Ειδικά Θέματα Παράλληλου Προγραμματισμού

2023-2024

Εργασία 3

Κωνσταντίνος Ελευθερίου

3200283

Ζήτημα Α

Η υλοποίηση της παραλληλοποίησης με OpenCL ξεκινάει με την φόρτωση της εικόνας και την αρχικοποίηση μεταβλητών που θα χρησιμοποιήσουμε εν συνέχεια.

Task 1 – Δημιουργία Context
Ξεκινάμε, δημιουργώντας ένα context, το context είναι ένα container για ένα συγκεκριμένο platform το οποίο περιέχει τα devices αλλά και μνήμη. Αυτό χρειάζεται για να γίνει manage η OpenCL.
Προτού όμως πάμε και δημιουργήσουμε το context, θα πρέπει να βρούμε ένα platform και τα αντίστοιχα devices GPU η CPU τα οποία είναι συμβατά μέσα σε αυτό (η device εάν είναι μοναδικό). Η έννοια του platform είναι σημαντική καθώς έχουμε αποδοτική μεταφορά δεδομένων μεταξύ των devices μέσα σε αυτό.

Αυτό γίνεται εύκολα με τις αντίστοιχες εντολές που μας προσφέρει η OpenCL:

```
// platform is essentially a container of devices, we are creating a context for a device
std::cout << "Creating context" << "\n";
;
cl_platform_id platform;
clGetPlatformIDs(1, &platform, nullptr);

// getting the device
cl_device_id device;
clGetDeviceIDs(platform, CL_DEVICE_TYPE_GPU, 1, &device, nullptr);
if (!device) {
    clGetDeviceIDs(platform, CL_DEVICE_TYPE_CPU, 1, &device, nullptr);
}
std::cout << "Device created" << "\n";

cl_device_type device_type;
clGetDeviceInfo(device, CL_DEVICE_TYPE, sizeof(cl_device_type), &device_type, nullptr);
if (device_type == CL_DEVICE_TYPE_GPU) {
```

```
    std::cout << "Device type: GPU" << std::endl;
} else if (device_type == CL_DEVICE_TYPE_CPU) {
    std::cout << "Device type: CPU" << "\n";
}
```

Ύστερα, δημιουργούμε το context:

```
cl_context context = clCreateContext(nullptr, 1, &device, nullptr, nullptr, nullptr);
```

Task 2 – Δημιουργία συνάρτησης blur σε OpenCL που θα εκτελεστεί από στο device.

Ύστερα, δημιουργούμε τον kernel κώδικα γραμμένο σε OpenCL που θα εκτελεστεί από το αντίστοιχο device που έχει βρεθεί στο context.

Για λόγους απλότητας, παραθέτω μόνο τα σημαντικά σημεία του kernel κώδικα.
Ο kernel κώδικας, λαμβάνει την **αρχική εικόνα** (input) την **τελική μορφή** που θα έχει μετά το πέρασμα blur (output), τα προϋπολογισμένα **weights**, το **πλάτος** και **ύψος** της εικόνας και ένα variable **axis** το οποίο μας λέει εάν το blurring θα γίνει vertically η horizontally.

```
"__kernel void blur(__global unsigned char* input, __global unsigned char* output, __global float*
weights, int width, int height, int axis) {\n"
"    int x = get_global_id(0);\n"
"    int y = get_global_id(1);\n"
"    int pixel = y * width + x;\n"
```

Στην OpenCL, χρησιμοποιείται η έννοια του global και local dimensions. Τα global dimensions είναι ένας τρόπος να ορίσουμε την παραλληλοποίηση ενός αντικειμένου. Τα global dimensions μπορεί να οριστεί ως 1D, 2D η και 3D. Ένα work item (η thread) εκτελείται για κάθε ένα σημείο του global dimension.

