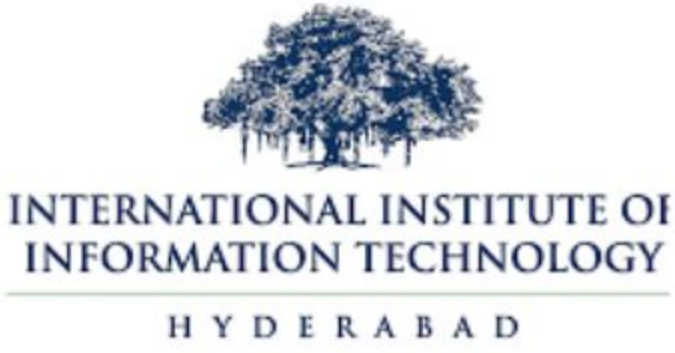


Introduction to Parallel Scientific Computing

Parallel Implementation of Fast Fourier Transform (CUDA)

Submitted By:	*Roll No.:*
Aman Joshi	2018201097
Praveen Balireddy	2018201052
Arpit Gupta	2018201048

INTRODUCTION

The aim of the project was to provide a parallel implementation of *Fast Fourier Transform (FFT)* method. FFT is a widely used method for various purposes. In our project we have implemented two uses of FFT. They are -

1. Multiplication of two polynomials
2. Image compression

Multiplication of polynomials usually takes O(n*m) where one polynomial has n and other has m terms. This can be reduced to O(n*log(m)) by using FFT.

One of the other uses of FFT include image compression. Fourier techniques allow us to decompose an image in the frequency domain, remove or alter particular frequencies and then reconstruct the image. This is beneficial for image compression as by removing certain frequencies, the compressed image contains less information.

Performance for both the implementations ,i.e., parallel and sequential have been visualized by plotting a graph between *time taken* and *number of threads*. We have exploited parallelism over 1D FFT function calls. This also parallelised 2D FFT methods.

FAST FOURIER TRANSFORM (FFT)

A **fast Fourier transform** (**FFT**) is an algorithm that computes the discrete Fourier transform (DFT) of a sequence, or its inverse (IDFT). Fourier analysis converts a signal from its original domain (often time or space) to a representation in the frequency domain and vice versa.

Let $x_0, \ldots, x_{N-1}$ be complex numbers. The DFT is defined by the formula

$$X_k = \sum_{n=0}^{N-1} x_n e^{-i 2\pi k n / N} \qquad k = 0, \ldots, N-1,$$

where $e^{i2\pi/N}$ is a primitive Nth root of 1.

Computing DFT directly from the above definition is often too slow. Therefore, FFT is used to do these calculation in a faster way. 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 from $O(n^2)$, which arises if one simply applies the definition of DFT, to O(n*log(n)) where n is the data size.

There are many FFT algorithms used for different problems. We have used *Cooley-Tukey* algorithm.

Cooley-Tukey Algorithm

This is a divide and conquer algorithm that recursively breaks down a DFT of any composite size $N = N_1 N_2$ into many smaller DFTs of sizes N_1 and N_2. The best known use of the Cooley–Tukey algorithm is to divide the transform into two pieces of size $N/2$ at each step, and is therefore limited to power-of-two sizes, but any factorization can be used in general.

Following images show our implementation of Cooley-Tukey algorithm.

```
// Performing Bit reversal ordering
int n = (int)a.size();

for (int i = 1, j = 0; i < n; ++i)
{
    int bit = n >> 1;
    for (; j >= bit; bit >>= 1)
        j -= bit;
    j += bit;
    if (i < j)
        swap(a[i], a[j]);
}
```

Fig.1 - Inplace bit reversal

For Sequential code we have used the above algorithm, It performs bit reversal in O(n logn) but constant space. However, for parallel code we have done it in O(n) time and O(n) extra space. However this O(n) is distributed among threads.

```
// Iteratinve FFT
// This part of FFT is parallelizable
for (int len = 2; len <= n; len <<= 1)
{
    double ang = 2 * M_PI / len * (invert ? -1 : 1);
    base wlen(cos(ang), sin(ang));
    for (int i = 0; i < n; i += len)
    {
        base w(1);
        for (int j = 0; j < len / 2; ++j)
        {
            base u = a[i + j], v = a[i + j + len / 2] * w;
            a[i + j] = u + v;
            a[i + j + len / 2] = u - v;
            w *= wlen;
        }
    }
}
```

