ΔΗΜΟΚΡΙΤΕΙΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΡΑΚΗΣ

ΠΟΛΥΤΕΧΝΙΚΗ ΣΧΟΛΗ

ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΣΧΕΔΙΑΣΜΟΣ ΕΝΣΩΜΑΤΩΜΕΝΩΝ ΣΥΣΤΗΜΑΤΩΝ

Φίλτρο Μέσης Τιμής για την Επεξεργασία Εικόνας
1ο Μέρος

Διδάσκων

Συρακούλης Γεώργιος

Ομάδα 24

Λαζαρίδου Νίνα ΑΜ:57260

Μούσλεχ Στυλιανός Γεώργιος ΑΜ:57382

Εισαγωγή

Στο πρώτο μέρος της εργασίας καλούμαστε να υλοποιήσουμε ένα συγκεκριμένο αλγόριθμο επεξεργασίας εικόνας και να τον βελτιστοποιήσουμε αυξάνοντας την απόδοση του αλγορίθμου.

Αρχικά περιγράφουμε τον αλγόριθμο που μας ζητήθηκε. Στη συνέχεια περιγράφουμε την βασική υλοποίηση με βάση τον ψευδοκώδικα που μας δόθηκε. Έπειτα εφαρμόζουμε μετασχηματισμούς βρόγχων και αλγοριθμικές τεχνικές βελτιστοποίησης ξεχωριστά κάθε φορά στην αρχική υλοποίηση για να μελετήσουμε την επίδραση της κάθε μεθόδου. Μετά συνδυάζουμε τις καλύτερες και φτιάχνουμε την τελική μας υλοποίηση. Για τις μετρήσεις χρησιμοποιήθηκε ο προσομοιωτής Armulator. Αφού το κάνουμε αυτό βρίσκουμε το μέγεθος του πίνακα δεδομένων και τον αριθμό των προσπελάσεων που γίνονται σε αυτόν στην αρχική και στην τελική μας έκδοση. Τέλος σχολιάζουμε συγκεντρωτικά τα αποτελέσματα και καταγράφουμε τα τελικά μας συμπεράσματα.

Περιγραφή αλγορίθμου

Στην παρούσα εργασία, καλούμαστε να υλοποιήσουμε ένα «Φίλτρο Μέσης Τιμής για την επεξεργασία εικόνας». Η βασική ιδέα αυτού του φίλτρου (γνωστό και ως mean or average filter) είναι να αντικαταστήσουμε την τιμή της φωτεινότητας του κάθε pixel μιας εικόνας με τον μέσο όρο των τιμών μιας «γειτονιάς» γύρω από το pixel συμπεριλαμβάνοντας και το ίδιο το pixel. Το αποτέλεσμα αυτής της επεξεργασίας στην εικόνα είναι να αποκτήσουμε μια πιο smooth/blurry έκδοση της εικόνας. Η ένταση του αποτελέσματος εξαρτάται από το μέγεθος του παραθύρου γύρω από το κάθε pixel.

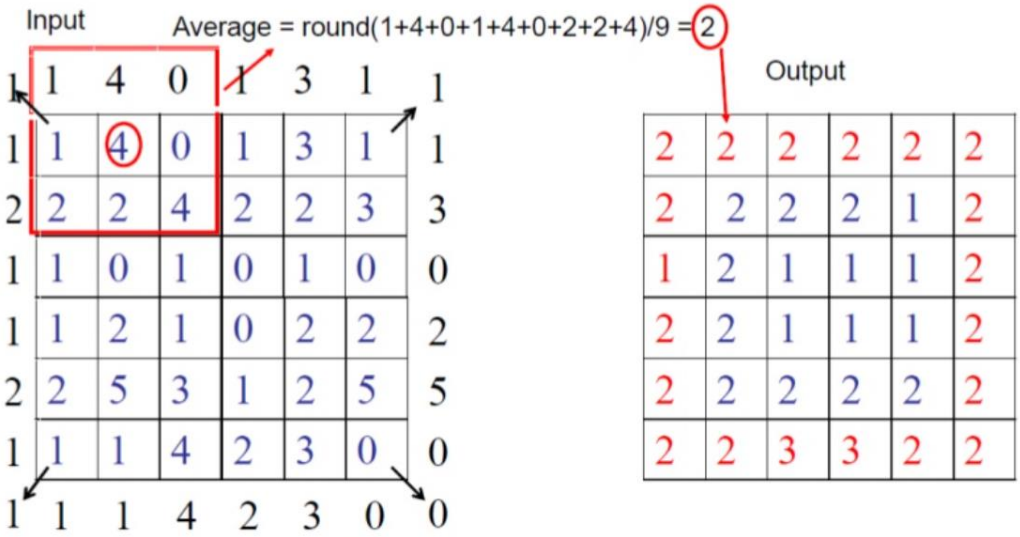
Δηλαδή έστω Y_{ij} η τιμή της φωτεινότητας για ένα pixel στην αρχική εικόνα μεγέθους N x M, τότε:

$\forall i, j \in N, M$

Για $Window\ Radius = 1 \Longrightarrow Kernel\ = 3x3$

$$Y_{i,j}^{new} = round\left(\frac{Y_{i,j} + Y_{i-1,j-1} + Y_{i-1,j} + Y_{i-1,j+1} + Y_{i,j-1} + Y_{i,j+1} + Y_{i+1,j-1} + Y_{i+1,j} + Y_{i+1,j+1}}{(2 \cdot WindowRadius + 1)^2}\right)$$

Παρατηρούμε ότι στα όρια της εικόνα θα πρέπει να πάρουμε κάποια απόφαση καθώς το Kernel μας βγαίνει εκτός εικόνας. Με βάση την εκφώνηση που μας δόθηκε θα δίνουμε τιμές στα pixels εκτός ορίων ίδιες με αυτά στα όρια , δηλαδή Για παράδειγμα:



Περιγραφή Βασικής Υλοποίησης

Η βασική υλοποίηση υπάρχει στο αρχείο «**meanFilter_Initial.c**».

Στην βασική μας υλοποίηση ακολουθήσαμε πιστά τον παρακάτω ψευδοκώδικα που μας δόθηκε στην εκφώνηση:

Κώδικας αλγορίθμου

```
procedure MeanFilter(WindowRadius:Integer);
var
      x,y,i,j,xx,yy : integer
      New: Array of array of integer;
begin
      SetLength(New,ImgWidth,ImgHeight);
      for x:=0 to ImgWidth-1 do
            for y:=0 to ImgHeight-1 do
            begin
            New[x,y]:=0;
            for i:=-WindowRadius to WindowRadius do
                  for j:=-WindowRadius to WindowRadius do
                  begin
                  xx:=Min(Max(x+i,0),ImgWidth-1);
                  yy:=Min(Max(y+j,0),ImgHeight-1);
                  New[x,y]:=New[x,y]+Image[xx,yy];
                  end;
            end;
            New[x,y]:=round(New[x,y]/sqr(2*WindowRadius+1));
      end;
end;
```

Πιο συγκεκριμένα , διαβάζαμε από την εικόνα μόνο το κανάλι Y , καθώς δουλεύουμε μόνο με την φωτεινότητα της εικόνας (Greyscale image) και γράφει το καινούριο κανάλι Y. Επιπλέον ο αλγόριθμος μας είναι παραμετροποιήσιμος όπως στην εκφώνηση για διαφορετικά μεγέθη ακτίνας της μάσκας.

(Μια μικρή υποσημείωση είναι ότι επειδή είχε κάποιο πρόβλημα ο Compiler του Armulator από τη χρήση της round() κάναμε εξαρχής μια δικιά μας συνάρτηση round() .)

Εδώ μπορούμε να δούμε και αποτελέσματα σε εικόνες και με διαφορετικό Window Radius:

1. Αρχική Εικόνα

2. Εικόνα με RADIUS=1

3. Εικόνα με RADIUS=2

4. Εικόνα με RADIUS=3

Στη συνέχεια της εργασίας θα κάνουμε μετρήσεις και συγκρίσεις για RADIUS=2

Τρέχοντας τον κώδικα στον Armulator , έχουμε τα παρακάτω αποτελέσματα :

Version	Instructions	Core Cycles	S Cycles	N Cycles	I Cycles	C Cycles	Total
1. Initial	194471492	348793219	248238865	84426542	29461316	0	362126723

Version	Description	Code	RO Data	RW Data	ZI Data	Debug	Total RO	Total RW	Total ROM
1. Initial	Object Totals	764	60	0	1161600	6044	21938 (21.42kB)	1161900 (1134.67kB)	21938 (21.42kB)
	Library Totals	20656	458	0	300	7292			
	Grand Totals	21420	518	0	1161900	13336			

Περιγραφή Μετασχηματισμών / Βελτιστοποιήσεων

Όπως αναφέραμε και στην εισαγωγή έχουμε 15 συνολικά εκδόσεις του κώδικα. Στην αρχή δοκιμάζουμε μετασχηματισμούς και βελτιστοποιήσεις στον αρχικό κώδικα ξεχωριστά την κάθε αλλαγή με σκοπό να δούμε καλύτερα τι βοηθάει ξεχωριστά και όχι προσθετικά. Στην συνέχεια συνδυάζοντας τις καλύτερες βελτιστοποιήσεις κάναμε 3 συνδυαστικές υλοποιήσεις που ακολουθούν «ξεχωριστές φιλοσοφίες» με σκοπό να βρούμε ποια συνδυαστική έκδοση είναι η καλύτερη, για να κρατήσουμε και ως τελική.

