Report

On computing short products

Author(s):
Mulders, Thom

Publication Date:
1997

Permanent Link:

https://doi.org/10.3929/ethz-a-006652192 →

Technical Report No. 276, Department of Computer Science, ETH Zurich
November 1997

ON COMPUTING SHORT PRODUCTS

THOM MULDERS
INSTITUTE OF SCIENTIFIC COMPUTING
ETH ZURICH, SWITZERLAND
MULDERS@INF.ETHZ.CH

Abstract. A polynomial consisting of only the low degree monomials of the (full) product of two univariate polynomials f and g is called a short product of f and g. A global algorithm, independent of the actual multiplication algorithm used, to compute short products is proposed. Its performance for several multiplication algorithms is studied. Also several applications of short products are pointed out.

1. Introduction

In [10] efficient algorithms for multiprecision floating point multiplication are developed. The main idea of these algorithms is the fact that, in order to perform such a multiplication, it is not necessary to first perform the full exact multiplication. In fact the products of the least significant digits do (in general) not contribute to the final result.

A similar situation occurs when one is computing in the residue ring $S = R[x]/(x^N)$ for some ring R and positive integer N. When $F, G \in S$ are represented by $f, g \in R[x]$ (both of degree $< N$), one could compute a representative $h \in R[x]$ (of degree $< N$) for FG by first computing fg and subsequently deleting the terms of degree $\geq N$. Of course one sees immediatly that the representative h can be taken to be

$$\sum_{k=0}^{N-1} \left(\sum_{i+j=k} f_i g_j \right) x^k, \tag{1}$$

where f_i (resp. g_i) denotes the coefficient of x^i of f (resp. g), so the products of high degree coefficients do not contribute to the final result.

In this paper we will give a global algorithm to compute expressions like (1) which will make use of any multiplication algorithm to compute full products. Furthermore, its performance for several multiplication algorithms is studied and optimal algorithms are derived. Finally, we will give some applications of short products and the gain which is yielded by the new algorithm.

We must stress that most of the material presented is also applicable to multiprecision integers. However, due to the phenomenon of carries, when multiplying integers, the details are more complicated in that case (see for example [10]). For that reason we restrict ourselves to the polynomial case, since that will suffice to expose the main ideas.

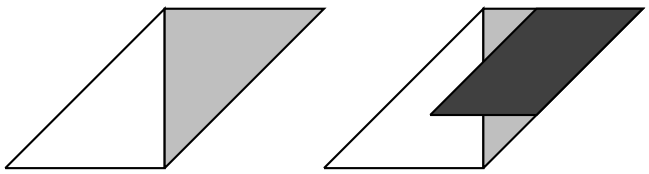

FIGURE 1. Computing a short product (case 1)

2. Computing short products (case 1)

From now on let R be any ring. For polynomials $f, g \in R[x]$ and a positive integer N we define the short product (notation similar to [10]):

$$f \times_N g = \sum_{k=0}^{N-1} \left(\sum_{i+j=k} f_i g_j \right) x^k. \tag{2}$$

It is clear how to compute a short product, and how it compares to full multiplication, when we use the classical polynomial multiplication algorithm. Now we will also investigate other polynomial multiplication algorithms.

We will first look at the problem of computing the short product $f \times_N g$ of polynomials f and g of length N (the length of a polynomial is its degree plus 1). The multiplication of f and g using the classical multiplication algorithm can be depicted by the left picture of figure 1 and the coefficient products contributing to $f \times_N g$ can be indicated by the shaded area in this picture.

The right picture of figure 1 is a pictorial representation of the algorithm we propose to compute $f \times_N g$. Compute the full product of the 'lower parts' of f and g (the dark area in the picture) and compute 2 short products of polynomials of lower degree (the 2 shaded areas in the picture). Then add appropriate terms to get the final result. For the computation of the full products we can take any multiplication algorithm. Notice that in practice we will use classical multiplication when the length of the polynomials gets smaller than some treshold d which depends on the multiplication algorithm and implementation used.

We will now investigate how the complexity of this algorithm compares to full multiplication. For this we will only look at multiplications in R, assuming that these form the major contribution to the complexity. For a specific implementation it may be worthwhile to take also additions into account, in order to optimize the program.

