

## HPC – Unit 5: GPU Architecture & CUDA Programming

May–June 2023 (Paper [6004]-493)

---

### Q5 a) CUDA: Programming Languages, and Three Applications

[8 Marks]

#### What is CUDA?

CUDA (Compute Unified Device Architecture) is NVIDIA's parallel computing platform and programming model that extends standard programming languages (primarily C/C++) to allow programs to harness the massively parallel processing power of NVIDIA GPUs. CUDA was introduced in 2006 and has since become the dominant framework for GPU computing.

CUDA exposes the GPU as a co-processor (called the device) that works alongside the CPU (called the host). The programmer writes special functions called kernels that execute in parallel across thousands of GPU threads simultaneously.

#### Programming Languages Supported in CUDA

- **CUDA C/C++:** The primary and most widely used language. Standard C/C++ code with CUDA-specific extensions including kernel launch syntax (`<<<...>>>`), qualifier keywords (`__global__`, `__device__`, `__shared__`), and built-in variables (`threadIdx`, `blockIdx`, `gridDim`, `blockDim`).
- **CUDA Fortran:** Supported via PGI/NVIDIA HPC compilers. Used in scientific computing communities (meteorology, CFD) that have legacy Fortran codebases.
- **Python (PyCUDA / CuPy / Numba CUDA):** PyCUDA provides Python bindings to the CUDA driver API. CuPy is a NumPy-compatible array library that uses CUDA internally. Numba allows writing CUDA kernels directly in Python using the `@cuda.jit` decorator, without leaving the Python ecosystem.
- **OpenCL:** While not CUDA itself, OpenCL is a competing open standard supported on NVIDIA GPUs via CUDA's underlying driver. It provides GPU programming in a vendor-neutral way.
- **CUDA Libraries (cuBLAS, cuDNN, cuFFT):** Precompiled CUDA libraries callable from C, Python, or MATLAB without writing explicit kernels. cuDNN is the backbone of every major deep learning framework (TensorFlow, PyTorch).

#### Three Major Applications of CUDA

**Application 1 — Deep Learning and Neural Networks:** Every major deep learning framework (TensorFlow, PyTorch, MXNet) runs training and inference on NVIDIA GPUs via CUDA. The forward pass (matrix multiplications, convolutions) and backward pass (gradient computation) of neural networks are ideal for GPU parallelism. NVIDIA's cuDNN library provides optimised implementations of convolution, pooling, batch normalisation, and RNN primitives. A single A100 GPU can deliver ~312 teraFLOPS for AI workloads — thousands of times faster than a single CPU core.

**Application 2 — Medical Imaging and CT Reconstruction:** CT scan reconstruction requires computing the inverse Radon transform over millions of data points. CUDA allows real-time or near-real-time reconstruction by parallelising the back-projection algorithm across thousands of GPU threads. Each thread handles one voxel's contribution, reducing reconstruction time from minutes (CPU) to seconds (GPU).

**Application 3 — Computational Finance (Monte Carlo Simulation):** Options pricing, risk modelling, and portfolio simulation using Monte Carlo methods require simulating millions of independent random

paths. Since each simulation path is independent, they map perfectly onto GPU threads. CUDA-based Monte Carlo can achieve 100x–500x speedups over CPU implementations, enabling real-time risk analysis in trading systems.

*Note: For SPPU exams, any three CUDA applications with a brief explanation each are acceptable. Other popular choices include: molecular dynamics simulation, weather forecasting, image processing, and cryptography.*

## Q5 b) Processing Flow of a CUDA-C Program

[6 Marks]

A CUDA-C program follows a well-defined flow that alternates between the CPU (host) and GPU (device). Understanding this flow is critical for writing correct and efficient CUDA programs.

