ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ & ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

Σχεδιασμός Ενσωματωμένων Συστημάτων
9ο Εξαμηνο ΗΜΜΥ

Συνεργάτες: Γεώργιος-Ταξιάρχης Γιαννιός, Α.Μ.: 03116156
Μπέτζελος Χρήστος, Α.Μ.: 03116067

1η Εργαστηριακή Άσκηση:
ΒΕΛΤΙΣΤΟΠΟΙΗΣΗ ΑΛΓΟΡΙΘΜΩΝ ΓΙΑ ΧΑΜΗΛΗ ΚΑΤΑΝΑΛΩΣΗ ΕΝΕΡΓΕΙΑΣ ΚΑΙ ΥΨΗΛΗ ΑΠΟΔΟΣΗ

Ζητούμενο 1ο : Loop Optimizations & Design Space Exploration

Στο προσωπικό μας υπολογιστή πραγματοποιήσαμε τα παρακάτω ερωτήματα:

1. Μέσω εντολών του Unix, καταγράψαμε τα χαρακτηριστικά του υπολογιστή στον οποίο πραγματοποιήσαμε την εργαστηριακή άσκηση. Για κάθε χαρακτηριστικό σημειώσαμε και την αντίστοιχη εντολή με την οποία βρήκαμε την ζητούμενη πληροφορία. Συγκεκριμένα:

```
lsb_release -r
uname -r
lscpu
sudo lshw -short
vi /proc/cpuinfo/
```

Έκδοση λειτουργικού	Έκδοση πυρήνα linux	CPU(s)	CPU MHz
Ubuntu 18.04	5.4.0-53-generic	4	800.055

L1d cache	L1i cache	L2 cache	L3 cache	RAM	Block size
32K	32K	256K	3072K	4GB	64

2. Στην συνέχεια μετασχηματήσαμε τον δοθέντα κώδικα ώστε να μετριέται ο χρόνος που χρειάζεται η συνάρτηση *phods_motion_estimation* για να εκτελεστεί. Η μέτρηση του χρόνου έγινε με την συνάρτηση gettimeofday με ακρίβεια μικροδευτερολέπτων. Ο κώδικας αφού εκτελέστηκε 10 φορές, μετρήσαμε το μέσο όρο, μέγιστο και ελάχιστο χρόνο εκτέλεσης.Τα αποτελέσματα της μέτρησης φαίνονται παρακάτω:

Η *main()*, μετά τον μετασχηματισμό της, ώστε να μετριέται ο χρόνος εκτέλεσης της *phods_motion_estimation*, είναι η εξής:

```
int main()
{
  double time;
  struct timeval ts,tf;

  int current[N][M], previous[N][M], motion_vectors_x[N/B][M/B],
      motion_vectors_y[N/B][M/B], i, j;

  read_sequence(current,previous);

  gettimeofday(&ts, NULL);
  phods_motion_estimation(current,previous,motion_vectors_x,
       motion_vectors_y);
  gettimeofday(&tf,NULL);

  time=(tf.tv_sec-ts.tv_sec)+(tf.tv_usec-ts.tv_usec) * 0.000001;
  printf("%lf\n", time);

  return 0;
}
```

Το σκριπτάκι σε python (time.py) που χρησιμοποιήσαμε για να τρέξουμε δέκα φορές τον κώδικα και να εκτυπώσουμε (min, average, max), είναι το εξής:

```
#!/usr/bin/python3

import sys
import time
from subprocess import check_output

if __name__ == '__main__':
    if len(sys.argv) != 3:
        print("Usage: time.py [phods/phods_opt] <iterations>")
        raise SystemExit

    cmd = ['./' + sys.argv[1]]
```

```
    iterations = int(sys.argv[2])

    sum_time = 0
    min_time = 1
    max_time = 0

    for i in range(iterations):
        value = float(check_output(cmd).decode())

        #print('Value', i, ':', value)
        sum_time += value
        if value < min_time:
            min_time = value
        if value > max_time:
            max_time = value

    avg_time = sum_time/iterations

    print('(min, average, max)= ', min_time, max_time,
str.format('{0:.6f}',avg_time))
```

Οι χρόνοι που εκτύπωσε το σκριπτάκι είναι οι εξής:

Minimum (s)	Average (s)	Maximum (s)
0.006673	0.006882	0.007483

3. Με δεδομένη την ύπαρξη της υποδομής για την μέτρηση του χρόνου που χρειάζεται το κρίσιμο κομμάτι της εφαρμογής για να εκτελεστεί, εφαρμόσαμε μετασχηματισμούς στον κώδικα ώστε να μειωθεί ο χρόνος εκτέλεσης. Για κάθε μετασχηματισμό, καταγράψαμε την αντίστοιχη αλλαγή και δικαιολογήσαμε το λόγο για τον οποίο την πραγματοποιήσαμε. Αφού ολοκληρώσαμε τους μετασχηματισμούς στον κώδικα, ξαναεκτελέσαμε τον κώδικα 10 φορές και μετρήσαμε μέσο όρο, μέγιστο και ελάχιστο χρόνο εκτέλεσης, αντίστοιχα με το ερώτημα 2. Τέλος συγκρίναμε τους χρόνους εκτέλεσης με αυτούς του ερωτήματος 2.

