Evan Lang
Yang Lu

FFT Multithreading on CPU & GPU

1. Description of FFT's

The Fast Fourier Transform is an algorithm that computes the discrete fourier transform of a signal of complex numbers. An FFT relies on the same basic equation of a DFT:

$$X_k = \sum_{n=0}^{N-1} x_n e^{-i2\pi kn/N} \qquad k = 0, \ldots, N-1,$$

N represents the total number of elements in the signal

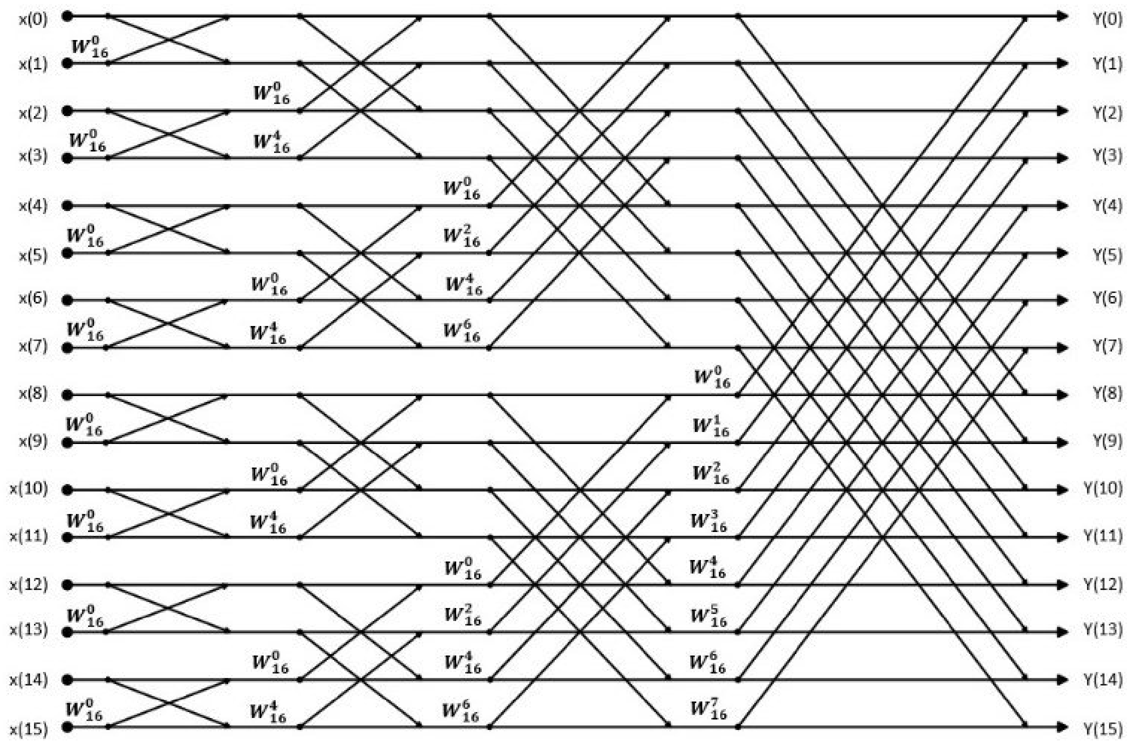
Utilizing this equation, a signal can be broken down into its component signals, enabling users to observe each component free of the interference of the rest. The primary difference between a DFT and an FFT lies in the difference in their time complexity. A classic DFT algorithm possesses a time complexity of $O(N^2)$, making it an extremely inefficient algorithm. FFT's meanwhile possess a time complexity of $O(N*\log_2(N))$, offering extreme savings in computational time compared to the DFT.

2. Our Serial FFT: The Cooley-Tukey algorithm

Invented in the 1960's by mathematician J. Tukey to speed up the fourier transforms used to detect secret nuclear tests hidden in seismographs taken by the United States government , the Cooley-Tukey algorithm was the first FFT of its time. Using a divide and conquer approach, Tukey was able to separate the Fourier transform into N/2 calculations that would run $\log_2(N)$ times. This is done by splitting the required calculations into even and odd sets of frequencies. Continually splitting these sets again and again until we are handling each element individually we are left with N/2 calculations (now referred to as the base calculations), owing to the symmetry of the signals.

For our serial code, we now need to run the base calculations for each time we split the set, resulting in the $\log_2(N)$ component of the time complexity. Each time we run the base calculations (with the current amount of runs now being represented by **S**) each element up to N/2 must interact with the next $2^{(S)}/2$ element. For example; on the first run S is equal to 1. This means the 0th element must now interact with the 1st element. On the next run of the base calculations, S is now equal to 2. This means the 0th element must now interact with the 2nd element. Next when S equals 3, 0 interacts with the 4th element and so on.

This behavior results in the striking visual depiction of each element's interactions as S increases to $\log_2(N)$ shown below:

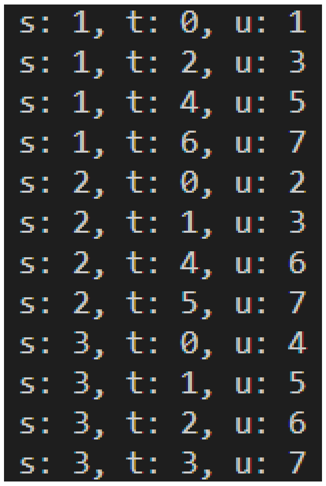


Named for its striking similarity to butterflies when $S > 1$. The butterfly operation encompasses the foundation of the Cooley-Tukey FFT. Finally, we must account for the periodicity of the signal by using a twiddle factor in each computation. This twiddle factor is $e^{-2\pi ji/N}$. Where j is the imaginary number $\sqrt{i}$, i is the index to be evaluated at ($0 \leq i \leq (N/2) - 1$), and N is the total number of elements.