Fig.2 - Iterative FFT

RESULTS

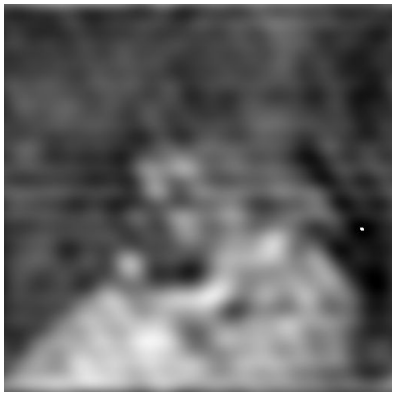
Following results show compression done on an image at various threshold level.

3(a)
(0.000001, 61.5 kB)

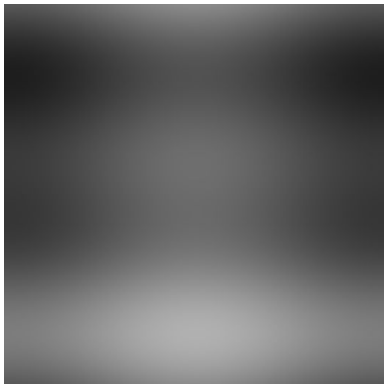
3(b)
(0.000010, 61.5 kB)

3(c)
(0.000100, 70.7 kB)

3(d)
(0.001000, 52.1 kB)



3(e)
(0.010000, 27.5 kB)



3(f)
(0.100000, 16.5 kB)

Fig.3 - Compressed image at various threshold level showing size of the compressed image. Format - (Threshold factor, size)

Sequential and parallel implementations of FFT has been compared based on the execution time of the program for different size of data (N).

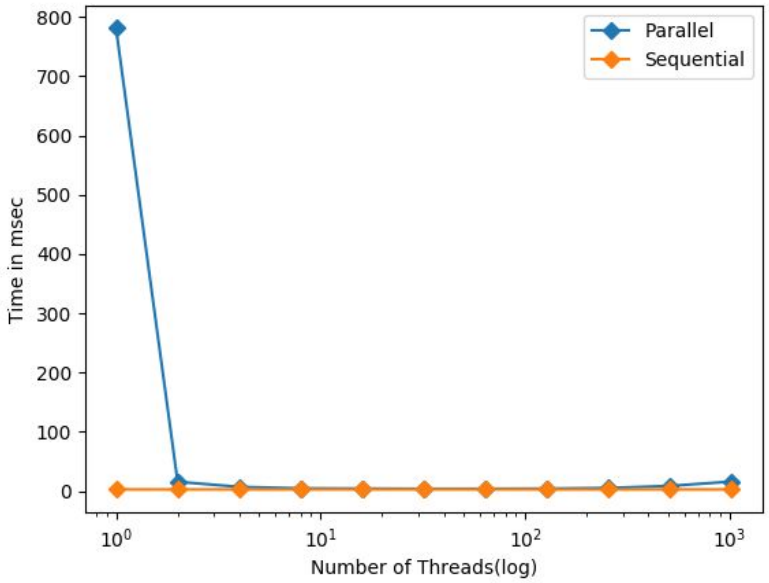


Fig.4(a) - Time taken by parallel and sequential implementations for different threads

