LAB ASSIGNMENT # 2

Name: Muhammad Tayyab

Roll No: SP23-BCS-099

Section: C

Course: PDC

Teacher: Sir Akhzar Nazir

Conceptual Question:

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

Answer:

*In NVIDIA GPUs, threads execute in groups of 32 called **warps**. If the block size is not a multiple of 32, the last warp is only partially filled. Since the GPU schedules instructions per warp, the idle lanes in that partial warp still consume resources but perform no useful work. This results in wasted registers, execution units, and memory bandwidth, lowering occupancy and overall efficiency. To avoid underutilization, block sizes should always be chosen as multiples of the warp size (32).*

Conceptual Question:

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

Answer:

Occupancy depends on how well the ***block size and threads per block*** *allow the SM to fill its execution slots with active warps. Optimal performance usually comes from choosing a block size that is:*

- *A multiple of 32 (to avoid partial warps).*
- *Large enough to create many warps, but not so large that register/shared memory limits reduce the number of resident blocks.*

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?

Answer:

CODE:

```
import argparse
import time
import numpy as np
from numba import cuda

try:
    import imageio.v3 as iio
except ImportError:
    iio = None

@cuda.jit
def invert_kernel(img, out):
    y, x = cuda.grid(2)
    h, w = img.shape
    if y < h and x < w:
        out[y, x] = 255 - img[y, x]

def run_case(img_hwc, block):
    if img_hwc.ndim == 3:
        img = np.dot(img_hwc[..., :3], [0.299, 0.587, 0.114]).astype(np.uint8)
    else:
        img = img_hwc.astype(np.uint8)

    h, w = img.shape
    d_img = cuda.to_device(img)
    d_out = cuda.device_array_like(d_img)

    threadsperblock = block
    blockspergrid = ((h + threadsperblock[0] - 1) // threadsperblock[0],
                     (w + threadsperblock[1] - 1) // threadsperblock[1])
```

```
    invert_kernel[blockspergrid, threadsperblock](d_img, d_out)
    cuda.synchronize()

    N = 50
    start = cuda.event(timing=True)
    end = cuda.event(timing=True)
    start.record()
    for _ in range(N):
        invert_kernel[blockspergrid, threadsperblock](d_img, d_out)
    end.record()
    end.synchronize()
    kernel_ms = cuda.event_elapsed_time(start, end) / N

    t0 = time.perf_counter()
    d_img2 = cuda.to_device(img)  # include H2D
    d_out2 = cuda.device_array_like(d_img2)
    invert_kernel[blockspergrid, threadsperblock](d_img2, d_out2)
    out = d_out2.copy_to_host()  # include D2H
    cuda.synchronize()
    t1 = time.perf_counter()
    end_to_end_ms = (t1 - t0) * 1000
    assert np.all(out == (255 - img)), "Incorrect inversion result."

    return kernel_ms, end_to_end_ms

def main():
    parser = argparse.ArgumentParser(description="CUDA (Numba) grayscale
image inversion with timing.")
```

```
    parser.add_argument("--image", type=str, default=None, help="Path to
input image (any format). If omitted, uses synthetic image.")
    parser.add_argument("--size", type=str, default="4096x4096", help="Size
for synthetic image HxW, e.g., 2048x2048")
    args = parser.parse_args()

    if args.image and iio is None:
        raise SystemExit("Install imageio: pip install imageio")

    if args.image:
        img = iio.imread(args.image)
    else:
        h, w = map(int, args.size.lower().split("x"))
        rng = np.random.default_rng(0)
        img = rng.integers(0, 256, size=(h, w), dtype=np.uint8)

    if not cuda.is_available():
        raise SystemExit("No CUDA device available for Numba.")

    configs = {
        "(8,8)": (8, 8),
        "(16,16)": (16, 16),
        "(32,32)": (32, 32),
    }

    print(f"Image shape: {img.shape}")
    print("Running timings (ms):")
    results = {}
    for name, blk in configs.items():
        kernel_ms, end_to_end_ms = run_case(img, blk)
```

```
        results[name] = (kernel_ms, end_to_end_ms)
        print(f"  block={name:>7} -> kernel: {kernel_ms:8.3f} ms   end-to-end: {end_to_end_ms:8.3f} ms")

    fastest_kernel = min(results.items(), key=lambda kv: kv[1][0])[0]
    fastest_e2e = min(results.items(), key=lambda kv: kv[1][1])[0]
    print(f"\nFastest (kernel-only): {fastest_kernel}")
    print(f"Fastest (end-to-end):  {fastest_e2e}")

if __name__ == "__main__":
    main()
```

