Code generation for polynomial multiplication

Ling Ding[1] and Éric Schost[2]

[1] ORCCA, Computer Science Department, The University of Western Ontario, London, Ontario, Canada
lding6@csd.uwo.ca
[2] ORCCA, Computer Science Department, The University of Western Ontario, London, Ontario, Canada
eschost@uwo.ca

Abstract. We discuss the family of "divide-and-conquer" algorithms for polynomial multiplication, that generalize Karatsuba's algorithm. We give explicit versions of *transposed* and *short* products for this family of algorithms and describe code generation techniques that result in high-performance implementations.

1 Introduction

Polynomial multiplication is a cornerstone of higher-level algorithms: fast algorithms for Euclidean division, GCD, Chinese remaindering, factorization, Newton iteration, etc, depend on fast (subquadratic) algorithms for polynomial multiplication [20]. This article describes implementation techniques for several aspects of this question; we focus on *dense* polynomial arithmetic, as opposed the sparse model [12].

Variants of polynomial multiplication. To fix notation, we let R be our base ring, and for $n \in \mathbb{N}_{>0}$, we let $\mathsf{R}[x]_n$ be the set of polynomials in $\mathsf{R}[x]$ of degree less than n. We will write the input polynomials as

$$A = a_0 + a_1 x + \cdots + a_{n-1} x^{n-1} \in \mathsf{R}[x]_n, \quad B = b_0 + b_1 x + \cdots + b_{n-1} x^{n-1} \in \mathsf{R}[x]_n;$$

note that the *number of terms* of A and B is at most n. Note also that we assume the same degree upper bound on A and B: this needs not be a sensible assumption in general, but makes sense in many applications (such as Newton iteration or GCD). Our first objective is to compute the coefficients of the product

$$C = AB = c_0 x + c_1 x + \cdots + c_{2n-2} x^{2n-2} \in \mathsf{R}[x]_{2n-1}.$$

This operation will be called *plain* multiplication. It turns out that two other forms of multiplication are useful: the first is the *transposed* multiplication, closely related to the *middle product* of [9]. The other noteworthy form is the *short product*, introduced in [14] and studied in [10]. Both are detailed in Section 3, together with mentions of their applications.

Our contribution: code generation for divide-and-conquer algorithms. Beyond from the naive algorithm, the main classes of fast algorithms are generalizations of Karatsuba's approach [11, 18, 21], where a given pattern is used in a divide-and-conquer fashion, and FFT-like approaches, that use or build suitable evaluation points [7, 15, 6], usually roots of unity.

In this paper, we focus on the former family. Despite its richness, we are not aware of a systematic treatment of algorithms for transposed or short product in this context. First, we fill this gap, giving explicit versions of such algorithms. We will see the algorithms of this family can be described in finite terms, by triples of graphs. Then, we describe a code generator that turns such graphs into C implementations, avoiding the need to reimplement everything from scratch; the performance of these implementations is among the best known to us.

Previous work. Most algorithms for plain multiplication discussed here are well-known: the most classical ones are due to Karatsuba [11] and Toom [18], with improvements in [1]; some less well-known ones are due to Winograd [21].

It is well-known that any algorithm for plain multiplication can be transformed into an algorithm for transposed multiplication; this is already in [21], and is developed in [9, 3], together with applications. However, while the *existence* of transposed algorithms was known, our general derivation of *explicit* divide-and-conquer algorithms is new, to our knowledge: the only explicit examples in [9, 3] describe Karatsuba multiplication. Similarly, the possibility of using any divide-and-conquer algorithm to perform short products is hinted at in [10], but no details are given; only the case of Karatsuba multiplication is developed in great detail. Our general presentation is, to our knowledge, new.

Computational model. Our problems are bilinear; computations will thus be done as follows: linear combinations of the inputs are computed (separately), followed by pairwise products of the values thus obtained; the result is deduced by a last series of linear combinations of these products. Our complexity estimates count the linear operations and the pairwise products.

2 Preliminaries: graphs for linear maps

We address first the linear part of the computations: we recall here how to compute linear maps using a graphical representation. The material in this section is well-known [5, Ch. 13].

Definition. A *linear graph* $\mathcal{G}$ consists of

- a directed acyclic graph (V, E) with k inputs and ℓ outputs,
- a weight function λ which assigns a weight $\lambda(e) \in \mathsf{R}$ to each edge e,
- orderings $(A_0, \dots, A_{k-1})$ and $(F_0, \dots, F_{\ell-1})$ of the inputs and outputs.

