

ΣΧΕΔΙΑΣΜΟΣ ΕΝΣΩΜΑΤΩΜΕΝΩΝ ΣΥΣΤΗΜΑΤΩΝ

1η ΕΡΓΑΣΙΑ

**Καλιωράκης Γιώργος ΑΜ:57069
Μερτζάνης Κωνσταντίνος ΑΜ:57131**

ΟΜΑΔΑ 45

ΠΕΡΙΕΧΟΜΕΝΑ

ΕΙΣΑΓΩΓΗ	3
Χρήση Λαπλασιανού φίλτρου για βελτίωση των ακμών σε μια εικόνα	3
ΠΙΝΑΚΑΣ ΔΕΔΟΜΕΝΩΝ.....	5
ΜΕΤΑΣΧΗΜΑΤΙΣΜΟΙ ΒΡΟΓΧΩΝ ΓΙΑ ΤΗΝ ΒΕΛΤΙΣΤΟΠΟΙΗΣΗ ΤΟΥ ΑΛΓΟΡΙΘΜΟΥ.....	5
1 ^η μέθοδος Loop Unrolling.....	5
2 ^η μέθοδος Loop Fission and Fusion.....	6
3 ^η μέθοδος Loop Interchange.....	8
4 ^η μέθοδος Loop Tiling.....	10
5 ^η μέθοδος Loop Collapsing.....	14
6 ^η μέθοδος Loop Inversion.....	15
UNSHARP MASKING.....	17
ΠΗΓΕΣ.....	18

ΕΙΣΑΓΩΓΗ

Η εργασία που ακολουθεί έχει ως στόχο της την βελτίωση των ακμών στην επεξεργασία μίας εικόνας. Σκοπός μας ήταν η σύνταξη ενός κώδικα σε γλώσσα C, ο οποίος θα υλοποιεί την παραπάνω λειτουργία(βελτίωση ακμών) με τη χρήση ενός Λαπλασιανού φίλτρου. Σε αυτόν έχουν εφαρμοστεί οι μετασχηματισμοί βρόχων για την κανονικοποίηση της δομής του αλλά και οποιοσδήποτε άλλος μετασχηματισμός θα μπορούσε να επιφέρει τη βελτίωση της απόδοσης του.

Στη συνέχεια θα γίνει μια αναφορά στη λειτουργία και την χρησιμότητα του φίλτρου ενώ θα παρουσιαστεί σε κάθε μία από τις περιπτώσεις βελτιστοποίησης τόσο το θεωρητικό υπόβαθρο όσο και τα αποτελέσματα που θα εξάγουμε από το Code Warrior και το AXD Debugger καθώς επίσης και ένας σχολιασμός για το κατά πόσο βελτιστοποιήθηκε ο κώδικάς μας ή όχι.

Χρήση Λαπλασιανού φίλτρου για βελτίωση των ακμών σε μια εικόνα

Η βελτίωση εικόνας είναι συνήθως μία διαδικασία φιλτραρίσματος δηλ. συνέλιξης με συγκεκριμένη δισδιάστατη μάσκα και στοχεύει στην ανάδειξη χαρακτηριστικών ή στην ελάττωση θορύβου και άλλων ανεπιθύμητων χαρακτηριστικών. Στη διαδικασία βελτίωσης εικόνας το αποτέλεσμα είναι επίσης εικόνα και όχι κάποιο χαρακτηριστικό. Επομένως το φιλτράρισμα εικόνας είναι ουσιαστικά η πράξη συνέλιξης μεταξύ της αρχικής εικόνας και ενός συνόλου συντελεστών.

Η βελτίωση των ακμών, πραγματοποιείται με ένα φίλτρο επεξεργασίας εικόνας, το οποίο ενισχύει τις αντιθέσεις στα σημεία που εμφανίζονται ακμές σε μια εικόνα. Το φίλτρο Laplace είναι ένα ιστροπικό χωρικό φίλτρο το οποίο χρησιμοποιείται για την ανίχνευση ξαφνικών μεταβάσεων έντασης σε μια εικόνα και την επισήμανση των ακμών της σε αυτή την περίπτωση. Ουσιαστικά πραγματοποιεί συνέλιξη μιας εικόνας με μια μάσκα :

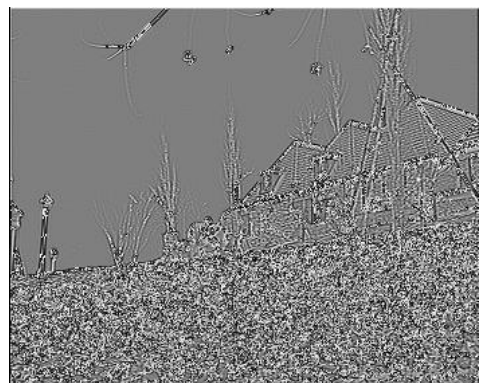
Int Laplacianfilter = {0, 1, 0},

{1,-4, 1},

{0, 1, 0}; και λειτουργεί ως

ανιχνευτής των pixel των ακμών. Το φίλτρο λειτουργεί αναγνωρίζοντας τις «αιχμηρές» άκμές στην εικόνα, όπως πχ είναι η ακμή μεταξύ ενός αντικειμένου και ενός φόντου με αντίθεση χρώματος, και αυξάνοντας την αντίθεση της εικόνας στην περιοχή αμέσως γύρω από την ακμή. Αυτό έχει ως αποτέλεσμα τη δημιουργία λεπτών φωτεινών και σκοτεινών τόνων σε κάθε πλευρά οποιονδήποτε άκρων της εικόνας.

Η εικόνα που επεξεργαστήκαμε είναι η παρακάτω. Δεξιά της παρατίθεται το αποτέλεσμα που λάβαμε μετά την πρώτη εφαρμογή του φίλτρου σε αυτήν. Η ανάδειξη των ακμών είναι κάτι παραπάνω από εμφανής.