Αρχικά, παρατηρείται πως τα κομμάτια που υπολογίζουν τα *distx* και *disty* βρίσκονται σε βρόχους εκτέλεσης με ίδια άκρα και είναι ανεξάρτητα μεταξύ τους. Συνεπώς τα κάναμε merge (**loop fusion**) σε έναν κοινό βρόχο.

```
/*For all candidate blocks in X dimension*/
/*For all candidate blocks in Y dimension*/
        for(i=-S; i<S+1; i+=S)
        {
          distx = 0;
         disty = 0;
          /*For all pixels in the block*/
          for(k=0; k<B; k++)
          {  ...
              ... }
```

Ξανατρέχουμε το σκριπτάκι στην ανανεωμένη phods.c (phods_v2.c) και έχουμε χρόνους:

Minimum (s)	Average (s)	Maximum (s)
0.006320	0.006508	0.006815

Παρατηρούμε μια μικρή βελτίωση στο χρόνο εκτέλεσης της τάξης του **5.4 %**

Στη συνέχεια, παρατηρείται ότι μέσα στα loops των x και y, τα *S* και *i* δεν κάνουν loop εώς κάποιο όριο που εξαρτάται από το input, αλλά πάνω σε σταθερές τιμές, οπότε χρησιμοποιήσαμε macros για να τα κάνουμε unroll. (**Loop Unrolling**). Ο κώδικας (phods_v3.c) μετά από αυτές τις αλλαγές είναι ο εξής:

```
for(x=0; x<N/B; x++)
{
    for(y=0; y<M/B; y++)
    {
        FOR_LOOP_S(4);
        FOR_LOOP_S(2);
        FOR_LOOP_S(1);
    }
}
```

όπου,

```
#define FOR_LOOP_S(s)           /
       min1 = 255*B*B;           /
       min2 = 255*B*B;           /
       FOR_LOOP_I(-s);           /
       FOR_LOOP_I(0);            /
       FOR_LOOP_I(s);            /
       vectors_x[x][y] += bestx; /
       vectors_y[x][y] += besty;
```

και :

```
#define FOR_LOOP_I(i)                  /
      distx = 0;                       /
      disty = 0;                       /
      for(k=0;k<B;k++)                 /
      {                                /
            [...]                      /
      }                                /
      if(distx<min1)                   /
      {                                /
       min1= distx;                    /
       bestx = i;                      /
      }                                /
      if(disty<min2)                   /
      {                                /
      min2=disty;                      /
      besty=i;
      }
```

Ξανατρέχουμε το σκριπτάκι στην ανανεωμένη phods.c (phods_v3.c) και έχουμε χρόνους:

Minimum (s)	Average (s)	Maximum (s)
0.006020	0.006175	0.006421

Παρατηρούμε πάλι μια μικρή βελτίωση στο χρόνο εκτέλεσης συγκριτικά με το phods_v2 της τάξης του **5.1%**

Τέλος, παρατηρήθηκε πώς στις παραπάνω μακροεντολές γίνεται πρόσβαση στα στοιχεία του πίνακα vectors_x[x][y] και vectors_y[x][y] πολλαπλές φορές μέσα σε κάθε επανάληψη, χωρίς αυτά να έχουν αλλάξει τιμή. Γι' αυτό

τροποποιήθηκε ο κώδικας ώστε αυτά τα στοιχεία να υπολογίζονται μόνο μία φορά σε κάθε επανάληψη του συνολικού διπλού βρόχου. Έτσι, όταν χρησιμοποιείται ξανά, δεν γίνεται ξανά πρόσβαση στην εξωτερική μνήμη, γιατι η τιμή βρίσκεται στην μνήμη cache του επεξεργαστή. Ο νέος ανανεωμένος κώδικας phods_v4.c είναι ο παρακάτω:

```
for(x=0; x<N/B; x++)
  {
       for(y=0; y<M/B; y++)
       {
      int array_x = vectors_x[x][y];
      int array_y = vectors_y[x][y];
      FOR_LOOP_S(3);
      FOR_LOOP_S(2);
      FOR_LOOP_S(1);
       }
  }
```

Επιπλέον, προφανώς αντικαταστήσαμε και στις μακροεντολές FOR_LOOP_I και FOR_LOOP_S το *vectors_x[x][y]* με το *array_x* και το *vectors_y[x][y]* με το array_y. Οι νέοι χρόνοι που προκύψανε είναι οι εξής:

Minimum (s)	Average (s)	Maximum (s)
0.003085	0.003243	0.003555

Αυτή τη φορά παρατηρούμε μια πολύ μεγαλύτερη βελτίωση στο χρόνο εκτέλεσης συγκριτικά με το phods_v3, που είναι της τάξης του **47.5%**.

Συμπερασματικά, όσον αφορά την συνολική βελτίωση του χρόνου, χάρη στο loop fusion, το loop unrolling και το data reuse, από την αρχική *phods*, στην τελική *phods_v4*, έχει μειωθεί το average κατά **52.9%**. Προφανώς, όπως και είδαμε από τα προηγούμενα, τον μεγαλύτερο ρόλο τον έπαιξε το data reuse, αφού οι εντολές πρόσβασης στη μνήμη, ως γνωστόν, είναι αυτές που επιβαρύνουν περισσότερο την εκτέλεση ενός προγράμματος.

4. Στον κώδικα που προέκυψε από το ερώτημα 3 (*phods_v4*), εφαρμόσαμε Design Space Exploration αναφορικά με την εύρεση του βέλτιστου μεγέθους μπλοκ B (μεταβλητή στον κώδικα) για την αρχιτεκτονική του υπολογιστή μας. Η αναζήτηση πραγματοποιήθηκε με εξαντλητική αναζήτηση και μόνο για τιμές του B που είναι κοινοί διαιρέτες του M και του N.