One assigns a matrix to a linear graph in a straightforward way. Each vertex is assigned a value, obtained by following the "flow" from inputs to outputs: going from a vertex v to a vertex v' along an edge e, the value at v is multiplied by

the weight $\lambda(e)$; the value at v' is obtained by summing the contributions of all incoming edges. The values obtained at each vertex are linear combinations of the values $a_0, \dots, a_{k-1}$ given at the inputs $A_0, \dots, A_{k-1}$. In particular, let $f_0, \dots, f_{\ell-1}$ be the values computed by the output nodes $F_0, \dots, F_{\ell-1}$; f_i can thus be written $f_i = L_{i,0}a_0 + \cdots + L_{i,k-1}a_{k-1}$, for some constants $L_{i,j}$, so that

$$\begin{bmatrix} f_0 \\ \vdots \\ f_{\ell-1} \end{bmatrix} = \mathbf{L} \begin{bmatrix} a_0 \\ \vdots \\ a_{k-1} \end{bmatrix}, \quad \text{with} \quad \mathbf{L} = \begin{bmatrix} L_{0,0} & \cdots & L_{0,k-1} \\ \vdots & & \vdots \\ L_{\ell-1,0} & \cdots & L_{\ell-1,k-1} \end{bmatrix}.$$

Thus, we say that the linear graph $\mathcal{G}$ *computes* the matrix $\mathbf{L}$.

Cost. To measure the number of operations attached to a linear graph, we first make our computational model more precise: we count at unit cost multiplications by constants, as well as operations of the form $\alpha = \pm\beta \pm \gamma$. Then, we define the *cost* of $\mathcal{G}$ as the number

$$c(\mathcal{G}) := |\{e \in E \mid \lambda(e) \neq \pm 1\}| + |E| - |V| + k.$$

We claim that, with $\mathbf{a} = [a_0 \ \cdots \ a_{k-1}]^t$, the matrix-vector product $\mathbf{a} \mapsto \mathbf{La}$ can be computed using $c(\mathcal{G})$ operations. Indeed, along the edges, each multiplication by a constant different from ± 1 costs one operation, which add up to $|\{e \in E \mid \lambda(e) \neq \pm 1\}|$. Then, if the input of a vertex v consists of s edges, computing the value at v uses another $s - 1$ operations of the form $\pm\beta \pm \gamma$; summing over all v gives an extra $|E| - |V| + k$ operations.

Transposition. The *transposition principle* asserts that an algorithm performing a matrix-vector product can be transposed, producing an algorithm that computes the transposed matrix-vector product, in almost the same complexity as the original one. In our model, the transposition principle is easy to prove. If $\mathcal{G}$ is a linear graph with k inputs and ℓ outputs, that computes a matrix $\mathbf{L}$, we define the transposed graph $\mathcal{G}^t$ exchanging inputs and outputs and reversing the edges, without changing the weights. Theorem 13.10 in [5] proves that $\mathcal{G}^t$ computes the transposed matrix of $\mathbf{L}$; besides the cost $c(\mathcal{G}^t)$ is given by $c(\mathcal{G}^t) = c(\mathcal{G}) - k + \ell$.

3 Polynomial multiplication and its variants

In this section, we describe three variants of polynomial multiplication (plain, transposed and short product), and give algorithms for all of them. The algorithms we consider will be called "divide-and-conquer", following the terminology of [19]. The most well-known representatives of this class are due to Karatsuba and Toom, though many more exist.

3.1 Divide-and-conquer algorithms

A *divide-and-conquer* algorithm of parameters (k, ℓ), with $k < \ell$, is a triple $\mathcal{G} = (\mathcal{G}_A, \mathcal{G}_B, \mathcal{G}_C)$ of linear graphs such that $\mathcal{G}_A$ and $\mathcal{G}_B$ have k inputs and ℓ outputs, and $\mathcal{G}_C$ has ℓ inputs and $2k - 1$ outputs (other conditions follow).

Let $\mathbf{A} = (A_0, \ldots, A_{k-1})$ and $\mathbf{B} = (B_0, \ldots, B_{k-1})$ be indeterminates and let $L_0(\mathbf{A}), \ldots, L_{\ell-1}(\mathbf{A})$ and $M_0(\mathbf{B}), \ldots, M_{\ell-1}(\mathbf{B})$ be the linear forms computed by respectively $\mathcal{G}_A$ and $\mathcal{G}_B$. Let further $N_i = L_i M_i$ and let $P_0, \ldots, P_{2k-2}$ be the linear forms computed by $\mathcal{G}_C$. Then, the last conditions for $\mathcal{G}_A$, $\mathcal{G}_B$ and $\mathcal{G}_C$ to form a divide-and-conquer algorithm is that for $i = 0, \ldots, 2k-2$,

$$P_i(\mathbf{N}) = \sum_{0 \le j < k,\ 0 \le j' < k,\ j+j'=i} A_j B_{j'},$$

where $P_i(\mathbf{N})$ stands for the evaluation of the linear form P_i at $N_0, \ldots, N_{\ell-1}$. For instance, Karatsuba's algorithm has $k = 2, \ell = 3$ and

- $L_0 = A_0$, $L_1 = A_0 + A_1$, $L_2 = A_1$
- $M_0 = B_0$, $M_1 = B_0 + B_1$, $M_2 = B_1$
- $P_0(\mathbf{N}) = N_0$, $P_1(\mathbf{N}) = N_1 - N_0 - N_2$, $P_2(\mathbf{N}) = N_2$.

Other examples due to Toom [18] and Winograd [21] are in the last section; note that in these examples, $\mathcal{G}_A = \mathcal{G}_B$.

