FFT

Abstract. beefy floating point units

1. notation

$\mathbb{i}$	$\sqrt{-1}$
$\mathbb{e}$	$2.71\ldots$
π	$3.14\ldots$
$\mathrm{EvalPoly}(a, \vec{x})$	$\sum_{0\le i<n} x_i a^i$, $\vec{x} = (x_0, \ldots, x_{n-1})$
ω_b^a	$\mathbb{e}^{2a\pi\mathbb{i}/b}$ when the base ring is $\mathbb{C}$, Otherwise some compatible b^{th} root of unity.
$\bar{i}^k$	length k bit-reversal of i. Only defined for $0 \le i < 2^k$

2. fft/ifft

2.1. **Bit Reversal.** Unless the output of the fft is specifically needed to be in the usual order

$$\left\{\mathrm{EvalPoly}(\omega_{2^k}^i, \vec{x})\right\}_{0\le i<2^k}$$

there is no reason not to give the output in bit-reversed order

$$\left\{\mathrm{EvalPoly}(\omega_{2^k}^{\bar{i}^k}, \vec{x})\right\}_{0\le i<2^k}.$$

The reason is that the bit-reversed output is much simpler and faster because it groups similar outputs close together. For example, $\mathrm{EvalPoly}(1, \vec{x})$ and $\mathrm{EvalPoly}(-1, \vec{x})$ are very similar computationally and are right next to each other in the bit-reversed output, but are very far apart in the usual order. Therefore, we will restrict exclusively to bit-reversed outputs.

The usual sequence for calculating a length 16 fft follows the columns below and ends with the fft in the last column.

(1)

x_i	y_i	z_i	w_i	$\mathrm{fft}(\vec{x})$
x_0	$x_0 + x_8$	$y_0 + y_4$	$z_0 + z_2$	$w_0 + w_1$
x_1	$x_1 + x_9$	$y_1 + y_5$	$z_1 + z_3$	$(w_0 - w_1)$
x_2	$x_2 + x_{10}$	$y_2 + y_6$	$(z_0 - z_2)\omega_4^0$	$w_2 + w_3$
x_3	$x_3 + x_{11}$	$y_3 + y_7$	$(z_2 - z_3)\omega_4^1$	$(w_2 - w_3)$
x_4	$x_4 + x_{12}$	$(y_0 - y_4)\omega_8^0$	$z_4 + z_6$	$w_4 + w_5$
x_5	$x_5 + x_{13}$	$(y_1 - y_5)\omega_8^1$	$z_5 + z_7$	$(w_4 - w_5)$
x_6	$x_6 + x_{14}$	$(y_2 - y_6)\omega_8^2$	$(z_4 - z_6)\omega_4^0$	$w_6 + w_7$
x_7	$x_7 + x_{15}$	$(y_3 - y_7)\omega_8^3$	$(z_5 - z_7)\omega_4^1$	$(w_6 - w_7)$
x_8	$(x_0 - x_8)\omega_{16}^0$	$y_8 + y_{12}$	$z_8 + z_{10}$	$w_8 + w_9$
x_9	$(x_1 - x_9)\omega_{16}^1$	$y_9 + y_{13}$	$z_9 + z_{11}$	$(w_8 - w_9)$
x_{10}	$(x_2 - x_{10})\omega_{16}^2$	$y_{10} + y_{14}$	$(z_8 - z_{10})\omega_4^0$	$w_{10} + w_{11}$
x_{11}	$(x_3 - x_{11})\omega_{16}^3$	$y_{11} + y_{15}$	$(z_9 - z_{11})\omega_4^1$	$(w_{10} - w_{11})$
x_{12}	$(x_4 - x_{12})\omega_{16}^4$	$(y_8 - y_{12})\omega_8^0$	$z_{12} + z_{14}$	$w_{12} + w_{13}$
x_{13}	$(x_5 - x_{13})\omega_{16}^5$	$(y_9 - y_{13})\omega_8^1$	$z_{13} + z_{15}$	$(w_{12} - w_{13})$
x_{14}	$(x_6 - x_{14})\omega_{16}^6$	$(y_{10} - y_{14})\omega_8^2$	$(z_{12} - z_{14})\omega_4^0$	$w_{14} + w_{15}$
x_{15}	$(x_7 - x_{15})\omega_{16}^7$	$(y_{11} - y_{15})\omega_8^3$	$(z_{13} - z_{15})\omega_4^1$	$(w_{14} - w_{15})$

