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

- 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/cl).

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 = blockIdx.x * blockDim.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]

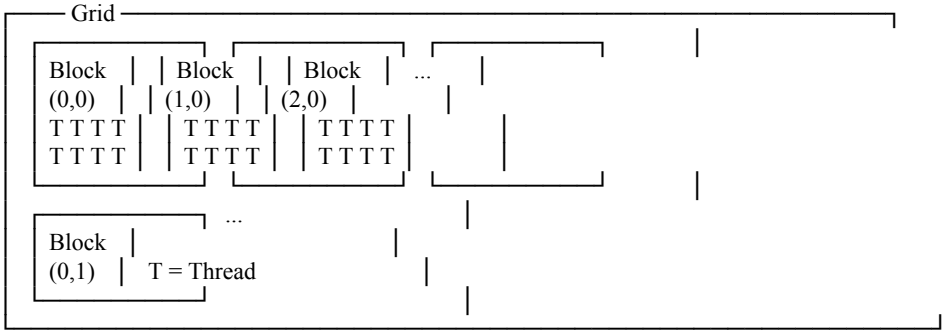
Thread Hierarchy

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 × 256 = 1024
Thread in block 2, position 10 → globalIdx = 2×256 + 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. ~100× 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 (~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 ≤ 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 × 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 + threadsPerBlock - 1) / threadsPerBlock is a ceiling division trick that ensures enough blocks are launched even if n is not a multiple of 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
```

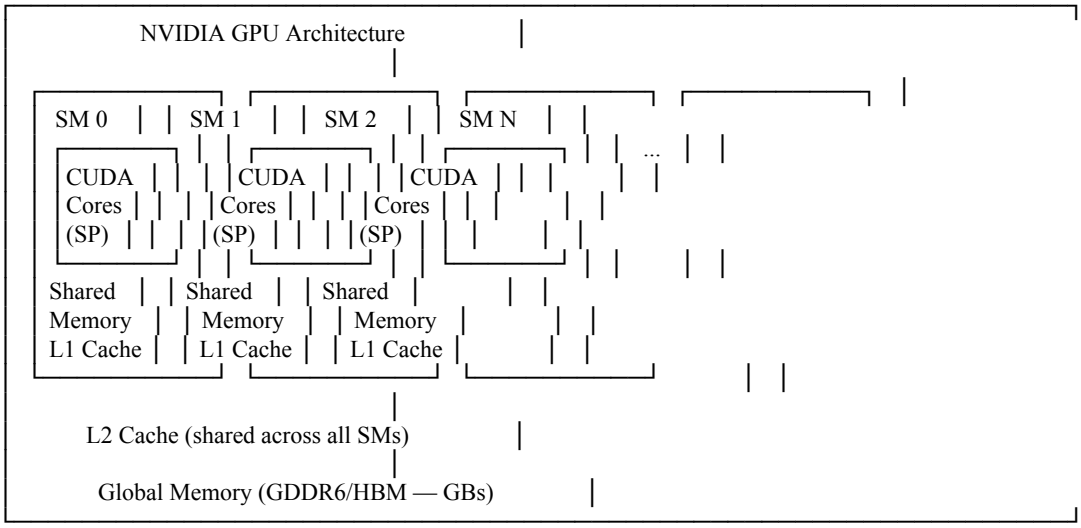
> *Note: The kernel launch is asynchronous — it returns to the host immediately after submission, before the kernel finishes. Use cudaDeviceSynchronize() or cudaMemcpy (which is synchronous by default) to ensure the kernel has completed before accessing its output.*

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) [6 Marks]

[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.

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
__global__	Host (and device*)	Device (GPU)	Kernels — launched with <<<>>>
__device__	Device only	Device (GPU)	Helper functions called from kernels
__host__	Host only	Host (CPU)	Regular C functions (default)
__host__ __device__	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
threadIdx	dim3	Thread index within its block (x, y, z components)
blockIdx	dim3	Block index within the grid (x, y, z components)
blockDim	dim3	Dimensions of each block (number of threads per block)
gridDim	dim3	Dimensions of the grid (number of blocks)

Nov–Dec 2025 (Paper [6584]-81)

Q5 a) CUDA: Programming Languages and Three Applications [8 Marks]

[REPEATED] – Full answer covering CUDA definition, all five language support categories (CUDA C/C++, Fortran, Python/PyCUDA/Numba, OpenCL, CUDA Libraries), and three detailed applications (Deep Learning, CT Reconstruction, Monte Carlo Finance) is in: Unit 5 Answer Doc → May–June 2023 → Q5 a).

Q5 b) Processing Flow of a CUDA-C Program with Diagram [6 Marks]

[REPEATED] – Full 8-step flow with ASCII diagram and all CUDA function signatures (cudaMalloc, cudaMemcpy, kernel launch, cudaDeviceSynchronize, cudaFree) is in: Unit 5 Answer Doc → May–June 2023 → Q5 b).

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

[REPEATED] – Full definitions with qualifier keywords and vector addition code example are in: Unit 5 Answer Doc → May–June 2023 → Q5 c).

Q6 a) Compare-Exchange and Compare-Split Operations on Parallel Computers [8 Marks]

Compare-exchange and compare-split are the fundamental primitives that parallel sorting networks are built upon. They define how two processors (or nodes) interact to merge or order their data. Understanding these two operations is the key to understanding bitonic sort, odd-even merge, and other sorting networks.

Compare-Exchange Operation

A compare-exchange is an operation performed between exactly two processors, say P_i and P_j (where i < j). Each processor holds a single element. After the operation, P_i holds the minimum of the two values and P_j holds the maximum, ensuring the pair is locally sorted in ascending order (P_i ≤ P_j).

