ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ

ΣΥΣΤΗΜΑΤΑ ΥΠΟΛΟΓΙΣΜΟΥ ΥΨΗΛΩΝ ΕΠΙΔΟΣΕΩΝ (ECE 415)

Ακαδημαϊκο έτος 2021-2022

1η Εργαστηριακή Άσκηση

Φοιτητες:

Ηλιάδης Ηλίας, ΑΕΜ: 2523

Μακρής Δημήτριος Κωσταντίνος, ΑΕΜ: 2787

Στα πλαίσια αυτής της εργασίας κληθήκαμε να υλοποιήσουμε διαδοχικά μια σειρά βελτιστοποιήσεων στον δοσμένο κώδικα ανίχνευσης ακμών του αρχείου **sobel_orig.c**, με στόχο τον καλύτερο δυνατό χρόνο εκτέλεσης του κώδικα και ποσοστό αξιοποίησης των πόρων του συστήματος. Η σειρά υλοποίησης εξαρτήθηκε άμεσα απο το αν η επόμενη «υποψήφια» βελτιστοποίηση ήταν πιο πρόσφορη στην μείωση του χρόνου εκτέλεσης σε σχέση με άλλες βελτιστοποιήσεις. Επίσης παρατηρήθηκε κατα την δίαρκεια υλοποίησης ότι βελτιστοποιήσεις οι οποίες επιβάρυναν αρχικά την επίδοση του προγράμματος αποδείχθηκαν χρήσιμες στην μείωση του χρόνου εκτέλεσης σε μεταγενέστερα στάδια. Παρακάτω σας παραθέτουμε όλες τις βελτιστοποιήσεις με την σειρά που έγιναν , χρησιμοποιώντας το σύστημα με τα εξής χαρακτηριστικά:

Επεξεργαστής: Intel(R) Core(TM) i5-1035G1 CPU @ 1.00GHz

Κύρια μνήμη: 8GiB

'Εκδοση λειτουργικού: 20.04.1 LTS (Focal Fossa)

'Εκδοση πυρήνα: 5.11.0-38-generic

Μεταγλωτιστής: icc (ICC) 2021.4.0 20210910

Βελτιστοποίηση #1: Loop Interchange

Αρχεία: loop_inter.c

Η συγκεκτιμένη βελτιστοποίηση εφαρμόστηκε στα εμφολευμένα for loop των συναρτήσεων **sobel** και **convolution2D.** Η εν λόγω αλλαγή είναι εφαρμόσιμη στην περίπτωσή μας , καθώς δεν υπάρχουν εξαρτήσεις δεδομένων μεταξύ επαναλήψεων και στα δυο εμφολευμένα loops. Με την υλοποίηση αυτή επιτυγχάνουμε βελτίωση της τοπικότητας στην μνήμη cache αλλάζοντας τον τρόπο προπέλασης των στοιχείων των πινάκων **input** και **output**. Συγκεκριμένα στον κώδικα του αρχείου **sobel_orig.c**, οι δύο παραπάνω πίνακες προσπελαύνονταν κατα στήλες (column-major order), αυξάνοντας έτσι το **cache miss rate** σε σχέση με την προσπέλαση κατά γραμμές, στην οποία τα στοιχεία κάθε γραμμής που προσπελαύνονται είναι αποθηκευμένα συνεχόμενα στην μνήμη και ανήκουν στο ίδιο cache block. Παρόλο που δεν έχουμε κάποια εικόνα απο το πως ομαδοποιούνται τα στοιχεία του πίνακα input στην cache του συστήματος, παρατηρούμε και με βάση τις μετρήσεις του πειράματος ότι βελτιστοποιείται η αξιοποίηση της μνήμης όσον αφορά την τοπικότητα.

```
for (i=1; i<SIZE-1; i+=1) {
    for (j=1; j<SIZE-1; j+=1 ) {
        /* Apply the sobel filter and calculate the magnitude *
         * of the derivative.                                 */
        p = pow(convolution2D(i, j, input, horiz_operator), 2) +
            pow(convolution2D(i, j, input, vert_operator), 2);
        res = (int)sqrt(p);
        /* If the resulting value is greater than 255, clip it *
         * to 255.                                             */
        if (res > 255)
            output[i*SIZE + j] = 255;
        else
            output[i*SIZE + j] = (unsigned char)res;
    }
}
```

```
int convolution2D(int posy, int posx, const unsigned char *input, char operator[][3]) {
    int i, j, res;

    res = 0;

    for (i = -1; i <= 1; i++) {
        for (j = -1; j <= 1; j++) {
            res += input[(posy + i)*SIZE + posx + j] * operator[i+1][j+1];
        }
    }
    return(res);
}
```

loop_inter.c			
compiled with -O0		compiled with -fast	
AVG	STDEV	AVG	STDEV
2.076051	0.006639	0.023626	0.000372

