Performance Analysis of DFT implementations on GPUs

Rishav Karki, PID: A59018992 , ECE 277 Final Project

Outline

Discrete Fourier Transform

- DFT transforms discrete time signals from the time domain to the frequency domain.
- Decomposes signals into constituent frequencies, revealing amplitude and phase information.
- Used in spectrum analysis and linear filtering of signals.
- Applications in audio processing, image processing, and telecommunications.

$$
\begin{aligned}
X_k &= \sum_{n=0}^{N-1} x_n \cdot e^{-\frac{i2\pi}{N}kn} \\
&= \sum_{n=0}^{N-1} x_n \cdot \left[\cos\left(\frac{2\pi}{N}kn\right) - i \cdot \sin\left(\frac{2\pi}{N}kn\right)\right]
\end{aligned}
$$

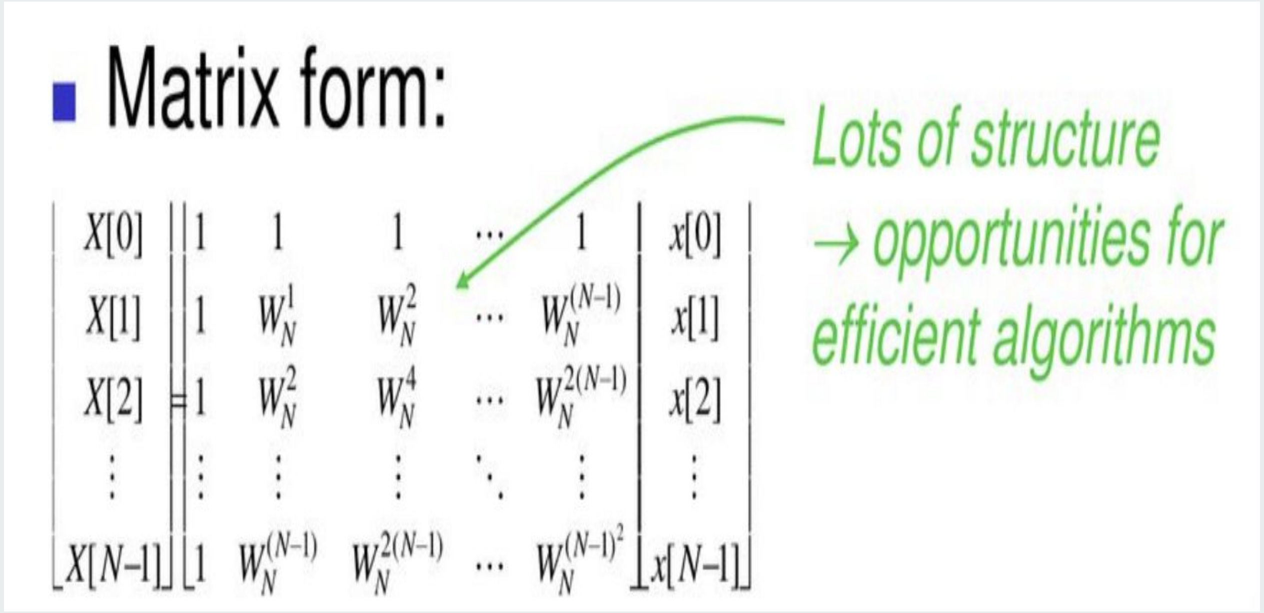
Discrete Fourier Transform: Algorithm

- Steps:
  - Initialize the input data vector and the kernel matrix with float2 variables, where ‘x’ represents the Real part of the complex number and ‘y’ represents the imaginary part. The kernel matrix contains the twiddle factors.
  - Multiply kernel matrix with input data to obtain the output DFT vector.

The step highlighted in red can be parallelized.

Discrete Fourier Transform: Matrix Vector Multiplication

- This step can be parallelized on a GPU using multiple threads, as we will see later in the CUDA kernel.



- Matrix form:

$$\begin{bmatrix} X[0] \\ X[1] \\ X[2] \\ \vdots \\ X[N-1] \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & W_N^1 & W_N^2 & \cdots & W_N^{(N-1)} \\ 1 & W_N^2 & W_N^4 & \cdots & W_N^{2(N-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & W_N^{(N-1)} & W_N^{2(N-1)} & \cdots & W_N^{(N-1)^2} \end{bmatrix} \begin{bmatrix} x[0] \\ x[1] \\ x[2] \\ \vdots \\ x[N-1] \end{bmatrix}$$

Lots of structure $\rightarrow$ opportunities for efficient algorithms

Fast Fourier Transform

- FFT is an efficient method for computing the Discrete Fourier Transform (DFT) of a sequence or signal.
- We will be looking at the Cooley Tukey FFT algorithm.
- Divides the DFT computation into smaller subproblems.
- Achieves a much lower time complexity of O(N log N) compared to the straightforward O(N^2) approach for DFT.

.