Αποτελέσματα του code warrior για τον αρχικό κώδικα (FILTER.c):

Image component sizes						
	Code	RO Data	RW Data	ZI Data	Debug	
	856	60	0	1226792	4660	Object Totals
	10196	314	0	300	4548	Library Totals
=====						
	Code	RO Data	RW Data	ZI Data	Debug	
	11052	374	0	1227092	9208	Grand Totals
=====						
	Total RO	Size(Code + RO Data)			11426 (11.16kB)	
	Total RW	Size(RW Data + ZI Data)			1227092 (1198.33kB)	
	Total ROM	Size(Code + RO Data + RW Data)			11426 (11.16kB)	
=====						

Code: Πρόκειται για τον κώδικά,ο οποίος καταλαμβάνει έναν συγκεκριμένο χώρο στη μνήμη.

RO data (Read Only data): Δεδομένα των οποίων την τιμή δεν μπορώ να μεταβάλλω. Για παράδειγμα οι σταθερές. Δηλαδή

```
#define N 288 /* frame dimension for QCIF format */
#define M 352 /* frame dimension for QCIF format */
#define filename "flower_cif_150_yuv444.yuv"
#define file_y "akiyo2y.yuv"
```

RW data (Read/ Write data): Δεδομένα τα οποία μπορώ να διαβάσω και να τους εκχωρήσω τιμή.

ZI data (Zero Initialized data):Οι global μεταβλητές. Τα δεδομένα τα οποία είναι έξω από τη main. Αυτά με το που «κατέβουν» στον μικροεπεξεργαστή θα πάρουν τιμή μηδέν. Για παράδειγμα

```
int current_y[N][M];
int A[N+2][M+2];
int newA[N+2][M+2];
```

Debugger Internals Statistics:

Internal Variables		Statistics						
Reference	Points	Instructions	Core_Cycles	S_Cycles	N_Cycles	I_Cycles	C_Cycles	Total
\$statistics		11016040	21008839	10510699	8054628	2670721	0	21236048

- Core_Cycles: Οι κύκλοι στον επεξεργαστή.
- S_Cycles (Sequential Cycles): Ο αριθμός των κύκλων στους οποίους προχωράμε σειριακά.
- N_Cycles (Non-Sequential Cycles): Πλήθος κύκλων που χρειάζεται να μην εκτελεστεί σειριακά η εντολή (π.χ. έχω μια διακλάδωση).

- C_Cycles: Οι κύκλοι στον συνεπεξεργαστή (π.χ. κάρτα γραφικών γενικού σκοπού)

ΠΙΝΑΚΑΣ ΔΕΔΟΜΕΝΩΝ

Ο πίνακας που προκύπτει με τα επεξεργασμένα δεδομένα είναι ο `current_y` και θα έχει διαστάσεις ίδιες με την εικόνα που εισαγάγαμε δηλαδή 288×352 με $N=288$ και $M=352$. Εφόσον τα δεδομένα είναι τύπου `integer` και κάθε ακέραιος έχει μέγεθος 4 Bytes το μέγεθος του πίνακα θα είναι

$$A = M \cdot N \cdot 4 \text{ Bytes} = 405504 \text{ Bytes}$$

Σε όλους τους κώδικες παρατηρούμε ότι κάνουμε αρκετές προσπελάσεις στον τελικό πίνακα `current_y`. Συγκεκριμένα οι προσπελάσεις αυτές στον πίνακα είναι 4:

- Όταν εκχωρούνται τιμές στον πίνακα μέσω της `fgetc()`,
- Όταν θέτονται τιμές από τον πίνακα `current_y` στον πίνακα `A`,
- Όταν γίνεται κλιμάκωση
- Όταν λαμβάνονται τιμές από τον πίνακα με την `fputc()`.

Άρα ο συνολικός αριθμός προσπελάσεων στον πίνακα `current_y` θα είναι $4 \cdot M \cdot N = 405504$.

Στις περιπτώσεις των `loop fission` και `loop tiling` παρατηρούμε όμως ότι προσπελαύνουμε τον πίνακα `current_y` 7 και 7 φορές αντίστοιχα άρα οι συνολικές προσπελάσεις στον πίνακα θα είναι $7 \cdot M \cdot N = 709632$ και στις 2 περιπτώσεις.

