Faster Multiplication in $\mathbb{F}_2[X]$

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

INRIA LORRAINE

CACAO

Post-doc position

The CACAO project (Nancy, France) offers a post-doc position for working on the Number Field Sieve ; in particular:

- implementation concerns.
- Software speed-up.
- Large scale distribution.

This is part an ongoing project on NFS, named CADO:

`http://cado.gforge.inria.fr/`

Faster Multiplication in $\mathbb{F}_2[X]$

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

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).

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.

- We measure the best window size with experiments.
- 64×64: ~ 75 Intel `core2` cycles ; ~ 85 AMD `k8` cycles.
- $64k \times 64$ would work the same way.

Using SIMD capabilities

What about 128×128 ?

- Karatsuba $\Rightarrow$ three 64×64.
- Schoolbook requires $a \times b_{\text{low}}$ and $a \times b_{\text{high}}$ $\Rightarrow$ 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 absolutely necessary.

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: $0, 1, \infty, x, x^{-1}$.
- Often better: $0, 1, \infty, x^{64}, x^{-64}$: avoids shifts.
- The degrees in recursive calls increase mildly.

See paper for timings.

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).
- Cantor’s additive FFT algorithm.
- Schönhage’s ternary FFT algorithm.

Cantor's additive FFT

Assume we are given fast polynomial multiplication in $F_k = \mathbb{F}_{2^{2^k}} = \mathbb{F}_2[\gamma]$.

We use it for multiplication in $\mathbb{F}_2[x]$.

- Separate coefficients of a and b in blocks of 2^{k-1} coefficients:
  - Write $a = A(x, x^{2^{k-1}})$, $A(x,t) = a_0(x) + a_1(x)t + a_2(x)t^2 + \cdots$.
  - $\tilde{a} = A(\gamma, t) \in F_k[t]$.
- $\tilde{a} \times \tilde{b}$ has coefficients in F_k.
- Since $\deg a_i b_j < 2^k$, then $c = a \times b$ is such that $\tilde{c} = \tilde{a} \times \tilde{b}$.

Multiplying in F_k

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 $W_i = \left\{\alpha \in \mathbb{F}_{2^{2^k}} \mid s_i(\alpha) = 0\right\} = \operatorname{Ker} s_i$.
  W_i is a sub-vector space of $\mathbb{F}_{2^{2^k}}$; $\dim W_i = i$.

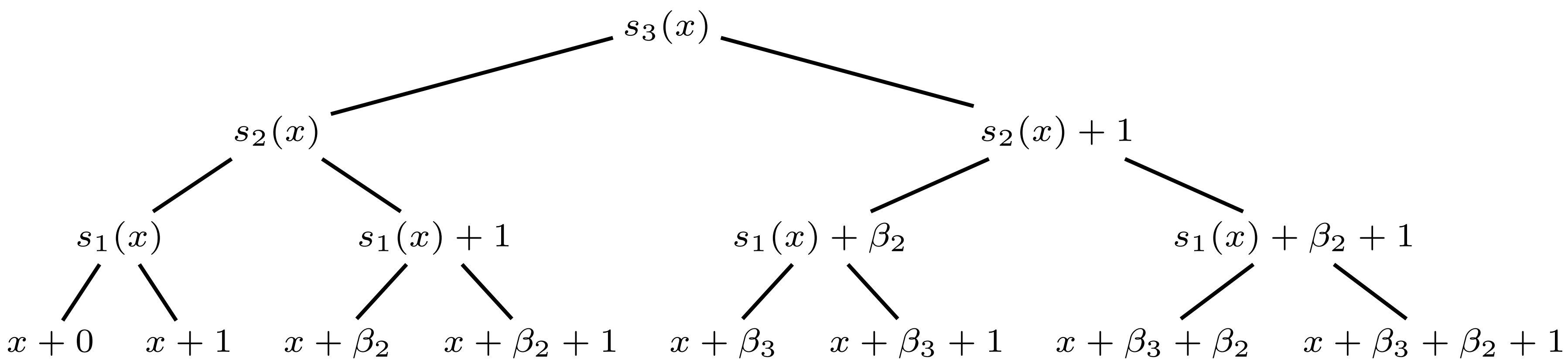
How do we multiply $h = f \times g$ in $F_k[x]$?