Έχουμε ότι M=176 και N=144, οπότε οι κοινοί διαιρέτες είναι οι [1, 2, 4, 8, 16]. Επομένως, για κάθε μία από τις 5 δυνατές τιμές του B, το πρόγραμμα εκτελέστηκε 10 φορές και ως μετρική απόδοσης ορίστηκε ο μέσος όρος του χρόνου εκτέλεσης σε αυτές τις 10 εκτελέσεις. Παρατηρούμε ότι η phods.c έχει από μόνη της καθορισμένη την τιμή του Block Size στο 16 (*#define B 16*), επομένως, χρειάστηκε τροποποίηση του κώδικα ώστε να δέχεται σαν παράμετρο το μέγεθος B. Συγκεκριμένα προσθέσαμε τις παρακάτω γραμμές στην αρχή της *main().*

```
if (argc == 1)
{
    B = 16;
}
else if (argc == 2)
{
    B = atoi(argv[1]);
}
else
{
    printf("phods_v4 usage: ./phods_v4 block_size\n");
    exit(1);
}
```

Το σκριπτάκι *time_block.py* που τρέξαμε ώστε να μας εμφανίσει τους μέσους χρόνους για όλες τις τιμές του B είναι το παρακάτω:

```
#!/usr/bin/python3

import sys
from subprocess import check_output

if __name__ == '__main__':
    if len(sys.argv) != 3:
        print("Usage: python3 time_block.py phods_v4 [iterations]")
        raise SystemExit

    if sys.argv[1] == 'phods_v4':
        block_sizes = [1, 2, 4, 8, 16]
        for N in block_sizes:
            cmd = ['./' + sys.argv[1], str(N)]
```

```
            iterations = int(sys.argv[2])

            sum_time = 0
            for i in range(iterations):
                value = float(check_output(cmd).decode())
                sum_time += value

            avg_time = suma/iterations
            print('Block size', N, ':', str.format('{0:.6f}', avg_time), 'sec')

    else:
        print("Usage: python3 time_block.py phods_v4 [iterations]")
        raise SystemExit
```

Τα αποτελέσματα που προκύψανε είναι τα εξής:

Block Size	Average Time
1	0.005719
2	0.004396
4	0.003655
8	0.003595
16	0.003218

Άρα, η τιμή του B για την οποία πετυχαίνεται η καλύτερη απόδοση είναι 16.

Τέλος παρατηρήσαμε ότι υπάρχει κάποια σχέση μεταξύ αυτής της τιμής και του μεγέθους block της cache που καταγράψαμε στο ερώτημα 1. Πιο ειδικά, όπως είδαμε και από τον αλγόριθμο *phods_motion_estimation* οι εικόνες (τρέχουσα - προηγούμενη) χωρίζονται σε blocks μεγέθους BxB για να συγκριθούν μεταξύ τους. Κάθε τέτοιο block μεταφερεται στη προσωρινή μνήμη και σε περίπτωση που χωράει, καταλαμβάνει συγκεκριμένο χώρο, όσο το cache block size. Όσο μικρότερο είναι το B, τόσο περισσότερα είναι τα blocks που θα κατανεμηθούν στην cache και επομένως τόσες περισσότερες προσπελάσεις στη μνήμη θα έχουμε. Το βέλτιστο προφανώς θα προέκυπτε στην περίπτωση που το B θα ισούταν με το μέγεθος του cache block, δηλαδή 64. (Θα καλυφθεί έτσι ο λιγότερος αριθμός blocks στην cache).

5. Στον κώδικα που προέκυψε από το ερώτημα 3, εφαρμόσαμε Design Space Exploration αναφορικά με την εύρεση του βέλτιστου μεγέθους μπλοκ θεωρώντας ορθογώνιο μπλοκ διαστάσεων B_x και B_y για την αρχιτεκτονική του υπολογιστή μας.Το μέγεθος του B_x να είναι διαιρέτης του N και τομέγεθος B_y να είναι διαιρέτης του M.

N=144 άρα, B_x = [1, 2, 3, 4, 6, 8, 9, 12, 16, 18, 24, 36, 48, 72, 144]
M=176 άρα, B_y = [1, 2, 4, 8, 11, 16, 22, 44, 88, 176]

Εφαρμόσαμε εξαντλητική αναζήτηση προκειμένου να βρούμε το βέλτιστο ζεύγος B_x, B_y το οποίο ελαχιστοποιεί τον χρόνο εκτέλεσης του προγράμματος. Για κάθε ζεύγος B_x και B_y, το πρόγραμμα εκτελέστηκε 10 φορές και ως μετρική απόδοσης ορίστηκε ο μέσος όρος του χρόνου εκτέλεσης σε αυτές τις 10 εκτελέσεις.
Έγινε προφανώς προσαρμογή του κώδικα ώστε να δέχεται πλέον 2 διαφορετικές διαστάσεις για το Block. Αυτός υπάρχει στο αρχείο phods_v4_2d.c. Οι αλλαγές που έπρεπε να κάνουμε και στην main() είναι οι εξής:

```
if (argc == 1)
    {
        Bx = 16;
        By = 16;
    }
    else if (argc == 3)
    {
        Bx = atoi(argv[1]);
        By = atoi(argv[1]);
    }
    else
    {
        printf("Usage: ./phods_v4_2d block_size_x block_size_y\n");
        exit(1);
    }
```