Παρακάτω αναλύουμε την κάθε έκδοση και το σκεπτικό πίσω από την καθεμιά καθώς και τα αποτελέσματα της κάθε έκδοσης ξεχωριστά σε σχέση με την αρχική. Αναλυτική σύγκριση των αποτελεσμάτων γίνεται στο τέλος της παρούσας εργασίας.

2η Έκδοση: Χρήση τοπικής μεταβλητής Sum

Ο κώδικας της έκδοσης βρίσκεται στο αρχείο : «**meanFilter_LocalSum.c**».

Στη συγκεκριμένη έκδοση κάνουμε μια μικρή αλλαγή εντός της συνάρτησης meanFilter() και πιο συγκεκριμένα στον υπολογισμό του αθροίσματος των τιμών της φωτεινότητας των pixel γύρω από το pixel που θέλουμε να υπολογίσουμε. Στην αρχική έκδοση προσθέτουμε την κάθε τιμή φωτεινότητας του pixel απευθείας στην θέση του pixel στον καινούριο πίνακα

```
newImg[y][x]=newImg[y][x] + image_Y[yy][xx];
```

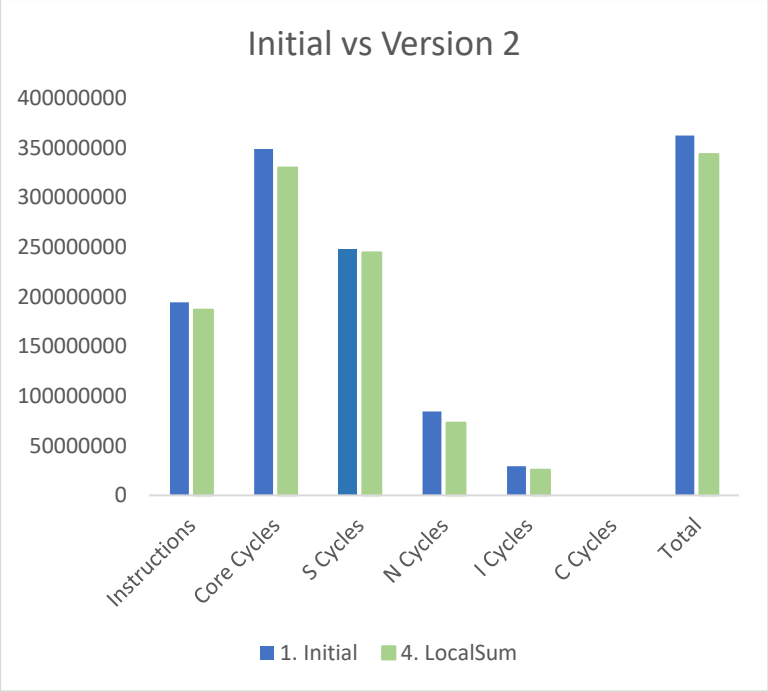
Μπορούμε να ορίσουμε μια τοπική μεταβλητή **sum** όπου εκεί θα προσθέτουμε την φωτεινότητα των pixel του kernel πριν διαιρέσουμε το άθροισμα με τον μέγεθος του kernel και τοποθετήσουμε το στρογγυλοποιημένο αποτέλεσμα στη θέση του νέου πίνακα έτσι:

```
sum=sum + image_Y[yy][xx];
```

Η διαφορά είναι ότι ουσιαστικά αν ζητάμε συνέχεια μια συγκεκριμένη θέση πίνακα από την μνήμη ο επεξεργαστής κάθε φορά πρέπει να πάρει την θέση μνήμης της πρώτης θέσης του πίνακα και μετά να κάνει πράξη για να πάει στη θέση του πίνακα που του λέμε, ενώ αν έχουμε μια απλή μεταβλητή μπορεί να την προσπελάσει κατευθείαν μειώνοντας έτσι τους κύκλους.

Τρέχοντας τον κώδικα έχουμε τα παρακάτω αποτελέσματα:

Version	Instructions	Core Cycles	S Cycles	N Cycles	I Cycles	C Cycles	Total
1. Initial	194471492	348793219	248238865	84426542	29461316	0	362126723
2. LocalSum	187066291	330062422	244608865	73100944	25686117	0	343395926



Παρατηρούμε ότι έχουμε μια μείωση των συνολικών κύκλων και της τάξης περίπου των 19 εκατομμύρια κύκλων. Έχουμε και λογική πτώση των instructions καθώς και της κάθε ξεχωριστής κατηγορίας κύκλων με σημαντικότερη πτώση στους Non Sequential cycles.

3η Έκδοση: Υπολογισμός του Kernel size σε pixel 1 φορά

Ο κώδικας της έκδοσης βρίσκεται στο αρχείο : ««**meanFilter_OncePow.c**».

Στη συγκεκριμένη έκδοση αποφασίσαμε να κάνουμε μια μικρή αλγοριθμική αλλαγή για να μειώσουμε τους κύκλους. Ο υπολογισμός των συνολικών pixel στο Kernel είναι πράξη τετραγώνου που κοστίζει σε κύκλους. Στην αρχική υλοποίηση αυτόν τον υπολογισμό τον κάνουμε για κάθε pixel, αλλά το μέγεθος αυτό δεν αλλάζει όσο κυλάει το Kernel πάνω στην εικόνα. Οπότε μπορούμε να το υπολογίσουμε 1 φορά στην αρχή της συνάρτησης αποθηκεύοντας το σε μια μεταβλητή και στη συνέχεια να χρησιμοποιούμε την μεταβλητή αυτή αντί να το υπολογίζουμε ξανά κάθε φορά. Οπότε αντί να κάνουμε μέσα στις εμφωλιασμένες for τον υπολογισμό έτσι:

```
newImg[y][x]=myround((double)newImg[y][x]/pow((double)(2*windowRadius+1),2));
```

Τον κάνουμε 1 φορά στην αρχή της συνάρτησης

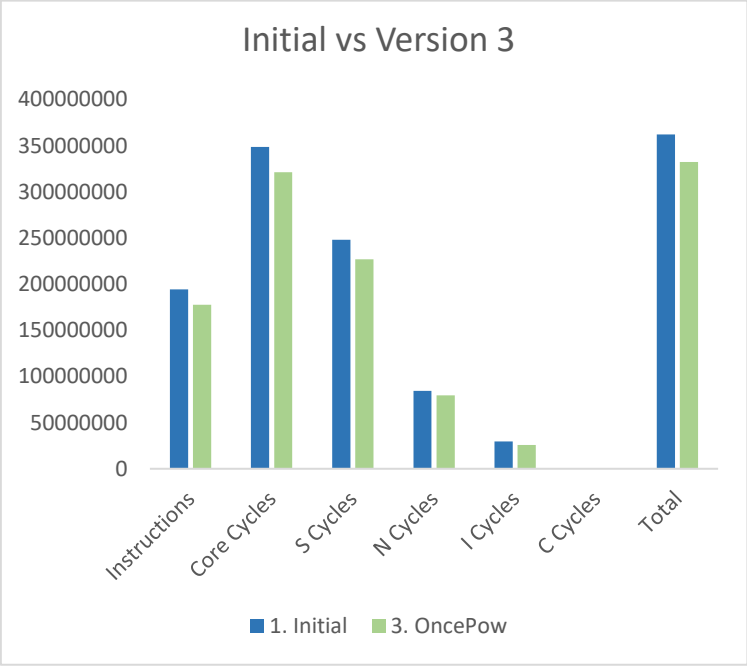
```
double pixels_inKernel;
pixels_inKernel = pow((double)(2*windowRadius+1),2);
```

Και μέσα στις For Loops απλά χρησιμοποιούμε την μεταβλητή:

```
newImg[y][x]=myround((double)newImg[y][x] / pixels_inKernel);
```

Τρέχοντας τον κώδικα έχουμε τα παρακάτω αποτελέσματα:

Version	Instructions	Core Cycles	S Cycles	N Cycles	I Cycles	C Cycles	Total
1. Initial	194471492	348793219	248238865	84426542	29461316	0	362126723
3. OncePow	177773596	321350589	227039801	79489769	25686140	0	332215710



Παρατηρούμε ότι έχουμε μια αρκετά μεγάλη μείωση των συνολικών κύκλων και σε αυτήν την έκδοση. Εδώ είναι της τάξης περίπου των 30 εκατομμύρια κύκλων. Έχουμε και λογική πτώση των instructions καθώς και της κάθε ξεχωριστής κατηγορίας κύκλων με σημαντικότερη πτώση στους Sequential cycles. Είναι επίσης προφανές ότι αν συνδυάσουμε την έκδοση 2 και 3 θα πέσουν ακόμα παραπάνω οι κύκλοι σε σχέση με τη αρχική.

4η Έκδοση: Procedure Inlining

Ο κώδικας της έκδοσης βρίσκεται στο αρχείο : «**meanFilter_ProcedureInlining2.c**».

Η «Γραμμικοποίηση Διαδικασίας» είναι μία μέθοδος που επιτυγχάνει την απάλειψη του επιπρόσθετου κόστους της διασύνδεσης διαδικασίας (procedure linkage overhead).

Η υλοποίηση που κάνουμε μεταφέρει τις συναρτήσεις min, max και myround μέσα στην συνάρτηση meanFilter. Επιπλέον αντικαταστήσαμε την συνάρτηση pow της βιβλιοθήκης math.h της C με απευθείας πολλαπλασιασμό της μεταβλητής με τον εαυτό της. Τέλος, για να δούμε την πλήρη δυνατότητα αυτής της μεθόδου μεταφέραμε τελικά όλες εναπομένουσες συναρτήσεις (read,write, meanFilter) μέσα στην main.