CUDA-C Program Processing Flow	
1. CPU: Initialise data (arrays, matrices, etc.) <code>int h_A[N]; // host array</code>	
2. CPU: Allocate GPU memory <code>cudaMalloc(&amp;d_A, N * sizeof(int));</code>	
3. CPU → GPU: Copy input data to device <code>cudaMemcpy(d_A, h_A, size, cudaMemcpyH2D);</code>	
4. CPU: Configure and launch kernel <code>myKernel&lt;&lt;&lt;gridDim, blockDim&gt;&gt;&gt;&gt;(d_A, d_B, d_C);</code>	
5. GPU: Kernel executes in parallel (thousands threads) <pre> __global__ void myKernel(int *A, int *B, ...) {     int i = blockDim.x * blockIdx.x + threadIdx.x;     C[i] = A[i] + B[i]; } </pre>	
6. CPU: Synchronise (wait for kernel to finish) <code>cudaDeviceSynchronize();</code>	
7. GPU → CPU: Copy results back to host <code>cudaMemcpy(h_C, d_C, size, cudaMemcpyD2H);</code>	
8. CPU: Process results, free GPU memory <code>cudaFree(d_A); cudaFree(d_B); cudaFree(d_C);</code>	

- `cudaMalloc` allocates memory on the GPU's DRAM (global memory).
- `cudaMemcpy` transfers data over the PCIe bus between host and device memory.
- The kernel launch `<<<gridDim, blockDim>>>` specifies the execution configuration — how many thread blocks and threads per block.
- `cudaDeviceSynchronize` is a barrier: the CPU waits here until the GPU kernel has completely finished before proceeding.
- `cudaFree` releases GPU memory — always required to prevent GPU memory leaks.

*Note: The PCIe data transfer (steps 3 and 7) is often the bottleneck. Minimise transfers by keeping data on the GPU for as long as possible across multiple kernel calls. NVIDIA's Unified Memory (`cudaMallocManaged`) can automate transfers but may add overhead.*

**Q5 c) CUDA Terms: Device, Host, Device Code, Kernel****[4 Marks]**

CUDA defines a clear distinction between the CPU-side and GPU-side of computation, and uses specific terminology to refer to each:

**Host**

The host refers to the CPU and its associated main memory (RAM). The host runs the main program, manages the overall program flow, allocates/frees GPU memory, and transfers data between CPU and GPU. Code running on the host is standard C/C++ compiled by the regular compiler (gcc/clang).

**Device**

The device refers to the GPU and its dedicated high-bandwidth memory (GDDR/HBM). The device executes kernels in parallel across thousands of threads. The GPU is a co-processor — it does not run the main program independently; it only executes when the host explicitly launches work on it.

**Device Code**

Device code is code that runs on the GPU. In CUDA-C, it is distinguished from host code by qualifier keywords:

- `__global__` — marks a function as a kernel (callable from host, executed on device).
- `__device__` — marks a helper function callable only from device code (not from the host).
- `__host__` — explicitly marks a function as host-only (the default; can be combined with `__device__` to compile for both).
- `__shared__` — marks a variable as shared memory (fast, on-chip, shared among threads in the same block).

**Kernel**

A kernel is a function that is executed in parallel across many GPU threads simultaneously. Each thread runs the same kernel code but operates on different data (SIMT — Single Instruction, Multiple Threads model). The kernel is defined with the `__global__` qualifier and launched from the host using the special `<<<gridDim, blockDim>>>` syntax.

```
__global__ void vectorAdd(float *A, float *B, float *C, int n) {
    int i = blockDim.x * blockIdx.x + threadIdx.x; // unique thread ID
    if (i < n) C[i] = A[i] + B[i];                // each thread adds one pair
}
```

```
// Host launch: 256 blocks of 256 threads each = 65536 total threads
vectorAdd<<<256, 256>>>>(d_A, d_B, d_C, n);
```

*Note: The key insight is that the kernel appears to the programmer as a single function, but actually runs as thousands of independent threads concurrently. The thread's unique index (computed from `blockIdx` and `threadIdx`) is how each thread knows which data element to process.*

