ΤΜΉΜΑ ΗΛΕΚΤΡΟΛΌΓΩΝ ΜΗΧΑΝΙΚΏΝ ΚΑΙ ΜΗΧΑΝΙΚΏΝ ΥΠΟΛΟΓΙΣΤΏΝ

ΠΑΝΕΠΙΣΤΉΜΙΟ ΘΕΣΣΑΛΊΑΣ

ΣΥΣΤΉΜΑΤΑ ΥΠΟΛΟΓΙΣΜΟΎ ΥΨΗΛΏΝ ΕΠΙΔΌΣΕΩΝ

Ηλιάδης Γρηγόρης, ΑΕΜ: 2522

Πουρναρόπουλος Φοίβος, ΑΕΜ: 2614

Σκοπός της συγκεκριμένης εργασίας είναι η εφαρμογή διαδοχικών βελτιστοποιήσεων στον αρχικό κώδικα sobel_orig.c, οι οποίες θα επιφέρουν μείωση στο συνολικό χρόνο εκτέλεσης.

Η δοκιμή όλων των πιθανών συνδυασμών έτσι ώστε να βρούμε ποια αλληλουχία βελτιστοποιήσεων παράγει τον ταχύτερο κώδικα είναι πρακτικά αδύνατη, εφόσον οι συνδυασμοί είναι πολλοί. Συνεπώς, σε κάθε βήμα επιλέγουμε βελτιστοποιήσεις οι οποίες είναι πιο πρόσφορες σχετικά με την επίδοση και ταυτόχρονα επιτρέπουν περαιτέρω βελτιστοποιήσεις. Παρακάτω, παρουσιάζονται και επεξηγούνται οι βελτιστοποιήσεις με τη σειρά που έγιναν.

Χαρακτηριστικά συστήματος:

Επεξεργαστής: Intel(R) Core(TM) i7-7500U CPU @ 2.70GHz

Μνήμη RAM: 16GB

Έκδοση λειτουργικού: 16.04.3 LTS (Xenial Xerus)

Έκδοση πυρήνα: 4.15.0-122-generic

Μεταγλωττιστής: icc (ICC) 19.1.2.254 20200623

1: Loop interchange, αρχεία: sobel1_loop_interchange.c

Πιο συγκεκριμένα, εναλλάσσουμε το εσωτερικό με το εξωτερικό loop των nested loops στη συνάρτηση convolution2D και στη συνάρτηση sobel. Η εναλλαγή αυτή είναι επιτρεπτή στην περίπτωση μας, καθώς δεν υπάρχουν εξαρτήσεις δεδομένων μεταξύ διαφορετικών επαναλήψεων. Με αυτό τον τρόπο εκμεταλλευόμαστε καλύτερα την τοπικότητα των δεδομένων στη μνήμη πετυχαίνοντας βελτίωση του χρόνου προσπέλασης σε αυτήν. Γνωρίζουμε ότι οι πίνακες αποθηκεύονται στη μνήμη κατά γραμμές. Στον αρχικό κώδικα, η προσπέλαση των πινάκων input και output γίνεται κατά στήλες, κάτι που προκαλεί πολλά περισσότερα cache misses σε σχέση με την προσπέλαση κατά γραμμές. Δεν γνωρίζουμε επακριβώς ποιο τμήμα της εικόνας(πίνακας input) μπαίνει στην cache σε κάθε επανάληψη. Στην συνάρτηση convolution2D, όπου οι προσπελάσεις στη μνήμη αφορούν αποκλειστικά τον πίνακα input, εκμεταλλευόμαστε πλήρως την τοπικότητα των δεδομένων. Όμως, παρόλο που το πρότυπο προσπέλασης των πινάκων στη συνάρτηση sobel

“χαλάει” την τοπικότητα (input μέσω της συνάρτησης convolution2D, output, input μέσω της συνάρτησης convolution2D , output,...) , το συγκεκριμένο loop interchange εξακολουθεί να αυξάνει σε κάθε περίπτωση την πιθανότητα να έχουμε cache hits.

```
40  int convolution2D(int posy, int posx, const unsigned char *input, char operator[][3]) {
41      int i, j, res;
42
43      res = 0;
44      for (j = -1; j <= 1; j++) {
45          for (i = -1; i <= 1; i++) {
46              res += input[(posy + i)*SIZE + posx + j] * operator[i+1][j+1];
47          }
48      }
49      return(res);
50  }
```

```
107     for (j=1; j<SIZE-1; j+=1) {
108         for (i=1; i<SIZE-1; i+=1 ) {
109             /* Apply the sobel filter and calculate the magnitude *
110              * of the derivative.                                 */
111             p = pow(convolution2D(i, j, input, horiz_operator), 2) +
112                 pow(convolution2D(i, j, input, vert_operator), 2);
113             res = (int)sqrt(p);
114             /* If the resulting value is greater than 255, clip it *
115              * to 255.                                            */
116             if (res > 255)
117                 output[i*SIZE + j] = 255;
118             else
119                 output[i*SIZE + j] = (unsigned char)res;
120         }
121     }
```

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

    res = 0;
    //(1) allagh to j me to i
    //kai ston pinaka input 1 grammh ana loop anti gia 3( o pinakas apo8hkeyetai kata grammes)
    //kai ston pinaka operator an kai amelhtaio
    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);
}
```

```
111     //(1) allagh i j beltiwsh ston pinaka output
112     for (i=1; i<SIZE-1; i+=1) {
113         for (j=1; j<SIZE-1; j+=1 ) {
114             /* Apply the sobel filter and calculate the magnitude *
115              * of the derivative.                                 */
116             p = pow(convolution2D(i, j, input, horiz_operator), 2) +
117                 pow(convolution2D(i, j, input, vert_operator), 2);
118             res = (int)sqrt(p);
119             /* If the resulting value is greater than 255, clip it *
120              * to 255.
121                                                                    */
122             if (res > 255)
123                 output[i*SIZE + j] = 255;
124             else
125                 output[i*SIZE + j] = (unsigned char)res;
126         }
127     }
```

Επεξήγηση sobel1_test: Στο αρχείο αυτό, εφαρμόσαμε μόνο 1 loop interchange το οποίο έγινε στη συνάρτηση convolution2D.Με βάση τις μετρήσεις παρατηρούμε ότι αυτή η αλλαγή παράγει πιο αργό κώδικα σε σχέση με τον αρχικό. Αυτό οφείλεται στο γεγονός ότι “χαλάει” το πρότυπο προσπέλασης στη μνήμη με αποτέλεσμα να αυξηθούν τα cache misses.

(-O0)

Αρχείο: sobel1_loop_interchange.c

Μέσος χρόνος εκτέλεσης: 2.264459 sec

Τυπική απόκλιση: 0.017053 sec

sobel1_test.c:

Μέσος χρόνος εκτέλεσης: 3.421974 sec

Τυπική απόκλιση: 0.007739 sec

Μείωση μέσου χρόνου εκτέλεσης σε σχέση με το προηγούμενο στάδιο: 33%

(-fast)

Αρχείο: sobel1_loop_interchange.c

Μέσος χρόνος εκτέλεσης: 0.053298 sec

Τυπική απόκλιση: 0.002009 sec

sobel1_test.c:

Μέσος χρόνος εκτέλεσης: 0.587794 sec

Τυπική απόκλιση: 0.006450 sec

2: Loop unroll στην συνάρτηση convolution2D, αρχείο: sobel2_unroll.c

Για μέγεθος εικόνας SIZE, η συνάρτηση convolution2D καλείται 2*((SIZE-2)^2) φορές. Συνεπώς, για μεγάλες τιμές του SIZE, όπως συμβαίνει στις περισσότερες εικόνες, μειώνεται κατά πολύ το πλήθος των διαχειριστικών εντολών που αφορούν το loop. Πιο συγκεκριμένα, μειώνονται οι αυξήσεις μετρητών, οι έλεγχοι και τα άλματα, γεγονός που εξοικονομεί πολλούς κύκλους μηχανής. Επίσης, μειώνεται το κόστος των πράξεων που απαιτούνται για τον υπολογισμό της εκάστοτε θέσης του πίνακα που θέλουμε να προσπελάσουμε. Επειδή στη συγκεκριμένη περίπτωση ο συνολικός αριθμός των επαναλήψεων είναι μικρός και γνωστός και κατά συνέπεια δεν αυξάνουμε τόσο το μέγεθος του εκτελέσιμου ώστε να αυξηθούν δυνητικά τα instruction cache misses, επιλέξαμε πλήρες loop unrolling, για να μην έχουμε καμία από τις παραπάνω επιβαρύνσεις.

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

    res = 0;
    //(1) allagh to j me to i
    //kai ston pinaka input 1 grammh ana loop anti gia 3( o pinakas apo8hkeyetai kata grammes)
    //kai ston pinaka operator an kai amelhtaio

    //2plo loop unroll
    //(2) loop unroll
    res += input[(posy -1)*SIZE + posx + -1] * operator[0][0];
    res += input[(posy -1)*SIZE + posx ] * operator[0][1];
    res += input[(posy -1)*SIZE + posx + 1] * operator[0][2];
    res += input[(posy)*SIZE + posx + -1] * operator[1][0];
    res += input[(posy)*SIZE + posx ] * operator[1][1];
    res += input[(posy)*SIZE + posx + 1] * operator[1][2];
    res += input[(posy + 1)*SIZE + posx + -1] * operator[2][0];
    res += input[(posy + 1)*SIZE + posx ] * operator[2][1];
    res += input[(posy + 1)*SIZE + posx + 1] * operator[2][2];

    return(res);
}
```