3. **Our Serial Code**:

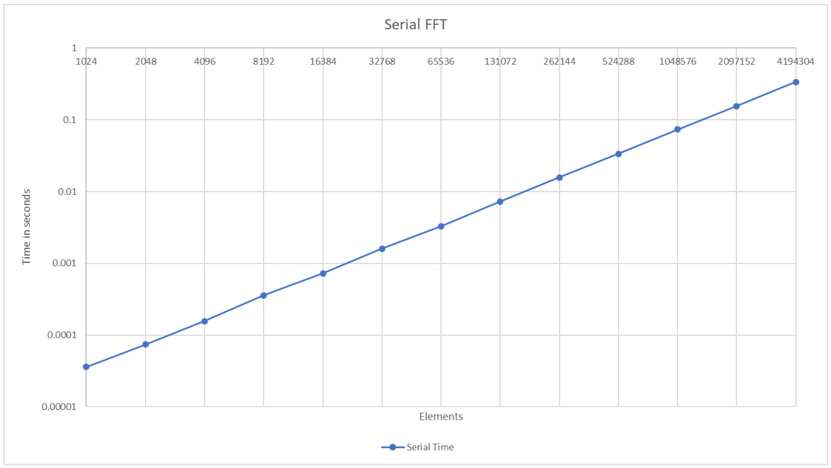
```
int logn = log2(input.n);
  //The primary S loop, which will run logn times
  for (int s = 1; s <= logn; s++) {
   int m = 1 << s;
   // The twiddle factor
   cplx w_m = cexp(-2.0 * PI * I / m);
//The Base Calculation Loops, which will run n/2 times, with the internal loop running (2^S)/2 times and the
external loop running n/(2^S)  times
         //These loops calculate the butterfly operation indexes
   for (int k = 0; k < input.n; k += m) {
           // The initial twiddle factor
           cplx w = 1.0;
            for (int j = 0; j < m / 2; j++) {
                // The indices of the elements
                int t = k + j;
                int u = t + m / 2;
                // The butterfly operation
                cplx temp = w * input.x[u];
                input.x[u] = input.x[t] - temp;
                input.x[t] = input.x[t] + temp;
                // Update the twiddle factor
                w = w * w_m;
           }
   }
}
```

In our serial code above, inspired by GeeksforGeeks by rohit_is (April, 2023), the current index is determined by both a combination of the k and j loops. This will run base calculations for $2^S/2$ indexes (represented by the j loop), before skipping the next half and running again for $2^S/2$ indexes (represented by the k loop, which increments by 2^S everytime the j loop completes). The combination of the j and k values determine our base index (represented by t), which is then matched with an index that is 2^S greater (represented by u). Everytime the j loop increments it will perform a butterfly operation and update the twiddle factor, and everytime the j loop completes the twiddle will be reset to 1. The twiddle factor is used in each butterfly operation to perform the necessary transform on each element, represented by the code in bold. This twiddle factor is $(e^{(2*pi*-i)/(2^S)})^{t\%(2^S)/2}$. In this case % means remainder. The order of operations for an 8 element signal is shown to the right.

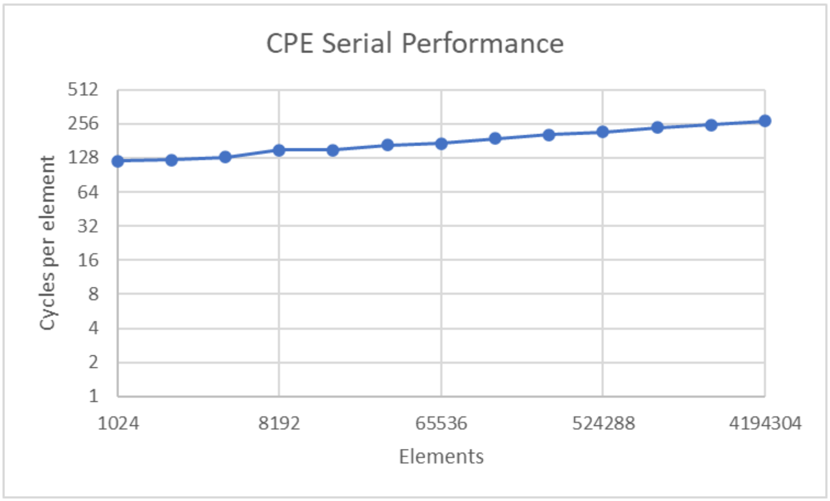


```
s: 1, t: 0, u: 1
s: 1, t: 2, u: 3
s: 1, t: 4, u: 5
s: 1, t: 6, u: 7
s: 2, t: 0, u: 2
s: 2, t: 1, u: 3
s: 2, t: 4, u: 6
s: 2, t: 5, u: 7
s: 3, t: 0, u: 4
s: 3, t: 1, u: 5
s: 3, t: 2, u: 6
s: 3, t: 3, u: 7
```

Finally the performance of the serial FFT is shown below:



The serial code maintains the time complexity of O(NlogN) displayed by the linear behavior when both axes are in log scale.



However we do see some slight CPE degradation as the number of elements increases. This could be due to the increasing number of cache misses present as m/2 spacing between indexed elements outruns our cache capacity.

