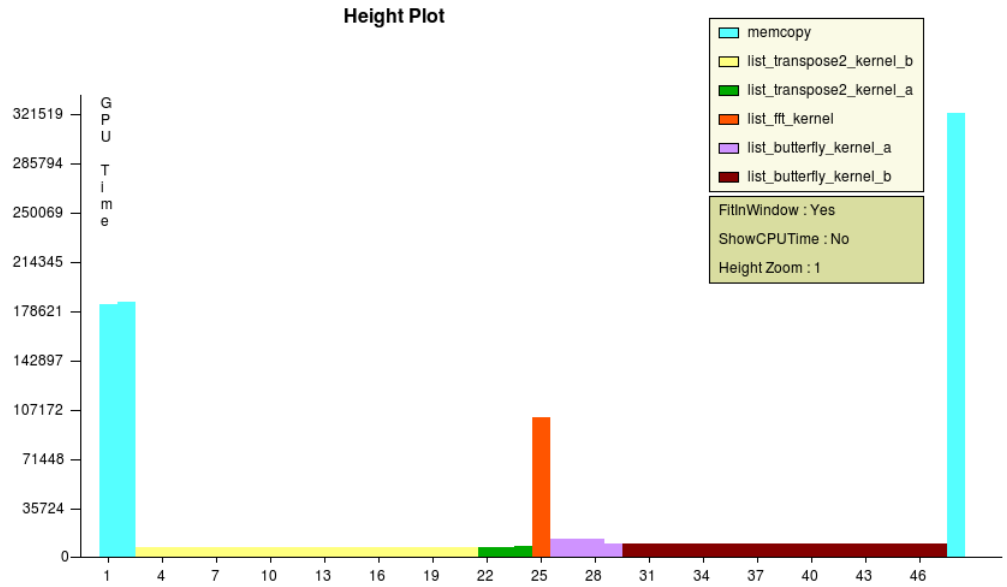

Figure 3: Kernel statistics for the Cooley-Tukey FFT with pre-computed jumped powers. The second kernel moves the extra $n/2$ powers of the primitive root of unity, which affects the overall performance. Note that those method names _a or _b as a suffix, since we implement the same algorithm for handling input data in different ranges.

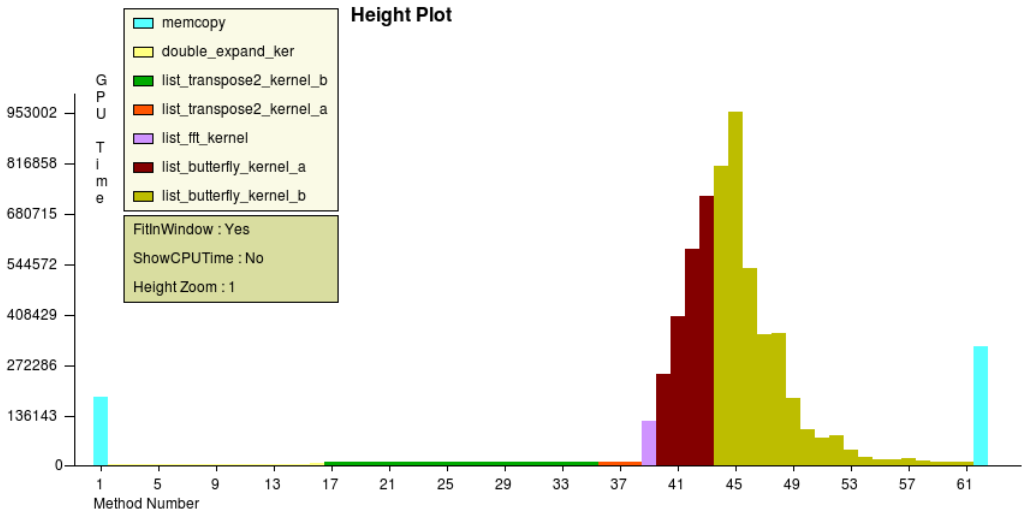

Figure 4: Kernel statistics for the Cooley-Tukey FFT on GPU without pre-computed jumped powers. Time to call the kernel *list_butterfly_kernel* increases significantly, which greatly downgrades the overall performance.

