Name: Omkar Rajesh Auti

PRN No: 21510042

**High Performance Computing Lab**

**Practical No. 11**

**Title of practical: Understanding concepts of CUDA Programming**

**Problem Statement 1:**

**Execute the following program and check the properties of your GPGPU.**

**#include <stdio.h>**

**#include <stdlib.h>**

**int main()**

**{**

**int deviceCount;**

**cudaGetDeviceCount(&deviceCount);**

**if (deviceCount == 0)**

**{**

**printf("There is no device supporting CUDA\n");**

**}**

**int dev;**

**for (dev = 0; dev < deviceCount; ++dev)**

**{**

**cudaDeviceProp deviceProp;**

**cudaGetDeviceProperties(&deviceProp, dev);**

**if (dev == 0)**

**{**

**if (deviceProp.major < 1)**

**{**

**printf("There is no device supporting CUDA.\n");**

**}**

**else if (deviceCount == 1)**

**{**

**printf("There is 1 device supporting CUDA\n");**

**}**

**else**

**{**

**printf("There are %d devices supporting CUDA\n", deviceCount);**

**}**

**}**

**printf("\nDevice %d: \"%s\"\n", dev, deviceProp.name);**

**printf(" Major revision number: %d\n", deviceProp.major);**

**printf(" Minor revision number: %d\n", deviceProp.minor);**

**printf(" Total amount of global memory: %d bytes\n", deviceProp.totalGlobalMem);**

**printf(" Total amount of constant memory: %d bytes\n", deviceProp.totalConstMem);**

**printf(" Total amount of shared memory per block: %d bytes\n", deviceProp.sharedMemPerBlock);**

**printf(" Total number of registers available per block: %d\n", deviceProp.regsPerBlock);**

**printf(" Warp size: %d\n", deviceProp.warpSize);**

**printf(" Multiprocessor count: %d\n",deviceProp.multiProcessorCount );**

**printf(" Maximum number of threads per block: %d\n", deviceProp.maxThreadsPerBlock);**

**printf(" Maximum sizes of each dimension of a block: %d x %d x %d\n", deviceProp.maxThreadsDim[0],deviceProp.maxThreadsDim[1], deviceProp.maxThreadsDim[2]);**

**printf(" Maximum sizes of each dimension of a grid: %d x %d x %d\n", deviceProp.maxGridSize[0], deviceProp.maxGridSize[1], deviceProp.maxGridSize[2]);**

**printf(" Maximum memory pitch: %d bytes\n", deviceProp.memPitch);**

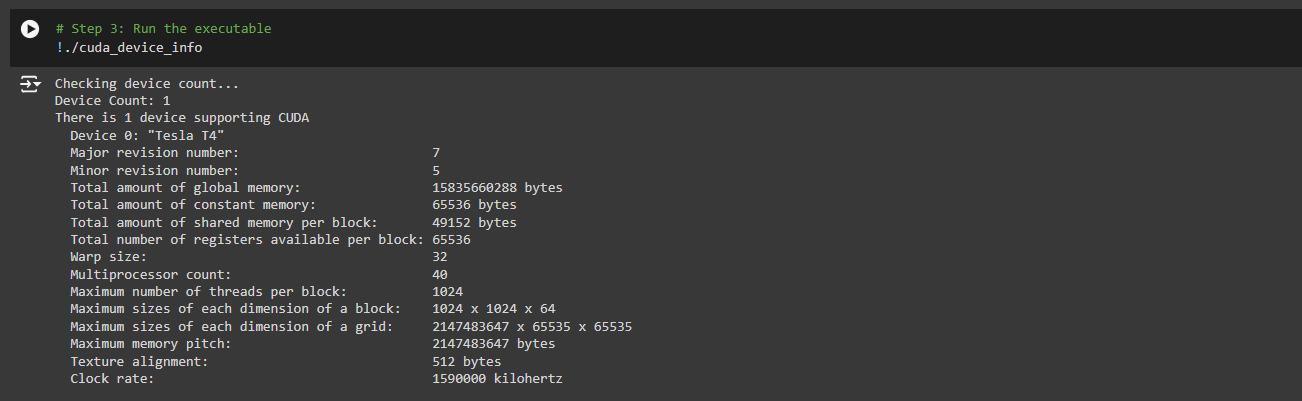
**printf(" Texture alignment: %d bytes\n", deviceProp.textureAlignment);**

