Parallel Fast Fourier Transform

ME 766: COURSE PROJECT

Prof. Shivasubramanian Gopalakrishnan

Group: Vatsal (140100026), Nikunj (15D100004), Sumant (15D100020), Shubham (150100060)

Fourier transform

- The Fourier Transform decomposes any function into a sum of sinusoidal basis functions. Each of these basis functions is a complex exponential of a different frequency. The Fourier Transform therefore gives us a unique way of viewing any function - as the sum of simple sinusoids.

$$\mathscr{F}\{g(t)\} = G(f) = \int_{-\infty}^{\infty} g(t)e^{-i2\pi ft}dt$$

DFT- $$\sum_{n=0}^{N-1} s(n)\left(cos\left(2\pi\frac{k}{N}n\right) - jsin\left(2\pi\frac{k}{N}n\right)\right)$$

- Complexity of DFT is 2N^2 which can be reduced to N^2 using periodicity of sin and cosine function.

Fast Fourier transform :

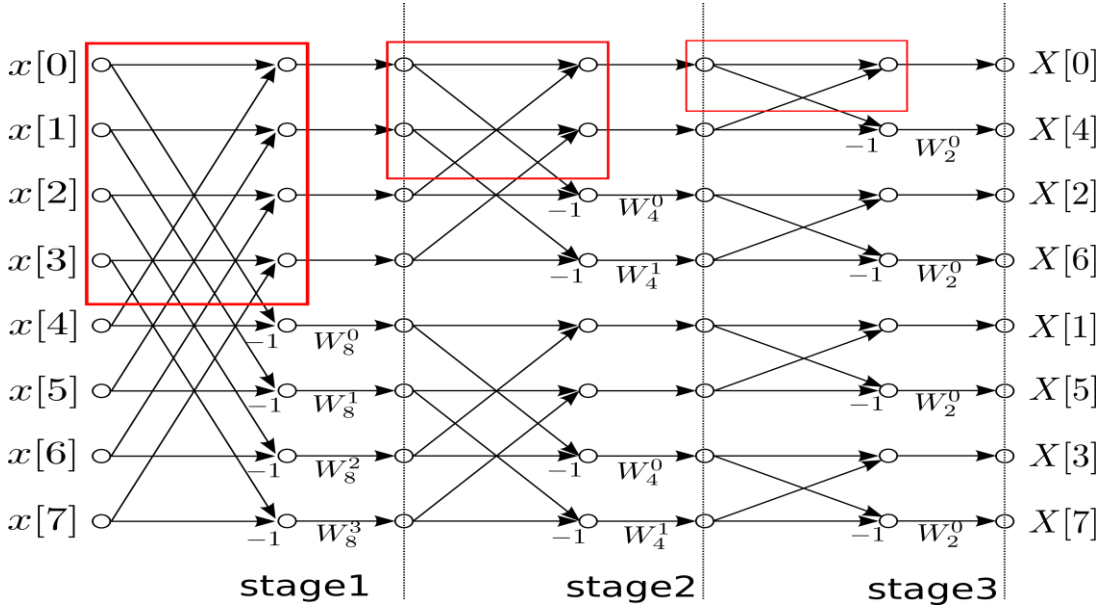
- An FFT algorithm computes the discrete Fourier transform (DFT) of a sequence, or its inverse (IFFT). Fourier analysis converts a signal from its original domain to a representation in the frequency domain and vice versa. An FFT rapidly computes such transformations by factorizing the DFT matrix into a product of sparse (mostly zero) factors. As a result, it manages to reduce the complexity of computing the DFT .
- Types of FFT algorithm : Cooley –Tuckey , Prime-factor FFT, Brunn's FFT, Rader's FFT, and more.
- We tried to implement Cooley-Tuckey algorithm with some variation.

Cooley-Tuckey FFT Algorithm

- A **radix-2** decimation-in-time (**DIT**) FFT is the simplest and most common form of the Cooley–Tukey algorithm. Radix-2 DIT divides a DFT of size N into two interleaved DFTs (hence the name "radix-2") of size N/2 with each recursive stage.
- Given DFT equation:

