**Lab Exercise 1.2 – Simple for Loop in a CUDA Program**

**Objective**

* Understand how a for loop works inside a CUDA kernel.
* Learn how each thread can repeatedly operate on its assigned data.

**Program:**

#include <iostream>

#include <cuda\_runtime.h>

\_\_global\_\_ void addConstantMultipleTimes(int \*a, int N) {

int tid = blockIdx.x \* blockDim.x + threadIdx.x; // Global thread ID

if (tid < N) {

for (int i = 0; i < 3; i++) { // Repeat 3 times

a[tid] += 1; // Add 1 to the element

}

}

}

int main() {

const int N = 5;

int h\_a[N] = {1, 2, 3, 4, 5};

int \*d\_a;

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

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

addConstantMultipleTimes<<<1, N>>>(d\_a, N);

cudaMemcpy(h\_a, d\_a, N \* sizeof(int), cudaMemcpyDeviceToHost);

std::cout << "Result after adding 1 three times:\n";

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

std::cout << h\_a[i] << " ";

}

std::cout << std::endl;

cudaFree(d\_a);

return 0;

}

**Step-by-Step Execution and Explanation**

**Step 1: Include Headers**

#include <iostream>

#include <cuda\_runtime.h>

* iostream is needed for printing results.
* cuda\_runtime.h provides CUDA functions.

**Step 2: Define the CUDA Kernel**

\_\_global\_\_ void addConstantMultipleTimes(int \*a, int N)

* \_\_global\_\_ marks this function as a **kernel** that runs on the GPU.
* Each thread will add **1** to an element **three times** using a for loop.

**Step 3: Calculate Global Thread ID**

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

* blockIdx.x: Block ID.
* blockDim.x: Number of threads per block.
* threadIdx.x: Thread ID inside the block.
* tid uniquely identifies which element this thread will modify.

**Step 4: Use a for Loop inside Kernel**

for (int i = 0; i < 3; i++) {

a[tid] += 1;

}

* Each thread repeats adding 1 **three times**.
* Example:
  + Start: a[tid] = 1
  + After 1st loop: 2
  + After 2nd loop: 3
  + After 3rd loop: 4

**Step 5: Host (CPU) Code - Main Function**

const int N = 5;

int h\_a[N] = {1, 2, 3, 4, 5};

* Create an array of 5 integers initialized to {1, 2, 3, 4, 5}.

**Step 6: Allocate Device Memory**

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

* Allocate memory for array d\_a on the GPU.

**Step 7: Copy Data from Host to Device**

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

* Copy h\_a from CPU memory to GPU memory d\_a.

**Step 8: Launch Kernel**

addConstantMultipleTimes<<<1, N>>>(d\_a, N);

* 1 block with N=5 threads.
* Each thread handles one element from the array.

**Step 9: Copy Result Back from Device to Host**

cudaMemcpy(h\_a, d\_a, N \* sizeof(int), cudaMemcpyDeviceToHost);

* After computation, copy modified data back to the CPU array h\_a.

**Step 10: Display Result**

std::cout << "Result after adding 1 three times:\n";

* Print the new values of the array.

**Step 11: Free Device Memory**

cudaFree(d\_a);

* Release the memory allocated on the GPU.

**Expected Output**

Result after adding 1 three times:

4 5 6 7 8

**Explanation:**

* Each original value was increased by 3:
  + 1 → 4
  + 2 → 5
  + 3 → 6
  + 4 → 7
  + 5 → 8

**Summary**

* A **for loop** inside a CUDA kernel works independently in each thread.
* Each thread repeats a task (here, adding 1) multiple times.
* CUDA allows massive parallel execution of simple tasks through thousands of threads.