Space- and Time-Efficient Polynomial Multiplication

Daniel S. Roche
Cheriton School of Computer Science
University of Waterloo
Waterloo, Ontario, Canada
droche@cs.uwaterloo.ca
www.cs.uwaterloo.ca/~droche/

ABSTRACT

Countless algorithms have been developed for the multiplication of univariate polynomials and multi-precision integers, but all those with sub-quadratic time complexity currently require at least $\Omega(n)$ extra space for the computation. A new routine based on the Karatsuba/Ofman algorithm is presented with the same time complexity of $O(n^{1.59})$ but only $O(\log n)$ extra space. A second routine based on the method of Schönhage/Strassen achieves the same pseudo-linear time and $O(1)$ extra space, but only under certain conditions. A preliminary implementation over $\mathbb{F}_p[x]$, where p fits into a single machine word, is presented and compared with existing software.

Categories and Subject Descriptors

F.2.1 [**Analysis of Algorithms and Problem Complexity**]: Numerical Algorithms and Problems—*Computations on polynomials*; G.1.0 [**Numerical Analysis**]: General—*Multiple precision arithmetic*; G.4 [**Mathematical Software**]: Algorithm design and analysis, Efficiency

General Terms

Algorithms, Performance, Theory

Keywords

Polynomial multiplication, integer multiplication, space efficiency, time-space tradeoff

1. INTRODUCTION

The multiplication of univariate polynomials and multi-precision integers is one of the most basic and crucial operations for efficient mathematical and symbolic computation. These routines form the low-level basis of any computer algebra system, and specific high-performance libraries such as NTL [14] and GMP [6] have been built around fast arithmetic operations.

ISSAC'09, July 28–31, 2009, Seoul, Republic of Korea.

Consider the problem of computing the product of two degree-n polynomials or integers whose length is n machine words. Significant algorithmic progress has been made in improving the time complexity from $O(n^2)$ for the classical algorithm. The two-way divide and conquer algorithm which we refer to as Karatsuba's was first introduced in [10] and has complexity $O(n^{1.59})$. This was later generalized to a k-way divide and conquer scheme by Toom and Cook [17, 4]; in particular, when $k = 3$, an $O(n^{1.47})$ algorithm is produced. The use of the Fast Fourier Transform (FFT) algorithm [5] made the "quasi-linear" time complexity of $O(n \log n \operatorname{loglog} n)$ possible — first for long integer multiplication [13], and later for polynomials over arbitrary algebras [3]. Current best results give an upper bound of $O(n \log n 2^{O(\log^* n)})$ time complexity for multiplication [7]. In a typical high-performance implementation, either for integers or polynomials, the classical algorithm is used for the smallest operands, followed by a range where the divide-and-conquer approaches are best (always Karatsuba and sometimes Toom-Cook 3-way as well), and FFT-based methods are used for the largest inputs.

1.1 Measuring Space Efficiency

Unfortunately, all known multiplication algorithms other than the classical method require at least a linear amount of extra storage space. Our goal is to improve this, specifically for the Karatsuba and FFT-based methods. First we must specify exactly what is meant by the term "extra space".

Branching programs are a fairly general model of computation and in particular lower bounds for branching programs also apply to straight line programs and random access machine models. Branching programs essentially allow random access to the read-only input and random writes to the output, but each position of the output can only be written to once and never read. In this model, a lower bound of $\Omega(n^2)$ time times space complexity is known [1]. This means that, while constant space is possible for the classical algorithm, all the other multiplication methods above will require at least a polynomial amount of space in the branching program model, and hence significant progress will not happen here.

For this reason, we depart from the traditional models of computation and space complexity measures and instead use what we consider to be a more practical model based on a typical modern architecture. Fortunately, the only significant change required is that we allow multiple writes *and* reads to/from the output space. Perhaps surprisingly, this small change allows significant improvements from the $\Omega(n^2)$

time times space lower bound in branching programs for multiplication.

Specifically, we partition the memory accessed by a program into three parts: a read-only input space, a read/write work space, and a read/write output space. The program has random access to all three, with unit cost for any read or write operation, and obviously the size of the output space can never be larger than the actual output from the algorithm. We only count the size of the work space, which we call the "extra space" required for the algorithm.

For the multiplication of multiple-precision integers, each memory location can hold one word-sized integer, and basic arithmetic operations on word-sized integers have unit cost. When multiplying univariate polynomials over a ring R, each memory location can also hold an element in R and arithmetic operations in R also have unit cost. While this is a more powerful model than those above, it corresponds more closely to actual computation on a modern computer.

Even in this more powerful model, known methods require at least linear extra space for every multiplication algorithm other than the classical one. In [12], Monagan shows how to implement the classical algorithm with $O(1)$ extra space, and discusses the importance of space efficiency in basic polynomial arithmetic. Since then, a number of authors have concentrated specifically on space efficiency in Karatsuba's algorithm. Upper and lower bounds of $\Theta(n)$ for the amount of extra space for Karatsuba multiplication are derived in [11] and then used to preallocate storage for all recursive calls. For polynomial multiplication, [16] shows how to reduce the space by half by reusing the output space for some intermediate products, and conjectures that less than linear extra space is not possible. Many of these results are summarized in the forthcoming book by Brent and Zimmermann, which states "The efficiency of an implementation of Karatsuba's algorithm depends heavily on memory usage" [2].