Fast Fourier Transform : Algorithm

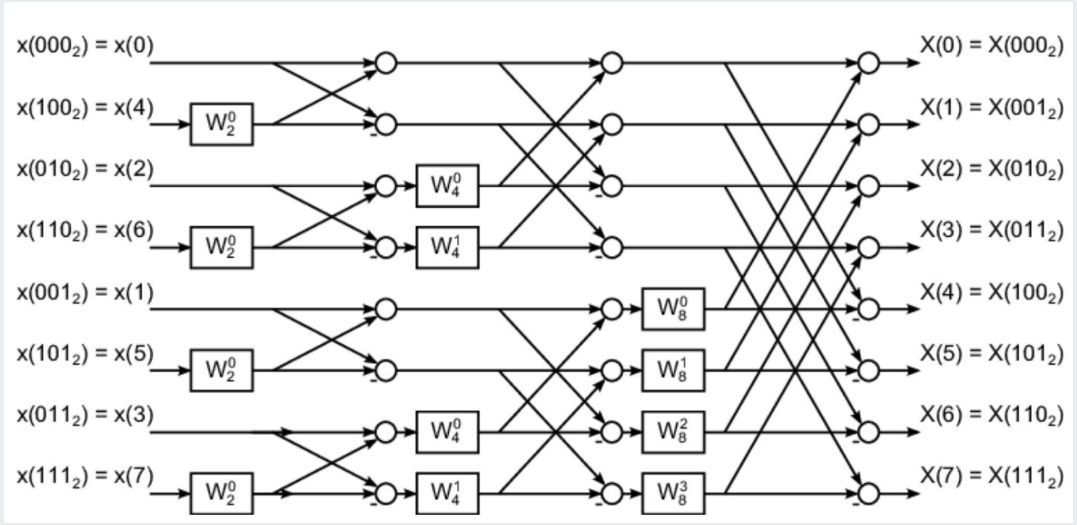
- Decimation in time : Express the N point DFT as a sum of two N/2 point DFTs.

$$
\begin{aligned}
X(k) &= \sum_{n=0}^{N-1} x(n)\, W_N^{kn}, \qquad k = 0,1,\ldots, N-1 \\
&= \sum_{n \text{ even}} x(n)\, W_N^{kn} + \sum_{n \text{ odd}} x(n)\, W_N^{kn} \\
&= \sum_{m=0}^{(N/2)-1} x(2m)\, W_N^{2mk} + \sum_{m=0}^{(N/2)-1} x(2m+1)\, W_N^{k(2m+1)}
\end{aligned}
$$

$$
\begin{aligned}
X(k) &= \sum_{m=0}^{(N/2)-1} f_1(m)\, W_{N/2}^{km} + W_N^k \sum_{m=0}^{(N/2)-1} f_2(m)\, W_{N/2}^{km} \\
&= F_1(k) + W_N^k F_2(k), \qquad k = 0,1,\ldots, N-1
\end{aligned}
$$