```
Compare-Exchange(P_i, P_j):  — assumes P_i communicates with P_j

  P_i sends its value to P_j, and simultaneously P_j sends its value to P_i.
  (full-duplex exchange — both happen at the same time)

  After exchange:
    P_i keeps: MIN(original_P_i, original_P_j)   [smaller value goes 'left']
    P_j keeps: MAX(original_P_i, original_P_j)   [larger value goes 'right']

Example: P_0 = 7,  P_1 = 3
  Exchange: P_0 sends 7 to P_1, P_1 sends 3 to P_0
  After:    P_0 = MIN(7,3) = 3,  P_1 = MAX(7,3) = 7 ✓

Example: P_0 = 2,  P_1 = 9  (already in order — no effective change)
  After:    P_0 = MIN(2,9) = 2,  P_1 = MAX(2,9) = 9 ✓
```

The compare-exchange is the building block of sorting networks. By composing many compare-exchange operations in a specific wiring pattern, we can sort any permutation of n elements. The key

property is that a sequence of compare-exchange operations forms a sorting network if and only if it correctly sorts all 2^n binary (0,1) sequences — this is the 0-1 Principle.

- Cost: one compare-exchange takes exactly 1 communication step (one send + one receive) regardless of how far apart P_i and P_j are in the network topology.
- The 0-1 Principle: if a network of compare-exchange operations correctly sorts all possible 0/1 input sequences, it correctly sorts all sequences of arbitrary values. This is enormously useful for proving sorting network correctness.

Compare-Split Operation

While compare-exchange operates on single elements (one per processor), compare-split generalises this to the case where each processor holds a sorted list of elements rather than a single value. This is the operation used when processors already have local sorted sequences and need to merge two sorted halves across processor boundaries.

```
Compare-Split(P_i, P_j):  — each processor holds n/p sorted elements

  P_i and P_j exchange their full sorted lists with each other.
  After the exchange, each processor has 2n/p elements total.

  P_i (the 'lower' processor) merges the two lists and keeps the
    LOWER HALF of the merged result (the n/p smallest elements).

  P_j (the 'upper' processor) keeps the UPPER HALF of the merged
    result (the n/p largest elements).

Example: n=8 elements on 2 processors, each with a sorted half.
  P_0 holds sorted: [1, 3, 5, 7]
  P_1 holds sorted: [2, 4, 6, 8]

