Σχεδιασμός Ενσωματωμένων Συστημάτων
9ο εξάμηνο ΣΗΜΜΥ
1η εργαστηριακή άσκηση

Ομάδα 6:
Δούκας Θωμάς Α.Μ.: 03116081
Ψαρράς Ιωάννης Α.Μ.:03116033

Ζητούμενο 1ο – Loop Optimizations & Design Space Exploration

1) Χαρακτηριστικά του υπολογιστή

Μέσω της εντολής ***lsb_release -a*** και ***uname -r*** προσδιορίζουμε την έκδοση του λειτουργικού συστήματος και του πυρήνα που χρησιμοποιείται. Συγκεκριμένα:

Πληροφορίες λειτουργικού συστήματος
Distributor ID: Ubuntu
Description: Ubuntu 20.04.1 LTS
Release: 20.04
Codename: focal

Πληροφορίες πυρήνα Linux
5.4.0-54-generic

Με την εντολή ***lscpu*** παρατηρούμε σχετικά με τους πυρήνες ότι:
CPU(s): 8
CPU MHz: 3194.002

Για την ιεραρχία μνήμης, με την εκτέλεση της εντολής ***free -m*** παρατηρούμε ότι η διαθέσιμη ram είναι 8GB ενώ χρησιμοποιώντας την εντολή ***lscpu*** εμφανίζονται τα παρακάτω αποτελέσματα:
L1d cache: 256 KiB
L1i cache: 512 KiB
L2 cache: 4 MiB
L3 cache: 16 MiB

2) Χρόνος εκτέλεσης *phods_motion_estimation()*

Στο συγκεκριμένο ερώτημα θα μελετήσουμε την λειτουργία του αλγόριθμου εκτίμησης κίνησης PHODS με σκοπό την ελαχιστοποίηση του απαιτούμενου χρόνου εκτέλεσης για το κρίσιμο κομμάτι κώδικα. Προς το σκοπό αυτό, μετασχηματίζουμε κατάλληλα τον δοθέντα κώδικα, ώστε να εμφανίζεται ο χρόνος που χρειάζεται η συνάρτηση phods_motion_estimation. Βασιζόμαστε στη συνάρτηση gettimeofday για τον προσδιορισμό των microsecond ανά εκτέλεση. Παρακάτω φαίνεται ο μετασχηματισμένος κώδικας:

```
160  int main()
161  {
162    int current[N][M], previous[N][M], motion_vectors_x[N/B][M/B],
163        motion_vectors_y[N/B][M/B], i, j;
164    struct timeval start_time, finish_time;
165
166    read_sequence(current,previous);
167
168    gettimeofday(&start_time, NULL);
169    phods_motion_estimation(current,previous,motion_vectors_x,motion_vectors_y);
170    gettimeofday(&finish_time, NULL);
171
172    //Calculate execution time of phods_motion_estimation()
173    printf("%d\n", (int)((finish_time.tv_sec - start_time.tv_sec ) *1000000 + finish_time.tv_usec-start_time.tv_usec) );
174
175    return 0;
176  }
177
```

Για την αυτοματοποίηση της διαδικασίας μεταγλώπτισης, εκτέλεσης και επεξεργασίας των αποτελεσμάτων, υλοποιήσαμε τα scripts Makefile, execute.sh, run.sh, measures.py τα οποία παρουσιάζονται αναλυτικότερα στο Παράρτημα Α.

Έπειτα από την εκτέλεση των παραπάνω (make, ./execute.sh phods) οι μετρήσεις που λάβαμε σε (microseconds) ήταν:

Min time: 6283
Max time: 6835
Median: 6410.0
Average time: ***6499.4***

Οι εντολές εκτέλεσης προκαλούν μεταγλώττιση, εκτέλεση του κώδικα 10 φορές, αποθήκευση αποτελεσμάτων στο αρχείο <code_filename>.out, επεξεργασία δεδομένων μέσω measures.py και εμφάνιση αποτελεσμάτων στο terminal. Ίδια διαδικασία ακολουθείται και για τα επόμενα ερωτήματα.

Ακολουθώντας την διαδικασία αρκετές φορές, διαπιστώσαμε ότι οι παραπάνω τιμές είναι χαρακτηριστικές για το δοθέν πρόγραμμα. Είναι σημαντικό να αναφερθεί ότι δεν επιτυγχάνει κάθε εκτέλεση το ίδιο αποτέλεσμα, καθώς αυτό εξαρτάται και από αστάθμητους παράγοντες όπως η χρονοδρομολόγηση άλλων εργασιών.

3) Μετασχηματισμοί κώδικα

Προκειμένου να μειωθεί ο χρόνος εκτέλεσης της συνάρτησης που αποτελεί το κρίσιμο κομμάτι κώδικα, πραγματοποιήσαμε τους ακόλουθους μετασχηματισμούς. Έπειτα από κάθε μετασχηματισμό εκτελούμε τις εντολές *make*, και ./execute.sh opt_phods.