$$X_k = \sum_{n=0}^{N-1} x_n e^{-\frac{2\pi i}{N} nk},$$

$$X_k = \underbrace{\sum_{m=0}^{N/2-1} x_{2m} e^{-\frac{2\pi i}{N/2} mk}}_{\text{DFT of even-indexed part of } x_n} + e^{-\frac{2\pi i}{N} k} \underbrace{\sum_{m=0}^{N/2-1} x_{2m+1} e^{-\frac{2\pi i}{N/2} mk}}_{\text{DFT of odd-indexed part of } x_n} = E_k + e^{-\frac{2\pi i}{N} k} O_k.$$




```
X0,...,N-1 ← ditfft2(X, N, s):          DFT of (X0, Xs, X2s, ..., X(N-1)s):
    if N = 1 then
        X0 ← X0                                  trivial size-1 DFT base case
    else
        X0,...,N/2-1 ← ditfft2(X, N/2, 2s)       DFT of (X0, X2s, X4s, ...)
        XN/2,...,N-1 ← ditfft2(X+s, N/2, 2s)     DFT of (Xs, Xs+2s, Xs+4s, ...)
        for k = 0 to N/2-1                       combine DFTs of two halves into full DFT:
            t ← Xk
            Xk ← t + exp(-2πi k/N) Xk+N/2
            Xk+N/2 ← t - exp(-2πi k/N) Xk+N/2
        endfor
    endif
```

Serial implementation

In the serial implementation of radix 2 DIT Cooley Tuckey algorithm.
At first a complex<double> data-type was used but for optimization we performed further analysis of this serial code using gprof profiling tool, we observed more 50% of the time spent on complex<double> data-type . The complex<double>data-type was then converted to double 2D array and the computation time was reduced significantly with upto 2X speed up.

```
void fft2 (double * X, long int N) {
   if(N < 2) {
}
      else {
      separate(X,N);      // all evens to lower half, all odds to upper half
      fft2(X,     N/2);   // recurse even items
      fft2(X+N, N/2);   // recurse odd  items
   //#pragma omp parallel for shared(X,N)
      for(int k=0; k<N; k+=2) {
         double e[2] = {X[k],X[k+1]}; // even,
         double o[2] = {X[k+N],X[k+N+1]};   // odd

         double w[2] = {cos(-1.*M_PI*k/N),sin(-1.*M_PI*k/N)};
         X[k]   = e[0] + w[0]*o[0] - w[1]*o[1];
         X[k+1] = e[1] + w[0]*o[1] + w[1]*o[0];
         X[k+N] = e[0] - w[0]*o[0] + w[1]*o[1];
         X[k+N+1] = e[1] - w[0]*o[1] - w[1]*o[0];
      }
   }
}
```

% time	calls	name
15.66	1245184	std::complex<double>::__rep() const
15.66	1114112	std::complex<double> std::exp<double>(std::complex<double> const&)
12.53	2226224	std::complex<double>& std::complex<double>::operator*=<double>(std::complex<double> const&)
12.53	1114112	std::__complex_exp(doublecomplex)
6.26	4456448	std::complex<double>::imag() const
6.26	4456448	std::complex<double>::real() const
6.26	2226224	std::complex<double> std::operator*<double>(std::complex<double> const&, std::complex<double> const&)
6.26	1114112	std::complex<double> std::operator-<double>(std::complex<double> const&, std::complex<double> const&)
6.26	1114112	std::complex<double> std::operator+<double>(std::complex<double> const&, std::complex<double> const&)
6.26	131071	separate(std::complex<double>*
6.26	1	fft2(std::complex<double>*

Output log when complex<double> used

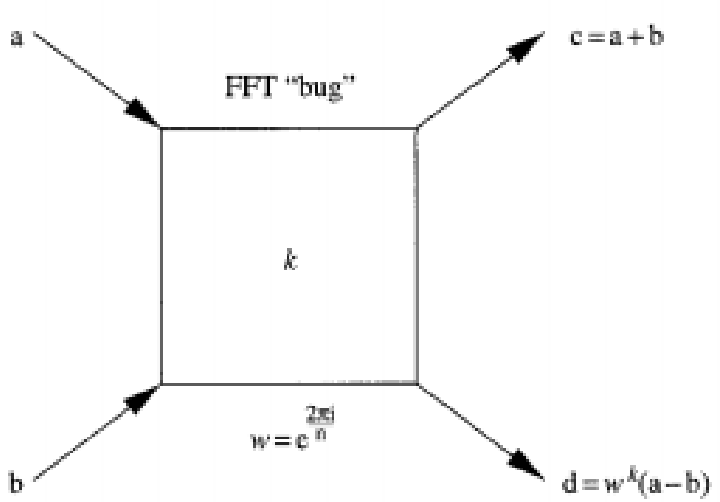
%time	calls	name
100.37	1	fft2(double*, long)
0	131071	separate(double*, long)

Output log when double 2-D array used

However due recursive nature of its implementation no speed up was observed when parallelized using Omp and time of execution was actually increased. Thus this method was not used.

