

ΔΗΜΟΚΡΙΤΕΙΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΡΑΚΗΣ



ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ & ΜΗΧΑΝΙΚΩΝ ΤΠΟΛΟΓΙΣΤΩΝ
ΣΧΕΔΙΑΣΜΟΣ ΕΝΣΩΜΑΤΩΜΕΝΩΝ ΣΥΣΤΗΜΑΤΩΝ

Αλγόριθμος εξαγωγής ακμών του Canny

Φοιτητές

Ιωάννης-Αριστείδης Μασλάρης
57348
Νικόλαος Χατζηπαπάς
57370

Καθηγητής
Γεώργιος Συρακούλης

Ιανουάριος 2021

Περιεχόμενα

1 Μέρος πρώτο	3
1.1 Εισαγωγή στον αλγόριθμο Canny	3
1.1.1 Φίλτραρισμα εικόνας με φίλτρο Gaussian	3
1.1.2 Προσδιορισμός κλίσης της εικόνας	3
1.1.3 Καταστολή μη μέγιστων τιμών	4
1.1.4 Κατωφλίωση υστέρησης.	5
1.2 Τλοποίηση αλγορίθμου σε γλώσσα C	6
1.3 Πρόγραμμα Arm Development Suite	11
1.4 Μέθοδοι Βελτιστοποίησης	13
1.4.1 Loop Unrolling	13
1.4.2 Loop Fusion	14
1.4.3 Loop Interchange	15
1.4.4 Loop Collapsing	17
1.4.5 Loop Inversion	18
1.4.6 Loop Tiling	19
1.4.7 Αντικατάσταση της συνάρτησης pow	20
1.4.8 Συνδιασμός μεθόδων βελτιστοποίησης	21
1.5 Πίνακας δεδομένων	25
1.5.1 Μέγεθος	25
1.5.2 Αριθμός προσπελάσεων	25
1.6 Συμπεράσματα	26
1.6.1 Επιλογή μάσκας kernel για Gaussian blur	26
1.6.2 Επιλογή τιμών κατωφλίου T1 και T2	27
1.6.3 Βελτιστοποίηση χρόνου	28
2 Μέρος δεύτερο	29
2.1 Μνήμη	29
2.1.1 ROM	29
2.1.2 RAM	29
2.1.3 Memory views	29
2.2 Αρχεία για εισαγωγή μνήμης	31
2.2.1 Αρχείο memory.map	31
2.2.2 Αρχείο scatter.txt	31
2.2.3 Αρχείο stack.c	31
2.3 Ορισμός αρχιτεκτονικής μνήμης	33
2.3.1 Αρχιτεκτονική 1	33
2.3.2 Αρχιτεκτονική 2	35
2.3.3 Αρχιτεκτονική 3	36
2.3.4 Σημασία ρολογιού επεξεργαστή	38
2.3.5 Αρχιτεκτονική 4	38
2.3.6 Αρχιτεκτονική 5	40
2.4 Συμπεράσματα	42

3 Μέρος τρίτο	45
3.1 Επαναχρησιμοποίηση δεδομένων	45
3.1.1 Επαναχρησιμοποίηση δεδομένων και συνέλιξη	45
3.2 Εισαγωγή buffers στον κώδικα	46
3.3 Δοκιμές δομών buffers	49
3.3.1 Δοκιμή 1	49
3.3.2 Δοκιμή 2	50
3.3.3 Δοκιμή 3	52
3.4 Συμπεράσματα	54



Μέρος πρώτο

1.1 Εισαγωγή στον αλγόριθμο Canny

Ο αλγόριθμος που πρότεινε ο Canny για ανίχνευση ακμών σε εικόνες υεωρείται ο βέλτιστος που μπορούμε να ακολουθήσουμε για ανίχνευση ακμών παρουσία λευκού υφορύβου. Για την υλοποίησή του απαιτούνται συγκεκριμένα βήματα. Πρόθεση του Canny ήταν να βελτιώσει τους ήδη υπάρχοντες αλγορίθμους όταν ερευνούσε την περιοχή της ανίχνευσης ακμών. Για να το πετύχει αυτό όρισε κάποια κριτήρια για να αξιολογήσει την αποτελεσματικότητα των αλγόριθμων αυτών. Τα κριτήρια στα οποία βασίζεται η ανάπτυξη του αλγορίθμου είναι:

1. Σωστή ανίχνευση των ακμών
2. Εντοπισμός σωστής ύφεσης ακμών
3. Μοναδική απόκριση σε κάθε ακμή

Για την υλοποίηση της μεθόδου απαιτείται να γίνουν τα ακόλουθα βήματα σειριακά.



Εικόνα 1.1: Αρχική ασπρόμαυρη εικόνα bus 420x280.

1.1.1 Φιλτράρισμα εικόνας με φίλτρο Gaussian

Το αποτέλεσμα της ανίχνευσης ακμών επηρεάζεται έντονα από το υόρυβο, οπότε έιναι αναγκαίο το φιλτράρισμα του. Αυτό γίνεται με ομαλοποίηση της φωτεινότητας κάθε εικονοστοιχείου σύμφωνα με τα γειτονικά του εικονοστοιχεία, μέσω συνέλιξης του kernel του φίλτρου με την εικόνα. Σε αυτό το στάδιο το μέγεθος του kernel παίζει σημαντικό ρόλο στο αποτέλεσμα της εύρεσης ακμών. Το οπτικό αποτέλεσμα είναι η υόλωση της εικόνας.

1.1.2 Προσδιορισμός κλίσης της εικόνας

Μία ακμή στην εικόνα μπορεί να έχει οποιαδήποτε κατεύθυνση. Σε αυτό το βήμα εντοπίζεται η κατεύθυνση αυτή μέσω τους προσδιορισμού της κλίσης του κάθε εικονοστοιχείου. Αυτό επιτυγχάνεται με την χρήση της οριζόντιας και κατακόρυφης μάσκας Sobel για τους άξονες x και



Εικόνα 1.2: Αποτέλεσμα εφαρμογής φίλτρου Gaussian 3x3

υ αντίστοιχα. Μετά την εφαρμογή της μάσκας προκύπτουν δύο πίνακες, ένας για τον άξονα x και ένας για τον άξονα y . Μέσω αυτών των πινάκων υπολογίζονται οι πίνακες που δείχνουν τα πλάτη και τις κλίσεις για το κάθε εικονοστοιχείο.



Εικόνα 1.3: Αποτέλεσμα μετά την εφαρμογή της μάσκας Sobel.

1.1.3 Καταστολή μη μέγιστων τιμών

Αυτό το βήμα αποσκοπεί στο φιλτράρισμα και την λέπτυνση των ακμών. Απαλείφουμε από τον πίνακα με τα πλάτη που προέκυψαν ως θέσεις ακμών εκείνες που τοπικά έχουν την μικρότερη τιμή. Εξετάζεται ουσιαστικά η φωτεινότητα του εικονοστοιχείου σε σχέση με τα γειτονικά που βρίσκονται στην κλίση του.



Εικόνα 1.4: Αποτέλεσμα καταστολής μη μέγιστων τιμών.



1.1.4 Κατωφλίωση υστέρησης.

Τέλος η κατωφλίωση υστέρησης στοχεύει στην περαιτέρω μείωση των εικονοστοιχείων που προέκυψαν από την καταστολή των μη μέγιστων τιμών. Αρχικά ορίζονται δύο τιμές κατωφλίου, οι T_1 και T_2 . Μόλις βρεθεί το πρώτο εικονοστοιχείο με τιμές μεγαλύτερη του T_2 προχωράμε διαδοχικά στα γειτονικά συνδεδεμένα εικονοστοιχεία μέχρι να βρουμε κάποιο με κλίση μικρότερη από T_1 . Όσα εικονοστοιχεία έχουν τιμή κάτω από T_1 γίνονται ίσα με το 0, ενώ όσα έχουν τιμή μεγαλύτερη του T_2 θέτονται ίσα με το 255. Όσα έχουν μια ενδιάμεση τιμή ανάμεσα στα T_1 και T_2 εξετάζεται αν αποτελούν μέρος μίας ακμής. Αν είναι θέτονται ίσα με 255, ενώ αν δεν είναι με το 0. Μέσω αυτής της διαδικασίας μπορεί να επιτευχθεί ακόμα καλύτερη λέπτυνση των ακμών αλλά και απομάκρυνση του θορύβου.



Εικόνα 1.5: Αποτέλεσμα με επιλογή $T_1=235$ και $T_2=240$.



1.2 Υλοποίηση αλγορίθμου σε γλώσσα C

Ο αλγόριθμος χωρίζεται σε τέσσερις βασικές συναρτήσεις. Κάθε μία αναφέρεται αντίστοιχα σε κάθε βήμα που θεωρητικά εξηγήσθηκε στην προηγούμενη ενότητα. Επίσης χρησιμοποιείται η συνάρτηση `read()` για την ανάγνωση των εικόνων και η συνάρτηση `write()` για την αποθήκευση του αποτελέσματος. Αρχικά ορίζουμε το αρχείο εικόνας που θα επεξεργαστούμε με τις διαστάσεις του, καθώς και τις μεταβλητές που θα χρειαστούμε. Στη συνέχεια διαβάζουμε την εικόνα.

```
1 #include <stdio.h>
2 #include <math.h>
3 #include <string.h>
4 #include <stdlib.h>
5
6 #define N 420 /* frame dimension for QCIF format */
7 #define M 280 /* frame dimension for QCIF format */
8
9 #define filename "bus_420x280.yuv"
10#define file_yuv "test.yuv"
11
12/* code for armulator*/
13// declare variables
14
15int current[M][N];
16int output_gaussian[M+2][N+2];
17int Eo[M+2][N+2];
18int Es[M+2][N+2];
19int Igx[M+2][N+2];
20int IgY[M+2][N+2];
21int Edges[M+2][N+2];
22int gaussianMask[3][3];
23int GxMask[3][3];
24int GyMask[3][3];
25int NonMax[M+2][N+2];
26int temp[M+2][N+2];
27
28int i, j, newPixel, row, col, kernelRow, kernelCol;
29
30int kernelSize = 5;
31int searched[M+2][N+2]={0};
32int limit=1;
33int limit_gauss=1;
34int T1=230;
35int T2=245;
36
40 // read image
41 void read()
42 {
43     FILE *frame_c;
44     if((frame_c=fopen(filename,"rb"))==NULL)
45     {
46         printf("current frame doesn't exist\n");
47         exit(-1);
48     }
49
50     for(i=0;i<M;i++)
51     {
52         for(j=0;j<N;j++)
53         {
54             current[i][j]=fgetc(frame_c);
55         }
56     }
57
58     fclose(frame_c);
59 }
60
61 //write image
62 void write()
63 {
64     FILE *frame_yuv;
65     frame_yuv=fopen(file_yuv,"wb");
66
67     for(i=1;i<M+1;i++)
68     {
69         for(j=1;j<N+1;j++)
70         {
71             if(i==1 || j==1 || i==M || j==N)
72             {
73                 fputc(0,frame_yuv);
74             }
75             else
76                 fputc(Edges[i][j],frame_yuv);
77         }
78     }
79
80     fclose(frame_yuv);
81 }
```

Εικόνα 1.6: Ορισμός μεταβλητών και συναρτήσεις `read()`,`write()`.

Ακολουθεί το φιλτράρισμα της εικόνας με φίλτρο Gauss. Αρχικά βάζουμε ένα περίγραμμα ενός εικονοστοιχείου γύρω από την εικόνα και αποθηκεύουμε το αποτέλεσμα στον πίνακα `temp`. Αυτό επιτρέπει ώστε η μάσκα να μπορεί να εφαρμοστεί και στα άκρα της εικόνας. Μετά από δοκιμές επιλέγουμε μία μάσκα διαστάσεων 3x3. Διατρέχεται όλη η εικόνα και κάθε εικονοστοιχείο της παίρνει την τιμή που προκύπτει ως εξής:

$$\frac{\text{Conv}(Kernel, \text{neighboring pixels})}{\text{kernel weight}}$$

Το αποτέλεσμα αποθηκεύεται στον πίνακα `output_gaussian`.



```
83 // =====
84 void gaussian_blur()
85 {
86
87     for(i=0;i<(M+2);++i)
88     {
89         for(j=0;j<(N+2);++j)
90         {
91             temp[i][j]=0;
92         }
93     }
94
95     for(i=1;i<(M+1);++i)
96     {
97         for(j=1;j<(N+1);++j)
98         {
99             temp[i][j]=current[i-1][j-1];
100        }
101    }
102
103 // Declare Gaussian mask
104 gaussianMask[0][0] = 1; gaussianMask[0][1] = 2; gaussianMask[0][2] = 1;
105 gaussianMask[1][0] = 2; gaussianMask[1][1] = 4; gaussianMask[1][2] = 2;
106 gaussianMask[2][0] = 1; gaussianMask[2][1] = 2; gaussianMask[2][2] = 1;
107
108 for (col=limit_gauss;col<=(N+limit_gauss);col++){
109     for (row=limit_gauss;row<=(M+limit_gauss);row++){
110         newPixel = 0;
111
112         //printf("%d",row);
113         for (kernelCol=-limit_gauss;kernelCol<=limit_gauss; kernelCol++){
114             for (kernelRow=-limit_gauss; kernelRow<=limit_gauss; kernelRow++){
115                 newPixel = newPixel + temp[row+kernelRow][col+kernelCol]*gaussianMask[limit_gauss+kernelRow][limit_gauss+kernelCol];
116             }
117         }
118         output_gaussian[row][col] = (int)(floor(newPixel)/16);
119     }
120 }
121
122 }
```

Εικόνα 1.7: Εφαρμογή Gaussian φίλτρου για θόλωση.

Αμέσως μετά, στην συνάρτηση sobel(), γίνεται προσδιορισμός της κλίσης της εικόνας. Ορίζουμε δύο μάσκες sobel, μία οριζόντια και μία κατακόρυφη. Γίνεται συνέληξη του πίνακα output_gaussian με τις μάσκες sobel και υπολογίζουμε τα Igx, Igx. Τέλος προσδιορίζουμε τα πλάτη και τις κλίσεις κάθισ εικονοστοιχείου με από τις σχέσεις:

$$E_s = \sqrt{I_{Gx}^2 + I_{Gy}^2}$$

$$E_o = \arctan\left(\frac{I_{Gy}}{I_{Gx}}\right)$$



```

124 void sobel()
125 {
126     /* Declare Sobel masks */
127     GxMask[0][0] = -1; GxMask[0][1] = 0; GxMask[0][2] = 1;
128     GxMask[1][0] = -2; GxMask[1][1] = 0; GxMask[1][2] = 2;
129     GxMask[2][0] = -1; GxMask[2][1] = 0; GxMask[2][2] = 1;
130
131     GyMask[0][0] = 1; GyMask[0][1] = 2; GyMask[0][2] = 1;
132     GyMask[1][0] = 0; GyMask[1][1] = 0; GyMask[1][2] = 0;
133     GyMask[2][0] = -1; GyMask[2][1] = -2; GyMask[2][2] = -1;
134
135     /* This is for Gx */
136     for (col=limit; col<=(N+limit); col++){
137         for (row=limit; row<=(M+limit); row++){
138             newPixel = 0;
139             for (kernelCol=-limit; kernelCol<=limit; kernelCol++){
140                 for (kernelRow=-limit; kernelRow<=limit; kernelRow++){
141                     newPixel = newPixel + output_gaussian[row+kernelRow][col+kernelCol]*GxMask[limit+kernelRow][limit+kernelCol];
142                 }
143             }
144             Igx[row][col] = (int)(floor(newPixel));
145         }
146     }
147     /* This is for Gy */
148     for (col=limit; col<=(N+limit); col++){
149         for (row=limit; row<=(M+limit); row++){
150             newPixel = 0;
151             for (kernelCol=-limit; kernelCol<=limit; kernelCol++){
152                 for (kernelRow=-limit; kernelRow<=limit; kernelRow++){
153                     newPixel = newPixel + output_gaussian[row+kernelRow][col+kernelCol]*GyMask[limit+kernelRow][limit+kernelCol];
154                 }
155             }
156             Igy[row][col] = (int)(floor(newPixel));
157         }
158     }
159     /* This is for Es */
160     for (col=limit; col<=(N+limit); col++){
161         for (row=limit; row<=(M+limit); row++){
162             Es[row][col] = sqrt(pow(Igx[row][col], 2.0)+pow(Igy[row][col], 2.0));
163
164             if(Es[row][col]>=255)
165             {
166                 Es[row][col] = 255;
167             }
168             //printf("%d ",Es[row][col]);
169         }
170     }
171
172     for (col=limit; col<=(N+limit); col++){
173         for (row=limit; row<=(M+limit); row++){
174             Eo[row][col] = (atan2(Igy[row][col], Igx[row][col])/3.14159) * 180.0;
175         }
176     }
177 }

```

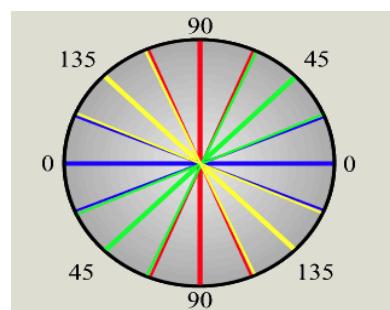
Εικόνα 1.8: Εφαρμογή μασκών Sobel.

Στη συνέχεια γίνεται η καταστολή των μη μέγιστων τιμών.

Ορίζουμε τον πίνακα NonMax στον οποίο αρχικά αποθηκεύουμε ένα αντίγραφο του πίνακα Es. Διατρέχοντας όλα τα εικονοστοιχεία της εικόνας, ελέγχουμε για καθένα από αυτά αν τοπικά υπάρχει εικονοστοιχείο με μικότερη τιμή πλάτους Es. Με τον όρο τοπικά αναφερόμαστε στα γειτονικά εικονοστοιχεία σε ακτίνα ενός εικονοστοιχείου από το κεντρικό. Ο τοπικός έλεγχος γίνεται χωρίζοντας τον χώρο γύρω από το κεντρικό εικονοστοιχείο σε τέσσερις περιοχές, όπως φαίνεται στην δίπλα εικόνα.

Σύμφωνα με την ακίση κάθε εικονοστοιχείου ελέγχουμε και μία από τις τέσσερις περιοχές γύρω του. Αν τα γειτονικά εικονοστοιχεία που βρίσκονται στην εκάστωτε περιοχή έχουν τιμή πλάτους μικρότερη από την τιμή του πλάτους του κτενρικού εικονοστοιχείου τότε θέτουμε τα ανάλογα κελιά του πίνακα NonMax ίσα με μηδέν.

Εικόνα 1.9: Διαχωρισμός τοπικής περιοχής εικονοστοιχείου





```

179 void nonSup() {
180
181     for (col=0; col<N; col++) {
182         for (row=0; row<M; row++) {
183             NonMax[row][col] = Es[row][col];
184         }
185     }
186
187     for (col=limit; col<(N+limit); col++) {
188         for (row=limit; row<(M+limit); row++) {
189
190             //Horizontal Edge
191             if (((-22.5 < Eo[row][col]) && (Eo[row][col] <= 22.5)) || ((157.5 < Eo[row][col]) && (Eo[row][col] <= -157.5)))
192             {
193                 if ((Es[row][col] < Es[row][col+1]) || (Es[row][col] < Es[row][col-1]))
194                 {
195                     NonMax[row][col] = 0;
196                 }
197             }
198
199
200             //Vertical Edge
201             if (((-112.5 < Eo[row][col]) && (Eo[row][col] <= -67.5)) || ((67.5 < Eo[row][col]) && (Eo[row][col] <= 112.5)))
202             {
203                 if ((Es[row][col] < Es[row+1][col]) || (Es[row][col] < Es[row-1][col]))
204                 {
205                     NonMax[row][col] = 0;
206                 }
207             }
208
209
210             //+45 Degree Edge
211             if (((-67.5 < Eo[row][col]) && (Eo[row][col] <= -22.5)) || ((112.5 < Eo[row][col]) && (Eo[row][col] <= 157.5)))
212             {
213                 if ((Es[row][col] < Es[row+1][col+1]) || (Es[row][col] < Es[row-1][col+1]))
214                 {
215                     NonMax[row][col] = 0;
216                 }
217             }
218
219
220             //-45 Degree Edge
221             if (((-157.5 < Eo[row][col]) && (Eo[row][col] <= -112.5)) || ((67.5 < Eo[row][col]) && (Eo[row][col] <= 22.5)))
222             {
223                 if ((Es[row][col] < Es[row+1][col+1]) || (Es[row][col] < Es[row-1][col-1]))
224                 {
225                     NonMax[row][col] = 0;
226                 }
227             }
228
229         }
230     }
231 }
```

Εικόνα 1.10: Συνάρτηση για την καταστολή των μη μέγιστων τιμών.

