Ειδικά Θέματα Παράλληλου Προγραμματισμού 2023-24

Σετ Ασκήσεων 3

Παναγιώτης Τριανταφυλλίδης

AM: 3200199

p3200199@aueb.gr

Το αντικείμενο της εργασίας είναι η φόρτωση και επεξεργασία εικόνων. Κάθε εικόνα αποτελείται από έναν αριθμό pixels όπου το χρώμα του κάθε πίξελ αναπαρίσταται από τις τιμές των 3 βασικών χρωμάτων Red , Green , Blue . Συνήθως σε αυτά τα χρωματικά κανάλια προσθέτουμε και ένα τέταρτο κανάλι, Alpha , το οποίο αναπαριστά το ποσοστό της διαφάνειας του pixel. Τα κανάλια αποθηκεύονται ως μεταβλητές τύπου unsigned char , που παίρνουν τιμές από το 0 (καθόλου συνεισφορά) έως το 255 (μέγιστη συνεισφορά). Η κάθε εικόνα αποθηκεύεται ως ένας πίνακας από unsigned chars μεγέθους [width * height * number_of_channels].

Οι μετρήσεις έγιναν στο παρακάτω σύστημα:

Επεξεργαστής	AMD Ryzen 5 3500U with Radeon Vega Mobile Gfx
Πλήθος πυρήνων	4
Πλήθος λογικών πυρήνων	8
Λειτουργικό σύστημα	Windows 10
Μεταφραστής	MSVC v143

Άσκηση 1

Το πρόβλημα

Στην πρώτη εργασία (HW1) μας είχε δοθεί η συνάρτηση gaussian_blur_separate_serial() , η οποία εφαρμόζει την τεχνική δύο περασμάτων από Gaussian Blur ώστε να επιταχύνει το αποτέλεσμα της θόλωσης της εικόνας “street_night.jpg”. Μας ζητείται να δημιουργήσουμε μία νέα συνάρτηση (gaussian_blur_separate_parallel()), στην οποία να φορτώνεται η ίδια εικόνα και έπειτα να γίνεται η θόλωση παράλληλα, χρησιμοποιώντας την OpenCL. Τέλος, να αποθηκεύσουμε την εικόνα σε ένα αρχείο με όνομα “image_blurred_final.jpg”.

Μέθοδος παραλληλοποίησης

Χρησιμοποιήθηκε το σειριακό πρόγραμμα της gaussian_blur_separate_serial για την δομή της gaussian_blur_separate_parallel αλλά και τα εργαλεία της βιβλιοθήκης της OpenCL για την εκτέλεση.

Για τον ορισμό του OpenCL context ορίζουμε ένα GPU device που διαθέτει ο υπολογιστής μας ως εξής:

```
// Helper function to get the OpenCL device
cl_device_id get_device() {
    cl_uint platform_count;
    clGetPlatformIDs(0, nullptr, &platform_count);
    std::vector<cl_platform_id> platforms(platform_count);
    clGetPlatformIDs(platform_count, platforms.data(), nullptr);

    cl_device_id device_id;
    for (auto platform : platforms) {
        cl_uint device_count;
        clGetDeviceIDs(platform, CL_DEVICE_TYPE_GPU, 1, &device_id,
&device_count);
        if (device_count > 0) {
            return device_id;
        }
    }

    // If no GPU device is found, use CPU
    for (auto platform : platforms) {
        cl_uint device_count;
        clGetDeviceIDs(platform, CL_DEVICE_TYPE_CPU, 1, &device_id,
&device_count);
        if (device_count > 0) {
            return device_id;
        }
    }

    throw std::runtime_error("No suitable OpenCL device found.");
}
```

και ορίστηκε το context ως εξής:

```
/ Create OpenCL context and device
cl_device_id device_id = get_device();
cl_context context = clCreateContext(nullptr, 1, &device_id, nullptr, nullptr,
nullptr);
cl_command_queue queue = clCreateCommandQueue(context, device_id, 0, nullptr);
```

Ταυτόχρονα ορίσαμε και την ουρά για την οργάνωση και τον συγχρονισμό εκτέλεσης εντολών στο device που ορίσαμε παραπάνω.