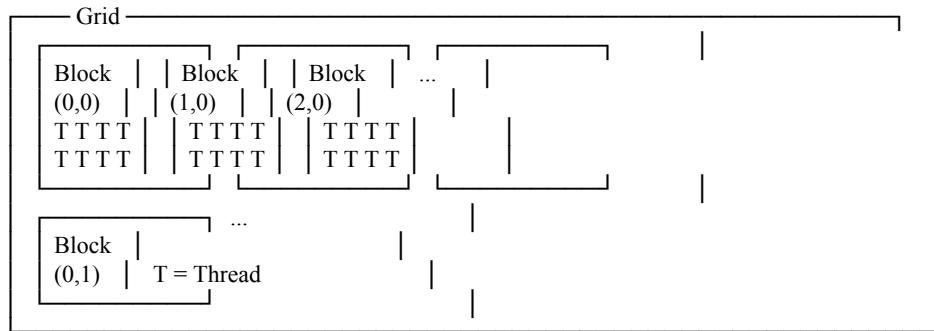
**Q6 a) CUDA Memory Model and Thread Hierarchy****[8 Marks]****Thread Hierarchy**

CUDA organises parallel threads in a three-level hierarchy: threads, blocks, and grids. This hierarchy maps directly to the GPU hardware.

- Thread: The smallest unit of execution. Each thread runs the kernel function independently with

its own registers and local memory. A thread is identified within its block by threadIdx (a 3D index: threadIdx.x, threadIdx.y, threadIdx.z).

- **Block (Thread Block):** A group of threads (up to 1024) that are guaranteed to run on the same SM. Threads within a block can synchronise using `__syncthreads()` and communicate via fast shared memory. Blocks are identified by blockIdx.x, blockIdx.y, blockIdx.z.
- **Grid:** A collection of blocks executing the same kernel. Blocks in a grid are independent — they cannot synchronise directly with each other (only via global memory atomics or kernel boundaries). Grid dimensions are specified by gridDim.



Global thread index formula (1D case):

```
int globalIdx = blockIdx.x * blockDim.x + threadIdx.x;
```

For a grid of 4 blocks, each with 256 threads:

Total threads =  $4 \times 256 = 1024$

Thread in block 2, position 10  $\rightarrow$  globalIdx =  $2 \times 256 + 10 = 522$

## CUDA Memory Model

CUDA defines several types of memory, each with different scope, lifetime, speed, and programmer control:

- **Registers:** Fastest memory, private to each thread, allocated automatically by the compiler. Variables declared inside a kernel without qualifiers go into registers. Extremely fast — zero extra latency.
- **Local Memory:** When a thread uses too many variables to fit in registers (register spilling), or uses large arrays, the excess goes to local memory. Despite the name, local memory is physically in global DRAM and is slow. The compiler handles this transparently.
- **Shared Memory (`__shared__`):** On-chip memory shared by all threads in a block. Declared with `__shared__` qualifier.  $\sim 100\times$  faster than global memory. Used for inter-thread communication within a block and as a programmable cache. Must be managed manually by the programmer.
- **Global Memory (`cudaMalloc`):** The largest memory space (GBs), off-chip GDDR/HBM. Accessible by all threads and by the host (via `cudaMemcpy`). High latency ( $\sim 500$  cycles) but high bandwidth. Most CUDA programs' bottleneck.
- **Constant Memory (`__constant__`):** Read-only from device, cached. Ideal for broadcasting a single value to all threads (e.g., a filter kernel in image processing). Only 64KB.
- **Texture Memory:** Read-only, spatially cached (optimised for 2D locality). Useful for image processing and lookup tables.

*Note: The golden rule of CUDA memory optimisation: access global memory as infrequently as possible. Load data into shared memory first, work on it there, and write results back to global memory at the end. This pattern is called shared memory tiling.*

**Q6 b) Block Dimension, Grid Dimension, and Vector Addition Kernel [6 Marks]****Block Dimension and Grid Dimension**

When launching a CUDA kernel, the programmer specifies two execution configuration parameters inside the <<< >>> brackets: the grid dimension (number of blocks) and the block dimension (number of threads per block).

- **blockDim:** A built-in dim3 variable giving the size of each block in threads. E.g., blockDim.x = 256 means 256 threads per block in the x-dimension. Can be up to (1024, 1024, 64) with the product  $\leq 1024$ .
- **gridDim:** A built-in dim3 variable giving the number of blocks in each dimension of the grid. E.g., gridDim.x = 128 means 128 blocks in the x-direction.
- **Total threads in 1D:** gridDim.x  $\times$  blockDim.x.