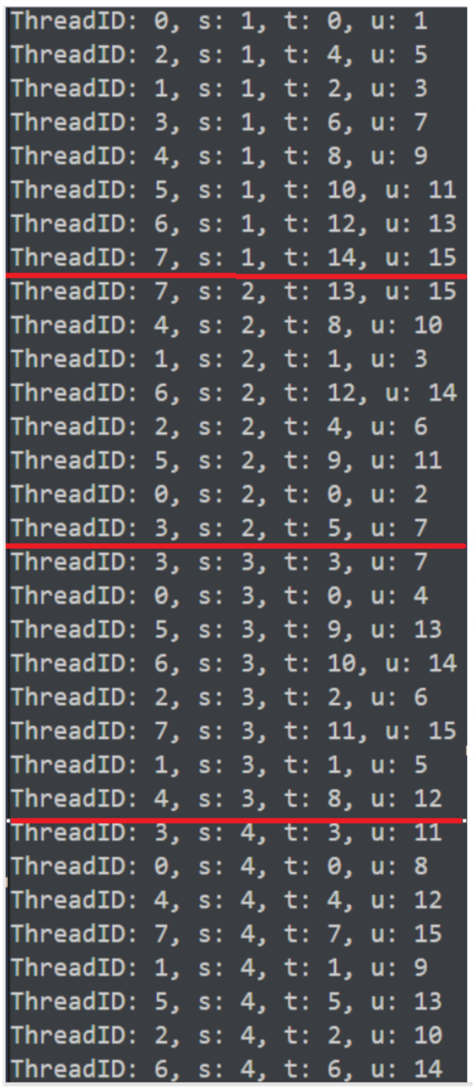
The output of our serial FFT's were validated against the matlab implementation of FFT to ensure correctness.

4. Multithreading with Pthreads

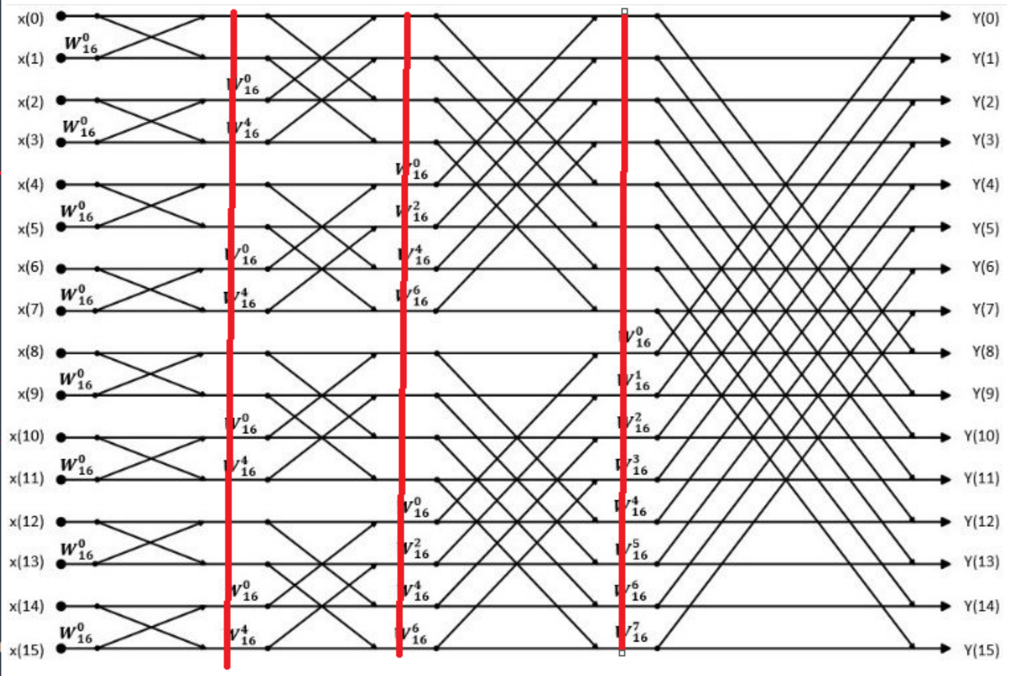
The first strategy used to optimize the code was CPU multithreading per odd-even index calculation using pthreads. This methodology proved to be extremely inefficient, as each thread was performing very little work. Consequently, an enormous number of threads were created. The original thought process was that assigning a thread to each calculation would speed up the process. However, this did not happen, as creating the threads turned out to be much more expensive than the calculations performed by each thread. One way to optimize this approach could have been to create threads first and then reuse them instead of destroying and creating them every time a calculation was performed. However, this strategy was much worse than the serial code, and a new approach was needed.

This train of thought led to a second revision of the aforementioned multithreading code. In this iteration, it was decided that the multithreading would happen one loop higher. This allowed for the calculation of multiple odd-even indexes so that an entire FFT stage worth of calculations could be performed in parallel. This method worked and even outperformed the serial code. However, it only slightly outperformed it, even with eight threads enabled. The reason for this was that since the threads were being created within each stage, the total number of threads created was equal to (the total number of stages) multiplied by (the number of pthreads specified). Although this technically resulted in a speedup, it was not significant enough to justify multithreading. Hence, a better way was needed. This thought process led to the final revision of the pthread code, described below.