**printf(" Clock rate: %d kilohertz\n", deviceProp.clockRate);**

**}**

**}**

**Output:**

****

**Problem Statement 2:**

**Write a program to where each thread prints its thread ID along with hello world. Lauch the kernel with one block and multiple threads.**

**Code:**

**%%writefile cuda\_device\_info.cu**

**#include <stdio.h>**

**#include <cuda\_runtime.h>**

**\_\_global\_\_ void helloWorldKernel() {**

**// Get the block ID and thread ID**

**int blockId = blockIdx.x;**

**int threadId = threadIdx.x;**

**printf("Hello World from block %d, thread %d\n", blockId, threadId);**

**}**

**int main() {**

**// Launch the kernel with 5 blocks and 10 threads per block**

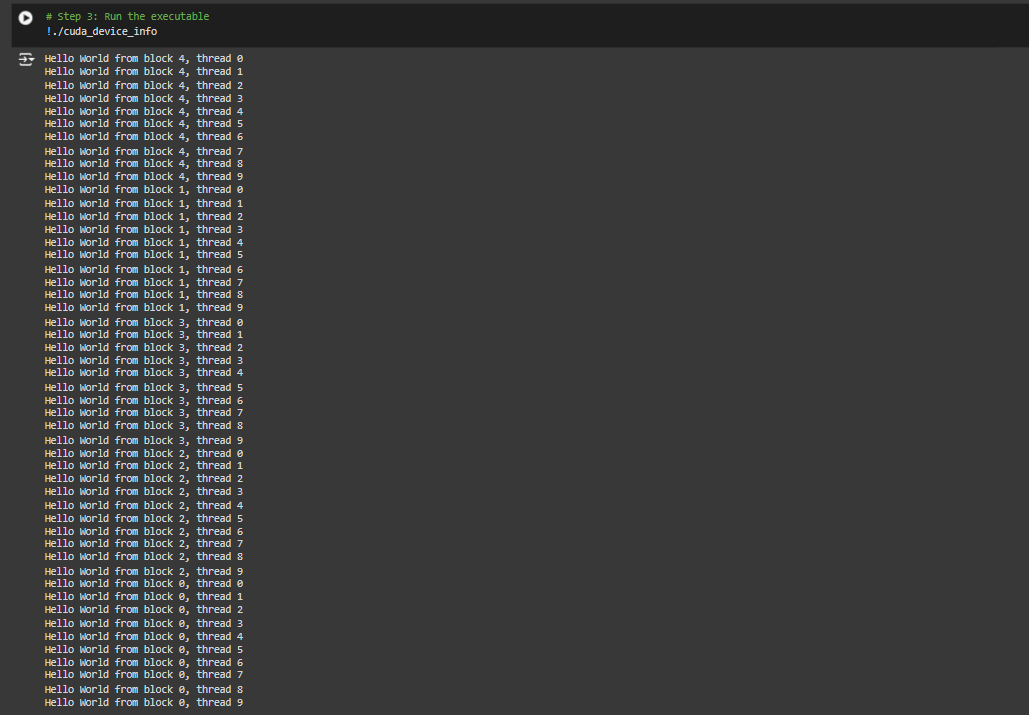
**helloWorldKernel<<<5, 10>>>(); // 5 blocks, 10 threads per block**

**cudaDeviceSynchronize(); // Wait for the kernel to finish**

**return 0;**

**}**

**Output:**

****

**Analysis:**

**Block and Thread Structure:**

* **There are 5 blocks (numbered 0 to 4), with each block containing 10 threads (numbered 0 to 9).**
* **Each thread within a block prints a message that includes its block and thread ID.**