3.2 Plain multiplication

Let $\mathcal{G} = (\mathcal{G}_A, \mathcal{G}_B, \mathcal{G}_C)$ be a divide-and-conquer algorithm of parameters (k, ℓ). We now recall the well-known derivation of an algorithm for plain multiplication using $\mathcal{G}$; note that this formalism does not cover evaluations at points in $\mathsf{R}(x)$, which are useful e.g. over GF(2) [22].

Given n and A, B in $\mathsf{R}[x]_n$, we let $h = \lfloor (n+k-1)/k \rfloor$ and $h' = n - (k-1)h$, so that $h' \le h$. To make the algorithm simpler, we also want $h' > 0$; this will be the case as soon as $n > (k-1)^2$. Then, we write

$$A = A_0 + A_1 x^h + \cdots + A_{k-1} x^{(k-1)h}, \quad B = B_0 + B_1 x^h + \cdots + B_{k-1} x^{(k-1)h},$$

$$C = C_0 + C_1 x^h + \cdots + C_{2k-2} x^{(2k-2)h}.$$

In Algorithm 1 below, we use the notation slice(A, p, q) to denote the "slice" of A of length q starting at index p, that is, $(A \text{ div } x^p) \bmod x^q$. Note that $A_0, \ldots, A_{k-2}$ are in $\mathsf{R}[x]_h$ and A_{k-1} in $\mathsf{R}[x]_{h'}$; the same holds for the B_i; similarly, $C_0, \ldots, C_{2k-4}$ are in $\mathsf{R}[x]_{2h-1}$, C_{2k-3} in $\mathsf{R}[x]_{h+h'-1}$ and C_{2k-2} in $\mathsf{R}[x]_{2h'-1}$.

To obtain C, we compute the linear combinations L_i of $A_0, \ldots, A_{k-1}$ and M_i of $B_0, \ldots, B_{k-1}$, the products $N_i = L_i M_i$, and the polynomials C_i as the linear combinations $P_i(\mathbf{N})$. To handle the recursive calls, we need bounds e_i and f_i such that $\deg(L_i) < e_i$ and $\deg(M_i) < f_i$ holds: we simply take $e_i = h$ if $L_i \ne A_{k-1}$, and $e_i = h'$ if $L_i = A_{k-1}$; the same construction holds for f_i. For simplicity, we assume that $e_i = f_i$ for all i: this is e.g. the case when $\mathcal{G}_A = \mathcal{G}_B$. If $e_i \ne f_i$, the recursive calls need to be slightly modified, by e.g. doing a recursive call in length $\min(e_i, f_i)$ and an extra $O(n)$ operations to complete the product.

The cost $T(n)$ of this algorithm is $O(n^{\log_k(\ell)})$; one cannot easily give a more precise statement, since the ratio $T(n)/n^{\log_k(\ell)}$ does not have a limit as $n \to \infty$. We give here closed form expressions for n of the form k^i; in this case, we can go down the recursion until $n = 1$, which simplifies the estimates.

Algorithm 1 $\mathsf{Mul(A, B, n)}$

Require: A, B, n, with $\deg(A) < n$ and $\deg(B) < n$
Ensure: $C = AB$
1: **if** $n \le (k-1)^2$ **then**
2: **return** AB naive multiplication
3: $h = \lfloor (n+k-1)/k \rfloor$, $h' = n - (k-1)h$
4: **for** $i = 0$ **to** $k-2$ **do**
5: $A_i = \text{slice}(A, ih, h)$
6: $B_i = \text{slice}(B, ih, h)$
7: $A_{k-1} = \text{slice}(A, (k-1)h, h')$
8: $B_{k-1} = \text{slice}(B, (k-1)h, h')$
9: compute the linear combinations $L_0, \dots, L_{\ell-1}$ of $A_0, \dots, A_{k-1}$
10: compute the linear combinations $M_0, \dots, M_{\ell-1}$ of $B_0, \dots, B_{k-1}$
11: **for** $i = 0$ **to** $\ell - 1$ **do**
12: $N_i = \mathsf{Mul}(L_i, M_i, e_i)$
13: recover $C_0, \dots, C_{2k-2}$ as linear combinations of $N_0, \dots, N_{\ell-1}$
14: **return** $C = C_0 + C_1 x^h + \dots + C_{2k-2} x^{(2k-2)h}$.

The number of bilinear multiplications is ℓ^i. As to the linear operations, let c_A, c_B, c_C be the costs of $\mathcal{G}_A, \mathcal{G}_B, \mathcal{G}_C$. On inputs of length n, a quick inspection shows that we do $c_A n/k + c_B n/k + c_C(2n/k - 1)$ operations at steps 9, 10 and 13 and $2(k-1)(n/k-1)$ additions at step 14, for a total of $(c_A + c_B + 2c_C + 2k - 2)n/k - (c_C + 2k - 2)$. For $n = k^i$, summing over all recursive calls gives an overall estimate of

$$t(i) = (c_A + c_B + 2c_C + 2k - 2)(\ell^i - k^i)/(\ell - k) - (c_C + 2k - 2)(\ell^i - 1)/(\ell - 1). \quad (1)$$

3.3 Transposed product

If A is fixed, the map $A, B \mapsto AB$ becomes linear in B. The *transposed product* is the transposed map; applications include Newton iteration [9], evaluation and interpolation [3], etc.

