Even faster integer multiplication

David Harvey

University of New South Wales

22nd May 2015
NICTA

(joint work with Joris van der Hoeven and Grégoire Lecerf, École polytechnique)

Let $\mathrm{M}(n) =$ cost of multiplying n-bit integers.

Complexity model is *deterministic bit complexity*.

(e.g. number of Boolean operations, or number of steps on a multi-tape Turing machine)

Primary school long multiplication algorithm:

$$\mathrm{M}(n) = O(n^2).$$

Karatsuba (1962) divide-and-conquer:

$$\mathrm{M}(n) = O(n^{\lg 3/\lg 2}) \approx O(n^{1.58}).$$

Schönhage–Strassen (1971) using fast Fourier transforms (FFTs):

$$\mathrm{M}(n) = O(n \lg n \lg \lg n).$$

[$\lg$ = base 2 logarithm]

Fürer (2007) using very clever FFTs:

$$\mathrm{M}(n) = O(n \lg n \, K^{\lg^* n})$$

for some unspecified constant $K > 1$.

Here $\lg^*$ is the iterated logarithm:

$$\lg^* x = \begin{cases} 0 & \text{if } x \le 1, \\ 1 + \lg^*(\lg x) & \text{if } x > 1. \end{cases}$$

It is a **very** slowly growing function!!! Example:

$$\lg^*(2^{2^{65536}}) = 6.$$

So $K^{\lg^* n}$ grows much more slowly than $\lg \lg n$.

Theorem (H.–van der Hoeven–Lecerf, 2014)

$$\mathsf{M}(n) = O(n \lg n \, 8^{\lg^* n}).$$

In other words, we can achieve $K = 8$ in Fürer's bound.

Muttering from audience...

"So what?
You optimised Fürer's algorithm?
Why should I care?"

Actually, our algorithm is different to Fürer's:

- Our algorithm can be adapted to polynomial multiplication in $\mathbf{F}_p[x]$. To our knowledge, Fürer's cannot.
- We tried pretty hard to optimise Fürer's algorithm. The best we could get is $K = 16$. I challenge anyone to beat this!
- Our algorithm is (IMHO) simpler than Fürer's.

In the rest of this talk I will give a sketch of the new algorithm.

Recall the usual *discrete Fourier transform* (DFT) over $\mathbf{C}$:

Given an input vector

$$a = (a_0, \ldots, a_{N-1}) \in \mathbf{C}^N,$$

the DFT is defined to be

$$\hat{a} = (\hat{a}_0, \ldots, \hat{a}_{N-1}) \in \mathbf{C}^N$$

where

$$\hat{a}_k = \sum_{j=0}^{N-1} \omega^{jk} a_j, \qquad \omega = e^{2\pi i/N}.$$

Example with $N = 8$:

$$\begin{array}{ccc} a \in \mathbf{C}^8 & & \hat{a} \in \mathbf{C}^8 \\ \begin{pmatrix} 0.21644849 - 0.23665015i \\ -0.69682296 + 0.97728396i \\ 0.32530267 + 0.50362843i \\ 0.41696975 - 0.77450417i \\ -0.11835476 - 0.13146835i \\ -0.06843155 + 0.55582899i \\ -0.89669982 - 0.83208469i \\ 0.47597590 + 0.39679403i \end{pmatrix} & \longmapsto & \begin{pmatrix} -0.34561227 + 0.45882805i \\ -0.87330681 + 1.75700379i \\ -1.24133222 - 1.69786242i \\ 2.60335181 - 2.93949434i \\ -0.60099456 - 1.85197756i \\ -1.12851292 + 0.47663759i \\ 2.58031397 + 1.61853792i \\ 0.73768096 + 0.28512574i \end{pmatrix} \end{array}$$

$$\omega = e^{2\pi i/8} = 0.70710678 + 0.70710678i$$

We cannot represent complex numbers exactly on a computer.

The input vector a may be considered to be exact, but the output $\hat{a}$ has been rounded to about 8 significant digits.

It is well known that the DFT can be computed using $O(N \lg N)$ operations in $\mathbf{C}$.

Such an algorithm is known as a *fast Fourier transform* (FFT).

Example: the Cooley–Tukey divide-and-conquer algorithm (1968).

But we are interested in **bit complexity**.

Suppose we work with complex numbers with P significant digits.

Arithmetic operations in $\mathbf{C}$ do not take unit time:

- Addition/subtraction in $\mathbf{C}$ has cost $O(P)$.
- We can simulate multiplication in $\mathbf{C}$ by using *integer* multiplication. So each multiplication in $\mathbf{C}$ has cost $O(\mathsf{M}(P))$.