Choosing the right block size matters for performance — 128 or 256 threads per block is a common starting point that gives good occupancy on most GPUs.

**CUDA Kernel for Vector Addition**

```
#include <stdio.h>
#include <cuda_runtime.h>

// Kernel: each thread adds one element of A and B, stores in C
__global__ void vectorAdd(float *A, float *B, float *C, int n) {
    // Compute the global thread index
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    // Guard: only process valid indices
    if (i < n) {
        C[i] = A[i] + B[i];
    }
}

int main() {
    int n = 1024;
    int size = n * sizeof(float);

    // 1. Allocate and initialise host arrays
    float *h_A = (float*)malloc(size);
    float *h_B = (float*)malloc(size);
    float *h_C = (float*)malloc(size);
    for (int i = 0; i < n; i++) { h_A[i] = i; h_B[i] = 2*i; }

    // 2. Allocate device memory
    float *d_A, *d_B, *d_C;
    cudaMalloc(&d_A, size);
    cudaMalloc(&d_B, size);
    cudaMalloc(&d_C, size);

    // 3. Copy inputs to device
    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

    // 4. Launch: 4 blocks of 256 threads = 1024 total threads
    int threadsPerBlock = 256;
    int blocksPerGrid = (n + threadsPerBlock - 1) / threadsPerBlock;
    vectorAdd<<<blocksPerGrid, threadsPerBlock>>>>(d_A, d_B, d_C, n);
```

```
// 5. Copy result back
cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);

// 6. Verify and cleanup
printf("C[0]=%.0f, C[n-1]=%.0f\n", h_C[0], h_C[n-1]);
cudaFree(d_A); cudaFree(d_B); cudaFree(d_C);
free(h_A); free(h_B); free(h_C);
}

// How threads compute addition:
// Thread 0 (block 0): i = 0*256+0 = 0 → C[0] = A[0]+B[0]
// Thread 1 (block 0): i = 0*256+1 = 1 → C[1] = A[1]+B[1]
// ...
// Thread 0 (block 3): i = 3*256+0 = 768 → C[768] = A[768]+B[768]
// All 1024 threads run simultaneously → vector addition done in 1 pass
```

*Note: The formula  $(n + \text{threadsPerBlock} - 1) / \text{threadsPerBlock}$  is a ceiling division trick that ensures enough blocks are launched even if  $n$  is not a multiple of  $\text{threadsPerBlock}$ . The if ( $i < n$ ) guard inside the kernel prevents out-of-bounds access for the extra threads.*

## Q6 c) Kernel, Kernel Launch, and Launch Arguments

[4 Marks]

### What is a Kernel?

A kernel is a function defined with the `__global__` qualifier that executes on the GPU device. When launched, it runs across  $N$  instances (threads) simultaneously, each with a unique thread ID used to determine which part of the data to process. Kernels cannot return values directly (return type is always void) and must use output arrays or global memory for results.

### Kernel Launch

A kernel is launched from host code using CUDA's special triple-chevron syntax:

```
kernelName<<<gridDim, blockDim, sharedMemBytes, stream>>>(arg1, arg2, ...);
```

### Arguments in a Kernel Launch

- `gridDim` (dim3 or int): Specifies the number of blocks in the grid. Can be 1D, 2D, or 3D. E.g., `dim3 grid(16, 16)` launches a 16×16 grid of 256 total blocks.
- `blockDim` (dim3 or int): Specifies the number of threads in each block. E.g., `dim3 block(16, 16)` launches 256 threads per block. Maximum 1024 threads per block.
- `sharedMemBytes` (optional, default 0): The number of bytes of dynamically-allocated shared memory per block. Used when the shared memory size is not known at compile time.
- `stream` (optional, default 0): A CUDA stream — a sequence of operations that execute in order on the GPU. Using multiple streams enables concurrent kernel execution and overlapping of computation with memory transfers.

```
// Example launch configurations:
myKernel<<<256, 128>>>(d_data, n); // 256 blocks, 128 threads/block
myKernel<<<dim3(16,16), dim3(16,16)>>>(img); // 2D grid for image processing
myKernel<<<grid, block, 4096, stream1>>>(d); // 4KB dynamic shared mem, stream 1
```

