Ειδικά Θέματα Παράλληλου Προγραμματισμού

Εργασία #2 – OpenCL (2022-23)

Γενικές Πληροφορίες

Η δεύτερη εργασία του μαθήματος αφορά προγραμματισμό με OpenCL, και συγκεκριμένα σε C. Όσον αφορά το μηχάνημα που χρησιμοποιήθηκε για τις μετρήσεις, ήταν με CPU Intel i5-10600KF (6 πυρήνων), με GPU Nvidia 1060 (6GB), σε gcc 8.1.0 (μέσω του Code::Blocks IDE σε Windows).

Άσκηση 1

Το Πρόβλημα

Στη δεύτερη εργασία, μας δόθηκε πάλι η σειριακή υλοποίηση ενός προγράμματος το οποίο εφαρμόζει Gaussian blur για τη θόλωση μιας εικόνας (όπως στην $2^η$ άσκηση της προηγούμενης εργασίας). Αυτή τη φορά όμως, πέρα από τις συναρτήσεις gaussian_blur_omp_loops και gaussian_blur_omp_tasks που κληθήκαμε να υλοποιήσουμε στην προηγούμενη εργασία (για βελτιστοποίηση του αλγορίθμου μέσω OpenMP loops και OpenMP tasks αντίστοιχα), κληθήκαμε να υλοποιήσουμε και μια νέα συνάρτηση, ονόματι gaussian_blur_opencl. Όλες οι συναρτήσεις λαμβάνουν ως ορίσματα μια εικόνα imgin, μια εικόνα imgout (στην οποία αποθηκεύεται η αλλοιωμένη εικόνα), και μια ακτίνα θόλωσης radius, η οποία όσο μεγαλύτερη είναι, τόσο μεγαλύτερο το θόλωμα της εικόνας. Στη συνέχεια, έπρεπε να γράψουμε ένα OpenCL kernel που υλοποιεί τον ίδιο αλγόριθμο με τη σειριακή υλοποίηση και να συγκρίνουμε χρόνους με τις 2 OpenMP υλοποιήσεις καθώς και τη σειριακή, εξάγοντας συμπεράσματα.

Το δεύτερο υποερώτημα της εργασίας αφορούσε την εύρεση του βέλτιστου μεγέθους ομάδας εργασίας (local work group size), έπειτα από πειραματισμούς με διάφορα μεγέθη.

Η αρχική σειριακή μέθοδος που μας δόθηκε ακολουθεί στην επόμενη σελίδα:

```
void gaussian_blur_serial(int radius, img_t *imgin, img_t *imgout)
{
	int i, j;
	int width = imgin->header.width, height = imgin->header.height;
	double row, col;
	double weightSum = 0.0, redSum = 0.0, greenSum = 0.0, blueSum = 0.0;

	for (i = 0; i < height; i++)
	{
		for (j = 0; j < width ; j++)
		{
			for (row = i-radius; row <= i + radius; row++)
			{
				for (col = j-radius; col <= j + radius; col++)
				{
					int x = clamp(col, 0, width-1);
					int y = clamp(row, 0, height-1);
					int tempPos = y * width + x;
					double square = (col-j)*(col-j)+(row-i)*(row-i);
					double sigma = radius*radius;
					double weight = exp(-square / (2*sigma)) /
(3.14*2*sigma);

					redSum += imgin->red[tempPos] * weight;
					greenSum += imgin->green[tempPos] * weight;
					blueSum += imgin->blue[tempPos] * weight;
					weightSum += weight;
				}
			}
			imgout->red[i*width+j] = round(redSum/weightSum);
			imgout->green[i*width+j] = round(greenSum/weightSum);
			imgout->blue[i*width+j] = round(blueSum/weightSum);

			redSum = 0;
			greenSum = 0;
			blueSum = 0;
			weightSum = 0;
		}
	}
}
```

Μέθοδος Παραλληλοποίησης

Το kernel που έφτιαξα πραγματοποιεί τους υπολογισμούς του σειριακού αλγορίθμου για κάθε pixel (δηλαδή τις εντολές που περικλείει το πρώτο εμφωλευμένο for της σειριακής υλοποίησης). Όπως γίνεται εύκολα αντιληπτό, αυτό μας οδηγεί στην χρήση global work size ίσο με width * height αν επιλέξουμε το kernel να είναι μονοδιάστατο, ή {width, height} αν επιλέξουμε να είναι δισδιάστατο. Επέλεξα να χρησιμοποιήσω δισδιάστατο, καθώς παρατήρησα σαφή βελτίωση στους μέσους χρόνους σε σχέση με τη μονοδιάστατη προσέγγιση (περισσότερα για αυτό στη συνέχεια).