- Evaluate f and g at points of some W_i.
- multiply pointwise to obtain $\{f(\alpha) \times g(\alpha),\ \alpha \in W_i\}$.
- Interpolate: recover h from $\{f(\alpha) \times g(\alpha),\ \alpha \in W_i\}$.

Multiplying in F_k (2)

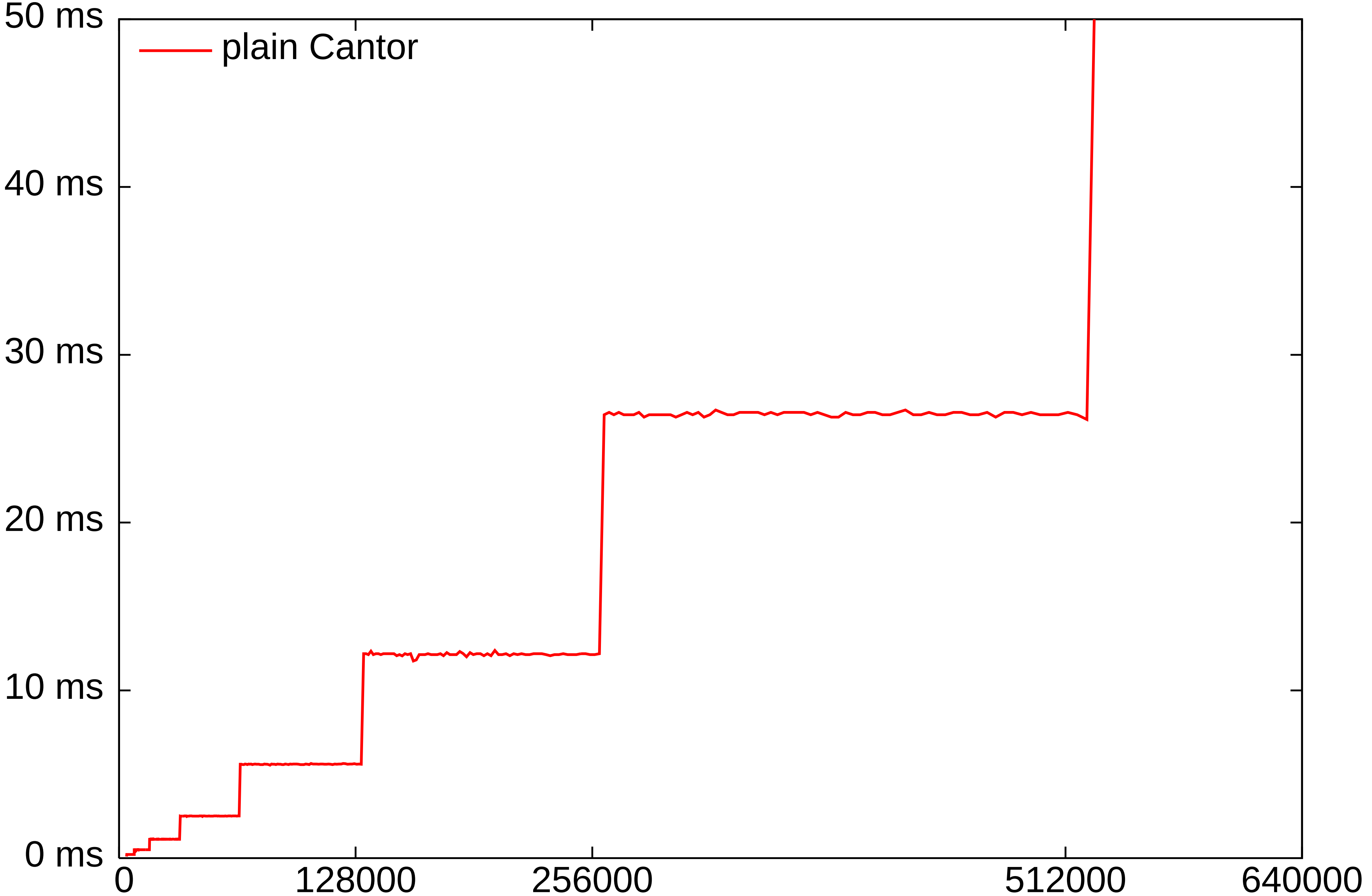
Multi-evaluating at W_i is done with a sub-product tree:

