Σχεδιασμός Ενσωματωμένων Συστημάτων

9ο Εξάμηνο ΗΜΜΥ

1η ΕΡΓΑΣΤΗΡΙΑΚΗ ΑΣΚΗΣΗ

ΒΕΛΤΙΣΤΟΠΟΙΗΣΗ ΑΛΓΟΡΙΘΜΩΝ ΓΙΑ ΧΑΜΗΛΗ ΚΑΤΑΝΑΛΩΣΗ ΕΝΕΡΓΕΙΑΣ ΚΑΙ ΥΨΗΛΗ ΑΠΟΔΟΣΗ

Φοίβος-Ευστράτιος Καλεμκερής (03116010)

Νικολέτα-Μαρκέλα Ηλιακοπούλου (03116111)

Ζητούμενο 1ο: Loop Optimizations & Design Space Exploration

Σημείωση: Το σύνολο του κώδικα που χρησιμοποιήθηκε για το 1ο μέρος της άσκησης βρίσκεται εντός του φακέλου 1. Τα python scripts που χρησιμοποιούνται βρίσκονται εντός του φακέλου helpers, οι γραφικές που αυτά παράγουν εντός του plots και οι έξοδοι των εκτελέσεων εντός του results. Ο φάκελος helpers περιλαμβάνει τα δύο script για τον υπολογισμό των διαιρετών. Ο τρόπος εκτέλεσης κάθε script περιγράφεται στο README.md που περιλαμβάνεται στον φάκελο 1.

1. Έκδοση λειτουργικού: 20.04-Ubuntu, Έκδοση πυρήνα: 5.4.0-54-generic (**uname -a**)
   Ιεραρχία μνήμης (**lscpu**): L1d cache: 128KB
   L1i cache: 128KB
   L2 cache: 1MB
   L3 cache: 32MB
   RAM
   Αριθμός πυρήνων (**lscpu**): 4
   Συχνότητα ρολογιού πυρήνων (**lscpu**): 2303.998 MHz

2. Θα χρειαστούμε τη συνάρτηση *gettimeofday* από τη βιβλιοθήκη <sys/time.h>, την οποία θα καλέσουμε πριν και μετά την εκτέλεση της συνάρτησης *phods_motion_estimation*, προκειμένου να βρούμε τις χρονικές στιγμές έναρξης και τερματισμού της συνάρτησης, από τη διαφορά των οποίων βρίσκουμε το χρόνο εκτέλεσής της.
   Μεταγλωττίζουμε με make στο φάκελο της άσκησης (το Makefile περιλαμβάνει την εντολή *gcc -O0 -o phods_initial phods_initial.c,* καθώς και οδηγίες για τους κώδικες των επόμενων μερών της άσκησης. Θα τρέχουμε απλά make για το compilation οποιουδήποτε από τα προγράμματα). Εκτελούμε τον κώδικα 10 φορές ανακατευθύνοντας την έξοδο στο αρχείο phods.out με τη χρήση του script run.sh (*./run.sh*), το οποίο εκτελεί το run_phods.sh στο οποίο πραγματοποιούνται οι εν λόγω

επαναλήψεις. Στη συνέχεια, εκτελούμε το python script min_max_avg.py (*python3 ../min_max_avg.py results/out/phods_initial.out*), το οποίο διαβάζει το προηγούμενο αρχείο εξόδου υπολογίζοντας και εκτυπώνοντας τα επιθυμητά αποτελέσματα:

Stats	Initial Time (sec)
Minimum	0.005109
Median	0.005312
Average	0.005841
Maximum	0.010261

Σε αυτό, όπως και σε όλα τα υπόλοιπα μέρη της άσκησης λαμβάνουμε εκτός από τα min, max και average και το median, δηλαδή το διάμεσο των αποτελεσμάτων. Αυτός παρέχει μια καλύτερη απεικόνιση του συνόλου των αποτελεσμάτων από τον αριθμητικό μέσο, αφού δεν επηρεάζεται από τυχόν ακραίες τιμές (outliers), που όπως θα δούμε θα προκύψουν πολλές φορές στα διάφορα μέρη της άσκησης, γεγονός λογικό, μιας και το σύστημα στο οποίο εκτελούμε τα προγράμματα δεν είναι απομονωμένο. Απεναντίας, τρέχουν παράλληλα κι άλλες διεργασίες, οι οποίες μεταβάλλουν το περιεχόμενο της μνήμης, ενώ ευθύνονται και για καθυστερήσεις λόγω του context switching.

3. Θέλουμε, στη συνέχεια να μεταβάλουμε τον αρχικό κώδικα με σκοπό τη βελτιστοποίησή του ως προς το χρόνο εκτέλεσης. Για το λόγο αυτό θα βασιστούμε στη σχεδιαστική αρχή του data reuse, θα προβούμε σε μετασχηματισμούς βρόχων με σκοπό την αξιοποίηση του spatial locality και θα φροντίσουμε ώστε, χρονοβόροι υπολογισμοί, να πραγματοποιούνται κατά το δυνατόν μόνο μία φορά. Το αποτέλεσμα αυτής της βελτιστοποίησης μπορεί να βρεθεί στο αρχείο phods_opt.c

   Αρχικά, παρατηρώντας τον κώδικα διαπιστώσαμε ότι πολλοί υπολογισμοί όπως για παράδειγμα αριθμητικές πράξεις και προσβάσεις στη μνήμη κατά το διάβασμα πινάκων, επαναλαμβάνονται αυτούσιοι πολλές φορές κατά την εκτέλεση του κώδικα. Ιδιαίτερα οι περιπτώσεις υπολογισμών που “κοστίζουν” πολύ, όπως διαιρέσεις και πολλαπλασιασμοί, κάνουν τον κώδικα μη αποδοτικό. Ειδικότερα, υπολογίσαμε τα N/B, M/B και 255*B*B μία φορά και τα αποθηκεύσαμε σε global μεταβλητές, τις NdiaB, MdiaB και bigInt, αντίστοιχα, μιας και παραμένουν σταθερά κατά τη διάρκεια της εκτέλεσης.