One problem with this approach is that each column accesses many different *twiddle factors* ω_b^a. In the case of a Schönhage–Strassen fft where the base ring is $\mathbb{Z}/(2^m+1)\mathbb{Z}$, this doesn't matter because each twiddle factor is a power of two and implemented via bit shifts. In other cases, these twiddle factors have to be either computed on the fly or precomputed and then retrieved from storage. In the case of precomputation, we have to have in memory the table

$$1, \omega_2^1, 1, \omega_4^1, 1, \omega_8^1, \omega_8^2, \omega_8^3, 1, \omega_{16}^1, \omega_{16}^2, \omega_{16}^3, \omega_{16}^4, \omega_{16}^5, \omega_{16}^6, \omega_{16}^7, 1, \dots \tag{2}$$

so that the columns can access this table sequentially. However, such a table is nice because once it is extended to accommodate an fft of a certain length, it can be reused for all ffts of smaller length.

By rearranging the twiddle factors as ($\omega = \omega_{16}$)

x_i	y_i	z_i	w_i	$\mathrm{fft}(\vec{x})$
x_0	$x_0 + \omega^0 x_8$	$y_0 + \omega^0 y_4$	$z_0 + \omega^0 z_2$	$w_0 + \omega^0 w_1$
x_1	$x_1 + \omega^0 x_9$	$y_1 + \omega^0 y_5$	$z_1 + \omega^0 z_3$	$w_0 + \omega^8 w_1$
x_2	$x_2 + \omega^0 x_{10}$	$y_2 + \omega^0 y_6$	$z_0 + \omega^8 z_2$	$w_2 + \omega^4 w_3$
x_3	$x_3 + \omega^0 x_{11}$	$y_3 + \omega^0 y_7$	$z_1 + \omega^8 z_3$	$w_2 + \omega^{12} w_3$
x_4	$x_4 + \omega^0 x_{12}$	$y_0 + \omega^8 y_4$	$z_4 + \omega^4 z_6$	$w_4 + \omega^2 w_5$
x_5	$x_5 + \omega^0 x_{13}$	$y_1 + \omega^8 y_5$	$z_5 + \omega^4 z_7$	$w_4 + \omega^{10} w_5$
x_6	$x_6 + \omega^0 x_{14}$	$y_2 + \omega^8 y_6$	$z_4 + \omega^{12} z_6$	$w_6 + \omega^6 w_7$
x_7	$x_7 + \omega^0 x_{15}$	$y_3 + \omega^8 y_7$	$z_5 + \omega^{12} z_7$	$w_6 + \omega^{14} w_7$
x_8	$x_0 + \omega^8 x_8$	$y_8 + \omega^4 y_{12}$	$z_8 + \omega^2 z_{10}$	$w_8 + \omega^1 w_9$
x_9	$x_1 + \omega^8 x_9$	$y_9 + \omega^4 y_{13}$	$z_9 + \omega^2 z_{11}$	$w_8 + \omega^9 w_9$
x_{10}	$x_2 + \omega^8 x_{10}$	$y_{10} + \omega^4 y_{14}$	$z_8 + \omega^{10} z_{10}$	$w_{10} + \omega^5 w_{11}$
x_{11}	$x_3 + \omega^8 x_{11}$	$y_{11} + \omega^4 y_{15}$	$z_9 + \omega^{10} z_{11}$	$w_{10} + \omega^{13} w_{11}$
x_{12}	$x_4 + \omega^8 x_{12}$	$y_8 + \omega^{12} y_{12}$	$z_{12} + \omega^6 z_{14}$	$w_{12} + \omega^3 w_{13}$
x_{13}	$x_5 + \omega^8 x_{13}$	$y_9 + \omega^{12} y_{13}$	$z_{13} + \omega^6 z_{15}$	$w_{12} + \omega^{11} w_{13}$
x_{14}	$x_6 + \omega^8 x_{14}$	$y_{10} + \omega^{12} y_{14}$	$z_{12} + \omega^{14} z_{14}$	$w_{14} + \omega^7 w_{15}$
x_{15}	$x_7 + \omega^8 x_{15}$	$y_{11} + \omega^{12} y_{15}$	$z_{13} + \omega^{14} z_{15}$	$w_{14} + \omega^{15} w_{15}$

(3)

there are still the same number of twiddle factor multiplications, but each twiddle factor itself can be reused in each column. Also, as with the previous method, there is a universal (bit-reversed) table

$$1, \quad \omega_2^1, \quad \omega_4^1, \omega_4^3, \quad \omega_8^1, \omega_8^5, \omega_8^3, \omega_8^7, \quad \omega_{16}^1, \omega_{16}^9, \omega_{16}^5, \omega_{16}^{13}, \omega_{16}^3, \omega_{16}^{11}, \omega_{16}^7, \omega_{16}^{15}, \quad \dots \tag{4}$$

that can be reused. The difference here is that the portion that is used for a specific fft is now half the size (making note of $\omega_4^3 = -\omega_4^1$, $\omega_8^5 = -\omega_8^1$, $\omega_8^7 = -\omega_8^3$, etc.).

