6 By Jianxin Wang, on parallel discrete Fourier transform

Introduction to Parallel Algorithms by Joseph JaJa in Chapter 8.3

6.1 Introduction

Fourier transform is widely used in different engineering applications, such as signal processing to analyze frequency , image compression and image reconstruction. My project is to parallelize the calculation of discrete fourier transform. The traditional fourier transform is serialized and very ineffecient, In order to parallelize the algorithm, we need to divide the calculation into chunks. Fast Fourier Transform algorithm achieves this by using divide and conquer to break it into smaller parts that don't interfere with each others. We can then use cuda to run different parts in parallel.

6.2 Discrete Fourier Transform

Given a n dimensional vector x with complex number. After discrete fourier transform of x, the output is n dimensional vector y. Each element y_j in y is calculated as

$$y_j = \sum_{k=0}^{n-1} w^{jk} x_k \tag{3}$$

where w is $e^{i\frac{2\pi}{n}} = \cos\frac{2\pi}{n} + i\sin\frac{2\pi}{n}$

6.3 Brutal Fourier Transform Algorithm

Naively to implement the above discrete fourier transform is using two for loops to transform the n dimensional vector. The runtime would be $\mathcal{O}(N^2)$ The implementation is as follow:

```
void Fourier_cpu_original (float *R, float *I, float *R_2, float *I_2, int n) {
   for (int i =0; i<n; i++){
   for (int j=0 ; j<n ; j++) {
        R_2[i] += R[j] * cos(2 * i * j * pi / n) - I[j] * sin(2 * i * j * pi / n);
        I_2[i] += R[j] * sin(2 * i * j * pi / n) + I[j] * cos(2 * i * j * pi / n);
   }
}
}
```

The input for the algorithm is complex numbers. R is the list of real part of input and I is the list of imaginary part of input. R_2 and I_2 are the output of real and imaginary part of the input after the Fourier transform. The algorithm calculates each output by adding the multiplication of the every input with cos and sin function. This algorithm is slow and we can optimize it by using divide and conquer to break the calculation into smaller parts.

6.4 Fast Fourier Transform Algorithm

The Fast Fourier Transform algorithm takes 2^n number of inputs and applies Discrete Fourier Transform to produce 2^n outputs. For example, given input $x_0, x_1, x_2, x_3, x_4, x_5, x_6, x_7$, those numbers will go through $\log_2 8 = 4$ layers to output the final number. The diagram below shows the overview of the algorithm:

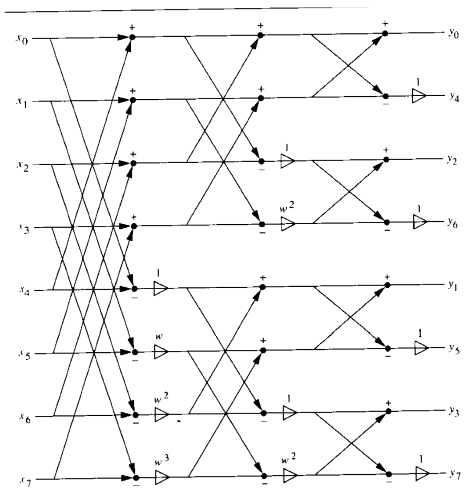

Figure 16: Overview of FFT algorithm

For the first layers:

$$x_0 = x_0 + x_4 \tag{4}$$
$$x_1 = x_1 + x_5 \tag{5}$$
$$x_2 = x_2 + x_6 \tag{6}$$
$$x_3 = x_3 + x_7 \tag{7}$$
$$x_4 = x_0 - x_4 \tag{8}$$
$$x_5 = w(x_1 - x_5) \tag{9}$$
$$x_6 = w^2(x_2 - x_6) \tag{10}$$
$$x_7 = w^3(x_3 - x_7) \tag{11}$$