Βελτιστοποίηση #2: Function inlining

Αρχεία: func_inline.c

Για την συγκεκριμένη βελτιστοποίηση, αντικαταστάθηκε η κλήση της συνάρτησης **convolution2D** στον κώδικα της συνάρτησης **sobel** με το σώμα της καλούμενης συνάρτησης. Η εν λόγω αλλαγή είναι πρόσφορη στην επίδοση του προγράμματος διότι το μέγεθος της συνάρτησης είναι μικρό με αποτέλεσμα να μην επιβαρύνεται η **instruction cache** καθώς εξαλείφεται και το overhead που προσθέτει η κάθε κλήση της συνάρτησης: τοποθέτηση των καταχωρητών που πρόκειται να επαναχρησιμοποιηθούν απο τον καλών στην στοίβα (stack), τοποθέτηση καταχωρητών ορίσματος, τοποθέτηση διεύθυνσης επιστροφής, επαναφορά καταχωρητών και επιστροφή τιμής και επαναρύθμιση του δείκτη στοίβας κατα την επιστροφή απο την καλούμενη συνάρτηση. Στην δική μας περίπτωση το συνολικό κόστος είναι αρκετά μεγάλο και επηρεάζει αισθητά τον χρόνο εκτέλεσης του

προγράμματος δεδομένου ότι η συνάρτηση καλείται συνολικά 2*((SIZE-2)^2) φορές. Επίσης η επιλογή αυτης της βελτιστοποίησης μας παρέχει την δυνατότητα ενσωμάτωσης βελτιστοποιήσεων οι οποίες δεν θα ήταν αποδοτικές στην περίπτωση όπου οι δυο συναρτήσεις δέν ήταν «ενωμένες». Να σημειωθεί ότι σε αυτό το στάδιο προτιμήσαμε αρχικά να «ξεδιπλώσουμε» το εσωτερικό loop της συνάρτησης convolution2D , πράγμα το οποίο δεν επέφερε κάποια μείωση στον χρόνο εκτέλεσης.

```
for (i=1; i<SIZE-1; i+=1) {
    for (j=1; j<SIZE-1; j+=1 ) {
        /* Apply the sobel filter and calculate the magnitude *
         * of the derivative.                                  */

        a1 = 0;
        a2 = 0;

        for (g = -1; g <= 1; g++) {
            for (k = -1; k <= 1; k++) {
                a1 += input[(i + k)*SIZE + j + g] * horiz_operator[k+1][g+1];
                a2 += input[(i + k)*SIZE + j + g] * vert_operator[k+1][g+1];
            }
        }
        p = pow(a1,2) + pow(a2,2);
        res = (int)sqrt(p);
        /* If the resulting value is greater than 255, clip it *
         * to 255.                                              */
        if (res > 255)
            output[i*SIZE + j] = 255;
        else
            output[i*SIZE + j] = (unsigned char)res;
    }
}
```

func_inline.c			
compiled with -O0		compiled with -fast	
AVG	STDEV	AVG	STDEV
1.919723	0.007497	0.023627	0.000124