   Έπειτα, εστιάσαμε την προσοχή μας στο εσωτερικό της συνάρτησης phods_motion_estimation. Παρατηρήσαμε, συγκεκριμένα, ότι πολλές πράξεις που περιλαμβάνουν μεταβλητές της εξωτερικής λούπας x και y, καθώς και σταθερές, επαναλαμβάνονται πολλές φορές αυτούσια στις εσωτερικές λούπες k, l. Για το λόγο αυτό, τις αποθηκεύσαμε σε προσωρινές μεταβλητές (που διατηρούν την τιμή τους για κάθε y επανάληψη) έξω από τις εσωτερικές επαναλήψεις. Χαρακτηριστικά, αποθηκεύσαμε τις τιμές του B*x και B*y που προηγουμένως υπολογίζονταν κάθε φορά

σε κάθε έλεγχο (if) του l βρόχου, στις μεταβλητές Bx, By, αντίστοιχα και στη συνέχεια χρησιμοποιήσαμε αυτές για να υπολογίσουμε τα παρακάτω:

- vector_x = vectors_x[x][y] + Bx;
- vector_y = vectors_y[x][y] + By;

Ύστερα, συνειδητοποιήσαμε ότι η αρχικοποίηση ολόκληρων των πινάκων vectors_x[x][y] και vectors_y[x][y] πριν την είσοδο στον κύριο βρόχο είναι ανούσια, μιας και σε κάθε x-y επανάληψη χρειαζόμαστε μία μόνο θέση του καθενός, αυτή που αντιστοιχεί στην τρέχουσα επανάληψη. Επιλέξαμε, επομένως, να αρχικοποιούμε εντός της y-loop μόνο τη θέση του πίνακα που πρόκειται στη συνέχεια να χρειαστούμε (πριν πραγματοποιήσουμε την εκχώρηση που δείξαμε προηγουμένως):

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

Η πλήρης εκμετάλλευση του data reuse οδήγησε στη μείωση του μέσου χρόνου εκτέλεσης κατά 40% σε σχέση με το αρχικό πρόγραμμα.

Ακολούθως, παρατηρώντας τη μεγάλη ομοιότητα των επαναλήψεων για τη X και Y διάσταση, αποφασίσαμε να εφαρμόσουμε loop merge προκειμένου να μειώσουμε περαιτέρω το χρόνο εκτέλεσης. Συγκεκριμένα, μεταφέραμε το περιεχόμενο της l-loop για την Y διάσταση μέσα στην l-loop της X, μεταφέροντας φυσικά και τις απαραίτητες αρχικοποιήσεις και τους μετασχηματισμούς που είχαμε πραγματοποιήσει στο προηγούμενο βήμα. Η αλλαγή αυτή είχε πράγματι τα επιθυμητά αποτελέσματα, μειώνοντας τον χρόνο κατά 50% σε σχέση με το προηγούμενο βήμα.

Τέλος, αποφασίσαμε να δοκιμάσουμε loop unroll στη while loop για το S, μιας και αυτή έχει σταθερό αριθμό επαναλήψεων, ίσο με 3 (S = 4, 2, 1) για κάθε επανάληψη y. Ωστόσο, ο μετασχηματισμός αυτός επέφερε μικρή αύξηση του χρόνου εκτέλεσης και, συνεπώς, απορρίφθηκε. Αντί αυτού, δοκιμάσαμε να αντικαταστήσουμε το while loop με ένα for loop (for(m=0; m<3; m++)), αξιοποιώντας την προηγούμενη παρατήρηση περί σταθερού αριθμού επαναλήψεων. Πράγματι, η αλλαγή αυτή προκάλεσε μικρή ελάττωση του χρόνου εκτέλεσης και, έτσι, επιλέξαμε να την κρατήσουμε στον τελικό κώδικα.

Μετά το σύνολο των παραπάνω μετασχηματισμών και αφού μεταγλωττίσουμε το πρόγραμμά μας (make), τρέχουμε- όπως και πριν- 10 φορές τον κώδικα που προκύπτει με το script run.sh (*./run.sh opt*) και, ύστερα, εκτελώντας όπως και προηγουμένως το script min_max_avg.py (*python3 ../min_max_avg.py results/out/phods_opt.out*), λαμβάνουμε την ελάχιστη, τη μέγιστη τιμή του χρόνου εκτέλεσης, το μέσο όρο και τον διάμεσο, όπως αυτοί φαίνονται στον επόμενο πίνακα και επιβεβαιώνουν τη μείωση που περιγράψαμε.

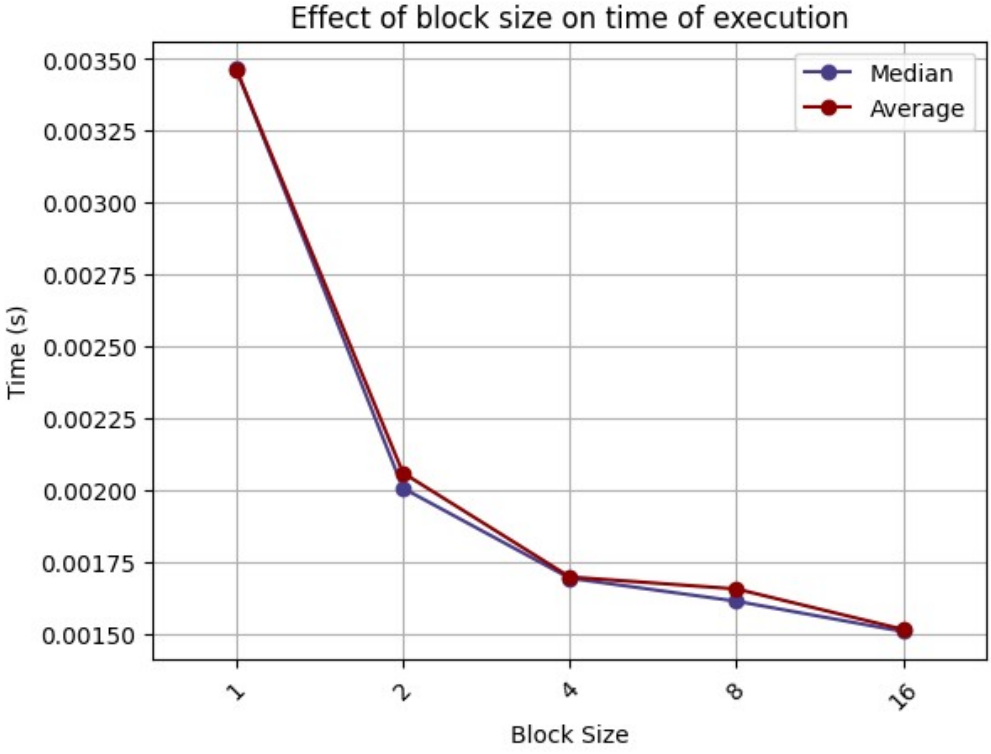
Stats	Optimized Time (sec)
Minimum	0.001493
Median	0.001685
Average	0.001774
Maximum	0.002586

