Δημιουργία Edges και Outline Effects στην επεξεργασία Εικόνας σε ασπρόμαυρη και σε έγρωμη

Τζίτζος Παύλος
Κουμπάνης Αθανάσιος

25/12/2023

Καθηγητής: Γ. Συρακούλης

Μπορείτε να δείτε το υλικό της αναφοράς εδώ .

Λίστα Αντικειμένων

Λίστα Πινάκων

Λίστα Εικόνων

Θεωρητικό Υπόβαθρο

Μετατροπή YUV σε RGB

Για να μετατρέψουμε την εικόνα σε γκρι πρέπει να μετατρεψουμε το χρωματικό χώρο YUV σε RGB. Τα YUV μοντελοποιούν την εικονα χρησιμοποιώντας 3 διαφορετικα στοιχεία :

Y (luminance) για φωτεινότητα, U για το μπλε χρώμα και V για το κόκκινο.

Σε αντίθεση με το YUV μοντέλο , το RGB μοντέλο χρησιμοποιεί 3 στοιχεία για τα χρώματα:

R (red) για το κόκκινο , G (green) για το πράσινο και B (blue) για το μπλέ.

Για τη μετατροπή από YUV σε RGB χρησιμοποιούμε τις εξισώσεις :

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 1 & 0 & 1.13983 \\ 1 & -0.39465 & -0.58060 \\ 1 & 2.03211 & 0 \end{bmatrix} \cdot \begin{bmatrix} Y \\ U \\ V \end{bmatrix}$$

Στο σημείο αυτό πρέπει να έχουμε υπόψιν μας ότι τα δύο αυτά μοντέλα έχουν διαφορετικό εύρος στις τιμές των στοιχείων τους. Το YUV παίρνει τις εξής τιμές :

$0 \leq Y \leq 1$ ή $0 \leq Y \leq 255$

$$-0.5 \leq U \leq 0.5$$

$$-0.5 \leq V \leq 0.5$$

Ενώ το RGB κυμαίνεται μεταξύ των : $0 \leq R$, G , $B \leq 255$.

Οπότε μετά από κάθε μετατροπή είναι αναγκαίο να κάνουμε κανονικοποίηση.

Μετατροπή Έγχρωμης RGB εικόνας σε Ασπρόμαυρη

Επειδή θέλουμε να εντοπίσουμε τις ακμές στην εικόνα μας πριν εφαρμόσουμε το γκαουσιανό φίλτρο μετατρέπουμε την εικόνα από έγχρωμη RGB σε ασπρόμαυρη. Υπάρχουν 3 τρόποι με τους οποίους μπορούμε να μετατρέψουμε μια εικόνα σε ασπρόμαυρη.

Μεθοδος Φωτός

$$I_{grayscale} = \frac{min(R,G,B) - max(R,G,B)}{2}$$

Μέθοδος Αριθμητικού Μέσου

$$I_{grayscale} = \frac{R+G+B}{3}$$

Μέθοδος Φωτεινότητας

$$I_{grayscale} = 0.3R + 0.59G + 0.11B$$

Η τελευταία είναι η πιο αποτελεσματική επειδή το μάτι αντιδρά διαφορετικά σε κάθε χρώμα οπότε βάζοντας βάρη σε κάθε τιμή RGB πετυχαίνουμε να προσαρμόσουμε την γκρι εικόνα στο πως την αντιλαμβανόμαστε .

Το εύρος τιμών των ασπρόμαυρων εικόνων κυμαίνεται από 0 έως 255.

Εφαρμογή του Γκαουσιανού Φίλτρου στην Ασπρόμαυρη εικόνα

Επειδή ο σκοπός της εργασίας είναι να αναγνωρίσουμε τις ακμές στην εικόνα και να τις χρωματίσουμε σε επόμενο βήμα, πρέπει να εφαρμόσουμε το γκαουσιανό φίλτρο στην εικόνα. Αυτό γίνεται με συνέλιξη. Επειδή όμως εφαρμόζουμε την συνέλιξη με μάσκα 3x3 στην πρώτη γραμμή θα χρειαστούμε επιπλέον μία γραμμή. Αλλά και μία στήλη στα αριστερά της εικόνας. Στα δεξία της εικόνας αλλά και στο κάτω μέρος της εικόνας θα χρειαστούμε επίσης μια επιπλέον στήλη και γραμμή αντίστοιχα. Για αυτό τον λόγο δημιουργούμε μια νέα εικόνα I_{padded} .

Η επαυξημένη εικόνα είναι :

$$I_{y,padded} = \begin{bmatrix} 0 & \cdots & 0 \\ \vdots & I_{y,grayscale} & \vdots \\ 0 & \cdots & 0 \end{bmatrix}$$

Η συνέλιξη του φίλτρου με την εικόνα είναι :

$$I_{grayscale,filtered} = (G_{kernel} \bigstar I_{grayscale})(x,y) = \sum_{i=0}^{N-1} \sum_{j=0}^{M-1} G_{kernel}(x-i, y-j) I_{grayscale}(i,j)$$

Υπολογισμός της Βαθμίδας της εικόνας

Εφόσον έχουμε εφαρμόσει το γκαουσιανό φίλτρο υπολογίζουμε την βαθμίδα της φιλτραρισμένης εικόνας που αντιπροσωπεύει τον ρυθμό αλλαγής της έντασης της φωτινότητας του κάθε pixel. Η βαθμίδα της εικόνας I ορίζεται ως:

$\nabla I = \begin{bmatrix} I_x & I_y \end{bmatrix}'$, όπου $I_x = \dfrac{\partial I}{\partial x}$ και $I_y = \dfrac{\partial I}{\partial y}$.

Για να υπολογίσουμε τις παραγώγους χρησιμποιήσαμε φίλτρα sobel και τα εφαρμόσαμε στην φιλτραρισμένη εικόνα με συνέλιξη. Οι μάσκες των φίλτρων sobel είναι :

$$S_{kernel,x} = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$

$$S_{kernel,y} = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

Υπολογισμός της Έντασης και της Γωνίας των pixel της εικόνας

Έχοντας την βαθμίδα υπολογισμένη μπορούμε να υπολογίσουμε της ένταση και την γωνία του κάθε pixel. Η ένταση αντιπροσωπεύει την ένταση του χρώματος γκρι / λευκό και είναι το μέτρο της βαθμίδας ενώ η γωνία δείχνει την κλίση της κατεύθυνσης της ένατσης πάνω σε ένα x,y επίπεδο , το επίπεδο της εικόνας. Οι τύποι για τον υπολογισμό τους είναι αντίστοιχα:

$$|\nabla I| = \sqrt{{I_x}^2 + {I_y}^2}$$

$$\theta = \arctan\left(\frac{I_x}{I_y}\right)$$

Η ένταση έχει εύρος τιμών μεταξύ 0 και 255.

Η γωνία έχει εύρος τιμών μεταξύ 0 έως 180 μοίρες.

Χρωματισμός των pixel βάση της Έντασης και της Γωνίας

Επειδή σε μια εικόνα μια ακμή μπορεί να δείχνει προς οποιαδήποτε κατεύθυνση χρησιμοποιούμε διαφορετικά χρώματα ανάλογα την γωνία που έχουμε υπολογίσει για εκείνο το pixel και το χρωματίζουμε . Έτσι όλα τα pixel που ανήκουν επάνω στην ακμή θα έχουν ίδιο χρώμα και η ένταση του χρώματος αυτού εξαρτάται από την ένταση (μέτρο) της κλίσης που υπολογίσαμε προηγουμένως. Ορίζουμε τις εξής περιοχές:

$\leq \theta°$	$\theta° <$	Στρογγυλοποίηση ($\theta°$)	Χρώμα ομάδας	Ακμές
0	22.5	0	κίτρινο	Κατακόρυφες
22.5	67.5	45	πράσινο	Διαγώνιες προς δεξιά
67.5	112.5	90	μπλέ	Οριζόντιες
112.5	157.5	135	κόκκινο	Διαγώνιες προς αριστερά
157.5	180	0	κίτρινο	Κατακόρυφες

Πίνακας 1 - Χρώματα των pixel ανάλογα την γωνία

Κώδικας C

Ο κώδικας υλοποιήθηκε αρχικά χρησιμοποιώντας το Visual Studio για να μπορέσουμε να κάνουμε debugging εύκολα και το github για να μπορέσουμε να διατηρήσουμε διαφορετικές εκδόσεις λόγω βελτιστοποιήσεων αλλά και για να μπορούμε να δουλέψουμε από διαφορετικές συσκευές στην ίδια εργασία. Η εικόνα με την οποία δοκιμάσαμε τον κώδικα μας είχε διαστάσεις 496x376 και ήταν της μορφής YUV 420.

read_image

1. Το πρώτο βήμα στην εκτέλεση του προγράμματος είναι να φορτώσουμε την εικόνα από την μνήμη. Η συνάρτηση αυτή κάνει αυτή την δουλειά , με παράμετρο filename που έχουμε δηλώσει στην αρχή του αρχείου.

write_id_image

2. Προκειμένου να ελέγξουμε μετά από κάθε αλγόριθμο το αποτέλεσμα γράψαμε συναρτήσεις για αυτόν τον σκοπό. Τα ονόματα αυτών των συναρτήσεων έχουν την παρακάτω δομή:

   void write_*id*_image() ,

   όπου *id* είναι το αναγνωριστικό της μεθόδου που ελέγχουμε.

id1_to_id2

3. Χρειάστηκε σε πολλά σημεία να μετατρέψουμε την εικόνα από μια μορφή σε μια άλλη. Οι συναρτήσεις αυτής της δομής εκτελούν αυτές τις μετατροπές:

   void *id1*_to_*id2*() ,

   όπου *id1* είναι το χρωματικό μοντέλο που διαθέτουμε και *id2* είναι το χρωματικό μοντέλο στο οποίο θέλουμε να μετατρέψουμε την εικόνα μας.

grayscale_id

4. Οι συναρτήσεις αυτές χρησιμοποιούνται από την void rgb_to_grayscale(int sel) που ανοίκει στην παραπάνω κατηγορία συναρτήσεων. Η ιδιαιτερότητα αυτής της συνάρτησης είναι δίνει την δυνατότητα να διαλέξουμε ποια μέθοδο επιθυμούμε να χρησιμοποιήσουμε κατά την μετατροπή μιας έγχρωμης rgb εικόνας σε ασπρόμαυρη. Οι συναρτήσεις αυτές έχουν την παρακάτω δομή :

   void grayscale_*id*() ,

όπου *id1* είναι το χρωματικό μοντέλο που διαθέτουμε και *id2* είναι το χρωματικό μοντέλο στο οποίο θέλουμε να μετατρέψουμε την εικόνα μας.

