Theoretical Computer Science 410 (2009) 2649–2658

Contents lists available at ScienceDirect

Theoretical Computer Science

journal homepage: www.elsevier.com/locate/tcs

A cache-friendly truncated FFT

David Harvey *

Courant Institute of Mathematical Sciences, New York University, 251 Mercer Street, New York, NY 10012-1185, United States

ARTICLE INFO

Article history:
Received 16 November 2008
Received in revised form 5 March 2009
Accepted 15 March 2009
Communicated by V. Pan

Keywords:
Polynomial multiplication
Fast Fourier transform

ABSTRACT

We describe a cache-friendly version of van der Hoeven's truncated FFT and inverse truncated FFT, focusing on the case of 'large' coefficients, such as those arising in the Schönhage–Strassen algorithm for multiplication in $\mathbf{Z}[x]$. We describe two implementations and examine their performance.

1. Introduction

In typical implementations of the FFT method for dense univariate polynomial multiplication, the input polynomials are zero-padded up to an appropriate power-of-two length, causing a jump in the running time when the lengths cross a power-of-two boundary. Various approaches to reducing these jumps have been proposed – for example, splitting into pieces of distinct power-of-two lengths, or using roots of unity of small odd order – but the most effective and elegant is the recent algorithm of van der Hoeven [16,17]. He introduces a novel TFT (truncated FFT) and ITFT (inverse truncated FFT), achieving relatively smooth performance without sacrificing the simplicity of a power-of-two transform length.

However, the transforms that he describes suffer from suboptimal locality. The transforms follow the divide-and-conquer FFT paradigm, recursively splitting the problem into two half-sized transforms. If the transform length is 2^ℓ, and only 2^k coefficients fit into a given level of cache, then only the deepest k layers of the transform take advantage of that cache; the remaining $\ell - k$ layers do not.

In this paper we address this difficulty, achieving superior temporal locality by reordering the sequence of butterfly operations in van der Hoeven's transforms. Our algorithm does not directly address spatial locality; this is discussed further in Section 6. Our strategy is similar to Bailey's algorithm [1]. Bailey rearranges the data into a $2^{\ell_1} \times 2^{\ell_2}$ matrix, where $\ell_1 + \ell_2 = \ell$, and then rewrites the transform as 2^{ℓ_2} column transforms of length 2^{ℓ_1} followed by 2^{ℓ_1} row transforms of length 2^{ℓ_2}. The divide-and-conquer algorithm may be regarded as the special case where $\ell_1 = 1$ and $\ell_2 = \ell - 1$. However, when $\ell_i \approx \ell/2$, the working set for each row and column is only about $2^{\ell/2}$ coefficients, greatly improving the algorithm's locality. This method can of course be applied recursively, until the working set for each subtransform fits into the lowest level of cache, making efficient use of the entire memory hierarchy.

It is straightforward to adapt this idea to the TFT, obtaining a decomposition of the TFT into TFTs of half the depth (Section 3). The corresponding decomposition of the ITFT is more involved; it becomes necessary to alternate between ITFTs on the rows and columns in a slightly complicated way (Section 4).

In Section 5 we discuss the performance of two implementations. The first is an implementation of the Schönhage–Strassen algorithm [14] for multiplication in $\mathbf{Z}[x]$. The second is an implementation of the Schönhage–Nussbaumer convolution algorithm [11,10] for the case of $(\mathbf{Z}/m\mathbf{Z})[x]$ where m is an odd word-sized modulus. In both cases the individual

* Tel.: +1 212 998 3210; fax: +1 212 995 4121.
E-mail address: dmharvey@cims.nyu.edu.

0304-3975/$ – see front matter © 2009 Elsevier B.V. All rights reserved.
doi:10.1016/j.tcs.2009.03.014

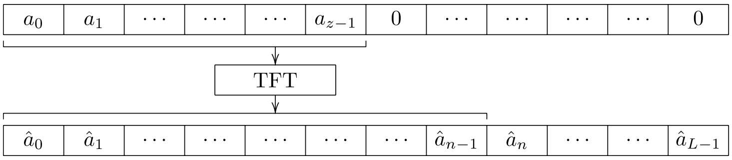

Fig. 1. The TFT.

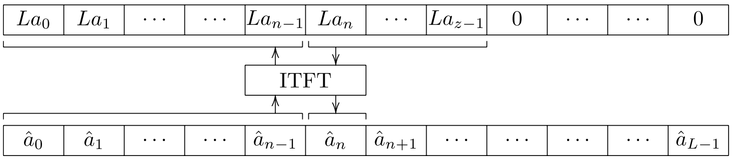

Fig. 2. The ITFT.

Fourier coefficients occupy relatively large blocks of memory, so spatial locality is largely automatic. A natural question is whether the new algorithms are suitable for the more conventional case of 'small' coefficients, such as double-precision real or complex coefficients. We offer some speculation in Section 6, although we have not attempted an implementation.

2. Notation and setup

Let R be a commutative ring in which 2 is invertible. We assume that R contains a principal Mth root of unity ω, where $M = 2^m$ for some integer $m \geq 1$; this means that $\omega^M = 1$ and moreover that $\sum_{i=0}^{M-1} \omega^{ij} = 0$ for all $0 < j < M$. We have in mind examples like $R = \mathbf{Z}/(2^{M/2} + 1)\mathbf{Z}$ and $\omega = 2$, which appears in the Schönhage–Strassen algorithm for multiplication in $\mathbf{Z}[x]$.

If $L \mid M$, we denote by ω_L the principal Lth root of unity $\omega^{M/L}$; we then have the compatibility relation $(\omega_{L'})^{L'/L} = \omega_L$ for any $L \mid L' \mid M$.

Now suppose that $L \mid M$, $L = 2^\ell$, and let $\zeta \in R^\times$. Let $(a_0, \dots, a_{L-1}) \in R^L$. The (weighted) discrete Fourier transform (DFT) is defined by

$$\hat{a}_j = \zeta^{j'} \sum_{i=0}^{L-1} \omega_L^{ij'} a_i, \quad 0 \leq j < L, \tag{1}$$

where j' denotes the length-ℓ bit-reversal of j.

We define the *truncated Fourier transform* (TFT) as follows. Let $1 \leq z \leq L$ and $1 \leq n \leq L$, and suppose that $a_z = \cdots = a_{L-1} = 0$. Then

$$\mathrm{TFT}(L, \zeta, z, n; (a_0, \dots, a_{z-1})) := (\hat{a}_0, \dots, \hat{a}_{n-1}).$$

In other words, the TFT computes a prescribed initial segment of the transform, assuming that some prescribed final segment of the untransformed data is zero (see Fig. 1).