Let $M(N)$ denote the number of multiplications in R needed to multiply two polynomials of length N using some polynomial muliplication algorithm M. Furthermore let $S_{M,\beta}(N)$ ($\beta \geq 1/2$) denote the number of multiplications in R needed when performing our algorithm using M for the full multiplications and taking the terms of degree $< \beta N$ as 'lower parts'. We then get the following formula:

$$S_{M,\beta}(N) = M(\beta N) + 2S_{M,\beta}((1-\beta)N). \tag{3}$$

We will now compute β minimizing $S_{M,\beta}$ for different M. For this we will look at the asymptotic case (N very big).

First we assume that $M(N) = N^\alpha$ (we leave out a possible constant factor). This includes classical multiplication ($\alpha = 2$), Karatsuba multiplication ($\alpha = \log(3)$) and its generalizations ($\alpha = \log(2r+1)/\log(r+1)$, $r \geq 1$).(See [8] 4.3.3, [12] 1.3.5)

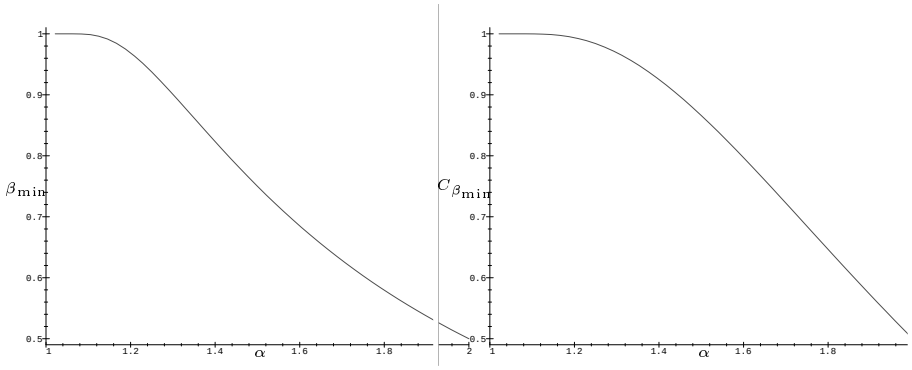

FIGURE 2. $\beta_{\min}$ and $C_{\beta_{\min}}$ for $1 \leq \alpha \leq 2$

By expanding formula 3 further, using $M(aN) = a^\alpha M(N)$, we get

$$\begin{aligned} S_{M,\beta}(N) &= (\beta^\alpha + 2\beta^\alpha(1-\beta)^\alpha + 4\beta^\alpha(1-\beta)^{2\alpha} + \cdots)M(N) \\ &\leq \frac{\beta^\alpha}{1-2(1-\beta)^\alpha} M(N) \\ &= C_\beta M(N) \end{aligned}$$

Now C_β is minimal for $\beta_{\min} = 1 - (1/2)^{(1/(\alpha-1))}$. Figure 2 gives $\beta_{\min}$ and $C_{\beta_{\min}}$ for all $1 \leq \alpha \leq 2$.

1. Classical multiplication ($\alpha = 2$):

   We have $\beta_{\min} = 1/2$ and $C_{\beta_{\min}} = 1/2$, so we gain 50%, which is not very surprising. Notice that an implementation of the short product computing only the products needed and adding them appropriately, will need the same number of multiplications but will in general be faster than our algorithm due to less overhead.

2. Karatsuba multiplication ($\alpha = \log(3)$):

   We have $\beta_{\min} \approx 0.694$ and $C_{\beta_{\min}} \approx 0.808$, so we gain about 19%. In [10] this algorithm is also proposed using $\beta = 1/2$. However, since $C_{1/2} = 1$ this algorithm wouldn't gain anything in the asymptotic case. In table 1 we compare the performence of the algorithm in the non-asymptotic case using $\beta = 1/2$ and $\beta = 0.694$. Here we list N/d (where N is the length and d is the treshold), u (the recursion-depth of the algorithm) and $C_\beta(u) = \beta^\alpha + 2\beta^\alpha(1-\beta)^\alpha + \cdots + 2^u\beta^\alpha(1-\beta)^{2u}$.

