**Extra Lab Exercise -3 – CUDA Lab Exercise on Thread Divergence**

**1. Explanation of the Program**

* **Threads in a warp** execute the same instruction at the same time.
* If some threads in a warp take one path in an if statement and others take a different path, **thread divergence** occurs.
* CUDA will execute the divergent paths sequentially, leading to reduced performance.

**2. Basic Steps in the Program**

* Create an array and fill it with numbers.
* Write a kernel that applies a condition to the array.
* Half of the threads will take one branch of the condition, and the other half will take another branch.

**3. Example Program (Demonstrating Thread Divergence)**

#include <iostream>

#include <cuda\_runtime.h>

#define N 64

// Kernel demonstrating thread divergence

\_\_global\_\_ void threadDivergenceKernel(int \*arr) {

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

if (idx < N) {

// Thread divergence occurs here

if (arr[idx] % 2 == 0) {

arr[idx] \*= 2; // Even threads

} else {

arr[idx] \*= 3; // Odd threads

}

}

}

int main() {

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

int \*d\_arr;

// Initialize the array with some values

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

h\_arr[i] = i + 1; // 1, 2, 3, ..., N

}

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

// Allocate memory on the device

cudaMalloc(&d\_arr, size);

// Copy data from host to device

cudaMemcpy(d\_arr, h\_arr, size, cudaMemcpyHostToDevice);

// Launch kernel with 1 block and 64 threads

threadDivergenceKernel<<<1, N>>>(d\_arr);

// Synchronize the device

cudaDeviceSynchronize();

// Copy result from device back to host

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

// Print the results

std::cout << "Results after applying thread divergence:" << std::endl;

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

std::cout << "arr[" << i << "] = " << h\_arr[i] << std::endl;

}

// Free device memory

cudaFree(d\_arr);

delete[] h\_arr;

return 0;

}

**Explanation of the Program:**

1. **Kernel: threadDivergenceKernel**
   * Each thread reads an element from the array arr.
   * Threads with even values in the array multiply their value by 2.
   * Threads with odd values multiply their value by 3.
   * **Thread divergence** occurs here because the even and odd threads follow different execution paths.
2. **Main Function:**
   * We allocate memory for an array on the host (h\_arr) and device (d\_arr).
   * The array is initialized with values from 1 to N on the host.
   * We then copy the data to the device and launch the kernel with N threads.
   * After the kernel execution, we copy the result back to the host and print it out.

**4. Key Points to Observe:**

* Threads with **even numbers** will execute one path (multiply by 2), and threads with **odd numbers** will execute a different path (multiply by 3).
* This divergence leads to sequential execution of both branches, even though the threads are part of the same warp.

**5. Output Example:**

Results after applying thread divergence:

arr[0] = 6

arr[1] = 6

arr[2] = 12

arr[3] = 9

arr[4] = 18

arr[5] = 15

arr[6] = 24

arr[7] = 21

arr[8] = 30

arr[9] = 27

arr[10] = 36

arr[11] = 33

arr[12] = 42

arr[13] = 39

arr[14] = 48

arr[15] = 45

...

* As you can see, **even numbers** are multiplied by 2, and **odd numbers** are multiplied by 3.

**6. Performance Considerations:**

* **Thread Divergence** occurs when some threads in a warp take one path and others take a different path. This can cause the warp to execute the instructions sequentially, one path after another, instead of in parallel.
* This reduces performance, especially when the condition being checked is frequently true or false for many threads in a warp.

**Example Performance Test:**

To observe the performance impact, you could modify the program to use **identical execution paths** for all threads and compare the performance of the divergent and non-divergent cases.

**7. Optimizing for Divergence:**

While we cannot eliminate thread divergence entirely, here are a few tips:

1. **Minimize conditional branching** within warps.
2. Use **predication** or rearrange data so that threads in a warp follow similar paths.
3. **Refactor the logic** to reduce the likelihood of threads following different execution paths.

**8. Conclusion:**

This lab demonstrates thread divergence in CUDA. The main takeaway is that **conditional branching within a warp** leads to inefficiency because divergent threads are executed sequentially. This can have a significant performance impact, especially with large numbers of threads.