Σχεδιασμός Ενσωματωμένων Συστημάτων

Φιλιππόπουλος Ορφέας el18082
Παπαρρηγόπουλος Θοδωρής el18040

1η Εργαστηριακή Άσκηση

1.2 Loop Optimizations & Design Space Exploration (folder 1/)

Note: Για δική σας ευκολία σε περίπτωση που θέλετε να τα τρέξετε, παρέχουμε το run_1_2_all.py script το οποίο τρέχει όλους τους κώδικες για το 1.2 μέρος και επίσης παρέχουμε το read_results_and_box_plot.py script το οποίο παίρνει τα αποτελέσματα (από τα κατάλληλα αρχεία που έγραψε το προηγούμενο script) και δημιουργεί τα κατάλληλα plots το φάκελο plots. Άμα δεν τρέξετε το run_1_2_all.py script τότε στο φάκελο plots φαίνονται τα τελικά αποτελέσματα που τρέξαμε στο υπολογιστή μας (οπότε και να ξανατρέξετε το read_results_and_box_plot.py script θα παράξει τα ίδια αποτελέσματα).

1.
Με την εντολή uname -a βρήκαμε την έκδοση του λειτουργικού και του kernel:

Έκδοση Kernel: Linux version 5.15.0-53-generic
Έκδοση Λειτουργικού: Ubuntu 9.4.0-1ubuntu1~20.04.1

Με την ενολή lscpu βρήκαμε τον αριθμό των φυσικών πυρήνων, των λογικών πυρήνων (threads), τη συχνότητα του ρολογιού του επεξεργαστή και τέλος τα μεγέθη και την ιεραρχία των μνημών μέχρι και τη RAM.

8 logical cores (threads)
4 physical cores
2.1GHz

L1d – 128KiB (cache line 64B)
L1i – 128KiB (cache line 64B)
L2 – 1MiB (cache line 64B)
L3 – 6MiB (cache line 64B)
RAM – 16.2GiB

τα cache lines τα βρήκαμε κάνοντας cat κάποια συγκεκριμένα αρχεία.

2. Αρχικά προσθέσαμε στη main του phods.c το παρακάτω κομμάτι κώδικα (πρακτικά, απλά χρησιμοποιήσαμε την gettimeofday 2 φορές, μία πριν την εκτέλεση της συνάρτησης και μία μετά

```
struct timeval start, end;

gettimeofday(&start, NULL);

phods_motion_estimation(current,previous,motion_vectors_x,motion_vectors_y);

gettimeofday(&end, NULL);

float time = (end.tv_sec - start.tv_sec) + (end.tv_usec - start.tv_usec)*0.000001;

printf("%lf\n", time);
```

και τέλος τυπώσαμε το χρόνο με ακρίβεια usec):

Προκειμένου να μπορούμε να χρονομετρήσουμε την εκτέλεση του συνάρτησης phods_motion_estimation.

Στη συνέχεια τρέξαμε το **phods.c** 10 φορές με το **exec_first.py script** (το οποίο κάνει και compile το πρόγραμμα με τις κατάλληλες παραμέτρους) και υπολογίσαμε ελάχιστο, μέγιστο και μέσο όρο εκτέλεσης. Τα αποτελέσματα που προέκυψαν είναι τα εξής:

max time	0.008777
mean time	0.007463
min time	0.006941

3.
Στο ερώτημα αυτό καλούμαστε να πραγματοποιήσουμε τους κατάλληλους μετασχηματισμούς του κώδικα προκειμένου να μειωθεί ο χρόνος εκτέλεσης. Για το λόγο αυτό θα βασιστούμε στη σχεδιαστική αρχή data reuse, θα αξιοποιήσουμε το spatial locality των δεδομένων και τέλος θα προσπαθήσουμε να μειώσουμε όσο το δυνατόν χρονοβόρους υπολογισμούς όπως τους πολλαπλασιασμούς/διαιρέσεις. Στο αρχείο phods_opt.c μπορείτε να βρείτε την υλοποίηση μας.

Αρχικά παρατηρώντας το κώδικα διαπιστώσαμε ότι πολλοί πολλαπλασιασμοί/διαιρέσεις καθώς και προσπελάσεις μνήμης (χρονοβόρες διαδικασίες) επαναλαβάνονται. Για το λόγο αυτό κάναμε κατάλληλους μετασχηματισμούς προκειμένου να τις/τους υπολογίζουμε μία φορά.

Το πρώτο πράγμα το οποίο πραγματοποιήσαμε ήταν το loop merging για τη for loop:

```
for(i=-S; i<S+1; i+=S)
```

και εφαρμόσαμε ακριβώς το ίδιο για το loop:

```
for(k=0; k<B; k++)
```

και τέλος:

```
for(l=0; l<B; l++)
```

γλυτώνοντας έτσι με μία πρώτη ματιά αρκετές εντολές σε assembly.