   We see that, except for $1 < N/d < 8$ and $11 \leq N/d < 16$, $C_{0.694} < C_{1/2}$. Here we have not taken into account the work for computing short products when $N \leq d$. For small N/d this might be an advantage for one or the other β, depending on N/d, d and the implementation. We see however that also in the non-asymptotic case the choice $\beta = 0.694$ is in general better than $\beta = 1/2$.

   Remark: A. Schönhage pointed out to the author that in [12] there is an exercise (page 35, exercise 17) concerning short products using Karatsuba. The algorithm proposed above was found independently of this exercise. The authors of the exercise had also this algorithm in mind but never published their solution.

	$\beta = 1/2$		$\beta = 0.694$	
N/d	u	$C_\beta(u)$	u	$C_\beta(u)$
2	0	.333	0	.561
4	1	.555	0	.561
8	2	.704	0	.561
11	2	.704	1	.732
16	3	.802	1	.732
32	4	.868	1	.732
35	4	.868	2	.785
64	5	.912	2	.785
115	5	.912	3	.801
128	6	.941	3	.801
256	7	.961	3	.801
375	7	.961	4	.806
512	8	.974	4	.806
1024	9	.983	4	.806
1224	9	.983	5	.807

TABLE 1. Comparison of $\beta = 1/2$ and $\beta = 0.694$ in the non-asymptotic case

N/d	$\beta_{\min}$	$S_{F,\beta_{\min}}(N)/M(N)$
50	0.829	0.933
100	0.902	0.964
300	0.953	0.985
500	0.966	0.990
1000	0.977	0.994

TABLE 2. Optimal values for β

3. Generalizations of Karatsuba multiplication ($\alpha = \log(2r+1)/\log(r+1)$):
   For increasing r we have that α approaches 1, in which case also $C_{\beta_{\min}}$ approaches 1. So the gain of our algorithm compared to full multiplication decreases for increasing r. Notice that these algorithms become impractical for larger values of r due to their large overhead (see also [8] and [12]).

Now we will assume that $M(N) = N\log(N)$. This includes the algorithms described in [3], [7] and [11].

By expanding formula 3 further we get

$$\begin{aligned} S_{M,\beta}(N) &= \beta N\log(\beta N) + 2\beta(1-\beta)N\log(\beta(1-\beta)N) + \\ &\quad 4\beta(1-\beta)^2 N\log(\beta(1-\beta)^2 N) + \cdots \end{aligned}$$

Now we cannot simply take this sum to infinity since the logarithms will become eventually negative. We therefore sum only up to the term $2^u\beta(1-\beta)^u N\log(\beta(1-\beta)^u N)$ where u is the depth of the algorithm (i.e. $(1-\beta)^{u+1}N \leq d$). Then, for several N/d, we plot $S_{F,\beta}(N)/M(N)$ and determine for which β this quotient is minimal. We get the values in table 2.

We see that when using these multiplication algorithms it is not worthwhile to use our algorithm. In the range where these algorithms are useful (i.e. large N) the gain of our algorithm is very small. This could also be deduced from figure 2, by replacing $N\log(N)$ by N^α for α close to 1.

2.1. **Implementation.** First we have to make a remark on the relation between theory and practice. The formula $M(N) = N^{\log(3)}$ for Karatsuba multiplication, used to compute the optimal β, is only an approximation to the actual cost. For some lengths N the formula is more accurate than for others. For that reason it may happen that there are more favourable subdivisions of the problem than the theory would predict. This is heavily dependent on the actual length N. For that reason we determine the actual values of parameters and tresholds in our implementations by experiment. This also gives not the best values in all cases but only a global optimum. Another advantage of this method is the fact that it takes the overhead of the algorithm into account.

We have implemented the algorithm in Aldor (previously $A^\#$) as part of the Σ^{IT}–library ([2]), using the built-in multiplication algorithms, i.e. classical and Karatsuba multiplication.

In the case of Karatsuba multiplication, experiments show that the best choice for β depends on N and that $0.68 \leq \beta \leq 0.71$ is the best in general. So we see that our theoretical model was not bad at all. We have chosen $\beta = 0.7$ and the treshold d to be twice the treshold for full multiplication.

