Two-Dimensional Discrete Fourier Transform on CPU and GPU

Pritish Nahar, Vinod Lasrado, Akshaya Nagarajan, and Pranav Ghaisas

***Abstract*—This Project explores three variations of 2D-DFT implementation, viz. using CPU threads, MPI and on GPUs using CUDA. The first multi-threaded CPU implementation involves the conventional DFT computation using the built-in features of C++11 threads. The second variation involves implementation of the Cooley-Tukey algorithm [2] using MPI. The third variation involves implementation of the conventional DFT algorithm on GPUs using CUDA. In addition to this, the project also covers Inverse DFT implementations for all the above-mentioned variations. Timing and Performance Analysis tests for all the above variations were conducted to determine the fastest implementation.**

***Index Terms*—Threads, CUDA, MPI, DFT, FFT, Cooley-Tuckey.**

I. Implementation

In this section, the approaches to implement the said algorithms using C++ Threads, MPI and CUDA will be discussed.

A. CPU Implementation using C++ 11 Threads

In this CPU implementation using threads, firstly the input file is read to determine the width and height of the matrix. The DFT computation of the rows and columns of the input file is distributed amongst 8 threads. If there are N number of threads, each thread is given **Height/N** rows and **Width/N** columns to work on. The row-wise and column-wise operations are performed by the **computeRow()** and **computeColumn()** functions respectively. The threads first perform the row transformation by calling the **computeRow()** function, and the output of this function is fed as input to the **computeColumn()** function to perform the column transformation. The output of **computeColumn()** function provides the final DFT result. The threads are synchronized after both the row and column transformation operations.

Similarly, the **Inverse DFT** is computed by calling the **computeRowIFT()** and **computeColumnIFT()** functions. The process of computation for IDFT follows the same steps as for the DFT computation.

B. CPU Implementation using MPI

In this CPU implementation using MPI, the *Cooley-Tukey* Algorithm was implemented. The **Cooley-Tukey Algorithm** states that FFT of an ***N*** length array is the summation of two smaller FFTs of length ***N/2***, where each individual FFT consists of only the even and odd numbered samples respectively, thereby reducing the number of operations (time complexity) to be performed [2].

The process with rank 0 reads the input file to determine width and height of the matrix. This process then sends width and height information of the matrix to the other processes. Each process using MPI calculates the shard size, i.e. the **Number of Rows** it needs to handle, to calculate the DFT. The load is divided in a way that the **Number of Rows** each process handles is equal to either $[Rows/(NumberofProcesses)]$ or $[Rows/(NumberofProcesses) + 1]$.

Process 0 also calculates the shard size for all the processes and sends them the rows they are responsible for, to compute the FFT. Each process then computes Fast Fourier Transform by calling the **computeFFTForMPI()** function for all the rows (using Cooley-Tukey Algorithm) it is responsible for. All processes post the FFT computation, send results to process 0, which collates the results into a single matrix. This matrix is then transposed and the row elements are sent again to other processes. Again, each process computes the FFT for the rows it is responsible for, and sends results to the process with rank 0. This process again collates the results into a matrix and transposes it, which is the FFT of the input 2D matrix.

The **Inverse DFT** computation has a load split and steps similar to the FFT computation, and the processes call **computeIFTForMPI()** function for computing the Inverse DFT.

C. GPU Implementation using CUDA

In this GPU implementation using CUDA, firstly the input file is read to determine the width and height of the matrix. Each CUDA thread is assigned an individual element of the matrix to operate on. The row-wise and column-wise operations are performed by the **computeRow()** and **computeColumn()** functions respectively. The CUDA threads first perform the row transformation by calling the **computeRow()** kernel function, and the output of this function is fed as input to the **computeColumn()** kernel function to perform the column transformation. The output of **computeColumn()** kernel function provides the final DFT result. The CUDA threads are synchronized by calling the **cudaDeviceSynchronize()** function after both the row and column transformation operations.