Έπειτα ο κώδικας για το Gaussian Blur γράφεται σε OpenCL C Language και αποθηκεύεται σε ένα αρχείο με όνομα kernel.cl:

```
// Save the kernel to a file
std::ofstream kernel_file("kernel.cl");
kernel_file << kernel_source;
kernel_file.close();
```

```
// The kernel source as a global variable
const char* kernel_source = R"(
#define KERNEL_RADIUS 8
__kernel void gaussian_blur(__global const uchar* input, __global uchar*
output, __global const float* weights, int width, int height, int axis) {
    int x = get_global_id(0);
    int y = get_global_id(1);
    int c = get_global_id(2);
    float sum_weight = 0.0f;
    float ret = 0.0f;
    for (int offset = -KERNEL_RADIUS; offset <= KERNEL_RADIUS; offset++) {
        int offset_x = axis == 0 ? offset : 0;
        int offset_y = axis == 1 ? offset : 0;
        int pixel_x = clamp(x + offset_x, 0, width - 1);
        int pixel_y = clamp(y + offset_y, 0, height - 1);
        int pixel_index = (pixel_y * width + pixel_x) * 4 + c;

        float weight = weights[offset + KERNEL_RADIUS];

        ret += weight * input[pixel_index];
        sum_weight += weight;
    }

    int output_index = (y * width + x) * 4 + c;
    output[output_index] = (uchar)clamp(ret / sum_weight, 0.0f, 255.0f);
}
)";
```

Μπορούμε να παρατηρήσουμε ότι κάθε work item δουλεύει σε τρισδιάστατο χώρο εκτέλεση του kernel. Ο ορισμός τους γίνεται στο global_work_size:

```
// Tune local worksize
size_t local_work_size[3] = { std::get<2>(combination),
std::get<1>(combination), std::get<0>(combination) };
```

Ενώ αντίστοιχα δημιουργούμε τον kernel απευθείας από το kernel_source:

```
// Create and build the program
const char* kernel_source_cstr = kernel_source;
cl_program program = clCreateProgramWithSource(context, 1, &kernel_source_cstr,
nullptr, nullptr);
clBuildProgram(program, 1, &device_id, nullptr, nullptr, nullptr);

// Create kernel
cl_kernel kernel = clCreateKernel(program, "gaussian_blur", nullptr);
```

Επίσης, παρατηρούμε ότι τα βάρη στην εκτέλεση της σειριακής έκδοσης υπολογίζονται δύο φορές (οριζόντια και κατακόρυφη θόλωση). Για αυτόν τον λόγο θα τα υπολογίσουμε μια φορά και θα τα περάσουμε σαν παραμέτρους από τον device κώδικα στον kernel.

Ωστόσο δεν θα υπολογίσουμε τον χρόνο εκτέλεσης τους στον συνολικό χρόνο της θόλωσης της εικόνας.

```
// Function to precompute Gaussian weights
std::vector<float> compute_gaussian_weights(int radius, float sigma) {
    std::vector<float> weights(radius * 2 + 1);
    float sum = 0.0;
    for (int i = -radius; i <= radius; i++) {
        float weight = exp(-(i * i) / (2.f * sigma * sigma));
        weights[i + radius] = weight;
    }
    return weights;
}
```

ενώ εδώ περνάμε τις παραμέτρους της καλώντας την compute gaussian weights:

```
// Precompute Gaussian weights
int radius = 8;
float sigma = 3.0f;
std::vector<float> weights = compute_gaussian_weights(radius, sigma);
```