In table 3 we list the timings of some experiments on a single CPU DEC Alpha 9000/300 with 96Mb main memory, running Digital Unix V4.0. The polynomials used have integer coefficients, in absolute value smaller than 100 and the timings are given in milliseconds (garbage collection included). The columns have the following meaning:

- N: the length of the polynomials.
- K: computing the full product using Karatsuba multiplication.
- SK1: computing the short product using Karatsuba multiplication with $\beta = 0.7$.
- IK1: improvement when using Karatsuba multiplication with $\beta = 0.7$.
- SK2: computing the short product using Karatsuba multiplication and $\beta = 1/2$.
- IK2: improvement when using Karatsuba multiplication with $\beta = 1/2$.
- C: computing the full product using classical multiplication.
- SC: computing the short product using classical multiplication.
- IC: improvement when using classical multiplication.

We see that we get indeed a gain of approximately 19% when using Karatsuba multiplication with $\beta = 0.7$ and a gain of 50% when using classical multiplication. The differences in the gain for Karatsuba multiplication can be explained by the remark at the beginning of this section.

Also we see that when using Karatsuba multiplication with $\beta = 1/2$, the gain decreases for increasing N.

N	K	SK1	IK1	SK2	IK2	C	SC	IC
50	33	26	21%	30	9%	73	46	37%
100	103	83	19%	90	13%	293	160	45%
200	303	243	20%	276	9%	1096	593	46%
400	946	736	22%	863	9%	4443	2246	49%
800	2746	2256	18%	2683	2%	17276	8706	50%
1600	8430	7216	14%	8513	-1%	69543	34820	50%
3200	24233	20516	15%	24734	-2%	276450	141034	49%
6400	74767	60817	19%	73850	1%	1113733	569033	49%
12800	230400	181150	21%	224300	3%	4269100	2302350	46%

TABLE 3. Timings for polynomials having integer coefficients

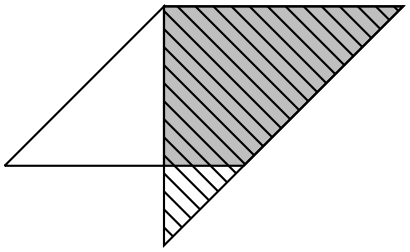

FIGURE 3. Computing a short product (case 2), alternative 1

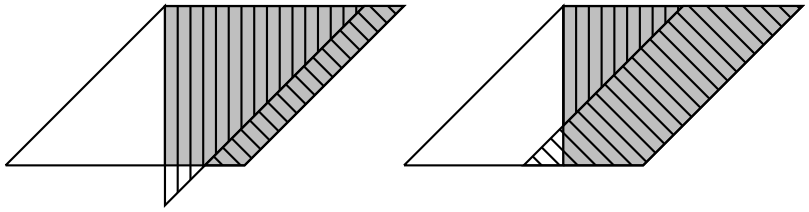

FIGURE 4. Computing a short product (case 2), alternative 2

3. Computing short products (case 2)

In this section we will look at the problem of computing the short product $f \times_N g$ of polynomials f of length N and g of length $\leq N$. This can be indicated by the shaded area in figure 3.

Of course we can apply our algorithm of the previous section by considering g as a polynomial of length N. This can be depicted by the lined area in figure 3. Doing this will not be as costly as when g would have had length N since eventually polynomials to be multiplied will become 0 (the lined but not shaded area in figure 3). In the case of classical multiplication this works fine and is this the way we've implemented it.

Another alternative is depicted in figure 4. Here we split f into parts f_1 and $f2$ and compute the full product of f_1 with g and another short product. For this there are two possibilities as figure 4 shows. In fact all three possibilities are different instances of the same method, depending on the choice of the length of f_1.

	l
$1 \leq u < 1.3$	0
$1.3 \leq u < 2.3$	1
$2.3 \leq u < 3.3$	2
$3.3 \leq u < 4.0$	3

TABLE 4. Length of f_1

N	K	SK1	IK1	C	SC	IC
850	3050	2634	14%	18216	10800	40%
1000	3667	3183	13%	22100	14517	34%
1300	5416	4066	25%	27467	20283	26%
1800	6334	6134	3%	38783	31850	18%
1900	7100	6233	12%	40417	34300	15%
2200	8417	7167	15%	45767	41300	10%
3000	13117	10983	16%	62967	58833	7%

TABLE 5. Timings for polynomials having integer coefficients

In the case of Karatsuba multiplication we could use the formula $M(N) = N\log(N)$ and derive formulae to compute the optimal length of f_1. This theoretical optimum is however not optimal in practice (see remark at beginning of section 2.1). Experiments show that the most efficient choice for the length of f_1 is a multiple of the length of g, a consequence of the preference of Karatsuba for equal–length polynomials. Depending on the length of f compared to the length of g one has to choose the length of f_1.