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

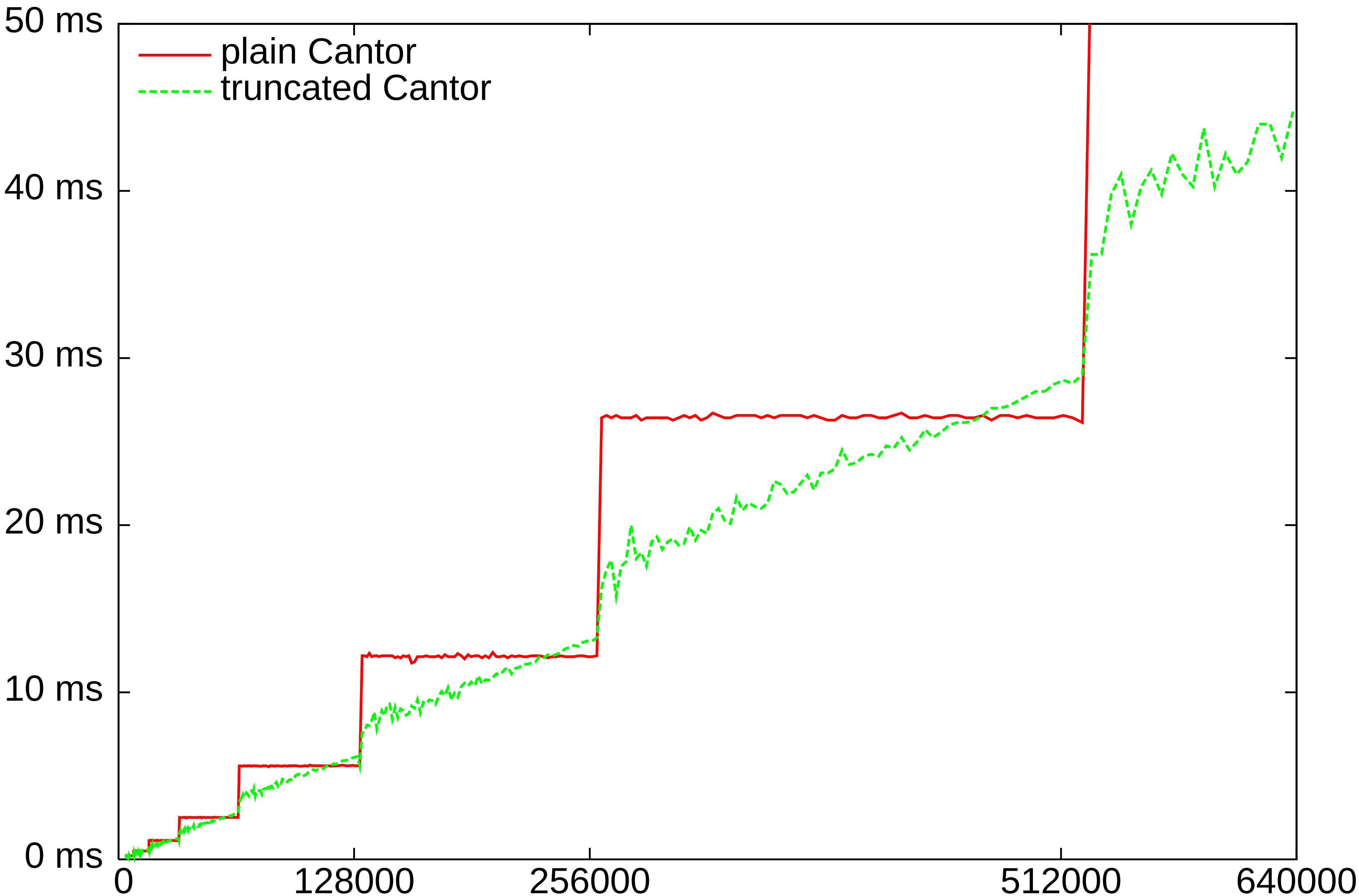


- right-child $= 1 +$ left-child.
- Only the constant coefficients are in extension fields.
- s_j is sparse, so reduction is cheap.