Therefore the cost of the classical FFT algorithm becomes

$$O(N \lg N \ \mathsf{M}(P)),$$

because we have to do $O(N \lg N)$ multiplications at precision P.

I will now demonstrate that this is *suboptimal*.

There is an alternative algorithm that is **faster than the FFT**.

First: use Bluestein's trick (1970) to convert the DFT to a convolution:

$$\hat{a}_k = \sum_{j=0}^{N-1} \omega^{jk} a_j = \omega^{k^2/2} \sum_{j=0}^{N-1} \omega^{-(k-j)^2/2} (\omega^{j^2/2} a_j).$$

The last sum is $\omega^{k^2/2}$ times the k-th coefficient of the convolution of the sequences $b_j = \omega^{-j^2/2}$, $a'_j = \omega^{j^2/2} a_j$.

Example: we want the convolution of

$$a' = (\omega^{j^2/2} a_j)_{j=0}^{7} \in \mathbf{C}^8 \qquad \text{and} \qquad b = (\omega^{-j^2/2})_{j=-7}^{7} \in \mathbf{C}^{15}$$

$$\begin{pmatrix} 0.21644849 - 0.23665015i \\ -1.01777085 + 0.63623004i \\ -0.50362843 + 0.32530267i \\ -0.68161974 + 0.55598113i \\ -0.11835476 - 0.13146835i \\ 0.27592905 - 0.48733140i \\ 0.83208469 - 0.89669982i \\ 0.28789789 + 0.54873797i \end{pmatrix} * \begin{pmatrix} 0.92387953 - 0.38268343i \\ 0.00000000 - 1.00000000i \\ -0.92387953 + 0.38268343i \\ 1.00000000 + 0.00000000i \\ -0.92387953 + 0.38268343i \\ 0.00000000 - 1.00000000i \\ 0.92387953 - 0.38268343i \\ 1.00000000 + 0.00000000i \\ 0.92387953 - 0.38268343i \\ 0.00000000 - 1.00000000i \\ -0.92387953 + 0.38268343i \\ 1.00000000 + 0.00000000i \\ -0.92387953 + 0.38268343i \\ 0.00000000 - 1.00000000i \\ 0.92387953 - 0.38268343i \end{pmatrix}$$

Use *Kronecker substitution* to convert this convolution to one enormous (complex) integer multiplication of $O(NP)$ bits, i.e., by packing the coefficients together with appropriate zero-padding.

$$\left(\begin{array}{r}21644848\\999999999898222914\\999999999949637156\\999999999931838025\\999999999988164524\\000000000027592905\\000000000083208469\\000000000028789789\end{array} + \begin{array}{r}-23665014\\999999999936376995\\999999999967469732\\999999999944401887\\000000000013146835\\000000000048733140\\000000000089669981\\999999999945126203\end{array}\, i\right) * \left(\begin{array}{r}92387952\\999999999999999999\\999999999907612047\\000000000099999999\\999999999907612047\\000000000000000000\\000000000092387953\\000000000100000000\\000000000092387952\\999999999999999999\\999999999907612047\\000000000099999999\\999999999907612047\\000000000000000000\\000000000092387953\end{array} + \begin{array}{r}-38268343\\000000000099999999\\999999999961731656\\999999999999999999\\999999999961731657\\000000000100000000\\000000000038268343\\000000000000000000\\000000000038268343\\000000000099999999\\999999999961731656\\999999999999999999\\999999999961731657\\000000000100000000\\000000000038268343\end{array}\, i\right)$$

$$= \left(\begin{array}{r} 1094102380983951 \\ 990665268894305367 \\ 001860157462423250 \\ 008216043613832169 \\ 996099477984043813 \\ 003104773390843541 \\ 005027433424335913 \\ \hline 996543877251201672 \\ 998655459320762055 \\ 983021375950836170 \\ 987197123475664085 \\ 993990054348798327 \\ 008602086679237944 \\ 016185379249163829 \\ 007906412324335913 \\ \hline 994379392351201673 \\ 007739065439778103 \\ 997392391356530803 \\ 992153163413240835 \\ 986957558334966157 \\ 009743318195194131 \\ 004759758958320288 \end{array} + \begin{array}{r} -3014674799279501 \\ 992391645398409032 \\ 981874909371667897 \\ 999358015050269195 \\ 989501219848715585 \\ 020804081553681573 \\ 005676144024113327 \\ \hline 995411719691144054 \\ 980425401136560701 \\ 987586678104136340 \\ 962880017675886672 \\ 018519775508855945 \\ 008722189063439298 \\ 025803139495863659 \\ 000188764324113327 \\ \hline 993045218191144054 \\ 983773026737281199 \\ 998448059405727307 \\ 986564919604218774 \\ 017847076958586749 \\ 014347655214723712 \\ 996032059742182086 \end{array} \, i \right) \Longrightarrow \begin{pmatrix} -0.34561227 + 0.45882803i \\ -0.13445406 + 1.95745988i \\ -1.69786240 + 1.24133218i \\ -1.28028765 + 3.71199823i \\ -0.60099456 - 1.85197755i \\ 0.86020866 - 0.87221890i \\ 1.61853792 - 2.58031394i \\ 0.79064123 - 0.01887643i \end{pmatrix}$$

