**Lab Assignment # 2**

* **Conceptual Question**
* Why does choosing a block size that is not a multiple of 32 (warp size) lead to underutilization of GPU hardware resources?
* Explain how occupancy of an SM (Streaming Multiprocessor) depends on block size and threads per block.
* **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?
* **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
* **Discussion Question**
* Why does increasing the number of threads per block not always improve performance? Consider register pressure, shared memory limits, and scheduling.

**CODING PART:**

import numpy as np

import cupy as cp

import time

H, W = 4096, 4096

img = np.random.randint(0, 256, size=(H, W), dtype=np.uint8)

print("GPU available?:", cp.cuda.runtime.getDeviceCount() > 0)

invert\_kernel = cp.RawKernel(r'''

extern "C" \_\_global\_\_

void invert\_kernel(const unsigned char\* input\_img, unsigned char\* output\_img,

int H, int W) {

int x = blockDim.x \* blockIdx.x + threadIdx.x;

int y = blockDim.y \* blockIdx.y + threadIdx.y;

if (x < H && y < W) {

output\_img[x \* W + y] = 255 - input\_img[x \* W + y];

}

}

''', 'invert\_kernel')

def run\_gpu(block\_dim):

bx, by = block\_dim

gx = (H + bx - 1) // bx

gy = (W + by - 1) // by

grid\_dim = (gx, gy)

d\_in = cp.array(img)

d\_out = cp.empty\_like(d\_in)

invert\_kernel(grid\_dim, (bx, by), (d\_in, d\_out, H, W))

cp.cuda.runtime.deviceSynchronize()

start = cp.cuda.Event()

end = cp.cuda.Event()

start.record()

invert\_kernel(grid\_dim, (bx, by), (d\_in, d\_out, H, W))

end.record()

end.synchronize()

elapsed\_ms = cp.cuda.get\_elapsed\_time(start, end)

out = cp.asnumpy(d\_out)

correct = np.all(out == (255 - img))

return elapsed\_ms, correct

blocks = [(8,8), (16,16), (32,32)]

for b in blocks:

t\_ms, correct = run\_gpu(b)

print(f"Block {b}: {t\_ms:.3f} ms, correct={correct}")

def invert\_cpu(in\_img):

return 255 - in\_img

for b in blocks:

t0 = time.perf\_counter()

out = invert\_cpu(img)

t1 = time.perf\_counter()

print(f"Block {b} (CPU): {(t1-t0)\*1000:.3f} ms, correct={np.all(out==(255-img))}")

def choose\_block\_size\_auto(img\_shape, max\_threads\_per\_block=1024):

candidates = [(8,8), (16,16), (32,32)]

chosen = None

for b in candidates[::-1]:

tp = b[0]\*b[1]

if tp <= max\_threads\_per\_block and tp % 32 == 0:

chosen = b

break

if chosen is None:

chosen = (16,16)

return chosen

print("Auto-chosen block size:", choose\_block\_size\_auto(img.shape, max\_threads\_per\_block=1024))

**My results:**

* (8,8) → **1.017 ms (fastest)**
* (16,16) → **1.884 ms**
* (32,32) → **3.479 ms (slowest)**

**Why is (8,8) fastest in my case?**

* Smaller blocks let the GPU schedule **more blocks per SM**, hiding memory latency better.
* Larger blocks (1024 threads) consumed more SM resources, reducing active block count → less parallelism.

**Conceptual Question**

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

* GPU threads execute in groups of 32 (warps).
* If block size is **not a multiple of 32**, the last warp will have **inactive (idle) threads** → wasted GPU cores.
* In my test, I used (8,8), (16,16), (32,32) which correspond to **64, 256, and 1024 threads per block**.
* All of these are multiples of 32, so **no warp underutilization occurred**. That’s why my correctness check passed in all cases.

Even though block sizes differed in performance, none of the blocks wasted threads since all were warp-aligned.

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

* **Occupancy = active warps per SM ÷ maximum warps supported by SM.**
* Small blocks (like (8,8) = 64 threads) → allow **many blocks to run simultaneously per SM**, giving high occupancy.
* Large blocks (like (32,32) = 1024 threads) → consume more registers and shared memory per block → fewer blocks fit per SM → lower occupancy.

In my results:

* (8,8) = **fastest (1.017 ms)** → high occupancy, many blocks scheduled at once.
* (32,32) = **slowest (3.479 ms)** → fewer blocks active, less latency hiding.

This proves that occupancy strongly depends on block size in practice.

**Analysis Question**

**Case A: 64 threads per block (like your 8×8)**  
**Case B: 256 threads per block (like your 16×16)**  
**Case C: 1024 threads per block (like your 32×32)**

* In my experiment, **Case A (64 threads) was fastest**.
* But the question assumes **Case B (256 threads) is fastest**. Why not smallest or largest?

***Explanation:***

* **Too small (Case A / 64 threads):** Doesn’t fully utilize warps and creates scheduling overhead. On my GPU, it worked best only because it allowed very high occupancy.
* **Too large (Case C / 1024 threads):** Each block consumes too many registers/shared memory → fewer blocks per SM → reduced occupancy.
* **Case B (256 threads):** Provides a **balance**: enough threads to hide latency, but not so large that it limits active blocks per SM.

In general, GPUs perform best in the **sweet spot (128–512 threads per block)**, though in my case (8,8) surprisingly beat the others due to how Colab’s GPU manages resources.

**Discussion Question**

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

1. **Register pressure** – Each thread needs registers. Larger blocks → more registers → fewer blocks fit in SM → reduced occupancy.
2. **Shared memory limits** – If a block uses a lot of shared memory, large blocks may prevent multiple blocks from running together.
3. **Scheduling flexibility** – Many small blocks let the GPU schedule across SMs more effectively, while very large blocks reduce scheduling options.
4. **Memory bandwidth** – More threads doesn’t always mean better memory coalescing; sometimes smaller blocks coalesce memory better.

My results show exactly this:

* (32,32) with 1024 threads/block was **slower** than (8,8) because it consumed more SM resources and reduced scheduling flexibility.