Multiplication of binary polynomials

R. P. Brent, P. Gaudry, E. Thomé, P. Zimmermann

See paper at ANTS VIII, 2008

Multiplication of univariate binary polynomials

R. P. Brent, P. Gaudry, E. Thomé, P. Zimmermann

See paper at ANTS VIII, 2008

INRIA
LORRAINE

CACAO

Plan

1. **Introduction**
2. **Small sizes**
3. **Medium sizes**
4. **Large sizes**

1. Introduction
2. Small sizes
3. Medium sizes
4. Large sizes

Why ?

We focus on polynomial multiplication over $\mathbb{F}_2[x]$.

This is used in many contexts:

- polynomial factorization, irreducibility tests ;
- (some) crypto applications ;
- less obvious: sparse linear algebra over $\mathbb{F}_2$;
- and more.

How does data look like ?

Binary polynomial $x^3 + x^2 + 1 \quad \rightarrow \quad$ machine integer$(1101)_2$ (“dense”).

- up to degree 63: one machine word (64-bit).
- degree 64 to 127: two words.
- …

In hardware:

- `add` is trivial ;
- `mul` is easy ; much easier than integer `mul`.
- Not our business.

In software:

- `add` is trivial (`xor`) ;
- `mul` is tedious (no PCMULQDQ yet !).

What do we do ?

We are interested in:

- software.
- speed everywhere: from 64 to 2^{32} coefficients (think recursion).

Existing software

Existing software typically has:

- Possibly fast multiplication for $1, 2 \ldots$ up to a few words.
- Karatsuba multiplication above.

Main reference: Victor Shoup's NTL: `shoup.net/ntl`

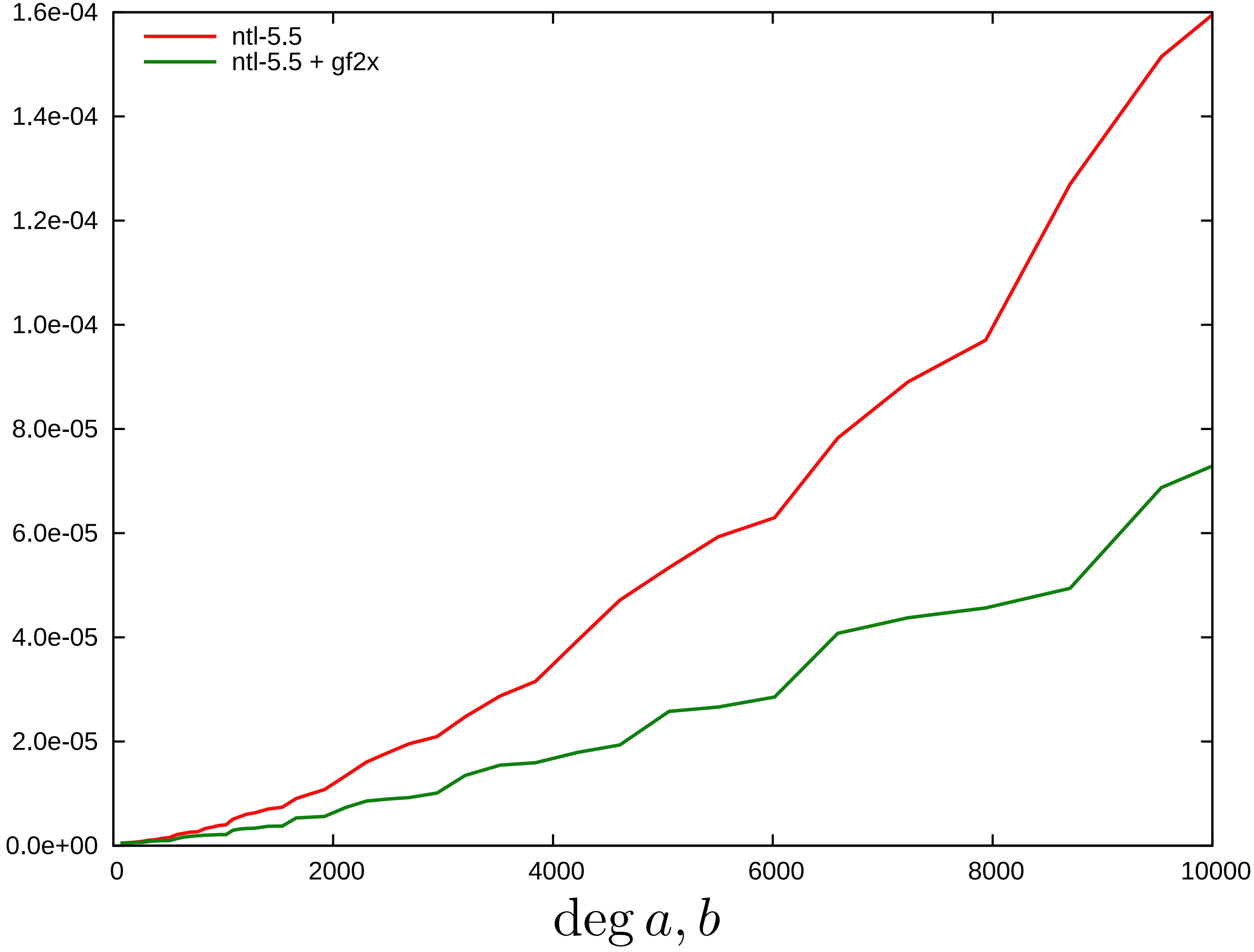
Very rarely (if ever), one finds:

- Code that takes advantage of CPU-specific instructions ;
- Toom-Cook multiplication ;
- Fast multiplication for unbalanced operands ;
- FFT (Schönhage ternary + Cantor additive).

All of this is in the `gf2x` software package.

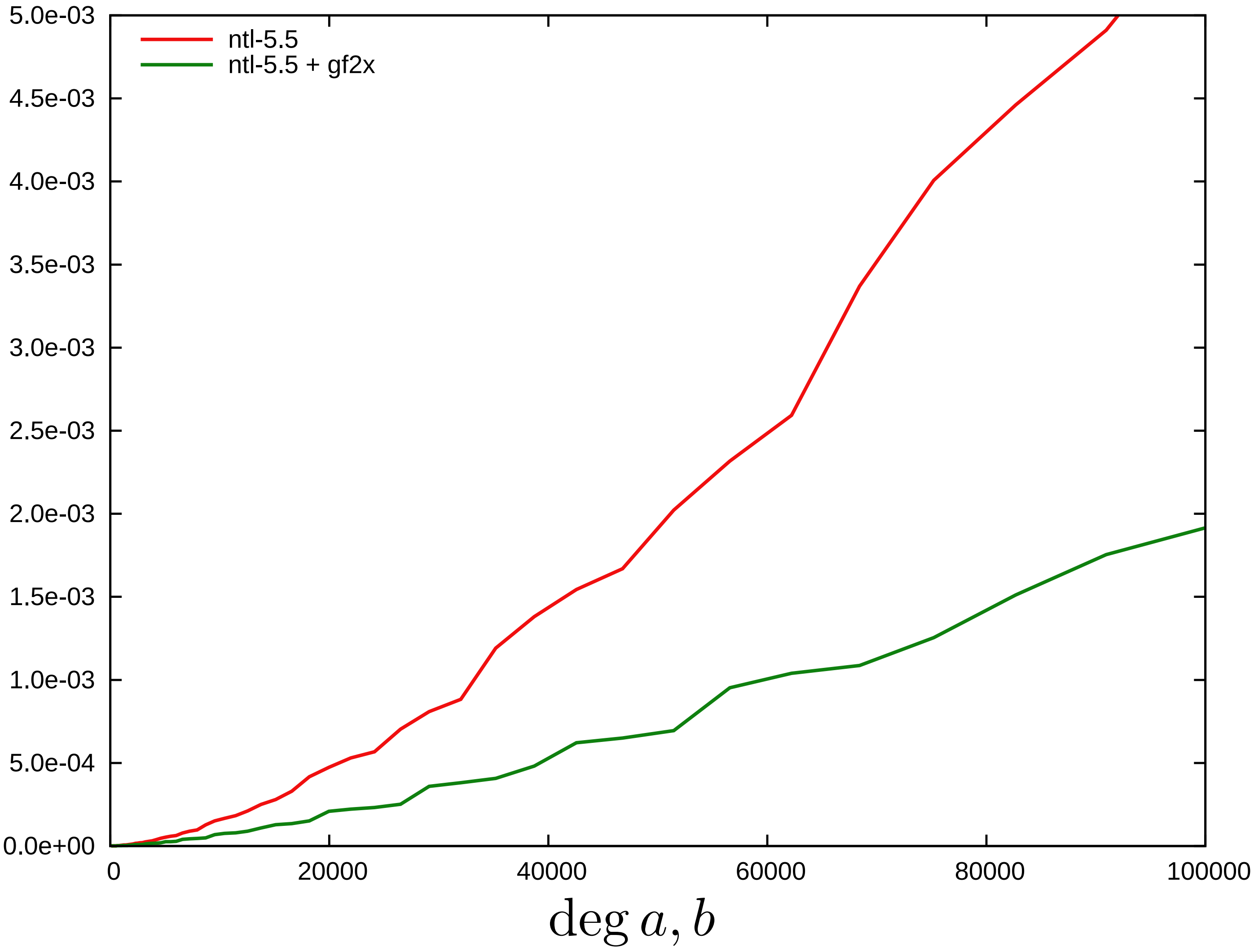
It pays off !

