CUDA

EVANGELOS CHASANIS

Το 2ο σετ ασκήσεων αφορά τον προγραμματισμό GPU χρησιμοποιώντας τεχνολογία CUDA. Αρχικά υλοποιείται το πρόγραμμα cuinfo.cu το οποίο συλλέγει πληροφορίες για τις συσκευές και στην συνέχεια για την 2η Άσκηση βελτιστοποιείται ο αλγόριθμος επεξεργασίας εικόνας Gaussian με την χρήση παράλληλης εκτέλεσης στις GPU.

Πληροφορίες μηχανήματος:

```
[ex24000@parallax ~]$ lscpu
Architecture:        x86_64
CPU op-mode(s):      32-bit, 64-bit
Byte Order:          Little Endian
CPU(s):              64
On-line CPU(s) list: 0-63
Thread(s) per core:  2
Core(s) per socket:  16
Socket(s):           2
NUMA node(s):        2
Vendor ID:           GenuineIntel
CPU family:          6
Model:               85
Model name:          Intel(R) Xeon(R) Gold 6130 CPU @ 2.10GHz
Stepping:            4
CPU MHz:             1000.377
BogoMIPS:            4200.00
Virtualization:      VT-x
L1d cache:           32K
L1i cache:           32K
L2 cache:            1024K
L3 cache:            22528K
NUMA node0 CPU(s):   0,2,4,6,8,10,12,14,16,18,20,22,24,26,28,30,32,34,36,38,40,42,44,46,48,50,52,54,56,58,60,62
NUMA node1 CPU(s):   1,3,5,7,9,11,13,15,17,19,21,23,25,27,29,31,33,35,37,39,41,43,45,47,49,51,53,55,57,59,61,63
Flags:               fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush dts acpi mmx fxsr s
se sse2 ss ht tm pbe syscall nx pdpe1gb rdtscp lm constant_tsc art arch_perfmon pebs bts rep_good nopl xtopology nonstop
_tsc cpuid aperfmperf pni pclmulqdq dtes64 monitor ds_cpl vmx smx est tm2 ssse3 sdbg fma cx16 xtpr pdcm pcid dca sse4_1
sse4_2 x2apic movbe popcnt tsc_deadline_timer aes xsave avx f16c rdrand lahf_lm abm 3dnowprefetch cpuid_fault epb cat_l3
 cdp_l3 invpcid_single pti intel_ppin ssbd mba ibrs ibpb stibp tpr_shadow vnmi flexpriority ept vpid ept_ad fsgsbase tsc
_adjust bmi1 hle avx2 smep bmi2 erms invpcid rtm cqm mpx rdt_a avx512f avx512dq rdseed adx smap clflushopt clwb intel_pt
```

Άσκηση 1

Το πρόβλημα

Ζητείται να δημιουργηθεί ένα πρόγραμμα που χρησιμοποιώντας το CUDA Runtime API, θα ανιχνεύει τις διαθέσιμες συσκευές CUDA σε ένα σύστημα και θα παρέχει το όνομα της συσκευής, την έκδοση της compute capability, την έκδοση του CUDA driver και runtime, το πλήθος των streaming multiprocessors (SMs), το συνολικό μέγεθος της καθολικής μνήμης, της μνήμης σταθερών και της κοινόχρηστης μνήμης ανά block. Επιπλέον, ζητείται η ανάπτυξη ενός τρόπου για τον υπολογισμό του συνολικού αριθμού πυρήνων (cores) μιας συσκευής NVIDIA GPU.

Μέθοδος επίλυσης

Χρησιμοποείται το struct cudDeviceProp dev_prop μέσα από το οποίο αντλείται η πληροφορίες που απαιτούνται. Αν δεν βρεθεί καμία CUDA συσκευή τότε τυπώνεται το αντίστοιχο μήνυμα και το πρόγραμμα τερματίζει. Αλλιώς για κάθε συσκευή τυπώνεται ότι απαιτείται από την εκφώνηση.