Τη μεγαλύτερη διαφορά την κάνουν οι συναρτήσεις min, max που καλούνται πολλαπλές φορές στην meanFilter και ακολουθούν οι pow και myround.

Ενδεικτικά το τμήμα κώδικα από:

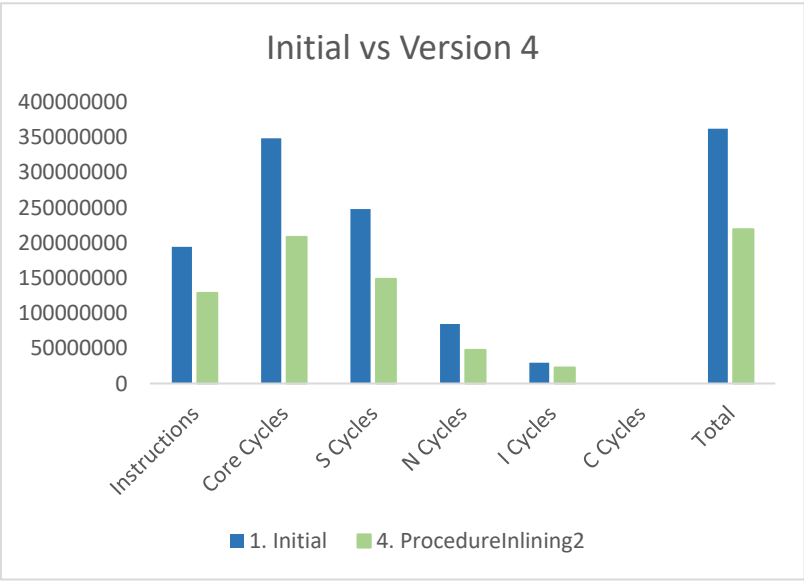
```
newImg[y][x]=myround((double)newImg[y][x] / pow((double)(2*windowRadius+1),2));
```

Θα γίνει:

```
newImg[y][x]=(int)(((double)newImg[y][x]/((double)((windowRadius<<1)+1)*((windowRadius<<1)+1)))+0.5);
```

Το αποτέλεσμα αυτής της δοκιμής ήταν το παρακάτω:

Version	Instructions	Core Cycles	S Cycles	N Cycles	I Cycles	C Cycles	Total
1. Initial	194471492	348793219	248238865	84426542	29461316	0	362126723
4. ProcedureInlining2	128676138	208530039	148766986	47991241	22636933	0	219395160



Παρατηρούμε ότι έχουμε μια αρκετά μεγάλη μείωση των συνολικών κύκλων, της τάξης των περίπου 140 εκατομμύρια κύκλων. Έχουμε πτώση της κάθε ξεχωριστής κατηγορίας κύκλων με τα Sequential και Non Sequential να γίνονται περίπου μισά.

5η Έκδοση: Χρήση Padding στην αρχική εικόνα

Ο κώδικας της έκδοσης βρίσκεται στο αρχείο : «**meanFilter_Padding.c**».

Σε αυτήν την έκδοση ασχοληθήκαμε με τις συνοριακές περιπτώσεις του αλγορίθμου μας. Πιο συγκεκριμένα ο αρχικός αλγόριθμος ελέγχει για κάθε pixel γύρω από το pixel που μας ενδιαφέρει αν είναι εκτός ορίων ώστε να του δώσει την τιμή του κεντρικού pixel του Kernel. Αυτός ο τρόπος κοστίζει πάρα πολύ καθώς για κάθε pixel εντός του kernel κάνουμε 4 if ελέγχους μέσω κλήσης συναρτήσεων min kai max.

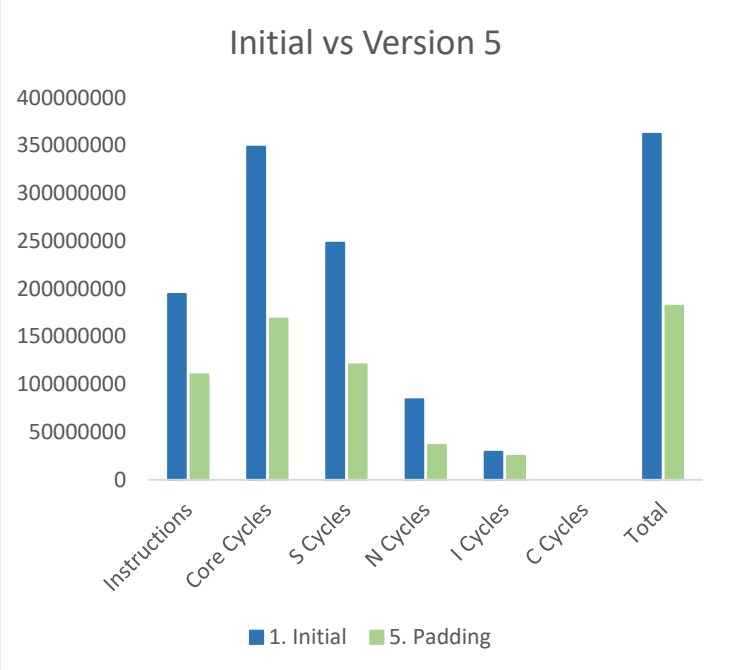
Μπορούμε να μειώσουμε πολύ το κόστος αυτό αν δημιουργήσουμε μια padded εικόνα μεγαλύτερη από την αρχική κατά 2*WindowRadius σε κάθε διάσταση. Έτσι ουσιαστικά δημιουργούμε ένα «πλαίσιο» γύρω από την αρχική εικόνα.

Για να έχουμε το ίδιο αποτέλεσμα με το αρχικό αποτέλεσμα, στις τιμές των padded pixels δίνουμε τις τιμές των ακριανών pixel της αρχικής εικόνας.

Αν λοιπόν τρέξουμε το mean filter εσωτερικά του padded image στις αρχικές διαστάσεις της εικόνας θα έχουμε το ίδιο αποτέλεσμα χωρίς να χρειαστεί να ελέγχουμε για συνοριακές περιπτώσεις.

Τρέχοντας τον κώδικα έχουμε τα παρακάτω αποτελέσματα:

Version	Instructions	Core Cycles	S Cycles	N Cycles	I Cycles	C Cycles	Total
1. Initial	194471492	348793219	248238865	84426542	29461316	0	362126723
5. Padding	110526555	168839072	120778585	36431216	24985187	0	182194988

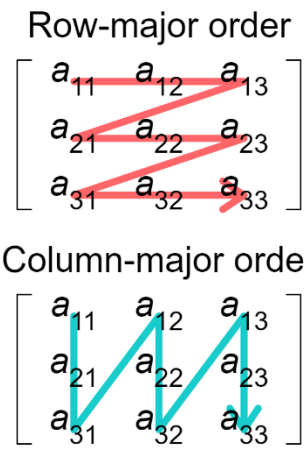


Όπως βλέπουμε σε αυτήν την έκδοση έχουμε τεράστια πτώση τόσο των instructions όσο και των κύκλων. Οι συνολικοί κύκλοι έπεσαν κάτω από τους μισούς σε σχέση με τους αρχικούς. Είχαμε μια πτώση της τάξης των 234 εκατομμυρίων κύκλων. Η τακτική του Padded Image φαίνεται να είναι πάρα πολύ βελτιωτική και μπορεί να συνδυαστεί με τις προηγούμενες βελτιστοποιήσεις και να οδηγήσει σε ακόμα χαμηλότερους κύκλους.

6η Έκδοση: Εφαρμογή Loop Interchange

Ο κώδικας της έκδοσης βρίσκεται στο αρχείο : «**meanFilter_LoopInterchange.c**».

Σε αυτήν την έκδοση κάναμε τον μετασχηματισμό βρόγχων Loop Interchange στον αρχικό αλγόριθμο. Ο μετασχηματισμός αυτός εφαρμόζεται σε εμφωλευμένες for και είναι η εναλλαγή των 2 μεταβλητών που διατρέχουν οι 2 for. Αυτό το κάνουμε με σκοπό να έχουμε προσπέλαση διδιάστατων πινάκων sequentially όπως είναι αποθηκευμένοι στη μνήμη κάτι που είναι πιο γρήγορο από Non sequential προσπελάσεις.



Στην περίπτωση μας, στον αρχικό αλγόριθμο, στην συνάρτηση meanFilter, η εξωτερική for αλλάζει το column και η εσωτερική το row του πίνακα (column-major order). Η C όμως είναι row-major γλώσσα που σημαίνει ότι τα στοιχεία του πίνακα είναι αποθηκευμένα με τον τρόπο που βλέπουμε στην πάνω διπλανή εικόνα. Οπότε άμα αλλάξουμε τις for μέσα στην meanFilter θα έχουμε sequential προσπελάσεις της μνήμης και οι συνολικοί κύκλοι θα πέσουν.