Σχόλια σχετικά με τον κώδικα του kernel

Στο kernel, προφανώς δεν είναι δυνατή η χρήση των 2 img_t pointers (img_in, img_out) που χρησιμοποιούνται στην κλήση της gaussian_blur_opencl συνάρτησης. Έτσι, μιας και οι μόνες παράμετροι του img_t struct που επηρεάζονται από τους υπολογισμούς είναι οι 3 RGB πίνακες του struct, επέλεξα να έχω 7 παραμέτρους: 3 const unsigned char pointers για καθέναν εκ των RGB πινάκων της εικόνας img_in που είχε ως είσοδο η συνάρτηση gaussian_blur_opencl, 3 unsigned char pointers για καθέναν εκ των 3 πινάκων για red, green και blue που έχει η τελική εικόνα img_out, που πάλι είχε ως είσοδο η συνάρτηση gaussian_blur_opencl, και μια integer παράμετρο για την ακτίνα θόλωσης (radius), η οποία και αυτή δίνεται ως παράμετρος στην κλήση της συνάρτησης gaussian_blur_opencl.

Ύστερα, έπαιρνα το id του κάθε thread στην 1η διάσταση (μέσω της εντολής get_global_id(0)), και το id του κάθε thread στην 2η διάσταση (μέσω της εντολής get_global_id(1)). Επίσης, προκειμένου να πάρω τις μεταβλητές width και height για τον υπολογισμό του for loop, χρησιμοποίησα τις εντολές get_global_size(0) και get_global_size(1) αντίστοιχα.

Σχόλια σχετικά με τη δημιουργία του OpenCL context και την κλήση του kernel

Πριν σχολιάσω την προσέγγισή μου, ήθελα να αναφέρω ότι δυστυχώς δεν μπόρεσα να χρησιμοποιήσω τη CPU μου μέσω OpenCL για την προσέγγιση του προβλήματος, καθώς δεν κατάφερα να κάνω τον μεταγλωττιστή της OpenCL να την αναγνωρίσει.

Για το setup της OpenCL και για τη δημιουργία του kernel, στα αρχικά βήματα (μέχρι και τη δημιουργία του kernel) βασίστηκα στον συμπληρωματικό κώδικα που μας είχε δοθεί, ο οποίος χρησιμοποιούσε OpenCL για να βελτιστοποιήσει την πρόσθεση 2 πινάκων ακεραίων σε έναν τρίτο, και δεν έκανα καμία ουσιαστική αλλαγή. Οι αξιόλογες τροποποιήσεις ξεκίνησαν έπειτα από εκείνο το σημείο.

Αρχικά, γνωρίζοντας ότι όλοι οι red, green και blue unsigned char πίνακες των εικόνων ήταν μεγέθους width * height, οι 3 buffers που έκανα allocate προκειμένου να περάσω τους 3 πίνακες (σύμφωνα με την ιδέα που αναφέρθηκε νωρίτερα) ως παραμέτρους στο kernel ήταν της μορφής:

```
imginRedBuf = clCreateBuffer(context, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR, sizeof(unsigned char) * pixels, imgin->red, &err);
```

Όπου τα pixels υπολογίζονταν ως width * height. Σχετικά με τα cl_mem_flags στην άνω κλήση, έκρινα προφανής την χρήση CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR, προκειμένου το kernel να έχει τις σωστές τιμές. Οι δημιουργίες των 3 buffers (imginRedBuf, imginGreenBuf, imginBlueBuf) έγιναν με τον ίδιο τρόπο.

Σχετικά με τους buffers για τους 3 unsigned char πίνακες της imgout, χρησιμοποίησα την εξής εντολή:

```
imgoutRedBuf = clCreateBuffer(context, CL_MEM_WRITE_ONLY, sizeof(unsigned char) * pixels, NULL, &err);
```

Η λογική στην άνω εντολή ήταν πως, μιας και το kernel μας θα κάνει update κάθε τιμή και στους 3 πίνακες ανεξάρτητα από τις τρέχουσες τιμές τους, δεν υπάρχει λόγος τα flags να είναι CL_MEM_WRITE_ONLY | CL_MEM_COPY_HOST_PTR σε συνδυασμό με imgout->red (και

αντίστοιχα) ως host pointer, και θα είναι πιο απλό για την GPU απλά να γράφει στον εν λόγω pointer, χωρίς να ασχολείται με τις τρέχουσες τιμές του.

