**Extra Lab Exercise -4 – CUDA Lab Exercise on Thread Divergence Performance Analysis**

The program will run both non-divergent and divergent kernels, and it will measure the execution time for each kernel over multiple iterations to minimize the effect of kernel launch overhead:

**With No Performance Impact:**

#include <iostream>

#include <cuda\_runtime.h>

#define N 1000000 // Increase problem size

// Kernel with no thread divergence (all threads follow the same path)

\_\_global\_\_ void nonDivergentKernel(int \*data) {

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

if (idx < N) {

data[idx] = data[idx] \* 2;

}

}

// Kernel with thread divergence (threads follow different paths)

\_\_global\_\_ void divergentKernel(int \*data) {

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

if (idx < N) {

if (idx % 2 == 0) {

data[idx] = data[idx] \* 2;

} else {

data[idx] = data[idx] / 2;

}

}

}

void runKernel(void (\*kernel)(int \*), int \*d\_data) {

int threadsPerBlock = 256;

int blocksPerGrid = (N + threadsPerBlock - 1) / threadsPerBlock;

// Launch kernel

kernel<<<blocksPerGrid, threadsPerBlock>>>(d\_data);

cudaDeviceSynchronize();

}

int main() {

int \*h\_data = new int[N];

int \*d\_data;

// Initialize host data

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

h\_data[i] = i + 1; // Initializing with 1, 2, 3, ...

}

// Allocate device memory

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

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

// CUDA event handles for timing

cudaEvent\_t start, stop;

cudaEventCreate(&start);

cudaEventCreate(&stop);

// Measure performance for non-divergent kernel

std::cout << "Running non-divergent kernel..." << std::endl;

cudaEventRecord(start);

runKernel(nonDivergentKernel, d\_data);

cudaEventRecord(stop);

cudaEventSynchronize(stop);

float nonDivergentTime;

cudaEventElapsedTime(&nonDivergentTime, start, stop);

// Measure performance for divergent kernel

std::cout << "Running divergent kernel..." << std::endl;

cudaEventRecord(start);

runKernel(divergentKernel, d\_data);

cudaEventRecord(stop);

cudaEventSynchronize(stop);

float divergentTime;

cudaEventElapsedTime(&divergentTime, start, stop);

// Print execution times

std::cout << "Non-Divergent Execution Time: " << nonDivergentTime / 1000.0f << " seconds" << std::endl;

std::cout << "Divergent Execution Time: " << divergentTime / 1000.0f << " seconds" << std::endl;

// Clean up

delete[] h\_data;

cudaFree(d\_data);

cudaEventDestroy(start);

cudaEventDestroy(stop);

return 0;

}

**With Performance impact**

#include <iostream>

#include <cuda\_runtime.h>

#define N 10000000 // Large problem size

#define NUM\_ITERATIONS 100 // Number of iterations to average execution time

// Kernel with no thread divergence (all threads follow the same path)

\_\_global\_\_ void nonDivergentKernel(int \*data) {

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

if (idx < N) {

data[idx] = data[idx] \* 2; // Simple operation for all threads

}

}

// Kernel with thread divergence (threads follow different paths)

\_\_global\_\_ void divergentKernel(int \*data) {

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

if (idx < N) {

// Complex divergence: multiple conditions

if (idx % 3 == 0) {

data[idx] = data[idx] \* 2;

} else if (idx % 3 == 1) {

data[idx] = data[idx] / 2;

} else {

data[idx] = data[idx] + 1;

}

}

}

// Function to run the kernel

void runKernel(void (\*kernel)(int \*), int \*d\_data) {

int threadsPerBlock = 256;

int blocksPerGrid = (N + threadsPerBlock - 1) / threadsPerBlock;

// Launch kernel

kernel<<<blocksPerGrid, threadsPerBlock>>>(d\_data);

cudaDeviceSynchronize();

}

int main() {

int \*h\_data = new int[N];

int \*d\_data;

// Initialize host data

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

h\_data[i] = i + 1; // Initializing with 1, 2, 3, ...

}

// Allocate device memory

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

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

// CUDA event handles for timing

cudaEvent\_t start, stop;

cudaEventCreate(&start);

cudaEventCreate(&stop);

// Measure performance for non-divergent kernel

float nonDivergentTime = 0.0f;

for (int i = 0; i < NUM\_ITERATIONS; ++i) {

cudaEventRecord(start);

runKernel(nonDivergentKernel, d\_data);

cudaEventRecord(stop);

cudaEventSynchronize(stop);

float time;

cudaEventElapsedTime(&time, start, stop);

nonDivergentTime += time;

}

// Measure performance for divergent kernel

float divergentTime = 0.0f;

for (int i = 0; i < NUM\_ITERATIONS; ++i) {

cudaEventRecord(start);

runKernel(divergentKernel, d\_data);

cudaEventRecord(stop);

cudaEventSynchronize(stop);

float time;

cudaEventElapsedTime(&time, start, stop);

divergentTime += time;

}

// Average time across iterations

nonDivergentTime /= NUM\_ITERATIONS;

divergentTime /= NUM\_ITERATIONS;

// Print execution times

std::cout << "Non-Divergent Execution Time: " << nonDivergentTime / 1000.0f << " seconds" << std::endl;

std::cout << "Divergent Execution Time: " << divergentTime / 1000.0f << " seconds" << std::endl;

// Clean up

delete[] h\_data;

cudaFree(d\_data);

cudaEventDestroy(start);

cudaEventDestroy(stop);

return 0;

}

**Explanation:**

1. **Problem Size (N = 10^7)**: The problem size has been increased to 10 million elements to ensure that the GPU is sufficiently stressed during execution.
2. **Divergent Kernel**: The divergentKernel introduces more complex branching, where threads diverge into different paths depending on the result of idx % 3. This should make the divergence more significant and result in more noticeable performance degradation.
3. **Kernel Launch and Timing**: The program runs each kernel 100 times (NUM\_ITERATIONS = 100) and averages the execution time across all iterations. This reduces the noise from kernel launch overhead and gives a more accurate measure of performance.
4. **CUDA Timing**: The cudaEvent API is used to measure the time taken by each kernel. The execution time is measured in milliseconds and then divided by 1000 to convert it to seconds.

**Expected Outcome:**

* **Non-Divergent Kernel**: The execution time will be faster, as there is no thread divergence and all threads perform the same operation.
* **Divergent Kernel**: The execution time will be slower due to the thread divergence, where threads within the same warp are forced to execute different paths, resulting in serialization and lower performance.