id1_filter{_id2}

5. Όπως δηλώνει το όνομα filter οι συναρτήσεις αυτές υλοποιούν τις συνελίξεις των εικόνων με τις μάσκες (kernels). Οι συναρτήσεις αυτές έχουν την παρακάτω δομή :

   void *id1*_filter{_*id2*}() ,

   όπου *id1* είναι ο τύπος του φίλτρου που εφαρμόζουμε πχ γκαουσιανό (gaussian) και το *id2* είναι μια παράμετρος που χρησιμοποιείται από τα φίλτρα sobel για να δείξει αν μιλάμε για την παράγωγο ως προς x ή ως προς y.

id_calc

6. Για τους υπολογισμούς που χρησιμοποιούν την βαθμίδα οι συναρτήσεις έχουν την μορφή αυτή :

   void *id*_calc() ,

   όπου *id* είναι ο τύπος της μαθηματικής ιδιότητας της βαθμίδας πχ angle

   Την ίδια δομή έχει και η συνάρτηση που υπολογίζει την βαθμίδα :

   void gradient_calc()

   Μια ματιά στο εσωτερικό αυτής της συνάρτησης :
   Η συνάρτηση αυτή καλεί τις συναρτήσεις φίλτρων sobel (sobel_filter_x και sobel_filter_y)

min{Number} , max{Number}

7. Επειδή σε διάφορα σημεία χρειάστηκε να υπλογισθεί το ελάχιστο ή το μέγιστο στοιχείο μεταξύ 3ων στοιχείων υλοποιήθηκαν οι συναρτήσεις αυτές που δίνουν το αποτέλεσμα αυτό :

   void min{*Number*}() ,

   όπου Number είναι ο αριθμός των στοιχείων που συγκρίνει πχ 2 για να βρει τον ελάχιστο μεταξύ τους.

Ομοίως για την :

void max{*Number*}()

Μια ματιά στο εσωτερικό των συνάρτησεων min3 , max3 :
Οι συναρτήσεις αυτές καλούν τις συναρτήσεις min2 , max2 αντίστοιχα.

Μια ματιά στο εσωτερικό των συνάρτησεων min2 , max2 :
Οι συναρτήσεις αυτές συγκρίνουν τα 2 στοιχεία που τους έχουν δωθεί και επιστρέφουν το μικρότερο ή το μεγαλύτερο από τα 2. Σε περίπτωση που είναι ίσα επιστρέφουν -1. Αυτη την περίπτωση την διαχειρίζονται οι συναρτήσεις που τις κάλεσαν.

find_min, find_max , linear_scaling και scale_magnitude_image

8. Προκειμένου να κάνουμε κανονικοποίηση στις τιμές των pixel της εικόνας εφαρμόζουμε μια απλή ευθεία γραμμή :
$$y = a(x - x_0) + y_0$$
όπου *id1* είναι ο τύπος του φίλτρου που

Πριν όμως εφαρμόσουμε την ευθεία γραμμή πρέπει να βρούμε το ελάχιστο και το μέγιστο των τιμών των pixel για αυτό χρησιμοποιούμε τις συναρτήσεις find_min() και find_max().

scale_rgb_image, find_min_{id}, find_max_{id}

9. Όταν μετατρέπουμε την εικόνα από YUV σε RGB δεν αρκεί να βρούμε τις τιμές των χρωμάτων RGB μόνο. Πρέπει να προσέξουμε το εύρος των τιμών τους να είναι σύμφωνο με την θεωρία όπως έχει αναφερθεί παραπάνω. Για αυτόν τον λόγο η συνάρτηση scale_rgb_image εφαρμόζει κανονικοποίηση πάνω σε κάθε τιμή των RGB πινάκων. Οι συναρτήσεις find_min_{id} και find_max_{id} βρίσκουν τη μικρότερη και τη μεγαλύτερη τιμή του πίνακα με αναγνωριστικό το id. Το id μπορεί να είναι r , g ή b ανάλογα σε ποιο frame εφαρμόζουμε την αναζήτηση. Η αναζήτηση είναι η πιο απλή αλγοριθμικά συγκρίνει όλα τα στοιχεία μεταξύ τους. Φυσικά υπάρχει περιθώριο αλλαγής.

colour_image

10. Το τελευταίο βήμα του απρογράμματος είναι να βάψει την εικόνα σύμφωνα με την γωνία και την ένταση(μέτρο) της βαθμίδας που έχει κάθε pixel.

Είδη Βελτιστοποίησης

Στην προσπάθεια να γράψουμε το παραπάνω πρόγραμμα χρησιμοποιώντας εργαλεία που μετράνε την κατανάλωση μνήμης , επεξεργαστή , χρόνου , πράξεων στον επεξεργαστή και άλλων χρήσιμων πόρων όπως έχει το Visual Studio αλλά και το Armulator παρατηρήσαμε ότι υπήρχαν υψηλές καταναλώσεις μνήμης και άλλων πόρων. Θα θέλαμε να κάνουμε τον κώδικά μας πιο ποιοτικό και για να το πετύχουμε αυτό έχουμε στη διάθεση μας διάφορες τεχνικές βελτιστοποίησης του κώδικά μας ανάλογα την μετρική που θέλουμε να αυξήσουμε ή να μειώσουμε.

Η ποιότητα κώδικα μπορεί να σημαίνει πολλά πράγματα και μπορεί να μετρηθεί με πολλούς τρόπους και σε διαφορετικά στάδια. Αρκεί να αναφέρουμε μερικά παραδείγματα:

1. Ταχύτητα Εκτέλεσης (Runtime Speed)
2. Μέγεθος Κώδικα
3. Ενέργεια που καταναλώνει ο επεξεργαστής κατά της εκτέλεση

Παρακάτω παρουσιάζονται ορισμένες τεχνικές Βελτιστοποίησης συνοπτικά.

Είδη Βελτιστοποίησης βάση του εύρους(scope) της μεθόδου

1. Τοπική Βελτιστοποίηση (Local Optimization) , που βασίζεται στην μέθοδο Απαρίθμησης Τοπικών Τιμών (Local Value Numbering - LVN) και αφορά μικρά κομμάτια κώδικα που μπορούν να εκτελεστούν σειριακά. Στην μέθοδο αυτή βασίζονται οι παρακάτω :
    a. Αναδήπλωση Σταθερών (Constant folding)
    b. Αλγεβρική Απλοποίηση (Algebraic simplification)

* Οι αλγόριθμοι αυτοί είναι εκτός του εύρους της συγκεκριμένης εργασίας αλλά προστέθηκαν για αναφορά μόνο.

2. Βελτιστοποίηση Περιοχής (Regional or Loop-Level Optimization). Αφορούν περιοχές στο πρόγραμμα που περιλαμβάνουν βρόχους και δομές ελέγχου. Μερικές μέθοδοι είναι :
    a. Απαρίθμηση Υπερ-τοπικών Τιμών (Superlocal Value Numbering), είναι μια επέκταση της LVN
    b. Ξεδίπλωμα Βρόχου (Loop unrolling)

3. Βελτιστοποίηση Προγράμματος (Global Optimization). Λόγω του μεγάλου εύρους (scope) και ότι περιλαμβάνουν περιοχές με βρόχους και δομές ελέγχου οι βελτιστοποιήσεις αυτές εφαρμόζονται κατά το στάδιο της ανάλυσης πριν ξαναγράψουν τον κώδικα.
    a. Εντοπισμός Μη αρχικοποιημένων μεταβλητών με ζωντανές δομές (Finding Uninitialized Variables with Live Sets)

   b. Γενικευμένη Τοποθέτηση Κώδικα (Global Code Placement)

4. Βελτιστοποίηση Διαδικασιών (Interprocedural Optimization)
   a. Αντικατάσταση Inline μεθόδων (Inline Substitution)
   b. Τοποθέτηση Διεργασιών (Procedure Placement)

Βαθμωτή Βελτιστοποίηση (Scalar Optimization)

Η βαθμωτή βελτιστοποίηση αφορά την βελτιστοποίηση του κώδικα που έχει αναλάβει ένα νήμα (thread) ελέγχου. Βασίζεται στις παραπάνω μεθόδους βελτιστοποίησης αλλά και στην Στατική Ανάλυση[ref]. Οι τεχνικές που αφορούν την μέθοδο αυτή είναι:

1. Καθαρισμός Άχριστου και Απροσπέλαστου Κώδικα
2. Μετακίνηση Κώδικα (Lazy Code Motion)
3. Εξιδίκευση Υπολογισμού
4. Απαλοιφή Περιττών Υπολογισμών
5. Ενεργοποίηση άλλων Μετασχηματισμών[ρεφ] , όπως είναι ο μετασχηματισμός loop unswitching που βγάζει έξω από έναν βρόχο τις δομές ελέγχου που δεν μεταβάλλονται (loop-invariant control-flow)

Στο σημείο θα αναφέρουμε τις μεθόδους βελτιστοποίησης που μας ενδιαφέρουν περισσότερο σε αυτή τη φάση της εργασίας:

1. Loop Unrolling
2. Loop Fusion - Loop Fission
3. Code Motion
4. Strength Reduction
5. Dead Code Elimination
6. Common Subexpression Elimination (CSE)
7. Function Inlining – Inline Expansion
8. Constant Folding - Copy Propagation

Βελτιστοποιημένος Κώδικας και Μετρήσεις

Για να πετύχουμε υψηλή ποιότητα στον κώδικα αρχικά θέλαμε να είναι όλες οι συναρτήσεις inline σαν πρώτο βήμα. Όμως αυτό δεν είναι εφικτό σε όλες για αυτό και τοποθετήσαμε μόνο τα βασικά blocks μέσα στην main. Το επόμενο βήμα είναι να πετύχουμε βελτιστοποίηση στην κατανάλωση του επεξεργαστή για και ειδικά στα πιο κρίσιμα σημεία. Το visual studio βοηθάει με τον CPU profiler σαν πρώτη εικόνα. Επίσης το ίδιο IDE δίνει την δυνατότητα να βλέπουμε την Process Memory με τα private bytes. Έτσι μπορούμε δοκιμάζοντας τον κώδικα να μειώσουμε την κατανάλωση μνήμης που χρειάζεται.

Μετρήσεις με το Visual Studio IDE

Σύμφωνα με το Visual Studio το μη-βελτιστοποιημένο πρόγραμμα χρρειάζεται 26 MB και χρειάζεται 2.4 s για να εκτελεστεί.Το βελτιστοποιημένο πρόγραμμα χρειάζεται 24 MB και εκτελείται σε 2.5553 s όταν έχουμε σβήσει όλα τα σχόλια , έχουμε κρατήσει μόνο τις συναρτήσεις που είναι απολύτως απαραίτητες για την παραγωγή του σωστού αποτελέσματος και εφόσον τις ενσωματώσαμε όλες στην main.