Στη συνέχεια υπολογίσαμε τα N/B, M/B και 255*B*B μία φορά και τα αποθηκεύσαμε, πριν ξεκινήσει το for loop (για τη μεταβλητή x) σε μεταβλητές, τις x_bound, y_bound και b_b, αντίστοιχα, μιας και παραμένουν σταθερά κατά τη διάρκεια της εκτέλεσης.

```
69      int x_bound = N/B;
70      int y_bound = M/B;
```

Επιπροσθέτως παρατηρήσαμε (αφού έχουμε κάνει το loop merging) ότι εσωτερικά των loops για τις μεταβλητές k και l στο εσωτερικό τους πραγματοποιούνται υπολογισμοί οι οποίοι περιλαμβάνουν μεταβλητές της εξωτερικής λούπας x και y καθώς και σταθερές. Πιο συγκεκριμένα αναφερόμαστε στους πολλαπλασιασμούς B*x και B*y και για το λόγο αυτό "μετακινήσαμε" τους πολλαπλασιασμούς αυτούς στη for loop για το x και για το y αντίστοιχα στις μεταβλητές b_x και b_y αντίστοιχα.

Ακόμα, παρατηρήσαμε ότι η πράξη B*x + vectors_x[x][y] και η πράξη B*y + vectors[x][y] εκτελούνται σε κάθε επανάληψη του l ενώ αφορούν μεταβλητές ανεξάρτητες απο τη λούπα αυτή. Για το λόγο αυτό μετακινήσαμε τις πράξης αυτές στη while στις μεταβλητές b_x_vec_x και b_y_vec_y αντίστοιχα.

Μία πολύ σημαντική παρατήρηση είναι το γεγονός ότι η αρχικοποίηση των vectors_x, vectors_y πινάκων δεν χρειάζεται να γίνεται ανεξάρτητα, πάνω από τη for loop για το x αλλά μπορεί να γίνεται παράλληλα με την εκτέλεση του αλγορίθμου μέσα στη for loop για το y. Κατά αυτό το τρόπο, σε συνδυασμό με τη παραπάνω αλλαγή, οι vectors_x[x][y] και vectors_y[x][y] όταν αρχικοποιούνται έρχονται στη cache και όταν χρησιμοποιούνται λίγο παρακάτω απο την εντολή B*x + vectors_x[x][y] και την B*y + vectors[x][y] αντίστοιχα έχουμε cache hit!.

Επιπροσθέτως διάσπαρτα στο κώδικα έχουμε "μπερδέψει" τις εντολές προκειμένου να αποφύγουμε RAW Hazards.

Οι παραπάνω αλλαγές φαίνονται στο παρακάτω κομμάτι κώδικα:

```
/*For all blocks in the current frame*/
for (x = 0; x < x_bound; x++)
{

  b_x = B * x;

  for (y = 0; y < y_bound; y++)
  {

    S = 4;

    vectors_x[x][y] = 0;
    vectors_y[x][y] = 0;

    b_y = B * y;

    while (S > 0)
    {

      min1 = b_b;
      min2 = b_b;

      b_vec_x = b_x + vectors_x[x][y];
      b_vec_y = b_y + vectors_y[x][y];

      /*For all candidate blocks in X dimension*/
      for (i = -S; i < S + 1; i += S)
```

Μέχρι στιγμής ο χρόνος ετκέλεσης (μαζί με το loop merging) έχει μειωθεί αρκετά. Ωστόσο θα

κάνουμε μία τελευταία αλλαγή και θα παρουσιάσουμε τα αποτελέσματα.

Όπως πριν, παρατηρούμε πως πολλές πράξης εσωτερικά του l for loop αφορούν τις παραπάνω μεταβλητές που τις ορίσαμε εσωτερικά του x και του y for loop αλλά και το k και για το λόγο αυτό κάνουμε ακριβώς την ίδια διαδικασία με πριν ορίζοντας τις κατάλληλες μεταβλητές εσωτερικά του k for loop. Σε επίπεδο κώδικα κάναμε το εξής:

```
/*For all pixels in the block*/
for(k=0; k<B; k++)
{

    b_x_k = b_x + k;
    b_x_vec_x = b_vec_x + k;
    b_x_vec_x_i = b_x_vec_x + i;
    check_x_i = (b_x_vec_x_i) < 0 || (b_x_vec_x_i) > N_1;
    check_x = (b_x_vec_x) < 0 || (b_x_vec_x) > N_1;

    for(l=0; l<B; l++)
```

Όσον αφορά περισσότερο το data reuse θα μπορούσαμε πχ να έχουμε έναν πίνακα μεγέθους BxB (μικρότερου μεγέθους μνήμη που μπορεί να τοποθετηθεί πλησιέστερα στον επεξεργαστή) στον οποίο θα “cacharame” BxB block από τον current πίνακα. Ωστόσο δεν πιστεύουμε πως έχει νόημα για τις συγκεκριμένες διαστάσεις του πίνακα καθώς “χωράει” ολόκληρος στην L1 cache μας (θα υπάρχουν μερικά conflicts μεταξύ του current και του previous πίνακα στην L1 cache αλλά πιστεύουμε πως δεν θα μας επηρέαζαν ιδιαίτερα). Μάλιστα το δοκιμάσαμε κίολας (αρχείο **phods_opt_data_reuse.c**) και ο χρόνος εκτέλεσης χειροτέρεψε (το overhead της αντιγραφής είναι μεγαλύτερο από τα πλεονεκτήματα της συγκεκριμένης υλοποίησης στις συγκεκριμένες διαστάσεις).

