Fast Integer Multiplication with Schönhage-Strassen's Algorithm

Alexander Kruppa

CACAO team at LORIA, Nancy

séminaire Algorithms INRIA Rocquencourt

Contents

Contents of this talk:

1. Basics of Integer Multiplication
   (a) by polynomial multiplication
   (b) Evaluation/Interpolation
   (c) Karatsuba's method
   (d) Toom-Cook method
   (e) FFT and weighted transforms
   (f) Schönhage-Strassen's Algorithm

Integer Multiplication

- Problem: given two n-word (word base β) integers

$$a = \sum_{i=0}^{n-1} a_i \beta^i,$$

$0 \leq a_i < \beta$ and likewise for b, compute

$$\begin{aligned} c = ab &= \sum_{i=0}^{2n-1} c_i \beta^i \\ &= \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} a_i b_j \beta^{i+j}, \end{aligned}$$

with $0 \leq c_i < \beta$.

by Polynomial Multiplication

- We can rewrite the problem as polynomial arithmetic:

$$A(x) = \sum_{i=0}^{n-1} a_i x^i$$

so that $a = A(\beta)$, likewise for $B(x)$, then

$$\begin{aligned} C(x) &= A(x)B(x) \\ &= \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} a_i b_j x^{i+j} \end{aligned}$$

so that $c = ab = C(\beta)$.

- Double sum has complexity $O(n^2)$ (Grammar School Algorithm), we can do much better

Evaluation/Interpolation

- Unisolvence Theorem: Polynomial of degree $d-1$ is determined uniquely by values at d distinct points

- Since $C(k) = A(k)B(k)$ for all $k \in R$ for ring R, reduce the polynomial multiplication to:
  1. Evaluate $A(x)$, $B(x)$ at $2n-1$ points $k_0, \ldots, k_{2n-2}$
  2. Pairwise multiply to get $C(k_i) = A(k_i)B(k_i)$
  3. Interpolate $C(x)$ from its values $C(k_i)$

Karatsuba's Method

- First algorithm to use this principle (Karatsuba and Ofman, 1962)
- Multiplies polynomials of degree 1: $A(x) = a_0 + a_1 x$
- Suggested points of evaluation: $0, 1, \infty$
- $A(0) = a_0, A(1) = a_0 + a_1, A(\infty) = a_1$ (same for $B(x)$)
- $C(0) = a_0 b_0, C(1) = (a_0 + a_1)(b_0 + b_1), C(\infty) = a_1 b_1$
- $c_0 = C(0), c_2 = C(\infty), c_1 = C(1) - c_0 - c_2$
- Product of $2n$ words computed with 3 pointwise multiplications of n words each, applied recursively: $O(n^{\log_2(3)}) = O(n^{1.585})$

Toom-Cook Method

- Generalized to polynomials of larger degree (Toom, 1963, Cook, 1966)
- Product of two n word integers with $A(x)$, $B(x)$ of degree d: $2d+1$ products of $n/(d+1)$ word integers
- For fixed d: complexity $O(n^{\log_{d+1}(2d+1)})$, e.g. $d = 2$: $O(n^{1.465})$
- Interpolation/Evaluation costly ($O(dn\log(d))$), cannot increase d arbitrarily for given n
- Choosing d as function of n allows algorithm in $O(n^{1+\epsilon})$, for any $\epsilon > 0$. Small exponents need very large n

Evaluation/Interpolation with FFT

- FFT solves problem of costly evaluation/interpolation
- Length-ℓ DFT of $a_0, ..., a_{\ell-1}$ in R computes $\tilde{a}_j = A(\omega_\ell^j)$, $0 \leq j < \ell$, with ω_ℓ an ℓ-th principal root of unity in R: ℓ-point polynomial evaluation
- Length-ℓ IDFT computes a_i from given $\tilde{a}_j$: ℓ-point polynomial interpolation
- With FFT algorithm, algebraic complexity only $O(\ell \log(\ell))$
- Problem: R needs to support length-ℓ FFT (preferably ℓ a power of 2): needs ℓ-th principal root of unity, ℓ a unit