Since the output of the fft is given in bit-reversed order, the inverse operation cannot simply replace ω by ω^{-1} and use the same calculation sequence. Thus the ifft has to invert the left-to-right operation by working from the right to the left and inverting each basic operation. This gives slightly different data access patterns but involves essentially the same calculations. For example, the operation $w_8 = z_8 + \omega^2 z_{10}, w_{10} = z_8 - \omega^2 z_{10}$ is inverted by $2z_8 = w_8 + w_{10}, 2z_{10} = \omega^{-2}(w_8 - w_{10})$ and the negative power $\omega^{-\bar{i}^k}$ can be looked up in the table by flipping all bits of i except the highest.

2.2. **Considerations for the base ring $\mathbb{C}$.** When arithmetic in the base ring necessarily includes roundoff error, it is important for the fft to be implemented as accurately as possible. Since both (1) and (3) compute the bit-reversed fft, we are free to use either. Using (3) for the forward transform and inverting (1) for the inverse transform gives calculation sequences for both that consist *entirely of fused-multiply-add operations*. The ifft will read often from the table (2), but this is of little consequence since large convolution lengths are not feasible for this base ring (see Table 1). One further issue that arises with the ifft when processing data with n-wide vectors is a factor of n increase in the size of

TABLE 1. Integer multiplication with complex 53 bit ffts: The two input numbers are viewed as polynomials evaluated at 2^m so that the fft has inputs in the range $[0, 2^m)$. The output coefficients from the ifft are eventually unreliable due to rounding errors: let $f(m)$ denote the maximum bit size of an answer which can be guaranteed correct by this algorithm.

m	upper bound on $f(m)$
22	3008
21	11968
20	30464
19	117632
18	417280
17	1246592
16	1984000

twiddle table (2). Since the table (2) is used only for the calculations in lane number 0, there must be $n-1$ "twists" of this data available for the other vector lanes.

The case of real input and output data is important and can be optimized. Feeding in purely real data to the fft and getting back real data from the ifft represents only a 50% utilization of the circuits (1) or (3), since the data possesses certain symmetries under complex conjugation in each column of the calculation. First, and most elementary, two real data sets $\vec{a}$ and $\vec{b}$ of the same length can be transformed with one complex transformation of the same length, and this is nothing but the linearity of the fft operation:

$$\begin{aligned}\text{EvalPoly}(\omega, \vec{a} + \mathbb{i}\vec{b}) &= \text{EvalPoly}(\omega, \vec{a}) + \mathbb{i}\,\text{EvalPoly}(\omega, \vec{b}) \\ \overline{\text{EvalPoly}(\omega^{-1}, \vec{a} + \mathbb{i}\vec{b})} &= \text{EvalPoly}(\omega, \vec{a}) - \mathbb{i}\,\text{EvalPoly}(\omega, \vec{b})\end{aligned}$$

This is fine for doubling the processing speed of an even number of data sets of the same length, but if we would like to process one data set twice as fast, the following identity can be used.

$$\text{EvalPoly}(\omega, (x_0, \ldots, x_{2n-1})) = \text{EvalPoly}(\omega^2, (x_0, x_2, \ldots, x_{2n-2})) + \omega\,\text{EvalPoly}(\omega^2, (x_1, x_3, \ldots, x_{2n-1})).$$

Multiplication of real coefficients $\{a_i\}$ and $\{b_i\}$ to form product coefficients $\{c_i\}$ can then take the form