  Exchange: P_0 sends [1,3,5,7] to P_1; P_1 sends [2,4,6,8] to P_0.
  Both merge: [1, 2, 3, 4, 5, 6, 7, 8]
  P_0 keeps lower half: [1, 2, 3, 4]
  P_1 keeps upper half: [5, 6, 7, 8]  ✓
```

The compare-split is the fundamental step in parallel merge sort and bitonic sort when the number of elements per processor is greater than 1. After all compare-split operations in the sorting network have been executed, each processor holds a contiguous sorted range of the global sequence.

Key Differences Between the Two Operations

Aspect	Compare-Exchange	Compare-Split
Elements per processor	1 (a single value)	n/p (a sorted sequence)
Communication	Exchange 1 element	Exchange n/p elements (entire local array)
After operation	Each proc has 1 element (min/max)	Each proc has n/p elements (lower/upper half)
Use case	Sorting networks (bitonic sort)	Parallel merge sort, parallel quicksort
Local work	None (trivial min/max)	O((n/p) log(n/p)) merge step
Cost	O(1) comp + O(1) comm	O(n/p) comp + O(n/p) comm

Application: Bitonic Sort

Bitonic sort is a classic parallel sorting network built entirely from compare-exchange operations. For n elements on n processors, it uses $\log_2(n)$ stages, each containing multiple rounds of compare-exchange. Each stage i has i rounds, giving a total of $1+2+...+\log_2(n) = \log_2(n)\times(\log_2(n)+1)/2$ rounds of compare-exchange. The parallel time is therefore $O(\log^2 n)$, compared to $O(n \log n)$ sequentially.

Example: 4-element bitonic sort on processors P0..P3, values [4,2,1,3]

Stage 1 (form bitonic sequences of length 2):
Round 1: CE(P0,P1) ascending + CE(P2,P3) descending
P0,P1: CE(4,2) → P0=2,P1=4
P2,P3: CE(1,3) → P2=3,P3=1 (descending: larger stays at P2)
After Stage 1: [2, 4, 3, 1] — bitonic sequence

Stage 2 (sort entire sequence of length 4):
Round 1: CE(P0,P2), CE(P1,P3)
CE(2,3)→P0=2,P2=3; CE(4,1)→P1=1,P3=4
After: [2,1,3,4]
Round 2: CE(P0,P1), CE(P2,P3)
CE(2,1)→P0=1,P1=2; CE(3,4)→P2=3,P3=4
After: [1,2,3,4] ✓ Sorted!

> *Note: Compare-split is the 'bulk' version of compare-exchange for when you're working with sorted sublists rather than single elements. You will see compare-exchange used in bitonic sort diagrams, and compare-split used in parallel merge sort discussions. Both operations require a full-duplex exchange of data between the two processors involved — this is a key exam point.*

Q6 b) Issues in Sorting on Parallel Computers [6 Marks]

[REPEATED] – Full answer covering load imbalance, communication overhead, merge bottleneck, scalability, and the sample sort example is in: Unit 6 Answer Doc → May–June 2023 → Q8 b). Note: this question appears in the Unit 5/6 boundary but covers sorting, which is Unit 6 syllabus content.

Q6 c) Odd-Even Transposition in Bubble Sort using Parallel Formulation [4 Marks]

[REPEATED] – Full answer with step-by-step 4-element example [3,1,4,2], complexity analysis O(n) parallel rounds, and the 0-1 principle note is in: Unit 6 Answer Doc → May–June 2023 → Q7 a).

May–June 2025 (Paper [6404]-94)

Q5 a) CUDA Architecture in Detail [9 Marks]

[REPEATED] – Full GPU architecture answer with SM structure, memory hierarchy table, SIMT execution model, and ASCII architecture diagram is in: Unit 5 Answer Doc → May–June 2024 → Q5 a).

Q5 b) Managing GPU Memory [9 Marks]

GPU memory management is one of the most critical aspects of writing efficient CUDA programs. The GPU has its own dedicated memory (GDDR/HBM) that is physically separate from the host CPU's

RAM, and the programmer must explicitly control when data moves between the two. Poor memory management is the single most common cause of slow CUDA programs.

1. Allocating and Freeing GPU Memory

Unlike CPU memory where malloc/free manage RAM, GPU memory uses dedicated CUDA runtime functions:

```
// Allocate n bytes on GPU global memory
cudaError_t cudaMalloc(void **devPtr, size_t n);
// devPtr is a pointer-to-pointer: cudaMalloc fills it with the GPU address

// Free previously allocated GPU memory
cudaError_t cudaFree(void *devPtr);

// Example: allocate a 1024-float array on GPU
float *d_array;
cudaMalloc((void**)&d_array, 1024 * sizeof(float));
// ... use d_array in kernels ...
cudaFree(d_array);    // always free to prevent GPU memory leak
```

- cudaMalloc does NOT zero-initialise memory. Use cudaMemset(devPtr, value, n) to initialise, similar to memset on the CPU.
- GPU memory is a finite, shared resource. On a typical GPU, global memory ranges from 8 GB to 80 GB. Exceeding it causes cudaMalloc to return cudaErrorMemoryAllocation — always check return codes!

2. Transferring Data: cudaMemcpy

Since GPU and CPU have separate physical memory spaces, data must be explicitly copied between them over the PCIe bus. The cudaMemcpy function handles this:

```
cudaMemcpy(dst, src, n, direction);

Directions:
  cudaMemcpyHostToDevice (H2D): RAM → GPU memory
  cudaMemcpyDeviceToHost (D2H): GPU memory → RAM
  cudaMemcpyDeviceToDevice (D2D): GPU memory → GPU memory (fast, on-chip)
  cudaMemcpyHostToHost (H2H): RAM → RAM (rarely used, same as memcpy)

// Full workflow example:
float h_A[N], h_B[N], h_C[N];  // host arrays
float *d_A, *d_B, *d_C;        // device pointers
cudaMalloc(&d_A, N*sizeof(float));
cudaMalloc(&d_B, N*sizeof(float));
cudaMalloc(&d_C, N*sizeof(float));
cudaMemcpy(d_A, h_A, N*sizeof(float), cudaMemcpyHostToDevice);  // H2D
cudaMemcpy(d_B, h_B, N*sizeof(float), cudaMemcpyHostToDevice);  // H2D
vectorAdd<<<(N+255)/256, 256>>>(d_A, d_B, d_C, N);  // launch kernel
cudaMemcpy(h_C, d_C, N*sizeof(float), cudaMemcpyDeviceToHost);  // D2H
```

- cudaMemcpy is synchronous by default — it blocks the CPU until the transfer completes. Use cudaMemcpyAsync with a CUDA stream for overlapping transfers with computation.
- PCIe bandwidth (≈16–32 GB/s) is 10–50× slower than GPU global memory bandwidth (≈600–2000 GB/s). Minimise host-device transfers — the most effective optimisation in CUDA programming is to restructure algorithms to keep data on the GPU as long as possible.

3. Unified Memory (cudaMallocManaged)

CUDA 6.0 introduced Unified Memory, which creates a single managed memory space accessible from both CPU and GPU. The CUDA runtime automatically migrates pages between CPU and GPU memory on demand, simplifying programming at the cost of some performance overhead.

```
float *data;
cudaMallocManaged(&data, N*sizeof(float));  // accessible on both CPU and GPU

// CPU can access directly:
for (int i = 0; i < N; i++) data[i] = i;

// GPU can access in kernel:
myKernel<<<grid, block>>>(data, N);
cudaDeviceSynchronize();  // ensure GPU done before CPU reads back

// No explicit cudaMemcpy needed!
printf("data[0] = %f\n", data[0]);
cudaFree(data);
```

- Advantage: eliminates manual H2D/D2H transfers, reducing code complexity.
- Disadvantage: page migration overhead can hurt performance if CPU and GPU frequently access the same pages. Best used when the access pattern is clear and non-overlapping.

4. Types of GPU Memory and How to Manage Each

Memory Type	How to Allocate	How to Access	Lifetime
Global	cudaMalloc / cudaMallocManaged	Any thread, any kernel	Until cudaFree
Shared	__shared__ keyword in kernel	Threads in same block	Duration of kernel block
Constant	cudaMemcpyToSymbol()	All threads (read-only, cached)	Duration of program
Local	Automatic (register overflow)	Single thread only	Duration of kernel
Texture	cudaBindTexture() / Texture Objects	All threads (read-only, 2D cached)	Until unbound

5. Shared Memory Management

Shared memory is the most powerful tool for reducing global memory accesses. Since shared memory is on-chip (48–96 KB per SM), accessing it is roughly 100× faster than global memory. The programming pattern is called 'shared memory tiling':

```
__global__ void tiledMatMul(float *A, float *B, float *C, int N) {
  __shared__ float tileA[TILE][TILE];  // tile of A in shared memory
  __shared__ float tileB[TILE][TILE];  // tile of B in shared memory

  int row = blockIdx.y*TILE + threadIdx.y;
  int col = blockIdx.x*TILE + threadIdx.x;
  float sum = 0.0f;

  for (int t = 0; t < N/TILE; t++) {
    // Cooperatively load tiles from global memory into shared memory
    tileA[threadIdx.y][threadIdx.x] = A[row*N + t*TILE + threadIdx.x];
    tileB[threadIdx.y][threadIdx.x] = B[(t*TILE+threadIdx.y)*N + col];
    __syncthreads();  // wait for all threads to finish loading
```