Αφού πραγματοποιήσαμε όλους τους παραπάνω μετασχηματισμούς (το αρχείο που το περιέχει είναι το **phods_opt.c**), τρέξαμε το κώδικα μας 10 φορές με το **exec_first_opt.py script** (το οποίο κάνει και compile του προγράμματος). Οι χρόνοι που προέκυψαν είναι οι εξής:

max time	0.001980
mean time	0.001866
min time	0.001837

Με βάση τα παραπάνω παρατηρούμε πολύ καλύτερες επιδόσεις σε σχέση με πριν. Για την ακρίβεια ο χρόνος μειώθηκε κατά 73.4375% (ή αλλιώς σχεδόν υποτετραπλασσιάστηκε).

4.
Στον κώδικα που προέκυψε απο το προηγούμενο ερώτημα εφαρμόζουμε Design Exploration pattern αναφορικά με την εύρεση του βέλτιστου μεγέθους μπλοκ B.

Οι κοινοί διαιρέτες του 144 και του 176 είναι:

1, 2, 4, 8, 16

Προκειμένου να μπορούμε να γράψουμε script το οποίο τρέχει αυτόματα όλες τις περιπτώσεις για όλα τα B, αλλάξαμε λίγο τον κώδικα του phods_opt.c προκειμένου να δέχεται το B σαν παράμετρο.

Στη συνέχεια τρέξαμε το **exec_opt_find_b.py** script (το οποιο κάνει και compile το κώδικα) το

οποίο αποθηκεύει στο **results_phods_opt_best_b.txt** τα αποτελέσματα.

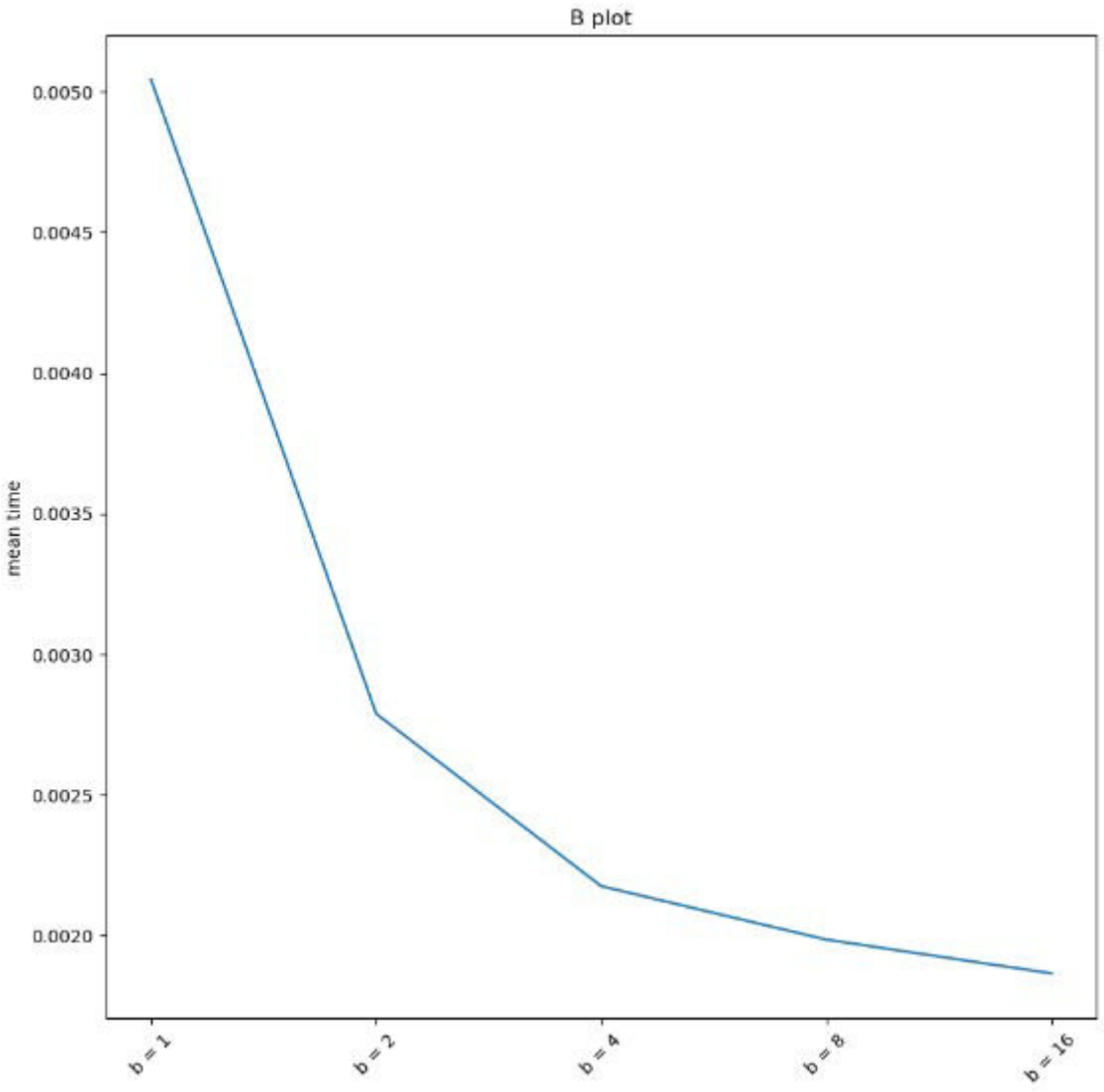
Με βάση αυτά λοιπόν παρατηρούμε ότι τη καλύτερη επίδοση τη πετυχαίνει όταν το B = 16 με χρόνους (το βέλτιστο το βρήκαμε με χρήση του **find_best_b.py script**):

max time	0.001939
mean time	0.001866
min time	0.001831

Σχεδόν ίδια με πριν καθώς και πριν τρέξαμε το κώδικα με B = 16.

