

# SPPU-BE-COMP-CONTENT – KSKA Git

## # ASSIGNMENT: NO: 1 (ONE)

Q1) What is CUDA? Explain different programming languages supporting CUDA. Discuss any 3 applications of CUDA.

- ANS
- CUDA stands for 'Compute Unified Device Architecture'
  - CUDA is a parallel computing platform developed by NVIDIA that enables programmers to use GPU Power for General Purpose Computing, (GPGPU)
  - It allows execution of thousands of threads in parallel, making it suitable for high performance Applications.
  - Programming Languages supported in CUDA:
- CUDA provides support for multiple Languages.
1. CUDA C/C++
    - Most commonly used.
    - Extensions like `_global_`, `_device_`, `_host_`
  2. CUDA Fortran.
    - Used in Scientific computing.
  3. Python (via Libraries)
    - Libraries like PyCUDA, CuPy
    - Easier for Rapid Development.
  4. OpenCL (Alternative Framework)
    - Cross platform parallel programming.
  5. Matlab CUDA - support



# SPPU-BE-COMP-CONTENT - KSKA Git

Page No. : \_\_\_\_\_

Date. : / /

- Used for simulations and prototyping.

## o Applications of CUDA:-

### 1. Scientific Computing and Simulations.

CUDA is widely used in research for solving complex mathematical models:

- Molecular dynamics and Drug Discovery
- Astrophysics simulations (galaxies, Black holes)

Example: Simulating millions of particles simultaneously using GPU Threads.

### 2. Deep Learning and Artificial Intelligence.

CUDA powers modern AI Framework like PyTorch & TensorFlow.

- Training Neural Networks
- Autonomous Vehicles.
- Recommendation system.
- Image recognition and NLP.

### 3. Image and Video Processing.

CUDA accelerates multimedia applications.

- Real time video encoding / Decoding.
- Image Filtering and enhancement, and computer vision tasks
- Rendering tools like Adobe Premiere Pro and Blender.

### 4. Gaming and Graphics Rendering.

CUDA enhances gaming performance.

- Real time ray tracing.
- Physics simulation (particles, collision)
- Advanced shading and Lighting effects.

→ Used in modern game engines for realistic graphics.

### 5. Medical Imaging and Healthcare.

### 6. Financial Modeling.

### 7. Cryptography and cybersecurity.

### 8. Data Analytics and Big Data.

# SPPU-BE-COMP-CONTENT - KSKA Git

Page No. : \_\_\_\_\_

Date. : / /

Q2.) Describe processing flow of CUDA along with CUDA-C functions Applications.

ANS. CUDA Processing Flow:-

[Host (CPU)]

↓  
1. Allocate Memory (cudaMalloc)

↓  
[Device memory (GPU)]

↓  
2. Copy Data (cudaMemcpy)

↓  
[Kernel execution on GPU]

↓  
3. Parallel Execution.

↓  
[Results stored in GPU]

↓  
4. Copy Back (cudaMemcpy)

↓  
[Host Output]

# Steps in CUDA Execution

1. Memory Allocation.

• cudaMalloc()

2. Data Transfer (Host → Device)

• cudaMemcpy()

3. Kernel Launch

• kernel <<< grid, block >>>()

4. Execution on GPU

• Thousands of threads execute

5. Data Transfer Back,

• cudaMemcpy()

6. Memory Deallocation.

• cudaFree()



→ Important CUDA C Functions.