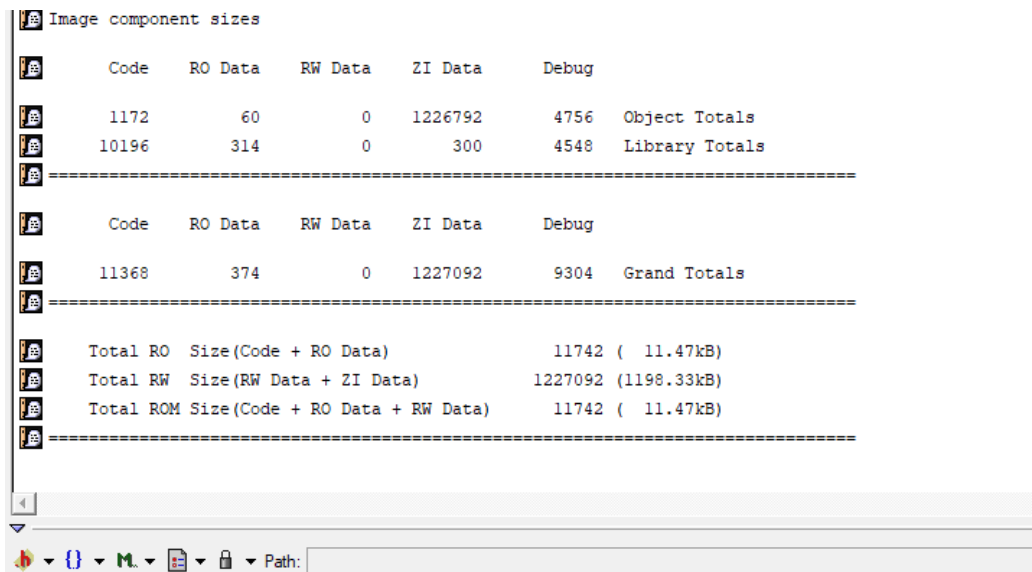
ΜΕΤΑΣΧΗΜΑΤΙΣΜΟΙ ΒΡΟΓΧΩΝ ΓΙΑ ΤΗΝ ΒΕΛΤΙΣΤΟΠΟΙΗΣΗ ΤΟΥ ΑΛΓΟΡΙΘΜΟΥ

1^η μέθοδος Loop Unrolling

Η `loop unrolling` είναι μια τεχνική μετασχηματισμού βρόχου που επιχειρεί να βελτιστοποιήσει την ταχύτητα εκτέλεσης του προγράμματος σε βάρος του δυαδικού μεγέθους του, που είναι μια προσέγγιση γνωστή ως χωροχρονική ανταλλαγή. Ο μετασχηματισμός μπορεί να γίνει χειροκίνητα από τον προγραμματιστή ή από έναν μεταγλωττιστή βελτιστοποίησης. Ο στόχος της χαλάρωσης του βρόχου είναι να αυξήσει την ταχύτητα του προγράμματος μειώνοντας ή εξαλείφοντας τις οδηγίες που ελέγχουν τον βρόχο, όπως είναι οι αριθμητικές μετρήσεις των δεικτών και οι δοκιμές "τέλος του βρόχου" σε κάθε επανάληψη καθώς και την απόκρυψη καθυστερήσεων, συμπεριλαμβανομένης της καθυστέρησης στην ανάγνωση δεδομένων από τη μνήμη. Για να εξαλειφθούν αυτά, οι βρόχοι μπορούν να ανατυπωθούν ως μια επαναλαμβανόμενη ακολουθία παρόμοιων ανεξάρτητων δηλώσεων. Η απελευθέρωση του βρόχου αποτελεί επίσης μέρος ορισμένων τυπικών τεχνικών επαλήθευσης, ιδίως των περιορισμένων ελέγχων μοντέλων.

Θέλει ιδιαίτερη προσοχή στη χρήση της συγκεκριμένης μεθόδου, έτσι ώστε να μην αυξηθεί «επικίνδυνα» το μέγεθος του κώδικα και καταλάβει χώρο στη μνήμη σημαντικό για άλλα δεδομένα.

Τα αποτελέσματα που πήραμε από το code warrior είναι τα παρακατω



Code	RO Data	RW Data	ZI Data	Debug	
1172	60	0	1226792	4756	Object Totals
10196	314	0	300	4548	Library Totals
=====					
Code	RO Data	RW Data	ZI Data	Debug	
11368	374	0	1227092	9304	Grand Totals
=====					
Total RO	Size(Code + RO Data)			11742 (11.47kB)	
Total RW	Size(RW Data + ZI Data)			1227092 (1198.33kB)	
Total ROM	Size(Code + RO Data + RW Data)			11742 (11.47kB)	
=====					

Παρατηρούμε πως τα Byte του κώδικα εμφανώς αυξήθηκαν κατά 324 Bytes, από 856 βλέπουμε να είναι 1172.

Internal Variables Statistics							
Reference Points	Instructions	Core Cycles	S_Cycles	N_Cycles	I_Cycles	C_Cycles	Total
\$statistics	9368079	17980587	8698324	7143987	2365485	0	18207796

Παρατηρούμε πως οι συνολικοί κύκλοι έχουν ελαττωθεί κατά 14%.

Έγινε αυτό που περιμέναμε και από το θεωρητικό σκέλος, δηλαδή να αυξηθεί το μέγεθος του κώδικα καθώς σπάσαμε βρόγχους επανάληψης σε πολλούς μικρότερους αλλά ταυτόχρονα πετύχαμε τη βελτιστοποίηση, δηλαδή τη μείωση των συνολικών κύκλων γιατί λόγω των πολλών επαναλήψεων χρειάστηκαν λιγότεροι έλεγχοι.

2^η μέθοδος Loop Fission and Fusion

Στην επιστήμη των υπολογιστών, η loop fission είναι μια βελτιστοποίηση του compiler στον οποίο ένας βρόχος διασπάται σε πολλαπλούς βρόχους πάνω από το ίδιο εύρος δείκτη, με το καθένα να παίρνει μόνο ένα μέρος του σώματος του αρχικού βρόχου. Ο στόχος είναι να σπάσει ένα μεγάλο σώμα βρόχου σε μικρότερα ώστε να επιτευχθεί καλύτερη αξιοποίηση της τοπικότητας αναφοράς. Αυτή η βελτιστοποίηση είναι πιο αποτελεσματική σε επεξεργαστές πολλαπλών πυρήνων που μπορούν να χωρίσουν μια εργασία σε πολλαπλές εργασίες για κάθε επεξεργαστή.

Αντιστρόφως, η loop fusion είναι μια βελτιστοποίηση μεταγλωττιστή και μετασχηματισμός βρόχου που αντικαθιστά πολλαπλούς βρόχους με ένα μόνο. Είναι δυνατό όταν δύο βρόχοι επαναλαμβάνονται στο ίδιο εύρος και δεν αναφέρονται τα δεδομένα του άλλου. Η σύντηξη βρόχου δεν βελτιώνει πάντα την ταχύτητα εκτέλεσης. Σε ορισμένες αρχιτεκτονικές, δύο βρόχοι μπορεί να έχουν καλύτερη απόδοση από έναν βρόχο επειδή, για παράδειγμα, υπάρχει αυξημένη τοποθεσία δεδομένων σε κάθε βρόχο.