Οπότε ο κώδικας από:

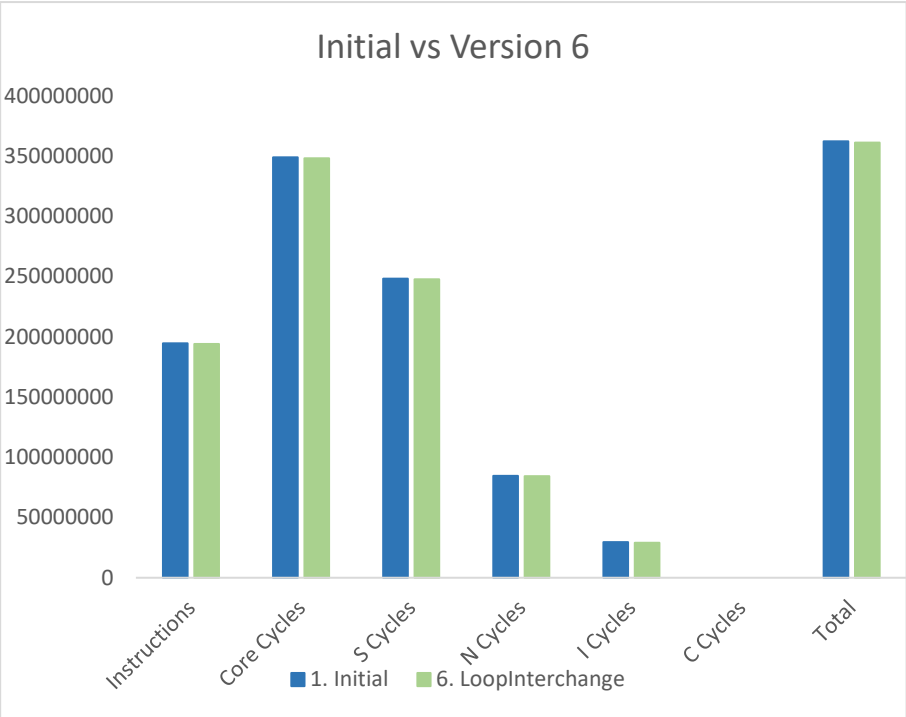
```
      for (x=0; x<imgWidth; x++) {
          for (y=0; y<imgHeight; y++) {
            newImg[y][x]=0;
            for (i=-windowRadius; i<=windowRadius; i++) {
              for (j=-windowRadius; j<=windowRadius; j++) {
```

Θα γίνει:

```
for (y=0; y<imgHeight; y++) {
   for (x=0; x<imgWidth; x++) {
     newImg[y][x]=0;
     for (j=-windowRadius; j<=windowRadius; j++) {
       for (i=-windowRadius; i<=windowRadius; i++) {
```

Τρέχοντας το κώδικα έχουμε:

Version	Instructions	Core Cycles	S Cycles	N Cycles	I Cycles	C Cycles	Total
1. Initial	194471492	348793219	248238865	84426542	29461316	0	362126723
6. LoopInterchange	194036232	348067347	247803387	84281236	29138872	0	361223495



Παρατηρούμε μια πτώση στους συνολικούς κύκλους αν και συγκριτικά με προηγούμενες εκδόσεις αρκετά μικρή, της τάξης του 1 εκατομμυρίου κύκλων. Επίσης παρατηρούμε την λογική πτώση των Non sequential cycles.

7η Έκδοση: Loop Collapse

Ο κώδικας της έκδοσης βρίσκεται στο αρχείο : «**meanFilter_LoopCollapse.c**».

Το Loop Collapsing σχετίζεται με το πώς μπαίνουν τα δεδομένα στη μνήμη. Ο πίνακας μπαίνει στη μνήμη σαν ένα vector. Δηλαδή τα στοιχεία του είναι αποθηκευμένα σειριακά (και κατά γραμμές στη C). Έτσι χρησιμοποιούμε έναν pointer για να διατρέχουμε τα στοιχεία του πίνακα αποφεύγοντας τις δύο for.

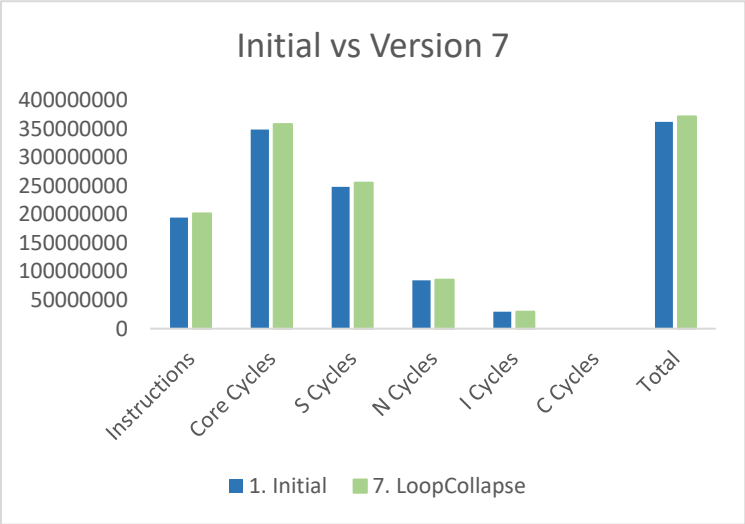
Ενδεικτικά, αριστερά φαίνεται ο αρχικός κώδικας και δεξιά ο βελτιστοποιημένος.

```
//write Y image to file
void write() {
  int i,j;
  FILE *frame_yuv;
  frame_yuv=fopen(file_out,"wb");
  for(i=0;i<N;i++) {
    for(j=0;j<M;j++) {
      fputc(newImg[i][j],frame_yuv);
    }
  }
  fclose(frame_yuv);
}
```

```
//write Y image to file
void write() {
  int i;
  int *p=&newImg[0][0];
  FILE *frame_yuv;
  frame_yuv=fopen(file_out,"wb");
  for(i=0;i<N*M;i++) {
    fputc(*p++,frame_yuv);
  }
  fclose(frame_yuv);
}
```

Τρέχοντας τον κώδικα έχουμε τα παρακάτω αποτελέσματα:

Version	Instructions	Core Cycles	S Cycles	N Cycles	I Cycles	C Cycles	Total
1. Initial	194471492	348793219	248238865	84426542	29461316	0	362126723
7. LoopCollapse	201668348	358229660	255757789	85764139	29862670	0	371384598



Περιμέναμε να δούμε βελτίωση με την χρήση της μεθόδου όμως τα αποτελέσματα δεν ήταν τα αναμενόμενα.

8η Έκδοση: Πρώτη υλοποίηση του Unroll

Ο κώδικας της έκδοσης βρίσκεται στο αρχείο : «**meanFilter_Unroll1.c**».

Η βελτιστοποίηση Loop Unrolling επιτυγχάνει τη μείωση του επιπρόσθετου κόστους του βρόχου (loop overhead) .

Στην έκδοση αυτή κάνουμε unroll σε όλες τις for που είναι εμφωλευμένες εντός κάποιας άλλης με παράγοντα UNROLL= 4. Επίσης, μέσα στην συνάρτηση meanFilter κάναμε πλήρως unroll στις δύο μικρές for που έτρεχαν μόνο κατά γραμμές και στήλες του kernel του φίλτρου. Στο πλήρες unroll θεωρήσαμε ότι γνωρίζουμε εκ των προτέρων το μέγεθος του kernel με το οποίο δουλεύουμε και ότι παραμένει σταθερό.

Ενδεικτικά ένα κομμάτι κώδικα στο οποίο κάναμε unroll είναι το παρακάτω. Αριστερά φαίνεται ο αρχικός κώδικας και δεξιά ο βελτιστοποιημένος

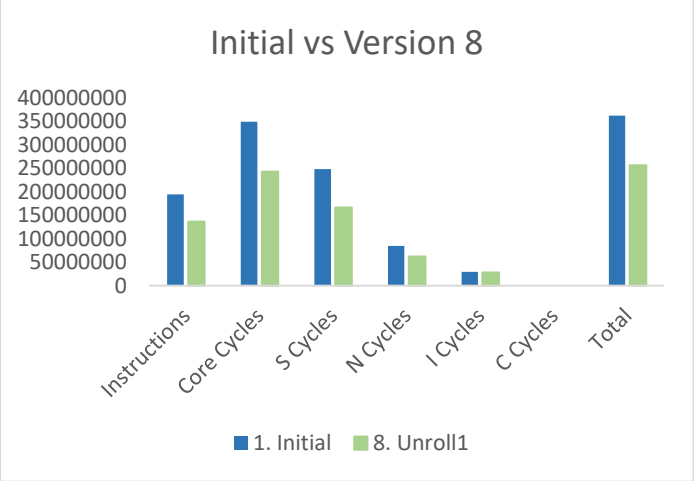
```
//write Y image to file
void write() {
  int i,j;
  FILE *frame_yuv;
  frame_yuv=fopen(file_out,"wb");
  for(i=0;i<N;i++) {
    for(j=0;j<M;j++) {
      fputc(newImg[i][j],frame_yuv);
    }
  }
  fclose(frame_yuv);
}
```

```
//write Y image to file
void write() {
  int i,j;
  int point;
  FILE *frame_yuv;
  frame_yuv=fopen(file_out,"wb");
  for(i=0; i<N; i++) {
    point=M%UNROLL; //UNROLL factor is 4
    for(j=0; j<point; j++) {
      fputc(newImg[i][j],frame_yuv);
    }
    for(j=point; j<M; j+=UNROLL) {
      fputc(newImg[i][j],frame_yuv);
      fputc(newImg[i][j+1],frame_yuv);
      fputc(newImg[i][j+2],frame_yuv);
      fputc(newImg[i][j+3],frame_yuv);
    }
  }
  fclose(frame_yuv);
}
```

Το unroll μπορεί να επιτελεί επαναληπτικές λειτουργίες με συγκεκριμένο βήμα. Για να διατρέξουμε όλα τα δεδομένα μας θα έπρεπε αυτά να είναι ακέραιο πολλαπλάσιο του βήματος το οποίο όμως δεν συμβαίνει πάντα. Έτσι τα δεδομένα που «περισσεύουν» τα διατρέχουμε με βήμα 1 (χωρίς unroll) και υπόλοιπα unrolled.