```
        // Compute using fast shared memory
        for (int k = 0; k < TILE; k++)
            sum += tileA[threadIdx.y][k] * tileB[k][threadIdx.x];
        __syncthreads();  // wait before overwriting tiles
    }
    C[row*N + col] = sum;
}

// __syncthreads() is critical: it's a block-level barrier ensuring all
// threads in the block reach the same point before any proceeds.
```

> *Note: Bank conflicts: shared memory is organised into 32 banks (one per warp thread). If multiple threads in a warp access the same bank (but different addresses), the accesses are serialised, degrading performance. Avoid bank conflicts by ensuring stride-1 access patterns or using padding in shared memory arrays.*

Q6 a) Modify DFS for Parallel Execution and Analyse Complexity [9 Marks]

[Core parallel DFS algorithm REPEATED] – The work-stealing approach, challenges, pseudocode, and performance discussion are in: Unit 6 Answer Doc → May–June 2023 → Q7 b).

The 2025 paper specifically asks to 'modify' DFS and 'analyse complexity', so here is an extended treatment of both aspects:

Modifications Required to Parallelise Sequential DFS

Sequential DFS is based on a single global stack that cannot be split safely because each stack entry depends on the context established by popping previous entries. To parallelise it, we make the following structural changes:

- Replace the single global stack with p independent local stacks, one per processor. Each processor runs DFS independently on its local stack.
- Add a work-stealing protocol: when a processor's local stack becomes empty, rather than terminating, it attempts to steal unexplored subtrees from another processor's stack. It steals from the bottom of the victim's stack (not the top), because bottom entries represent larger, more independent subtrees — stealing the top would just steal the next sequential node, not a useful chunk of work.
- Replace sequential 'visited' marking with thread-safe atomic operations: use atomic compare-and-swap (CAS) to mark a node as visited, ensuring only one processor claims each node even when multiple processors might discover it simultaneously.
- Add a global termination detection protocol (e.g., Dijkstra's token-ring algorithm) to recognise when all stacks are empty and no work-stealing is in progress — this is the signal that DFS has completed.

Complexity Analysis

Let V = vertices, E = edges, p = processors.

Sequential DFS: $T_s = O(V + E)$

Parallel DFS with work stealing:
Computation per processor: $O((V + E) / p)$ [ideal, evenly distributed]

Work-stealing overhead:
Each steal moves one subtree between processors.
At most O(V) steals can occur (each node stolen at most once).
Each steal costs O(t_steal) = O(t_s + branch_size × t_w)

Total parallel time:
$T_p = O((V + E) / p) + O(p \times t_{steal})$ [if steals are cheap]

Speedup: $S = T_s / T_p = O((V+E)) / O((V+E)/p + p \cdot t_{steal})$
For $(V+E) >> p^2 \cdot t_{steal}$: $S \approx O(p)$ [near-linear speedup]
When graph is a linear chain: $S \approx O(1)$ [no exploitable parallelism]

Efficiency: $E = S/p$ — depends heavily on graph structure.
Best case (balanced tree): $E \approx 1 - O(p \cdot t_{steal} / (V+E))$
Worst case (linear path): $E = O(1/p)$ [poor]

> *Note: The critical insight for the exam: parallel DFS has O(V+E) total work regardless of p (no redundant computation), but extracting that parallelism requires the work-stealing overhead. The algorithm is work-optimal but not always time-optimal because the critical path length (the longest sequential chain of dependencies) limits achievable speedup.*

Q6 b) Dijkstra's Algorithm in Parallel Formulation [9 Marks]

Sequential Dijkstra's Algorithm Recap

Dijkstra's algorithm finds the shortest paths from a single source s to all other vertices in a weighted graph with non-negative edge weights. The sequential algorithm maintains a priority queue of (distance, vertex) pairs, greedily extracting the minimum-distance unvisited vertex and relaxing its edges.

```
Sequential Dijkstra:
  dist[s] = 0; dist[v] = ∞ for all v ≠ s
  priority_queue Q = {(0, s)}
  while Q is not empty:
    (d, u) = Q.extract_min()       // O(log V) with binary heap
    for each edge (u, v) with weight w:
      if dist[u] + w < dist[v]:
        dist[v] = dist[u] + w
        Q.decrease_key(v, dist[v])  // O(log V)
  Total: O((V + E) log V) with binary heap
        O(V² )          with simple array (better for dense graphs)
```

Challenges in Parallelising Dijkstra

Dijkstra has a fundamental sequential dependency: the next vertex to process must be the one with the globally minimum current distance. This means the 'extract-minimum' step cannot be trivially parallelised — you cannot process multiple vertices simultaneously if they might have the same minimum distance and affect each other's relaxation. This is the key difficulty.

Parallel Formulation: Partitioned Priority Queue

The most common parallel approach is to distribute the vertex set across p processors, each maintaining its own local portion of the distance array and a local priority queue. The algorithm proceeds in rounds:

```
Parallel Dijkstra (p processors, each owns V/p vertices):

Initialise: dist[s]=0 on P(s), dist[v]=∞ elsewhere
```

```
repeat V times:
  Step 1 — Local minimum: each processor finds the minimum-distance
           unvisited vertex among its V/p assigned vertices.
           Local cost: O(V/p)

  Step 2 — Global minimum: all-reduce (MPI_Allreduce with MIN operator)
           to find the globally minimum-distance vertex u across all procs.
           Cost: O(log p × t_s)

  Step 3 — Broadcast u: the processor owning u broadcasts u and dist[u]
           to all other processors.
           Cost: O(log p × t_s)

  Step 4 — Edge relaxation: each processor checks its V/p vertices:
           for each vertex v owned by this processor:
             if edge (u, v) exists with weight w:
               dist[v] = min(dist[v], dist[u] + w)
           Local cost: O(V/p) [for dense graph, O(E/p) for sparse]

Total parallel time:
  T_p = V × [O(V/p) + O(log p × t_s) + O(V/p)]
      = O(V²/p) + O(V log p × t_s)

Sequential time: T_s = O(V²)  [for dense graph]
Speedup: S = V² / (V²/p + V·log p·t_s) ≈ p when V >> p·log p·t_s
```

Delta-Stepping: A More Scalable Parallel Dijkstra

The V rounds of global synchronisation in the basic approach (one round per vertex) limit scalability. Delta-stepping addresses this by processing multiple vertices per round — all vertices whose tentative distance falls within a 'bucket' of width Δ are processed concurrently:

- All vertices in the current bucket [kΔ, (k+1)Δ) are relaxed in parallel within each round.
- Vertices may be 'light' edges (weight ≤ Δ) within a round, processed first, or 'heavy' edges (weight > Δ), processed after.
- With a well-chosen Δ, delta-stepping achieves near-linear speedup on real-world road networks and social graphs, making it the basis of practical parallel shortest-path libraries.

Complexity Summary

Variant	Parallel Time	Communication Rounds	Best For
Simple partitioned	O(V²/p + V log p)	V rounds of all-reduce	Dense graphs, small p
Priority-queue based	O((V+E) log V / p + V log p)	V rounds	Sparse graphs
Delta-stepping	O(VΔ/p + V/Δ × log p)	O(V/Δ) rounds	Large-scale real graphs

> *Note: Dijkstra's algorithm is inherently difficult to parallelise efficiently because of the global minimum extraction step — it imposes V global synchronisations. The delta-stepping algorithm by Meyer & Sanders (1998) is the state-of-the-art solution, allowing multiple vertices to be settled simultaneously when their distance estimates fall in the same delta bucket. This is implemented in modern libraries like PBGL (Parallel Boost Graph Library) and Galois.*

Additional Concepts & Quick Reference

Sorting Primitive Comparison

Primitive	Input	Output	Cost	Used In
Compare-Exchange	2 single values	Sorted pair (min, max)	O(1) comp + O(1) comm	Bitonic sort, odd-even sort
Compare-Split	2 sorted lists (n/p each)	Lower half / Upper half	O(n/p) comp + O(n/p) comm	Parallel merge sort

GPU Memory Access Best Practices

- Coalesced access: ensure threads in a warp access consecutive memory addresses. When thread k accesses address base + k, all 32 accesses are served in one memory transaction. Random access requires 32 separate transactions — 32× slower.
- Shared memory tiling: load a tile of global memory into shared memory cooperatively (all threads in a block load together), then process the tile using fast shared memory. This pattern is the foundation of fast matrix multiply, convolution, and reduction kernels.
- Avoid branch divergence: if threads in the same warp take different if/else branches, both paths are executed serially. Keep conditional logic uniform across a warp, or move conditional logic to a pre-processing step that splits work into uniform groups.
- Prefer D2D transfers over H2D when possible: cudaMemcpyDeviceToDevice is an on-GPU operation, limited only by GPU memory bandwidth (≈600 GB/s), which is 20× faster than PCIe H2D transfers.