The definition of the *inverse truncated Fourier transform* (ITFT) is more involved. Let $f \in \{0, 1\}$. Suppose that $1 \leq z \leq L$ and $1 \leq n + f \leq L$, and moreover that $z \geq n$. Suppose as before that $a_z = \cdots = a_{L-1} = 0$. Then

$$\mathrm{ITFT}(L, \zeta, z, n, f; (\hat{a}_0, \dots, \hat{a}_{n-1}, La_n, \dots, La_{z-1})) := \begin{cases} (La_0, \dots, La_{n-1}) & f = 0, \\ (La_0, \dots, La_{n-1}, \hat{a}_n) & f = 1. \end{cases}$$

In other words, the ITFT takes as input an initial segment of the transformed data together with the *complementary* final segment of the untransformed data (some components of which are known to be zero), and returns the initial segment of the untransformed data, and optionally (if $f = 1$) the next transformed coordinate (see Fig. 2). When $z = n = L$, $f = 0$ and $\zeta = 1$, the TFT and ITFT reduce to the usual DFT and inverse DFT, with inputs in normal order and outputs in bit-reversed order.

It is not obvious *a priori* that the ITFT is well defined, and in particular that the coordinates $\hat{a}_0, \dots, \hat{a}_{n-1}, a_n, \dots, a_{L-1}$ are linearly independent. Van der Hoeven deduced this from the correctness of his algorithm for computing the ITFT; it will follow in the same way from the proof of correctness of our cache-friendly ITFT algorithm in Section 4.

Van der Hoeven allowed the input and output coordinates to come from a wider class of subsets of $\{0, \dots, L-1\}$. In this paper we restrict ourselves to the initial and final segments mentioned above, which suffices for our intended application to univariate polynomial multiplication.

The TFT and ITFT may be used to deduce a polynomial multiplication algorithm in $R[X]$ as follows. Suppose that $g, h \in R[X]$, and let $u = gh$. Let $z_1 = 1 + \deg g$, $z_2 = 1 + \deg h$, $n = z_1 + z_2 - 1$, and assume that $n \leq L$. Let $g_0, \dots, g_{z_1-1}$ be the coefficients of g and $h_0, \dots, h_{z_2-1}$ be the coefficients of h. Compute

$$(\hat{g}_0, \dots, \hat{g}_{n-1}) = \mathrm{TFT}(L, 1, z_1, n; (g_0, \dots, g_{z_1-1})),$$
$$(\hat{h}_0, \dots, \hat{h}_{n-1}) = \mathrm{TFT}(L, 1, z_2, n; (h_0, \dots, h_{z_2-1})),$$

and then compute $\hat{u}_i = \hat{g}_i\hat{h}_i$ in R for $0 \le i < n$. Then $\hat{u}_0, \ldots, \hat{u}_{n-1}$ are the first n Fourier coefficients of u, and moreover $u_n = \cdots = u_{L-1} = 0$ since $n = \deg u + 1$. Therefore we recover u via

$$(Lu_0, \ldots, Lu_{n-1}) = \mathrm{ITFT}(L, 1, n, n, 0; (\hat{u}_0, \ldots, \hat{u}_{n-1})).$$

(This multiplication algorithm has not used the parameters f or ζ in a nontrivial way; these enter the picture when the algorithms are called recursively in Sections 3 and 4.)

The standard FFT algorithms compute the DFT (or inverse DFT) using $\ell L/2$ 'butterfly operations'. In contrast, van der Hoeven showed that the TFT and ITFT may be computed using at most $\ell n/2 + L$ butterfly operations, and we will see that this estimate holds for our cache-friendly TFT and ITFT algorithms as well. Furthermore, in the multiplication algorithm sketched above, only n pointwise multiplications are performed, compared to the L multiplications incurred by the standard FFT method. Therefore, in this simplified algebraic complexity model, the ratio of the running time of the TFT/ITFT-based multiplication algorithm to the running time of the usual FFT multiplication algorithm is $n/L + O(\ell^{-1})$, indicating that the performance is relatively smooth as a function of n.

Algorithms 1 and 2 below (CacheFriendlyTFT and CacheFriendlyITFT) implement the TFT and ITFT in a cache-friendly manner. They operate on an array $x_0, \ldots, x_{L-1}$, where $L = 2^\ell$. In general all L elements of the array, even those elements not containing input or output, are used in intermediate computations.

For the TFT, the first z elements are expected to contain the inputs $a_0, \ldots, a_{z-1}$, and the outputs $\hat{a}_0, \ldots, \hat{a}_{n-1}$ are written in-place to the same array. For the ITFT, the first z elements are expected to contain the inputs $\hat{a}_0, \ldots, \hat{a}_{n-1}, La_n, \ldots, La_{z-1}$, and the outputs $La_0, \ldots, La_{n-1}$ (optionally followed by $\hat{a}_n$ if $f = 1$) are written in-place to the same array.

Both algorithms make use of the following well-known decomposition of (1). Let $L = L_1L_2$ where $L_1 = 2^{\ell_1}$ and $L_2 = 2^{\ell_2}$ (so that $\ell_1 + \ell_2 = \ell$). Write $i = i_2 + L_2 i_1$ where $0 \le i_1 < L_1$ and $0 \le i_2 < L_2$, and similarly for j. Then $j' = j_1' + L_1 j_2'$, where j_1' and j_2' are respectively the length-ℓ_1 and length-ℓ_2 bit-reversals of j_1 and j_2. We obtain

$$\hat{a}_j = \hat{a}_{j_2+L_2j_1} = \zeta^{j_1'+L_1j_2'} \sum_{i_2=0}^{L_2-1}\sum_{i_1=0}^{L_1-1} \omega_L^{(i_2+L_2i_1)(j_1'+L_1j_2')} a_{i_2+L_2i_1} = (\zeta^{L_1})^{j_2'} \sum_{i_2=0}^{L_2-1} \omega_{L_2}^{i_2j_2'} \left((\zeta\omega_L^{i_2})^{j_1'} \sum_{i_1=0}^{L_1-1} \omega_{L_1}^{i_1j_1'} a_{i_2+L_2i_1} \right).$$

Therefore if we put

$$b_k = b_{k_2+L_2k_1} = (\zeta\omega_L^{k_2})^{k_1'} \sum_{m=0}^{L_1-1} \omega_{L_1}^{mk_1'} a_{k_2+L_2m}, \tag{2}$$

we obtain

$$\hat{a}_j = (\zeta^{L_1})^{j_2'} \sum_{r=0}^{L_2-1} \omega_{L_2}^{rj_2'} b_{r+L_2j_1}. \tag{3}$$

In other words, if a, b and $\hat{a}$ are thought of as $L_1 \times L_2$ matrices, then b is the result of applying an appropriately weighted DFT to each of the columns of a, and $\hat{a}$ is the result of applying an appropriately weighted DFT to each of the rows of b.