For the second layer:

$$x_0 = x_0 + x_2 \tag{12}$$
$$x_1 = x_1 + x_3 \tag{13}$$
$$x_2 = x_0 - x_2 \tag{14}$$
$$x_3 = w^2(x_1 - x_3) \tag{15}$$
$$x_4 = x_6 + x_4 \tag{16}$$
$$x_5 = x_7 + x_5 \tag{17}$$
$$x_6 = x_4 - x_6 \tag{18}$$
$$x_7 = w^2(x_5 - x_7) \tag{19}$$

For the output layer:

$$y_0 = x_0 + x_1 \tag{20}$$
$$y_4 = x_0 - x_1 \tag{21}$$
$$y_2 = x_3 + x_2 \tag{22}$$
$$y_6 = x_2 - x_3 \tag{23}$$
$$y_1 = x_5 + x_4 \tag{24}$$
$$y_5 = x_4 - x_5 \tag{25}$$
$$y_3 = x_7 + x_6 \tag{26}$$
$$y_7 = x_6 - x_7 \tag{27}$$

After each layer, the algorithm breaks the blocks in each layer into half. Every part in each layer is not interfering with each other. In the example I shown here, first layer has one block, the second layer has two blocks, and the third layer has four blocks. Also in each block, the output in first half of the block is calculated by addition and output in second half of the block is calculated by subtraction and multiplying with the corresponding w value. The run time for this algorithm is $\mathcal{O}(N \log N)$

To avoid using if condition to separate the threads for different calculation causing divergence issues, we used some mathematical trick to obtain the instruction for each thread's operation.

Following is the detail illustration of the algorithm.

First, for each thread id, it need to know which position it will be in the block according to its level, its associated w value and its operation sign.

```
            int temp = 1<<l;
            int k = threadIdxx/(n/temp);
            int i = threadIdxx - k*(n/temp);
            int kmax = max(0 ,  (i - (n/(2*temp)))*temp);
            float wkreal = cos( (2*pi*((float ) kmax))/( (float) n));
            float wkimag = sin( (2*pi*((float ) kmax))/( (float) n));
            int sign = 1 - 2*(  (i/(n/(2*temp)))  % 2);
```

The k value is telling which block the thread belong to in the level and i is telling the corresponding index in the block. *kmax* is the power value for w for the corresponding thread. *wkreal* is the value of cos and *wkimag* is the value of sin. Sign variable is telling whether it is plus or minus operation for that index.

Then, once we gathered the corresponding information for each thread, we can then use the thread to calculate the value for each index.

The c algorithm implementation for calculation is as follow.

```
  if (l % 2 == 0)
            {
            R_2[threadIdxx] = 0.5 * (1+sign) * (R[threadIdxx] + sign * (wkreal * R[threadIdxx +
    sign* (n/(2*temp))] -  wkimag * I[threadIdxx + sign* (n/(2*temp))]))  + 0.5 * (1 - sign) * (
    wkreal*(R[threadIdxx + sign* (n/(2*temp))]-R[threadIdxx]) - wkimag *(I[threadIdxx + sign* (n
    /(2*temp))]-I[threadIdxx]));

           I_2[threadIdxx] = 0.5 * (1+sign) * (I[threadIdxx] + sign * (wkreal * I[threadIdxx +
    sign* (n/(2*temp))]) +  wkimag * R[threadIdxx + sign* (n/(2*temp))]) + 0.5 * (1 - sign) * (
    wkreal * (I[threadIdxx + sign* (n/(2*temp))]-I[threadIdxx]) + wkimag *(R[threadIdxx + sign* (
    n/(2*temp))]-R[threadIdxx]));
            }

            else {
              R[threadIdxx] = 0.5 * (1+sign) * (R_2[threadIdxx] + sign * (wkreal * R_2[threadIdxx
     + sign* (n/(2*temp))] -  wkimag * I_2[threadIdxx + sign* (n/(2*temp))]))  + 0.5 * (1 - sign)
     * (wkreal*(R_2[threadIdxx + sign* (n/(2*temp))]-R_2[threadIdxx]) - wkimag *(I_2[threadIdxx +
     sign* (n/(2*temp))]-I_2[threadIdxx]));

              I[threadIdxx] = 0.5 * (1+sign) * (I_2[threadIdxx] + sign * (wkreal * I_2[threadIdxx
     + sign* (n/(2*temp))]) +  wkimag * R_2[threadIdxx + sign* (n/(2*temp))]) + 0.5 * (1 - sign)
    * (wkreal * (I_2[threadIdxx + sign* (n/(2*temp))]-I_2[threadIdxx]) + wkimag *(R_2[threadIdxx
    + sign* (n/(2*temp))]-R_2[threadIdxx]));
             }
```