If A is in $\mathsf{R}[x]_n$, multiplication-by-A maps $B \in \mathsf{R}[x]_n$ to $C = AB \in \mathsf{R}[x]_{2n-1}$. For $k \in \mathbb{N}$, we identify $\mathsf{R}[x]_k$ with its dual; then, the transposed product $A, C \mapsto B = CA^t$ maps $C \in \mathsf{R}[x]_{2n-1}$ to $B \in \mathsf{R}[x]_n$. Writing down the matrix of this map, we deduce the explicit formula [9, 3]

$$B = (C\widetilde{A} \text{ div } x^{n-1}) \bmod x^n,$$

where $\widetilde{A} = x^{n-1}A(1/x)$ is the reverse of A. This formula gives a quadratic algorithm for the transposed product; actually, any algorithm for the plain product can be used, by computing $C\widetilde{A}$ and discarding the unnecessary terms.

However, one can do better. As a consequence of the transposition principle, algorithms for the plain product yield algorithms for the transposed one, with only $O(n)$ cost difference: this was mentioned in [21], and developed further in [9] and [3]. However, none of the previous references gave an explicit form for the transposed version of divide-and-conquer algorithms, except for Karatsuba.

In Algorithm 2, we provide such an explicit form, on the basis of a divide-and-conquer algorithm $\mathcal{G}$ of parameters (k, ℓ). The polynomial A is subdivided as before, and the linear operations applied to the slices A_i are unchanged. The other input is now C; we apply to it the transposes of the operations seen in Algorithm 1, in the reverse order. Summing $C_0, \ldots, C_{2k-2}$ in Algorithm 1 becomes here the subdivision of C into $C_0, \ldots, C_{2k-2}$, using the degree information obtained in the previous section. Then, we follow the transposed graph $\mathcal{G}_C^t$ to obtain $N_0, \ldots, N_{\ell-1}$; we enforce the degree constraints $\deg(N_i) < 2e_i - 1$ by truncation (these truncations are the transposes of injections between some $\mathsf{R}[x]_{2e_i-1}$ and $\mathsf{R}[x]_{2e_j-1}$ that were implicit at step 13 of Algorithm 1). After this, we apply the algorithm recursively, follow the transposed graph $\mathcal{G}_B^t$ to obtain $B_0, \ldots, B_{k-1}$, and obtain B as the sum $B_0 + \cdots + B_{k-1}x^{(k-1)h}$.

Algorithm 2 $\mathsf{TranMul(A, C, n)}$

Require: A, C, n, with $\deg(A) < n$ and $\deg(C) < 2n - 1$
Ensure: $B = CA^t$
1: **if** $n \leq (k-1)^2$ **then**
2:     **return** CA^t naive transposed multiplication
3: $h = \lfloor (n + k - 1)/k \rfloor$, $h' = n - (k-1)h$
4: **for** $i = 0$ **to** $k - 2$ **do**
5:     $A_i = \mathrm{slice}(A, ih, h)$
6: $A_{k-1} = \mathrm{slice}(A, (k-1)h, h')$
7: **for** $i = 0$ **to** $2k - 4$ **do**
8:     $C_i = \mathrm{slice}(C, ih, 2h - 1)$
9: $C_{2k-3} = \mathrm{slice}(C, (2k-3)h, h + h' - 1)$
10: $C_{2k-2} = \mathrm{slice}(C, (2k-2)h, 2h' - 1)$
11: compute the linear combinations $L_0, \ldots, L_{\ell-1}$ of $A_0, \ldots, A_{k-1}$
12: compute the transposed linear combinations $N_0, \ldots, N_{\ell-1}$ of $C_0, \ldots, C_{2k-2}$, with N_i truncated modulo x^{2e_i-1}
13: **for** $i = 0$ **to** $\ell - 1$ **do**
14:     $M_i = \mathsf{TranMul}(L_i, N_i, e_i)$
15: compute the transposed linear combinations $B_0, \ldots, B_{k-1}$ of $M_0, \ldots, M_{\ell-1}$, with B_{k-1} truncated modulo $x^{h'}$
16: **return** $B_0 + \cdots + B_{k-1}x^{(k-1)h}$

The cost $T'(n)$ of this algorithm is still $O(n^{\log_k(\ell)})$. Precisely, let c_A, c_B, c_C be the costs of $\mathcal{G}_A, \mathcal{G}_B, \mathcal{G}_C$, and consider the case where $n = k^i$. The number of bilinear multiplications does not change compared to the direct version. As to linear operations, the cost of step 12 is $(c_C - \ell + 2k - 1)(2n/k - 1)$ and that of step 15 is $(c_B - k + \ell)n/k$. After simplification and summation, we obtain that for $n = k^i$, the overall number of linear operations is now

$$t'(i) = (c_A + c_B + 2c_C + 3k - \ell - 2)(\ell^i - k^i)/(\ell - k) - (c_C + 2k - \ell - 1)(\ell^i - 1)/(\ell - 1).$$

With $t(i)$ given in Eq. (1), we obtain $t'(i) - t(i) = k^i - 1 = n - 1$, as implied by the transposition principle: the transposed algorithm uses $n - 1$ more operations.