(-O0)

Μέσος χρόνος εκτέλεσης: 1.788373 sec

Τυπική απόκλιση: 0.015977 sec

Μείωση μέσου χρόνου εκτέλεσης σε σχέση με το προηγούμενο στάδιο: 21%

(-fast)

Μέσος χρόνος εκτέλεσης: 0.045832 sec

Τυπική απόκλιση: 0.001202 sec

Δίνουμε τη δυνατότητα στο μεταγλωττιστή να αναδιοργανώσει πιο αποτελεσματικά τη σειρά εκτέλεσης των εντολών.

```
119     //(1) allagh i j beltiwsh ston pinaka output
120     for (i=1; i<SIZE-1; i+=1) {
121         for (j=1; j<SIZE-1; j+=1 ) {
122             resx = 0;
123             resy = 0;
124             /* Apply the sobel filter and calculate the magnitude *
125              * of the derivative.                                 */
126             resx += input[(i -1)*SIZE + j + -1] * horiz_operator[0][0];
127             resx += input[(i -1)*SIZE + j ] * horiz_operator[0][1];
128             resx += input[(i -1)*SIZE + j + 1] * horiz_operator[0][2];
129             resx += input[(i)*SIZE + j + -1] * horiz_operator[1][0];
130             resx += input[(i)*SIZE + j ] * horiz_operator[1][1];
131             resx += input[(i)*SIZE + j + 1] * horiz_operator[1][2];
132             resx += input[(i + 1)*SIZE + j + -1] * horiz_operator[2][0];
133             resx += input[(i + 1)*SIZE + j ] * horiz_operator[2][1];
134             resx += input[(i + 1)*SIZE + j + 1] * horiz_operator[2][2];
135 
136             resy += input[(i -1)*SIZE + j + -1] * vert_operator[0][0];
137             resy += input[(i -1)*SIZE + j ] * vert_operator[0][1];
138             resy += input[(i -1)*SIZE + j + 1] * vert_operator[0][2];
139             resy += input[(i)*SIZE + j + -1] * vert_operator[1][0];
140             resy += input[(i)*SIZE + j ] * vert_operator[1][1];
141             resy += input[(i)*SIZE + j + 1] * vert_operator[1][2];
142             resy += input[(i + 1)*SIZE + j + -1] * vert_operator[2][0];
143             resy += input[(i + 1)*SIZE + j ] * vert_operator[2][1];
144             resy += input[(i + 1)*SIZE + j + 1] * vert_operator[2][2];
145 
146             p = pow(resx, 2) + pow(resy, 2);
147             res = (int)sqrt(p);
```

3: Function inlining της συνάρτησης convolution2D στην sobel, αρχείο: sobel3_inline.c

Η συγκεκριμένη βελτιστοποίηση έχει ως αποτέλεσμα τη μείωση των κλήσεων της συνάρτησης convolution2D, οι οποίες είναι αρκετά ακριβές. Για την κλήση μιας συνάρτησης πρέπει να αποθηκευτούν registers στην stack, να γίνουν push τα arguments, αλλαγή του instruction pointer, αποθήκευση του return address κλπ. Ειδικά στην περίπτωση μας, που η συνάρτηση convolution2D είναι μικρή σε μέγεθος και καλείται πολλές φορές, η επιβάρυνση είναι ακόμα μεγαλύτερη, διότι το κόστος της κλήσης είναι συγκρίσιμο με το κόστος της εκτέλεσης του σώματος της συνάρτησης. Επίσης, το ότι η συνάρτηση είναι μικρή και εμφανίζεται μόνο 2 φορές στον κώδικα της sobel, δεν αυξάνει σημαντικά το μέγεθος του εκτελέσιμου αρχείου, πράγμα που δε μπορεί δυνητικά να αυξήσει σημαντικά τα instruction cache misses. Τέλος, το inlining μας δίνει τη δυνατότητα να κάνουμε επιπλέον βελτιστοποιήσεις που δεν μπορούσαμε να κάνουμε χωρίς αυτό, όπως strength reduction, common subexpression elimination κλπ.

(-O0)

Μέσος χρόνος εκτέλεσης: 1.758620 sec

Τυπική απόκλιση: 0.003587 sec

Μείωση μέσου χρόνου εκτέλεσης σε σχέση με το προηγούμενο στάδιο: 1.66%

(-fast)

Μέσος χρόνος εκτέλεσης: 0.044820 sec

Τυπική απόκλιση: 0.002383 sec

4:Loop unrolling, αρχεία: sobel4_unroll2.c, sobel4_unroll4.c, sobel4_unroll8.c, sobel4_unroll16.c

Για την περαιτέρω βελτίωση του χρόνου εκτέλεσης, εφαρμόσαμε loop unrolling στα loops της συνάρτησης sobel. Ειδικότερα, δοκιμάσαμε unrolling κατά 2, 4, 8 και 16. Επιπλέον, επειδή η τιμή του SIZE μπορεί να αλλάξει, δεν μπορούμε να γνωρίζουμε εκ των προτέρων αν ο αριθμός επαναλήψεων διαιρείται ακριβώς με τους αριθμούς 2,4,8 και 16, για αυτό το λόγο γράφτηκε κώδικας που εκτελεί τις επαναλήψεις που περισσεύουν εφόσον χρειάζεται.Η βελτίωση είναι σημαντική καθώς μειώνουμε σε μεγάλο βαθμό τον αριθμό των loops και το κόστος διαχείρισής τους, όπως προαναφέρθηκε στην δεύτερη κατά σειρά βελτιστοποίηση. Όπως φαίνεται και πειραματικά, οι περιπτώσεις των 4 και 8 παράγουν τον ταχύτερο κώδικα.

```
    //(1) allagh i j beltiwsh ston pinaka output
    for (i=1; i<SIZE-1; i+=1) {
        for (j=1 , k = 1; k <=((SIZE-2)/2);k++, j+=2 ) {

            ////////////////////////////////////////////////////////////////////
            resx = 0;
            resy =0;
            /* Apply the sobel filter and calculate the magnitude *
             * of the derivative.                                 */
            resx += input[(i -1)*SIZE + j -1] * horiz_operator[0][0];
            resx += input[(i -1)*SIZE + j ] * horiz_operator[0][1];
            resx += input[(i -1)*SIZE + j + 1] * horiz_operator[0][2];
            resx += input[i*SIZE + j + -1] * horiz_operator[1][0];
            resx += input[i*SIZE + j ] * horiz_operator[1][1];
            resx += input[i*SIZE + j + 1] * horiz_operator[1][2];
            resx += input[(i + 1)*SIZE + j -1] * horiz_operator[2][0];
            resx += input[(i + 1)*SIZE + j ] * horiz_operator[2][1];
            resx += input[(i + 1)*SIZE + j + 1] * horiz_operator[2][2];


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

            p = pow(resx, 2) + pow(resy, 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;
            ////////////////////////////////////////////////////////////////////
```

```
159
160             ////////////////////////////////////////////////////////////////////
161             resx = 0;
162             resy =0;
163             /* Apply the sobel filter and calculate the magnitude *
164              * of the derivative.                                 */
165             resx += input[(i -1)*SIZE + j] * horiz_operator[0][0];
166             resx += input[(i -1)*SIZE + j+1 ] * horiz_operator[0][1];
167             resx += input[(i -1)*SIZE + j + 2] * horiz_operator[0][2];
168             resx += input[i*SIZE + j] * horiz_operator[1][0];
169             resx += input[i*SIZE + j +1] * horiz_operator[1][1];
170             resx += input[i*SIZE + j +2] * horiz_operator[1][2];
171             resx += input[(i + 1)*SIZE + j] * horiz_operator[2][0];
172             resx += input[(i + 1)*SIZE + j + 1 ] * horiz_operator[2][1];
173             resx += input[(i + 1)*SIZE + j + 2] * horiz_operator[2][2];
174
175
176             resy += input[(i -1)*SIZE + j] * vert_operator[0][0];
177             resy += input[(i -1)*SIZE + j +1] * vert_operator[0][1];
178             resy += input[(i -1)*SIZE + j + 2] * vert_operator[0][2];
179             resy += input[i*SIZE + j] * vert_operator[1][0];
180             resy += input[i*SIZE + j +1] * vert_operator[1][1];
181             resy += input[i*SIZE + j + 2] * vert_operator[1][2];
182             resy += input[(i + 1)*SIZE + j ] * vert_operator[2][0];
183             resy += input[(i + 1)*SIZE + j +1] * vert_operator[2][1];
184             resy += input[(i + 1)*SIZE + j + 2] * vert_operator[2][2];
185
186             p = pow(resx, 2) + pow(resy, 2);
187             res = (int)sqrt(p);
188             /* If the resulting value is greater than 255, clip it *
189              * to 255.
190                                                                    */
191             if (res > 255)
192                 output[i*SIZE + j +1] = 255;
193             else
194                 output[i*SIZE + j + 1] = (unsigned char)res;
195             ////////////////////////////////////////////////////////////////////
196
197         }
```

```
198         for(k = 0; k < (SIZE-2) % 2 ; k++, j++ ) {
199 
200             ////////////////////////////////////////////////////////////////////
201             resx = 0;
202             resy =0;
203             /* Apply the sobel filter and calculate the magnitude *
204              * of the derivative.                                 */
205             resx += input[(i -1)*SIZE + j -1] * horiz_operator[0][0];
206             resx += input[(i -1)*SIZE + j ] * horiz_operator[0][1];
207             resx += input[(i -1)*SIZE + j + 1] * horiz_operator[0][2];
208             resx += input[i*SIZE + j + -1] * horiz_operator[1][0];
209             resx += input[i*SIZE + j ] * horiz_operator[1][1];
210             resx += input[i*SIZE + j + 1] * horiz_operator[1][2];
211             resx += input[(i + 1)*SIZE + j -1] * horiz_operator[2][0];
212             resx += input[(i + 1)*SIZE + j ] * horiz_operator[2][1];
213             resx += input[(i + 1)*SIZE + j + 1] * horiz_operator[2][2];
214 
215 
216             resy += input[(i -1)*SIZE + j -1] * vert_operator[0][0];
217             resy += input[(i -1)*SIZE + j ] * vert_operator[0][1];
218             resy += input[(i -1)*SIZE + j + 1] * vert_operator[0][2];
219             resy += input[i*SIZE + j -1] * vert_operator[1][0];
220             resy += input[i*SIZE + j ] * vert_operator[1][1];
221             resy += input[i*SIZE + j + 1] * vert_operator[1][2];
222             resy += input[(i + 1)*SIZE + j  -1] * vert_operator[2][0];
223             resy += input[(i + 1)*SIZE + j ] * vert_operator[2][1];
224             resy += input[(i + 1)*SIZE + j + 1] * vert_operator[2][2];
225 
226             p = pow(resx, 2) + pow(resy, 2);
227             res = (int)sqrt(p);
228             /* If the resulting value is greater than 255, clip it *
229              * to 255.
230                                                                    */
231             if (res > 255)
232                 output[i*SIZE + j] = 255;
233             else
234                 output[i*SIZE + j] = (unsigned char)res;
235             ////////////////////////////////////////////////////////////////////
236         }
237     }
```

```
238 
239     /* Now run through the output and the golden output to calculate *
240      * the MSE and then the PSNR.                                    */
241     for (i=1; i<SIZE-1; i++) {
242         for (j=1 , k = 1; k <=((SIZE-2)/2);k++, j+=2 ) {
243             t = pow((output[i*SIZE+j] - golden[i*SIZE+j]),2);
244             PSNR += t;
245             t = pow((output[i*SIZE+j+1] - golden[i*SIZE+j+1]),2);
246             PSNR += t;
247         }
248         for(k = 0; k < (SIZE-2) % 2 ; k++, j++ ) {
249             t = pow((output[i*SIZE+j] - golden[i*SIZE+j]),2);
250             PSNR += t;
251         }
252     }
253 
```