Η τιμή αυτή πιστεύουμε ότι σχετίζεται με το cache line της L1 cache μας. Όταν κάνουμε access ένα στοιχείο από πχ τον πίνακα current στην επόμενη loopa του l θα κάνουμε access το επόμενο του, και πάει λέγοντας με το επόμενου του επόμενου κλπ. Για αυτό όταν έχουμε B = 16 πετυχαίνουμε αισθητά τον καλύτερο χρόνο γιατί τότε φέρνουμε σε chunks των 16 (cache line = 64B → 16*sizeof(int) = 64B) στοιχείων τα στοιχεία του πίνακα current ενώ εμείς αξιοποιούμε τα 16, με αποτέλεσμα να έχουμε πιθανόν 15 συνεχόμενα cache hits. Στις υπόλοιπες περιπτώσεις θα είχαμε πιθανότατα (το σύστημά μας δεν είναι απομωνομένο και για αυτό δεν μπορούμε να πούμε τίποτα με σιγουριά) 7, 3, 1, 0 συνεχόμενα cache hits αντίστοιχα.

Συμπληρωματικά δείχνουμε γραφικά και τους χρόνους για τα υπόλοιπα B (δεν κάναμε box plot γιατί το scale των χρόνων είναι αρκετά διαφορετικό):



Παρατηρεί λοιπόν κανείς ότι για B = 1 έχουμε πολύ μεγάλους χρόνους (λογικό καθώς σε κάθε λούπα για ένα block εμείς απλά χρησιμοποιούμε 1 στοιχείο ενώ μας έρχονται άλλα 63), για B = 2 έχουμε και πάλι αυξημένους χρόνους για τον ίδιο λόγο με B = 1 (αξιοποιούμε απλά τα 2 από τα 64 στοιχεία) και τέλος το ίδιο για B = 4, 8. Με λίγα λόγια το B = 16 μεγιστοποιεί την αξιοποίηση των 64 σοιχείων που μας έρχονται στη cache (16 από τα 64). Το γράφημα μας επιβεβαιώνει και την αρχική αιτιολόγησή μας.

5.
Στον κώδικα που προέκυψε απο το προηγούμενο ερώτημα εφαρμόζουμε Design Exploration pattern αναφορικά με την εύρεση του βέλτιστου μεγέθους μπλοκ, για τα ορθογώνια μπλοκ διαστάσεων Bx, By, όπου Bx διαιρέτης του N και By διαιρέτης του M.

Οι πιθανές διαστάσεις για τα Bx είναι:

1, 2, 3, 4, 6, 8, 9, 12, 16, 18, 24, 36, 48, 72, 144

και για τα By είναι:

1, 2, 4, 8, 11, 16, 22, 44, 88, 176

Θα εφαρμόσουμε εξαντλητική εξερεύνηση των καλύτερων τιμών χρησιμοποιώντας το script exec.py (κάνει και το compile του προγράμματος). Επίσης τροποποίησαμε λίγο το κώδικα για την αποφυγή κάποιον warnings και τον αποθηκεύσαμε στο phods_opt_bx_by.c.

Τα αποτελέσματα βρίσκονται στο αρχείο **results_phods_opt_bx_by.txt**.