Εκτελούμε τον πυρήνα δύο φορές, μια για κάθετο και οριζόντιο πέρασμα, ώστε να υπολογιστεί η θολωμένη εικόνα. Ταυτόχρονα καθορίζουμε τον αριθμό των work items με το global_work_size σε 3D χώρο εκτέλεσης στον kernel. Κάθε work item θα επεξεργαστεί ένα συγκεκριμένο κανάλι χρώματος ενός συγκεκριμένου pixel.

```
// Allocate buffers
cl_mem input_buffer = clCreateBuffer(context, CL_MEM_READ_ONLY |
CL_MEM_COPY_HOST_PTR, width * height * 4 * sizeof(unsigned char), img_in,
nullptr);
cl_mem output_buffer = clCreateBuffer(context, CL_MEM_WRITE_ONLY, width *
height * 4 * sizeof(unsigned char), nullptr, nullptr);
cl_mem weight_buffer = clCreateBuffer(context, CL_MEM_READ_ONLY |
CL_MEM_COPY_HOST_PTR, weights.size() * sizeof(float), weights.data(), nullptr);

unsigned char* img_horizontal_blur = new unsigned char[width * height * 4];

// Set kernel arguments for horizontal blur
clSetKernelArg(kernel, 0, sizeof(cl_mem), &input_buffer);
clSetKernelArg(kernel, 1, sizeof(cl_mem), &output_buffer);
clSetKernelArg(kernel, 2, sizeof(cl_mem), &weight_buffer);
clSetKernelArg(kernel, 3, sizeof(int), &width);
clSetKernelArg(kernel, 4, sizeof(int), &height);
int axis = 0;
clSetKernelArg(kernel, 5, sizeof(int), &axis);

// Define the global work size (width * height * channels)
channels = 4;
size_t global_work_size[3] = { static_cast<size_t>(width),
static_cast<size_t>(height), static_cast<size_t>(channels) };

// Timer to measure performance
auto start = std::chrono::high_resolution_clock::now();
```

```
// Execute kernel for horizontal blur
clEnqueueNDRangeKernel(queue, kernel, 3, nullptr, global_work_size,
local_work_size, 0, nullptr, nullptr);
clFinish(queue);  // Ensure the kernel execution is finished

    // Read the horizontal blur result back to host memory
clEnqueueReadBuffer(queue, output_buffer, CL_TRUE, 0, width * height * 4 *
sizeof(unsigned char), img_horizontal_blur, 0, nullptr, nullptr);

    // Prepare for vertical blur
clSetKernelArg(kernel, 0, sizeof(cl_mem), &output_buffer);
clSetKernelArg(kernel, 1, sizeof(cl_mem), &input_buffer);
axis = 1;
clSetKernelArg(kernel, 5, sizeof(int), &axis);

// Execute kernel for vertical blur
clEnqueueNDRangeKernel(queue, kernel, 3, nullptr, global_work_size,
local_work_size, 0, nullptr, nullptr);
clFinish(queue);  // Ensure the kernel execution is finished

// Read the final result back to host memory
clEnqueueReadBuffer(queue, input_buffer, CL_TRUE, 0, width * height * 4 *
sizeof(unsigned char), img_in, 0, nullptr, nullptr);

// Timer to measure performance
auto end = std::chrono::high_resolution_clock::now();
// Computation time in milliseconds
int time = (int)std::chrono::duration_cast<std::chrono::milliseconds>(end -
start).count();
```

Υπάρχουν κάποια σημεία κλειδιά τα οποία πρέπει να προσέξουμε παραπάνω:

- το axis συμβολίζει την οριζόντια (0) ή κατακόρυφη θόλωση (1).
- κάποιες μεταβλητές δεν τις περνάμε στο kernel στην κατακόρυφη θόλωση (πχ weights, width, height) καθώς έχουν ήδη οριστεί από την οριζόντια θόλωση.
- αλλάζουμε μόνο τις μεταβλητές των εικόνων εισόδου και εξόδου στο kernel σε κάθε θόλωση και τον άξονα.
- περιμένουμε πάντα να τελειώσει η ουρά εκτέλεσης πριν συνεχίσουμε στο επόμενο βήμα κάθε φορά.

Τέλος εγγράφουμε την θολωμένη εικόνα στο “image_blurred_final.jpg”.

```
// Write the blurred image into a JPG file
stbi_write_jpg("image_blurred_final.jpg", width, height, 4, img_in, 90);
```

Ωστόσο, ίσως να παρατηρήθηκε από κάποιους αλλά το local_work_size δεν παίρνει σταθερές τιμές. Αυτό γιατί στην συνάρτηση main του προγράμματος πειραματιζόμαστε για διαφορετικούς συνδυασμούς.