5.3. Univariate polynomial multiplication over finite fields

As a direct application of fast Fourier transforms, we have implemented FFT based univariate polynomial multiplications over finite fields. Figure 8 compares the modpn FFT based polynomial multiplication against our GPU Stockham FFT-based one. The input two polynomials are randomly generated with the same given degree. When the degree is relatively large, the speedup we achieved is about 21 - 37, comparing to the modpn polynomial multiplication.

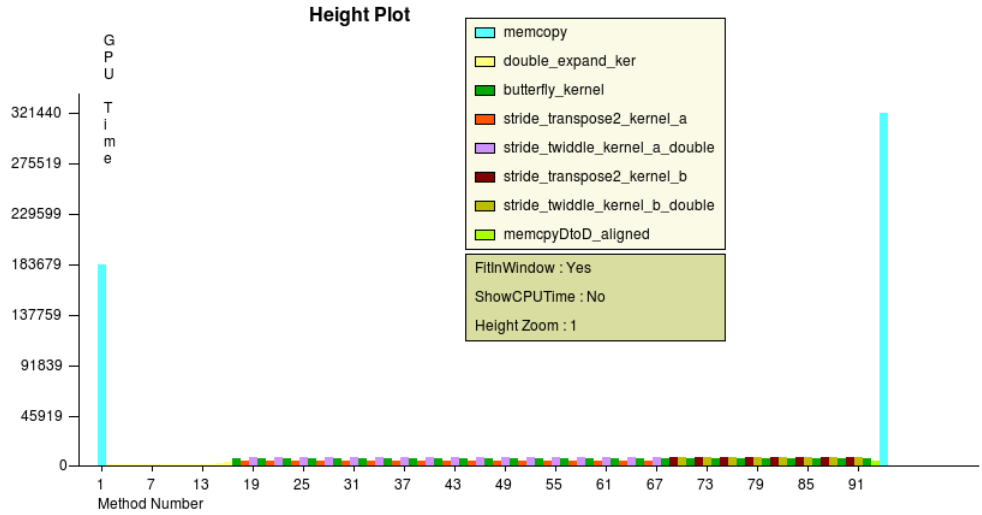

Figure 5: Kernel statistics for the Stockham FFT on GPU. This is our best implementation for 1D FFT. The pre-computation is achieved by the kernel call *double_expand_ker*. The steps $\mathbf{A}_1$, $\mathbf{A}_2$ and $\mathbf{A}_3$ are realized by *stride_transpose2_kernel*, *stride_twiddle_kernel* and *butterfly_kernel*, respectively. All of them are running very efficiently.

	CT FFT	CT FFT + transfer	Stockham FFT	Stockham FFT + transfer	modpn FFT
12	1	1	2	2	1
13	2	2	2	3	1
14	1	2	2	3	3
15	2	2	3	3	4
16	3	3	3	4	10
17	4	5	3	5	16
18	6	9	4	7	37
19	11	15	6	10	71
20	22	28	9	16	174
21	44	56	16	28	470
22	83	105	29	52	997
23	165	210	56	101	2070
24	330	418	113	201	4194
25	667	842	230	405	8611
26	1338	1686	473	822	17617

Figure 6: Timing of FFT codes on CPU and GPU in milliseconds. The first column is the logarithm of FFT size in base 2. The second and the fourth column show the timing of our Cooley-Tukey and Stockham FFT implementations, without counting the data transfer between GPU and CPU, respectively. The third and the fifth column shows these FFT implementations with the data transfer. The last column shows the `modpn` FFT timing.

