**Lab Exercise 9.3 – Using cuFFT – A CUDA Library for Signal Processing (Fast Fourier Transform)**

**Objective:**

* Learn how to perform Fast Fourier Transform (FFT) using NVIDIA’s cuFFT library on GPU.
* Understand the basic steps to use cuFFT in a CUDA application.
* Compare GPU-based FFT performance with CPU-based methods for signal processing.

**1. What is cuFFT?**

cuFFT is a CUDA library provided by NVIDIA to compute 1D, 2D, and 3D Fast Fourier Transforms on NVIDIA GPUs. It significantly accelerates signal and image processing tasks by parallel execution on the GPU.

**2. Requirements:**

* CUDA Toolkit installed
* A CUDA-enabled GPU
* C++ compiler and nvcc

**3. Program: 1D Complex-to-Complex FFT using cuFFT**

#include <iostream>

#include <cufft.h>

#define N 8 // Number of complex elements

int main() {

cufftComplex \*h\_data, \*d\_data;

cufftHandle plan;

// Allocate host memory

h\_data = (cufftComplex\*)malloc(sizeof(cufftComplex) \* N);

// Initialize input data (e.g., sin wave)

for (int i = 0; i < N; ++i) {

h\_data[i].x = i; // Real part

h\_data[i].y = 0; // Imaginary part

}

// Allocate device memory

cudaMalloc((void\*\*)&d\_data, sizeof(cufftComplex) \* N);

cudaMemcpy(d\_data, h\_data, sizeof(cufftComplex) \* N, cudaMemcpyHostToDevice);

// Create 1D FFT plan

cufftPlan1d(&plan, N, CUFFT\_C2C, 1);

// Execute FFT (forward transform)

cufftExecC2C(plan, d\_data, d\_data, CUFFT\_FORWARD);

// Copy result back to host

cudaMemcpy(h\_data, d\_data, sizeof(cufftComplex) \* N, cudaMemcpyDeviceToHost);

// Display FFT result

std::cout << "FFT Result:" << std::endl;

for (int i = 0; i < N; ++i) {

std::cout << h\_data[i].x << " + " << h\_data[i].y << "i" << std::endl;

}

// Clean up

cufftDestroy(plan);

cudaFree(d\_data);

free(h\_data);

return 0;

}

**4. Explanation:**

* cufftComplex represents complex numbers with real and imaginary parts.
* cufftPlan1d() creates a plan for 1D FFT.
* cufftExecC2C() performs forward FFT on complex data.
* The result is stored back in the same buffer.

**Code Overview**

#include <iostream>

#include <cufft.h>

#define N 8 // Number of complex elements

* Includes standard output and cuFFT headers.
* N defines the length of the signal — here it is 8 complex numbers (small for demonstration).

**Memory Allocation and Initialization**

cufftComplex \*h\_data, \*d\_data;

cufftHandle plan;

// Allocate host memory

h\_data = (cufftComplex\*)malloc(sizeof(cufftComplex) \* N);

// Initialize input data (e.g., sine wave)

for (int i = 0; i < N; ++i) {

h\_data[i].x = i; // Real part

h\_data[i].y = 0; // Imaginary part

}

* cufftComplex is a struct that stores two floats: .x (real) and .y (imaginary).
* A simple signal is initialized on the host with real values 0 to 7, and all imaginary parts as 0.

**🔹 Copy to Device and Plan Setup**

cudaMalloc((void\*\*)&d\_data, sizeof(cufftComplex) \* N);

cudaMemcpy(d\_data, h\_data, sizeof(cufftComplex) \* N, cudaMemcpyHostToDevice);

* Allocates space on the GPU (d\_data).
* Copies the initialized signal from host (h\_data) to device (d\_data).

cufftPlan1d(&plan, N, CUFFT\_C2C, 1);

* Creates an FFT **plan** for a 1D FFT of length N.
* CUFFT\_C2C means **Complex-to-Complex** FFT.
* 1 is the batch size (number of independent FFTs — here we only run one).

**🔹 Execute FFT on GPU**

cufftExecC2C(plan, d\_data, d\_data, CUFFT\_FORWARD);

* Executes a **forward FFT** on d\_data.
* Input and output buffers are the same (in-place).
* CUFFT\_FORWARD indicates a forward FFT (time-domain to frequency-domain).

**5. Compilation:**

nvcc -o fft\_example fft\_example.cu -lcufft

Run it:

./fft\_example

**6. Expected Output:**

You’ll see 8 complex values (real + imaginary) which represent the frequency components of the original input signal.

**7. Use Cases of cuFFT:**

* Audio signal analysis
* Image processing (e.g., filtering in frequency domain)
* Radar and sonar data analysis
* Biomedical signal (ECG, EEG) analysis

**8. Conclusion:**

This lab introduced:

* GPU-accelerated FFT using cuFFT
* Basic structure of cuFFT-based programs
* Performance benefit over CPU-based FFTs (especially for large datasets)