For the base case $L = 2$ the routines compute the TFT/ITFT directly. If $L = 2^\ell \ge 4$, they write $L = L_1L_2$ where $L_1 = 2^{\lfloor \ell/2 \rfloor}$ and $L_2 = 2^{\lceil \ell/2 \rceil}$, so that $1 < L_1 < L$ and $1 < L_2 < L$. They treat the array as an $L_1 \times L_2$ matrix, and recurse into TFTs/ITFTs on the columns and rows. The column transforms correspond to recursively applying the TFT/ITFT to the transform given by (2); the row transforms similarly correspond to the transform given by (3). (Van der Hoeven's TFT and ITFT algorithms are essentially the special case obtained by taking $L_1 = 2$ and $L_2 = L/2$.)

We will denote by c_u the uth column $(x_u, x_{u+L_2}, \ldots, x_{u+(L_1-1)L_2})$ and by r_u the uth row $(x_{uL_2}, x_{uL_2+1}, \ldots, x_{uL_2+L_2-1})$. A real implementation would use auxiliary variables to describe such sub-arrays; for example, a pointer to the first element and a stride parameter.

Common to both routines is the decomposition $n = n_2 + L_2n_1$ where $0 \le n_1 \le L_1$ and $0 \le n_2 < L_2$, and where $n_1 = L_1$ implies $n_2 = 0$. This partitions the first n cells of the array into n_1 complete rows followed by n_2 cells in the subsequent row. The parameter z is decomposed similarly into z_1 and z_2.

3. A cache-friendly TFT

We first consider the TFT; the idea is to compute only those parts of the DFT that are requested. We handle the column transforms first, followed by the row transforms.

Theorem 1. *Algorithm* 1 *correctly computes the TFT. The base case is executed at most* $\min((n-1)\ell/2 + L - 1, L\ell/2)$ *times.*

Proof. We first consider the base case $L = 2$. The relevant DFT is given by $(\hat{a}_0, \hat{a}_1) = (a_0 + a_1, \zeta(a_0 - a_1))$. If $z = 1$ then $a_1 = 0$, and the transform becomes simply $(\hat{a}_0, \hat{a}_1) = (a_0, \zeta a_0)$. If $n = 2$ then both $\hat{a}_0$ and $\hat{a}_1$ must be computed; if $n = 1$ then only $\hat{a}_0$ is needed. Lines 2–4 handle the various cases.

Now we consider the recursive case, for $L = 2^\ell \ge 4$. Fig. 3(a)–(c) show the possible input configurations, for $L = 64$, $L_1 = L_2 = 8$. Cells labelled a contain some a_i; cells labelled $\cdot$ contain uninitialised data, but implicitly represent $a_i = 0$.

Algorithm 1: CACHEFRIENDLYTFT$(L, \zeta, z, n; (x_0, \ldots, x_{L-1}))$

Input: $L = 2^\ell \geq 2$, $\zeta \in R^\times$,
$1 \leq z \leq L$, $1 \leq n \leq L$,
$x_i = a_i$ for $0 \leq i < z$
Output: $x_i = \hat{a}_i$ for $0 \leq i < n$

1 **if** $L = 2$ **then**
  `// base case`
2   **if** $n = 2$ **and** $z = 2$ **then** $(x_0, x_1) \leftarrow (x_0 + x_1, \zeta(x_0 - x_1))$
3   **if** $n = 2$ **and** $z = 1$ **then** $x_1 \leftarrow \zeta x_0$
4   **if** $n = 1$ **and** $z = 2$ **then** $x_0 \leftarrow x_0 + x_1$
5   **return**
6 **end**

`// recursive case`
7 $L_1 \leftarrow 2^{\lfloor \ell/2 \rfloor}$, $L_2 \leftarrow 2^{\lceil \ell/2 \rceil}$
8 $n_2 \leftarrow n \bmod L_2$, $n_1 \leftarrow \lfloor n/L_2 \rfloor$, $n_1' \leftarrow \lceil n/L_2 \rceil$
9 $z_2 \leftarrow z \bmod L_2$, $z_1 \leftarrow \lfloor z/L_2 \rfloor$
10 **if** $z_1 > 0$ **then** $z_2' \leftarrow L_2$ **else** $z_2' \leftarrow z_2$

`// column transforms`
11 **for** $0 \leq u < z_2$ **do** CACHEFRIENDLYTFT$(L_1, \omega_L^u \zeta, z_1 + 1, n_1'; c_u)$
12 **for** $z_2 \leq u < z_2'$ **do** CACHEFRIENDLYTFT$(L_1, \omega_L^u \zeta, z_1, n_1'; c_u)$

`// row transforms`
13 **for** $0 \leq u < n_1$ **do** CACHEFRIENDLYTFT$(L_2, \zeta^{L_1}, z_2', L_2; r_u)$
14 **if** $n_2 > 0$ **then** CACHEFRIENDLYTFT$(L_2, \zeta^{L_1}, z_2', n_2; r_{n_1})$

Diagram (a) shows the case $z_1 = 0$, in which case $z_2' = z_2$. Diagram (b) shows the case $z_1 > 0$ and $z_2 = 0$, and diagram (c) shows the case $z_1 > 0, z_2 > 0$. In these latter cases $z_2' = L_2$. Lines 11–12 apply the TFT recursively to the columns to evaluate the first n_1' rows of (2). Line 11 handles those columns containing $z_1 + 1$ nonzero entries; line 12 handles those containing only z_1 nonzero entries.

After lines 11–12 have been executed, we have $x_i = b_i$ for $0 \leq i_1 < n_1'$ and $0 \leq i_2 < z_2'$, and we also know that $b_i = 0$ for $z_2' \leq i < L_2$ (the latter statement is non-vacuous only if $z_1 = 0$). Fig. 4 illustrates the situation: cells labelled b contain some b_i; cells labelled $\cdot$ contain unspecified data but implicitly represent $b_i = 0$; cells labelled ? are meaningless. Diagram (a) shows the case $z_2' < L_2$, and diagram (b) shows $z_2' = L_2$.