Since when the length of f is large compared to the length of g the short product is almost all of the full product, we compute the full product in that case.

3.1. **Implementation.** In the case of classical multiplication we compute the short product in case 2 by considering g as a polynomial of higher degree (alternative 1). In the case of Karatsuba multiplication, when the length of f is u times the length of g, we choose (guided by some experiments) the length of f_1 to be l times the length of g where l is as in table 4. When $u \geq 4$ we compute the full product of f and g.

In table 5 we list the timings of some experiments using our code. Here N is the length of f, while we take the length of g to be 800. The other columns have the same meaning as in table 3.

4. Computing short products (general case)

In this section we will look at the problem of computing the short product $f \times_N g$ of polynomials f and g of arbitrary length. This can be indicated by the shaded area in figure 5. We assume that f is at least as long as g.

In this case we simply consider f to be of length N and apply the algorithm of case 2. An instance of this is indicated by the two lined areas in figure 5. Again this is not as costly as when f would have had length N since eventually polynomials to be multiplied will be 0.

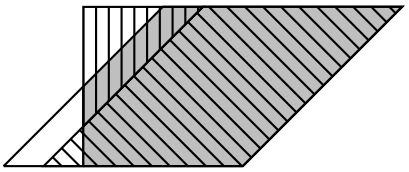

FIGURE 5. Computing a short product (general case)

N_1	N	K	SK1	IK1	C	SC	IC
800	800	2862	2375	17%	16533	9417	43%
800	880	2866	2741	4%	16533	10500	36%
800	920	2862	2891	-1%	16533	11900	28%
800	1200	2866	2862	0%	16533	15266	8%
1600	1600	5912	5500	7%	33517	27316	18%
1600	1680	5912	5700	4%	33517	28783	14%
1600	1720	5912	5850	1%	33517	30784	8%
1600	2000	5912	5912	0%	33517	33850	-1%
2000	2000	7512	6512	13%	41534	35317	15%
2000	2080	7512	6712	11%	41534	36767	11%
2000	2120	7512	6841	9%	41534	39666	4%
2000	2400	7512	7512	0%	41534	41684	0%
2400	2400	9570	8400	12%	50700	45150	11%
2400	2480	9570	8804	8%	50700	47283	7%
2400	2520	9570	8966	6%	50700	48216	5%
2400	2800	9570	9570	0%	50700	51650	-2%

TABLE 6. Timings for polynomials having integer coefficients

Let the difference of N and the length of f be w times the length of g. When w gets closer to 1 the short product is almost all of the full product, in which case we simply compute the full product. For what w we do this is determined experimentally.

4.1. **Implementation.** After doing some experiments we've chosen to compute the full product when $w > 0.5$ for classical multiplication, and when $w > 0.15$ for Karatsuba multiplication. In table 6 we list the timings of some experiments using our code. Here N_1 is the length of f, while we take the length of g to be 800. We compute $f \times_N g$. The other columns have the same meaning as in table 3.

5. Other coefficient rings

Until now we have only considered the values of parameters in our implementation, when the polynomials have integer coefficients. For other coefficient rings one can also determine these parameters by experiment. In $A^{\#}$ these values can be exported by the coefficient ring. For example in our implementation, when the coefficients are elements of some finite field and we are using Karatsuba multiplication, we use full multiplication only when $w > 0.5$ (was $w > 0.15$ for integer coefficients).

N_1	N	K	SK1	IK1	C	SC	IC
1600	1600	1791	1437	20%	10366	5500	47%
1600	1760	1791	1629	9%	10366	6483	37%
1600	1840	1791	1708	5%	10366	6934	33%
1600	2400	1791	1766	1%	10366	9317	10%
3200	3200	3520	3175	10%	20800	15850	24%
3200	3360	3520	3360	5%	20800	16866	19%
3200	3440	3520	3433	2%	20800	17333	17%
3200	4000	3520	3479	1%	20800	19700	5%
4000	4000	4791	3979	17%	25917	21734	16%
4000	4160	4791	4108	14%	25917	23000	11%
4000	4240	4791	4158	13%	25917	23466	9%
4000	4800	4791	4616	4%	25917	25900	0%

TABLE 7. Timings for polynomials having coefficients in $\mathbf{Z}/1009\mathbf{Z}$

In table 7 we list the timings of some experiments with polynomials having coefficients in $\mathbf{Z}/1009\mathbf{Z}$. The length of g is 1600. The columns have the same meaning as in table 6.