The **Inverse DFT** is computed by calling the **computeRowInv()** and **computeColumnInv()** functions. The process of

Department of Electrical and Computer Engineering, Georgia Institute of Technology, Atlanta, GA, 30332 USA

computation for IDFT follows the same steps as for the DFT computation.

II. Performance Measurements

All the algorithms were tested on a machine with the following specs:

```
[pnahar7@rich133-k40-17 p3]$ lscpu
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                8
On-line CPU(s) list:   0-7
Thread(s) per core:    1
Core(s) per socket:    4
Socket(s):             2
NUMA node(s):          2
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 79
Stepping:              1
CPU MHz:               2600.000
BogoMIPS:              5199.25
Virtualization:        VT-x
L1d cache:             32K
L1i cache:             32K
L2 cache:              256K
L3 cache:              10240K
NUMA node0 CPU(s):     0-3
NUMA node1 CPU(s):     4-7
[pnahar7@rich133-k40-17 p3]$
```

Fig. 1. Specifications of the GPU node

(Command used to enter the GPU node - qsub -I -q coc-ice -l nodes=1:ppn=8:gpus=1,walltime=2:00:00,pmem=2gb)

A. Table of Execution Times Computed

The following table represents the time in seconds required for execution(including output file write) of all the inputs matrix files (4x4, 8x8, 128x128, 512x512, 1024x1024, 2048x2048) tested using CPU threads, MPI and CUDA.

TABLE I
Execution Times in Seconds

Method	4	8	128	256	512	1024	2048
Threads	0.005	0.005	0.097	0.90	8.05	69.44	718.85
MPI	0.68	0.73	0.70	0.81	0.85	2.48	3.26
CUDA	0.25	0.24	0.25	0.34	0.62	1.96	3.89

B. Graph Analysis

The following graphs were plotted based on the trends observed in the above table. However, it is difficult to make an accurate judgment regarding the behavior of the graphs due to a small sample size.

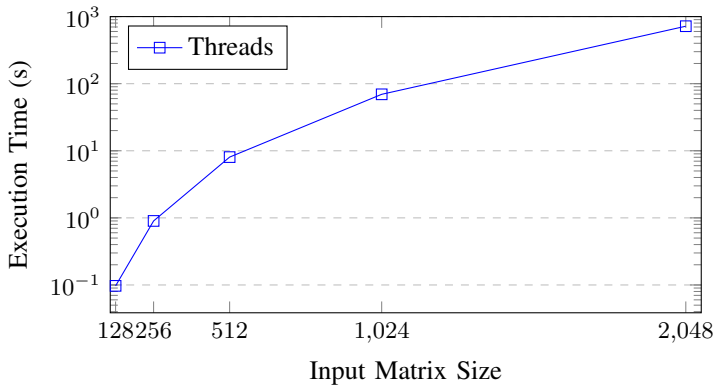

Fig. 2. Performance of Our C++ Threads implementation for different Input Matrices

Observation for Graph 1: The algorithm execution time for C++ threads seems to increase exponentially as size of the Input Matrix is increased. The time required for the matrix of size 2048*2048 is much higher than that for all other matrices.

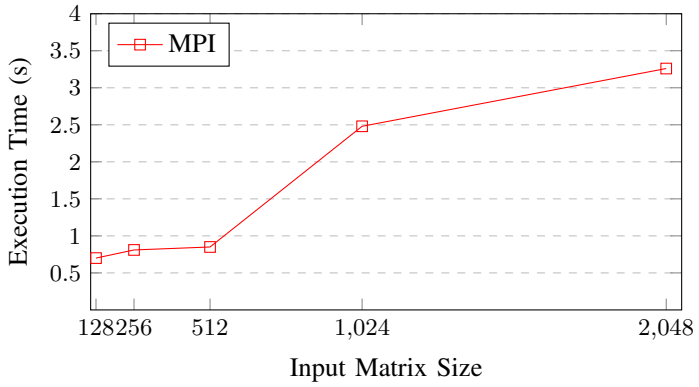

Fig. 3. Performance of Our MPI Implementation for different Input Matrices

Observation for Graph 2: The algorithm execution time for MPI increases for increased size of the Input Matrix.