Δημιουργήσαμε πάλι σκριπτάκι *(time_block2)* το οποίο είναι το εξής:

```
#!/usr/bin/python3

import sys
from subprocess import check_output

if __name__ == '__main__':
	if len(sys.argv) != 3:
	print("Usage: python3 time_block2.py phods_v4_2d [iterations]")
	raise SystemExit

	if sys.argv[1] == 'phods_v4_2d':
	N = 144
	M = 176
	minimum = 1
	bx = 1
	by = 1

	for bx in range (1,N+1,1):
		if N%bx == 0:
		for by in range (1,M+1,1):
			if M%by == 0:
				cmd = ['./' + sys.argv[1], str(bx), str(by)]
				iterations = int(sys.argv[2])

				sum_time = 0
				for i in range(iterations):
				value = float(check_output(cmd).decode())
				sum_time += value

				avg_time = sum_time/iterations
				print('Block size', bx,'x',by,':',
str.format('{0:.6f}', avg_time), 'sec')

			if avg_time < minimum:
				minimum = avg_time
				bx_min = bx
				by_min = by

	print('Best block size:', bx_min, 'x', by_min, ':', str.format('{0:.6f}',
minimum), 'sec')

	else:
	print("Usage: python3 time_block2.py phods_v4_2d [iterations]")
	raise SystemExit
```

Αποτελέσματα από το time_block2:

```
Block size 1 x 1 : 0.005855 sec
Block size 1 x 2 : 0.005895 sec
Block size 1 x 4 : 0.006029 sec
Block size 1 x 8 : 0.005884 sec
Block size 1 x 11 : 0.005823 sec
Block size 1 x 16 : 0.006032 sec
Block size 1 x 22 : 0.005984 sec
Block size 1 x 44 : 0.005939 sec
Block size 1 x 88 : 0.005894 sec
Block size 1 x 176 : 0.006261 sec
Block size 2 x 1 : 0.004408 sec
Block size 2 x 2 : 0.004447 sec
Block size 2 x 4 : 0.004424 sec
Block size 2 x 8 : 0.004433 sec
Block size 2 x 11 : 0.004562 sec
Block size 2 x 16 : 0.004453 sec
Block size 2 x 22 : 0.004368 sec
Block size 2 x 44 : 0.004405 sec
Block size 2 x 88 : 0.004740 sec
Block size 2 x 176 : 0.004528 sec
Block size 3 x 1 : 0.003949 sec
Block size 3 x 2 : 0.003894 sec
Block size 3 x 4 : 0.004158 sec
Block size 3 x 8 : 0.004018 sec
Block size 3 x 11 : 0.003973 sec
Block size 3 x 16 : 0.003922 sec
Block size 3 x 22 : 0.003879 sec
Block size 3 x 44 : 0.004064 sec
Block size 3 x 88 : 0.003918 sec
Block size 3 x 176 : 0.003901 sec
Block size 4 x 1 : 0.003720 sec
Block size 4 x 2 : 0.003585 sec
Block size 4 x 4 : 0.003992 sec
Block size 4 x 8 : 0.003743 sec
Block size 4 x 11 : 0.003660 sec
Block size 4 x 16 : 0.003670 sec
Block size 4 x 22 : 0.003675 sec
Block size 4 x 44 : 0.004166 sec
Block size 4 x 88 : 0.003650 sec
Block size 4 x 176 : 0.003604 sec
Block size 6 x 1 : 0.003500 sec
Block size 6 x 2 : 0.003477 sec
Block size 6 x 4 : 0.003614 sec
Block size 6 x 8 : 0.003552 sec
Block size 6 x 11 : 0.003512 sec
Block size 6 x 16 : 0.003485 sec
Block size 6 x 22 : 0.003421 sec
Block size 6 x 44 : 0.003357 sec
Block size 6 x 88 : 0.003642 sec
Block size 6 x 176 : 0.003489 sec
Block size 8 x 1 : 0.003392 sec
Block size 8 x 2 : 0.003321 sec
Block size 8 x 4 : 0.003489 sec
Block size 8 x 8 : 0.003718 sec
```

```
Block size 8 x 11 : 0.003450 sec
Block size 8 x 16 : 0.003483 sec
Block size 8 x 22 : 0.003471 sec
Block size 8 x 44 : 0.003751 sec
Block size 8 x 88 : 0.003644 sec
Block size 8 x 176 : 0.003467 sec
Block size 9 x 1 : 0.003341 sec
Block size 9 x 2 : 0.003239 sec
Block size 9 x 4 : 0.003302 sec
Block size 9 x 8 : 0.003262 sec
Block size 9 x 11 : 0.003508 sec
Block size 9 x 16 : 0.003261 sec
Block size 9 x 22 : 0.003247 sec
Block size 9 x 44 : 0.003193 sec
Block size 9 x 88 : 0.003403 sec
Block size 9 x 176 : 0.003458 sec
Block size 12 x 1 : 0.003224 sec
Block size 12 x 2 : 0.003129 sec
Block size 12 x 4 : 0.003198 sec
Block size 12 x 8 : 0.003191 sec
Block size 12 x 11 : 0.003227 sec
Block size 12 x 16 : 0.003326 sec
Block size 12 x 22 : 0.003188 sec
Block size 12 x 44 : 0.003236 sec
Block size 12 x 88 : 0.003261 sec
Block size 12 x 176 : 0.003113 sec
Block size 16 x 1 : 0.003336 sec
Block size 16 x 2 : 0.003414 sec
Block size 16 x 4 : 0.003404 sec
Block size 16 x 8 : 0.003200 sec
Block size 16 x 11 : 0.003347 sec
Block size 16 x 16 : 0.003324 sec
Block size 16 x 22 : 0.003482 sec
Block size 16 x 44 : 0.003280 sec
Block size 16 x 88 : 0.003210 sec
Block size 16 x 176 : 0.003266 sec
Block size 18 x 1 : 0.003038 sec
Block size 18 x 2 : 0.003071 sec
Block size 18 x 4 : 0.003066 sec
Block size 18 x 8 : 0.002949 sec
Block size 18 x 11 : 0.002988 sec
Block size 18 x 16 : 0.003080 sec
Block size 18 x 22 : 0.003079 sec
Block size 18 x 44 : 0.003041 sec
Block size 18 x 88 : 0.003126 sec
Block size 18 x 176 : 0.003021 sec
Block size 24 x 1 : 0.003089 sec
Block size 24 x 2 : 0.003076 sec
Block size 24 x 4 : 0.003164 sec
Block size 24 x 8 : 0.003335 sec
Block size 24 x 11 : 0.003063 sec
Block size 24 x 16 : 0.003141 sec
Block size 24 x 22 : 0.003102 sec
Block size 24 x 44 : 0.003068 sec
Block size 24 x 88 : 0.003111 sec
Block size 24 x 176 : 0.003260 sec
```

```
Block size 36 x 1 : 0.002734 sec
Block size 36 x 2 : 0.002736 sec
Block size 36 x 4 : 0.002671 sec
Block size 36 x 8 : 0.002649 sec
Block size 36 x 11 : 0.002716 sec
Block size 36 x 16 : 0.002996 sec
Block size 36 x 22 : 0.002673 sec
Block size 36 x 44 : 0.002688 sec
Block size 36 x 88 : 0.002638 sec
Block size 36 x 176 : 0.002755 sec
Block size 48 x 1 : 0.002686 sec
Block size 48 x 2 : 0.002730 sec
Block size 48 x 4 : 0.002913 sec
Block size 48 x 8 : 0.002736 sec
Block size 48 x 11 : 0.002738 sec
Block size 48 x 16 : 0.002767 sec
Block size 48 x 22 : 0.002713 sec
Block size 48 x 44 : 0.002667 sec
Block size 48 x 88 : 0.002777 sec
Block size 48 x 176 : 0.002729 sec
Block size 72 x 1 : 0.002758 sec
Block size 72 x 2 : 0.002687 sec
Block size 72 x 4 : 0.002690 sec
Block size 72 x 8 : 0.002728 sec
Block size 72 x 11 : 0.002770 sec
Block size 72 x 16 : 0.002666 sec
Block size 72 x 22 : 0.002796 sec
Block size 72 x 44 : 0.002698 sec
Block size 72 x 88 : 0.002755 sec
Block size 72 x 176 : 0.002734 sec
Block size 144 x 1 : 0.002718 sec
Block size 144 x 2 : 0.002638 sec
Block size 144 x 4 : 0.002622 sec
Block size 144 x 8 : 0.002602 sec
Block size 144 x 11 : 0.002737 sec
Block size 144 x 16 : 0.002887 sec
Block size 144 x 22 : 0.002660 sec
Block size 144 x 44 : 0.002740 sec
Block size 144 x 88 : 0.002631 sec
Block size 144 x 176 : 0.002689 sec
```