Διάγραμμα unroll factor με επίδοση: Η αύξηση στα 16 οφείλεται στο γεγονός ότι φτάνουμε σε τοπικό ελάχιστο (4,8), αλλά στην περίπτωση του 16 αυξάνουμε σημαντικά το μέγεθος του εκτελέσιμου με αποτέλεσμα να έχουμε περισσότερα misses στην instruction cache. Επιπλέον, στην περίπτωση που υπάρχει μόνο μία shared cache για data και instructions, η αύξηση των εντολών ενδέχεται να προκαλεί και misses που αφορούν δεδομένα, γεγονός που καθυστερεί ακόμα περισσότερο το σύστημα.

Unroll κατά 2

(-O0)

Μέσος χρόνος εκτέλεσης: 1.754997 sec

Τυπική απόκλιση: 0.001150 sec

(-fast)

Μέσος χρόνος εκτέλεσης: 0.054632 sec

Τυπική απόκλιση: 0.001191 sec

Unroll κατά 4

(-O0)

Μέσος χρόνος εκτέλεσης: 1.748794 sec

Τυπική απόκλιση: 0.002209 sec

(-fast)

Μέσος χρόνος εκτέλεσης: 0.075320 sec

Τυπική απόκλιση: 0.001081 sec

Unroll κατά 8

(-O0)

Μέσος χρόνος εκτέλεσης: 1.757707 sec

Τυπική απόκλιση: 0.002375 sec

(-fast)

Μέσος χρόνος εκτέλεσης: 0.058367 sec

Τυπική απόκλιση: 0.001750 sec

Unroll κατά 16

(-O0)

Μέσος χρόνος εκτέλεσης: 1.766627 sec

Τυπική απόκλιση: 0.006540 sec

(-fast)

Μέσος χρόνος εκτέλεσης: 0.062203 sec

Τυπική απόκλιση: 0.001278 sec

(-O0)Μείωση μέσου χρόνου εκτέλεσης σε σχέση με το προηγούμενο στάδιο για unroll κατά 2: 0.21%

5: Loop fusion, αρχεία: sobel5_unroll2-fusion.c, sobel5_unroll2-fusion-inter.c, sobel5_unroll4-fusion.c, sobel5_unroll4-fusion-inter.c , sobel5_unroll8-fusion.c, sobel5_unroll16-fusion.c

Στο προηγούμενο στάδιο, τα 2 loops της συνάρτησης sobel έγιναν unrolled κατά τον ίδιο αριθμό, κάτι που σε συνδυασμό με το ότι δεν παραβιάζονται οι εξαρτήσεις των δεδομένων μας επιτρέπει σε αυτό το στάδιο να εφαρμόσουμε loop fusion μεταξύ των 2 loops. Η εξάρτηση δεδομένων δεν παραβιάζεται διότι η προσπέλαση στον πίνακα output , για τον υπολογισμό του PSNR, που αφορά το δεύτερο loop εισάγεται στο τέλος της κάθε επανάληψης έτσι ώστε να χρησιμοποιούμε τις σωστές τιμές του πίνακα output που παράχθηκαν από το πρώτο loop. Η συγκεκριμένη αλλαγή μειώνει τη διαχειριστική επιβάρυνση των loops καθώς αυτά συνενώνονται και παράλληλα παρέχει περισσότερο κώδικα στο σώμα του loop για τυχόν μεταγενέστερες βελτιστοποιήσεις του μεταγλωττιστή ή κατά την εκτέλεση του κώδικα. Στην βελτιστοποίηση που εφαρμόσαμε (loop fusion μεταξύ των loops της sobel), παρατηρήσαμε ότι ο συνδυασμός loop unrolling κατά 2 και loop fusion επιφέρει την καλύτερη μέχρι στιγμής επίδοση σε σχέση με τους υπόλοιπους συνδυασμούς (loop unroll 2, loop unroll 4, loop unroll 8, loop unroll 16, unroll 4 και fusion, unroll 8 και fusion, unroll 16 και fusion), κάτι που μας κάνει να προτιμήσουμε το unroll κατά 2 στο προηγούμενο στάδιο. Αυτό μπορεί να οφείλεται στο πρότυπο προσπέλασης στην μνήμη, το οποίο εκμεταλλευόμαστε καλύτερα με το unroll κατά 2 και fusion σε σχέση με τα υπόλοιπα σενάρια που ενδέχεται να “χαλάνε” περισσότερο την τοπικότητα των αναφορών στη μνήμη. Οι αντίστοιχες μετρήσεις επιβεβαιώνουν τα παραπάνω σχόλια.

Επεξήγηση αρχείου unroll+fusion_inter: Στο αρχείο αυτό δοκιμάσαμε να χωρίσουμε τις προσπελάσεις στον πίνακα output που αφορούν το δεύτερο loop με σκοπό να δούμε πώς επηρεάζει το συγκεκριμένο πρότυπο προσπέλασης στη μνήμη τον συνολικό χρόνο εκτέλεσης. Πειραματικά, φαίνεται ότι αυτή η αλλαγή αυξάνει το συνολικό χρόνο εκτέλεσης του προγράμματος και κατά συνέπεια οι επόμενες βελτιστοποιήσεις δεν βασίζονται σε αυτό.

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

            p = pow(resx, 2) + pow(resy, 2);
            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;
            ////////////////////////////////////////////////////////////////////
            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++ ) {

            ////////////////////////////////////////////////////////////////////
            resx = 0;
            resy =0;
            /* Apply the sobel filter and calculate the magnitude *
             * of the derivative.                                 */
            resx += input[(i -1)*SIZE + j -1] * horiz_operator[0][0];
            resx += input[(i -1)*SIZE + j ] * horiz_operator[0][1];
            resx += input[(i -1)*SIZE + j + 1] * horiz_operator[0][2];
            resx += input[i*SIZE + j + -1] * horiz_operator[1][0];
            resx += input[i*SIZE + j ] * horiz_operator[1][1];
            resx += input[i*SIZE + j + 1] * horiz_operator[1][2];
            resx += input[(i + 1)*SIZE + j -1] * horiz_operator[2][0];
            resx += input[(i + 1)*SIZE + j ] * horiz_operator[2][1];
            resx += input[(i + 1)*SIZE + j + 1] * horiz_operator[2][2];


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

            p = pow(resx, 2) + pow(resy, 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;
            ////////////////////////////////////////////////////////////////////

            t = pow((output[i*SIZE+j] - golden[i*SIZE+j]),2);
            PSNR += t;

