On the Flop and Flap Count of the 2,8-Split-Radix FFT

Paul Caprioli
High Performance Kernels LLC
paul@hpkfft.com

***Abstract*—We define *flap* to mean a floating point amalgamated operation, i.e., an add, multiply, or multiply-add operation, and derive asymptotic flop and flap counts for the radix-2, radix-4, radix-8, 2,4-split-radix, and 2,8-split-radix FFT algorithms. We find that, of these, the 2,8-split-radix algorithm requires the fewest flaps. Lastly, we opine on real-value scaling of an FFT.**

I. Aperitivo

The Fast Fourier Transform (FFT) maps a sequence of N complex numbers in the time domain

$$\mathbf{x} = \langle x_0, x_1, \ldots, x_{N-1} \rangle$$

into another sequence of N complex numbers in the frequency domain

$$\mathcal{F}(\mathbf{x}) = \mathbf{X} = \langle X_0, X_1, \ldots, X_{N-1} \rangle$$

such that

$$X_k = \sum_{n=0}^{N-1} \omega_N^{kn}\, x_n \tag{1}$$

where

$$\omega_N = \exp\left(\frac{-2\pi i}{N}\right) \quad \text{and} \quad i = \sqrt{-1}.$$

The FFT has been studied extensively, and in this brief article we cannot do justice to the resulting body of prior art. The background section of “High Performance Kernels for FFT via Modern C++” by Caprioli and Jenkins [3] offers ample motivation and references.

II. Antipasto

A. Flops and Flaps

In this paper, we make the usual assumption that the values of ω_N^n have been precomputed for all $n \in \{0, 1, \ldots, N-1\}$. Note that these are points on the unit circle in the complex plane. Since typically a very large number of FFTs need to be computed for a given length N, it makes sense to do this work up-front. (If you’ve downloaded this paper onto a mobile device, consider how many FFTs were used to accomplish that feat.) Therefore, we will ignore division and trigonometric function evaluation and state the following:

Definition 1. A floating point operation (flop) is the addition, subtraction, or multiplication of two floating point numbers.

The IEEE Standard for Floating-Point Arithmetic [8] also specifies a three operand fused multiply-add (FMA) which calculates $(a \times b) + c$ without intermediate rounding. Hardware is often designed to execute an FMA instruction with the same latency and throughput as a multiply instruction despite the former’s counting as two flops. In fact, overall throughput is shared, as a multiply is effectively implemented as if it were $(a \times b) + 0$. This, not the fused computation with only final rounding, motivates:

Definition 2. A floating point amalgamated operation (flap) is a fused multiply-add, a fused multiply-subtract, or a flop that is not itself a component of the aforementioned operations.

Hardware vendors generally agree that an FMA flap is two flops and proceed to tout their floating point capability in terms of flops per second. That is fine and reasonable, and naturally they use FMA-centric benchmarks to showcase their results. But when comparing the computational complexity of FFT algorithms, it makes sense to consider flaps. For example, $a \times (b + c)$ is also two flops, but unlike an FMA, it requires two flaps. An algorithm that replaces this with an FMA would, ceteris paribus, be preferred.

B. Complex Multiplication

The standard 6 flop implementation of a complex product uses 4 flaps, as implied by the extra parentheses below:

$$\begin{aligned} xy &= (a + bi)(c + di) \\ &= \big(ac - (bd)\big) + \big(ad + (bc)\big)i \end{aligned} \tag{2}$$

We will be ignoring data movement in this paper, but that is not meant to suggest it is unimportant. Performance is highly dependent on loads, stores, and how the memory access patterns interact with the cache hierarchy. Furthermore, parallelizing using SIMD vector registers is essential to good performance, and this requires data shuffling instructions, which are also not considered when counting flops or flaps.

The quality of an algorithm’s implementation, though hard to define, is crucial to its performance. For example, the complex multiplication code given in Intel’s 2011 (and later) optimization manual [6] takes advantage of hardware’s ability to perform basic shuffles on load ports and is easily updated to use an FMA as follows:

```
vmovsldup (%rax), %ymm0
vmovshdup (%rax), %ymm1
vmovups   (%rdx), %ymm4
vshufps $0xB1, %ymm4, %ymm4, %ymm5
vmulps         %ymm1, %ymm5, %ymm5
vfmaddsub231ps %ymm0, %ymm4, %ymm5
```

Assuming that rax contains a pointer to the sequence

$$\langle 0+1i,\ 2+3i,\ \ldots\rangle$$

and rdx to

$$\langle 4+5i,\ 6+7i,\ \ldots\rangle,$$

the code above sets the registers as follows:

```
ymm0 = { 0, 0,  2,  2, ...}
ymm1 = { 1, 1,  3,  3, ...}
ymm4 = { 4, 5,  6,  7, ...}
ymm5 = { 5, 4,  7,  6, ...}
ymm5 = { 5, 4, 21, 18, ...}
ymm5 = {-5, 4, -9, 32, ...}
```

Note that only the last two instructions[1] perform flops; the first four are data movement. Since each ymm register holds four complex points (eight single precision floating point numbers), the multiply is 8 flaps and is also 8 flops. The FMA is 8 flaps and is 16 flops. In summary, the operation count is exactly

	(N)
ADD	0
MUL	2
FMA	2
flops	6
flaps	4

where N is the number of complex multiplications.