Μετά γίνεται η διπλή κατωφλίωση υστέρησης. Αρχικά στην συνάρτηση hysteresis ελέγχουμε ποια εικονοστοιχεία έχουν τιμή πλάτους μικρότερη του κατωφλίου T1, ενώ έπειτα ελέγχουμε ποια εικονοστοιχεία έχουν τιμή πλάτους μεγαλύτερη του κατωφλίου T2. Για τα εικονοστοιχεία της πρώτης περίπτωσης σημειώνουμε στην αντίστοιχη θέση του τελικού πίνακα Edges την τιμή 0. Μόλις εντοπίσουμε ένα εικονοστοιχείο της δεύτερης περίπτωσης τότε σημειώνουμε στην αντίστοιχη θέση του τελικού πίνακα Edges την τιμή 255 και καλείται η συνάρτηση neighbor_loop.

```

317 void hysteresis(){
318
319     for (col=limit; col<(N+limit); col++) {
320         for (row=limit; row<(M+limit); row++) {
321             if(NonMax[row][col]<T1)
322             {
323                 Edges[row][col]=0;
324             }
325
326             else if(NonMax[row][col]>T2)
327             {
328                 Edges[row][col]=255;
329                 neighbor_loop(row,col);
330             }
331
332             else
333             {
334                 Edges[row][col]=0;
335             }
336
337         }
338     }
339 }
```

Εικόνα 1.11: Διπλή κατωφλίωση εικονοστοιχείων.



Στην συνάρτηση neighbor_loop υλοποιείται η εξής διαδικασία. Ξεκινώντας από ένα εικονοστοιχείο με τιμή πλάτους μεγαλύτερη του T2, ελέγχεται η περιοχή γύρω του σύμφωνα με την τιμή κλίσης του. Για όποιο εικονοστοιχείο εντοπίζεται με τιμή πλάτους μεγαλύτερη του T1 σημειώνουμε στην αντίστοιχη θέση του πίνακα Edges την τιμή 255 καθώς και στην αντίστοιχη θέση του πίνακα searched την τιμή 1 ώστε αργότερα να μην ελέγξουμε αυτό το εικονοστοιχείο ξανά. Τέλος καλούμε όλες τις συναρτήσεις στην main.

```
236 void neighbor_loop(int Row, int Col){  
237     //Horizontal Edge  
238     if (((-22.5 < Eo[Row][Col]) && (Eo[Row][Col] <= 22.5)) || ((157.5 < Eo[Row][Col]) && (Eo[Row][Col] <= -157.5)))  
239     {  
240         if(NonMax[Row][Col+1]>T1 && searched[Row][Col+1]!=0)  
241         {  
242             Edges[Row][Col+1]=255;  
243             neighbor_loop(Row,Col+1);  
244             searched[Row][Col+1]=1;  
245         }  
246     }  
247     else if(NonMax[Row][Col-1]>T1 && searched[Row][Col-1]!=0)  
248     {  
249         Edges[Row][Col-1]=255;  
250         neighbor_loop(Row,Col-1);  
251         searched[Row][Col-1]=1;  
252     }  
253 }  
254  
255 //Vertical Edge  
256 else if (((-112.5 < Eo[Row][Col]) && (Eo[Row][Col] <= -67.5)) || ((67.5 < Eo[Row][Col]) && (Eo[Row][Col] <= 112.5)))  
257 {  
258     if(NonMax[Row+1][Col]>T1 && searched[Row+1][Col]!=0)  
259     {  
260         Edges[Row+1][Col]=255;  
261         neighbor_loop(Row+1,Col);  
262         searched[Row+1][Col]=1;  
263     }  
264     else if(NonMax[Row-1][Col]>T1 && searched[Row-1][Col]!=0)  
265     {  
266         Edges[Row-1][Col]=255;  
267         neighbor_loop(Row-1,Col);  
268         searched[Row-1][Col]=1;  
269     }  
270 }  
271  
272 //+45 Degree Edge  
273 else if (((-67.5 < Eo[Row][Col]) && (Eo[Row][Col] <= -22.5)) || ((112.5 < Eo[Row][Col]) && (Eo[Row][Col] <= 157.5)))  
274 {  
275     if(NonMax[Row-1][Col+1]>T1 && searched[Row-1][Col+1]!=0)  
276     {  
277         Edges[Row-1][Col+1]=255;  
278         neighbor_loop(Row-1,Col+1);  
279         searched[Row-1][Col+1]=1;  
280     }  
281     else if(NonMax[Row+1][Col-1]>T1 && searched[Row+1][Col-1]!=0)  
282     {  
283         Edges[Row+1][Col-1]=255;  
284         neighbor_loop(Row+1,Col-1);  
285         searched[Row+1][Col-1]=1;  
286     }  
287 }  
288  
289 // -45 Degree Edge  
290 else if (((-157.5 < Eo[Row][Col]) && (Eo[Row][Col] <= -112.5)) || ((67.5 < Eo[Row][Col]) && (Eo[Row][Col] <= 22.5)))  
291 {  
292     if(NonMax[Row+1][Col+1]>T1 && searched[Row+1][Col+1]!=0)  
293     {  
294         Edges[Row+1][Col+1]=255;  
295         neighbor_loop(Row+1,Col+1);  
296         searched[Row+1][Col+1]=1;  
297     }  
298     else if(NonMax[Row-1][Col-1]>T1 && searched[Row-1][Col-1]!=0)  
299     {  
300         Edges[Row-1][Col-1]=255;  
301         neighbor_loop(Row-1,Col-1);  
302         searched[Row-1][Col-1]=1;  
303     }  
304 }  
305  
306 return;  
307 }  
308  
309 }  
310  
311 }  
312  
313 }  
314  
315 }
```

Εικόνα 1.12: Κατωφλίωση υστέρησης.



1.3 Πρόγραμμα Arm Development Suite

Για την επίτευξη της χρονικής βελτιστοποίησης του αλγορίθμου χρησιμοποιήθηκε το πρόγραμμα Arm Development Suite. Μετά την εκτέλεση ενός κώδικα επιλέγοντας το κουμπί make εμφανίζεται ο παραχάτω πίνακας:

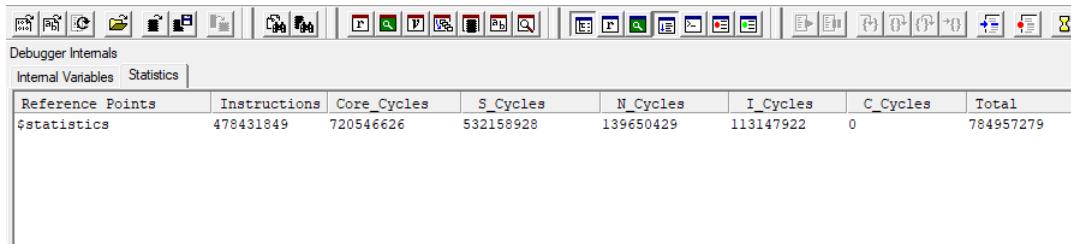
Image component sizes					
	Code	RO Data	RW Data	ZI Data	Debug
6336	60	16	4749076	9700	Object Totals
22356	610	0	300	7624	Library Totals

	Code	RO Data	RW Data	ZI Data	Debug
28692	670	16	4749376	17324	Grand Totals

	Total RO Size(Code + RO Data)	(28.67kB)
Total RW Size(RW Data + ZI Data)	4749392	(4638.08kB)
Total ROM Size(Code + RO Data + RW Data)	29378	(28.69kB)

Εικόνα 1.13: Επιλογή make.

Τα Object Totals αναφέρονται στα bytes που οφείλονται στον κώδικα του προγραμματιστή, ενώ τα Library Totals στα bytes που οφείλονται στις βιβλιοθήκες που συμπεριλήφθηκαν. Στην στήλη Code αναγράφεται ο αριθμός σε bytes που καταλαμβάνει ο κώδικας στην μνήμη. Η στήλη RO data (Read Only data) είναι αρχικοποιημένη στο 60. Με την χρήση μεταβλητών που είναι μόνο για ανάγνωση (πχ τύπου const) ο αριθμός αυτός αυξάνεται. Η στήλη RW data (Read Write data) αφορά επαναγράψιμες αρχικοποιημένες μεταβλητές. Δείχνει τον αριθμό σε bytes που καταλαμβάνουν αυτές οι μεταβλητές. Η στήλη ZI data (Zero Initialized data) αναγράφει το μέγεθος σε bytes των μη αρχικοποιημένων επαναγράψιμων μεταβλητών. Η στήλη Debug αφορά δεδομένα που οφείλονται στο debugging. Στην γραμμή Grand Totals απεικονίζεται το άθροισμα των Object Totals και Library Totals. Τέλος στην τελευταία γραμμή μπορούμε να δούμε το μέγεθος της μνήμης ROM που απαιτείται.



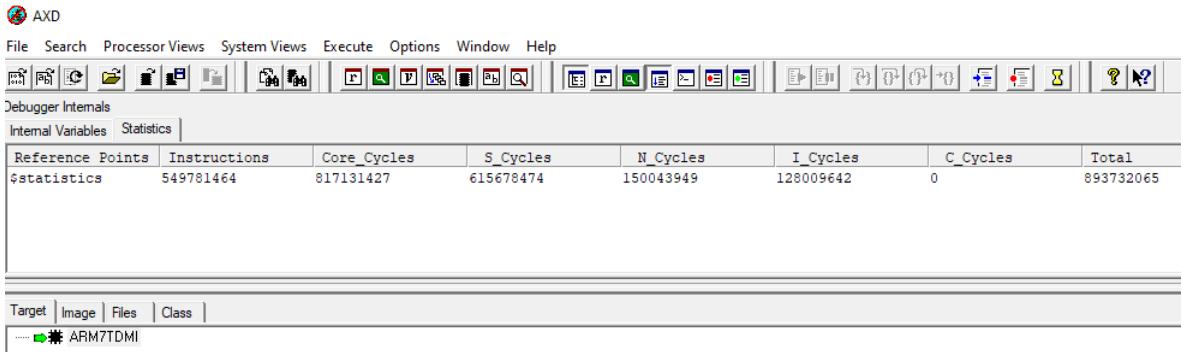
Εικόνα 1.14: AXD.

Μετά την εκτέλεση του προγράμματος ανοίγουμε το πρόγραμμα AXD Debugger. Από αυτό μπορούμε να δούμε τους κύκλους εκτέλεσης του προγράμματος. Το Instructions μας δείχνει τον αριθμό εντολών σε Assembly, ενώ το Core Cycles τον συνολικό αριθμό κύκλων στον επεξεργαστή. Το S Cycles μας δείχνει τον αριθμό των κύκλων στους οποίους γίνονται διαδοχικές (sequential) κλήσεις στην μνήμη. Σε αυτούς τους κύκλους η τιμή του program counter αυξάνεται κατά ένα. Από την άλλη το N Cycles απεικονίζει τους κύκλους στους οποίους γίνονται μη διαδοχικές (non sequential) κλήσεις στην μνήμη. Το I Cycles δείχνει τους κύκλους στους οποίους ο επεξεργαστής δεν κάνει κλήση (call) στην μνήμη. Τέλος το C Cycles αφορά τη χρήση βοηθητικού επεξεργαστή. Βασικός στόχος είναι η μείωση του αριθμού Core Cycles. Επιπλέον είναι σημαντική η μετατροπή των N Cycles σε S Cycles καθώς μέσω αυτής μειώνεται ο χρόνος εκτέλεσης του προγράμματος.



1.4 Μέθοδοι Βελτιστοποίησης

Εκτελώντας το αρχικό πρόγραμμα χωρίς τις μεθόδους βελτιστοποίησης τα αποτελέσματα τα οποία παίρνουμε είναι τα εξής:



Εικόνα 1.15: Αποτελέσματα χωρίς βελτιστοποιήσεις.

1.4.1 Loop Unrolling

Η μέθοδος του ξετυλίγματος βρόγχου είναι μια από τις τεχνικλες μετασχηματισμού βρόγχου που προσπαθούν να βελτιστοποιήσουν την ταχύτητα εκτέλεσης ενός προγράμματος. Ο αλγόριθμος μπορεί να υλοποιηθεί χειροκίνητα από τον προγραμματιστή ή από έναν compiler βελτιστοποίησης. Στους σύγχρονους επεξεργαστές η τεχνική αυτή είναι συνήθως αντιπαραγωγική καθώς με το ξελύτιγμα αυξάνονται οι απαιτήσεις σε μνήμη. Ουσιαστικά για να μειωθούν οι κύκλοι του επεξεργαστή αυξάνονται οι απαιτήσεις σε μνήμη ROM καθώς γράφουμε περισσότερο κώδικα. Η μείωση των κύκλων του επεξεργαστή είναι λογική. Έστω ότι σε μια for το βήμα διπλασιαστεί και γίνουν αντίστοιχα δύο αναθέσεις σε κάθε εκτέλεση. Μετά την μεταγλώττιση του προγράμματος σε γλώσσα Assembly η εντολή branch θα εκτελεστεί τις μισές φορές σε σχέση με πριν.

Υλοποίηση σε κώδικα

Ενδεικτικά αναλύουμε την υλοποίηση της τεχνικής loop unroll στην σάρωση των εικονοστοιχείων της συνάρτησης gaussian-blur. Μετατρέπουμε την for απανάληψη που αφορά τις γραμμές ώστε να τις σαρώνει ανά τέσσερις, δηλαδή ο δείκτης της αυξάνεται κάθε φορά κατά 4. Εσωτερικά, υπολογίζουμε την νέα τιμή τεσσάρων εικονοστοιχείων. Δηλαδή για κάθε επανάληψη της for που αφορά τις γραμμές εκτελούνται εντολές που διαφορετικά θα εκτελούνταν σε τέσσερις επαναλήψεις της συγκεκριμένης for.

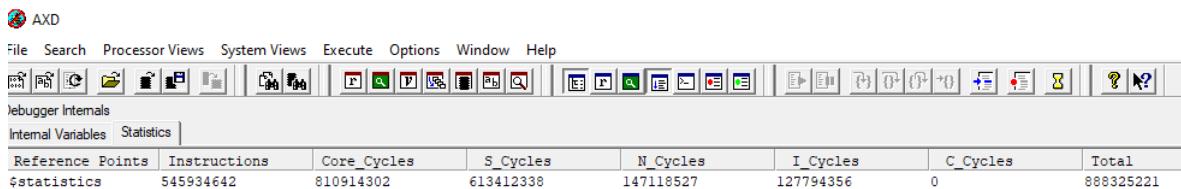


```
109 for (col=limit_gauss,col<=(N+limit_gauss);col++){
110   for (row=limit_gauss;row<=(M+limit_gauss);row+=4){
111
112     newPixel = 0;
113     newPixel2 = 0;
114     newPixel3 = 0;
115     newPixel4 = 0;
116
117     for (kernelCol=-limit_gauss;kernelCol<=limit_gauss; kernelCol++){
118       for (kernelRow=-limit_gauss; kernelRow<=limit_gauss; kernelRow++){
119
120         newPixel = newPixel + temp[row+kernelRow][col+kernelCol]*gaussianMask[limit_gauss+kernelRow][limit_gauss+kernelCol];
121         newPixel2 = newPixel2 + temp[row+kernelRow+1][col+kernelCol]*gaussianMask[limit_gauss+kernelRow][limit_gauss+kernelCol];
122         newPixel3 = newPixel3 + temp[row+kernelRow+2][col+kernelCol]*gaussianMask[limit_gauss+kernelRow][limit_gauss+kernelCol];
123         newPixel4 = newPixel4 + temp[row+kernelRow+3][col+kernelCol]*gaussianMask[limit_gauss+kernelRow][limit_gauss+kernelCol];
124
125     }
126   }
127 }
```

Εικόνα 1.16: Ενδεικτική υλοποίηση της μεθόδου Loop Unrolling.

Αποτελέσματα

Παρακάτω φαίνεται το αποτέλεσμα στην βελτίωση του χρόνου εκτέλεσης που είχε η εφαρμογή της μεθόδου.



Εικόνα 1.17: Αποτελέσματα βελτιστοποίησης με Loop Unrolling.

Η βελτίωση έγκειται στο γεγονός ότι το πλήθος των επαναλήψεων σάφωσης γραμμών έχει μειωθεί στο ένα τέταρτο του αρχικού. Άρα και το overhead της συγκεκριμένης for έχει μειωθεί αντίστοιχα.

1.4.2 Loop Fusion

Στην επιστήμη των υπολογιστών η σύντηξη βρόγχων είναι τεχνική με την οποία μπορεί να επιτευχθεί βελτιστοποίηση χρόνου. Αφορά την αντικατάσταση πολλών βρόγχων με έναν. Είναι πιθανόν σε ένα πρόγραμμα δύο ή περισσότεροι βρόγχοι να έχουν ίδιο εύρος και μην υπάρχουν εξαρτήσεις μεταξύ των δεδομένων τους. Σε αυτή την περίπτωση είναι η σύντηξη αυτών των βρόγχων μπορεί να δώσει πολύ καλύτερα αποτελέσματα αναφορικά με τον χρόνο εκτέλεσης του προγράμματος.

Υλοποίηση σε κώδικα

Ενδεικτικά αναλύουμε την υλοποίηση της τεχνικής loop fusion στην σάφωση των εικονοστοιχείων της συνάρτησης sobel και πιο συγκεκριμένα στον υπολογισμό των Igx, Igy. Αρχικά ο



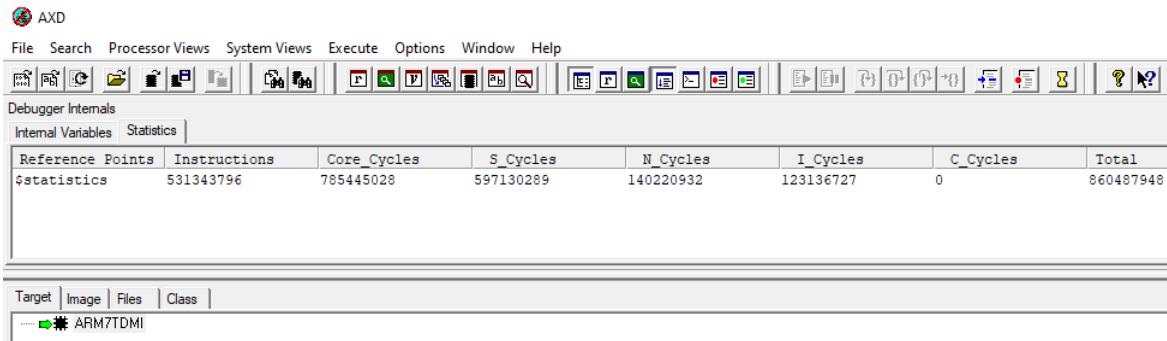
υπολογισμός τους εκτελούνται σε δύο χωριστές ομάδες επαναλήψεων. Εφόσον αυτές έχουν τα ίδια όρια μπορούν να ενωθούν σε μία ομάδα, μέσα στην οποία γίνεται ο υπολογισμός των Igx,Igy.

```
133 /* This is for Gx */
134 for (col=limit;col<=(M+limit);col++){
135     for (row=limit;row<=(M+limit);row++){
136
137         newPixelx = 0;
138         newPixely = 0;
139         for (kernelCol=-limit;kernelCol<=limit; kernelCol++){
140             for (kernelRow=-limit; kernelRow<=limit; kernelRow++){
141
142                 newPixelx = newPixelx + output_gaussian[row+kernelRow][col+kernelCol]*GxMask[limit+kernelRow][limit+kernelCol];
143                 newPixely = newPixely + output_gaussian[row+kernelRow][col+kernelCol]*GyMask[limit+kernelRow][limit+kernelCol];
144             }
145         }
146         Igx[row][col] = (int)(floor(newPixelx));
147         Igy[row][col] = (int)(floor(newPixely));
148
149         Es[row][col] = sqrt(pow(Igx[row][col], 2.0)+pow(Igy[row][col], 2.0));
150
151         if(Es[row][col]>=255)
152         {
153             Es[row][col] = 255;
154         }
155
156         Eo[row][col] = (atan2(Igy[row][col],Igx[row][col])/3.14159) * 180.0;
157
158     }
159 }
160 }
```

Εικόνα 1.18: Κώδικας υλοποίησης της μεθόδου Loop Fusion.

Αποτελέσματα

Παρακάτω φαίνεται το αποτέλεσμα στην βελτίωση του χρόνου εκτέλεσης που είχε η εφαρμογή της μεθόδου.



Εικόνα 1.19: Αποτελέσματα βελτιστοποίησης με Loop Fusion.

Η βελτίωση έγκειται στο γεγονός ότι δύο ομάδες for επαναλήψεων αντικαθιστώνται με μία και έτσι εξαφανίζεται και το Overhead της μίας ομάδας for επαναλήψεων.

1.4.3 Loop Interchange

Η τεχνική της εναλλαγής βρόγχου είναι επίσης μια πολύ καλή τεχνική για την μείωση του χρόνου εκτέλεσης. Ο βασικός σκοπός αυτής της τεχνικής είναι να εκμεταλλευτεί τον τρόπο αποθήκευσης των στοιχείων ενός πίνακα στην μνήμη της CPU cache. Πολύ σημαντικό ρόλο για την μείωση του χρόνου εκτέλεσης ενός προγράμματος είναι η κλήσεις στην μνήμη να είναι διαδοχικές (sequential).