Αρχικά, παρατηρούμε πως στον αρχικό κώδικα, *phods.c*, οι υπολογισμοί που αφορούν τους άξονες X και Y βρίσκονται σε 2 διαφορετικά loops με ίδια άκρα και δεν υπάρχει καμία εξάρτηση μεταξύ τους. Με βάση την παρατήρηση αυτή, πραγματοποιήσαμε **merge** αυτών των δύο κομματιών, τοποθετώντας όλους τους υπολογισμούς στο ίδιο loop. Η συγκεκριμένη αλλαγή φαίνεται παρακάτω:

```
for(k=0; k<B; k++)
{
  for(l=0; l<B; l++)
  {
    p1 = current[B*x+k][B*y+l];

    if((B*x + vectors_x[x][y] + i + k) < 0 ||
       (B*x + vectors_x[x][y] + i + k) > (N-1) ||
       (B*y + vectors_y[x][y] + l) < 0 ||
       (B*y + vectors_y[x][y] + l) > (M-1))
    {
      p2 = 0;
    } else {
      p2 = previous[B*x+vectors_x[x][y]+i+k][B*y+vectors_y[x][y]+l];
    }

    if((B*x + vectors_x[x][y] + k) <0 ||
       (B*x + vectors_x[x][y] + k) > (N-1) ||
       (B*y + vectors_y[x][y] + i + l) < 0 ||
       (B*y + vectors_y[x][y] + i + l) > (M-1))
    {
      q2 = 0;
    } else {
      q2 = previous[B*x+vectors_x[x][y]+k][B*y+vectors_y[x][y]+i+l];
    }

    distx += abs(p1-p2);
    disty += abs(p1-q2);
  }
}

if(distx < min1)
{
  min1 = distx;
  bestx = i;
}

if(disty < min2)
{
  min2 = disty;
  besty = i;
}
```

Σχετικά με τον μετασχηματισμό αναφέρουμε πως η εντολή *p1=current[B*x+k][B*y+l];* εκτελείται αυτούσια και στις 2 περιπτώσεις και άρα διατηρείται μια φορά στην τελική έκδοση του κώδικα.

Αυτή η αλλαγή βελτίωσε εμφανώς τον χρόνο εκτέλεσης. Παραθέτουμε τα αποτελέσματα της αλλαγής:

Min time: 5555
Max time: 6092
Median: 5621.5
Average time: ***5746.7***

Η συμπεριφορά που φαίνεται στις παραπάνω μετρήσεις είναι αναμενόμενη, καθώς πρακτικά οι επαναλήψεις που αφορούν το πιο απαιτητικό κομμάτι του προγράμματος μειώθηκαν στο μισό.

Στη συνέχεια, παρατηρήσαμε ότι στα στάδια της συνάρτησης *phods_motion_estimation()* υπάρχουν αρκετές τιμές που επαναχρησιμοποιούνται συχνά. Για να αποφευχθεί ο συνεχής υπολογισμός τους, γεγονός που επιβραδύνει την εκτέλεση του προγράμματος, δημιουργήσαμε την μεταβλητή *temp[9].* Με τον τρόπο αυτό, χρονοβόρες πράξεις όπως πολλαπλασιασμοί και διαιρέσεις, εκτελούνται πλέον μόνο μία φορά, κατά την αποθήκευσή τους στον πίνακα.

Η παραπάνω **data reuse** τεχνική μειώνει σημαντικά την απαιτητική σε χρόνο και κόστος ανάγκη για επίσκεψη στην κύρια μνήμη και την εύρεση των τιμών που χρησιμοποιούνται συχνά. Ταυτόχρονα, επιδιώξαμε την αναδιαμόρφωση των απαιτητικών πράξεων αυτών, με υλοποιήσεις πιο συμφέρουσες ως προς το κόστος.

Επιπλέον, αποδείχθηκε ωφέλιμη προς το χρόνο εκτέλεσης η μετακίνηση συγκεκριμένων υπολογισμών σε ανώτερα επίπεδα επαναλήψεων. Οι μετασχηματισμοί αυτοί παρατίθενται ακολούθως

```
//temp = {B*B*255, B*x, B*y, B*x+k, B*y+l, B*x+k+vec_x, B*y+l+vec_y, N/B, M/B}
int temp[9] = {0, 0, 0, 0, 0, 0, 0, N/B, M/B };
temp[0] = B*B;
temp[0] = (temp[0] << 8 ) - temp[0];
```

```
for(x=0; x<temp[7]; x++)
{
  temp[1] = B*x;
  for(y=0; y<temp[8]; y++)
  {
    vectors_x[x][y] = 0;
    vectors_y[x][y] = 0;
    temp[2] = B*y;

    S = 4;
    while(S > 0)
    {
      min1 = temp[0];
      min2 = temp[0];

      for(i=-S; i<S+1; i+=S)
      {
        distx = 0;
        disty = 0;

        /*For all pixels in the block*/
        for(k=0; k<B; k++)
        {
          temp[3] = temp[1] + k;
          temp[5] = vectors_x[x][y] + temp[3];
          for(l=0; l<B; l++)
          {
            //Calculations on X & Y dimension...

          }
        }

        if(distx < min1)
        {
          min1 = distx;
          bestx = i;
        }

        if(disty < min2)
        {
          min2 = disty;
          besty = i;
        }
      }

      S = S>>1;
      vectors_x[x][y] += bestx;
      vectors_y[x][y] += besty;
    }
  }
}
```

Το κέρδος από αυτή την διαδικασία είναι αρκετά σημαντικό, όπως φαίνεται και στις μετρήσεις που ακολουθούν.

Min time: 2240
Max time: 2450
Median: 2295.5
Average time: **2308.7**

Ο μέσος χρόνος εκτέλεσης πέφτει περισσότερο από 50%.