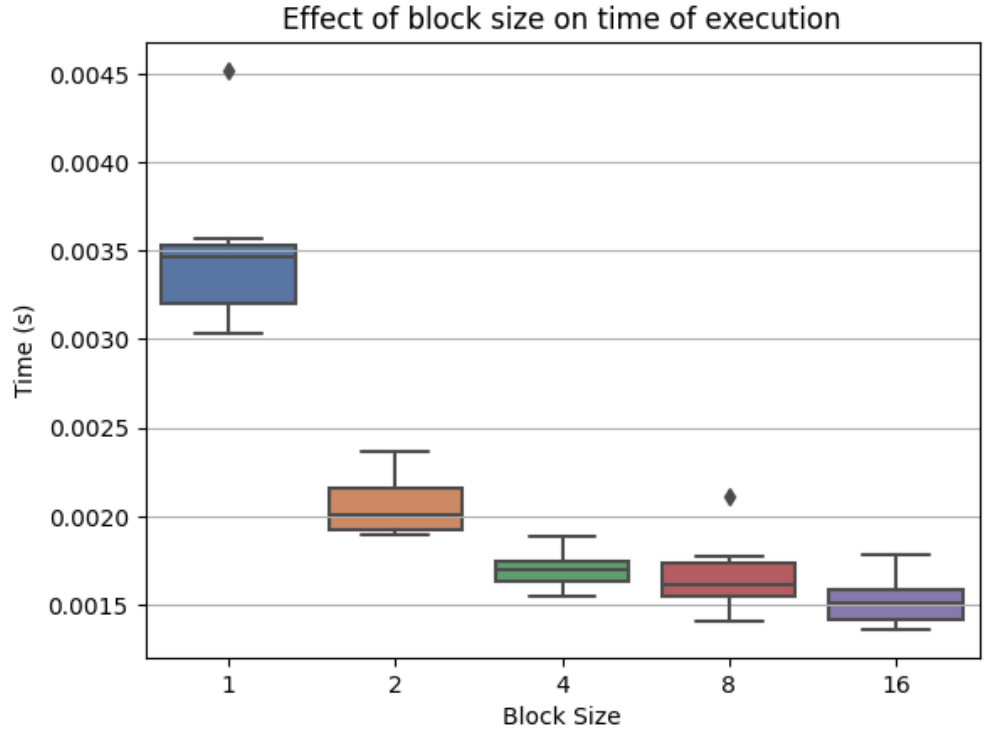
4. Στον κώδικα που προέκυψε από το προηγούμενο ερώτημα επιθυμούμε στη συνέχεια να εφαρμόσουμε Design Space Exploration αναφορικά με την εύρεση του βέλτιστου μεγέθους μπλοκ B. Μεταβάλλουμε τον προηγούμενο κώδικα, ούτως ώστε να δέχεται ως είσοδο το B (βλ. αρχείο phods_B.c). Τρέχουμε 10 φορές τη νέα υλοποίηση με το script run.sh (*./run.sh B*), για τις διάφορες τιμές του B που είναι κοινοί διαιρέτες του N και του M. ΄Επειτα, με το script best_block.py (*python3 scripts/best_block.py B*) λαμβάνουμε το καλύτερο μέγεθος block, αλλά και τα στατιστικά του. Παρατηρούμε ότι για B=16 έχουμε τον καλύτερο χρόνο.

Stats	B DSE Time (sec)
Minimum	0.001360
Median	0.001509
Average	0.001516
Maximum	0.001787

Η τιμή αυτή σχετίζεται με το line size της cache μας, το οποίο το βρήκαμε 64 KB. Η τιμή 16 που δώσαμε είναι μικρότερη του line size της cache μας, αλλά η μεγαλύτερη δυνατή, οπότε κάνει το μέγιστο fit. Παρατηρούμε επίσης από τον κώδικα, στον τρόπο επεξεργασίας των δεδομένων, ότι υπάρχει πολύ καλή εκμετάλλευση του spatial locality όσο επεξεργαζόμαστε ένα block, αλλά όχι data reuse όταν μεταβαίνουμε στο επόμενο. Αυτό επιβεβαιώνει ότι όσο μεγαλώνει το block size το οποίο κάθε φορά κάνουμε access, τόσα περισσότερα hits (κι επομένως λιγότερα conflict misses) έχουμε στην cache μας.