Fast Fourier Transform : Cooley Tukey Algorithm

- Divide and conquer: Complex FFT butterflies

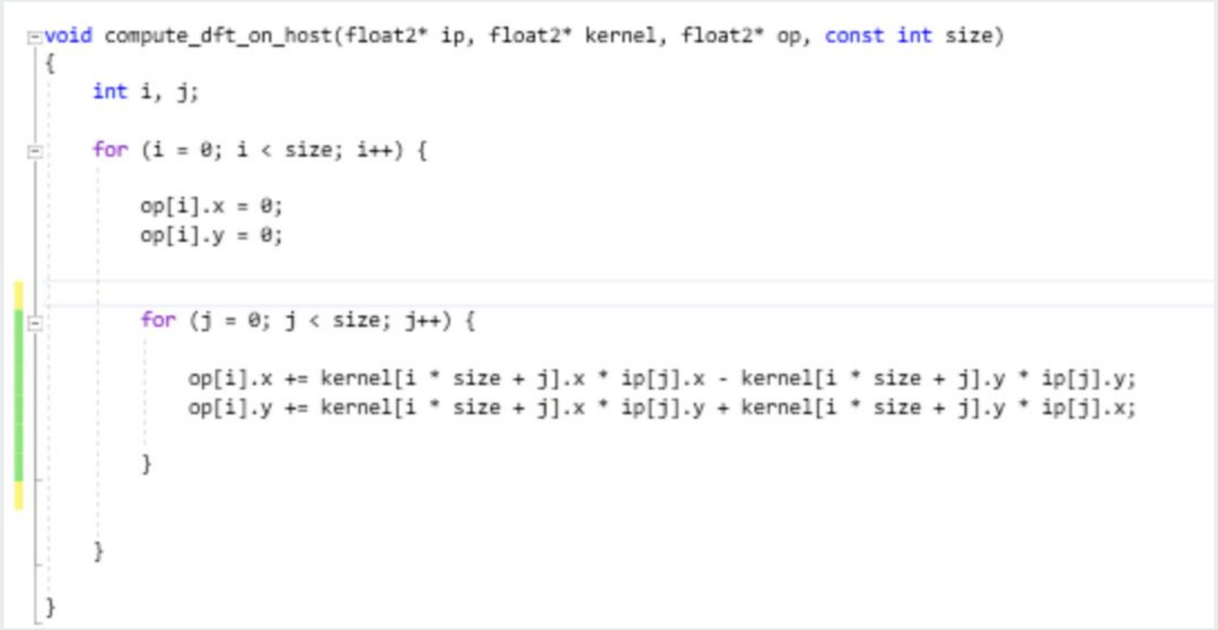


Fast Fourier Transform : Algorithm

- Steps:
  - Initialize input data vector.
  - Rearrange the input data vector such that the bit reversed addresses are contiguous.
  - Run a for loop for log N number of iterations.
    - Compute the twiddle factors at stage 'm', which correspond to 2^m point FFT.
    - Access pairs of input data at stage 'm' , where the elements of the pair are separated by a stride of 'm/2' addresses.
    - Multiply the second element of each pair with the twiddle factor to rotate the phase.
    - Add and subtract the phase rotated second element of each pair from the first element to generate two outputs per pair of inputs.

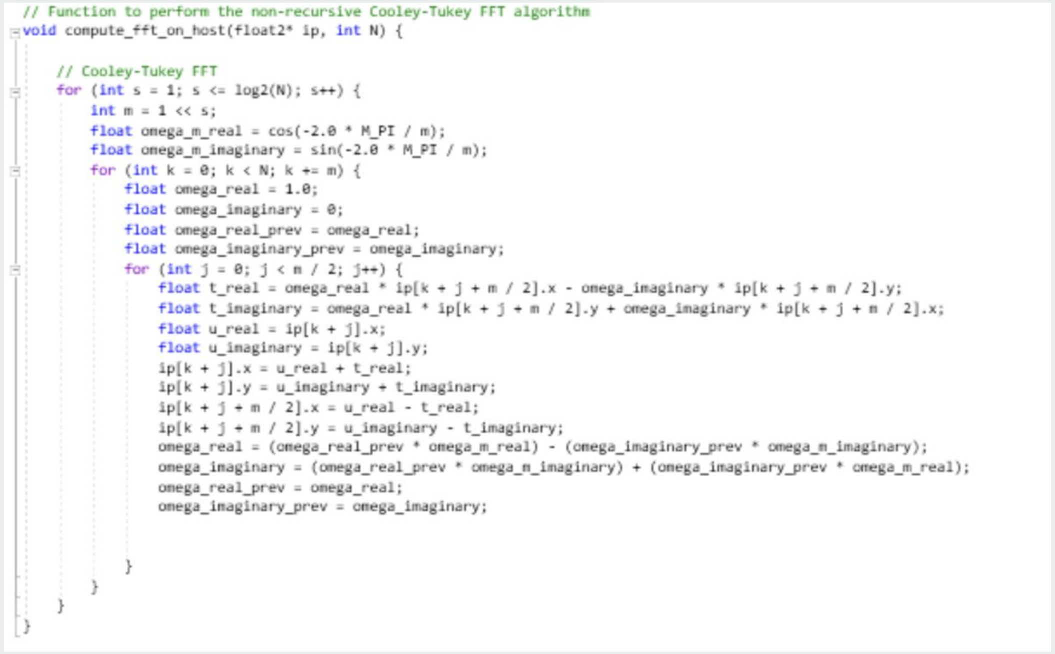
The steps highlighted in red can be parallelized, as we will see later in the CUDA kernel.

DFT on host



```
void compute_dft_on_host(float2* ip, float2* kernel, float2* op, const int size)
{
    int i, j;

    for (i = 0; i < size; i++) {

        op[i].x = 0;
        op[i].y = 0;


        for (j = 0; j < size; j++) {

            op[i].x += kernel[i * size + j].x * ip[j].x - kernel[i * size + j].y * ip[j].y;
            op[i].y += kernel[i * size + j].x * ip[j].y + kernel[i * size + j].y * ip[j].x;

        }


    }

}
```