6. Conclusion and future work

We have presented in detail various issues in implementing efficient fast Fourier transforms over finite fields on the GPU. Our experimental results show that the Stockham formula is well-suited for massively-threaded architectures. In particular, it avoids pre-computing extra powers of primitive roots of unity in a natural way, without downgrading the performance. Our implementation exhibits a significant performance improvement over a reference C implementation. For multiplying two dense univariate polynomials, we have achieved about 30x speedup with respect to the best code available to us. As future work, we would like to implement multidimensional FFTs, and to revisit various modular algorithms in symbolic computation, like evaluation/interpolation based subresultant chain construction

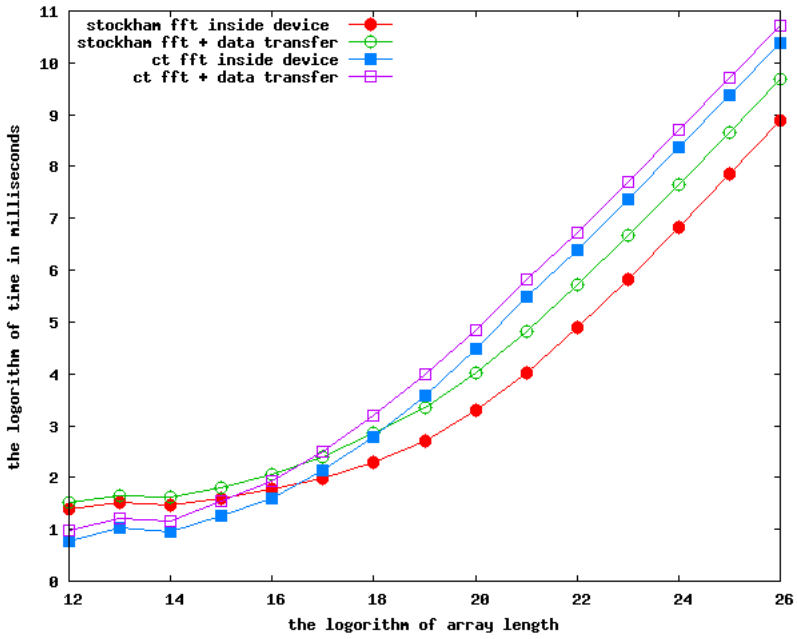

Figure 7: We visualize the comparison among GPU FFT implementations according to the timing from Figure 6. Both the FFT size and the time are scaled by the base 2 logarithm.

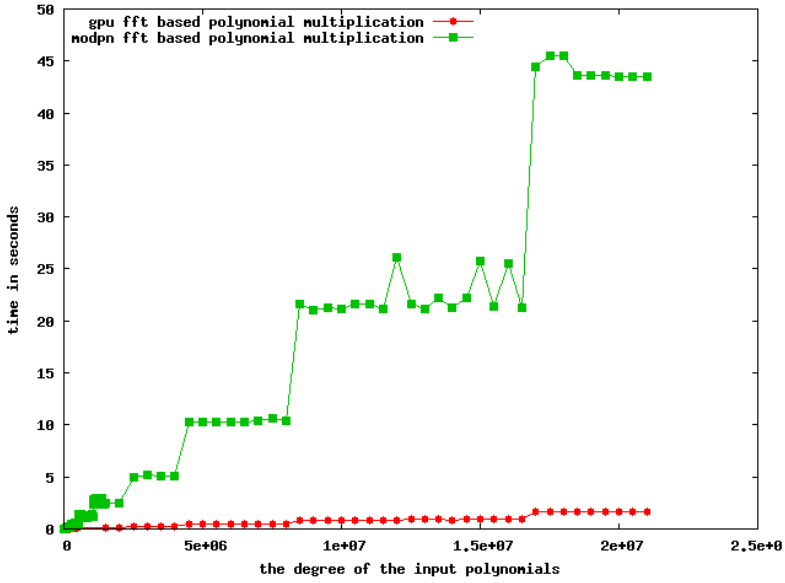

Figure 8: FFT-based dense polynomial multiplication on GPU and CPU. The data transfer has been counted for the GPU code.

of polynomials.

[1] *The MAGMA Computational Algebra System*. http://magma.maths.usyd.edu.au/magma/.
[2] *Nvidia CUDA*. http://www.nvidia.com/object/cuda_home_new.html.
[3] *Nvidia CUDA programming guide 2.3*. 2009.
[4] D. Cantor and E. Kaltofen. On fast multiplication of polynomials over arbitrary algebras. *Acta Informatica*, 28:693–701, 1991.