Timings in seconds, Core2 2.4GHz



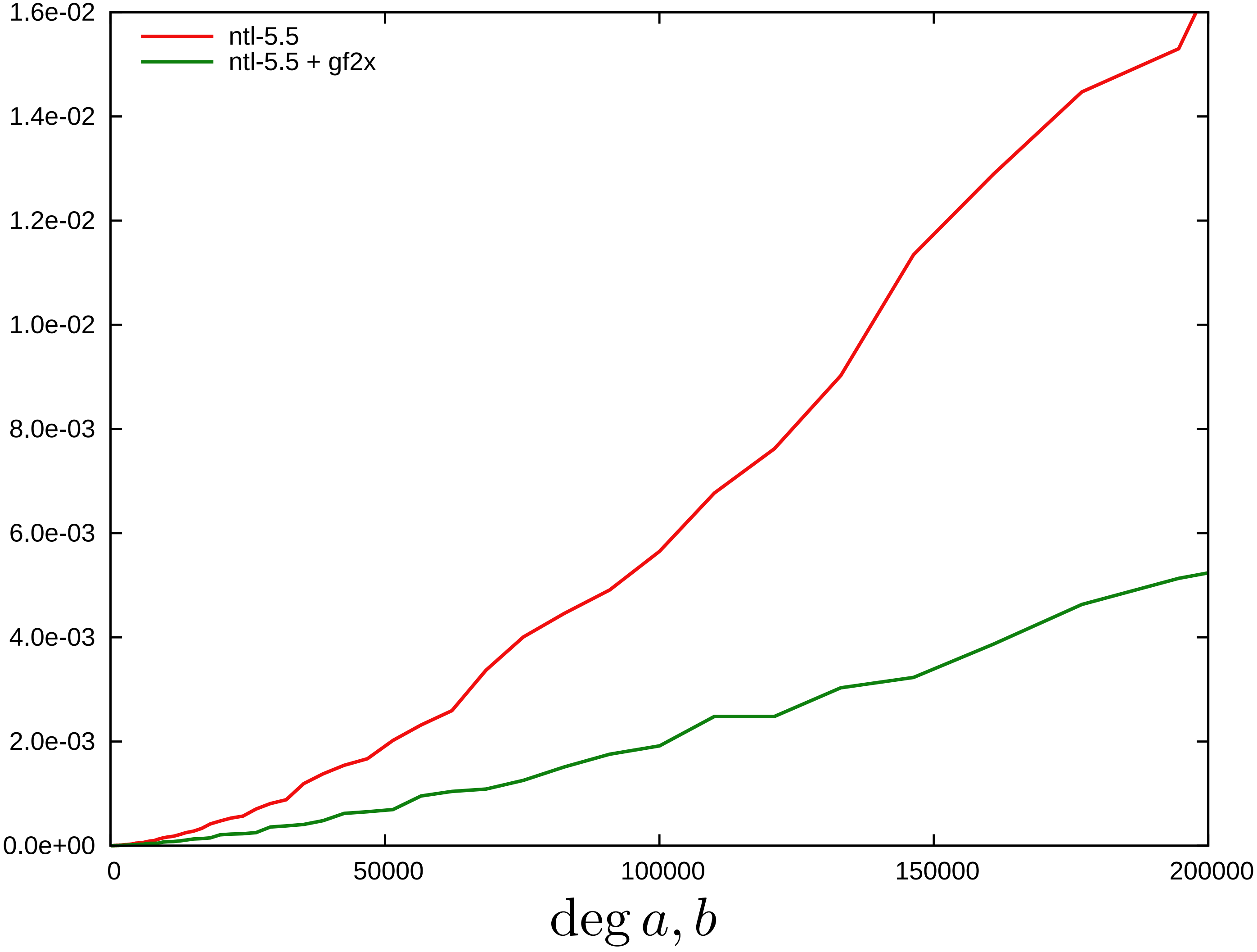
It pays off !

Timings in seconds, Core2 2.4GHz



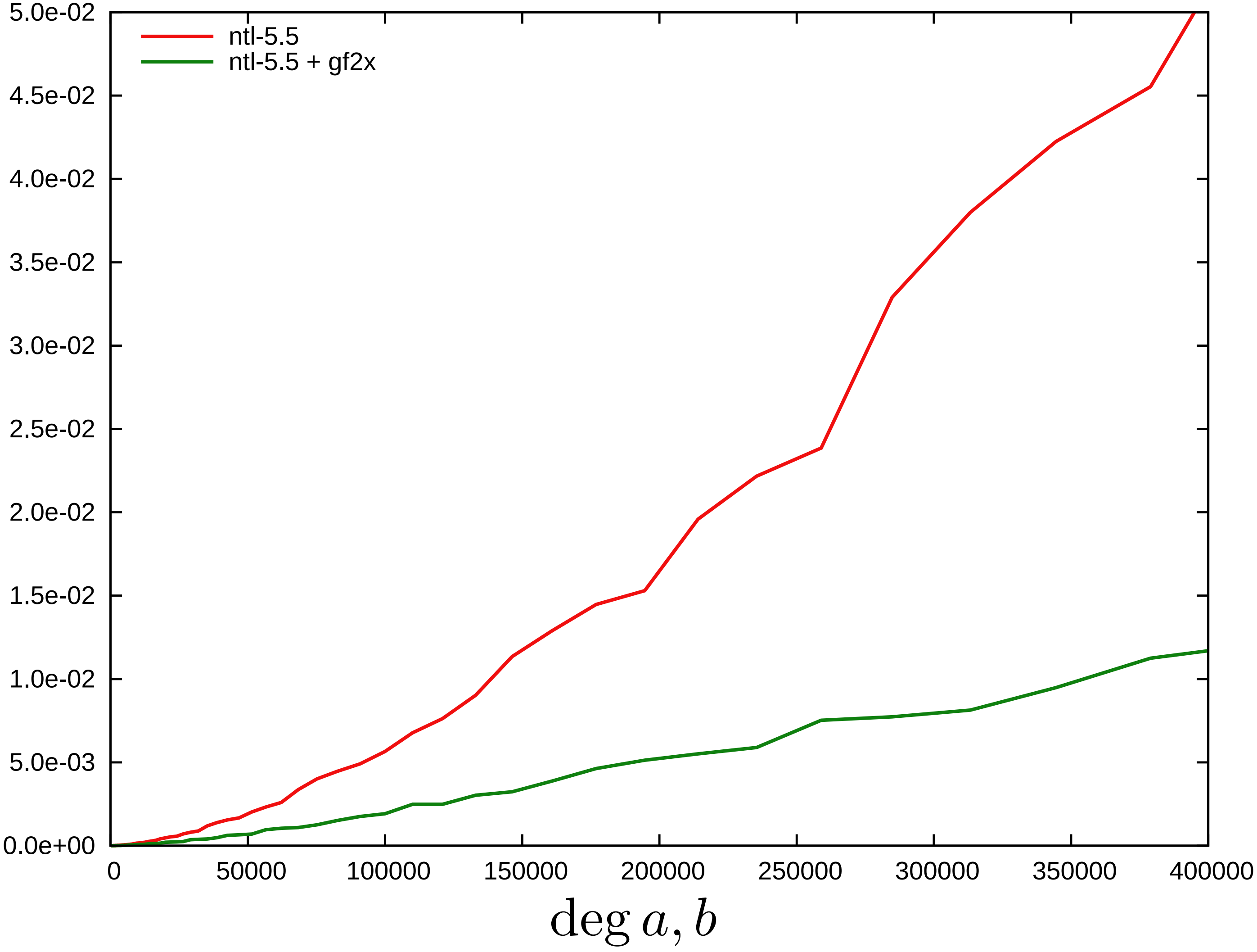
It pays off !

Timings in seconds, Core2 2.4GHz



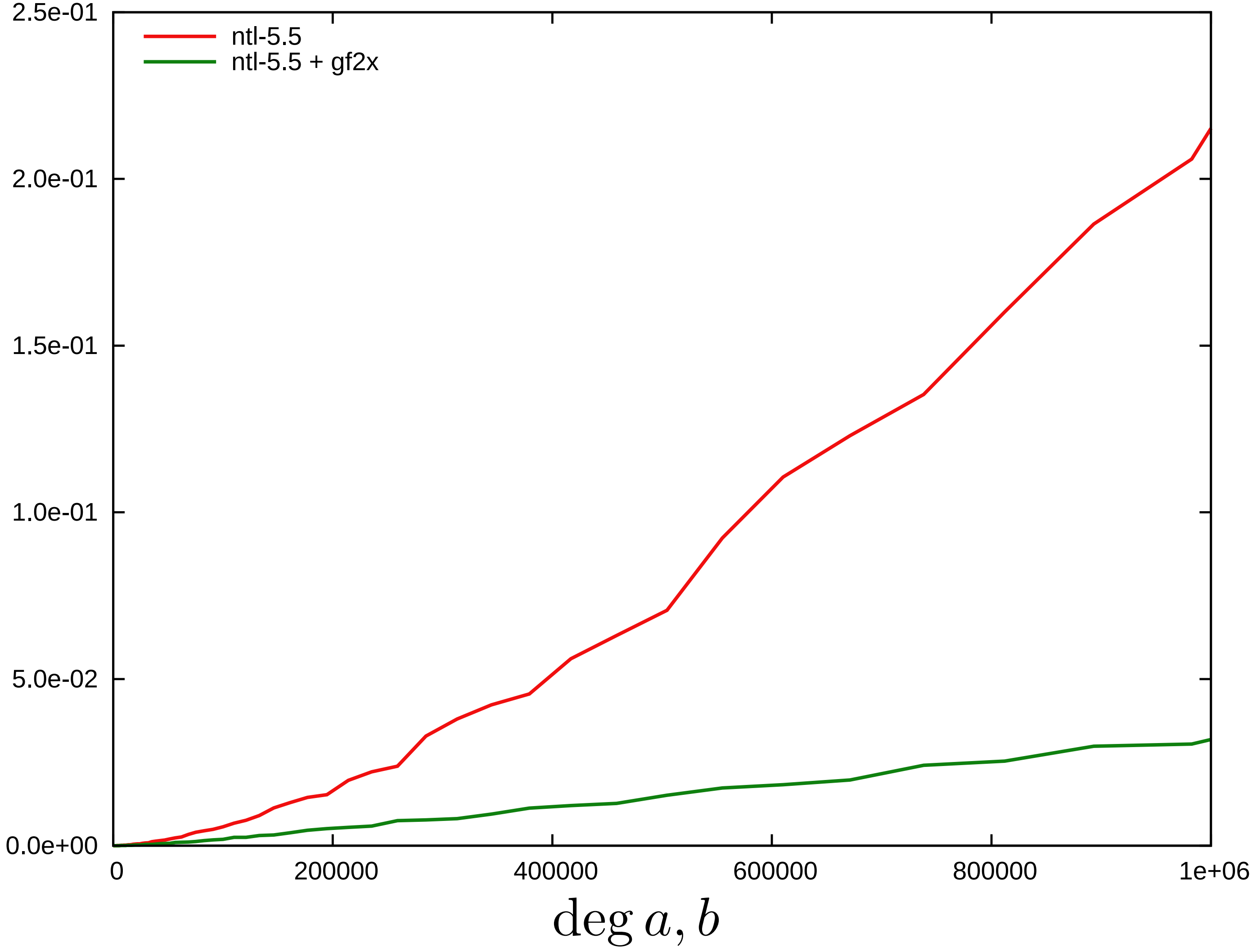
It pays off !