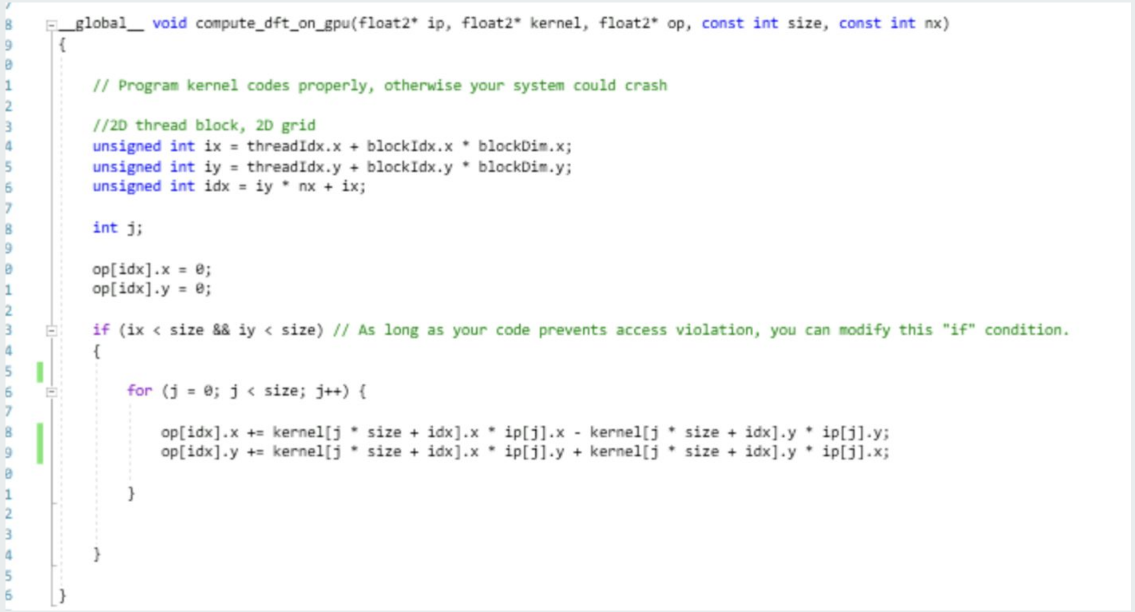
FFT on host



```
// Function to perform the non-recursive Cooley-Tukey FFT algorithm
void compute_fft_on_host(float2* ip, int N) {

    // Cooley-Tukey FFT
    for (int s = 1; s <= log2(N); s++) {
        int m = 1 << s;
        float omega_m_real = cos(-2.0 * M_PI / m);
        float omega_m_imaginary = sin(-2.0 * M_PI / m);
        for (int k = 0; k < N; k += m) {
            float omega_real = 1.0;
            float omega_imaginary = 0;
            float omega_real_prev = omega_real;
            float omega_imaginary_prev = omega_imaginary;
            for (int j = 0; j < m / 2; j++) {
                float t_real = omega_real * ip[k + j + m / 2].x - omega_imaginary * ip[k + j + m / 2].y;
                float t_imaginary = omega_real * ip[k + j + m / 2].y + omega_imaginary * ip[k + j + m / 2].x;
                float u_real = ip[k + j].x;
                float u_imaginary = ip[k + j].y;
                ip[k + j].x = u_real + t_real;
                ip[k + j].y = u_imaginary + t_imaginary;
                ip[k + j + m / 2].x = u_real - t_real;
                ip[k + j + m / 2].y = u_imaginary - t_imaginary;
                omega_real = (omega_real_prev * omega_m_real) - (omega_imaginary_prev * omega_m_imaginary);
                omega_imaginary = (omega_real_prev * omega_m_imaginary) + (omega_imaginary_prev * omega_m_real);
                omega_real_prev = omega_real;
                omega_imaginary_prev = omega_imaginary;


            }
        }
    }
}
```

CUDA kernel : DFT

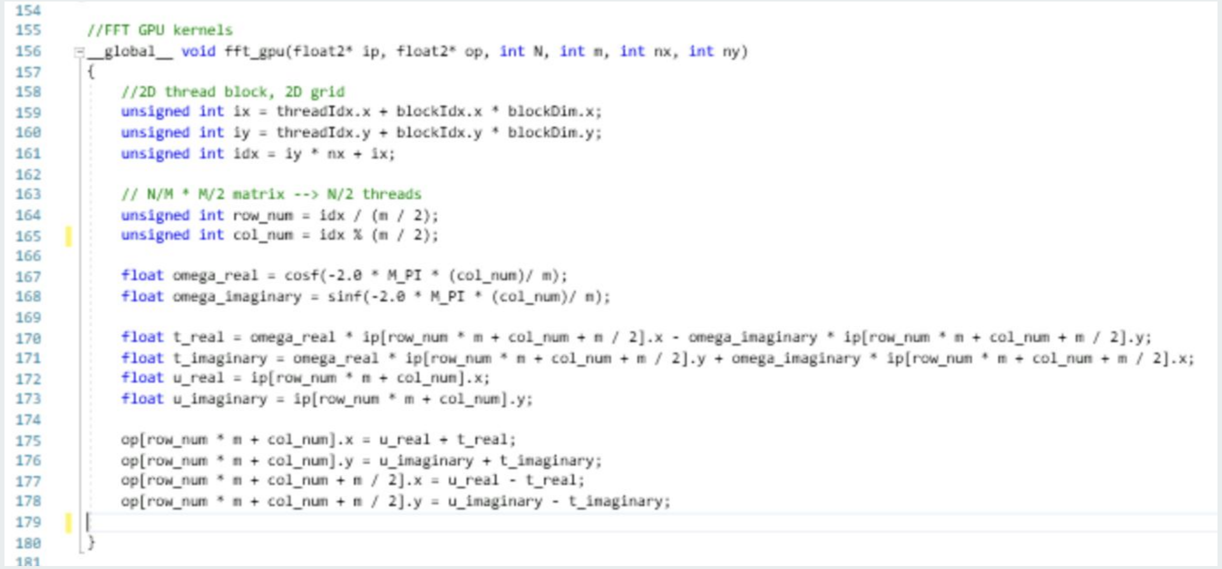
- Kernel is configured as a 2D thread block, 2D grid of threads.
- The DFT GPU kernel is called once for an N point DFT, reducing the time complexity theoretically to O(N), as it generates 1 output per thread, but has a for loop where it goes through N inputs to compute 1 output.
- Aligned and coalesced global memory accesses in every warp ensure memory transactions are reduced. We take advantage of the fact that the kernel matrix is symmetric.



```
__global__ void compute_dft_on_gpu(float2* ip, float2* kernel, float2* op, const int size, const int nx)
{

    // Program kernel codes properly, otherwise your system could crash

    //2D thread block, 2D grid
    unsigned int ix = threadIdx.x + blockIdx.x * blockDim.x;
    unsigned int iy = threadIdx.y + blockIdx.y * blockDim.y;
    unsigned int idx = iy * nx + ix;

    int j;

    op[idx].x = 0;
    op[idx].y = 0;

    if (ix < size && iy < size) // As long as your code prevents access violation, you can modify this "if" condition.
    {

        for (j = 0; j < size; j++) {

            op[idx].x += kernel[j * size + idx].x * ip[j].x - kernel[j * size + idx].y * ip[j].y;
            op[idx].y += kernel[j * size + idx].x * ip[j].y + kernel[j * size + idx].y * ip[j].x;

        }


    }

}
```