Τα αποτελέσματα:

Version	Instructions	Core Cycles	S Cycles	N Cycles	I Cycles	C Cycles	Total
1. Initial	194471492	348793219	248238865	84426542	29461316	0	362126723
8. Unroll1	136386882	242980055	166429673	61872369	28011517	0	256313559



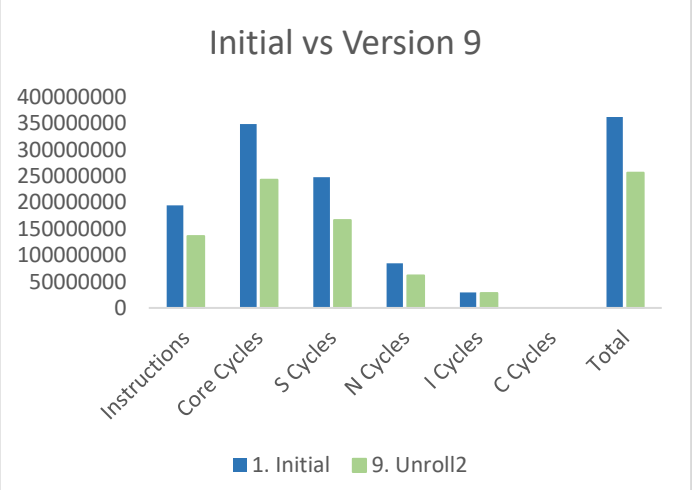
Τα total cycles έπεσαν στα 256εκ., η βελτιστοποίηση επομένως μείωσε τους συνολικούς κύκλους χονδρικά κατά 105εκ. Αντιλαμβανόμαστε λοιπόν ότι το Unroll παίζει μεγάλο ρόλο στην βελτιστοποίηση της απόδοσης του κώδικα. Ωστόσο, είναι εμφανές ότι όσο μεγαλύτερο το unroll τόσο πιο δυσανάγνωστος καταλήγει να είναι ο κώδικας.

9η Έκδοση: Δεύτερη υλοποίηση του Unroll

Ο κώδικας της έκδοσης βρίσκεται στο αρχείο : «**meanFilter_Unroll2.c**».

Σε συνέχεια της προηγούμενης βελτιστοποίησης μπορούμε να κάνουμε partial unroll και στις εξωτερικές for των εμφωλευμένων βρόχων. Χρησιμοποιήσαμε και πάλι παράγοντα UNROLL=4 και τα αποτελέσματα αυτού του περαιτέρω «ξετυλίγματος» φαίνονται παρακάτω:

Version	Instructions	Core Cycles	S Cycles	N Cycles	I Cycles	C Cycles	Total
1. Initial	194471492	348793219	248238865	84426542	29461316	0	362126723
8. Unroll1	136386882	242980055	166429673	61872369	28011517	0	256313559
9. Unroll2	136384902	242977415	166427363	61872039	28011113	0	256310515



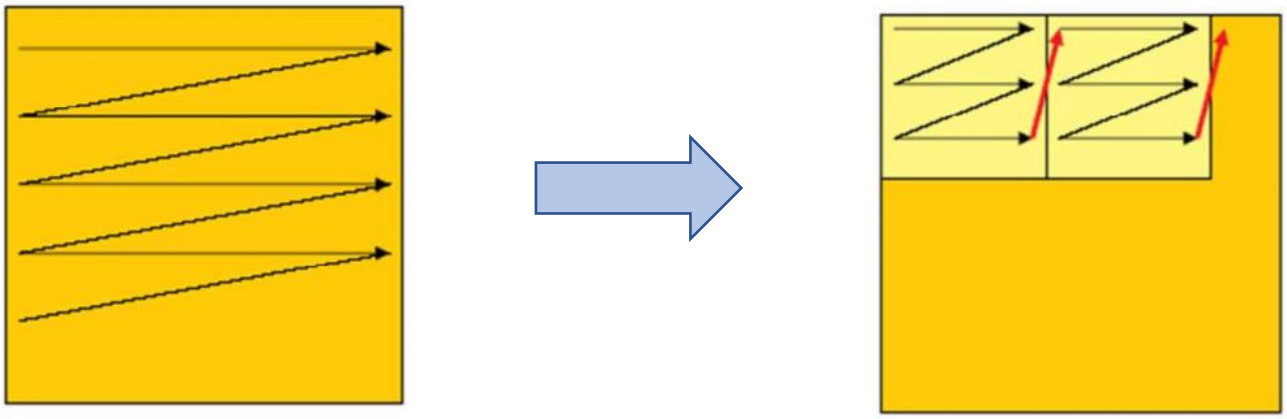
Εδώ η διαφορά με την προηγούμενη έκδοση Unroll είναι της τάξης των 3 χιλιάδων κύκλων που είναι μία πολύ μικρή βελτιστοποίηση σε σχέση με τον συνολικό αριθμό κύκλων.

10η Έκδοση: Loop Tiling

Ο κώδικας της έκδοσης βρίσκεται στο αρχείο : «**meanFilter_LoopTiling.c**».

Το Loop Tiling διατρέχει τα δεδομένα εντός ενός επαναληπτικού βρόγχου κατά μπλοκ δεδομένων. Η βελτιστοποίηση που προκαλεί σχετίζεται περισσότερο με το πώς φέρνει τα δεδομένα ο επεξεργαστής στη μνήμη cache. Με τη σωστή χρήση του loop tiling μπορούμε να μειώσουμε το latency της κλήσης των δεδομένων από τη μνήμη ορίζοντας τα χρήσιμα δεδομένα που μένουν αποθηκευμένα στην cache μέχρι την επαναχρησιμοποίηση αυτού του χώρου.

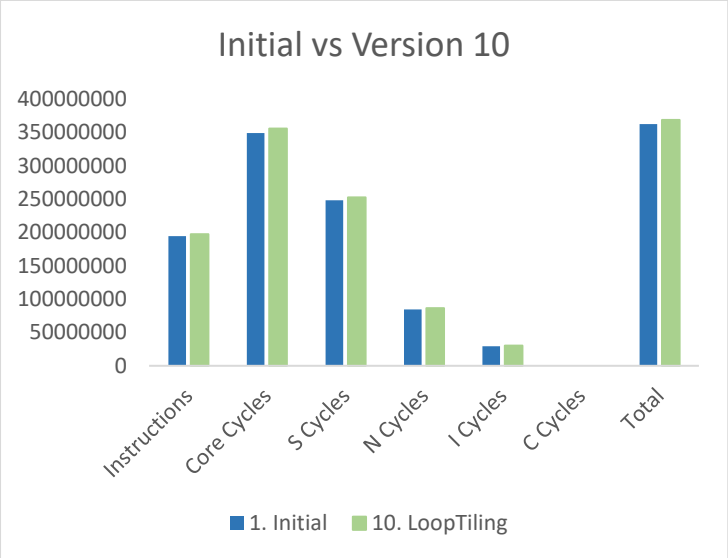
Οπτικοποίηση της μεθόδου σε έναν δισδιάστατο πίνακα:



Καθώς δεν χρησιμοποιούμε κάποια δομή μνήμης cache για το πρόγραμμά μας, δεν περιμένουμε βελτίωση στην απόδοση του κώδικα. Περιμένουμε μάλιστα αύξηση των total cycles καθώς αυξάνουμε τον αριθμό των for loops στον κώδικα και επιπλέον είναι λογική μία αύξηση των non sequential cycles λόγω της αλλαγής σειράς πρόσβασης σε έναν πίνακα (πιο συχνά σε μη διαδοχικές θέσεις μνήμης).

Το αποτέλεσμα αυτής της δοκιμής ήταν το παρακάτω:

Version	Instructions	Core Cycles	S Cycles	N Cycles	I Cycles	C Cycles	Total
1. Initial	194471492	348793219	248238865	84426542	29461316	0	362126723
10. LoopTiling	197267969	355361929	252157726	86422550	30115157	0	368695433



Όπως περιμέναμε, έχουμε αύξηση των συνολικών κύκλων κατά περίπου 6 εκατομμύρια κύκλους.

11η Έκδοση: Loop Inversion

Ο κώδικας της έκδοσης βρίσκεται στο αρχείο : «**meanFilter_LoopInversion.c**».

Το Loop Inversion είναι μία βελτιστοποίηση στην οποία ένας βρόγχος while αντικαθίσταται από ένα if block που εσωκλείει ένα do…while βρόχο. Παρόλο που φαίνεται η μέθοδος να μειώνει την απόδοση με την προσθήκη του ελέγχου if η μέθοδος αυτή μπορεί να είναι αποδοτική καθώς προκαλεί 2 λιγότερα instruction jumps.

Συγκριτικά τα δύο workflows:

```
while (condition) {
  ...
}
```

Workflow:

1. check condition;
2. if false, jump to outside of loop;
3. run one iteration;
4. jump to top.

```
if (condition) do {
  ...
} while (condition);
```

Workflow:

1. check condition;
2. if false, jump to beyond the loop;
3. run one iteration;
4. check condition;
5. if true, jump to step 3.