6. The opposite short product

Instead of computing only the coefficients of the low degree terms of a product of polynomials, one can of course also compute only the coefficients of the high degree terms. We call this the opposite short product. For polynomials $f, g \in R[x]$ of degree d_f resp. d_g it is defined as:

$$f \times^N g = \sum_{k=N}^{d_f+d_g} \left(\sum_{i+j=k} f_i g_j \right) x^k. \tag{4}$$

In order to compute the opposite short product one can adjust the algorithm for the short product or use the formula

$$f \times^N g = \mathrm{rev}(\mathrm{rev}(f, d_f) \times_{(d_f+d_g-N+1)} \mathrm{rev}(g, d_g), d_f + d_g),$$

where the ith coefficient of $\mathrm{rev}(h, n)$ is the $(n-i)$th coefficient of h.

Everything we saw for the short product in the previous sections holds in a slightly adjusted way for the opposite short product.

7. Applications

In this section we will show some applications of the short product. In some cases we will compute the theoretical gain obtained by replacing, where possible, the full product by the short product. We will only look at Karatsuba multiplication since in some applications it makes no sense to use classical multiplication and in others it is clear what the gain will be when using classical multiplication. When f and g are polynomials of length N, we will denote the cost for a full Karatsuba multiplication of f and g by $T(N)$ and the cost of computing the short product $f \times_N g$ by $S(N)$. From the preceding we know that $S(N) \approx 0.8\,T(N)$. When computing the cost of an algorithm we only take multiplications into account.

7.1. **Power series inversion and quotient.** Power series can be inverted by using Newtons iteration method. If $p \in R[[x]]$ is the power series to be inverted ($p(0)$ a unit in R), $q_0 = 1/p(0)$ and $q_{n+1} = q_n(1+(1-q_np)) \pmod{x^{2^{n+1}}}$ for $n \geq 0$, then $q_n = 1/p \pmod{x^{2^n}}$ for $n \geq 0$ (see also [4] 4.9).

Now let's look at the cost of computing q_{n+1}. For this consider p to be a polynomial of length 2^{n+1} and denote by f_s^t the polynomial of length $t-s$ whose ith coefficient is the $(s+i)$th coefficient of f. When using full multiplications we have to compute q_np and, since $1-q_np = 0 \pmod{x^{2^n}}$, only $q_n(q_np)_{2^n}^{2^{n+1}}$. This will cost $3T(2^n)$. In fact, computing $q_n \times^{2^n} p_0^{2^n}$ and $q_n \times_{2^n} p_{2^n}^{2^{n+1}}$ suffices to compute $(q_np)_{2^n}^{2^{n+1}}$ and we only need $q_n \times_{2^n} (q_np)_{2^n}^{2^{n+1}}$. So computing (opposite) short products this computation costs only $3\,S(2^n)$, so we can save 20%.

Since the foregoing holds for all n we see that by using short products we can save 20% when computing $1/p \pmod{x^{2^n}}$ from $1/p(0)$. The total cost for this is then $\sum_{i=0}^{n-1} 3\,S(2^i) \leq 1.2\,T(2^n)$.

When computing a quotient $p_1/p_2 \pmod{x^{2^n}}$ of power series, we can first compute $1/p_2 \pmod{x^{2^n}}$ and multiply this by p_1. Since we can save again 20% at this last multiplication we get also a gain of 20% for the computation of the quotient. The total cost for this is then $2\,T(2^n)$.