Η γλώσσα C είναι μια γλώσσα row major. Αυτό σημαίνει ότι στην μνήμη cache τα στοιχεία ενός πίνακα αποθηκεύονται κατά γραμμές. Δηλαδή μπαίνουν πρώτα όλες οι στήλες της πρώτης γραμμής ενός πίνακα, ακολουθούν όλες οι στήλες της δεύτερης γραμμής και η διαδικασία αυτή συνεχίζεται μέχρι να αποθηκευτούν όλα τα στοιχεία του πίνακα. Για την υλοποίηση του αλγορίθμου εξαγωγής ακμών του Canny απαιτήθηκε πολλές φορές η χρήση εμφωλευμένων for. Για να γίνουν διαδοχικές κλήσεις στην μνήμη cache πρέπει η εξωτερική for να διατρέχει τις γραμμές ενός πίνακα και η εσωτερική for τις στήλες. Αν υλοποιηθούν με την αντίθετη σειρά γίνονται μη διαδοχικές κλήσεις (non sequential) κλήσεις στην μνήμη και ο χρόνος εκτέλεσης του προγράμματος αυξάνεται.

Υλοποίηση σε κώδικα

Ενδεικτικά θα δείξουμε την υλοποίηση της μεθόδου loop interchange στην συνάρτηση gaussianBlur. Σε κάθε ομάδα for επαναλήψεων οι οποίες σαρώνουν κάποιον πίνακα, αντιστρέφουμε τα όριά τους ώστε η σάρωση να γίνεται κατά γραμμές.

```
109     for (row=limit_gauss;row<=(N+limit_gauss);row++){
110         for (col=limit_gauss;col<=(N+limit_gauss);col++){
111             newPixel = 0;
112             for (kernelRow=-limit_gauss; kernelRow<=limit_gauss; kernelRow++){
113                 for (kernelCol=-limit_gauss;kernelCol<=limit_gauss; kernelCol++){
114                     newPixel = newPixel + temp[row+kernelRow][col+kernelCol]*gaussianMask[limit_gauss+kernelRow][limit_gauss+kernelCol];
115                 }
116             }
117             output_gaussian[row][col] = (int)(floor(newPixel)/16);
118         }
119     }
120 }
121 }
```

Εικόνα 1.20: Κώδικας υλοποίησης της μεθόδου Loop Interchange.

Αποτελέσματα

Παρακάτω φαίνεται το αποτέλεσμα στην βελτίωση του χρόνου εκτέλεσης που είχε η εφαρμογή της μεθόδου.

Debugger Internals								
Internal Variables		Statistics						
Reference ...	Instructions	Core_Cycles	S_Cycles	N_Cycles	I_Cycles	C_Cycles	Total	
\$statistics	540510251	804328750	606264874	147880964	124192397	0	878338235	

Εικόνα 1.21: Αποτελέσματα βελτιστοποίησης με Loop Interchange.

Η βελτίωση έγκειται στο γεγονός ότι στην C οι πίνακες αποθηκεύονται στην μνήμη κατά στείλες, άρα γειτονικά στοιχεία μίας γραμμής ενός πίνακα είναι αποθηκευμένα το ένα δίπλα στο άλλο. Οπότε η προσπέλασή τους απαιτεί μόνο την άυξηση του δείκτη του πίνακα κατά ένα.



1.4.4 Loop Collapsing

Εμφωλευμένες for επαναλήψεις μπορούν να μετατραπούν σε μία για να μειωθεί το overhead της συνολικής επανάληψης και έτσι να βελτιωθεί ο χρόνος εκτέλεσης. Η τελική επανάληψη που θα προκύψει πρέπει να ρυθμίζεται ώστε να εκτελεί το πλήθος των επαναλήψεων που εκτελεί να είναι ίδιο με αυτό του συνόλου των επαναλήψεων που αντικαθιστά. Κλήση σε στοιχεία πινώκων εντός της επανάληψης γίνεται με δείκτες οι οποίοι αυξάνονται μετά το πέρας κάθε επανάληψης.

Υλοποίηση σε κώδικα

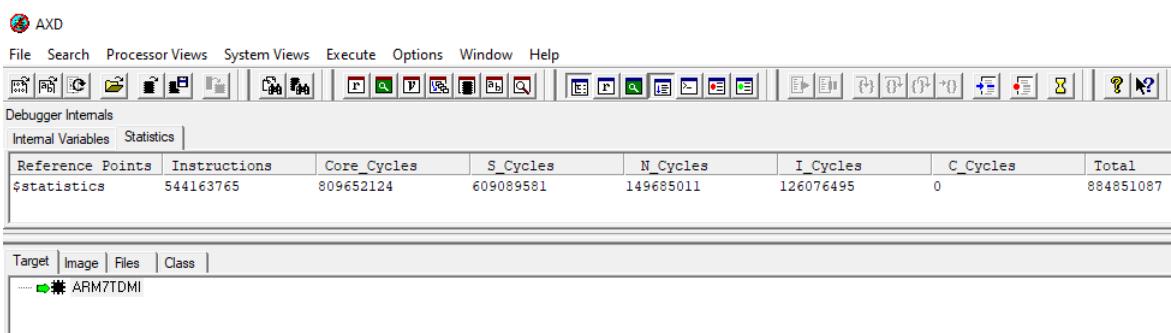
Ενδεικτικά αναλύουμε την υλοποίηση της μεθόδου loop collapsing στον υπολογισμό των τιμών του πλάτους Es και της κλίσης Eo των εικονοστοιχείων. Αρχικά ο υπολογισμός γινόταν με δύο for επαναλήψεις οι οποίες σάρωναν όλα τα εικονοστοιχεία. Μετατρέπουμε τις δύο for σε μία, η οποία όμως εκτελεί το ίδιο πλήθος επαναλήψεων που εκτελεί και ομάδα των δύο for. Αναφορά στις μεταβλητές που χρειάζονται για τους υπολογισμούς γίνεται με την χρήση δεικτών. Έτσι με κάθε επανάληψη αυξάνονται όλοι οι δείκτες και δείχνουν στην αμέσως επόμενη θέση στη μνήμη.

```
171 }  
172  
173  
174  
175  
176 }  
177  
178  
179  
180  
181  
182  
183  
184  
185 }  
  
for (i=limit;i<=N*M;i++) {  
    *s = sqrt(pow(*igx, 2.0)+pow(*igy, 2.0));  
    *o = atan2(*igy,*igx)/3.14159 * 180;  
  
    if (*s>=255)  
    {  
        *s = 255;  
    }  
    *s++;  
    *o++;  
    *igx++;  
    *igy++;  
}
```

Εικόνα 1.22: Κώδικας υλοποίησης loop collapsing.

Η βελτίωση έγκειται στο γεγονός ότι πλέον έχουμε μία for αντί για δύο. Άρα το overhead της δεύτερης for δεν συμπεριλαμβάνεται.

Αποτελέσματα



Εικόνα 1.23: Αποτελέσματα βελτιστοποίησης με loop collapsing.



1.4.5 Loop Inversion

Στην επιστήμη των υπολογιστών η τεχνική της αναστροφής βρόγχου είναι μια ακόμα τεχνική για την βελτιστοποίηση του χρόνου εκτέλεσης ενός προγράμματος. Η βελτιστοποίηση που μπορεί να επιτευχθεί με αυτή την τεχνική είναι πολύ μικρή. Ένα παράδειγμα υλοποίησής της αποτελεί η μετατροπή μιας while σε do while. Με αυτό τον τρόπο ο έλεγχος συνθήκης γίνεται στο τέλος. Έτσι όταν φτάσουμε στη συνθήκη τερματισμού δεν γίνεται jump και γλιτώνουμε ουσιαστικά μια εκτέλεση της εντολής branch σε assembly.

Υλοποίηση σε κώδικα

Ενδεικτικά αναλύουμε την υλοποίηση της μεθόδου loop inversion στην αρχικοποίηση του πίνακα temp στον οποίο δημιουργείται το περίγραμμα του ενός εικονοστοιχείου γύρω από την εικόνα. Εν πρώτοις η αρχικοποίηση γίνεται με την χρήση μίας while επανάληψης μέσα σε μία for επανάληψη. Επιλέγουμε τον συνδιασμό for-while αποκλειστικά για την επίδειξη της μεθόδου. Μετατρέπουμε την while σε do-while. Ο πρώτος έλεγχος γίνεται αμέσως πριν το do με μία if, στη συνέχεια ακολουθούν οι εντολές της αρχικοποίησης κάθε στήλης του πίνακα temp και τέλος γίνεται ο έλεγχος της while.

```
88 |     for(i=0;i<(M+2);++i)
89 |     {
90 |     |     if(j<(N+2)){
91 |     |     |     do{
92 |     |     |     |     temp[i][j]=0;
93 |     |     |     |     j++;
94 |     |     |     }while(j<(N+2));
95 |     |
96 |     }
```

Εικόνα 1.24: Κώδικας υλοποίησης μεθόδου loop inversion.

Αποτελέσματα

Παρακάτω φαίνεται το αποτέλεσμα στην βελτίωση του χρόνου εκτέλεσης που είχε η εφαρμογή της μεθόδου.

Debugger Internals							
Internal Variables		Statistics					
Reference ...	Instructions	Core_Cycles	S_Cycles	N_Cycles	I_Cycles	C_Cycles	Total
\$statistics	548936311	815614407	614964106	149350710	127913661	0	892228477

Εικόνα 1.25: Αποτελέσματα βελτιστοποίησης με loop inversion.

Η βελτίωση έγκειται στο γεγονός ότι ο έλεγχος εξόδου γίνεται στο τέλος και έτσι δεν χρειάζεται να εκτελεστεί η τελευταία branch εντολή σε επίπεδο assembler.



1.4.6 Loop Tiling

Χρησιμοποιείται σε εφαρμογές οι οποίες λειτουργούν με πολύ μεγάλα σύνολα δεδομένων. Ο στόχος είναι να φορτώσουμε ένα υποσύνολο των δεδομένων (tile) στην cache μνήμη και να κάνουμε όλες τις πράξεις που χρειάζεται σε αυτά τα δεδομένα πριν χρησιμοποιήσουμε ένα νέο υποσύνολο των δεδομένων. Ανάλογα με τις πράξεις που γίνονται καθώς και με την δομή του συνόλου δεδομένων, μία επανάληψη μπορεί να χρειαστεί να προσπελάσει διαφορετικά υποσύνολα δεδομένων. Έτσι, εφόσον είναι αδύνατον να είναι φορτωμένο στην cache μνήμη το σύνολο των δεδομένων, προκαλείται ένα πλήθος cache miss περιστατικών ενώ επίσης χρειάζεται συνεχώς να φορτώνουμε στην μνήμη διαφορετικά υποσύνολα των δεδομένων. Όλα αυτά οδηγούν ένα σύστημα να ξοδεύει άσκοπα κύκλους για τις εργασίες του. Με προσεκτικό προγραμματισμό της σειράς εκτέλεσης των εντολών μπορεί ο χρόνος εκτέλεσης να μειωθεί σημαντικά για κάποιες εφαρμογές.

Υλοποίηση σε κώδικα

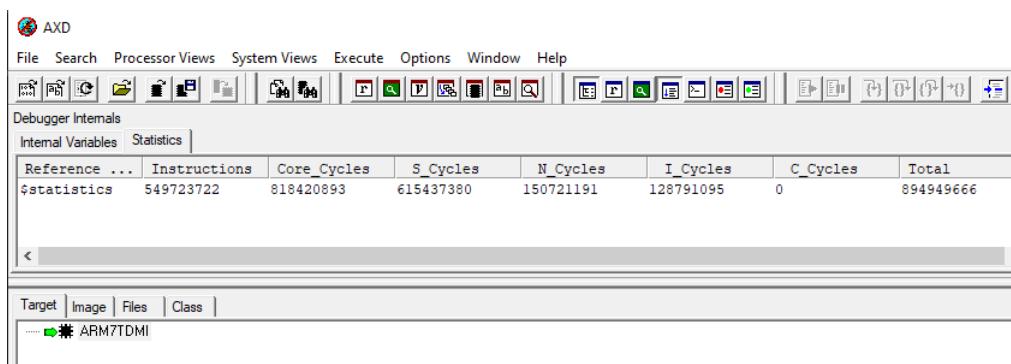
Ενδεικτικά αναλύουμε την υλοποίηση της μεθόδου loop tiling στον υπολογισμό των τιμών του πλάτους Es των εικονοστοιχείων στην συνάρτηση Sobel. Αρχικά ο υπολογισμός γινόταν με χρήση μίας διπλής for για την σάρωση όλων των εικονοστοιχείων. Αντικαθυστούμε τις δύο for με τέσσερις, οι οποίες χωρίζουν τους πίνακες που σαρώνονται κατά τον υπολογισμό των Es σε τέσσερα κομμάτια. Έτσι κάθε ένα κομμάτι σαρώνεται χωριστά.

```
165     for (colout=0;colout<=1;colout++){
166         for (rowout=0;rowout<=1;rowout++){
167             for (col=limit+((N/2)*colout);col<=((N)/2+((N/2)*colout));col++){
168                 for (row=limit+((M/2)*rowout);row<=((M)/2+((M/2)*rowout));row++){
169                     Es[row][col] = sqrt(pow(Igx[row][col], 2.0)+pow(Igy[row][col], 2.0));
170
171                     if(Es[row][col]>=255)
172                     {
173                         Es[row][col] = 255;
174                     }
175     }
```

Εικόνα 1.26: Κώδικας βελτιστοποίησης με Loop Tilling.

Αποτελέσματα

Παρακάτω φαίνεται το αποτέλεσμα στην βελτίωση του χρόνου εκτέλεσης που είχε η εφαρμογή της μεθόδου.



Εικόνα 1.27: Αποτελέσματα βελτιστοποίησης με loop tiling.



Η μέθοδος αυτή δεν αποδίδει στην συγκεκριμένη εφαρμογή που υλοποιείται στην εργασία. Ο μέθοδος loop tiling όπως αναφέρθηκε και πριν βελτιστοποιεί τον χρόνο εκτέλεσης αναφορικά με την μνήμη cache που διαθέτουμε καθώς κάθε υποομάδα δεδομένων είναι ολόκληρη φορτωμένη στην μνήμη και άμεσα διαθέσημη. Έτσι μειώνονται οι κύκλοι κλήσης στην μνήμη. Στα πλαίσια της εργασίας όμως διαθέτουμε άπειρη θεωρητικά μνήμη, άρα η συγκεκριμένη μέθοδος δεν μπορεί να προσφέρει κάποια βελτίωση αφού όλα τα δεδομένα είναι άμεσα διαθέσιμα. Αντίθετα αυξάνει τον χρόνο εκτέλεσης αφού αυξάνονται και οι for επαναλήψεις που χρησιμοποιούμε, άρα και το συνολικό overhead.

1.4.7 Αντικατάσταση της συνάρτησης pow

Η συνάρτηση pow στην C έχει έναν γενικό αλγόριθμο ώστε να υλοποιεί έναν αριθμό σε οποιαδήποτε δύναμη. Στον κώδικα που υλοποιήθηκε η συνάρτηση αυτή χρησιμοποιήθηκε για τον υπολογισμό του πίνακα Es μετά την εφαρμογή της μάσκας Sobel. Επειδή η δύναμη που απαιτήθηκε είναι το 2, αντικαθιστώντας αυτή την συνάρτηση με την πράξη του πολλαπλασιασμού παρατηρήθηκε μια ικανοποιητική μείωση των συνολικών κύκλων του επεξεργαστή.

Υλοποίηση σε κώδικα

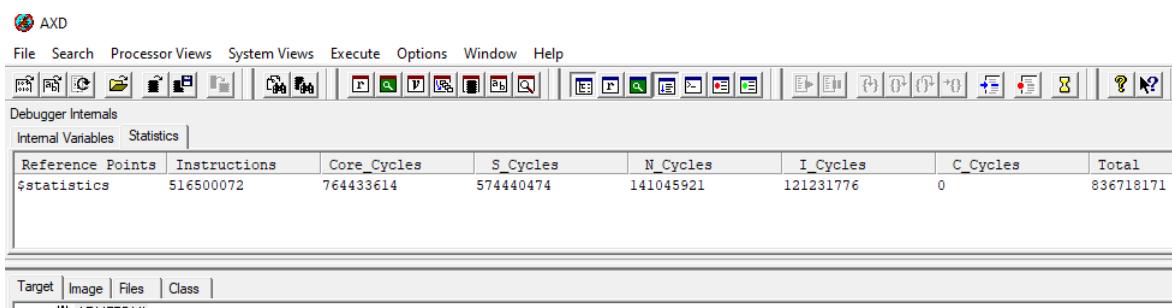
Ενδεικτικά αναλύουμε την υλοποίηση της μεθόδου αυτής στον υπολογισμό των τιμών του πλάτους Es των εικονοστοιχείων στην συνάρτηση Sobel. Ο υπολογισμός αρχικά γινόταν χριστι- μοποιώντας την συνάρτηση pow της βιβλιοθήκης math. Αντικαθηστούμε την συνάρτηση πολ- λαπλασιάζοντας το στοιχείο, το οποίο θέλουμε να υψήσουμε στο τετράγωνο, με τον εαυτό του.

```
162 |     |     | Es[row][col] = sqrt((Ixg[row][col]*Ixg[row][col])+(Igy[row][col]*Igy[row][col]));  
163 |
```

Εικόνα 1.28: Κώδικας μετά την αφαίρεση της συνάρτησης pow.

Αποτελέσματα

Παρακάτω φαίνεται το αποτέλεσμα στην βελτίωση του χρόνου εκτέλεσης που είχε η εφαρμογή της μεθόδου.



Εικόνα 1.29: Αποτέλεσμα βελτιστοποίησης με αφαίρεση της pow.



1.4.8 Συνδιασμός μεθόδων βελτιστοποίησης

Τέλος γίνεται συνδιασμός όλων των προηγούμενων μεθόδων με στόχο τον ελάχιστο δυνατό χρόνο εκτέλεσης. Το αποτέλεσμα φάινεται παρακάτω.

Υλοποίηση σε κώδικα

Γίνεται εφαρμογή όλων των προηγούμενων μεθόδων σε διάφορα σημεία του κώδικα. Επίσης όπου ήταν δυνατόν έγινε και συνδιασμός δύο μεθόδων στην ίδια ομάδα for επαναλήψεων. Συνολικά στον κώδικα, σε κάθε for επανάληψη εφαρμόστηκε loop interchange και η σάρωση των πινάκων πλέον γίνεται κατά γραμμές. Στην συνάρτηση gaussian_blur εφαρμόστηκε η μέθοδος loop unroll όπως είδαμε και προηγουμένος. Στην συνάρτηση Sobel, σχετικά με τον υπολογισμό των Igx,Igy εφαρμόστηκε συνδιασμός των μεθόδων loop fusion και loop unroll. Αρχικά οι δύο ξεχωριστές ομάδες for επαναλήψεων που υπολόγιζαν τα Igx και Igy αντίστοιχα συγχονεύτικαν σε μία αφού είχαν ίδια όρια. Εσωτερικά αυτής έγινε loop unroll της επανάληψης που αφορούσε την σάρωση των στηλών.

```
155 // compute Igx and Igy
156 for (row=limit;row<=(N+limit);row++){
157     for (col=limit;col<=(N+limit);col+=4){
158         newPixel = 0;
159         newPixel2 = 0;
160         newPixel3 = 0;
161         newPixel4 = 0;
162
163         newPixelx = 0;
164         newPixel2y = 0;
165         newPixel3y = 0;
166         newPixel4y = 0;
167
168         for (kernelRow=-limit; kernelRow<=limit; kernelRow++){
169             for (kernelCol=-limit;kernelCol<=limit; kernelCol++){
170                 newPixel = newPixel + output_gaussian[row+kernelRow][col+kernelCol]*GxMask[limit+kernelRow][limit+kernelCol];
171                 newPixel2 = newPixel2 + output_gaussian[row+kernelRow][col+kernelCol+1]*GxMask[limit+kernelRow][limit+kernelCol];
172                 newPixel3 = newPixel3 + output_gaussian[row+kernelRow][col+kernelCol+2]*GxMask[limit+kernelRow][limit+kernelCol];
173                 newPixel4 = newPixel4 + output_gaussian[row+kernelRow][col+kernelCol+3]*GxMask[limit+kernelRow][limit+kernelCol];
174
175                 newPixelx = newPixelx + output_gaussian[row+kernelRow][col+kernelCol]*GyMask[limit+kernelRow][limit+kernelCol];
176                 newPixel2y = newPixel2y + output_gaussian[row+kernelRow][col+kernelCol+1]*GyMask[limit+kernelRow][limit+kernelCol];
177                 newPixel3y = newPixel3y + output_gaussian[row+kernelRow][col+kernelCol+2]*GyMask[limit+kernelRow][limit+kernelCol];
178                 newPixel4y = newPixel4y + output_gaussian[row+kernelRow][col+kernelCol+3]*GyMask[limit+kernelRow][limit+kernelCol];
179
180             }
181         }
182         Igx[row][col] = (int)(floor(newPixel));
183         Igx[row][col+1] = (int)(floor(newPixel2));
184         Igx[row][col+2] = (int)(floor(newPixel3));
185         Igx[row][col+3] = (int)(floor(newPixel4));
186
187         Igy[row][col] = (int)(floor(newPixelx));
188         Igy[row][col+1] = (int)(floor(newPixel2y));
189         Igy[row][col+2] = (int)(floor(newPixel3y));
190         Igy[row][col+3] = (int)(floor(newPixel4y));
191     }
192 }
```

Εικόνα 1.30: Κώδικας υλοποίησης συνολικής βελτιστοποίησης.