Βελτιστοποίηση #3: Loop Unrolling στο σώμα της convolution2D

Αρχεία: unroll_funcLoop.c

Αφου μεταφέραμε το σώμα την συνάρτησης **convolution2D,** επιλέξαμε να ξεδιπλώσουμε το nested loop της εν λόγω συνάρτησης. Με αυτον τον τρόπο μειώνουμε την πολυπλοκότητα που προκύπτει απο την διαχειριστική πληροφορία των for loop καθώς το συγκεκριμένο loop καλείται **2*((SIZE-2)^2)** φορές. Συγκεκριμένα απαλασσόμαστε απο εντολές που διαχειρίζονται το iteration του loop όταν φτάνουμε στο τέλος αυτου, εντολές jump, μειώνονται τα penalty που προκύπτουν απο τον branch predictor όπως και οι προσπελάσεις στην μνήμη. Επίσης έγινε πλήρες loop unrolling καθώς οι επαναλήψεις είναι λίγες με αποτέλεσμα να μην αυξηθεί σημαντικά το μέγεθος του κώδικα ώστε να επιβαρυνθεί η instruction cache.

```
for (i=1; i<SIZE-1; i+=1) {
    for (j=1; j<SIZE-1; j+=1 ) {

        a1 = 0;
        a2 = 0;
        /* Apply the sobel filter and calculate the magnitude *
         * of the derivative.                                 */

        //Function inlining implementation
        a1 += input[(i -1)*SIZE + j -1] * horiz_operator[0][0];
        a1 += input[(i -1)*SIZE + j ] * horiz_operator[0][1];
        a1 += input[(i -1)*SIZE + j + 1] * horiz_operator[0][2];
        a1 += input[i*SIZE + j + -1] * horiz_operator[1][0];
        a1 += input[i*SIZE + j ] * horiz_operator[1][1];
        a1 += input[i*SIZE + j + 1] * horiz_operator[1][2];
        a1 += input[(i + 1)*SIZE + j -1] * horiz_operator[2][0];
        a1 += input[(i + 1)*SIZE + j ] * horiz_operator[2][1];
        a1 += input[(i + 1)*SIZE + j + 1] * horiz_operator[2][2];


        a2 += input[(i -1)*SIZE + j -1] * vert_operator[0][0];
        a2 += input[(i -1)*SIZE + j ] * vert_operator[0][1];
        a2 += input[(i -1)*SIZE + j + 1] * vert_operator[0][2];
        a2 += input[i*SIZE + j -1] * vert_operator[1][0];
        a2 += input[i*SIZE + j ] * vert_operator[1][1];
        a2 += input[i*SIZE + j + 1] * vert_operator[1][2];
        a2 += input[(i + 1)*SIZE + j  -1] * vert_operator[2][0];
        a2 += input[(i + 1)*SIZE + j ] * vert_operator[2][1];
        a2 += input[(i + 1)*SIZE + j + 1] * vert_operator[2][2];

        p = pow(a1,2) + pow(a2,2);
        res = (int)sqrt(p);
        /* If the resulting value is greater than 255, clip it *
         * to 255.                                             */
        if (res > 255)
```

unroll_funcLoop.c			
compiled with **-O0**		compiled with **-fast**	
AVG	STDEV	AVG	STDEV
1.527475	0.002947	0.020677	0.000145

Βελτιστοποίηση #4: Loop Unrolling στους Βρόγχους της συνάρτησης sobel

Αρχεία: sobel_unrolledLoop2.c, sobel_unrolledLoop4.c, sobel_unrolledLoop8.c, sobel_unrolledLoop16.c.

Σε αυτό το σημείο της υλοποίησης θεωρήσαμε λογικό να εφαρμόσουμε την μέθοδο του loop unrolling και στα υπόλοιπα loop της συνάρτησης sobel ούτως ώστε να αυξηθεί η επίδοση του προγράμματος. Ο κώδικας που εφαρμόζει το «ξεδίπλωμα» των βρόγχων είναι γραμμένος με τέτοιο τρόπο ώστε να εκτελούνται όλες οι επαναλήψεις, ακόμα και αυτές που περισσεύουν σε περίπτωση που ο συνολικός αριθμός επαναλήψεων δεν διαιρείται ακριβως με τον βαθμό ξεδιπλώματος που έχουμε επιλέξει (παρακάτω παρατίθεται η εικόνα του κώδικα). Πειραματατιστήκαμε στο μερικό ξεδίπλωμα των βρόγχων με βαθμούς ξεδιπλώματος 2, 4, 8, και 16. Συνολικά καταφέραμε να ελαχιστοποιήσουμε τις επαναλήψεις των for loop όπως και το συνολικό κόστος διαχείρισης και υπολογισμού που συνεπάγονται απο τα προηγούμενα. Πειραματικά με βάση τις μετρήσεις αποδείχθηκε ότι η περίπτωση ξεδιπλώματος 4ου βαθμού παράγει τον ταχύτερο κώδικα απο τις υπόλοιπες περιπτώσεις. Αυτό μας οδήγησε στο συμπέρασμα πως με "μικρό" unrolling υποβοηθούμε τον μεταγλωττιστή να τρέχει ο ίδιος μικροαλλαγές σε εντολές load/store (καθώς έχουμε διαπέραση πίνακα, κι έτσι αυξήσεις στην τιμή που δείχνει ο register), επιταχύνοντας έτσι το τρέξιμο του κώδικα. Απο την άλλη, το ξεδίπλωμα 16ου βαθμού χειροτερεύει την επίδοση σε σχέση με την Βελτιστοποίηση #3, πράγμα που ενδεχομένως οφείλεται στην σημαντική αύξηση του κώδικα με αποτέλεσμα να αυξάνονται τα instruction cache misses και κατα συνέπεια και το run time.

```
for (i=1; i<SIZE-1; i+=1) {
    for (j=1 , k = 1; k <=((SIZE-2)/2);k++, j+=2 ) {
```