Επεξήγηση υπολογισμού CUDA Driver/Runtime version:

- Για να εκτυπωθεί ο κύριος αριθμός έκδοσης, διαιρώ το CUDART_VERSION με 1000. Για παράδειγμα, αν το CUDART_VERSION είναι 11000, η διαίρεση με 1000 δίνει 11, που αντιπροσωπεύει τον κύριο αριθμό έκδοσης.
- Για να εκτυπωθεί ο δευτερεύον αριθμός έκδοσης, χρησιμοποιείται ο τελεστής του υπολοίπου % για να εξαχθούν τα τελευταία δύο ψηφία μετά τη διαίρεση του CUDART_VERSION με 100. Για παράδειγμα, αν το CUDART_VERSION είναι

11060, η διαίρεση με 100 δίνει 110 με υπόλοιπο 60. Στη συνέχεια, διαιρούμε το υπόλοιπο με 10 για να πάρουμε τον δευτερεύοντα αριθμό έκδοσης.

Χρησιμοποίησα πληροφορίες που βρήκα από το Stack Overflow για να αναπτύξω τη συνάρτηση getSPcores, η οποία υπολογίζει το συνολικό αριθμό πυρήνων CUDA για μια συσκευή βάσει των πληροφοριών της συσκευής. Βασίστηκα στην έκδοση της CUDA Compute Capability, η οποία είναι 6.1 στην περίπτωσή μας, και γνωρίζοντας ότι η συσκευή είναι Tesla, γνωρίζουμε ότι ανήκει στην αρχιτεκτονική Pascal.

Με βάση αυτές τις πληροφορίες, η συνάρτηση υπολογίζει τον αριθμό των πυρήνων CUDA για την κάθε αρχιτεκτονική, όπως Fermi, Kepler, Maxwell, Pascal κ.λπ. Έτσι, με βάση την έκδοση 6.1 της CUDA Compute Capability και το γεγονός ότι η συσκευή είναι από την αρχιτεκτονική Pascal, χρησιμοποίησα την κατάλληλη περίπτωση στη συνάρτηση getSPcores για να υπολογίσω τον συνολικό αριθμό των πυρήνων CUDA.

Link για το Stack Overflow: https://stackoverflow.com/questions/32530604/how-can-i-get-number-of-cores-in-cuda-device

Compile με : nvcc -o cuinfo cuinfo.cu -Xcompiler -fopenmp

Τα αποτελέσματα του προγράμματος είναι τα παρακάτω:

```
Last login: Fri May  3 19:59:08 2024 from 87.203.127.167
[ex24000@parallax ~]$ ./cuinfo
Found 1 CUDA device(s):
Device Name: Tesla P40
CUDA Compute Capability: 6.1
CUDA Driver/Runtime Version: 11.6
Number of SMs: 30
Total Global Memory: 24032444416 bytes
Total Constant Memory: 65536 bytes
Shared Memory Per Block: 49152 bytes
Total CUDA Cores: 3840
[ex24000@parallax ~]$
```

Επομένως υπολογίστηκαν 3840 cores σε μια NVIDIA GPU(Tesla P40).

Το οποίο είναι σωστό όπως φαίνεται παρακάτω:

Link : https://www.techpowerup.com/gpu-specs/tesla-p40.c2878

Άσκηση 2

Το πρόβλημα

Το πρόβλημα που πρέπει να λυθεί είναι η επιτάχυνση της διεργασίας θόλωσης εικόνας μέσω της χρήσης της GPU με χρήση OpenMP. Αυτό σημαίνει να εκτελεστεί η λειτουργία Gaussian Blur παράλληλα σε πολλαπλά νήματα στη GPU, εκμεταλλευόμενοι τα πλεονεκτήματα της παραλληλίας που προσφέρει η συσκευή CUDA. Αυτό πρέπει να συμβεί ακολουθώντας τις απαιτήσεις της εκφώνησης.