*Note: The kernel launch is asynchronous — it returns to the host immediately after submission, before the kernel finishes. Use `cudaDeviceSynchronize()` or `cudaMemcpy` (which is synchronous by default) to ensure the kernel has completed before accessing its output.*

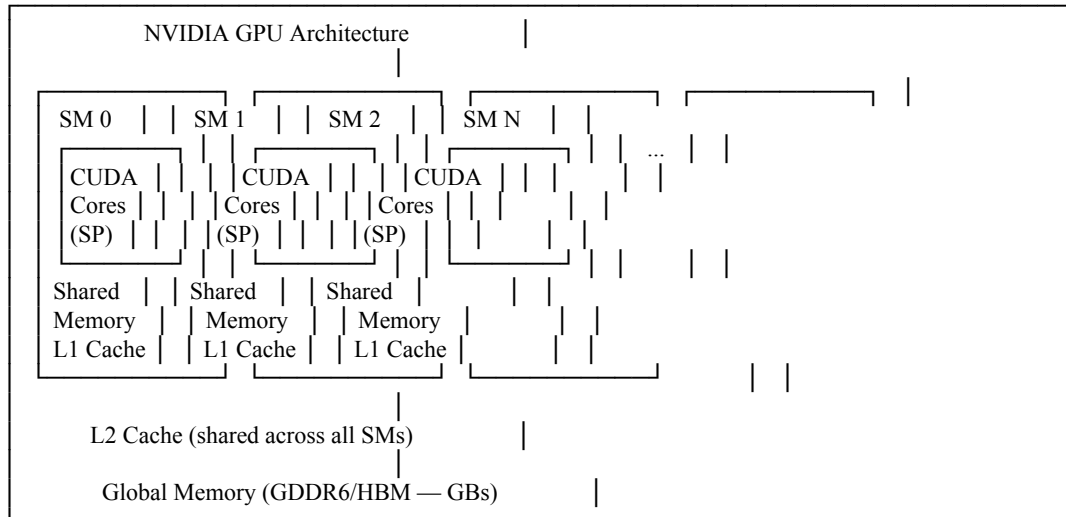
## May–June 2024 (Paper [6263]-94)

### Q5 a) CUDA Architecture in Detail

[8 Marks]

#### High-Level Architecture Overview

A modern NVIDIA GPU consists of multiple Streaming Multiprocessors (SMs), each of which is itself a massively parallel processor. The entire GPU chip sits on a PCIe card and connects to the CPU via a high-speed bus (PCIe Gen4/5 or NVLink for server GPUs).



#### Streaming Multiprocessor (SM)

Each SM is an independent processor containing:

- **CUDA Cores (SP — Scalar Processors):** Each CUDA core executes one floating-point or integer instruction per clock per thread. Modern GPUs have 128 CUDA cores per SM, with 100+ SMs on high-end chips (e.g., A100: 108 SMs × 64 FP64 cores = 6912 FP64 cores).
- **Tensor Cores:** Specialised matrix-multiply-accumulate units. Each Tensor Core performs a 4×4 matrix multiply per clock, delivering 8× more FLOPS than regular CUDA cores for AI workloads.
- **Special Function Units (SFUs):** Handle transcendental functions (sin, cos, sqrt, exp).
- **Warp Scheduler:** Schedules and issues instructions. A warp is a group of 32 threads that execute in lockstep (SIMT). The scheduler can hide memory latency by switching between warps when one warp is waiting for memory.
- **Register File:** Each SM has a large register file (65,536 32-bit registers on Ampere architecture), shared among all active threads.
- **Shared Memory / L1 Cache:** Fast on-chip memory (48KB–96KB per SM, software-configurable between shared memory and L1 cache). Orders of magnitude faster than global memory.

