**Lab Exercise 11.4 – Performance Impact of Constant Memory in CUDA**

**Objective**

To explore how **constant memory** improves the performance of matrix-vector multiplication in CUDA, compared to using **global memory** only.

**Background**

In CUDA:

* **Global memory** is accessible to all threads but is relatively slow.
* **Constant memory** is small (64KB) but **cached** and **broadcast-efficient**—perfect for values shared across threads like a vector in matrix-vector multiplication.

**Requirements**

* NVIDIA CUDA-capable GPU
* CUDA Toolkit (includes nvcc)
* Any C++ compiler supporting C++11 for timing (e.g., MSVC, g++, clang++)

**Experiment Details**

You will multiply a large matrix by a vector using:

1. A naive implementation using only global memory.
2. An optimized version using constant memory for the vector.

You'll measure the time taken by each kernel.

**matrix\_vector\_constant\_vs\_global.cu**

#include <cuda\_runtime.h>

#include <iostream>

#include <chrono>

#include <cstdlib>

#define N 2024 // Matrix and vector size

#define BLOCK\_SIZE 256

// Constant memory declaration

\_\_constant\_\_ float d\_vector\_const[N];

// Global memory kernel

\_\_global\_\_ void matVecGlobal(const float\* matrix, const float\* vector, float\* result, int n) {

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

if (row < n) {

float sum = 0.0f;

for (int col = 0; col < n; ++col) {

sum += matrix[row \* n + col] \* vector[col];

}

result[row] = sum;

}

}

// Constant memory kernel

\_\_global\_\_ void matVecConst(const float\* matrix, float\* result, int n) {

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

if (row < n) {

float sum = 0.0f;

for (int col = 0; col < n; ++col) {

sum += matrix[row \* n + col] \* d\_vector\_const[col];

}

result[row] = sum;

}

}

// Host utility

void initializeMatrixVector(float\* mat, float\* vec, int n) {

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

mat[i] = static\_cast<float>(rand()) / RAND\_MAX;

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

vec[i] = static\_cast<float>(rand()) / RAND\_MAX;

}

int main() {

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

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

float \*h\_result\_global = new float[N];

float \*h\_result\_const = new float[N];

initializeMatrixVector(h\_matrix, h\_vector, N);

float \*d\_matrix, \*d\_vector, \*d\_result;

cudaMalloc(&d\_matrix, N \* N \* sizeof(float));

cudaMalloc(&d\_vector, N \* sizeof(float));

cudaMalloc(&d\_result, N \* sizeof(float));

cudaMemcpy(d\_matrix, h\_matrix, N \* N \* sizeof(float), cudaMemcpyHostToDevice);

cudaMemcpy(d\_vector, h\_vector, N \* sizeof(float), cudaMemcpyHostToDevice);

dim3 blockSize(BLOCK\_SIZE);

dim3 gridSize((N + BLOCK\_SIZE - 1) / BLOCK\_SIZE);

// ---- Global Memory Kernel ----

std::cout << "[Global Memory Version]\n";

auto start = std::chrono::high\_resolution\_clock::now();

matVecGlobal<<<gridSize, blockSize>>>(d\_matrix, d\_vector, d\_result, N);

cudaDeviceSynchronize();

auto end = std::chrono::high\_resolution\_clock::now();

std::chrono::duration<double, std::milli> duration = end - start;

std::cout << "Time: " << duration.count() << " ms\n";

cudaMemcpy(h\_result\_global, d\_result, N \* sizeof(float), cudaMemcpyDeviceToHost);

// ---- Constant Memory Kernel ----

std::cout << "[Constant Memory Version]\n";

cudaMemcpyToSymbol(d\_vector\_const, h\_vector, N \* sizeof(float));

start = std::chrono::high\_resolution\_clock::now();

matVecConst<<<gridSize, blockSize>>>(d\_matrix, d\_result, N);

cudaDeviceSynchronize();

end = std::chrono::high\_resolution\_clock::now();

duration = end - start;

std::cout << "Time: " << duration.count() << " ms\n";

cudaMemcpy(h\_result\_const, d\_result, N \* sizeof(float), cudaMemcpyDeviceToHost);

// Cleanup

delete[] h\_matrix;

delete[] h\_vector;

delete[] h\_result\_global;

delete[] h\_result\_const;

cudaFree(d\_matrix);

cudaFree(d\_vector);

cudaFree(d\_result);

return 0;

}

**Compile and Run**

nvcc matrix\_vector\_constant\_vs\_global.cu -o matrix\_vector\_test

./matrix\_vector\_test

**Sample Output**

[Global Memory Version]

Time: 7.128 ms

[Constant Memory Version]

Time: 3.056 ms