Καθώς δεν είχαμε while loops στον κώδικά μας δοκιμάσαμε να αλλάξουμε τα for loops σε while loops και μετά να τα μεταποιήσουμε με Loop Inversion. Θεωρήσαμε ότι τα while loops δεν θα μείωναν την απόδοση και το tradeoff της προσθήκης ελέγχου αλλά με μείωση των instruction jumps περιμέναμε να προκαλέσει συνολικά μείωση των κύκλων εκτέλεσης.

Τα συγκριτικά αποτελέσματα φαίνονται εδώ:

Version	Instructions	Core Cycles	S Cycles	N Cycles	I Cycles	C Cycles	Total
1. Initial	194471492	348793219	248238865	84426542	29461316	0	362126723
11.LoopInversion	194472816	348794547	248240190	84426544	29461317	0	362128051

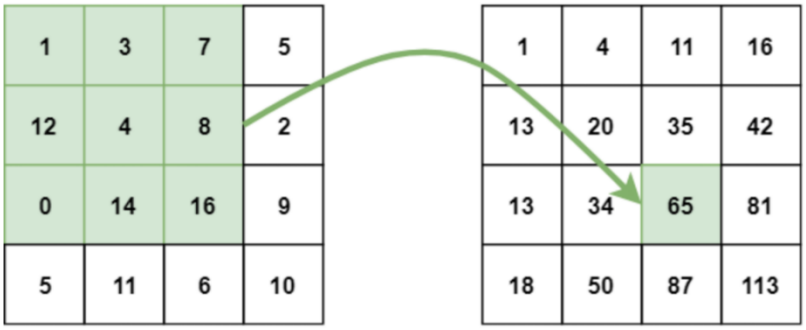
Τελικά δεν είχαμε βελτίωση με την χρήση αυτής της μεθόδου.

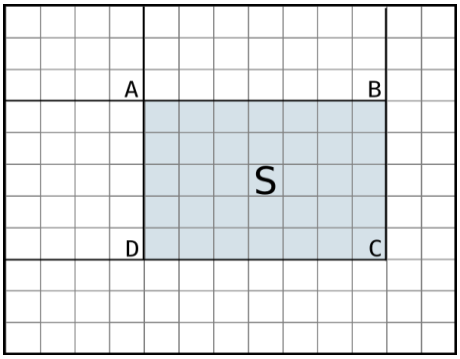
12η Έκδοση: Χρήση Integral Image

Ο κώδικας της έκδοσης βρίσκεται στο αρχείο : «**meanFilter_IntegralImage.c**».

Στην συγκεκριμένη έκδοση κάνουμε χρήση του summed-area table ή αλλιώς Integral Image για τον γρήγορο υπολογισμό του αθροίσματος των τιμών των pixel του Kernel.

Ο συγκεκριμένος πίνακας είναι ένας πίνακας όπου η τιμή του κάθε pixel είναι το άθροισμα των τιμών όλων των pixel αριστερά και από πάνω του.





Έτσι όταν θέλουμε να πάρουμε το άθροισμα των τιμών των pixel σε μια περιοχή της εικόνας αρκεί να κάνουμε την πράξη με SUM= C+A-D-B στην Integral Image που δημιουργήσαμε.

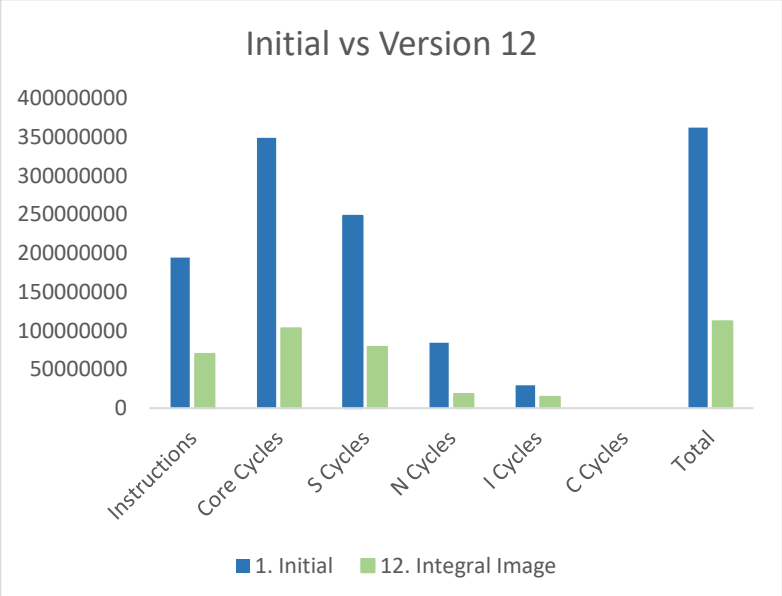
Φυσικά για να μην έχουμε το ίδιο πρόβλημα με τους ελέγχους των οριακών συνθηκών πρέπει να κάνουμε την integral image με βάση την padded εικόνα.

Επιπλέον όπως βλέπουμε για να πάρουμε το άθροισμα μιας περιοχής πρέπει να χρησιμοποιήσουμε pixel αριστερά και πάνω από την εικόνα, οπότε θα κάνουμε και padding την integral image κατά 1 γραμμή και 1 στήλη.

Μετά από το pre processing για την δημιουργία της εικόνας ο υπολογισμός του μέσου όρου σε κάθε pixel είναι πολύ πιο γρήγορος καθώς πρέπει κάθε φορά να πάρουμε μόνο 4 τιμές από την μνήμη και δεν έχουμε πλέον εσωτερικές for.

Τρέχοντας τον κώδικα έχουμε τα παρακάτω αποτελέσματα:

Version	Instructions	Core Cycles	S Cycles	N Cycles	I Cycles	C Cycles	Total
1. Initial	194471492	348793219	248238865	84426542	29461316	0	362126723
12. Integral Image	70088896	102892233	78890271	18459378	14792306	0	112141955



Παρατηρούμε τεράστια πτώση τόσο σε cycles όσο και σε instructions, μάλιστα μεγαλύτερη από τη χρήση απλά padded image. Οι κύκλοι πέφτουν συνολικά κατά περίπου 250 εκατομμύρια. Είναι το καλύτερο αποτέλεσμα που έχουμε σε μη συνδυαστική έκδοση

Συνδυαστικές Εκδόσεις: 13η 14η και 15η

Έχοντας δοκιμάσει ξεχωριστά όλες τις πιθανές βελτιστοποιήσεις , θα δουλέψουμε πλέον αθροιστικά με σκοπό να καταλήξουμε στην καλύτερη δυνατή έκδοση.

Με βάση τις προηγούμενες βελτιστοποιήσεις, μπορούμε να κάνουμε διαφορετικούς συνδυασμούς με σκοπό να ελαχιστοποιήσουμε τους κύκλους. Η βασικότερη ειδοποιός διαφορά είναι η χρήση σκέτου Padded Image , ή χρήση και Integral Image ή χρήση μόνο του απλού πίνακα της εικόνας.

Οπότε η 3 εκδόσεις μας είναι οι εξής

13.Combined_V1:

- Χρήση Integral Image
- Χρήση Padded Image
- Χρήση Τοπικής μεταβλητής Sum
- Υπολογισμός του Kernel size σε pixel 1 φορά
- Procedure Inlining
- Loop Interchange
- Loop Unroll (factor 4)
- Loop Fusion (Εδώ είναι η πρώτη φορά που μας δόθηκε η ευκαιρία να προσθέσουμε Loop Fusion λόγω του κώδικα στο pre processing padding)

14.Combined_V2:

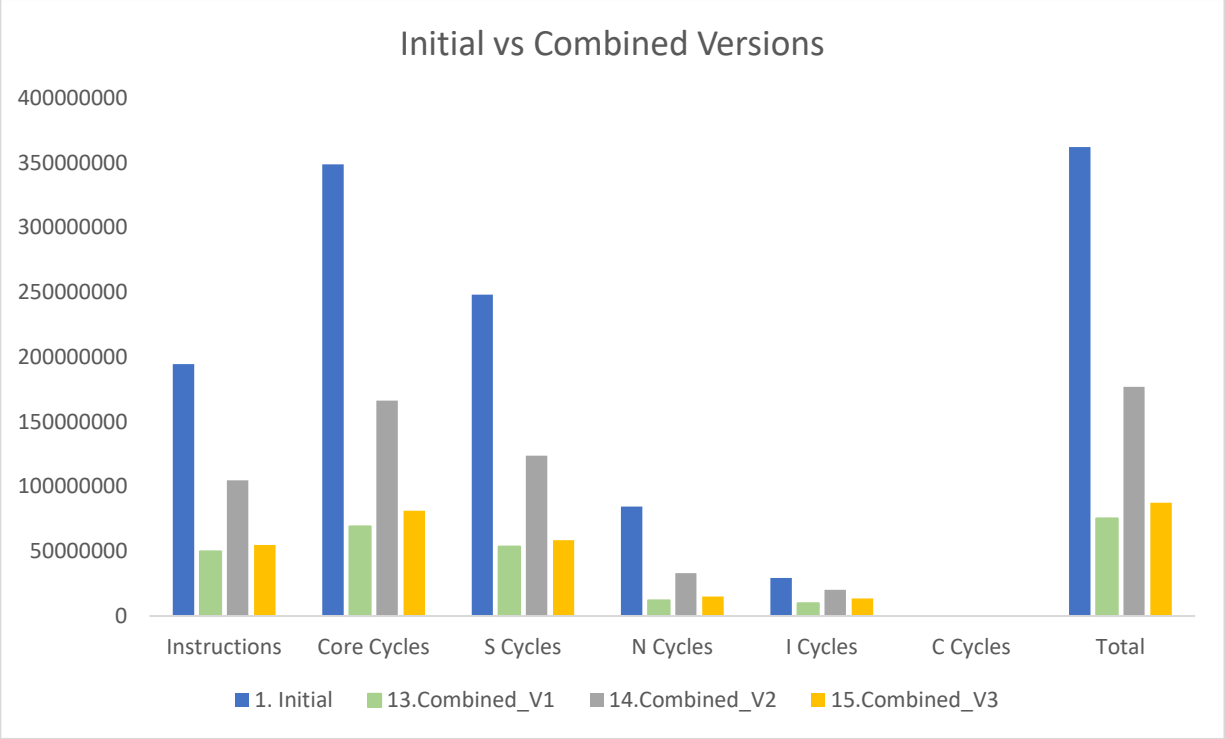
- Χρήση Τοπικής μεταβλητής Sum
- Υπολογισμός του Kernel size σε pixel 1 φορά
- Procedure Inlining
- Loop Interchange
- Loop Unroll (factor 4)