Αναπτύσσοντας την συνάρτηση padder() σε 3 διαφορετικούς βρόχους έριξε την επεξεργαστική ισχύ από 25% που ήταν προηγουμένως και για τον μη-βελτιστοποιημένο και για τον βελτιστοποιημένο κώδικα σε 22% κατά την εκτέλεση της συγκεκριμένης συνάρτησης.

Έπειτα εφαρμόζουμε loop unrolling στην συνάρτηση που κάνει κανονικοποίηση το κόκκινο frame αφού του φορτώσουμε δεδομένα από την εικόνα YUV.

Η υψηλή κατανάλωση της μνήμης επιχειρήσαμε να την μειώσουμε δεσμεύοντας όλο και λιγότερη μνήμη στις global μεταβλητές.

Από 26MB σε 24MB βάζοντας όλες τις απαραίτητες συναρτήσεις μέσα στην main.

Από 24MB σε 21MB μειώνοντας κατά 3 τους πίνακες (σβήσαμε τους frame_grayscale_r , g , b) και κρατήσαμε μόνο τους πίνακες που αρχηκά λεγόντουσαν frame_y , frame_u , frame_v και frame_r , frame_g , frame_b και τους μετονομάσαμε σε frame_1_a , frame_1_b frame_1c και frame_2_a frame_2_b frame_2_c

Από 21 MB σε 17MB σβήνοντας τους πίνακες coloured_image που χρησιμοποιούνται στο βήαμ 7 και τους αντικαταστήσαμε με τους παραπάνω frame_1_{a,b,c} και frame_2_{a,b,c}.

Μετρήσεις με το CodeWarrior IDE

Χρειάστηκε να γίνουν κάποιες αλλαγές στον κώδικα , προκειμένου να μπορέσει να εκτελεστεί σωστά από το CodeWarrior IDE. Αυτές είναι η περιοχή pragma arm section και οι διορθώσεις των pow(), arctan() διότι είχαν unhandled overflow. Επίσης χρειάστηκαν τα αρχεία scatter.txt, memory.map και stack. Το πρώτο δείχνει τον χώρο αποθήκευσης (δομή

μνήμης στο λογισμικό) , το δεύτερο την δομή της μνήμης στο υλικό και το τρίτο δηλώνει την δομή δεδομένων την οποία έχει η μνήμη.

Οι μέθοδοι βελτιστοποίησης που χρησιμοποιήσαμε ήταν διαδοχικά :

1. Inline Substitution – Code Motion. Αντί για να βάλουμε inline μπροστά από τις συναρτήσεις τις βάλαμε μέσα στην main γιατί παρατηρήσαμε ότι βελτίωνε τις μετρικές του προγράμματος.
2. Loop unrolling. Στους περισσότερους βρόχους εφαρμόσαμε την τεχνική αυτή.
3. Loop fussion. Στον βρόχο που δημιουργεί την επαυξημένη εικόνα εφαρμόσαμε την τεχνική αυτή. Σπάσαμε την δομή for-for-if-else σε for-for , for, for.
4. Loop fission. Όταν εκτελούμε μετατροπές από ένα χρωματικό χώρο σε άλλο κρατήσαμε τους βρόχους ενωμένους.
5. Dead Code Elimination. Υπήρχαν κάποια μονοπάτια στον κώδικα που δεν εκτελούνταν ποτέ. Τα απαλειψαμε.
6. Strength Reduction. Πολλές αλγεβρικές πράξεις που υπήρχαν στον κώδικα τις κόψαμε σε δύο κομμάτια μειώνοντας την βαρύτητα της συνολικής πράξης.
7. Copy Propagation. Στα #define και στην μεταβλητή neighborhood_of_image αντικαταστήσαμε αρχικά τις τιμές πχ filename με τη πραγματική τιμή. Όμως η neighborhood_of_image είχε κι άλλα περιθώρια βελτιστοποίησης και ενώ την κάναμε να φορτώνει δεδομένα από τον πίνακα και να αρχικοποιείται τελικά ανοιγωντά την στον χώρο (κάθε στοιχείο φορτώνεται σε ξεχωριστή γραμμή) είχε πιο πολύ κόστος. Τελικά με τις δύο for πετύχαμε βελτιστοποίηση του κώδικα.
8. Common Subexpression Elimination. Οι τιμές των μεταβλητών slope_id υπολογίζονται στην αχή του προγράμματος και όχι μέσα στην for-for δομή όπου και χρησιμοποιούνται.

Τα στοιχεία του αρχικού προγράμματος φαίνονται παρακάτω:

initial	Object	7248	60	108	22197156	24016
	Library	23180	610	0	300	8216
	Grand	30428	670	108	22197456	32232
	RO	31098	30.36914063	kB		
	RW	22197564	21.16924667	MB		
	ROM	31206	30.47460938	kB		

Πίνακας 2 - Απαιτήσεις Μνήμης Αρχικού Προγράμματος

Ενώ του τελικού είναι:

optimized-1	Object	6240	168	0	0	10364
	Library	17664	466	0	300	7240

	Grand	23904	634	0	300	17604
	RO	24538	23.96289063	kB		
	RW	300	0.29296875	kB		
	ROM	24538	23.96289063	kB		

Πίνακας 3 - Απαιτήσεις Μνήμης Βελτιστοποιημένου Προγράμματος

Και οι εντολές και οι κύκλοι στον επεξεργαστή είναι:

NO MEMORY	Instuctions	Core Cycles	S Cycles	N Cycles	I Cycles	C Cycles	Total
initial	620226123	102298801	728855498	232897531	120254198	0	1082007227
optimized-1	844442376	1287663527	974504597	243573763	167334313	0	1385412673

Πίνακας 4 - Σύνολο Εντολών Αρχικού και Βελτιστοποιημένου Προγράμματος

Παρατηρούμε ότι έχουμε αύξηση των εντολών και των κύκλων εκτέλεσης. Συνεπώς δεν μπορούμε να κάνουμε πολλά για να βελτιστοποιήσουμε τον κώδικα έτσι όπως είναι αυτή τη στιγμή. Εφαρμόζοντας όλες τις παραπάνω τεχνικές δεν λύνεται το βασικότερο πρόβλημα που είναι η επαναχρησιμοποίηση μεταβλητών μέσα στην cache. Αν είχαμε την δυνατότητα να έχουμε μόνο ένα πίνακα της εικόνας δίπλα στον επεξεργαστή, τότε το πλήθος των εντολών θα ήταν πάρα πολύ μικρότερο.

Μνήμη Ενσωματωμένων Συστημάτων με χρήση ARM επεξεργαστών

Έχοντας πλέον αναπτύξει τον κώδικα του προγράμματος εκτελώντας τα παρακάτω βήματα:

1. Αρχικές Προδιαγραφές
2. Υλοποίηση και δοκιμή κώδικα
3. Εξήγηση κώδικα
4. Βελτιστοποίηση Κώδικα
5. Μετρήσεις για σύγκριση

θα κοιτάξουμε το επόμενο κομμάτι που είναι η μνήμη.

Στην περίπτωση μας, μας ενδιαφέρει μόνο η μνήμη μέσα στον επεξεργαστή (on-chip). Η μνήμη αυτή αποτελείται από δύο κομμάτια: την μνήμη ROM στην οποία θα έχουμε τον κώδικα του προγράμματος και τα αρχικά δεδομένα (Zero-Initialized Data) και την μνήμη RAM στην οποία θα έχουμε τις δομές heap και stack και μερικά δεδομένα που προκύπτουν από το πρόγραμμα καθώς λειτουργεί.

Η διευθυνσιοδότηση της μνήμης είναι δομημένη σειριακά:

1. ROM: 0x000000 ~ ROM_MAX
2. RAM: ROM_MAX ~ RAM_MAX

Μέσα στην RAM έχουμε τις δομές heap και stack. Κάθε μία από αυτές διαθέτει μια διεύθυνση βάσης από όπου ξεκινάνε να αυξάνονται σε μέγεθος. Οι διευθύνσεις αυτές πρέπει να δηλώθούν ώστε να έχουμε ολοκληρωμένη εικόνα του μοντέλου μνήμης του συστήματος μας. Είναι σημαντικό να μην ταυτίζονται και πρέπει να προβλέψουμε να μην επικαλυφθούν οι δύο δομές καθώς μεγαλώνουν σε μέγεθος.

Στο αρχικό πρόγραμμα , το μη-βελτιστοποιημένο τοποθετούμε μνήμη και έχουμε συνολικά 256 MB:

1. ROM: 0x00 000 000 ~ 0x04 000 000 = 64 MB
2. RAM: 0x04 000 000 ~ 0x10 000 000 = 192 MB

Στην RAM θα δηλώσουμε και τις διευθύνσεις της βάσης της δομής heap και της στοίβας :

1. Βάση heap: 0x6 000 000
2. Βάση στοίβας: 0x8 000 000

Οπότε δίνουμε χώρο στην heap να αναπτυχθεί είτε προς τα πάνω (προς τις μικρότερε διευθύνσεις) είτε προς τα κάτω(μεγαλύτερες διευθύνσεις) και την stack να χρησιμοποιήσουν μέχρι 32 MB. Τα υπόλοιπα 64 MB είναι Zero-Initialized data και global μεταβλητές και είναι χώρος που αφορά τους πίνακες των frame.

Στο βελτιστοποιημένο πρόγραμμα που έχουμε από την προηγούμενη ενότητα πάμε να προσθέσουμε μνήμη. Επειδή χρειαζόμαστε περίπου 16 MB για την ROM και 48 MB (συνολικά

περίπου 64 MB) για την RAM δηλώνουμε αντίστοιχα τις διευθύνσεις ROM_MAX και RAM_MAX ως 0x1 000 000 και 0x4 000 000. Έτσι έχουμε:

1. ROM: 0x0 000 000 ~ 0x1 000 000 = 16 MB
2. RAM: 0x1 000 000 ~ 0x4 000 000 = 48 MB

Στην RAM θα δηλώσουμε και τις διευθύνσεις της βάσης της δομής heap και της stack:

1. Βάση heap: 0x1 000 000
2. Βάση στοίβας: 0x2 000 000

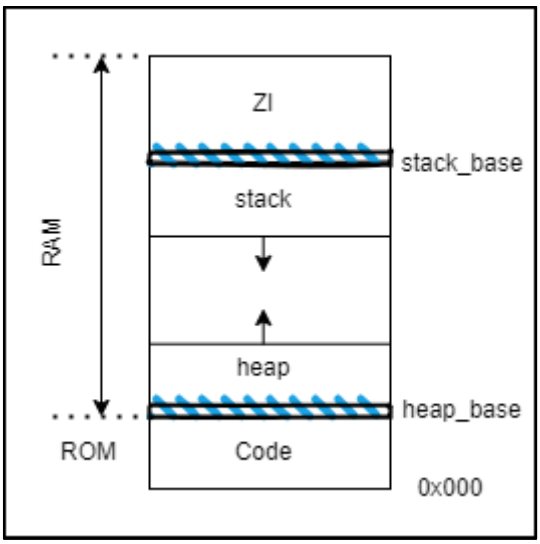
Οπότε δίνουμε χώρο στην heap και την stack να χρησιμοποιήσουν μέχρι 16 MB.

