Assignment-16

1. What is CUDA?

   CUDA stands for Compute Unified Device Architecture. It is a parallel computing platform and programming model developed by NVIDIA for general computing on graphical processing units (GPUs). It allows developers to harness the computational power of NVIDIA GPUs for parallel processing tasks, such as scientific simulations, image processing, deep learning, and more.

2. What is the prerequisite for learning CUDA?

   Strong understanding of parallel computing concepts, familiarity with C or C++ programming languages, and basic knowledge of GPU architecture and memory hierarchy.

3. Which are the languages that support CUDA?

   CUDA primarily supports C and C++ programming languages. However, there are also libraries and frameworks available for other languages like Python (with libraries like PyCUDA or Numba), Fortran, and even MATLAB for CUDA programming.

4. What do you mean by a CUDA ready architecture?

   CUDA-ready architecture refers to GPUs that are designed and optimized to work efficiently with CUDA programming. These GPUs have specialized hardware components and software support to accelerate parallel processing tasks performed using CUDA.

5. How CUDA works?

   Big Task: Imagine you have a huge math problem to solve.

Ask for Help: Instead of doing it alone, you ask your GPU friend for help.

Divide the Work: Your GPU friend splits the math problem into smaller parts.

Everyone Works Together: Each part is solved at the same time by different parts of the GPU.

Faster Solution: Because everyone is working together, the math problem gets solved much faster than if you did it alone.

6. What are the benefits and limitations of CUDA programming?

   Benefits:

   Accelerated performance for parallelizable tasks.

   Utilization of GPU resources for computational tasks.

   Scalability across different CUDA-enabled GPUs.

   Support for a wide range of parallel computing applications.

   Limitations:

   Requires a dedicated NVIDIA GPU.

   Complexity in managing memory hierarchy.

   Limited support for non-NVIDIA GPUs.

   Requires understanding of parallel programming concepts.

7. Understand and explain the CUDA program structure with an example.

   Include Necessary Headers:

   #include <iostream>: Include the standard input/output stream library for printing results.

   Define CUDA Kernel Function:

   global void vectorAdd(int *a, int *b, int *c, int n): Define the CUDA kernel function vectorAdd, which adds corresponding elements of input vectors a and b and stores the result in the output vector c. This function will be executed on the GPU.

   Main Function:

   int main(): Define the main function.

   Vector Size and Host Vectors:

   int n = 1024;: Define the size of the vectors.

int *h_a, *h_b, *h_c;: Declare host pointers for input vectors a, b, and the output vector c.

h_a = new int[n];, h_b = new int[n];, h_c = new int[n];: Allocate memory on the host for vectors a, b, and c.

Initialize Input Vectors:

for (int i = 0; i < n; i++) { h_a[i] = i; h_b[i] = i * 2; }: Initialize input vectors a and b with some values.

Allocate Device Memory:

int *d_a, *d_b, *d_c;: Declare device pointers for vectors a, b, and c.

cudaMalloc((void*)&d_a, n * sizeof(int));, cudaMalloc((void)&d_b, n * sizeof(int));, cudaMalloc((void*)&d_c, n * sizeof(int));: Allocate memory on the GPU for vectors a, b, and c.

Copy Input Data from Host to Device:

cudaMemcpy(d_a, h_a, n * sizeof(int), cudaMemcpyHostToDevice);: Copy input vector a from host to GPU.

cudaMemcpy(d_b, h_b, n * sizeof(int), cudaMemcpyHostToDevice);: Copy input vector b from host to GPU.

Launch CUDA Kernel:

int threadsPerBlock = 256;: Define the number of threads per block.

int blocksPerGrid = (n + threadsPerBlock - 1) / threadsPerBlock;: Calculate the number of blocks needed in the grid.

vectorAdd<<<blocksPerGrid, threadsPerBlock>>>(d_a, d_b, d_c, n);: Launch the vectorAdd CUDA kernel on the GPU.

Copy Output Data from Device to Host:

cudaMemcpy(h_c, d_c, n * sizeof(int), cudaMemcpyDeviceToHost);: Copy the output vector c from GPU to host.

Print Results:

for (int i = 0; i < 10; i++) { std::cout << h_c[i] << " "; }: Print the first 10 elements of vector c.

Free Device and Host Memory:

cudaFree(d_a);, cudaFree(d_b);, cudaFree(d_c);: Free memory allocated for vectors a, b, and c on the GPU.

delete[] h_a;, delete[] h_b;, delete[] h_c;: Free memory allocated for vectors a, b, and c on the host.

Return 0:

return 0;: Indicate successful program completion

8. Explain CUDA thread organization.

CUDA thread organization refers to how threads are arranged and managed within CUDA programs for execution on the GPU. Understanding CUDA thread organization is crucial for efficiently utilizing the computational power of the GPU. Here are the key aspects of CUDA thread organization:

Threads: Threads are the smallest units of execution in CUDA programs. Each thread executes a specific portion of the overall computation.

Thread Hierarchy:

Grid: A grid is the top-level organization of threads in a CUDA program. It consists of multiple thread blocks.

Thread Block: A thread block is a group of threads that can cooperate with each other via shared memory and synchronization. All threads within a block have access to the same resources and can synchronize with each other.

Threads: Threads within a block are organized in one, two, or three dimensions, forming a thread block grid. Each thread has a unique identifier within its block.

Thread Indexing:

Thread Index: Each thread is identified by its unique index within its block. In a one-dimensional block grid, threads are indexed from 0 to (blockDim.x - 1). Similarly, in two or three-dimensional block grids, threads are indexed in a multidimensional manner.

Block Index: Each block is identified by its unique index within the grid. In a one-dimensional grid, blocks are indexed from 0 to (gridDim.x - 1). In multidimensional grids, blocks are indexed accordingly.

Global Thread Index: Threads are often referenced using their global thread index, which is a unique combination of block index and thread index within the grid. This index allows threads to access unique portions of data or perform specific computations based on their position within the grid.

Thread Synchronization:

Threads within the same block can synchronize their execution using synchronization primitives such as barriers (__syncthreads()). Synchronization ensures that all threads have finished executing certain instructions before proceeding.

Threads from different blocks cannot directly synchronize with each other.

Memory Hierarchy:

Each thread has access to different types of memory:

Registers: Each thread has its own set of registers for storing temporary data.

Shared Memory: Threads within the same block can share data through shared memory, which has low latency and is visible to all threads within the block.

Global Memory: All threads in the grid have access to global memory, which is the main memory space on the GPU. Accessing global memory is slower compared to shared memory.

Local Memory: Local memory is used for storing local variables and is slower than shared memory.

Thread Divergence:

Thread divergence occurs when different threads within the same block take different execution paths based on conditional statements. This can lead to inefficiencies due to serialization of execution.

Overall, understanding CUDA thread organization allows developers to effectively utilize the parallelism offered by GPUs and optimize their CUDA programs for performance.

9. Install and try CUDA sample program and explain the same. (installation steps).

   Not able to install CUDA in local system due to unavailability of NVIDIA GPU.