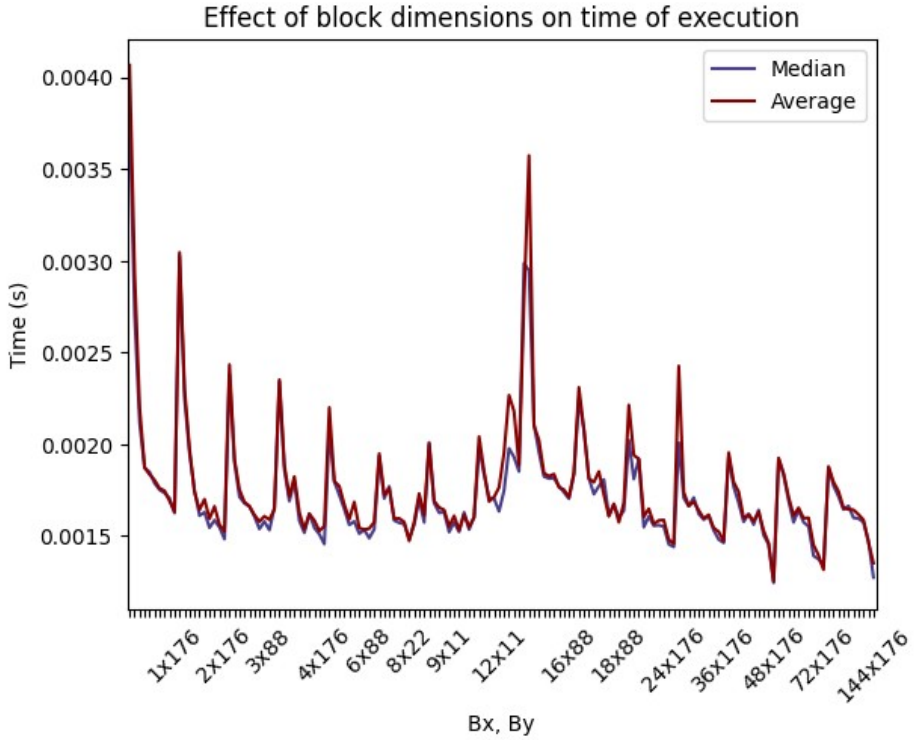
Τη χρονική συμπεριφορά της υλοποίησης αποτυπώσαμε και στα δύο ακόλουθα διαγράμματα, τα οποία παράγουμε με το script plot_B.py (*python3 scripts/plot_B.py*). Το πρώτο είναι ένα απλό γράφημα, όπου φαίνονται οι μέσοι και οι διάμεσοι χρόνοι για κάθε τιμή του B, ενώ το δεύτερο αποτελεί boxplot που συνοψίζει το σύνολο της πληροφορίας για κάθε μέγεθος block (min, max, median, outliers).


Effect of block size on time of execution
Median
Average
0.00350
0.00325
0.00300
0.00275
0.00250
0.00225
0.00200
0.00175
0.00150
Time (s)
1
2
4
8
16
Block Size


Effect of block size on time of execution
0.0045
0.0040
0.0035
0.0030
0.0025
0.0020
0.0015
Time (s)
1
2
4
8
16
Block Size

5. Στον κώδικα που προέκυψε από το ερώτημα 3, θα εφαρμόσουμε στη συνέχεια Design Space Exploration αναφορικά με την εύρεση του βέλτιστου μεγέθους μπλοκ, θεωρώντας αυτή τη φορά ορθογώνιο μπλοκ διαστάσεων Bx, By, όπου Bx διαιρέτης του N και By διαιρέτης του M. Θα προσπαθήσουμε, αρχικά, να διατυπώσουμε μια ευριστική μέθοδο για την εύρεση του καλύτερου παραλληλογράμμου:
    - Έχοντας ως αφετηρία την παρατήρηση από το προηγούμενο ερώτημα ότι σημειώνουμε την καλύτερη χρονική επίδοση για B=16, μπορούμε να ξεκινήσουμε την αναζήτηση δίνοντας στα Bx, By την τιμή 16.
    - Έπειτα, δοκιμάζουμε τη σταδιακή αύξηση του Bx κρατώντας σταθερό το By και κρατώντας πάντα τους χρόνους των 2 τελευταίων επαναλήψεων. Αν παρατηρήσουμε χρόνο χειρότερο από τις 2 τελευταίες επαναλήψεις σταματάμε κρατώντας την προηγούμενη τιμή.
    - Ύστερα, μειώνουμε σταδιακά το Bx εκκινώντας από το 16x16 και επαναλαμβάνουμε την ίδια μεθοδολογία.
    - Έπειτα, από τα διαφορετικά Bx που ελέγξαμε, κρατάμε το Bx που μας έχει δώσει τον καλύτερο χρόνο, κι εφαρμόζουμε την ίδια μεθοδολογία για το By. Παίρνουμε, διαλέγουμε λοιπόν το By που μας δίνει τον βέλτιστο χρόνο για το Bx που είχαμε διαλέξει προηγουμένως. Η προσέγγιση αυτή θεωρείται “άπληστη”, και πολλές φορές ενδέχεται να μη δώσει βέλτιστη λύση, σίγουρα όμως δίνει μια προσεγγιστικά καλή.

    Προκειμένου να λάβουμε περισσότερο ακριβή αποτελέσματα θα προβούμε στην εξαντλητική αναζήτηση του χώρου σχεδιασμού. Ειδικότερα με τη βοήθεια του script run.sh και πάλι εκτελούμε το πρόγραμμα phods_rect για όλους τους συνδυασμούς Bx, By (*./run.sh rect*). Τα αποτελέσματα είναι πάρα πολλά για να αποτυπωθούν σε ένα διάγραμμα. Για να δείξουμε, ωστόσο, την τάση των τιμών για κάθε συνδυασμό, σχεδιάσαμε το script plot_rect.py (*./plot_rect.py*). Αυτό υπολογίζει τη διάμεση τιμή και το μέσο όρο των 10 εκτελέσεων για κάθε συνδυασμό και την αποτυπώνει σε ένα γράφημα.