Το θέσιμο των kernel arguments έγινε πάλι παρόμοια με τον κώδικα του συμπληρωματικού κώδικα, μόνο που την παράμετρο radius την έθεσα ως clSetKernelArg(kernel, 6, sizeof(int), &radius);

καθώς προφανώς δεν χρειάζεται η χρήση buffer.

Ύστερα, χρονομέτρησα την εκτέλεση πάλι σύμφωνα με τον συμπληρωματικό κώδικα, χρησιμοποιώντας event για την εκτέλεση του kernel, και εξάγοντας πόση ώρα πήρε το event για την εκτέλεσή του. Τέλος, χρησιμοποίησα την εντολή clEnqueueReadBuffer για να περάσω τους 3 πίνακες της εικόνας imgout πίσω στο struct από την GPU, και απελευθέρωσα όλη την OpenCL μνήμη που είχα δεσμεύσει.

Εύρεση Κατάλληλου local work size

Προκειμένου να βρω το κατάλληλο μέγεθος work group size, χρησιμοποίησα έναν συνδυασμό εντολών. Αρχικά, αναζήτησα το μέγιστο δυνατό local work size που υποστηρίζει η GPU μου, μέσω της εντολής:

```
clGetDeviceInfo(device, CL_DEVICE_MAX_WORK_GROUP_SIZE, sizeof(size_t),
&maxWorkGroupSize, NULL);
```

όπου maxWorkGroupSize ήταν μια μεταβλητή τύπου size_t που χρησιμοποιούσα. Το αποτέλεσμα για τη συσκευή μου ήταν 1024.

Στη συνέχεια, αναζήτησα το CL_KERNEL_PREFERRED_WORK_GROUP_SIZE_MULTIPLE για το kernel που είχα δημιουργήσει, το οποίο μου επέστρεψε την τιμή 32. Επίσης, αναζήτησα το μέγιστο δυνατό CL_KERNEL_WORK_GROUP_SIZE για το kernel το οποίο είχα δημιουργήσει, το οποίο μου επέστρεψε την τιμή 256, την οποία τιμή θεώρησα άνω όριο για το γινόμενο localSizeX, localSizeY (του δισδιάστατου kernel) σε όποιους υπολογισμούς ακολούθησαν.

Όμως, υπήρχε ένας πάρα πολύ σημαντικός περιορισμός στους υπολογισμούς μου. Εάν δοκίμαζα να χρησιμοποιήσω local work size το οποίο δεν ήταν ακριβής διαιρέτης του {width, height} (όπως πχ {2, 128} προκειμένου να προσεγγίσω το CL_KERNEL_WORK_GROUP_SIZE καθώς και να πληρώ την απαίτηση του CL_KERNEL_PREFERRED_WORK_GROUP_SIZE_MULTIPLE), η OpenCL πετούσε error CL_INVALID_WORK_GROUP_SIZE. Έτσι, κατέληξα σε local group size το οποίο είναι ακριβής διαιρέτης κάθε διάστασης του global size αντίστοιχα, και το γινόμενο των localSizeX, localSizeY να ήταν <= του CL_DEVICE_MAX_WORK_GROUP_SIZE. Αυτό μου κατέστησε ιδιαίτερα δύσκολη την εύρεση του βέλτιστου local work size. Συγκεκριμένα, ό,τι συνδυασμός εν τέλει δοκίμασα, ανεξάρτητα από την είσοδο του προγράμματος, έπειτα από πολλές δοκιμές ήταν πάντα πάρα πολύ κοντά με την περίπτωση που είχα NULL ως local work size στην εντολή clEnqueueNDRangeKernel (αν όχι ολίγον χειρότερος). Παρόλα αυτά, αναζήτησα best practices σχετικά με OpenCL, μέσω του οποίου έμαθα ότι η επιλογή NULL αναφορικά με το local_work_size argument της εντολής clEnqueueNDRangeKernel ενδέχεται να δημιουργήσει θέματα ανάλογα με το σύστημα στο οποίο γίνονται οι δοκιμές, οπότε επέλεξα να χρησιμοποιήσω το βέλτιστο non-NULL συνδυασμό που βρήκα στους υπολογισμούς μου. Ακολουθούν τα αποτελέσματα.