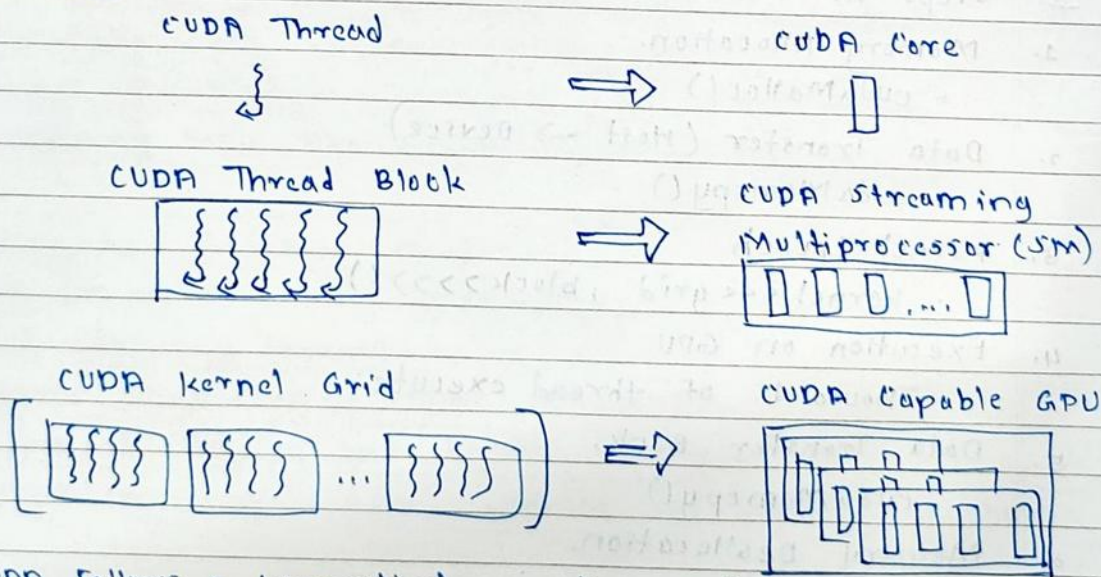
- `cudaMalloc` → Allocate GPU Memory.
- `cudaMemcpy` → Transfer Data.
- `cudaFree` → Free Memory.
- `_global_` → Kernel Function.
- `_device_` → Runs on GPU
- `_host_` → Runs on CPU.

Q3) Explain how the CUDA C program executes at the kernel level with example.

ANS. CUDA Kernel:

A CUDA kernel is a function written in CUDA C/C++ language that runs on the GPU (device) and is executed by many threads in parallel. It is declared using the `_global_` keyword and launched from the CPU (Host).

Diagram:-



◦ CUDA follows a hierarchical execution Model.

1. Grid - Collection of Blocks.
2. Block - Collection of Threads.
3. Thread - Smallest Execution Unit.



# SPPU-BE-COMP-CONTENT - KSKA Git

Page No. : \_\_\_\_\_

Date. : / /

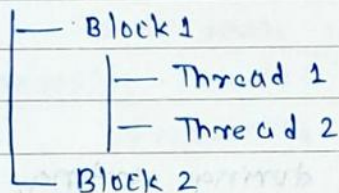
Each Thread executes the same kernel code but operates on different data.

## # STEPS of Kernel Function.

1. Host (CPU) prepares data.
2. Data is copied from Host Memory  $\rightarrow$  Device Memory.
3. GPU Executes kernel in parallel Threads.
4. Kernel is launched using special Syntax.
5. Results are copied back to Host.

## $\Rightarrow$ Execution Hierarchy.

Grid



Example:- Vector Addition

\_\_global\_\_ void add(int \*a, int \*b, int \*c)

```
{
    int i = threadIdx.x;
    c[i] = a[i] + b[i];
}
```

Kernel Launch:

add<<<1, N>>>>(a,b,c);

- Each thread computes one element.
- Thread index identifies work.
- Execution happens in parallel.

## • Key Points:-

- ① Threads execute simultaneously
- ② Efficient for large data.
- ② Synchronization required in some cases.



# SPPU-BE-COMP-CONTENT - KSKA Git

Page No. : \_\_\_\_\_

Date. : / /

Q4) What are the issues in sorting on parallel computers with Example?

ANS. Issues in Sorting on Parallel Computers.

1. Load Balancing problem.

In parallel sorting, the input data is divided among multiple processors. Ideally, each processor should get an equal amount of Work.