Επίσης στην συνάρτηση Sobel, στο σημείο υπολογισμού των τιμών του πλάτους Es και της κλίσης Eo εφαρμόστηκε συνδιασμός των μεθόδων loop fussion, loop collapsing και loop unroll. Αρχικά οι δύο ομάδες επαναλήψεων γίνονται μία. Στη μία ομάδα for που προέκυψε έχουμε δύο for επαναλήψεις με τα ίδια όρια, άρα γίνονται μία εφαρμόζοντας loop collapsing. Τέλος η μία for επανάληψη που έμεινε ρυθμίζεται ώστε ο δέικτης της να αυξάνεται κατά 4 κάθε φορά, ενώ εδωτερικά της υπολογίζονται οι τιμές των Es, Eo για τέσσερα εικονοστοιχεία κάθε φορά.

```
193
194
195 for (i=0;i<=N*M;i+=4) {
196     *s = sqrt((*igx)*(*igx)+(*igy)*(*igy));
197     *o = atan2(*igx,*igy)/3.14159 * 180;
198
199     if(*s>=255)
200                {
201            *s = 255;
202            }
203        *s++;
204        *o++;
205        *igx++;
206        *igy++;
207
208        *s = sqrt((*igx)*(*igx)+(*igy)*(*igy));
209        *o = atan2(*igx,*igy)/3.14159 * 180;
210
211     if(*s>=255)
212                {
213            *s = 255;
214            }
215        *s++;
216        *o++;
217        *igx++;
218        *igy++;
219
220        *s = sqrt((*igx)*(*igx)+(*igy)*(*igy));
221        *o = atan2(*igx,*igy)/3.14159 * 180;
222
223     if(*s>=255)
224                {
225            *s = 255;
226            }
227        *s++;
228        *o++;
229        *igx++;
230        *igy++;
231
232        *s = sqrt((*igx)*(*igx)+(*igy)*(*igy));
233        *o = atan2(*igx,*igy)/3.14159 * 180;
234
235     if(*s>=255)
236                {
237            *s = 255;
238            }
239        *s++;
240        *o++;
241        *igx++;
242        *igy++;
243    }
244 }
```

Εικόνα 1.31: Κώδικας υλοποίησης συνολικής βελτιστοποίησης.

Τέλος για την αρχικοποίηση του πίνακα NonMax και στην συνάρτηση hysteresis εφαρμόστηκε η μέθοδος loop unroll.



```
249
250
251
252
253
254
255
256
257
258      for (col=0; col<N; col++) {
          for (row=0; row<M; row+=4) {

              NonMax[row][col] = Es[row][col];
              NonMax[row+1][col] = Es[row+1][col];
              NonMax[row+2][col] = Es[row+2][col];
              NonMax[row+3][col] = Es[row+3][col];

          }
      }
```

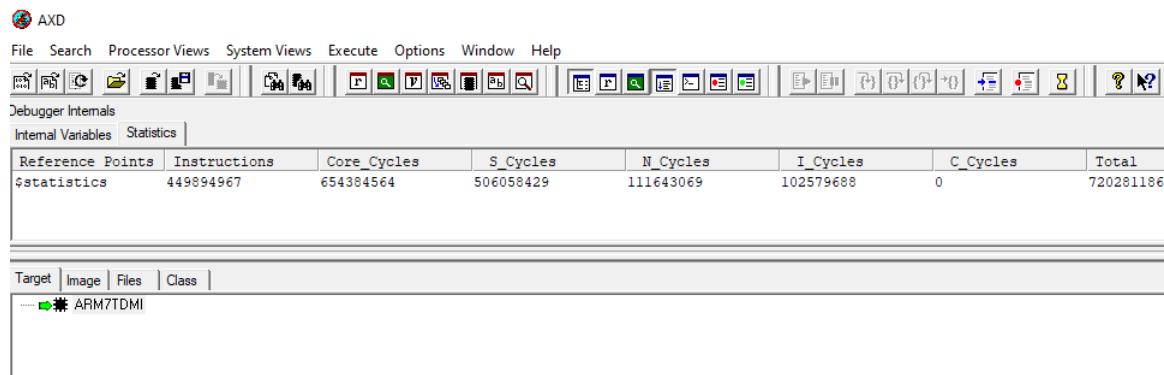
Εικόνα 1.32: Αρχικοποίηση NonMax.

```
383 void hysteresis(){
384
385     for (row=0; row<(M); row++) {
386         for (col=0; col<(N); col+=4)
387         {
388
389             if(NonMax[row][col]<T1)
390             {
391                 Edges[row][col]=0;
392             }
393
394             else if(NonMax[row][col]>T2)
395             {
396                 Edges[row][col]=255;
397                 neighbor_loop(row+1,col+1);
398             }
399
400             if(NonMax[row][col+1]<T1)
401             {
402                 Edges[row][col+1]=0;
403             }
404
405             else if(NonMax[row][col+1]>T2)
406             {
407                 Edges[row][col+1]=255;
408                 neighbor_loop(row+1,col+2);
409             }
410
411             if(NonMax[row][col+2]<T1)
412             {
413                 Edges[row][col+2]=0;
414             }
415
416             else if(NonMax[row][col+2]>T2)
417             {
418                 Edges[row][col+2]=255;
419                 neighbor_loop(row+1,col+3);
420             }
421
422             if(NonMax[row][col+3]<T1)
423             {
424                 Edges[row][col+3]=0;
425             }
426
427             else if(NonMax[row][col+3]>T2)
428             {
429                 Edges[row][col+3]=255;
430                 neighbor_loop(row+1,col+4);
431             }
432
433         }
434     }
435 }
```

Εικόνα 1.33: Κώδικας υλοποίησης συνολικής βελτιστοποίησης.



Αποτελέσματα



Εικόνα 1.34: Αποτελέσματα συνολικής βελτιστοποίησης.

Ο συνδιασμός μεθόδων που καταλήξαμε είναι αυτός που έδωσε τον μικρότεροχρόνο εκτέλεσης. Δεν χρισμοποιήθηκε η μέθοδος loop tiling αφού αυξάνει τους κύκλους εκέλεσης. Επίσης δεν χρησιμοποιήθηκε η μέθοδος loop inversion καθώς συνολικά απέδιδε χειρότερα και μας περιόριζε στο να δημιουργήσουμε καλύτερους δυνσιασμούς.



1.5 Πίνακας δεδομένων

1.5.1 Μέγεθος

Μετά την μεταγλώττιση του κώδικα μέσω του προγράμματος Arm Development Suite μπορούμε να δούμε τον πίνακα δεδομένων με την επιλογή make. Τα αποτελέσματα αναφορικά με το RW μέγευθος ήταν περίπου ίδια για όλες τις υλοποιήσεις. Στην ακόλουθη εικόνα παρουσιάζεται το μέγευθος του πίνακα δεδομένων για την βέλτιστη υλοποίηση.

Image component sizes					
	Code	RO Data	RW Data	ZI Data	Debug
4828		60	16	4749064	8272
22356		610	0	300	7624
<hr/>					
	Code	RO Data	RW Data	ZI Data	Debug
27184		670	16	4749364	15896
<hr/>					
Total RO	Size(Code + RO Data)			27854 (27.20kB)	
Total RW	Size(RW Data + ZI Data)			4749380 (4638.07kB)	
Total ROM	Size(Code + RO Data + RW Data)			27870 (27.22kB)	
<hr/>					

Εικόνα 1.35: Αποτελέσματα επιλογής make.

Βλέπουμε ότι το συνολικό μέγευθος του πίνακα δεδομένων είναι 4638.07 kB. Ο πίνακας απαρτίζεται από τα επαναγράψιμα αρχικοποιημένα δεδομένα (RW) και από τα επαναγράψιμα μη αρχικοποιημένα δεδομένα (ZI). Τα RW δεδομένα έχουν μέγευθος 16 B, ενώ τα ZI έχουν μέγευθος 4749364 B.

1.5.2 Αριθμός προσπελάσεων

Στην εικόνα βλέπουμε τον αριθμό των προσπελάσεων του πίνακα δεδομένων (διαδοχικές και μη διαδοχικές κλήσεις στην μνήμη) για όλες τις υλοποιήσεις. Ο αριθμός των προσπελάσεων του τελικού αρχείου είναι κατά 19.3% μικρότερος σε σχέση με τον αρχικό κώδικα. Στον αρχικό κώδικα οι προσπελάσεις του πίνακα δεδομένων είναι ίσες με περίπου 765 εκατομμύρια, ενώ στον τελικό κώδικα είναι ίσες με περίπου 617 εκατομμύρια.

Μέθοδος	Sequencial Cycles	Non-Sequencial Cycles	Άθροισμα	Ποσοστό βελτίωσης
Αρχικό χωρίς βελτιστοποίησεις	615.678.474	150.043.949	765.722.423	0%
Loop Inversion	614.964.106	149.350.710	764.314.816	0.22%
Loop Unrolling	613.412.338	147.118.527	760.530.865	0.68%
Loop Collapsing	609.089.581	149.685.011	758.774.592	0.9%
Loop Interchange	606.264.874	147.880.964	754.145.838	1.5%
Loop Fusion	597.130.289	140.220.932	737.351.221	3.7%
Pow	574.440.474	141.045.921	715.486.395	6.6%
Final best combination	506.058.429	111.643.069	617.701.498	19.3%

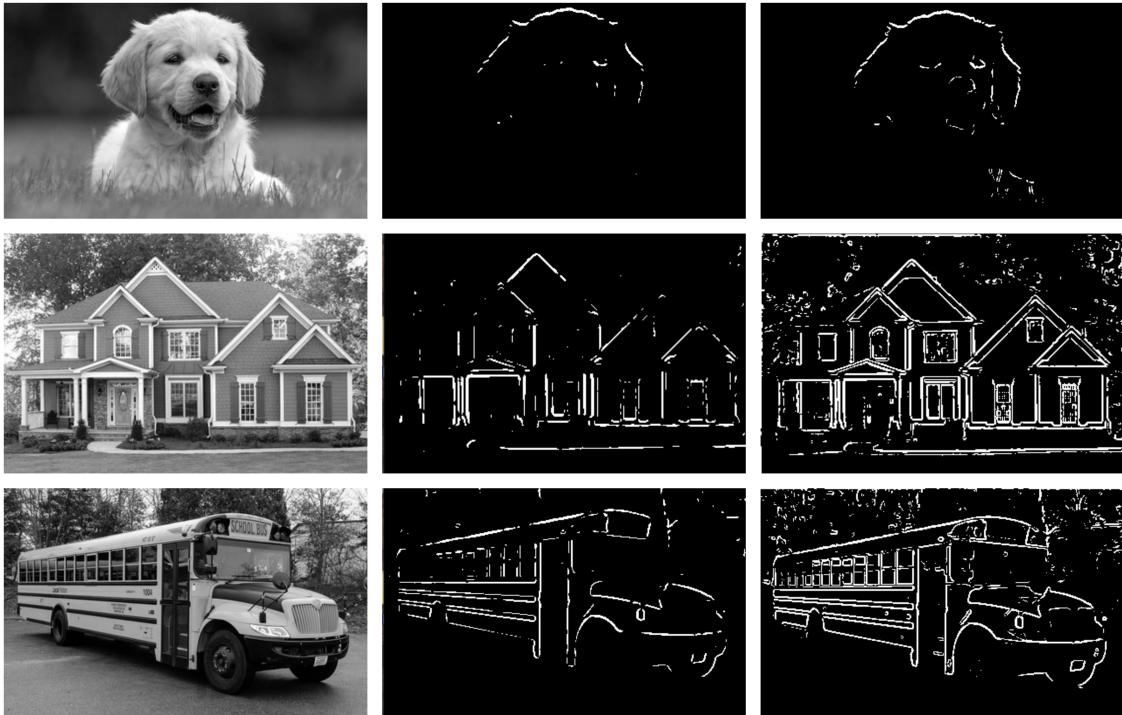
Εικόνα 1.36: Κύκλοι προσπέλασης μνήμης και ποσοστό μείωσης ως προς την αρχική για κάθε υλοποίηση.



1.6 Συμπεράσματα

1.6.1 Επιλογή μάσκας kernel για Gaussian blur

Η επιλογή του μεγέθους της μάσκας παίζει σημαντικό ρόλο. Με την χρήση ενός μεγαλύτερου φίλτρου λαμβάνονται υπ' όψιν περισσότερα γειτονικά εικονοστοιχεία. Δοκιμάστηκαν δύο διαφορετικά φίλτρα, ένα μεγέθους 3×3 με διακύμανση $\sigma=0.6$ και ένα μεγέθους 5×5 με διακύμανση $\sigma=1$.



Εικόνα 1.37: Σύγκριση kernel 5×5 με 3×3 σε διάφορες εικόνες.

Στην παραπάνω εικόνα βλέπουμε την διαφοροποίηση στο τελικό αποτέλεσμα με την εφαρμογή ενός φίλτρου 5×5 και ενός φίλτρου 3×3 . Στα αριστερά απεικονίζεται η αρχική εικόνα σε ασπρόμαυρο, στην μέση το αποτέλεσμα με χρήση 5×5 kernel και στα δεξιά το αποτέλεσμα με την χρήση ενός 3×3 φίλτρου. Από τα αποτελέσματα είναι εμφανές ότι η επιλογή του 3×3 φίλτρου με $\sigma=0.6$ έδωσε πολύ καλύτερα αποτελέσματα αναφορικά με την έρευση ακμών στις εικόνες.

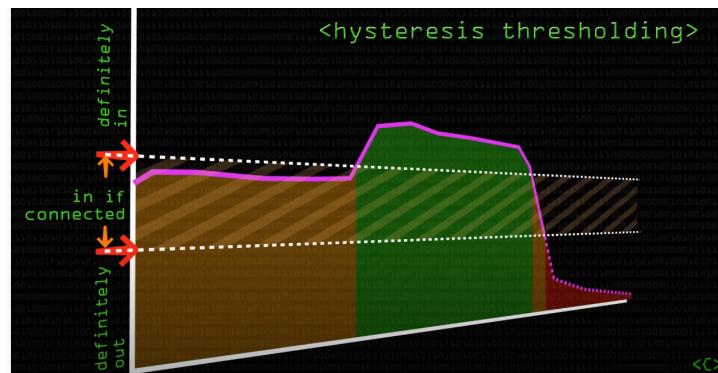


Εικόνα 1.38: Σύγκριση kernel 5×5 με 3×3 στην εικόνα flower.

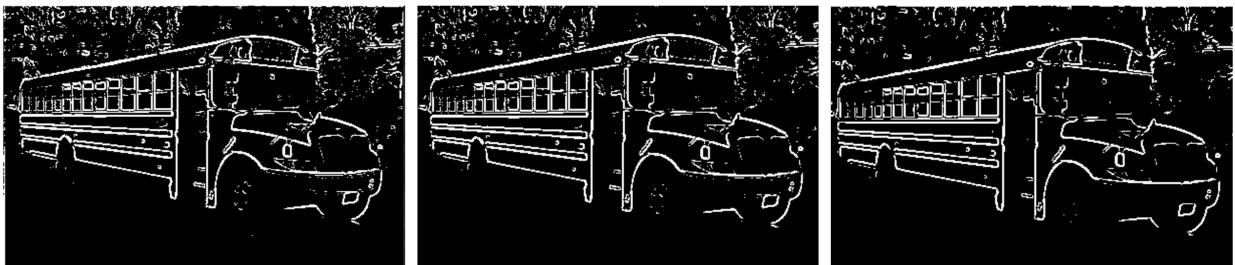


1.6.2 Επιλογή τιμών κατωφλίου T1 και T2

Στο τελευταίο βήμα για την υλοποίηση της κατωφλίωσης υστέρησης είναι σημαντική η επιλογή των δύο τιμών κατωφλίου, T1 και T2. Κατά την διαδικασία αυτή όλα τα εικονοστοιχεία με τιμή κάτω από T1 γίνονται ίσα με το 0, δηλαδή γίνονται μαύρα. Από την άλλη όσα έχουν τιμή πάνω από T2 γίνονται ίσα με το 255, δηλαδή γίνονται άσπρα. Μόλις βρεθεί ένα εικονοστοιχείο με τιμή μεγαλύτερη από T2 εξατάζονται με βάση της κλίση του οι τιμές των αντίστοιχων γειτόνων που βρίσκονται πάνω σε αυτή και είναι πιθανό να αποτελούν ακμή. Αν κάποιο γειτονικό εικονοστοιχείο έχει τιμή μεγαλύτερη από T1 γίνεται άσπρο και στη συνέχεια εξατάζονται τα γειτονικά εικονοστοιχεία που βρίσκονται πάνω στην κλίση του. Η διαδικασία αυτή συνεχίζεται μέχρι να βρεθεί ένα εικονοστοιχείο με τιμή μικρότερη από T1. Στην εικόνα απεικονίζονται με ροζ χρώμα οι τιμές των εικονοστοιχείων ενός πίνακα. Βλέπουμε ότι η τιμή των T1 και T2 μπορεί να επηρεάσει σε μεγάλο βαθμό το αποτέλεσμα.



Εικόνα 1.39: Εικόνα με ενδεικτικές τιμές εικονοστοιχείων.



Εικόνα 1.40: Σύγκριση αποτελεσμάτων διαφορετικών τιμών T1 και T2 στην εικόνα bus, η αριστερή έχει T1=180 και T2=190, η μεσαία έχει T1=205 και T2=210 και η δεξιά έχει T1=230 και T2=245.

Όπως φαίνεται στην παραπόνω εικόνα με την επιλογή χαμηλότερων τιμών για τα T1 και T2 παρουσιάζεται περισσότερος ψόρυβος. Η επιλογή των τιμών T1=230 και T2=245 δίνει τα καλύτερα αποτελέσματα.



1.6.3 Βελτιστοποίηση χρόνου

Τηλοποιόντας μία μία τις τεχνικές μπορούμε να δούμε σε τι βαθμό επιτυγχάνεται βελτιστοποίηση του αρχικού κώδικα. Τα αποτελέσματα αναφορικά με την βελτιστοποίηση στους κύκλους εκτέλεσης του επεξεργαστή φαίνονται στην ακόλουθη εικόνα.

Μέθοδος	Κύκλοι επεξεργαστή	Ποσοστό βελτίωσης
Αρχικό χωρίς βελτιστοποίησης	817.131.427	0%
Loop Inversion	815.614.407	0.19%
Loop Unrolling	810.914.302	0.8%
Loop Collapsing	809.652.124	0.9%
Loop Interchange	804.328.750	1.5%
Loop Fusion	785.445.028	3.9%
Pow	764.433.614	6.4%
Final best combination	654.384.564	20%

Εικόνα 1.41: Ποσοστό βελτίωσης με την χρήση των μεθόδων.

Όπως φαίνεται η τεχνική του Loop Inversion βελτιώνει μόλις κατά 0.19% τους συνολικούς κύκλους εκτέλεσης, ενώ η τεχνική του Loop Unrolling μειώνει κατά 0.8%. Αντίστοιχα μικρό ποσοστό σημειώνει και η τεχνική του Loop Collapsing με 0.9% βελτίωση χρόνου και η τεχνική του Loop Interchange με 1.5%. Ακολουθεί η μέθοδος του Loop Fusion με ποσοστό 3.9%, ενώ σημαντική μείωση στο 6.4% μπορεί να επιτευχθεί με την αντικατάσταση της συνάρτησης `pow` με τον πράξη του πολλαπλασιασμού. Με τον συνδιασμό όλων των μεθόδων το προκύπτει πολύ σημαντική μείωση της τάξεως του 20% στους κύκλους του επεξεργαστή.