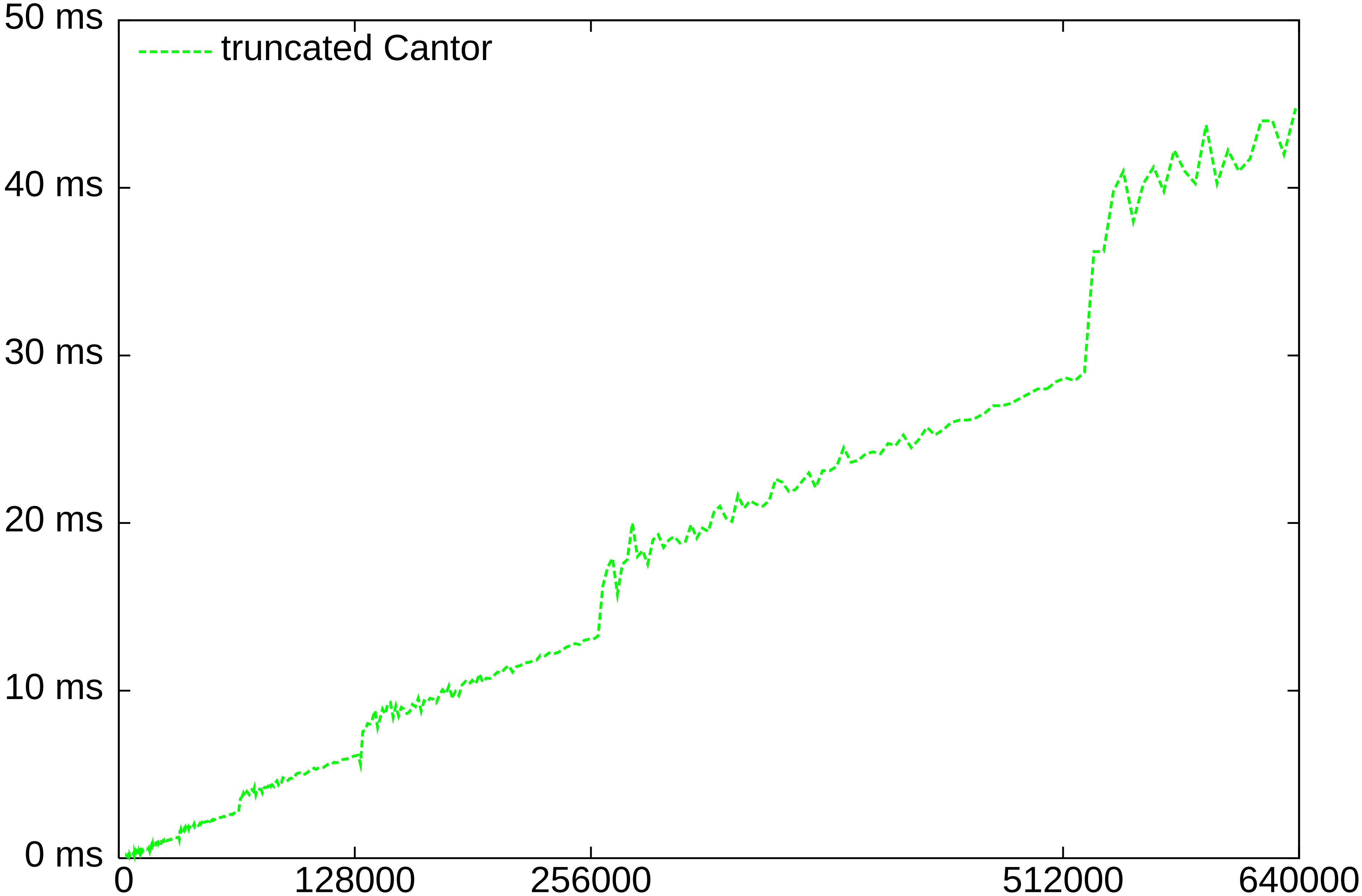
Performance of additive FFT



Performance of additive FFT



Performance of additive FFT



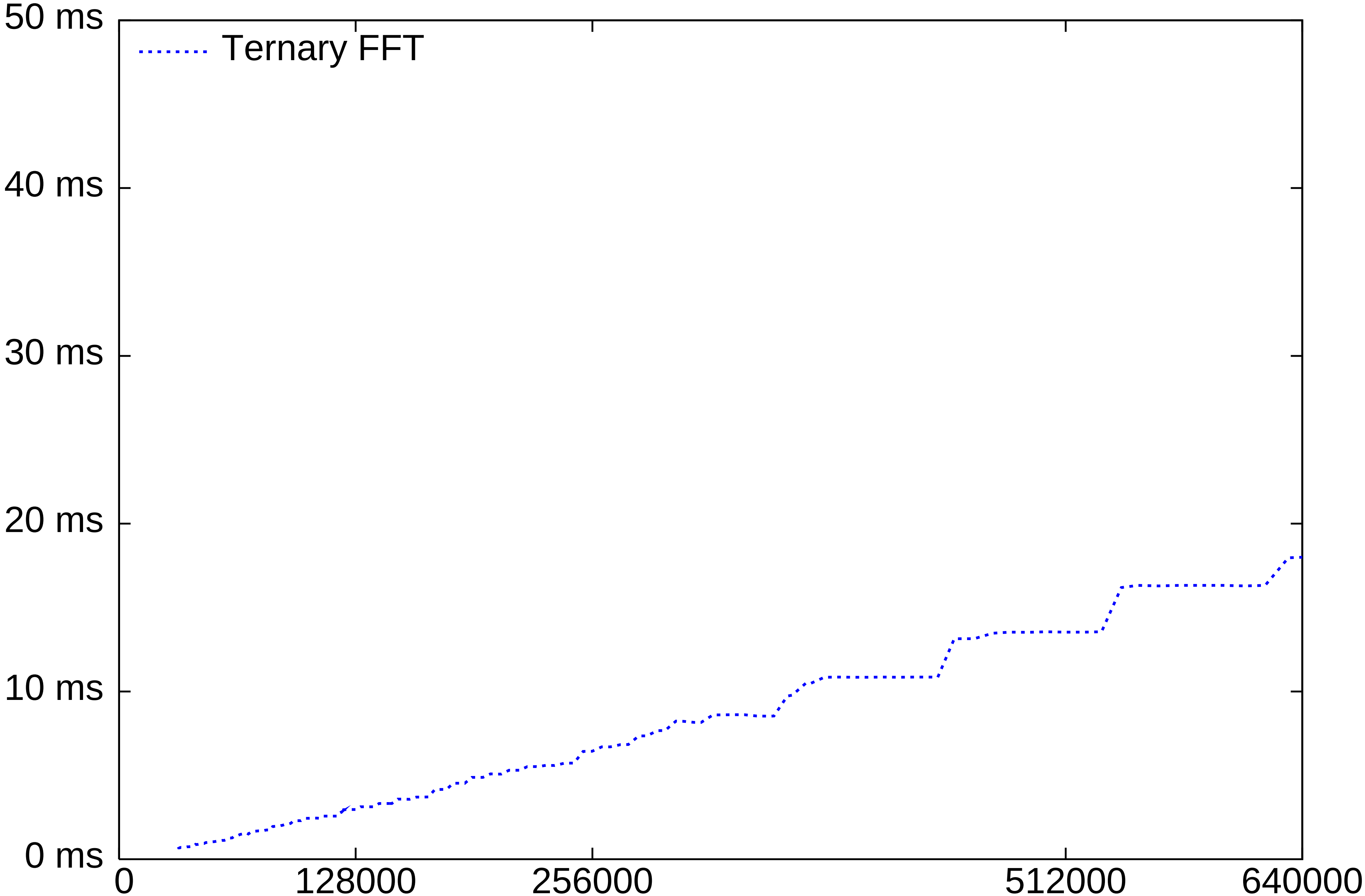
Schönhage's ternary FFT algorithm

- FFT typically calls for 2^n roots of unity ; bad for $\mathrm{char}\, K = 2$.
- Schönhage (1977): work in $R = \mathbb{F}_2[x]/x^{2L} + x^L + 1$, where $L = \lambda 3^{k-1}$.
- x^λ is a 3^k-th root of 1 in R.
- Use ternary FFT to multiply polynomials of degree $< 3^k$ in $R[t]$.
  - Evaluate $\hat{f} = \left\{ f(x^{\lambda i}),\ 0 \le i < 3^k \right\}$. (same for $\hat{g}$).
  - Multiply pointwise to obtain $\widehat{fg}$; multiplications in R: recurse.
  - Interpolate to recover fg ; FFT again since $\hat{\hat{f}} = f$.