Τέλος, παρατηρήσαμε ότι τα loops *while(S>0){...}* και *for(int i=-S; i<S+1; i+=S){...}* επαναλαμβάνονται για συγκεκριμένες σταθερές τιμές της μεταβλητής S ενώ δεν υπάρχει κάποια άλλη εξάρτηση. Θεωρούμε απαραίτητη την πραγματοποίηση κατάλληλου μετασχηματισμού για την ελαχιστοποίηση του χρόνου εκτέλεσης, μέσω της μείωσης των εντολών ελέγχου βρόχων όπως (iteration condition). Στα πλαίσια αυτά αφαιρούμε το κομμάτι κώδικα

```
S=4
while(S>0){
        …
        for(int i=-S; i<S+1; i+=S){
                …
        }
}
S = S>>1;
```

το οποίο και αντικαθιστούμε με όμοιες ανεξάρτητες μεταξύ τους εκφράσεις.
Επομένως, εφαρμόσαμε σε αυτά **loop unrolling** κάνοντας χρήση των μακροεντολών *S_LOOP* και *ITERATION*. Οι βασικές λειτουργίες της τελικής έκδοσης του προγράμματος, *opt_phods.c* παρατίθενται στη συνέχεια.

```
for(x=0; x<temp[7]; x++){
    temp[1] = B*x;
    for(y=0; y<temp[8]; y++){

        //Initialize the vector motion matrices
        vectors_x[x][y] = 0;
        vectors_y[x][y] = 0;

        // temp[1] = B*x;
        temp[2] = B*y;

        //Repeat for s
        S_LOOP(4);
        S_LOOP(2);
        S_LOOP(1);
        }
    }
```

```
#define S_LOOP(s)                                   \
    min1 = temp[0];                                 \
    min2 = temp[0];                                 \
    ITERATION(-s);                                  \
    ITERATION(0);                                   \
    ITERATION(s);                                   \
    vectors_x[x][y] += bestx;                       \
    vectors_y[x][y] += besty;                       \
```

```
#define ITERATION(i)                                \
    distx = 0;                                      \
    disty = 0;                                      \
                                                    \
    /*For all pixels in the block*/                 \
    for(k=0; k<B; k++){                             \
        temp[3] = temp[1] + k;                      \
        temp[5] = vectors_x[x][y] + temp[3];        \
        for(l=0; l<B; l++){                         \
            /*Calculations on X & Y dimension*/     \
            temp[4] = temp[2] + l;                  \
            temp[6] = vectors_y[x][y] + temp[4];    \
                                                    \
            p1 = current[temp[3]][temp[4]];         \
                                                    \
            if(( temp[5] + i ) < 0 ||               \
                ( temp[5] + i ) > N-1 ||            \
                (temp[6]) < 0 ||                    \
                (temp[6]) > M-1){                   \
                p2 = 0;                             \
            }                                       \
            else                                    \
                p2 = previous[temp[5] + i][temp[6]];\
                                                    \
            if((temp[5]) <0 ||                      \
                (temp[5]) > N-1 ||                  \
                (temp[6] + i) < 0 ||                \
                (temp[6] + i) > M-1){               \
                q2 = 0;                             \
            }                                       \
            else                                    \
                q2 = previous[temp[5]][temp[6] + i];\
                                                    \
            distx += abs(p1-p2);                    \
            disty += abs(p1-q2);                    \
                                                    \
        }                                           \
    }                                               \
                                                    \
        if(distx < min1){                           \
            min1 = distx;                           \
            bestx = i;                              \
        }                                           \
                                                    \
        if(disty < min2){                           \
            min2 = disty;                           \
            besty = i;                              \
        }                                           \
```

Η δραστικότητα του συγκεκριμένου μετασχηματισμού φαίνεται στις παρακάτω μετρήσεις:

Min time: 2059
Max time: 2186
Median: 2095.0
Average time: ***2098.1***

Οι μετασχηματισμοί που πραγματοποιήθηκαν μειώνουν το μέσο χρόνο εκτέλεσης του προγράμματος από 6499.4usec σε 2098.1usec (**-68%**).

4) Design Space Exploration (B block)

Στο παρόν ερώτημα θα πραγματοποιήσουμε έρευνα για τη βέλτιστη τιμή της μεταβλητής του B block. Υποψήφιες τιμές για αυτή αποτελούν οι κοινοί διαιρέτες των διαστάσεων των διαδοχικών εικόνων βίντεο MxN (144x176). Προσδιορίζουμε τα πιθανά Block εκτελώντας την υλοποιήση του πρόγραμματος *divisors.py*, από το οποίο προκύπτουν τα εξής:

Common divisors of 144 176 [1, 2, 4, 8, 16]

Ο κώδικας *opt_phods.c* τροποποιήθηκε κατάλληλα έτσι ώστε να λαμβάνει την τιμή B ως είσοδο.

```c
int main(int argc, char *argv[])
{
    if(argc != 2){
        printf("Usage: ./<compiled_name> <Block size>\n");
        exit(0);
    }
    else{
        B = atoi(argv[1]);

        int current[N][M], previous[N][M], motion_vectors_x[N/B][M/B],
            motion_vectors_y[N/B][M/B], i, j;
        struct timeval start_time, finish_time;

        read_sequence(current,previous);
        gettimeofday(&start_time, NULL);
        phods_motion_estimation(current,previous,motion_vectors_x,motion_vectors_y);
        gettimeofday(&finish_time, NULL);

        //Calculate execution time of phods_motion_estimation()
        printf("%d ", (int)((finish_time.tv_sec - start_time.tv_sec ) *1000000 + finish_time.tv_usec-start_time.tv_usec) );

        return 0;
    }
}
```

Από την αυτοματοποιημένη εκτέλεση (*make, ./execute.sh dse_phods*) προέκυψαν οι εξής μετρήσεις:

B.size	Execution Times									
1	4980	4593	4682	4652	4560	4725	4762	4580	5024	4567
2	3448	2890	2918	2909	2955	3087	3096	2924	2931	2938
4	2467	2459	2403	2394	2420	2318	2485	2516	2404	2391
8	2076	2091	2133	2154	2060	2809	2243	2240	2092	2100
16	2040	2321	2091	2088	2148	2089	2054	2141	2045	2048

Minimum average time = **2106.5 for block size = 16**

Επισημαίνεται πως τα παραπάνω αποτελέσματα είναι κοντά σε επιδόσεις για κάποιες από τις διαφορετικές διαστάσεις. Το γεγονός αυτό σε συνδυασμό με το ότι οι μετρήσεις παρουσιάζουν μικρή διακύμανση οδηγεί στην εμφάνιση διαφορετικού αποτελέσματος σε μελλοντικές εκτελέσεις παρά τη χρήση του ίδιου συστήματος.