Timings in seconds, Core2 2.4GHz



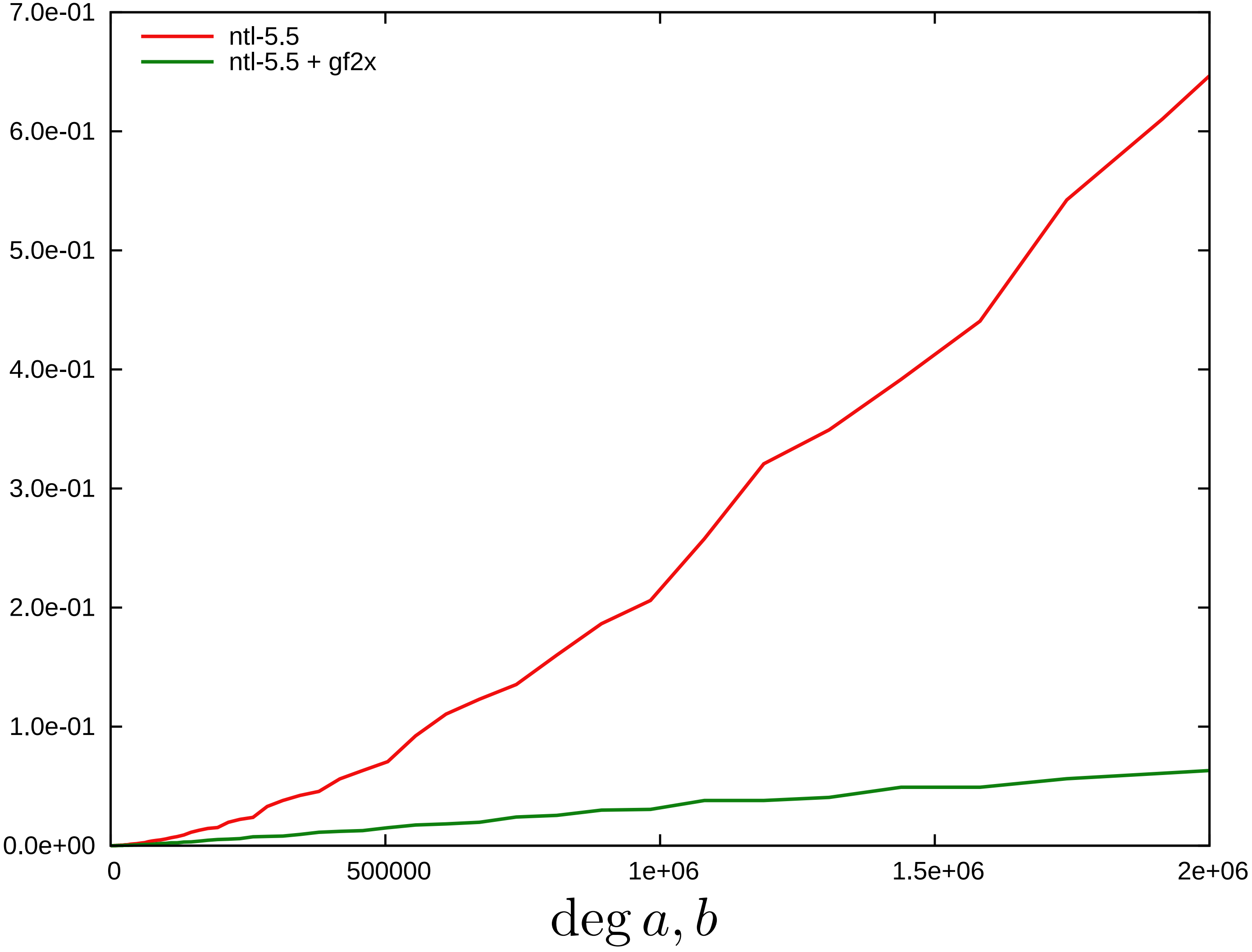
It pays off !

Timings in seconds, Core2 2.4GHz



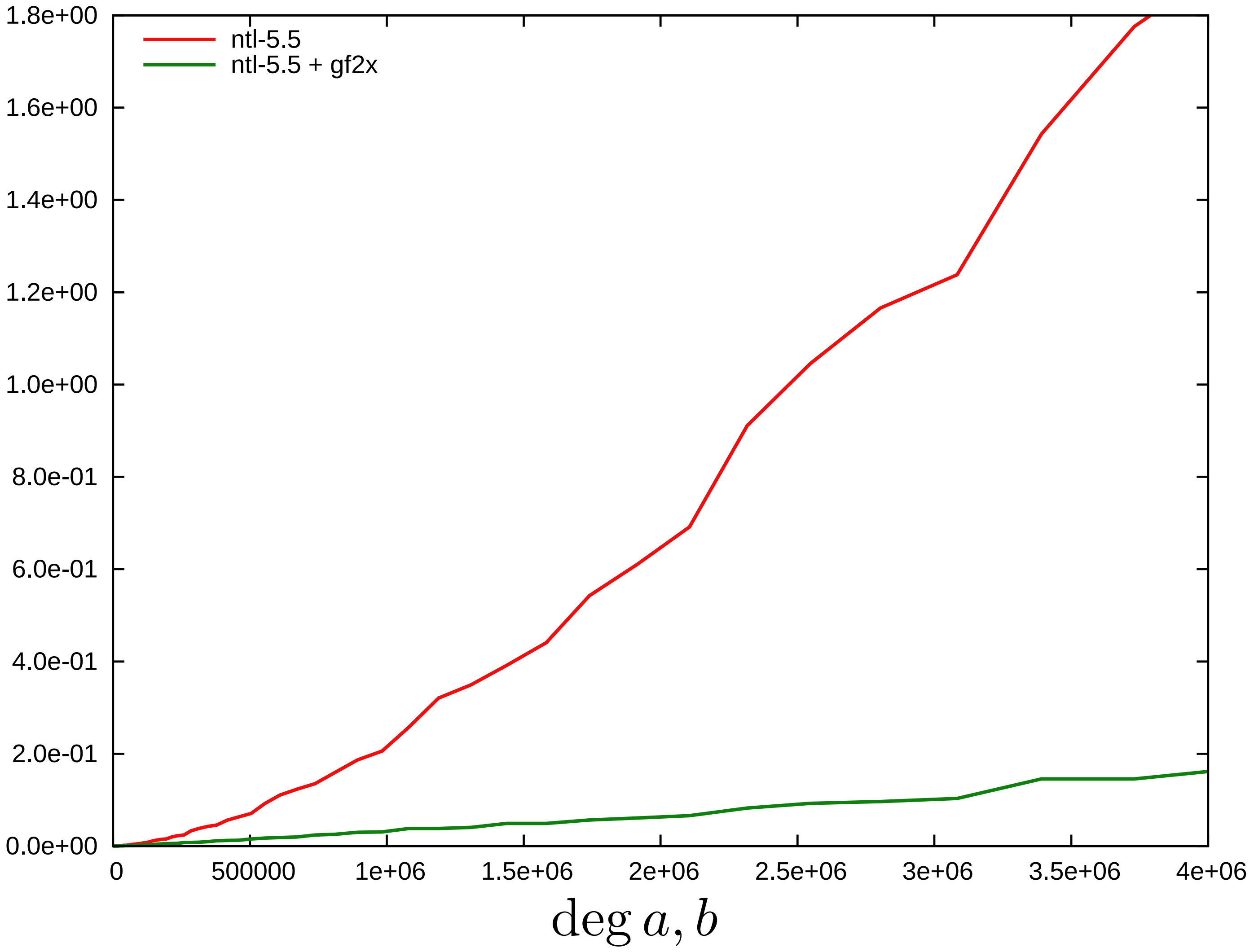
It pays off !

Timings in seconds, Core2 2.4GHz



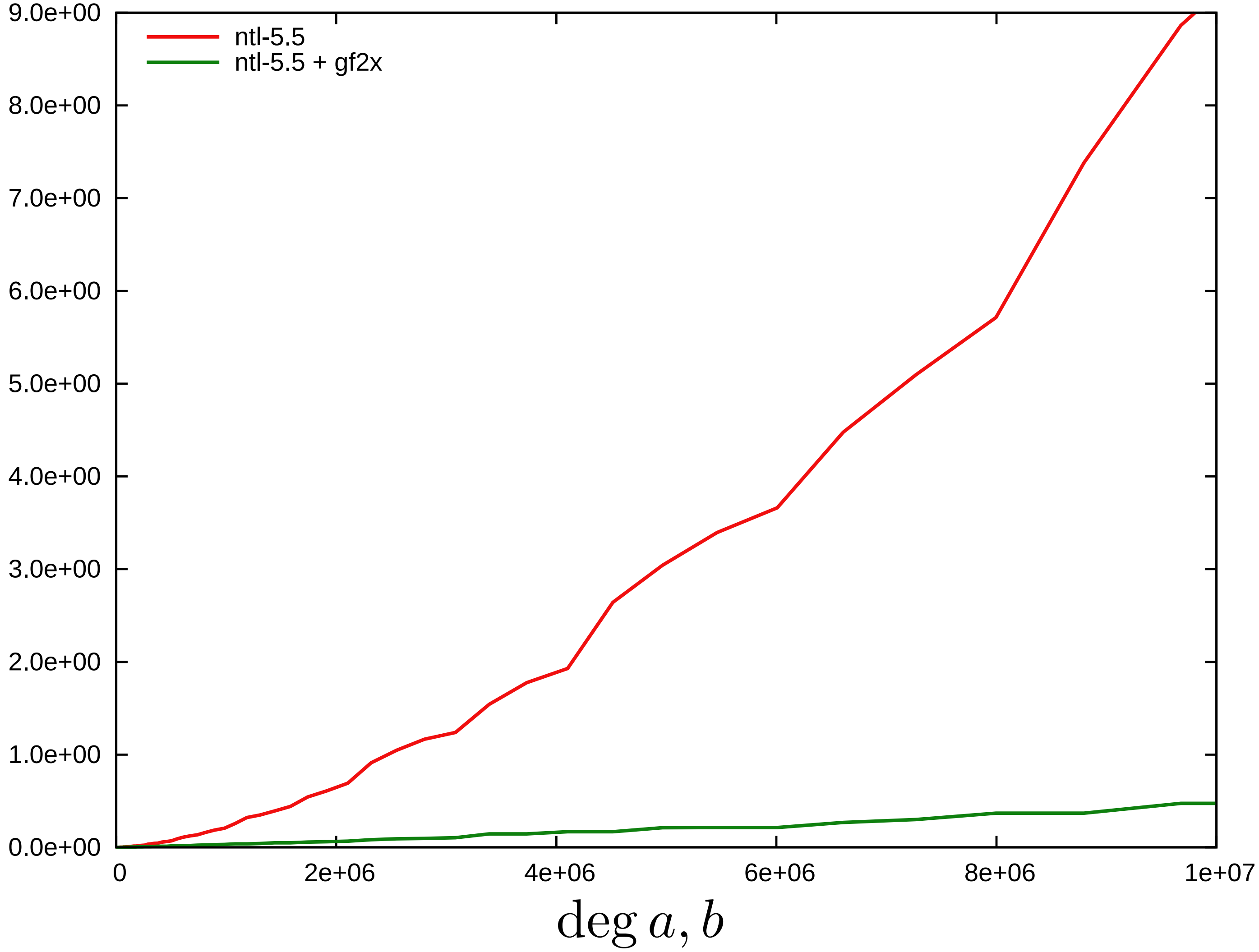
It pays off !