**Execution Order:**

* **The output is not in sequential order (i.e., block 0, followed by block 1, and so on). Instead, blocks appear in a seemingly random order (4, 1, 3, 2, and then 0). This is expected behaviour in parallel processing, as blocks are scheduled and executed independently and may complete in any order.**
* **The threads within each block are executed in sequential order from thread 0 to thread 9. This suggests that within each block, thread execution is ordered, but block order is not guaranteed.**

**Parallel Processing Analysis:**

* **This pattern is a typical outcome of multi-threaded execution in a block-based parallel programming model, such as with CUDA or OpenMP, where the scheduler assigns blocks and threads to available resources, allowing for efficient concurrent execution.**

**Interpretation of Output:**

* **The interleaving of blocks reflects the flexibility of parallel scheduling, which allows independent tasks (like blocks) to run concurrently without enforcing a strict order.**
* **The consistent order within each block (thread 0 to thread 9) might imply a controlled or synchronized sequence within the block, even if the blocks themselves are scheduled freely across processing units.**

**Problem Statement 3:**

**Write a program to where each thread prints its thread ID along with hello world. Lauch the kernel with multiple blocks and multiple threads.**

**Code:**

**%%writefile cuda\_device\_info.cu**

**#include <stdio.h>**

**#include <cuda\_runtime.h>**

**\_\_global\_\_ void helloWorldKernel() {**

**// Get the block ID and thread ID within the block**

**int blockId = blockIdx.x;**

**int threadIdInBlock = threadIdx.x;**

**// Get the global thread ID across all blocks**

**int globalThreadId = blockId \* blockDim.x + threadIdInBlock;**

**printf("Hello World from block %d, thread %d (global thread ID: %d)\n", blockId, threadIdInBlock, globalThreadId);**

**}**

**int main() {**

**int numBlocks = 2;          // Number of blocks**

**int threadsPerBlock = 5;    // Number of threads per block**

**// Launch the kernel with multiple blocks and multiple threads**

**helloWorldKernel<<<numBlocks, threadsPerBlock>>>();**

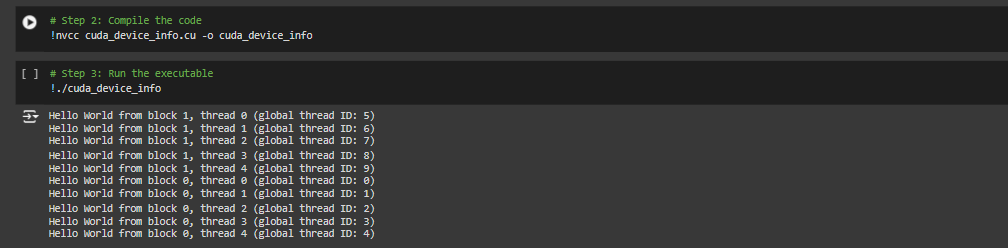
**// Synchronize the device**

**cudaDeviceSynchronize(); // Wait for the kernel to finish**

**return 0;**

**}**

**Output:**

****

**Analysis:**

**Block and Thread Structure:**

* **There are 2 blocks (block 0 and block 1), each with 5 threads.**
* **Threads within each block have local thread IDs from 0 to 4.**
* **Each thread also has a global thread ID, which is a unique identifier across all blocks and threads, calculated as global\_thread\_id = block\_id \* num\_threads\_per\_block + thread\_id.**

**Global Thread ID Calculation:**

* **For block 0, thread IDs from 0 to 4 correspond to global thread IDs from 0 to 4.**
* **For block 1, thread IDs from 0 to 4 correspond to global thread IDs from 5 to 9.**
* **This global ID uniquely identifies each thread across all blocks, allowing for a single identifier in the entire grid, useful in indexing and memory operations in parallel processing.**

**Execution Order:**

* **The output does not follow strict sequential order (block 0 then block 1), likely due to parallel scheduling.**
* **Block 1 messages appear first, followed by block 0, which indicates that block execution can start and finish independently, and block order is not enforced.**

**Parallel Execution and Scheduling:**

* **The output suggests that while threads within each block execute in a specific order (0 to 4), the blocks themselves may execute in any order due to parallel scheduling.**
* **This unordered block execution reflects parallel processing's non-deterministic scheduling, where independent units (blocks) are scheduled based on resource availability, leading to varied completion times.**

