**Lab Exercise 9.2 – Performance difference between using cuBLAS vs. a naive CUDA kernel**

**Performance difference between using cuBLAS vs. a naive CUDA kernel** for matrix multiplication.

**Objective**

Compare the performance (execution time) of:

1. **cuBLAS-based matrix multiplication**
2. **Naive custom CUDA kernel for matrix multiplication**

#include <iostream>

#include <cuda\_runtime.h>

#include <cublas\_v2.h>

#include <chrono>

#define N 1024 // Matrix size

// ----------------------------

// Naive CUDA Matrix Multiply

// ----------------------------

\_\_global\_\_ void matMulKernel(float\* A, float\* B, float\* C, int n) {

int row = blockIdx.y \* blockDim.y + threadIdx.y;

int col = blockIdx.x \* blockDim.x + threadIdx.x;

if (row < n && col < n) {

float value = 0;

for (int k = 0; k < n; ++k)

value += A[row \* n + k] \* B[k \* n + col];

C[row \* n + col] = value;

}

}

// ----------------------------

// Host Helper

// ----------------------------

void checkCuda(cudaError\_t result, const char \*msg) {

if (result != cudaSuccess) {

std::cerr << msg << ": " << cudaGetErrorString(result) << "\n";

exit(EXIT\_FAILURE);

}

}

// ----------------------------

// Main

// ----------------------------

int main() {

size\_t size = N \* N \* sizeof(float);

float \*h\_A = new float[N \* N];

float \*h\_B = new float[N \* N];

float \*h\_C1 = new float[N \* N]; // Output from naive kernel

float \*h\_C2 = new float[N \* N]; // Output from cuBLAS

// Initialize host matrices

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

h\_A[i] = 1.0f;

h\_B[i] = 1.0f;

}

float \*d\_A, \*d\_B, \*d\_C1, \*d\_C2;

checkCuda(cudaMalloc(&d\_A, size), "cudaMalloc d\_A");

checkCuda(cudaMalloc(&d\_B, size), "cudaMalloc d\_B");

checkCuda(cudaMalloc(&d\_C1, size), "cudaMalloc d\_C1");

checkCuda(cudaMalloc(&d\_C2, size), "cudaMalloc d\_C2");

checkCuda(cudaMemcpy(d\_A, h\_A, size, cudaMemcpyHostToDevice), "Memcpy A");

checkCuda(cudaMemcpy(d\_B, h\_B, size, cudaMemcpyHostToDevice), "Memcpy B");

// -----------------------------------------

// 1. Naive CUDA Kernel Performance

// -----------------------------------------

dim3 block(16, 16);

dim3 grid((N + 15) / 16, (N + 15) / 16);

cudaEvent\_t start1, stop1;

cudaEventCreate(&start1);

cudaEventCreate(&stop1);

cudaEventRecord(start1);

matMulKernel<<<grid, block>>>(d\_A, d\_B, d\_C1, N);

cudaEventRecord(stop1);

cudaEventSynchronize(stop1);

float time\_naive = 0;

cudaEventElapsedTime(&time\_naive, start1, stop1);

std::cout << "Naive CUDA kernel time: " << time\_naive << " ms\n";

// -----------------------------------------

// 2. cuBLAS Performance

// -----------------------------------------

cublasHandle\_t handle;

cublasCreate(&handle);

const float alpha = 1.0f;

const float beta = 0.0f;

cudaEvent\_t start2, stop2;

cudaEventCreate(&start2);

cudaEventCreate(&stop2);

cudaEventRecord(start2);

cublasSgemm(handle,

CUBLAS\_OP\_N, CUBLAS\_OP\_N,

N, N, N,

&alpha,

d\_B, N,

d\_A, N,

&beta,

d\_C2, N);

cudaEventRecord(stop2);

cudaEventSynchronize(stop2);

float time\_cublas = 0;

cudaEventElapsedTime(&time\_cublas, start2, stop2);

std::cout << "cuBLAS time: " << time\_cublas << " ms\n";

// -----------------------------------------

// Clean up

// -----------------------------------------

cudaMemcpy(h\_C1, d\_C1, size, cudaMemcpyDeviceToHost);

cudaMemcpy(h\_C2, d\_C2, size, cudaMemcpyDeviceToHost);

cublasDestroy(handle);

cudaFree(d\_A);

cudaFree(d\_B);

cudaFree(d\_C1);

cudaFree(d\_C2);

delete[] h\_A;

delete[] h\_B;

delete[] h\_C1;

delete[] h\_C2;

return 0;

}

**Expected Output**

Naive CUDA kernel time: 43.938 ms

cuBLAS time: 5.00049 ms