

# Fast and Small: Multiplying Polynomials without Extra Space

Daniel S. Roche



Symbolic Computation Group  
School of Computer Science  
University of Waterloo



CECM Day  
SFU, Vancouver, 24 July 2009

# Preliminaries

We study algorithms for **univariate polynomial multiplication**:

## The Problem

**Given:** A ring  $R$ , an integer  $n$ ,  
and  $f, g \in R[x]$  with degrees less than  $n$

**Compute:** Their product  $f \cdot g \in R[x]$

## The Model

- Ring operations have unit cost
- Random reads from input, random **reads**/writes to output
- Space complexity determined by size of auxiliary storage

# Univariate Multiplication Algorithms

	<b>Time Complexity</b>	<b>Space Complexity</b>
<b>Classical Method</b>	$O(n^2)$	$O(1)$
<b>Divide-and-Conquer</b> Karatsuba/Ofman '63	$O(n^{\log_2 3})$ or $O(n^{1.59})$	$O(n)$
<b>FFT-based</b> Schönhage/Strassen '71 Cantor/Kaltofen '91	$O(n \log n \log \log n)$	$O(n)$

# Univariate Multiplication Algorithms

	<b>Time Complexity</b>	<b>Space Complexity</b>
<b>Classical Method</b>	$O(n^2)$	$O(1)$
<b>Divide-and-Conquer</b> Karatsuba/Ofman '63	$O(n^{\log_2 3})$ or $O(n^{1.59})$	$O(n)$
<b>FFT-based</b> Schönhage/Strassen '71 Cantor/Kaltofen '91	$O(n \log n \log \log n)$	$O(n)$

Goal: Keep time complexity the same, reduce space

# The Evolution of Multiplication

**Small and slow**



# The Evolution of Multiplication

**Big and fast**



# The Evolution of Multiplication

**Small and fast**



## Previous Work

- **Savage & Swamy 1979**  $O(n^2)$  time-space lower bound for straight line programs
- **Abrahamson 1985**:  $O(n^2)$  time-space lower bound for branching programs



## Previous Work

- **Savage & Swamy 1979**  $O(n^2)$  time-space lower bound for straight line programs
- **Abrahamson 1985**:  $O(n^2)$  time-space lower bound for branching programs
- **Monagan 1993**: Importance of space efficiency for multiplication over  $\mathbb{Z}_p[x]$
- **Maeder 1993**: Bounds extra space for Karatsuba multiplication so that storage can be preallocated — about  $2n$  extra memory cells required.
- **Thomé 2002**: Karatsuba multiplication for polynomials using  $n$  extra memory cells.

# Present Contributions

- New Karatsuba-like algorithm with  $O(\log n)$  space
- New FFT-based algorithm with  $O(1)$  space  
*under certain conditions*
- Implementations in C over  $\mathbb{Z}/p\mathbb{Z}$

# Standard Karatsuba Algorithm

**Idea:** Reduce one degree- $2k$  multiplication to three of degree  $k$ .

- Originally noticed by Gauss (multiplying complex numbers), rediscovered and formalized by Karatsuba & Ofman

**Input:**  $f, g \in \mathbb{R}[x]$  each with degree less than  $2k$ .

Write  $f = f_0 + f_1x^k$  and  $g = g_0 + g_1x^k$ .



# Low-Space Karatsuba Algorithms

Version "0"

**Read-Only Input Space:**



**Read/Write Output Space:**



**To Compute:**  $f \cdot g$

# Low-Space Karatsuba Algorithms

Version "1"

- 1 The low-order coefficients of the output are initialized as  $h$ , and the product  $f \cdot g$  is added to this.

**Read-Only Input Space:**



**Read/Write Output Space:**



**To Compute:**  $f \cdot g + h$

# Low-Space Karatsuba Algorithms

Version "2"

- 1 The low-order coefficients of the output are initialized as  $h$ , and the product  $f \cdot g$  is added to this.
- 2 The first polynomial  $f$  is given as a sum  $f^{(0)} + f^{(1)}$ .