Πειραματικά αποτελέσματα – μετρήσεις

Η χρονομέτρηση έγινε με τις εντολές :

```
// Timer to measure performance
auto start = std::chrono::high_resolution_clock::now();

auto end = std::chrono::high_resolution_clock::now();

// Computation time in milliseconds
int time = (int)std::chrono::duration_cast<std::chrono::milliseconds>(end -
start).count();
```

Για αρχή εκτελέσαμε την gaussian_blur_separate_parallel 4 φορές για κάθε συνδυασμό που μας δίνει γινόμενο 256 αλλά και για την σειριακή έκδοση. Έπειτα για την σύγκριση μεταξύ τους πήραμε τον μέσο όρο.

```
const char* filename = "street_night.jpg";
const int NUM_TESTS = 4; // Number of times to execute each test

// Execute serial Gaussian blur four times and calculate average time
std::cout << "Serial Gaussian blur times (4 executions) and average time:\n";
std::vector<double> serialTimes;
double totalSerialTime = 0.0;
for (int i = 0; i < NUM_TESTS; ++i) {
    double executionTime = gaussian_blur_separate_serial(filename);
    totalSerialTime += executionTime;
    serialTimes.push_back(executionTime);
}
double avgSerialTime = totalSerialTime / NUM_TESTS;
for (const auto& time : serialTimes) {
    std::cout << time << "ms ";
}
std::cout << " Average time: " << avgSerialTime << "ms\n";
// Define a vector of tuples to store the combinations
std::vector<std::tuple<int, int, int>> combinations = {
    {2, 4, 32},
    {2, 8, 16},
    {4, 4, 16},
    {2, 2, 64},
    {4, 8, 8},
    {1, 16, 16},
    {1, 8, 32},
    {1, 4, 64},
    {1, 2, 128}
};
```

```
// Execute parallel Gaussian blur for each combination, calculate average time,
and print results
std::cout << "\nParallel Gaussian blur times (4 executions) and average time
for each combination:\n";
std::tuple<int, int, int> minAvgCombination;
double minAvgTime = std::numeric_limits<double>::max();
for (const auto& combination : combinations) {
    std::cout << "Combination: " << std::get<0>(combination) << " "
        << std::get<1>(combination) << " "
        << std::get<2>(combination) << " : ";
    std::vector<double> times;
    double totalParallelTime = 0.0;
    for (int i = 0; i < NUM_TESTS; ++i) {
        double executionTime = gaussian_blur_separate_parallel(filename,
combination);
        totalParallelTime += executionTime;
        times.push_back(executionTime);
    }
    double avgParallelTime = totalParallelTime / NUM_TESTS;
    if (avgParallelTime < minAvgTime) {
        minAvgTime = avgParallelTime;
        minAvgCombination = combination;
    }
    for (const auto& time : times) {
        std::cout << time << "ms ";
    }
    std::cout << " Average time: " << avgParallelTime << "ms\n";
}

// Print the combination with the minimum average time
std::cout << "\nCombination with the minimum average time: "
    << std::get<0>(minAvgCombination) << " "
    << std::get<1>(minAvgCombination) << " "
    << std::get<2>(minAvgCombination) << " : "
    << minAvgTime << "ms\n";
```