Τα αποτελέσματα που πήραμε τρέχοντας τον κώδικα στο code warrior είναι τα παρακάτω

1) Loop Fission

Image component sizes						
	Code	RO Data	RW Data	ZI Data	Debug	
	1056	60	0	1226792	4788	Object Totals
	10196	314	0	300	4548	Library Totals
=====						
	Code	RO Data	RW Data	ZI Data	Debug	
	11252	374	0	1227092	9336	Grand Totals
=====						
	Total RO	Size(Code + RO Data)			11626	(11.35kB)
	Total RW	Size(RW Data + ZI Data)			1227092	(1198.33kB)
	Total ROM	Size(Code + RO Data + RW Data)			11626	(11.35kB)
=====						

2) Loop Fusion

Image component sizes						
	Code	RO Data	RW Data	ZI Data	Debug	
	752	60	0	1226792	4660	Object Totals
	10196	314	0	300	4548	Library Totals
=====						
	Code	RO Data	RW Data	ZI Data	Debug	
	10948	374	0	1227092	9208	Grand Totals
=====						
	Total RO	Size(Code + RO Data)			11322	(11.06kB)
	Total RW	Size(RW Data + ZI Data)			1227092	(1198.33kB)
	Total ROM	Size(Code + RO Data + RW Data)			11322	(11.06kB)
=====						

Παρατηρούμε ότι στο loop fission το μέγεθος του κώδικα αυξάνεται από 856 σε 1056 Bytes ενώ στο loop fusion μειώνεται από 856 σε 752 Byte. Το εξαγόμενο της παρατήρησης του αποτελέσματος είναι απολύτως φυσιολογικό καθώς περιμένουμε σε μέθοδο(loop fission) η οποία διασπά τους βρογχους σε περισσότερους να προκύψει μεγαλύτερου μεγέθους κώδικα όπως και το αντιστροφο στην άλλη μέθοδο(loop fusion),δηλαδή να προκύψει μικρότερου μεγέθους.

Τα αποτελέσματα από το Debugger είναι τα παρακάτω

1) Loop Fission

Internal Variables		Statistics					
Reference Points	Instructions	Core_Cycles	S_Cycles	N_Cycles	I_Cycles	C_Cycles	Total
\$statistics	12654451	24362477	12126440	9386732	3077474	0	24590646

2) Loop Fusion

Internal Variables		Statistics					
Reference Points	Instructions	Core_Cycles	S_Cycles	N_Cycles	I_Cycles	C_Cycles	Total
\$statistics	10118174	19092595	10001059	6951154	2366952	0	19319165

Παρατηρούμε ότι οι συνολικοί κύκλοι στην πρώτη περίπτωση αυξήθηκαν κατά 15,8%, ενώ μειώθηκαν κατά 9% ενώ στην δεύτερη περίπτωση

Loop Fission

Η sram αποτελεί την γέφυρα μεταξύ της RAM και της CPU, στην οποία εκχωρείται μια μικρή ποσότητα δεδομένων της RAM και λόγω της πολύ υψηλότερης ταχύτητας της κρυφής μνήμης, μειώνεται το χάσμα στην ταχύτητα επεξεργασίας δεδομένων μεταξύ της CPU και της RAM. Λόγω της μικρής χωρητικότητας της cache, τα δεδομένα που δέχεται από την RAM είναι περιορισμένου μεγέθους, ενώ ταυτόχρονα έχει το πλεονέκτημα της άμεσης προώθησης δεδομένων που «θυμάται», δηλαδή που έχουν ξαναχρησιμοποιηθεί. Επομένως για να αποτελέσει εργαλείο στην μείωση των κύκλων πρέπει ο κώδικάς μας να συντάσσεται με βάση την επαναχρησιμοποίηση δεδομένων. Στην συγκεκριμένη περίπτωση δεν συμβαίνει κάτι τέτοιο καθώς ένα στοιχείο του πίνακα (current_y) θα επεξεργαστεί/προσπελαστεί σε διαφορετικά χρονικά σημεία λόγω του σπασίματος των επαναλήψεων και στη συνέχεια θα πρέπει να ανακτηθεί εκ νέου από την κρυφή μνήμη, καθώς σε αυτήν θα βρίσκεται πλέον άλλο στοιχείο.

Loop Fusion

Η μεγάλη ελάττωση του πλήθους των εντολών στον επεξεργαστή, ενδέχεται να αποτελεί την αιτία της μείωσης των συνολικών κύκλων κατά την υλοποίηση αυτής της μεθόδου βελτιστοποίησης. Επίσης λόγω της συνένωσης βρόχων κοινών ορίων, με το επακόλουθο της επεξεργασίας /προσπέλασης ενός στοιχείου του πίνακα πολλές φορές χωρίς αυτό να μεταβάλλεται, ενισχύεται η συνεισφορά της sram. Αυτό σημαίνει ότι το στοιχείο που υπόκειται σε επεξεργασία, επαναχρησιμοποιείται συνεχώς στον ίδιο βρόχο με αποτέλεσμα η προώθησή του από την cache στην CPU να είναι άμεση και αρκετά ταχύτερη από την περίπτωση όπου η μεταβλητή χάνεται και πρέπει να ανακτηθεί ξανά από την RAM. Η τοπικότητα αναφοράς της cache χρησιμοποιείται ευεργετικά, σε αντίθεση με την προηγούμενη περίπτωση.

3^η μέθοδος Loop Interchange

Loop Interchange(ανταλλαγή βρόχων) είναι η διαδικασία ανταλλαγής της σειράς των δύο μεταβλητών επανάληψης που χρησιμοποιούνται από έναν ένθετο βρόχο. Η μεταβλητή που χρησιμοποιείται στον εσωτερικό βρόχο μεταβαίνει στον εξωτερικό βρόχο και αντίστροφα. Συχνά

γίνεται για να διασφαλιστεί ότι τα στοιχεία μιας πολυδιάστατης συστοιχίας είναι προσβάσιμα με τη σειρά με την οποία βρίσκονται στη μνήμη, βελτιώνοντας την τοπικότητα αναφοράς.