$$\begin{array}{c} a_0 + \mathbb{i}b_0 \\ a_1 + \mathbb{i}b_1 \\ \cdots \\ a_{15} + \mathbb{i}b_{15} \end{array} \overset{\text{fft}_{16}}{\Longrightarrow} \begin{array}{c} f_0 \\ f_8 \\ f_4 \\ f_{12} \\ \cdots \\ f_{14} \\ f_1 \\ \cdots \\ f_7 \\ f_{15} \end{array} \Longrightarrow \begin{array}{r} \hat{a}_0 \cdot \hat{b}_0 = \hat{c}_0 \\ \hat{a}_8 \cdot \hat{b}_8 = \hat{c}_8 \\ \hat{a}_4 \cdot \hat{b}_4 = \hat{c}_4 \\ \cdots \\ \hat{a}_{14} \cdot \hat{b}_{14} = \hat{c}_{14} \\ \hat{a}_1 \cdot \hat{b}_1 = \hat{c}_1 \\ \cdots \\ \hat{a}_7 \cdot \hat{b}_7 = \hat{c}_7 \\ \hat{a}_{15} \cdot \hat{b}_{15} = \hat{c}_{15} \end{array} \Longrightarrow \begin{array}{l} \hat{e}_0 + \mathbb{i}\hat{o}_0 \\ \hat{e}_4 + \mathbb{i}\hat{o}_4 \\ \hat{e}_2 + \mathbb{i}\hat{o}_2 \\ \hat{e}_6 + \mathbb{i}\hat{o}_6 \\ \hat{e}_1 + \mathbb{i}\hat{o}_1 \\ \hat{e}_5 + \mathbb{i}\hat{o}_5 \\ \hat{e}_3 + \mathbb{i}\hat{o}_3 \\ \hat{e}_7 + \mathbb{i}\hat{o}_7 \end{array} \overset{\text{ifft}_8}{\Longrightarrow} \begin{array}{l} c_0 + \mathbb{i}c_1 \\ c_2 + \mathbb{i}c_3 \\ c_4 + \mathbb{i}c_5 \\ c_6 + \mathbb{i}c_7 \\ c_8 + \mathbb{i}c_9 \\ c_{10} + \mathbb{i}c_{11} \\ c_{12} + \mathbb{i}c_{13} \\ c_{14} + \mathbb{i}c_{15} \end{array}$$

Denoting by $\bar{i}$ the result of flipping all bits in i but its lowest (set) bit, the $\hat{a}_i$ and $\hat{b}_i$ are recovered from the f_i, and the e_i and the o_i are recovered from the $\hat{c}_i$ by

$$\begin{aligned} f_i &= \hat{a}_i + \mathbb{i}\hat{b}_i, \quad \hat{c}_{i+0} = \hat{e}_i + \omega^i \hat{o}_i \\ \overline{f_{\bar{i}}} &= \hat{a}_i - \mathbb{i}\hat{b}_i, \quad \hat{c}_{i+8} = \hat{e}_i - \omega^i \hat{o}_i \end{aligned}.$$

3. TRUNCATION

If the idea of zero padding the real input data to complex data in the previous section sounded bad, then the idea of zero padding the overall input data to the next power-of-two size should sound even worse. For this reason, a *truncated fft* is necessary, which assumes certain portions of the input and output are zero. If the desired output length is denoted by n we expect runtime $\approx n \log n$ so the more constant the ratio is, the better the truncation is working. Figure 1 show a graph of such a ratio when the fft is performed over $\mathbb{F}_p$ for a 50 bit prime p and the input is further truncated to length $n/2$, which corresponds to assuming the top half of the n inputs are zero. For example, this is exactly the situation when squaring a polynomial. As expected, the truncated fft/ifft performs worst right after a power of two, but slightly unexpected is how bad the fft is there after $n > 2^{23}$ and how well the ifft is doing there. There are performance bumps at $3 \cdot 2^n$, and, rather surprisingly, the non-truncated ifft is eventually beating the non-truncated fft by about 7%. This is surprising for the non-truncated version because they are performing the exact same calculations, just in a different order. It is sometimes worth optimizing the basecases of these algorithms, and the result of optimizing only the length 16 basecases of the non-truncated fft and ifft is shown in Figure 2, where the fft is still generally lagging behind the fft. The small sizes where also given more stable timings in this graph by repeated calls to the function. Finally, the corresponding graph for FLINT's Schönhage–Strassen fft is shown in Figure 3, where very large coefficients had to be used to ensure that the graph can continue to $n = 2^{18}$ while keeping the coefficient size constant.