1.2 Overview

The current aim is to develop schemes for multiplication which achieve less than $O(n^2)$ time complexity and use as little extra space as possible. In Section 2 a recursive algorithm is presented with the same time complexity as Karatsuba but only a constant amount of extra space at each step, for a total of $O(\log n)$ extra space. Section 3 gives an algorithm for FFT-based multiplication with the same time complexity but only $O(1)$ extra space, under certain conditions. Implementation issues for these new algorithms are discussed in Section 4.

2. SPACE-EFFICIENT KARATSUBA MULTIPLICATION

Gauss was perhaps the first to notice that the multiplication of two complex numbers $(a + b\boldsymbol{i})$ and $(c + d\boldsymbol{i})$ can be performed with only three multiplications in $\mathbb{R}$ as

$$(a + b\boldsymbol{i})(c + d\boldsymbol{i}) = ac - bd + ((a + b)(c + d) - ac - bd)\,\boldsymbol{i}.$$

Karatsuba and Ofman used this idea to develop a scheme for long integer multiplication in [10]. The description of the algorithm is cleanest when multiplying polynomials in $\mathsf{R}[x]$ with equal sizes that are both divisible by 2, so for simplicity we will present that case first. The corresponding methods for odd-sized operands, different-sized operands, and multiple-precision integers will follow fairly easily from this case.

2.1 Standard Karatsuba Algorithm

For polynomials $f, g \in \mathsf{R}[x]$ each with degree less than $2k$, the Karatsuba algorithm first splits each input polynomial into high- and low-degree parts by writing

$$f = f_0 + f_1 x^k, \qquad g = g_0 + g_1 x^k, \tag{1}$$

where each of f_0, f_1, g_0, g_1 is a polynomial in $\mathsf{R}[x]$ with degree less than k. We then compute three intermediate products:

$$\alpha = f_0 \cdot g_0, \qquad \beta = f_0 \cdot f_1, \qquad \gamma = (f_0 + f_1) \cdot (g_0 + g_1). \tag{2}$$

Finally, these products are combined to produce the product of f and g as follows:

$$f \cdot g = \alpha + (\gamma - \alpha - \beta) \cdot x^k + \beta \cdot x^{2k}. \tag{3}$$

A straightforward implementation might allocate n units of extra storage at each recursive step to store the intermediate product γ, resulting in an algorithm that uses a linear amount of extra space and performs approximately $4n$ additions of ring elements besides the three recursive calls.

There is of course significant overlap between the three terms of (3). To see this more clearly, split each polynomial α, β, γ into its low-order and high-order coefficients as in (1). Then we have (with no overlap):

$$f \cdot g = \alpha_0 + (\gamma_0 + \alpha_1 - \alpha_0 - \beta_0)x^k + (\gamma_1 + \beta_0 - \alpha_1 - \beta_1)x^{2k} + \beta_1 x^{3k} \tag{4}$$

Examining this formulation, we see that the difference $\alpha_1 - \beta_0$ occurs twice, so the number of additions can be reduced to $7n/2$ at each recursive step. This has been noticed by others, who have also made improvements in the amount of extra space, but never less than $O(n)$ [11, 16, 2].

2.2 Improved algorithm: general formulation

Again, we first present our algorithm in the easiest case, for multiplication of univariate polynomials that have equal and even sizes.

The key to obtaining $O(\log n)$ extra space for Karatsuba-like multiplication is by solving a slightly more general problem. In particular, two extra requirements are added to the algorithm at each recursive step.

CONDITION 2.1. *The low-order n coefficients of the output space are pre-initialized and must be added to. That is, half of the product space is initialized with a polynomial $h \in \mathsf{R}[x]$ of degree less than n.*

With Condition 2.1, the computed result should now equal $h + f \cdot g$.

CONDITION 2.2. *The first operand to the multiplication f is given as two polynomials which must first be summed before being multiplied by g. That is, rather than a single polynomial $f \in \mathsf{R}[x]$, we are given two polynomials $f^{(0)}, f^{(1)} \in \mathsf{R}[x]$, each with degree less than n.*

With both these conditions, the result of the computation should be $h + (f^{(0)} + f^{(1)}) \cdot g$.

Of course, these conditions should not be made on the very first call to the algorithm, and this will be discussed in the next subsection. We are now ready to present the algorithm in the easiest case that $f, g \in \mathsf{R}[x]$ with $\deg f =$

$\deg g = 2k-1$ for some $k \in \mathbb{N}$. If A is an array in memory, we use the notation of $A[i..j]$ for the sub-array from indices i (inclusive) to j (exclusive), with $0 \le i < j \le |A|$. If array A contains a polynomial f, then the array element $A[i]$ is the coefficient of x^i in f. The three read-only input operands $f^{(0)}, f^{(1)}, g$ are stored in arrays A, B, C, respectively, and the output is written to array D. From the first condition, $D[0..2k]$ is initialized with h.

ALGORITHM `SE_KarMult_1+2`.