Εισαγωγή βαθμού unrolling

```
for(k = 0; k < (SIZE-2) % 2 ; k++, j++ )
```

Διάτρεξη περισσευόμενων επαναλήψεων

```
for (i=1; i<SIZE-1; i++) {
    for (j=1 , k = 1; k <=((SIZE-2)/2);k++, j+=2 ) {
        t = pow((output[i*SIZE+j] - golden[i*SIZE+j]),2);
        PSNR += t;
        t = pow((output[i*SIZE+j+1] - golden[i*SIZE+j+1]),2);
        PSNR += t;
    }
    for(k = 0; k < (SIZE-2) % 2 ; k++, j++ ) {
        t = pow((output[i*SIZE+j] - golden[i*SIZE+j]),2);
        PSNR += t;
    }
}

PSNR /= (double)(SIZE*SIZE);
PSNR = 10*log10(65536/PSNR);
```

sobel_unrolledLoop2.c			
compiled with -O0		compiled with -fast	
AVG	STDEV	AVG	STDEV
1.521347	0.003942	0.020442	0.000306

sobel_unrolledLoop4.c			
compiled with -O0		compiled with -fast	
AVG	STDEV	AVG	STDEV
1.517405	0.002776	0.051451	0.000384

sobel_unrolledLoop8.c			
compiled with -O0		compiled with -fast	
AVG	STDEV	AVG	STDEV
1.524263	0.005822	0.046007	0.000347

sobel_unrolledLoop16.c			
compiled with -O0		compiled with -fast	
AVG	STDEV	AVG	STDEV
1.546367	0.006694	0.04455	0.000233

Βελτιστοποίηση #5: Strengh Reduction (1) - Αντικατάσταση της συνάρτησης pow με απλό γινόμενο του ορίσματος με τον εαυτό του

Αρχεία: noPow.c

Παρατηρήσαμε (ίσως και πρόωρα, σε ότι έχει να κάνει με strength reduction) ότι γίνεται επανειλημμένη χρήση της συνάρτησης **pow(double, double)** μέσα στα δύο nested loops της συνάρτησης sobel. Θεωρήσαμε αποδοτικό να αντικαταστήσουμε την κλήση της συνάρτησης με έναν απλό πολλαπλασιασμό του ορίσματος με τον εαυτό του και τελικά αποδείχθηκε με βάση τις μετρήσεις ότι η αλλαγή αυτη επιφέρει πολύ σημαντική αύξηση στην επλιδοση του προγράμματος μας. Αυτό συμβαίνει καθώς η συνάρτηση pow πρέπει να εφαρμόσει ένα γενικό αλγόριθμο ο οποίος πρέπει να λειτουργεί σε όλες τις περιπτώσεις και να είναι σε θέση να μπορεί να ανυψώσει οποιονδήποτε εκθέτη που μπορεί να αναπαρασταθεί απο έναν double αριθμό, ενώ στην περίπτωση του απλού γινομένου ο επεξεργαστής θα έρθει αντιμέτωπος με 1 – 2 εντολές assembly.

noPow.c			
compiled with -O0		compiled with -fast	
AVG	STDEV	AVG	STDEV
0.507798	0.003603	0.018743	0.000319