        }
    }
```

Unroll 2 & fusion

(-O0)

Μέσος χρόνος εκτέλεσης: 1.729657 sec

Τυπική απόκλιση: 0.002184 sec

(-fast)

Μέσος χρόνος εκτέλεσης: 0.059500 sec

Τυπική απόκλιση: 0.001077 sec

Unroll 2 & fusion inter

(-O0)

Μέσος χρόνος εκτέλεσης: 1.761746 sec

Τυπική απόκλιση: 0.008712 sec

(-fast)

Μέσος χρόνος εκτέλεσης: 0.061243 sec

Τυπική απόκλιση: 0.000568 sec

Unroll 4 & fusion

(-O0)

Μέσος χρόνος εκτέλεσης: 1.745788 sec

Τυπική απόκλιση: 0.003158 sec

(-fast)

Μέσος χρόνος εκτέλεσης: 0.082920 sec

Τυπική απόκλιση: 0.001304 sec

Unroll 4 & fusion inter

(-O0)

Μέσος χρόνος εκτέλεσης: 1.768716 sec

Τυπική απόκλιση: 0.002899 sec

(-fast)

Μέσος χρόνος εκτέλεσης: 0.083798 sec

Τυπική απόκλιση: 0.001548 sec

Unroll 8 &fusion

(-O0)

Μέσος χρόνος εκτέλεσης: 1.762881 sec

Τυπική απόκλιση: 0.002548 sec

(-fast)

Μέσος χρόνος εκτέλεσης: 0.056836 sec

Τυπική απόκλιση: 0.001345 sec

Unroll 16 & fusion

(-O0)

Μέσος χρόνος εκτέλεσης: 1.778750 sec

Τυπική απόκλιση: 0.002675 sec

(-fast)

Μέσος χρόνος εκτέλεσης: 0.054650 sec

Τυπική απόκλιση: 0.001255 sec

(-O0)Μείωση μέσου χρόνου εκτέλεσης σε σχέση με το προηγούμενο στάδιο για unroll κατά 2 και fusion: 1.44%

6: Loop invariant code motion, αρχείο: sobel6_linvcode.c

Παρατηρούμε ότι οι παρακάτω υπολογισμοί είναι ανεξάρτητοι από τις επαναλήψεις του nested loop της συνάρτησης sobel: (i - 1) * SIZE, i* SIZE, (i+1)*SIZE. Για το λόγο αυτό, μπορούμε σε κάθε επανάληψη i να αποθηκεύσουμε τις αντίστοιχες ποσότητες σε μεταβλητές εκτός του loop με αποτέλεσμα να εκτελούνται μόνο μία φορά, σε αντίθεση με την προηγούμενη έκδοση κώδικα που θα έπρεπε να εκτελεστούν ((SIZE-2)/2)+ ((SIZE-2) % 2) φορές για κάθε i. Αυτό συνεπάγεται και βελτίωση στην επίδοση.

```
119     //(1) allagh i j beltiwsh ston pinaka output
120     for (i=1; i<SIZE-1; i+=1) {
121         t1 = (i-1) * SIZE;
122         t2 = i * SIZE;
123         t3 = (i+1) * SIZE;
124         for (j=1 , k = 1; k <=((SIZE-2)/2);k++, j+=2 ) {
125 
126             ////////////////////////////////////////////////////////////////////
127             resx = 0;
128             resy =0;
129             /* Apply the sobel filter and calculate the magnitude *
130              * of the derivative.                                 */
131             resx += input[t1 + j -1] * horiz_operator[0][0];
132             resx += input[t1 + j ] * horiz_operator[0][1];
133             resx += input[t1 + j + 1] * horiz_operator[0][2];
134             resx += input[t2 + j + -1] * horiz_operator[1][0];
135             resx += input[t2 + j ] * horiz_operator[1][1];
136             resx += input[t2 + j + 1] * horiz_operator[1][2];
137             resx += input[t3 + j -1] * horiz_operator[2][0];
138             resx += input[t3 + j ] * horiz_operator[2][1];
139             resx += input[t3 + j + 1] * horiz_operator[2][2];
140 
141 
142             resy += input[t1 + j -1] * vert_operator[0][0];
143             resy += input[t1 + j ] * vert_operator[0][1];
144             resy += input[t1 + j + 1] * vert_operator[0][2];
145             resy += input[t2 + j -1] * vert_operator[1][0];
146             resy += input[t2 + j ] * vert_operator[1][1];
147             resy += input[t2 + j + 1] * vert_operator[1][2];
148             resy += input[t3 + j  -1] * vert_operator[2][0];
149             resy += input[t3 + j ] * vert_operator[2][1];
150             resy += input[t3 + j + 1] * vert_operator[2][2];
151 
```

(-O0)

Μέσος χρόνος εκτέλεσης: 1.677607 sec

Τυπική απόκλιση: 0.003429 sec

(-fast)

Μέσος χρόνος εκτέλεσης: 0.059993 sec

Τυπική απόκλιση: 0.001334 sec

(-O0)Μείωση μέσου χρόνου εκτέλεσης σε σχέση με το προηγούμενο στάδιο: 3%

7: Common subexpression elimination, αρχείο: sobel7_subexpr2.c

Για τη συγκεκριμένη βελτιστοποίηση παρατηρήσαμε ότι ο υπολογισμός των εκφράσεων (i-1)* SIZE + j = t1+ j, i * SIZE + j = t2 + j και (i+1)*SIZE + j = t3 + j επαναλαμβάνεται πολλές φορές, λόγω του function inlining και του unrolling που προηγήθηκαν, διότι υπολογίζεται σε κάθε προσπέλαση των πινάκων input, output και golden. Αποθηκεύοντας σε μεταβλητές τις εκφράσεις που επαναχρησιμοποιούνται πολλές φορές στον κώδικα ουσιαστικά τις υπολογίζουμε μία μόνο φορά και έπειτα τις επαναχρησιμοποιούμε μειώνοντας το κόστος υπολογισμού της απομάκρυνσης από την αρχή των πινάκων input, output και golden.

Γι’αυτό το λόγο, οι εκφράσεις ((SIZE - 2) / 2) και ((SIZE – 2) % 2) αποθηκεύονται σε μεταβλητές για να μην χρειάζεται να τις υπολογίζουμε κάθε φορά που εξετάζουμε τις συνθήκες εξόδου των loops της συνάρτησης sobel.

```
117     /* For each pixel of the output image */
118     t4 = (SIZE-2)/2;
119     t5 = (SIZE-2)%2;
120     //(1) allagh i j beltiwsh ston pinaka output
121     for (i=1; i<SIZE-1; i+=1) {
122         t1 = (i-1) * SIZE;
123         t2 = i * SIZE;
124         t3 = (i+1) * SIZE;
125         for (j=1 , k = 1; k <=t4;k++, j+=2 ) {
126             j1 = t1 + j;
127             j2 = t2 + j;
128             j3 = t3 + j;
129             ////////////////////////////////////////////////////////////////////////
130             resx = 0;
131             resy =0;
132             /* Apply the sobel filter and calculate the magnitude *
133              * of the derivative.                                 */
134             resx += input[j1 -1] * horiz_operator[0][0];
135             resx += input[j1 ] * horiz_operator[0][1];
136             resx += input[j1 + 1] * horiz_operator[0][2];
137             resx += input[j2 + -1] * horiz_operator[1][0];
138             resx += input[j2 ] * horiz_operator[1][1];
139             resx += input[j2 + 1] * horiz_operator[1][2];
140             resx += input[j3 -1] * horiz_operator[2][0];
141             resx += input[j3 ] * horiz_operator[2][1];
142             resx += input[j3 + 1] * horiz_operator[2][2];
143 
144 
145             resy += input[j1 -1] * vert_operator[0][0];
146             resy += input[j1 ] * vert_operator[0][1];
147             resy += input[j1 + 1] * vert_operator[0][2];
148             resy += input[j2 -1] * vert_operator[1][0];
149             resy += input[j2 ] * vert_operator[1][1];
150             resy += input[j2 + 1] * vert_operator[1][2];
151             resy += input[j3  -1] * vert_operator[2][0];
152             resy += input[j3 ] * vert_operator[2][1];
153             resy += input[j3 + 1] * vert_operator[2][2];
154 
```

(-O0)

Μέσος χρόνος εκτέλεσης: 1.664173 sec

Τυπική απόκλιση: 0.002140 sec

(-fast)

Μέσος χρόνος εκτέλεσης: 0.060589 sec

Τυπική απόκλιση: 0.000976 sec

(-O0)Μείωση μέσου χρόνου εκτέλεσης σε σχέση με το προηγούμενο στάδιο: 0.8%

8: Strength reduction, αρχείο: sobel8_strength_reduction.c

Η βελτιστοποίηση αυτή αφορά την αντικατάσταση “ακριβών” πράξεων με άλλες που είναι πιο “φθηνές” με ισοδύναμο αποτέλεσμα. Πιο συγκεκριμένα, οι γραμμές 124-127 της προηγούμενης έκδοσης όπου σε κάθε i πρέπει να κάνουμε τους αντίστοιχους πολλαπλασιασμούς, αντικαθίστανται με κατάλληλες αρχικοποιήσεις και διαδοχικές προσθέσεις του SIZE μετά από κάθε επανάληψη, πράξεις οι οποίες κάνουν ταχύτερο τον κώδικα. Στην αρχικοποίηση της t3 σημειώνεται ότι χρησιμοποιούμε αριστερή ολίσθηση αντί για τον πολλαπλασιασμό 2 * SIZE καθώς ο δεύτερος είναι πιο “ακριβός” σε σχέση με την ολίσθηση. Ομοίως, χρησιμοποιούμε δεξιά ολίσθηση αντί για διαίρεση με το 2 στην αρχικοποίηση της t4. Επιπλέον, αντικαθιστούμε τις αναφορές στους πίνακες horiz_operator και vert_operator με τον αριθμό που αντιστοιχεί στην σωστή θέση κάθε φορά. Στις περιπτώσεις όπου έχουμε πολλαπλασιασμό με το μηδέν, σβήνουμε την αντίστοιχη γραμμή καθώς δεν συμβάλλει στο αποτέλεσμα, ενώ στις περιπτώσεις που έχουμε πολλαπλασιασμό/διαίρεση με δύναμη του 2, χρησιμοποιούμε αριστερή/δεξιά ολίσθηση αντίστοιχα. Μία ακόμα αντικατάσταση, που βελτιώνει το συνολικό χρόνο εκτέλεσης σημαντικά, είναι της συνάρτησης pow με διαδοχικούς πολλαπλασιασμούς. Τέλος, μία από τις πιο ακριβές πράξεις είναι αυτή της τετραγωνικής ρίζας που υλοποιείται μέσω της συνάρτησης sqrt(). Για να μπορέσουμε να αποφύγουμε την κλήση της συγκεκριμένης συνάρτησης, δεδομένου ότι στο πρόβλημά μας χρειαζόμαστε τις τετραγωνικές ρίζες των αριθμών από 0 έως και 65025, χρησιμοποιούμε ένα lookup table 65026 θέσεων, το οποίο αρχικοποιείται με τρόπο ώστε η θέση i να περιέχει την τετραγωνική ρίζα του ακέραιου αριθμού i.

Η προσπέλαση του lookup table γίνεται μόνο εφόσον το p δεν είναι μεγαλύτερο του 65025 (sqrt(p) > 255 => p > 255^2 = 65025), ώστε να έχουμε ακόμα λιγότερες προσπελάσεις στον πίνακα σε σχέση με πριν. Με αυτό τον τρόπο, υπολογίζουμε όλες τις πιθανές τιμές που μπορεί να χρειαστούμε και τις αποθηκεύουμε στον πίνακα τον οποίο προσπελαύνουμε κάθε φορά που χρειαζόμαστε μία από αυτές, εξαλείφοντας τις κλήσεις της sqrt που επιβαρύνουν αρκετά το σύστημα. Ο κώδικάς μας δεν είναι τόσο μεγάλος ώστε πολλαπλές προσπελάσεις στο lookup table να είναι πολύ μεγαλύτερες από το κόστος αρχικοποίησής του. Στην πραγματικότητα, όμως, οι κώδικες είναι πολύ μεγαλύτεροι σε σχέση με τον κώδικα που χρησιμοποιούμε στην συγκεκριμένη εργασία, πράγμα που σημαίνει ότι ο χρόνος πολλαπλών προσπελάσεων στο lookup table θα είναι πολύ μεγαλύτερος από το χρόνο αρχικοποίησης, με αποτέλεσμα ο τελευταίος να θεωρείται αμελητέος. Για τον λόγο αυτό αρχικοποιούμε το lookup table εκτός του τμήματος κώδικα που θέλουμε να χρονομετρήσουμε. Οι παραπάνω αλλαγές προκαλούν τη μεγαλύτερη βελτίωση στο χρόνο εκτέλεσης του κώδικα σε σχέση με τις υπόλοιπες βελτιστοποιήσεις που υλοποιήθηκαν, γεγονός το οποίο φαίνεται και στα αντίστοιχα διαγράμματα.