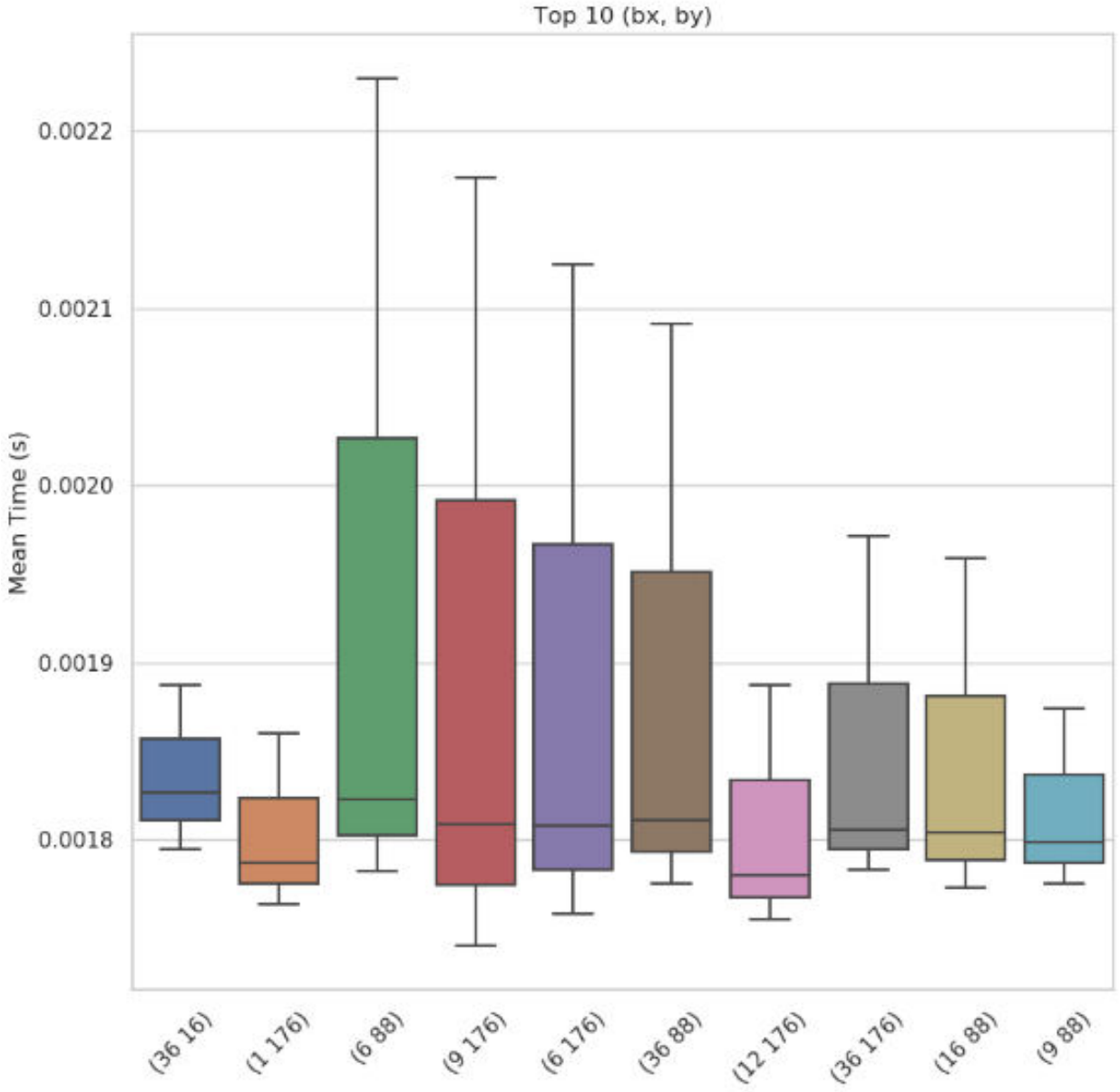
Μέσω του script **find_best_bx_by.py** βρήκαμε το καλύτερο συνδυασμό Bx By ο οποίος είναι:

bx: 12
by:176

με mean time 0.001768.

Παρατηρούμε αισθητή διαφορά σε σχέση με πριν (όπου Bx = 16 και By = 16)!

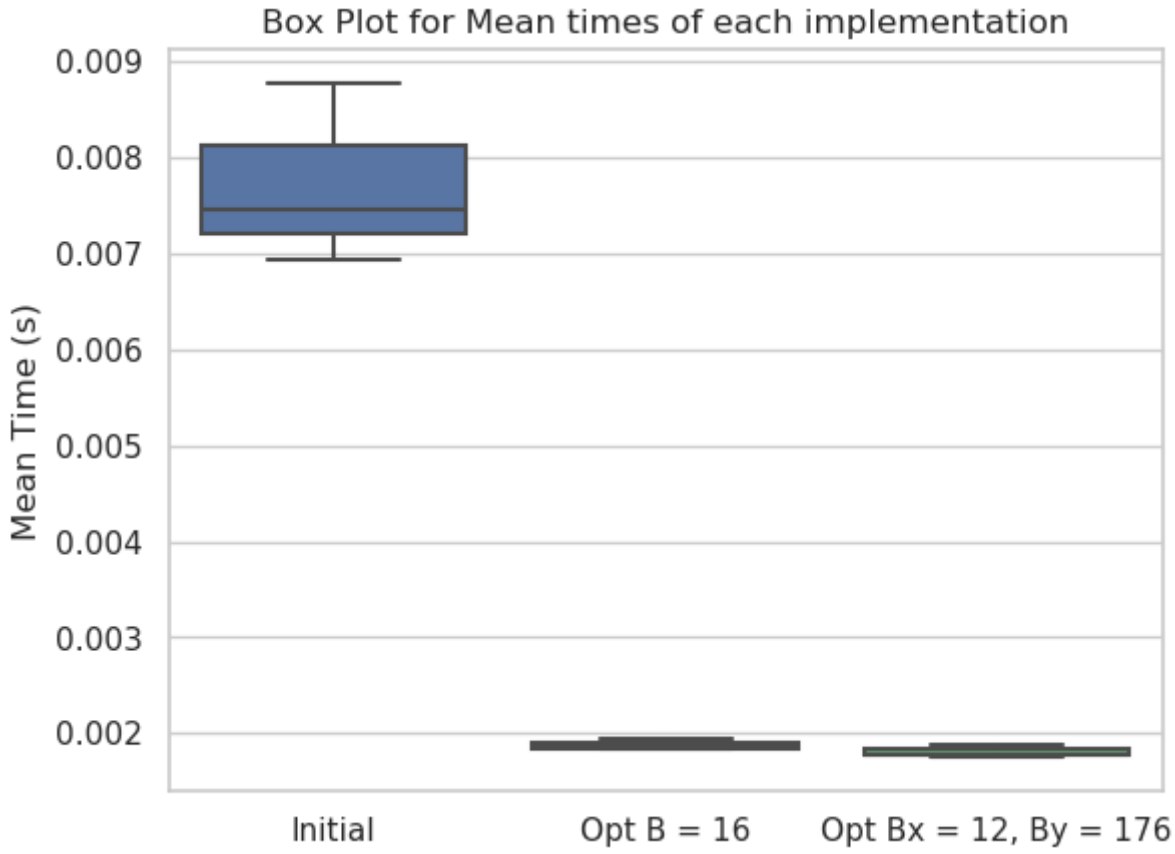
Επίσης για λόγους πληρότητας plot-αρουμε τα 10 πρώτα καλύτερα configurations (πάλι με το script **find_best_bx_by.py)**.