**Read-Only Input Space:**



**Read/Write Output Space:**



**To Compute:**  $(f^{(0)} + f^{(1)}) \cdot g + h$

# Dirty Details

Restrict modulus to 29 bits to allow for delayed reductions

## In the Karatsuba step

- Only 4 values are added/subtracted in one position
- Delay reductions, perform two “corrections”

## Classical algorithm

- Switch over at  $n \leq 32$  (determined experimentally)
- Perform arithmetic in double-precision `long` `longs`; delay reductions (a la Monagan)

## Problem: code explosion

3 “versions” of algorithms (based on extra constraints)

×

Karatsuba or classical

×

odd-sized or even-sized operands

×

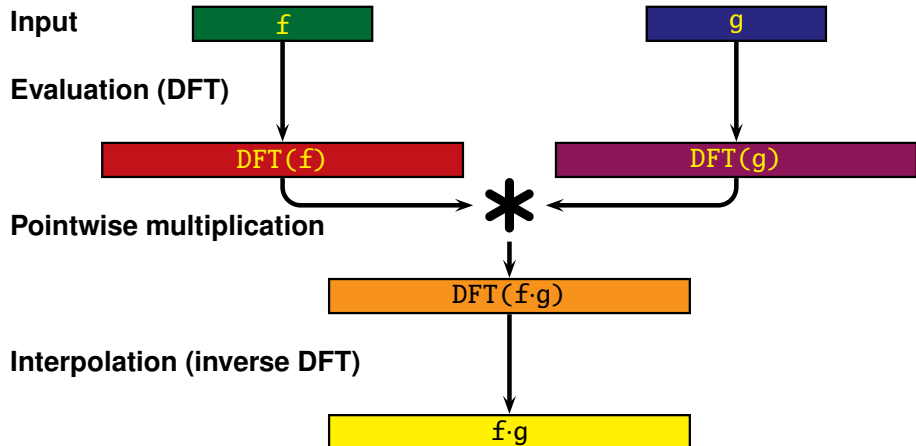
equal-sized operands or “one different”

**Solution:** Use “supermacros” in C:

Same file is included multiple times with some parameter values changed (crude form of code generation).



# DFT-Based Multiplication



# Simplifying Assumptions

## From now on:

- $\deg f + \deg g < n = 2^k$  for some  $k \in \mathbb{N}$
- The base ring  $R$  contains a  $2^k$ -PRU  $\omega$

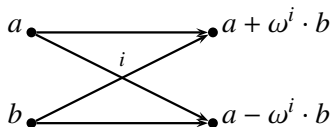
That is, assume “virtual roots of unity” have already been found, and optimize from there.

# Usual Formulation of the FFT

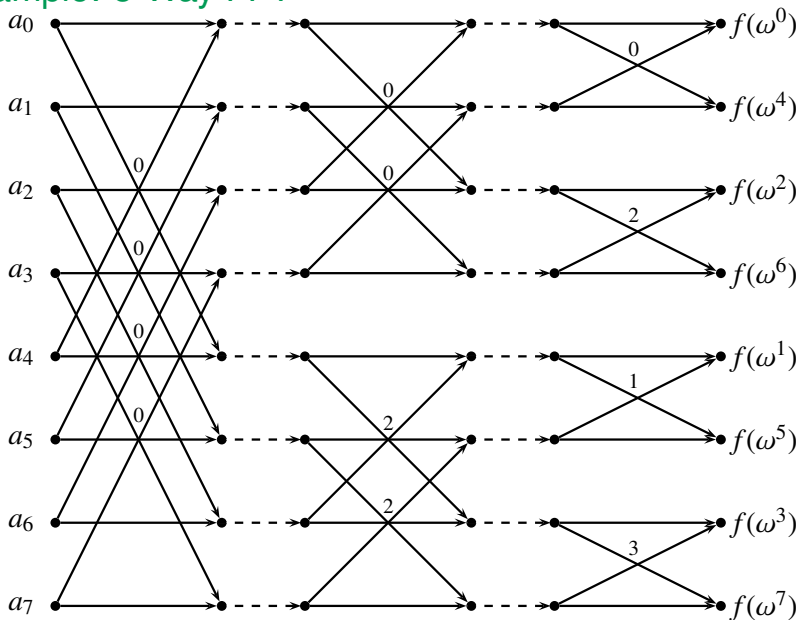
Perform two  $\frac{n}{2}$ -DFTs followed by  $\frac{n}{2}$  2-DFTs:

- Write  $f(x) = f_{\text{even}}(x^2) + x \cdot f_{\text{odd}}(x^2)$   
(i.e.  $\deg f_{\text{even}}, \deg f_{\text{odd}} < n/2$ )
- Compute  $\text{DFT}_{\omega^2}(f_{\text{even}})$  and  $\text{DFT}_{\omega^2}(f_{\text{odd}})$
- Compute each  $f(\omega^i) = f_{\text{even}}(\omega^{2i}) + \omega \cdot f_{\text{odd}}(\omega^{2i})$

Make use of “butterfly circuit” for each size-2 DFT:

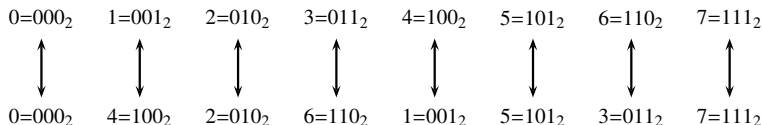


## Example: 8-Way FFT



# Reverted Binary Ordering

In-Place FFT permutes the ordering into **reverted binary**:



**Problem:** Powers of  $\omega$  are not accessed in order

Possible solutions:

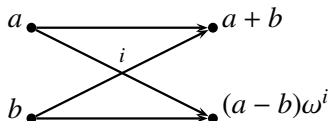
- Precompute all powers of  $\omega$  — **too much space**
- Perform steps out of order — **terrible for cache**
- Permute input before computing — **costly**

# Alternate Formulation of FFT

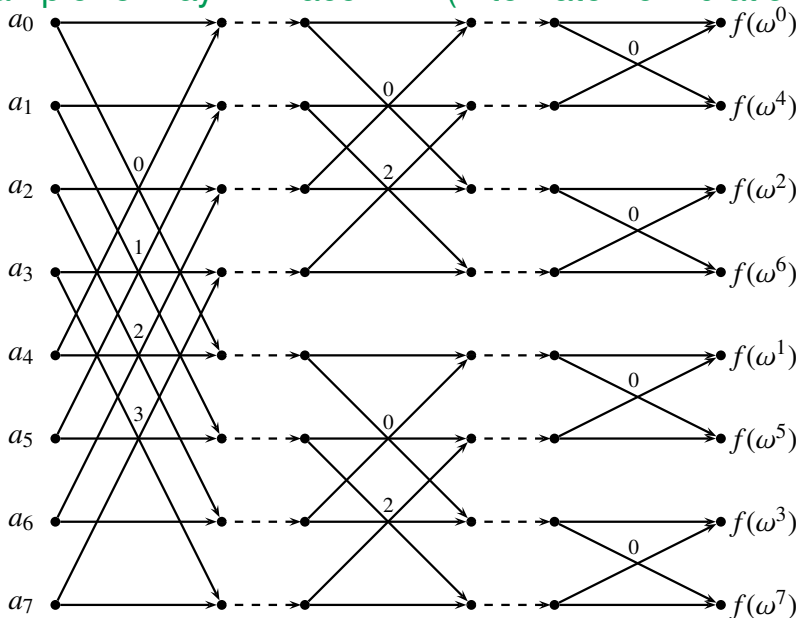
Perform  $\frac{n}{2}$  2-DFTs followed by two  $\frac{n}{2}$ -DFTs