Ο κύριος σκοπός της ανταλλαγής βρόχων είναι να επωφεληθεί από την CPU cache κατά την πρόσβαση σε στοιχεία συστοιχίας. Όταν ένας επεξεργαστής αποκτήσει πρόσβαση για ένα στοιχείο πίνακα για πρώτη φορά, θα ανακτήσει ένα ολόκληρο μπλοκ δεδομένων από τη μνήμη στην προσωρινή μνήμη. Αυτό το μπλοκ είναι πιθανό να έχει πολλά περισσότερα διαδοχικά στοιχεία μετά την πρώτη, έτσι ώστε στην επόμενη πρόσβαση στο στοιχείο του πίνακα, θα οδηγηθεί απευθείας από την κρυφή μνήμη (η οποία είναι ταχύτερη από την απόκτηση της αργής κύριας μνήμης). Η αποτελεσματικότητα της loop interchange εξαρτάται και πρέπει να εξεταστεί υπό το φως του μοντέλου κρυφής μνήμης που χρησιμοποιείται από το hardware και το μοντέλο συστοιχίας που χρησιμοποιείται από τον compiler.

Τα αποτελέσματα από το code warrior είναι τα παρακάτω

Image component sizes						
	Code	RO Data	RW Data	ZI Data	Debug	
	836	60	0	1226792	4700	Object Totals
	10196	314	0	300	4548	Library Totals
=====						
	Code	RO Data	RW Data	ZI Data	Debug	
	11032	374	0	1227092	9248	Grand Totals
=====						
	Total RO	Size(Code + RO Data)			11406 (11.14kB)	
	Total RW	Size(RW Data + ZI Data)			1227092 (1198.33kB)	
	Total ROM	Size(Code + RO Data + RW Data)			11406 (11.14kB)	
=====						

Παρατηρούμε ότι ο κώδικας έχει μικρύνει ελάχιστα κατά 20 Bytes.

Τα αποτελέσματα από το Debugger είναι παρακάτω

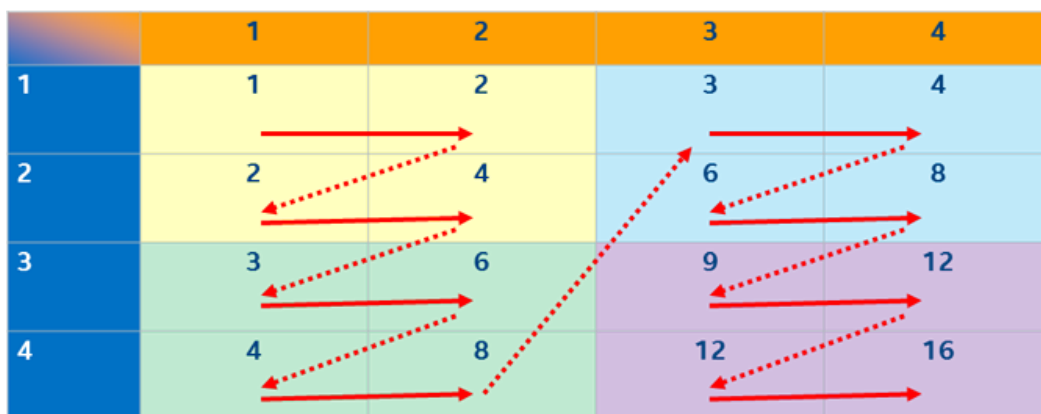
Internal Variables		Statistics						
Reference Points	Instructions	Core_Cycles	S_Cycles	N_Cycles	I_Cycles	C_Cycles	Total	
\$statistics	12231574	22224039	11725305	8055541	3348129	0	23128975	

Παρατηρούμε ότι οι συνολικοί κύκλοι αυξήθηκαν κατά 8%.

Ο λόγος για τον οποίο η διαφορά στο μέγεθος του κώδικα είναι μικρή είναι λόγω μικροδιαφορών μεταξύ των δύο κωδίκων. Η αύξηση των συνολικών κύκλων δικαιολογείται στο τρόπο με τον οποίο στη C τα δεδομένα αποθηκεύονται κατά γραμμή στη μνήμη και διαβάζονται από αυτήν, ενώ τώρα πραγματοποιούμε καταχώρηση κατά στήλη/

4^η μέθοδος Loop Tilling

Η loop tilling, γνωστή επίσης και ως αποκλεισμός βρόχου, είναι ένας μετασχηματισμός βρόχου που χωρίζει νοητά έναν πίνακα σε μικρότερους υποπίνακες(tiles). Αυτός ο μετασχηματισμός επιτρέπει την πρόσβαση στα δεδομένα σε μπλοκ(υποπίνακες), με το μέγεθος του μπλοκ να ορίζεται ως παράμετρος αυτού του μετασχηματισμού. Η loop tilling μπορεί να εντοπίσει τα διαφορετικά επίπεδα μνήμης (συμπεριλαμβανομένων των επιπέδων μνήμης cache) και μπορεί να συντονιστεί για να μεγιστοποιήσει την επαναχρησιμοποίηση δεδομένων σε ένα συγκεκριμένο επίπεδο της ιεραρχίας μνήμης. Ο μετασχηματισμός περιλαμβάνει τη χρήση συγκεκριμένων μεγεθών μπλοκ και μπορούν να διορθωθούν κατά τον χρόνο μεταγλώττισης και υπολογίζονται βάσει του μεγέθους δεδομένων και του μεγέθους μνήμης (π.χ., μνήμης cache) .



Χωρίσαμε τους δισδιάστατους πίνακες σε tiles όπως φαίνεται παρακάτω. Στην πρώτη περίπτωση κρατώντας σταθερές τις γραμμές και στη δεύτερη περίπτωση κρατώντας σταθερές τις στήλες.