Μέρος δεύτερο

2.1 Μνήμη

2.1.1 ROM

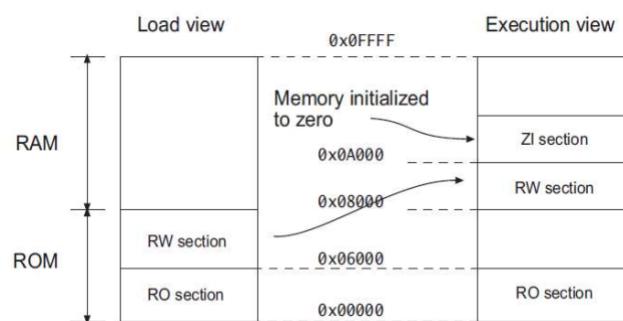
Η μνήμη μόνο για ανάγνωση (ROM, Read-only memory) είναι ένας τύπος μη πτητικής μνήμης που χρησιμοποιείται κυρίως σε ηλεκτρονικούς υπολογιστές αλλά και ηλεκτρονικές συσκευές. Σε αυτήν αποθηκεύεται λογισμικό το οποίο σπάνια θα μεταβληθεί κατά τα χρόνια λειτουργίας του συστηματος, γνωστό και ως firmware. Συνήθως με τον όρο ROM εννοούμε μία μνήμη της οποίας το περιεχόμενό δεν αλλάζει μετά την κατασκευή της. Για την διόρθωση λαθών ή την αναβάθμιση του λογισμικού θα πρέπει να αντικατασταθεί από μία νέα.

2.1.2 RAM

Η μνήμη τυχαίας προσπέλασης (RAM, Random access memory) είναι όρος που χρησιμοποιείται για ηλεκτρονικές διατάξεις προσωρινής αποθήκευσης ψηφιακών δεδομένων (μνήμης υπολογιστή), οι οποίες επιτρέπουν πρόσβαση στα αποθηκευμένα δεδομένα στον ίδιο χρόνο οπουδήποτε και αν βρίσκονται αυτά. Υπάρχουν δύο βασικοί τύποι RAM, η δυναμική RAM (DRAM) και η στατική RAM (SRAM). Η DRAM είναι η πιο κοινή μορφή αλλά πρέπει να ανανεώνεται (refresh) χιλιάδες φορές ανά δευτερόλεπτο, ενώ η SRAM δεν χρειάζεται κάτι τέτοιο. Η SRAM, ως διάταξη, είναι πιο δαπανηρή στην κατασκευή της και επομένως στην αγορά της σε σχέση με την DRAM.

2.1.3 Memory views

Η μνήμη του συστήματος χωρίζεται σε δύο περιοχές, την περιοχή Load και την περιοχή Execution. Ο διαχωρισμός αυτός γίνεται τελείως διαισθητικά και αφορά την κατάσταση αδρανείας και την κατάσταση εκτέλεσης του συστήματος. Η περιοχή Load υφίσταται κατά την λειτουργία αδρανείας. Τότε το ενσωματωμένο σύστημα δεν εκτελεί καμία λειτουργία και έτσι αποθηκεύονται στην ROM μνήμη βασικά δεδομένα όπως ο κώδικας και κάποια δεδομένα για την εκκίνηση. Η περιοχή Execution υφίσταται κατά την λειτουργία εκτέλεσης. Τότε το ενσωματωμένο σύστημα εκτελεί τις λειτουργίες του. Τα δεδομένα μεταφέρονται στις περιοχές εκτέλεσης, κάποιες μεταβλητές και κάποιο μέρος του κώδικα μεταφέρονται στην RAM, ενώ κάποια παραμένουν στην ROM.



Εικόνα 2.1: Memory views κατά την μετάβαση σε κατάσταση εκτέλεσης.



H Load ROM ισούται με τον αριθμό σε bytes του κώδικα, των δεδομένων RO και των δεδομένων RW. H Executable ROM ισούται με τον αριθμό σε bytes του κώδικα και των δεδομένων RO. Το μέγεθος της RAM (RW+ZI) που δηλώνεται στο αρχείο scatter.txt ισούται με τον αριθμό σε bytes των δεδομένων RW και ZI. Στο αρχείο memory.map ως συνολικό μέγεθος RAM δηλώνουμε τον αριθμό σε bytes των δεδομένων RW, των δεδομένων ZI και του μεγέθους της μνήμης που έχουμε δώσει για τα stack-heap.



2.2 Αρχεία για εισαγωγή μνήμης

2.2.1 Αρχείο memory.map

Σε αυτό το αρχείο γίνεται ο ορισμός της αρχιτεκτονικής της μνήμης του ενσωμετωμένου συστήματος. Μπορούμε να δηλώσουμε διάφορες υπομονάδες μνήμης με τα χαρακτηριστικά τους όπως, την διεύθυνση έναρξης, το μέγεθος, το όνομά τους, το μέγεθος του bus τους, τον τύπο τους, καθώς και την εγγραφής και ανάγνωσης N Και S κύκλων.

```
1 START1 SIZE1 TAG1 BUS1 TYPE1 S Cycles1 N Cycles1
2 START2 SIZE2 TAG2 BUS2 TYPE2 S Cycles2 N Cycles2
```

Εικόνα 2.2: Παράδειγμα αρχείου memory map.

2.2.2 Αρχείο scatter.txt

Σε αυτό το αρχείο περιγράφουμε ακριβώς ποια δεδομένα θα εγγραφούν σε ποιές περιοχές τις μνήμης για κάθε χρονική στιγμή (Load και Execution). Μπορούμε να ορίσουμε γενικά για την θέση των τύπων δεδομένων που έχουμε RO, RW, ZI άλλα και συγκεκριμένες μεταβλητές μέσω επικετών στον πηγαίο κώδικα. Αρχικά ορίζουμε ότι κατά την κατάσταση αδρανείας όλα τα δεδομένα βρίσκονται στην ROM. Στη συνέχεια περιγράφουμε τι συμβαίνει κατά την λειτουργία εκτέλεσης, δηλαδή δηλώνουμε ποια από τα δεδομένα που βρίσκονται στην ROM παραμένουν σε αυτήν και ποιά μεταφέρονται στην RAM ή σε οποιαδήποτε άλλη υπομονάδα μνήμης έχουμε ορίσει στο αρχείο memory.MAP.

```
1 MEMORY1 START SIZE
2 {
3 MEMORY1 START SIZE
4 {
5 *.o ( +RO )
6 }
7 MEMORY2 START SIZE
8 {
9 * ( +ZI,+RW )
10 }
11 }
12 }
```

Εικόνα 2.3: Παράδειγμα αρχείου memory map.

Στην προηγούμενη εικόνα βλέπουμε ότι μέσω του αρχείου scatter ορίζουμε το σύνολο των δεδομένων μας να αποθηκεύονται στην MEMORY1 κατά την κατάσταση αδρανείας. Μόλις το σύστημα ξεκινήσει να λειτουργεί τότε η δομή της μνήμης είναι αυτή της κατάστασης εκτέλεσης. Ορίζουμε ώστε κατά την κατάσταση εκτέλεσης τα δεδομένα τύπου RO να παραμένουν στην ROM ενώ τα δεδομένα τύπου ZI, RW να μεταφέρονται στην MEMORY2.

2.2.3 Αρχείο stack.c

Στο αρχείο stack.c ορίζουμε την διεύθυνση έναρξης των δομών heap και stack. Η δομή heap αποθηκεύει δεδομένα σχετικά με τις συναρτήσεις δυναμικής κατανομής μνήμης οι οποίες είναι οι malloc, realloc, calloc, free. Πολλές από τις συναρτήσεις των βιβλιοθηκών της C χρισμοποιούν αυτές τις συναρτήσεις όπως για παράδειγμα οι συναρτήσεις fopen και fclose. Συγκεκριμένα για



κάθε κλήση του ζεύγους συναρτήσεων fopen και fclose δεσμεύονται 592 byte στο heap. Η δομή stack αποθηκεύει τις τοπικές μεταβλητές, δηλαδή αυτές οι οποίες ορίζονται μέσα στην main και στις συναρτήσεις που ορίζουμε εμείς. Επίσης αποθηκεύει κάποια δεδομένα κατά για την εκκίνηση. Το ακριβές μέγεθος των δομών heap και stack δεν μπορεί να προσδιοριστεί εύκολα και με ακρίβεια. Από την μία είναι χρονοβόρο να ελέγξουμε το πλήθος χρήσης των συναρτήσεων δυναμικής κατανομής μνήμης και αφετέρου δεν γνωρίζουμε πάντα ακριβός το μέγεθος των δεδομένων που αποθηκεύονται στην δομή stack. Προσθέτουμε το αρχείο stack.c μαζί με τον αρχείο που περιέχει τον κώδικας υλοποίησης. Έπειτα εκτελούμε αυτά τα δύο αρχεία. Το αρχείο stack.c έχει την παρακάτω μορφή:

```
1 #include <rt_misic.h>
2
3     __value_in_regs struct __initial_stackheap __user_initial_stackheap(
4     unsigned R0, unsigned SP, unsigned R2, unsigned SL){
5         |
6         struct __initial_stackheap config;
7
8         //config.heap_limit = 0x00724B40;
9
10        //config.stack_limit = 0x00004000;
11        //config.stack_limit = 0x00100000;
12
13        /* works */
14        //config.heap_base = 0x01260000;
15        //config.stack_base = 0x01460000;
16
17        config.heap_base = 0x48D97C;
18        config.stack_base = 0x49008C;
19
20
21        /* factory defaults */
22        //config.heap_base = 0x00060000;
23        //config.stack_base = 0x00080000;
24        return config;}
```

Εικόνα 2.4: Παράδειγμα αρχείου stack.c.



2.3 Ορισμός αρχιτεκτονικής μνήμης

2.3.1 Αρχιτεκτονική 1

Αρχικά δοκιμάζουμε την υλοποίηση μία απλής αρχιτεκτονικής με μία μνήμη ROM και μνήμη RAM. Υπολογίζουμε τα μεγέθη τους από το Make log, πιο συγκεκριμένα:

	Code	RO Data	RW Data	ZI Data	Debug
Object Totals	6152	40	32	4749092	10792
Library Totals	18176	474	0	300	7184
	Code	RO Data	RW Data	ZI Data	Debug
Grand Totals	24328	514	32	4749392	17976
	Total RO	Size(Code + RO Data)			24842 (24.26kB)
	Total RW	Size(RW Data + ZI Data)			4749424 (4638.11kB)
	Total ROM	Size(Code + RO Data + RW Data)			24874 (24.29kB)

Εικόνα 2.5: Make log του βέλτιστου κώδικα του πρώτου μέρους.

Βλέπουμε ποιά είναι η ανάγκη σε μέγεθος για τις μνήμες ROM και RAM. Το μέγεθος των δεδομένων που αποθηκέυονται στην ROM (Code + RO Data + RW Data) είναι ίσο με 24874 bytes, ενώ το μέγεθος των δεδομένων που αποθηκέυονται στην RAM (RW + ZI) είναι 4749424 bytes. Έχοντας αυτά τα δεδομένα πάμε και ορίζουμε τις παραμέτρους αρχικά του αρχείου memory.MAP. Πρώτα ορίζουμε ότι θα έχουμε μία ROM και μία RAM με μέγεθος διαύλου ίσο με 4 bytes. Άρα θα πρέπει το μέγεθος που θα έχουν μνήμες ROM και RAM να είναι πολλαπλάσιο του 4. Επίσης δεν πρέπει να γίνεται σπατάλη στις μνήμες αφού το κόστος τους είναι υψηλό και αυξάνει με την ταχύτητά τους, οπότε θα πρέπει να είμαστε όσο το δυνατό πιο ακριβείς γίνεται στον ορισμό του μεγέθους τους. Η σωστή επιλογή μεγέθους μνημών είναι αυτή που εξασφαλίζει την βέλτιστη απόδοση με το μικρότερο δυνατό κόστος. Άρα, σύμφωνα με τα προηγούμενα, επιλέγουμε για μέγεθος κάθε μνήμης το αμέσως επόμενο πολλαπλάσιο του 4 μετά την ανάγκες που υπολογίζουμε από το Make log. Οι ανάγκες σε μνήμη υπολογίζονται πολύ εύκολα στην προκύμενη περίπτωση, κατευθείαν από το Make log. Βλέπουμε ότι η απαιτούμενη ROM είναι στα 24874 bytes, ενώ η απαιτούμενη RAM είναι στα 4759424 bytes. Στην RAM εκτός της ελάχιστης αναγκαίας τιμής πρέπει να δώσουμε και τον χώρο για την αποθήκευση των δομών heap και stack. Για αυτές τις δομές δίνουμε αυθαίρετα 10000 bytes, άρα το απαιτούμενο μέγεθος της RAM είναι 4759424 bytes. Άρα επιλέγουμε τα εξής μεγέθη:

	Memory size Dec	Memory size Hex
ROM	24844	610C
RAM	4759424	489F80

Εικόνα 2.6: Επιλογή μεγεθών για τις μνήμες της 1ης αρχιτεκτονικής.

Τώρα ορίζουμε από ποιά διεύθυνση ξεκινάνε οι δύο μνήμες. Θέτουμε την ROM να ξεκινάει από την 0(dec), ενώ θέτουμε την RAM να ξεκινάει από την διεύθυνση που τελειώνει η ROM,



δηλαδή από την διεύθηση 24844(dec). Επίσης η ROM ορίζουμε να είναι τύπου R, ενώ η RAM τύπου RW. Τέλος θέτουμε ταχύτητες 1/1 για την ROM και 250/50 για την RAM. Το αρχείο memory.MAP τελικά είναι:

```
1 00000000 610C ROM 4 R 1/1 1/1
2 610C 489F80 RAM 4 RW 250/50 250/50
```

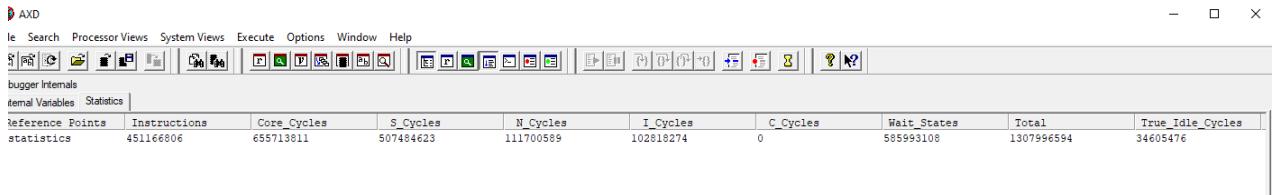
Εικόνα 2.7: Παραμετροποίηση αρχείου memory.MAP.

Στο αρχείο scatter.txt ορίζουμε πως τα δεδομένα μοιράζονται στην μνήμη για τα Load και Execution views. Αρχικά ορίζουμε ότι στο Load view όλα τα δεδομένα Code, RO, RW βρίσκονται στην ROM. Αυτά έχουν μέγεθος 24874 bytes το οποίο όμως δεν είναι πολλαπλάσιο του 4. Για να είναι λειτουργική η αρχιτεκτονική θέτουμε το μέγεθος Load view ROM ίσο με το αμέσως επόμενο πολλαπλάσιο του 4 το οποίο είναι 24876. Στη συνέχεια ορίζουμε ότι κατά το Execution view θα παραμένουν στην ROM μόνο τα δεδομένα τύπου RO και ο κώδικας. Το μέγεθος αυτό είναι ίσο με το αντίστοιχο που ορίστηκε για την ROM στο αρχείο memory.MAP, δηλαδή είναι 24844 bytes. Τέλος ορίζουμε ότι κατά το Execution view θα εγγράφονται στην μνήμη RAM τα δεδομένα τύπου ZI, RW. Το μέγεθος αυτών, όπως φαίνεται από το Make log, είναι 4749424 bytes, το οποίο είναι πολλαπλάσιο του 4.

```
1 ROM 0x0 0x612C
2 {
3 ROM 0x0 0x610C
4 {
5 *.o ( +RO )
6 }
7 RAM 0x610C 0x487870
8 {
9 * ( +ZI,+RW )
10 }
11
12 }
```

Εικόνα 2.8: Παραμετροποίηση αρχείου scatter.txt.

Βλέπουμε τώρα πως αποδίδει ο βέλτιστος κώδικας του πρώτου μέρους της εργασίας με την αρχιτεκτονική μνήμης που ορίσαμε. Τρέχουμε τον κώδικα στον AXD και παίρνουμε το εξής αποτέλεσμα.



Εικόνα 2.9: Αρχική επιλογή ROM και DRAM.

Βλέπουμε ότι στο πεδίο Statistics έχει εμφανιστεί μία νέα παράμετρος, η wait states. Αυτή αφορά τους κύκλους στους οποίους ο επεξεργαστής περιμένει να λάβει δεδομένα από την μνήμη. Αφού ο επεξεργαστής τοποθετήσει τις διευθύνσεις μνήμης από τις οποίες θέλει να διαβάσει δεδομένα, περιμένει μέχρι να βρεθούν από την μνήμη και να αποσταλούν. Επειδή οι μνήμες είναι πιο



αργές από την ταχύτητα επεξεργασίας του επεξεργαστή πάντα όμως υπάρχουν wait states. Αυτό φανερώνει μία σημαντική σπατάλη, όχι μόνο στην απόδοση του επεξεργαστή αλλά και σε ενέργεια αφού το ρολόι του επεξεργαστή συνεχίζει να λειτουργεί και όσο περιμένει τα δεδομένα που έχει ζητήσει. Με διαφορετικές αρχιτεκτονικές μνήμης όμως δούμε ότι ο αριθμός των wait states μειώνεται.

2.3.2 Αρχιτεκτονική 2

Στην δεύτερη αρχιτεκτονική γίνεται χρήση μιας γρήγορης μνήμης SRAM. Για τον υπολογισμό του μεγέθους των μνημών ανοίγουμε το Make log.

	Code	RO Data	RW Data	ZI Data	Debug
Object Totals	6152	60	32	4749092	10788
Library Totals	18176	474	0	300	7184
Grand Totals	24328	534	32	4749392	17972
Total RO Size(Code + RO Data)				24862	(24.28kB)
Total RW Size(RW Data + ZI Data)				4749424	(4638.11kB)
Total ROM Size(Code + RO Data + RW Data)				24894	(24.31kB)

Εικόνα 2.10: Αποτελέσματα make με την χρήση SRAM.

Από τον πίνακα βλέπουμε τις απαιτήσεις για κάθε μνήμη. Στόχος είναι η επιλογή των μικρότερων δυνατών τιμών για την κάθε μνήμη ώστε να επιτευχθεί η καλύτερη δυνατή βελτιστοποίηση. Επιπλέον πρέπει οι τιμές των μνημών να είναι ακέραια πολλαπλάσια του 4. Από τον πίνακα βλέπουμε ότι η Load ROM (Code+RO+RW) είναι ίση με 24862 bytes. Επειδή $(24894 \bmod 4) = 2$ επιλέγουμε την αμέσως επόμενη τιμή που είναι 24896 bytes. Ομοίως για την Executable ROM (Code+RO) που είναι 24862 επειδή $(24862 \bmod 4) = 2$ επιλέγουμε την τιμή 24864. Το μέγεθος των συνολικών RW και ZI δεδομένων έχει παραμείνει ίδιο και είναι ίσο με 4749424 bytes. Τα δεδομένα που τοποθετούνται στην καινούργια γρήγορη SRAM είναι ίσα με 164 bytes. Για την επίτευξη της επιλογής των μικρότερων δυνατών μνημών επιλέγουμε DRAM (ZI+RW) στο scatter αρχείο ίση με την προηγούμενη τιμή μείον 164 bytes, δηλαδή αφαιρούμε τα bytes που τοποθετούνται στην SRAM. Ομοίως στο memory map αρχείο αφαιρούμε από την συνολική προηγούμενη DRAM (που περιλαμβάνει και το μέγεθος των δομών stack και heap) τα 164 bytes της καινούργιας SRAM. Στη διεύθυνση που τελειώνει η DRAM ξεκινάει η γρήγορη SRAM. Η SRAM έχει ταχύτητες 1/1 για εγγραφή και ανάγνωση τόσο σε sequential όσο και σε non-sequential κλήσεις.

	Memory size Dec	Memory size Hex
ROM	24864	6120
DRAM	4759260	489EDC
SRAM	164	A4

Εικόνα 2.11: Αποτελέσματα με την χρήση Επιλογή μεγεθών για τις μνήμες της 1ης αρχιτεκτονικής.



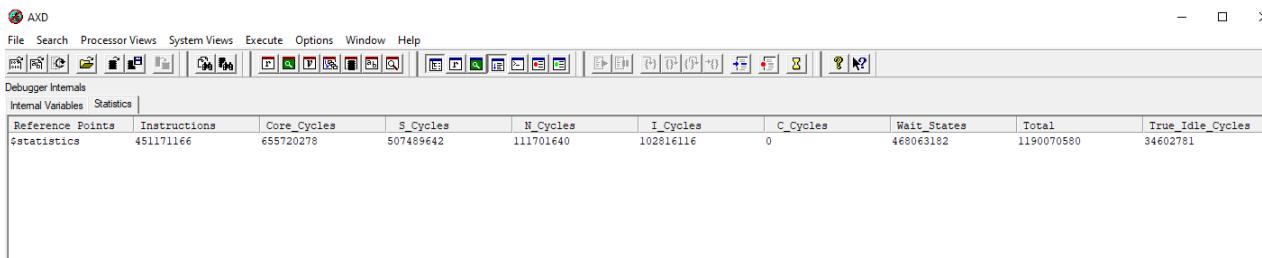
```
1 00000000 6120 ROM 4 R 1/1 1/1
2 6120 489EDC DRAM 4 RW 250/50 250/50
3 49008C A4 SRAM 4 RW 1/1 1/1
```

Εικόνα 2.12: Παραμετροποίηση αρχείου memory.MAP.

```
1 ROM 0x0 0x6140
2 {
3 ROM 0x0 0x6120
4 {
5 *.o ( +RO )
6 }
7 DRAM 0x6120 0x4877CC
8 {
9 * ( +ZI,+RW )
10 }
11 SRAM 0x48FFE8 0xA4
12 {
13 * (ram)
14 }
15
16 }
```

Εικόνα 2.13: Παραμετροποίηση αρχείου scatter.txt.