Same clumping technique as before $\Rightarrow$ multiplication in $\mathbb{F}_2[x]$.

- In effect, we multiply modulo $x^N + 1$ (for some $N > \deg ab$).
- See details in paper.

Schönhage FFT



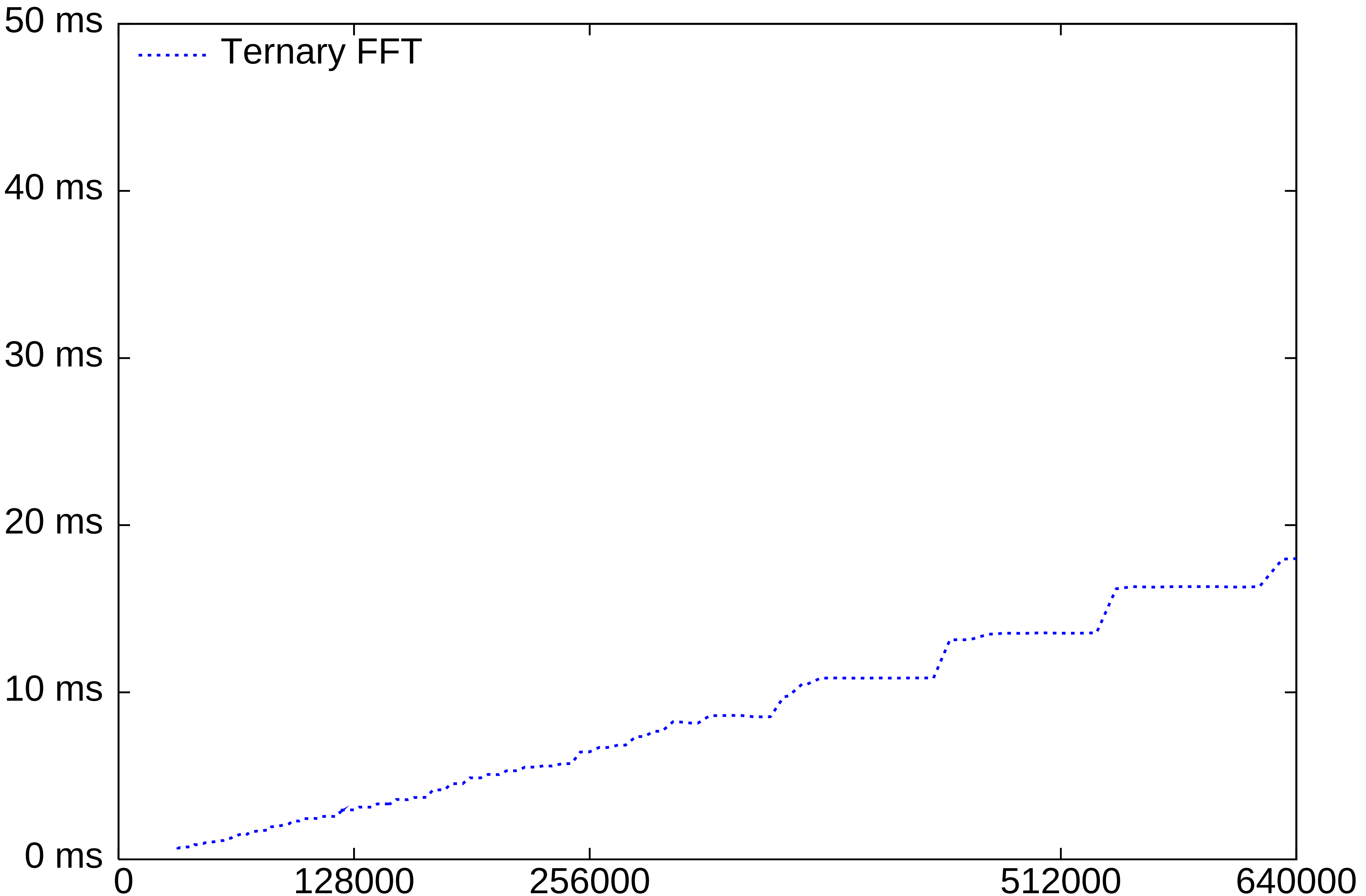
Splitting the ternary FFT

There is a (mild) staircase effect.