Βελτιστοποίηση #6: Loop fusion μεταξύ των δύο nested loop της συνάρτητης sobel

Αρχεία: fusion.c

Σε προηγούμενη βελτιστοποίηση εφαρμόσαμε «ξεδίπλωμα» και στα δύο εσωτερικά loop των δυο nested loop της συνάρτησης sobel με τον ίδιο βαθμό, έχοντας και τον ίδιο αριθμο επαναλήψεων. Επίσης δεν υπάρχουν εξαρτήσεις δεδομένων μεταξύ των δύο loop καθώς ο υπολογισμός του **PSNR** γίνεται αφότου ανατεθούν όλοι οι υπολογισμοί του **μέσου τετραγωνικού σφάλματος** που παράγονται στις επαναλήψεις του πρώτου nested loop στον πίνακα **output,** τα δεδομένα του οποίου χρησιμοποιούνται για τον υπολογισμό του PSNR. Με βάση τα παραπάνω μπορούμε να εφαρμόσουμε loop fusion μεταξυ των δύο βρόγχων της συνάρτησης sobel. Η αλλαγή αυτή είναι πρόσφορη καθώς ενώνοντας τα loop μειώνεται η συνολική διαχειριστική πολυπλοκότητα και δημιουργεί ανάχωμα για μεταγενέστερες βελτιστοποιήσεις όπως strength reduction, common subexpression elimination κτλ. Τέλος παρατηρήθηκε πειραματικά με βάση τις μετρήσεις ότι η εφαργμογή του loop fusion είναι πιο αποδοτική στον κώδικα του αρχείου **sobel_unrolledLoop2.c** , ενώ στις περιπτώσεις ξεδιπλώματος μεγαλύτερου βαθμου «έριχνε» την επίδοση. Το τελευταιο ενδεχομένος συμβάνει λόγω αυξημένου data locality.

fusion.c			
compiled with **-O0**		compiled with **-fast**	
AVG	STDEV	AVG	STDEV
0.49704	0.003372	0.029183	0.000471

```
    p = a1*a1 + a2*a2;
    res = (int)sqrt(p);
    /* If the resulting value is greater than 255, clip it *
     * to 255.
                                                          */
    if (res > 255)
        output[i*SIZE + j + 1] = 255;
    else
        output[i*SIZE + j + 1] = (unsigned char)res;

    //LOOP FUSION
    //Now run through the output and the golden output to calculate the MSE
    t = (output[i*SIZE+j] - golden[i*SIZE+j])*(output[i*SIZE+j] - golden[i*SIZE+j]);
    PSNR += t;
    t = (output[i*SIZE+j+1] - golden[i*SIZE+j+1])*(output[i*SIZE+j+1] - golden[i*SIZE+j+1]);
    PSNR += t;

}
```

Βελτιστοποίηση #7: Loop invariant code motion

Αρχεία: loopInvariant.c

Στη συνέχεια, μετακινήσαμε τμήματα κώδικα εκτός των loops, (γινόμενα του μετρητή i με το SIZE), καθώς αυτές οι πράξεις είναι ανεξάρτητες των επαναλήψεων με τον μετρητή j, κι έτσι καταφέραμε μείωση του χρόνου κατά 0.06 δευτερολέπτων, καθώς ο compiler δεν υποχρεώνεται να επαναλάβει πράξεις για περισσότερες φορές, εκμεταλλευόμενος προϋπάρχοντα δεδομένα πιο άμεσα.

loopInvariant.c			
compiled with -O0		compiled with -fast	
AVG	STDEV	AVG	STDEV
0.436876	0.0032	0.029083	0.000154

Βελτιστοποίηση #8: Common Subexpression Elimination

Αρχεία: subEx_elimination.c

Έπειτα, πολλές πράξεις που επαναλαμβάνονταν, όπως προσθέσεις των μεταβλητών s1, s2, s3 με τον μετρητή j, αποφασίσαμε να εξαλειφθούν από το loop, πραγματοποιώντας έτσι καλύτερη διαχείριση των registers και μειώνοντας τον αριθμό των προσθέσεων, με αποτέλεσμα τη μείωση του χρόνου εκτέλεσης κατά 0.04 δευτερολέπτων.

subEx_elimination.c			
compiled with -O0		compiled with -fast	
AVG	STDEV	AVG	STDEV
0.394558	0.005069	0.028934	0.00019

