

# SPPU-BE-COMP-CONTENT - KSKA Git

Q1) What are the advantages of using CUDA to perform vector addition and Matrix multiplication compared to using a CPU?

ANS. GPU (CUDA) is better than CPU becoz, CUDA (by NVIDIA) uses thousands of cores, while a CPU typically has fewer cores.

Key Advantages:-

1. Massive Parallelism.

- Vector Addition: Each element computed by a separate thread

- Vector Multiplication: Each thread computes one element of result Matrix.

CPU → Sequential or limited parallel.

GPU → Thousands of parallel operations.

2. Faster Computation

- CPU executes many operations simultaneously.

- Significant speedup for Large Datasets.

3. Better Throughput.

- Handles large scale data efficiently.

- Ideal for scientific and numerical Computations.

4. Efficient Memory Usage (with Optimization)

- Shared memory reduces global memory access.

- Improves speed.

5. Scalability.

- Performance increases with more GPU cores.

CUDA is faster than CPU for:-

① Large vectors



# SPPU-BE-COMP-CONTENT - KSKA Git

② Large Matrices.

③ Repetitive computations.

Q2.)

How do you launch a CUDA kernel to perform different operations?

ANS.

Kernel Launch Syntax:-

kernel-name <<< numBlocks, threadsPerBlock >>> (arguments);

Steps to Launch a kernel.

→

STEP 1:- Define kernel

```
_global_ void add(int *A, int *B, int *C) {  
    int i = threadIdx.x + blockIdx.x * blockDim.x;  
    C[i] = A[i] + B[i];  
}
```

STEP 2:- Launch kernel

add <<< numBlocks, threadsPerBlock >>> (A, B, C);

⇒

Example Configuration

Vector Addition

```
int threadsPerBlock = 256;  
int numBlocks = (N + 255) / 256;  
add <<< numBlocks, threadsPerBlock >>> (A, B, C);
```

→

For Note:-

- ① numBlocks - Number of blocks in Grid.
- ② threadsPerBlock - Threads inside each block.
- ③ Each thread executes same kernel but on different data.



# SPPU-BE-COMP-CONTENT - KSKA Git

Q3.) How can you Optimize the performance of a CUDA program for adding two large vectors and Matrix Multiplication

ANS: Optimization Techniques:-

1. Memory Coalescing.

- Access global memory in contiguous way.
- Reduces memory Latency.

2. Use shared Memory.

- Stores frequently used data in fast shared Memory.
- Especially useful in matrix based Multiplication.

3. Minimize Global Memory Access.

- Global Memory is slow
- Reuse data in shared Memory.

4. Proper thread and Block size.

- Choose Optimal size (e.g., 256 thread/block)
- Improves GPU Utilization.

5. Avoid Thread Divergence

- Threads in same block should follow same execution path.

6. Loop Unrolling.

- Reduces Loop Overhead.

7. Use Tiling (for Matrix Multiplication)

- Divide matrix into small blocks (tiles)
- Load tiles into shared Memory.

• Example concept:-

Load tile  $\rightarrow$  Compute Partial Result  $\rightarrow$  Repeat  $\rightarrow$  Final Result



# SPPU-BE-COMP-CONTENT - KSKA Git

## 8. Synchronization.

- Use `_syncthreads()` correctly.
- Avoid unnecessary synchronization.

## 9. Occupancy Optimization

- Ensure maximum active threads per SM (Streaming Multiprocessor)

### ⇒ Example Optimization (Matrix multiplication Idea)

- Load submatrix into shared Memory.
- Perform Multiplication.
- Reduce Global Memory cells.