Στην συγκεκριμένη περίπτωση, εμείς θέλουμε να θολώσουμε χρησιμοποιώντας τεχνικές παραλληλοποίησης μιας εικόνας διαστάσεων width x height. Παρόλα αυτά, έχουμε και την διάσταση των 4 channels (RGBA), επομένως, αν και θα μπορούσαμε να ορίσουμε ως global dimension ως έναν τρισδιάστατό χώρο, δηλαδή, width x height x channel work-items έχοντας 1 thread για κάθε pixel της εικόνας, θεωρώ πως είναι προτιμότερο ίσως να μείνει σαν ένα 2D όπου στην μια διάσταση θα είναι το width και στην άλλη το height. Συνολικά λοιπόν, έχουμε width x height work items. Ο λόγος που θα ήταν προτιμότερο είναι πως με το να έχουμε ένα 2D global dimension, είναι καθαρά στο ότι έτσι μπορούμε να κρατήσουμε την δομή πιο απλή (και ίσως πιο ευέλικτη).

Ταυτόχρονα, η απλούστευση των διαστάσεων μπορεί ίσως να συμβάλει στην καλύτερη χρήση της μνήμης cache και, κατά συνέπεια, στη βελτίωση της συνολικής απόδοσης, καθώς έτσι, κάθε work item στις δυο διαστάσεις θα έχει να υπολογίσει ένα for loop για τα 4 αυτά channels.

Όσον αφορά την εντολή της **get_global_id,**
Η get_global_id(0) λαμβάνει το id στην πρώτη διάσταση, ενώ η εντολή get_global_id(1) στην δεύτερη.

Ύστερα, αφού πλέον έχουμε ορίσει το kernel source, αποθηκεύουμε το αρχείο τοπικά.

```
bool success = saveKernelSource(kernel_source, "kernel.cl");
```

Task 3 – Προϋπολογισμός των blur weights

Συνεχίζουμε, προϋπολογίζοντάς τα blur weights, καθώς είναι ίδια και ξανά-υπολογίζονται σε κάθε pixel που εφαρμόζεται το φίλτρο.

```
std::vector<float> precalculateBlurWeights() {
    std::vector<float> weights = std::vector<float>(2 * KERNEL_RADIUS + 1);
    for (int offset = -KERNEL_RADIUS; offset <= KERNEL_RADIUS; ++offset) {
        weights[offset + KERNEL_RADIUS] = std::exp(-(offset * offset) / (2.0f * SIGMA * SIGMA));
    }
    return weights;
}
```

Task 4 – Εφαρμογή blur στην εικόνα μέσω του kernel κώδικα και των 2 axis.

Στο σημείο που έχουμε φτάσει, θέλουμε να εκτελέσουμε των κώδικα kernel στο device το οποίο έχει βρεθεί του αντίστοιχου context.

Προτού όμως γίνει αυτό, θα πρέπει αρχικά να δούμε πως θα μπορέσουμε να στείλουμε δεδομένα προς αυτά. Αυτό μπορεί να γίνει μέσω τον command queues. Το πρόγραμμα μας τοποθετεί δεδομένα σε αυτό το queue και τα στέλνει στα αντίστοιχα devices.

```
cl_command_queue command_queue = clCreateCommandQueue(context, device, 0, nullptr);
```

Στην συνέχεια, κάνουμε allocate τα δεδομένα που θα χρησιμοποιήσουμε, η μνήμη που χρησιμοποιείται είναι από την μνήμη του context που ζουν και τα devices. Αυτή η μνήμη μπορεί να χρησιμοποιηθεί από το OpenCL kernel.

```
// creating an OpenCL buffer to store the weights, input image, intermediate image and output image
cl_mem weights_buffer = clCreateBuffer(context, CL_MEM_READ_ONLY, (2 * KERNEL_RADIUS + 1) * sizeof(float), nullptr, nullptr);

cl_mem input_buffer = clCreateBuffer(context, CL_MEM_READ_ONLY, width * height * 4 * sizeof(unsigned char), nullptr, nullptr);

cl_mem intermediate_buffer = clCreateBuffer(context, CL_MEM_READ_WRITE, width * height * 4 * sizeof(unsigned char), nullptr, nullptr);

cl_mem output_buffer = clCreateBuffer(context, CL_MEM_WRITE_ONLY, width * height * 4 * sizeof(unsigned char), nullptr, nullptr);
```