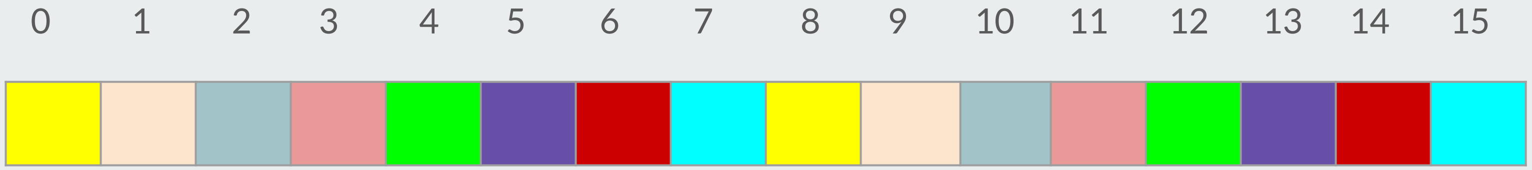
CUDA kernel : FFT

- Kernel is configured as a 2D thread block, 2D grid of threads.
- The FFT GPU kernel is called log N number of times for an N point FFT, reducing the time complexity theoretically to O(log N), as it computes N outputs per stage parallely using N/2 threads.
- Aligned and coalesced global memory accesses in every warp ensure memory transactions are reduced.
- No loop overheads within warp boosts performance.

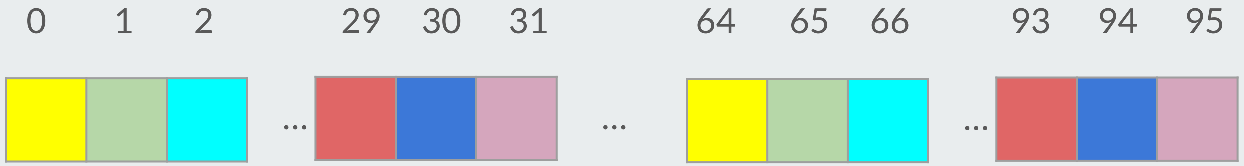


```
//FFT GPU kernels
__global__ void fft_gpu(float2* ip, float2* op, int N, int m, int nx, int ny)
{
    //2D thread block, 2D grid
    unsigned int ix = threadIdx.x + blockIdx.x * blockDim.x;
    unsigned int iy = threadIdx.y + blockIdx.y * blockDim.y;
    unsigned int idx = iy * nx + ix;

    // N/M * M/2 matrix --> N/2 threads
    unsigned int row_num = idx / (m / 2);
    unsigned int col_num = idx % (m / 2);

    float omega_real = cosf(-2.0 * M_PI * (col_num)/ m);
    float omega_imaginary = sinf(-2.0 * M_PI * (col_num)/ m);

    float t_real = omega_real * ip[row_num * m + col_num + m / 2].x - omega_imaginary * ip[row_num * m + col_num + m / 2].y;
    float t_imaginary = omega_real * ip[row_num * m + col_num + m / 2].y + omega_imaginary * ip[row_num * m + col_num + m / 2].x;
    float u_real = ip[row_num * m + col_num].x;
    float u_imaginary = ip[row_num * m + col_num].y;

    op[row_num * m + col_num].x = u_real + t_real;
    op[row_num * m + col_num].y = u_imaginary + t_imaginary;
    op[row_num * m + col_num + m / 2].x = u_real - t_real;
    op[row_num * m + col_num + m / 2].y = u_imaginary - t_imaginary;

}
```

FFT memory access patterns



- Strided memory access in stage 4 of a 16 point FFT
- 8 threads in the warp access contiguous memory locations.