**Problem Statement 4:**

**Write a program to where each thread prints its thread ID along with hello world. Lauch the kernel with 2D blocks and 2D threads.**

**Code:**

**%%writefile cuda\_device\_info.cu**

**#include <stdio.h>**

**#include <cuda\_runtime.h>**

**\_\_global\_\_ void helloWorldKernel() {**

**// Get the 2D thread ID within the block**

**int threadIdX = threadIdx.x;**

**int threadIdY = threadIdx.y;**

**// Get the 2D block ID**

**int blockIdX = blockIdx.x;**

**int blockIdY = blockIdx.y;**

**// Get the global thread ID in 2D grid**

**int globalThreadIdX = blockIdX \* blockDim.x + threadIdX;**

**int globalThreadIdY = blockIdY \* blockDim.y + threadIdY;**

**printf("Hello World from block (%d, %d), thread (%d, %d) (global thread ID: (%d, %d))\n",**

**blockIdX, blockIdY, threadIdX, threadIdY, globalThreadIdX, globalThreadIdY);**

**}**

**int main() {**

**dim3 threadsPerBlock(2, 2); // Size of the block (2x2 threads)**

**dim3 numBlocks(2, 2);       // Number of blocks (2x2 blocks)**

**// Launch the kernel with 2D blocks and 2D threads**

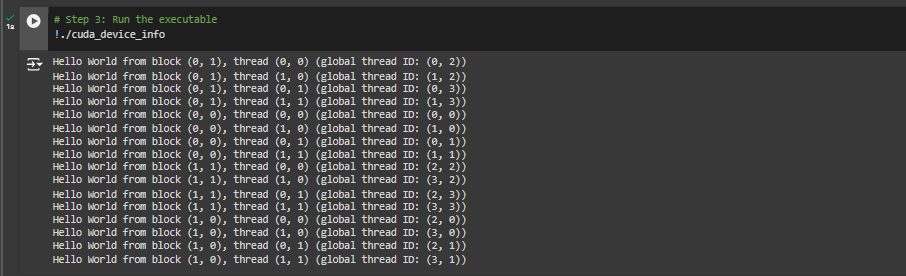
**helloWorldKernel<<<numBlocks, threadsPerBlock>>>();**

**cudaDeviceSynchronize(); // Wait for the kernel to finish**

**return 0;**

**}**

**Output:**

****

**Analysis:**

**2D Block and Thread Structure:**

* **The setup shows a two-dimensional block structure, where each block is identified by coordinates (x, y).**
* **Each block contains threads arranged in a two-dimensional grid, also identified by (x, y) coordinates.**
* **The format for each output line is:  
  "Hello World from block (block\_x, block\_y), thread (thread\_x, thread\_y) (global thread ID: (global\_x, global\_y))"**

**Global Thread ID Calculation:**

* **Each thread is assigned a unique global thread ID as a coordinate (global\_x, global\_y), which seems to relate to the block and thread coordinates.**
* **From the output, we can infer a formula where global\_x and global\_y are determined as follows:**
  + **global\_x = block\_x \* threads\_per\_block\_x + thread\_x**
  + **global\_y = block\_y \* threads\_per\_block\_y + thread\_y**
* **This formula allows the system to map each thread in its block to a unique global identifier, useful for tasks that need distinct identification across the entire grid.**

**Execution Order:**

* **The blocks do not follow a sequential order (like (0,0), (0,1), (1,0), (1,1)), suggesting that each block executes independently, as per the scheduling order.**
* **Within each block, however, threads are listed in order, indicating sequential processing within each block, even though blocks themselves may complete at different times.**

**Parallel Execution Insights:**

* **The independence of block execution shows that the program allows blocks to run in parallel, leveraging the underlying system's scheduling to determine the order.**
* **The sequential nature within each block (order of threads from (0,0) to (1,1)) ensures that operations within a block are synchronized or ordered but allows flexibility across blocks.**