Weighted Transform

- Since $(\omega_\ell^j)^\ell = 1$ for all $j \in \mathbb{N}$, $C_1(x)x^\ell + C_0(x)$ has same DFT coefficients as $C_1(x) + C_0(x)$: implicit modulus $x^\ell - 1$ in DFT

- FFT convolution gives $C(x) = (A(x)B(x)) \bmod (x^\ell - 1)$: cyclic convolution

- Can change that modulus with weighed transform: compute $C(wx) = (A(wx)B(wx)) \bmod (x^\ell - 1)$. Then

$$\begin{aligned} A(wx)B(wx) &= C_1(wx)x^\ell w^\ell + C_0(wx) \\ C(wx) &= C_1(wx)x^\ell w^\ell + C_0(wx) \bmod (x^\ell - 1) \\ &= C_1(wx)w^\ell + C_0(wx) \end{aligned}$$

  so that $C(x) = (A(x)B(x)) \bmod (x^\ell - w^\ell)$

- With $w^\ell = -1$, we get modulus $x^\ell + 1$: negacyclic convolution, but need 2ℓ-th root of unity in R

Schönhage-Strassen's Algorithm: Basic Idea

- First algorithms to use FFT (Schönhage and Strassen 1971)
- Uses ring $R_n = \mathbb{Z}/(2^n+1)\mathbb{Z}$ for transform, with $\ell = 2^k \mid n$
- Then $2^{n/\ell} \equiv -1 \pmod{2^n+1}$: so $2^{n/\ell} \in R_n$ is 2ℓ-th root of unity, multiplication by powers of 2 is fast! ($O(n)$)
- Allows length ℓ weighted transform for negacyclic convolution
- Write input $a = A(2^M)$, $b = B(2^M)$, compute $C(x) = A(x)B(x) \pmod{x^\ell + 1}$. Then $c = C(2^M) = ab \pmod{2^{M\ell}+1}$
- Point-wise products modulo $2^n + 1$ use SSA recursively: choose next level's ℓ', M' so that $M'\ell' = n$

Improvements to Schönhage-Strassen’s Algorithm

Motivation for Improving SSA

- Integer multiplication is fundamental to arithmetic, used in PRP testing, ECPP, polynomial multiplication
- Schönhage-Strassen's algorithm [SSA]: good asymptotic complexity $O(n \log n \log \log n)$, fast in practice for large operands, exact (only integer arithmetic)
- Used in GMP, widely deployed
- We improved algorithmic aspects of Schönhage-Strassen
- Validated by implementation based on GMP 4.2.1 [GMP]

Schönhage-Strassen's Algorithm

- SSA reduces multiplication of two S-bit integers to ℓ multiplications of approx. $4S/\ell$-bit integers

- Example: multiply two numbers a, b of 2^{20} bits each $\Rightarrow$ product has at most 2^{21} bits

  1. Choose $N = 2^{21}$ and a good ℓ, for this example $\ell = 512$. We compute $ab \bmod (2^N + 1)$

  2. Write a as polynomial of degree ℓ, coefficients $a_i < 2^M$ with $M = N/\ell$, $a = a(2^M)$. Same for b

  3. $ab = a(2^M)b(2^M) \bmod (2^N + 1)$, compute $c(x) = a(x)b(x) \bmod (x^\ell + 1)$

  4. Convolution theorem: Fourier transform and pointwise multiplication

5. FFT needs ℓ-th root of unity: map to $\mathbb{Z}/(2^n+1)\mathbb{Z}[x]$ with $\ell \mid n$. Then $2^{2n/\ell}$ has order ℓ

6. We need $2^n + 1 > c_i$: choose $n \geq 2M + \log_2(\ell) + 1$

7. Compute $c(x) = a(x)b(x) \bmod (x^\ell + 1)$, evaluate $ab = c(2^M)$ - and we're done!

8. Benefits:
   - Root of unity is power of 2
   - Reduction $\bmod(2^n + 1)$ is fast
   - Point-wise products can use SSA recursively without padding