Βελτιστοποίηση #9: Replacing values from operating arrays

Αρχεία: replace_operator.c

Ένα ακόμα βήμα στην βελτιστοποίηση του κώδικα ήταν η αυτούσια χρήση των τιμών που περιέχονται στους πίνακες horiz_operator και vert_operator, εκμεταλλευόμενοι έτσι την αποφυγή προσπέλασης πινάκων για κάθε πράξη μέσα στο loop, αλλά επικέντρωση στις πράξεις καθαυτές. Συνεπώς, ο χρόνος μειώθηκε κατά 0.15 δευτερόλεπτα, μέγεθος αρκετά σημαντικό.

replace_operator.c			
compiled with -O0		compiled with -fast	
AVG	STDEV	AVG	STDEV
0.240354	0.001428	0.028529	0.000317

Βελτιστοποίηση #10: Strength Reduction(2)-Replacing multiplication by left shift

Αρχεία: strength_reduction.c

Σε μία περαιτέρω προσπάθεια να εξαλείψουμε εναπομείναντες "ακριβές" πράξεις, αντικαταστήσαμε τα γινόμενα με <<, καθώς σαν πράξεις τα shift είναι πιο "οικονομικά". Έτσι, επιτύχαμε μία ελαχιστοποίηση του χρόνου κατά 0.02 δευτερόλεπτα κατά την εκτέλεση.

strength_reduction.c			
compiled with -O0		compiled with -fast	
AVG	STDEV	AVG	STDEV
0.225782	0.002946	0.029102	0.000291

Βελτιστοποίηση #11: Data alignment

Αρχεία: data_alignment.c

Τέλος, παρατηρήσαμε πως οι πράξεις με τους operators πραγματοποιούνταν κατά στήλες, κι όχι κατά γραμμές, όπως θα ήταν καλύτερο, συνεπώς εφόσον η αλλαγή αυτών δεν επηρέαζε το τελικό αποτέλεσμα εκμεταλλευτήκαμε την χωρική τοπικότητα πραγματοποιώντας πράξεις γραμμικά, κάτι που επέφερε μία ακόμη μείωση του χρόνου εκτέλεσης, σε κλάση 0.002 δευτερολέπτων.

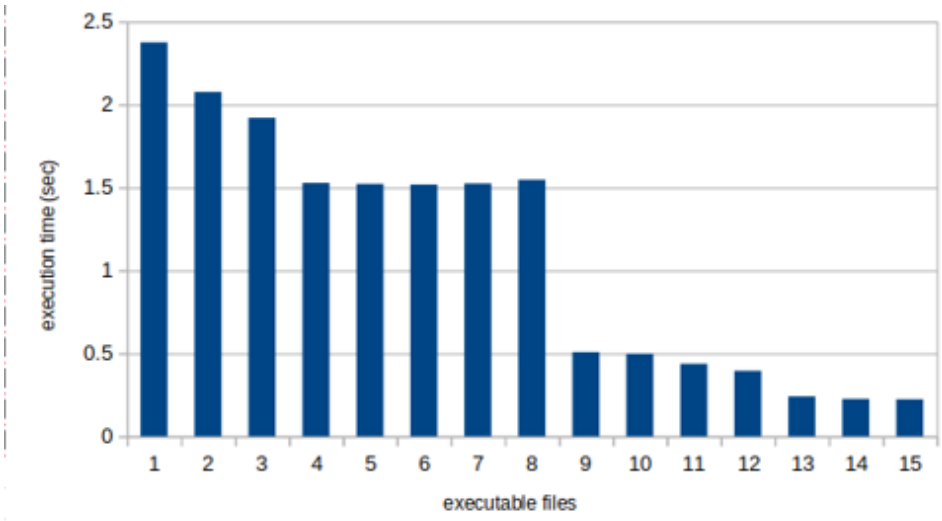
data_alignment.c			
compiled with -O0		compiled with -fast	
AVG	STDEV	AVG	STDEV
0.223368	0.001788	0.028065	0.000356