Πειραματικά Αποτελέσματα - Μετρήσεις

Αρχικά, θέλω να επιδείξω διάφορους συνδυασμούς που δοκίμασα για local work size του kernel μου. Όλες οι δοκιμές ήταν σε συνδυασμούς που ήταν ακριβοί διαιρέτες του width και του height αντίστοιχα, αλλά παρέμεναν μικρότεροι του 256 (δηλαδή της τιμής του CL_KERNEL_WORK_GROUP_SIZE για την υλοποίησή μου). Καθώς ήταν αρκετά δύσκολη η εξαγωγή συμπερασμάτων με μικρή ακτίνα (radius), δοκίμασα ακτίνα r=40 με την εικόνα 1500x1500, προκειμένου οποιεσδήποτε διαφορά μεταξύ των προσεγγίσεων να γίνει όσο πιο εμφανής γίνεται. Παρόλα αυτά, οι διαφορές ήταν μηδαμινές, της τάξης του <1%. Παραθέτω τα αποτελέσματά μου. Οι χρόνοι είναι σε δευτερόλεπτα.

Δοκιμές για local work size στην εικόνα 1500x1500, με ακτίνα r=40

Local Dimensions	First Run	Second Run	Third Run	Fourth Run	Average
5x50	9.26	9.3	9.27	9.3	9.2825
10x25	9.31	9.35	9.36	9.32	9.335
2x125	9.32	9.55	9.29	9.42	9.395
1x250	9.54	9.28	9.47	9.28	9.3925
15x15	10.55	10.54	10.5	10.6	10.5475
10x20	10.15	10.11	10.18	10.17	10.1525
NULL	9.36	9.5	9.37	9.34	9.3925

Όπως παρατηρείται, όλοι οι συνδυασμοί που έρχονται όσο το δυνατόν κοντινότερα στο CL_KERNEL_WORK_GROUP_SIZE (ενώ παραμένουν ακριβοί διαιρέτες των αντίστοιχων διαστάσεων), είναι οι πιο αποδοτικοί. Οι διαστάσεις 15x15 και 10x20 με τις οποίες πειραματίστηκα, εμφανίζονται σαφώς χειρότερες, κάτι το οποίο θεωρώ αναμενόμενο καθώς απέχουν σημαντικά από το μέγιστο δυνατό πλήθος local work items ανά kernel (δηλαδή 256). Λόγω αυτών των δοκιμών, επέλεξα να συνεχίσω τους υπολογισμούς μου με kernel 5x50.

Συγκρίσεις

Για τη σύγκριση, έτρεξα 4 δοκιμές για σειριακή, OpenMP loops με 4 threads, OpenMP tasks με 4 threads, και OpenCL. Έκρινα σκόπιμο να μην δοκιμάσω πάλι σταδιακά με 1, 2, 3 και 4 OpenMP threads τη φορά όπως στην προηγούμενη εργασία, και να κρατήσω κατευθείαν τα βέλτιστα. Ακολουθούν τα αποτελέσματα (οι χρόνοι είναι σε δευτερόλεπτα).