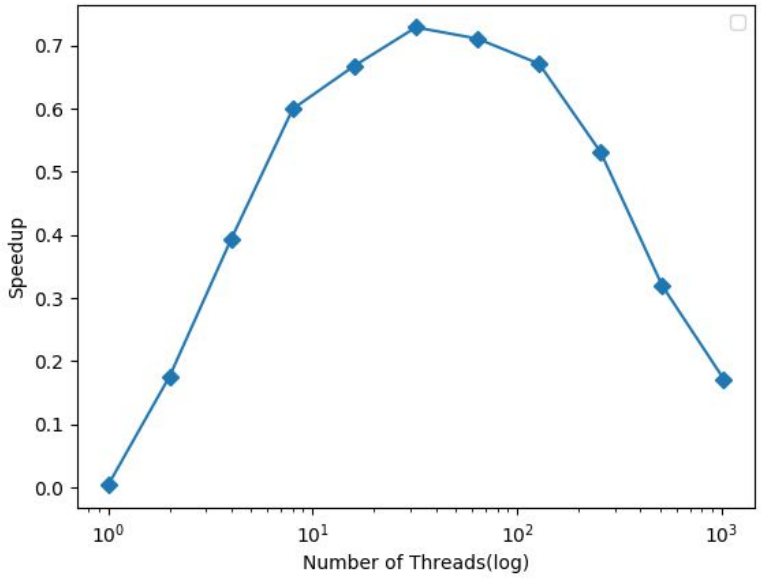


Fig.4(b) - Speedup of parallel implementation as compared to sequential

Fig.4 - Visualization for N = 1000

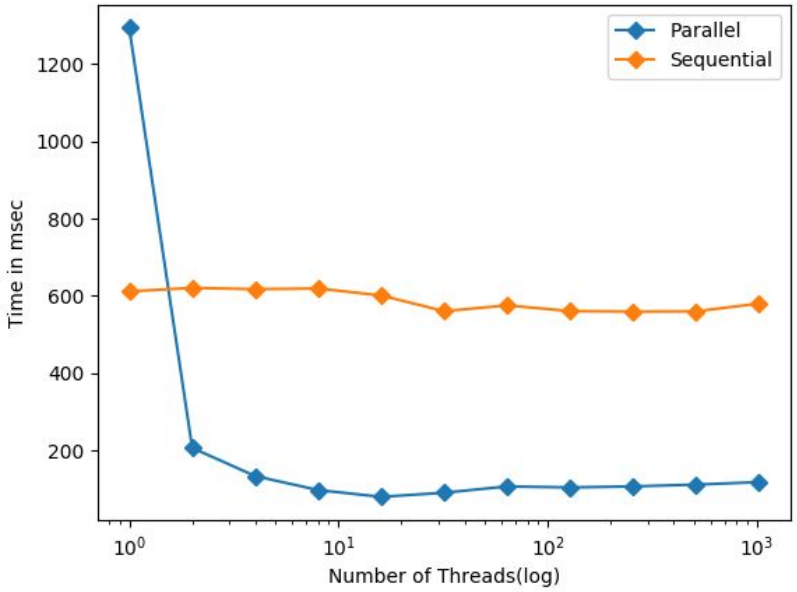


Fig.5(a) - Time taken by parallel and sequential implementations for different threads

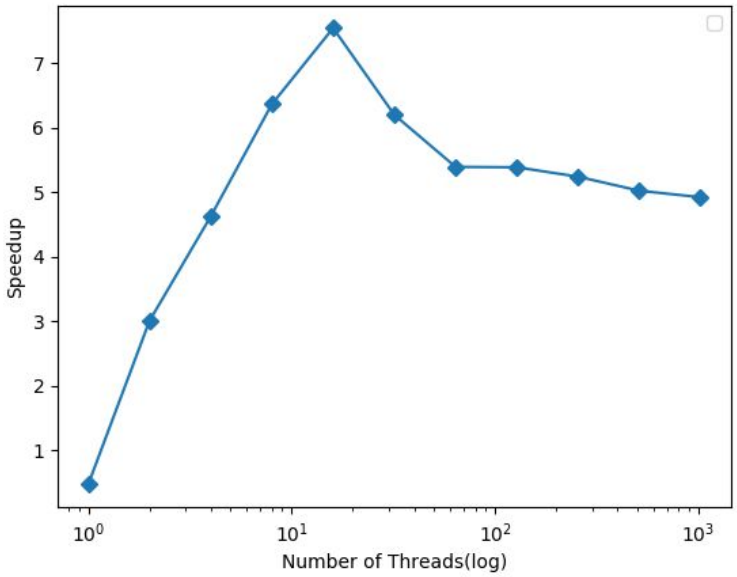


Fig.5(b) - Speedup of parallel implementation as compared to sequential

Fig.5 - Visualization for N = 100000

CONCLUSIONS

- As we are increasing threshold factor for compression of images, we are losing more and more information as lower frequency components of the images are filtered out. Thus, image is becoming more blur.
- For lower number of threads, parallel implementation takes more time due to overhead involved in creation of threads. This can be observed from the fact that speedup is less than 1. But, as number of threads are increased expected behaviour is observed as parallel implementation executes faster than sequential.
- For small value of N, difference in execution time is not that significant (as shown in Fig.4(a)).

REFERENCES

- Link to youtube video referred for image compression
- Link to wikipedia article referred for Cooley-Tukey algorithm
- Link to article on how to perform 2D-FFT using 1D-FFT
- Link for article on JPEG compression of images using FFT
- Link for polynomial multiplication using FFT
- Link for CUDA basics