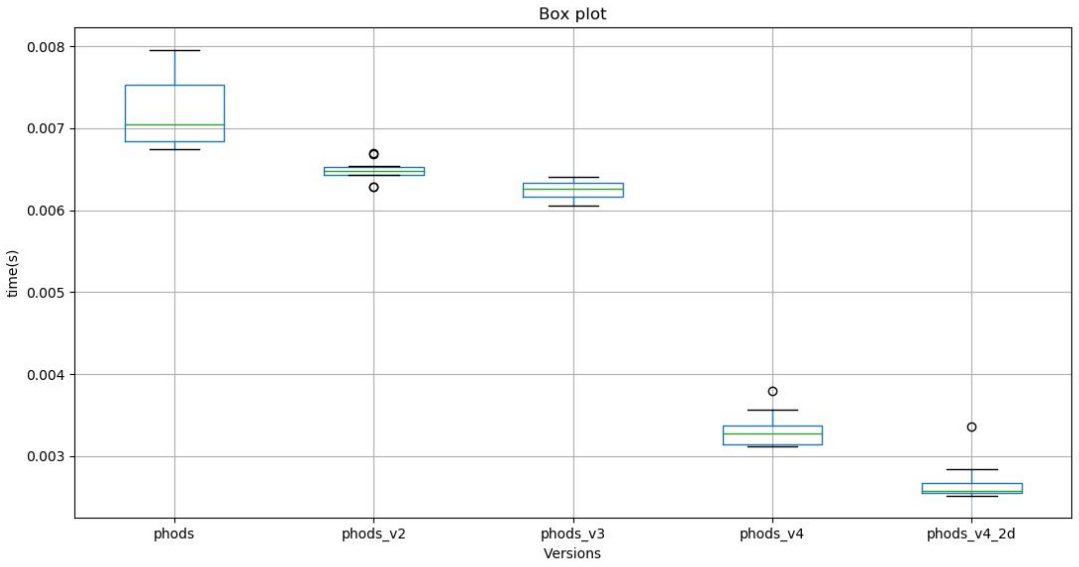
Καταγράψαμε το ζεύγος B_x, B_y για το οποίο πετυχαμε την καλύτερη απόδοση :

```
Best block size: 144 x 8 : 0.002602 sec
```

Ευριστικός τρόπος περιορισμού της αναζήτησης:
Με βάση το προηγούμενο ερώτημα, το B=16 ήταν η βέλτιστη επιλογή για τον μικρότερο χρόνο εκτέλεσης. Επομένως, η αναζήτηση θα μπορούσε να περιοριστεί στα blocks τα οποία έχουν μέγεθος μεγαλύτερο ή ίσο του 16x16.