1^η περίπτωση

288x16

288x22

288x32

288x44

288x88

2^η περίπτωση

16x352

24x352

32x352

48x352

96x352

Τα αποτελέσματα από το code warrior για την πρώτη και για την δεύτερη περίπτωση είναι τα παρακάτω

1^η περίπτωση

Image component sizes						
	Code	RO Data	RW Data	ZI Data	Debug	
	1044	60	0	1226796	4848	Object Totals
	10196	314	0	300	4548	Library Totals
=====						
	Code	RO Data	RW Data	ZI Data	Debug	
	11240	374	0	1227096	9396	Grand Totals
=====						
	Total RO	Size(Code + RO Data)			11614	(11.34kB)
	Total RW	Size(RW Data + ZI Data)			1227096	(1198.34kB)
	Total ROM	Size(Code + RO Data + RW Data)			11614	(11.34kB)
=====						

Παρατηρούμε ότι ο κώδικας έχει μεγαλώσει κατά 188 Bytes

2^η περίπτωση

Image component sizes					
Code	RO Data	RW Data	ZI Data	Debug	
1024	60	0	1226796	4844	Object Totals
10196	314	0	300	4548	Library Totals
=====					
Code	RO Data	RW Data	ZI Data	Debug	
11220	374	0	1227096	9392	Grand Totals
=====					
Total RO	Size(Code + RO Data)			11594	(11.32kB)
Total RW	Size(RW Data + ZI Data)			1227096	(1198.34kB)
Total ROM	Size(Code + RO Data + RW Data)			11594	(11.32kB)
=====					

Παρατηρούμε ότι ο κώδικας μεγαλώνει κατά 168 Bytes.

Τα αποτελέσματα από τον Debugger και για τις δύο περιπτώσεις.

1^η περίπτωση

288x16

Internal Variables		Statistics						
Reference Points		Instructions	Core_Cycles	S_Cycles	N_Cycles	I_Cycles	C_Cycles	Total
\$statistics		11568644	22319721	11043918	8675240	2868517	0	22587675

288x22

Internal Variables		Statistics						
Reference Points		Instructions	Core_Cycles	S_Cycles	N_Cycles	I_Cycles	C_Cycles	Total
\$statistics		11473418	22151823	10948740	8618096	2841391	0	22408227

288x32

Internal Variables		Statistics						
Reference Points		Instructions	Core_Cycles	S_Cycles	N_Cycles	I_Cycles	C_Cycles	Total
\$statistics		11394063	22011908	10869425	8570476	2818786	0	22258687

288x44

Internal Variables		Statistics						
Reference Points		Instructions	Core_Cycles	S_Cycles	N_Cycles	I_Cycles	C_Cycles	Total
\$statistics		11346450	21927959	10821836	8541904	2805223	0	22168963

288x88

Internal Variables		Statistics						
Reference	Points	Instructions	Core_Cycles	S_Cycles	N_Cycles	I_Cycles	C_Cycles	Total
\$statistics		11282966	21816027	10758384	8503808	2787139	0	22049331

2^η περίπτωση

16x352

Internal Variables		Statistics						
Reference	Points	Instructions	Core_Cycles	S_Cycles	N_Cycles	I_Cycles	C_Cycles	Total
\$statistics		12797680	23536019	12272986	8668823	3497414	0	24439223

24x352

Internal Variables		Statistics						
Reference	Points	Instructions	Core_Cycles	S_Cycles	N_Cycles	I_Cycles	C_Cycles	Total
\$statistics		12710908	23364671	12186262	8601119	3480494	0	24267875

32x352

Internal Variables		Statistics						
Ref...	Ins...	Cor...	S...	N...	I...	C...	Total	
\$statist	12667522	23278997	12142900	8567267	3472034	0	24182201	

48x352

Internal Variables		Statistics						
Reference	Points	Instructions	Core_Cycles	S_Cycles	N_Cycles	I_Cycles	C_Cycles	Total
\$statistics		12624136	23193323	12099538	8533415	3463574	0	24096527

96x352

Internal Variables		Statistics						
Ref...	Ins...	Cor...	S...	N...	I...	C...	Total	
\$statist:12580750	23107649	12056176	8499563	3455114	0		24010853	

Βλέπουμε πως σε κάθε περίπτωση έχουμε αύξηση του αριθμού των κύκλων. Στις εφαρμογές του tiling έχουμε καλύτερα εξαγόμενα από τις εφαρμογές του inverted tiling.

Παρατηρούμε μια λογική αύξηση του μεγέθους του κώδικα στα 1044 και 1024 Bytes αντίστοιχα. Μια μεταβολή λογική, αρκεί να αναλογιστούμε πως λόγω της εφαρμογής του tiling σε βρόχους του κώδικα, είχαμε έναν βρόχο επανάληψης παραπάνω σε κάθε περίπτωση.

Από την 1^η περίπτωση, η καλύτερη απόδοση συναντάται στο tile 288x88, όπου είχαμε αύξηση κατά περίπου 3,8% στους κύκλους.

Από την 2^η περίπτωση, η καλύτερη επίδοση συναντάται στο tile 96x352, όπου είχαμε αύξηση κατά περίπου 13%.

Στην πρώτη περίπτωση όπου ο εσωτερικός βρόχος εκτελεί στα στοιχεία προσπέλαση κατά γραμμή, έχουμε μικρότερο αριθμό συνολικών κύκλων γιατί τα δεδομένα αποθηκεύονται στη μνήμη με τον ίδιο τρόπο και όχι κατά στήλη.

Τέλος παρατηρούμε πως τα ZI DATA εμφανίζουν μια αύξηση της τάξης των 4 Byte, ακριβώς όσο το μέγεθος δήλωσης ενός στοιχείου τύπου int στη γλώσσα C. Αυτό συνέβη επειδή στον κώδικα προστέθηκαν οι ακέραιες μεταβλητές jj και ii σε κάθε περίπτωση.

5^η μέθοδος Loop Collapsing

Η μέθοδος loop collapsing ορίζει πως μερικοί βρόχοι επανάληψης θα συμπιηχθούν σε έναν ενιαίο βρόχο με σκοπό τη μείωση της επιβάρυνσης βρόχου για τη βελτίωση της απόδοσης χρόνου εκτέλεσης. Ενώ αποτελεί ακόμα μία μέθοδο βελτιστοποίησης, δεν είναι συνήθης η χρήση της σε compilers γλώσσας C.