Extract middle N coefficients and invert the Kronecker substitution. A final multiplication by $\omega^{k^2/2}$ gives the desired DFT.

Let's compare old and new algorithms:

$$N \lg N \; \mathsf{M}(P) \qquad \text{vs} \qquad \mathsf{M}(NP).$$

Assume that we use a "fast" FFT-based integer multiplication algorithm, so that

$$\mathsf{M}(N) = N(\lg N)^{1+\epsilon}.$$

What happens if $P \sim N$?

(For example: a transform of length 1024, with 1024-bit coefficients.)

Then we get

$$N^2(\lg N)^{2+\epsilon} \qquad \text{vs} \qquad N^2(\lg N)^{1+\epsilon}.$$

We save a factor of $\lg N$!

This is very counterintuitive.

What is the source of the $\lg N$ factor improvement?

In the old algorithm, we get one factor of $\lg N$ from the FFT, and another factor of $\lg N$ from the coefficient arithmetic.

In the new algorithm, these two processes are "merged". The FFT *inside* the integer multiplication algorithm handles both problems simultaneously.

Instead of

$$(\lg N)^2$$

we get

$$\lg(N^2) = O(\lg N).$$

So much for DFTs.

How do we apply this idea to integer multiplication?

Suppose we wish to multiply two large n-bit integers.

First convert to a polynomial multiplication problem:

Split the inputs into about $n/\lg n$ chunks of $\lg n$ bits.

Now we must multiply two integer polynomials, with degree about $n/\lg n$, and coefficients with $\lg n$ bits.

Perform this polynomial multiplication by using DFTs over $\mathbf{C}$, with working precision of $P = O(\lg n)$.

So far this is all very standard.

(Schönhage–Strassen's famous paper actually has two algorithms, the famous one and a not-so-famous one. We have basically been following the not-so-famous one.)

Next we use a generalised Cooley–Tukey technique to decompose the DFT of length $n/\lg n$ into many "short DFTs" of length about $\lg n$.

This is basically what Fürer did.

Where we differ from Fürer is in how we evaluate these short DFTs.

Fürer works over a special coefficient ring (not $\mathbf{C}$) so that he can multiply by certain roots of unity in linear time. This allows him to perform the short DFTs very efficiently.

Instead, we use the Bluestein–Kronecker trick from the first part of the talk.

This reduces the original integer multiplication to many small integer multiplications, which are handled recursively.

Some analysis shows that this achieves the desired bound

$$\mathrm{M}(n) = O(n \lg n \, K^{\lg^* n}).$$

Doing everything very carefully leads to the constant $K = 8$.

Can this be improved?

Can we do better than $K = 8$?

A *Mersenne prime* is a prime of the form $p = 2^q - 1$.

The largest known Mersenne prime is $2^{57,885,161} - 1$.

It is unknown if there are infinitely many Mersenne primes.

Let $\pi_m(x)$ be the number of Mersenne primes less than x.

Conjecture (Lenstra–Pomerance–Wagstaff, early 1980s)

We have

$$\pi_m(x) \sim c \log \log x$$

as $x \to \infty$, *where* $c = e^\gamma / \log 2$.

[This is quite a subtle heuristic. Integers of the form $2^q - 1$ do not have the same "probability" of being prime as random nearby integers.]

Theorem (H.–van der Hoeven–Lecerf)

Assume the Lenstra–Pomerance–Wagstaff conjecture. Then

$$\mathrm{M}(n) = O(n \lg n \, 4^{\lg^* n}).$$

The idea of the proof is to replace the coefficient ring $\mathbf{C}$ by the finite field $\mathbf{F}_p[i]$, where p is a Mersenne prime of appropriate size.

The point is that there are fast algorithms for multiplication modulo integers of the form $2^q - 1$, and this saves a constant factor at each recursion level.

[Actually we can get by with something slightly weaker than the LPW conjecture; we just need nontrivial lim inf and lim sup.]

Thank you!