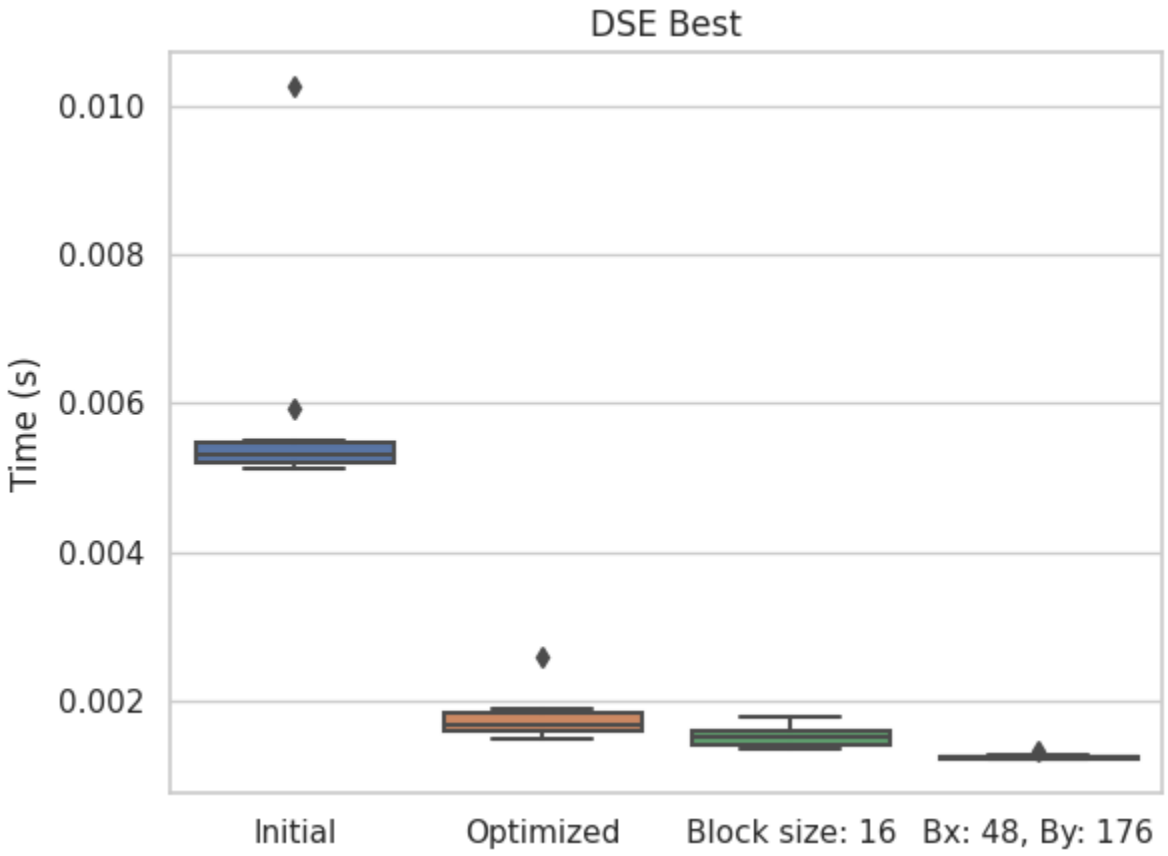


Στον άξονα x του παραπάνω διαγράμματος εμφανίζονται μόνο οι διαστάσεις του ορθογωνίου για τα τοπικά ελάχιστα, αφού δε χωρούν και τα 150 labels.
Παρατηρούμε πως για σταθερό Bx οι συναρτήσεις median (Bx=σταθ, By) και average (Bx=σταθ, By) είναι ποιοτικά φθίνουσες για όλα τα Bx. Επίσης για σταθερό By οι συναρτήσεις median (Bx=σταθ, By) και average (Bx=σταθ, By) είναι ποιοτικά φθίνουσες για Bx < 16, για την τιμή Bx = 16 σημειώνουν πολύ κακούς χρόνους σε σχέση με τους υπόλοιπους συνδυασμούς, κι έπειτα για μεγαλύτερες τιμές του Bx, είναι και πάλι ποιοτικά φθίνουσες. Γενικότερα, καταλαβαίνουμε ότι μεγάλες τιμές του Bx και By είναι προτιμητέες και μας δίνουν πολύ καλύτερα αποτελέσματα. Στις περισσότερες εκτελέσεις, ο συνδυασμός (Bx, By) = (144, 176) υπήρξε ο βέλτιστος, δίνοντας μας το καλύτερο fit του block στην cache. Στην προκειμένη, βέβαια ο καλύτερος χρόνος εμφανίζεται για (Bx, By) = (48, 176). Το γεγονός αυτό μπορεί να επιβεβαιωθεί και με τη χρήση του script best_block.py για την ανάλυση των δεδομένων εξόδου (*python3 scripts/best_block.py rect*), το οποίο μας δίνει ότι το καλύτερο ορθογώνιο είναι πράγματι το [48, 176], τα στατιστικά του οποίου φαίνονται ακολούθως.