Τα αποτελέσματα που πήραμε τρέχοντας τον κώδικα στο code warrior είναι τα παρακάτω

🔧	Image component sizes					
🔧	Code	RO Data	RW Data	ZI Data	Debug	
🔧	760	60	0	816156	4672	Object Totals
🔧	10196	314	0	300	4548	Library Totals
🔧	=====					
🔧	Code	RO Data	RW Data	ZI Data	Debug	
🔧	10956	374	0	816456	9220	Grand Totals
🔧	=====					
🔧	Total RO	Size(Code + RO Data)			11330 (11.06kB)	
🔧	Total RW	Size(RW Data + ZI Data)			816456 (797.32kB)	
🔧	Total ROM	Size(Code + RO Data + RW Data)			11330 (11.06kB)	
🔧	=====					

Παρατηρούμε ότι ο κώδικας έχει μικρύνει κατά 100 Bytes.

Τα αποτελέσματα από το Debugger είναι παρακάτω

Internal Variables	Statistics						
Reference Points	Instructions	Core_Cycles	S_Cycles	N_Cycles	I_Cycles	C_Cycles	Total
\$statistics	10105666	19372784	9681336	7248512	2782174	0	19712022

Παρατηρούμε ότι οι συνολικοί κύκλοι μειώθηκαν κατά 7,2%

Ο κώδικας, ως λογικό επακόλουθο της σύμπτυξης διαφορετικών βρόχων επανάληψης σε έναν ενιαίο, παρουσιάζει μείωση της τάξης των 100 Byte.

Επίσης παρατηρούμε μια κατακόρυφη πτώση στον όγκο των ZI DATA της τάξης των 40 KByte, κάτι για το οποίο ευθύνεται χρήση ενός pointer που αντικαθιστά τον πίνακα δεδομένων

6^η μέθοδος Loop Inversion

Μια άλλη βελτιστοποίηση με σκοπό τη μείωση του αριθμού των οδηγιών διακλάδωσης είναι η loop inversion μέθοδος. Η loop inversion είναι ένας αρκετά απλός μετασχηματισμός: ένας βρόχος επανάληψης που υλοποιείται πχ με for ή while μετατρέπεται σε βρόχο do-while τυλιγμένο από μια εντολή if. Όταν χρησιμοποιείται σωστά, μπορεί να βελτιώσει την απόδοση λόγω του instruction pipelining. Το instruction pipeline, χρησιμοποιείται στην CPU επιτρέποντας να εκτελούνται παράλληλα τα στάδια διαφορετικών εντολών αυξάνοντας την απόδοση του υπολογιστή. Στη σωλήνωση αυτή τα στοιχεία της επεξεργασίας είναι μέρη του επεξεργαστή που υλοποιούν τα επιμέρους στάδια εκτέλεσης μια εντολής (προσαγωγή, αποκωδικοποίηση, υπολογισμός τελεστών, προσαγωγή τελεστών, εκτέλεση εντολής κτλ.). Πρόκειται για μία μέθοδο παραλληλισμού εντολών στο υλικό του επεξεργαστή.

Συγκρίνοντας τις 2 μεθόδους επανάληψης(while-dowhile), μπορούμε εύκολα να δούμε ότι η dowhile μπορεί να μην εκτελέσει κανένα άλμα, υπό τον όρο ότι υπάρχει ακριβώς ένα βήμα μέσα στον βρόχο(έως ότου η συνθήκη να πάψει να είναι αληθής), και γενικά πως ο αριθμός των αλμάτων θα είναι κατά ένας μικρότερος από τον αριθμό των επαναλήψεων. Η πρώτη μέθοδος θα πρέπει να πηδήσει πίσω για να ελέγξει την συνθήκη, μόνο και μόνο για να βγει από το βρόχο όταν η κατάσταση είναι ψευδής.

Όμως τα άλματα στις σύγχρονες αρχιτεκτονικές CPU μπορεί να είναι αρκετά «κοστοβόρα» και χρονικώς επιζήμια. Καθώς η CPU ολοκληρώνει την εκτέλεση των ελέγχων πριν το άλμα(στην περίπτωση της do-while), οι οδηγίες πέρα από το άλμα είναι ήδη στο μέσο του αγωγού. Όλη αυτή η επεξεργασία πρέπει να απορριφθεί στις περιπτώσεις που το άλμα εκτελείται πριν τον έλεγχο της συνθήκης και εν τέλει η συνθήκη είναι ψευδής(θα μπορούσε να συμβεί στην περίπτωση της while). Έτσι εισάγεται περεταίρω καθυστέρηση.

Υλοποίηση Do-While

Στην δικιά μας περίπτωση, εφόσον είναι αναγκαία κάθε φορά η αρχικοποίηση των άκρων του εύρους της επανάληψης, η εντολή if η οποία θα ελέγχει τα όριά της επανάληψης είναι περιττή και στο πλαίσιο της βελτιστοποίησης του κώδικα, δεν χρησιμοποιήθηκε.

Τα αποτελέσματα απο το code warrior είναι τα παρακάτω

Do-while

Image component sizes					
Code	RO Data	RW Data	ZI Data	Debug	
848	60	0	1226792	4736	Object Totals
10196	314	0	300	4548	Library Totals
=====					
Code	RO Data	RW Data	ZI Data	Debug	
11044	374	0	1227092	9284	Grand Totals
=====					
Total RO	Size(Code + RO Data)			11418 (11.15kB)	
Total RW	Size(RW Data + ZI Data)			1227092 (1198.33kB)	
Total ROM	Size(Code + RO Data + RW Data)			11418 (11.15kB)	
=====					

While

Image component sizes					
Code	RO Data	RW Data	ZI Data	Debug	
896	60	0	1226792	4836	Object Totals
10196	314	0	300	4548	Library Totals
=====					
Code	RO Data	RW Data	ZI Data	Debug	
11092	374	0	1227092	9384	Grand Totals
=====					
Total RO	Size(Code + RO Data)			11466 (11.20kB)	
Total RW	Size(RW Data + ZI Data)			1227092 (1198.33kB)	
Total ROM	Size(Code + RO Data + RW Data)			11466 (11.20kB)	
=====					

Παρατηρούμε πως και στις δύο περιπτώσεις η μεταβολή του μεγέθους του κώδικα είναι μικρή. Στην περίπτωση των βρόχων do-while ο κώδικας μειώθηκε κατά 8 Bytes ενώ στη δεύτερη περίπτωση των βρόχων while αυξήθηκε κατά 40 Bytes.