III. Primo

A. Radix-2 FFT

The radix-2 decimation-in-time decomposition [4] can be derived by separating the original equation (1) into even and odd terms:

$$X_k = \sum_{n=0}^{N/2-1} \omega_{N/2}^{kn} x_{2n} + \omega_N^k \sum_{n=0}^{N/2-1} \omega_{N/2}^{kn} x_{2n+1} \tag{3}$$

In the odd-term summation above, we have made use of the fact that

$$\omega_N^{k(2n+1)} = \omega_N^{2kn+k} = \omega_{N/2}^{kn}\,\omega_N^k$$

and pulled the last factor, which is independent of n, outside the summation. It is called a twiddle factor.[2]

Observing that each summation in (3) is of the form (1) leads to the radix-2 recursive algorithm shown in the column on the right. It is worth mentioning that the subtraction arises since

$$\omega_N^{k+N/2} = \omega_N^k\,\omega_N^{N/2} = \omega_N^k\,e^{-\pi i} = -\omega_N^k,$$

and the resulting opportunity for reusing subexpressions is the reason for producing output points two at a time.

Radix-2 FFT

```
input   : $x_0, x_1, \ldots, x_{N-1}$
output  : $X_0, X_1, \ldots, X_{N-1}$
requires: $N = 2^p$ for $p \geq 0$

function fft_2<N>
    if $N < 2$ then
        $X_0 \leftarrow x_0$
    else
        $\omega \leftarrow \exp(-2\pi i/N)$
        $\mathbf{u} \leftarrow$ fft_2<N/2>($x_0, x_2, \ldots, x_{N-2}$)
        $\mathbf{v} \leftarrow$ fft_2<N/2>($x_1, x_3, \ldots, x_{N-1}$)
        for $k \leftarrow 0$ to $N/2-1$ do
            $X_k \leftarrow u_k + \omega^k v_k$
            $X_{k+N/2} \leftarrow u_k - \omega^k v_k$
        end
    end
end
```

The **for** loop of function fft_2<N> above iterates $N/2$ times and contains one complex multiplication (by a twiddle factor) and two complex additions. Note that we do not distinguish subtractions from additions in our accounting.

To count the total number of operations required by this algorithm, we must understand the recursion. The sequence of calls made by fft_2<64> is illustrated in Fig. 1. The first line contains 64 points, with each point represented by a triangle.[3] The points on the top line require computing two 32-point FFTs, which are shown in the second line of the figure. Each of these requires two 16-point FFTs, shown in the third line, and so forth. Thus, the number of lines in the figure (the depth of the recursion) is $\log_2 N$.

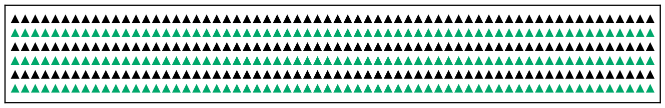

Fig. 1. Recursion for radix-2 for $N = 64$

In an optimized implementation, some attention would be paid to the base case. In particular, a two-point FFT is simply $\mathcal{F}\langle x_0, x_1\rangle = \langle x_0 + x_1,\ x_0 - x_1\rangle$, since inside the loop $k = 0$ and $\omega^0 = 1$. However, such details can be ignored in an asymptotic analysis; so in terms of real operations, we obtain

	$(N \log_2 N)$
ADD	2.0
MUL	1.0
FMA	1.0
flops	5.0
flaps	4.0

[1]We considered calling these *flinstrs* but restrained ourselves.

[2]Those who would harrumph at *flap* are advised first to contemplate their assent to *twiddle*.

[3]At least triangles are pointy. Since the point will be calculated by a calling a ½-length FFT, we'd have preferred digons, but sadly these are degenerate and thus ill-suited for figures.

B. Radix-4 FFT

Separating summation (1) into terms modulo four yields

$$X_k = \sum_{n=0}^{N/4-1} \omega_{N/4}^{kn} x_{4n} + \omega_N^k \sum_{n=0}^{N/4-1} \omega_{N/4}^{kn} x_{4n+1} + \omega_N^{2k} \sum_{n=0}^{N/4-1} \omega_{N/4}^{kn} x_{4n+2} + \omega_N^{3k} \sum_{n=0}^{N/4-1} \omega_{N/4}^{kn} x_{4n+3} \tag{4}$$

and the algorithm

Radix-4 FFT

```
input   : $x_0, x_1, \ldots, x_{N-1}$
output  : $X_0, X_1, \ldots, X_{N-1}$
requires: $N = 2^p$ for $p \geq 0$

function fft_4<$N$>
    if $N < 4$ then
        $\mathbf{X} \leftarrow$ fft_2<$N$>($\mathbf{x}$)
    else
        $\omega \leftarrow \exp(-2\pi i/N)$
        $\mathbf{u} \leftarrow$ fft_4<$N/4$>($x_0, x_4, \ldots, x_{N-4}$)
        $\mathbf{y} \leftarrow$ fft_4<$N/4$>($x_1, x_5, \ldots, x_{N-3}$)
        $\mathbf{v} \leftarrow$ fft_4<$N/4$>($x_2, x_6, \ldots, x_{N-2}$)
        $\mathbf{z} \leftarrow$ fft_4<$N/4$>($x_3, x_7, \ldots, x_{N-1}$)
        for $k \leftarrow 0$ to $N/4 - 1$ do
            $X_k \leftarrow (u_k + \omega^{2k} v_k) + (\omega^k y_k + \omega^{3k} z_k)$
            $X_{k+N/4} \leftarrow (u_k - \omega^{2k} v_k) - i(\omega^k y_k - \omega^{3k} z_k)$
            $X_{k+2N/4} \leftarrow (u_k + \omega^{2k} v_k) - (\omega^k y_k + \omega^{3k} z_k)$
            $X_{k+3N/4} \leftarrow (u_k - \omega^{2k} v_k) + i(\omega^k y_k - \omega^{3k} z_k)$
        end
    end
end
```