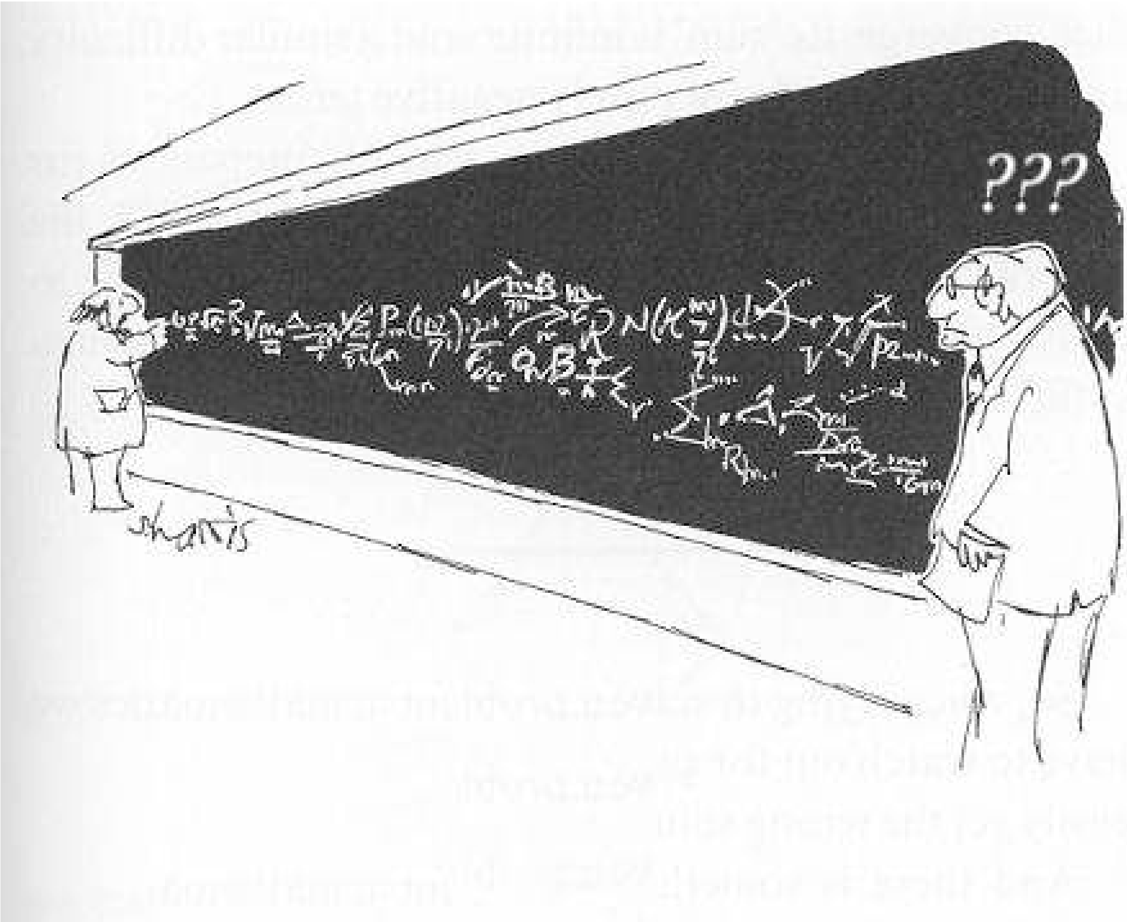
$$\mathbb{Z} \Longrightarrow \mathbb{Z}_{2^N+1} \Longrightarrow \mathbb{Z}[x] \bmod (x^\ell + 1) \Longrightarrow \mathbb{Z}_{2^n+1}[x] \bmod (x^\ell + 1) \Longrightarrow \mathbb{Z}_{2^n+1}$$

No, recurse

n small?