3.4 Short product

The *short product* is a truncated product: to A, B in $\mathsf{R}[x]_n$, it associates $C = AB \bmod x^n \in \mathsf{R}[x]_n$; it was introduced and described in [14, 10], and finds a natural role in many algorithms involving power series operations, such as those relying on Newton iteration [4]. The situation is similar to that of the transposed product: the previous references describe Karatsuba's version in detail, but hardly mention other algorithms in the divide-and-conquer family. Thus, as for transposed product, we give here an explicit version of the short product algorithm, starting from a divide-and-conquer algorithm $\mathcal{G}$ of parameters (k, ℓ).

For Karatsuba's algorithm, two strategies exist in the literature; the latter one, due to [10], extends directly to the general case. Instead of slicing the input polynomials, we "decimate" them: for $A \in \mathsf{R}[x]_n$, we write $A = \sum_{i<k} A_i(x^k)x^i$ (the same holds for B). Here, the polynomial A_i belongs to $\mathsf{R}[x]_{h_i}$, with $h_i = \lfloor (n+k-1-i)/k \rfloor$; we denote it by $A_i = \mathrm{decimation}(A, i, h_i)$. Then, with $C_i = \sum_{j+j'=i} A_j B_{j'}$, we deduce

$$C = \sum_{i<2k-1} C_i(x^k)\, x^i = \sum_{i<k-1} (C_i + xC_{i+k})(x^k)\, x^i + C_{k-1}(x^k)\, x^{k-1}.$$

We compute the linear combinations L_i of $A_0, \ldots, A_{k-1}$ and M_i of $B_0, \ldots, B_{k-1}$, the products $N_i = L_i M_i$, and finally C_i using the linear forms $P_0, \ldots, P_{2k-2}$. We need to compute C_i modulo x^{h_i}. For $i < \ell$, let thus i' be the largest index such that the product $L_i M_i$ appears with a non-zero coefficient in the linear form $P_{i'}$ (this depends on the divide-and-conquer algorithm), and let $g_i = h_{i'}$ Since the h_i form a decreasing sequence, it suffices to compute $L_i M_i \bmod x^{g_i}$.

These steps are summarized in Algorithm 3, where we reuse the notation introduced above. Here, it suffices that $n \geq k$ to ensure that all h_i, and thus all g_i, are positive, since smallest is $h_{k-1} = \lfloor n/k \rfloor$. As for the previous algorithms, the cost is $O(n^{\log_k(\ell)})$; however, the precise analysis is much more delicate [10], so we do not give any closed-form estimate here, even for n of the form k^i.

4 Code generation

The algorithms given in the previous section all share the same shape; they only depend on the datum of a divide-and-conquer algorithm $\mathcal{G}$, that is, three linear graphs. We wrote a java program that inputs such graphs and generates C implementations; we describe it here.

Coefficient arithmetic. We focus on coefficient types that can be represented using machine data types: polynomials with `double` coefficients and polynomials with coefficients in $\mathbb{Z}/p\mathbb{Z}$, where p is an integer (typically a prime) that fits in a machine word; p is not known in advance (the mpfq library [8] is able to exploit possible prior knowledge of p).

In the first case, due to cancellations, operations on `double`s do not satisfy the ring axioms. Nevertheless, we support this type, since we want to compare the running times between `double` and modular coefficients, and to measure to what extent divide-and-conquer algorithms suffer from precision loss.

Algorithm 3 ShortMul(A, B, n)

Require: A, B, n, with $\deg(A) < n$, $\deg(B) < n$
Ensure: $C = AB \bmod x^n$
1: **if** $n = 1$ **then**
2:     **return** AB
3: **for** $i = 0$ **to** $k - 1$ **do**
4:     $A_i = \text{decimation}(A, i, h_i)$
5:     $B_i = \text{decimation}(B, i, h_i)$
6: compute the linear combinations $L_0, \ldots, L_{\ell-1}$ of $A_0, \ldots, A_{k-1}$
7: compute the linear combinations $M_0, \ldots, M_{\ell-1}$ of $B_0, \ldots, B_{k-1}$
8: **for** $i = 0$ **to** $\ell - 1$ **do**
9:     $N_i = \mathsf{ShortMul}(L_i \bmod x^{g_i}, M_i \bmod x^{g_i}, g_i)$
10: compute the linear combinations $C_0, \ldots, C_{2k-2}$ of $N_0, \ldots, N_{\ell-1}$, truncating C_i modulo x^{h_i}
11: **return** $C = \sum_{i=0}^{k-2} (C_i + xC_{i+k})(x^k)\, x^i + C_{k-1}(x^k)\, x^{k-1}$

In the second case, we use `unsigned long`s. Since our implementations are all done on 64 bit platforms (Intel Core2 or AMD 64), long machine words can hold up to 64 bits (we will actually slightly reduce this bound, for reasons explained later). The implementation of operations modulo p follows well-known recipes; we recall some of them here.