Timings in seconds, Core2 2.4GHz



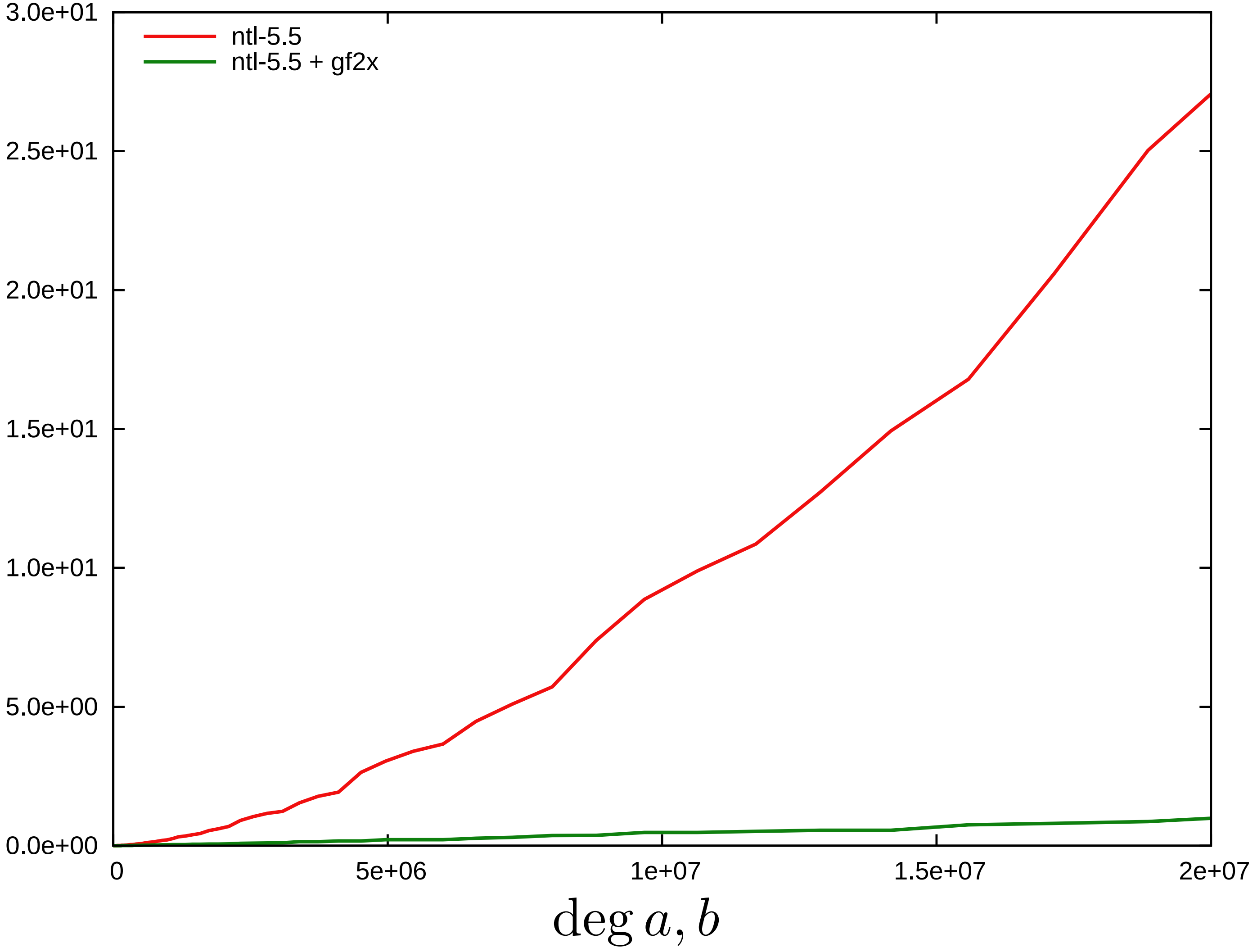
It pays off !

Timings in seconds, Core2 2.4GHz



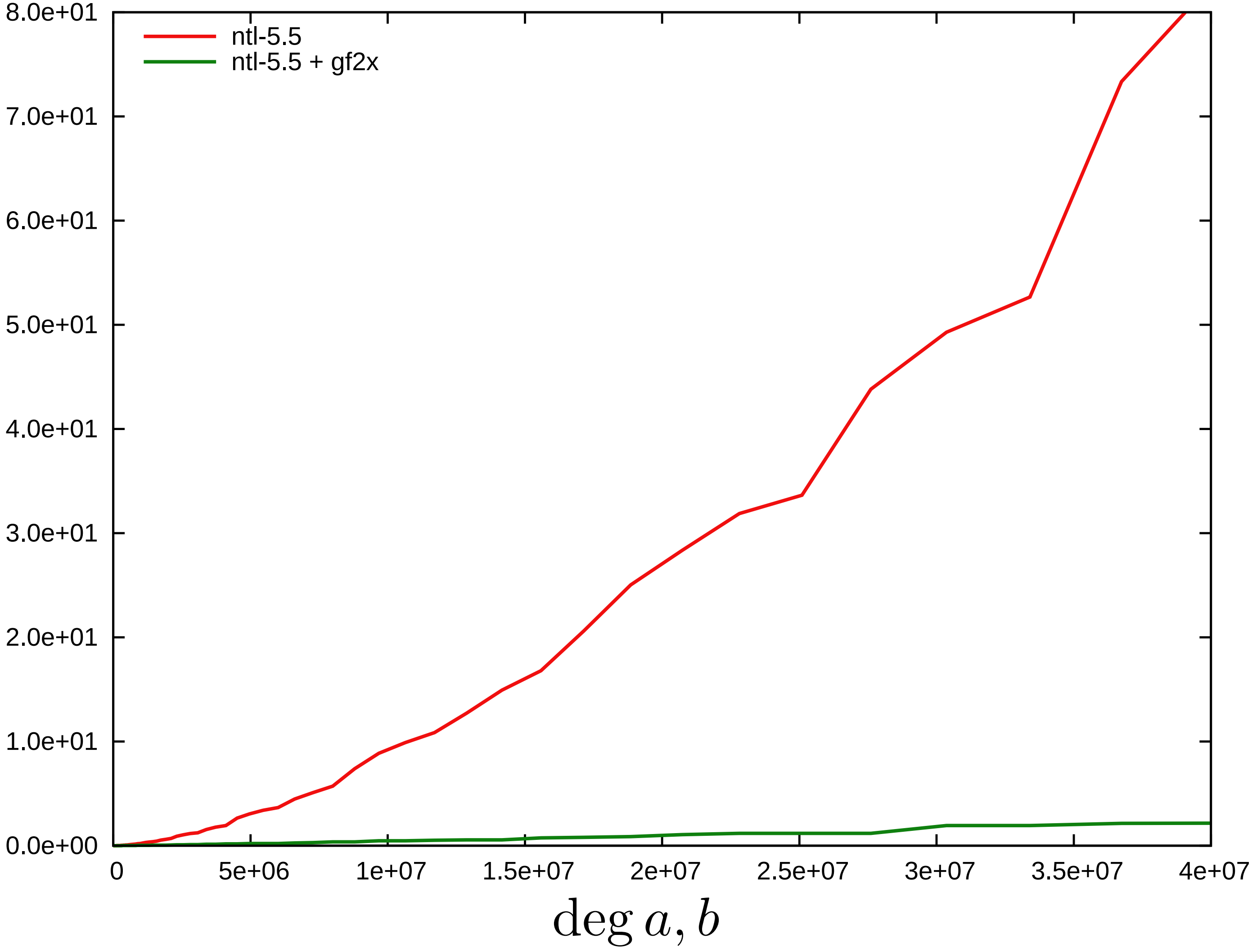
It pays off !

Timings in seconds, Core2 2.4GHz



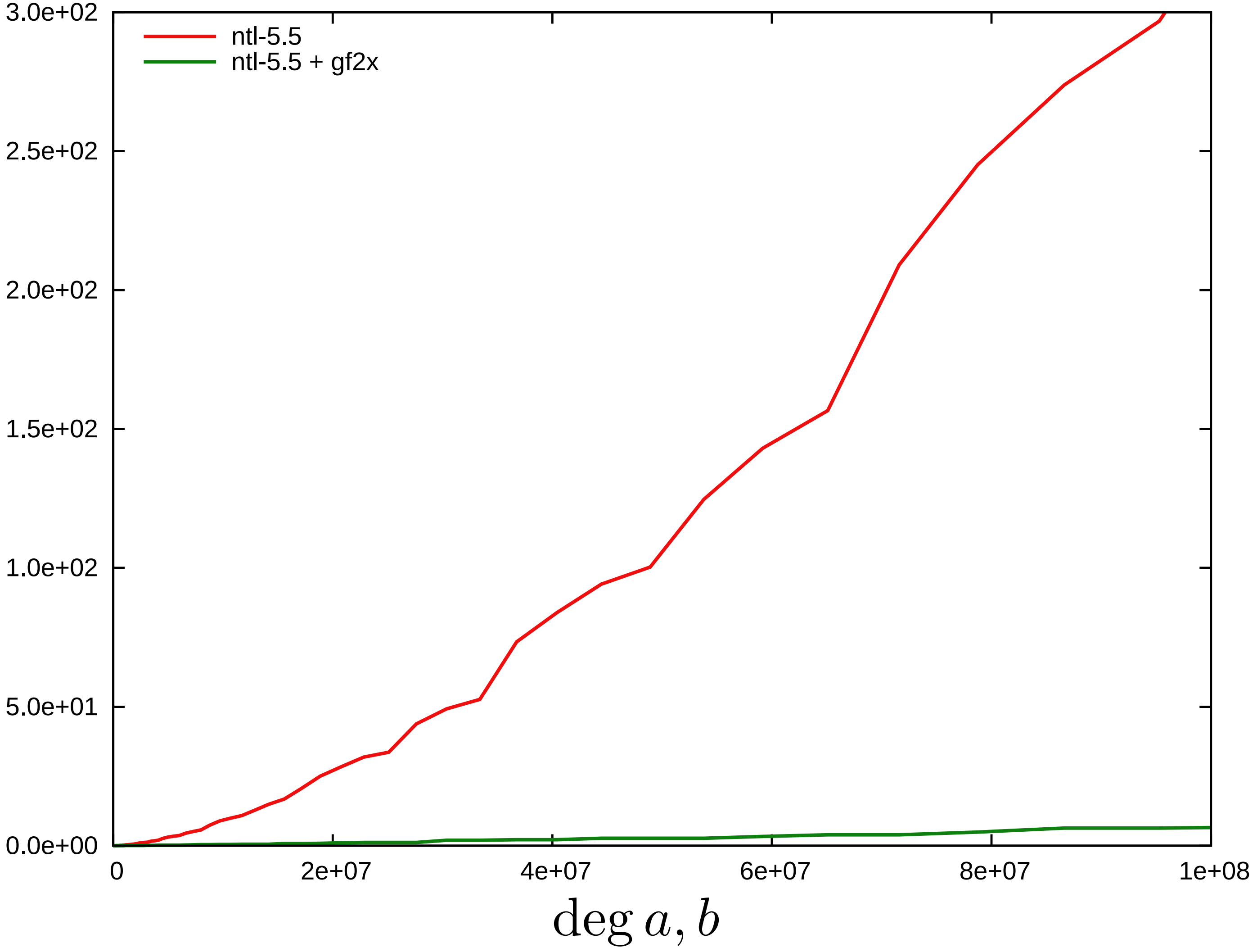
It pays off !

Timings in seconds, Core2 2.4GHz



It pays off !

Timings in seconds, Core2 2.4GHz



Notations

- Splitting of a polynomial $a(x)$ in s-bit slices:

$$\begin{aligned} a(x) &= A(x, x^s), \\ A(x,t) &= A_0(x) + A_1(x)t + A_2(x)t^2 + \cdots \\ &\text{and } \deg A_i < s. \end{aligned}$$

- Reconstruction: from $A(x,t)$, compute $a(x) = A(x, x^s)$.

Only of notational interest ; “computationally, nothing happens”.

1. Introduction
2. Small sizes
3. Medium sizes
4. Large sizes

Below degree 64: `mul1`

Classical: $c = a \times b$ computed with a (fixed-) window method.

- Tabulate multiples $g \times b$, for $\deg g < s$ ($s =$ window size).
- Split $a = A_0 + A_1 x^s + A_2 x^{2s} + \cdots$.
- Accumulate $c = A_0 \times b + (A_1 \times b)x^s + (A_2 \times b)x^{2s} + \cdots$.

Operations required: shifts, XORs.

For degree below 64, we work with machine words only.

- ⚠ For $\deg b = 63$, the computation $A_i \times b$ overflows !
- Necessary “repair” step is rather easy (see paper).

`mul1` (cont'd)

What is the best window size ?

- Use trial and error. Typically 3 or 4.
- 64×64: ~ 75 CPU cycles on Intel `core2` and `i7` ;
  ~ 85 CPU cycles on AMD `k8`.