14.Combined_V3:

- Χρήση Padded Image
- Χρήση Τοπικής μεταβλητής Sum
- Υπολογισμός του Kernel size σε pixel 1 φορά
- Procedure Inlining
- Loop Interchange
- Loop Unroll (factor 4)
- Loop Fusion (Εδώ είναι η πρώτη φορά που μας δόθηκε η ευκαιρία να προσθέσουμε Loop Fusion λόγω του κώδικα στο pre processing του padding)

Τρέχοντας τις 3 αυτές συνδυαστικές εκδόσεις:

Version	Instructions	Core Cycles	S Cycles	N Cycles	I Cycles	C Cycles	Total
1. Initial	194471492	348793219	248238865	84426542	29461316	0	362126723
13.Combined_V1	49909220	69215378	53593744	12073789	9794482	0	75462015
14.Combined_V2	104866987	166285942	123725764	33111451	20136492	0	176973707
15.Combined_V3	54818000	81193817	58608427	15165377	13666218	0	87440022



Παρατηρούμε αρχικά ότι και οι 3 συνδυαστικές εκδόσεις έχουν πολύ καλό αποτέλεσμα σε σχέση με το αρχικό. Μάλιστα ακόμα και να μην χρησιμοποιήσουμε padding ή Integral Image καταφέραμε να ρίξουμε τους κύκλους στους μισούς. Εκεί που η διαφορά γίνεται πλέον πολύ μεγάλη είναι στις περιπτώσεις που χρησιμοποιούμε padding ή integral image όπου οι κύκλοι συγκριτικά με τους αρχικούς έχουν γίνει ελάχιστοι. Η καλύτερη μας έκδοση είναι η Combined_V1 όπου έχουμε πτώση περίπου 287 εκατομμυρίων κύκλων

Προσπέλαση και Αποθήκευση Πινάκων

Οι πίνακες δεδομένων που χρησιμοποιούνται στον αρχικό αλγόριθμο είναι οι εξής:

```
int image_Y[N][M]; // Original image only Y(luminance)
int newImg[N][M]; //final image after filter, only Y(luminance)
```

Όπου έχουμε για κάθε πίνακα τις εξής προσπελάσεις μνήμης:

image_Y Iterations:
1 x read N*M
1 x iteration (loop) ((2*RADIUS+1)^2)*(N*M)

newImg Iterations:
2 x iteration N*M
1 x iteration ((2*RADIUS+1)^2)*(N*M)
1 x write N*M
Και στον βελτιστοποιημένο αντίστοιχα:

```
int image_Y[N+2*RADIUS][M+2*RADIUS]; // Original image only Y(luminance)
int newImg[N][M];  //final image after filter, only Y(luminance)
int integralImg[N+(2*RADIUS)+1][M+(2*RADIUS)+1];
```

Όπου οι προσπελάσεις μνήμης είναι :

image_Y Iterations:
1 x read N*M
4 x iteration RADIUS*RADIUS
2 x iteration RADIUS*M
2 x iteration N*RADIUS
1 x iteration Mp (pre-processing)
1 x iteration (Np-1)*Mp (pre-processing)

newImg Iterations:
1 x iteration M*N (meanFilter)
1 x write N*M

integralImg Iterations:
1 x Np+1
1 x Mp
1 x Mp
2 x (Np-1)*Mp
2 x Np*(Mp-1)
4 x N*M

Όπου Np=N+2*RADIUS
Και Mp=M+2*RADIUS

Παρακάτω φαίνονται συγκριτικά οι προσπελάσεις που γίνονται στους πίνακες αυτούς θεωρώντας RADIUS=2 κατά την εκτέλεση του προγράμματος στην αρχική και στην βελτιστοποιημένη εκδοχή του κώδικα.

	INITIAL VERSION	**BEST VERSION**
Image_Y	26NM	2NM + 6(N+M) + 20
newImg	28NM	2NM
integralImg	-	8NM + 15(N+M) + M + 61
TOTAL	54*N*M	12*N*M +21(N+M) + M + 81

Η διαφορά είναι: 42*N*M-21(N+M)-M-81

Παρατηρούμε, λοιπόν, ότι στην τελική εκδοχή του κώδικα απαιτούνται πολύ λιγότερες προσπελάσεις στους πίνακες δεδομένων.

Οι πίνακες είναι τύπου δεδομένων int επομένως η απαιτούμενη μνήμη για την αποθήκευση των πινάκων:
Memory_Initial = [(N*M)+(N*M)]*4byte= 2*N*M*4byte

Memory_Best = [(N*M)+(N+2*RADIUS)(M+2RADIUS)+(N+2*RADIUS+1)(M+2*RADIUS+1)]*4byte=

= [3*N*M +9(N+M) + 41]*4byte

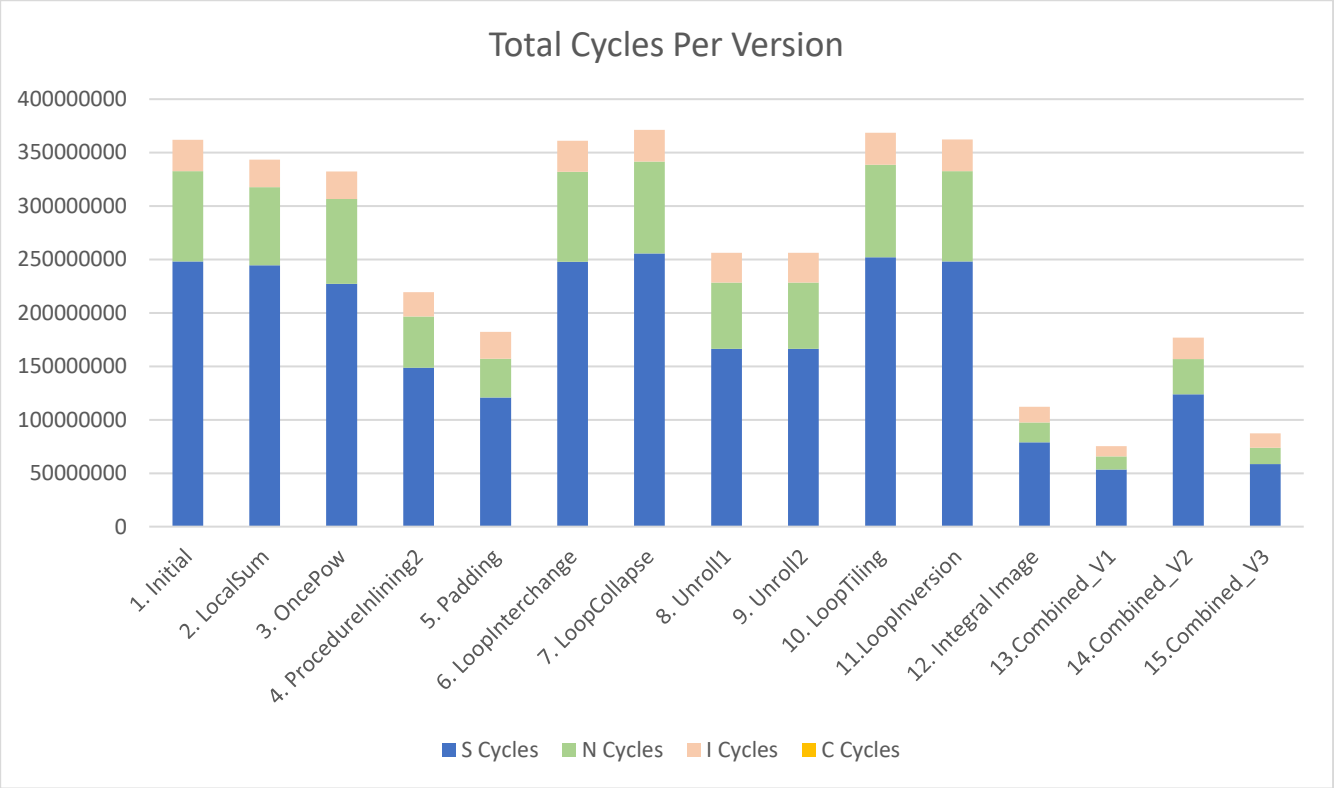
Διαφορά=[(N*M)+9(N+M)+41]*4byte

Παρατηρούμε, λοιπόν, ότι στην τελική εκδοχή του κώδικα η απαιτούμενη μνήμη για τους πίνακες δεδομένων είναι περισσότερη.
Μπορούμε να δούμε και τις γενικότερες απαιτήσεις μνήμης ανά έκδοση στον παρακάτω πίνακα