FIGURE 1. Truncation effectiveness with 50 bit coefficients

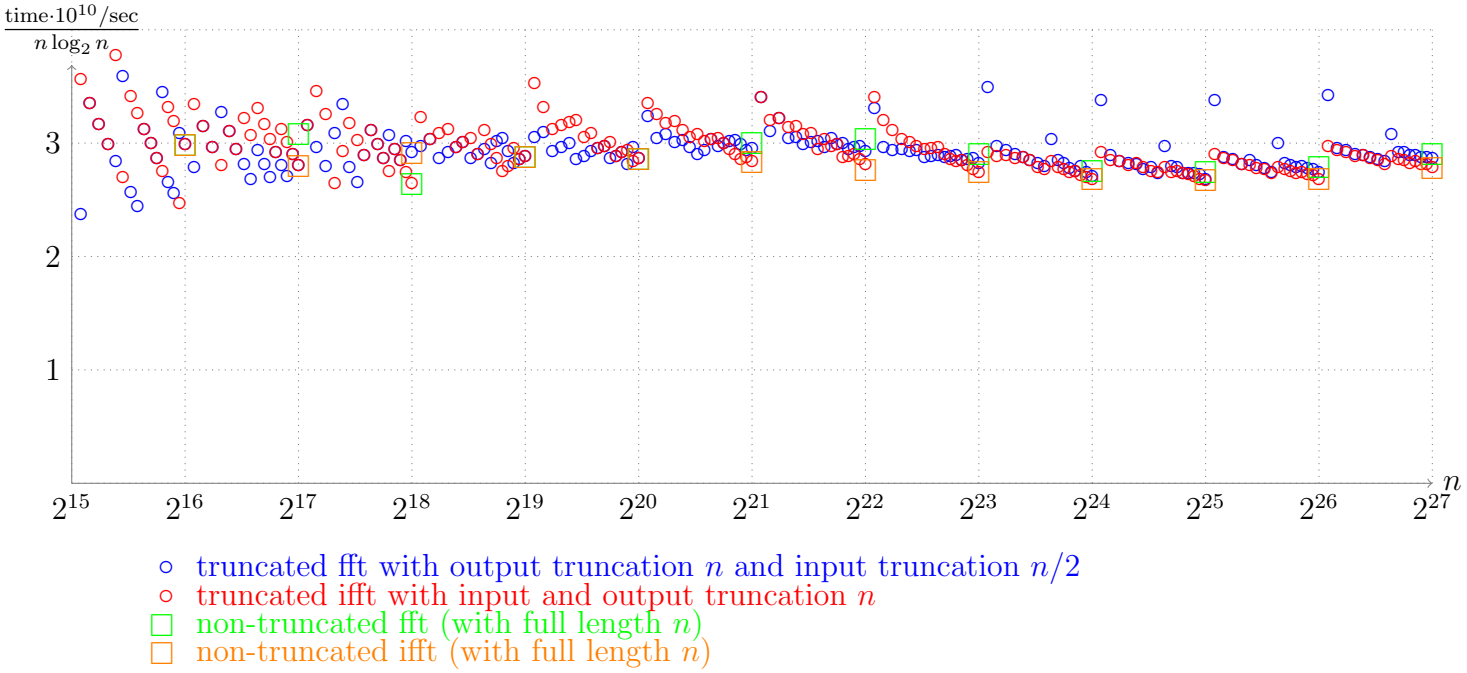


FIGURE 2. Truncation effectiveness with 50 bit coefficients and optimized length 16 basecase