Input: $k \in \mathbb{N}$ and $f^{(0)}, f^{(1)}, g, h \in \mathsf{R}[x]$ with degrees less than $2k$ in arrays A, B, C, D, respectively

Output: $h + (f^{(0)} + f^{(1)}) \cdot g$ stored in array D

1: $D[k..2k] \leftarrow D[k..2k] + D[0..k]$
2: $D[3k-1..4k-1] \leftarrow$
$A[0..k] + A[k..2k] + B[0..k] + B[k..2k]$
3: $D[k..3k-1] \leftarrow$
`SE_KarMult_1+2`$(C[0..k], C[k..2k], D[3k-1..4k-1])$

4: $D[3k-1..4k-1] \leftarrow D[k..2k] + D[2k..3k-1]$
5: $D[0..2k-1] \leftarrow$ `SE_KarMult_1+2`$(A[0..k], B[0..k], C[0..k])$

6: $D[2k..3k-1] \leftarrow D[2k..3k-1] - D[k..2k-1]$
7: $D[k..2k] \leftarrow D[3k-1..4k-1] - D[0..k]$
8: $D[2k..4k-1] \leftarrow$
`SE_KarMult_1+2`$(A[k..2k], B[k..2k], C[k..2k])$
9: $D[k..2k] \leftarrow D[k..2k] - D[2k..3k]$
10: $D[2k..3k-1] \leftarrow D[2k..3k-1] - D[3k..4k-1]$

Table 1 summarizes the computation by showing the actual values (in terms of the input polynomials and intermediate products) stored in each part of the output array D after each step of the algorithm. Between the recursive calls on Steps 3, 5, and 8, we perform some additions and rearranging to prepare for the next multiplication. Notice that a few times a value is added somewhere only so that it can be cancelled off at a later point in the algorithm. An example of this is the low-order half of h, h_0, which is added to h_1 on Step 1 only to be cancelled when we subtract $(\alpha_0 + h_0)$ from this quantity later, on Step 7.

The final value stored in D after Step 10 is

$$(h_0 + \alpha_0) + (h_1 + \alpha_1 + \gamma_0 - \alpha_0 - \beta_0)x^k + (\beta_0 + \gamma_1 - \alpha_1 - \beta_1)x^{2k} + \beta_1 \cdot x^{3k}, \quad (5)$$

which we notice is exactly the same as (4) with the addition of $h_0 + h_1 x^k$ as specified by Condition 2.1.

The base case of the algorithm will be to switch over to the classical algorithm for multiplication; the exact size at which the classical algorithm should be preferred will depend on the implementation. We do not give the details, but it is straightforward to implement the classical multiplication without using any auxiliary storage space, even with the two extra conditions. This proves the correctness of the following theorem:

THEOREM 2.3. *Let $f^{(0)}, f^{(1)}, g, h \in \mathsf{R}[x]$ be polynomials each with degree one less than $n = 2^b$ for some $b \in \mathbb{N}$. Algorithm `SE_KarMult_1+2` correctly computes $h + (f^{(0)} + f^{(1)}) \cdot g$ using the output space (which is initialized with h) and $O(\log n)$ extra space, and has time complexity $O(n^{\log_2 3})$, or $O(n^{1.59})$.*

PROOF. The size of the input polynomials n must be a power of 2 so that each recursive call is an even-sized arguments (until the last recursive call when each of $f^{(0)}, f^{(1)}, g, h$ is just a scalar in R). Correctness follows from the discussion above. We can see that there are exactly three recursive calls on input of one-half the size of the original input, and this gives the stated time complexity bounds. Finally, we see that only $O(1)$ extra space is used at each recursive step, for a total of $O(\log n)$ extra space, as $\log_2 n$ is the depth of recursion. □

2.3 Initial calls

The initial call to compute the product of two polynomials f and g will not satisfy Conditions 2.1 and 2.2 above. So there must be top-level versions of the algorithm which do *not* solve the more general problem of $h + (f^{(0)} + f^{(1)}) \cdot g$.

Working backwards, first denote by `SE_KarMult_1` an algorithm similar to Algorithm `SE_KarMult_1+2`, but which does not satisfy Condition 2.2. Namely, the input will be just three polynomials $f, g, h \in \mathsf{R}[x]$, with h stored in the low-order half of the output space, and the algorithm computes $h + f \cdot g$. In this version, two of the additions on Step 2 are eliminated. The function call on Step 3 is still to `SE_KarMult_1+2`, but the other two on Steps 5 and 8 are recursive calls to `SE_KarMult_1`.

Similarly, `SE_KarMult` will be the name of the top-level call which does not satisfy either Condition 2.1 or 2.2, and therefore simply computes a single product $f \cdot g$ into an uninitialized output space. Here again we save two array additions on Step 2, and Step 1 is replaced with the instruction:

$$D[0..k] \leftarrow C[0..k] + C[k..2k],$$

so that the first two function calls on Steps 3 and 5 are recursive calls to `SE_KarMult`, and only the last one on Step 8 is a call to `SE_KarMult_1`.