Version	Description	Code	RO Data	RW Data	ZI Data	Debug	Total RO	Total RW	Total ROM
1. Initial	Object Totals	764	60	0	1161600	6044	21938 (21.42kB)	1161900 (1134.67kB)	21938 (21.42kB)
	Library Totals	20656	458	0	300	7292			
	Grand Totals	21420	518	0	1161900	13336			
2. LocalSum	Object Totals	756	60	0	1161600	6080	21930 (21.42kB)	1161900 (1134.67kB)	21930 (21.42kB)
	Library Totals	20656	458	0	300	7292			
	Grand Totals	21412	518	0	1161900	13372			
3. OncePow	Object Totals	748	60	0	1161600	6064	21922 (21.41kB)	1161900 (1134.67kB)	21922 (21.41kB)
	Library Totals	20656	458	0	300	7292			
	Grand Totals	21404	518	0	1161900	13356			
4. ProcedureInlining2	Object Totals	696	60	0	1161600	5232	16274 (15.89kB)	1161900 (1134.67kB)	16274 (15.89kB)
	Library Totals	15204	314	0	300	6220			
	Grand Totals	15900	374	0	1161900	10992			
5. Padding	Object Totals	924	60	0	1173984	5708	22098 (21.58kB)	1161900 (1134.67kB)	22098 (21.58kB)
	Library Totals	20656	458	0	300	7292			
	Grand Totals	21580	518	0	1174284	13000			
6. LoopInterchange	Object Totals	776	60	0	1161600	6084	21950 (21.44kB)	1161900 (1134.67kB)	21950 (21.44kB)
	Library Totals	20656	458	0	300	7292			
	Grand Totals	21432	518	0	1161900	13376			
7. LoopFusion	Object Totals	696	60	0	1161600	6032	22094 (21.58kB)	1161900 (1134.67kB)	22094 (21.58kB)
	Library Totals	20880	458	0	300	7420			
	Grand Totals	21576	518	0	1161900	13452			
8.Unroll1	Object Totals	2168	60	0	1161600	9624	23342 (22.79kB)	1161900 (1134.67kB)	23342 (22.79kB)
	Library Totals	20656	458	0	300	7292			
	Grand Totals	22824	518	0	1161900	16916			
9.Unroll2	Object Totals	2384	60	0	1161600	10296	23558 (23.01kB)	1161900 (1134.67kB)	23558(23.01kB)
	Library Totals	20656	458	0	300	7292			
	Grand Totals	23040	518	0	1161900	17588			
10. LoopTiling	Object Totals	880	60	0	1161600	6284	22054 (21.54kB)	1161900 (1134.67kB)	22054 (21.54kB)
	Library Totals	20656	458	0	300	7292			
	Grand Totals	21536	518	0	1161900	13576			
11.LoopInversion	Object Totals	792	60	0	1161600	6204	21966(21,45kB)	1161900 (1134.67kB)	21966(21,45kB)
	Library Totals	20656	458	0	300	7292			
	Grand Totals	21448	518	0	1161900	13496			
12.IntegralImage	Object Totals	1244	60	0	1173984	6332	22418 (21.89kB)	1174284 (1146.76kB)	22418 (21.89kB)
	Library Totals	20656	458	0	300	7292			
	Grand Totals	21900	518	0	1174284	13624			
13.Combined_V1	Object Totals	1956	60	0	1173984	6928	17534(17.12kB)	1174284 (1146.76kB)	17534 (17.12kB)
	Library Totals	15204	314	0	300	6220			
	Grand Totals	17160	374	0	1174284	13148			
13.Combined_V2	Object Totals	1212	60	0	1161600	6384	16790 (16.40kB)	1161900 (1134.67kB)	16790 (16.40kB)
	Library Totals	15204	314	0	300	6220			
	Grand Totals	16416	374	0	1161900	12604			
14.Combined_V3	Object Totals	1476	60	0	1173984	6353	170654 (16.65kB)	1174284 (1146.76kB)	17054 (16.65kB)
	Library Totals	15204	314	0	300	6220			
	Grand Totals	16680	374	0	1174284	12572			

Συνολικός Σχολιασμός/Συμπεράσματα

Γενικότερα είδαμε πώς κάθε έκδοση επηρεάζει τους συνολικούς κύκλους που χρειάζεται ο επεξεργαστής ξεχωριστά και συνδυαστικά.



Όπως αναφέραμε και προηγουμένως τις μεγαλύτερες ξεχωριστές βελτιώσεις τις είδαμε κάνοντας χρήση Padded Image ή Integral Image .

Ο πιο καλός μετασχηματισμός βρόγχου φάνηκε να είναι το Loop Unroll

Επίσης πολύ καλή βελτίωση είχε το procedure inlining καθώς στον αρχικό κώδικα καλούμε συναρτήσεις min,max,pow,round πάρα πολλές φορές.

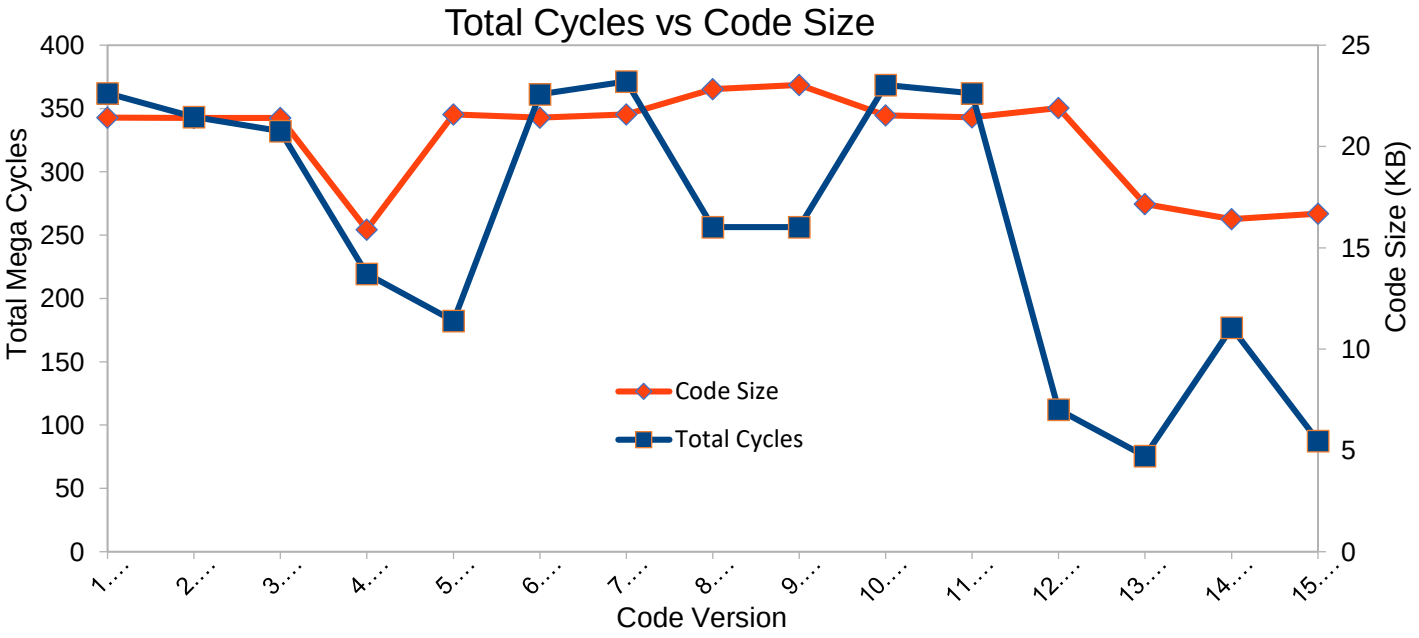
Οι συνδυαστικές εκδόσεις είχαν πάρα πολύ καλά αποτελέσματα με καλύτερη την V1, όπου ρίχνει τους κύκλους κάτω του ενός τετάρτου των αρχικών και την V3 να είναι αρκετά κοντά της.

Η καλύτερη συνολικά έκδοση ήταν η **13.Combined_V1**

Γενικότερα φάνηκε οι συνθήκες if να έχουν το μεγαλύτερο κόστος σε κύκλους, καθώς και οι προσπελάσεις στην μνήμη.

Φυσικά, παρατηρήσαμε ότι πολλοί από αυτούς τους μετασχηματισμούς μας και κυρίως τον βρόγχων κάνανε τον κώδικα μεγαλύτερο σε μέγεθος:

Κάποιες άλλες όμως αλγοριθμικές αλλαγές τον μικραίνουν ,καθώς μπορούμε να αποφύγουμε να έχουμε την βιβλιοθήκη math.h ή να μην χρειαζόμαστε συναρτήσεις min,max,myround γενικότερα.



Είδαμε, επίσης, ότι στη πιο βέλτιστη μας έκδοση είχαμε πολύ μεγαλύτερες ανάγκες μνήμης για την αποθήκευση δεδομένων.

Τέλος πολλές από αυτές τις αλλαγές κάνανε τον κώδικα πιο δύσκολο στην ανάγνωση και στην εύρεση λαθών όταν γινόντουσαν, γεγονός που μπορεί να κάνει κώδικα γραμμένο με τέτοιο τρόπο, δύσκολο στη συντήρηση και στην αναβάθμιση.

Κλείνοντας δείχνουμε τα αποτελέσματα της βέλτιστης υλοποίησης για Radius=2 στις εικόνας που μας δόθηκαν