Next, lines 13–14 apply the TFT recursively to the first n_1' rows to evaluate (3). Fig. 5 shows the possible output configurations. Cells labelled $\hat{a}$ contain some $\hat{a}_i$; cells labelled ? contain meaningless data. Diagram (a) shows the case $n_2 > 0$, where $n_1' = n_1 + 1$, and diagram (b) shows the case $n_2 = 0$, where $n_1' = n_1$. Line 13 handles the first n_1 rows, where $\hat{a}_i$ must be computed for $0 \leq i_2 < L_2$; line 14 handles the remaining partial row, where $\hat{a}_i$ is needed only for $0 \leq i_2 < n_2$.

We prove the complexity estimate by induction on L. For $L = 2$ the bound is $\min((n-1)/2 + 1, 1) = 1$, so the estimate holds. Now assume that $L \geq 4$, and let $\ell_1 = \log_2 L_1$ and $\ell_2 = \log_2 L_2$.

We first verify that the number of calls to the base case is bounded by $L\ell/2$. By induction, lines 11–12 call the base case at most $L_2(L_1\ell_1/2)$ times, and lines 13–14 call it at most $n_1'(L_2\ell_2/2) \leq L_1(L_2\ell_2/2)$ times. The sum is $L_1L_2(\ell_1 + \ell_2)/2 = L\ell/2$.

Second, we must verify that the number of calls is bounded by $(n-1)\ell/2 + L - 1$. Let $\delta = n_1' - n_1 \in \{0, 1\}$. Lines 11–12 call the base case at most $L_2((n_1 + \delta - 1)\ell_1/2 + L_1 - 1)$ times, line 13 calls it at most $n_1(L_2\ell_2/2)$ times, and line 14 calls it at most $\delta((n_2 - 1)\ell_2/2 + L_2 - 1)$ times. The sum of these terms is $\frac{1}{2}X + Y$ where

$$\begin{aligned}
X &= L_2(n_1 - 1)\ell_1 + n_1L_2\ell_2 + \delta(L_2\ell_1 + (n_2 - 1)\ell_2) \\
&= (n - n_2)\ell - L_2\ell_1 + \delta(L_2\ell_1 + (n_2 - 1)\ell_2) \\
&= (n - 1)\ell + (\delta - 1)L_2\ell_1 + (n_2 - 1)(\delta\ell_2 - \ell), \\
Y &= L_2(L_1 - 1) + \delta(L_2 - 1) = L - 1 + (\delta - 1)(L_2 - 1).
\end{aligned}$$

If $\delta = 1$, then $n_2 \geq 1$ and $(n_2 - 1)(\delta\ell_2 - \ell) = -\ell_1(n_2 - 1) \leq 0$. If $\delta = 0$ then $n_2 = 0$ and $(\delta - 1)L_2\ell_1 + (n_2 - 1)(\delta\ell_2 - \ell) = -L_2\ell_1 + \ell_1 + \ell_2$, which is non-positive since $L_2 = 2^{\ell_2} \geq \ell_2 + 1$. The desired estimate holds in both cases. □

4. A cache-friendly inverse TFT

The ITFT cannot be implemented by simply running the TFT in reverse, because when the ITFT commences there is insufficient information to perform all the row transforms. In particular, if $n \not\equiv 0 \bmod L_2$, then the $\lfloor n/L_2 \rfloor$th row contains some $\hat{a}_i$ but does not contain the corresponding b_i needed to apply (3).

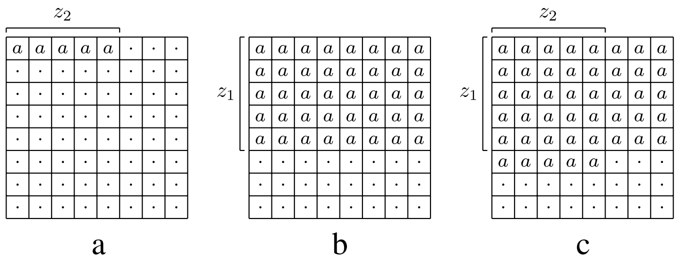

Fig. 3. Before line 11 of CacheFriendlyTFT.

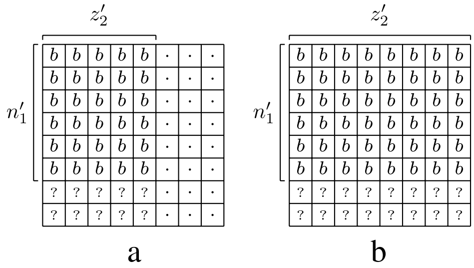

Fig. 4. After line 12 of CacheFriendlyTFT.

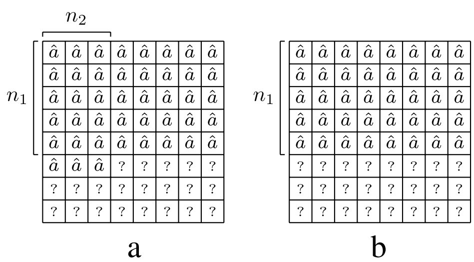

Fig. 5. After lines 13–14 of CacheFriendlyTFT.

To circumvent this difficulty, we proceed as follows. We first perform as many row transforms as possible. We are then able to perform *some* of the column transforms. When these are complete, it becomes possible to execute the last row transform that was inaccessible before. After this row transform, the remainder of the column transforms may be completed. Algorithm 2 gives a precise statement.

Theorem 2. *Algorithm* 2 *correctly computes the ITFT. The base case is executed at most* $\min((n+f-1)\ell/2+L-1, L\ell/2)$ *times.*

Proof. We first consider the base case $L = 2$. As before, the relevant DFT is given by $(\hat{a}_0, \hat{a}_1) = (a_0+a_1, \zeta(a_0-a_1))$. If $n = 2$, then we must have $z = 2$ and $f = 0$, and we are computing the map $(\hat{a}_0, \hat{a}_1) \mapsto (2a_0, 2a_1) = (\hat{a}_0 + \zeta^{-1}\hat{a}_1, \hat{a}_0 - \zeta^{-1}\hat{a}_1)$. This is handled by line 2. Now suppose that $n = 1$. If $f = 1$ and $z = 2$, we must compute the map $(\hat{a}_0, 2a_1) \mapsto (2a_0, \hat{a}_1) = (2\hat{a}_0 - 2a_1, \zeta(\hat{a}_0 - 2a_1))$ (van der Hoeven's 'cross butterfly'). This is handled by line 3. Lines 4–6 handle the analogous cases where $f = 0$ (the second output is not needed) or where $z = 1$ (a_1 is assumed to be zero). Finally suppose that $n = 0$. Then we must have $f = 1$. If $z = 2$, we must compute $(2a_0, 2a_1) \mapsto \hat{a}_0 = (2a_0 + 2a_1)/2$. This is handled by line 7. The $z = 1$ case (where we assume $a_1 = 0$) is handled by line 8.