Μέθοδος επίλυσης

Αρχικά ορίστηκε η συνάρτηση **gaussian_blur_omp_device** η οποία με τις κατάλληλες οδηγίες offloading, είναι υπεύθυνη για την θόλωση στην GPU. Απαίτηση της άσκησης ήταν να χρησιμοποιηθεί κατάλληλος συνδυασμός των εντολών target, teams, distribute, parallel for collapse(2), η οποία υλοποιήθηκε. Ακόμα χρησιμοποιείται map(from:color_out[0:height*width]) ώστε μόλις υπολογιστούν να επιστρέψουν στον host, και map(to:color_in[0:height*width] και height, width, radius) για να περνάν οι τιμές που απαιτούνται στην GPU για να εκτελεστεί πάνω σε αυτά οι θόλωση και να υπολογιστούν τα out.

Μεταφορά αρχείων και έλεγχος αποτελεσμάτων

Για την μεταφορά του αρχείου και της εικόνας από το μηχάνημα μου στο Parallax χρησιμοποίησα την εντολή:

scp -P 2229 gaussian-blur.c ex24000@gatepc73.cs.uoi.gr:~/

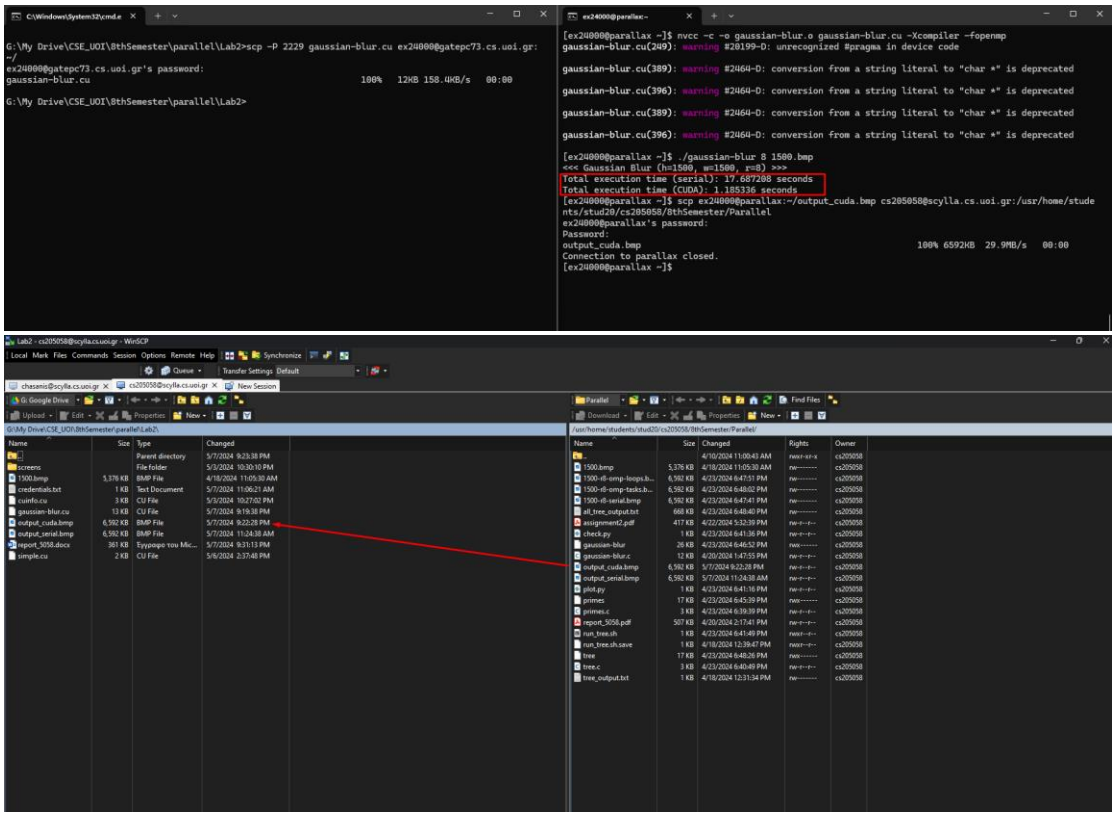
scp -P 2229 1500.bmp ex24000@gatepc73.cs.uoi.gr:~/

Για την μεταφορά από το Parallax στον ακαδημαϊκό λογαριασμό μου στην Scylla(δεν βρήκα τρόπο να τα μεταφέρω κατευθείαν στο μηχάνημά μου):

scp ex24000@parallax:~/<αρχείο> cs205058@scylla.cs.uoi.gr:/usr/home/students/stud20/cs205058/8thSemester/Parallel

Μετά χρησιμοποίησα το WinSCP για να τα μεταφέρω στον υπολογιστή μου και να ελέγξω τα αποτελέσματα της θόλωσης.