Στην συνέχεια, γεμίζουμε την μνήμη που κάναμε allocate με την εικόνα μας, και τα pre-calculated weights στην αντίστοιχη θέση μνήμης που έγινε allocate για τα weights. Η εντολή αυτή μεταφέρει τα δεδομένα από την μνήμη του host στην global μνήμη του device. Από εκεί, καθώς η μνήμη είναι global είναι προσβάσιμη από όλα τα work-items στο OpenCL context, η οποία είναι και πιο γρήγορη.

```
cl_int clStatus = clEnqueueWriteBuffer(command_queue, input_buffer, CL_TRUE, 0, width * height * 4 * sizeof(unsigned char), img_in, 0, nullptr, nullptr);

clStatus = clEnqueueWriteBuffer(command_queue, weights_buffer, CL_TRUE, 0, (2 * KERNEL_RADIUS + 1) * sizeof(float), weights, 0, nullptr, nullptr);
```

Τέλος, αφού έχουμε δημιουργήσει και την κατάλληλη μνήμη δημιουργούμε το πρόγραμμα και το kernel, περνάμε τα attributes του kernel πού εξαρχής είχαμε κάνει allocate, και ορίζουμε το global work size και το local work size.

Όπως είπαμε και προηγουμένως, δημιουργεί μια ‘ζώνη’ παραλληλοποίησης, ενώ το local work size μας δίνει μια περιοχή η ένα group από work-items η threads τα οποία δουλεύουν μαζί. Αυτό γίνεται διότι ένα work group εκτελείται μαζί από ένα compute unit (η SM). Κάθε ένα work item που βρίσκεται στο ίδιο group με κάποιο άλλο, μπορούν να συγχρονιστούν, γιαυτο και είναι σημαντικός ο καθορισμός του size.

```
clStatus = clBuildProgram(program, 1, &device, nullptr, nullptr, nullptr);
if (clStatus != CL_SUCCESS) {
    std::cout << "Error building program" << "\n";
    return -1;
}

// create the kernel
cl_kernel kernel = clCreateKernel(program, "blur", nullptr);

// setting the kernel arguments for the horizontal pass
clSetKernelArg(kernel, 0, sizeof(cl_mem), &input_buffer);
clSetKernelArg(kernel, 1, sizeof(cl_mem), &intermediate_buffer);
clSetKernelArg(kernel, 2, sizeof(cl_mem), &weights_buffer);
clSetKernelArg(kernel, 3, sizeof(int), &width);
clSetKernelArg(kernel, 4, sizeof(int), &height);

int axis = 0; // horizontal
clSetKernelArg(kernel, 5, sizeof(int), &axis);

// setting the global and local work sizes
size_t global_work_size[2] = { static_cast<size_t>(width), static_cast<size_t>(height) };
size_t local_work_size[2] = {16, 16};
```

Το kernel εκτελείται για το πρώτο pass:

```
clStatus = clEnqueueNDRangeKernel(command_queue, kernel, 2, nullptr, global_work_size,
local_work_size, 0, nullptr, nullptr);
```

Ορίζουμε και πάλι τα attributes για την εκτέλεση του περάσματος 2:
Αυτήν την φορά ως input buffer τοποθετείται το buffer του αποτελέσματος του προηγουμένου step, ενώ ως output buffer ο χώρος στο οποίο θα κρατηθεί η τελική εικόνα.

```
clSetKernelArg(kernel, 0, sizeof(cl_mem), &intermediate_buffer);
clSetKernelArg(kernel, 1, sizeof(cl_mem), &output_buffer);
clSetKernelArg(kernel, 2, sizeof(cl_mem), &weights_buffer);
clSetKernelArg(kernel, 3, sizeof(int), &width);
```

```
clSetKernelArg(kernel, 4, sizeof(int), &height);

axis = 1; // vertical
clSetKernelArg(kernel, 5, sizeof(int), &axis);
```