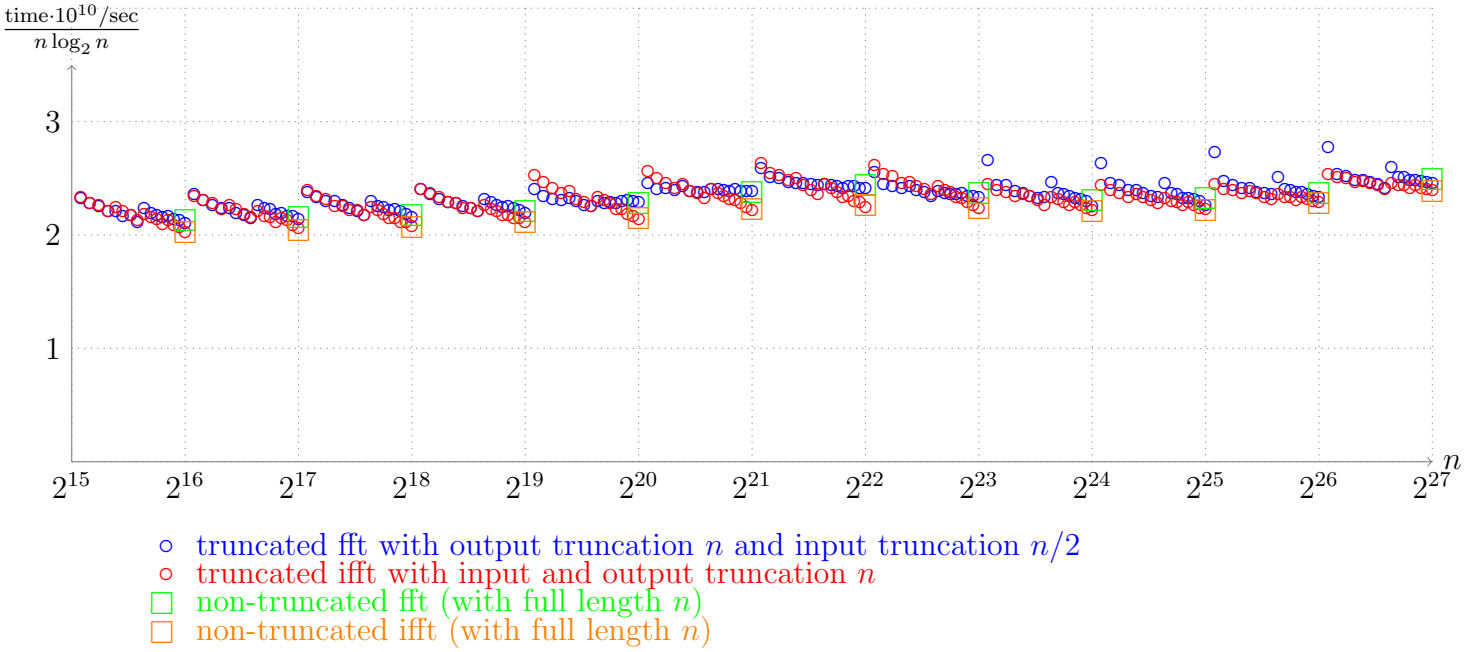
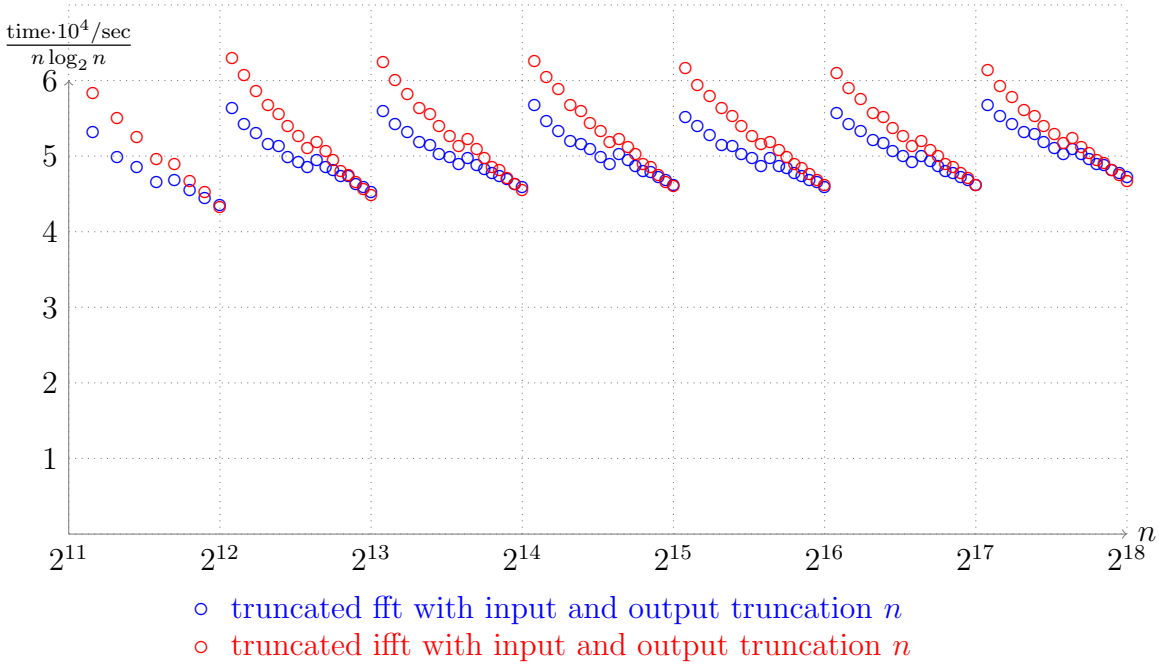


FIGURE 3. Truncation effectiveness of {i}`fft_mfa_truncate_sqrt2` with 2^{16} bit coefficients



4. INTEGER MULTIPLICATION

Figures 4, 5 and ? show timings for three integer multiplication algorithms:

- variable prime fft: between three and nine 50 bit primes are used along with the truncated matrix Fourier algorithm. Intended for large operands.
- four prime fft: four fixed 50 bit primes are used to combat the slow nature of a variable number of primes at smaller sizes.