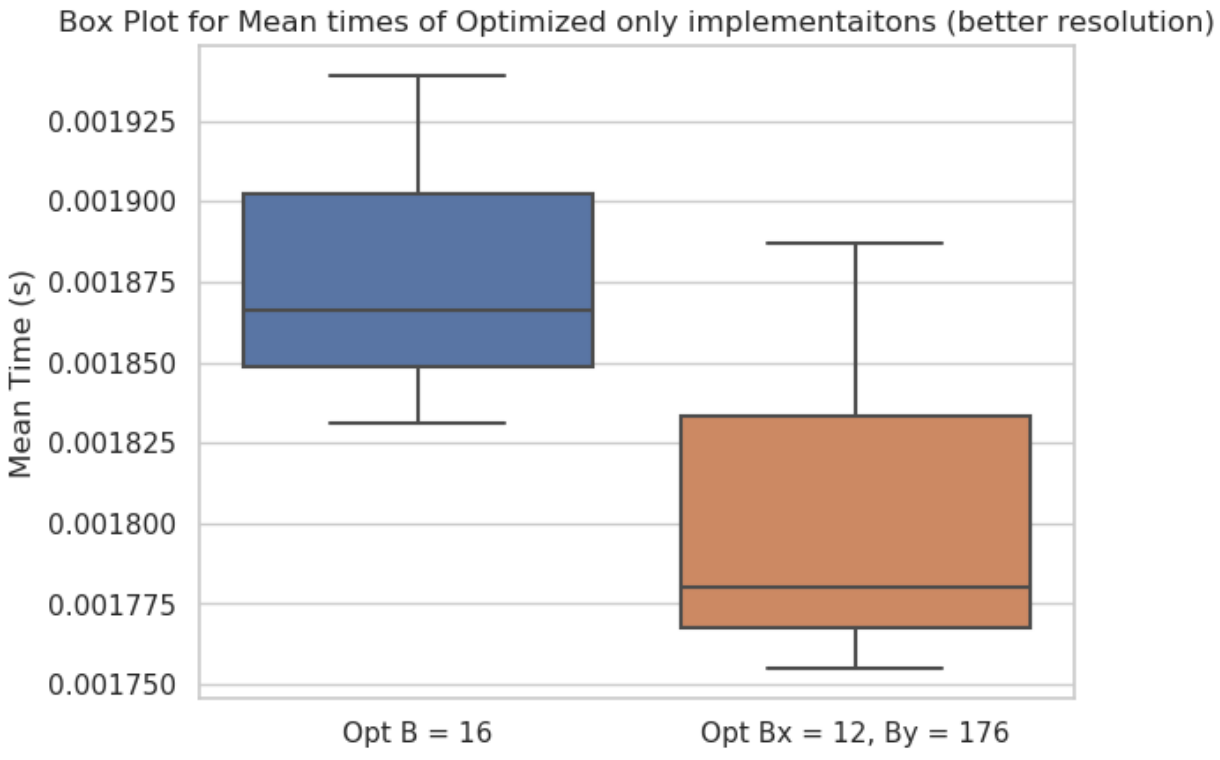
Ψάχνουμε τώρα μια ευρισιτική για να προσεγγίσουμε την εύρεση του καλύτερου παραλληλόγραμμου. Δοκιμάζουμε να κρατήσουμε σταθερό το By και να αυξάνουμε το Bx. Αν δούμε ότι υπάρχει αύξηση του χρόνου εκτέλεσης σε 3 συνεχόμενες αυξήσεις σταματάμε την αύξηση και κρατάμε την προηγούμενη τιμή. Στη συνέχεια μειώνουμε το Bx και επαναλαμβάνουμε αυτή την μεθοδολογία. Τελικά, από αυτή την διαδικασία κρατάμε το Bx που μας έφερε την καλύτερη απόδοση όσο κρατούσαμε σταθερό το By και το ίδιο για το By.
Η μεθοδολογία αυτή ακολουθεί τη λογική των greedy αλγορίθμων και μπορεί να κολλήσει σε τοπικό ελάχιστο και να μην βρει πότε το ολικό ελάχιστο.

6.
Μέσω του script **read_results_and_box_plot.py** διαβάζουμε από τα αρχεία τις κατάλληλες τιμές για κάθε υλοποίηση και κάνουμε τα παρακάνω boxplots.