5) Design Space Exploration (Bx, By dimensions)

Στο ερώτημα αυτό πραγματοποιήσαμε διερεύνηση για το βέλτιστο blocksize στην περίπτωση που αυτό είναι ορθογώνιο και επομένως οι διαστάσεις του **Bx**, **By** είναι διαφορετικές. Συγκεκριμένα, οι μεταβλητές των διαστάσεων Bx, By θα πρέπει να είναι διαιρέτες των N και M αντίστοιχα. Έπειτα από τροποποίηση του *opt_phods.c* σε *dse_rect_phods.c* για να δέχεται παραμέτρους για τις διαστάσεις, με χρήση του *divisors.py* βρίσκουμε τις παρακάτω υποψήφιες τιμές:

***Bx** (Divisors of 144): [1, 2, 3, 4, 6, 8, 9, 12, 16, 18, 24, 36, 48, 72, 144]*

***By** (Divisors of 176): [1, 2, 4, 8, 11, 16, 22, 44, 88, 176]*

Από την αυτοματοποιημένη εκτέλεση (*make, ./execute.sh dse_rect_phods*) προκύπτουν αποτελέσματα για όλους τους συνδυασμούς διαστάσεων. Για την μικρότερη μέση τιμή προσδιορίζεται ο συνδυασμός :

Minimum average time = **1887.9 for block size = 144x176**

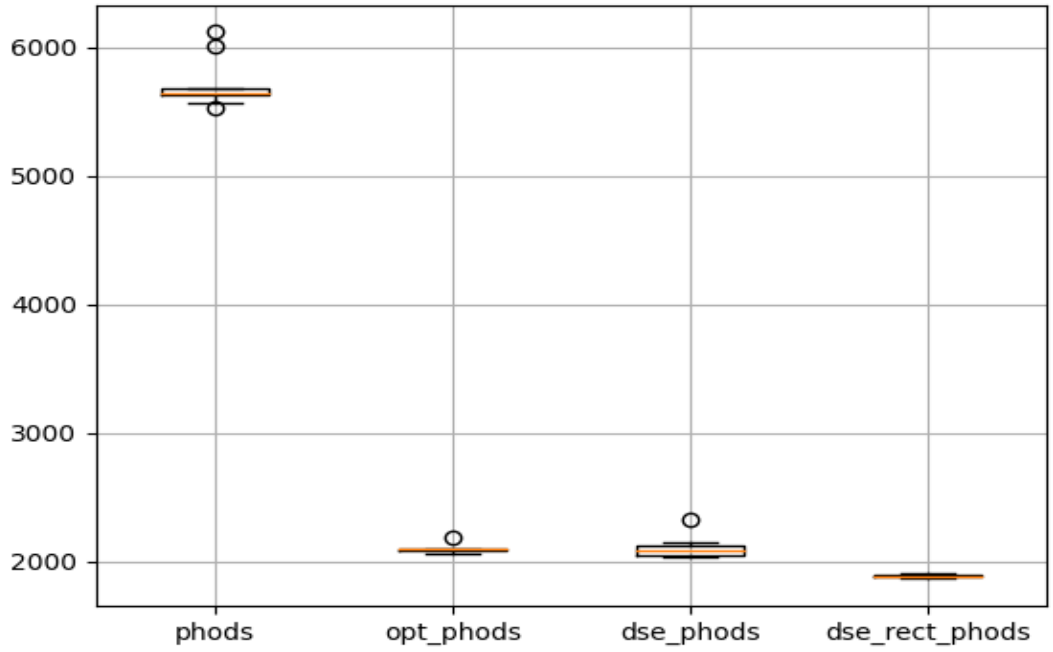
Αναφέρουμε και πάλι πως αρκετά από τα διαφορετικά block sizes προσέγγισαν την παραπάνω τελική τιμή.

