

Asiignment 2

Name: Urwah Muhammad

Roll No.: SP23-BCS-123

Section: SP23-BCS-C

Subject: Parallel and Distributed Computing

Submitted to: Sir Akhzar Nazir



**Assignment 2**

# Conceptual Questions

* Non-multiples of 32 waste threads, because the GPU still allocates a full warp if the there are fewer threads i.e. If there are 33 threads, 2 warps will be allocated (64 threads) where 32 are given to one warp and 1 thread is given to the 2nd warp, wasting the other 31 threads.
* Occupancy is the ratio of active warps on an SM to its maximum capacity. Too small block sizes don’t launch enough warps, while too large ones consume too many resources and limit blocks per SM. An optimal mid-range block size (128–512 threads) balances resource use and keeps occupancy high.

# Practical / Coding Question

## Code:

#include <iostream>

#include <cuda\_runtime.h>

#include <opencv2/opencv.hpp>

// CUDA kernel for RGB inversion

\_\_global\_\_ void invert\_image(const unsigned char\* input, unsigned char\* output,

int width, int height, int channels) {

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

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

if (x < width && y < height) {

int idx = (y \* width + x) \* channels;

for (int c = 0; c < channels; c++) {

output[idx + c] = 255 - input[idx + c];

}

}

}

// Run inversion with timing (CUDA events)

float run\_inversion(const cv::Mat& img, cv::Mat& result, dim3 blockDim, int runs = 50) {

int width = img.cols;

int height = img.rows;

int channels = img.channels();

size\_t size = width \* height \* channels \* sizeof(unsigned char);

unsigned char \*d\_input, \*d\_output;

cudaMalloc(&d\_input, size);

cudaMalloc(&d\_output, size);

cudaMemcpy(d\_input, img.data, size, cudaMemcpyHostToDevice);

dim3 gridDim((width + blockDim.x - 1) / blockDim.x,

(height + blockDim.y - 1) / blockDim.y);

// CUDA events for timing

cudaEvent\_t start, stop;

cudaEventCreate(&start);

cudaEventCreate(&stop);

// Warm-up

invert\_image<<<gridDim, blockDim>>>(d\_input, d\_output, width, height, channels);

cudaDeviceSynchronize();

cudaEventRecord(start);

for (int i = 0; i < runs; i++) {

invert\_image<<<gridDim, blockDim>>>(d\_input, d\_output, width, height, channels);

}

cudaEventRecord(stop);

cudaEventSynchronize(stop);

float elapsed\_ms;

cudaEventElapsedTime(&elapsed\_ms, start, stop);

// Copy back one result

cudaMemcpy(result.data, d\_output, size, cudaMemcpyDeviceToHost);

// Cleanup

cudaFree(d\_input);

cudaFree(d\_output);

cudaEventDestroy(start);

cudaEventDestroy(stop);

return elapsed\_ms / runs; // average time

}

int main() {

cv::Mat img = cv::imread("input.jpg", cv::IMREAD\_COLOR);

if (img.empty()) {

std::cerr << "Error: Could not load image!\n";

return -1;

}

cv::Mat result(img.size(), img.type());

dim3 block\_sizes[] = { dim3(8,8), dim3(16,16), dim3(32,32) };

for (auto block : block\_sizes) {

float t = run\_inversion(img, result, block);

std::cout << "Block size (" << block.x << "," << block.y << ") -> "

<< t << " ms (avg)\n";

std::string fname = "inverted\_" + std::to\_string(block.x) + "x" +

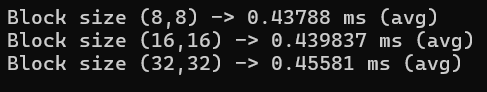
std::to\_string(block.y) + ".jpg";

cv::imwrite(fname, result);

}

return 0;

}



Block size **(8,8)** runs fastest (0.43788 ms) because smaller blocks give more scheduling flexibility and better latency hiding. Larger blocks like (32,32) reduce SM occupancy, so performance drops slightly.

# Analysis Question

Neither the smallest (64) nor the largest (1024) block size is optimal.  
The **best performance (256 threads per block)** comes from striking a balance:

* Enough threads to fully utilize GPU resources.
* But not so many that register pressure or shared memory limits reduce the number of concurrent blocks.

This is why **Case B is the fastest**.

## Code

#include <iostream>

#include <cuda\_runtime.h>

#include <opencv2/opencv.hpp>

// CUDA kernel: image inversion

\_\_global\_\_ void invert\_image(const unsigned char\* input, unsigned char\* output,

int width, int height, int channels) {

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

int total = width \* height;

if (idx < total) {

int pixel\_start = idx \* channels;

for (int c = 0; c < channels; c++) {

output[pixel\_start + c] = 255 - input[pixel\_start + c];

}

}

}

// Run kernel with given block size and return execution time (ms)

float run\_case(const cv::Mat& img, cv::Mat& result, int threadsPerBlock) {

int width = img.cols;

int height = img.rows;

int channels = img.channels();

int total = width \* height;

size\_t size = total \* channels \* sizeof(unsigned char);

unsigned char \*d\_input, \*d\_output;

cudaMalloc(&d\_input, size);

cudaMalloc(&d\_output, size);

cudaMemcpy(d\_input, img.data, size, cudaMemcpyHostToDevice);

int blocks = (total + threadsPerBlock - 1) / threadsPerBlock;

// CUDA events for accurate timing

cudaEvent\_t start, stop;

cudaEventCreate(&start);

cudaEventCreate(&stop);

cudaEventRecord(start);

invert\_image<<<blocks, threadsPerBlock>>>(d\_input, d\_output, width, height, channels);

cudaEventRecord(stop);

cudaDeviceSynchronize();

float ms = 0;

cudaEventElapsedTime(&ms, start, stop);

cudaMemcpy(result.data, d\_output, size, cudaMemcpyDeviceToHost);

cudaFree(d\_input);

cudaFree(d\_output);

cudaEventDestroy(start);

cudaEventDestroy(stop);

return ms;

}

int main() {

cv::Mat img = cv::imread("input.jpg", cv::IMREAD\_COLOR);

if (img.empty()) {

std::cerr << "Error: Could not load image!\n";

return -1;

}

cv::Mat result(img.size(), img.type());

# Discussion Question

Increasing threads per block doesn’t always help because larger blocks consume more registers and shared memory, which can limit how many blocks fit on an SM. This reduces occupancy and scheduling flexibility, sometimes making performance worse instead of better