6.Τέλος απεικονίσαμε τα αποτελέσματα των ερωτημάτων 2-5 σε ένα διάγραμμα boxplot και συγκρίναμε-αναλύσαμε τα πειραματικά αποτελέσματα που λάβαμε. Χρησιμοποιήσαμε το παρακάτω σκριπτάκι *box_plots.py*.

```
#!/usr/bin/python3

import sys
import time
from subprocess import check_output
import pandas as pd
import matplotlib.pyplot as plt

if __name__ == '__main__':
	iterations = 10

	phods=[]
	phods_v2=[]
	phods_v3=[]
	phods_v4=[]
	phods_v4_2d=[]

	for i in range(iterations):
	cmd = ['./' + "phods"]
	value = float(check_output(cmd).decode())
	phods.append(value)

	cmd = ['./' + "phods_v2"]
	value = float(check_output(cmd).decode())
	phods_v2.append(value)

	cmd = ['./' + "phods_v3"]
	value = float(check_output(cmd).decode())
	phods_v3.append(value)

	cmd = ['./' + "phods_v4"]
	value = float(check_output(cmd).decode())
	phods_v4.append(value)

	cmd = ['./' + "phods_v4_2d", str(144), str(8)]
	value = float(check_output(cmd).decode())
	phods_v4_2d.append(value)

	df = pd.DataFrame({'phods': phods, 'phods_v2': phods_v2, 'phods_v3':
phods_v3, 'phods_v4': phods_v4, 'phods_v4_2d': phods_v4_2d})

	boxplot = df.boxplot(column=['phods', 'phods_v2', 'phods_v3', 'phods_v4',
'phods_v4_2d'])
	plt.title("Box plot")
	plt.xlabel("Versions")
	plt.ylabel("time(s)")
	plt.show()
```



Στο παραπάνω διάγραμμα, απεικονίζονται 5 boxes καθένα από τα οποία αφορά κάποιον μετασχηματισμό που εφαρμόσαμε στα προηγούμενα ερωτήματα. Πιο ειδικά:

1. phods: Αρχικός κώδικας (χωρίς μετασχηματισμούς)
2. phods_v2: Εφαρμογή **Loop fusion** (Merge)
3. phods_v3: Εφαρμογή **Loop Unrolling**
4. phods_v4: Εφαρμογή βέλτιστου block **B=16**
5. phods_v5: Εφαρμογή βέλτιστου ζεύγους **Bx=144, By=8**

Παρατηρήσεις:

Σε κάθε νέα έκδοση (που προκύπτει από κάποιον μετασχηματισμό), παρατηρούμε ότι έχουμε βελτίωση στο χρόνο εκτέλεσης τόσο στη μέση τίμη όσο και στις min, max τιμές. Συγκρίνοντας τα boxes με το αρχικό, φαίνεται ότι έχει μειωθεί αρκετά το εύρος τιμών του χρόνου εκτέλεσης.

Ζητούμενο 2ο: Αυτοματοποιημένη βελτιστοποίηση κώδικα

1. Αφού εγκαταστήσαμε επιτυχώς το εργαλείο Orio, το χρησιμοποιήσαμε για βελτιστοποίηση του κώδικα `tables.c`, του οποίου η λειτουργία αφορά απλές προσπελάσεις και πράξεις με πίνακες. Αρχικά, μετρήσαμε τον χρόνο εκτέλεσης του . Η μέτρηση του χρόνου έγινε με την συνάρτηση gettimeofday με ακρίβεια μικροδευτερολέπτων. Εκτελέσαμε τον κώδικα 10 φορές και μετρήσαμε το μέσο όρο, μέγιστο και ελάχιστο χρόνο εκτέλεσης. Χρησιμοποιήσαμε πάλι το σκριπτάκι time.py που είχαμε δημιουργήσει στο ερώτημα 2. Τα αποτελέσματα της μέτρησης φαίνονται παρακάτω:

Minimum (s)	Average (s)	Maximum (s)
0.045432	0.048513	0.059486

2. Προσαρμόσαμε το αρχείο tables_orio.c, για να εφαρμόσουμε τον μετασχηματισμό loop unrolling στο κομμάτι κώδικα του ερωτήματος 1. Για κάθε έναν από τους παρακάτω τρόπους που αναγράφονται στη σελίδα του Orio (Exhaustive, Randomsearch, Simplex) πραγματοποιήσαμε Design Space Exploration για να βρούμε το κατάλληλο loop unrolling factor για αυτό το κομμάτι κώδικα. Επειδή για $N = 10^8$, ο compile του Orio εμφάνιζε το συγκεκριμένο error,μειώσαμε με βάση και την υπόδειξη που μας δόθηκε το N $N = 10^7$.