Yes, multiply

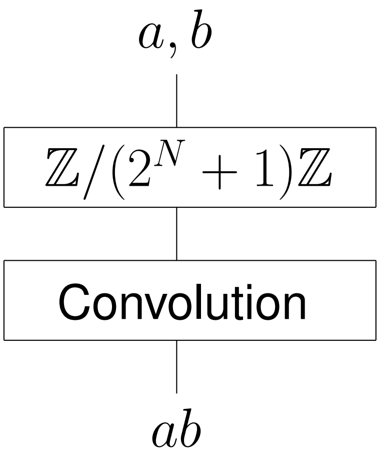
High-Level Optimizations



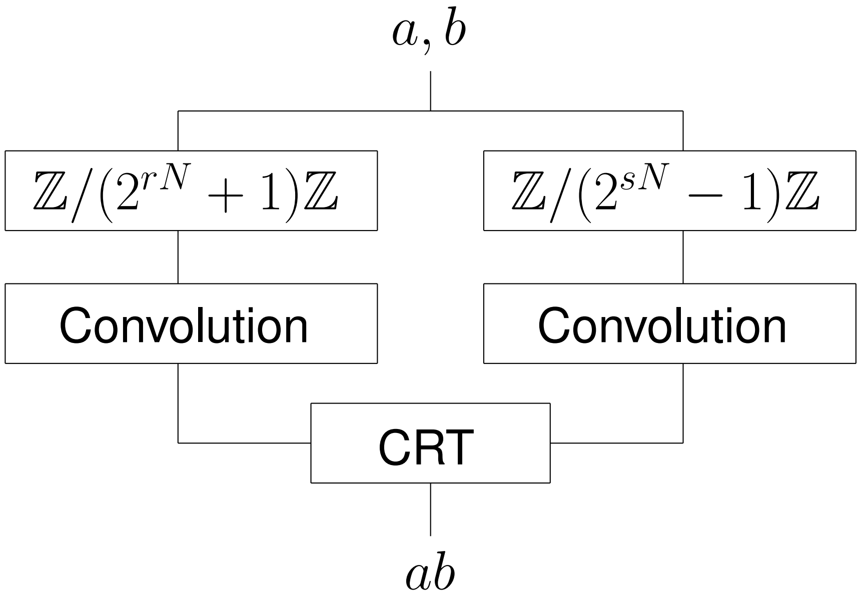
Mersenne Transform

- Convolution theorem implies reduction $\bmod (x^\ell - 1)$
- Convolution $\bmod(x^\ell + 1)$ needs weights θ^i with ord$(\theta) = 2\ell$, needs $\ell \mid n$ to get 2ℓ-th root of unity in R_n
- Computing $ab \bmod (2^N + 1)$ to allows recursive use of SSA, but is *not* required at top level
- Map a and b to $\mathbb{Z}/(2^N - 1)\mathbb{Z}$ instead:
  compute $c(x) = a(x)b(x) \bmod (x^\ell - 1)$
- Condition relaxes to $\ell \mid 2n$. Twice the transform length, smaller n
- No need to apply/unapply weights

CRT Reconstruction



with $2^N + 1 > ab$



with $(2^{rN} + 1)(2^{sN} - 1) > ab$

- At least one of $(2^{rN}-1, 2^{sN}+1)$ and $(2^{rN}+1, 2^{sN}-1)$ is coprime
- Our implementation uses $(2^{rN}+1, 2^{N}-1)$: always coprime, good speed
- Smaller convolution, finer-grained parameter selection

The $\sqrt{2}$ Trick

- If $4 \mid n$, 2 is a quadratic residue in $\mathbb{Z}/(2^n+1)\mathbb{Z}$
- In that case, $\sqrt{2} \equiv 2^{3n/4} - 2^{n/4}$: simple form, multiplication by $\sqrt{2}$ takes only 2 shift, 1 subtraction modulo 2^n+1
- Offers root of unity of order $4n$, allows $\ell \mid 2n$ for Fermat transform, $\ell \mid 4n$ for Mersenne transform
- Sadly, higher roots of 2 usually not available in $\mathbb{Z}/(2^n+1)\mathbb{Z}$, or have no simple form

Low-Level Optimizations

Arithmetic modulo $2^n + 1$

- Residues stored semi-normalized ($< 2^{n+1}$), each with $m = n/w$ full words plus one extra word ≤ 1

- Adding two semi-normalized values:

```
c = a[m] + b[m] + mpn_add_n (r, a, b, m);
r[m] = (r[0] < c);
MPN_DECR_U (r, m + 1, c - r[m]);
```

  Assures `r[m]` $= 0$ or 1, `c` $-$ `r[m]` $>$ `r[0]` $\Rightarrow$ `r[m]` $= 1$ so carry propagation must terminate.