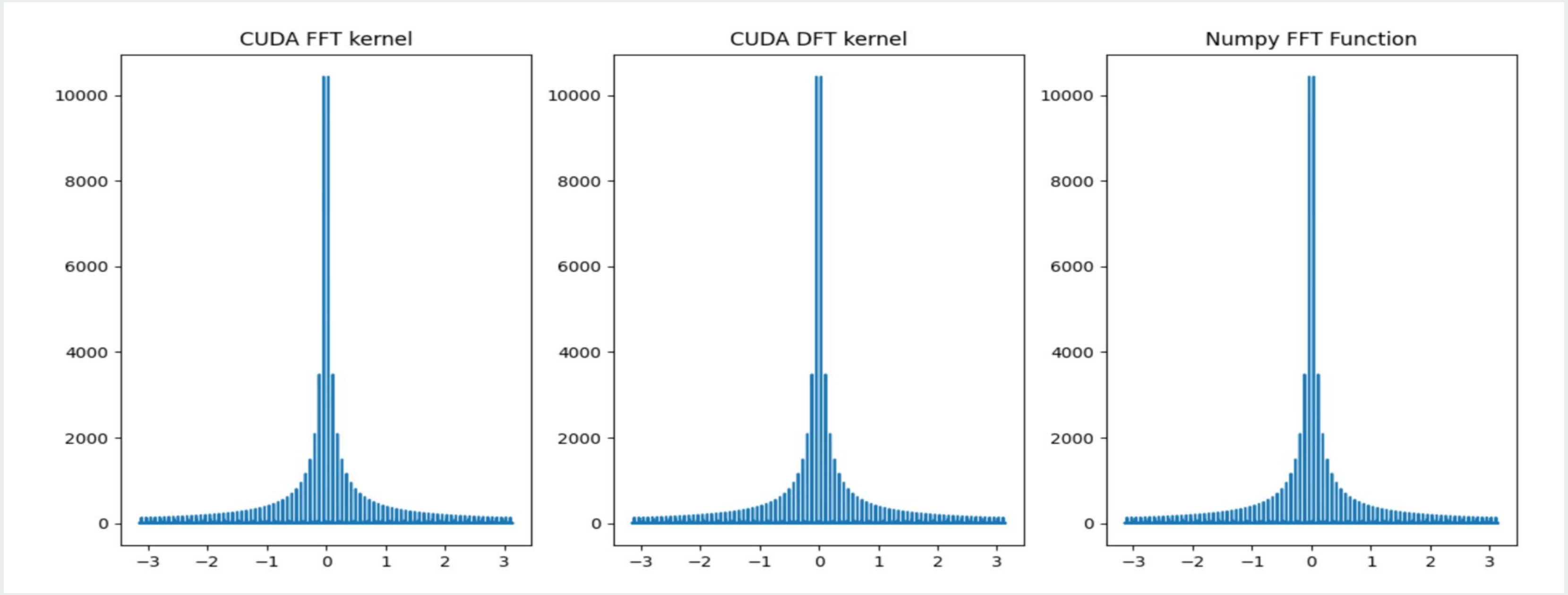


- Strided memory access in stage 7 of a 128 point FFT
- 32 threads in the warp access 2 contiguous sets of memory locations (0-31 and 64-95).
- Aligned and coalesced accesses for N point FFT with N>32, and a threadblock with at least 16 threads.