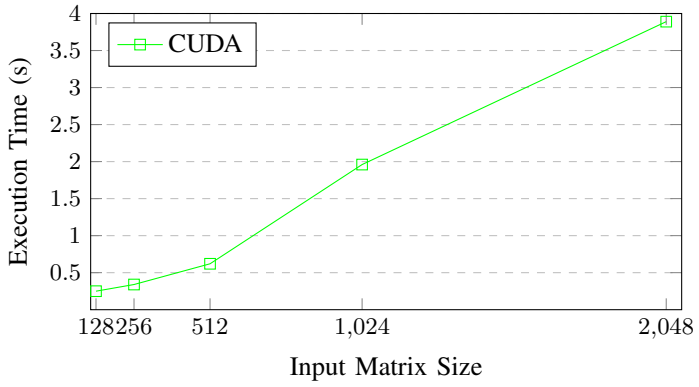

Fig. 4. Performance of Our CUDA Implementation for different Input Matrices

Observation for Graph 3: The algorithm execution time for CUDA increases for increased size of the Input Matrix.

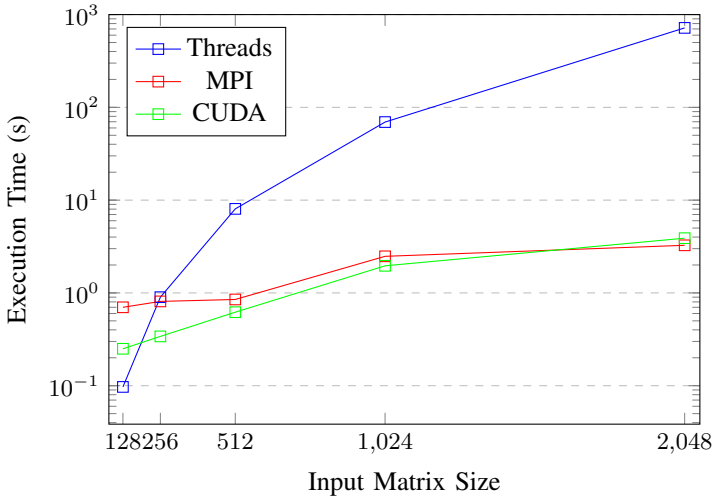

Fig. 5. Performance of all Our Implementations for different Input Matrices

Observation for Graph 4: The algorithm execution time for Threads, MPI and CUDA are around the same for smaller input matrices, but increases rapidly for Threads for huge sizes of the Input Matrix like 1024x1024 and beyond.

III. Conclusion

In this project we implemented Discrete Fourier Transform on CPU using C++ Threads, on GPU using CUDA, and Fast Fourier Transform on CPU using MPI. We also implemented the Inverse Fourier Transform, for all the three variations. Our implementation was evaluated for symmetric matrices of size 4x4, 8x8, 128x128, 256x256, 512x512, 1024x1024 and 2048x2048. The execution time increases with an increase in the size of the matrix. **CUDA shows the best performance** because to it's ability to work independently on every element in the matrix, and the ability to simultaneously deploy a huge number of threads. The other frameworks cannot support such a huge number of parallel processes or threads, and hence have to work on multiple rows serially, reducing the scope for **parallelization**. The Results for input file 2048x2048 are off for CUDA and C++ threads implementation. We believe that this is due to the error caused by **precision loss** during cosine and sine computations which get compounded, since there are 2048 additions for every element in a row/column transform.

References

[1] *http ://mathworld.wolfram.com/Danielson-LanczosLemma.html*
[2] *https ://en.wikipedia.org/wiki/Cooley-Tukey FFT algorithm*

IV. Additional Notes

We have a **single CMake file** for this running this project. Modules loaded to test the project:

- gcc/4.9.0
- openmpi/1.8
- cuda/9.1
- cmake/3.9.1

Refer **README.txt** for additional details regarding execution.