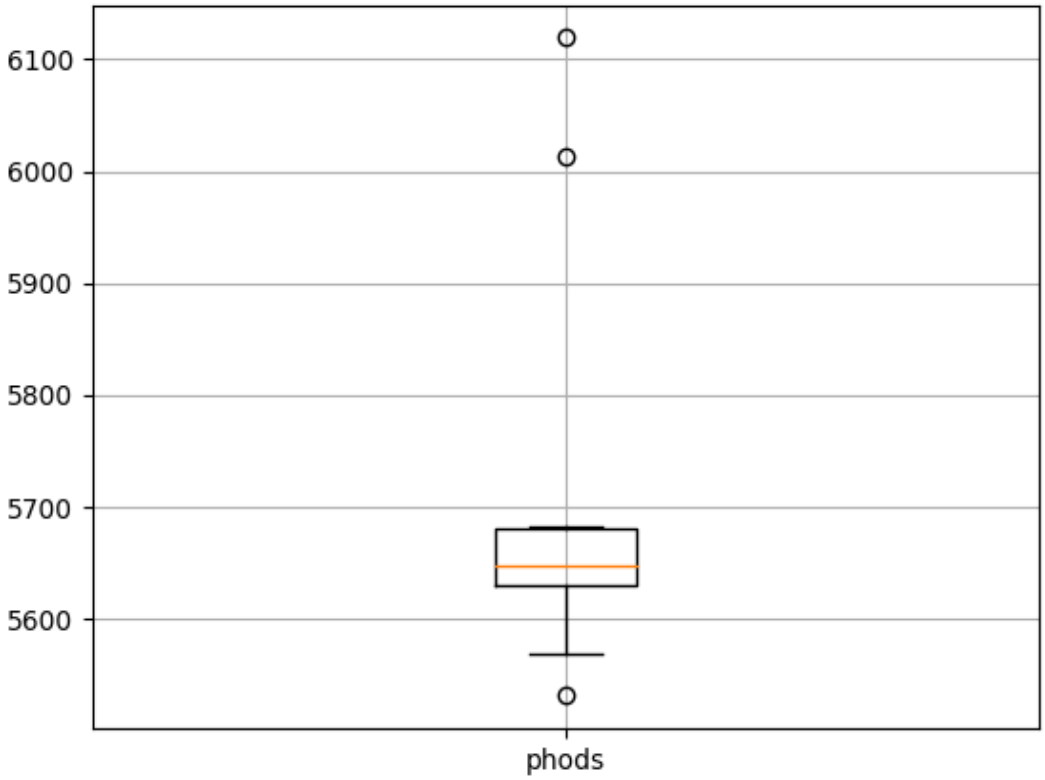
6) Διαγράμματα Boxplot

Για τις μετρήσεις που παρουσιάστηκαν στα ερωτήματα 2,3 αλλά και τα αποτελέσματα που αντιστοιχούν στις καλύτερες διαστάσεις των block δημιουργούμε διαγράμματα boxplot. Με την εκτέλεση της εντολής *python3 boxplot.py* προκύπτουν τα διαγράμματα που παρουσιάζονται παρακάτω.

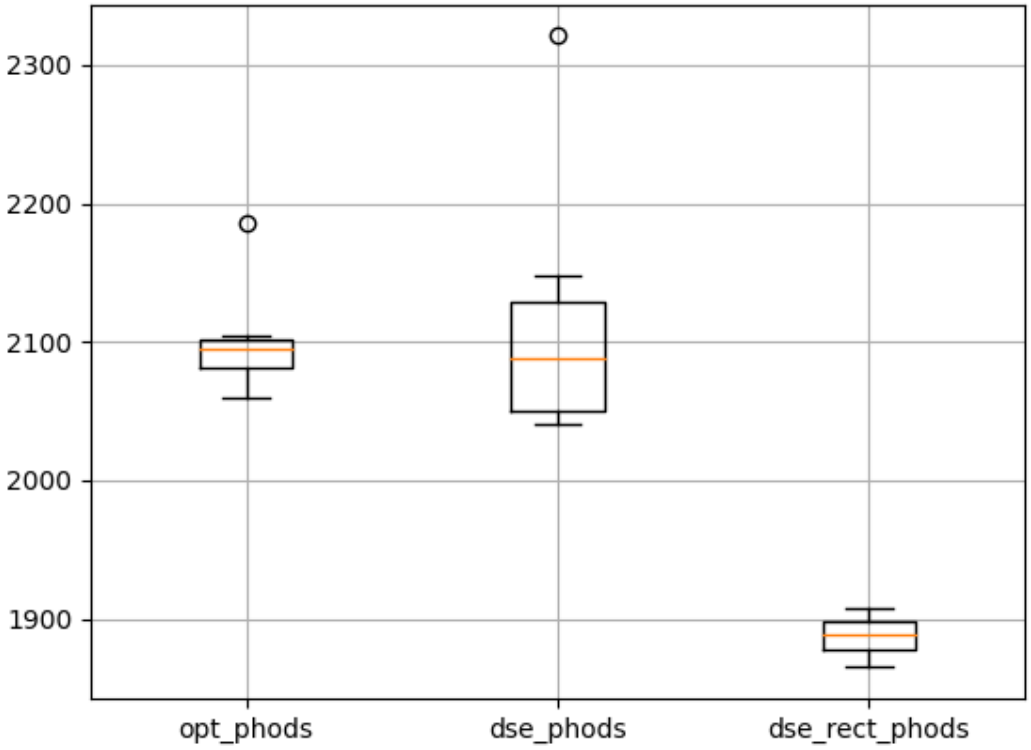
Αρχικά, σχεδιάσαμε στο ίδιο διάγραμμα τα boxplot που αντιστοιχούν στις βέλτιστες μετρήσεις των ερωτημάτων. Ωστόσο, τόσο η ύπαρξη απομακρυσμένων τιμών όσο και οι μεγάλες διαφορές των χρόνων εκτέλεσης για τις διαφορετικές εκδοχές του προγράμματος, καθιστούν το παρακάτω διάγραμμα μη πρακτικό. Το παραθέτουμε στην συνέχεια καθώς οπτικοποιεί τις βελτιώσεις που επιφέρουν στο *phods.c* οι τροποποιήσεις.

Παρουσιάζεται επίσης και το boxplot της αρχικής έκδοσης. Παρατηρούμε εμφανίζονται αρκετές αποκλίνουσες τιμές οι οποίες, όπως αναφέρθηκε οφείλονται πιθανώς στην ύπαρξη επιπλέον διεργασιών στο σύστημα εκτέλεσης του προγράμματος.





Ως εκ τούτου, συνοψίζουμε τα σημαντικότερα αποτελέσματα στο τελικό διάγραμμα. Παρατηρούμε ότι στην περίπτωση του *opt_phods.c* όπου το block size είναι προκαθορισμένο στα 16, επιτυγχάνονται χαμηλές τιμές. Αντίστοιχα, στην βέλτιστη περίπτωση που προκύπτει για τον *dse_phods.c* επιτυγχάνονται εξίσου χαμηλές τιμές. Τα δύο διαγράμματα αναφέρονται στο ίδιο μέγεθος block=16 γεγονός που ενισχύει την ιδιότητα που αναφέραμε παραπάνω περί διακύμανσης των αποτελεσμάτων. Συγκεκριμένα, η εκδοχή *dse_phods* όχι μόνο εμφανίζει μικρότερη τιμή διαμέσου (median) αλλά επιτυγχάνει και καλύτερο ελάχιστο χρόνο εκτέλεσης.