Ο αρχικός βρόγχος στο αρχείο *table.c* είναι ο παρακάτω:

```
for (i=0; i<=N-1; i++
{
    //This loop needs to be modified after Orio's execution...
    y[i] = y[i] + a1*x1[i] + a2*x2[i] + a3*x3[i];
}
```

➢ Exhaustive: Έχουμε δύο περιπτώσεις

- Για N≤1000 έχουμε Unroll Factor 2

```
for (int loop_loop_38=0; loop_loop_38 < 1; loop_loop_38++)  {
        int i;
        for (i = 0; i <= N - 2; i = i + 2) {
        y[i] = y[i] + a1 * x1[i] + a2 * x2[i] + a3 * x3[i];
        y[(i + 1)] = y[(i + 1)] + a1 * x1[(i + 1)] + a2 * x2[(i + 1)] + a3 * x3[(i + 1)];
        }
        for (i = N - ((N - (0)) % 2); i <= N - 1; i = i + 1)
        y[i] = y[i] + a1 * x1[i] + a2 * x2[i] + a3 * x3[i];
  }
```

- Για N≤10000000 έχουμε Unroll Factor 3

```
for (int loop_loop_38=0; loop_loop_38 < 1; loop_loop_38++)  {
        int i;
        for (i = 0; i <= N - 3; i = i + 3) {
        y[i] = y[i] + a1 * x1[i] + a2 * x2[i] + a3 * x3[i];
        y[(i + 1)] = y[(i + 1)] + a1 * x1[(i + 1)] + a2 * x2[(i + 1)] + a3 * x3[(i + 1)];
        y[(i + 2)] = y[(i + 2)] + a1 * x1[(i + 2)] + a2 * x2[(i + 2)] + a3 * x3[(i + 2)];
        }
        for (i = N - ((N - (0)) % 3); i <= N - 1; i = i + 1)
        y[i] = y[i] + a1 * x1[i] + a2 * x2[i] + a3 * x3[i];
  }
```

➢ Randomsearch: Χρησιμοποιήσαμε τις ακόλουθες παραμέτρους:

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

- Για N≤1000 έχουμε Unroll Factor 5

```
for (int loop_loop_40=0; loop_loop_40 < 1; loop_loop_40++)  {
        int i;
        for (i = 0; i <= N - 5; i = i + 5) {
        y[i] = y[i] + a1 * x1[i] + a2 * x2[i] + a3 * x3[i];
        y[(i + 1)] = y[(i + 1)] + a1 * x1[(i + 1)] + a2 * x2[(i + 1)] + a3 * x3[(i + 1)];
        y[(i + 2)] = y[(i + 2)] + a1 * x1[(i + 2)] + a2 * x2[(i + 2)] + a3 * x3[(i + 2)];
        y[(i + 3)] = y[(i + 3)] + a1 * x1[(i + 3)] + a2 * x2[(i + 3)] + a3 * x3[(i + 3)];
        y[(i + 4)] = y[(i + 4)] + a1 * x1[(i + 4)] + a2 * x2[(i + 4)] + a3 * x3[(i + 4)];
        }
        for (i = N - ((N - (0)) % 5); i <= N - 1; i = i + 1)
        y[i] = y[i] + a1 * x1[i] + a2 * x2[i] + a3 * x3[i];
  }
```

- Για N≤10000000 έχουμε Unroll Factor 15

```
for (int loop_loop_40=0; loop_loop_40 < 1; loop_loop_40++)  {
        int i;
        for (i = 0; i <= N - 15; i = i + 15) {
        y[i] = y[i] + a1 * x1[i] + a2 * x2[i] + a3 * x3[i];
        y[(i + 1)] = y[(i + 1)] + a1 * x1[(i + 1)] + a2 * x2[(i + 1)] + a3 * x3[(i + 1)];
        y[(i + 2)] = y[(i + 2)] + a1 * x1[(i + 2)] + a2 * x2[(i + 2)] + a3 * x3[(i + 2)];
        y[(i + 3)] = y[(i + 3)] + a1 * x1[(i + 3)] + a2 * x2[(i + 3)] + a3 * x3[(i + 3)];
        y[(i + 4)] = y[(i + 4)] + a1 * x1[(i + 4)] + a2 * x2[(i + 4)] + a3 * x3[(i + 4)];
        y[(i + 5)] = y[(i + 5)] + a1 * x1[(i + 5)] + a2 * x2[(i + 5)] + a3 * x3[(i + 5)];
        y[(i + 6)] = y[(i + 6)] + a1 * x1[(i + 6)] + a2 * x2[(i + 6)] + a3 * x3[(i + 6)];
        y[(i + 7)] = y[(i + 7)] + a1 * x1[(i + 7)] + a2 * x2[(i + 7)] + a3 * x3[(i + 7)];
        y[(i + 8)] = y[(i + 8)] + a1 * x1[(i + 8)] + a2 * x2[(i + 8)] + a3 * x3[(i + 8)];
        y[(i + 9)] = y[(i + 9)] + a1 * x1[(i + 9)] + a2 * x2[(i + 9)] + a3 * x3[(i + 9)];
        y[(i + 10)] = y[(i + 10)] + a1 * x1[(i + 10)] + a2 * x2[(i + 10)] + a3 * x3[(i + 10)];
        y[(i + 11)] = y[(i + 11)] + a1 * x1[(i + 11)] + a2 * x2[(i + 11)] + a3 * x3[(i + 11)];
        y[(i + 12)] = y[(i + 12)] + a1 * x1[(i + 12)] + a2 * x2[(i + 12)] + a3 * x3[(i + 12)];
        y[(i + 13)] = y[(i + 13)] + a1 * x1[(i + 13)] + a2 * x2[(i + 13)] + a3 * x3[(i + 13)];
        y[(i + 14)] = y[(i + 14)] + a1 * x1[(i + 14)] + a2 * x2[(i + 14)] + a3 * x3[(i + 14)];
        }
        for (i = N - ((N - (0)) % 15); i <= N - 1; i = i + 1)
        y[i] = y[i] + a1 * x1[i] + a2 * x2[i] + a3 * x3[i];
  }
/*@ end @*/

}
```

