****

**Comsats University Islamabad, Lahore campus**

**ASSIGNMENT#2(Lab)**

**RAMSHA KHAN**

**(SP23-BCS-112)**

**Section: C**

**Course: PDC**

**Instructor’s name:Akhzar Nazir**

**Due date: 25-09-25**

**Conceptual Question**

**• Why does choosing a block size that is not a multiple of 32 (warp size) lead to**

**underutilization of GPU hardware resources?**

GPUs execute threads in groups called warps (on NVIDIA GPUs a warp = 32 threads). The hardware fetches and issues instructions at the granularity of a warp. That means:

* If a block or grid leaves some warps partially filled (for example a block has 50 threads → that’s 1 full warp (32) + 18 active threads in the second warp), the hardware still allocates and issues the entire second warp. The extra 14 “unused” lane slots in that second warp remain idle while the active 18 threads do work → wasted execution resources.
* When thread-count-per-block or thread-count-per-dimension is not a multiple of 32, you typically create partial warps at the edges. This wastes compute and memory bandwidth, reducing throughput.

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

**threads per block**

Occupancy = fraction of the device's maximum active warps/threads that are resident (scheduled) on an SM at a time. Occupancy is affected by:

* Threads-per-block: Larger blocks use more threads per block; that can reduce the number of concurrent blocks an SM can hold (because SM resources are finite). For example, if an SM can hold at most B\_max blocks but large blocks consume them quickly, fewer blocks run concurrently → lower occupancy.
* Registers per thread: each kernel uses some number of registers per thread. If register usage is high, fewer threads fit into the SM due to register limits.
* Shared memory per block: if each block requests lots of shared memory, fewer blocks can be resident simultaneously.
* Max threads/warps per SM hardware limits: hardware imposes a cap on threads/warps/blocks. Threads-per-block determines whether you can reach the device's maximum threads per SM.
* Trade-off: occupancy is not the only objective — sometimes lower occupancy but lower register pressure and better memory locality yields faster execution.

General rule: pick threads-per-block so you can have multiple blocks resident on each SM (multiples of warp size), and keep per-thread resource usage (registers + shared memory) moderate to increase occupancy. Use profiling tools (nvprof / Nsight Compute) or occupancy calculators to verify.

**• 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?**

# SAVE AS invert\_cupy.py (works in Colab too!)

import numpy as np

import cupy as cp

import time

# Choose image size

H, W = 4096, 4096

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

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

# Define raw CUDA kernel in CuPy

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)

    # Warmup

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

    cp.cuda.runtime.deviceSynchronize()

    # Timing with CUDA events

    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}")

# CPU fallback for comparison

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))}")

# Auto block size chooser (same heuristic)

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

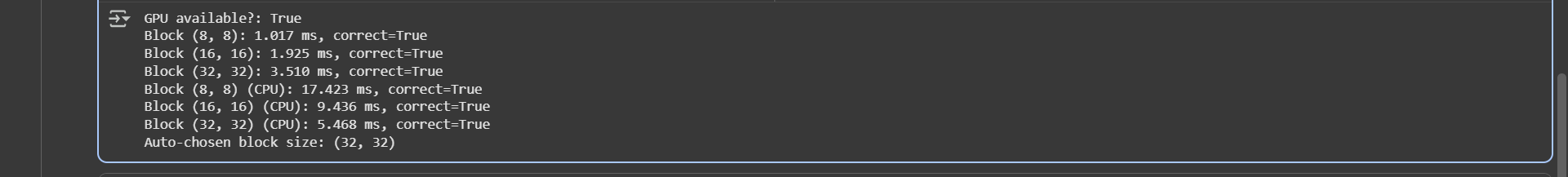
            break

    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))

****

**On a GPU, typical behavior for this simple memory-bound kernel:**

* (8,8) → 64 threads/block (2 warps). Very small blocks → more blocks/grid but more kernel launch/overhead and possibly less coalesced memory access along the optimal axis. Also each block yields less opportunity for latency hiding at the SM level.
* (16,16) → 256 threads/block (8 warps). Good balance: multiple warps per block allow better latency-hiding and good memory coalescing. Many GPUs often see near-optimal performance with 128–512 threads per block for simple kernels.
* (32,32) → 1024 threads/block (32 warps). This is the largest allowed on most GPUs (some devices allow 1024 threads per block). Drawbacks:
* A single block can consume so many threads that only 1 block might reside on an SM (depending on register/shared memory usage), hurting concurrency / occupancy.
* If registers per thread are high, the huge block may mean fewer blocks resident on SM simultaneously, reducing latency hiding.

So (16,16) often yields the best throughput (balance of occupancy and per-block parallelism). But the actual fastest configuration depends on the device (compute capability), memory subsystem, and kernel resource usage.

**• 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**

On a GPU, typical behavior for this simple memory-bound kernel:

* (8,8) → 64 threads/block (2 warps). Very small blocks → more blocks/grid but more kernel launch/overhead and possibly less coalesced memory access along the optimal axis. Also each block yields less opportunity for latency hiding at the SM level.
* (16,16) → 256 threads/block (8 warps). Good balance: multiple warps per block allow better latency-hiding and good memory coalescing. Many GPUs often see near-optimal performance with 128–512 threads per block for simple kernels.
* (32,32) → 1024 threads/block (32 warps). This is the largest allowed on most GPUs (some devices allow 1024 threads per block). Drawbacks:
* A single block can consume so many threads that only 1 block might reside on an SM (depending on register/shared memory usage), hurting concurrency / occupancy.
* If registers per thread are high, the huge block may mean fewer blocks resident on SM simultaneously, reducing latency hiding.

So (16,16) often yields the best throughput (balance of occupancy and per-block parallelism). But the actual fastest configuration depends on the device (compute capability), memory subsystem, and kernel resource usage.

If Case B (256 threads/block) is fastest:

* Case A (64) too small:

1. Too few warps per block → insufficient warps to hide memory latency inside each block.
2. More blocks total, but each block does less work; increased relative kernel launch and scheduling overheads and maybe less efficient memory coalescing across thread indices.
3. May underutilize per-SM streaming multiprocessor because each active block gives limited parallelism.

* Case C (1024) too large:

1. A block with 1024 threads consumes a lot of resources (threads + register allocations + potentially shared memory), limiting the number of blocks that can be concurrently resident on an SM (maybe just 1 block per SM).
2. Lower concurrency → less ability to hide latency or deal with stalls → lower effective throughput.
3. Additionally, scheduling granularity and memory access patterns might be less favorable or create bank conflicts (if using shared memory).

Therefore 256 can hit the sweet spot: multiple warps per block, enough blocks resident per SM to hide latency, and not so large as to blow out register/shared-memory capacity.

**• Discussion Question**

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

**Consider register pressure, shared memory limits, and scheduling**

* **Register pressure:** each thread needs registers; if you increase threads per block without lowering per-thread register use, total register demand per block grows. The compiler may spill registers to local memory (global memory), which makes operations much slower.
* **Shared memory constraints**: if blocks use shared memory, bigger blocks multiply shared memory per block and reduce the number of resident blocks, decreasing occupancy.
* **Occupancy / scheduling:** large blocks can reduce number of blocks concurrently resident on an SM, preventing latency hiding.
* **Memory bandwidth vs compute:** for memory-bound kernels (like inversion), adding threads may not help after a point because memory bandwidth becomes the limiter; more threads only increase memory contention without more throughput.
* **Divergence & serialization:** if threads diverge more with larger blocks, some lanes idle and overall performance reduces.
* **Cache behavior:** larger thread-blocks might thrash caches or reduce locality in unexpected ways.