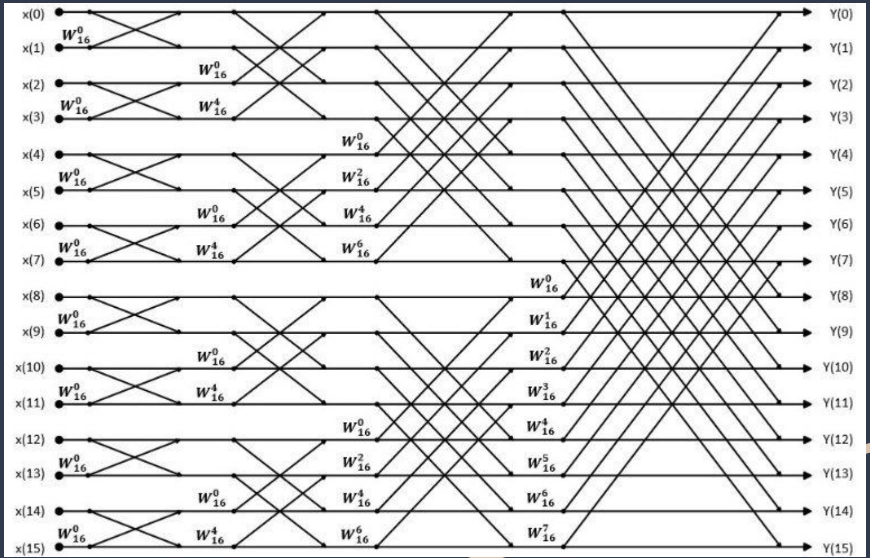
FFT Multithreading on CPU & GPU

Evan Lang & Yang Lu

What is a FFT



DFT (Discrete Fourier Transform) O(n^2):

$$X(k) = \sum_{n=0}^{N-1} x(n) e^{-j2\pi kn/N}$$

- Usually used to derive a frequency domain representation of a signal

FFT (Fast Fourier Transform): O(nlog(n))

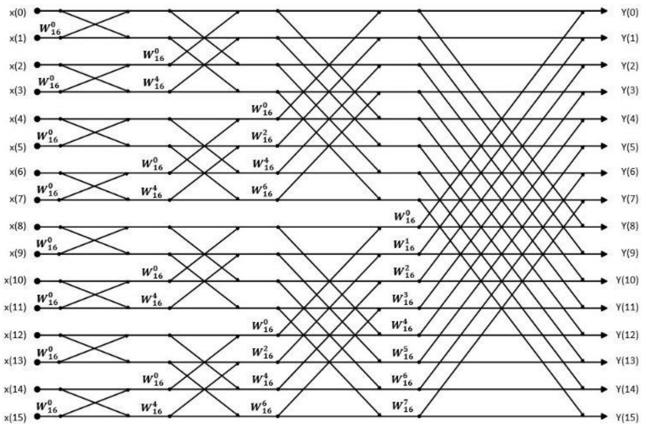
- Divide and conquer approach to the DFT algorithm
- Today we will be specifically focused on the radix-2 Cooley-Tukey DIF_{RN} FFT algorithm.
- Divides DFT in $\log_2 n$ subproblems
- Requires n/2 operations per subproblem

Our Serial Code

```c
void fft_helper(fft_helper_input input)
{
    // The size of each subproblem is 2^s
    int m = 1 << input.s;
    // The twiddle factor
    cplx w_m = cexp(-2.0 * PI * I / m);
    // For each subproblem
    for (int k = 0; k < input.n; k += m) {
      // The initial twiddle factor
      cplx w = 1.0;
      // For each pair of elements in the subproblem
      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;
      }
    }
}

// Perform an iterative FFT on a vector of complex numbers
void fft(cplx *x, int n) {
  // Assume n is a power of 2
  int logn = log2(n);
  // Rearrange the elements of x according to the bit-reversed order
  for (int i = 0; i < n; i++) {
    unsigned int j = reverse(i, logn);
    if (j > i) {
      swap(&x[i], &x[j]);
    }
  }
  //Begin the butterfly operations per S
  for (int s = 1; s <= logn; s++) {
    // Create a struct to hold the inputs to fft_helper
    fft_helper_input input = {n, s, x};
    fft_helper(input);
  }
}
```