```
68  double sobel(unsigned char *input, unsigned char *output, unsigned char *golden)
69  {
70      double PSNR = 0, t;
71      int i, j, k, resx, resy, t1, t2, t3, t4, t5, j1, j2, j3;
72      unsigned int p;
73      int sqr[65026];
74      int res;
75      struct timespec  tv1, tv2;
76      FILE *f_in, *f_out, *f_golden;
77
78
79      for(i = 0; i < 65026; i++){
80          sqr[i] = sqrt(i);
81      }
```

```
123      //strength reduction
124      t1 = 0;
125      t2 = SIZE;
126      t3 = SIZE << 1;
127      t4 = (SIZE-2)>>1;
128      t5 = (SIZE-2) % 2;
129      //(1) allagh i j beltiwsh ston pinaka output
130      for (i=1; i<SIZE-1; i+=1) {
131          for (j=1 , k = 1; k <=t4 ;k++, j+=2 ) {
132              j1 = t1 + j;
133              j2 = t2 + j;
134              j3 = t3 + j;
135              ////////////////////////////////////////////////////////////////////
136              resx = 0;
137              resy =0;
138              /* Apply the sobel filter and calculate the magnitude *
139               * of the derivative.                                 */
140              resx -= input[j1 -1];
141              //resx += input[j1] * horiz_operator[0][1];
142              resx += input[j1 + 1];
143              resx -= input[j2 + -1] << 1;
144              //resx += input[j2] * horiz_operator[1][1];
145              resx += input[j2 + 1] << 1;
146              resx -= input[j3 -1];
147              //resx += input[j3 ] * horiz_operator[2][1];
148              resx += input[j3 + 1];
149
150
151              resy += input[j1 -1];
152              resy += input[j1] << 1;
153              resy += input[j1 + 1];
154              //resy += input[j2 -1] * vert_operator[1][0];
155              //resy += input[j2 ] * vert_operator[1][1];
156              //resy += input[j2 + 1] * vert_operator[1][2];
157              resy -= input[j3 -1];
158              resy -= input[j3] << 1;
159              resy -= input[j3 + 1];
160
161
162              //p = pow(resx, 2) + pow(resy, 2);
163              p = resx*resx + resy*resy;
164              //res = (int)sqrt(p);
```

```
167                                          */
168              if (p > 65025)
169                  output[j2] = 255;
170              else
171                  output[j2] = (unsigned char)sqr[p];
172              ////////////////////////////////////////////////////////////////////
173
```

```
211              //t = pow((output[j2] - golden[j2]),2);
212              t = (output[j2] - golden[j2])*(output[j2]- golden[j2]);
213              PSNR += t;
214              //t = pow((output[j2+1] - golden[j2+1]),2);
215              t = (output[j2+1] - golden[j2+1])*(output[j2+1]- golden[j2+1]);
216              PSNR += t;
217
```

```
258                                                 */
259             if (p > 65025)
260                 output[j2] = 255;
261             else
262                 output[j2] = (unsigned char)sqr[p];
263             ////////////////////////////////////////////////////////////
264
265             //t = pow((output[j2] - golden[j2]),2);
266             t = (output[j2] - golden[j2])*(output[j2]- golden[j2]);
267             PSNR += t;
268
269         }
270         t1 += SIZE;
271         t2 += SIZE;
272         t3 += SIZE;
273     }
274
```

(-O0)

Μέσος χρόνος εκτέλεσης: 0.313675 sec

Τυπική απόκλιση: 0.001716 sec

(-fast)

Μέσος χρόνος εκτέλεσης: 0.056205 sec

Τυπική απόκλιση: 0.001273 sec

(-O0)Μείωση μέσου χρόνου εκτέλεσης σε σχέση με το προηγούμενο στάδιο: 81.15%

Κατά συνέπεια η συγκεκριμένη βελτιστοποίηση έχει το μεγαλύτερο ποσοστό μείωσης του μέσου χρόνου εκτέλεσης μεταξύ διαδοχικών σταδίων, που την καθιστά την πιο πρόσφορη.

9: Καλύτερη εκμετάλλευση της τοπικότητας για περαιτέρω μείωση του χρόνου εκτέλεσης, αρχείο: sobel9.c

Ομαδοποιήσαμε τις εντολές στον κώδικα της συνάρτησης sobel (lines) έτσι ώστε να γίνονται οι προσπελάσεις στη μνήμη διαδοχικά στα στοιχεία της ίδιας γραμμής του πίνακα input. Αυτό έχει ως αποτέλεσμα να βελτιώνουμε το ποσοστό των cache hits, σε αντίθεση με την προηγούμενη έκδοση που είχαμε αναφορές στη μνήμη που δεν αφορούσαν πρώτα την γραμμή i, μετά την γραμμή i+1 κλπ.

```
135             resx -= input[j1 -1];
136             resy += input[j1 -1];
137             resx += input[j1 + 1];
138             resy += input[j1 + 1];
139             resy += input[j1] << 1;
140             resx -= input[j2 + -1] << 1;
141             resx += input[j2 + 1] << 1;
142             resx -= input[j3 -1];
143             resy -= input[j3 -1];
144             resx += input[j3 + 1];
145             resy -= input[j3 + 1];
146             resy -= input[j3] << 1;
```

(-O0)

Μέσος χρόνος εκτέλεσης: 0.296195 sec

Τυπική απόκλιση: 0.001906 sec

(-fast)

Μέσος χρόνος εκτέλεσης: 0.054700 sec

Τυπική απόκλιση: 0.001747 sec

(-O0)Μείωση μέσου χρόνου εκτέλεσης σε σχέση με το προηγούμενο στάδιο: 5.6%

10: Υποβοήθηση μεταγλωττιστή, αρχείο: sobel10_compiler_help.c

Στην τρέχουσα βελτιστοποίηση δίνουμε στον μεταγλωττιστή πληροφορία που δε μπορεί να εξάγει από την ανάλυσή του, με σκοπό να τον βοηθήσουμε στις αυτόματες βελτιστοποιήσεις του. Αρχικά, βάζουμε το πρόθεμα register στις μεταβλητές που θεωρούμε ότι πρέπει να μπουν σε καταχωρητές. Επίσης, βάζουμε restrict στους δείκτες input, output, golden δηλώνοντας ότι οι συγκεκριμένες θέσεις μνήμης θα προσπελαστούν αποκλειστικά μέσω των παραπάνω δεικτών. Αυτές οι προσθήκες δεν φαίνεται να έχουν βελτίωση όσον αφορά την εκτέλεση του κώδικα χωρίς καμία βελτιστοποίηση του compiler. Αντιθέτως, όταν χρησιμοποιήσουμε το flag –fast κατά τη μεταγλώττιση, παρατηρούμε μείωση στο συνολικό χρόνο εκτέλεσης, γεγονός που επιβεβαιώνει ότι τα προθέματα register και restrict δίνουν απαραίτητες για τον μεταγλωττιστή πληροφορίες που οδηγούν στο να παραχθεί ταχύτερος κώδικας.