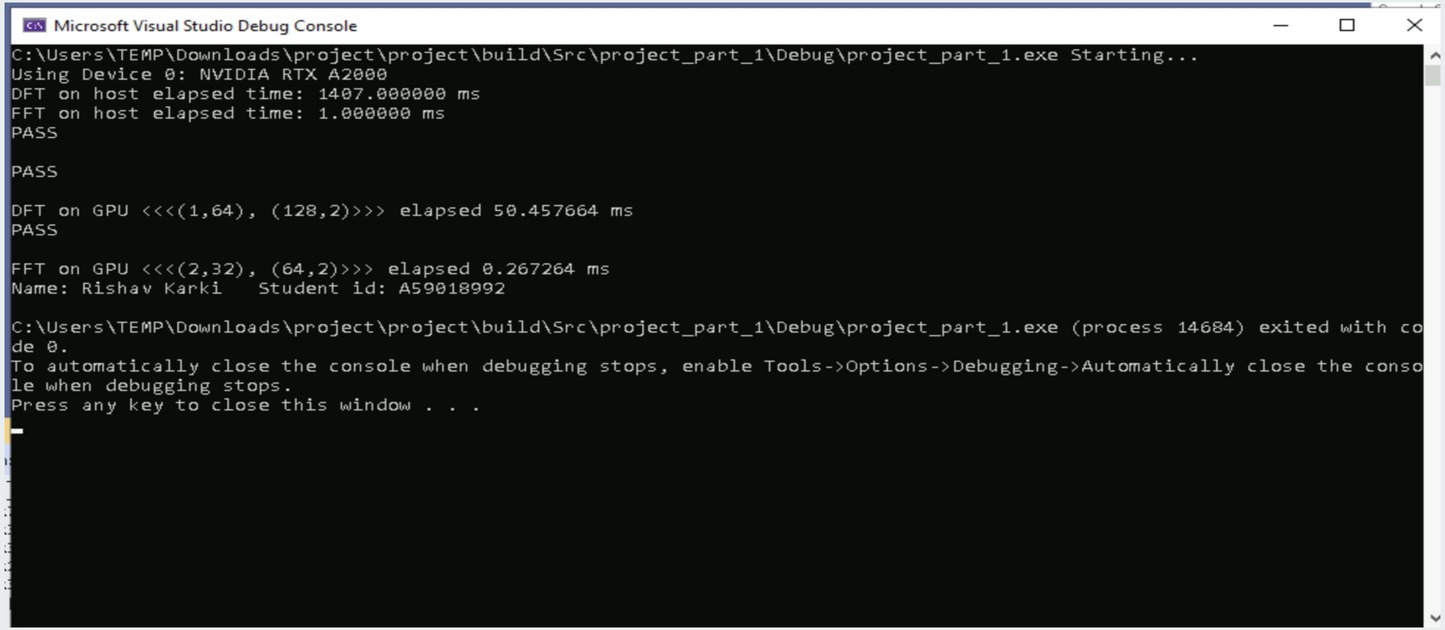
Validation

- Compare Host DFT and Host FFT outputs for the same input vector.
- Compare Host DFT and GPU DFT outputs for the same input vector.
- Compare Host FFT and GPU FFT outputs for the same input vector.
- Use CUDA events to track the run time of the algorithms on GPU.
- Visual inspection for square wave inputs in Python, using PyBind 11.
- Collect kernel run time information using the Python script by varying thread block and grid dimensions to evaluate performance on NVIDIA RTX A2000 (EBU1 server) GPU.