Τα υπόλοιπα 32 MB είναι Zero-Initialized data και είναι χώρος που αφορά τους πίνακες των frame.

Για τις μνήμες πέρα από το μέγεθος τους μπορούμε να καθορίσουμε και την διεπαφή τους. Μπορούμε να δηλώσουμε το πλάτος του διαύλου επικοινωνίας και τους χρόνους μη-διαδοχικής και διαδοχικής προσπέλασης. Στις παραπάνω ρυθμίσεις είχαμε :

1. ROM: δίαυλο 4bytes, μη-διαδοχική/διαδοχική προσπέλαση 1/1
2. RAM: δίαυλο 4bytes, μη-διαδοχική/διαδοχική προσπέλαση 250/50

Ένα σχεδιάγραμμα της μνήμης μπορεί να είναι το παρακάτω:



Εικόνα 1 - Δομή της Μνήμης που προτείνουμε

Λογισμικό για ARM επεξεργαστή

Η διαδικασία υλοποίησης λογισμικού για επεξεργαστές της ARM είναι μια ροή που αποτελείται από τα παρακάτω βήματα:

1. Κώδικας C/C++/Java/Python (.c , .h, .cpp, .hpp, .py, .java, .jar)
2. Κώδικας Μηχανής (.o , .obj)
3. Εικόνα .axf

Ο κώδικας που γράφουμε περνάει από τον μεταγλωτιστή (compiler) και παράγει τον κώδικα μηχανής, ο οποίος με την σειρά του περάνει από τον ARM Linker και παίρνουμε το τελικό εκτελέσιμο αρχείο σε μορφή .axf . Το αρχείο .axf μπορούμε να το εξετάσουμε με τον debugger και να πάρουμε διάφορες μετρήσεις και δεδομένα για τις εντολές , τους κύκλους του επεξεργαστή και τους χρόνους τους συστήματος.

Τα αρχεία scatter.txt , mem.c χρησιμοποιούνται από τον compiler ώστε να τοποθετηθούν σωστά τα δεδομένα μέσα στα αρχεία .o (ή .obj).

Τα αρχεία memory.map χρησιμοποιούνται από τον debugger ώστε η εικόνα .axf να μπορέσει να τρέξει με τις καθορισμένες ταχύτητες, χρόνους.

Αρχεία scatter.txt

Είναι αρχεία που δείχνουν την δομή της μνήμης. Πόσες μνήμες, τι ονόματα έχουν και το εύρος των διευθύνσεων τους. Όταν παράγεται ο κώδικας μηχανής ο compiler έχει χρησιμοποιήσει την δομή που του δώσαμε και τοποθέτησε τα δεδομένα στις αντίστοιχες θέσεις. Όταν θα παραχθεί η εικόνα .axf έχει δηλωθεί η μνήμη και μάλιστα έχουν δηλωθεί δύο μνήμες:

1. LOAD MEM: είναι η μνήμη που είναι συμπιεσμένος ο εκτελέσιμος κώδικας
2. EXECUTE MEM: είναι η μνήμη που θα τοποθετηθούν όλα τα δεδομένα όπως έχουμε δηλώσει στο αρχείο scatter.txt κατά την αποσυμπίεση.

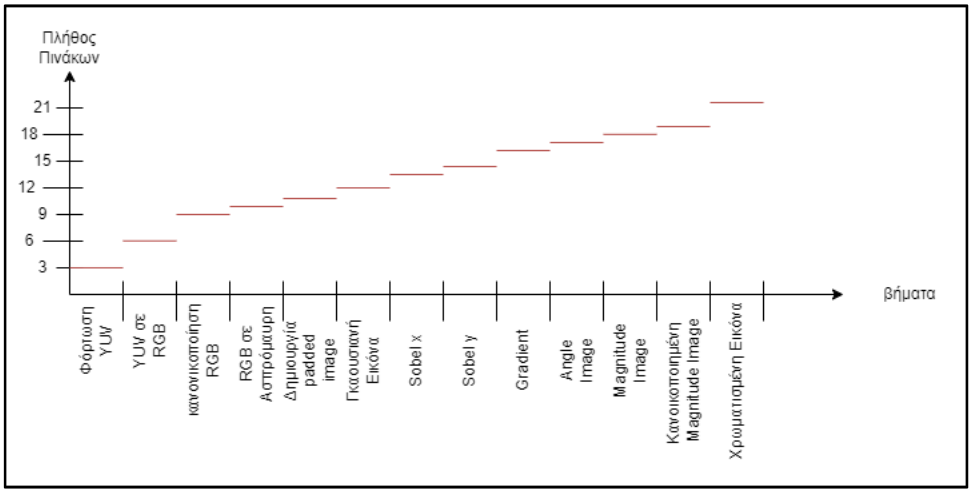
Αρχεία *.map

Είναι αρχεία που δείχνουν τα χαρακτηριστικά των δηλωμένων μνημών. Κατά την εκτέλεση μέσω του debugger του .axf αρχείου πρέπει να δηλώσουμε τα χαρακτηριστικά του επεξεργαστή και της μνήμης. Για τους επεξεργαστές έρχονται με το λογισμικό. Για την μνήμη μπορούμε να χρησιμοποιήσουμε την προεπιλεγμένη επιλογή (μνήμη του υπολογιστή) ή να δηλώσουμε εμείς την επιθυμιτή μνήμη μέσω των αρχείων .map . Μια γραμμή σε αυτό το αρχείο περιλμβάνει:

1. Αρχική Διεύθυνση μνήμης (να συμβαδίζει με το αρχείο scatter.txt)
2. Τελική Διεύθυνση μνήμης (να συμβαδίζει με το αρχείο scatter.txt)
3. Όνομα μνήμης (να συμβαδίζει με το αρχείο scatter.txt)
4. Εύρος Διάυλου επικοινωνίας σε bytes
5. Τύπος μνήμης ως προς την ανάγνωση και εγγραφή (R, RW)
6. Χρόνος Ανάγνωσης (ns): Μη διαδοχικής Προσπέλασης / Διαδοχικής Προσπέλασης
7. Χρόνος Εγγραφής (ns): Μη διαδοχικής Προσπέλασης / Διαδοχικής Προσπέλασης

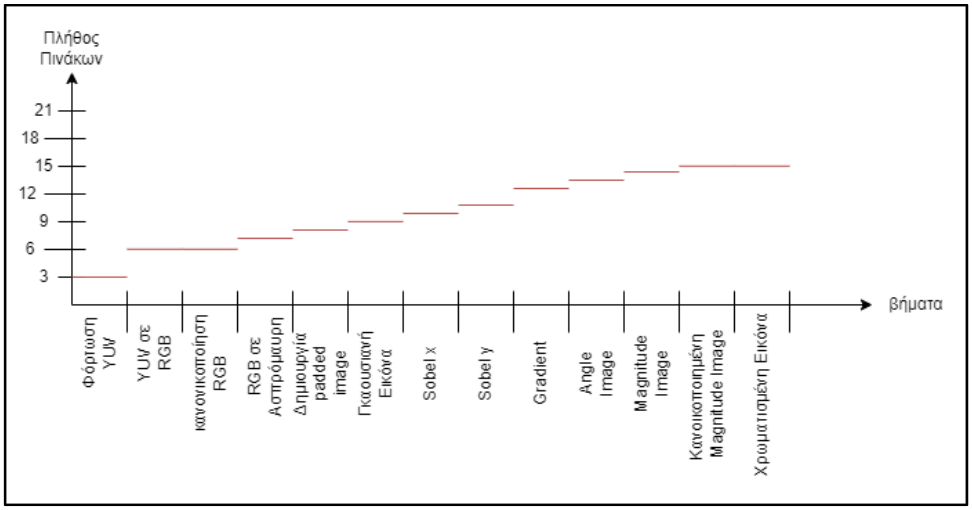
Βελτιστοποίηση σε Επίπεδο Μνήμης

Διαπιστώσαμε ότι η ανάγκες του προγράμματος σε μνήμη είναι πολύ μεγάλες. Για να μειώσουμε αυτές τις ανάγκες θα προσπαθήσουμε να μειώσουμε το πλήθος πινάκων σε συνάρτηση με τον χρόνο. Η χρήση της μνήμης στον αρχικό κώδικα ήταν:



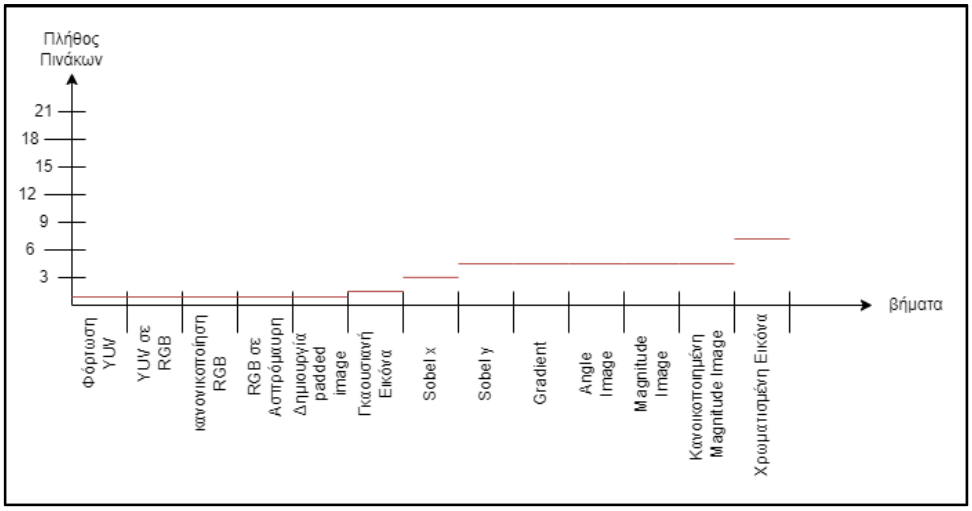
Εικόνα 2 - Διάγραμμα πλήθους πινάκων σε κάθε βήμα του αρχικού προγράμματος

Η χρήση 21 πινάκων χωρίς να επαναχρησιμοποιούμε κάποιους από αυτούς δεν είναι επιθυμητό. Στην αμέσως επόμενη βελτιωμένη έκδοση έχουμε:



Εικόνα 3 - Διάγραμμα πλήθους πινάκων σε κάθε βήμα του βελτιστοποιημένου προγράμματος

Από τους 21 το πλήθος των πινάκων έπεσε σε 16. Πάλι όμως ήταν πάρα πολλοί για μια εφαρμογή ενσωματωμένων συστημάτων πόσο μάλλον για την επεξεργασία μιας εικόνας. Στην τελευταία έκδοση (optimized-2) έχουμε μειώσει το πλήθος των πινάκων σε 7:



Εικόνα 4 - Διάγραμμα πλήθους πινάκων σε κάθε βήμα του βελτιστοποιημένου προγράμματος με μνήμη

Παρακάτω εξηγούμε τα βήματα που ακολουθήσαμε και καταλήξαμε στην τελική μορφή.