Replicate the butterfly diagram:

- Each computation of the butterfly will be defined by the S-loop
- Each series of numbers within a stage(S) will be defined by the K loop
- Each calculation within k will be called by the j loop
- Each calculation performed will be a odd even pair on the indices



First Multithreaded Strategy

```c
// Perform the butterfly operations
for (long s = 1; s <= logn; s++) {
    long m = 1 << s;
    // The twiddle factor
    cplx w_m = cexp(-2.0 * PI * I / m);
    // For each subproblem
    for (long k = 0; k < n; k += m) {
        // The initial twiddle factor
        cplx w = 1.0;
        // For each pair of elements in the subproblem
        long mDiv2 = m/2;
        pthread_t tid[mDiv2];
        fft_helper_bad_input input[mDiv2];
        for (long j = 0; j <mDiv2; j++) {
            //fft_helper_bad(&input);
            input[j].x = x;
            input[j].j = j;
            input[j].k = k;
            input[j].m = m;
            input[j].n = n;
            input[j].w_m = w_m;
            input[j].w = w;
            w = input[j].w * input[j].w_m;
            if(j >= num_threads){
                pthread_join(tid[j-num_threads], NULL);
            }
            pthread_create(&tid[j], NULL, fft_helper_bad, &input[j]);
            //pthread_join(tid[j], NULL);
        }
        for(long j = mDiv2-num_threads; j <mDiv2; j++){
            pthread_join(tid[j], NULL);
        }
    }
```

- Threading within the j loop.
  - Each calculation would be threaded
  - Very minimal operations performed per thread
  - Cost of thread overhead outweighed the threading

```c
void *fft_helper_bad(void *arg)
{
    fft_helper_bad_input *input = (fft_helper_bad_input *) arg;
    long k = input -> k;
    long m = input -> m;
    long j = input -> j;
    long n = input->n;

    // The indices of the elements
    long t = k + j;
    long u = t + m / 2;

    // The butterfly operation
    cplx temp = input -> w * input -> x[u];
    input -> x[u] = input -> x[t] - temp;
    input -> x[t] = input -> x[t] + temp;
    //printf("k: %d, j: %d, t: %d, u: %d\n", k, j, t, u);
}
```

Second Multithreaded Strategy

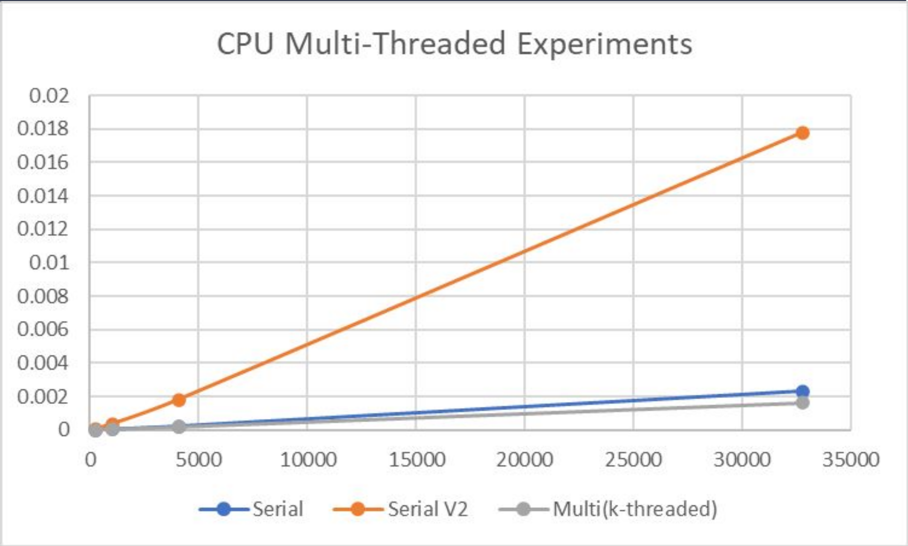
```
// Perform the butterfly operations
for (long s = 1; s <= logn; s++) {
    long m = 1 << s;
    fft_helper_kthreaded_input input[num_threads];
    pthread_t tid[num_threads];
    long k_per_thread = ceil(1.0*n/num_threads);
    if (k_per_thread < m){
        k_per_thread = m;
    }
    long next_kstart = 0;
    for(long th_count = 0; th_count < num_threads; th_count++){
        input[th_count].m = m;
        input[th_count].x = x;
        input[th_count].kstart = next_kstart;
        next_kstart += k_per_thread;
        next_kstart = ceil(1.0*next_kstart/m)*m;
        input[th_count].kend = k_per_thread + input[th_count].kstart;
        if(input[th_count].kend > n)
        {
            input[th_count].kend = n;
        }
        if(input[th_count].kstart > n)
        {
            break;
        }
        fft_helper_kthreaded(&input[th_count]);
    }
    for(long th_count = 0; th_count < num_threads; th_count++){
        pthread_join(tid[th_count], NULL);
    }
}
}
```