Η διαδικασία φαίνεται παρακάτω:



Σύμφωνα με τις απιτήσεις της εκφώνησης το numTeams θα πρέπει να είναι >= 30 και ο αριθμός νημάτων >=32 και πολλαπλάσιος του οπότε οι πιθανόι συνδυασμοί teams με threads είναι:

Teams	Threads
30	128
40	96
60	64
120	32

Compile με: clang -fopenmp -fopenmp-targets=nvptx64-nvidia-cuda -I /usr/local/cuda/include -L /usr/local/cuda/lib64 gaussian-blur.c -o gaussian-blur -lcudart -lm

Συνδυασμένες Οδηγίες:

```
void gaussian_blur_omp_device(int radius, img_t
*imgin, img_t *imgout) {
    int width = imgin->header.width, height =
imgin->header.height;

    unsigned char *red_out = imgout->red;
    unsigned char *green_out = imgout->green;
    unsigned char *blue_out = imgout->blue;

    unsigned char *red_in = imgin->red;
    unsigned char *green_in = imgin->green;
    unsigned char *blue_in = imgin->blue;

    int numThreadsPerTeam = 128;
    int numTeams = 30;

    #pragma omp target teams distribute parallel
for collapse(2) num_teams(numTeams) \
    thread_limit(numThreadsPerTeam) \
    map(to: red_in[:width*height],
green_in[:width*height], blue_in[:width*height],
height, width, radius) \
    map(from: red_out[:width*height],
green_out[:width*height], blue_out[:width*height])
    for (int i = 0; i < height; i++) {
        for (int j = 0; j < width; j++) {
            double weightSum = 0.0, redSum = 0.0,
greenSum = 0.0, blueSum = 0.0;

            for (int row = i - radius; row <= i +
radius; row++) {
```

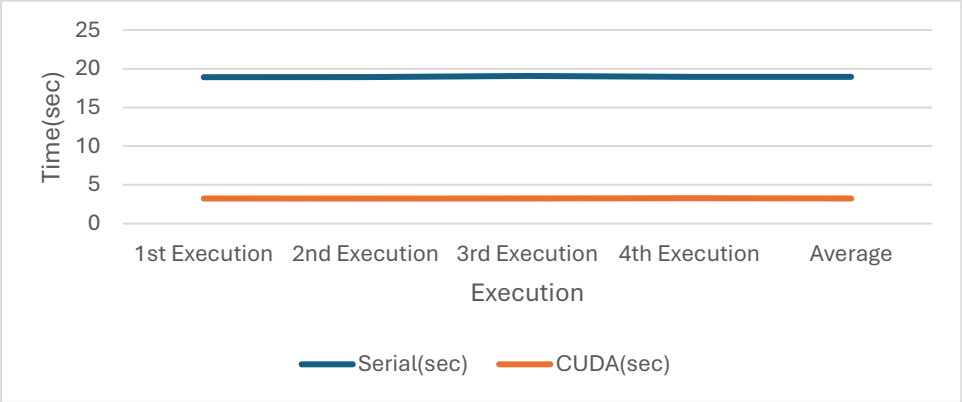
```
                for (int col = j - radius; col <=
j + radius; col++) {
                    int x = clamp(col, 0, width -
1);
                    int y = clamp(row, 0, height -
1);
                    int tempPos = y * width + x;
                    double square = (col - j) *
(col - j) + (row - i)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

                    redSum += red_in[tempPos] *
weight;
                    greenSum += green_in[tempPos]
* weight;
                    blueSum += blue_in[tempPos] *
weight;
                    weightSum += weight;
                }
            }

            red_out[i * width + j] = (unsigned
char)(redSum / weightSum + 0.5);
            green_out[i * width + j] = (unsigned
char)(greenSum / weightSum + 0.5);
            blue_out[i * width + j] = (unsigned
char)(blueSum / weightSum + 0.5);
        }
    }
}
```