- Conditional branch only in `mpn_add_n` loop and (almost always non-taken) in carry propagation of `MPN_DECR_U`

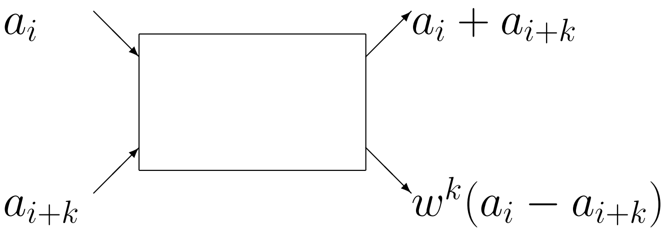
- Similar for subtraction, multiplication by 2^k

Improving Cache Locality

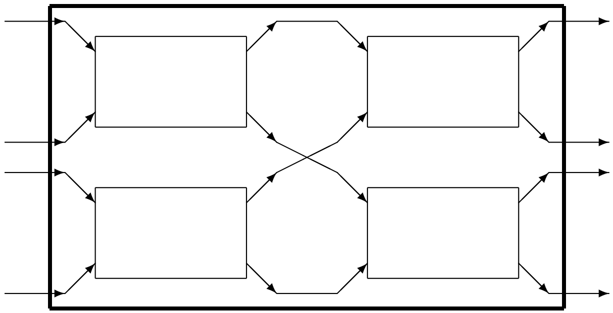
- SSA behaves differently than, say, complex floating point FFT: elements have hundreds or thousands of bytes instead of 16

- Recursive implementation preferable, reduces working data set size quickly, overhead small compared to arithmetic

- Radix 4 transform fuses two levels of butterflies on four inputs. Half as many recursion levels, 4 operands usually fit into level 1 cache



Radix-2 butterfly



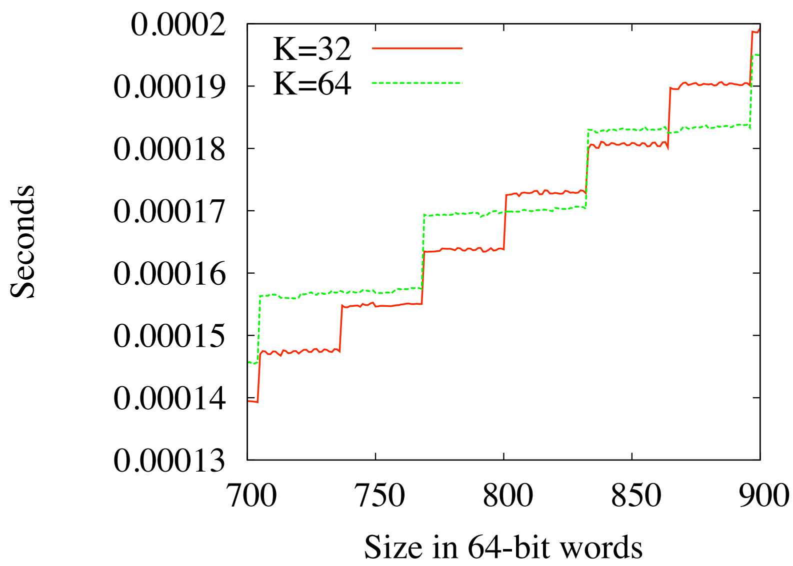
Radix-4 butterfly

- Bailey's 4-step algorithm (radix $\sqrt{\ell}$ transform)
  - groups half of recursive levels into first pass, other half into second pass
  - Each pass is again a set of FFTs, each of length $\sqrt{\ell}$
  - If length $\sqrt{\ell}$ transform fits in level 2 cache: only two passes over memory per transfrom
  - Extremely effective for complex floating-point FFTs, we found it useful for SSA with large ℓ as well
- Fusing different stages of SSA
  - Do as much work as possible on data while it is in cache
  - When cutting input a into M-bit size coefficients $a = \sum_{0 \leq i < \ell} a_i 2^{iM}$, also apply weights for negacyclic transform and perform first FFT level (likewise for b)
  - Similar ideas for other stages

Fine-Grained Tuning

- Up to version 4.2.1, GMP uses simple tuning scheme: transform length grows monotonously with input size