The number of additions (and subtractions) at each recursive step determine the hidden constant in the $O(n^{1.59})$ time complexity measure. We mentioned that a naïve implementation of Karatsuba's algorithm uses $4n$ additions at each recursive step, and it is easy to improve this to $7n/2$. By inspection, Algorithm `SE_KarMult_1+2` uses approximately $9n/2$ additions at each recursive step, and therefore both `SE_KarMult_1` and `SE_KarMult` use only $7n/2$ additions at each recursive step, matching the best known existing algorithms. So although calls to `SE_KarMult_1+2` will eventually dominate, incurring a slight penalty in extra arithmetic operations, most of the initial calls, particularly for smaller values of n (where Karatsuba's algorithm is actually used), will be to `SE_KarMult_1` or `SE_KarMult`, and should not be any more costly in time than a good existing implementation. This gives us hope that our space-efficient algorithm might be useful in practice; see Section 4 for more discussion on this topic.

2.4 Unequal and odd-sized operands

So far our algorithm only works when both input polynomials have the same degree which is one less than a power of two, since the size of both operands at each recursive call must be even. Special cases to handle the cases when the input polynomials have even degree (i.e. odd size) or different degres will resolve these issues and give a general-purpose multiplication algorithm.

First consider the case that $\deg f^{(0)}$, $\deg f^{(1)}$, $\deg g$, and $\deg h$ are all equal to $2k$ for some $k \in \mathbb{N}$, so that each polynomial has an odd number of coefficients. It is easy

	$D[0..k]$	$D[k..2k]$	$D[2k..3k-1]^*$	$D[3k-1..4k-1]^*$
0	h_0	h_1	—	—
1	h_0	h_0+h_1	—	—
2	h_0	h_0+h_1	—	$f_0^{(0)}+f_1^{(0)}+f_0^{(1)}+f_1^{(1)}$
3	h_0	$h_0+h_1+\gamma_0$	γ_1	$f_0^{(0)}+f_1^{(0)}+f_0^{(1)}+f_1^{(1)}$
4	h_0	$h_0+h_1+\gamma_0$	γ_1	$h_0+h_1+\gamma_0+\gamma_1$
5	$h_0+\alpha_0$	α_1	γ_1	$h_0+h_1+\gamma_0+\gamma_1$
6	$h_0+\alpha_0$	α_1	$\gamma_1-\alpha_1$	$h_0+h_1+\gamma_0+\gamma_1$
7	$h_0+\alpha_0$	$h_1+\gamma_0+\gamma_1-\alpha_0$	$\gamma_1-\alpha_1$	$h_0+h_1+\gamma_0+\gamma_1$
8	$h_0+\alpha_0$	$h_1+\gamma_0+\gamma_1-\alpha_0$	$\beta_0+\gamma_1-\alpha_1$	β_1
9	$h_0+\alpha_0$	$h_1+\alpha_1+\gamma_0-\alpha_0-\beta_0$	$\beta_0+\gamma_1-\alpha_1$	β_1
10	$h_0+\alpha_0$	$h_1+\alpha_1+\gamma_0-\alpha_0-\beta_0$	$\beta_0+\gamma_1-\alpha_1-\beta_1$	β_1

*The last two sub-arrays shift by one to $D[2k..3k]$ and $D[3k..4k-1]$ at Step 8.

Table 1: Values stored in D through the steps of Algorithm `SE_KarMult_1+2`

to get the same asymptotic result here, by simply pulling off the low-order coefficients of the input and writing the result $h+(f^{(0)}+f^{(1)})\cdot g$ as

$$h_0+h_1x+\hat{h}x^2+(f_0^{(0)}+f^{\hat{(0)}}x+f_0^{(1)}+f^{\hat{(1)}}x)\cdot(g_0+\hat{g}x).$$

A rearrangement produces a single call to `SE_KarMult_1+2` with even-length arguments, $\hat{h}+(f^{\hat{(0)}}+f^{\hat{(1)}})\cdot\hat{g}$, three additions of a scalar times a polynomial, and a few more scalar products and additions. Since we can multiply a polynomial by a scalar and add to a pre-initialized result without using any extra space, this still achieves the same asymptotic time complexity and $O(\log n)$ extra space, albeit with a few extra arithmetic operations. There are probably more efficient ways to tackle the odd-sized case, by some slight shifting of the intermediate results in the output array, but we have not yet fully investigated them.

This handles the multiplication of any two polynomials with the same degree, but what if the polynomials have different degrees? In this case, we use the well-known trick of blocking the larger polynomial into sub-arrays whose length equal the degree of the smaller one. That is, given $f,g\in\mathsf{R}[x]$ with $n=\deg f$, $m=\deg g$ and $n>m$, we write $n=qm+r$ and reduce the problem to computing q products of a degree-m by a degree-$(m-1)$ polynomial and one product of a degree-m by a degree-$(r-1)$ polynomial. Further special cases of the algorithms above are now needed to handle polynomial multiplication where the degrees of f and g differ by one, but this is relatively straightforward as above.

Each of the q initial products overlap in exactly m coefficients (half the size of the output), so Condition 2.1 actually works quite naturally here, and the q m-by-$(m-1)$ products are calls to a version of `SE_KarMult_1`. The single m-by-$(r-1)$ product is performed *first* (so that the entire output is uninitialized), and this is done by a recursive call to this same procedure multiplying arbitrary unequal-length polynomials.