- Write  $f = f_{\text{low}} + x^{n/2} \cdot f_{\text{high}}$
- Compute  $f_0 = f_{\text{low}} + f_{\text{high}}$  and  $f_1 = f_{\text{low}}(\omega x) - f_{\text{high}}(\omega x)$
- Compute each  $f(\omega^{2i}) = f_0(\omega^{2i})$  and  $f(\omega^{2i+1}) = f_1(\omega^{2i})$

Modified “butterfly circuit”:



## Example: 8-Way In-Place FFT (Alternate Formulation)



## Folded Polynomials

Recall the basis for the “alternate” FFT formulation:

$$\begin{aligned}f_0 &= f_{\text{low}} + f_{\text{high}} \\f_1 &= f_{\text{low}}(\omega x) - f_{\text{high}}(\omega x)\end{aligned}$$

A generalization (recalling that  $n = 2^k$ ):

### Definition (Folded Polynomials)

$$f_i = f(\omega^{2^{i-1}} x) \quad \text{rem } x^{2^{k-i}} - 1$$

### Theorem

$$f(\omega^{2^i(2j+1)}) = f_{i+1}(\omega^{2^{i+1}j})$$

So by computing each  $f_i$  at all powers of  $\omega^i$ , we get the values of  $f$  at all powers of  $\omega$ .



# Recursively Applying the Alternate Formulation

Example (Reverted Binary Ordering of  $0, 1, \dots, 15$ )

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

$\text{DFT}_\omega(f)$  in binary reversed order  
can be computed by DFTs of  $f_i$ s:

$\text{DFT}_\omega(f)$

=

...  $\text{DFT}_{\omega^8}(f_3)$   $\text{DFT}_{\omega^4}(f_2)$   $\text{DFT}_{\omega^2}(f_1)$

# FFT-Based Multiplication without Extra Space

**Idea:** Solve half of remaining problem at each iteration

f

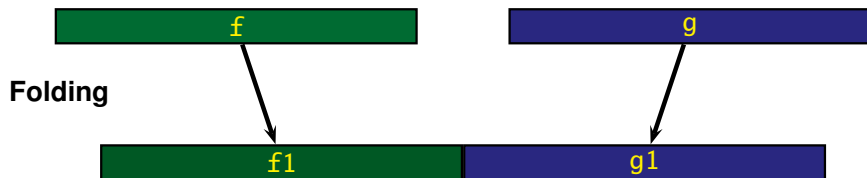
g

**Input**

(empty)