Τέλος, εκτελούμε και το 2ο pass και αποθηκεύουμε την εικόνα:

```
clStatus = clEnqueueNDRangeKernel(command_queue, kernel, 2, nullptr, global_work_size,
local_work_size, 0, nullptr, nullptr);
if (clStatus != CL_SUCCESS) {
    std::cout << "Error executing kernel for vertical pass" << "\n";
    return -1;
}
clFinish(command_queue);


// Copy the output image from the output buffer
clStatus = clEnqueueReadBuffer(command_queue, output_buffer, CL_TRUE, 0, width * height * 4 *
sizeof(unsigned char), img_out, 0, nullptr, nullptr);
if (clStatus != CL_SUCCESS) {
    std::cout << "Error copying the output image from the output buffer" <<  "\n";
    return -1;
}
// Write the output image to a file
stbi_write_jpg("image_blurred_final.jpg", width, height, 4, img_out, 90);
```

Πρώτη Εκτέλεση	Δεύτερη Εκτέλεση	Τρίτη Εκτέλεση	Τέταρτη Εκτέλεση	Μέσος Όρος
0.253 sec	0.149 sec	0.141 sec	0.142 sec	**0.171 sec**

Παραλληλοποίηση της Separate Gaussian Blur με OpenCL

Εύρεση κατάλληλου local work size

Θα ξεκινήσουμε τώρα να δούμε, έναν ιδανικό αριθμό για το local dimensions. Η OpenCL μας δίνει διάφορες δυνατότητες όπως:

1) Εύρεση μεγίστου work group size

Αυτό μπορεί να γίνει μέσω της εντολής:

clGetDeviceInfo(device, CL_DEVICE_MAX_WORK_GROUP_SIZE, sizeof(size_t), &maxWorkGroupSize, nullptr);

Για το σύστημα μου, αυτό είναι 256 work items ανά group, το οποίο είναι 16x16 και οτιδήποτε άλλο το οποίο μας δίνει αυτά τα work items, σε διαστάσεις του local_work_size.

2) Προτιμότερο work group size

Αυτό μπορεί να γίνει μέσω της εντολής:

clGetKernelWorkGroupInfo(kernel, device, CL_KERNEL_PREFERRED_WORK_GROUP_SIZE_MULTIPLE, sizeof(size_t), &preferredWorkGroupSizeMultiple, nullptr);

Η OpenCL μας προτείνει 32 work items ανά group άρα οποιονδήποτε συνδυασμό 4x8, 8x4, 32x1, 1x32 κλπ.

Σε αυτό το σημείο, πρέπει να αναφερθεί πως, δεν γίνεται να ορίσουμε ένα local dimension (localDimensionX, localDimensionY) , τέτοιο ώστε:

Τα localDimensionX, localDimensionY να μην διαιρούνται ακριβώς με τα αντίστοιχα 2 dimensions του global dimension (στην συγκεκριμένη περίπτωση, αυτά είναι width και height).

Χρόνοι εκτέλεσης

Local Dimensions	Πρώτη Εκτέλεση	Δεύτερη Εκτέλεση	Τρίτη Εκτέλεση	Τέταρτη Εκτέλεση	Μέσος Όρος
32x1	0.332 sec	0.158 sec	0.403 sec	0.153 sec	0.261 sec
1x32	0.371 sec	0.250 sec	0.197 sec	0.157 sec	0.243 sec
8x4	0.229 sec	0.154 sec	0.150 sec	0.145 sec	0.169 sec
4x8	**0.158 sec**	**0.144 sec**	**0.144 sec**	**0.146 sec**	**0.148 sec**
16x2	0.164 sec	0.150 sec	0.144 sec	0.155 sec	0.153 sec
2x16	0.158 sec	0.158 sec	0.149 sec	0.146 sec	0.152 sec
32x8	0.321 sec	0.212 sec	0.201 sec	0.177 sec	0.227 sec
8x32	0.191 sec	0.188 sec	0.160 sec	0.156 sec	0.173 sec
16x16	0.253 sec	0.149 sec	0.141 sec	0.142 sec	0.171 sec