Each level in the calculation need to be updated before they propagate to the next level. The R, I and R_2, I_2 are to save the updated value from current level calculation for the next level calculation. If it is a even level, updated value will save to the R_2 and I_2. If it is a odd level, updated value will save to the R and I. For different sign, there are different operations. If it is a plus sign, the calculation will be

```

R_2[threadIdxx] = R[threadIdxx] + sign * (wkreal * R[threadIdxx + sign* (n/(2*temp))] -  wkimag *
     I[threadIdxx + sign* (n/(2*temp))])

```

```
I_2[threadIdxx] = I[threadIdxx] + sign * (wkreal * I[threadIdxx + sign* (n/(2*temp))]) +  wkimag
    * R[threadIdxx + sign* (n/(2*temp))
```

If it is a minus sign:

```

 R_2[threadIdxx] =  wkreal*(R[threadIdxx + sign* (n/(2*temp))]-R[threadIdxx]) - wkimag *(I[
    threadIdxx + sign* (n/(2*temp))]-I[threadIdxx]);

I_2[threadIdxx] = wkreal * (I[threadIdxx + sign* (n/(2*temp))]-I[threadIdxx]) + wkimag *(R[
    threadIdxx + sign* (n/(2*temp))]-R[threadIdxx]);
```

However the final output layer index is not the same as the input layer index. The relationship between the input and output index is bit reversal. I adapted the implementation from http://groups.csail.mit.edu/cag/streamit/results/fft/code/c/unifft.c
Here is the c code to reverse the index:

```
__host__ __device__ int bitrev(int input, int numbits)
{
  int i, result=0;
  for (i=0; i < numbits; i++)
  {
    result = (result << 1) | (input & 1);
    input >>= 1;
  }
  return result;
}
```