- Threading within k-loop
    - Much better than previous
    - However, still not better than serial
    - S loop still creating an enormous amount of threads per iteration

Is there a way to optimize this so that fewer threads are created?

```
void *fft_helper_kthreaded(void *arg)
{
    fft_helper_kthreaded_input *input = (fft_helper_kthreaded_input *) arg;
    cplx *x = input->x;
    long m = input->m;
    long kstart = input->kstart;
    long kend = input->kend;
    long mDiv2 = m/2;
    cplx w_m = cexp(-2.0 * PI * I / m);
    for (long k = kstart; k < kend; k += m) {
        // The initial twiddle factor
        cplx w = 1.0;
        //For each pair of elements in the subproblem
        for (long j = 0; j <mDiv2; j++) {
            // The indices of the elements
            long t = k + j;
            long u = t + m / 2;

            // The butterfly operation
            cplx temp = w * x[u];
            x[u] =  x[t] - temp;
            x[t] += temp;
            w = w * w_m;
        }
    }
}
```

Results:



- The multi-threaded j loop doesn't fit
  - The overhead of creating threads is so high
  - Computation of each calculation is not worth the high overhead
- Multi-threading on k is beating Serial
  - It is barely doing so with 8 threads.
  - Trend doesn't change at higher N
  - We have to be able to do better right?
- Serial V2
  - Was another serial we designed that leveraged 2 arrays instead of the complex number type.
  - Otherwise same code
  - Much worse than the other Serial
    - Needing to access memory from 2 arrays at inefficient locations