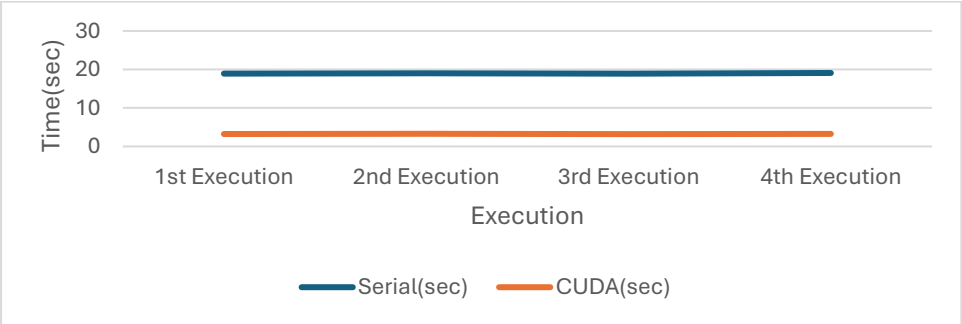
Με 30 ομάδες και 128 νήματα ανά ομάδα:

	Serial(sec)	CUDA(sec)
1st Execution	18.9209	3.25235
2nd Execution	18.91997	3.228374
3rd Execution	19.06571	3.253551
4th Execution	18.95839	3.27147
Average	**18.96624**	**3.251436**



Με 40 ομάδες και 96 νήματα ανά ομάδα:

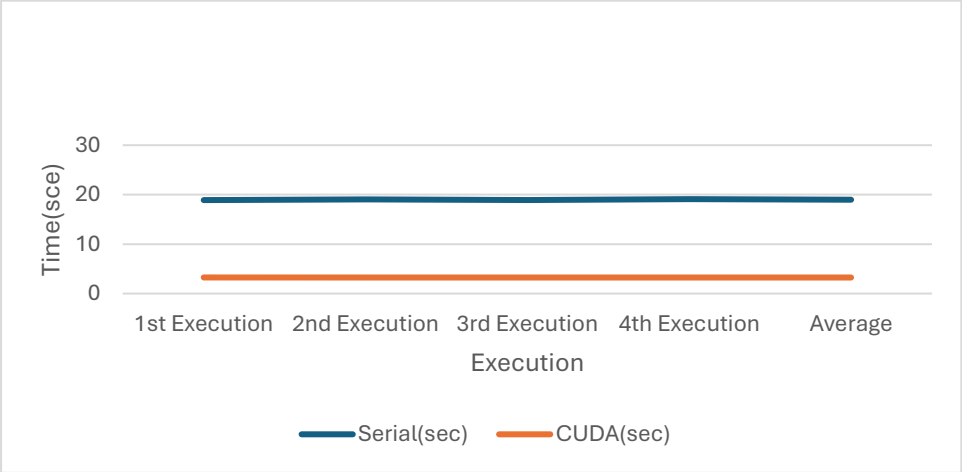
	Serial(sec)	CUDA(sec)
1st Execution	18.91677	3.237775
2nd Execution	18.97251	3.259477
3rd Execution	18.9054	3.246534
4th Execution	19.08657	3.263474
Average	**18.97031**	**3.251815**



Με 60 ομάδες και 64 νήματα ανά ομάδα:

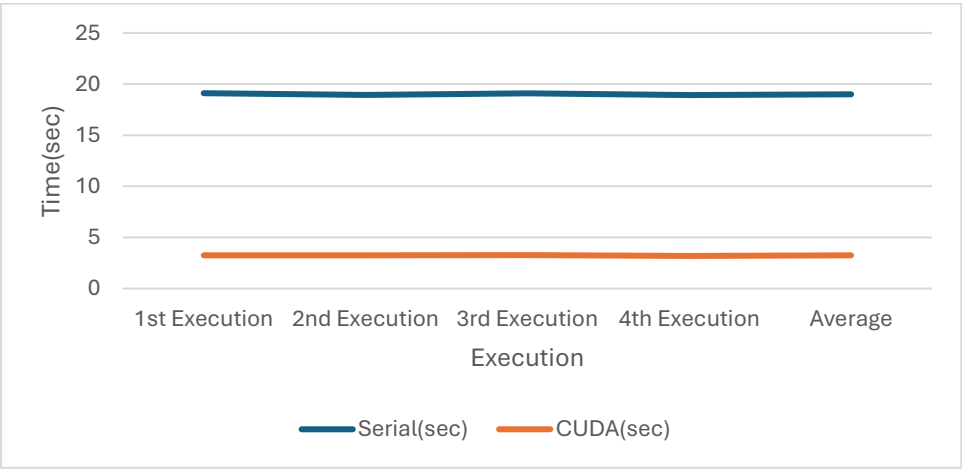
	Serial(sec)	CUDA(sec)