```
61  * image, the output produced by the algorithm and the output used as    *
62  * golden standard for the comparisons.                                  */
63  double sobel(unsigned char *restrict input, unsigned char *restrict output, unsigned char *restrict golden)
64  {
65      register double PSNR = 0, t;
66      register int i, j, k, resx, resy, t1, t2, t3, t4, t5, j1, j2, j3;
67      unsigned int p;
68      int sqr[65026];
69      int res;
70      struct timespec  tv1, tv2;
71      FILE *f_in, *f_out, *f_golden;
72
```

(-O0)

Μέσος χρόνος εκτέλεσης: 0.296888 sec

Τυπική απόκλιση: 0.003066 sec

(-fast)

Μέσος χρόνος εκτέλεσης: 0.054204 sec

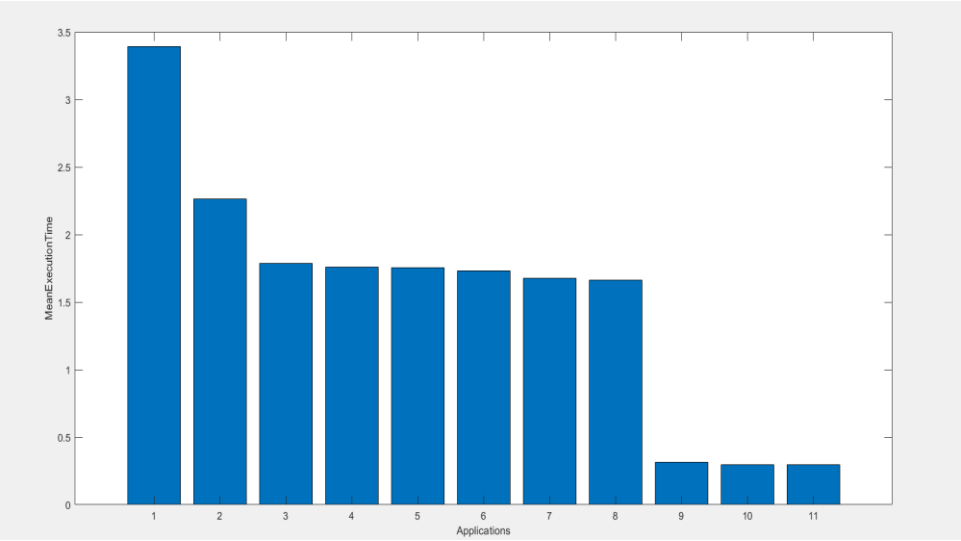
Τυπική απόκλιση: 0.002609 sec

Η τρέχουσα βελτιστοποίηση δεν μειώνει τον μέσο χρόνο εκτέλεσης με flags -O0. Όμως, το ποσοστό μείωσης του μέσου χρόνου εκτέλεσης σε σχέση με το προηγούμενο στάδιο για -fast είναι 0.9%.

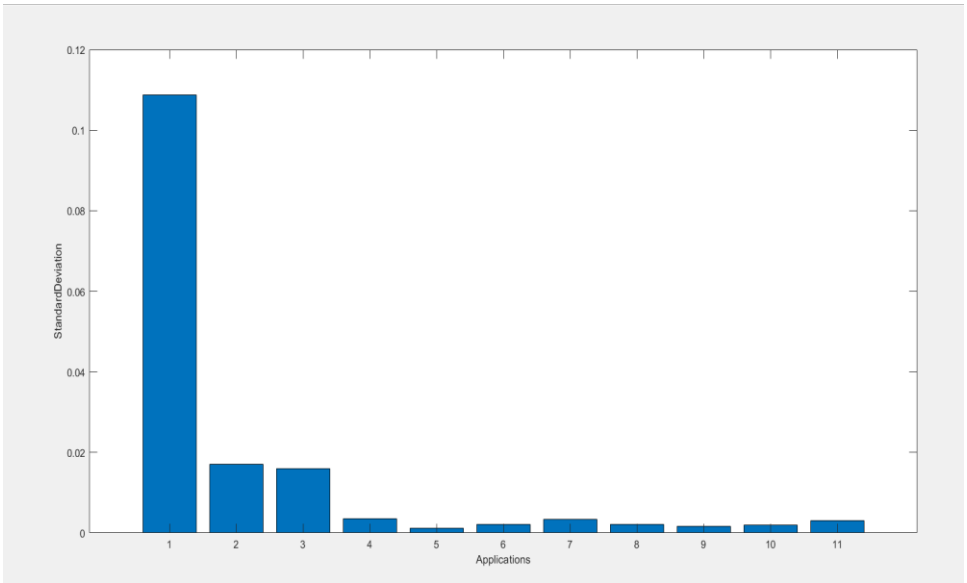
ΔΙΑΓΡΑΜΜΑΤΑ

Στα παρακάτω διαγράμματα χρησιμοποιούμε τα ακόλουθα αρχεία με τη σειρά που καταγράφονται:

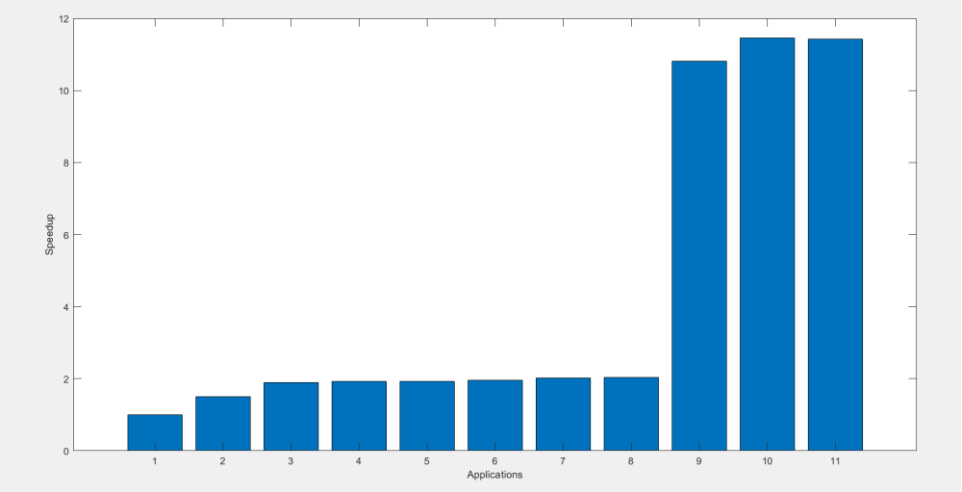
Διάγραμμα Μέσου χρόνου εκτέλεσης(-O0)- Εφαρμογής



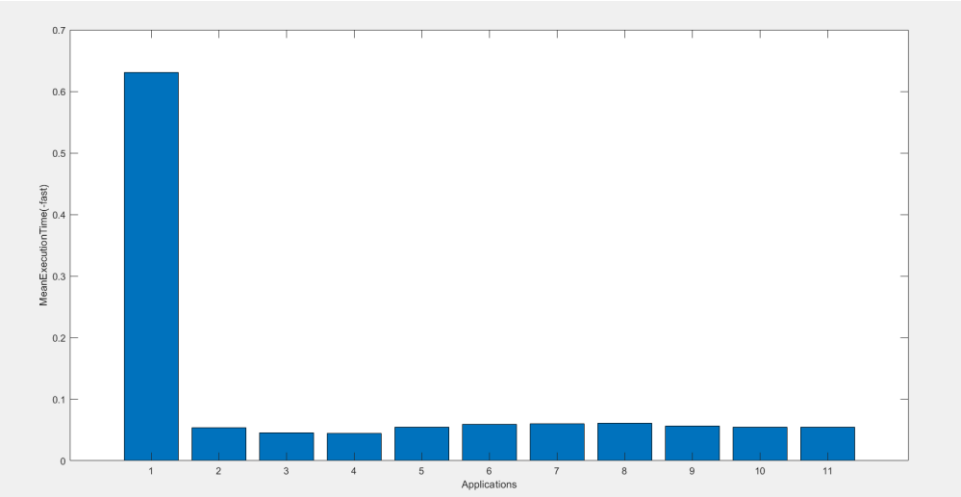
Διάγραμμα Τυπικής απόκλισης (-O0) - Εφαρμογής



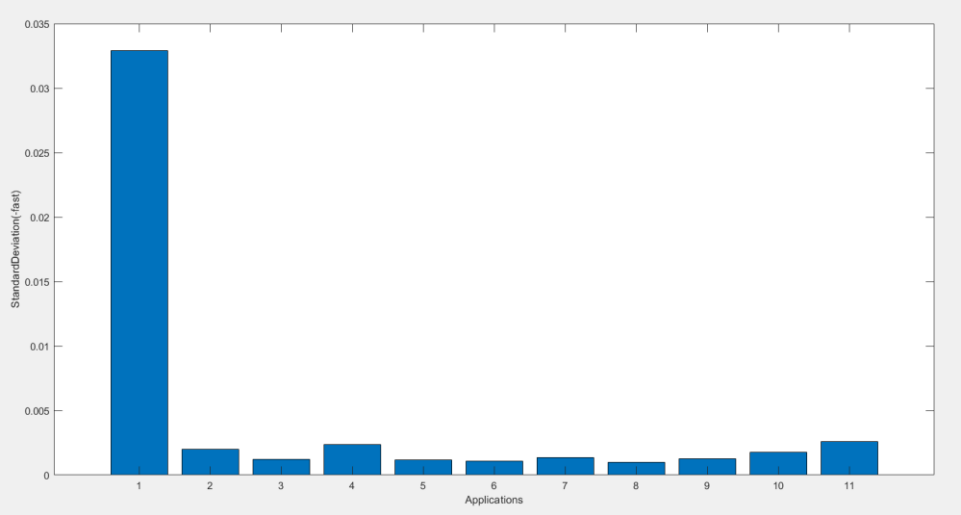
Διάγραμμα Speedup(σε σχέση με τον αρχικό κώδικα για -O0) - Εφαρμογής



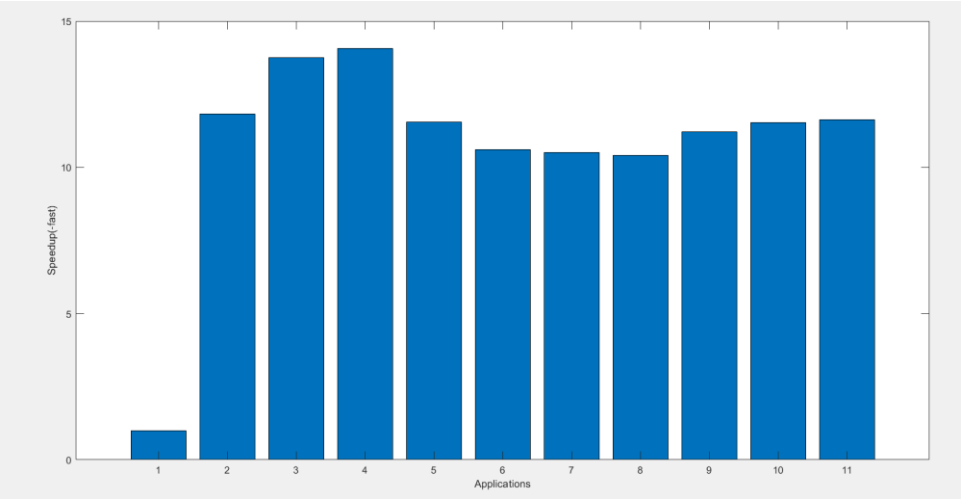
Διάγραμμα Μέσου χρόνου εκτέλεσης(-fast)- Εφαρμογής



Διάγραμμα Τυπικής απόκλισης (-fast) - Εφαρμογής