The combination of the Fast Fourier Transform and bit reversal is as follow:

```
void Fourier_cpu (float *R, float *I, float *R_2, float *I_2, float *R_out, float *I_out, int n){
   for (int l = 0; l < (int)(log(n)/log(2)); l++){ //loop through each individual level
    for (int threadIdxx = 0; threadIdxx < n ; threadIdxx++){ //simulating threadIdx.x
    // Calculating the result
            int temp = 1<<l;
            int k = threadIdxx/(n/temp);
            int i = threadIdxx - k*(n/temp);
            int kmax = max(0 ,  (i - (n/(2*temp)))*temp);
            float wkreal = cos( (2*pi*((float ) kmax))/( (float) n));
            float wkimag = sin( (2*pi*((float ) kmax))/( (float) n));
            int sign = 1 - 2*(  (i/(n/(2*temp)))  % 2);
            if (l % 2 == 0)
            {
            R_2[threadIdxx] = 0.5 * (1+sign) * (R[threadIdxx] + sign * (wkreal * R[threadIdxx +
    sign* (n/(2*temp))] -  wkimag * I[threadIdxx + sign* (n/(2*temp))]))  + 0.5 * (1 - sign) * (
    wkreal*(R[threadIdxx + sign* (n/(2*temp))]-R[threadIdxx]) - wkimag *(I[threadIdxx + sign* (n
    /(2*temp))]-I[threadIdxx]));

           I_2[threadIdxx] = 0.5 * (1+sign) * (I[threadIdxx] + sign * (wkreal * I[threadIdxx +
    sign* (n/(2*temp))]) +  wkimag * R[threadIdxx + sign* (n/(2*temp))]) + 0.5 * (1 - sign) * (
    wkreal * (I[threadIdxx + sign* (n/(2*temp))]-I[threadIdxx]) + wkimag *(R[threadIdxx + sign* (
    n/(2*temp))]-R[threadIdxx]));
            }
            else {
              R[threadIdxx] = 0.5 * (1+sign) * (R_2[threadIdxx] + sign * (wkreal * R_2[threadIdxx
     + sign* (n/(2*temp))] -  wkimag * I_2[threadIdxx + sign* (n/(2*temp))]))  + 0.5 * (1 - sign)
     * (wkreal*(R_2[threadIdxx + sign* (n/(2*temp))]-R_2[threadIdxx]) - wkimag *(I_2[threadIdxx +
     sign* (n/(2*temp))]-I_2[threadIdxx]));

              I[threadIdxx] = 0.5 * (1+sign) * (I_2[threadIdxx] + sign * (wkreal * I_2[threadIdxx
     + sign* (n/(2*temp))]) +  wkimag * R_2[threadIdxx + sign* (n/(2*temp))]) + 0.5 * (1 - sign)
    * (wkreal * (I_2[threadIdxx + sign* (n/(2*temp))]-I_2[threadIdxx]) + wkimag *(R_2[threadIdxx
    + sign* (n/(2*temp))]-R_2[threadIdxx]));
```

```

            }
        }
    }
// Bit Reversal
    for(int j = 0; j < n; j++){
    int out_index = bitrev(j, (int)(log(n)/log(2)));
    int check = (int)(log(n)/log(2));
    if (check % 2 == 0) // Different n will result R_out and I_out taking value from either (R, I
    ) or (R_2, I_2)
    {
      R_out[j] = R[out_index];
      I_out[j] = I[out_index];
    }
    else
    {
    R_out[j] = R_2[out_index];
    I_out[j] = I_2[out_index];
    }
    }
}
```

6.5 Cuda Parallelism

We can parallelize the Fast Fourier Transform using multiple blocks and multiple threads. To avoid using syncthreads, I launch a new kernel on each level. After for loop, I launch a kernel just for bit reversal

6.5.1 Kernel Setup

Fast Fourier Kernel setup is as follow

```

__global__ void Fourier(float *R, float *I, float *R_2, float *I_2, int n, int l){
            int temp = 1<<l;
          int idx = threadIdx.x + blockIdx.x*blockDim.x; \\allocate the idx for each thread in
    block
            int k = idx/(n/temp);
            int i = idx - k*(n/temp);
            int kmax = max(0 ,  (i - (n/(2*temp)))*temp);
            float wkreal = cos( (2*pi*((float ) kmax))/( (float) n));
            float wkimag = sin( (2*pi*((float ) kmax))/( (float) n));
            int sign = 1 - 2*(  (i/(n/(2*temp)))  % 2);

            if (l % 2 == 0)
             {
             R_2[idx] = 0.5 * (1+sign) * (R[idx] + sign * (wkreal * R[idx + sign* (n/(2*temp))] -
      wkimag * I[idx + sign* (n/(2*temp))]))  + 0.5 * (1 - sign) * (wkreal*(R[idx + sign* (n/(2*
    temp))]-R[idx]) - wkimag *(I[idx + sign* (n/(2*temp))]-I[idx]));

            I_2[idx] = 0.5 * (1+sign) * (I[idx] + sign * (wkreal * I[idx + sign* (n/(2*temp))]) +
      wkimag * R[idx + sign* (n/(2*temp))]) + 0.5 * (1 - sign) * (wkreal * (I[idx + sign* (n/(2*
    temp))]-I[idx]) + wkimag *(R[idx + sign* (n/(2*temp))]-R[idx]));
             }
             else {
               R[idx] = 0.5 * (1+sign) * (R_2[idx] + sign * (wkreal * R_2[idx + sign* (n/(2*temp)
    )] -  wkimag * I_2[idx + sign* (n/(2*temp))]))  + 0.5 * (1 - sign) * (wkreal*(R_2[idx + sign*
     (n/(2*temp))]-R_2[idx]) - wkimag *(I_2[idx + sign* (n/(2*temp))]-I_2[idx]));

               I[idx] = 0.5 * (1+sign) * (I_2[idx] + sign * (wkreal * I_2[idx + sign* (n/(2*temp)
    )]) +  wkimag * R_2[idx + sign* (n/(2*temp))]) + 0.5 * (1 - sign) * (wkreal * (I_2[idx + sign
    * (n/(2*temp))]-I_2[idx]) + wkimag *(R_2[idx + sign* (n/(2*temp))]-R_2[idx]));
             }
}