- The addition $c = a + b \bmod p$ is done by computing $c' = a + b - p$; if it is negative, we add p to it. This is done by using the sign bit of c' as a mask [17], using shifts, `and`s and additions. The same trick is used for subtraction and multiplication by small constants (used in the linear combination steps).
- Multiplications are done using Montgomery's algorithm [13].
- The algorithms may do divisions by constants in the linear combination steps; division by α is done by computing $\beta = 1/\alpha$ modulo p and multiplying by β modulo p. Division by 2 receives a special treatment: writing $p = 2q+1$, we obtain that $p - q = 1/2 \bmod p$. Thus, for an integer $a = 2u + v$, with $v \in \{0, 1\}$, $a/2 \bmod p$ is given by $u + v(p - q) \bmod p$.
- Some algorithms use roots of unity of low order (e.g., $\sqrt{-1}$) when p allows it.

Input and output. On input a triple of matrices, the code generator produces C code for plain, transposed, short multiplication, as well as two related operations, square (where both inputs are the same, so some savings are possible) and short square (similar, but with the same truncation as in the short product). Suppose for instance that we consider the Karatsuba algorithm; the code generator takes as input the following matrices:

```
0 0 0 0 0      0 0 0 0 0      0  0  0 0 0 0 0
0 0 0 0 0      0 0 0 0 0      0  0  0 0 0 0 0
1 0 0 0 0      1 0 0 0 0      0  0  0 0 0 0 0
1 1 0 0 0      1 1 0 0 0      1  0  0 0 0 0 0
0 1 0 0 0      0 1 0 0 0      -1 1 -1 0 0 0 0
                              0  0  1 0 0 0 0
```

For a given multiplication type (plain, transposed or short) and a given data type, we produce several functions: a top-level function, which allocates some workspace and does some precomputations (e.g., the modular inverses or roots of unity needed for the linear combinations), the main recursive function, and functions for the linear combinations.

Memory management. Intermediate results are stored in temporary memory, in successive slots of length either $\lceil n/k \rceil$ or $2\lceil n/k \rceil$. At code generation, when we determine that a memory area can be reused, we reuse it. We can thus determine how much workspace will be needed in a single call to the main recursive function in length n. In general, if the call in length n uses $rn+s$ space, the *total* amount will be $rn + rn/k + \cdots + s\log_k(n) \le rkn/(k-1) + s\log_k(n)$. This memory is allocated by the top-level function. Efforts are made to avoid using too much memory, similarly to what one would do when writing the code by hand. When an output of the linear combinations aliases an input, we reuse the input in all other operations (e.g., for Karatsuba, the linear combinations are $L_0 = a_0, L_1 = a_0 + a_1, L_2 = a_1$: no copy is made and only an addition takes place).

Naive product. We implemented naive algorithms (for plain, transposed and short products), for degrees up to 16. Our code for this case is generated automatically as well, so as to unroll loops, since the compiler was not doing a very good job by itself. We do not perform modular reduction after each step: we first compute the whole result without any reduction, and apply the reduction in the end. In degree $< n$, this reduces the number of reductions from n^2 to n. However, this slightly reduces the possible size of the modulus: only 60-bit modulus can now be used. No assembly code was used: using gcc's custom `__uint128_t` type, we obtained code of satisfying quality after compilation.

5 Experiments

We finally give the results of experiments an Intel Core2 Duo CPU T7300 with 4Gb RAM, set to 800Mhz clock speed. The timings are in seconds, for 500 repetitions of the same computation. Our experiments use Karatsuba's algorithm and its generalizations by Toom, of parameters $(k, 2k-1)$: for the standard evaluation points $(0, \pm1, \pm2, \pm1/2, \ldots, \infty)$, we use linear graphs from [1]. Computing modulo p, with $p = 4r+1$, we wrote a version of Toom's algorithm (called i-Toom below) that evaluates at $(0, \pm1, \pm\sqrt{-1}, 2, \infty)$ using FFT techniques in size 4. We also use a less known algorithm of parameters $(3, 6)$ due to Winograd, with only additions and subtractions in its linear combinations [21, Ch. IVc]. Complexity predicts that it should be slower than Karatsuba, but the simple structure of the linear combinations made it worthwhile to experiment with.

Comparison between divide-and-conquer algorithms. Figure 1 compares the algorithms of Karatsuba, Toom ($k = 3$) and Winograd, for plain product, using `unsigned long`s (transposed and short products behave similarly). As predicted, Winograd's algorithm does not perform very well. More surprisingly, Toom's algorithm appears useful for most degrees (examples using other divide-and-conquer algorithm are given below). Jumps appear for all algorithms; these

are due to crossing degree thresholds determined by both the parameter k of the graphs, and the threshold for the switch to the naive algorithm: increasing the latter smooths the curves noticeably. Finally, profiling using Valgrind shows that in all cases, 65% to 70% is the time is spent in the naive algorithm.

