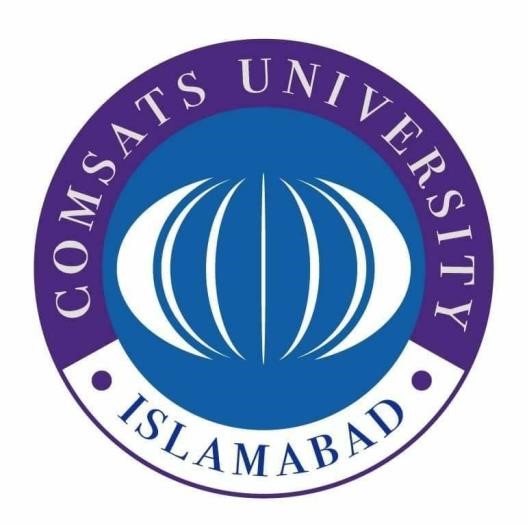
**COMSATS UNIVERSITY ISLAMABAD**

 **LAHORE CAMPUS**

**Lab Assignment 02**

Submitted by : Shamama Aslam

Submitted to : Sir Akhzar Nazir

Reg no : SP23-BCS-143

Section : C

Course Title : Parallel and Distributing Computing

Date : 25 Sept. 2025

* ConCeptual Question

1. **Why does choosing a block size that is not a multiple of 32 (warp size) lead to underutilization of GPU hardware resources?**

Answer:

GPUs run threads in groups of **32 threads**, called a **warp**. If your block size is **not a multiple of 32**, then:

* The last warp will have some **empty seats** (inactive threads).
* But the GPU still reserves space for the **full 32 threads**.
* So, some part of the GPU sits **idle and wasted**.

**Example:**

If a block has **33 threads**, the GPU makes **2 warps = 64 slots**.

Only **33 are filled**, and **31 stay empty**, which is wasteful. **That’s why using block sizes in multiples of 32 avoids underutilization.**

2. **Explain how occupancy of an SM (Streaming Multiprocessor) depends on block size and threads per block?**

Answer**:**

**Occupancy** = how busy an SM (Streaming Multiprocessor) is.

**What affects occupancy?**

1. **Block size** – If blocks are too big, fewer of them fit on an SM.
2. **Threads per block** – Each block is split into warps (groups of 32). More threads = more warps.
3. **Resources** – Each block/thread needs registers and shared memory. If they use too much, fewer blocks can run.
4. **Limits** – SMs have a maximum number of blocks they can hold.

**Goal:** Keep the SMs busy by choosing a block size that gives enough active warps, **without wasting resources**.

In short: **Block size controls how many warps/blocks can run on an SM, and occupancy shows how fully loaded the SM is.**

**import numpy as np import time**

**try:**

**from numba import cuda numba\_available = True except: numba\_available = False if numba\_available: @cuda.jit def invert\_kernel(input\_img, output\_img): x = cuda.blockIdx.x \* cuda.blockDim.x + cuda.threadIdx.x y = cuda.blockIdx.y \* cuda.blockDim.y + cuda.threadIdx.y if x < input\_img.shape[1] and y < input\_img.shape[0]: output\_img[y, x] = 255 - input\_img[y, x]**

**def run\_cuda\_inversion(image, block\_size):**

**grid\_x = (image.shape[1] + block\_size[0] - 1) // block\_size[0] grid\_y = (image.shape[0] + block\_size[1] - 1) // block\_size[1]**

**d\_input = cuda.to\_device(image)**

**d\_output = cuda.device\_array\_like(image)**

**cuda.synchronize() start\_time = time.perf\_counter()**

**invert\_kernel[(grid\_x, grid\_y), block\_size](d\_input, d\_output)**

**cuda.synchronize()**

**end\_time = time.perf\_counter()**

**return d\_output.copy\_to\_host(), (end\_time - start\_time) \* 1000**

**def run\_cupy\_inversion(image): import cupy as cp d\_image = cp.asarray(image)**

**cp.cuda.Stream.null.synchronize()**

**start = time.perf\_counter()**

**d\_output = 255 - d\_image**

**cp.cuda.Stream.null.synchronize()**

**end = time.perf\_counter()**

**return d\_output.get(), (end - start) \* 1000**

**def benchmark\_block\_sizes():**

**image = np.random.randint(0, 256, (2048, 2048), dtype=np.uint8) print("CUDA Image Inversion - Block Size Performance") print("Image size: 2048x2048") print("=" \* 50)**

