The Fast Fourier Transform

1. The conquer and divide step when $2K = N$ described by

$$\sum_{l=0}^{N-1} e^{-i2\pi kl/N} x_l = \sum_{p=0}^{K-1} e^{-i2\pi kp/K} x_{2p} + e^{-2\pi k/N} \sum_{p=0}^{K-1} e^{-i2\pi kp/K} x_{2p+1}$$

   splits the terms of the sum for the discrete Fourier transform into add and terms. Construct a similar equation for use when $N = 3^n$ that divides the sum into three parts such that l divided by 3 has remainder 0, 1 or 2.

Suppose $3K = N$ then a similar equation that divides the discrete Fourier transform into three parts is

$$\begin{aligned}
&\sum_{l=0}^{N-1} e^{-i2\pi kl/N} x_l = \Big(\sum_{l\equiv 0 \,(\mathrm{mod}\, 3)} + \sum_{l\equiv 1 \,(\mathrm{mod}\, 3)} + \sum_{l\equiv 2 \,(\mathrm{mod}\, 3)} \Big) e^{-i2\pi kl/N} x_l \\
&= \sum_{p=0}^{K-1} e^{-i2\pi (3p)l/N} x_{3p} + \sum_{p=0}^{K-1} e^{-i2\pi (3p+1)l/N} x_{3p+1} + \sum_{p=0}^{K-1} e^{-i2\pi (3p+2)l/N} x_{3p+2} \\
&= \sum_{p=0}^{K-1} e^{-i2\pi pl/K} x_{3p} + e^{-i2\pi l/N} \sum_{p=0}^{K-1} e^{-i2\pi pl/K} x_{3p+1} + e^{-i4\pi l/N} \sum_{p=0}^{K-1} e^{-i2\pi pl/K} x_{3p+2}.
\end{aligned}$$

Note that the last three sums are exactly Fourier transforms of size $N/3 = K$ for the vectors with components x_{3p}, x_{3p+1} and x_{3p+2} respectively.

2. Let $z = a + bi$ and $w = u + iv$ be complex numbers. It takes four real-values multiplications when using the foil method to find the product zw. Look up fast complex multiplication, describe it and explain how many real-valued multiplications the fast algorithm uses for find zw.

For reference, note that by the foil method we have

$$zw = (a + bi)(u + iv) = au - bv + i(av + bu).$$

Now, set $M = a + b$ and $N = u + v$ and form the three products

$$\eta_1 = au, \qquad \eta_2 = MN \qquad \text{and} \qquad \eta_3 = bv.$$

Since

$$\eta_2 = (a + b)(u + v) = au + av + bu + bv = \eta_1 + av + bu + \eta_3,$$

it follows that

$$zw = \eta_1 - \eta_3 + i(\eta_2 - \eta_1 - \eta_3).$$

Therefore, fact complex multiplication finds the product zw using only three multiplications. It should be noted, however, that the number of additions and subtractions have increased from two to five.

Suppose T_* is the computational effort to multiply two real floating-point numbers and $T_\pm$ is the effort to perform either addition or subtraction. For fast complex multiplication to be faster than the foil method we must have

$$3T_* + 5T_\pm \leq 4T_* + 2T_\pm \qquad \text{or equivalently} \qquad 3T_\pm \leq T_*.$$

Therefore, if on any particular CPU architecture it happens that floating point multiplication is less than three times as long as an addition and subtraction, then the foil method would actually be faster.

3. Compute the number of real-valued double-precision floating point multiplications, additions and subtractions achieved per second for test runs of the fast Fourier transform detailed in the following table:

Processor	Cores	N	Seconds
AMD A6-9225	1	1048576	0.72
AMD A6-9225	2	1048576	0.48
Intel Xeon E5-2520	1	1048576	0.82
Intel Xeon E5-2520	12	1048576	0.21

Explain your reasoning and how you counted the total number of operations. How many evaluations of the exponential function are performed?

Upon examining the FORTRAN code, we find that all the computation for the fast Fourier transform is performed by the routine `fftwork`. The conquer and divide algorithm calls this routine recursively for smaller and smaller transforms. Each time the loop

```
do k=0,N2-1
    even=b(p+k)
    odd=b(p+k+N2)
    w=exp(cmplx(0D0,-2*M_PI*k/N,kind(w)))
    b(p+k)=even+w*odd
    b(p+k+N2)=even-w*odd
end do
```

executes where `N2` $= N/2$ and N is the size of the transform size. Depending on how the compiler optimizes things a different number of additions, subtractions, multiplications and even divisions may be performed each time through the loop. For example, the subexpression `w*odd` appears twice in the loop so an optimizing compiler would generate code that only computes it once. A choice can further be made whether to use fast complex multiplication or the foil method to compute this product. Also note that the expression `-2*M_PI/N` is constant throughout the loop, so it need be computed only once. However, dividing by N first and then multiplying by k can make a subtle change in the results because of different rounding errors, so many compilers will not perform this transformation.

In order to determine exactly what the generated code will do we will inspect the assembly language output from the FORTRAN compiler. This can be obtained using the command

```
$ gfortran -S -O3 -ffast-math fast.f90
```

Here the flag `-S` indicates to stop after creating the assembler code rather than making an executable problem. The flag `-ffast-math` tells the compiler to perform additional floating-point optimizations that are mathematically correct but might change the rounding behavior of the resulting program in subtle ways.

