|  |
| --- |
| PARALLEL AND DISTRIBUTED COMPUTING |
| ASSIGNMENT 2 |
| OMAR ISMAIL – SP23-BCS-110 |

**CODING EXAMPLE:**

**IMAGE INVERSION:**

from numba import cuda

import numpy as np

import time

@cuda.jit

def invert\_img(image):

x, y = cuda.grid(2) # get 2D thread indices

if x < image.shape[0] and y < image.shape[1]:

image[x, y] = 255 - image[x, y]

image\_matrix = np.random.randint(0, 256, (128, 128), dtype=np.uint8)

d\_image = cuda.to\_device(image\_matrix)

threadsperblock = (8, 8)

blockspergrid\_x = (image\_matrix.shape[0] + threadsperblock[0] - 1) // threadsperblock[0]

blockspergrid\_y = (image\_matrix.shape[1] + threadsperblock[1] - 1) // threadsperblock[1]

blockspergrid = (blockspergrid\_x, blockspergrid\_y)

start = time.time()

invert\_img[blockspergrid, threadsperblock](d\_image)

inverted = d\_image.copy\_to\_host()

end = time.time()

print("Original:\n", image\_matrix)

print("Inverted:\n", inverted)

print("Execution time: \n", end - start)

**NOTE: NUMBA IS BROKEN IN GOOGLE COLAB DUE TO PTX VERSION 8.4 AND NUMBA REQUIRING VERSION 8.5 AND FOR DOWNGADING NUMBA IT REQUIURES PYTHON 3.10 BUT IT IS NOT DOWNGRADING.**

**ANALYSIS QUESTION:**

**OVERVIEW: Automatically select threads per block and blocks per grid according to image.**

def auto\_config(image\_shape, max\_threads\_per\_block):

rows, cols = image\_shape

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

(32, 8), (8, 32), (8, 16), (16, 8)]

# Pick the largest valid block size that does not exceed the image size

for bx, by in candidates:

if bx <= rows and by <= cols and bx \* by <= max\_threads\_per\_block:

threadsperblock = (bx, by)

break

else:

threadsperblock = (min(16, rows), min(16, cols))

# Compute grid size

blockspergrid\_x = (rows + threadsperblock[0] - 1) // threadsperblock[0]

blockspergrid\_y = (cols + threadsperblock[1] - 1) // threadsperblock[1]

blockspergrid = (blockspergrid\_x, blockspergrid\_y)

return blockspergrid, threadsperblock

from numba import cuda

import numpy as np

import time

@cuda.jit

def brighten\_img(image):

x, y = cuda.grid(2) # get 2D thread indices

if x < image.shape[0] and y < image.shape[1]:

image[x, y] = image[x, y] \* 1.2

image\_matrix = np.random.randint(0, 256, (128, 128), dtype=np.uint8)

d\_image = cuda.to\_device(image\_matrix)

blockspergrid, threadsperblock = auto\_config((image\_matrix.shape[0], 1024))

start = time.time()

invert\_img[blockspergrid, threadsperblock](d\_image)

inverted = d\_image.copy\_to\_host()

end = time.time()

print("Original:\n", image\_matrix)

print("Inverted:\n", inverted)

print("Execution time: \n", end - start)

**DISCUSSION QUESTION:**

**Why does increasing the number of threads per block not always improve performance? Consider register pressure, shared memory limits, and scheduling.**

Limitations in GPU hardware resources like registers and shared memory, which leads to register spilling and reduced occupancy. Excessive threads per block can also exceed the SM's capacity for running blocks in parallel, reducing overall GPU utilization. Finally, factors like warp divergence and inefficient memory access patterns can be exacerbated by larger block sizes, further hindering performance.

**CONCEPTUAL QUESTION:**

* Choosing a block size not a multiple of the 32-thread warp size leads to underutilization because the GPU schedules threads in warps, so any remaining threads in an incomplete warp are idle, wasting computational resources
* Occupancy of an SM depends on the block size and the threads per block because a higher number of active warps per SM increases occupancy and improves performance by keeping the SM's execution units busy and hiding latency.