We now suppose that $L \geq 4$ and consider the four cases below. Figs. 6–10 illustrate the various stages of the algorithm for each of these cases. Cells labelled a, b and $\hat{a}$ indicate respectively La_i, L_2b_i or $\hat{a}_i$; cells labelled $\cdot$ are uninitialised, but implicitly represent $a_i = 0$; cells containing ? contain unspecified data not used in subsequent computations. A symbol in parentheses indicates that the symbol is only valid if $f = 1$; if $f = 0$ the cell behaves like a ? cell. Cells in bold are those about to be transformed by a recursive call.

Case (a): $z_1 = 0$. This implies that $0 < n_2 \leq z_2 = z_2' < L_2$, $n_1 = 0$, $m = n_2$, $m' = z_2$, and $f' = 1$. Line 17 has no effect since $n_1 = 0$. Line 18 computes $x_i = L_2b_i$ for $n_2 \leq i < z_2$, and destroys x_i for $n_2 \leq i_2 < z_2$, $1 \leq i_1 < L_1$. Line 19 has no effect since $z_2 = z_2'$. Line 20 computes $x_i = L_2b_i$ for $0 \leq i < n_2$, computes $x_{n_2} = x_n = \hat{a}_n$ if $f = 1$, and destroys x_i for $n_2 + f \leq i < L_2$. Line 21 computes $x_i = La_i$ for $0 \leq i < n_2 = n$, and destroys x_i for $0 \leq i_2 < n_2$, $1 \leq i_1 < L_1$. Line 22 has no effect since $m = n_2$.

Case (b): $z_1 > 0$ and $n_2 = 0$. This implies that $z_1 \geq n_1 > 0$, $z_2' = L_2$, $m = 0$, $m' = z_2$ and $f' = f$. Line 17 computes $x_i = L_2b_i$ for $0 \leq i < n_1L_2 = n$. Lines 18–19 compute $x_i = La_i$ for $0 \leq i < n_1L_2 = n$, and if $f = 1$ also compute $x_i = L_2b_i$ for

Algorithm 2: CacheFriendlyITFT$(L, \zeta, z, n, f; (x_0, \ldots, x_{L-1}))$

Input: $L = 2^\ell \geq 2, \zeta \in R^\times$,
$f \in \{0, 1\}, 1 \leq n + f \leq L, 1 \leq z \leq L, z \geq n$,
$x_i = \hat{a}_i$ for $0 \leq i < n, x_i = La_i$ for $n \leq i < z$

Output: $x_i = La_i$ for $0 \leq i < n$,
$x_n = \hat{a}_n$ if $f = 1$

1 **if** $L = 2$ **then**
    // base case
2   **if** $n = 2$ **then** $(x_0, x_1) \leftarrow (x_0 + \zeta^{-1}x_1, x_0 - \zeta^{-1}x_1)$
3   **if** $n = 1$ **and** $f = 1$ **and** $z = 2$ **then** $(x_0, x_1) \leftarrow (2x_0 - x_1, \zeta(x_0 - x_1))$
4   **if** $n = 1$ **and** $f = 1$ **and** $z = 1$ **then** $(x_0, x_1) \leftarrow (2x_0, \zeta x_0)$
5   **if** $n = 1$ **and** $f = 0$ **and** $z = 2$ **then** $x_0 \leftarrow 2x_0 - x_1$
6   **if** $n = 1$ **and** $f = 0$ **and** $z = 1$ **then** $x_0 \leftarrow 2x_0$
7   **if** $n = 0$ **and** $z = 2$ **then** $x_0 \leftarrow (x_0 + x_1)/2$
8   **if** $n = 0$ **and** $z = 1$ **then** $x_0 \leftarrow x_0/2$
9   **return**
10 **end**

// recursive case
11 $L_1 \leftarrow 2^{\lfloor \ell/2 \rfloor}, L_2 \leftarrow 2^{\lceil \ell/2 \rceil}$
12 $n_2 \leftarrow n \bmod L_2, n_1 \leftarrow \lfloor n/L_2 \rfloor$
13 $z_2 \leftarrow z \bmod L_2, z_1 \leftarrow \lfloor z/L_2 \rfloor$
14 **if** $n_2 + f > 0$ **then** $f' \leftarrow 1$ **else** $f' \leftarrow 0$
15 **if** $z_1 > 0$ **then** $z_2' \leftarrow L_2$ **else** $z_2' \leftarrow z_2$
16 $m \leftarrow \min(n_2, z_2), m' \leftarrow \max(n_2, z_2)$

// row tranforms
17 **for** $0 \leq u < n_1$ **do** CacheFriendlyITFT$(L_2, \zeta^{L_1}, L_2, L_2, 0; r_u)$

// rightmost column transforms
18 **for** $n_2 \leq u < m'$ **do** CacheFriendlyITFT$(L_1, \omega_L^u\zeta, z_1 + 1, n_1, f'; c_u)$
19 **for** $m' \leq u < z_2'$ **do** CacheFriendlyITFT$(L_1, \omega_L^u\zeta, z_1, n_1, f'; c_u)$

// last row transform
20 **if** $f' = 1$ **then** CacheFriendlyITFT$(L_2, \zeta^{L_1}, z_2', n_2, f; r_{n_1})$

// leftmost column transforms
21 **for** $0 \leq u < m$ **do** CacheFriendlyITFT$(L_1, \omega_L^u\zeta, z_1 + 1, n_1 + 1, 0; c_u)$
22 **for** $m \leq u < n_2$ **do** CacheFriendlyITFT$(L_1, \omega_L^u\zeta, z_1, n_1 + 1, 0; c_u)$

$0 \leq i_2 < L_2, i_1 = n_1$; they destroy x_i for $L_2(n_1 + f) \leq i < L$. If $f = 1$, then line 20 computes $x_{n_1 L_2} = x_n = \hat{a}_n$ and destroys x_i for $n_1 L_2 < i < (n_1 + 1)L_2$. Lines 21–22 have no effect since $m = n_2 = 0$.