#### Memory Hierarchy

Memory Type	Location	Scope	Speed	Size
Registers	On-chip (SM)	Per-thread	~1 cycle	Tiny (per thread)



Shared Memory	On-chip (SM)	Per-block	~10 cycles	48–96 KB/SM
L1 Cache	On-chip (SM)	Per-SM	~20 cycles	Configurable
L2 Cache	On-chip (GPU)	All threads	~200 cycles	4–40 MB
Global Memory	Off-chip GDDR/HBM	All threads	~500 cycles	8–80 GB
Constant Memory	Off-chip (cached)	All threads, read-only	~20 cycles if cached	64 KB

### Thread Execution Model (SIMT)

Threads are organised hierarchically: threads are grouped into blocks (up to 1024 threads each), and blocks are organised into a grid (up to  $2^{31}-1$  blocks per dimension). All blocks in a grid execute the same kernel. The GPU scheduler maps blocks onto SMs; each SM executes one or more blocks concurrently using its warp schedulers.

*Note: A key design principle: the GPU trades single-thread performance for massive throughput. A single GPU thread is much slower than a CPU thread, but having 10,000+ threads running simultaneously delivers enormous aggregate performance.*

### Q5 b) Processing Flow of CUDA with CUDA-C Functions

[6 Marks]

**[REPEATED] – See Q5 b) in May–June 2023 for the complete processing flow with diagram. The 2024 version additionally expects the CUDA-C function signatures, which are all included in that answer (cudaMalloc, cudaMemcpy, kernel launch, cudaDeviceSynchronize, cudaFree).**

### Q5 c) Advantages and Limitations of CUDA

[4 Marks]

#### Advantages

- Massive parallelism: Modern NVIDIA GPUs have thousands of CUDA cores, enabling parallel execution of tens of thousands of threads simultaneously — orders of magnitude more than CPU cores.
- High memory bandwidth: GPU GDDR6/HBM memory provides 600–2000 GB/s bandwidth vs. CPU DDR5's ~100 GB/s. This is critical for data-intensive tasks.
- Rich ecosystem: cuBLAS, cuDNN, cuFFT, cuSPARSE, Thrust, NCCL, and many other libraries provide highly optimised routines. Deep learning frameworks (PyTorch, TensorFlow) use CUDA internally.
- Energy efficiency: For compute-intensive tasks, GPUs often deliver better FLOPS-per-watt than CPUs.
- Relative ease of programming: CUDA-C extends standard C/C++ with minimal new syntax. A programmer familiar with C can write basic GPU code quickly.
- Profiling tools: NVIDIA Nsight, nvprof, and NCU provide detailed performance analysis.

#### Limitations

- NVIDIA-only: CUDA is proprietary to NVIDIA GPUs. Code written in CUDA does not run on AMD GPUs (which use ROCm/HIP) or Intel GPUs. This creates vendor lock-in.
- Host-device transfer bottleneck: Moving data over the PCIe bus is slow (16–32 GB/s) compared



to GPU memory bandwidth. If the algorithm requires frequent transfers, the speedup is severely limited.

- Not suited for all workloads: Problems with irregular memory access patterns, complex branching, or inherently sequential logic perform poorly on GPUs. CPUs are better for latency-sensitive, single-threaded tasks.
- Memory capacity constraints: Even high-end datacenter GPUs (A100: 80GB, H100: 80GB) have far less memory than CPU systems (multiple TB). Large datasets must be streamed in chunks.
- Debugging difficulty: Race conditions, memory access errors, and synchronisation bugs on the GPU are harder to debug than CPU code. Tools like cuda-memcheck/compute-sanitizer help but add overhead.
- Learning curve for optimisation: Writing correct CUDA code is accessible, but writing high-performance CUDA code (coalesced memory access, shared memory tiling, occupancy tuning) requires deep hardware knowledge.