The **for** loop of function fft_4<N> iterates $N/4$ times, and after common subexpression elimination, contains three complex multiplications and eight complex additions. Note that multiplication by 1, -1, i, or $-i$ does not require any flops, but rather, at most, data movement followed by changing addition to subtraction or vice versa. This is quite nice—we're computing points four at a time and don't need multiplications to put the pieces together. Unfortunately, though, three of the four subproblem results do require complex multiplication by twiddle factors. (In the previous algorithm, it was only one out of two.)

The sequence of calls for fft_4<64> is illustrated in Fig. 2. The first line contains 64 points, with each point represented by a square.[4] The points on the top line require computing four 16-point FFTs, which are shown in the third line of the figure. The second line is empty since no 32-point FFTs are used. Likewise, no 8-point FFTs are computed. The fifth line shows the 4-point FFTs, which themselves invoke no further recursion.

[4]Squares, which have four sides, are used to indicate that the point will be calculated by a calling a ¼-length FFT.

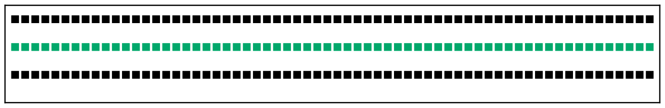
Fig. 2. Recursion for radix-4 for $N = 64$

The box in Fig. 2 above, which has depth $\log_2 N$, is one-half full, so we must apply this factor to our operation counts. (An alternative approach would have been to show there are $\log_4 N$ recursive calls and use the fact that $\log_4 N = \frac{1}{2}\log_2 N$.) Asymptotically the real operation counts are as follows:

	$(N \log_2 N)$
ADD	2.0
MUL	0.75
FMA	0.75
flops	4.25
flaps	3.5

Again, for an asymptotic analysis, we do not need to examine the details of the base case. However, an optimized implementation would benefit from adding a specialized base case for $N = 4$ that avoids all multiplications:

Four Point FFT

```
input   : $x_0, x_1, x_2, x_3$
output  : $X_0, X_1, X_2, X_3$

function fft_4<4>
    $X_0 \leftarrow (x_0 + x_2) + (x_1 + x_3)$
    $X_1 \leftarrow (x_0 - x_2) - i(x_1 - x_3)$
    $X_2 \leftarrow (x_0 + x_2) - (x_1 + x_3)$
    $X_3 \leftarrow (x_0 - x_2) + i(x_1 - x_3)$
end
```

C. Radix-8 FFT

Separating the original equation (1) into terms modulo eight yields

$$X_k = \sum_{n=0}^{N/8-1} \omega_{N/8}^{kn} x_{8n} + \omega_N^k \sum_{n=0}^{N/8-1} \omega_{N/8}^{kn} x_{8n+1} + \omega_N^{2k} \sum_{n=0}^{N/8-1} \omega_{N/8}^{kn} x_{8n+2} + \omega_N^{3k} \sum_{n=0}^{N/8-1} \omega_{N/8}^{kn} x_{8n+3} + \omega_N^{4k} \sum_{n=0}^{N/8-1} \omega_{N/8}^{kn} x_{8n+4} + \omega_N^{5k} \sum_{n=0}^{N/8-1} \omega_{N/8}^{kn} x_{8n+5} + \omega_N^{6k} \sum_{n=0}^{N/8-1} \omega_{N/8}^{kn} x_{8n+6} + \omega_N^{7k} \sum_{n=0}^{N/8-1} \omega_{N/8}^{kn} x_{8n+7} \tag{5}$$

The radix-8 recursive algorithm is apparent from the equation above, though it is worthwhile to write it out to see the common subexpressions.

A key observation is that the complex product $(1 \pm i)z$ can be rewritten as $z \pm iz$, which does not require multiplications. This is applicable to the radix-8 algorithm since

$$\begin{aligned}\omega_N^{k+N/8} &= \omega_N^{N/8}\omega_N^k \\ &= e^{-\pi i/4}\omega_N^k \\ &= \sqrt{1/2}\,(1-i)\,\omega_N^k.\end{aligned}$$

Therefore, combining the subproblem results will take fewer operations than a simpler reckoning would suggest. On the other hand, seven of the eight sums in (5) require complex multiplication by twiddle factors.

Finally, note that the pair of real expressions

$$\begin{aligned}&a + \sqrt{1/2}\,b \\ &a - \sqrt{1/2}\,b\end{aligned}$$

can be computed in two obvious ways. The first is to perform a single multiply, $\sqrt{1/2}\,b$, followed by one add and one subtract. This would cost 3 flops and 3 flaps. The second would be to use two FMAs, which would cost 4 flops but only 2 flaps. The FMA approach is generally preferred since, from a throughput perspective, it requires fewer instructions, and from a latency perspective, the instruction dependency graph is shallower. The two FMAs can execute in parallel. Nevertheless, we will consider both implementations in our analysis below.