All these special cases are available for more careful examination in the implementation discussed later, and they give the following, which again is the first sub-quadratic multiplication algorithm to do better than $O(n)$ extra space.

THEOREM 2.4. *For any $f,g\in\mathsf{R}[x]$ with degrees less than n, the product $f\cdot g$ can be computed using $O(\log n)$ extra space and $O(n^{\log_2 3})$ or $O(n^{1.59})$ operations in R and on word-sized integers.*

3. SPACE-EFFICIENT FFT-BASED MULTIPLICATION

The fastest practical algorithms for integer and polynomial multiplication are based on Fourier transforms. Suppose we want to compute the product of $f,g\in\mathsf{R}[x]$, where $\deg f+\deg g<n$. Using the notation of [8], we say that $\omega\in\mathsf{R}$ is an n-PRU (primitive root of unity) iff $\omega^n=1$ and $\omega^i\neq 1$ for $1\le i<n$. Then the discrete Fourier transform of a sequence $(a_0,a_1,\ldots,a_{n-1})\in\mathsf{R}^n$ at ω is defined as

$$\mathrm{DFT}_\omega(a_0,a_1,\ldots,a_{n-1})=\left(\sum_{j=0}^{n-1}a_j\cdot\omega^{ij}\right)_{0\le i\le n-1}.$$

This transform can be used for both multi-point evaluation and interpolation of polynomials. If we write $f\in\mathsf{R}[x]$ as

$$f=f_0+f_1x+f_2x^2+\cdots+f_{n-1}x^{n-1}$$

(noticing that all f_i with $i>\deg f$ will equal zero), it is easy to see that

$$\begin{aligned}\mathrm{DFT}_\omega(f_0,\ldots,f_{n-1}) &= \big(f(\omega^0),\ldots,f(\omega^{n-1})\big)\\ \frac{1}{n}\mathrm{DFT}_{\omega^{-1}}(f(\omega^0),\ldots,f(\omega^{n-1})) &= (f_0,\ldots,f_{n-1})\end{aligned}$$

For the remainder, we will write $\mathrm{DFT}_\omega(f)$ as a shorthand for the discrete Fourier transform of the coefficients. The Fast Fourier Transform (FFT) [5] is an algorithm to compute DFT_ω using $O(n\log n)$ operations in R which has become one of the most useful and important algorithms in computer science. To compute the product of $f,g\in\mathsf{R}[x]$, we then simply use the FFT three times to compute:

$$f\cdot g=\frac{1}{n}\mathrm{DFT}_{\omega^{-1}}\left(\mathrm{DFT}_\omega(f)*\mathrm{DFT}_\omega(g)\right),$$

where $*$ signifies pairwise multiplication of vector elements. This works provided the degree of the product is less than n *and* n is a power of 2, say 2^k for some $k\in\mathbb{N}$.

The real difficulty lies in the requirement that R contains an n-PRU ω. When it does not, we can construct a so-called virtual root of unity by working in a larger field and incurring an extra multiplicative factor of $\mathrm{loglog}\,n$ in the complexity [13, 3]. But here we explicitly dodge this issue by simply assuming that the ring R already contains an n-PRU ω.

We will also assume in all cases that $\deg f + \deg g = n-1 = 2^k - 1$, so that the size of the output is the same as the size of the FFT that we need to compute. This is in some sense unavoidable, as even the truncated version of the FFT algorithm requires $2^{\lceil \log_2 n \rceil}$ space at some intermediate step (where n is the size of the output) [18]. When these conditions are met, the algorithm we now present will use only $O(1)$ extra space to multiply f and g, but even in the most general case we will save $O(n)$ auxiliary space compared to the standard implementation, and hence the algorithm presented can have broader implications.

3.1 Reverted binary ordering

The FFT algorithm has been well-studied, and in particular can be implemented completely in-place. That is, the procedure overwrites the input sequence with the DFT of that sequence, using only a constant amount of extra space. Although "self-sorting" methods are known which compute the output in order (see e.g. [15]), the standard and simplest in-place algorithm computes the output in the "reverted binary ordering". This is defined by the operator rev_k which transforms a k-bit binary number into another by reversing the order of the binary digits. So, for example, $\mathrm{rev}_6(58) = 23$ because $58 = 111010_2$ and $23 = 010111_2$. The reverted binary ordering of the numbers $0, 1, \ldots, 2^k - 1$ is simply the sequence $\mathrm{rev}_k(0), \mathrm{rev}_k(1), \ldots, \mathrm{rev}_k(2^k - 1)$. For $k = 4$, this sequence is

$$0, 8, 4, 12, 2, 10, 6, 14, 1, 9, 5, 13, 3, 11, 7, 15.$$

The following property will be useful for us:

LEMMA 3.1. *For all $k \in \mathbb{N}$, the reverted binary order of the positive k-bit integers, $(\mathrm{rev}_k(i))_{0<i<2^k}$, can be written as a concatenation of sequences:*

$$\left(\left(2^{k-i} \,\mathrm{rev}_i(j) + 2^{k-i-1} \right)_{0 \le j < 2^i} \right)_{0 \le i < k}$$

PROOF. For any $k \in \mathbb{N}$, it is easy to see that the sequence $(\mathrm{rev}_k(i))_{0 \le i < 2^k}$ can be written as the concatenation of two sequences as

$$(2\,\mathrm{rev}_{k-1}(i))_{0 \le i < 2^{k-1}}, (2\,\mathrm{rev}_{k-1}(i) + 1)_{0 \le i < 2^{k-1}}.$$

Applying this formula recursively to the left-most concatenated subsequence k times, then removing the leading 0, gives the stated result. □

3.2 Folded polynomials

A standard implementation of FFT-based multiplication needs to compute two size-n DFT's (of f and of g) in two size-n arrays (one of which could be the output space), then multiply pairwise elements to compute $\mathrm{DFT}_\omega(f \cdot g)$. The final DFT operation could be computed in-place in the output space.