Issue: Uneven data distribution leads to some processors finishing early while others are still working.

• Causes Idle time and reduces efficiency.

Example: In parallel Quicksort, if pivot selection is poor, one processor may get most of the data while others get very little.

2. Communication Overhead.

• Processors must exchange data during sorting.

Issue: Processors must wait for others to reach certain points.

• Leads to delays and inefficient utilization.

Example: In parallel Merge Sort, sorted sublists must be exchanged and merged across processors → heavy data transfer.

3. Data Dependency and Ordering constraints.

• Sorting requires maintaining a global Order.

Issue: Operations are not fully independent.

• Ensuring correct ordering across processors is complex.

Example:- Two processors sorting different halves still need to ensure elements are globally ordered.

4. Scalability Issues.

Increasing the Number of processors does not always



# SPPU-BE-COMP-CONTENT - KSKA Git

Page No. : \_\_\_\_\_

Date. : / /

improve performance.

Issue:-

• Beyond a limit, communication and synchronization overhead outweigh benefits.

• Diminishing returns with more processors.

Example: Sorting 1 million elements with 100 processors may not be faster than with 10 processors.

5. Memory Access Conflicts.

Occurs in shared memory systems.

Issue: • Multiple processors accessing the same memory location leads to contention.

• Slows down execution.

Example: Parallel Insertion sort where processors frequently access shared arrays.

Q5.)

Explain Dijkstra's algorithm in parallel Formulation.

ANS.

Dijkstra's Algorithm in Parallel Formulation.

In a graph  $G(V, E)$ , the goal is to find the shortest path from a source vertex to all other vertices.

Parallelization focuses on:-

- Finding the minimum distance vertex in parallel.
- Performing edge relaxation concurrently.

• Sequential Formula:-

The key operation in Dijkstra's algorithm is Edge Relaxation.

$$\text{dist}[v] = \min[\text{dist}[v], \text{dist}[u] + w(u, v)]$$

• Parallel Formulation:-

① STEP 1: Initialization

• Set distance of source = 0.



# SPPU-BE-COMP-CONTENT - KSKA Git

Page No. : \_\_\_\_\_

Date. : / /

- All other vertices =  $\infty$
- Mark all vertices as visited.

## ② STEP 2: Parallel Minimum Selection.

- Each processor is assigned a subset of vertices.
- Each processor finds local minimum distance vertex.
- A global reduction is used to find the overall minimum.

## ③ STEP 3: Parallel Relaxation.

- Once the minimum vertex  $u$  is selected.
- All processors simultaneously update neighbours of  $u$
- Each processor handles a subset of Edges:

$$\text{dist}[v] = \min(\text{dist}[v], \text{dist}[u] + w(u,v))$$

## ④ STEP 4: Synchronization.

- After relaxation, processors synchronize.
- Mark vertex  $u$  as visited

## ⑤ STEP 5: Repeat.

- Continue until all vertices are processed.

### • Parallel Approaches:-

- ① Shared Memory Model (OpenMP)
- ② Distributed Memory model (MPI)
- ③ GPU based parallel Dijkstra (CUDA)

### • Time Complexity:-

1. Sequential  $O(V^2)$  or  $O(V \log V)$
2. Parallel  $O\left(\frac{V^2}{P} + \log P\right)$

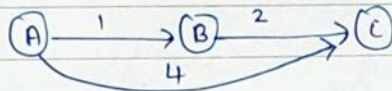


# SPPU-BE-COMP-CONTENT – KSKA Git

Page No. : \_\_\_\_\_

Date. : / /

→ Example:-



Parallel Execution:-

- Processor 1 handles  $A \rightarrow B$
- Processor 2 handles  $A \rightarrow C$ .
- Relaxation happens simultaneously.

Result:

- Shortest path  $A \rightarrow C = 3$  (via B)

→ Challenges in Parallel Dijkstra:-

1. Sequential Bottleneck.
2. Load Balancing.
3. Communication cost.
4. Synchronization Overhead.