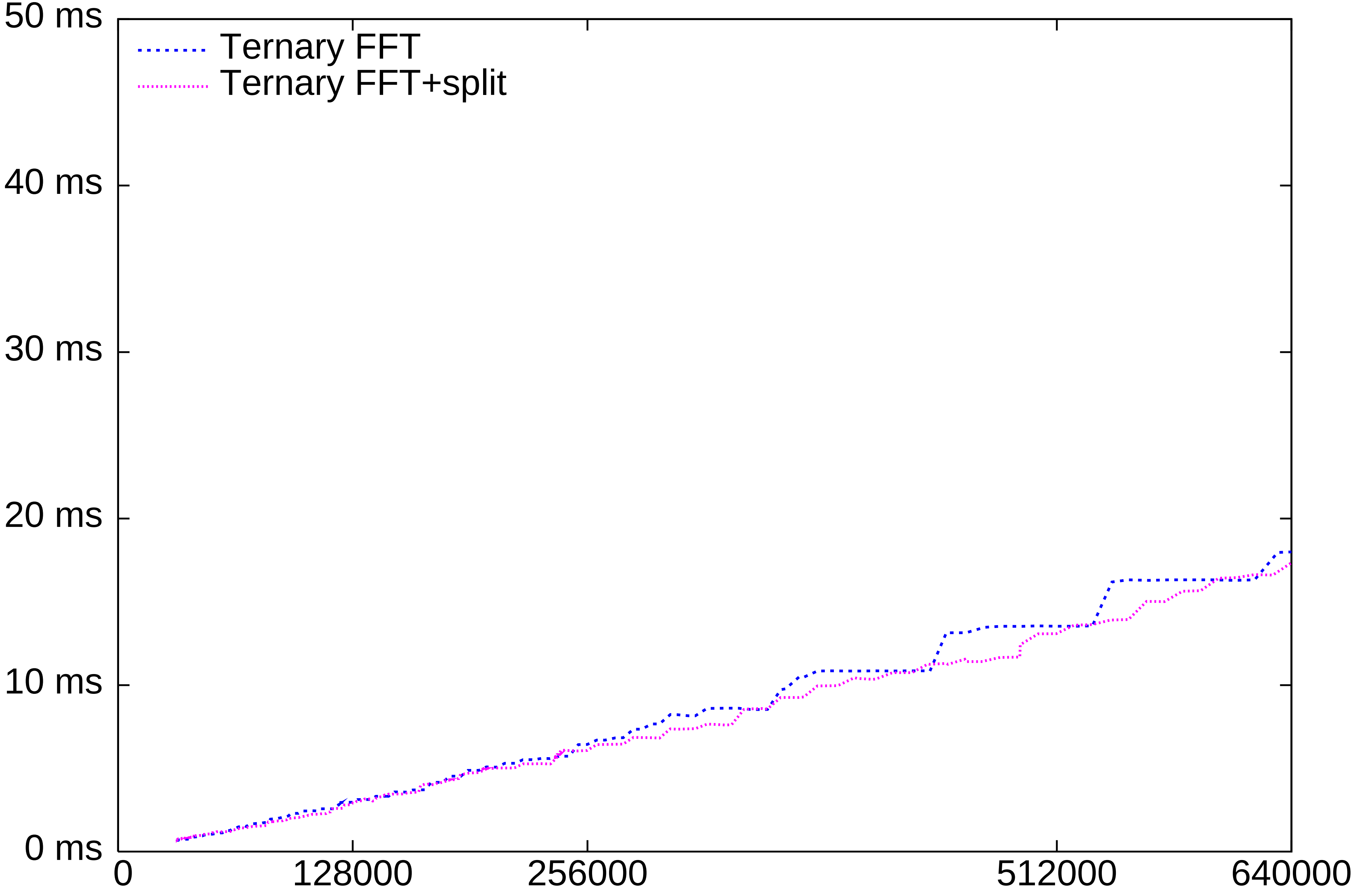
We can compute a product of degree $< N$ by splitting:

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

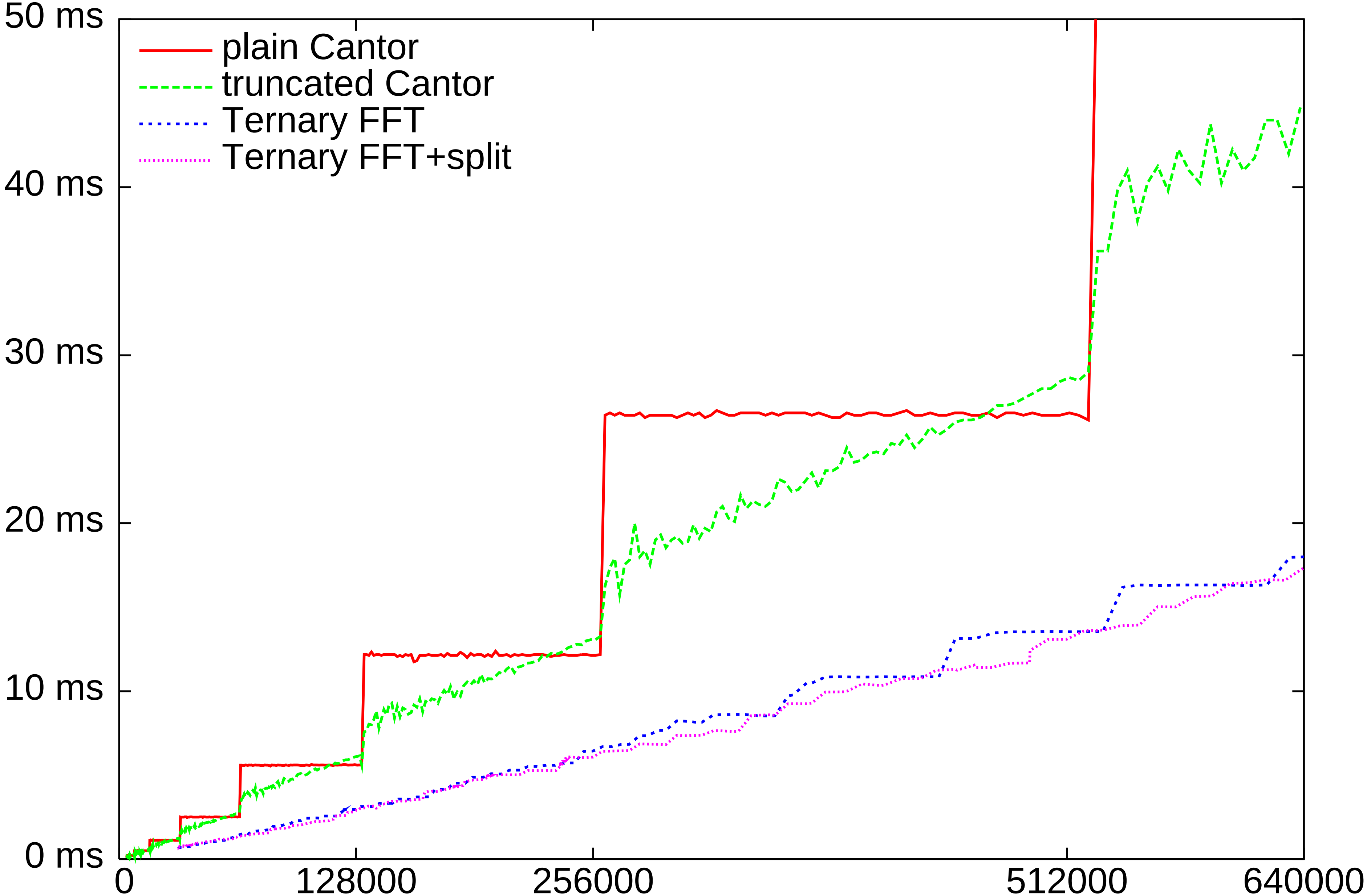
Schönhage FFT + splitting



Schönhage FFT + splitting



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.
- Openly available implementation of two FFT algorithms.
- Accessible from `rpbrent.com/gf2x.html`