Our approach avoids the need for the extra n-sized array for the initial DFT computation by computing first computing half of $\mathrm{DFT}(f \cdot g)$, then half of what remains, half of what remains again, and so forth until the operation is complete. The partial DFTs computed are defined by the following polynomials (here and hereafter rem indicates the remainder in exact division):

DEFINITION 3.2. *Let $f \in \mathsf{R}[x]$ and $m \in \mathbb{N}$ such that $\theta \in \mathsf{R}$ is a $2m$-PRU. Then the* folded polynomial $P\langle f, \theta\rangle \in \mathsf{R}[x]$ *is*

$$f(\theta \cdot x) \operatorname{rem} x^m - 1$$

If $f = \sum_{j \ge 0} f_j x^j$, then we can also write $P\langle f, \theta\rangle$ as

$$\sum_{j \ge 0} \theta^j f_j x^{j \operatorname{rem} m}.$$

Using this formula, we can easily compute $P\langle f, \theta\rangle$ using the space of the output and only $O(1)$ extra space in $O(\deg f)$ time.

We also observe the following property due to the fact that $(\theta^{2i})^m - 1 = 0$ for any $i \in \mathbb{N}$:

LEMMA 3.3. *Let $f \in \mathsf{R}[x]$ and $m \in \mathbb{N}$ such that $\theta \in \mathsf{R}$ is a $2m$-PRU. Then, for all $i \in \mathbb{N}$,*

$$P\langle f, \theta\rangle(\theta^{2i}) = f(\theta^{2i+1}).$$

Finally, we need the following fact relating the folded polynomials to the reverted binary ordering:

LEMMA 3.4. *Let $f \in \mathsf{R}[x]$ and $k \in \mathbb{N}$ such that $\omega \in \mathsf{R}$ is a 2^k-PRU. Then the following two sequences are equivalent:*

(a) $\left(f(\omega^{\mathrm{rev}_k(i)}) \right)_{0 < i < 2^k}$

(b) $\left(\left(P\langle f, \omega^{2^{k-i-1}} \rangle (\omega^{2^{k-i}\,\mathrm{rev}_i(j)}) \right)_{0 \le j < 2^i} \right)_{0 \le i < k}$

PROOF. From Lemma 3.3, we know that

$$P\langle f, \omega^{2^{k-i-1}} \rangle \left(\omega^{2^{k-i}\,\mathrm{rev}_i(j)} \right) = f\left(\omega^{2^{k-i}\,\mathrm{rev}_i(j) + 2^{k-i-1}} \right).$$

Therefore sequence (b) is equivalent to

$$\left(\left(f(\omega^{2^{k-i}\,\mathrm{rev}_i(j) + 2^{k-i-1}}) \right)_{0 \le j < 2^i} \right)_{0 \le i < k}.$$

By Lemma 3.1, this is exactly the same as sequence (a). □

3.3 Constant-space algorithm

Our strategy for FFT-based multiplication with $O(1)$ extra space is then to compute

$$\mathrm{DFT}_{\omega^{2^{k-i}}} \left(P\langle f \cdot g, \omega^{2^{k-i-1}} \rangle \right)$$

for $i = k-1, k-2, \ldots, 0$, storing them in reverse order in the output space, so that we end up with the sequence of Lemma 3.4 for the polynomial $f \cdot g$. Since the evaluations of this polynomial at powers of ω are now stored in reverted binary order, a single in-place FFT operation on the entire array produces the coefficients of the result in order. This approach is presented in Algorithm `SE_FFTMult`.

THEOREM 3.5. *Let $n = \deg f + \deg g + 1$ be the size of the output polynomial. Algorithm `SE_FFTMult` works correctly as stated in time $O(n \log n)$ using $O(2^k - n)$ extra space.*

PROOF. The value of θ is $\omega^{2^{k-i-1}}$ for the duration of each iteration through Steps 3–11. The **for** loops on Steps 4 and 6 compute $P\langle f, \theta\rangle$ and $P\langle g, \theta\rangle$, respectively. The DFT_{θ^2}'s of these two polynomials are computed in place on Steps 8 and 9, with the results being permuted into the reverted binary ordering. These are then multiplied in the loop at Step 10 so that before each iteration through Step 12, the sequence

$$\left(P\langle f \cdot g, \theta\rangle (\theta^{2\,\mathrm{rev}_i\, j}) \right)_{0 \le j < 2^i}$$

is stored in the sub-array $C[2^i..2^{i+1}]$.

ALGORITHM `SE_FFTMult`.