Note: This trivially extends to a routine for $64k \times 64$.

Using SIMD capabilities

What about 128×128 ?

- Karatsuba ⇒ three 64×64.
- Schoolbook requires $a \times b_{\text{low}}$ and $a \times b_{\text{high}}$ ⇒ two 128×64.
- BUT $a \times b_{\text{low}}$ and $a \times b_{\text{high}}$ can be computed in a SIMD-manner.
- SIMD instructions on `x86_64` provide the necessary shifts and XORs.
  - Accessible with compiler builtins (`gcc`, `icc`, MSVC).
  - Assembly is not an absolute necessity.

128×128:

- ~ 129 Intel `core2` cycles ;
- ~ 226 AMD `k8` cycles.
- Faster than Karatsuba here.

1. Introduction
2. Small sizes
3. Medium sizes
4. Large sizes

Medium sizes

Classical: from 2 to 9 machine words, hard-code Karatsuba multiplication.

$\Rightarrow$ No branching.

Example for `mul4`:

```
mul2 (c, a, b);
mul2 (c + 4, a + 2, b + 2);
aa[0] = a[0] ^ a[2]; aa[1] = a[1] ^ a[3];
bb[0] = b[0] ^ b[2]; bb[1] = b[1] ^ b[3];
c24 = c[2] ^ c[4];
c35 = c[3] ^ c[5];
mul2 (ab, aa, bb);
c[2] = ab[0] ^ c[0] ^ c24; c[3] = ab[1] ^ c[1] ^ c35;
c[4] = ab[2] ^ c[6] ^ c24; c[5] = ab[3] ^ c[7] ^ c35;
```

Medium sizes

Classical: from 2 to 9 machine words, hard-code Karatsuba multiplication.

$\Rightarrow$ No branching.

Cycle counts, Intel `core2`.

$\deg$	NTL	LIDIA	ZEN	this paper
63	99	117	158	**75**
127	368	317	480	**132**
191	703	787	1 005	**364**
255	1 130	988	1 703	**410**
319	1 787	1 926	2 629	**806**
383	2 182	2 416	3 677	**850**
447	3 070	2 849	4 960	**1 242**
511	3 517	3 019	6 433	**1 287**

What comes next ?

Toom-3: $\deg a < 3k$, write $a = A(x, x^k)$, $A(x,t) = a_0(x) + a_1(x)t + a_2(x)t^2$.

- Evaluate $(A(x, x_i))_{i=0,1,2,3,4}$ and $(B(x, x_i))_{i=0,1,2,3,4}$
- Multiply: $C(x, x_i) = A(x, x_i)B(x, x_i)$.
- Interpolate: recover $C(x,t)$ from $(C(x, x_i))_{i=0,1,2,3,4}$

Misbelief:
- This is only for $\#K \geq 4\ldots$
- because we need 5 evaluation points (in $\mathbb{P}^1(K)$).

- We can use: $0, 1, \infty, x, x^{-1}$; call this TC3.
- Often better: $0, 1, \infty, x^{64}, x^{-64}$: avoids shifts ; call this TC3W.
- The degrees in recursive calls increase mildly.

Timings

$1 + \deg$	NTL	`gf2x`	method
$1\,536$	0.008	0.004	TC3
$4\,096$	0.033	0.017	K2
$8\,000$	0.098	0.046	TC3W
$10\,240$	0.160	0.080	TC3W
$16\,384$	0.295	0.140	TC3W
$24\,576$	0.567	0.240	TC3W
$32\,768$	0.887	0.395	TC4
$57\,344$	2.331	0.976	TC4
$65\,536$	2.667	1.067	TC4
$131\,072$	7.937	3.040	TC4

`ms`, Intel Core2 2.4GHz ; cycles:$\times 2.4 \cdot 10^6$.

1. Introduction
2. Small sizes
3. Medium sizes
4. Large sizes

Large sizes

We are also interested in multiplication in the FFT range.

Several options:

- integer FFT and (huge) padding (Krönecker-Schönhage).
- Schönhage's ternary FFT algorithm.
- Cantor's additive FFT algorithm.
- . . .

Why is Krönecker's trick inefficient here ?

Assume we know how to multiply (fast) in $\mathbb{Z}$.

We use that in for multiplying in $\mathbb{F}_p[x]$.

If $a(x) = a_0 + a_1 x + \cdots + a_{N-1} x^{N-1}$, and $b(x)$ similar:

- Choose $B = 2^k$ such that $Np^2 < 2^k$.
- Form the integer $\tilde{a} = a_0 + a_1 B + \cdots$. Same for $\tilde{b}$. We have:

$$\tilde{a} \cdot \tilde{b} = \sum_{\ell} \underbrace{\sum_{i+j=\ell} a_i b_j}_{\tilde{c}_\ell < Np^2 < B} B^\ell .$$

- The coefficients of $c = a \times b$ are recovered with $c_i = \tilde{c}_i \bmod p$.

Cost: $\mathsf{M}\left(N \log\left(Np^2\right)\right)$, to be compared to $\mathsf{M}(N \log p)$ (quasi-linearity w.r.t input size). This is very expensive for p small.

Other ways to do the FFT

Assume we are given one quotient ring R of $\mathbb{F}_2[x]$ of degree d with

- (reasonably) fast multiplication ;
- fast multipoint evaluation and interpolation on some subset W.

Then we can use this to compute products in $\mathbb{F}_2[x]$ up to $\frac{1}{2}d.\#W$ bits.

- Split a and b in blocks of $\frac{d}{2}$ coefficients: $a = A(x, x^{d/2})$. Map A to $R[t]$.
- Multi-evaluate A and B at W.
- Compute pointwise products $\{A(w) \cdot B(w), w \in W\}$.
- Interpolate: recognize C such that $\forall w \in W, C(w) = A(w) \cdot B(w)$.
  Note that $C = AB$ as long as $\deg(AB) < \#W$.
- Lift C to $\mathbb{F}_2[x, t]$. Recover $ab = C(x, x^{d/2})$.

Where to do the FFT ?

Several possibilities. Pitfall: W can not be a set of 2^n-th roots of 1 !

- Let $R = \mathbb{F}_{2^k}$ with $2^k - 1$ having a large smooth factor K.
  Let $W = \{K$-th roots of $1\}$.
- Let $R = \mathbb{F}_2[x]/x^{2L} + x^L + 1$, where $L = \lambda 3^{k-1}$.
  Then x^λ generates $W = \{3^k$-th roots of $1\}$.
- Let $R = \mathbb{F}_{2^{2^k}}$ and W be a sub-vector space.

“Butterflies” go “3-way”.

$$x, y \to x + \alpha y, x - \alpha y,$$

$$x, y, z \to x + \alpha y + \alpha^2 z, x + j\alpha y + j^2\alpha^2 z, x + j^2\alpha y + j\alpha z.$$

Where to do the FFT ?

Several possibilities. Pitfall: W can not be a set of 2^n-th roots of 1 !

- Let $R = \mathbb{F}_{2^k}$ with $2^k - 1$ having a large smooth factor K.
  Let $W = \{K$-th roots of $1\}$. not looked at.
- Let $R = \mathbb{F}_2[x]/x^{2L} + x^L + 1$, where $L = \lambda 3^{k-1}$.
  Then x^λ generates $W = \{3^k$-th roots of $1\}$. ternary FFT.
- Let $R = \mathbb{F}_{2^{2^k}}$ and W be a sub-vector space. additive FFT.

“Butterflies” go “3-way”.

$$x, y \to x + \alpha y, x - \alpha y,$$

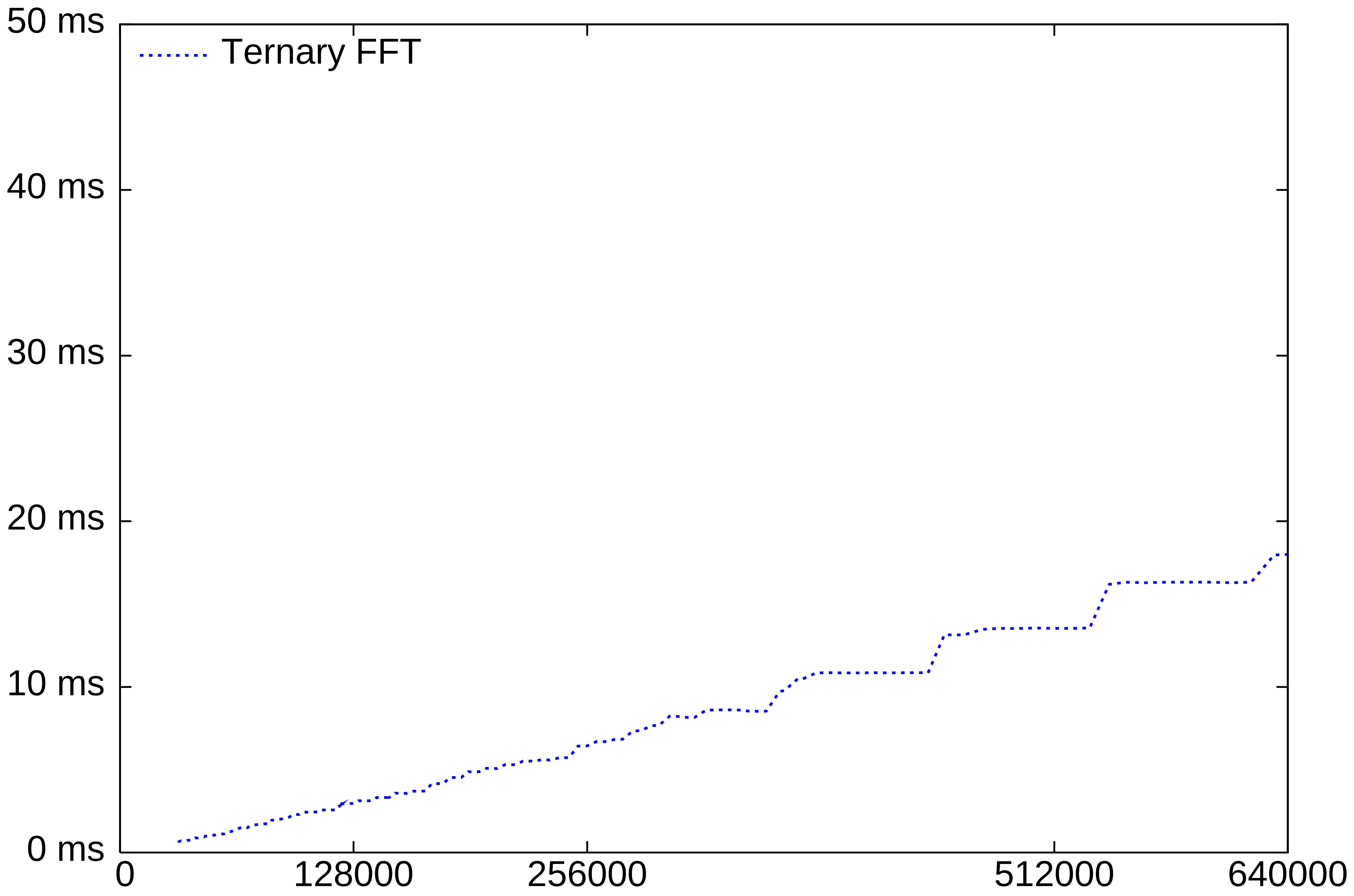
$$x, y, z \to x + \alpha y + \alpha^2 z, x + j\alpha y + j^2\alpha^2 z, x + j^2\alpha y + j\alpha z.$$

Ternary FFT (Schönhage)

The ternary FFT achieves $O(N \log N \log \log N)$ complexity if one uses it recursively also for the pointwise products, BUT:

- This requires computing the pointwise products modulo $X^{3L} + 1$.
- One doesn't *have* to. One top-level step gives already good results.
- Have to choose a sensible value for K.