Results : Discrete Fourier Transform of a Square wave

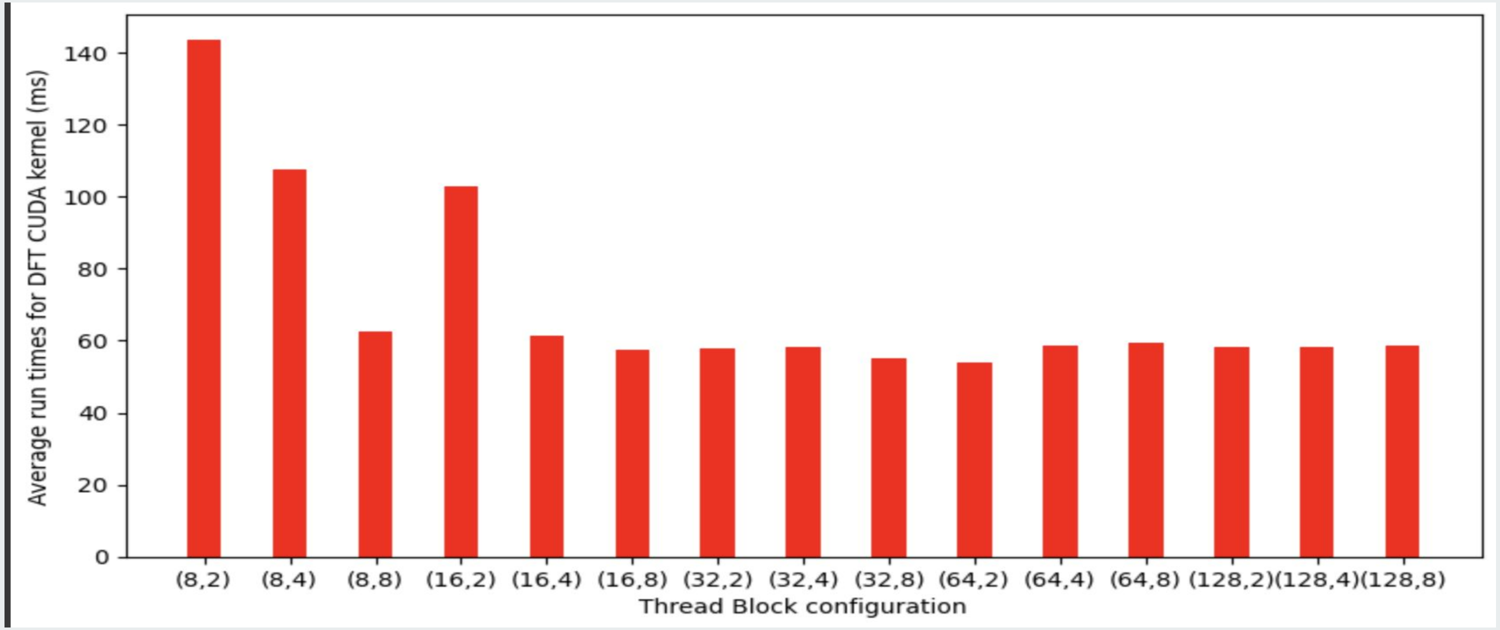


Results : DFT, FFT comparisons on host and GPU

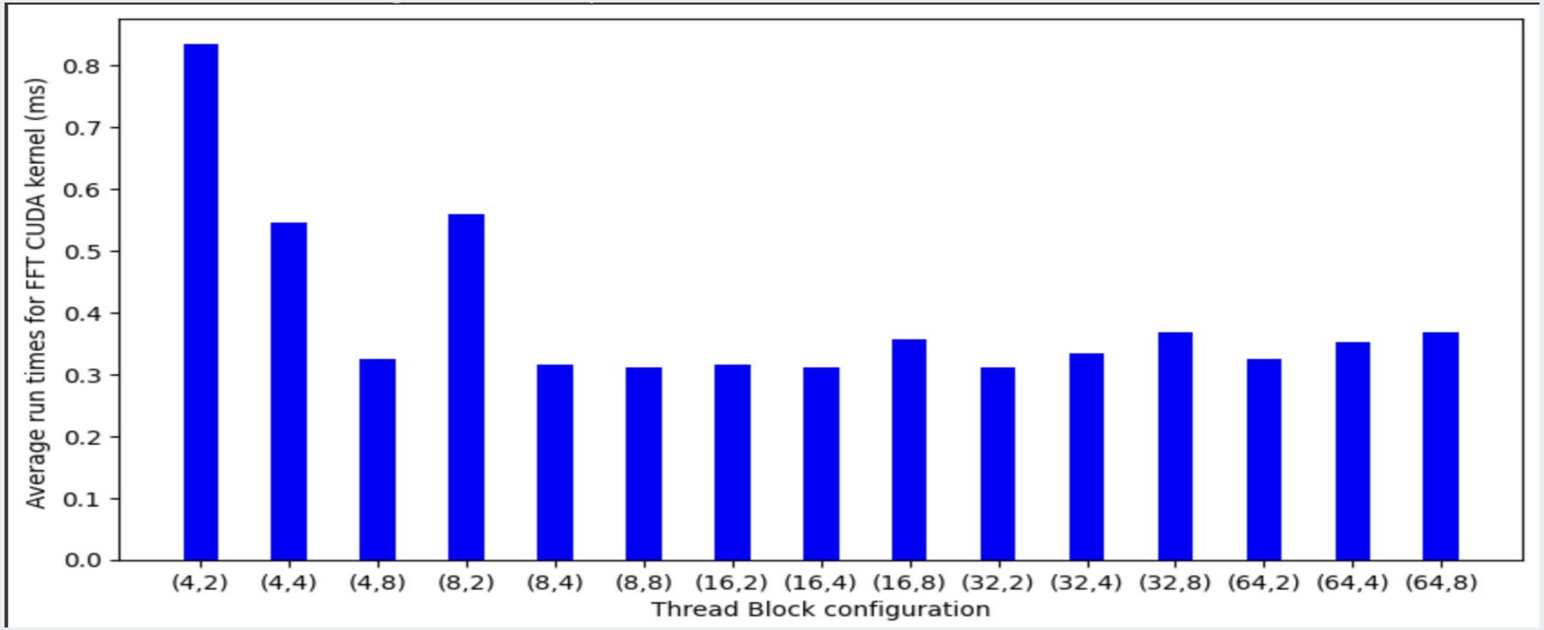

Microsoft Visual Studio Debug Console

```
C:\Users\TEMP\Downloads\project\project\build\Src\project_part_1\Debug\project_part_1.exe Starting...
Using Device 0: NVIDIA RTX A2000
DFT on host elapsed time: 1407.000000 ms
FFT on host elapsed time: 1.000000 ms
PASS

PASS

DFT on GPU <<<(1,64), (128,2)>>> elapsed 50.457664 ms
PASS

FFT on GPU <<<(2,32), (64,2)>>> elapsed 0.267264 ms
Name: Rishav Karki   Student id: A59018992

C:\Users\TEMP\Downloads\project\project\build\Src\project_part_1\Debug\project_part_1.exe (process 14684) exited with co
de 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the conso
le when debugging stops.
Press any key to close this window . . .
```

Results: CUDA DFT Kernel



Results: CUDA FFT Kernel



Analysis

- The number of threads in a thread block should be a multiple of the warp size (32 threads).
    - Impact on memory bandwidth utilization.
    - Reduce memory transactions per memory request, as aligned and coalesced memory accesses are inherently supported in the CUDA kernels.
    - More threads in a thread block implies more thread level parallelism in SIMT paradigm.
- Reducing the number of threads in a block increases the number of thread blocks.
    - Less resources partitioned among blocks and threads
    - Higher achieved occupancy, more active warps on SMs exposing more parallelism.
- Optimal grid and block heuristics should strike a balance.
    - Good memory load/store throughput and good achieved occupancy.
    - The best thread block configuration for 2^{14} point DFT is (64,2).
    - The best thread block configuration for 2^{14} point FFT is (8,8) and (16,4).

Conclusion

- CPU host run times for 2^{14} point DFT : 1380 ms
- CPU host run times for 2^{14} point FFT : 1ms
- GPU run times for 2^{14} point DFT with thread block configuration (64,2) = 54.073 ms
  - 25.52X speedup on GPU vs CPU
- GPU run times for 2^{14} point FFT with thread block configuration (8,8) and (16,4) = 0.311 ms
  - 3.21X speedup on GPU vs CPU

References

- Professional CUDA C Programming
- A GPU Based Memory Optimized Parallel Method ForFFT Implementation -Fan Zhanga, , Chen Hua, Qiang Yina, Wei Hua
- https://www.cmlab.csie.ntu.edu.tw/cml/dsp/training/coding/transform/fft.html

Steps to run the program

- Extract project.zip
- Launch Cmake. Source Code path : “C:/Users/TEMP/Downloads/project/project”
- Build project in visual studio.
- Run run_pybind.py in VS code from path C:/Users/TEMP/Downloads/project/project/py_src