**Lab Exercise- Advanced Thread Synchronization Techniques - Barriers and Locks in CUDA**

In this lab, you will explore **advanced thread synchronization techniques** in CUDA, such as **barriers** and **locks**, for complex applications like **parallel prefix sums** (scan) and **parallel sorting**. These techniques are crucial when multiple threads need to synchronize at specific points during execution or when they access shared resources.

The lab consists of the following tasks:

1. **Implementing a Parallel Prefix Sum (Scan) using Barriers**.
2. **Implementing Parallel Sorting using Locks and Synchronization**.

**1. Implementing a Parallel Prefix Sum (Scan) using Barriers**

A **prefix sum** (or **scan**) is a fundamental parallel algorithm that computes the cumulative sum of an array of numbers. For example, for an array [1, 2, 3, 4], the prefix sum would be [1, 3, 6, 10].

In CUDA, implementing a parallel prefix sum requires **barriers** to synchronize threads, ensuring that all threads in a block have completed their computations at a certain point.

**Objective:**

* Implement a parallel prefix sum using CUDA.
* Use barriers to synchronize threads within a block to ensure correct calculations.

**CUDA Example: Parallel Prefix Sum with Barriers**

#include <iostream>

#include <cuda\_runtime.h>

\_\_device\_\_ void prefix\_sum\_kernel(int \*arr, int N) {

// Shared memory to hold partial sums

\_\_shared\_\_ int temp[256];

int tid = threadIdx.x;

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

if (gid < N) {

temp[tid] = arr[gid];

}

\_\_syncthreads(); // Barrier: Ensure all threads have written to shared memory

// Perform prefix sum in shared memory

for (int offset = 1; offset < blockDim.x; offset \*= 2) {

if (tid >= offset) {

temp[tid] += temp[tid - offset];

}

\_\_syncthreads(); // Barrier: Ensure all threads have completed their addition

}

if (gid < N) {

arr[gid] = temp[tid];

}

}

int main() {

const int N = 1024;

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

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

int \*d\_arr;

// Initialize input array

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

h\_arr[i] = 1; // Each element is set to 1 for simplicity

}

// Allocate memory on device

cudaMalloc((void\*\*)&d\_arr, size);

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

// Launch kernel to perform parallel prefix sum

int threadsPerBlock = 256;

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

prefix\_sum\_kernel<<<blocksPerGrid, threadsPerBlock>>>(d\_arr, N);

// Copy the result back to host

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

// Print the first 10 results

for (int i = 0; i < 10; i++) {

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

}

// Free memory

delete[] h\_arr;

cudaFree(d\_arr);

return 0;

}

**Explanation**:

* The kernel prefix\_sum\_kernel calculates the prefix sum of the array in parallel using **shared memory** to hold partial sums.
* **\_\_syncthreads()** is used to synchronize threads within a block at crucial points to ensure that threads correctly exchange values in the shared memory before continuing their computations.
* The algorithm uses a **binary tree reduction** approach, where each thread adds its value to the one before it in the shared memory. This approach requires barriers at each step to ensure that all threads in the block synchronize before the next step.

**2. Implementing Parallel Sorting using Locks and Synchronization**

**Parallel sorting** is a common problem that can be efficiently solved using parallel algorithms. One such approach is **bitonic sorting**, where elements are compared and swapped in a sequence that follows a fixed pattern. This algorithm can be made parallel by synchronizing the threads at key stages.

**Objective:**

* Implement a parallel sorting algorithm using **locks** and synchronization to avoid race conditions when accessing shared resources.
* Use **barriers** to ensure that all threads reach a certain stage before continuing.

**CUDA Example: Parallel Bitonic Sort with Locks**

#include <iostream>

#include <cuda\_runtime.h>

// CUDA kernel to perform bitonic sort

\_\_global\_\_ void bitonic\_sort\_kernel(int \*arr, int N, int stage) {

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

int pairIdx = tid ^ stage; // Pairing element index

// Synchronize threads within a block

\_\_syncthreads();

// Compare and swap if necessary

if (tid < N && pairIdx < N) {

if (arr[tid] > arr[pairIdx]) {

int temp = arr[tid];

arr[tid] = arr[pairIdx];

arr[pairIdx] = temp;

}

}

// Synchronize threads before next stage

\_\_syncthreads();

}

int main() {

const int N = 1024;

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

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

int \*d\_arr;

// Initialize input array with random values

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

h\_arr[i] = rand() % 1000;

}

// Allocate memory on device

cudaMalloc((void\*\*)&d\_arr, size);

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

// Launch kernel for bitonic sort

int threadsPerBlock = 256;

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

for (int stage = 1; stage < N; stage \*= 2) {

bitonic\_sort\_kernel<<<blocksPerGrid, threadsPerBlock>>>(d\_arr, N, stage);

}

// Copy the result back to host

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

// Print the first 10 sorted elements

for (int i = 0; i < 10; i++) {

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

}

// Free memory

delete[] h\_arr;

cudaFree(d\_arr);

return 0;

}

**Explanation**:

* The bitonic\_sort\_kernel kernel performs the sorting operation using the **bitonic sorting** algorithm. This involves comparing and swapping elements in a fixed pattern across different stages of sorting.
* **\_\_syncthreads()** is used to synchronize the threads after each stage to ensure that they all reach the same point before the next comparison.
* In this example, we use a **pairing index** (pairIdx) to compare elements in a "bitonic sequence" where elements are compared and swapped based on the bitonic pattern.

**3. Challenges and Performance Considerations**

* **Barrier Synchronization**: When using \_\_syncthreads(), you must be careful to avoid situations where threads in a block wait on threads that are not yet finished. This can lead to **deadlocks** or inefficiencies.
* **Locks**: In CUDA, locking mechanisms are not directly available. We often rely on atomic operations or other techniques (like using **shared memory** and proper synchronization) to avoid race conditions in cases like sorting or reductions.
* **Memory Coalescing**: For better performance, ensure that memory access is coalesced, i.e., threads should access consecutive memory locations. This is important for both shared and global memory access.

**4. Conclusion**

In this lab, you learned how to:

1. **Implement parallel prefix sums** using barriers to synchronize threads and perform efficient parallel reductions.
2. **Implement parallel sorting** (bitonic sort) using locks and synchronization mechanisms like \_\_syncthreads() to avoid race conditions and ensure correct sorting.