**Lab Exercise 3.2 – Create and Manage Threads, Thread-Safe Programming Techniques in CUDA**

**Objective:**

* Learn how to create and manage threads in CUDA.
* Understand and implement thread-safe programming practices.
* Explore synchronization techniques to prevent race conditions.

**1. Key Concepts**

| **Concept** | **Description** |
| --- | --- |
| CUDA Threads | Each CUDA kernel launch creates many lightweight threads. |
| Thread Safety | Ensures correct results when multiple threads access shared resources. |
| \_\_syncthreads() | Synchronizes threads within a block to coordinate access to shared memory. |
| Atomic Operations | Prevents race conditions when multiple threads update the same variable. |

**2. CUDA Program: Safe Sum using Atomic Operations**

This program demonstrates how to safely accumulate values from multiple threads.

#include <iostream>

#include <cuda\_runtime.h>

\_\_global\_\_ void unsafeSum(int \*data, int \*result, int N) {

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

if (tid < N) {

// Not thread-safe: multiple threads can write simultaneously

\*result += data[tid];

}

}

\_\_global\_\_ void safeSum(int \*data, int \*result, int N) {

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

if (tid < N) {

// Thread-safe version using atomic operation

atomicAdd(result, data[tid]);

}

}

int main() {

const int N = 100;

int h\_data[N], h\_result = 0;

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

h\_data[i] = 1;

int \*d\_data, \*d\_result;

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

cudaMalloc((void\*\*)&d\_result, sizeof(int));

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

cudaMemcpy(d\_result, &h\_result, sizeof(int), cudaMemcpyHostToDevice);

int threadsPerBlock = 32;

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

// Use the thread-safe kernel

safeSum<<<blocksPerGrid, threadsPerBlock>>>(d\_data, d\_result, N);

cudaMemcpy(&h\_result, d\_result, sizeof(int), cudaMemcpyDeviceToHost);

std::cout << "Thread-safe sum result: " << h\_result << std::endl;

cudaFree(d\_data);

cudaFree(d\_result);

return 0;

}

**3. Explanation**

* threadIdx.x + blockIdx.x \* blockDim.x: Calculates the global thread ID.
* atomicAdd(): Ensures that multiple threads do not simultaneously modify result, avoiding a **race condition**.
* unsafeSum: A version that is **not thread-safe**—may produce incorrect results if run instead of safeSum.

**4. Compile and Run**

nvcc -o thread\_safe\_sum thread\_safe\_sum.cu

./thread\_safe\_sum

Expected output:

Thread-safe sum result: 100

**5. Summary**

* CUDA threads are easy to create using kernel launches with <<<blocks, threads>>>.
* Shared or global data must be handled carefully to avoid race conditions.
* **Atomic operations** like atomicAdd are essential for **thread-safe** programming when shared data is modified.
* For intra-block data sharing, also consider using **shared memory with synchronization** using \_\_syncthreads().