**Lab Exercise 7.1 – Memory Model and Data Transfer in CUDA (Shared, Global, Constant, Local)**

**Objective:**

* Understand CUDA's memory model.
* Learn how to transfer data between the host (CPU) and the device (GPU).
* Explore different types of memory in CUDA (e.g., global, shared, constant, and local memory).
* Understand the impact of data transfer and memory access patterns on performance.

**1. Introduction to CUDA Memory Model:**

In CUDA, the memory model defines how memory is organized on the GPU. The memory is divided into different types with varying scopes and lifetimes:

* **Global Memory**: This memory is accessible by all threads. It is the largest but also the slowest memory available.
* **Shared Memory**: This memory is shared by all threads within a block. It is much faster than global memory, but it is limited in size.
* **Constant Memory**: This memory is optimized for read-only data that is accessed by all threads.
* **Local Memory**: This memory is specific to each thread and is used for variables that do not fit into registers.

The interaction between the host (CPU) and the device (GPU) involves copying data between the host memory (RAM) and the device memory (GPU memory).

**Lab Exercise: Demonstrating Different Memory Types in CUDA**

**Objective:**

Demonstrate the usage of Global, Shared, Constant, and Local memory in a CUDA kernel by performing a simple vector addition.

**Requirements:**

* CUDA-enabled GPU and toolkit installed.
* Basic knowledge of CUDA programming.

**Steps:**

1. **Set Up the CUDA Environment:** Ensure that your system has the necessary CUDA toolkit and GPU drivers installed.
2. **Define the Problem:** Let's consider the problem of adding two arrays (vectors) element-wise using parallel threads.

#include <iostream>

#include <cmath>

#define N 1024 // Size of the vectors

// Constant memory for storing a constant scalar

\_\_constant\_\_ float constant\_value;

// CUDA Kernel for vector addition

\_\_global\_\_ void vectorAdd(float \*A, float \*B, float \*C) {

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

// Local memory (registers) can be used automatically by the compiler for simple variables

float tempA, tempB;

if (idx < N) {

// Local memory usage (temporary variables within each thread)

tempA = A[idx];

tempB = B[idx];

// Shared memory for intermediate results (within each block)

\_\_shared\_\_ float sharedMemory[256];

// Load data into shared memory (each thread loads its corresponding element)

sharedMemory[threadIdx.x] = tempA + tempB;

\_\_syncthreads(); // Synchronize threads within the block

// Write results to global memory (shared between all threads)

C[idx] = sharedMemory[threadIdx.x] \* constant\_value; // Apply constant factor

}

}

int main() {

// Host vectors

float A[N], B[N], C[N];

// Allocate device memory

float \*d\_A, \*d\_B, \*d\_C;

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

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

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

// Initialize vectors

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

A[i] = i;

B[i] = i \* 2;

}

// Copy data to device

cudaMemcpy(d\_A, A, N \* sizeof(float), cudaMemcpyHostToDevice);

cudaMemcpy(d\_B, B, N \* sizeof(float), cudaMemcpyHostToDevice);

// Set a constant value in constant memory

float constant\_val = 2.0f;

cudaMemcpyToSymbol(constant\_value, &constant\_val, sizeof(float));

// Launch kernel

vectorAdd<<<(N + 255) / 256, 256>>>(d\_A, d\_B, d\_C);

// Copy the result back to the host

cudaMemcpy(C, d\_C, N \* sizeof(float), cudaMemcpyDeviceToHost);

// Print some results

for (int i = 0; i < 10; i++) { // Print first 10 results

std::cout << "C[" << i << "] = " << C[i] << std::endl;

}

// Free device memory

cudaFree(d\_A);

cudaFree(d\_B);

cudaFree(d\_C);

return 0;

}

**Explanation:**

1. **Global Memory**:
   * This memory type is the largest and is used for storing the vectors A, B, and C on the GPU device. Each thread can access global memory, but it’s slower than other memory types.
2. **Shared Memory**:
   * Shared memory is used inside the kernel to store temporary results that are accessed by threads within the same block. Here, we use sharedMemory to store the sum of corresponding elements of A and B within each block. Threads within the same block can access and update this memory.
3. **Constant Memory**:
   * Constant memory is used for read-only data that is accessible by all threads. In this example, we store a constant scalar value constant\_value in the constant memory. All threads use this value to multiply the sum of A and B.
4. **Local Memory**:
   * Local memory is used for thread-specific variables that do not fit into registers. In this example, variables tempA and tempB are stored in local memory, though in many cases, the compiler might optimize such variables to register memory automatically.

**Observations:**

* **Performance**: You will notice that shared memory improves performance within each block as it is much faster than global memory.
* **Constant Memory**: It improves access speeds when all threads are reading the same value.
* **Local Memory**: It’s used for thread-specific data and can be slower than registers but is necessary for data that doesn’t fit in registers.