[5] S. Chellappa, F. Franchetti, and M. Püschel. How to write fast numerical code. In *Summer School on Generative and Transformational Techniques in Software Engineering*, volume 5235 of *LNCS*, pages 196–259, 2008.

[6] J. Cooley and J. Tukey. An algorithm for the machine calculation of complex Fourier series. *Math. Comp.*, 19:297–301, 1965.

[7] J.-G. Dumas, T. Gautier, and C. Pernet. Finite field linear algebra subroutines. In *ISSAC '02: Proceedings of the 2002 international symposium on symbolic and algebraic computation*, pages 63–74, New York, NY, USA, 2002. ACM.

[8] A. Filatei, X. Li, M. Moreno Maza, and E. Schost. Implementation techniques for fast polynomial arithmetic in a high-level programming environment. In *Proc. ISSAC'06*, pages 93–100, New York, NY, USA, 2006. ACM Press.

[9] F. Franchetti and M. Püschel. *Encyclopedia of Parallel Computing*. Springer, 2011.

[10] F. Franchetti, M. Püschel, Y. Voronenko, S. Chellappa, and J. M. F. Moura. Discrete Fourier transform on multicore. *IEEE Signal Processing Magazine, special issue on "Signal Processing on Platforms with Multiple Cores"*, 26(6):90–102, 2009.

[11] J. Gathen and J. Gerhard. *Modern Computer Algebra*. Cambridge University Press, 1999.

[12] N. K. Govindaraju, B. Lloyd, Y. Dotsenko, B. Smith, and J. Manferdelli. High performance discrete fourier transforms on graphics processors. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–12, Piscataway, NJ, USA, 2008. IEEE Press.

[13] GPUmat. *GPU toolbox for MATLAB*.

[14] X. Li, M. Moreno Maza, R. Rasheed, and É. Schost. The modpn library: Bringing fast polynomial arithmetic into maple. In *MICA'08*, pages 73–80, 2008.

[15] X. Li, M. Moreno Maza, and E. Schost. Fast arithmetic for triangular sets: From theory to practice. *J. Symb. Comp.*, 44(7):891–907, 2008.

[16] P. LinBox. *Exact computational linear algebra*. http://www.linalg.org/.

[17] MathWorks. *Parallel Computing Toolbox*.

[18] M. Monagan and R. Pearce. Parallel sparse polynomial multiplication using heaps. In *ISSAC '09: Proceedings of the international symposium on Symbolic and algebraic computation*, pages 263–270, New York, NY, USA, 2009. ACM.

[19] P. L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44(170):519–521, 1985.

[20] K. Moreland and E. Angel. The FFT on a gpu. In *HWWS '03: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 112–119, Aire-la-Ville, Switzerland, 2003. Eurographics Association.

[21] M. Moreno Maza and Y. Xie. Balanced dense polynomial multiplication on multicores. In *Proc. of PDCAT'09*. IEEE Computer Society, 2009.

[22] M. Moreno Maza and Y. Xie. FFT-based dense polynomial arithmetic on multi-cores. In D. Mewhort, editor, *Proc. HPCS 2009*, volume 5976 of *LNCS*, Heidelberg, 2010. Springer-Verlag Berlin.

[23] 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. SPIRAL: Code generation for DSP transforms. *Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation"*, 93(2):232– 275, 2005.

[24] G. Ruetsch and P. Micikevicius. Optimizing matrix transpose in cuda. 2009.

[25] V. Shoup. *NTL: The Number Theory Library*.

[26] T. G. Stockham, Jr. High-speed convolution and correlation. In *AFIPS '66 (Spring): Proceedings of the April 26-28, 1966, Spring joint computer conference*, pages 229–233, New York, NY, USA, 1966. ACM.

[27] P. N. Swarztrauber. FFT algorithms for vector computers. *Parallel Comput.*, 1(1):45–63, 1984.

[28] C. Van Loan. *Computational frameworks for the fast Fourier transform*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1992.