# FFT-Based Multiplication without Extra Space

**Idea:** Solve half of remaining problem at each iteration



# FFT-Based Multiplication without Extra Space

**Idea:** Solve half of remaining problem at each iteration

$f$

$g$

**In-Place FFTs (alternate formulation)**

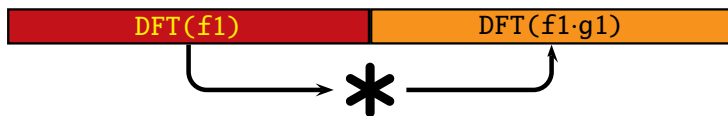
$DFT(f_1)$   $DFT(g_1)$

# FFT-Based Multiplication without Extra Space

**Idea:** Solve half of remaining problem at each iteration

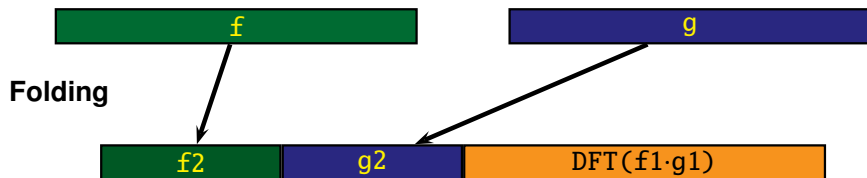


**Pointwise Multiplication**



# FFT-Based Multiplication without Extra Space

**Idea:** Solve half of remaining problem at each iteration



# FFT-Based Multiplication without Extra Space

**Idea:** Solve half of remaining problem at each iteration



**In-Place FFTs (alternate formulation)**

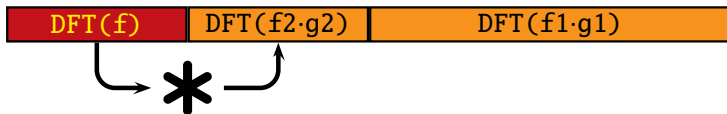


# FFT-Based Multiplication without Extra Space

**Idea:** Solve half of remaining problem at each iteration



**Pointwise Multiplication**





# FFT-Based Multiplication without Extra Space

**Idea:** Solve half of remaining problem at each iteration



**( $k$  iterations)**



# FFT-Based Multiplication without Extra Space

**Idea:** Solve half of remaining problem at each iteration

$f$

$g$

**In-Place Reverse FFT (usual formulation)**

$f \cdot g$

# Analysis

Time cost of the various stages:

- **Folding:**  $O(n)$  cost times  $\log n$  folds =  $O(n \log n)$
- **FFTs:**  $O(m \log m)$  for  $m = n, n/2, n/4, \dots, 1 = O(n \log n)$
- **Multiplications:**  $n/2 + n/4 + \dots + 1 = O(n)$

Total cost:  $O(n \log n)$  time and  $O(1)$  extra space  
when the following conditions hold:

- $n = \deg f + \deg g + 1$  is a power of 2
- $R$  contains an  $n$ -PRU  $\omega$

# Modular Arithmetic

Use floating-point Barrett reduction (from NTL):

- Pre-compute an approximation of  $1/p$
- Given  $a, b \in \mathbb{Z}_p$ , compute an approximation of  $q = \lfloor a \cdot b \cdot (1/p) \rfloor$
- Then  $ab - qp$  equals  $ab \bmod p$  plus or minus  $p$ .

The cost of this method:

- 2 double multiplications
- 2 int multiplications
- 1 int subtraction
- 3 conversions between int and double
- 2 “correction” steps to get exact result  
↪ not necessary until the very end!

# Implementation Benchmarking

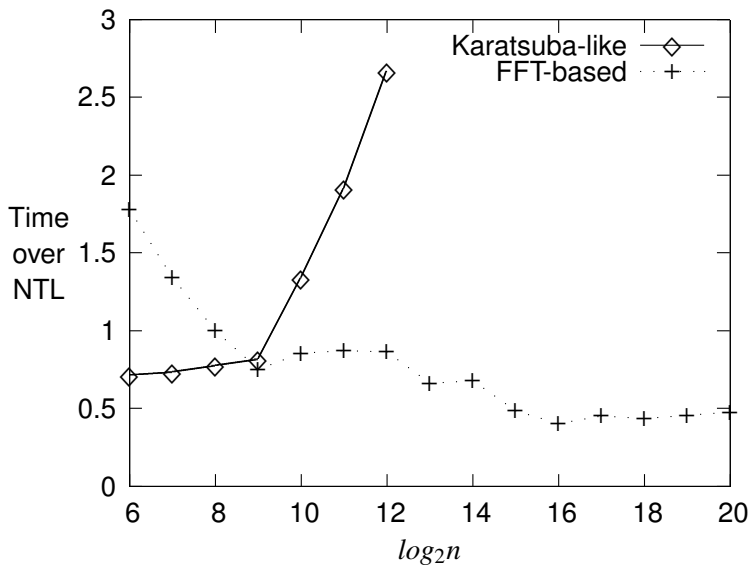
Details of tests:

- 2.5 GHz 64-bit Athalon, 256KB L1, 1MB L2, 2GB RAM
- $p = 167772161$  (28 bits)
- Comparing CPU time (in seconds) for the computation

## Disclaimer

We are comparing apples to oranges.

# Timing Benchmarks



# Future Directions

- Efficient implementation over  $\mathbb{Z}$  (GMP)
- Similar results for  
Toom-Cook 3-way or  $k$ -way
- What modulus bit restriction is “best”?
- Is completely in-place (overwriting input) possible?