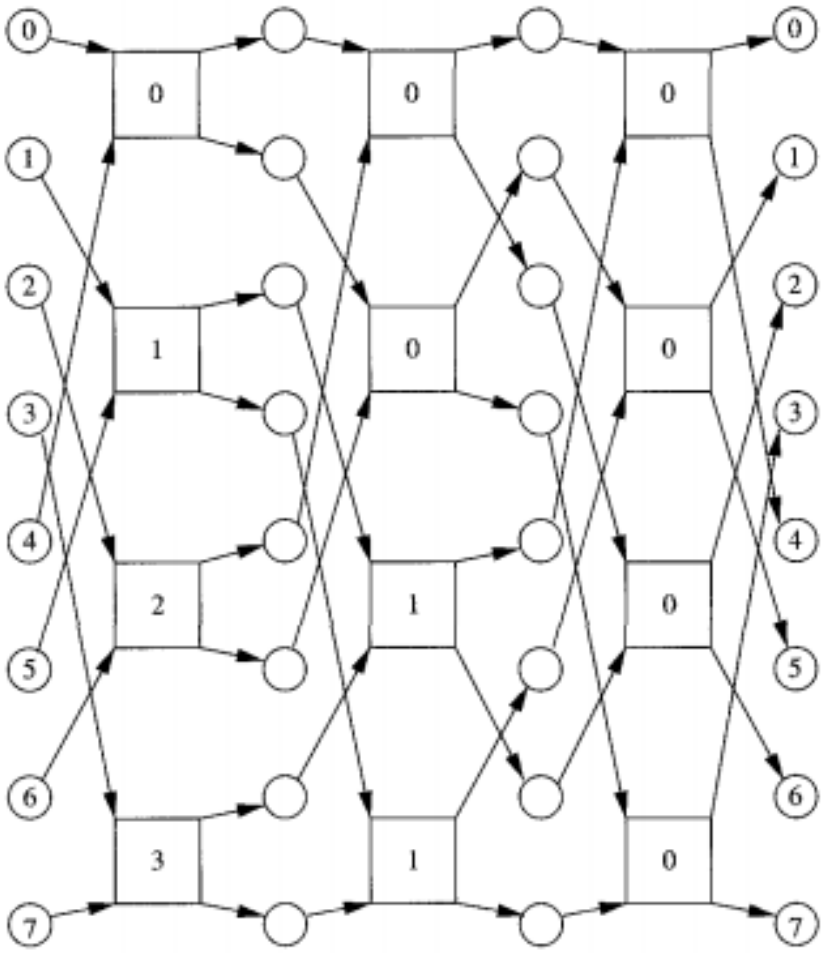
OpenMP implementation

- As it was not possible to implement parallelization on recursive algorithm we took a new approach as discussed in the book ‘Introduction to Parallel Computing’, W. P. Petersen
- It is the workspace version of self sorting FFT which simply toggles back and forth between the input (x) and the workspace/output (y)



- Each "bug" computes: c = a + b and $d=w^k (a-b)$, where w = the n^{th} root of unity
- Output from the bugs will write into areas where the next elements are read - a dependency situation will occur if the output array is the same as the input

OpenMP implementation



signal flow diagram for n = 8

Overview of algorithm

- set up sine and cosine tables needed for the FFT calculation

  w[i*2+0] = cos (2*pi*i/N)
  w[i*2+1] = sin (2*pi*i/N)

- Run the fft function
  - mj=1
  - for j=1:m-2 where $N = 2^m$
    - Mj*=2
    - if toggle = 1 run step function with x as input and reset toggle else vice versa

- Step function: carries out one step of the workspace version of CFFT2
  - For j=0:n/2*mj
    - For k=0:mj
    - Perform one bug operation with proper memory address

CUDA IMPLEMENTATION

- The CUDA Implementation involved building the following kernels

➢ Data Generation

```
__global__ void datagen (cuDoubleComplex* x_d, int n_local, int nSamples)
```

- Sample data was generated by **summing up the sinusoids** of **known frequency** to test the ouput of computed fourier transform
- For generation of large sample data the task of generating one sample point was assigned to one thread

```
for(int i=0; i<n_local; i++) {
    x_d[idx+i] = make_cuDoubleComplex(0,0);
    // sum several known sinusoids into x[]
    for(int j=0; j<nFreqs; j++)
        x_d[i+idx] = cuCadd(x_d[i+idx],
            make_cuDoubleComplex(sin( 2*M_PI*(float)(freq[j])
```

➢ Fast Fourier Transform Computation

```
__global__ void fft (cuDoubleComplex* X_d, int local_n, int numBlock, int block_size, cuDoubleComplex* b)
```