Τέλος παρατηρούμε, πως ο κώδικας *dse_rect_phods.c* που πραγματοποιεί εξαντλητική αναζήτηση των βέλτιστων διαστάσεων του block επιτυγχάνει σημαντικά μικρότερο χρόνο εκτέλεσης. Οι τιμές του συνδυασμού 144x176, ίσου με το μέγεθος της εικόνας, εμφανίζουν σχετικά μικρή διακύμανση ενώ ο μέσος όρος μειώνεται σημαντικά σε σχέση με την αρχική υλοποίηση (**-70.95%**).

Ζητούμενο 2ο – Αυτοματοποιημένη βελτιστοποίηση κώδικα

Στο συγκεκριμένο ζητούμενο θα χρησιμοποιήσουμε το εργαλείο Orio για να πραγματοποιήσουμε αυτόματη βελτιστοποίηση κώδικα. Θα προσπαθήσουμε να εκτελέσουμε ***loop unroll*** στο απαιτητικό κομμάτι του προγράμματος *tables.c* ο οποίος εκτελεί απλές προσπελάσεις και πράξεις με πίνακες.

1) Χρόνος εκτέλεσης tables.c

Με την εντολή *gettimeofday* τροποποιούμε τον κώδικα *tables.c* ώστε να προσδιορίζει τον χρόνο εκτέλεσης, ενώ επαναχρησιμοποιούμε και τα scripts που υλοποιήθηκαν στο ζητούμενο 1 για την αυτοματοποίηση της διαδικασίας.

```
gettimeofday(&start_time, NULL);

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

gettimeofday(&finish_time, NULL);

//Calculate execution time of phods_motion_estimation()
printf("%d\n", (int)((finish_time.tv_sec - start_time.tv_sec ) *1000000 + finish_time.tv_usec-start_time.tv_usec) );
```

Τα αποτελέσματα που προκύπτουν σε microseconds είναι:
Min time: 419819
Max time: 599381
Median: 567550.5
Average time: 551646.2

2) Orio Loop Unrolling

Για την εφαρμογή του μετασχηματισμού ***loop unrolling*** μέσω του εργαλείου Orio πραγματοποιούμε αλλαγές στο αρχείο *tables_orio.c*. Η αυτόματη βελτιστοποίηση θα πραγματοποιηθεί 3 φορές, για τους αλγόριθμους *Simplex*, *Randomsearch* και *Exhaustive*. Για καθέναν από αυτούς χρησιμοποιούμε τις ακόλουθες αρχικοποιήσεις:

```
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;
}
```

```
def search {
        arg algorithm = 'Randomsearch';
        arg time_limit = 10;
        arg total-tuns = 10;
}

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

Εκτελώντας *orcc tables_orio.c* σε κάθε μια από τις περιπτώσεις προκύπτουν:

	Simplex	Random Search	Exhaustive
Unroll Factor	6	15	23

Η διαφορά στα αποτελέσματα οφείλεται σε δύο κύριους παράγοντες. Αρχικά, λόγω της διαφορετικής προσέγγισης κάθε αλγορίθμου, δεν εξασφαλίζεται πως και οι τρεις θα εξετάσουν τα ίδια unroll factors. Επιπλέον, διαφοροποιήσεις όσον αφορά την κατάσταση του συστήματος την στιγμή εκτέλεσης του κάθε αλγορίθμου μπορούν να επηρεάσουν την λήψη μετρήσεων για ίδια unroll factors ανάμεσα στους 3 αλγόριθμους.

3) Σύγκριση χρόνων εκτέλεσης

Για τα αποτελέσματα των αλγορίθμων αναζήτησης του unroll factor αντικαθιστούμε το περιεχόμενο του πίνακα *tables.c* με τα αποτελέσματα που προκύπτουν από το Orio.

Simplex

```
for (int i = 0; i <= N - 6; i = i + 6) {
  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)];
}
for (int i = N - ((N - (0)) % 6); i <= N - 1; i = i + 1)
  y[i] = y[i] + a1 * x1[i] + a2 * x2[i] + a3 * x3[i];
}
```

Randomsearch

```
for (int 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 (int i = N - ((N - (0)) % 15); i <= N - 1; i = i + 1)
  y[i] = y[i] + a1 * x1[i] + a2 * x2[i] + a3 * x3[i];
```

Exhaustive