Παρακάτω παρουσιάζουμε το boxplot που προέκυψε (δεν έχουμε το ερώτημα 3 γιατί είναι ακριβώς το ίδιο με το ερώτημα 4, δηλαδή και τα 2 με B = 16):



Ωστόσο επειδή έχουν αρκετά διαφορετικό scale στους χρόνους δεν φαίνεται τόσο καλά και για αυτό το λόγο προσθέτουμε και το boxplot για τις Opt B = 16 και
Opt Bx = 12, By = 176 περιπτώσεις:



Παρατηρούμε λοιπόν από τα παραπάνω ότι οι αρχικές βελτιστοποιήσεις που πραγματοποιήσαμε όντως βελτίωσαν σε αρκετά μεγάλο ποσοστό το χρόνο εκτέλεσης (με B = 16, που ήταν το βέλτιστο εκ των **τετραγωνικών** block). Ωστόσο και το εξαντλητικό Design Space Exploration που εφαρμόσαμε μας έδωσε μη τετραγωνικό block, με Bx = 12 και By = 176 τα οποία κατάφεραν να μειώσουν περαιτέρω το χρόνο εκτέλεσης (περίπου 7%).

1.3 Αυτοματοποιημένη βελτιστοποίηση κώδικα (folder 2/)

Note: Για δική σας ευκολία σε περίπτωση που θέλετε να τα τρέξετε, παρέχουμε το run_1_3_all.py script το οποίο τρέχει όλους τους κώδικες για το 1.3 μέρος και επίσης παρέχουμε το read_results_and_box_plot.py script το οποίο παίρνει τα αποτελέσματα (από τα κατάλληλα αρχεία που έγραψε το προηγούμενο script) και δημιουργεί τα κατάλληλα plots το φάκελο plots. Άμα δεν τρέξετε το run_1_3_all.py script τότε στο φάκελο plots φαίνονται τα τελικά αποτελέσματα που τρέξαμε στο υπολογιστή μας (οπότε και να ξανατρέξετε το read_results_and_box_plot.py script θα παράξει τα ίδια αποτελέσματα).

Στο μέρος αυτό θα χρησιμοποιήσουμε το εργαλείο Orio για αυτόματη βελτιστοποίηση κώδικα (και στη προκειμένη περίπτωση για αυτόματη εύρεση του loop unrolling factor). Ποιο συγκεκριμένα έχουμε ένα πρόγραμμα (tables.c) το οποίο εκτελεί 100.000.000 επαναλήψεις και πρέπει να βρούμε το loop unrolling factor που μας δίνει τα καλύτερα αποτελέσματα.

1.

Αρχικά, αφού εγκαταστήσαμε επιτυχώς το Orio στον υπολογιστή μας, τρέξαμε 10 φορές το tables.c (αφού προσθέσαμε κώδικα ο οποίος χρονομετρά το πρόγραμμα) και υπολογίσαμε τα mean, max, min times του προγράμματος. Πιο συγκεκριμένα έχουμε:

Max time	0.573897
Mean time	0.489608
Min time	0.443157

Παρατηρούμε λοιπόν ότι **χωρίς loop unrolling** πετυχαίνουμε κατά μέσο όρο 0.489608s χρόνο.

2.

Προσαρμόσαμε το αρχείο tables_orio.c προκειμένου να βρούμε τα UFs χρησιμοποιώντας κάθε φορά διαφορετικό τρόπο (Exhaustive, Randomsearch, Simplex) και τρέχοντας την εντολή sudo orcc tables_orio.c βρήκαμε τα UFs της κάθε περίπτωσης:

Exhaustive	UF = 9
Randomsearch	UF = 17
Simplex	UF = 22

Αξιοσημείωτο είναι ότι επειδή ο πίνακας είναι μεγέθους 100.000.000 * sizeof(double) = 800.000.000 και είναι πολύ μεγάλος, χρησιμοποιήσαμε και το -mcmodel=large flag στον gcc,

δηλαδή:

```
def build {
        arg build_command = 'gcc -O0 -mcmodel=large';
}
```

Τα configurations της κάθε υλοποίησης μπορούν να βρεθούν στα:

/2/simulations_source_codes/exhaustive/tables_orio.c
/2/simulations_source_codes/random/tables_orio.c
/2/simulations_source_codes/simplex/tables_orio.c

Ωστόσο για το exhaustive χρησιμοποιήσαμε τα ήδη υπάρχοντα, για το random απλά χρησιμοποιήσαμε απλά arg algorithm = 'Randomsearch'; configuration (τα υπόλοιπα ίδια με το exhaustive πέρα από το ότι βάλαμε 15 runs) και τέλος για το simplex χρησιμοποιήσαμε αυτό που μας δίνονται.