Είναι ενδιαφέρον να δει κανείς πως το αρχικό κόστος σε κάθε εκτέλεση είναι σχετικά μεγαλύτερο από ότι κάθε άλλη εκτέλεση. Αυτό ίσως συμβαίνει λόγο του kernel launch overhead που υπάρχει, και άρα το compilation μπορεί να είναι αρκετά αργό την πρώτη φορά.

Από τα δεδομένα αυτά όσον αφορά τους χρόνους εκτέλεσης, μπορούμε να αναλύσουμε τα εξής:

- Η **καλύτερη απόδοση** με μέσο χρόνο γύρω στα 0,148 δευτερόλεπτα και 0,152 δευτερόλεπτα, δίνεται από τα dimensions 4x8 και 2x16

- Οι "**χειρότερες**" **επιδόσεις**, βρίσκονται στα dimensions 32x1 και 1x32, με τους υψηλότερους μέσους χρόνους.

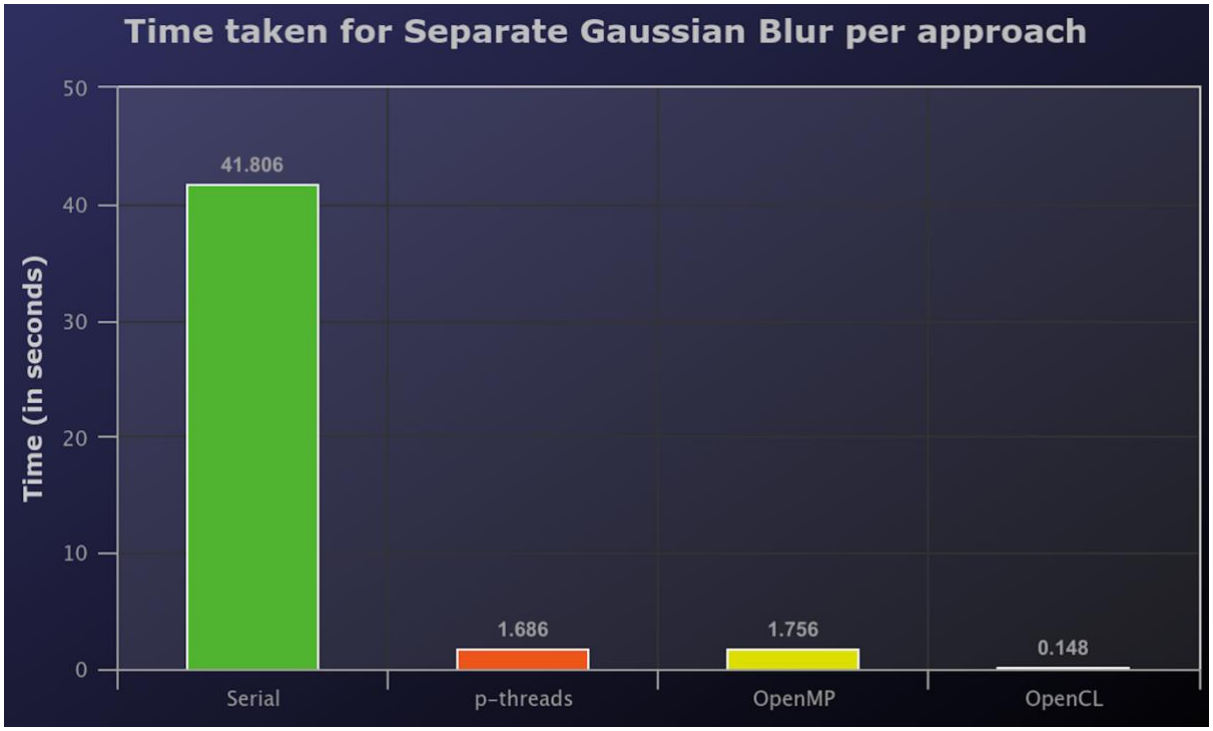
- Οι επιλογές 8x4 και 16x16 έχουν επίσης αρκετά καλές επιδόσεις, αλλά είναι ελαφρώς πιο αργές από τις αυτές με τις καλύτερες επιδόσεις.