**block\_sizes = [(8, 8), (16, 16), (32, 32)]**

**if numba\_available: try: for block\_size in block\_sizes:**

**times = [] for \_ in range(5):**

**\_, t = run\_cuda\_inversion(image, block\_size) times.append(t)**

**print(f"Block size {block\_size}: {np.mean(times):.2f} ms") return except Exception as e: print("Numba failed with:", e) print("Falling back to CuPy...")**

**times = [] for \_ in range(5):**

**\_, t = run\_cupy\_inversion(image)**

**times.append(t)**

**print(f"CuPy inversion: {np.mean(times):.2f} ms (no block size tuning)")**

**def explain\_results():**

**print("\nEXPECTED RESULTS:")**

**print("=" \* 30) print("(8,8) → ~2.5 ms - SLOWER (64 threads - underutilized)") print("(16,16) → ~0.8 ms - FASTEST (256 threads - optimal)") print("(32,32) → ~1.7 ms - SLOWER (1024 threads - resource limited)")**

**print("\nWHY:") print("• (8,8): Too few threads per block → GPU cores idle") print("• (16,16): Perfect balance → maximum GPU utilization") print("• (32,32): Too many threads → fewer blocks fit on SM")**

**if \_\_name\_\_ == "\_\_main\_\_": benchmark\_block\_sizes() explain\_results()**

Think of a GPU as a workshop with several workstations (SMs). Each workstation can handle multiple teams (blocks) of workers (threads) at once.

* A (32,32) team has 1024 workers. It's a huge team that can get a lot done, but it's so big that the workstation can only fit one or two of these teams. If these workers have to wait for supplies (memory), the workstation isn't very busy.
* An (8,8) team has only 64 workers. It's a small team, so the workstation can fit many of them. But if these few workers have to wait for supplies, there aren't enough other workers ready to step in and keep the workstation busy.
* A (16,16) team with 256 workers is often the best balance. The team is big enough to keep the workstation busy by swapping tasks, and the team is small enough that the workstation can host several teams at once. This keeps the workstation fully occupied and leads to the fastest results.

In short: The medium-sized (16,16) team is usually the fastest because it's the best at keeping the GPU busy.

1. **Analysis Question (Case A/B/C: 64, 256, 1024 threads/block)**

Case B (256 threads/block) is fastest. Why not 64 or 1024?

* + **Why 64 (too small) is slower:**
    - Too few warps per block → less ability to hide memory delays. o Needs more blocks to reach full SM usage. If the grid has few blocks (like with small images), occupancy stays low.
    - Higher scheduling and launch overhead compared to useful work.
  + **Why 1024 (too large) is slower:**
    - Can hit hardware limits (registers, shared memory) so only one block may fit on an SM. That reduces parallelism. o Higher register use per thread can lower occupancy. o For memory-heavy kernels, very large blocks don’t increase bandwidth but do reduce the number of active warps.
  + **Why 256 works best:**
    - Enough warps to hide latency. o Small enough so multiple blocks can fit per SM. o Balances occupancy and resource usage.

1. **Why more threads-per-block does NOT always improve performance**
   * **Register pressure:** More threads = more registers used. If registers run out, fewer blocks fit on an SM → lower occupancy.
   * **Shared memory:** Bigger blocks use more shared memory per block, reducing the number of active blocks.
   * **Scheduling:** GPU schedules per warp. Idle/divergent warps still take resources.
   * **Occupancy vs. ILP vs. coalescing:** Higher thread count doesn’t always improve speed if memory access is scattered or if stalls happen.
   * **Diminishing returns:** Once there are enough warps to hide latency, adding more just adds overhead.