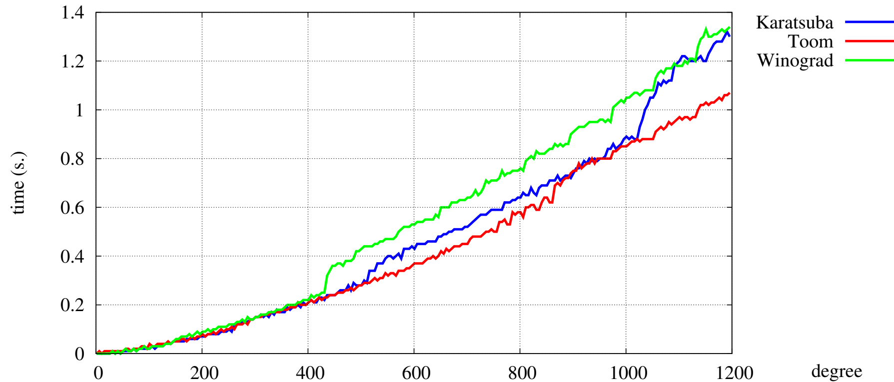

Fig. 1. Comparison between divide-and-conquer algorithms

Comparison between multiplication types. Figure 2 compares plain, transposed and short product, square and short square, with `unsigned long`s and Toom's algorithm ($k = 3$); the results for other divide-and-conquer algorithms are similar. The transposed product is faster than its plain counterpart, even if operation count predicts it should be slightly slower. Indeed, in the naive transposed product, fewer modular reduction are needed than in the plain one (since the output is twice as short); this is not accounted for in our model and seems to explain the savings. The time for a short product is about 60% to 70% that of a plain product, as in [10] for Karatsuba. The square product and the short square are faster than their non-square counterparts, but not by much.

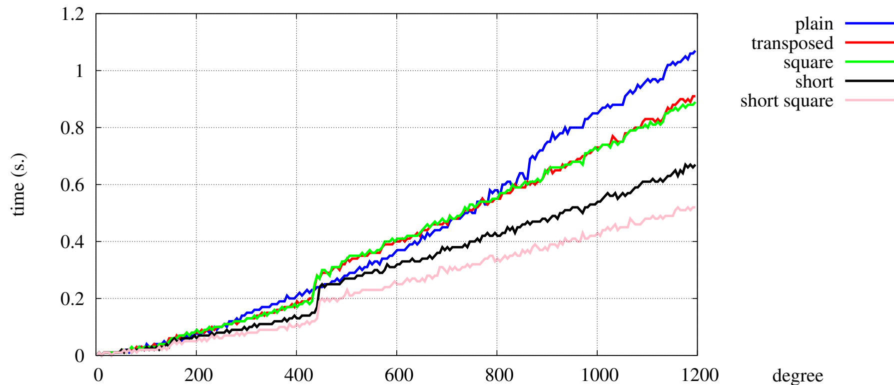

Fig. 2. Comparison between multiplication types

Comparison with other systems. For primes of size 60 bit, the library NTL v5.5 [16] and the computer algebra system Magma v2.15-6 [2] are the fastest implementations known to us. We use the two available representations for NTL, `lzz_p` and `ZZ_p` (our 60 bit primes are too large for the former, so we used 52

bit primes in that case). Figure 3 gives running times, where our code uses "standard" Toom multiplication for $k = 3$ or $k = 4$ or i-Toom for $p = 4r + 1$. Even though some other implementations use asymptotically faster algorithms (the staircases indicate FFT multiplication), our code performs better in these degree ranges. From degree 10000 on, Toom's algorithm with $k = 5$ is the best of our divide-and-conquer algorithms, but does no better than NTL's FFT.