Αρχικά, για να μειώσουμε τον χώρο που καταναλώνουμε από τα Zero-Initialized δεδομένα πετάμε τις περιττές αρχικοποιήσεις. Έπειτα παρατηρούμε ότι οι μάσκες των φίλτρων kernel έχουν διαφορά μεταξύ τους 90 μοίρες είναι δηλαδή περιστραμένα δεξιόστροφα. Επομένως δεν χρειάζεται να έχουμε ξεχωριστό πίνακα αρκεί να διατρέξουμε τα στοιχεία του ίδιο πίνακα κατά γραμμή ή κατά στήλη ανάλογα ποιον θα κρατήσουμε. Εμείς κρατήσαμε την μάσκα x οπότε τον ίδιο πίνακα τον διατρέχουμε κατά γραμμές όταν πρόκειται για το φίλτρο x και κατά στήλες όταν πρόκειται για το φίλτρο y. Επίσης, αντί να φορτώνουμε τα τρία frames Y, U, V ενώ χρειαζόμαστε μόνο την γκρί εικόνα, φορτώνουμε μόνο το Y frame και το κανονικοποιούμε για να είμαστε μεταξύ των τιμών 0 έως 255. Τα αποτελέσματα φαίνονται παρακάτω:

optimized-2-v1	Object	3936	112	0	0	9748
	Library	17596	466	0	300	7164
	Grand	21532	578	0	300	16912
	RO	22110	21.59179688	kB		
	RW	300	0.29296875	kB		
	ROM	22110	21.59179688	kB		

Πίνακας 5 - Απαιτήσεις μνήμης του βελτιστοποιημένου προγράμματος με μνήμη έκδοση v1

Έπειτα αντί να χρησιμοποιούμε 3 πίνακες, έναν για να φορτώνουμε τα δεδομένα, έναν για την κανονικοποίηση και έναν για την επαυξημένη εικόνα, μπορούμε να χρησιμοποιήσουμε μόνο έναν. Κάνοντας τις απαραίτητες αλλαγές έχουμε :

optimized-2-v2	Object	3936	112	0	0	9764
	Library	17596	466	0	300	7164
	Grand	21532	578	0	300	16928
	RO	22110	21.59179688	kB		
	RW	300	0.29296875	kB		
	ROM	22110	21.59179688	kB		

Πίνακας 6 - Απαιτήσεις μνήμης του βελτιστοποιημένου προγράμματος με μνήμη έκδοση v2

Επειδή τους πίνακες frame_padded και frame_filtered_y δεν τους χρειαζόμαστε εφόσον υπολογίσουμε τα φίλτρα sobel μπορούμε να τους επαναχρησιμοποιήσουμε για την γωνία και το πλάτος. Οπότε αντί να χρησιμοποιούμε ξεχωριστά array για αυτές τις διαδικασίες επαναχρησιμοποιούμε ήδη δεσμευμένες θεσης μνήμης. Κάνοντας τις απαραίτητες αλλαγές έχουμε τα παρακάτω αποτελέσματα:

optimized-2-v3	Object	3200	112	0	0	9100
	Library	17596	466	0	300	7164
	Grand	20796	578	0	300	16264
	RO	21374	20.87304688	kB		
	RW	300	0.29296875	kB		
	ROM	21374	20.87304688	kB		

Πίνακας 7 - Απαιτήσεις μνήμης του βελτιστοποιημένου προγράμματος με μνήμη έκδοση v3

Συγκεντρωτικά φαίνονται όλα παρακάτω:

WITH MEMORY	Instuctions	Core Cycles	S Cycles	N Cycles	I Cycles	C Cycles	Wait Cycles	Total	True Idle Cycles
initial	620,226,214	1,022,989,014	728,855,589	232,897,632	120,254,219	0	1,240,908,894	2,322,916,334	35,284,909
optimized-1	844,024,822	1,286,099,875	973,524,658	243,016,796	168,038,328	0	1,291,521,998	2,676,101,780	51,433,781
optimized-2-v1	560,637,052	857,471,806	649,000,191	163,202,201	94,488,489	0	814,989,422	1,721,680,303	27,328,414
optimized-2-v2	560,637,052	857,471,806	649,000,191	163,202,201	94,488,489	0	814,989,422	1,721,680,303	27,328,414
optimized-2-v3	470,994,406	694,339,163	534,629,122	123,891,439	84,076,059	0	614,427,236	1,357,023,856	25,006,719

Πίνακας 8 - Εντολές και Κύκλοι του επεξεργαστή μαζί με την μνήμη.

Παρατηρούμε ότι το μέγεθος της ROM μνήμης μειώνεται όλο και πιο πολύ , οπότε δοκιμάζοντας μειώσαμε και την μνήμη που δηλώνουμε στα memory.map και scatter.txt . Επίσης δηλώνοντας προσεκτικά τις βάσεις των stack , heap προκύπτει ο παρακάτω συγκεντρωτικός πίνακας:

	ROM			RAM					Total MB
	Start Address	End Address	MB	Start Address	Heap Base	Stack Base	End Address	MB	
initial	0x00	0x04000000	64	0x04000000	0x06000000	0x08000000	0x10000000	192	256
optimized-1	0x00	0x01000000	16	0x01000000	0x01000000	0x02000000	0x04000000	48	64
optimized-2-v1	0x00	0x01000000	16	0x01000000	0x01000000	0x02000000	0x04000000	48	64
optimized-2-v2	0x00	0x01000000	16	0x01000000	0x01000000	0x02000000	0x04000000	48	64
optimized-2-v3	0x00	0x00800000	8	0x00800000	0x00800000	0x01000000	0x02000000	24	32

Πίνακας 9 - Δομή μνήμης των προγραμμάτων με μνήμη

Έχουμε φτάσει την συμπιεσμένη μνήμη ROM στα 8 MB και την μνήμη RAM κατά την εκτέλεση του προγράμματος (και μετά την αποσυμπίεση) στα 24 MB. Έχουμε κάνει σχεδόν

ότι ήταν δυνατό για να μειώσουμε το μέγεθος της απαιτούμενης μνήμης. Τώρα ας κοιτάξουμε την δομή και την ταχύτητα της μνήμης.

Αρχικά θα δοκιμάσουμε να μειώσουμε την ταχύτητα που χρειάζεται ο επεξεργαστής για την επικοινωνία με την μνήμη. Έχουμε λοιπόν τον παρακάτω πίνακα με διαφορετικά αρχεία .map για την έκδοση v1 του βελτιστοποιημένου κώδικα:

Base	Dec	From .map file :										
cw-optimized-2-v1	name	Nread_ns	Nwrite_ns	Sread_ns	Swrite_ns	ns	s	Wait_States	Total	clock	cputime	sys_clock
memory.map	ROM	1	1	1	1	87764820	14	814989422	1721680303	34433606	1721680303	3443
	RAM	250	250	50	50	456071460	18					
memory2.map	ROM	1	1	1	1	87764820	14	1629978844	2536669725	50733394	2536669725	5073
	RAM	500	500	100	100	755859900	34					
memory3.map	ROM	1	1	1	1	87764820	14	407494711	1314185592	26283711	1314185592	2628
	RAM	125	125	25	25	306177240	10					
memory4.map	ROM	1	1	1	1	87764820	14	910745500	1817436381	36348727	1817436381	3634
	RAM	250	250	100	100	371193020	20					
memory5.map	ROM	1	1	1	1	659101960	11	767111383	1673802264	33476045	1673802264	3347
	RAM	250	250	25	25	498510680	17					

Πίνακας 10 - Μετρήσεις της μνήμης του βελτιστοποιημένου κώδικα v1

Αντίστοιχες μετρήσεις πήραμε για την έκδοση v2 και την v3:

Base	Dec	From .map file :										
cw-optimized-2-v2	name	Nread_ns	Nwrite_ns	Sread_ns	Swrite_ns	ns	s	Wait_States	Total	clock	cputime	sys_clock
memory.map	ROM	1	1	1	1	87764820	14	814989422	1721680303	34433606	1721680303	3443
	RAM	250	250	50	50	456071460	18					
memory2.map	ROM	1	1	1	1	87764820	14	1629978844	2536669725	50733394	2536669725	5073
	RAM	500	500	100	100	755859900	34					
memory3.map	ROM	1	1	1	1	87764820	14	407494711	1314185592	26283711	1314185592	2628
	RAM	125	125	25	25	306177240	10					
memory4.map	ROM	1	1	1	1	87764820	14	910745500	1817436381	36348727	1817436381	3634
	RAM	250	250	100	100	371193020	20					
memory5.map	ROM	1	1	1	1	87764820	14	767111383	1673802264	33476045	1673802264	3347
	RAM	250	250	25	25	498510680	17					