Case (c): $z_1 > 0, n_2 > 0$ and $n_2 \leq z_2$. This implies that $z_2' = L_2, 0 \leq n_1 < L_1, m = n_2, m' = z_2$, and $f' = 1$. Line 17 computes $x_i = L_2 b_i$ for $0 \leq i < n_1 L_2$. For each $n_2 \leq i_2 < L_2$, lines 18–19 compute $x_i = La_i$ for $0 \leq i_1 < n_1$, compute $x_i = L_2 b_i$ for $i_1 = n_1$, and destroy x_i for $n_1 < i_1 < L_1$. Line 20 computes $x_i = b_i$ for $0 \leq i_2 < n_2, i_1 = n_1$, computes $x_n = \hat{a}_n$ if $f = 1$, and destroys x_i for $n_2 + f \leq i_2 < L_2, i_1 = n_1$. Finally, for each $0 \leq i_2 < n_2$, lines 21–22 compute $x_i = La_i$ for $0 \leq i_1 < n_1 + 1$ and destroy x_i for $n_1 + 1 \leq i_1 < L_1$.

Case (d): $z_1 > 0, n_2 > 0$ and $n_2 > z_2$. The discussion for this case is essentially the same as for (c), with m and m' exchanged, and with slightly different diagrams.

Now we verify the complexity bound. The argument is similar to that used for the TFT. For $L = 2$ the bound is $\min((n + f - 1)/2 + 1, 1) = 1$, so the estimate holds. Now assume that $L \geq 4$, and let $\ell_1 = \log_2 L_1$ and $\ell_2 = \log_2 L_2$.

We first verify that the number of calls to the base case is bounded by $L\ell/2$. By induction, lines 18–19 and 21–22 call the base case at most $L_2(L_1\ell_1/2)$ times altogether. Lines 17 and 20 call it at most $L_1(L_2\ell_2/2)$ times (note that if line 20 is executed then $n_1 \leq L_1 - 1$). The sum is $L_1 L_2(\ell_1 + \ell_2)/2 = L\ell/2$.

Second, we must verify that the number of calls is bounded by $(n + f - 1)\ell/2 + L - 1$. Line 17 calls the base case at most $n_1(L_2\ell_2/2)$ times, lines 18–19 call it at most $(L_2 - n_2)((n_1 + f' - 1)\ell_1/2 + L_1 - 1)$ times, line 20 calls it at most

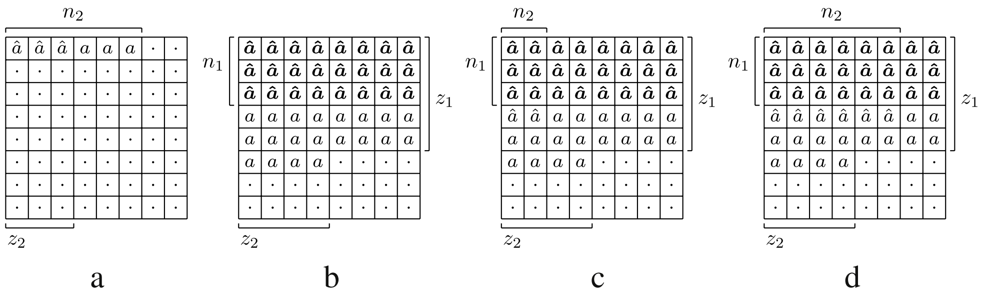

Fig. 6. Before line 17 of CacheFriendlyITFT. The bold rows are about to be transformed by line 17.

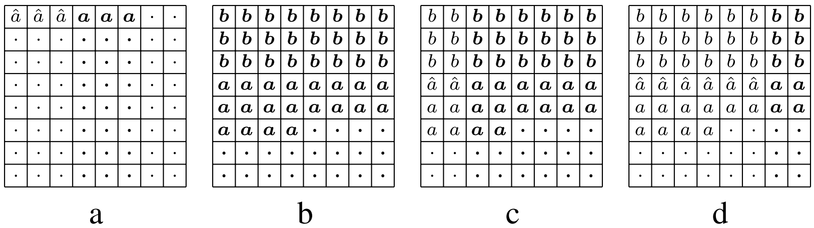

Fig. 7. After line 17 of CacheFriendlyITFT. The bold columns are about to be transformed by lines 18–19.

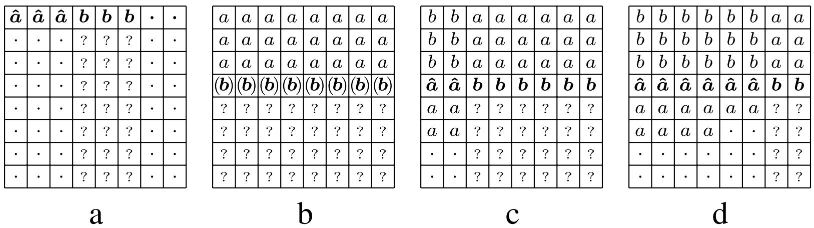

Fig. 8. After lines 18–19 of CacheFriendlyITFT. The bold row is about to be transformed by line 20.

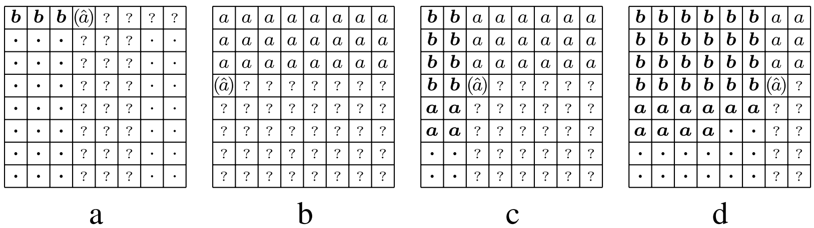

Fig. 9. After line 20 of CacheFriendlyITFT. The bold columns are about to be transformed by lines 21–22.

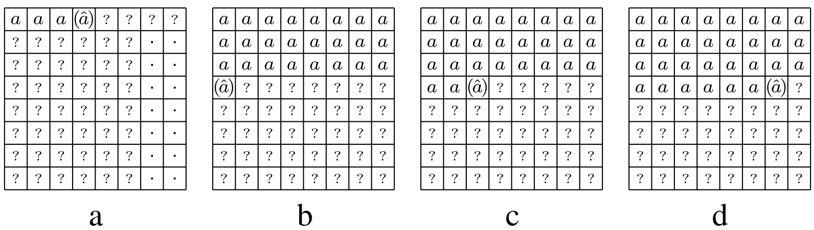

Fig. 10. After lines 21–22 of CacheFriendlyITFT.