Ternary FFT (Schönhage)



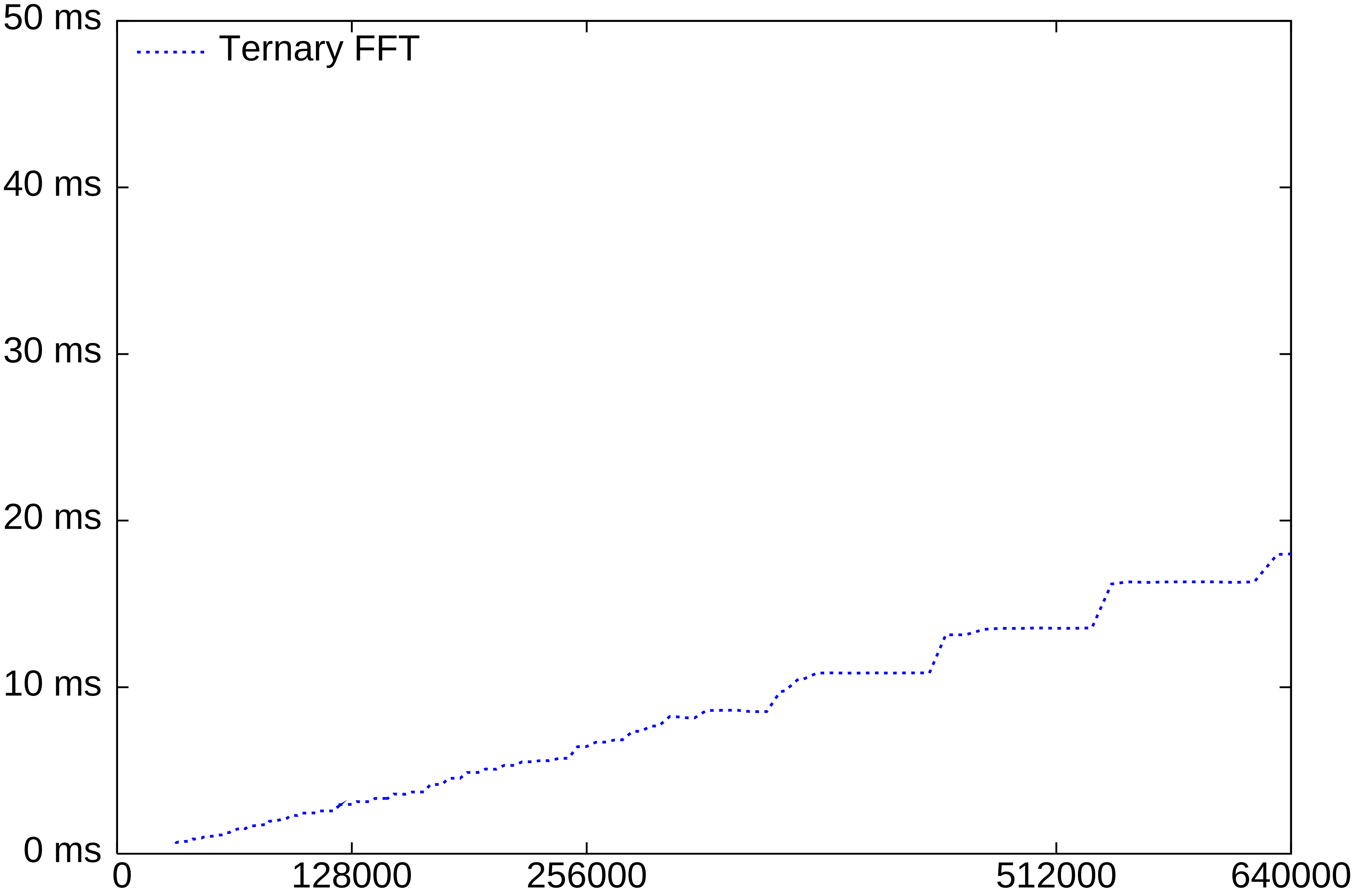
Splitting the ternary FFT

There is a (mild) staircase effect.

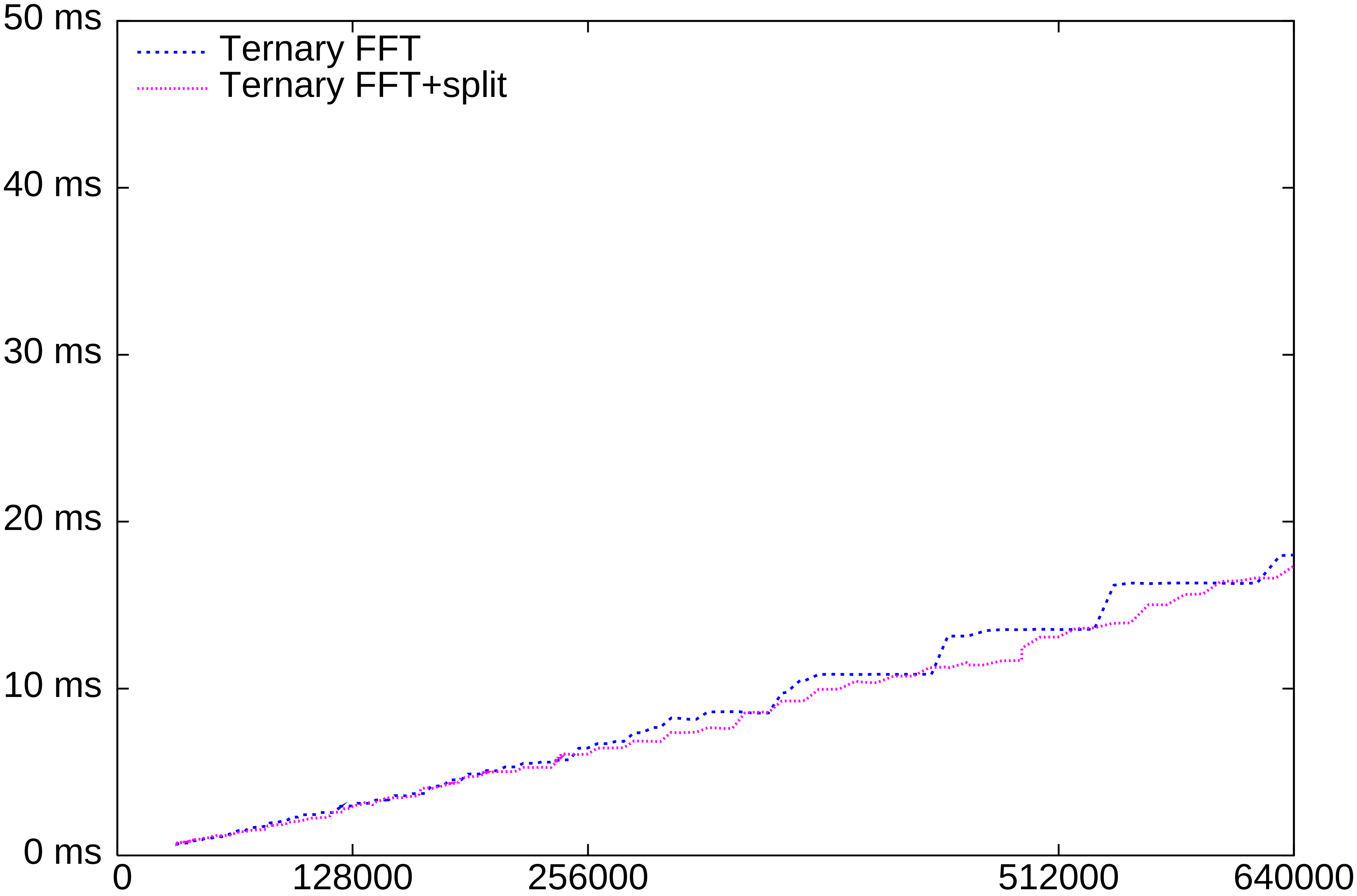
We can compute a product of degree $< N$ by splitting:

- Compute one product modulo $N' > N/2$.
- Compute another product modulo $N'' > N'$.
- Very simple XORs do the reconstruction.

Schönhage FFT + splitting



Schönhage FFT + splitting



Cantor's additive FFT

A completely different approach. Use a field $R = F_k = \mathbb{F}_{2^{2^k}} = \mathbb{F}_2[\gamma]$.

Which evaluation set W do we use ?

Let $s_1(x) = x^2 + x$, and $s_i(x) = \underbrace{s_1(s_1(\cdots s_1}_{i \text{ times}}(x)\cdots))$.

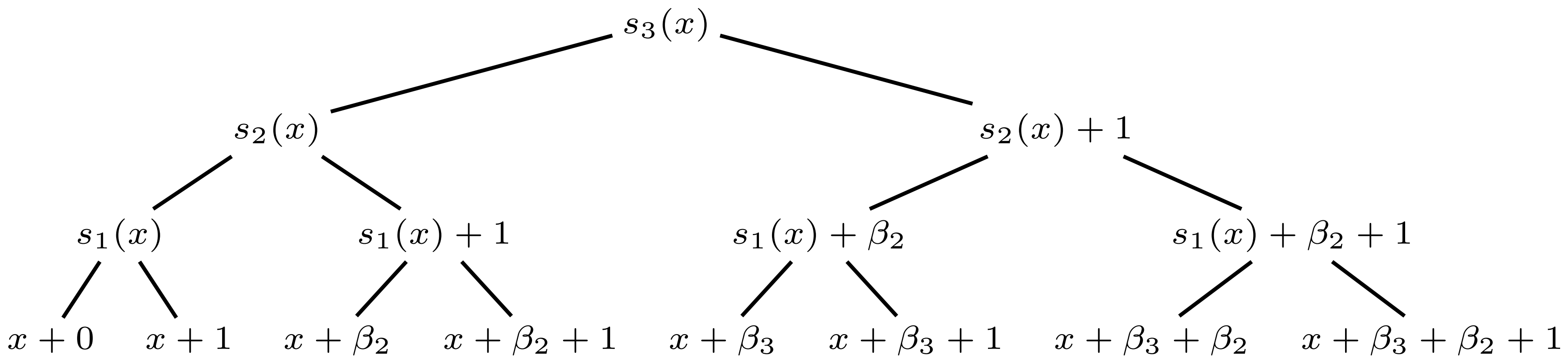
s_i satisfies many properties:

- s_i is sparse ; s_i is linear ; $s_{2^k} = x^{2^{2^k}} + x$.
- Let $2^k \geq i$ and $W_i = \{\alpha \in \mathbb{F}_{2^{2^k}} \mid s_i(\alpha) = 0\} = \operatorname{Ker} s_i$.
  W_i is a sub-vector space of $\mathbb{F}_{2^{2^k}}$; $\dim W_i = i$.