1st Execution	18.90313	3.257027
2nd Execution	19.0458	3.25602
3rd Execution	18.93933	3.248268
4th Execution	19.09491	3.248528
Average	**18.99579**	**3.252461**

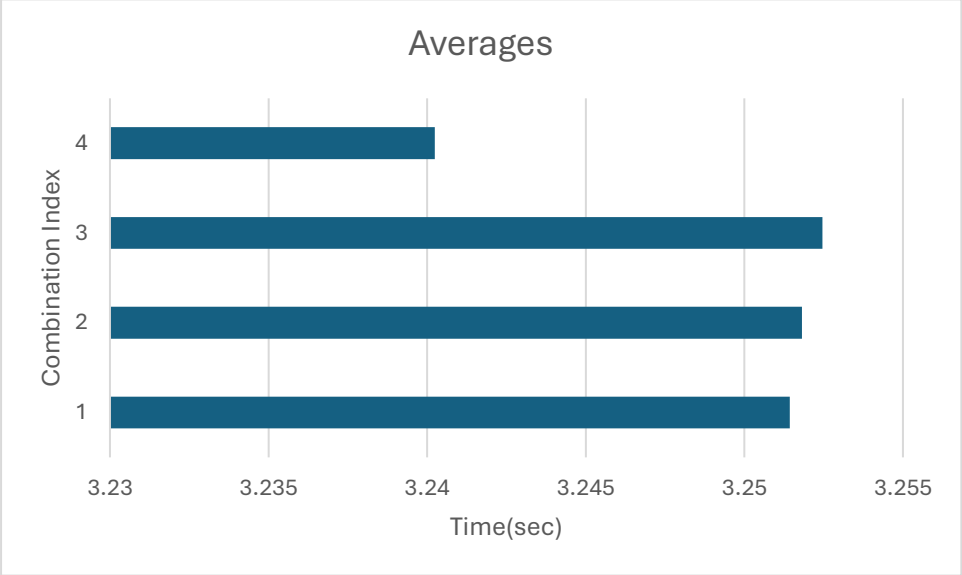


Με 120 ομάδες και 32 νήματα ανά ομάδα:

	Serial(sec)	CUDA(sec)
1st Execution	19.10755	3.246777
2nd Execution	18.9429	3.254651
3rd Execution	19.08392	3.269519
4th Execution	18.92399	3.19001
Average	**19.01459**	**3.240239**



Φαίνεται να υπάρχει ελάχιστη(σχεδόν μηδενική) διαφορά στον χρόνο ανεξαρτήτως συνδυασμού ομάδας και αριθμού νημάτων ανά ομάδα, όπως δείχνει και η γραφική παράσταση παρακάτω:



Αυτό προκύπτει από το γεγονός ότι οι διάφοροι συνδυασμοί του αριθμού των ομάδων και του αριθμού νημάτων ανά ομάδα οδηγούν σε έναν συνολικό αριθμό νημάτων που αντιστοιχεί στον συνολικό αριθμό πυρήνων της συσκευής. Επομένως, οι πόροι της συσκευής εκμεταλλεύονται πλήρως, διασφαλίζοντας ότι ο ρυθμός λειτουργίας των πυρήνων είναι σταθερός ανεξαρτήτως του συνδυασμού του αριθμού των ομάδων και του αριθμού νημάτων ανά ομάδα