Γραφική αναπαράσταση πειραματικών αποτελεσμάτων

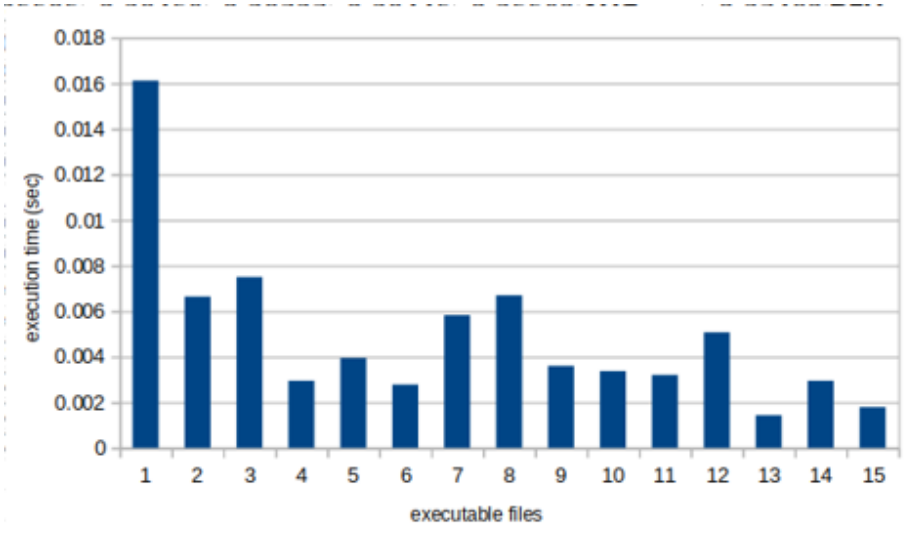
Παρακάτω παραθέτουμε τα διαγράμματα των μετρήσεων που λάβαμε απο τα πειράματα. Η σειρά με την οποία αναπαριστώνται οι χρόνοι αντιστοιχούν με την σειρά των εφαρμογών που παρουσιάζονται στα excel στην καθεμία απο τις οποίες εφαρμόστηκαν διαδοχικά οι βελτιστοποιήσεις που παρουσιάσαμε παραπάνω.

1.sobel_orig
2. loop_inter
3. func_inline
4. unroll_funcLoop
5. sobel_unrolledLoop2
6. sobel_unrolledLoop4
7. sobel_unrolledLoop8
8.sobel_unrolledLoop16
9. noPow
10. fusion
11. loopInvariant
12. subex_Elimination
13. replace_operator
14. strength_reduction
15. data_alignment

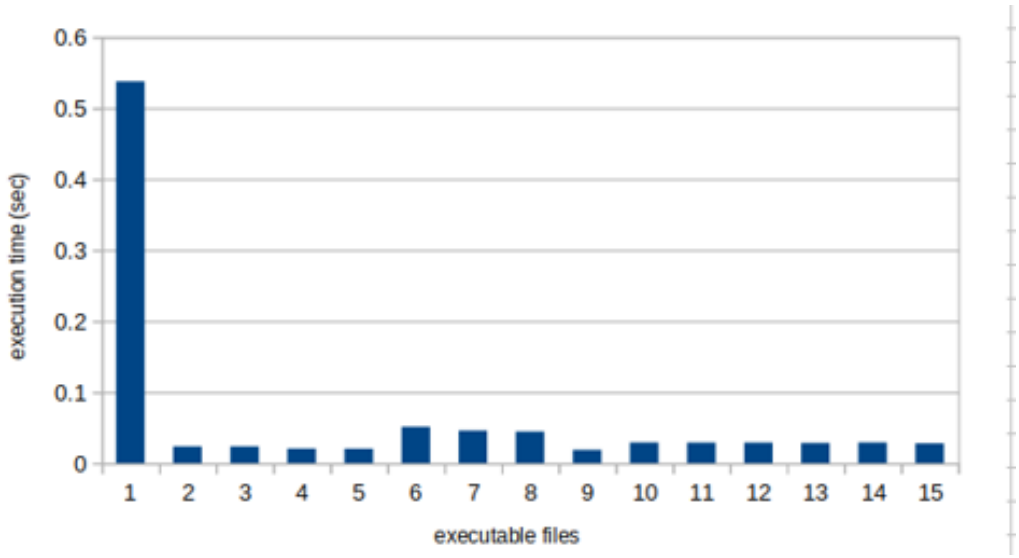
A) Μέσος χρόνος εκτέλεσης εφαρμογής με compiler flag -O0



Β) Τυπική απόκλιση χρόνων εκτέλεσης εφαρμογής με compiler flag -O0



Γ) Μέσος χρόνος εκτέλεσης εφαρμογής με compiler flag -fast



Δ) Τυπική απόκλιση χρόνων εκτέλεσης εφαρμογής με compiler flag -fast