Με την εκτέλεση του προγράμματος στον AXD παίρνουμε τα εξής αποτελέσματα.



Εικόνα 2.14: Αποτελέσματα με την χρήση SRAM.

Παρατηρούμε ότι υπάρχει μια σημαντική μείωση των wait cycles καθώς είναι κατά περίπου 120.000.000 λιγότερα σε σχέση με την αρχιτεκτονική 1. Επιπλέον οι συνολικοί κύκλοι (total) έχουν μειωθεί κατά περίπου 100.000.000 σε σχέση με την αρχιτεκτονική 1.

2.3.3 Αρχιτεκτονική 3

Στην τρίτη αρχιτεκτονική γίνεται χρήση μιας πιο αργής αλλά μεγαλύτερης μνήμης SRAM. Στην νέα SRAM βάζουμε τις μεταβλητές που είχαμε βάλει και την δεύτερη αρχιτεκτονική ενώ προσθέτουμε και τον πίνακα Εο. Η επιλογή του συγκεκριμένου πίνακα έγινε αφού αυτός μαζί με τον πίνακα Es χρησιμοποιούνται τις περισσότερες φορές από οποιοδήποτε άλλον μέσα στον κώδικα. Οι συγκεκριμένοι πίνακες χρησιμοποιούνται σε τέσσερεις λειτουργίες του κώδικα. Δεν βάζουμε και τον πίνακα Es στην SRAM αφού έτσι το μέγεθός της αυξάνεται σημαντικά πράγμα που την κάνει πολύ ακριβή δεδομένης της ταχήτυτά της. Το μέγεθος του πίνακα Εο υπολογίζεται χρησιμοποιώντας την εικόνα "bus_420x280.yuv" ως εξής. Ο πίνακας έχει διαστάσεις 422x282 και είναι τύπου int, άρα το μέγεθός του δίνεται από τον εξής πολλαπλασιασμό:

$$422 \times 282 \times 4 = 476016 \text{ bits}$$



Άρα το ελάχιστο μέγευθος της SRAM πρέπει είναι

$$476016 + 164 = 476180 \text{ bits}$$

Ορίζουμε το μέγευθος διαιώλου να είναι 4 bytes, άρα όταν πρέπει να φροντίσουμε τα μεγέθη των μνημών να είναι πολλαπλάσια του 4. Επίσης προσθέτουμε στην DRAM 10000 bytes για τις ανάγκες των stack και heap. Τελκά τα μεγέθη που επιλέγουμε είναι τα εξής.

	Memory size Dec	Memory size Hex
ROM	24864	6120
DRAM	4283244	415B6C
SRAM	476180	74414

Εικόνα 2.15: Αποτελέσματα με την χρήση Επιλογή μεγευθών για τις μνήμες της 3ης αρχιτεκτονικής.

Ορίζουμε τα αρχεία memory.MAP, scatter.txt και stack.c με τον τρόπο που ορίστηκε στην αρχιτεκτονική 1 σύμφωνα με τα μεγέθη που υπολογίσαμε για την αρχιτεκτονική 3.

```
1 00000000 6120 ROM 4 R 1/1 1/1
2 6120 415B6C DRAM 4 RW 250/50 250/50
3 41BC8C 74414 SRAM 4 RW 30/10 30/10
```

Εικόνα 2.16: Παραμετροποίηση αρχείου memory.MAP.

```
1 ROM 0x0 0x6140
2 {
3 ROM 0x0 0x6120
4 {
5 * .o ( +RO )
6 }
7 DRAM 0x6120 0x41345C
8 {
9 * ( +ZI,+RW )
10 }
11 SRAM 0x41BC8C 0x74414
12 {
13 * (ram)
14 }
15
16 }
```

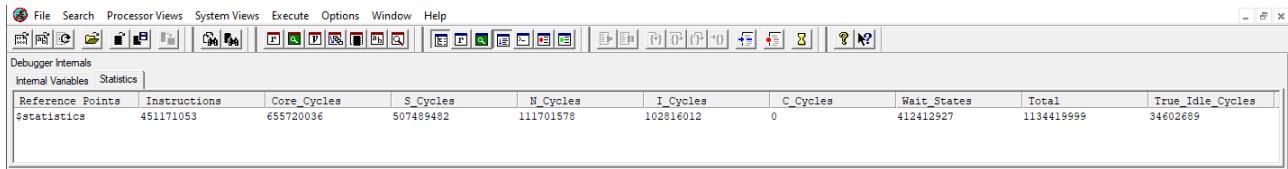
Εικόνα 2.17: Παραμετροποίηση αρχείου scatter.txt.

```
1 #include <rt_misc.h>
2
3 __value_in_regs struct __initial_stackheap __user_initial_stackheap(
4     unsigned R0, unsigned SP, unsigned R2, unsigned SL){
5
6     struct __initial_stackheap config;
7
8     //config.heap_limit = 0x00724B40;
9
10    //config.stack_limit = 0x00004000;
11    //config.stack_limit = 0x00100000;
12
13    /* works */
14    //config.heap_base = 0x01260000;
15    //config.stack_base = 0x01460000;
16
17    config.heap_base = 0x41957C;
18    config.stack_base = 0x41BC8C;
19
20
21    /* factory defaults */
22    //config.heap_base = 0x00060000;
23    //config.stack_base = 0x00080000;
24    return config;
}
```

Εικόνα 2.18: Παραμετροποίηση αρχείου stack.c.



Βλέπουμε το αποτέλεσμα παρακάτω.



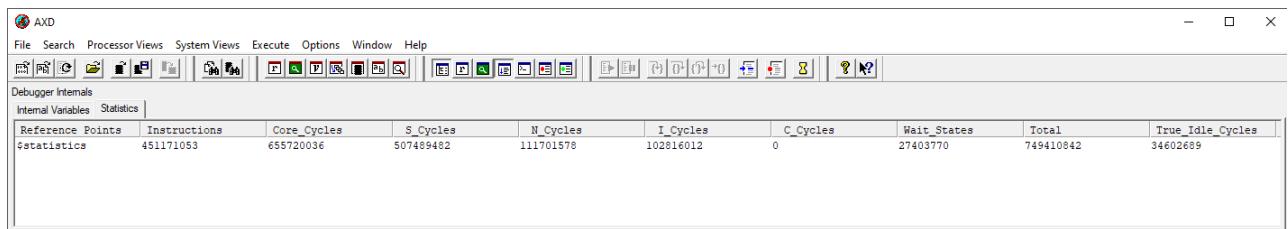
Εικόνα 2.19: Αποτελέσματα 3ης αρχιτεκτινικής.

Βλέπουμε ότι το πλήθος των wait states μειώνεται σε σχέση με την αρχιτεκτονική 2, παρ' όλο που η ταχύτητα της SRAM έχει μειωθεί. Όμως ο πίνκας που έχουμε προσθέσει στην SRAM της αρχιτεκτονικής 3 είναι αρκετά μεγάλος και χρησιμοποιείται πολλές φορές. Οπότε έτσι εξισορροπείται η μείωση της ταχύτητας. Άρα καταλήγουμε με μία πιο αργή μνήμη να έχουμε καλύτερο αποτέλεσμα εάν τοποθετήσουμε έξυπνα μεταβλητές σε αυτήν.

2.3.4 Σημασία ρολογιού επεξεργαστή

Για να δούμε την σημασία του ρολογιού επεξεργαστή τρέχουμε τον κώδικα με την αρχιτεκτονική μνήμης 3 αλλά ορίζουμε στον AXD το ρολόι του επεξεργαστή ίσο με 5MHz. Τα αποτελέσματα είναι τα παρακάτω.

Βλέπουμε το αποτέλεσμα παρακάτω.



Εικόνα 2.20: Αποτελέσματα 3ης αρχιτεκτινικής με ρολόι επεξεργαστή 5MHz.

Βλέπουμε ότι το πλήθος των κύκλων αναμονής μειώνεται και μαζί τους και το πλήθος των συνολικών κύκλων. Η μείωση των κύκλων αναμονής είναι λογική αφού πλέον ο επεξεργαστής λειτουργεί με μικρότερη συχνότητα, άρα τα διαστήματα αναμονής δεδομένων από την μνήμη μοιράζονται σε λιγότερους κύκλους επεξεργαστή. Η μείωση του πλήθους των συνολικών κύκλων οφείλεται μόνο στην μείωση των κύκλων αναμονής καθώς οι υπόλοιποι κύκλοι παραμένουν ίδιοι.

2.3.5 Αρχιτεκτονική 4

Αυτή η αρχιτεκτονική είναι ακριβώς ίδια με την αρχιτεκτονική 3 με την μόνη διαφορά ότι οι υπομονάδες μνήμης έχουν μέγεθος διαύλου ίσο με 2 bytes. Τα αρχεία mamory.MAP, scatter.txt και stack.c για αυτή στην αρχιτεκτονική φαίνονται παρακάτω.



```
00000000 6120 ROM 4 R 1/1 1/1
6120 415B6C DRAM 4 RW 250/50 250/50
41BC8C 74414 SRAM 2 RW 30/10 30/10
```

Εικόνα 2.21: Αρχείο memory.MAP 4ης αρχιτεκτονικής.

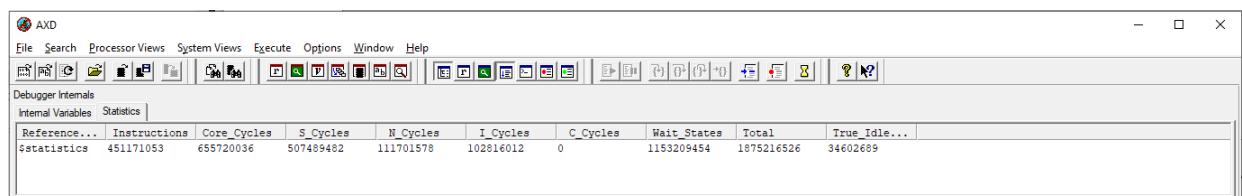
```
1 ROM 0x0 0x6140
2 {
3 ROM 0x0 0x6120
4 {
5 *.o ( +RO )
6 }
7 DRAM 0x6120 0x41345C
8 {
9 * ( +ZI,+RW )
10 }
11 SRAM 0x41BC8C 0x74414
12 {
13 * (ram)
14 }
15
16 }
```

Εικόνα 2.22: Αρχείο scatter.txt 4ης αρχιτεκτονικής.

```
1 #include <rt_misc.h>
2
3 __value_in_regs struct __initial_stackheap __user_initial_stackheap(
4     unsigned R0, unsigned SP, unsigned R2, unsigned SI){
5
6     struct __initial_stackheap config;
7
8 //config.heap_limit = 0x00724B40;
9
10 //config.stack_limit = 0x00004000;
11 //config.stack_limit = 0x00100000;
12
13 /* works */
14 //config.heap_base = 0x01260000;
15 //config.stack_base = 0x01460000;
16
17 config.heap_base = 0x41957C;
18 config.stack_base = 0x41BC8C;
19
20
21 /* factory defaults */
22 //config.heap_base = 0x00060000;
23 //config.stack_base = 0x00080000;
24     return config;
}
```

Εικόνα 2.23: Αρχείο stack.c 4ης αρχιτεκτονικής.

Τα αποτελέσματα με την χρήση αυτής της αρχιτεκτονικής είναι τα παρακάτω.



Εικόνα 2.24: BEST FINAL WITH BUS=2.

Βλέπουμε ότι καθώς το μέγεθος διαύλου μειώνεται οι κύκλοι αναμονής αυξάνονται. Αυτό είναι λογικό αφού με μικρότερο μέγεθος διαύλου μειώνεται και το πλήθος των δεδομένων που μπορούμε να μεταφέρουμε σε κάθε κύκλο μνήμης, άρα η αναμονή του επεξεργαστή είναι μεγαλύτερη.



2.3.6 Αρχιτεκτονική 5

Στην τέταρτη αρχιτεκτονική χρησιμοποιούμε μια μνήμη ROM και μια αρκετά γρήγορη SRAM μνήμη με χρόνους ανάγνωσης-εγγραφής 40/20 τόσο σε διαδοχικές (sequential) όσο και σε μη διαδοχικές (non-sequential) κλήσεις στη μνήμη. Έχουν τοποθετηθεί δηλαδή όλα τα δεδομένα RW και ZI σε αυτή τη μνήμη. Οι τιμές για τις LOAD ROM και Executable ROM είναι ίδιες με αυτές της αρχιτεκτονικής 1. Αντίστοιχα ίδια είναι η τιμή του μεγέθους της SRAM με την DRAM της αρχιτεκτονικής 1.

	Memory size Dec	Memory size Hex
ROM	24844	610C
RAM	4759424	489F80

Εικόνα 2.25: Επιλογή μεγεθών για τις μνήμες της 5ης αρχιτεκτονικής.

Ορίζουμε τα αρχεία memory.MAP, stack.c και scatter.txt ως εξής.

```
1 00000000 610C ROM 4 R 1/1 1/1
2 610C 489F80 SRAM 4 RW 40/20 40/20
3
```

Εικόνα 2.26: Αρχείο memory.MAP 5ης αρχιτεκτονικής.

```
1 ROM 0x0 0x612C
2 {
3 ROM 0x0 0x610C
4 {
5 *.o ( +RO )
6 }
7 RAM 0x610C 0x487870
8 {
9 * ( +ZI,+RW )
10 }
11 }
12
```

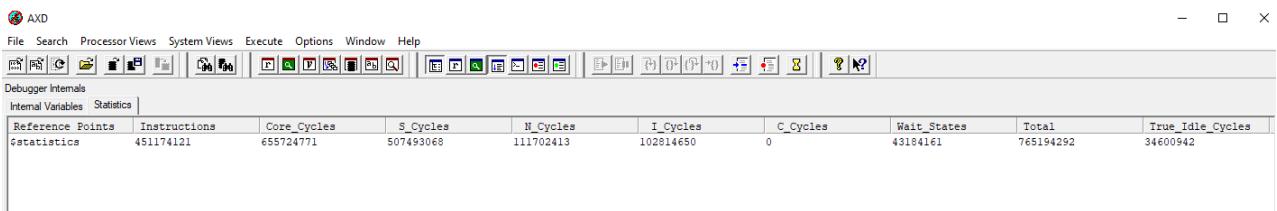
Εικόνα 2.27: Αρχείο scatter 5ης αρχιτεκτονικής.

```
1 #include <rt_misc.h>
2
3 __value_in_regs struct __initial_stackheap __user_initial_stackheap(
4     unsigned R0, unsigned SP, unsigned R2, unsigned SL){
5
6     struct __initial_stackheap config;
7
8     //config.heap_limit = 0x00724B40;
9
10    //config.stack_limit = 0x00004000;
11    //config.stack_limit = 0x00100000;
12
13    /* works */
14    //config.heap_base = 0x01260000;
15    //config.stack_base = 0x01460000;
16
17    config.heap_base = 0x48D97C;
18    config.stack_base = 0x49008C;
19
20
21    /* factory defaults */
22    //config.heap_base = 0x00060000;
23    //config.stack_base = 0x00080000;
24    return config;
}
```

Εικόνα 2.28: Αρχείο stack 5ης αρχιτεκτονικής.



Τα αποτελέσματα με την χρήση αυτής της αρχιτεκτονικής είναι τα παρακάτω.



Εικόνα 2.29: Αποτελέσματα 5ης αρχιτεκτονικής με μία γρήγορη SRAM.

Όπως βλέπουμε οι κύκλοι αναμονής (wait cycles) έχουν μειωθεί πάρα πολύ καθώς είναι περίπου 43.000.000. Επιπλέον έχουν μειωθεί σημαντικά και οι συνολικοί κύκλοι. Το κόστος όμως αυτής της μνήμης είναι αρκετά υψηλό καθώς έχουμε χρησιμοποιήσει μια μεγάλης χωρητικότητας SRAM μνήμη. Με αυτόν τον τρόπο πετυχαίνουμε πολύ σημαντική μείωση των κύκλων αναμονής όμως με πολύ μεγάλο κόστος.



2.4 Συμπεράσματα

Καταλήγουμε σε πέντε παραμέτρους οι οποίες επηρεάζουν το αποτέλεσμα της ταχύτητας εκτέλεσης του αλγορίθμου. Αυτές είναι το πλήθος επιπέδων μνήμης, η ταχύτητα εγγραφής και ανάγνωσης N και S κύκλων κάθε μνήμης, η τοποθέτηση μεταβλητών στις υπομονάδες μνήμης, το μέγεθος του διαύλου και το ρολόι του επεξεργαστή.

Τα σχόλια σχετικά με τις παραμέτρους θα γίνουν σε συνάρτηση με τον πίνακα που ακολουθεί. Σε αυτόν τον πίνακα βλέπουμε συγκεντρωτικά τα αποτελέσματα για κάθε αρχιτεκτονική που δοκιμάσαμε.

Αρχιτεκτονική μνήμης	Wait Cycles	Total	Ποσοστό βελτίωσης Wait Cycles	Ποσοστό βελτίωσης Total
1	585.993.108	1.307.996.594		
2	468.063.182	1.190.070.580	20.1%	9.00%
3	412.412.927	1.134.419.999	29.6%	13.3%
3 με ρολόι 5 Mhz	27.403.770	749.410.842	95.3%	42.7%
4	1.153.209.454	1.875.216.526	-96.8%	-43.36%
5	43.184.161	765.194.292	92.63%	41.5%

Εικόνα 2.30: Πίνακας αποτελεσμάτων.

Αρχικά βλέπουμε ότι για τρία επίπεδα μνήμης (ROM,DRAM,SRAM) έχουμε καλύτερη απόδοση. Η σύγχριση μπορεί να γίνει μεταξύ της πρώτης αρχιτεκτονικής και των αρχιτεκτονικών 2 και 3. Αυτές έχουν το ίδιο μέγεθος διάυλου και ρολόι επεξεργαστή αλλά διαφέρουν στα επίπεδα μνήμης που χρησιμοποιούμε. Οι αρχιτεκτονικές 2 και 3 αποτελούνται από τρία επίπεδα μνήμης και βλέπουμε ότι απόδιδουν αρκετά καλύτερα από την αρχιτεκτονική 1 που αποτελείται από ένα επίπεδο. Στις αρχιτεκτονικές 2 και 3 οι κύκλοι αναμονής και οι συνολικοί κύκλοι εκτέλεσης είναι μικρότεροι από αυτούς που προκύπτουν χρησιμοποιώντας την αρχιτεκτονική 1. Η βελτίωση έγκειται στο γεγονός ότι η μνήμη RAM του τρίτου επιπέδου έιναι πιο γρήγορη από αυτήν του δευτέρου επιπέδου, ενώ επίσης σημαντικό ρόλο παίζει και η επιλογή των μεταβλητών που βάζουμε στην πιο γρήγορη μνήμη τρίτου επιπέδου. Αυτές οι παρατηρήσεις αναλύονται λεπτομερώς παρακάτω.

Όπως φάνηκε και προηγουμένως η ταχύτητα εγγραφής και ανάγνωσης των N και S κύκλων της μνήμης παίζει σημαντικό ρόλο στην ταχύτητα εκτέλεσης του προγράμματος. Η απόδοση με την αρχιτεκτονική 2 είναι καλύτερη από αυτήν που προκύπτει χρησιμοποιώντας την αρχιτεκτονική 1 αφού πλέον ένα μέρος των μεταβλητών του προγράμματος τοποθετείται σε μία μνήμη με πιο γρήγορη εγγραφή και ανάγνωση. Επίσης μπορούμε να δούμε ότι η αρχιτεκτονική 5 απόδιδει καλύτερα από όλες. Όπως είδαμε και στην προηγούμενη ενότητα η αρχιτεκτονική 5 έχει δύο επίπεδα μνήμης, ένα επίπεδο μνήμης ROM και ένα επίπεδο μνήμης RAM. Παρ' όλα αυτά η απόδοσή της βλέπουμε ότι ξεπερνάει και αυτήν των μνημών τριών επιπέδων. Αυτό οφείλεται στο γεγονός ότι η RAM της αρχιτεκτονικής 5 είναι υπερβολικά γρήγορη, άρα μπορεί και επικοινωνεί ταχύτερα με τον επεξεργαστή μειώνοντας έτσι το πλήθος των κύκλων αναμονής καθώς και το πλήθος των συνολικών κύκλων εκτέλεσης. Μία τέτοια μνήμη όμως όπως αυτή της αρχιτεκτονικής 5 δεν είναι μία ρεαλιστική επιλογή για ένα ενσωματωμένο σύστημα. Οι ταχύτητες εγγραφής και ανάγνωσής της θα την καθιστούσαν πολύ ακριβή και έτσι η χρήση της δε συμφέρει για την ανάπτυξη του συστήματος. Αυτό που μπορούμε να κάνουμε είναι να χρησιμοποιήσουμε μνήμες μικρότερων ταχυτήτων στις οποίες όμως θα τοποθετήσουμε μεθοδικά μεταβλητές που χρησιμοποιούνται συχνά



από τον επεξεργαστή. Έτσι το κόστος του ενσωματωμένου συστήματος παραμένει όσο το δυνατόν χαμηλότερο διατηρώντας τα επιθυμητά επίπεδα απόδοσης.