Πίνακας 11 - Μετρήσεις της μνήμης του βελτιστοποιημένου κώδικα v2

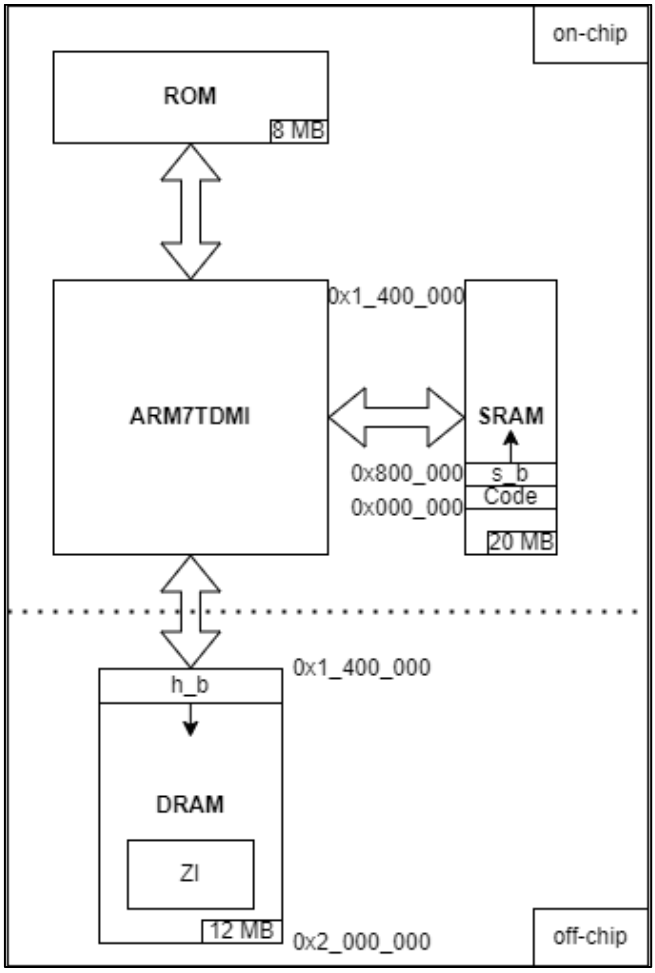
Base	Dec	From .map file :										
cw-optimized-2-v3	name	Nread_ns	Nwrite_ns	Sread_ns	Swrite_ns	ns	s	Wait_States	Total	clock	cputime	sys_clock
memory.map	ROM	1	1	1	1	659101960	11	614427236	1357023856	27140477	1357023856	2714
	RAM	250	250	50	50	799853980	13					
memory2.map	ROM	1	1	1	1	659101960	11	1228854472	1971451092	39429021	1971451092	3942
	RAM	500	500	100	100	88398700	26					
memory3.map	ROM	1	1	1	1	659101960	11	307213618	1049810238	20996204	1049810238	2099
	RAM	125	125	25	25	655581620	7					

memory 4.map	ROM	1	1	1	1	659101960	11	67289800	1415495520	28309910	1415495520	2830
	RAM	250	250	100	100	969287260	14					
memory 5.map	ROM	1	1	1	1	659101960	11	585191404	1327788024	26555760	1327788024	2655
	RAM	250	250	25	25	215137340	13					

Πίνακας 12 - Μετρήσεις της μνήμης του βελτιστοποιημένου κώδικα v3

Παρατηρούμε ότι η καλύτεροι χρόνοι είναι όταν η διεπαφή της μνήμης είναι στα 125 ns και 25 ns για τις μη-διαδοχικές και τις διαδοχικές εγγραφές/αναγνώσεις αντίστοιχα.

Θεωρώντας ότι είναι μία μνήμη εκτός του chip δεν μπορύμε παρά να “διορθώσουμε” αυτή την θεώρηση. Έτσι λοιπόν παίρνουμε τους παραπάνω χρόνους και τους αναθέτουμε στην καινύργια DRAM μας (off-chip) και θεωρούμε και μία on-chip μνήμη την SRAM. Έτσι πλέον μπορούμε να έχουμε την παρακάτω δομή:



Εικόνα 5 - Δομή της Μνήμης με on-chip και off-chip μνήμες

Να σημειωθεί ότι το διπλό βέλος μεταξύ του επεξεργαστή και της μνήμης ROM υποδεικνύει τις δύο λειτουργίες: την αποσυμπίεση της μνήμης LOAD_ROM και την ανάγνωση των δεδομένων μόνο για ανάγνωση.

Μια επιπλέον διαφοροποίηση με την προηγούμενη δομή της μνήμης είναι ότι τοποθετήσαμε τα Zero-Initialized δεδομένα μέσα στην DRAM και μέσα στην SRAM κρατήσαμε την ίδια δομή με πριν της stack και της heap.

Παρατηρούμε από τον Πίνακα-9 ότι τα wait states (καταστάσεις αναμονής του διάυλου και του επεξεργαστή) είναι 1,240,908,894 και φτάνουν στην έκδοση 3 του κώδικα τις 614,427,236. Αν εκμεταλλευτούμε την νέα δομή της μνήμης, έχουμε:

WITH MEMORY	Instuctions	Core Cycles	S Cycles	N Cycles	I Cycles	C Cycles	Wait Cycles	Total	True Idle Cycles
scatter.txt	470,994,406	694,339,163	534,629,122	123,891,439	84,076,059	0	614,427,236	1,357,023,856	25,006,719
scatter2.txt	470,994,406	694,339,163	534,629,122	123,891,439	84,076,059	0	0	742,596,504	25,006,716
scatter3.txt	470,994,406	694,339,163	534,629,122	123,891,439	84,076,059	0	0	742,596,504	25,006,716

Πίνακας 13 - Εντολές και Κύκλοι του Επεξεργαστή μαζί με τις 3 διαφορετικές δομές μνήμης

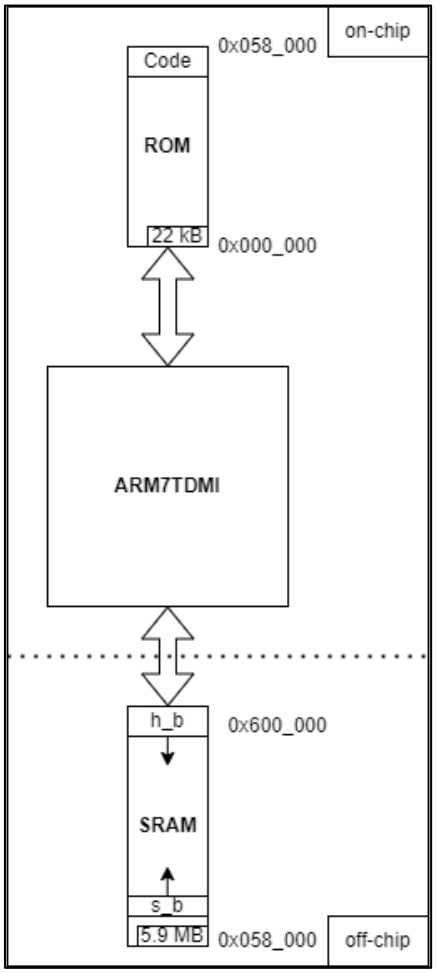
Οι μετρήσεις της μνήμης φαίνονται αναλυτικά παρακάτω:

Base	Hex	From scatter.txt :		From .map file :									
cw-optimized-2-v3	name	start	limit	Nread_ns	Nwrite_ns	Sread_ns	Swrite_ns	Nreads	Nwrites	Sreads	Swrites	ns	s
scatter.txt	ROM	0	800000	1	1	1	1	49F7FD0	0	1E1FB08A	0	53B2F54	E
	RAM	800000	2000000	7D	7D	19	19	1A3DE69	11F10B6	C518D5	F901A3	123FE4D8	A
scatter2.txt	ROM	0	8000000	1	1	1	1	49F7FBA	0	1E1FB056	0	2749340	B
	SRAM	800000	1400000	1	1	1	1	1A3DE5D	11F10AD	C518D5	F9019E	1E79F360	1
	DRAM	1400000	2000000	7D	7D	19	19	0	0	0	0	0	0
scatter3.txt	ROM	0	800000	1	1	1	1	1F4D41D	19DB953	13A7488	1A01B0F	1B2F1924	12
	SRAM	800000	1400000	A	A	A	A	1A3DE5D	11F10AD	C518D5	F9019E	1E79F360	1
	DRAM	1400000	2000000	7D	7D	19	19	0	0	0	0	0	0

Πίνακας 14 - Μετρήσεις της μνήμης του βελτιστοποιημένου κώδικα v3 με τις διαφορετικές δομές μνήμης

Παρατηρούμε ότι οι εγγραφές και ανγνώσεις στην μνήμη DRAM είναι μηδενικές. Όποτε είναι επιπλέον χώρος που δεν χρησιμοποιείται.

Επαναδιατάζουμε την μνήμη:



Εικόνα 6 - Δομή της Μνήμης on-chip ROM και off-chip SRAM μνήμες

Αυτή την φορά δεν σπάμε την μνήμη σε δύο κομμάτια. Και έχουμε τα παρακάτω αποτελέσματα συγκριτικά με τις προηγούμενες δομές:

WITH MEMORY	Instuctions	Core Cycles	S Cycles	N Cycles	I Cycles	C Cycles	Wait Cycles	Total	True Idle Cycles
scatter.txt	470,994,406	694,339,163	534,629,122	123,891,439	84,076,059	0	614,427,236	1,357,023,856	25,006,719
scatter2.txt	470,994,406	694,339,163	534,629,122	123,891,439	84,076,059	0	0	742,596,504	25,006,716
scatter3.txt	470,994,406	694,339,163	534,629,122	123,891,439	84,076,059	0	0	742,596,504	25,006,716
scatter4.txt	470,994,406	694,339,163	534,629,122	123,891,439	84,076,059	0	243,790,188	986,386,808	25,006,719

Πίνακας 15 - Εντολές και Κύκλοι των 4ων διαφορετικών δομών μνήμης

Και για την μνήμη παίρνουμε τις παρακάτω μετρήσεις:

Base	Hex	From scatter.txt :		From .map file :									
cw-optimized-2-v3	name	start	limit	Nread_ns	Nwrite_ns	Sread_ns	Swrite_ns	Nreads	Nwrites	Sreads	Swrites	ns	s
scatter4.txt	ROM	0	58000	1	1	1	1	49F7FD0	0	1E1FB08A	0	27491908	B
	SRAM	58000	600000	64	64	32	32	1A3DE69	11F10B6	C518D5	F901A3	1712E03C	6

Πίνακας 16 - Μετρήσεις της μνήμης του βελτιστοποιημένου κώδικα v3 μαζί με την βέλτιστη δομή μνήμης

Για τους χρόνους που έτρεξε ο επεξεργαστής για τα παραπάνω scatter αρχεία πήραμε τα παρακάτω αποτελέσματα:

Base	Dec			
cw-optimized-2-v3	name	clock (s)	cputime (in clock units)	sys_clock
scatter4.txt	ROM	19.727736	986386808	1972
	SRAM			
scatter3.txt	ROM	14.85193	742596504	1485
	SRAM			
	DRAM			
scatter2.txt	ROM	14.85193	742596504	1485
	SRAM			
	DRAM			
scatter.txt	ROM	26.283711	1314185592	2628
	RAM			

Πίνακας 17 - Χρόνοι Εκτέλεσης

Θα μπορούσαμε να διαλέξουμε μία SRAM ή μία DRAM που να έχει μέγεθος 16Mbits και να μπορούμε να διαβάσουμε και να γράψουμε με χρόνους 100 ns και 50 ns για μη διαδοχικούς και διαδοχικούς κύκλους αντίστοιχα.