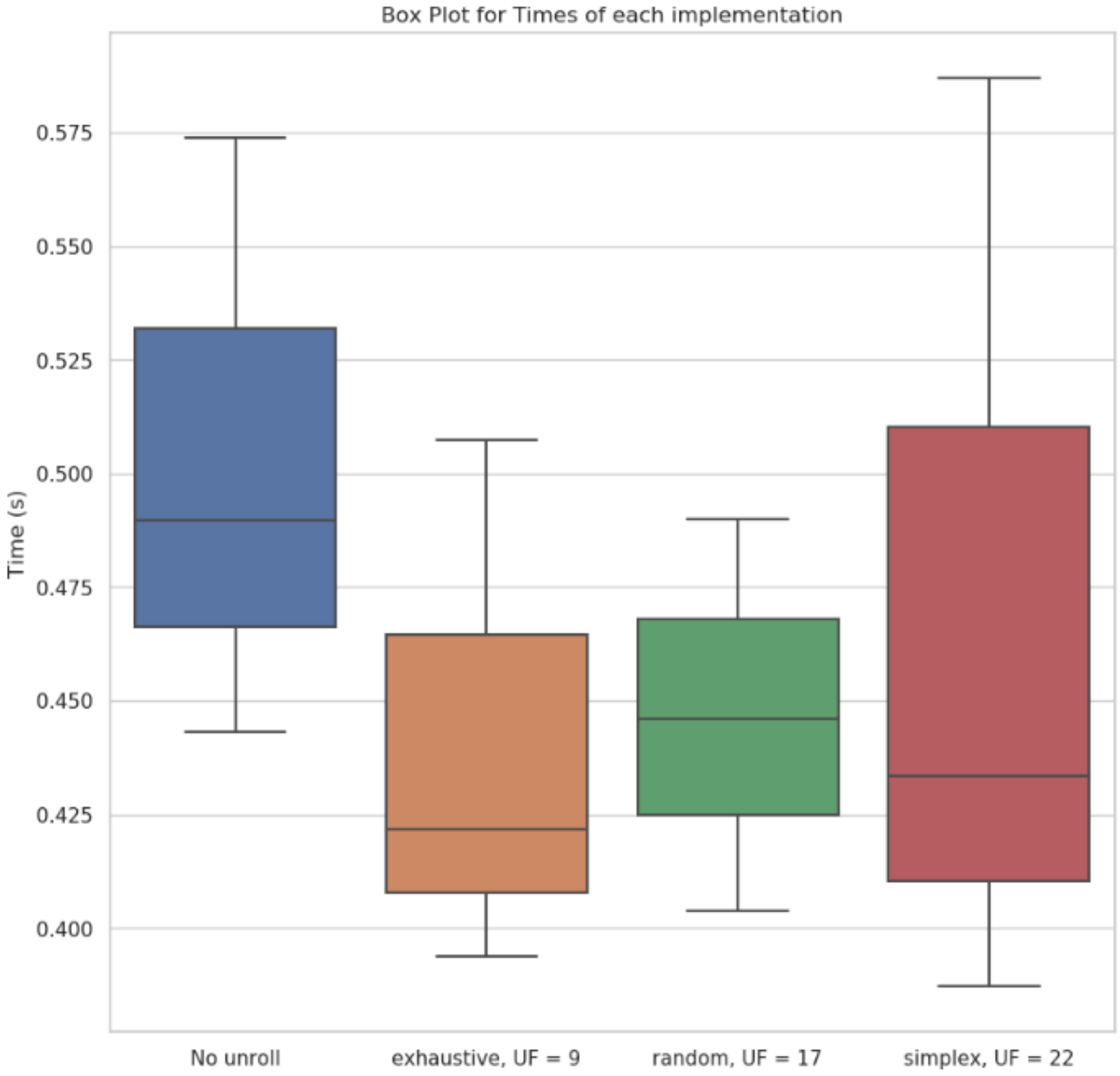
Τα αποτελέσματα όπως παρατηρούμε είνα διαφορετικά μεταξύ τους. Αυτό είναι λογικό καθώς ο Exhaustive θα "ψάξει" όλα τα πιθανά UFs και θα κρατήσει το βέλτιστο, ο RandomSearch θα κάνει τυχαία 15 δοκιμές (runs) και θα κρατήσει τη βέλτιστη ενώ τέλος ο Simplex προσπαθεί με ευριστικό τρόπο να βρει το βέλτιστο UF και δεν είναι απαραίτητο ότι θα το βρει. Ωστόσο το πλεονέκτημα των 2 τελευταίων (παρόλο που δεν βρίσκουν απαραίτητα το βέλτιστο UF) είναι ότι είναι αρκετα πιο γρήγοροι και σε μεγάλα, εκθετικά, Search Spaces προτιμούνται για αυτόν ακριβώς το λόγο.

Το script πουχρησιμοποιήθηκε για τα παραπάνω είναι το run_simulation.py στο φάκελο /scripts

3.

Τέλος με το script run_1_3_all.py τρέξαμε όλα τα παραπάνω προγράμματα (κάναμε copy paste τα αποτελέσματα του orio σε κάθε περίπτωση στα files
tables_exhaustive.c, tables_random.c και tables_simplex.c) και τα αντίστοιχα αποτελέσματα αποθηκεύτηκαν στο φάκελο /restuls στα αντίστοιχα files.

Τέλος με το script read_results_and_box_plot.py κάναμε τα ζητούμενα box plos.
Τα αποτελέσματα που προέκυψαν είναι:



Όπως περιμέναμε ο exhaustive τρόπος πετυχαίνει τον λιγότερο χρόνο ενώ ο random και simplex πετυχαίνουν παρόμοια αποτελέσματα (με λίγο καλυτερα αυτά του simplex ο οποίος με ευριστικό τρόπο προσπαθεί να βρει το βέλτιστο UF). Και οι 3 τρόποι πετυχαίνουν αρκετά καλύτερα αποτελέσματα από την αρχική υλοποίηση.