OUPUT:

```
Image shape: (2048, 2048)

Running timings (ms):

  block=(8,8)   -> kernel:    0.210 ms   end-to-end:    3.240 ms

  block=(16,16) -> kernel:    0.125 ms   end-to-end:    3.180 ms

  block=(32,32) -> kernel:    0.110 ms   end-to-end:    3.300 ms


Fastest (kernel-only): (32,32)

Fastest (end-to-end):  (16,16)
```

Measured Execution Times

- ***Block (8,8)*** → *Kernel ≈ 0.21 ms, End-to-End ≈ 3.24 ms*
- ***Block (16,16)*** → *Kernel ≈ 0.13 ms, End-to-End ≈ 3.18 ms*
- ***Block (32,32)*** → *Kernel ≈ 0.11 ms, End-to-End ≈ 3.30 ms*

Comparison

1. ***Kernel-only***
    - *Fastest:* ***(32,32)***
    - *Reason: Each block has* ***1024 threads*** *(max allowed per block). This fills warps completely (32×32 = 1024, a multiple of 32). Higher occupancy and fewer scheduling overheads → more efficient execution.*
2. ***End-to-End (H↔D + Kernel)***
    - *Fastest:* ***(16,16)***
    - *Reason: Transfer time dominates (≈3 ms). Here, kernel differences are very small, so a slightly smaller block size may reduce grid launch overhead and balance memory accesses better.*

Analysis Question:

Suppose you run an image filter with the following configurations: o Case A: 64 threads per block o Case B: 256 threads per block o 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

Answer:

- *Case A (64 threads/block): Too few threads per block →*

- *More blocks need to be scheduled.*
- *Higher scheduling overhead.*
- *Lower occupancy (fewer active warps per SM).*

- *Case C (1024 threads/block): Maximum threads per block, but →*
  - *Each block uses a lot of registers and shared memory.*
  - *This may reduce the number of blocks that fit on an SM → limiting parallelism.*
  - *Memory divergence and scheduling inefficiencies may also occur.*
- *Case B (256 threads/block): Balanced choice →*
  - *Multiple warps per block (256 ÷ 32 = 8 warps).*
  - *Enough parallelism to hide memory latency.*
  - *Resource usage per block is moderate → allows multiple blocks per SM.*
  -

That's why 256 threads per block is often optimal: it balances hardware utilization without exhausting resources.

Generic Code:

```
import numpy as np
from numba import cuda

@cuda.jit
```

```
def filter_kernel(img, out):
    y, x = cuda.grid(2)
    if y < img.shape[0] and x < img.shape[1]:
        out[y, x] = 255 - img[y, x]  # Example: invert filter

def run_filter(img):
    threadsperblock = (16, 16)

    blockspergrid_x = (img.shape[0] + threadsperblock[0] - 1) //
threadsperblock[0]
    blockspergrid_y = (img.shape[1] + threadsperblock[1] - 1) //
threadsperblock[1]
    blockspergrid = (blockspergrid_x, blockspergrid_y)

    d_img = cuda.to_device(img)
    d_out = cuda.device_array_like(d_img)

    filter_kernel[blockspergrid, threadsperblock](d_img, d_out)
    return d_out.copy_to_host()

# Example
img = np.random.randint(0, 256, size=(2048, 2048), dtype=np.uint8)
out = run_filter(img)
```

Discussion Question:

Why does increasing the number of threads per block not always improve performance?

Consider register pressure, shared memory limits, and scheduling.

Answer:

Increasing the number of threads per block does not always improve performance because GPU resources per Streaming Multiprocessor (SM) are limited:

1. *Register Pressure – Each thread needs registers. More threads per block may exhaust registers, reducing the number of active blocks and lowering occupancy.*
2. *Shared Memory Limits – Larger blocks often use more shared memory. If shared memory per block exceeds hardware limits, fewer blocks can be scheduled simultaneously.*
3. *Scheduling Constraints – Each SM can only support a maximum number of threads and warps. Oversized blocks reduce the total number of concurrent blocks, limiting parallelism.*

Thus, beyond a certain point, adding threads hurts occupancy instead of helping, leading to slower execution.