### Q6 a) How CUDA-C Executes at the Kernel Level (with Example)

[8 Marks]

When a CUDA kernel is launched, the GPU's hardware thread scheduling system takes over:

- The kernel launch places a task in the GPU's command queue for the specified stream.
- The GPU's Giga Thread Engine distributes blocks across available SMs. Each SM gets one or more blocks depending on the resource requirements (registers, shared memory, active threads) and SM capacity.
- Within each SM, the Warp Scheduler groups the block's threads into warps of 32 threads. Each warp executes instructions in SIMT fashion — all 32 threads execute the same instruction each cycle, but on different data.
- If a warp stalls (e.g., waiting for a global memory load which takes ~500 cycles), the scheduler immediately switches to another ready warp. This latency hiding through warp switching is the fundamental reason why GPUs can sustain high throughput despite high memory latency.

Example: Matrix-Multiply inner kernel executing at the kernel level.

```
__global__ void matMul(float *A, float *B, float *C, int N) {
    int row = blockIdx.y * blockDim.y + threadIdx.y; // which row of C
    int col = blockIdx.x * blockDim.x + threadIdx.x; // which col of C
    float sum = 0.0f;
    if (row < N && col < N) {
        for (int k = 0; k < N; k++) {
            sum += A[row * N + k] * B[k * N + col]; // dot product
        }
        C[row * N + col] = sum;
    }
}
```

```
// Launch for N=1024:
dim3 block(16, 16); // 256 threads per block
dim3 grid(64, 64); // 4096 blocks → 4096×256 = 1M threads
matMul<<<grid, block>>>>(dA, dB, dC, 1024);
```

```
// Each thread computes exactly one element of C.
// All 1M threads run concurrently → 1M multiply-adds in one kernel execution.
```

**Q6 b) CUDA Memory Model (Brief)**

[REPEATED] – See Q6 a) in May–June 2023 for the complete CUDA memory model with all memory types, their properties, and the shared memory tiling note.

[6 Marks]

**Q6 c) Applications of CUDA**

[4 Marks]

- Deep Learning / AI: Training and inference of neural networks (CNNs, Transformers, RNNs) using PyTorch, TensorFlow, and JAX — all of which rely on cuDNN and cuBLAS internally.
- Scientific Simulation: Molecular dynamics (GROMACS, NAMD), climate modelling, computational fluid dynamics (CFD), and finite-element analysis — all benefit from GPU parallelism for large-scale simulations.
- Medical Imaging: CT reconstruction, MRI reconstruction, radiotherapy dose calculation (real-time GPU-accelerated back-projection).
- Computer Vision and Image Processing: Real-time video analysis, object detection, SLAM (simultaneous localisation and mapping) for robotics and autonomous vehicles.
- Computational Finance: Monte Carlo risk simulations, options pricing, and real-time fraud detection.
- Cryptography and Blockchain: GPU mining of proof-of-work cryptocurrencies (though now partially replaced by ASICs); GPU-accelerated cryptographic hash functions.
- Genomics and Bioinformatics: DNA sequence alignment (BWA, GATK on GPU), protein folding (AlphaFold uses GPU clusters).

**Additional Concepts & Quick Reference****CUDA Function Qualifiers Summary**

Qualifier	Callable From	Executes On	Use Case
<code>__global__</code>	Host (and device*)	Device (GPU)	Kernels — launched with <code>&lt;&lt;&lt;&gt;&gt;&gt;</code>
<code>__device__</code>	Device only	Device (GPU)	Helper functions called from kernels
<code>__host__</code>	Host only	Host (CPU)	Regular C functions (default)
<code>__host__ __device__</code>	Both	Both	Utility functions needed on CPU and GPU

Note: \*CUDA dynamic parallelism (Compute Capability 3.5+) allows device code to launch kernels, but this is advanced and rarely tested at the BE level.

**Built-in Thread Variables**

Variable	Type	Description
<code>threadIdx</code>	<code>dim3</code>	Thread index within its block (x, y, z components)
<code>blockIdx</code>	<code>dim3</code>	Block index within the grid (x, y, z components)
<code>blockDim</code>	<code>dim3</code>	Dimensions of each block (number of threads per block)
<code>gridDim</code>	<code>dim3</code>	Dimensions of the grid (number of blocks)