```

```
}
```

Bit Reversal Kernel setup is as follow:

```
__host__ __device__ int bitrev(int input, int numbits)
{
  int i, result=0;
  for (i=0; i < numbits; i++)
  {
    result = (result << 1) | (input & 1);
    input >>= 1;
  }
  return result;
} // the bit reversal function


__global__ void bit_reverse (float *R, float *I, float *R_2, float *I_2, float *out_R, float *
    out_I, int n) {
    int idx = threadIdx.x + blockIdx.x*blockDim.x;
    int out_index = bitrev(idx, (int)(log2f(n)));
    int check = (int)(log2f(n));
    if (check % 2 == 0)
    {
    out_R[idx] = R[out_index];
    out_I[idx] = I[out_index];
    }
    else
    {
    out_R[idx] = R_2[out_index];
    out_I[idx] = I_2[out_index];
    }
} // the bit reversal kernel
```

Calling the kernel in main function

```
 for (int l = 0; l < (int)(log(n)/log(2)); l++){
       Fourier<<< num_blocks_per_grid   ,  num_elements_per_block   >>>(dA_R, dA_I, dB_R, dB_I,
    n, l);
    }

    bit_reverse<<< num_blocks_per_grid   ,  num_elements_per_block  >>> (dA_R, dA_I, dB_R, dB_I,
    dout_R, dout_I, n);
```

6.6 Results

Here is the table showing the speed for different size of number in different implementations mentioned above:

Number of elements in input	1024	2048	4096	8192
Brutal FT	105.27	421.39	1681.68	6745.41
FFT CPU	1.14	2.50	5.42	11.67
FFT GPU	0.32	0.35	0.38	0.42

Table 3: The run time comparison of Brutal FT, FFT CPU, and FFT GPU for number of elements in input. All times are in seconds

The result shows the superiority of parallel computing in speed. The run time for GPU FFT is significantly less than Brutal FT and FFT CPU.

6.7 Other Ideas

We can synchronize the thread in block partially. The advantage of FFT algorithm is that each level is divided by blocks. The thread in each block in each level can be synchronize. The cooperative group provides the flexibility to synchronize different groups of the thread. In FFT, we can synchronize just threads in different block (Depending on the number of blocks) as the blocks are not interfering with each other. Due to time constraints, I wasn't able to implement it. I will explore it in the future.

6.8 Conclusion

Before taking this class, I frequently use cuda in deep learning. This class gives me the opportunity to dive deep into cuda. Through working on this project, I have better understanding how to turn a sequential algorithm into a parallelized algorithm. I also learn how to use GPU to further accelerate the run time of parallelized algorithm. I think the most challenging part is implementing the FFT in a version that can apply to GPU. Overall, I was able to work through the whole process and I am excited to research more about it in the future.