$f'((n_2+f-1)\ell_2/2+L_2-1)$ times, and lines 21–22 call it at most $n_2(n_1\ell_1/2+L_1-1)$ times. The sum of these terms is $\frac{1}{2}X+Y$, where

$$\begin{aligned}
X &= n_1L_2\ell_2 + L_2(n_1+f'-1)\ell_1 - (f'-1)n_2\ell_1 + f'(n_2+f-1)\ell_2 \\
&= (n-n_2)\ell + (f'-1)(L_2-n_2)\ell_1 + f'(n_2+f-1)\ell_2 \\
&= (n+f-1)\ell + (f'-1)(L_2-n_2)\ell_1 + (n_2+f-1)(\ell_2 f'-\ell), \\
Y &= L_2(L_1-1) + f'(L_2-1) = L-1+(f'-1)(L_2-1).
\end{aligned}$$

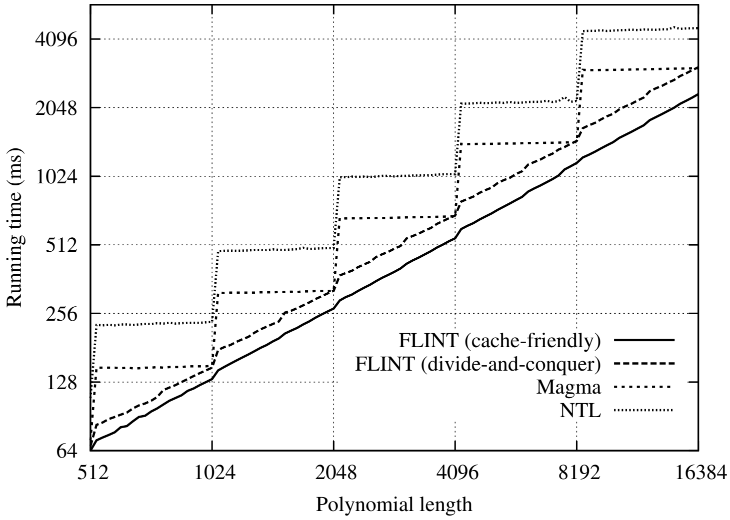

Fig. 11. Performance of several implementations of the Schönhage–Strassen algorithm for 8000-bit coefficients.

If $f' = 1$ then $n_2 + f \geq 1$ and the bound follows since $\ell_2 f' - \ell = -\ell_1 \leq 0$. If $f' = 0$ then $n_2 = f = 0$ and the bound follows since $-L_2\ell_1 + \ell \leq 0$ (as in the proof of Theorem 1). $\square$

5. Empirical performance and applications

5.1. The Schönhage–Strassen algorithm

Both the Magma computer algebra system (version 2.14-15, [3]) and Victor Shoup's NTL library (version 5.4.2, [12]) use the Schönhage–Strassen algorithm [14] for multiplication of dense polynomials in $\mathbf{Z}[x]$ when (roughly speaking) the coefficient size of the input polynomials (in bits) is larger than their degree. The algorithm may be sketched as follows. Suppose that $f, g \in \mathbf{Z}[x]$, and put $h = fg$. Let $R = \mathbf{Z}/(2^{kN/2} + 1)\mathbf{Z}$, where we choose $N = 2^n > \deg h$ and $kN/2$ larger than the size of the coefficients of h. Multiply the polynomials in $R[x]/(x^N - 1)$, using an FFT with respect to the principal Nth root of unity $\omega_N = 2^k \in R$, and lift the result back to $\mathbf{Z}[x]$. Arithmetic in R is especially efficient owing to the ease of reduction modulo $2^{kN/2} + 1$ and of multiplication by powers of ω_N.

The author, in joint work with William Hart, implemented the Schönhage–Strassen algorithm using the techniques of this paper to improve smoothness and locality. The implementation is part of the `fmpz_poly` module in the FLINT library (version 1.0.13, [8]), which is used as the default back-end for arithmetic in $\mathbf{Z}[x]$ in the Sage computer algebra system (version 3.1.1, [13]).

The following performance measurements were conducted on a 16-core 2.6 GHz Opteron server running Ubuntu Linux. This is a 64-bit processor with a 64 KB L1 cache and 1 MB L2 cache. Only a single core was used for the tests. Our own code and NTL were compiled with gcc 4.1.3, and linked with GMP (GNU Multiple Precision Arithmetic Library, [4]) version 4.2.3. We also applied an assembly patch of Pierrick Gaudry that improves the performance of GMP on the Opteron. Magma also uses Gaudry's patch, and links statically against GMP.

Fig. 11 compares four implementations for the case of polynomials with random non-negative 8000-bit coefficients, with lengths ranging from 512 to 16 384 in 5% increments. The graphs for Magma and NTL exhibit the jumps characteristic of FFT-based multiplication algorithms. The two graphs for FLINT show the multiplication performance obtained for van der Hoeven's divide-and-conquer truncated transforms, and for the cache-friendly truncated transforms. The latter is between 15% and 35% faster than the former for this range of polynomial lengths, and the relative improvement in performance increases with the degree. Note that the Fourier coefficients are about 16 000 bits long ($\approx$2 KB), so about 32 coefficients fit into the L1 cache and about 512 coefficients fit into the L2 cache.

5.2. The Schönhage–Nussbaumer algorithm

The author implemented the cache-friendy transforms in the context of the Schönhage–Nussbaumer algorithm [11,10] for multiplication in $S[x]$ where $S = \mathbf{Z}/m\mathbf{Z}$ and where m is an odd word-sized modulus. The implementation is part of the `zn_poly` polynomial arithmetic library (version 0.9, [7]). The code has been used in several number-theoretic applications, including computations of zeta functions of hyperelliptic curves over prime fields of large characteristic [5], computations of L-functions of hyperelliptic curves over $\mathbf{Q}$ [9], computing Hilbert class polynomials [15], and an ongoing project with Joe Buhler to extend the verification of Vandiver's conjecture and computation of irregular primes and cyclotomic invariants carried out in [2].

The basic idea of the Schönhage–Nussbaumer algorithm is to split the input polynomials into pieces of length $M/4$, and then map the problem to a convolution in $R[z]/(z^K - 1)$ for $R = S[y]/(y^{M/2} + 1)$, where $K \mid M$ so that R contains a principal Kth root of unity (namely $y^{M/K}$), and where K is large enough to accommodate the product. Our implementation performs the FFTs over R using the transforms of Sections 3 and 4, ensuring relatively smooth performance as a function of the input polynomial length. The pointwise multiplications are handled using a multipoint Kronecker substitution method [6], switching to Nussbaumer's algorithm for sufficiently large M. (Note that we do *not* perform an FFT over $\mathbf{Z}/m\mathbf{Z}$; such an FFT is usually not possible since $\mathbf{Z}/m\mathbf{Z}$ rarely contains appropriate roots of unity.)