The algorithm is shown in the column to the right. The **for** loop of function `fft_8<N>` iterates $N/8$ times, and inside the loop the computation of the t variables entails 7 complex twiddle factor multiplications and 16 complex additions. Then, the computation of $(t_5 \pm it_7)$ requires 2 complex additions. Finally, the complex points X are computed either with 4 real multiplications (since the constant $\sqrt{1/2}$ is real but $(t_5 \pm it_7)$ is complex) and 16 real additions, or preferably, with 8 real FMAs and 8 real additions.

The sequence of calls for `fft_8<64>` is illustrated in Fig. 3. The first line contains 64 points, with each point represented by an octagon.[8] The points on the top line require computing eight 8-point FFTs, which are shown in the fourth line. No other FFT subproblems are needed, so all other lines in the figure are empty. In general, the box, which has depth $\log_2 N$, is one-third full. Of course, we could also have derived this fraction using the relationship $\log_8 N = \frac{1}{3}\log_2 N$.

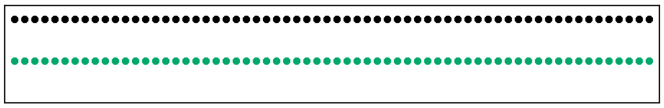

Fig. 3. Recursion for radix-8 for $N = 64$

The operation counts from examining the code inside the **for** loop need to be multiplied by $N/8$, the iteration count, and by $\frac{1}{3}\log_2 N$, the number of recursive function calls.

[8]Octagons indicate decomposition into subproblems having one-eighth as many points as the current problem.

Radix-8 FFT

```
input    : $x_0, x_1, \ldots, x_{N-1}$
output   : $X_0, X_1, \ldots, X_{N-1}$
requires : $N = 2^p$ for $p \geq 0$

function fft_8<N>
    if $N < 8$ then
        $\mathbf{X} \leftarrow$ fft_4<N>($\mathbf{x}$)
    else
        $\omega \leftarrow \exp(-2\pi i/N)$
        $\mathbf{u} \leftarrow$ fft_8<N/8>($x_0, x_8, \ldots, x_{N-8}$)
        $\mathbf{y} \leftarrow$ fft_8<N/8>($x_1, x_9, \ldots, x_{N-7}$)
        $\hat{\mathbf{u}} \leftarrow$ fft_8<N/8>($x_2, x_{10}, \ldots, x_{N-6}$)
        $\hat{\mathbf{y}} \leftarrow$ fft_8<N/8>($x_3, x_{11}, \ldots, x_{N-5}$)
        $\mathbf{v} \leftarrow$ fft_8<N/8>($x_4, x_{12}, \ldots, x_{N-4}$)
        $\mathbf{z} \leftarrow$ fft_8<N/8>($x_5, x_{13}, \ldots, x_{N-3}$)
        $\hat{\mathbf{v}} \leftarrow$ fft_8<N/8>($x_6, x_{14}, \ldots, x_{N-2}$)
        $\hat{\mathbf{z}} \leftarrow$ fft_8<N/8>($x_7, x_{15}, \ldots, x_{N-1}$)
        for $k \leftarrow 0$ to $N/8 - 1$ do
            $t_0 \leftarrow (\quad u_k + \omega^{4k}v_k) + (\omega^{2k}\hat{u}_k + \omega^{6k}\hat{v}_k)$
            $t_1 \leftarrow (\omega^k y_k + \omega^{5k}z_k) + (\omega^{3k}\hat{y}_k + \omega^{7k}\hat{z}_k)$
            $t_2 \leftarrow (\quad u_k + \omega^{4k}v_k) - (\omega^{2k}\hat{u}_k + \omega^{6k}\hat{v}_k)$
            $t_3 \leftarrow (\omega^k y_k + \omega^{5k}z_k) - (\omega^{3k}\hat{y}_k + \omega^{7k}\hat{z}_k)$
            $t_4 \leftarrow (\quad u_k - \omega^{4k}v_k) - i(\omega^{2k}\hat{u}_k - \omega^{6k}\hat{v}_k)$
            $t_5 \leftarrow (\omega^k y_k - \omega^{5k}z_k) - (\omega^{3k}\hat{y}_k - \omega^{7k}\hat{z}_k)$
            $t_6 \leftarrow (\quad u_k - \omega^{4k}v_k) + i(\omega^{2k}\hat{u}_k - \omega^{6k}\hat{v}_k)$
            $t_7 \leftarrow (\omega^k y_k - \omega^{5k}z_k) + (\omega^{3k}\hat{y}_k - \omega^{7k}\hat{z}_k)$

            $X_k \leftarrow t_0 + t_1$
            $X_{k+N/8} \leftarrow t_4 + \sqrt{1/2}(t_5 - it_7)$
            $X_{k+2N/8} \leftarrow t_2 - it_3$
            $X_{k+3N/8} \leftarrow t_6 - \sqrt{1/2}(t_5 + it_7)$
            $X_{k+4N/8} \leftarrow t_0 - t_1$
            $X_{k+5N/8} \leftarrow t_4 - \sqrt{1/2}(t_5 - it_7)$
            $X_{k+6N/8} \leftarrow t_2 + it_3$
            $X_{k+7N/8} \leftarrow t_6 + \sqrt{1/2}(t_5 + it_7)$
        end
    end
end
```

Asymptotically, we obtain either of the following real operation counts:

	$(N\log_2 N)$	$(N\log_2 N)$
ADD	$2.16\overline{6}$	$1.83\overline{3}$
MUL	0.750	$0.58\overline{3}$
FMA	$0.58\overline{3}$	$0.91\overline{6}$
flops	$4.08\overline{3}$	4.25
flaps	3.5	$3.3\overline{3}$

We see that, compared to radix-4, either the flop count or the flap count is reduced, but not both. Bergland [2] chose the former, but current hardware is more likely to benefit from the latter.

