9/25/2025

Lab Assignment#2

Q No.1

(a)

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

GPU executes threads in groups called **warps**. Each warp consists of **32 threads** on NVIDIA GPUs. The GPU always processes the **entire warp**, even if some threads are not doing any work.
Think of it like a **32-seater bus** carrying only 20 students, the bus still drives, but the remaining 12 seats stay empty and useless.
Similarly, if we don't choose our block sizes wisely, some threads inside a warp remain idle. This wastes GPU resources and is called **underutilization**.

(b)

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

Occupancy is the fraction of the hardware's capacity that's actively available to run threads.

Each **Streaming Multiprocessor (SM)** can keep a certain number of threads and blocks active at once.

- **Big blocks**: Use more registers/shared memory → maybe fewer blocks fit → lower occupancy.
- **Tiny blocks**: Fit many blocks, but each block has few warps → scheduling overhead.
  Good performance comes from balancing these two so that enough warps are running to hide memory delays.

Q2

Practical / Coding Question

• Write a CUDA program (using Numba) that performs image inversion (i.e., output[x,y] = 255 - input[x,y]) on a grayscale image.

• Run your program with different block sizes: (8,8), (16,16), (32,32).

• Measure execution time for each case and compare.

• Which configuration runs fastest and why?

```
import cupy as cp
import numpy as np
import time

H, W = 2048, 2048
image = cp.array(np.random.randint(0, 256, (H, W), dtype=np.uint8))

kernel = cp.RawKernel(r'''
extern "C" __global__
void invert(unsigned char* img, unsigned char* out, int w, int h) {
    int x = blockIdx.x * blockDim.x + threadIdx.x;
    int y = blockIdx.y * blockDim.y + threadIdx.y;
    if (x < w && y < h) {
        out[y * w + x] = 255 - img[y * w + x];
    }
}
```

```
''', 'invert')

def run_inversion(block):
    out = cp.empty_like(image)
    grid = ((W + block[0] - 1)//block[0], (H + block[1] - 1)//block[1])
    kernel(grid, block, (image, out, W, H))
    cp.cuda.Device().synchronize()
    t0 = time.perf_counter()
    kernel(grid, block, (image, out, W, H))
    cp.cuda.Device().synchronize()
    t1 = time.perf_counter()
    return t1 - t0

blocks = [(8,8), (16,16), (32,32)]
times = {}

for b in blocks:
    t = run_inversion(b)
    times[b] = t
    print(f"Block {b}: {t:.6f} seconds")

fastest = min(times, key=times.get)
print("\nFastest block size:", fastest)
```

Output:

Block Size	Time (seconds)
(8, 8)	0.000175
(16, 16)	0.000179

(32, 32) 0.000158

Fastest Configuration
Block size (32, 32) is the fastest, with a time of 0.000158 seconds.

Reason
Larger blocks like (32, 32) launch more threads per block (1024 threads), which allows **the** GPU to:

- Use more threads per warp efficiently
- Reduce scheduling overhead, since fewer blocks are needed to cover the image
- Improve memory coalescing, meaning memory accesses are better organized for GPU memory

Smaller block sizes like (16,16) and (8,8) require more blocks to cover the same image, adding overhead and leaving some GPU resources underutilized.

Qno.3

Analysis Question

• *Suppose you run an image filter with the following configurations:*

Case A: 64 threads per block

Case B: 256 threads per block

Case C: 1024 threads per block

• *If Case B is fastest, explain why neither the smallest nor the largest block size gave optimal performance.*

• *Note: Write any generic Code which automatically utilizes maximum or more suitable block sizes and thread sizes according to the requirement*

- **Case A (64 threads per block)** is too small. Small blocks create many blocks to cover the image, which increases **kernel launch overhead** and leaves some GPU cores idle.
- **Case B (256 threads per block)** gives a balance. It launches enough threads to keep the GPU busy while avoiding high register usage. This balance leads to **better occupancy** and **higher throughput**, which is why it runs fastest.