➢ Simplex: Στη περίπτωση του αλγορίθμου simplex, χρησιμοποιήσαμε τις ακόλουθες παραμέτρους:

```
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;
}
```

- Για N≤1000 έχουμε Unroll Factor 4

```
for (int loop_loop_45=0; loop_loop_45 < 1; loop_loop_45++)  {
        int i;
        for (i = 0; i <= N - 4; i = i + 4) {
        y[i] = y[i] + a1 * x1[i] + a2 * x2[i] + a3 * x3[i];
        y[(i + 1)] = y[(i + 1)] + a1 * x1[(i + 1)] + a2 * x2[(i + 1)] + a3 * x3[(i + 1)];
        y[(i + 2)] = y[(i + 2)] + a1 * x1[(i + 2)] + a2 * x2[(i + 2)] + a3 * x3[(i + 2)];
        y[(i + 3)] = y[(i + 3)] + a1 * x1[(i + 3)] + a2 * x2[(i + 3)] + a3 * x3[(i + 3)];
        }
        for (i = N - ((N - (0)) % 4); i <= N - 1; i = i + 1)
        y[i] = y[i] + a1 * x1[i] + a2 * x2[i] + a3 * x3[i];
  }
```

- Για N≤10000000 έχουμε Unroll Factor 11

```
for (int loop_loop_45=0; loop_loop_45 < 1; loop_loop_45++)  {
        int i;
        for (i = 0; i <= N - 11; i = i + 11) {
        y[i] = y[i] + a1 * x1[i] + a2 * x2[i] + a3 * x3[i];
        y[(i + 1)] = y[(i + 1)] + a1 * x1[(i + 1)] + a2 * x2[(i + 1)] + a3 * x3[(i + 1)];
        y[(i + 2)] = y[(i + 2)] + a1 * x1[(i + 2)] + a2 * x2[(i + 2)] + a3 * x3[(i + 2)];
        y[(i + 3)] = y[(i + 3)] + a1 * x1[(i + 3)] + a2 * x2[(i + 3)] + a3 * x3[(i + 3)];
        y[(i + 4)] = y[(i + 4)] + a1 * x1[(i + 4)] + a2 * x2[(i + 4)] + a3 * x3[(i + 4)];
        y[(i + 5)] = y[(i + 5)] + a1 * x1[(i + 5)] + a2 * x2[(i + 5)] + a3 * x3[(i + 5)];
        y[(i + 6)] = y[(i + 6)] + a1 * x1[(i + 6)] + a2 * x2[(i + 6)] + a3 * x3[(i + 6)];
        y[(i + 7)] = y[(i + 7)] + a1 * x1[(i + 7)] + a2 * x2[(i + 7)] + a3 * x3[(i + 7)];
        y[(i + 8)] = y[(i + 8)] + a1 * x1[(i + 8)] + a2 * x2[(i + 8)] + a3 * x3[(i + 8)];
        y[(i + 9)] = y[(i + 9)] + a1 * x1[(i + 9)] + a2 * x2[(i + 9)] + a3 * x3[(i + 9)];
        y[(i + 10)] = y[(i + 10)] + a1 * x1[(i + 10)] + a2 * x2[(i + 10)] + a3 * x3[(i + 10)];
        }
        for (i = N - ((N - (0)) % 11); i <= N - 1; i = i + 1)
        y[i] = y[i] + a1 * x1[i] + a2 * x2[i] + a3 * x3[i];
  }
/*@ end @*/
```

Σε κάθε μία περίπτωση καταγράψαμε το unroll factor που προέκυψε από την εξερεύνηση. Είδαμε ότι διαφέρουν μεταξύ τους, καθώς προσεγγίσαμε το πρόβλημα με διαφορετικό αλγόριθμο.

Algorithm	Unroll factor
Exhaustive	2 και 3
Randomsearch	5 και 15
Simplex	4 και 11

3. Αφού έχει παραχθεί ο μετασχηματισμένος κώδικας για κάθε έναν από τους τρεις τρόπους εξερεύνησης του χώρου σχεδιασμού, αντικαταστήσαμε το αντίστοιχο κομμάτι κώδικα στο αρχείο `tables.c`. Προφανώς επειδή εμείς έχουμε $N = 10^7$ επιλέγουμε τις δεύτερες περιπτώσεις των loops και για τους τρεις αλγορίθμους. Μετρήσαμε τον χρόνο εκτέλεσης τους. Η μέτρηση του χρόνου έγινε με την συνάρτηση gettimeofday με ακρίβεια μικρο-δευτερολέπτων. Εκτελέσαμε τον κώδικα 10 φορές και να μετρήσαμε το μέσο όρο, μέγιστο και ελάχιστο χρόνο εκτέλεσης (time.py). Παρουσιάζουμε τα αποτελέσματα της κάθε μέτρησης.

	Minimum (s)	Average (s)	Maximum (s)
Exhaustive	0.042233	0.043519	0.046476
Randomsearch	0.042779	0.043983	0.048376
Simplex	0.042181	0.043321	0.047977

Με βάση τα αποτελέσματα των παραπάνω μετρήσεων, είναι προφανές ότι και οι τρεις μέθοδοι μείωσαν αρκετά τους χρόνους εκτέλεσης συγκριτικά με τη μη μετασχηματισμένη έκδοσή τους. Κατά μέσο όρο, ο πιο γρήγορος ήταν ο Simplex με τον Exhaustive να ακολουθεί.