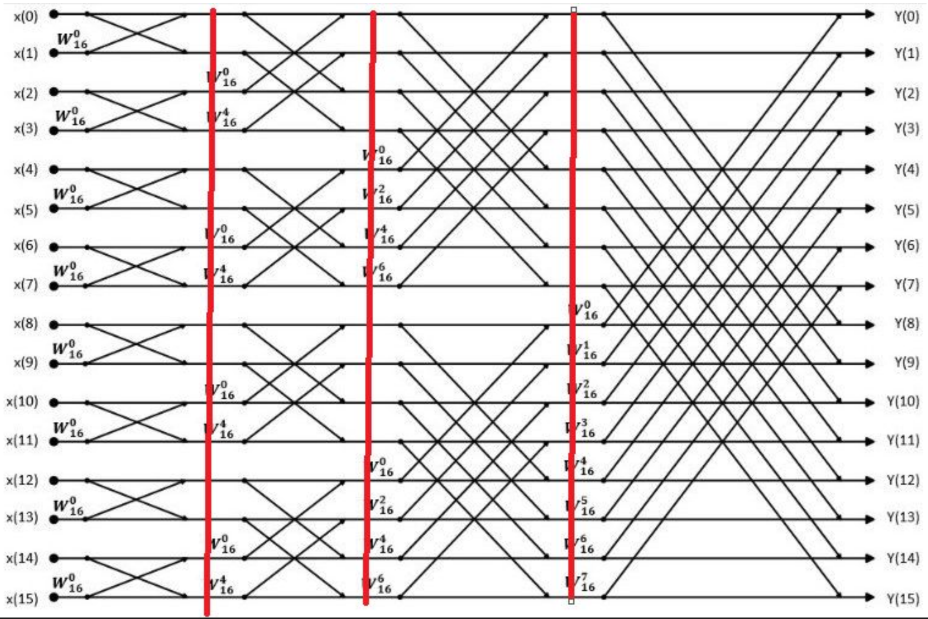
Final Multithreaded strategy

```c
void *fft_helper_multi(void *arg)
{
  //pthread_mutex_lock(&mutexA);
    fft_helper_multi_input *input = (fft_helper_multi_input *) arg;
  long logn = log2(input->n);
  long rc;
  for (long s = 1; s <= logn ; s++) {
    long m = 1 << s;
    // The twiddle factor
    cplx w_m = cexp(-2.0 * PI * I / m);
    // Starting point for each thread
    long kset;
    long jset;
    long wset;
    long TimesBefore;
    //Determines the starting k, j, and twiddle value for each thread based on the thread id and s value
    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 (long 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(long i = 0; i < wset; i++){
          w = w * w_m;
        }
      }
      // For each pair of elements in the subproblem
      for (long j = jset; j < m / 2 && j < (jset+input->totalrun/2); j++) {
        // Calculates the index of each of the 2 elements
        long t = k + j;
        long 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;
      }
    }
    //This barrier prevents the threads from moving on to the next s value until all threads have finished the current s value
    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;
}
```

- Each thread handles (NUM_ELEMENTS/2)/NUM_THREADS operations per s value
- Starting values for each thread are determined by the threadID
- Performs a limited transformation on the range of elements allocated to each thread

```
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
```

A barrier is present at the end of the s loop to maintain data coherency



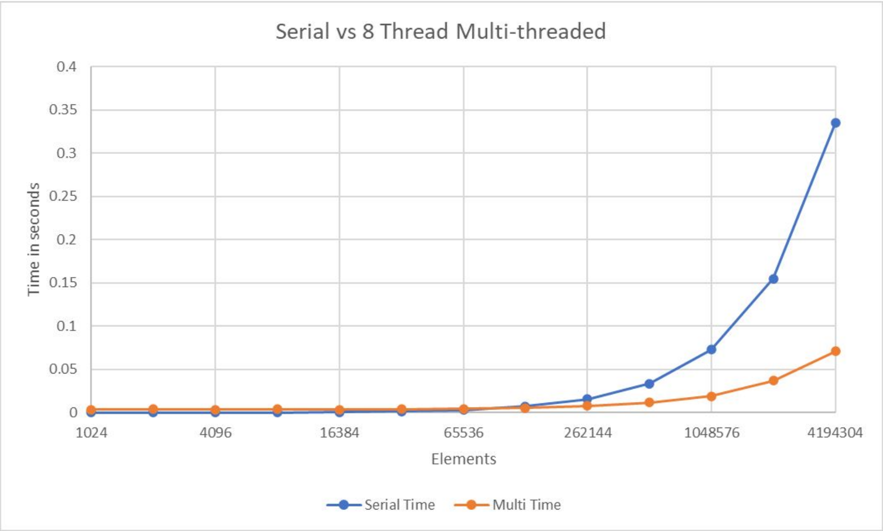

```
Input vector:
[1.00+0.00i, 2.00+0.00i, 3.00+0.00i, 4.00+0.00i, 5.00+0.00i, 6.00+0.00i, 7.00+0.00i, 8.00+0.00i, 9.00+0.00i, 10.00+0.00i, 11.00+0.00i, 12.00+0.00i, 13.00+0.00i, 14.00+0.00i, 15.00+0.00i, 16.00+0.00i]

0.004000 seconds

Output vector:
[136.00+0.00i, -8.00+40.22i, -8.00+19.31i, -8.00+11.97i, -8.00+8.00i, -8.00+5.35i, -8.00+3.31i, -8.00+1.59i, -8.00+0.00i, -8.00-1.59i, -8.00-3.31i, -8.00-5.35i, -8.00-8.00i, -8.00-11.97i, -8.00-19.31i, -8.00-40.22i]
```