Υλοποίηση Φίλτρων με buffers και Χρήση διαφορετικών περιοχών μνήμης

Buffers (optimized-3-v1)

Μέχρι τώρα κάναμε πράξεις με τους πίνακες ολόκληρους φορτώναμε από την μνήμη το στοιχείο i,j και μερικά τριγύρω του και κάναμε πράξεις με αυτά. Η διαδικασία αυτή σε εντολές assembly θα ήταν:

1. Υπολόγισε την διεύθυνση μνήμης
2. Φόρτωσε τα δεδομένα από την μνήμη σε καταχωρητή
3. Κάνε τις πράξεις πχ πρόσθεση / αφαίρεση / πολλαλασιασμό κτλ
4. Υπολόγισε την διεύθυνση αποθήκευσης
5. Αποθήκευσε τα δεδομένα

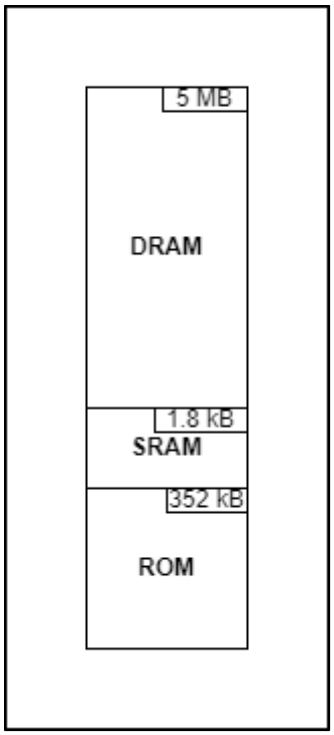
Μπορούμε να εξαλήψουμε πράξεις πολλαπλασιασμού στα συγκλεκριμένα δύο φίλτρα κάνοντας μόνο προσθέσεις και αφαιρέσεις.
Μπορούμε να μειώσουμε την καθυστέρηση μεταφοράς των δεδομένων από και προς την μνήμη φαίρνοντας ολόκληρες γραμμές πιο κοντά. Έτσι έχοντας μάσκες 3x3 μπορούμε να φαίρνουμε κοντά στον επεξεργαστή, όσο γίνεται, όλα τα στοιχεία των τριών γραμμών του πίνακα. Έτσι αντί να έχουμε πχ 150 ns καθυστέρηση στην μεταφορά των δεδομένων από την μνήμη προς τον επεξεργαστή για ένα δεδομένο πχ το στοιχείο i,j θα έχουμε μόνο 10 ns εφόσον είναι σε μια πιο κοντινή μνήμη. Λόγω της φύσης των φίλτρων όταν υπολογίζουμε το στοιχείο i,j χρειαζόμαστε τα στοιχεία γύρω του κατά 1 θέση. Φέρνοντας τα δεδομένα κατά γραμμές όταν φτάσουμε στην τελευταία γραμμή μεταβαίνουμε επόμενη γραμμή. Με την χρήση των buffers τα στοιχεία από τις προηγούμενες γραμμές είναι κοντά στην μνήμη εκτός από την 3η γραμμή για την οποία χρείαζεται να φορτώσουμε από την αργή μνήμη τα δεδομένα. Έχουμε έτσι κάθε φορά απαίτηση για μια γραμμή δεδομένων από την μνήμη και 3ων όπως την πρώτη φορά.

Κάνοντας CMake την εικόνα .axf έχουμε τις παρακάτω μετρήσεις:

Object	2972	60	0	5200152	8568
Library	17488	466	0	300	7080
Grand	20460	526	0	5200452	15648
RO	20986	20.49414063	kB		
RW	5200452	4.959537506	MB		
ROM	20986	20.49414063	kB		

Πίνακας 18 - Μετρήσεις Αντικειμένων στην εικόνα .axf της έκδοσης v1

Η δομή μνήμης που προτείνουμε είναι:



Εικόνα 7 - Δομή της Μνήμης on-chip ROM, SRAM και off-chip DRAM μνήμες για το την έκδοση v1 με buffers

Επιπλέον λειτουργίες (optimized-3-v2)

Έχουν προστεθεί μερικά κομμάτια κώδικα που υλοποιούν τις παρακάτω λειτουργίες:

1. Επιλογή Εικόνας από μιά ποικιλία εικόνων
2. Έξοδος μόνο αν το επιλέξει ο χρήστης.
3. Μπορεί να τρέξει για οποιοδήποτε μέγεθος εικόνας μέχρι το 498 x 372 αν δηλωθεί στον κώδικα πριν την εκτέλεση του.
4. Μπορεί να τρέξει για εικόνες YUV 420, 444 ή binary.

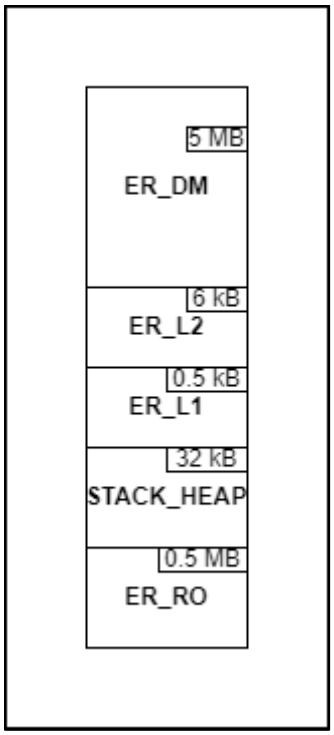
Ο χρήστης διαλέγει έναν αριθμό από το 0 έως το 13, με το 0 να είναι η έξοδος και το 1 έως το 12 οι 12 διαφορετικές εικόνες που μπορεί να εφαρμόσει πάνω τους τον αλγόριθμο.

Κάνοντας CMake την εικόνα .axf έχουμε τις παρακάτω μετρήσεις:

Object	6272	168	20	5221136	10368
Library	20808	738	0	300	8196
Grand	27080	906	20	5221436	18564
RO	27986	27.33007813	kB		
RW	5221456	4.979568481	MB		
ROM	28006	27.34960938	kB		

Πίνακας 19 - Μετρήσεις αντικειμένων στην εικόνα .axf της έκδοσης v2

Η δομή μνήμης που προτείνουμε είναι:



Εικόνα 8 - Δομή της Μνήμης της έκδοσης v3 με buffers και τις επιπλέον λειτουργίες

Χρήση Δεικτών (optimized-3-v3)

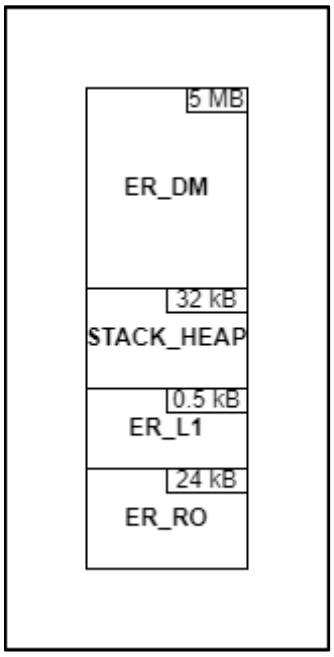
Μια εναλλακτική προσέγγιση του αλγορίθμου είναι να χρησιμοποιήσουμε δείκτες. Θεωρητικά και σε αυτή την προσέγγιση και στις προηγούμενες έπρεπε να υπολογιστεί η διεύθυνση μνήμης που θα αποθηκευτούν τα δεδομένα ή από τις θέσεις που θα χρειαστούμε δεδομένα για τους υπολογισμούς.

Κάνοντας CMake την εικόνα .axf έχουμε τις παρακάτω μετρήσεις:

Object	5784	212	60	5215140	10584
Library	17392	466	0	300	7080
Grand	23176	678	60	5215440	17664
RO	23854	23.29492188	kB		
RW	5215500	4.973888397	MB		
ROM	23914	23.35351563	kB		

Πίνακας 20 - Μετρήσεις αντικειμένων στην εικόνα .axf της έκδοσης v3

Η δομή μνήμης που προτείνουμε είναι:



Εικόνα 9 - Δομή της Μνήμης της έκδοσης v3 χωρίς buffers και χρήση δεικτών

Δομή της μνήμης - Μετρήσεις Μνήμης

Ο linker μας δίνει την δυνατότητα στα αρχεία scatter να δηλώσουμε είτε μεμωνομένες μεταβλητές είτε ολόκληρες συναρτήσεις μέσα σε συγκεκριμένους τομείς της μνήμης. Η διαβάθμηση που επιλέξαμε βασίζεται σε μια μνήμη ROM για την συμπιεσμένη εικόνα .axf, μια μνήμη που περιλαμβάνει τα δεδομένα μόνο για ανάγνωση, μια μνήμη L1 Cache που είναι μια περιοχή με κάποιες global μεταβλητές και κάποια .bss αρχεία, μια περιοχή μνήμης που την ονομάσαμε L2 Cache και διαθέτει τους buffers και τέλος μια περιοχή μνήμης Data Memory όπου βρίσκονται τα Zero Initialized Data των Πινάκων. Το memory.map αρχείο αυτής της διάταξης ακολουθεί μια ιεραχείαστην οποία όσο μεγαλύτερη η περιοχή μνήμης

τόσο και πιο αργή. Οπότε η πιο αργή περιοχή είναι η Data Memory και ακολουθούν οι L2 Cache, L1 Cache, ROM και LOAD ROM, με την τελευταία να είναι η πιο γρήγορη.

Επιπλέον έχει χρησιμοποιηθεί ξεχωριστή περιοχή μνήμης για την stack και την heap στις εκδόσεις v2 και v3.

Οι μετρήσεις της μνήμης φαίνονται αναλυτικά στον παρακάτω πίνακα:

Base	Hex	From scatter.txt :		From .map file :									
project	name	start	limit	Nread_ns	Nwrite_ns	Sread_ns	Swrite_ns	Nreads	Nwrites	Sreads	Swrites	ns	s
cw-optimized-3-v1	ROM	0	58000	1	1	1	1	3F5C70E	0	1867A76F	0	1EDB87C4	9
	SRAM	58000	700	7D	7D	19	19	90A478	B476F6	C54806	F92B74	921E338	4
	DRAM	58700	500000	7D	7D	19	19	71278B	43E148	4	59	276617EC	1
cw-optimized-3-v2	ROM	0	6E00	1	1	1	1	2F8F968	0	23686EF	2BCAD34	276617EC	1D
	SRAM	58000	8000	1	1	1	1	C34C98	FCF737	14E31E0	18F5E35	20B9C3D0	1
	SRAM	60000	200	32	32	D	D	7E47B4	317A84	10D8C8	B3DB7	2B63470C	0
	DRAM	60200	1800	64	64	19	19	3B2B60	110778	0	0	1DC3DC60	0
	DRAM	62000	500000	96	96	32	32	294C9B	301610	0	0	37DDAAE0	0
cw-optimized-3-v3	ROM	0	5D90	1	1	1	1	3574C96	0	E2FB978	0	34868718	5
	ROM	0	5D90	1	1	1	1	0	0	0	0	0	0
	SRAM	5D90	1AC	1	1	1	1	EBACAC	372723	E18A0	872C2	187B6FD4	0
	SRAM	5F3C	8000	A	A	5	5	7444AF	B507D4	A9F76F	F08503	374BB324	0
	DRAM	DF3C	4F9360	64	64	32	32	4C0DF8	32E6D0	0	0	31979E20	0