Input: $f, g \in \mathsf{R}[x]$ stored in arrays A, B, $k \in \mathbb{N}$ such that $\deg f + \deg g < 2^k$, and $\omega \in \mathsf{R}$ a 2^k-PRU

Output: The coefficients of $f \cdot g$ stored in a length-2^k array C

1: $\theta \leftarrow \omega$
2: **for** $i = k-1, k-2, \ldots, 0$ **do**
3: $\quad C[0..2^{i+1}] \leftarrow \mathbf{0}$
4: $\quad$ **for** $j = 0, 1, \ldots, \deg f$ **do**
5: $\quad\quad C[j \operatorname{rem} 2^i] \leftarrow C[j \operatorname{rem} 2^i] + A[j] \cdot \theta^j$
6: $\quad$ **for** $j = 0, 1, \ldots, \deg g$ **do**
7: $\quad\quad C[2^i + j \operatorname{rem} 2^i] \leftarrow C[2^i + j \operatorname{rem} 2^i] + B[j] \cdot \theta^j$
8: $\quad$ `InPlaceFFT`$_{\theta^2}(C[0..2^i])$
9: $\quad$ `InPlaceFFT`$_{\theta^2}(C[2^i..2^{i+1}])$
10: $\quad$ **for** $j = 0, 1, \ldots, 2^i - 1$ **do**
11: $\quad\quad C[2^i + j] \leftarrow C[2^i + j] \cdot C[j]$
12: $\quad \theta \leftarrow \theta^2$
13: $C[0] \leftarrow f(1) \cdot g(1)$
14: `InPlaceFFT`$_{\omega^{-1}}(C[0..2^k])$
15: $C[0..2^k] \leftarrow C[0..2^k]/(2^k \in \mathsf{R})$

Therefore from Lemma 3.4, the polynomial $f \cdot g$ evaluated at each power of ω is stored in reverted binary order in the array C after Step 13. The last two steps simply compute

$$f \cdot g = \frac{1}{n} \mathrm{DFT}_{\omega^{-1}}(\mathrm{DFT}_{\omega}(f \cdot g)),$$

and therefore the algorithm is correct.

For the time complexity, Steps 13 and 15 each cost $O(n)$ and Step 14 is $O(n \log n)$. The loops on lines 4 and 6 each cost $O(n)$ at each iteration, for a total cost of $O(nk) = O(n \log n)$. All other steps in the **for** loop from lines 2–12 are also $O(n)$ except for the two calls to `InPlaceFFT`. These cost $O(i \cdot 2^i)$ at each iteration, for a total cost of $O(n \log n)$.

Finally, it is clear that no more than a constant amount of extra space besides the array C is needed. Since $|C| = 2^k$, the stated result follows. □

In particular, when the size of the output is a power of 2, Theorem 3.5 tells us that the algorithm uses only a constant amount of extra space, as promised.

4. IMPLEMENTATION

A preliminary implementation of the algorithms presented above in the `C` language is available from the author's website at `http://www.cs.uwaterloo.ca/~droche/`. The current implementation works for polynomials over a finite field $\mathbb{F}_p$, where p is a single-precision integer. For the FFT-based multiplication method, we further require that the size of the output is a power of 2, and that $\mathbb{F}_p$ contains a PRU of that size, as mentioned above.

For benchmarking, we chose to compare with the popular `C++` library NTL [14]. There are some more recently-developed libraries which already claim imrpovement over NTL, notably David Harvey's `zn_poly` [9], but NTL seems to be more widely used, at least at the moment. Furthermore, our aim here is not to claim the "fastest" implementation of polynomial arithmetic, but merely to demonstrate that the space-efficient algorithms presented *can* be useful in practice.

With that goal in mind, we should mention a few fundamental differences in our code versus NTL before stating the results. First, we have a tighter restriction on the largest modulus that can be used, so that some modular reductions can be delayed in our algorithms. Second, NTL is not thread-safe, and in fact pre-allocates and reuses the "scratch space" for the multiplication algorithms. Finally, NTL is compiled into a static library in `C++`, whereas we are using direct compilation in `C`.

Size	Iterations	NTL	Karatsuba	FFT-Based
64	100000	1.76	**1.26**	3.13
128	100000	5.22	**3.85**	7.01
256	100000	15.43	**12.02**	15.45
512	10000	4.59	3.75	**3.45**
2^{10}	10000	8.63	11.58	**7.40**
2^{11}	10000	18.51	35.48	**16.15**
2^{12}	1000	4.05	10.81	**3.50**
2^{14}	1000	24.54	—	**16.67**
2^{16}	100	19.03	—	**7.70**
2^{18}	10	8.21	—	**3.58**
2^{20}	10	33.49	—	**15.99**

Table 2: Benchmarks versus NTL

Table 2 shows the results of some benchmarking tests on a relatively modest desktop machine with a 2.5 GHz 64-bit Athalon processor, 256 KB L1 cache, 1 MB L2 cache, and 2 GB RAM. Each line in the table gives the size of each input polynomial to the multiplication, the number of randomly-chosen product computations, the CPU time (in seconds) for NTL multiplication, and finally the CPU time for our new Karatsuba-like and FFT-based multiplication algorithms, respectively. In our Karatsuba implementation, we set the crossover to the classical method at size 32, and we can see that the crossover from FFT to Karatsuba should be somewhere around 512. In NTL, these crossover points are 16 and 90, respectively.