```
for (int i = 0; i <= N - 23; i = i + 23) {
  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)];
  y[(i + 15)] = y[(i + 15)] + a1 * x1[(i + 15)] + a2 * x2[(i + 15)] + a3 * x3[(i + 15)];
  y[(i + 16)] = y[(i + 16)] + a1 * x1[(i + 16)] + a2 * x2[(i + 16)] + a3 * x3[(i + 16)];
  y[(i + 17)] = y[(i + 17)] + a1 * x1[(i + 17)] + a2 * x2[(i + 17)] + a3 * x3[(i + 17)];
  y[(i + 18)] = y[(i + 18)] + a1 * x1[(i + 18)] + a2 * x2[(i + 18)] + a3 * x3[(i + 18)];
  y[(i + 19)] = y[(i + 19)] + a1 * x1[(i + 19)] + a2 * x2[(i + 19)] + a3 * x3[(i + 19)];
  y[(i + 20)] = y[(i + 20)] + a1 * x1[(i + 20)] + a2 * x2[(i + 20)] + a3 * x3[(i + 20)];
  y[(i + 21)] = y[(i + 21)] + a1 * x1[(i + 21)] + a2 * x2[(i + 21)] + a3 * x3[(i + 21)];
  y[(i + 22)] = y[(i + 22)] + a1 * x1[(i + 22)] + a2 * x2[(i + 22)] + a3 * x3[(i + 22)];
}
for (int i = N - ((N - (0)) % 23); i <= N - 1; i = i + 1)
  y[i] = y[i] + a1 * x1[i] + a2 * x2[i] + a3 * x3[i];
```

Χρησιμοποιώντας το script execute (make, ./execute.sh tables.c) επαναλαμβάνεται η εκτέλεση του κώδικα 10 φορές και πραγματοποιείται επεξεργασία των αποτελεσμάτων όπως και στο ζητούμενο 1. Τα αποτελέσματα εμφανίζονται στη συνέχεια.

	No Unrolling	Simplex	Random Search	Exhaustive
Execution Time (usec)	Min time: 419819 Max time: 599381 Median: 567550.5 Average time: **551646.2**	Min time: 357478 Max time: 413171 Median: 377440.5 Average time: **378535.3**	Min time: 354981 Max time: 374948 Median: 358327.0 Average time: **361295.9**	Min time: 355767 Max time: 383426 Median: 367116.0 Average time: **367369.5**

Όπως είναι αναμενόμενο, παρατηρούμε, ότι και οι 3 αλγόριθμοι εμφανίζουν μικρότερους χρόνους εκτέλεσης από την αρχική υλοποίηση, χωρίς την τεχνική του ***loop unrolling***. Συγκρίνοντας τους υπόλοιπους φαίνεται πως ο Exhaustive εμφανίζει μικρότερο μέσο όρο των χρόνων εκτέλεσης σε σχέση με τον αντίστοιχο του Simplex. Το συγκεκριμένο αποτέλεσμα κρίνεται φυσιολογικό καθώς ο πρώτος πραγματοποιεί εξαντλητική αναζήτηση.

Ωστόσο φαίνεται ο Randomsearch να επιτυγχάνει τα καλύτερα αποτελέσματα. Το γεγονός αυτό, πιθανότατα, οφείλεται στις διακυμάνσεις που παρατηρούνται σε διαφορετικές εκτελέσεις του ίδιου προγράμματος tables.c.

Παράρτημα Α - Παρουσίαση κώδικα για αυτοματοποιημένη εκτέλεση ερωτημάτων

Στο συγκεκριμένο σημείο θα παρουσιάσουμε συνοπτικά τα script που χρησιμοποιήθηκαν ώστε να αυτοματοποιηθεί η εκτέλεση των ερωτημάτων του πρώτου και δεύτερου ζητούμενου. Επειδή τα στάδια υλοποίησης ερωτημάτων επέβαλαν τη σταδιακή τροποποίηση των προγραμμάτων στο παρόν παράρτημα παρατίθενται οι τελικές υλοποιήσεις.
Για ζητούμενο 1:

Makefile

```
all: phods opt_phods dse_phods dse_rect_phods

phods: phods.c
    @gcc phods.c -o phods -O0

opt_phods: opt_phods.c
    @gcc opt_phods.c -o opt_phods -O0

dse_phods: dse_phods.c
    @gcc dse_phods.c -o dse_phods -O0

dse_rect_phods: dse_rect_phods.c
    @gcc dse_rect_phods.c -o dse_rect_phods -O0

clean:
    @rm phods opt_phods dse_phods dse_rect_phods *.out *.err
```

run.sh

```
#!/bin/bash
#Execute code 10 times

if [ -e $1 ] && [ $1 == "phods" ]
then
    for t in {1..10}
    do
        ./$1
    done
elif [ -e $1 ] && [ $1 == "opt_phods" ]
then
    for t in {1..10}
    do
        ./$1
    done
elif [ -e $1 ] && [ $1 == "dse_phods" ]
then
    for B in 1 2 4 8 16
    do
        echo "$B"
        for t in {1..10}
        do
            ./$1 $B
        done
        echo
    done
elif [ -e $1 ] && [ $1 == "dse_rect_phods" ]
then
    for Bx in 1 2 3 4 6 8 9 12 16 18 24 36 48 72 144
    do
        for By in 1 2 4 8 11 16 22 44 88 176
        do
            echo "$Bx $By"
            for t in {1..10}
            do
                ./$1 $Bx $By
            done
            echo
        done
    done
fi
```

execute.sh

```
#!/bin/bash
#Used to specify output files

if [ ! $# -eq 1 ]
then
    echo "Usage: ./execute <executable_code>"
else
    if [ -e $1 ]
    then
        ./run.sh $1 1>$1.out 2>$1.err
        #cat $1.err
        #cat $1.out
        python3 measures.py $1
    else
        echo "File does not exist!"
        echo "Check if code is compiled."
    fi
fi
```

divisors.py > ...

```
import math

# method to print the divisors
def divisors(n) :
    div=[]
    i = 1
    while i <= math.sqrt(n):
        if (n % i == 0) :
            if (n / i == i) :
                div.append(int(i))
            else :
                div.append(int(i))
                div.append(int(n/i))
        i = i + 1
    div.sort()
    print("Divisors of", n, div)

# method to print common divisors
def common_div(a, b):
    div=[]
    for i in range(1, min(a, b)+1):
        if a%i==b%i==0:
            div.append(int(i))
    div.sort()
    print("Common divisors of", a, b, div )

divisors(144)
divisors(176)
common_div(144, 176)
```

boxplot.py > ...