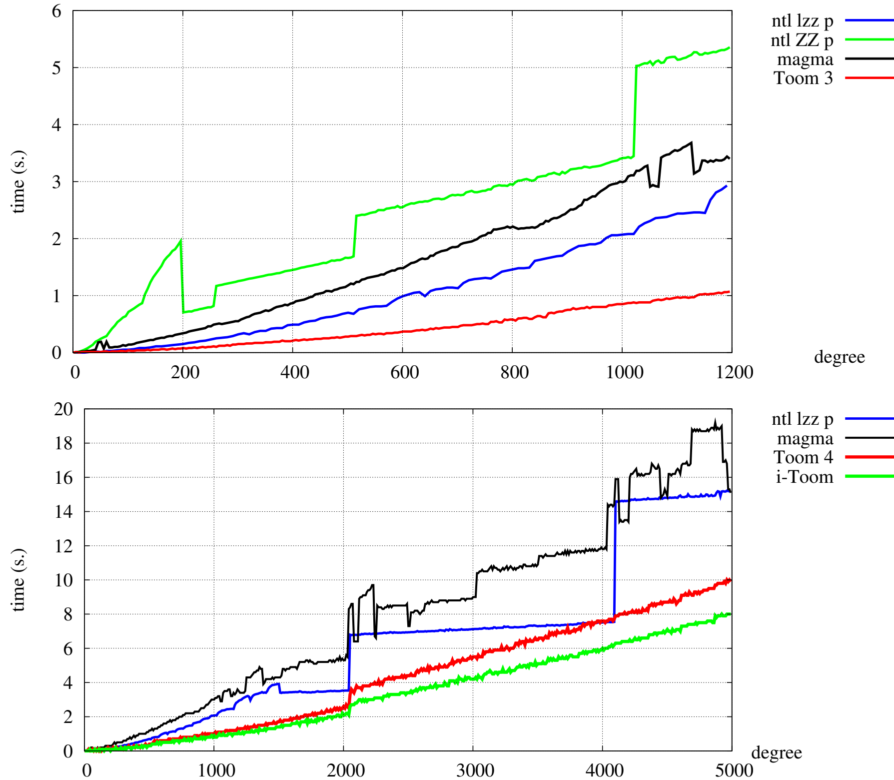

Fig. 3. Comparison with other systems

Comparison between data types. Operations with `double` coefficients are faster than with `unsigned long`s, but only by a factor of about 1.7 to 1.9 (for all variants, and for all divide-and-conquer algorithms). Divide-and-conquer algorithms do poorly in terms of precision with `double` coefficients: for useful kinds of inputs (such as solutions of ODE's), the cancellation errors make results unusable for degrees from 50 on.

6 Conclusion

Our approach offers several advantages: after paying the small price of writing the code generator, it becomes straightforward to experiment various divide-and-conquer algorithms, test optimizations, etc. Also, we now have general versions of transposed and short product. For plain products, performance is comparable to,

and actually better than, that of software using FFT multiplication in significant degree ranges. For short products, our advantage is actually higher, since it is rather difficult to obtain an efficient short product using FFT multiplication.

Acknowledgments. We acknowledge the support of the Canada Research Chairs Program, of the MITACS MOCAA project and of NSERC, and thank the referees for their helpful comments.

References

1. M. Bodrato and A. Zanoni. Integer and polynomial multiplication: towards optimal Toom-Cook matrices. In *ISSAC'07*, pages 17–24. ACM, 2007.
2. W. Bosma, J. Cannon, and C. Playoust. The Magma algebra system. I. The user language. *J. Symbolic Comput.*, 24(3-4):235–265, 1997.
3. A. Bostan, G. Lecerf, and É. Schost. Tellegen's principle into practice. In *ISSAC'03*, pages 37–44. ACM, 2003.
4. R. P. Brent and H. T. Kung. Fast algorithms for manipulating formal power series. *J. ACM*, 25(4):581–595, 1978.
5. P. Bürgisser, M. Clausen, and M. A. Shokrollahi. *Algebraic complexity theory*, volume 315 of *Grund. Math. Wissen.* Springer-Verlag, 1997.
6. D. G. Cantor and E. Kaltofen. On fast multiplication of polynomials over arbitrary algebras. *Acta Informatica*, 28(7):693–701, 1991.
7. J. Cooley and J. Tukey. An algorithm for the machine calculation of complex Fourier series. *Mathematics of computation*, 19, 1965.
8. P. Gaudry and E. Thomé. The mpFq library and implementing curve-based key exchanges. In *SPEED*, pages 49–64, 2007.
9. G. Hanrot, M. Quercia, and P. Zimmermann. The middle product algorithm. I. *Appl. Algebra Engrg. Comm. Comput.*, 14(6):415–438, 2004.
10. G. Hanrot and P. Zimmermann. A long note on Mulders' short product. *J. Symb. Comput.*, 37(3):391–401, 2004.
11. A. Karatsuba and Y. Ofman. Multiplication of multidigit numbers on automata. *Soviet Math. Dokl.*, 7:595–596, 1963.
12. M. Monagan and R. Pearce. Parallel sparse polynomial multiplication using heaps. In *ISSAC'09*. ACM, to appear.
13. P. L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44(170):519–521, 1985.
14. T. Mulders. On short multiplications and divisions. *Appl. Algebra Engrg. Comm. Comput.*, 11(1):69–88, 2000.
15. A. Schönhage and V. Strassen. Schnelle Multiplikation großer Zahlen. *Computing*, 7:281–292, 1971.
16. V. Shoup. A library for doing number theory. `http://www.shoup.net/ntl/`.
17. V. Shoup. A new polynomial factorization algorithm and its implementation. *J. Symb. Comp.*, 20(4):363–397, 1995.
18. A. Toom. The complexity of a scheme of functional elements realizing the multiplication of integers. *Doklady Akad. Nauk USSR*, 150(3):496–498, 1963.
19. J. van der Hoeven. Relax, but don't be too lazy. *J. Symbolic Comput.*, 34(6):479–542, 2002.
20. J. von zur Gathen and J. Gerhard. *Modern computer algebra.* Cambridge University Press, Cambridge, second edition, 2003.
21. S. Winograd. *Arithmetic complexity of computations*, volume 33 of *CBMS-NSF Regional Conference Series in Applied Mathematics*. SIAM, 1980.
22. P. Zimmermann. irred-ntl patch. `http://www.loria.fr/zimmerma/irred/`.