Multi-evaluation at W_i

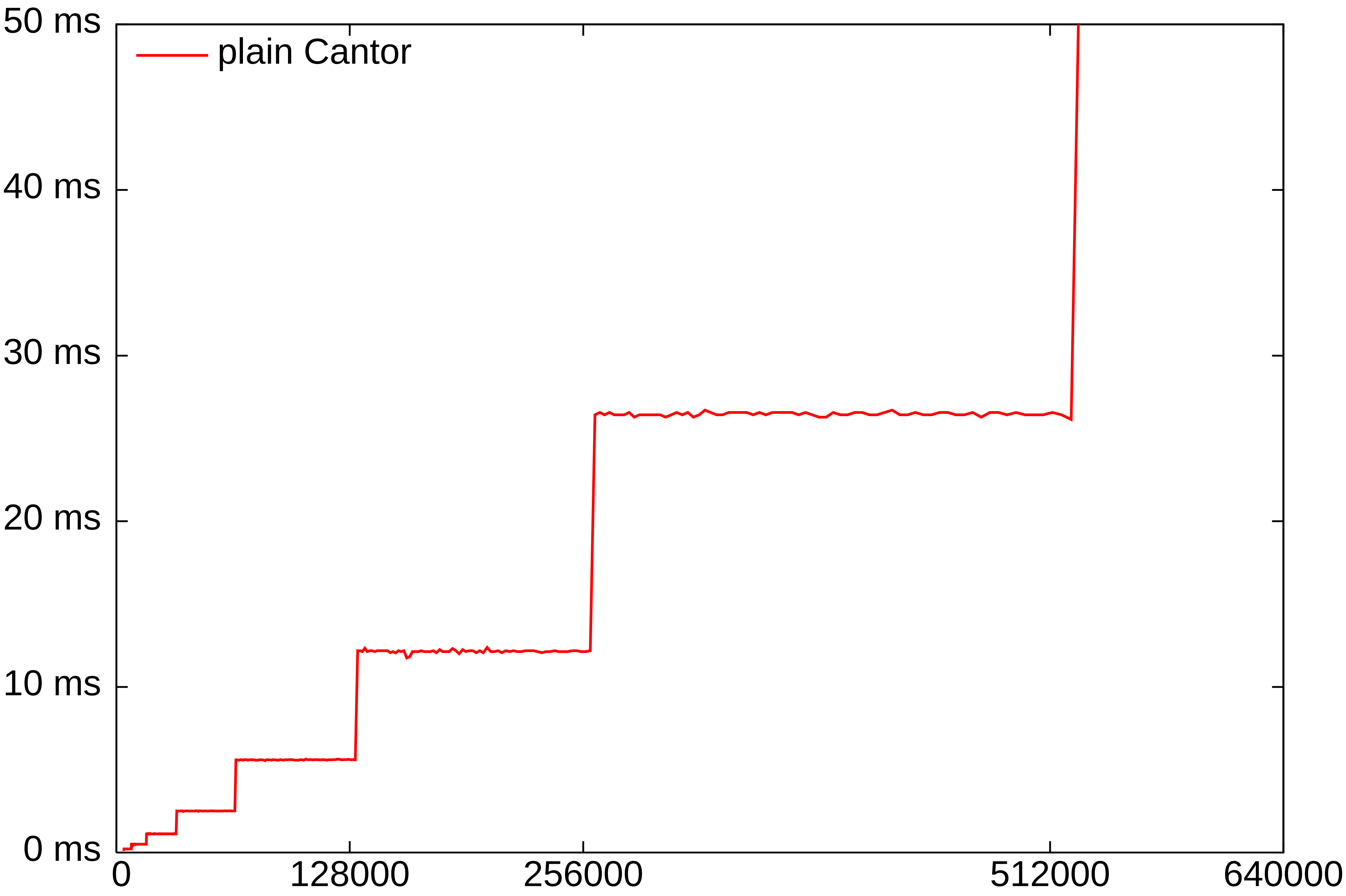
Use a sub-product tree:

$$\{f(\alpha),\ \alpha \in W_i\} = \{f \bmod (x+\alpha),\ \alpha \in W_i\}\,.$$



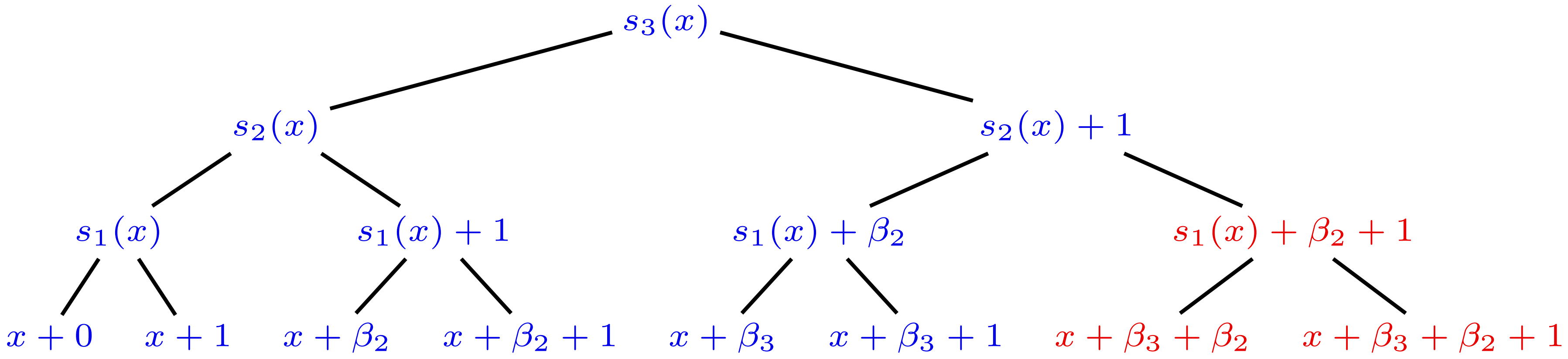
- right-child $= 1 +$ left-child.
- Only the constant coefficients are in extension fields.
- s_j is sparse, so reduction is cheap.

Cantor: staircase effect



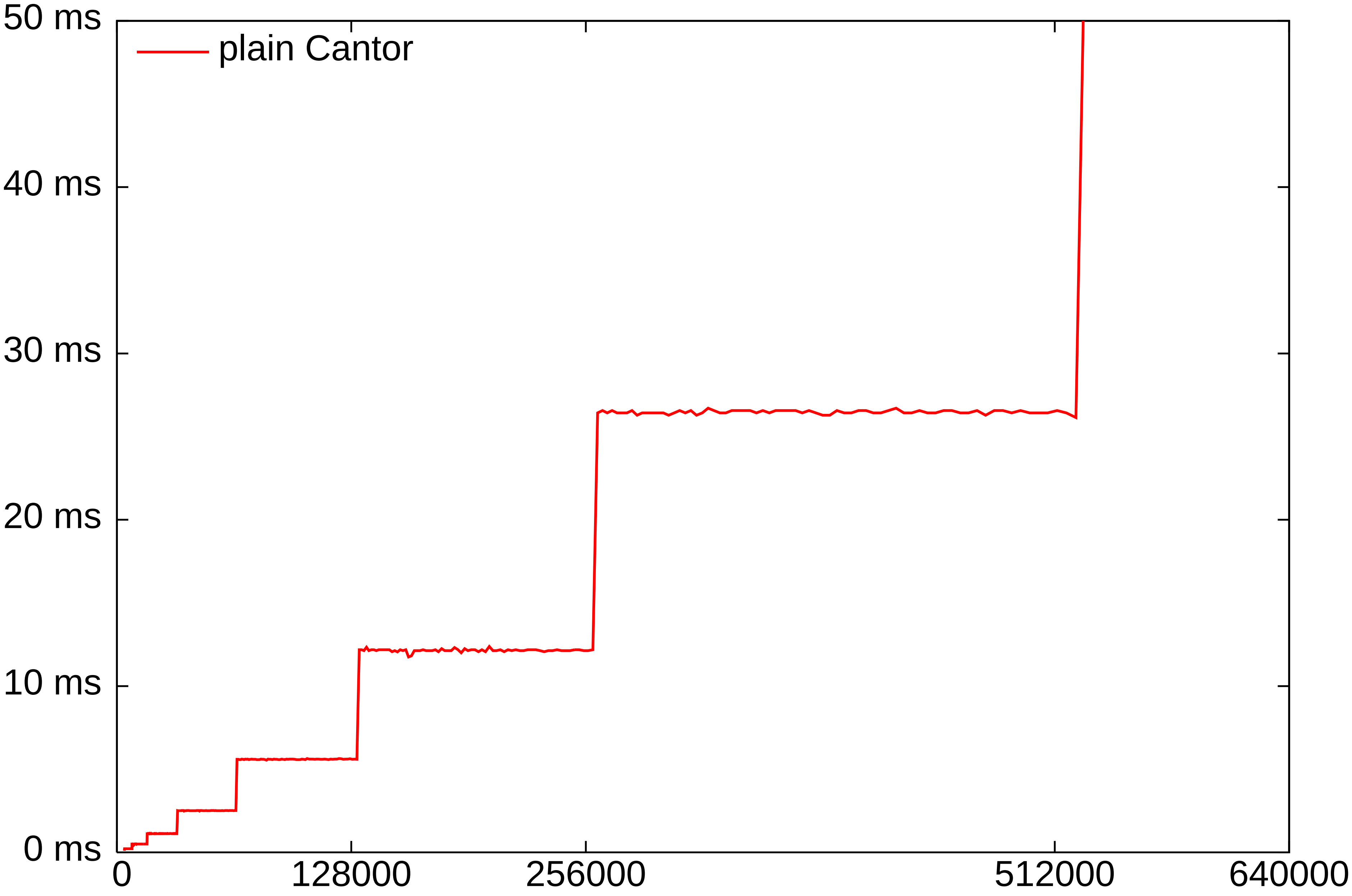
A truncated variant

- Evaluate at no more points than needed. Example for 6 points:

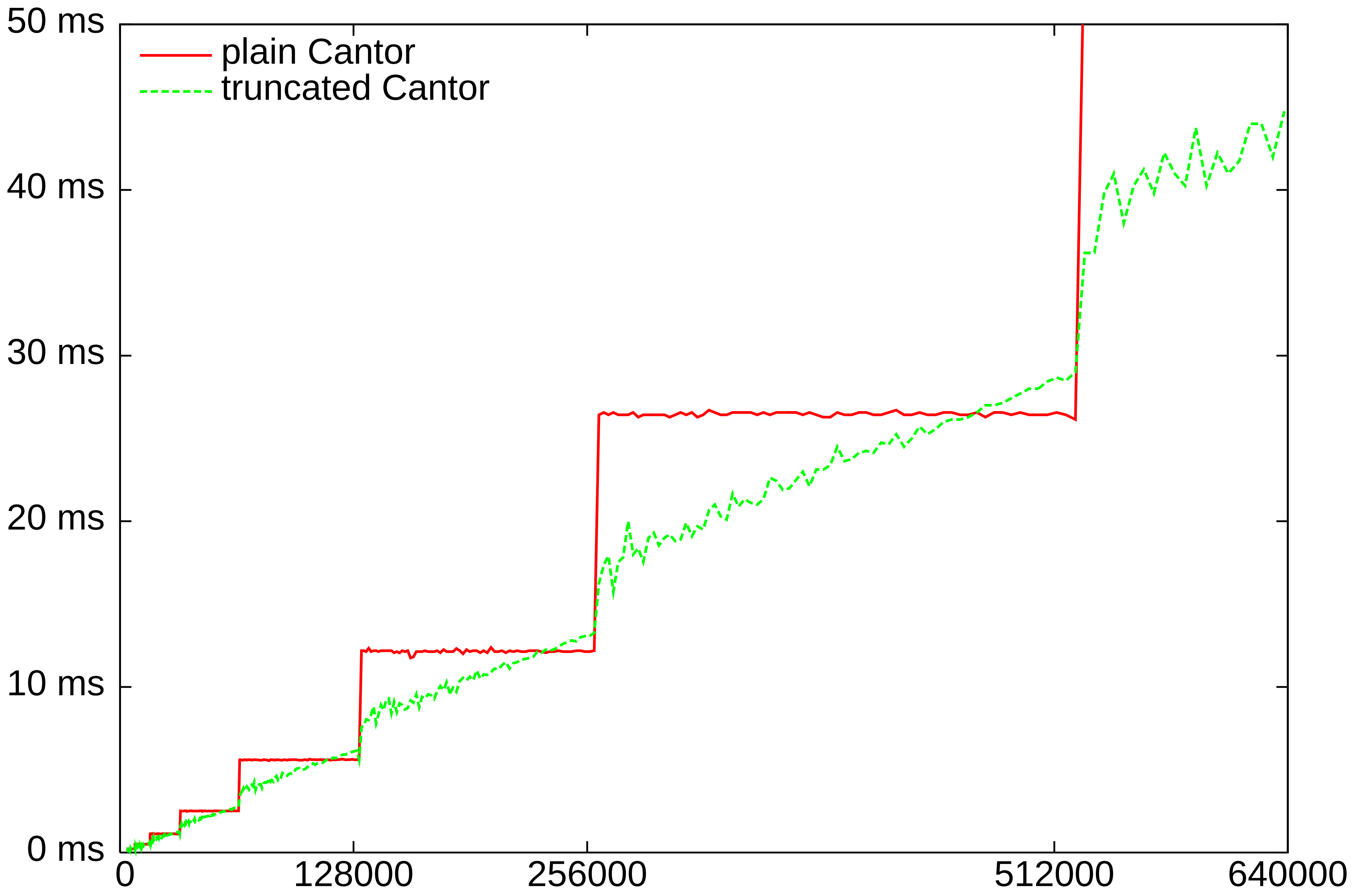


- Work modulo $s_2(x)\,(s_1(x)+\beta_2)$ instead of $s_3(x)$.
- Interpolation is tricky. Uses the sub-product tree twice. See paper.

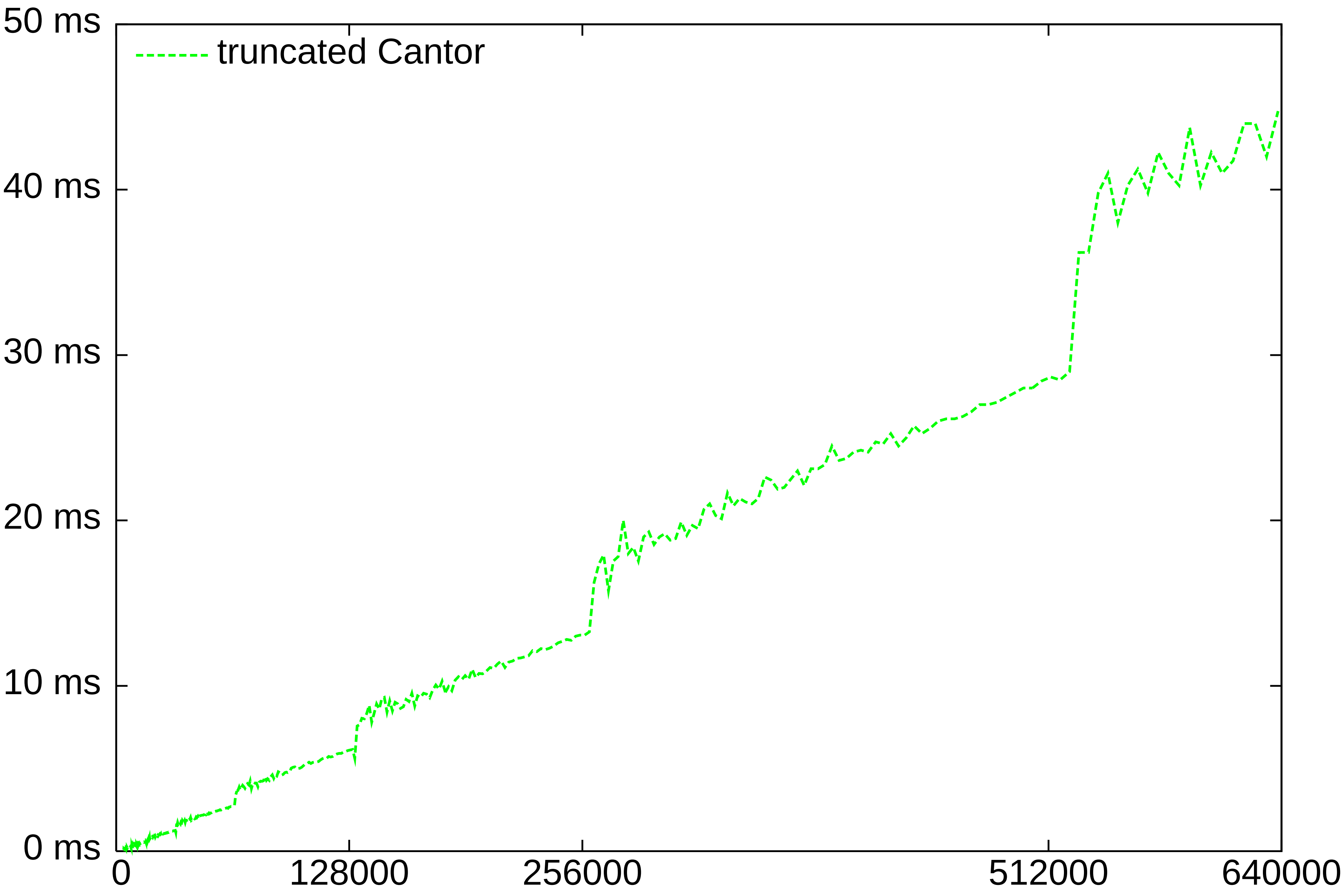
Performance of additive FFT



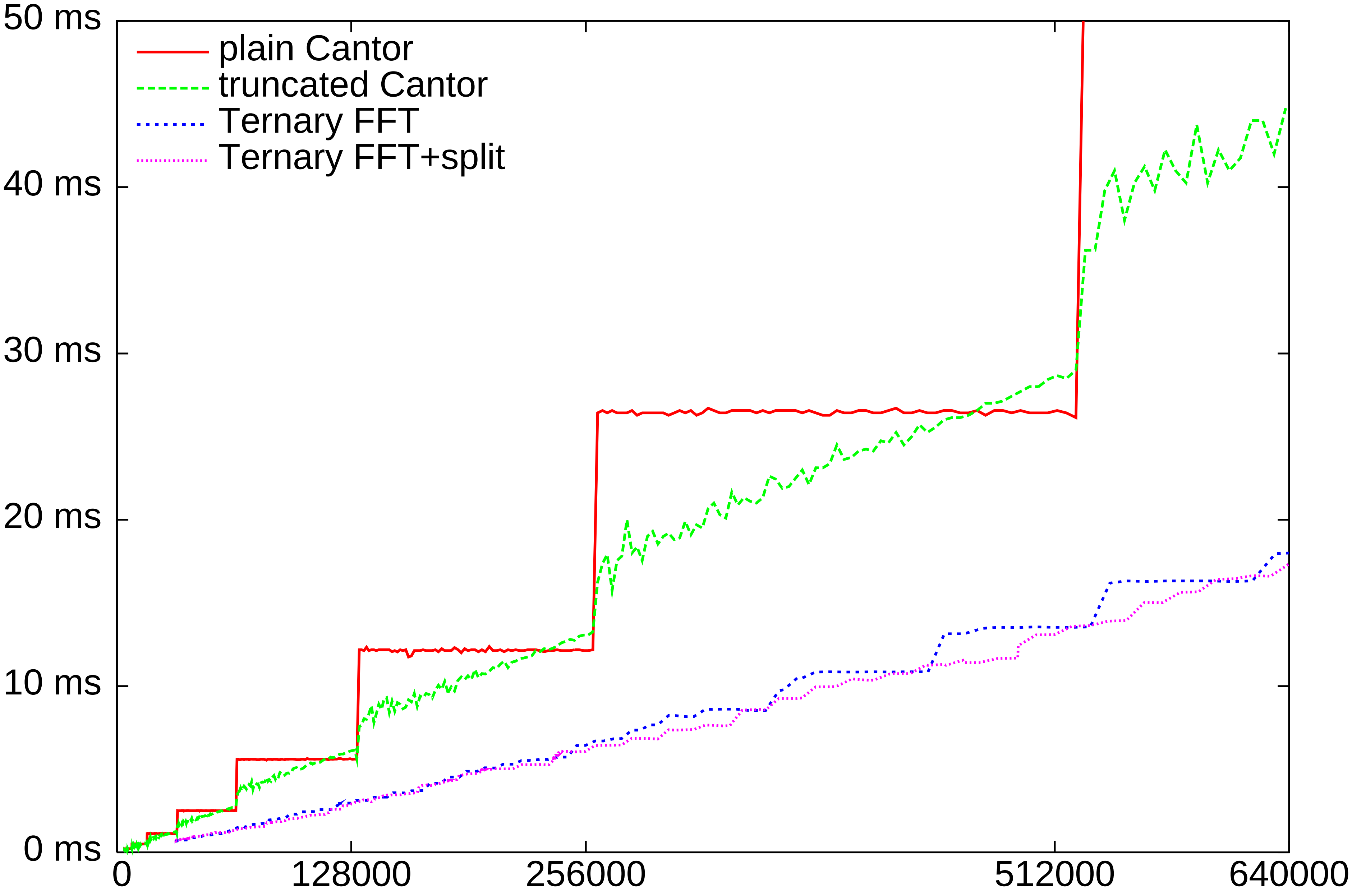
Performance of additive FFT



Performance of additive FFT



Comparison Cantor – Schönhage



Comparison Cantor – Schönhage

A word of caution:

- Additive FFT has cheap pointwise products.
- Ternary FFT has cheap evaluation / interpolation.

When transforms can be reused (matrices over $\mathbb{F}_2[x]$), additive FFT wins.
Example for $\deg ab < 2^{20}$:

- Additive FFT: 57 ms, 2.3 ms in pointwise mults.
- Ternary FFT: 28 ms, 18 ms in pointwise mults.
- $n \times n$ matrix mult: $c_{\text{eval/interp}} * n^2 + c_{\text{pointwise}} * n^3$
- Additive FFT faster for 3×3 matrices and above.

Conclusion

- Significant speed-ups over existing software.
- URL: `gf2x.gforge.inria.fr` (versions > 0.9 no longer have the additive FFT, because not routinely tested).
- Usable as an add-on to NTL 5.5: `NTL_GF2X_LIB=<path>`.

In the works: an update to expose the different steps of the transform, for algorithms where caching transforms is desired.