**Ειδικά Θέματα Παράλληλου Προγραμματισμού 2023-24**

Σετ Ασκήσεων 3

Παναγιώτης Τριανταφυλλίδης

ΑΜ: 3200199

[p3200199@aueb.gr](mailto: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 σύγκρισή 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.*