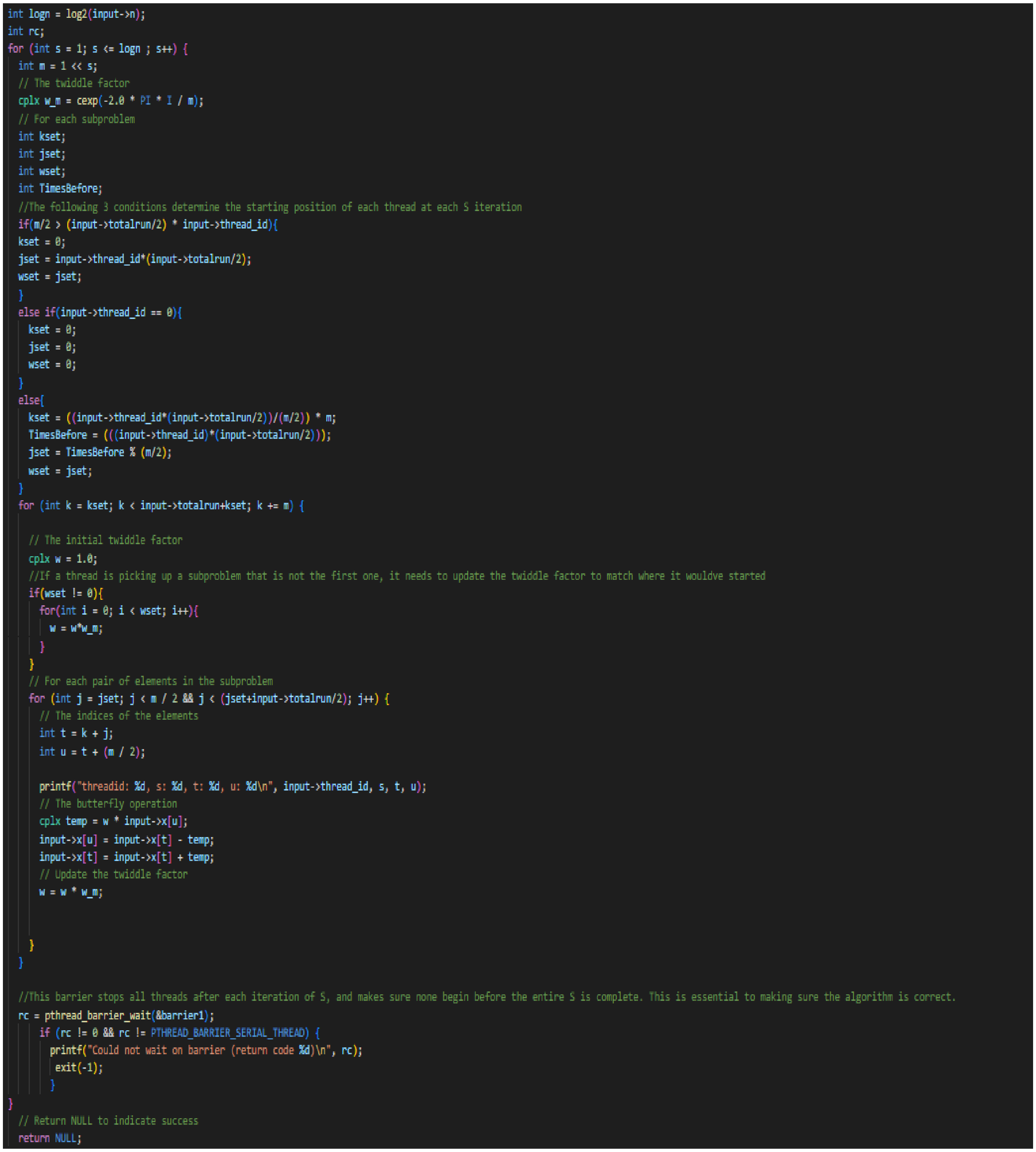
Ultimately rather than thread any of the internal loops, it was decided we would split each stage of S between the threads. This means each thread would handle (N/2)/num_threads computations per S. However, this presents 2 problems. First, each thread would have to know or calculate the starting indexes for j and k, the twiddle factor that would be present when its first computation started, and the final indexes for j and k allowed to it in order to avoid performing calculations assigned to the next thread. In order to accomplish this, there are 3 conditional statements before the k and j loop commence. Using the thread_id, total number of elements, and current values of s, these values are calculated. However, this still leaves us with a second problem; without something to prevent each thread from continuing onto the next S before the other threads have finished the current S, the integrity of the data in the signal can not be guaranteed. Within each S there are no data dependencies between the calculations. Each element is used only once per S. However, every single iteration of S depends on the results of the previous stage. Horrifyingly, a thread may complete an S before the others and access an index before the previous computations have been able to properly transform it, ruining the correctness of the FFT. In order to prevent this, a barrier was placed at the end of the S loop. A visual depiction of this barrier is shown below, along with the order of operations for a multithreaded FFT using 8 threads on a signal consisting of 16 elements:



```
ThreadID: 0, s: 1, t: 0, u: 1
ThreadID: 2, s: 1, t: 4, u: 5
ThreadID: 1, s: 1, t: 2, u: 3
ThreadID: 3, s: 1, t: 6, u: 7
ThreadID: 4, s: 1, t: 8, u: 9
ThreadID: 5, s: 1, t: 10, u: 11
ThreadID: 6, s: 1, t: 12, u: 13
ThreadID: 7, s: 1, t: 14, u: 15
ThreadID: 7, s: 2, t: 13, u: 15
ThreadID: 4, s: 2, t: 8, u: 10
ThreadID: 1, s: 2, t: 1, u: 3
ThreadID: 6, s: 2, t: 12, u: 14
ThreadID: 2, s: 2, t: 4, u: 6
ThreadID: 5, s: 2, t: 9, u: 11
ThreadID: 0, s: 2, t: 0, u: 2
ThreadID: 3, s: 2, t: 5, u: 7
ThreadID: 3, s: 3, t: 3, u: 7
ThreadID: 0, s: 3, t: 0, u: 4
ThreadID: 5, s: 3, t: 9, u: 13
ThreadID: 6, s: 3, t: 10, u: 14
ThreadID: 2, s: 3, t: 2, u: 6
ThreadID: 7, s: 3, t: 11, u: 15
ThreadID: 1, s: 3, t: 1, u: 5
ThreadID: 4, s: 3, t: 8, u: 12
ThreadID: 3, s: 4, t: 3, u: 11
ThreadID: 0, s: 4, t: 0, u: 8
ThreadID: 4, s: 4, t: 4, u: 12
ThreadID: 7, s: 4, t: 7, u: 15
ThreadID: 1, s: 4, t: 1, u: 9
ThreadID: 5, s: 4, t: 5, u: 13
ThreadID: 2, s: 4, t: 2, u: 10
ThreadID: 6, s: 4, t: 6, u: 14
```