We compared the performance of the cache-friendly transforms to the divide-and-conquer transforms for a range of polynomial lengths (10^4 to 3×10^7) and modulus sizes (5 to 63 bits). We observed a modest improvement in speed of up to 15%, depending on the polynomial length and modulus. As expected, polynomials of higher degree enjoy a greater relative improvement, as locality plays a greater role in such multiplications. Somewhat counterintuitively, the modulus size had the opposite effect on relative performance. This may be explained by noting that the FFTs in our implementation operate on arrays with each element of $\mathbf{Z}/m\mathbf{Z}$ occupying a single machine word, so the total FFT time does not depend on the modulus; on the other hand, the pointwise multiplications are faster for smaller moduli, as the Kronecker substitution reduces them to smaller integer multiplications. The implementation thus spends a smaller proportion of the total time in the FFTs when the modulus is larger, leading to a smaller relative improvement derived from the cache-friendly transforms.

6. The small coefficient case

In the applications described in Section 5, elements of the coefficient ring R occupy moderately large blocks of memory. However, FFTs are also commonly applied over 'small' coefficients, such as double-precision floating point numbers, or residues modulo a word-sized prime p where $\mathbf{Z}/p\mathbf{Z}$ contains suitable roots of unity. We have not attempted an implementation in this context, but in this section we make several relevant observations.

An essential consideration in the small coefficient case is spatial locality, which we have largely ignored in this paper. In typical contemporary cache hardware, the cache is organised into cache lines, each capable of storing several words from consecutive locations in main memory. If an algorithm operates on coefficients spaced out in memory, then only a single word of each cache line will be utilised, greatly reducing the effective size of the cache. Moreover, the mapping from physical addresses to cache lines often depends on only the last few bits of the address. If two coefficients are separated by a large power-of-two distance in memory – exactly the situation during the column transforms of a matrix FFT – then the cache cannot simultaneously hold both of them (although this can be mitigated to some extent by cache associativity). The standard solution to these problems is to transpose the matrix for the duration of the column transforms, using a cache-friendly matrix transpose algorithm, so that the subtransforms always operate on consecutive data. A similar approach would be needed to adapt our TFTs/ITFTs to the small coefficient case.

A second remark is that in the small coefficient case, it is quite reasonable to zero-pad the inputs so that there is no 'partial row'. The rationale is that the lowest level of cache can hold a large number of coefficients, making the penalty for zero-padding quite small. For example, suppose that the cache can hold 2^{13} coefficients (typical for a 64 KB L1 cache with double-precision floating-point coefficients), and that we are multiplying polynomials whose product has length $n = 12\,801 = 100 \times 2^7 + 1$. This requires a transform length of 2^{14}, which we may decompose into a $2^7 \times 2^7$ matrix. If we zero-pad the inputs so that n increases to $12928 = 101 \times 2^7$, an integral number of rows, the running time penalty incurred is at most 1%. This approach simplifies the ITFT routine considerably, since it may be implemented by simply reversing the steps of the TFT, removing the need for the special row transform (line 20 of Algorithm 2). The reduction in code complexity is likely worthwhile. We also note that the presence of a partial row makes it more difficult to maintain spatial locality during the special row transform.

Finally, in the implementations described in Section 5, the parameter $\zeta = \omega^s$ is represented simply by the integer s. With this representation, computing roots of unity (for example, computing ζ^{L_1} in line 13 of Algorithm 1) is very cheap compared to the cost of arithmetic in R. In the small coefficient case this is no longer necessarily true, and the cost of computing or storing roots of unity must be taken into account.

Acknowledgments

Many thanks to William Hart for his collaboration in implementing these algorithms in FLINT, to William Hart, Andrew Sutherland, Joris van der Hoeven and the referees for their comments on a draft of this paper, and to the Department of Mathematics at Harvard University for supplying the hardware on which the performance measurements were carried out.

References

[1] David H. Bailey, FFTs in external or hierarchical memory, J. Supercomput. 4 (1990) 23–35.
[2] Joe Buhler, Richard Crandall, Reijo Ernvall, Tauno Metsänkylä, M. Amin Shokrollahi, Irregular primes and cyclotomic invariants to 12 million, J. Symbolic Comput. 31 (1–2) (2001) 89–96. Computational algebra and number theory (Milwaukee, WI, 1996).
[3] Wieb Bosma, John Cannon, Catherine Playoust, The Magma algebra system. I. The user language, J. Symbolic Comput. 24 (3–4) (1997) 235–265.
[4] Torbjörn Granlund, The GNU multiple precision arithmetic library, 2008. http://gmplib.org/.
[5] David Harvey, Kedlaya's algorithm in larger characteristic, Int. Math. Res. Notices 2007 (2007) 29 pp. Article ID rnm095, doi:10.1093/imrn/rnm095.

[6] David Harvey, Faster polynomial multiplication via multipoint Kronecker substitution, arXiv preprint, 2008. cs.SC/0712.4046v1.
[7] David Harvey, The zn_poly library, 2008. http://www.cims.nyu.edu/~harvey/zn_poly/.
[8] William Hart, David Harvey, The FLINT library, 2008. http://www.flintlib.org/.
[9] Kiran S. Kedlaya, Andrew Sutherland, Computing L-series of hyperelliptic curves, in: ANTS VIII, in: Lecture Notes in Computer Science, vol. 5011, Springer, 2008, pp. 312–326.
[10] Henri J. Nussbaumer, Fast polynomial transform algorithms for digital convolution, IEEE Trans. Acoust. Speech Signal Process. 28 (2) (1980) 205–215.
[11] A. Schönhage, Schnelle Multiplikation von Polynomen über Körpern der Charakteristik 2, Acta Informat. 7 (4) (1976–77) 395–398.
[12] Victor Shoup, NTL: A library for doing number theory, 2007. http://www.shoup.net/ntl/.
[13] William Stein, David Joyner, Sage: System for algebra and geometry experimentation, Communications in Computer Algebra (ACM SIGSAM Bulletin) 39 (2) (2005) 61–64. http://sagemath.org/.
[14] A. Schönhage, V. Strassen, Schnelle Multiplikation grosser Zahlen, Computing (Arch. Elektron. Rechnen) 7 (1971) 281–292.
[15] Andrew V. Sutherland, Computing Hilbert class polynomials with the Chinese Remainder Theorem, 2009 (in press). URL: http://arxiv.org/abs/0903.2785.
[16] Joris van der Hoeven, The truncated Fourier transform and applications, in: ISSAC 2004, ACM, New York, 2004, pp. 290–296.
[17] Joris van der Hoeven, Notes on the truncated Fourier transform, unpublished, retrieved September 2008 from http://www.math.u-psud.fr/~vdhoeven/, 2005.