Διάγραμμα Speedup (σε σχέση με τον αρχικό κώδικα για -fast) - Εφαρμογής

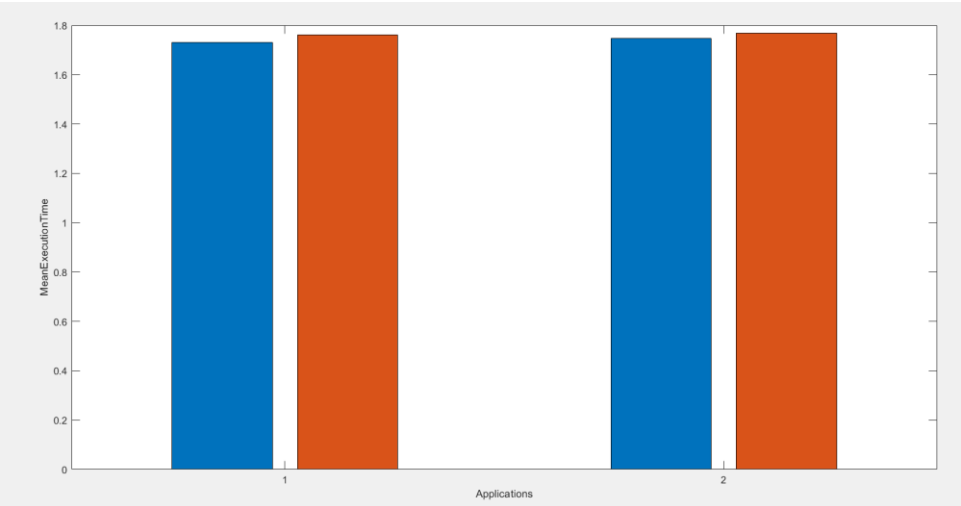


Σύγκριση μέσου χρόνου εκτέλεσης fusion_unroll με fusion_unroll_inter - Εφαρμογής

Ομάδα 1: unroll factor = 2

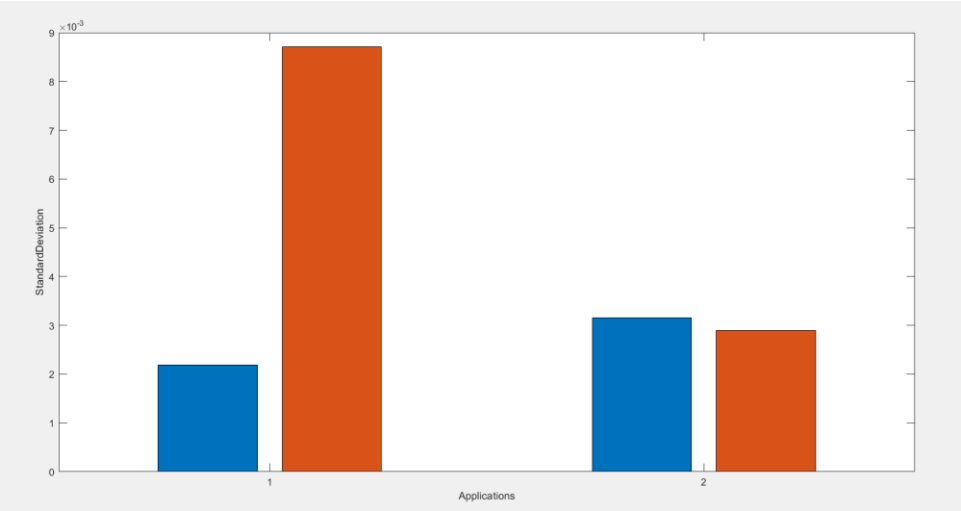
Ομάδα 2: unroll factor = 4

Μπλε χρώμα: fusion_unroll, Πορτοκαλί χρώμα: fusion_unroll_inter

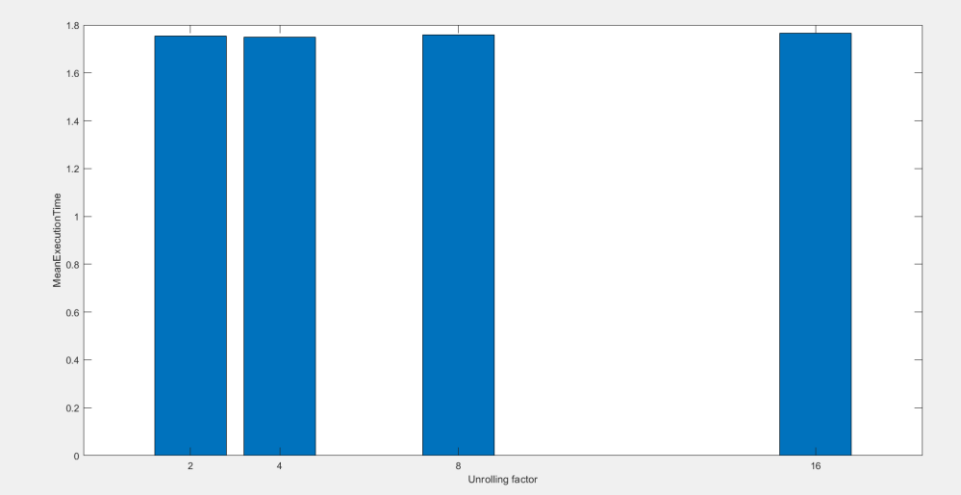


Σύγκριση τυπικής απόκλισης fusion_unroll με fusion_unroll_inter - Εφαρμογής

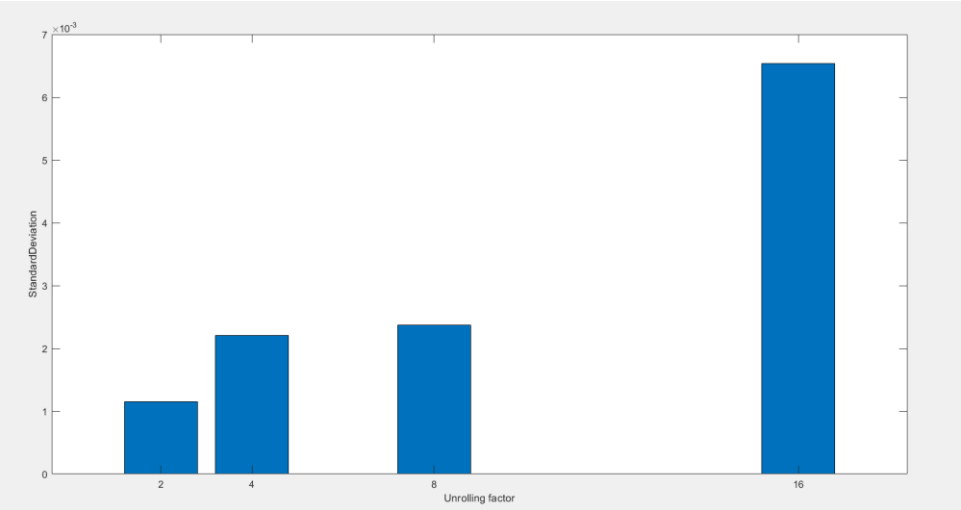
Για όμοιες ομάδες και χρώματα όπως προηγουμένως, το διάγραμμα είναι το εξής:



Σύγκριση μέσου χρόνου εκτέλεσης - Unroll factor(2,4,8,16)



Σύγκριση τυπικής απόκλισης - Unroll factor(2,4,8,16)



Σχολιασμός compilation με -fast

Εκτελέσαμε μία παράλληλη σειρά πειραμάτων στην οποία οι ίδιοι κώδικες που προέκυψαν από τα παραπάνω βήματα βελτιστοποίησης έγιναν compile με flag -fast αντί για -O0. Επίσης, στο compilation έχει προστεθεί το flag -qopt-report-file=tests/name, όπου name το όνομα του εκάστοτε εκτελέσιμου κώδικα. Με αυτό το flag παράγεται ένα optimazation report κατά το compilation από τον ίδιο τον compiler στο οποίο εξηγεί ποιες βελτιστοποίησεις πραγματοποίησε ή όχι και γιατί. Εμείς έχουμε επιλέξει επίπεδο 2(default) λεπτομέρειας για το report.

Παρακάτω παραθέτουμε μερικές παρατηρήσεις πάνω στα report που συλλέξαμε και σχολιάζουμε κάποιες από τις βασικές βελτιστοποιήσεις που κάνει ο compiler:

-Loop interchange

Στιγμιότυπο από report icc κώδικα sobel_orig.c