In these images the red lines represent the occurrence of the barrier

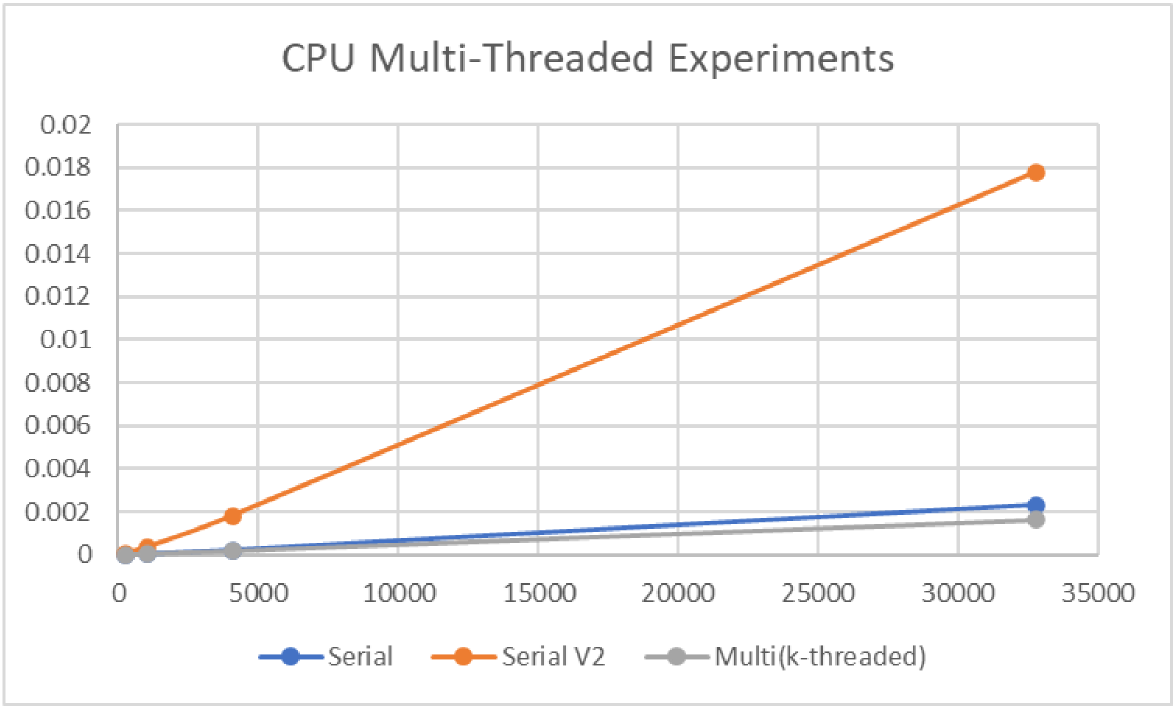


```c
int logn = log2(input->n);
int rc;
for (int s = 1; s <= logn ; s++) {
  int m = 1 << s;
  // The twiddle factor
  cplx w_m = cexp(-2.0 * PI * I / m);
  // For each subproblem
  int kset;
  int jset;
  int wset;
  int TimesBefore;
  //The following 3 conditions determine the starting position of each thread at each S iteration
  if(m/2 > (input->totalrun/2) * input->thread_id){
  kset = 0;
  jset = input->thread_id*(input->totalrun/2);
  wset = jset;
  }
  else if(input->thread_id == 0){
    kset = 0;
    jset = 0;
    wset = 0;
  }
  else{
    kset = ((input->thread_id*(input->totalrun/2))/(m/2)) * m;
    TimesBefore = (((input->thread_id)*(input->totalrun/2)));
    jset = TimesBefore % (m/2);
    wset = jset;
  }
  for (int k = kset; k < input->totalrun+kset; k += m) {

    // The initial twiddle factor
    cplx w = 1.0;
    //If a thread is picking up a subproblem that is not the first one, it needs to update the twiddle factor to match where it wouldve started
    if(wset != 0){
      for(int i = 0; i < wset; i++){
        w = w*w_m;
      }
    }
    // For each pair of elements in the subproblem
    for (int j = jset; j < m / 2 && j < (jset+input->totalrun/2); j++) {
      // The indices of the elements
      int t = k + j;
      int u = t + (m / 2);

      printf("threadid: %d, s: %d, t: %d, u: %d\n", input->thread_id, s, t, u);
      // The butterfly operation
      cplx temp = w * input->x[u];
      input->x[u] = input->x[t] - temp;
      input->x[t] = input->x[t] + temp;
      // Update the twiddle factor
      w = w * w_m;


    }
  }

  //This barrier stops all threads after each iteration of S, and makes sure none begin before the entire S is complete. This is essential to making sure the algorithm is correct.
  rc = pthread_barrier_wait(&barrier1);
      if (rc != 0 && rc != PTHREAD_BARRIER_SERIAL_THREAD) {
        printf("Could not wait on barrier (return code %d)\n", rc);
         exit(-1);
        }
}
  // Return NULL to indicate success
  return NULL;
```

The code for the final multithreaded FFT is shown above(this code can be found in **fft_pthread.c**)