Τα αποτελέσματα από τον Debugger είναι τα παρακάτω

while loop

Internal Variables		Statistics					
Ref...	Ins...	Cor...	S_...	N_...	I_...	C_...	Total
\$statist:	11024428	21230973	10522559	8161501	2774122	0	21458182

Παρατηρούμε ότι οι συνολικοί κύκλοι έχουν αυξηθεί κατά 1%

Do-while loop

Internal Variables		Statistics					
Reference Points	Instructions	Core_Cycles	S_Cycles	N_Cycles	I_Cycles	C_Cycles	Total
\$statistics	11016155	21009068	10510838	8054698	2670741	0	21236277

Παρατηρούμε ότι οι συνολικοί κύκλοι έχουν παραμείνει σχεδόν αμετάβλητοι-σε σχέση με τους αρχικούς. Αν συγκρίνουμε τα αποτελέσματα των 2 μεθόδων (while-dowhile) καταλήγουμε στο συμπέρασμα που υποστηρίξαμε και στο θεωρητικό κομμάτι. Ότι λόγω του γεγονότος ότι η do while θα πραγματοποιήσει ένα άλμα λιγότερο από τις συνολικές επαναλήψεις που θα υλοποιήσει (σε αντίθεση με την while) εισάγει λιγότερη καθυστέρηση, λιγότερα instructions (στην δικιά μας περίπτωση δεν έχουμε ούτε τους if ελέγχους λόγω της εκχώρησης τιμών λίγο πριν) με αποτέλεσμα λιγότερους συνολικούς κύκλους επεξεργαστή.

Η μικρή διαφορά στο μέγεθος του κώδικα ενδεχομένως να οφείλεται σε μικρές διαφορές ανάμεσα στους δύο κώδικες.

UNSHARP MASKING

Η unsharp masking αποτελεί μία μέθοδο ακονίσματος/βελτιστοποίησης μιας εικόνας. Το όνομα της τεχνικής, προκύπτει από το γεγονός ότι χρησιμοποιεί μια θολή (unsharp) εκδοχή της αρχικής εικόνας, για να δημιουργήσει μια μάσκα για αυτήν. Στη συνέχεια αυτή η μάσκα συνδυάζεται με την αρχική (αυθεντική) εικόνα, δημιουργώντας μια εικόνα που είναι λιγότερο θαμπή από την αρχική. Η προκύπτουσα εικόνα, αν και πιο καθαρή, μπορεί να είναι μια λιγότερο ακριβής αναπαράσταση του θέματος της εικόνας. Στο πλαίσιο της επεξεργασίας σήματος, η unsharp mask είναι γενικά ένα γραμμικό ή μη γραμμικό φίλτρο που ενισχύει τα συστατικά υψηλής συχνότητας ενός σήματος.

Τύπος unsharp masking: $\text{sharpened} = \text{original} + (\text{original} - \text{blurred}) \times \text{amount}$.



ΠΗΓΕΣ

- https://ocw.aoc.ntua.gr/modules/document/file.php/ECE102/%CE%91%CF%83%CE%BA%CE%AE%CF%83%CE%B5%CE%B9%CF%82/exercise1_updated.pdf
- https://bohr.wlu.ca/hfan/cp467/11/notes/cp467_11_lecture6_sharpening.pdf
- <http://homepages.inf.ed.ac.uk/rbf/HIPR2/unsharp.htm>
- <https://software.intel.com/en-us/articles/efficient-use-of-tiling>
- https://northstar-www.dartmouth.edu/doc/idl/html_6.2/Filtering_an_Imagehvr.html#wp1027822
- <https://www.sciencedirect.com/topics/engineering/laplacian-filter>
- <https://www.sciencedirect.com/topics/computer-science/loop-tiling>
- <http://www.nullstone.com/htmls/category/collapse.htm>
- https://en.wikipedia.org/wiki/Loop_unrolling
- https://en.wikipedia.org/wiki/Loop_interchange
- https://en.wikipedia.org/wiki/Loop_fission_and_fusion
- http://www.cs.columbia.edu/~ecj2122/research/x86_loop_optimizations/x86_loop_optimizations.pdf
- <https://www.tutorialcup.com/cprogramming/c-instructions.htm>
- https://stackoverflow.com/questions/20826718/what-is-the-loop-inversion-technique?fbclid=IwAR3zmCyVit6s-i_Hy6YpnBlk5uw3VilPbLLEsr_FOt8Q8l-oUzuEITh2N-E
- [https://el.wikipedia.org/wiki/%CE%A3%CF%89%CE%BB%CE%AE%CE%BD%CF%89%CF%83%CE%B7_\(%CF%85%CF%80%CE%BF%CE%BB%CE%BF%CE%B3%CE%B9%CF%83%CF%84%CE%AD%CF%82\)](https://el.wikipedia.org/wiki/%CE%A3%CF%89%CE%BB%CE%AE%CE%BD%CF%89%CF%83%CE%B7_(%CF%85%CF%80%CE%BF%CE%BB%CE%BF%CE%B3%CE%B9%CF%83%CF%84%CE%AD%CF%82))
- https://en.wikipedia.org/wiki/Unsharp_masking
- <https://stackoverflow.com/questions/42363726/bluring-an-image-in-c-c>