IV. Secondo

A. 2,4-Split-Radix FFT

The radix-4 FFT algorithm was shown to be superior to the radix-2, both in terms of flops and flaps. However, the radix-2 decomposition into even and odd terms had one advantage: it required that none of the even terms be multiplied by twiddle factors. This motivates the split radix algorithm [10, 5, 9], which combines a radix-2 and a radix-4 FFT, using the former for the even terms and the latter for the odd:

$$X_k = \sum_{n=0}^{N/2-1} \omega_{N/2}^{kn} x_{2n} + \omega_N^k \sum_{n=0}^{N/4-1} \omega_{N/4}^{kn} x_{4n+1} + \omega_N^{3k} \sum_{n=0}^{N/4-1} \omega_{N/4}^{kn} x_{4n+3} \tag{6}$$

In implementing (6) algorithmically, it is noteworthy that the first subproblem has twice the length of the other two. That poses no difficulty; FFTs are periodic, and results can simply be repeated as necessary. For example, consider the middle term in (6) when computing $X_{k+N/4}$. In front of the summation

$$\omega_N^{k+N/4} = \omega_N^k e^{-2\pi i/4} = -i\,\omega_N^k,$$

but inside the summation

$$\omega_{N/4}^{(k+N/4)n} = \omega_{N/4}^{kn} e^{-2\pi i n} = \omega_{N/4}^{kn}.$$

2,4-Split-Radix FFT

```
input  : $x_0, x_1, \ldots, x_{N-1}$
output : $X_0, X_1, \ldots, X_{N-1}$
requires: $N = 2^p$ for $p \geq 0$

function fft_2_4<$N$>
    if $N < 4$ then
        $\mathbf{X} \leftarrow$ fft_2<$N$>($\mathbf{x}$)
    else
        $\omega \leftarrow \exp(-2\pi i/N)$
        $\mathbf{u} \leftarrow$ fft_2_4<$N/2$>($x_0, x_2, \ldots, x_{N-2}$)
        $\mathbf{y} \leftarrow$ fft_2_4<$N/4$>($x_1, x_5, \ldots, x_{N-3}$)
        $\mathbf{z} \leftarrow$ fft_2_4<$N/4$>($x_3, x_7, \ldots, x_{N-1}$)
        for $k \leftarrow 0$ to $N/4 - 1$ do
            $X_k \leftarrow u_k + (\omega^k y_k + \omega^{3k} z_k)$
            $X_{k+N/4} \leftarrow u_{k+N/4} - i(\omega^k y_k - \omega^{3k} z_k)$
            $X_{k+2N/4} \leftarrow u_k - (\omega^k y_k + \omega^{3k} z_k)$
            $X_{k+3N/4} \leftarrow u_{k+N/4} + i(\omega^k y_k - \omega^{3k} z_k)$
        end
    end
end
```

The **for** loop above iterates $N/4$ times and contains two complex multiplications and six complex additions. Regarding the recursion, the first level for a 64-point FFT is shown in the top of Fig. 4. The topmost line has 64 points, and its even and odd elements are indicated by triangles and squares, respectively. The even points will be computed by a 32-point FFT, which is shown in the second row. The odd points will be computed by two 16-point FFTs, shown in the third row. The 32-point and 16-point subproblems will be recursively computed by `fft_2_4` until the base case is reached. This is shown in the bottom of Fig. 4.

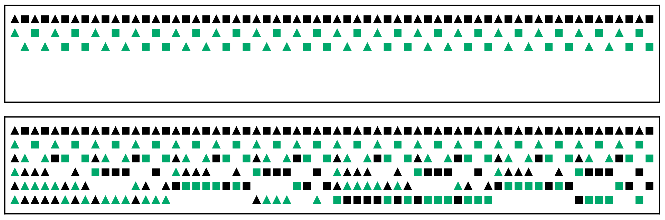

Fig. 4. Recursion for 2,4-split-radix after first step (top) and after five steps (bottom) for $N = 64$

In general, the number of points in a row is given by the sum of the number of even points in the row immediately above it and the number of odd points in the row two levels above it. Letting g_r represent the fraction of points in row r, the sequence $\langle g_r \rangle$ is

$$\left\langle 1, \frac{1}{2}, \frac{3}{4}, \frac{5}{8}, \frac{11}{16}, \frac{21}{32}, \frac{43}{64}, \frac{85}{128}, \frac{171}{256}, \frac{341}{512}, \frac{683}{1024}, \frac{1365}{2048}, \frac{2731}{4096}, \ldots \right\rangle.$$