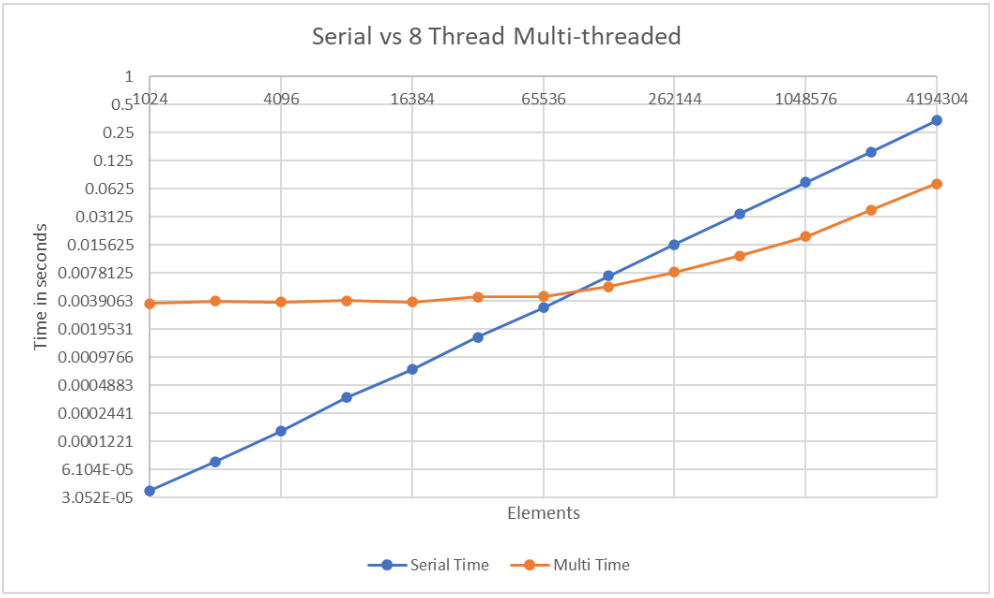
5. Multithreaded performance

Inefficient Pthread FFT Strategy

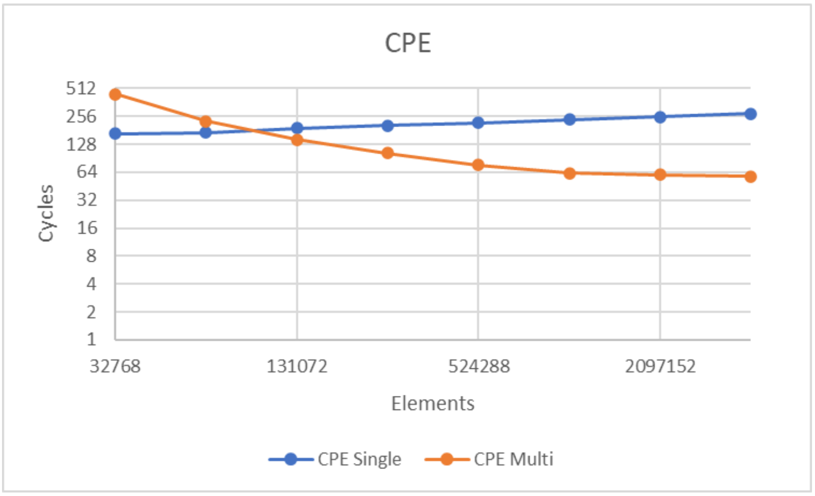


The graph above compares the performance of the second implementation of the multithreading code with the serial and serial v2 codes. As can be seen, using eight threads only slightly outperforms the serial code. It is worth noting that serial v2 was a test code that utilized two lists, with one containing the real part and the other containing the imaginary part. The discrepancy in performance between serial and serial v2 is likely due to inefficient data access across the two lists resulting from the nature of the Cooley-Tukey algorithm used in serial v2. In contrast, the serial code utilized a complex class that contained both the real and imaginary parts in a single index, minimizing inefficient data access. Lastly, it should be noted that these results were obtained using a **Core i5 13600K CPU**.

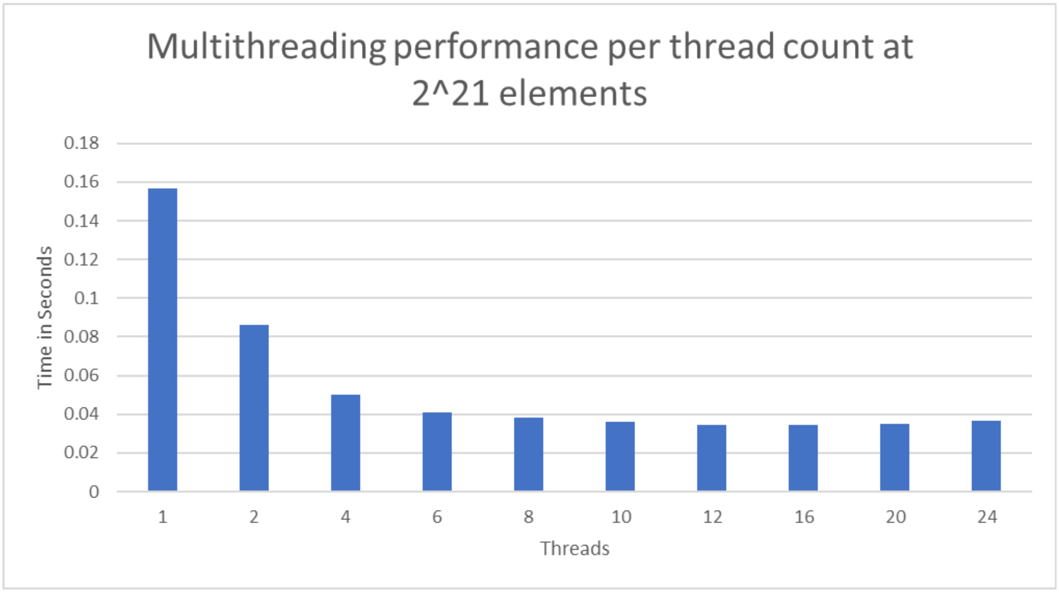
Final FFT Pthread Strategy



The graph above displays both the advantages and shortcomings of our pthread approach to the FFT. Below 2^{17} elements, the performance of the multithreaded approach is flat at ~.004 seconds per run. This can be attributed to the overhead of creating and launching the threads. This results in a massive slowdown when compared to the serial at the same values. However, once we reach 2^{17} elements, the overhead forced upon us by the barrier is no longer the chief contributor to the time. At this point each thread is handling such a large set of elements that the true performance can be seen. Starting from here the O(NlogN) complexity begins to show itself. Here, the multithreaded code begins to outpace the serial with a relative performance uplift of ~7x. This relationship with the overhead can also be seen in the cpe graphs, where the CPE of the multithreaded approach improves as the number of elements increases:



Finally, we can show the scaling of the multithreaded approach as we increase the thread count:



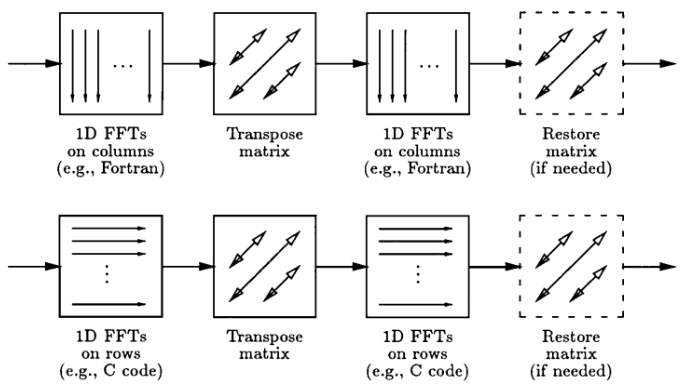
Here we can see the logarithmic relationship our performance has with the thread count. As the thread count increases there is a notable increase in performance. However as we go, the benefits adding additional threads provides gets smaller and smaller. Once we reach 12 threads, any benefit we receive by adding more is marginal. **It should be noted these tests were conducted using an Intel i7-13700k**. Due to the heterogeneous nature of this CPU, once we exceed 16 threads we have run out of available threads for the P Cores. At this point the CPU begins to utilize the single threaded E-Cores, which are less performant than the P cores. Due to the barrier present in the code, each stage of the FFT can only be completed as fast as the slowest thread finishes. Now that we are using e core threads, this time to finish increases, resulting in some noticeable performance loss at 20 and 24 threads.

6. Multithreading FFT via GPU

The original approach to parallelizing the FFT code on a GPU was similar to the strategy used for pthreading on the CPU. It was assumed that since the GPU did not require the creation and destruction of threads, parallelizing each odd-even index calculation performed by the FFT could lead to improved performance. However, this idea was quickly disproven, as running this code on CUDA did not yield any results that outperformed the optimized CPU multithreaded code in any significant way. The reason for this was that too many operations were still being performed by the CPU, and the GPU was only being used for the final step of the operation. To fully leverage the massive number of cores available on the GPU, the Cooley-Tukey algorithm for FFTs needed to be applied in a different way. The 1D Cooley-Tukey algorithm works well on the CPU but is not as efficient on the GPU due to the high overhead of launching multiple kernels.

This is where 2D FFT's come into play. A 2D FFT can be computed by performing the 1D Cooley-Tukey FFT on a 2D matrix's columns or rows then transposing the matrix and repforming the 1D fft in the same manor as before. The advantage of performing a 2D FFT on a 1D list is that by casting to a 2D matrix, the GPU can easily block the calculations, implement efficient memory access patterns, and alow for better memory aliassing. An image of what this process looks like is shown in the diagram below.

Figure 23.1 Sequential row-column 2D FFT algorithm—two implementations.


1D FFTs on columns (e.g., Fortran)
Transpose matrix
1D FFTs on columns (e.g., Fortran)
Restore matrix (if needed)
1D FFTs on rows (e.g., C code)
Transpose matrix
1D FFTs on rows (e.g., C code)
Restore matrix (if needed)

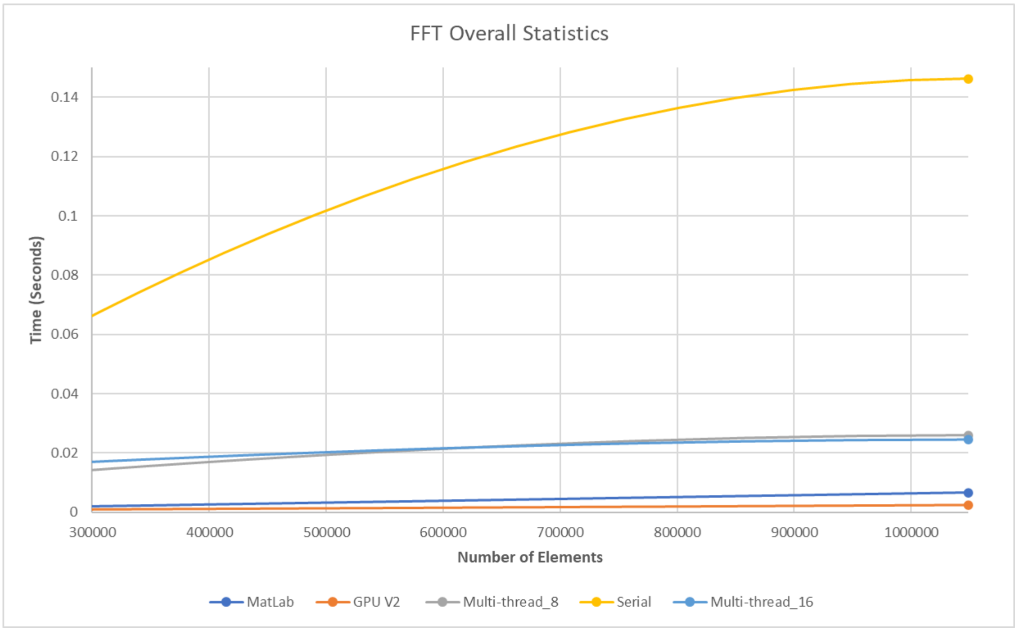
(Chu and George 2000)

The initial concept behind creating the pseudocode involved creating a block of shared data, performing a row-wise Cooley-Tuckey FFT, followed by a column-wise Cooley-Tuckey FFT on each block for all blocks, and eventually writing the shared data back to the primary device variable. The complete implementation of this kernel is provided below. Note that a row-wise FFT followed by a column-wise FFT was chosen over the transpose step for the sake of simplicity. Moreover, since everything is broken up into blocks of shared memory, a transpose operation isn't expected to provide too much of an improvement over the current implementation.