**Code:**

**%%writefile cuda\_device\_info.cu**

**#include <stdio.h>**

**#include <cuda\_runtime.h>**

**\_\_global\_\_ void helloWorldKernel() {**

**// Get the 2D thread ID within the block**

**int threadIdX = threadIdx.x;**

**int threadIdY = threadIdx.y;**

**// Get the 2D block ID**

**int blockIdX = blockIdx.x;**

**int blockIdY = blockIdx.y;**

**// Get the global thread ID in 2D grid**

**int globalThreadIdX = blockIdX \* blockDim.x + threadIdX;**

**int globalThreadIdY = blockIdY \* blockDim.y + threadIdY;**

**// Calculate the 1D global ID**

**int globalThreadId1D = (blockIdY \* gridDim.x + blockIdX) \* (blockDim.x \* blockDim.y) + (threadIdY \* blockDim.x) + threadIdX;**

**printf("Hello World from block (%d, %d), thread (%d, %d) (global thread ID: (%d, %d), 1D global ID: %d)\n",**

**blockIdX, blockIdY, threadIdX, threadIdY, globalThreadIdX, globalThreadIdY, globalThreadId1D);**

**}**

**int main() {**

**dim3 threadsPerBlock(2, 2); // Size of the block (2x2 threads)**

**dim3 numBlocks(2, 2);       // Number of blocks (2x2 blocks)**

**// Launch the kernel with 2D blocks and 2D threads**

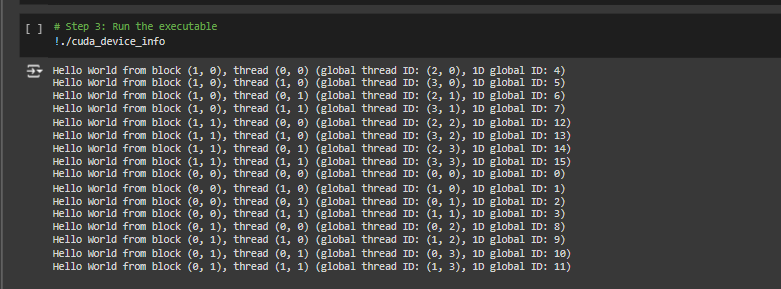
**helloWorldKernel<<<numBlocks, threadsPerBlock>>>();**

**cudaDeviceSynchronize(); // Wait for the kernel to finish**

**return 0;**

**}**

**Output:**

****

**Analysis:**

**2D Block and Thread Structure:**

* **Each line shows a thread within a specific 2D block, identified as (block\_x, block\_y).**
* **Each thread within a block is located by a 2D coordinate (thread\_x, thread\_y).**
* **This setup reflects a hierarchical structure where threads are organized in a 2D grid within each block.**

**Global Thread ID (2D):**

* **The global thread ID (x, y) is a unique identifier that represents each thread’s position across all blocks, calculated based on both block and thread positions.**
* **From the output, the calculation for each global coordinate appears to be:**
  + **global\_x = block\_x \* threads\_per\_block\_x + thread\_x**
  + **global\_y = block\_y \* threads\_per\_block\_y + thread\_y**
* **This mapping allows each thread to have a unique position in a 2D plane that spans all blocks.**

**1D Global ID:**

* **The 1D global ID provides a unique linear (1D) identifier for each thread, which is likely calculated by flattening the 2D grid into a single-dimensional array.**
* **The formula appears to follow a row-major order:**
  + **1D global ID = global\_x \* total\_grid\_y + global\_y**
* **This conversion simplifies referencing and accessing each thread, especially in applications requiring a linear index (e.g., indexing in arrays).**

**Execution Order:**

* **The blocks are processed in a non-sequential order, suggesting that block execution depends on system scheduling and is not strictly sequential.**
* **Within each block, threads are listed in a fixed order (typically row-by-row), indicating that threads within a block are processed sequentially.**

**Parallel Execution Insights:**

* **The non-deterministic execution order of blocks indicates parallelism, where the execution of different blocks is independent, allowing for concurrent processing.**
* **This layout is particularly efficient for large-scale parallel computing tasks where each block can operate independently, while each thread within a block can handle sub-tasks of the larger process.**