This is described by the recurrence

$$g_r = \frac{g_{r-1}}{2} + \frac{g_{r-2}}{2} + [r = 0] \tag{7}$$

where

$$[r = 0] \overset{\text{def}}{=} \begin{cases} 1 & \text{if } r = 0 \\ 0 & \text{otherwise} \end{cases}$$

and has the closed form solution

$$g_r = \frac{2}{3} + \frac{1}{3}\left(\frac{-1}{2}\right)^r. \tag{8}$$

As Knuth [7] observed, “once such an equation has been found, it is a simple matter to prove it by induction, and we need not even mention that we used generating functions to discover it.” So we shan't. What is important is that

$$\lim_{r \to \infty} g_r = 2/3. \tag{9}$$

Therefore, asymptotically, we obtain

	$(N \log_2 N)$
ADD	2.0
MUL	$0.6\overline{6}$
FMA	$0.6\overline{6}$
flops	4.0
flaps	$3.3\overline{3}$

as real operation counts for the 2,4-split-radix. This algorithm is seen to require fewer flops than any of the previous but the same number of flaps as the radix-8 FFT.

B. 2,8-Split-Radix FFT

The radix-8 algorithm was better able to utilize a hardware FMA since $\omega_8 = \sqrt{1/2}(1-i)$, and the constant $\sqrt{1/2}$ could be amalgamated into existing additions or subtractions. So, in order to take advantage of this optimization while also reducing the number of complex twiddle multiplications, let's consider the following decomposition:

$$\begin{aligned}X_k = &\sum_{n=0}^{N/2-1} \omega_{N/2}^{kn} x_{2n} \\ &+ \omega_N^k \sum_{n=0}^{N/8-1} \omega_{N/8}^{kn} x_{8n+1} + \omega_N^{3k} \sum_{n=0}^{N/8-1} \omega_{N/8}^{kn} x_{8n+3} \\ &+ \omega_N^{5k} \sum_{n=0}^{N/8-1} \omega_{N/8}^{kn} x_{8n+5} + \omega_N^{7k} \sum_{n=0}^{N/8-1} \omega_{N/8}^{kn} x_{8n+7}\end{aligned} \tag{10}$$

Above, we have used radix-2 for the even terms, as in the 2,4-split-radix, but we are now using radix-8 for the odd terms. We again have a choice as to how to implement $a \pm \sqrt{1/2}\,b$ and provide both analyses. Choosing the FMA implementation results in this algorithm requiring fewer flaps than the others.

The algorithm is as follows:

2,8-Split-Radix FFT

```
input   : $x_0, x_1, \ldots, x_{N-1}$
output  : $X_0, X_1, \ldots, X_{N-1}$
requires: $N = 2^p$ for $p \geq 0$

function fft_2_8<$N$>
    if $N < 8$ then
        $\mathbf{X} \leftarrow$ fft_4<$N$>($\mathbf{x}$)
    else
        $\omega \leftarrow \exp(-2\pi i/N)$
        $\mathbf{u} \leftarrow$ fft_2_8<$N/2$>($x_0, x_2, \ldots, x_{N-2}$)
        $\mathbf{y} \leftarrow$ fft_2_8<$N/8$>($x_1, x_9, \ldots, x_{N-7}$)
        $\hat{\mathbf{y}} \leftarrow$ fft_2_8<$N/8$>($x_3, x_{11}, \ldots, x_{N-5}$)
        $\mathbf{z} \leftarrow$ fft_2_8<$N/8$>($x_5, x_{13}, \ldots, x_{N-3}$)
        $\hat{\mathbf{z}} \leftarrow$ fft_2_8<$N/8$>($x_7, x_{15}, \ldots, x_{N-1}$)
        for $k \leftarrow 0$ to $N/8-1$ do
            $t_1 \leftarrow (\omega^k y_k + \omega^{5k} z_k) + (\omega^{3k}\hat{y}_k + \omega^{7k}\hat{z}_k)$
            $t_3 \leftarrow (\omega^k y_k + \omega^{5k} z_k) - (\omega^{3k}\hat{y}_k + \omega^{7k}\hat{z}_k)$
            $t_5 \leftarrow (\omega^k y_k - \omega^{5k} z_k) - (\omega^{3k}\hat{y}_k - \omega^{7k}\hat{z}_k)$
            $t_7 \leftarrow (\omega^k y_k - \omega^{5k} z_k) + (\omega^{3k}\hat{y}_k - \omega^{7k}\hat{z}_k)$

            $X_k \leftarrow u_k + t_1$
            $X_{k+N/8} \leftarrow u_{k+N/8} + \sqrt{1/2}(t_5 - it_7)$
            $X_{k+2N/8} \leftarrow u_{k+2N/8} - it_3$
            $X_{k+3N/8} \leftarrow u_{k+3N/8} - \sqrt{1/2}(t_5 + it_7)$
            $X_{k+4N/8} \leftarrow u_k - t_1$
            $X_{k+5N/8} \leftarrow u_{k+N/8} - \sqrt{1/2}(t_5 - it_7)$
            $X_{k+6N/8} \leftarrow u_{k+2N/8} + it_3$
            $X_{k+7N/8} \leftarrow u_{k+3N/8} + \sqrt{1/2}(t_5 + it_7)$
        end
    end
end
```

The **for** loop in the preferred flap-optimized implementation has 4 complex multiplications, 14 complex additions, and 8 real FMAs. If we had instead computed $\sqrt{1/2}(t_5 \pm it_7)$ as two common subexpressions, we would have needed 4 more real multiplications, but all of the 8 final FMAs would have become additions. This would have added 4 flaps (the multiplications) but overall would have saved 4 flops.