We get an even better algorithm for computing a power series quotient by using the following method. Let $q = p_1/p_2 \pmod{x^n}$ and write $p_1 \pmod{x^n} = p_{10} + p_{11}x^{n/2}$, $p_2 \pmod{x^n} = p_{20} + p_{21}x^{n/2}$ and $q = q_0 + q_1x^{n/2}$. Then

$$\frac{p_{10} + p_{11}x^{n/2} - q_0p_{20}}{x^{n/2}} - q_0p_{21} = q_1p_{20} \pmod{x^{n/2}},$$

so we can compute q by computing first q_0, then $q_0 \times^{n/2} p_{20}$ and $q_0 \times_{n/2} p_{21}$ and finally q_1. If $K(n)$ is the cost for this computation, then we see that $K(n) = 2\,K(n/2) + 2\,S(n/2)$. From this it follows that $K(2^n) = 2^nK(1) + \sum_{i=1}^{n} 2^iS(2^{n-i}) \leq 1.6\,T(2^n)$.

An algorithm which is still better can be achieved by using a polynomial variant of the algorithm in [6] for integer division as a subalgorithm (see also section 7.2). This subalgorithm computes $\frac{p_{10}+p_{11}x^{n/2}-q_0p_{20}}{x^{n/2}}$ in $2\,T(n/2)$. If $L(n)$ is the cost for the computation of q using this subalgorithm, then we see that $L(n) = 2\,T(n/2) + S(n/2) + L(n/2)$. From this it follows that $L(2^n) = 2.8\sum_{i=0}^{n-1} T(2^i) \leq 1.4\,T(2^n)$.

7.2. **Polynomial quotient.** The problem of computing the quotient of polynomials can be translated to the computation of some low order terms of the quotient of so-called reciprocal polynomials viewed as power series (see [4] 4.9). In this way all results of section 7.1 become valid for polynomial quotients.

The computation of quotient and remainder can be performed by the polynomial variant of the algorithm for integers in [6]. This looks as follows. When p_1 has length $2n-1$ and p_2 has length n, write $p_1 = p_{10} + p_{11}x^n$ and $p_2 = p_{20} + p_{21}x^{n/2}$. Compute quotient and remainder of p_{11} divided by p_{21}, giving $p_{11} = \tilde{q}p_{21} + \tilde{r}$ and write $\tilde{r}x^n - \tilde{q}p_{20}x^{n/2} + p_{10} = h_0 + h_1x^{n/2} + h_2x^n$. Then compute quotient and remainder of $h_1 + h_2x^{n/2}$ divided by p_{21}, giving $h_1 + h_2x^{n/2} = \hat{q}p_{21} + \hat{r}$. Then $\hat{q} + \tilde{q}x^{n/2}$ is the quotient and $h_0 + \hat{r}x^{n/2} - \hat{q}p_{20}$ is the remainder when dividing p_1 by p_2.

When $K(n)$ is the cost for these computations we get the relation $K(n) = 2\,K(n/2) + 2\,T(n/2)$. From this we get $K(2^n) \leq 2\,T(2^n)$ as was also proven in [6].

When we only need the quotient we can adjust the previous algorithm in the following way. Since we need only h_2 and the highest coefficient of h_1 in order to determine $\hat{q}$, we only have to compute $\tilde{q} \times^{n/2-1} p_{20}$. Also we don't have to compute the final remainder. When $L(n)$ is the cost for these computations we get the relation $L(n) = K(n/2) + S(n/2) + L(n/2)$. From this we get $L(2^n) \leq 1.4\,T(2^n)$.

In the algorithms above we compute the quotient from left to right, i.e. highest terms first. When a division of polynomials is exact, i.e. the remainder is 0, one can also compute the quotient from right to left, i.e. lowest terms first, and both methods can be combined by computing the upper half of the quotient from left to right and the lower half from right to left (see [13] and [5], [9] for the integer case). The above-mentioned algorithm to compute only the quotient can be adjusted to compute the quotient from right to left. Combining both algorithms, by letting them both compute half of the quotient, gives an algorithm whose costs are $C(2^n) = 2\,L(2^{n-1}) \leq 2.8\,T(2^{n-1}) \approx 0.93\,T(2^n)$. We see that we can perform exact division in less time than a multiplication of the same size would take.

Notice that the algorithm to compute an exact quotient from right to left can also be applied to compute a power series quotient. In this way we get an algorithm costing only $1.4\,T(2^n)$ compared to $2\,T(2^n)$ previously.