Η εικόνα μας τώρα, βρίσκεται σε διαστάσεις, 2048x1024, άρα το global dimension είναι στα 2048x1024. Εάν πάρουμε ως local dimension αυτό που μας προτείνει η OpenCL, αυτό είναι οποιοδήποτε διάσταση (x, y) με 32 work-items, τότε, καθώς το σύστημά μου έχει 8 (εκτιμώμενα) cores ανά compute unit, τα 32 αυτά work-items διαιρούνται ακριβώς με τα cores και επομένως έχουμε ένα αποτελεσματικό device utilization.

Τελικές Συγκρίσεις

Τεχνική	Πρώτη Εκτέλεση	Δεύτερη Εκτέλεση	Τρίτη Εκτέλεση	Τέταρτη Εκτέλεση	Μέσος Όρος
Serial	41.966	41.949	41.412	41.899	**41.806**
Pthreads	1.664	1.678	1.735	1.668	**1.686**
OpenMP	1.773	1.743	1.785	1.769	**1.756**
OpenCL	0.158	0.144	0.144	0.146	**0.148**

Όλες οι μέθοδοι (εκτός από την σειριακή) εκτελούν τον αποδοτικό αλγόριθμο της Gaussian blur, με το αποδοτικότερο configuration.



Ο χρόνος σειριακής εκτέλεσης χρησιμεύει ως base-line αλγόριθμο για την σύγκριση των αποτελεσμάτων που βρήκαμε με τις διάφορες τεχνικές παραλληλοποίησης που χρησιμοποιήσαμε, αναμένεται λοιπόν να είναι ο μεγαλύτερος.

Εν συνέχεια, είδαμε τεχνικές παραλληλοποίησης τα Pthreads και την OpenMP. Και οι δύο αυτές τεχνικές είναι CPU-based. Η απόδοση τους είναι αρκετά κοντά μεταξύ τους κάτι το οποίο είναι αναμενόμενο. Η τεχνική παραλληλοποίησης με pthreads φαίνεται να προσφέρει ελάχιστα καλύτερη απόδοση, κάτι το οποίο μπορεί να δικαιολογηθεί λόγο του lower overhead που υπάρχει στο thread management. Τέλος, παρατηρούμε ότι οι τεχνικές αυτές καταφέρνουν να κάνουν 20 φορές πιο γρήγορη την διαδικασία της θόλωσης εικόνας

Η τεχνική παραλληλοποίησης μέσω της OpenCL μας δίνει δραστικά μεγαλύτερες ταχύτητες σε σχέση με τις 2 τελευταίες, και αυτό γιατί, όπως είπαμε, μπορεί να εκμεταλλευτεί την δύναμη των GPU στο ότι μπορεί να χειρίζεται παράλληλα χιλιάδες νήματα καθιστώντας την ιδανική σε τέτοιες περιπτώσεις.

Επιπλέον, η ευελιξία της OpenCL συνδυάζεται με τη δυνατότητα προσαρμογής της απόδοσης, καθιστώντας την ιδανική για hardware-specific βελτιστοποιήσεις. Λόγω του χαρακτήρα της ως η πιο low-level τεχνική προγραμματισμού, επωφελείται πλήρως από τις δυνατότητες του υλικού, κάτι που εξηγεί τις εξαιρετικές αποδόσεις της. Η αποδόσεις είναι τόσο καλές που φτάνουν την τάξη του να είναι έως και 200 φόρες πιο γρήγορες από ότι την σειριακή εκτέλεση, και έως 20 φορές καλύτερες από τις τεχνικές με pthreads και OpenMP.