Η ταχύτητα μνήμης φαίνεται ότι από μόνη της δεν μπορεί να καθορίσει την βελτιστη αρχιτεκτονική της μνήμης που θα χρισμοποιηθούμε. Εφόσον το χαμηλό κόστος του ενσωματομένου συστήματος αποτελεί σημαντικό παράγοντα στην ανάπτυξή του, θα πρέπει να συμβιβαζόμαστε σε χαμηλότερες ταχύτητες μνήμης. Αν συγχρίνουμε όμως τις αρχιτεκτονικές 2 και 3 βλέπουμε ότι η τελευταία αποδίδει καλύτερα. Σε σχέση με την βασική αρχιτεκτονική 1 η βελτίωση που φέρνουν οι αρχιτεκτονικές 2 και 3 είναι 20.1% και 29.6% αντίστοιχα. Ας υψηλούμε ότι οι δύο αυτές αρχιτεκτονικές είναι ίδιες με την διαφορά ότι η SRAM της αρχιτεκτονικής 3 είναι μεγαλύτερη και πιο αργή από αυτήν της αρχιτεκτονικής 2. Οι DRAM παραμένουν ίδιες. Στην SRAM της αρχιτεκτονικής 3 βάζουμε και τον πίνακα Εο. Αυτός είναι ένας από τους πιο χρισμοποιούμενους πίνακες του αλγορίθμου και άρα πληροφορίες του διαβάζονται και εγγράφονται πολύ συχνά κατά την εκτέλεση. Είναι λογικό αφού τον μετακινούμε από την DRAM στην γρηγορότερη SRAM ο χρόνος εκτέλεσης του προγράμματος να βελτιώνεται αφού ο χρόνος αναμονής του επεξεργαστή μειώνεται.

Η αρχιτεκτονική 4 υλοποιήθηκε με σκοπό της εξαγωγή συμπερασμάτων σχετικά με την επιλογή του μεγέθους διαύλου των μνημών. Η αρχιτεκτονική αυτή είναι ίδια με την αρχιτεκτονική 3 με την διαφορά ότι για την SRAM επιλέχθηκε μήπως διαύλου 2 αντί για 4. Με αυτόν τον τρόπο τροφοδοτείται η κεντρική μονάδα επεξεργασίας (CPU) με πιο αργό ρυθμό με δεδομένα από την SRAM. Αυτό όπως βλέπουμε από τα αποτελέσματα επηρεάζει σε μεγάλο βαθμό του κύκλους αναμονής και κατά συνέπεια και του συνολικούς κύκλους. Οι κύκλοι αναμονής έχουν σχεδόν διπλασιαστεί σε σχέση με την πρώτη αρχιτεκτονική, ενώ οι συνολικοί κύκλοι έχουν αυξηθεί κατά 43.36%. Από τις αρχιτεκτονικές που εξετάστηκαν είναι η μόνη η οποία δε σημείωσε βελτίωση σε σχέση με την αρχιτεκτονική 1. Με μικρότερες τιμές του μεγέθους διαύλου οι συνολικοί κύκλοι αυξάνονται, ενώ με αντίστοιχη αύξηση του μήκους διαύλου οι κύκλοι συνολικού κύκλου μειώνονται. Έτσι καταλήγουμε στο συμπέρασμα ότι η επιλογή του μεγέθους διαύλου παίζει πολύ σημαντικό ρόλο και πρέπει να επιλέγουμε όσο το δυνατόν μεγαλύτερη τιμή. Η επιλογή της τιμής του μήκους διαύλου πρέπει να είναι ακέραιο πολλαπλάσιο του 2.

Η αρχιτεκτονική 3 με την επιλογή ρολογιού στα 5 Mhz σημείωσε το καλύτερο αποτέλεσμα στο συνολικούς (Total) κύκλους. Αυτό είναι αναμενόμενο καθώς κάνοντας τον επεξεργαστή πιο αργό και μειώνονται οι κύκλοι αναμονής (wait cycles). Αυτό το ρολόι είναι κατά 10 φορές μικρότερο από το ρολόι που χρησιμοποιήθηκε στις υπόλοιπες υλοποιήσεις. Επειδή όμως με αυτόν τον τρόπο μειώνεται η ταχύτητα του επεξεργαστή η λύση της μείωσης της συχνότητας του ρολογιού δεν κρίθηκε ως η καλύτερη δυνατή.

Καταλήγοντας η αρχιτεκτονική που προτείνεται για τον σχεδιασμό του ενσωματομένου συστήματος είναι αρχιτεκτονική 3 με την επιλογή της συχνότητας στα 50 Mhz. Η βελτίωση που έγινε είναι ιδιαίτερα σημαντική καθώς υπήρξε πτώση των κύκλων αναμονής κατά 29.6% και των συνολικών κύκλων κατά 13.3%. Η λύση είναι ρεαλιστική και δεν απαιτεί την επιλογή εξαιρετικά μεγάλου μεγέθους SRAM όπως στην αρχιτεκτονική 5. Επιπλέον δεν χρησιμοποιείται πιο χαμηλή συχνότητα ρολογιού όπως στην αρχιτεκτονική 3 με ρολόι 5Mhz. Σε σχέση με τις αρχιτεκτονικές 2 και 5 έχει αρκετά καλύτερα αποτελέσματα. Έτσι μπορούμε να συμπεράνουμε πως είναι πολύ σημαντική μια έξυπνη κατανομή των μεταβλητών που χρησιμοποιούνται στο πρόγραμμα στις μνήμες. Με την καταχώρηση μεταβλητών που χρησιμοποιούνται πολύ συχνά σε πιο γρήγορη μνήμη αλλά και



με την ταυτόχρονη χρήση μια πιο αργής αλλά και οικονομικής μνήμης DRAM για τις μεταβλητές μεγάλου μεγέθους που δεν χρησιμοποιούνται τόσο συχνά μπορεί να επιτευχθεί το χαλύτερο δυνατό αποτέλεσμα λαμβάνοντας υπόψιν τόσο το οικονομικό όσο και το κομμάτι των κύκλων και της ταχύτητας εκτέλεσης.



Μέρος τρίτο

3.1 Επαναχρησιμοποίηση δεδομένων

Σε πολλές εφαρμογές προκύπτει να χρησιμοποιούμε δεδομένα σε μία λειτουργία, τα οποία έχουμε χρησιμοποιήσει μόλις πριν. Κάθε φορά όμως που χρησιμοποιούμε δεδομένα από την μνήμη επ-ωμιζόμαστε την καθυστέρησης εύρεσης και μεταφοράς του από την μνήμη στον επεξεργαστή. Θα ήταν ωφέλιμο λοιπόν, δεδομένα τα οποία ξέρουμε ότι θα τα χρησιμοποιήσουμε ξανά μετά από μερικές λειτουργίες να τα αποθηκεύουμε προσωρινά σε μία μνήμη η οποία βρίσκεται "πιο κοντά" στον επεξεργαστή. Με την έννοια "πιο κοντά" εννοούμε ότι η μνήμη αυτή θα είναι ταχύτερη από την κύρια μνήμη που βρίσκονται κανονικά τα δεδομένα, ενώ επίσης αυτή η μνήμη μπορεί να βρίσκεται κυριολεκτικά πιο κοντά στον επεξεργαστή. Ένα τέτοιο επίπεδο μνήμης είναι η cache που διαθέτουν πολλοί επεξεργαστές, ενώ οι σύγχρονοι υπολογιστές έχουν περισσότερα από ένα επίπεδα μνήμης cache.

Για να εφαρμόσουμε την τεχνική της επαναχρησιμοποίησης δεδομένων θα πρέπει να εντοπίσουμε στον κώδικα σημεία όπου μεταξύ διαδοχικών λειτουργιών γίνεται κλήση στη μνήμη για τα ίδια δεδομένα. Τότε δημιουργούμε μία δομή με buffers στους οποίους τοποθετούμε προσωρινά τα δεδομένα αυτά, μέχρι πλέον να μην τα χρειαζόμαστε άμεσα. Τα δεδομένα που βρίσκονται στους buffers συνεχώς και όποτε χρειάζεται θα ανανεώνονται με σκοπό σε αυτούς να βρίσκεται μόνο πληροφορία την οποία άμεσα στους επόμενους κύκλους θα χρειαστούμε περισσότερες από μία φορές. Σίγουρα για να πετυχαίνουμε το καλύτερο αποτέλεσμα με την χρήση buffers, θα πρέπει αυτοί να τοποθετούνται σε υπομονάδες μνήμης υψηλής ταχύτητας. Αν όχι σε μία δομή cache, τότε τουλάχιστον στην γρηγορότερη μνήμη που διαθέτει το εκάστοτε σύστημα. Εφαρμογές οι οποίες εμπλέκουν την επεξεργασία και ανάλυση εικόνας καθίστανται ιδανικές για την εφαρμογή της επαναχρησιμοποίησης δεδομένων καθώς η πράξη της συνέλιξης είναι πολύ συχνή σε αυτές.

3.1.1 Επαναχρησιμοποίηση δεδομένων και συνέλιξη

Θα δούμε τώρα γιατί εφαρμογές που χρησιμοποιούν την συνέλιξη μπορούν να επωφεληθούν από την τοποθέτηση buffers και την επαναχρησιμοποίηση δεδομένων. Ας υμηθούμε ότι στην επεξεργασία εικόνας η συνέλιξη αφορά την εφαρμογή φίλτρων σε αυτήν για να επιτύχουμε διάφορα αποτελέσματα όπως ύσλωση ή εύρεση ακμών. Η συνέλιξη εκτελείται πάντα χρησιμοποιώντας ένα μικρό κομμάτι της εικόνας, μία γειτονιά εικονοστοιχείων. Από μόνη της η ανάκτηση ενός μέρους της εικόνας από την μνήμη, όσο μικρό και αν είναι αυτό, όταν γίνεται συνεχόμενα μπορεί να προκαλέσει καθυστερήσεις που οφείλονται μόνο σε αναμονή του επεξεργαστή για τα δεδομένα που χρειάζεται από την μνήμη. Πόσο μάλλον όταν μιλάμε για εφαρμογή της συνέλιξης σε γειτονικά εικονοστοιχεία, τότε γίνεται ανάκτηση ίδιας πληροφορίας ξανά και ξανά.

Στην εικόνα που ακολουθεί βλέπουμε την εφαρμογή κάποιου φίλτρου στα τρία πράσινα εικονοστοιχεία, το ποίο απαιτεί συνέλιξη της γειτονιάς με ακτίνα ένα εικονοστοιχείο από το κεντρικό. Με πορτοκαλί χρώμα φαίνονται τα εικονοστοιχεία τα οποία επικαλύπτονται από δύο γειτονιές, δηλαδή τα χρειάζονται δύο κεντρικά εικονοστοιχεία για τον υπολογισμό. Οπότε θα μπορούσαμε αυτά να τα έχουμε αποθηκευμένα στους buffers ώστε οι συνεχόμενες ανακλήσεις τους να γίνονται πιο γρήγορα σε σχέση με το να ήταν στην κύρια και πιο αργή μνήμη. Καθώς προχωράμε στις επόμενες γραμμές ανανεώνουμε τα εικονοστοιχεία τα οποία είναι αποθηκευμένα στους buffers.



Εικόνα 3.1: Επαναχρησιμοποίηση δεδομένων στην συνέληξη.

3.2 Εισαγωγή buffers στον κώδικα

Για την επίτευξη της επαναχρησιμοποίησης δεδομένων είναι απαραίτητη η υλοποίηση αλλαγών στον κώδικα. Για την αποτελεσματικότερη αξιοποίηση των buffers πρέπει να χρησιμοποιηθούν στην θέση πινάκων που χρησιμοποιούνται συχνά στον κώδικα. Πρώτα από όλα ορίζουμε τους buffers. Στις περιπτώσεις που θα αναλυθούν στην συνέχεια γίνεται ο ορισμός τους σε διαφορετικές μνήμες για να εξεταστεί η επίδραση που έχει αυτός ο ορισμός. Στην ακόλουθη εικόνα φαίνεται ο ορισμός τους στο section με όνομα cache.

```
#pragma arm section zidata="cache"  
int buffer1[N+2];  
int buffer2[N+2];  
int buffer3[N+2];  
#pragma arm section
```

Εικόνα 3.2: Ορισμός των buffers.

Έχουμε ορίσει τρεις line buffers. Το συνολικό επιπλέον μέγεθος που απαιτούν σε μηνημη είναι $(N + 2)x3x32/8$ Bytes. Ο αριθμός αυτό προκύπτει πολλαπλασιάζοντας τον αριθμό των στηλών της εικόνας επί τον αριθμό 3 που είναι το σύνολο των buffers επί 32 που είναι το μέγεθος σε bit της μεταβλητής τύπου int. Διαιρούμε με το 8 για να πάρουμε τον αριθμό σε Bytes. Ξεκινώντας από την συνάρτηση gaussian_blur γίνεται η πρώτη χρησιμοποίηση των buffers. Στην συνάρτηση αυτή υλοποιείται το φίλτρο gaussian στην εικόνα. Για την εφαρμογή αυτού του φίλτρου χρησιμοποιείται μια μάσκα με μέγεθος 3x3. Με την χρήση των buffers μπορούμε για τον υπολογισμό της συνέλιξης για κάθε pixel αντί να διαβάζουμε τις τιμές που απαιτούνται για τον υπολογισμό της από την αργή μνήμη DRAM που είναι αποθηκευμένος ο πίνακας temp να εκμεταλλευτούμε την επαναχρησιμοποίηση δεδομένων.

Κάθε φορά τα pixels της γραμμής που θέλουμε να εξετάσουμε βρίσκονται στον buffer2, ενώ ο buffer1 και ο buffer3 έχουν την πάνω και την κάτω γραμμή αντίστοιχα. Όταν αλλάζουμε γραμμή γίνεται ένα shift στους buffers και μόνο ο buffer3 διαβάζει μια καινούργια γραμμή από την αργή μνήμη DRAM. Ο buffer2 παίρνει την τιμή που είχε πριν ο buffer3 και ο buffer1 παίρνει την τιμή που είχε πριν ο buffer2. Έτσι χρησιμοποιούμε μια γρήγορη μνήμη για να γράψουμε και να διαβάσουμε τις περισσότερες νέες τιμές που χρειαζόμαστε. Για την σωστή υλοποίηση της λογικής του κώδικα μας χρησιμοποιείται πλέον μια μονή for στην οποία έχει υλοποιηθεί Unroll για τον υπολογισμό των newPixel, newPixel2, newPixel3 και newPixel4. Αφού υπολογιστούν οι τιμές αποθηκεύονται στον πίνακα output_gaussian.



```

// =====
// blur image using Gaussian kernel
void gaussian_blur()
{
    for(i=0;i<(M+2);i++)
    {
        for(j=0;j<(N+2);j+=4)
        {
            temp[i][j]=0;
            temp[i][j+1]=0;
            temp[i][j+2]=0;
            temp[i][j+3]=0;
        }
    }

    for(i=1;i<(M+1);i++)
    {
        for(j=1;j<(N+1);j+=4)
        {
            temp[i][j]=current[i-1][j-1];
            temp[i][j+1]=current[i-1][j];
            temp[i][j+2]=current[i-1][j+1];
            temp[i][j+3]=current[i-1][j+2];
        }
    }

    // declare Gaussian mask
    gaussianMask[0][0] = 1; gaussianMask[0][1] = 2; gaussianMask[0][2] = 1;
    gaussianMask[1][0] = 2; gaussianMask[1][1] = 4; gaussianMask[1][2] = 2;
    gaussianMask[2][0] = 1; gaussianMask[2][1] = 2; gaussianMask[2][2] = 1;
    // for every pixel
    for (row=limit_gauss;row<=M+limit_gauss;row++){
        if (row==1) {

            for(k=0;k<N+2;k++) {
                buffer1[k] = temp[0][k];
                buffer2[k] = temp[1][k];
                buffer3[k] = temp[2][k];
            }
        }
    }
}

```

Εικόνα 3.3: Συνάρτηση gaussian_blur.

```

} else {

    for (k=0;k<N+2;k++) {
        buffer1[k] = buffer2[k];
        buffer2[k] = buffer3[k];
        buffer3[k] = temp[row+1][k];
    }
}

for (col=limit_gauss;col<=N+limit_gauss;col+=4){
    newPixel = 0;
    newPixel2 = 0;
    newPixel3 = 0;
    newPixel4 = 0;

    // for every kernel pixel
    for (kernelCol=-limit_gauss;kernelCol<=limit_gauss; kernelCol++){
        newPixel = newPixel + buffer1[col+kernelCol]*gaussianMask[0][limit_gauss+kernelCol];
        newPixel = newPixel + buffer2[col+kernelCol]*gaussianMask[1][limit_gauss+kernelCol];
        newPixel = newPixel + buffer3[col+kernelCol]*gaussianMask[2][limit_gauss+kernelCol];

        newPixel2 = newPixel2 + buffer1[col+1+kernelCol]*gaussianMask[0][limit_gauss+kernelCol];
        newPixel2 = newPixel2 + buffer2[col+1+kernelCol]*gaussianMask[1][limit_gauss+kernelCol];
        newPixel2 = newPixel2 + buffer3[col+1+kernelCol]*gaussianMask[2][limit_gauss+kernelCol];

        newPixel3 = newPixel3 + buffer1[col+2+kernelCol]*gaussianMask[0][limit_gauss+kernelCol];
        newPixel3 = newPixel3 + buffer2[col+2+kernelCol]*gaussianMask[1][limit_gauss+kernelCol];
        newPixel3 = newPixel3 + buffer3[col+2+kernelCol]*gaussianMask[2][limit_gauss+kernelCol];

        newPixel4 = newPixel4 + buffer1[col+3+kernelCol]*gaussianMask[0][limit_gauss+kernelCol];
        newPixel4 = newPixel4 + buffer2[col+3+kernelCol]*gaussianMask[1][limit_gauss+kernelCol];
        newPixel4 = newPixel4 + buffer3[col+3+kernelCol]*gaussianMask[2][limit_gauss+kernelCol];
    }
}

output_gaussian[row][col] = (int)(floor(newPixel)/16);
output_gaussian[row][col+1] = (int)(floor(newPixel2)/16);
output_gaussian[row][col+2] = (int)(floor(newPixel3)/16);
output_gaussian[row][col+3] = (int)(floor(newPixel4)/16);
}
}

```

Εικόνα 3.4: Συνάρτηση gaussian_blur.