The first level for a 64-point FFT is shown in the top of Fig. 5. The topmost line has 64 points, and its even and odd elements are indicated by triangles and octagons, respectively. The even points will be computed by a 32-point FFT, which is shown in the second row. The odd points will be computed by four 8-point FFTs, shown in the fourth row. The subproblems will be recursively computed by `fft_2_8` until the base case is reached. This is shown in the bottom of Fig. 5.

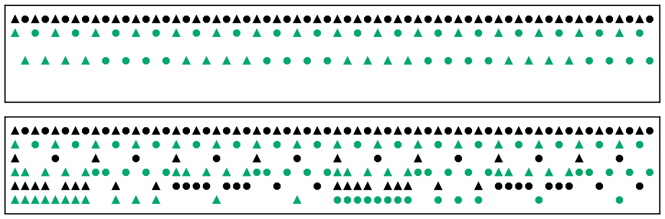

Fig. 5. Recursion for 2,8-split-radix after first step (top) and after five steps (bottom) for $N = 64$

Letting g_r represent the fraction of points in row r above, the sequence $\langle g_r \rangle$ is

$$\left\langle 1, \frac{1}{2}, \frac{1}{4}, \frac{5}{8}, \frac{9}{16}, \frac{13}{32}, \frac{33}{64}, \frac{69}{128}, \frac{121}{256}, \frac{253}{512}, \frac{529}{1024}, \frac{1013}{2048}, \frac{2025}{4096}, \ldots \right\rangle,$$

which is described by

$$g_r = \frac{g_{r-1}}{2} + \frac{g_{r-3}}{2} + [r = 0] \tag{11}$$

and has solution

$$\begin{aligned}g_r = \frac{1}{2} &+ \left(\frac{1}{4} - \frac{\sqrt{7}}{28}i\right)\left(\frac{-1}{4} + \frac{\sqrt{7}}{4}i\right)^r \\ &+ \left(\frac{1}{4} + \frac{\sqrt{7}}{28}i\right)\left(\frac{-1}{4} - \frac{\sqrt{7}}{4}i\right)^r.\end{aligned} \tag{12}$$

Note that

$$\lim_{r\to\infty} g_r = 1/2, \tag{13}$$

so asymptotically, we obtain either of the following operation counts:

	$(N\log_2 N)$	$(N\log_2 N)$
ADD	2.25	1.75
MUL	0.75	0.5
FMA	0.5	1.0
flops	4.0	4.25
flaps	3.5	3.25

V. Contorno

It is useful for an FFT to support multiplication by a real-valued scaling factor, α, by modifying (1) as follows:

$$X_k = \alpha \sum_{n=0}^{N-1} \omega_N^{kn} x_n \tag{14}$$

or

$$X_k = \sum_{n=0}^{N-1} \omega_N^{kn} (\alpha x_n) \tag{15}$$

We have implemented something closer to (15) for two reasons. First, users of an FFT library may wish to scale by $0 < \alpha < 1$ to avoid floating point overflow, so it is better for that purpose that we scale in the leaf nodes of the recursion tree (which are computed first) rather than in the final stages of an algorithm. Secondly, the base case of the recursion does not require twiddle factors. This gives us a convenient and flap-efficient place to introduce scaling.

The three base cases needed for the algorithms discussed in this paper are shown at the right. Note that each requires only two more flaps (one real $\times$ complex multiplication) than its unscaled variant. The remainder of the points are scaled by changing additions or subtractions into FMAs. For the eight point base case, another multiplication calculates $\alpha\sqrt{1/2}$, but this is done once for the entire FFT, not inside any recursively invoked function. In practice, both α and $\alpha\sqrt{1/2}$ are passed as function arguments to `scale_fft_8`, so scaling the transform requires only $N/4 + 1$ extra flaps.

Given the efficiency of using FMAs for most of the scaling, FFT libraries should provide an interface that both scales and transforms in a single function call. Of course, the biggest benefit to users (in addition to convenience) is not having to incur the load and store expense of looping over the input data merely in order to scale it. Furthermore, we suggest that the scaling factor α should be provided at run-time, at each transform function's call site, not beforehand. There is nothing to be gained by precomputing $\alpha\sqrt{1/2}$ and suchlike in advance.

Note that the scaling operations themselves are often directly supported in hardware. For example, $y \pm \alpha i z$ for $\alpha \in \mathbb{R}$ and $y, z \in \mathbb{C}$ can be computed with Arm's Scalable Vector Extension [1] by first replicating α into every element of a SIMD vector register and then using `fcmla #90` or `fcmla #270`. Also, the important special case where $\alpha = 1$ is supported using either `fcadd #90` or `fcadd #270`.

VI. Digestivo

Having an asymptotic complexity of $3.25\,N\log_2 N$ flaps, the 2,8-split-radix FFT achieves an advantage over the other algorithms discussed in this paper. In light of this result, we use the 2,8-split-radix FFT in our High Performance Kernels for FFT library [3].