7.3. **Fraction free Gaussian elimination.** When performing 1-step fraction-free Gaussian elimination on a matrix one has to perform many exact divisions of the form $(ab - cd)/q$ (see [1]). When the matrix has polynomial entries we can apply our methods in this situation. First we can use the exact division algorithm of section 7.2. When we perform an exact division $q = p_1/p_2$ of polynomials, using our method, and q has length n, we only need to know the $n/2$ highest and $n/2$ lowest coefficients of p_1. This means that when we want to compute $(ab - cd)/q$, it is sufficient to compute both a short and an opposite short product of a and b, resp. c and d (see also [13] for the case of classical multiplication).

When the initial entries of the matrix all have degree l, then in general at the kth stage, a, b, c and d will have degree kl and q will have degree $(k-1)l$, and thus the quotient will have degree $(k+1)l$. When performing full multiplication and the bidirectional version of the full division algorithm of section 7.2 each instance of computing $(ab - cd)/q$ at the kth stage takes $2\,T(kl) + 4\,T((k+1)l/2) = 2\,T(kl) + 4/3\,T((k+1)l)$. When using short products and the last algorithm of section 7.2 this takes $4\,S((k+1)l/2) + 2.8/3\,T((k+1)l) = 2\,T((k+1)l)$. When we assume that $T((k+1)l) \approx T(kl)$ (which holds for large k) we go from $10/3\,T(kl)$ to $2\,T(kl)$, which means a gain of 40%.

Notice that in a similar way one can get improvements when performing 2-step fraction-free Gaussian elimination (see [1]). In this case one has to compute expressions of the form $(ab + cd + ef)/q$ besides expressions as above.

7.4. **Applications involving long integers.** As mentioned before, all results in this paper can in principal be ported to the case of long integer arithmetic. However, the algorithms will be more complicated than in the polynomial case (see [5], [6] and [9]), mostly due to the phenomenon of carries when performing integer arithmetic.

Also one can get an improved version of the algorithm described in [10] to perform multiprecision floating point multiplication, by using an adapted version of the algorithm to compute short products mentioned in this paper.

References

[1] Bareiss, E.H., *Sylvester's Identity and Multistep Integer–Preserving Gaussian Elimination*, Math. Comp. 22, 565–578, 1968.

[2] Bronstein, M., Σ^{IT} – *a strongly-typed embeddable computer algebra library*, Proc. DISCO'96, LNCS 1128, Springer, 22–33, 1996.

[3] Cantor, G., Kaltofen, E., *On fast multiplication of polynomials over arbitrary algebras*, Acta Inf. 28, 693–701, 1991.

[4] Geddes, K.O., Czapor, S.R., Labahn, G., *Algorithms for Computer Algebra*, Kluwer Academic Publishers, 1992.

[5] Jebelean, T., *An Algorithm for Exact Division*, J. Symbolic Computation 15, 169–180, 1993.

[6] Jebelean, T., *Practical Integer Division with Karatsuba Complexity*, Proc. ISSAC'97, 339–341, 1997.

[7] Kaminski, M., *An Algorithm for Polynomial Multiplication That Does Not Depend on the Ring Constants*, J. Algorithms 9, 137–147, 1988.

[8] Knuth, D.E., *The Art of Computer Programming, Vol. 2, Seminumerical Algorithms*, 2nd ed., Addison–Wesley, 1981.

[9] Krandick, W., Jebelean, T., *Bidirectional Exact Integer Division*, J. Symbolic Computation 21, 441–455, 1996.

[10] Krandick, W., Johnson J.R., *Efficient Multiprecision Floating Point Multiplication with Exact Rounding*, Tech. Rep. 93-76, RISC-Linz, Johannes Kepler University, Linz, Austria, 1993.

[11] Schönhage, A., *Schnelle Multiplikation von Polynomen über Körpern der Charakteristik* 2, Acta Inf. 7, 395–398, 1977.

[12] Schönhage, A., Grotefeld, A.F.W., Vetter, E., *Fast Algorithms - A multitape Turing Machine Implementation*, BI Wissenschaftsverlag, 1994.

[13] Schönhage, A., Vetter, E., *A New Approach to Resultant Computations and Other Algorithms with Exact Division*, Proc. of the 2nd Annual European Symposiun on Algorithms, LNCS 855, Springer, 448–459, 1994.