- complex fft: 53 bit complex floating point with the real optimizations of Section 2.2. No answers here are provably correct, but coefficient sizes were taken up to about 75% of the observed failure points in (1). No truncation is performed in the fft because both ends of the top half are needed due to the real optimizations, and implementing a double truncated fft did not seem worth the effort. No truncation is performed in the ifft either as this seemed to have a disastrous effect on the accuracy. Instead we supplement the fft sizes with small odd multiples of powers two.

For each plotted value of n, the average, maximum, and minimum timings of integer multiplications where the product always has n bits and the smaller operand ranges between $n/2$ and $n/4$ bits are shown with three dots.

FIGURE 4. n bit integer product on mobile Zen 2

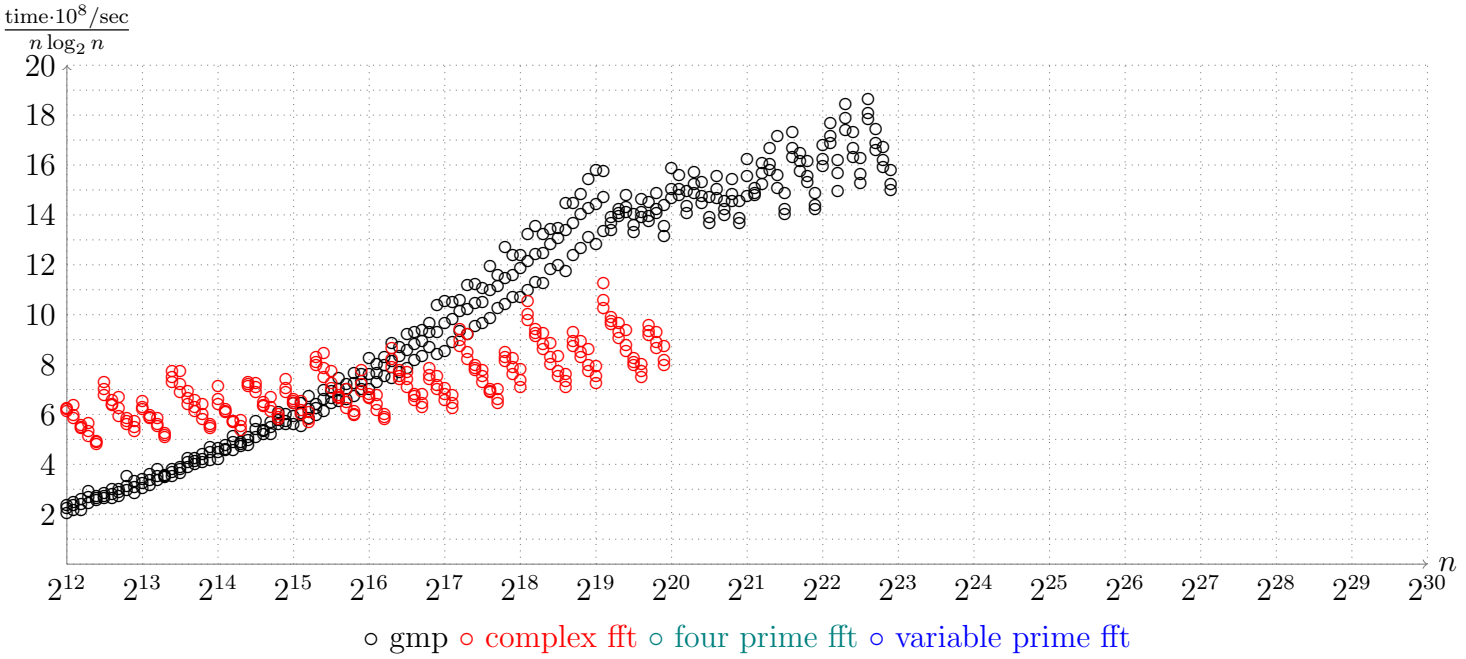
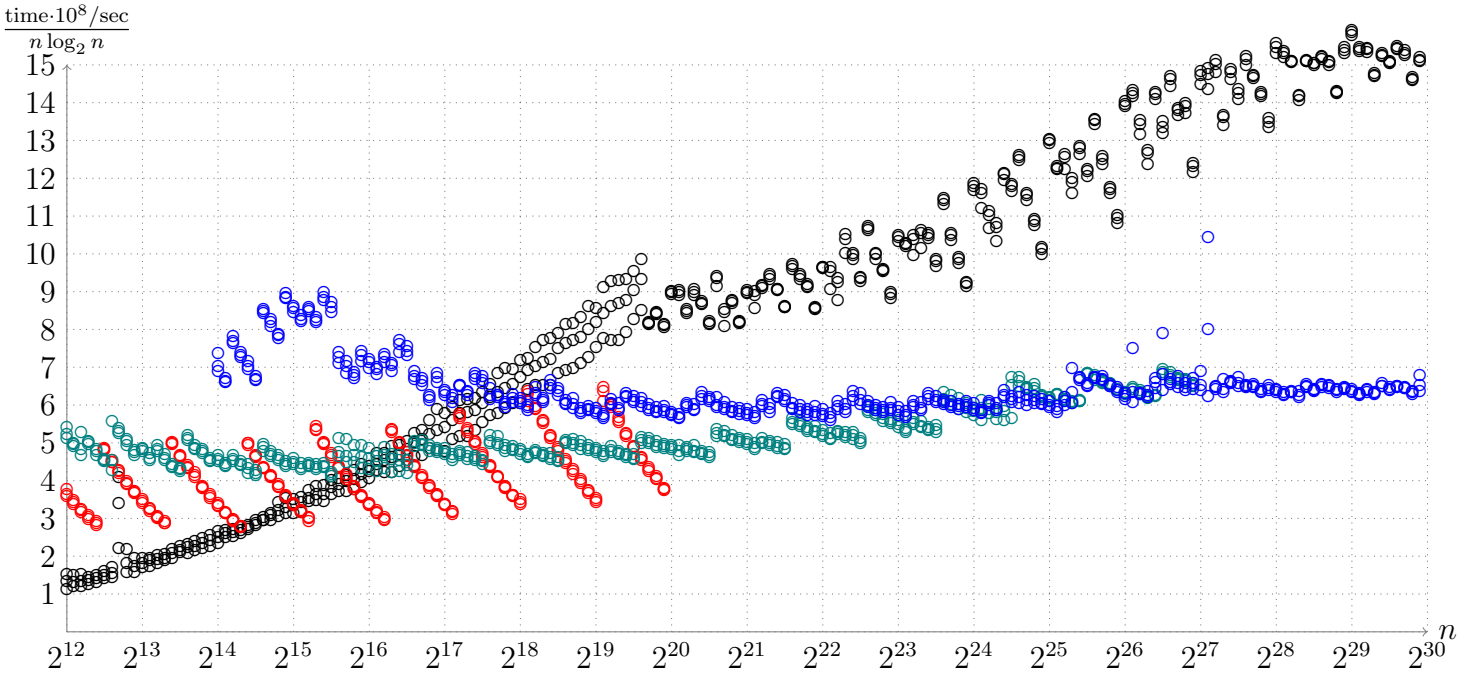