- We used the **cuDoubleComplex** object from the **<cuComplex.h >** header file for data-storage and to carry out arithmetic operations on the device while using the **complex<double>** object on the host
- The serial algorithm involved the usage of recursive calls for FFT computations and to implement recursion on CUDA required the knowledge and understanding of **Dynamic Parallelism**

```
separate(X_d,local_n,b);      // all evens to lower h
__syncthreads();
fft <<< 1, 1 >>> (X_d, local_n/2, 1, 1, b);   // recu
__syncthreads();
fft <<< 1, 1 >>> (X_d+local_n/2, local_n/2, 1, 1, b);
__syncthreads();
```

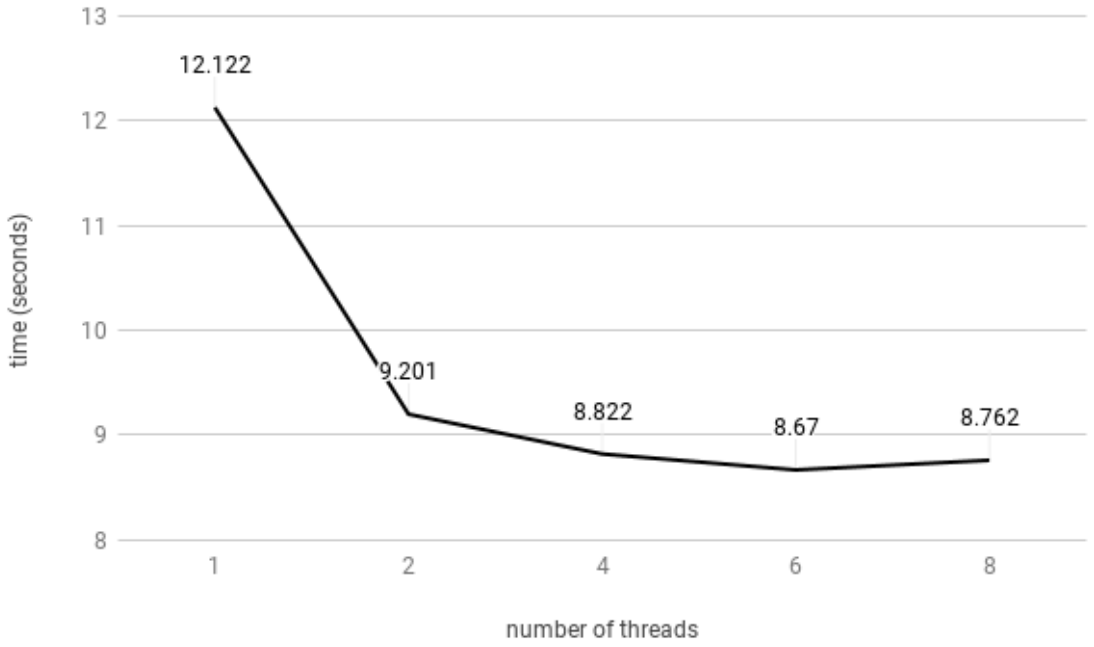
The idea behind the CUDA implementation was to **divide** the larger problem of **size N** into **smaller m-sized** problems for each thread and then combine the results of each thread into one complex<double> array for the final FFT answer

HURDLES

1. The biggest problem that we faced during the implementation was that of **Dynamic Parallelism.** CUDA codes based on Dynamic Parallelism require a **compute_35** or higher architecture.
   - The RADIX-2 DIT algorithm involves recursion and to implement that in CUDA require **nested kernel calls** which is called Dynamic Parallelism. The nested calls required that all the threads **finish updating the FFT array** before re-accessing them in recursion and to do that we had to use **__syncthreads()** at several locations which reduced the speed of the algorithm
2. The next problem was that of **object compatibility** between device and host objects
   - CUDA prevents the use of **complex<double>** objects inside the kernels(i.e on the device) so we first used the **thrust::complex<double>** object inside kernels which gave object incompatibility with **std::complex<double>** while using cudaMemcpy()
   - To solve this we then used the **<cuComplex.h>** header file and initialized kernel objects as **cuDoubleComplex** which is binary compatible with **std::complex<double>**
3. The CUDA FFT output was very inconsistent with the serialized algorithm output due to memory access issues and repercussions of Dynamic Parallelism so a different algorithmic approach is required for FFT implementation on CUDA

Conclusions:

- Graphical comparison of time taken versus number of threads was obtained as follows:



- Speed-up achieved is about six times more than serial implementation
- Parallelization of FFT is useful for only sample points greater than 2^{20}. If number of points are less than that the recursive algorithm is optimal.
- Parallelization using cuda is complex in recursive case as dynamic parallelization is difficult to implement and also it slower the code to high extent.