7. GPU Code:

```
__global__ void fft_kernel(float2* data, int N) {
  // Shared memory for storing intermediate results
  __shared__ float2 shared[BLOCK_SIZE][BLOCK_SIZE + 1];

  // Compute the global indices
  int x = blockIdx.x * blockDim.x + threadIdx.x;
  int y = blockIdx.y * blockDim.y + threadIdx.y;
  int index = y * N + x;

  // Copy the data to the shared memory
  shared[threadIdx.y][threadIdx.x] = data[index];
  __syncthreads();

  // Perform the row-wise FFT
  for (int k = 0; k < blockDim.x; k++) {
    float2 w;
    w.x = cosf(-2.0f * PI * k / N);
    w.y = sinf(-2.0f * PI * k / N);
    float2 t;
    t.x = shared[threadIdx.y][k].x * w.x - shared[threadIdx.y][k].y * w.y;
    t.y = shared[threadIdx.y][k].x * w.y + shared[threadIdx.y][k].y * w.x;
    shared[threadIdx.y][k] = t;
  }
  __syncthreads();

  // Perform the column-wise FFT
  for (int k = 0; k < blockDim.y; k++) {
    float2 w;
    w.x = cosf(-2.0f * PI * k / N);
    w.y = sinf(-2.0f * PI * k / N);
    float2 t;
    t.x = shared[k][threadIdx.x].x * w.x - shared[k][threadIdx.x].y * w.y;
    t.y = shared[k][threadIdx.x].x * w.y + shared[k][threadIdx.x].y * w.x;
    shared[k][threadIdx.x] = t;
  }
  __syncthreads();

  // Copy the data back to the global memory
  data[index] = shared[threadIdx.y][threadIdx.x];
}

void fft(float2* data, int N) {
  int blockSize = BLOCK_SIZE;
  dim3 dimBlock(blockSize, blockSize, 1);
  dim3 dimGrid(N / blockSize, N / blockSize, 1);

  // Call the CUDA kernel
  fft_kernel<<<dimGrid, dimBlock>>>(data, N);
  cudaDeviceSynchronize();
}
```

8. GPU Performance Results:



As expected, the 2D GPU code outperforms the serial and multithreaded 8- and 16-thread computations. As an additional bonus, it even outperforms the default MatLab FFT. It is worth noting that these results were obtained using an **i5 13600K** processor and an **Nvidia RTX 4080** graphics card with the Cuda 12.1 toolkit, 32 threads per block, and num. elements/threads per block blocks.

9. Conclusions and lessons learned

The fast fourier transform acts as one of the bedrocks of modern computing. Its performance, while a massive improvement over DFT, can still be improved by a wide range of multithreading practices. With our simple pthread strategy we could find a performance benefit of up to 8x at very large data sets. Even more impressive, the performance increase offered by utilizing the gpu blew both the serial and pthread approach away. Unsurprisingly, the lack of data dependencies within each S of the FFT lends itself to easy parallelization, however there are still improvements to our code that could have been made. Within the multithreaded pthread code, there exists conditionals used to calculate the start k, j, and twiddle factor. These conditionals could be removed, ridding us of any performance penalty associated with branch misprediction. Instead, these starting positions could be precalculated and passed in to each thread, trading the significantly smaller penalty of additional memory access with the relatively massive performance hit that comes with branch misprediction. Additionally, while we did not have time to implement it ourselves, the introduction of AVX vectorization within the serial code could also have significantly increased performance. The same lack of dependencies within each stage of S that made multithreading so easy would also enable a relatively simple implementation of vectors, which at large data sets could have had massive performance improvements over the standard serial.

10. **Some notes about the code**

Within both the serial and multithreaded implementations of the FFT, arrays of complex numbers utilizing the cplx library were used as our inputs.

These libraries were included:
<stdio.h>
<stdlib.h>
<math.h>
<complex.h>
<pthread.h>
<time.h>

Instead of running both CPU codes separately, a file named fft_final.c has been included. This file generates a filled array of complex numbers for both the serial and multithreaded approaches. The size of this array can be changed by altering the value at line 11 called long int N as well as altering the number initializing the global array used in line 14. The code will then run both num_iters times (changeable at line 13, currently it is set to 10) and report the timed average of these runs for each. Compile the program using the commented line on line 7:

```
gcc fft_final.c -o fft_final -lpthread
```

And run the computed fft_final program

The programs are individually available in fft_pthread.c and fft_serial.c for easy readability as fft_final.c includes examples of other pthread attempts that muddle the readability of the code

11. **Code Download:** https://github.com/yanglu149/FFT-Optimization-

References:

Chu, Eleanor, and Alan George. “Inside the FFT Black Box.” *Computational Mathematics*, 1999, https://doi.org/10.1201/9781420049961.

rohit_is. “Iterative Fast Fourier Transformation for Polynomial Multiplication.” GeeksforGeeks, GeeksforGeeks, 4 Apr. 2023, https://www.geeksforgeeks.org/iterative-fast-fourier-transformation-polynomial-multiplication/.

rohit_is. “Fast Fourier Transformation for Polynomial Multiplication.” GeeksforGeeks, GeeksforGeeks, 15 Mar. 2023, https://www.geeksforgeeks.org/fast-fourier-transformation-poynomial-multiplication/.