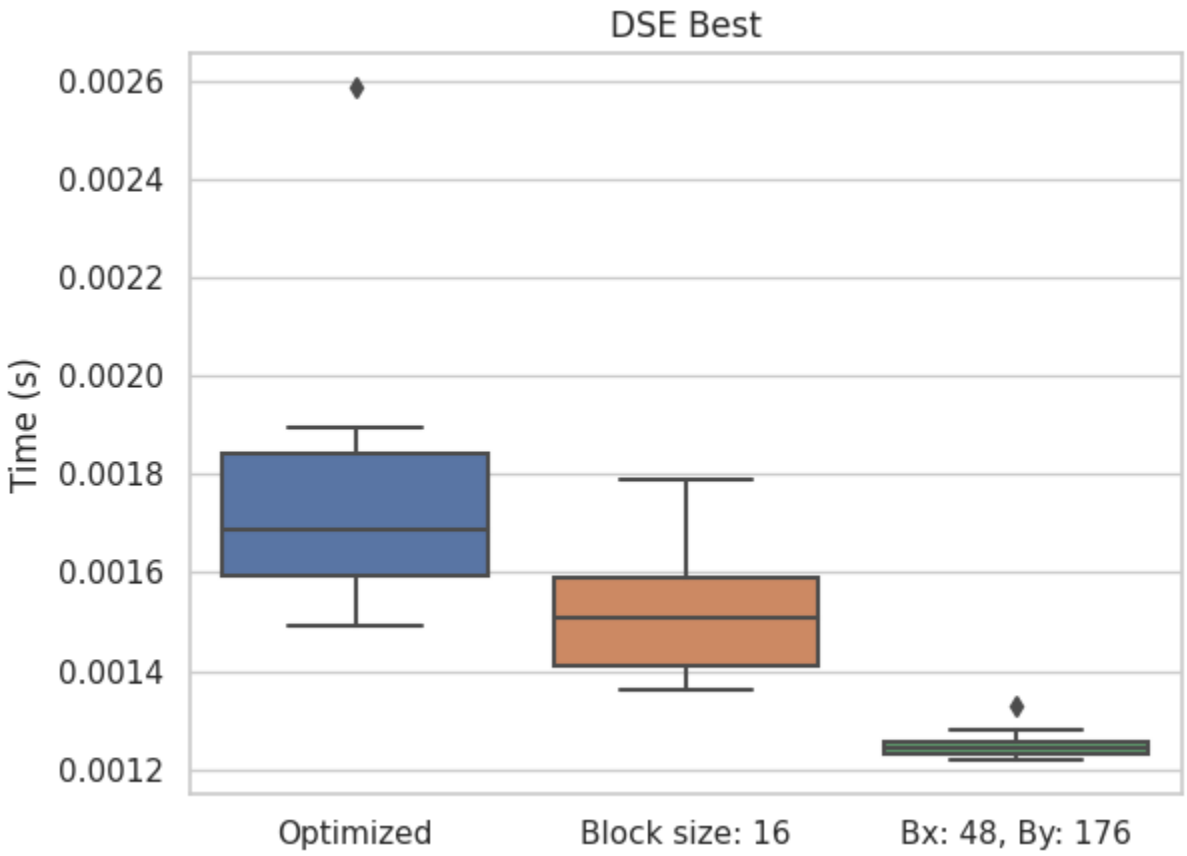
Stats	**Bx-By DSE Time (sec)**
Minimum	0.001219
Median	0.001244
Average	0.001252
Maximum	0.001330

Τέλος, με βάση την ποιοτική συμπεριφορά του διαγράμματος που παραθέσαμε, έρχεται να προστεθεί και άλλη μια πιθανή ευριστική μέθοδος, που δύναται να οδηγήσει σε καλά αποτελέσματα. Δεδομένης της συμπεριφοράς που περιγράψαμε παραπάνω, θα μπορούσαμε να εφαρμόσουμε δειγματοληψία ανά περιοχές και στην περιοχή που θα βρούμε τον καλύτερο χρόνο να εφαρμόσουμε εξαντλητική αναζήτηση.

6. Τέλος, ενδιαφέρον παρουσιάζει η συγκριτική εποπτεία των επιδόσεων των προηγούμενων υλοποιήσεων. Ο καλύτερος τρόπος για την απεικόνιση της πληροφορίας που έχουμε καταγράψει είναι ένα boxplot, προκειμένου να διακρίνονται τόσο, οι ελάχιστοι και μέγιστοι χρόνοι εκτέλεσης της κάθε υλοποίησης, οι διάμεσες τιμές, καθώς και ακραίες τιμές (outliers) που προκύπτουν κατά τρόπο τυχαίο και είναι λογικό να εμφανίζονται όταν ο χώρος εκτέλεσης των προγραμμάτων δεν είναι απομονωμένος. Αυτές αποτυπώνονται στο διάγραμμα ορθώς έξω από τα όρια της κατανομής των τιμών, διευκολύνοντάς μας να προβούμε σε συμπεράσματα που ανταποκρίνονται στην πραγματικότητα. Για το σχεδιασμό της εν λόγω γραφικής χρησιμοποιήσαμε το script best_times.py (*python3 boxplot.py*), το οποίο τρέξαμε αρχικά με σκοπό να εμφανίσουμε τα καλύτερα αποτελέσματα των τεσσάρων υλοποιήσεων: της αρχικής (initial), της βελτιστοποιημένης (opt), της βελτιστοποιημένης με το block size ως είσοδο (B) και της βελτιστοποιημένης με παραλληλόγραμμο block και τα Bx, By ως εισόδους. Ωστόσο, όπως φαίνεται και στη συνέχεια, η αρχική υλοποίηση παρουσιάζει πολύ χειρότερες επιδόσεις από τις άλλες 3, ενώ υπάρχουν γι' αυτή και δύο outliers. Λόγω αυτών, η κλίμακα του γραφήματος δεν είναι η ιδανική, ενώ η εποπτική ανάλυσή του καθίσταται ιδιαίτερα δύσκολη.



Για το λόγο αυτό, προσαρμόσαμε το παραπάνω script, παραλείποντας την αρχική υλοποίηση και εμφανίζοντας μόνο τις 3 βελτιστοποιημένες. Τώρα, είναι ευκολότερο να διακρίνουμε τα χαρακτηριστικά τους.



Μπορούμε, λοιπόν να πούμε συμπερασματικά ότι, αν και η βελτιστοποίηση της αρχικής υλοποίησης που περιγράψαμε προηγουμένως βελτίωσε σημαντικά την επίδοση, τα πιο ενδιαφέροντα αποτελέσματα προέκυψαν από το design space exploration που πραγματοποιήσαμε. Ειδικότερα, διαπιστώσαμε ότι αναφορικά με την υλοποίηση του τετράγωνου block, το μέγεθος block 16 είναι το καλύτερο. Ωστόσο, αυτή δεν είναι η βέλτιστη λύση, αφού όπως φαίνεται- για παραλληλόγραμμο block- και συγκεκριμένα block διαστάσεων 48-176, έχουμε εκτός από πολύ χαμηλή μέση και διάμεση τιμή χρόνου εκτέλεσης και πολύ μικρή διασπορά των τιμών. Αυτό σημαίνει ότι η εκτέλεση της συγκεκριμένης υλοποίησης απαιτεί σταθερά χαμηλό χρόνο, χωρίς αυξομειώσεις και, συνεπώς, επηρεάζεται λιγότερο από εξωτερικούς παράγοντες. Αξίζει, όσον αφορά την τελευταία παρατήρηση να επισημάνουμε ότι οι εκτελέσεις των Optimized και Τετραγωνικού block με μέγεθος 16 θα έπρεπε να πάρουν τον ίδιο ακριβώς χρόνο, αφού η Optimized έτσι κι αλλιώς χρησιμοποιεί block size ίσο με 16. Η διαφορά που παρατηρείται παραπάνω είναι τυχαία και οφείλεται αποκλειστικά σε εξωτερικούς παράγοντες, όπως άλλες διεργασίες που έτρεχαν την ώρα που εκτελούταν κάθε μία εξ αυτών, με αποτέλεσμα διαφορετική κατάσταση της μνήμης για τις δύο εκτελέσεις, διαφορετικές καθυστερήσεις για context switching κλπ.