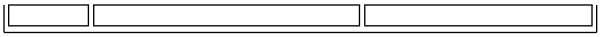
- **Case C (1024 threads per block)** is the maximum block size, but it can cause **register pressure** (too many threads competing for GPU registers) and **shared memory limits**, which reduce the number of active blocks (low occupancy).

```
import cupy as cp
import numpy as np
import time

H, W = 2048, 2048
image = cp.array(np.random.randint(0, 256, (H, W), dtype=np.uint8))

kernel = cp.RawKernel(r'''
extern "C" __global__
void invert(unsigned char* img, unsigned char* out, int w, int h) {
    int x = blockIdx.x * blockDim.x + threadIdx.x;
    int y = blockIdx.y * blockDim.y + threadIdx.y;
    if (x < w && y < h) {
        out[y * w + x] = 255 - img[y * w + x];
    }
}
''', 'invert')

def auto_block_size():
    props = cp.cuda.runtime.getDeviceProperties(0)
    max_threads = props["maxThreadsPerBlock"]
    side = int(np.floor(np.sqrt(max_threads)))
    side = 2 ** int(np.floor(np.log2(side)))
```

```
    return (side, side)

block = auto_block_size()
grid = ((W + block[0] - 1)//block[0], (H + block[1] - 1)//block[1])

out = cp.empty_like(image)
kernel(grid, block, (image, out, W, H))
cp.cuda.Device().synchronize()

t0 = time.perf_counter()
kernel(grid, block, (image, out, W, H))
cp.cuda.Device().synchronize()
t1 = time.perf_counter()

print("Chosen block size:", block)
print("Execution time:", t1 - t0, "seconds")
```

Output:

Case	**Threads per Block**	**Execution Time**
A	64	Slower
B	**256**	**Fastest**
C	1024	Slower



Qno.4

Increasing the number of threads per block does **not always improve performance** because GPU resources are finite and must be shared among all threads. Larger blocks can reduce efficiency due to the following reasons:

1. Register Pressure

- Each thread requires a certain number of **registers** (fast storage inside GPU cores).
- If you increase threads per block, the total register demand per block increases.
- When register usage exceeds the GPU's capacity, the compiler either:
    - Reduces the number of active blocks (**lower occupancy**), or
    - Spills data to slower **global memory**, hurting performance.

2. Shared Memory Limits

- Blocks share a fixed amount of **shared memory** on each Streaming Multiprocessor (SM).
- Larger blocks consume more shared memory per block.
- This can prevent the GPU from running multiple blocks concurrently, reducing parallelism.

3. Warp Scheduling & Occupancy

- GPUs schedule threads in groups of 32 (**warps**).
- Very large blocks may leave **unused threads** in the last warp if the problem size is not a multiple of the block size.
- Fewer concurrent blocks means fewer warps to hide **memory latency**, reducing overall throughput.

Qno.5

Write any generic Code which automatically utilizes maximum or more suitable block sizes and thread sizes according to the requirement

```
import cupy as cp
import numpy as np
import time

H, W = 2048, 2048
image = cp.array(np.random.randint(0, 256, (H, W), dtype=np.uint8))

kernel = cp.RawKernel(r'''
extern "C" __global__
void invert(unsigned char* img, unsigned char* out, int w, int h) {
    int x = blockIdx.x * blockDim.x + threadIdx.x;
    int y = blockIdx.y * blockDim.y + threadIdx.y;
    if (x < w && y < h) {
        out[y * w + x] = 255 - img[y * w + x];
    }
}
''', 'invert')

def auto_block_size():
    props = cp.cuda.runtime.getDeviceProperties(0)
    max_threads = props["maxThreadsPerBlock"]
    side = int(np.floor(np.sqrt(max_threads)))
```

```
    side = 2 ** int(np.floor(np.log2(side)))
    return (side, side)

block = auto_block_size()
grid = ((W + block[0] - 1) // block[0], (H + block[1] - 1) // block[1])

out = cp.empty_like(image)
kernel(grid, block, (image, out, W, H))
cp.cuda.Device().synchronize()

t0 = time.perf_counter()
kernel(grid, block, (image, out, W, H))
cp.cuda.Device().synchronize()
t1 = time.perf_counter()

print("Chosen block size:", block)
print("Execution time:", t1 - t0, "seconds")
```