Το δεύτερο σημείο στο οποίο γίνεται χρήση των buffers είναι στην συνάρτηση sobel. Σε αυτή την συνάρτηση γίνεται χρήση δύο μασκών και η χρήση των buffers είναι ιδιαίτερα σημαντική. Σε παρόμοια λογική με την συνάρτηση gaussian_blur για η γραμμή στην οποία βρίσκεται το pixel που εξετάζεται είναι αποθηκευμένη στον buffer2, η προηγούμενη γραμμή στον buffer1 και η επόμενη γραμμή στον buffer3. Όταν γίνεται αλλαγή γραμμής στον buffer3 γίνεται εγγραφή τιμών από την αργή μνήμη, ο buffer2 παίρνει την τιμή του buffer3 και ο buffer1 παίρνει την τιμή του buffer2. Έτσι με έξυπνο τρόπο μπορούμε να μειώσουμε τις προσπελάσεις στην αργή μνήμη και τα δεδομένα που επαναχρησιμοποιούνται να βρίσκονται στην πιο γρήγορη μνήμη. Με αυτόν τον τρόπο πετυχαίνουμε μικρότερους χρόνους ανάγνωσης των τιμών των pixels για τον υπολογισμό των συνελίξεων. Στις ακόλουθες εικόνες φαίνεται ο κώδικας υλοποίησης:

```
// =====
// find gradients
void sobel()
{
    // declare Sobel masks
    GxMask[0][0] = -1; GxMask[0][1] = 0; GxMask[0][2] = 1;
    GxMask[1][0] = -2; GxMask[1][1] = 0; GxMask[1][2] = 2;
    GxMask[2][0] = -1; GxMask[2][1] = 0; GxMask[2][2] = 1;

    GyMask[0][0] = 1; GyMask[0][1] = 2; GyMask[0][2] = 1;
    GyMask[1][0] = 0; GyMask[1][1] = 0; GyMask[1][2] = 0;
    GyMask[2][0] = -1; GyMask[2][1] = -2; GyMask[2][2] = -1;

    // compute Igx and Igy
    for (row=limit;row<=(M+limit);row++){

        if (row==1) {

            for(k=0;k<N+2;k++) {
                buffer1[k] = output_gaussian[0][k];
                buffer2[k] = output_gaussian[1][k];
                buffer3[k] = output_gaussian[2][k];
            }
        } else {

            for (k=0;k<N+2;k++) {
                buffer1[k] = buffer2[k];
                buffer2[k] = buffer3[k];
                buffer3[k] = output_gaussian[row+1][k];
            }
        }

        for (col=limit;col<=(N+limit);col+=4){

            newPixel = 0;
            newPixel2 = 0;
            newPixel3 = 0;
            newPixel4 = 0;

            newPixelx = 0;
            newPixel2y = 0;
            newPixel3y = 0;
            newPixel4y = 0;
        }
    }
}
```

Εικόνα 3.5: Συνάρτηση sobel.



```
for (kernelCol=-limit;kernelCol<=limit; kernelCol++){
    newPixel = newPixel + buffer1[col+kernelCol]*GxMask[0][limit_gauss+kernelCol];
    newPixel = newPixel + buffer2[col+kernelCol]*GxMask[1][limit_gauss+kernelCol];
    newPixel = newPixel + buffer3[col+kernelCol]*GxMask[2][limit_gauss+kernelCol];

    newPixel2 = newPixel2 + buffer1[col+1+kernelCol]*GxMask[0][limit_gauss+kernelCol];
    newPixel2 = newPixel2 + buffer2[col+1+kernelCol]*GxMask[1][limit_gauss+kernelCol];
    newPixel2 = newPixel2 + buffer3[col+1+kernelCol]*GxMask[2][limit_gauss+kernelCol];

    newPixel3 = newPixel3 + buffer1[col+2+kernelCol]*GxMask[0][limit_gauss+kernelCol];
    newPixel3 = newPixel3 + buffer2[col+2+kernelCol]*GxMask[1][limit_gauss+kernelCol];
    newPixel3 = newPixel3 + buffer3[col+2+kernelCol]*GxMask[2][limit_gauss+kernelCol];

    newPixel4 = newPixel4 + buffer1[col+3+kernelCol]*GxMask[0][limit_gauss+kernelCol];
    newPixel4 = newPixel4 + buffer2[col+3+kernelCol]*GxMask[1][limit_gauss+kernelCol];
    newPixel4 = newPixel4 + buffer3[col+3+kernelCol]*GxMask[2][limit_gauss+kernelCol];

    newPixelx = newPixelx + buffer1[col+kernelCol]*GyMask[0][limit_gauss+kernelCol];
    newPixelx = newPixelx + buffer2[col+kernelCol]*GyMask[1][limit_gauss+kernelCol];
    newPixelx = newPixelx + buffer3[col+kernelCol]*GyMask[2][limit_gauss+kernelCol];

    newPixel2y = newPixel2y + buffer1[col+1+kernelCol]*GyMask[0][limit_gauss+kernelCol];
    newPixel2y = newPixel2y + buffer2[col+1+kernelCol]*GyMask[1][limit_gauss+kernelCol];
    newPixel2y = newPixel2y + buffer3[col+1+kernelCol]*GyMask[2][limit_gauss+kernelCol];

    newPixel3y = newPixel3y + buffer1[col+2+kernelCol]*GyMask[0][limit_gauss+kernelCol];
    newPixel3y = newPixel3y + buffer2[col+2+kernelCol]*GyMask[1][limit_gauss+kernelCol];
    newPixel3y = newPixel3y + buffer3[col+2+kernelCol]*GyMask[2][limit_gauss+kernelCol];

    newPixel4y = newPixel4y + buffer1[col+3+kernelCol]*GyMask[0][limit_gauss+kernelCol];
    newPixel4y = newPixel4y + buffer2[col+3+kernelCol]*GyMask[1][limit_gauss+kernelCol];
    newPixel4y = newPixel4y + buffer3[col+3+kernelCol]*GyMask[2][limit_gauss+kernelCol];
}

Igx[row][col] = (int)(floor(newPixel));
Igx[row][col+1] = (int)(floor(newPixel2));
Igx[row][col+2] = (int)(floor(newPixel3));
Igx[row][col+3] = (int)(floor(newPixel4));

Igy[row][col] = (int)(floor(newPixelx));
Igy[row][col+1] = (int)(floor(newPixel2y));
Igy[row][col+2] = (int)(floor(newPixel3y));
Igy[row][col+3] = (int)(floor(newPixel4y));
```

Εικόνα 3.6: Συνάρτηση sobel.

3.3 Δοκιμές δομών buffers

Παρακάτω αναλύουμε πως απέδωσαν τρεις διαφορετικές προσεγγίσεις στην τοποθέτηση των buffers στις υπομονάδες μνήμης. Κάθε φορά έχουμε τρεις buffers με μέγευθος όσο και οι στήλες τις εικόνας. Οι αλλαγές έγκειται στην υπομονάδα μνήμης που βρίσκονται οι buffers κάθε φορά.

3.3.1 Δοκιμή 1

Αρχικά τρέχουμε των κώδικα που αναλύσαμε στο μέρος 3.3 τοποθετώντας τους buffers στην DRAM. Αυτό το κάνουμε αρχικοποιώντας τις μεταβλητές buffer1, buffer2, buffer3 εκτός του block κώδικα pragma ram. Έτσι αυτόματα μέσω του αρχείο scatter τοποθετούνται στην DRAM, όπου δηλαδή και οι υπόλοιπες μεταβλητές τύπου ZI. Στη συνέχεια πρέπει να αλλάξουμε τις ρυθμίσεις των αρχείων memory.MAP Και scatter ως προς τις διεύθυνσης μνήμης έναρξης των υπομονάδων μνήμης και τα μεγέθη τους. Η ROM αυξάνεται λόγο της αύξησης του κώδικα, ενώ επίσης και η DRAM αφού σε αυτήν τοποθετούμε τους buffers. Μαζί με την αλλαγή της RAM θα πρέπει να αλλάξουν και οι διεύθυνσης έναρξης των stack και heap δομών. Τέλος θα πρέπει να αλλάξει και η διεύθυνση έναρξης της SRAM. Από το make log μπορούμε να δούμε ακριβώς τα απαιτούμενα μεγέθη και να υπολογίσουμε τις διευθύνσεις έναρξης των υπομονάδων και των stack, heap δομών. Παραμετροποιούμε το αρχείο memory.MAP όπως φάινεται παρακάτω.

Στη συνέχεια παραμετροποιούμε τα αρχεία scatter Και stack αντίστοιχα όπως φαίνεται στις παρακάτω εικόνες.

Τρέχουμε τον κώδικα και πάρνουμε τα εξής αποτελέσματα.



```
1 00000000 636C ROM 4 R 1/1 1/1
2 636C 416F34 DRAM 4 RW 250/50 250/50
3 41D2A0 757E0 SRAM 4 RW 30/10 30/10
```

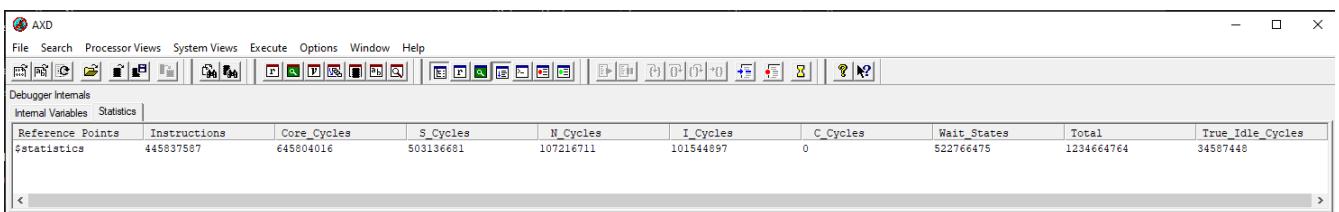
Εικόνα 3.7: Memory.MAP αρχείο.

```
1 ROM 0x0 0x638C
2 {
3 ROM 0x0 0x636C
4 {
5 *.o ( +RO )
6 }
7 DRAM 0x636C 0x414824
8 {
9 * ( +ZI,+RW )
10 }
11 SRAM 0x41D2A0 0x757E0
12 {
13 * (ram)
14 }
15 }
16 }
```

Εικόνα 3.8: Memory.MAP αρχείο.

```
17 //buffers out
18 config.heap_base = 0x41AB90;
19 config.stack_base = 0x41D2A0;
```

Εικόνα 3.9: Scatter αρχείο.



Εικόνα 3.10: Αποτελέσματα δοκιμής 1.

Βλέπουμε ότι το πλήθος των wait states και των συνολικών κύκλων έχει αυξηθεί από την εκτέλεση της αρχιτεκτονικής 3 του δεύτερου μέρους της εργασίας. Αυτό είναι λογικό αφού προσθέντας τους buffers προσθέτουμε και το overhead του γεμίσματος τους καθώς και οποιοδήποτε άλλο overhead λόγο αλλαγής του κώδικα για να εφαρμοστεί η επαναχρησιμοποίηση δεδομένων με του buffers. Επίσης βάζοντας τους buffers στην αργή RAM, εκεί δηλαδή που ήδη βρίσκονται και οι πίνακες με τους οποίους γεμίζουμε τους buffers δεν κερδίζουμε και τίποτα σε κύκλους. Άρα από την μία αυξάνουμε το overhead χωρίς να φροντίζουμε οι buffers να προσπελάζονται πιο γρήγορα απ' ότι η μνήμη στην οποία βρίσκονται ήδη οι πίνακες που περιέχουν την πληροφορία που επαναχρησιμοποιείται.

3.3.2 Δοκιμή 2

Στην δεύτερη δοκιμή γίνεται εισαγωγή των τριών line buffers στην SRAM με ταχύτητες ανάγνωσης εγγραφής 30/10 σε διαδοχικές και μη διαδοχικές κλήσεις της. Αρχικά ορίζουμε τους buffers στο section με όνομα ram το οποίο εισάγουμε στην μνήμη SRAM στο αρχείο memory.MAP. Λόγω της εισαγωγής των buffers αλλά και λόγω της προσθήκης bits του κώδικα μας απαιτείται ιδιαίτερη προσοχή στην αλλαγή των τιμών των διευθύνσεων και των μεγεθών για τις μνήμες μας. Για την επίτευξη της καλύτερης δυνατής βελτιστοποίησης και της χρήσης των μικρότερων δυνατών μνημών που επαρκούν για την υλοποίησή μας επιλέχθηκαν οι ελάχιστες τιμές των μνημών με την



χρήση των αποτελεσμάτων από το Make, αφού κάνουμε compile τα αρχεία μας.

```
#pragma arm section zidata="ram"
int buffer1[N+2];
int buffer2[N+2];
int buffer3[N+2];
int Eo[M+2][N+2];
int gaussianMask[3][3];
int GxMask[3][3];
int GyMask[3][3];
int i, j, k, newPixel, newPixel2, newPixel3, newPixel4, newPixelY;
int newPixel12y, newPixel3y, newPixel4y;
int row, col, kernelRow, kernelCol;
int limit=1;
int limit_gauss=1;
#pragma arm section
```

Εικόνα 3.11: Ορισμός buffers στην δεύτερη δοκιμή.

```
00000000 636C ROM 4 R 1/1 1/1
636C 415B6C DRAM 4 RW 250/50 250/50
41BED8 757E0 SRAM 4 RW 30/10 30/10
```

Εικόνα 3.12: Memory.MAP αρχείο στην δεύτερη δοκιμή.

```
ROM 0x0 0x638C
{
ROM 0x0 0x636C
{
*.o (+RO )
}
DRAM 0x636C 0x41345C
{
*( +ZI,+RW )
}
SRAM 0x41BED8 0x757E0
{
*( ram)
}

}
```

Εικόνα 3.13: Scatter αρχείο στην δεύτερη δοκιμή.

Από τα αποτελέσματα βλέπουμε ότι έχει υπάρξει μείωση των συνολικών κύκλων καθώς ο αριθμός των συνολικών κύκλων είναι 1.109.516.400, ενώ των κύκλων αναμονής (wait states) είναι 397.618.111. Υπάρχει δηλαδή μείωση των συνολικών κύκλων κατά 10% σε σχέση με την δοκιμή 1 και κατά 0.022% σε σχέση με τα αποτελέσματα της αρχιτεκτονικής 3 χωρίς την χρήση των buffers. Έτσι μπορούμε να πούμε ότι η χρήση τους μας βοηθάει στην μείωση των συνολικών κύκλων αλλά και των κύκλων αναμονής.



```
#include <rt_misc.h>

_value_in_regs struct __initial_stackheap __user_initial_stackheap(
    unsigned R0, unsigned SP, unsigned R2, unsigned SL){

    struct __initial_stackheap config;

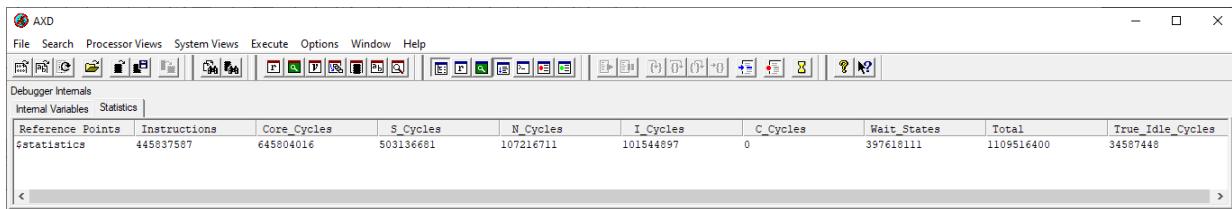
    //config.heap_limit = 0x000724B40;
    //config.stack_limit = 0x000004000;
    //config.stack_limit = 0x00100000;

    /* works */

    config.heap_base = 0x4197C8;
    config.stack_base = 0x41BED8;

    /* factory defaults */
    //config.heap_base = 0x00060000;
    //config.stack_base = 0x00080000;
    return config;
}
```

Εικόνα 3.14: Stack αρχείο στην δεύτερη δοκιμή.



Εικόνα 3.15: Αποτελέσματα δεύτερης δοκιμής.

3.3.3 Δοκιμή 3

Τώρα τοποθετούμε τους buffers σε μία νέα υπομονάδα μνήμης, ταχύτερη αλλά μικρότερη από τις άλλες ώστε να δούμε αν θα αποδώσουν καλύτερα. Φτιάχνουμε την υπομονάδα μνήμης cache η οποία ανταποκρίνεται σε μία αντίστοιχη μνήμη cache, όπως δηλώνει και το όνομά της. Το μέγευθός της είναι τόσο όσο και το μέγευθος των τριών buffers συνολικά, δηλαδή 5064 bytes. Εφόσον είναι μία πολύ μικρή μνήμη την κάνουμε πολύ γρήγορη ώστε να εκμεταλλευτούμε το trade off μεταξύ μεγέθους και ταχύτητας. Ορίζουμε να έχει ταχύτητα 1/1 για read αλλά και για write λειτουργίες. Παραμετροποιούμε τα αρεχεία memory.MAP, scatter και stack όπως φαίνεται στις παρακάτω εικόνες.

```
00000000 6380 ROM 4 R 1/1 1/1
6380 415B6C DRAM 4 RW 250/50 250/50
41BEEC 757E0 SRAM 4 RW 30/10 30/10
4916CC 13C8 CACHE 4 RW 1/1 1/1
```

Εικόνα 3.16: Memory.MAP αρχείο.



```
ROM 0x0 0x63A0
{
ROM 0x0 0x6380
{
*.o ( +RO )
}
DRAM 0x6380 0x41345C
{
* ( +ZI,+RW )
}
SRAM 0x41BEEC 0x757E0
{
* (ram)
}
CACHE 0x4916CC 0x13C8
{
* (cache)
}

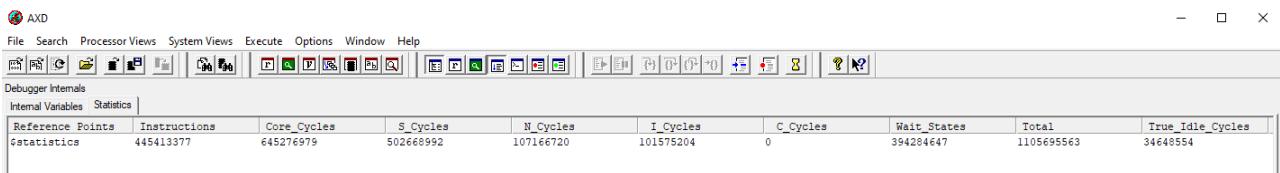
}
```

Εικόνα 3.17: Αρχείο scatter.

```
21 //cache
22 config.heap_base = 0x4197DC;
23 config.stack_base = 0x41BEEC;
```

Εικόνα 3.18: Αρχείο stack.

Τρέχουμε τον κώδικα και παίρνουμε τα εξής αποτελέσματα.



Εικόνα 3.19: Αποτελέσματα 3ης δοκιμής.

Ο αριθμός των wait states και των συνολικών κύκλων έχει ελαφρώς μειωθεί σε σχέση με την δοκιμή 2. Αναλογιζόμενοι και το πιθανό κόστος μίας μνήμης σαν cache που προτείναμε στην δοκιμή 3 ίσως και να μην συμφέρει για το ποσοστό της βελτίωσης.



3.4 Συμπεράσματα

Στον παρακάτω πίνακα βλέπουμε τα αποτελέσματα συνολικά για τις τρεις δοκιμές που κάναμε.

Αρχιτεκτονική μνήμης	Wait Cycles	Total	Ποσοστό βελτίωσης Wait Cycles	Ποσοστό βελτίωσης Total
3	412.412.927	1.134.419.999		
1η δοκιμή	522.766.475	1.234.664.764	-26.76%	-0.09%
2η δοκιμή	397.618.111	1.109.516.400	0.036%	0.022%
3η δοκιμή	394.284.647	1.105.695.563	0.044%	0.026%

Εικόνα 3.20: Πίνακας αποτελεσμάτων.

Παρατηρούμε ότι η εφαρμογή της τεχνικής επαναχρησιμοποίησης δεδομένων δίνει μία μικρή βελτίωση στο πλήθος των κύκλων αναμονής και των συνολικών κύκλων. Για να υπάρχει όμως βελτίωση θα πρέπει οι buffers να τοποθετούνται σε αρκετά γρήγορη μνήμη, τουλάχιστον στο αμέσως επόμενο πιο γρήγορο επίπεδο μνήμης από αυτό που βρίσκονται αποθηκευμένα τα δεδομένα που γράφονται στους buffers. Αυτό είναι λογικό αφού με την προσθήκη των buffers προστίθεται και overhead ενώ δεν κερδίζουμε τίποτα σε κύκλους αφού παίρνουμε δεδομένα και τα εγγράφουμε στην ίδια υπομονάδα μνήμης πριν τα στείλουμε στον επεξεργαστή. Όπως φαίνεται από την δοκιμή 1, η τοποθέτηση των buffers στην ίδια υπομονάδα μνήμης με τα δεδομένα δίνει χειρότερα αποτελέσματα από την υλοποίηση της αρχιτεκτονικής 3 χωρίς αυτούς. Στη συνέχεια αν δούμε την δοκιμή 2 βλέπουμε ότι έχουμε μία μικρή βελτίωση στο πλήθος των κύκλων. Οι buffers πλέων είναι τοποθετημένοι στην πιο γρήγορη υπομονάδα μνήμης που διαθέτει το ενσωματωμένο σύστημα η οποία είναι πολύ πιο γρήγορη από την υπομονάδα μνήμης που βρίσκονται κανονικά τα δεδομένα κατά την εκτέλεση του προγράμματος. Τώρα τα δεδομένα που θέλουμε άμεσα διαθέσιμα βρίσκονται "πιο κοντά" στον επεξεργαστή, γι αυτό βλέπουμε και το πλήθος των κύκλων αναμονής και των συνολικών κύκλων μειώνεται να μειώνεται. Στην δοκιμή 3 βάζουμε ένα ακόμη επίπεδο μνήμης με ταχύτητα εγγραφής και ανάγνωσης 1/1. Αυτή είναι μία πολύ γρήγορη και πολύ ακριβή μνήμη, οπότε το μέγεθός της θέλουμε να το περιορίσουμε ώστε μόνο να χωράει τους buffers. Σε αυτή την δοκιμή βλέπουμε μία πολύ μικρή βελτίωση των κύκλων αναμονής και των συνολικών κύκλων σε σχέση με την δοκιμή 2. Άρα αναλογιζόμενοι και το πιθανό κόστος του νέου επιπέδου μνήμης που προσθέτουμε ίσως να μην συμφέρει.