Ζητούμενο 2ο : Αυτοματοποιημένη βελτιστοποίηση κώδικα

Σημείωση: Το σύνολο του κώδικα που χρησιμοποιήθηκε για το 2ο μέρος της άσκησης βρίσκεται εντός του φακέλου 2. Οι έξοδοι των εκτελέσεων βρίσκονται εντός του φακέλου results, ενώ το script για το boxplot και η γραφική που παράγει βρίσκονται στο root μαζί με τους κώδικες C. Ο τρόπος εκτέλεσης κάθε script περιγράφεται στο README.md που περιλαμβάνεται στον φάκελο 2.

Στο μέρος αυτό της άσκησης καλούμαστε να χρησιμοποιήσουμε το εργαλείο Orio προκειμένου να πραγματοποιήσουμε αυτοματοποιημένη βελτιστοποίηση κώδικα. Συγκεκριμένα, αναζητούμε το βέλτιστο Unroll Factor για δεδομένη for loop 1.000.000 επαναλήψεων, εφαρμόζοντας διαφορετικές μεθόδους αναζήτησης με τη βοήθεια του Orio.

1. Εκτελούμε, αρχικά, τον δοθέντα κώδικα tables.c (έχουμε μετονομάσει το αρχείο σε tables_initial.c προς επίτευξη συνέπειας στην ονοματοδοσία) με τη βοήθεια του script run.sh (*./run.sh initial*), όπως και στην προηγούμενη άσκηση, και λαμβάνουμε τα στατιστικά για την έξοδό του με το script min_max_avg.py (*python3 ../min_max_avg.py results/out/tables_initial.out*). Τα αποτελέσματα παρουσιάζονται στον ακόλουθο πίνακα:

Stats	Initial Time (sec)
Minimum	0.336552
Median	0.351880
Average	0.351203
Maximum	0.368733

2. Ακολούθως, προσαρμόζουμε το αρχείο tables_orio.c αντικαθιστώντας το for loop με το αντίστοιχο προς βελτιστοποίηση κομμάτι του αρχικού κώδικα και το εκτελούμε (*sudo orcc tables_orio.c*) για κάθε έναν από τους 3 τρόπους αναζήτησης που προτείνονται:
    - Exhaustive
    - Random
    - Simplex

    Θα πρέπει, πρώτα, να αλλάξουμε την οδηγία μεταγλώττισης στο αρχείο tables_orio.c, προσθέτοντας τον όρο -mcmodel=large, προκειμένου να είμαστε σε θέση να δεσμεύσουμε μνήμη για τον πίνακα των 100.000.000 θέσεων που δίνεται στον προς βελτιστοποίηση κώδικα.

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

    Για την εκτέλεση της εξαντλητικής αναζήτησης δε χρειάζονται αλλαγές στο tables_orio.c, αφού ο τρόπος αναζήτησης είναι ήδη ορισμένος:

    ```
    def search {
      arg algorithm = 'Exhaustive';
    }
    ```

Για τους άλλους δύο αλγορίθμους αναζήτησης, ωστόσο, θα πρέπει να προσαρμόσουμε το κομμάτι αυτό του κώδικα:

```
def search {
  arg algorithm = 'Randomsearch';
  arg time_limit = 10;
  arg total_runs = 10;
}
```

```
def search {
  arg algorithm = 'Simplex';
  arg time_limit = 10;
  arg total_runs = 10;
  arg simplex_local_distance = 2;
  arg simplex_reflection_coef = 1.5;
  arg simplex_expansion_coef = 2.5;
  arg simplex_contraction_coef = 0.6;
  arg simplex_shrinkage_coef = 0.7;
}
```

Μετά την εκτέλεση του tables_orio.c για κάθε τρόπο αναζήτησης, προκύπτουν οι ακόλουθοι unroll factors.