To reiterate, the fundamental differences between our implementation and NTL make these comparisons somewhat meaningless, but these results are promising and seem to suggest that our space-efficient algorithms might be useful in practice. Of course, all of these benchmarks were performed on sizes which are powers of 2, precisely the cases where we expect our algorithms to gain the most advantage. More implementation work remains before we can determine in what cases (if any) our new routines are the best practical choice.

5. CONCLUSIONS

We have shown two new methods for multiplication which match existing "fast" algorithms in asymptotic time complexity, but need considerably less auxiliary storage space to compute the result. In particular, by breaking the traditional model and allowing multiple reads and writes into the output space, we have demonstrated that the $\Omega(n^2)$ time-space tradeoff lower bound for multiplication can be improved upon. Since this model is more realistic for modern architectures, and in fact is already being used elsewhere, our new algorithms may gain a practical advantage over existing approaches in some cases.

Much work remains to be done on this topic. First, while a straightforward adaptation of the space-efficient Karatsuba multiplication to the multiplication of multi-precision inte-

gers is not difficult, the extra challenges introduced by the presence of carries, combined with the extreme efficiency of existing libraries such as GMP [6], mean that an even more careful implementation would be needed to gain an advantage in this case. For instance, it would probably be better to use a subtractive version of Karatsuba's algorithm to avoid some carries. This is also likely the area of greatest potential utility of our new algorithms, as routines for long integer multiplication are used in many different areas of computing.

There are also some more theoretical questions left open here. One direction for further research would be to see if a scheme similar to the one presented here for Karatsuba-like multiplication with low space requirements could also be adapted to the Toom-Cook 3-way divide-and-conquer method, or even their arbitrary k-way scheme. One might also try to reduce the amount of extra space for Karatsuba multiplication below $O(\log n)$, or to remove some of the requirements of our FFT-based multiplication method that uses $O(1)$ extra space. Finally, a natural question is to ask whether polynomial or long integer multiplication can be done completely in-place — that is, transforming the input to the output using only a constant amount of extra space, such as what has been done for the FFT algorithm. Even allowing an exponential time complexity, it is not clear whether this is possible, nor is there any proof of its impossibility of which the author is aware.

6. REFERENCES

[1] Karl Abrahamson. Time-space tradeoffs for branching programs contrasted with those for straight-line programs. *Foundations of Computer Science, 1985., 27th Annual Symposium on*, pages 402–409, Oct. 1986.

[2] Richard Brent and Paul Zimmermann. Modern computer arithmetic. Online: `http://www.loria.fr/~zimmerma/mca/mca-0.2.pdf`, June 2008. Version 0.2.

[3] David G. Cantor and Erich Kaltofen. On fast multiplication of polynomials over arbitrary algebras. *Acta Inform.*, 28(7):693–701, 1991.

[4] Stephen A. Cook. *On the Mininum Computation Time of Functions*. PhD thesis, Harvard University, 1966.

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

[6] Torbjörn Granlund et. al. GNU Multiple Precision Arithmetic Library. Online: `http://gmplib.org/`, 2008.

[7] Martin Fürer. Faster integer multiplication. In *STOC '07: Proceedings of the thirty-ninth annual ACM symposium on Theory of computing*, pages 57–66, New York, NY, USA, 2007. ACM Press.

[8] Joachim von zur Gathen and Jürgen Gerhard. *Modern computer algebra*. Cambridge University Press, Cambridge, second edition, 2003.

[9] David Harvey. zn_poly: a library for polynomial arithmetic. Online: `http://www.cims.nyu.edu/~harvey/zn_poly/`, 2008.

[10] A. Karatsuba and Yu. Ofman. Multiplication of multidigit numbers on automata. *Dokl. Akad. Nauk SSSR*, 7:595–596, 1963.

[11] Roman Maeder. *Storage allocation for the Karatsuba integer multiplication algorithm*, pages 59–65. 1993.

[12] Michael Monagan. *In-place arithmetic for polynomials over Zn*, pages 22–34. 1993.

[13] A. Schönhage and V. Strassen. Schnelle Multiplikation grosser Zahlen. *Computing (Arch. Elektron. Rechnen)*, 7:281–292, 1971.

[14] Victor Shoup. NTL: A Library for doing Number Theory. Online: `http://www.shop.net/ntl/`, 2008.

[15] Clive Temperton. Self-sorting in-place fast fourier transforms. *SIAM Journal on Scientific and Statistical Computing*, 12(4):808–823, 1991.

[16] Emmanuel Thomé. Karatsuba multiplication of polynomials with temporary space of size $\leq$ n. Online: `http://www.loria.fr/~thome/publis/`, September 2002.

[17] A. L. Toom. The complexity of a scheme of functional elements realizing the multiplication of integers. *Dokl. Adad. Nauk. SSSR*, 150(3):496–498, 1963.

[18] Joris van der Hoeven. The truncated Fourier transform and applications. In *ISSAC 2004*, pages 290–296. ACM, New York, 2004.