```
LOOP BEGIN at sobel_orig.c(108,3) inlined into sobel_orig.c(155,9)
   remark #15411: vectorization support: conversion from float to int will be emulated   [ sobel_orig.c(112,5) ]
   remark #15301: OUTER LOOP WAS VECTORIZED

   LOOP BEGIN at sobel_orig.c(45,3) inlined into sobel_orig.c(155,9)
   <Peeled>
      remark #25444: Loopnest Interchanged: ( 1 2 ) --> ( 2 1 )
```

Γίνεται loop interchange στο nested loop στη συνάρτηση sobel line 108.

-Inline

Στους κώδικες sobel1_loop_interchange.c και sobel2_unroll.c όπου δεν είχαμε κάνει inline τη συνάρτηση convolution2D στην sobel ο compiler την κάνει inline.

Στιγμιότυπο από το report icc κώδικα sobel_orig.c

```
-> INLINE: (111,12) convolution2D(int, int, const unsigned char *, char (*)[3])
-> INLINE: (112,9) convolution2D(int, int, const unsigned char *, char (*)[3])
```

-Loop fusion

Τα δύο διαδοχικά nested loops στην συνάρτηση sobel γίνονται fusion.(Για το δεύτερο loop αναφέρεται "lost in fusion").

Στιγμιότυπο από report icc κώδικα sobel_orig.c

```
LOOP BEGIN at sobel_orig.c(45,3) inlined into sobel_orig.c(155,9)
<Peeled>
   remark #25444: Loopnest Interchanged: ( 1 2 ) --> ( 2 1 )
   remark #25045: Fused Loops: ( 45 45 )

   remark #15335: loop was not vectorized: vectorization possible but seems inefficient. Use vector always directive or -vec-threshold0 to override    [ sobel_orig.c(45,3) ]

   LOOP BEGIN at sobel_orig.c(44,2) inlined into sobel_orig.c(155,9)
   <Peeled>
      remark #25045: Fused Loops: ( 44 44 )

      remark #15335: loop was not vectorized: vectorization possible but seems inefficient. Use vector always directive or -vec-threshold0 to override
   LOOP END

   LOOP BEGIN at sobel_orig.c(44,2) inlined into sobel_orig.c(155,9)
      remark #25046: Loop lost in Fusion
   LOOP END
LOOP END

LOOP BEGIN at sobel_orig.c(44,2) inlined into sobel_orig.c(155,9)
   remark #25444: Loopnest Interchanged: ( 1 2 ) --> ( 2 1 )
   remark #25046: Loop lost in Fusion
LOOP END
```

-Loop unroll

Μπορεί να εφαρμοστεί σε οποιοδήποτε loop καθώς δεν χρειάζεται έλεγχος για εξάρτηση δεδομένων. Ο compiler επιλέγει τον πιο αποδοτικό βαθμό του unroll με βάση κάποιων μετρικών.

Για παράδειγμα στο report από icc του κώδικα sobel_orig.c γίνεται unroll 3

```
LOOP BEGIN at sobel_orig.c(45,3) inlined into sobel_orig.c(155,9)
<Peeled>
   remark #25436: completely unrolled by 3
```

-Vectorization

Μία από τις βασικότερες βελτιστοποιήσεις που πραγματοποίει είναι το vectorization. Για τον σκοπό αυτό χρησιμοποιούνται SIMD(Single Instruction Multiple Data) instructions. Πιο συγκεκριμένα ένα vector(διάνυσμα) είναι ένας instruction operand-SIMD operand ο οποίος περιέχει ένα σετ από δεδομένα πακεταρισμένα σε ένα μονοδιάστατο πίνακα. Η χρήση εντολών SIMD επιτρέπει οι πράξεις που απαιτούνται για το κάθε στοιχείο του vector να εφαρμοστούν παράλληλα για όλα τα στοιχεία του. Με αυτόν τον τρόπο επιτυγχάνεται παραλληλοποίηση σε επίπεδο δεδομένων.

Από τα report προκύπτει ότι vectorization εφαρμόζεται σε κάθε αρχείο στο εσωτερικό loop του nested loop στην συνάρτηση sobel.

Στιγμιότυπο από report του icc του κώδικα sobel_orig.c

```
LOOP BEGIN at sobel2_unroll.c(121,3) inlined into sobel2_unroll.c(169,9)
   remark #25045: Fused Loops: ( 121 140 )

   remark #15411: vectorization support: conversion from float to int will be emulated   [ sobel2_unroll.c(125,5) ]
   remark #15301: FUSED LOOP WAS VECTORIZED
```

Παρόλα αυτά το vectorization δεν είναι αποδοτικό για όλα τα loop, όπως και φαίνεται και από το στιγμιότυπο του report του icc από κώδικα sobel_orig.c.

```
29
30   LOOP BEGIN at sobel_orig.c(71,2) inlined into sobel_orig.c(155,9)
31      remark #15335: loop was not vectorized: vectorization possible but seems inefficient. Use vector always directive or -vec-threshold0 to override
32      remark #25438: unrolled without remainder by 2
33   LOOP END
```

Για την εφαρμογή αυτής της βελτιστοποίησης ο compiler πρέπει να γνωρίζει ότι δεν υπάρχουν εξαρτήσεις ανάμεσα στα δεδομένα για τον λόγο αυτό πολλές φορές παράγει δύο εκδοχές για ένα loop, ένα vectorized και ένα unvectorized και πραγματοποιεί real time test για να ελέγξει αν υπάρχει εξάρτηση στα δεδομένα και κρατάει στην συνέχεια την κατάλληλη εκδοχή. Επίσης οι εντολές SIMD είναι ποιό αποδοτικές σε ένα vectorized loop όταν τα δεδομένα είναι aligned σε μία διεύθυνση μνήμης που είναι πολλαπλάσια του πλάτους του SIMD register. Για να το πετύχει αυτό ο compiler μπορεί να κάνει "peel", δηλαδή να αφήσει, μερικές αρχικές διαπεράσεις, ώστε ο vectorized kernel να μπορεί να κάνει πράξεις σε δεδομένα τα οποία είναι καλύτερα aligned. Οι επαναλήψεις που μένουν μπορεί να βελτιστοποιηθούν σε ένα ξεχωριστό "reminder" loop.

Τέτοιο παράδειγμα εντοπίσαμε στο report του icc από κώδικα sobel_orig.c

```
LOOP BEGIN at sobel_orig.c(45,3) inlined into sobel_orig.c(155,9)
<Peeled>
```

```
LOOP BEGIN at sobel_orig.c(108,3) inlined into sobel_orig.c(155,9)
<Remainder loop for vectorization>
```

Από τις βελτιστοποιήσεις αυτές που σχολιάσαμε το vectorization έχει την μεγαλύτερη συμβολή στην βελτίωση του χρόνου εκτέλεσης. Παρ'όλα αυτά ο compiler πραγματοποιεί πολλές ακόμα βελτιστοποιήσεις τις οποίες είναι δύσκολο να αναγνωρίσουμε και σίγουρα συμβάλλουν δραματικά και αυτές στην ελαχιστοποίηση του χρόνου εκτέλεσης.Πάντως όσο πιο πολλές βελτιστοποιήσεις εφαρμόζουμε τόσο ποιο πολύ βοηθάμε τον compiler να πραγματοποιήσει επιπλέον βελτιστοποιήσεις, γεγονός που φαίνεται και από την προοδευτική ελάττωση του χρόνου εκτέλεσης των εκτελέσιμων που παράχθηκαν με -fast όσο εφαρμόζονταν επιπλέον βήματα βελτιστοποίησης.