Πίνακας 21 - Μετρήσεις Μνήμης για τις τρεις εκδόσεις

Οι μετρήσεις των χρόνων εκτέλεσης είναι :

Base Dec

project	name	clock (s)	cputime (in clock units)	sys_clock
cw-optimized-3-v1	ROM SRAM DRAM	16.571789	828589496	1657
cw-optimized-3-v2	ROM SRAM SRAM DRAM DRAM	1.407247	1125799188	2251
cw-optimized-3-v3	ROM SRAM SRAM SRAM *DRAM*	8.94913	447456546	894

Πίνακας 22 - Μετρήσεις Χρόνων από την εκτέλεση των τριών εκδόσεων

Συμπεράσματα

Συγκρίνοντας την αρχική υλοποίηση του αλγορίθμου με την τελική υπάρχει διαφορά στο πόσο εύκολα διαβάζεται ο κώδικας και στο πόσο περίπλοκος είναι. Όμως το πλήθος των εντολών έχει πέσει γύρω στο 50% και αυτό είναι ένας από τους αρχικούς στόχους. Το πλήθος των κύκλων αναμονής έχει επίσης μειωθεί δραματικά, πάνω από 50%. Αυτός ήταν ο δεύτερος στόχος της εργασίας αυτής. Και ο τρίτος στόχος ήταν να σπάσουμε να τα δεδομένα σε κομμάτια ώστε ακόμα και αν η μνήμη είναι αργή να τα έχουμε πιο κοντά στον επεξεργαστή μειώνοντας την αναμονή λόγω αργής μνήμης. Οι συνολικές μετρήσεις όπου συγκρίνονται όλες οι διαφορετικές εκδοχές υλοποίησης και μνήμης φαίνονται στους παρακάτω πίνακες:

NO MEMORY	Instuctions	Core Cycles	S Cycles	N Cycles	I Cycles	C Cycles	Wait Cycles	Total	True Idle Cycles
initial	620,226,123	102,298,801	728,855,498	232,897,531	120,254,198	0	0	1,082,007,227	0
optimized-1	844,442,376	1,287,663,527	974,504,597	243,573,763	167,334,313	0	0	1,385,412,673	0
optimized-2-v3	470,994,406	694,339,163	534,629,122	123,891,439	84,076,059	0	0	1,357,023,856	0
optimized-3-v1	377,499,895	555,538,749	434,852,992	95,472,978	61,677,728	0	0	592,003,698	0

Πίνακας 23 - Πλήθος εντολών ανάλογα το είδος της εντολής. Η μνήμη είναι η μέγιστη επιτρεπτή

WITH MEMORY	Instuctions	Core Cycles	S Cycles	N Cycles	I Cycles	C Cycles	Wait Cycles	Total	True Idle Cycles
initial	620,226,214	1,022,989,014	728,855,589	232,897,632	120,254,219	0	1,240,908,894	2,322,916,334	35,284,909
optimized-1	844,024,822	1,286,099,875	973,524,658	243,016,796	168,038,328	0	1,291,521,998	2,676,101,780	51,433,781
optimized-2-v1	560,637,052	857,471,806	649,000,191	163,202,201	94,488,489	0	814,989,422	1,721,680,303	27,328,414
optimized-2-v2	560,637,052	857,471,806	649,000,191	163,202,201	94,488,489	0	814,989,422	1,721,680,303	27,328,414
optimized-2-v3	470,994,406	694,339,163	534,629,122	123,891,439	84,076,059	0	614,427,236	1,357,023,856	25,006,719
optimized-2-v3-sc2	470,994,406	694,339,163	534,629,122	123,891,439	84,076,059	0	0	1,357,023,856	25,006,719
optimized-3-v1	381,483,784	563,899,854	438,704,966	99,609,423	61,993,414	0	228,281,693	828,589,496	19,633,046
optimized-3-v2	653,306,711	930,738,307	736,021,782	147,543,646	158,215,971	0	84,017,789	1,125,799,188	58,705,205
optimized-3-v3	223,667,396	404,578,018	266,387,788	102,916,528	44,871,430	0	33,280,800	447,456,546	5,548,129

Πίνακας 24 - Πλήθος Εντολών ανάλογα το είδος. Οι κώδικες έχουν την μνήμη που προτείνουμε

	ROM	RAM	Total MB

	Start Address	End Address	MB	Start Address	Heap Base	Stack Base	End Address	MB	
initial	0x00000000	0x04000000	64 MB	0x04000000	0x06000000	0x08000000	0x10000000	192	256
optimized-1	0x00000000	0x01000000	16 MB	0x01000000	0x01000000	0x02000000	0x04000000	48	64
optimized-2-v1	0x00000000	0x01000000	16 MB	0x01000000	0x01000000	0x02000000	0x04000000	48	64
optimized-2-v2	0x00000000	0x01000000	16 MB	0x01000000	0x01000000	0x02000000	0x04000000	48	64
optimized-2-v3	0x00000000	0x00800000	8 MB	0x00800000	0x00800000	0x01000000	0x02000000	24	32
optimized-3-v1	0x00000000	0x00058000	352 KB	0x00058000	0x00058000	0x00058700	0x00500000	4.7	5
optimized-3-v2	0x00000000	0x0006E000	0.5 MB	0x0006E000	0x00058000	0x00060000	0x00500000	5.1	5.6
optimized-3-v2	0x00000000	0x00005D90	24 KB	0x00005D90	0x00005F3C	0x0000DF3C	0x004F9360	5.1	5.3

Πίνακας 25 - Μεγέθη μνημών ανά έκδοση κώδικα

Base Dec

WITH MEMORY	clock (us)	cputime (in clock units)	sys_clock
initial	46.458326	2322916334	4645
optimized-1	53.522035	2676101780	5352
optimized-2-v1-mem-3	26.283711	1314185592	2628
optimized-2-v2-mem-3	26.283711	1314185592	2628
optimized-2-v3-mem-3	20.996204	1049810238	2099
optimized-2-v3-mem-3-sc2	14.85193	742596504	1485
optimized-3-v1	16.571789	828589496	1657
optimized-3-v2	1.407247	1125799188	2251
optimized-3-v3	8.94913	447456546	894

Πίνακας 26 - Χρόνοι Εκτέλεσης όλων των εκδόσεων

Έχουμε πετύχει τέσσερα πράγματα:

1. Μείωση του όγκου της μνήμης από 256 MB σε 5.5 MB (97%)
2. Μείωση του πλήθους πράξεων από 620,226,214 σε 223,667,396 (64%)
3. Μείωση της αναμονής λόγω αργής μνήμης από 1,240,908,894 κύκλους σε 33,280,800 κύκλους (97%)
4. Μείωση του χρόνου εκτέλεσης από 46.5 μs σε 8.9 μs (81%)

Τελικά πιο πολύ ρόλο στην καθυστέρηση μνήμης παίζουν οι χρόνοι ανάνγωσης/ εγγραφής αλλά και το πλήθος των δεδομένων.
Το πλήθος των εντολών μπορεί να μειωθεί σε ορισμένες περιπτώσεις αλλά αν η υλοποίηση δεν βοηθά, δεν θα υπάρχει αισθητή αλλαγή όπως έγινε από την αρχική υλοποίηση στην optimized-1 έκδοση.
Η χρήση buffers είναι μια καλή στρατηγική που βολεύει πολύ για τα φίλτρα και μεγάλους πίνακες αλλά σε περιπτώσεις όπου υπάρχει άμεση εξάρτηση μεταξύ των δεδομένων που αποτρέπει να γίνουν οι επόμενοι υπολογισμοί δεν μπορεί η τεχνική αυτή να χρησιμοποιηθεί όπως στην ανάγνωση ή την εγγραφή της εικόνας ή στον υπολογισμό της εικόνας γωνίας

από τα φίλτρα sobel x και y όπου πρέπει να υπολογιστούν οι τιμές και μετά να υπολογιστεί η γωνία. Ακόμα και με τους buffers η καθυστέρηση είναι αισθητή.
Η δομή της μνήμης που προτείναμε ήταν χωρισμένη σε δύο κομμάτια ROM και RAM. Αν σε φυσικό επίπεδο (physical memory) μπορεί να είναι μια μνήμη ή μπορεί να είναι περισσότερες πχ δύο DRAM ή δύο ROM. Αυτό δεν μας απασχολεί εφόσον οι διευθύνσεις είναι όπως τις έχουμε δηλώσει στα αρχεία scatter.txt.
Η stack και η heap είχαν μέγαλο όγκο 32 KB αλλά ακόμα και έτσι πιο πολύ χώρο έπιαναν οι πίνακες παρά τα υπόλοιπα δεδομένα. Αν μπορούσαμε να είχαμε μια μεγαλή μνήμη κοντά στον επεξεργαστή και πολύ γρήγορη δεν θα είχαμε κανένα πρόβλημα. Για να προσεγγήσουμε αυτή την ιδεατή κατάσταση χρησιμοποιήσαμε τεχνικές όπως οι buffers, επαναχρησιμοποίηση των ίδιων πινάκων και προσπαθήσαμε να κρατάμε όσα περισσότερα δεδομένα γίνεται στις μνήμες Cache L1, L2 αντί να τα στέλνουμε στην αργή μνήμη. Ακόμα με τις τεχνικές loop unrolling, loop merging καταφέραμε να εκτελούμε περισσότερες εντολές πριν μεταβούμε σε επόμενο βρόχο.

Αναφορές

1. Engineering A Compiler by Keith D. Cooper and Linda Torczon, 3rd Edition, Elsevier publications, 2023
2. Digital Image Processing by R. Gonzalez and R. Woods, 4th Edition, Pearson, 2018
3. Digital Video and HD - Algorithms and Interfaces by Charles Poynton, 2nd Edition, Elsevier publications, 2012
4. Compilers: Principles, Techniques and Tools by Alfred V. Aho, Monica S. Lam, Ravi Sethi and Jeffrey D. Ullman, 2nd Edition, Addison-Wesley, 2006
5. Best Practice Guide - Deep Learning - Scientific Figure on ResearchGate. Available from: https://www.researchgate.net/figure/Schematic-illustration-of-a-convolutional-operation-The-convolutional-kernel-shifts-over_fig2_332190148 [accessed 26 Nov, 2023]
6. Computer Organization and Architecture – Design for Performance by William Stallings, 11th Edition, Pearson, 2019
7. Embedded System Design – Embedded Systems Foundations of Cyber-Physical Systems and the Internet of Things by Peter Marwedel, 4th Edition, Springer, 2021
8. ARM System Developer's Guide Designing and Optimizing System Software by Andrew N. Sloss, Dominic Symes, Chris Wright, Elsevier publications, 2004