Upon inspecting the resulting file `fast.s` the assembler code generated for the loop given above is seen to be

```
.L14:
    pxor    %xmm1, %xmm1
    cvtsi2sd    %ebp, %xmm1
    mulsd   .LC4(%rip), %xmm1
    mulsd   16(%rsp), %xmm1
    movapd  %xmm1, %xmm0
    movsd   %xmm1, 8(%rsp)
    call    cos@PLT
    movsd   8(%rsp), %xmm1
    xorpd   .LC5(%rip), %xmm1
    movsd   %xmm0, (%rsp)
    movapd  %xmm1, %xmm0
    call    sin@PLT
    movsd   (%rsp), %xmm2
.L8:
    movsd   8(%rbx), %xmm1
    movsd   (%rbx), %xmm3
    addl    $1, %ebp
    movsd   0(%r13), %xmm4
    movsd   8(%r13), %xmm5
    movapd  %xmm1, %xmm7
    movapd  %xmm3, %xmm6
    mulsd   %xmm0, %xmm7
    mulsd   %xmm2, %xmm1
    mulsd   %xmm3, %xmm0
    mulsd   %xmm2, %xmm6
    addsd   %xmm0, %xmm1
    movapd  %xmm4, %xmm0
    addsd   %xmm7, %xmm4
    addsd   %xmm6, %xmm0
    subsd   %xmm6, %xmm4
    subsd   %xmm7, %xmm0
    movsd   %xmm0, 0(%r13)
    movapd  %xmm5, %xmm0
    subsd   %xmm1, %xmm5
    addsd   %xmm1, %xmm0
    movsd   %xmm0, 8(%r13)
    addq    %r14, %r13
    movsd   %xmm4, (%rbx)
    movsd   %xmm5, 8(%rbx)
    addq    %r14, %rbx
    cmpl    %ebp, %r12d
    jne .L14
```

It is now a simple matter to verify that multiplication instruction `mulsd` appears exactly six times in the interior of the loop. Counting the `addsd` and `subsd` instructions reveals that four floating-point additions and three subtractions are performed each time through the loop. Surprisingly, this is one more than the six needed to perform one each of complex addition, subtraction and multiplication.

Note that $1/N$ has been precomputed using the floating-point division instruction `divsb` (not shown) before the loop and stored in `16(%rsp)`. This converts the division appearing in the expression `-2*M_PI*k/N` to multiplication by $1/N$. While faster than division, this multiplication is still performed each time through the loop rather than being combined as part of the constant `-2*M_PI/N` previously identified. Further examination of the assembler reveals the instruction sequence

```
mulsd   %xmm0, %xmm7
mulsd   %xmm2, %xmm1
mulsd   %xmm3, %xmm0
mulsd   %xmm2, %xmm6
```

which suggests that the foil method was used to compute `w*odd` rather than fast complex multiplication. Without further experimentation, it is not clear whether the fast algorithm is actually slower or whether the compiler simply missed an optimization opportunity to avoid changing the rounding behavior of the final result. One can also see that the complex exponential has been converted into separate calls to `sin` and `cos` by the formula

$$e^{i\theta} = \cos(\theta) + i\sin(\theta).$$

We now count the total number of additions and multiplications which will be used. Since $1048576 = 2^{20}$, then the `fftwork` will recurse 20 times before reaching the identity transform. Upon returning from each recursive call, the above loop will be performed at each level. The instructions inside the loop are therefore executed

$$N/2 + 2(N/2^2) + 2^2(N/2^3) + \cdots + 2^{19}(N/2^{20}) = 20(N/2) = (N/2)\log_2 N$$

times. This results in

$$\begin{aligned}
N_* = \text{number of multiply} &= 6(N/2)\log_2 N = 62914560\\
N_\pm = \text{add and subtract} &= 7(N/2)\log_2 N = 73400320\\
N_{\text{FLOP}} = \text{total arithmetic} &= 13(N/2)\log_2 N = 136314880\\
N_{\text{trans}} = \text{transcendental} &= 2(N/2)\log_2 N = 20971520.
\end{aligned}$$

Dividing by the timings for the different computers detailed in the question yields

Processor	Cores	Seconds	N_*/sec	$N_\pm$/sec	N_{FLOP}/sec
AMD A6-9225	1	0.72	87×10^6	102×10^6	189×10^6
AMD A6-9225	2	0.48	131×10^6	153×10^6	284×10^6
Intel Xeon E5-2520	1	0.82	77×10^6	90×10^6	166×10^6
Intel Xeon E5-2520	12	0.21	300×10^6	350×10^6	649×10^6

4. Download the code `fasttime.f90` for determining the speed of the fast Fourier transform from our website. Compile and run it on your computer. Compare the speed of this code to the one developed in class.

I ran the `fasttime.f90` code on a AMD Ryzen 7 desktop. The results were

```
$ gfortran -O3 -ffast-math -fopenmp fasttime.f90
$ ./a.out
N=1048576
B(0)=(14.440159752937522 14.440159752937522)
fft took 0.28700000047683716 seconds.
B(0)=(14.440159752937522 14.440159752937522)
parallel fft took 0.18600000441074371 seconds.
```

For comparison the code `fft.f90` from March 7 with slight modifications yielded the following results

```
$ gfortran -O3 -ffast-math fft.f90
$ ./a.out
 Elapsed time   0.270000011      sec
$ gfortran -O3 -ffast-math -fopenmp fft.f90
$ ./a.out
 Elapsed time   0.143999994      sec
```

In summary we have

Program	Serial	Parallel
`fasttime.f90`	0.287	0.186
`fft.f90`	0.270	0.144

which suggests the program written in class was slightly faster.