FIGURE 5. n bit integer product on mobile Zen 2



5. FFT ON SIZES OTHER THAN POWERS OF TWO

Suppose the vector to transform has length thrice a power of two: $x_0, x_1, \ldots, x_{3\cdot 2^k-1}$. If, for each $0 \le i < 2^k$, we first perform

$$\begin{aligned} y_i^{(0)} &= x_{i+0\cdot 2^k} + x_{i+1\cdot 2^k} + x_{i+2\cdot 2^k} \\ y_i^{(1)} &= \omega_{3\cdot 2^k}^{i}(x_{i+0\cdot 2^k} + \omega_3^1 x_{i+1\cdot 2^k} + \omega_3^{-1} x_{i+2\cdot 2^k}) \\ y_i^{(2)} &= \omega_{3\cdot 2^k}^{-i}(x_{i+0\cdot 2^k} + \omega_3^{-1} x_{i+1\cdot 2^k} + \omega_3^1 x_{i+2\cdot 2^k}), \end{aligned}$$

then the three ffts of the length 2^k sequences $y^{(0)}, y^{(1)}, y^{(2)}$ will give the fft of the original sequence. However, while the transforms of $y^{(0)}$ and $y^{(1)}$ will have the same ordering, the sequence $y^{(2)}$ will transform to a different ordering due to the negative power. More precisely,

$$\begin{aligned} \text{EvalPoly}(\omega_{3\cdot 2^k}^{3i}, x) &= \text{EvalPoly}(\omega_{2^k}^{i}, y^{(0)}) \\ \text{EvalPoly}(\omega_{3\cdot 2^k}^{3i+1}, x) &= \text{EvalPoly}(\omega_{2^k}^{i}, y^{(1)}) \\ \text{EvalPoly}(\omega_{3\cdot 2^k}^{3i+2}, x) &= \text{EvalPoly}(\omega_{2^k}^{i+1}, y^{(2)}). \end{aligned}$$