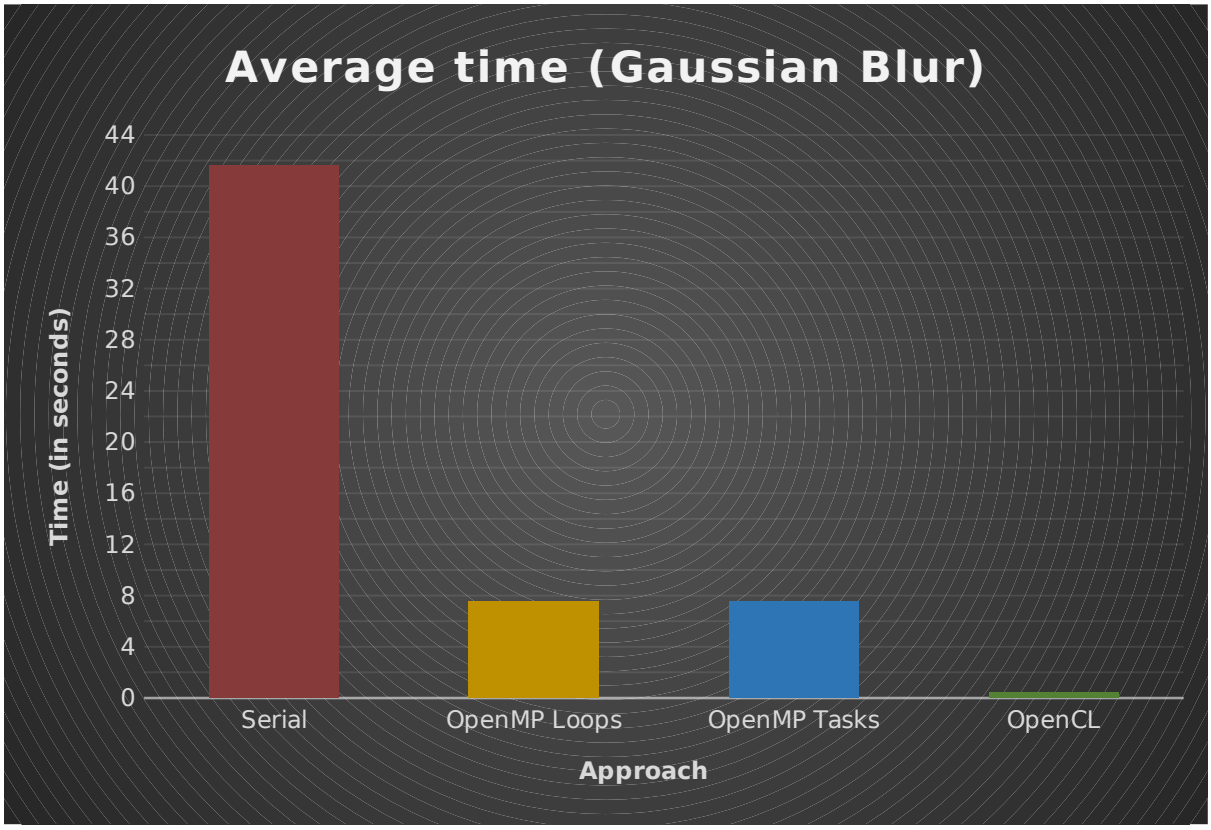
Approach	First Run	Second Run	Third Run	Fourth Run	Average
Serial	41.44	41.79	41.88	41.35	41.615
OpenMP Loops	10.74	10.8	10.96	10.71	10.8025
OpenMP Tasks	10.69	11.07	11.16	10.71	10.9075

OpenCL	0.45	0.47	0.43	0.45	0.45

Καθώς έμεινα έκπληκτος με τα αποτελέσματα, επέλεξα να κάνω έρευνα σχετικά με τα μέγιστα συνιστάμενα threads που υποστηρίζει το OpenMP, και έμαθα ότι ήταν ο αριθμός cores της CPU μου. Η CPU μου έχει 6 cores, οπότε ως επόμενο δοκίμασα την υλοποίηση OpenMP και με 6 threads. Όπως αναμενόταν, η διαφορά είναι και πάλι τεράστια μεταξύ της OpenCL υλοποίησης και της βέλτιστης δυνατής OpenMP υλοποίησης. (οι χρόνοι είναι πάλι σε δευτερόλεπτα)

Approach	**First Run**	**Second Run**	**Third Run**	**Fourth Run**	**Average**
Serial	41.44	41.79	41.88	41.35	41.615
OpenMP Loops - 6 Threads	7.52	7.5	7.47	7.57	7.515
OpenMP Tasks - 6 Threads	7.51	7.48	7.55	7.58	7.53
OpenCL	0.45	0.47	0.43	0.45	0.45

Από τον τελικό πίνακα, τον οποίο θα κρατήσω καθώς θεωρώ ότι αντικατοπτρίζει βέλτιστα τη διαφορά μεταξύ των υλοποιήσεων, προκύπτει το εξής barchart:



Σχόλια

Στο συγκεκριμένο πρόβλημα, μου έκανε τεράστια εντύπωση η διαφορά μεταξύ OpenMP και OpenCL. Περίμενα σαφή βελτίωση, αλλά ποτέ σε τέτοιο βαθμό. Ένα speedup που ρεαλιστικά θα περίμενα θα ήταν της τάξης του 10-20, αλλά ποτέ της τάξης του 100.

Νιώθω ότι το συγκεκριμένο πρόβλημα δέχεται και μια μικρή περαιτέρω βελτίωση από τη μεριά της OpenCL, καθώς με κατάλληλο padding, με μετατροπή του global work size στην κοντινότερη δυνατή του 2, θα ήταν δυνατή η πλήρης εκμετάλλευση της OpenCL (όπως μας δείχνει το CL_KERNEL_PREFERRED_WORK_GROUP_SIZE_MULTIPLE, που στην περίπτωσή μου ήταν 32).

Επίσης, θα ήθελα να είχα καταφέρει να συγκρίνω αποτελέσματα OpenCL GPU με OpenCL CPU, αλλά δυστυχώς δεν κατάφερα να κάνω την OpenCL να αναγνωρίσει την CPU μου.