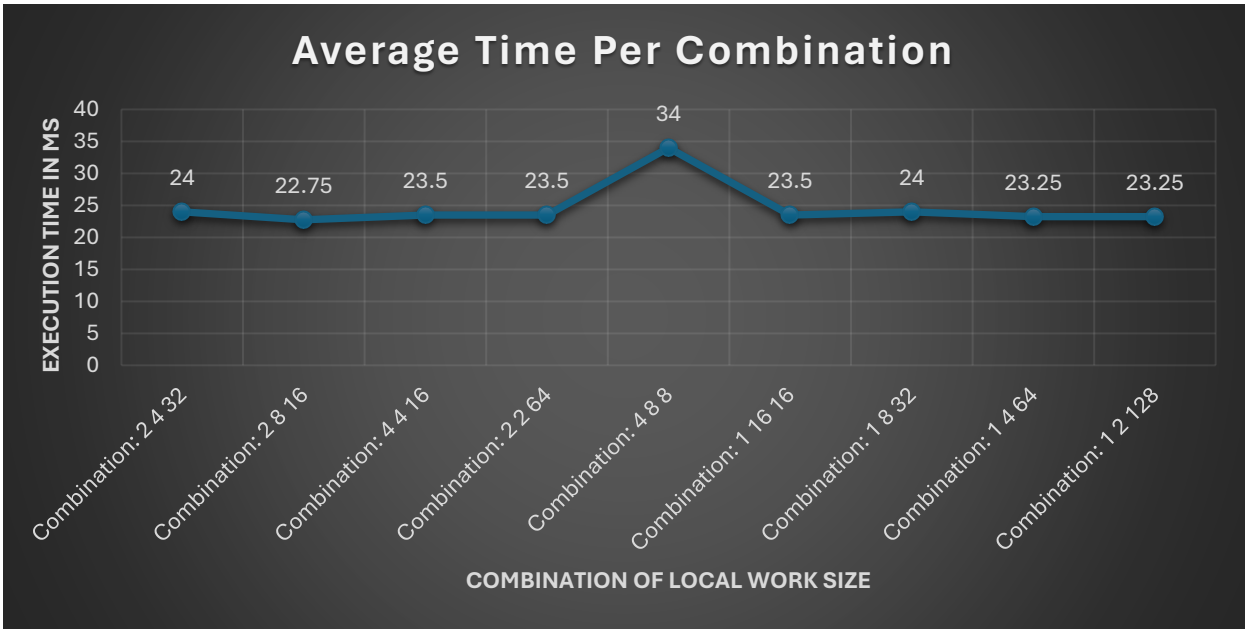
Οι μετρήσεις είναι οι εξής:

Serial	Combination: 2 4 32	Combination: 2 8 16	Combination: 4 4 16	Combination: 2 2 64	Combination: 4 8 8	Combination: 1 16 16	Combination: 1 8 32	Combination: 1 4 64	Combination: 1 2 128
24761	26	23	23	23	35	23	24	23	24
24053	23	23	24	24	34	24	25	23	23
27603	23	23	24	23	35	24	23	23	23
25827	24	22	23	24	32	23	24	24	23

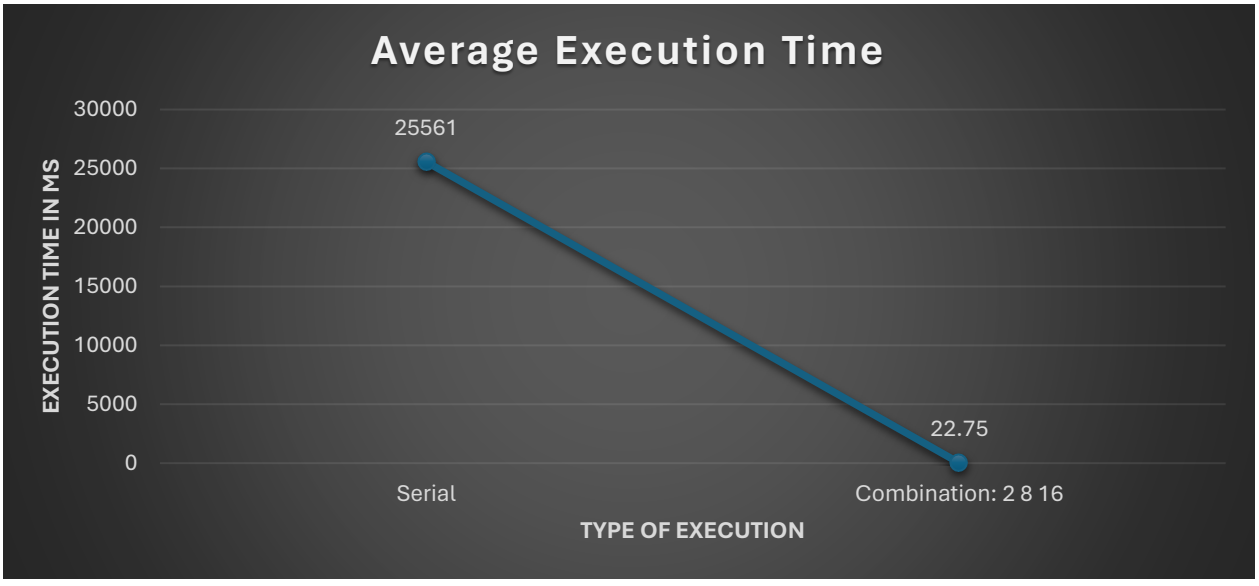
και αντίστοιχα οι μέσοι όροι τους:

Serial	Combination: 2 4 32	Combination: 2 8 16	Combination: 4 4 16	Combination: 2 2 64	Combination: 4 8 8	Combination: 1 16 16	Combination: 1 8 32	Combination: 1 4 64	Combination: 1 2 128
25561	24	22.75	23.5	23.5	34	23.5	24	23.25	23.25

Διαγραμματικά:



και συγκριτικά ο συνδυασμός (2x8x16) με το serial:



Σχόλια

Η επίτευξη του βέλτιστου συνδυασμού διαστάσεων εργασίας (2x8x16) υποδεικνύει πόσο κρίσιμη είναι η σωστή ρύθμιση των παραμέτρων στις παράλληλες επεξεργασίες. Ο συνδυασμός αυτός επιτρέπει την καλύτερη αξιοποίηση των διαθέσιμων πυρήνων του επεξεργαστή GPU, ελαχιστοποιώντας τον χρόνο αναμονής και μεγιστοποιώντας την αποδοτικότητα της υπολογιστικής ισχύος. Σε περιπτώσεις όπου οι παράμετροι δεν είναι καλά ρυθμισμένες, όπως σε συνδυασμούς που επιτυγχάνουν χρόνο γύρω στα 34 ms, παρατηρείται σημαντική μείωση της αποδοτικότητας, καταδεικνύοντας την ανάγκη για λεπτομερή ανάλυση και πειραματισμό με διάφορες παραμέτρους.

Επιπλέον, οι διαφορές μεταξύ των παράλληλων συνδυασμών μας δείχνουν πως δεν υπάρχει ένας "χρυσός κανόνας" που να ισχύει για όλες τις εφαρμογές ή τα δεδομένα. Η φύση της κάθε εργασίας και τα χαρακτηριστικά της εικόνας που επεξεργαζόμαστε μπορούν να επηρεάσουν τον βέλτιστο συνδυασμό. Για παράδειγμα, διαφορετικές εικόνες μπορεί να απαιτούν διαφορετικά μεγέθη τοπικών εργασιών για να επιτευχθεί η καλύτερη απόδοση. Η προσεκτική επιλογή και ο συνδυασμός των παραμέτρων μπορεί να οδηγήσει σε βελτιώσεις απόδοσης που να μην είναι αρχικά προφανείς. Αν και όλοι οι συνδυασμοί είναι βελτιωμένοι σε σχέση με τη σειριακή εκτέλεση, οι διαφορές στους χρόνους εκτέλεσης μεταξύ των διαφόρων συνδυασμών δείχνουν ότι η σωστή ρύθμιση των παραμέτρων μπορεί να έχει σημαντική επίπτωση στην απόδοση.

Τέλος, η δραματική μείωση του χρόνου εκτέλεσης από 25561 ms σε 22.75 ms δείχνει τη δύναμη των GPUs στην παράλληλη επεξεργασία. Το ποσοστό βελτίωσης που προκύπτει, περίπου 99.9%, αποδεικνύει ότι η παράλληλη επεξεργασία μπορεί να προσφέρει τεράστια βελτίωση στην απόδοση για κατάλληλες εργασίες όπως το Gaussian blur. Αυτή η απόδοση οφείλεται στην ικανότητα της GPU να διαχειρίζεται ταυτόχρονα μεγάλο αριθμό υπολογισμών, κάτι που είναι εξαιρετικά δύσκολο να επιτευχθεί με σειριακές μεθόδους σε CPU.

** *Στα συμπιεσμένα αρχεία συμπεριλαμβάνεται το βιβλίο εργασίας excel που περιέχει τις μετρήσεις και για την 3η άσκηση. Η εκτέλεση έγινε σε περιβάλλον IDE: Microsoft Visual Studio και σε γλώσσα C++ στην έκδοση v20. Οι παραγόμενες εικόνες βρίσκονται στην ίδιο φάκελο με τον πηγαίο κώδικα.*