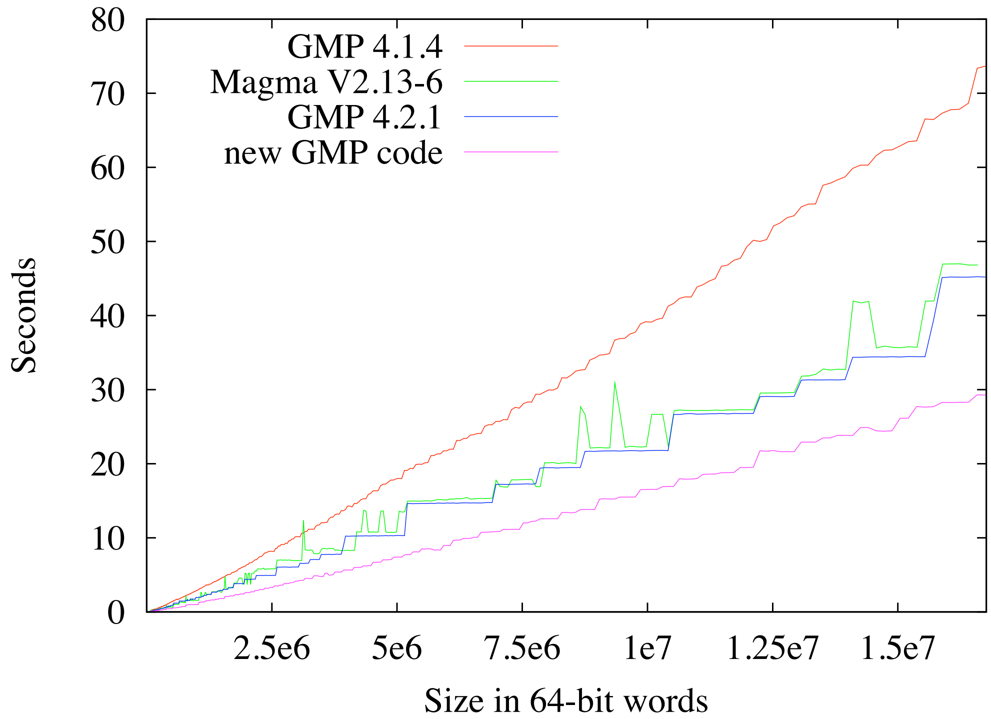
- Not optimal: time over input size graphs for different transform lengths intersect multiple times:



- New tuning scheme determines intervals of input size and optimal transform length
- Also determines pairs of Mersenne/Fermat transform lenghts for CRT
- Time-consuming (ca. 1h up to 1M words) but yields significant speedup

Timings and Comparisons

- Our code is about 40% faster than GMP 4.2.1 and Magma 2.13-6, more than twice as fast as GMP 4.1.4



- New code by William Hart for FLINT is competitive with ours, up to 30% faster for some input sizes

- Prime95 and Glucas implement complex floating point FFT for integer multiplication, mostly for arithmetic mod $2^n - 1$ (Lucas-Lehmer test for Mersenne numbers)

  - Considerably faster: Prime95 10x on Pentium 4, 2.5x on Opteron; Glucas 5x on Pentium 4, 2x on Opteron

  - Danger of round-off error due to floating point arithmetic

  - Provably correct rounding possible with about 2x the transform length

  - Prime95 written in assembly, non-portable

Untested Ideas

- Special code for point-wise multiplication:
  - Length $3 \cdot 2^k$ transform
  - Karatsuba and Toom-Cook with reduction mod $2^n + 1$ in interpolation step
  - Short-length, proven correct complex floating-point FFT
- Truncated Fourier transform
- Fürer's new algorithm has lower theoretical complexity $O(n \log(n) 2^{\log^*(n)})$. How fast is it in practice?

References

[GMP] T. Granlund, *The GNU Multiple Precision Arithmetic library*, `http://gmplib.org/`

[SSA] A. Schönhage and V. Strassen, *Schnelle Multiplikation großer Zahlen*, Computing 7 (1971)

Source tarball with new code available at

`<http://www.loria.fr/~kruppaal>`