For more information, or to download our library, please visit our website: `https://hpkfft.com`

Scaled Two Point FFT

```
input   : $\alpha$, $x_0, x_1$
output  : $X_0, X_1$
requires: $\alpha \in \mathbb{R}$

function scale_fft_2
    $\hat{x}_1 \leftarrow \alpha x_1$
    $X_0 \leftarrow \alpha x_0 + \hat{x}_1$
    $X_1 \leftarrow \alpha x_0 - \hat{x}_1$
end
```

Scaled Four Point FFT

```
input   : $\alpha$, $x_0, x_1, x_2, x_3$
output  : $X_0, X_1, X_2, X_3$
requires: $\alpha \in \mathbb{R}$

function scale_fft_4
    $\hat{x}_2 \leftarrow \alpha x_2$
    $X_0 \leftarrow (\alpha x_0 + \hat{x}_2) + \alpha(x_1 + x_3)$
    $X_1 \leftarrow (\alpha x_0 - \hat{x}_2) - \alpha i(x_1 - x_3)$
    $X_2 \leftarrow (\alpha x_0 + \hat{x}_2) - \alpha(x_1 + x_3)$
    $X_3 \leftarrow (\alpha x_0 - \hat{x}_2) + \alpha i(x_1 - x_3)$
end
```

Scaled Eight Point FFT

```
input   : $\alpha$, $x_0, x_1, \ldots, x_7$
output  : $X_0, X_1, \ldots, X_7$
requires: $\alpha \in \mathbb{R}$

function scale_fft_8
    $\hat{x}_4 \leftarrow \alpha x_4$
    $t_0 \leftarrow (\alpha x_0 + \hat{x}_4) + \alpha(x_2 + x_6)$
    $t_1 \leftarrow (x_1 + x_5) + (x_3 + x_7)$
    $t_2 \leftarrow (\alpha x_0 + \hat{x}_4) - \alpha(x_2 + x_6)$
    $t_3 \leftarrow (x_1 + x_5) - (x_3 + x_7)$
    $t_4 \leftarrow (\alpha x_0 - \hat{x}_4) - \alpha i(x_2 - x_6)$
    $t_5 \leftarrow (x_1 - x_5) - (x_3 - x_7)$
    $t_6 \leftarrow (\alpha x_0 - \hat{x}_4) + \alpha i(x_2 - x_6)$
    $t_7 \leftarrow (x_1 - x_5) + (x_3 - x_7)$

    $X_0 \leftarrow t_0 + \alpha t_1$
    $X_1 \leftarrow t_4 + \alpha\sqrt{1/2}(t_5 - i t_7)$
    $X_2 \leftarrow t_2 - \alpha i t_3$
    $X_3 \leftarrow t_6 - \alpha\sqrt{1/2}(t_5 + i t_7)$
    $X_4 \leftarrow t_0 - \alpha t_1$
    $X_5 \leftarrow t_4 - \alpha\sqrt{1/2}(t_5 - i t_7)$
    $X_6 \leftarrow t_2 + \alpha i t_3$
    $X_7 \leftarrow t_6 + \alpha\sqrt{1/2}(t_5 + i t_7)$
end
```

References

[1] Arm. 2023. *Arm A-profile A64 Instruction Set Architecture*. Arm Limited. `https://developer.arm.com/documentation/ddi0602/2023-12`

[2] Glenn Bergland. 1968. A Fast Fourier Transform Algorithm Using Base 8 Iterations. *Math. Comp.* 22 (April 1968), 275 – 279. `https://doi.org/10.1090/S0025-5718-1968-0226899-X`

[3] Paul Caprioli and Robby Jenkins. 2023. High Performance Kernels for FFT via Modern C++. `https://doi.org/10.5281/zenodo.8253863`

[4] James W. Cooley and John W. Tukey. 1965. An Algorithm for the Machine Calculation of Complex Fourier Series. *Math. Comp.* 19, 90 (1965), 297–301.

[5] Pierre Duhamel and Henk Hollmann. 1984. Split Radix FFT Algorithm. *Electronics Letters* 20 (February 1984), 14 – 16. `https://doi.org/10.1049/el:19840012`

[6] Intel. 2011. *Intel 64 and IA-32 Architectures Optimization Reference Manual, Order Number: 248966-023*. Intel Corporation. `https://www.intel.com/content/www/us/en/content-details/814198`

[7] Donald E. Knuth. 1997. *The Art of Computer Programming, Volume 1 (3rd ed.): Fundamental Algorithms*. Addison Wesley Longman, Massachusetts, USA.

[8] IEEE Std 754-2019 (Revision of IEEE 754-2008). 2019. *IEEE Standard for Floating-Point Arithmetic*. Institute of Electrical and Electronics Engineers, New York, NY, USA. `https://doi.org/10.1109/IEEESTD.2019.8766229`

[9] Henrik Sorensen, Michael Heideman, and C. Sidney Burrus. 1986. On Computing the Split-Radix FFT. *IEEE Transactions on Acoustics, Speech, and Signal Processing* 34, 1 (1986), 152–156. `https://doi.org/10.1109/TASSP.1986.1164804`

[10] R. Yavne. 1968. An Economical Method for Calculating the Discrete Fourier Transform. In *Proceedings of the December 9-11, 1968, Fall Joint Computer Conference, Part I* (San Francisco, California) *(AFIPS '68)*. Association for Computing Machinery, New York, NY, USA, 115–125. `https://doi.org/10.1145/1476589.1476610`