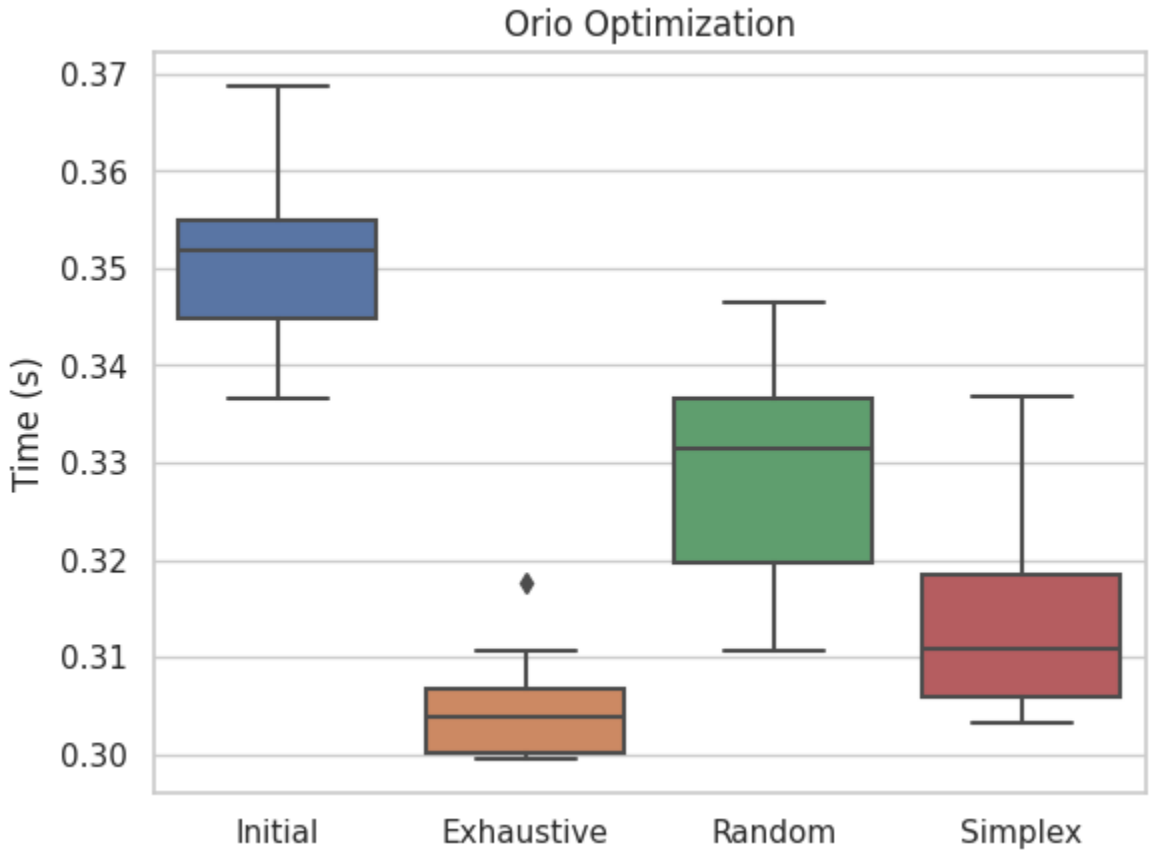
	Exhaustive	**Random**	**Simplex**
Unroll Factor (UF)	5	15	14

Τα νούμερα που προκύπτουν για τον Unroll Factor από κάθε αναζήτηση είναι διαφορετικά, γεγονός που οφείλεται στο διαφορετικό αλγόριθμο αναζήτησης κάθε φορά. Ειδικότερα, η Exhaustive Search, αποτελεί εξαντλητική αναζήτηση. Δοκιμάζει, δηλαδή, όλους τους δυνατούς unroll factors (στο εύρος που έχουμε ορίσει: [1, 32]). Αναμένουμε, συνεπώς, αυτή να παράξει το βέλτιστο unroll για τον αρχικό βρόχο. Πράγματι, η προσδοκία μας αυτή θα επιβεβαιωθεί μετά τη λήψη μετρήσεων επίδοσης για τα διαφορετικά προγράμματα. Ωστόσο, αυτό δε σημαίνει ότι οι άλλοι δύο αλγόριθμοι είναι κακοί. Πιο συγκεκριμένα, εκείνοι χρειάζονται λιγότερη ώρα για την ολοκλήρωση της αναζήτησης, αφού ο Random αφενός πραγματοποιεί τυχαίες επιλογές factors (10 runs εδώ) και διαλέγει την καλύτερη ανάμεσα σ’ αυτές, ο Simplex αφετέρου χρησιμοποιεί μια ευριστική λύση για να καταλήξει σε μια απόφαση. Φυσικά, δεν αποκλείεται οι δύο αυτοί αλγόριθμοι να συμβεί να επιστρέψουν ίδιο αποτέλεσμα με τον εξαντλητικό, ειδικά όταν το εύρος αναζήτησης είναι μικρό όπως στην προκειμένη, γεγονός άλλωστε που παρατηρήθηκε ορισμένες φορές.

3. Στη συνέχεια, αντικαθιστούμε τον αρχικό βρόχο με τον “ξετυλιγμένο” που προέκυψε από την αναζήτηση του Orio. Φτιάχνουμε για ευκολία 3 διαφορετικά προγράμματα: tables_exhaustive.c, tables_random.c, tables_simplex.c, καθένα εκ των οποίων περιλαμβάνει το αποτέλεσμα της αντίστοιχης αναζήτησης. Έπειτα τα μεταγλωττίζουμε με make και τα τρέχουμε με το script ./run.sh (*./run.sh initial/exhaustive/random/simplex*). Τα αποτελέσματα των εκτελέσεων των βελτιστοποιημένων προγραμμάτων λαμβάνονται και πάλι με τη βοήθεια του script min_max_avg.py (*python3 ../min_max_avg.py results/out/tables_version.out*) και φαίνονται στον ακόλουθο πίνακα, όπου για λόγους πληρότητας έχουμε συμπεριλάβει και τους χρόνους που αντιστοιχούν στην αρχική υλοποίηση.

ORIO	Initial	Exhaustive	Random	Simplex
Minimum	0.336552	0.299499	0.310582	0.303279
Median	0.351880	0.303757	0.331473	0.310764
Average	0.351203	0.304864	0.328935	0.314324
Maximum	0.368733	0.317617	0.346418	0.336829

Τοποθετήσαμε, στη συνέχεια, τα στοιχεία που λαμβάνουμε από τις προηγούμενες εκτελέσεις σε ένα boxplot, προκειμένου το περιεχόμενο του πίνακα να γίνει περισσότερο αντιληπτό εποπτικά:



Παρατηρώντας το boxplot με τις διάφορες μεθόδους αναζήτησης του Orio για το Unroll Factor του κώδικα, παρατηρούμε πως και με τις τρεις βρίσκεται ένας UF που βελτιώνει την επίδοση του αρχικού κώδικα. Όπως προβλέψαμε, βέβαια, τα καλύτερα αποτελέσματα σημειώνονται για τη βελτιστοποίηση του exhaustive search, ενώ ακολουθεί η simplex μέθοδος και, τέλος, η random που σημειώνει τα “χειρότερα” αποτελέσματα βελτίωσης, λόγω της τυχαιότητας.