Εμφωλευμένες Οδηγίες:

```
void gaussian_blur_omp_device(int radius, img_t
*imgin, img_t *imgout) {
    int width = imgin->header.width, height =
imgin->header.height;

    unsigned char *red_out = imgout->red;
    unsigned char *green_out = imgout->green;
    unsigned char *blue_out = imgout->blue;

    unsigned char *red_in = imgin->red;
    unsigned char *green_in = imgin->green;
    unsigned char *blue_in = imgin->blue;

    int numThreadsPerTeam = 128;
    int numTeams = 30;

    #pragma omp target map(to:
red_in[:width*height], green_in[:width*height],
blue_in[:width*height], height, width, radius)
map(from: red_out[:width*height],
green_out[:width*height], blue_out[:width*height])
    {
        #pragma omp teams
thread_limit(numThreadsPerTeam)
num_teams(numTeams)
        {
            #pragma omp distribute
            for (int i = 0; i < height; i++) {
                #pragma omp parallel for
collapse(2)
```

```
                for (int j = 0; j < width; j++) {
                    double weightSum = 0.0, redSum
= 0.0, greenSum = 0.0, blueSum = 0.0;

                    for (int row = i - radius; row
<= i + radius; row++) {
                        for (int 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 +=
red_in[tempPos] * weight;
                            greenSum +=
green_in[tempPos] * weight;
                            blueSum +=
blue_in[tempPos] * weight;
                            weightSum += weight;
                        }
                    }

                    red_out[i * width + j] =
(unsigned char)(redSum / weightSum + 0.5);
```

```
                    green_out[i * width + j] =
(unsigned char)(greenSum / weightSum + 0.5);
                    blue_out[i * width + j] =
(unsigned char)(blueSum / weightSum + 0.5);
                }
            }
        }
    }
}
```

Δεν υπάρχει κάποια διαφορά στον χρόνο εκτέλεσης του προγράμματος μεταξύ συνδυασμένων και εμφωλευμένων οδηγιών.

Η παρατήρηση μικρής ή ανύπαρκτης διαφοράς στον χρόνο εκτέλεσης μεταξύ των εσωτερικών και των ενσωματωμένων οδηγιών του OpenMP είναι μια συχνή εμφάνιση, ειδικά όταν πρόκειται για σχετικά μικρά φορτία εργασίας ή όταν το κόστος της παραλληλοποίησης είναι χαμηλό σε σύγκριση με τον χρόνο εκτέλεσης των μεμονωμένων εργασιών. Αυτό το φαινόμενο οφείλεται σε αρκετούς παράγοντες.

Καταρχάς, για μικρά φορτία εργασίας, το overhead της δημιουργίας και του συγχρονισμού νημάτων σε εμφωλευμένη παραλληλία ενδέχεται να μην επηρεάζει σημαντικά το συνολικό χρόνο εκτέλεσης. Επίσης, εάν οι εργασίες μέσα στην παράλληλη περιοχή είναι fine-grained και ισορροπημένες, το overhead της διαχείρισης της εμφωλευμένης παραλληλίας ενδέχεται να είναι αμελητέο σε σύγκριση με το υπολογιστικό φορτίο.

Δεύτερον, σύγχρονοι μεταγλωττιστές είναι συχνά σε θέση να βελτιστοποιήσουν τον παραγόμενο κώδικα για να ελαχιστοποιήσουν το κόστος της εμφωλευμένης παραλληλίας.

Τρίτον, η επίδραση της εμφωλευμένης παραλληλίας μπορεί να ποικίλλει ανάλογα με την αρχιτεκτονική του υλικού και τις δυνατότητες του συστήματος εκτέλεσης.

Με βάση αυτούς τους παράγοντες, η παρατήρηση ελάχιστης ή καθόλου διαφοράς στον χρόνο εκτέλεσης μεταξύ των εσωτερικών και των ενσωματωμένων οδηγιών του OpenMP σε συγκεκριμένα σενάρια είναι πράγματι φυσιολογική. Αυτό υπογραμμίζει την αποτελεσματικότητα του συστήματος εκτέλεσης OpenMP και την αποτελεσματικότητα των βελτιστοποιήσεων του μεταγλωττιστή.