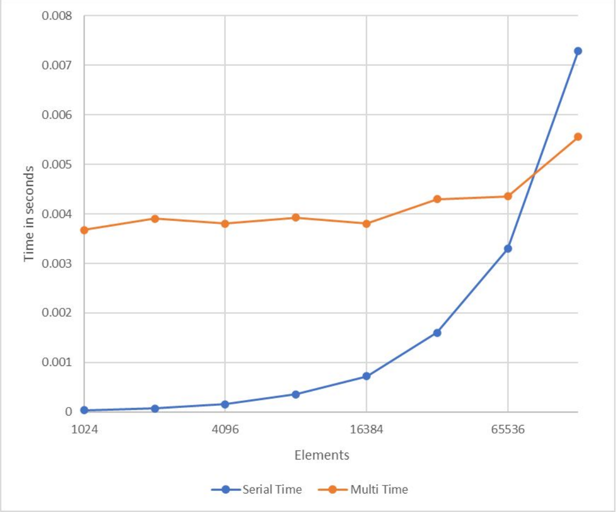
Performance vs serial

Below 2^{17} elements the serial code maintains a large performance advantage over the multithreaded implementation

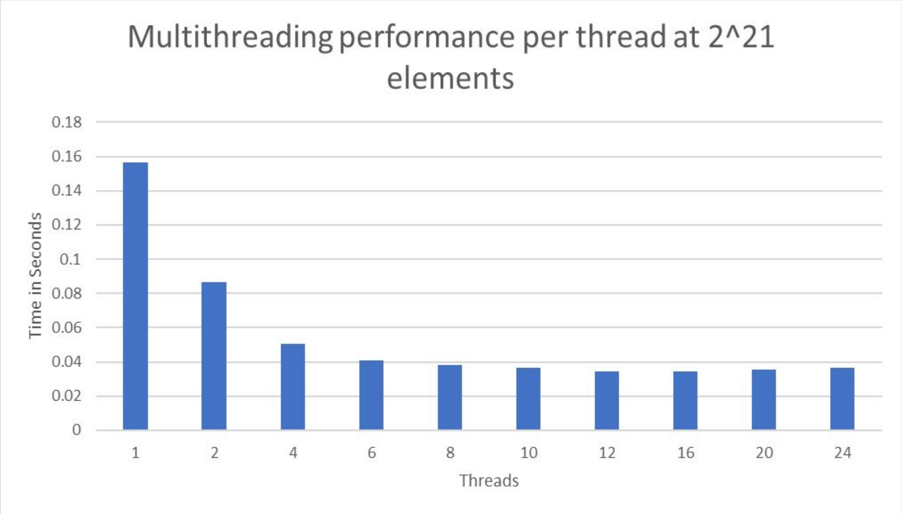
The multithreaded code scales significantly better than the serial above 2^{17} elements maintaining almost an 8x advantage in performance at 2^{22} elements

All data presented is the average runtime of 10 runs of each form of FFT





Performance scaling with threads

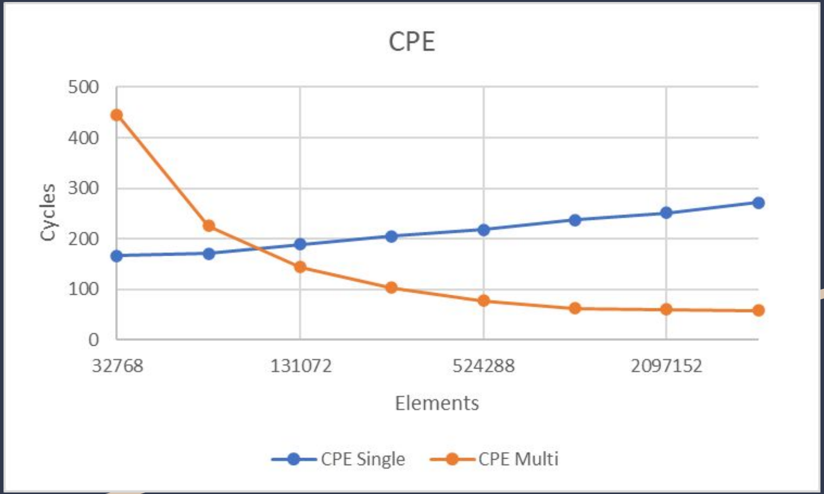


Initially the performance increase provided by the threads is large, with the an almost 2x increase in performance from 1 to 2 threads