```
import seaborn as sns
import matplotlib.pyplot as plt

#This will be a list of lists of ints
meas = []

#Read data
for filename in ["phods", "opt_phods", "dse_phods", "dse_rect_phods"]:
    file = open("Boxplot/" + filename + ".data")
    #From str to int
    line = list(map(int, file.readline().split()))
    meas.append(line)
    file.close()

# Thank you guy at stack overflow
#Only phods boxplot
fig, ax = plt.subplots()
ax.boxplot(meas[0])
ax.set_xticklabels(["phods"])
plt.grid()
plt.savefig('Images/phods_boxplot.png')

#All runs
fig, ax = plt.subplots()
ax.boxplot(meas)
ax.set_xticklabels(["phods", "opt_phods", "dse_phods", "dse_rect_phods"])
plt.grid()
plt.savefig('Images/all_boxplot.png')

#Readable boxplot
del(meas[0])
fig, ax = plt.subplots()
ax.boxplot(meas)
ax.set_xticklabels(["opt_phods", "dse_phods", "dse_rect_phods"])
plt.grid()
plt.savefig('Images/final_boxplot.png')
```

measures.py > ...

```python
import numpy as np
import sys

def bp_data(array):
    #Boxplot data
    file = open("Boxplot/" + sys.argv[1] + ".data", "w")
    for num in array:
        file.write(str(num) + " ")
    file.close()

if(len(sys.argv) == 2):
    #Open file
    try:
        file = open(sys.argv[1] + ".out")
    except:
        print("Error! File does not exist")
        exit(1)

    if(sys.argv[1]=="phods" or sys.argv[1]=="opt_phods"):
        #Read data
        line = file.readline()

        data = []

        while line:
            data.append(int(line))
            line = file.readline()
        file.close()

        #Convert to np array
        arr = np.array(data)

        #All times are in usec
        print("Min time: ", arr.min())
        print("Max time: ", arr.max())
        print("Median: ", np.median(arr))
        print("Average time: ", np.average(arr))

        bp_data(arr)

    elif(sys.argv[1]=="dse_phods" or sys.argv[1]=="dse_rect_phods"):
        #Read data
        line = file.readline().split()

        index = [] #Block size
        data = [] #Times
        avg_times = []

        while line:
            if (len(line) == 1):
                #Append size for dse_phods
                index.append(int(line[0]))
            elif (len(line) == 2):
                #Append size for dse_opt_phods
                index.append(line[0]+"x"+line[1])
            else:
                #Append times. List of lists of int
                data.append( list(map(int, line)) )
            line = file.readline().split()
        file.close()

        #Calculate avg times for each block
        for block in data:
            avg_times.append(np.average(block))

        #Print minum average and block size
        min_avg_time = min(avg_times)
        block_size = index[avg_times.index(min_avg_time)]
        print("Minimum average time =", min_avg_time, "for block size =", block_size)
        bp_data(block)


else:
    print("Usage: python3 measures.py <measures_file>")
```

Για ζητούμενο 2:

execute.sh

```
#!/bin/bash

if [ ! $# -eq 1 ]
then
    echo "Usage: ./execute <executable_code> <lu-algorithm>"
else
    if [ -e $1 ]
    then
        ./run.sh $1 1>$1.out 2>$1.err
        cat $1.err
        cat $1.out
        python3 measures.py $1
    else
        echo "File does not exist!"
        echo "Check if code is compiled."
    fi
fi
```

Makefile

```
all: tables

tables: tables.c
    @gcc tables.c -o tables -O0

clean:
    @rm tables *.err *.out
```

run.sh

```
#!/bin/bash
#Execute code 10 times

if [ -e $1 ] && [ $1 == "tables" ]
then
    for t in {1..10}
    do
        ./$1
    done
fi
```

measures.py > ...

```
import numpy as np
import sys

if(len(sys.argv) == 2):
    #Open file
    try:
        file = open(sys.argv[1] + ".out")
    except:
        print("Error! File does not exist")
        exit(1)

    if(sys.argv[1]=="tables"):
        #Read data
        line = file.readline()

        data = []

        while line:
            data.append(int(line))
            line = file.readline()
        file.close()

        #Convert to np array
        arr = np.array(data)

        #All times are in usec
        print("Min time: ", arr.min())
        print("Max time: ", arr.max())
        print("Median: ", np.median(arr))
        print("Average time: ", np.average(arr))

else:
    print("Usage: python3 measures.py <measures_file>")
```

Παράρτημα Β - Οδηγίες εκτέλεσης

Για την εκτέλεση των ερωτημάτων του πρώτου ζητούμενου απαιτείται ο χρήστης μπορεί να ακολουθήσει την εξής διαδικασία:

- *make*
- *./execute.sh phods*
- *./execute.sh opt_phods*
- *python3 divisors.py*
- *./execute.sh dse_phods*
- *./execute.sh dse_rect_phods*
- *python3 boxplot.py* (only after phods, opt_phods, dse_phods, dse_rect_phods execution)

Τα αποτελέσματα εμφανίζονται στο τερματικό ενώ τα boxplots αποθηκεύονται στο φάκελο images.

Για επιπλέον διευκόλυνση στη διαδικασία εκτέλεσης υλοποιήσαμε το script run_all.sh η εκτέλεση το οποίου (*./run_all.sh*) αναλαμβάνει την παραπάνω διαδικασία.

run_all.sh

```
#!/bin/bash

./execute.sh phods
./execute.sh opt_phods
./execute.sh dse_phods
./execute.sh dse_rect_phods
python3 boxplot.py
```

Αντίστοιχα για το δεύτερο ζητούμενο :
Για την εκτέλεση ο χρήστης απαιτείται μετά την εγκατάσταση του Orio να εφαρμόσει την κατάλληλη παραμετροποίηση του αρχείου *tables_orio.c*. Στη συνέχεια εκτελώντας *orcc tables_orio.c* τα αποτελέσματα εμφανίζονται στο αρχείο *_tables_orio.c*
Οι μετρήσεις λαμβάνονται και επεξεργάζονται μετά την εκτέλεση των παρακάτω:

- make
- ./execute tables

Η ίδια διαδικασία επαναλαμβάνεται για κάθε αλγόριθμο αναζήτησης.