Starting at 6 threads the benefit of adding more threads begins to be marginal, with only single to low double digit percent improvements to the performance

The optimal performance was found at 12 threads. Increasing the thread count from 12 began to impart small regressions in total performance

Limitations and inefficiencies



- At smaller sizes of N the overhead of the multi threading strategy resulted in huge CPE's values.
- As the amount of elements increases the effect of this overheard is dissipated.
- Presence of conditionals in the s loop could lead to performance loss from branch misses
- Could be avoided by pre-calculating starting positions and passing them into the thread

GPU Version 1

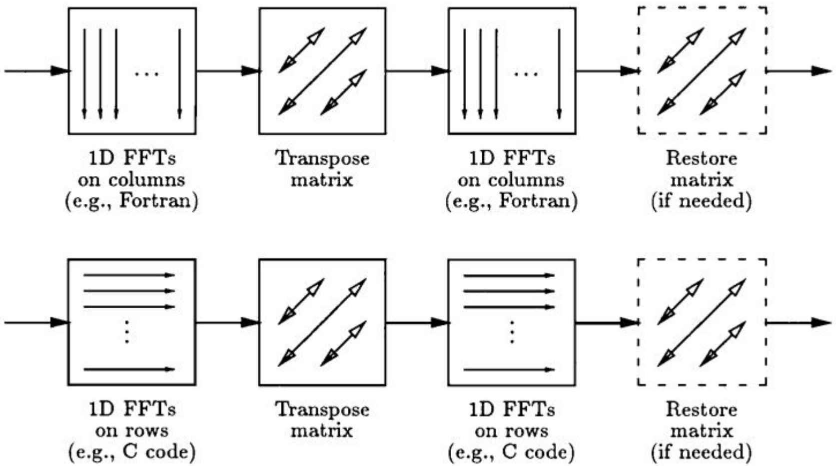
```
__global__ void FFT_kernel(float2 *d_data, int N, int m, int k)
{
    int j = threadIdx.x + blockDim.x * blockIdx.x; //j Loop controlled by tid
    if (j < m/2)
    {
        int t = j + k;
        int u = t + m/2;
        // printf("s: %d, k: %d, j: %d\n", (int)log2f(m), k, j);
        float2 twiddle_factor = make_float2(cosf(2 * PI * j / m), sinf(2 * PI * j / m));
        float2 temp = cuCmulf(d_data[u], twiddle_factor);
        d_data[u] = cuCsubf(d_data[t], temp);
        d_data[t] = cuCaddf(d_data[t], temp);
    }
}


void FFT(float2 *h_data, int N)
{
    float2 *d_data;
    cudaMalloc((void **)&d_data, sizeof(float2) * N);
    cudaMemcpy(d_data, h_data, sizeof(float2) * N, cudaMemcpyHostToDevice);
    int log2n = log2f(N);
    int threads_per_block = 256;
    int num_blocks = (N - 1) / threads_per_block + 1;

    for (int s = 1; s <= log2n; s++)
    {
        int m = 1 << s;
        for (int k = 0; k < N; k += m)
        {
            FFT_kernel<<<num_blocks, N/2>>>(d_data, N, m, k);
            cudaDeviceSynchronize();
        }
    }

    cudaMemcpy(h_data, d_data, sizeof(float2) * N, cudaMemcpyDeviceToHost);
    cudaFree(d_data);
}
```

- Started with threading on j loop
    - Results still very close to serial
- Needed a new approach
    - 2D FFT

GPU Version 2

```
__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];
}
```

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



Final Results