



**ΔΗΜΟΚΡΙΤΕΙΟ
ΠΑΝΕΠΙΣΤΗΜΙΟ
ΘΡΑΚΗΣ**

*"Σχεδιασμός Ενσωματωμένων
Συστημάτων"*

Δημοκρίτειο Πανεπιστήμιο Θράκης
ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ
ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

*«Ανιχνευτής ακμών κατά Prewitt και Robert cross
στην επεξεργασία εικόνας»
Μέρος 1^ο*

ΟΜΑΔΑ 48

Φαίδρα Μπογιάνογλου Βραχνού ΑΜ:58422

- Ξάνθη, 2024 -

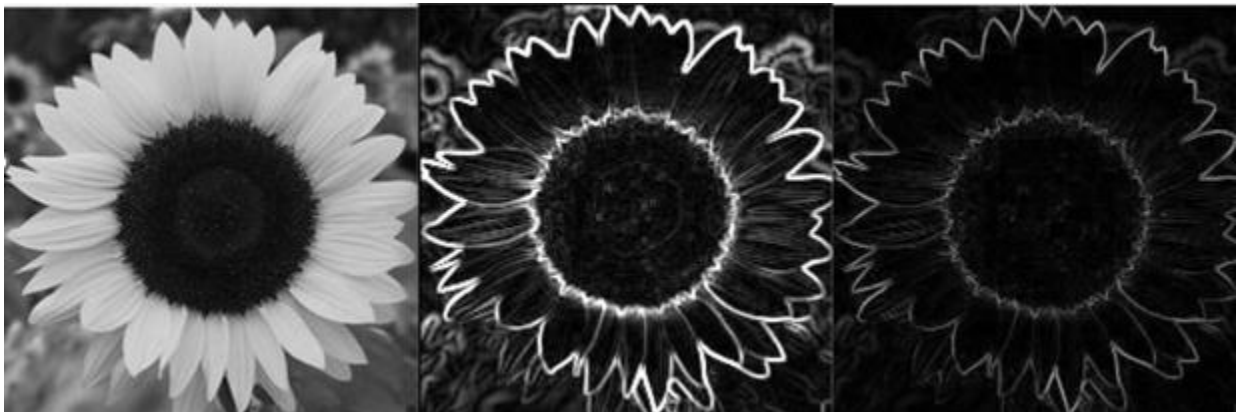
Μέρος 1^ο

Εισαγωγή

Η εργασία στα πλαίσια του μαθήματος "**Σχεδιασμός Ενσωματωμένων Συστημάτων**", αφορά την υλοποίηση και βελτιστοποίηση δύο κλασικών αλγορίθμων ανίχνευσης ακμών, του **Prewitt** και του **Roberts Cross**, σε γλώσσα C. Σε αυτή την εργασία θα υλοποιήσουμε τους δύο αλγόριθμους ανίχνευσης ακμών σε εικόνες μορφής YUV 4:4:4, όπου θα επεξεργαστούμε το κανάλι φωτεινότητας (Y). Οι συγκεκριμένοι αλγόριθμοι εφαρμόζουν διαφορετικές μάσκες για να εντοπίσουν περιοχές έντονης αλλαγής φωτεινότητας, οι οποίες αντιστοιχούν στις λεγόμενες ακμές της εικόνας.

Στόχος μας είναι η εφαρμογή τεχνικών βελτιστοποίησης στον κώδικα μέσω μετασχηματισμών βρόχων, για την κανονικοποίηση της δομής του αλγορίθμου και τη βελτίωση απόδοσής του.

Η βελτιστοποίηση των δύο αυτών αλγορίθμων είναι ιδιαίτερα σημαντική σε συστήματα χαμηλής ισχύος και περιορισμένων πόρων, όπως αυτά που χρησιμοποιούνται στα ενσωματωμένα συστήματα. Μέσω της βελτίωσης του κώδικα και της αποδοτικής διαχείρισης δεδομένων, επιτυγχάνονται καλύτερη απόδοση και μειωμένη κατανάλωση μνήμης.



Εικόνα 1: Σύγκριση των μεθόδων Prewitt και Robert Cross.

Περιγραφή Αλγορίθμου

Η ανίχνευση ακμών πραγματοποιείται με βάση τη διαφορική προσέγγιση, όπου οι αλλαγές φωτεινότητας υπολογίζονται τοπικά μέσω της συνέλιξης της εικόνας με δύο ειδικές μάσκες (kernel). Οι δύο αλγόριθμοι που χρησιμοποιούνται είναι:

1. Prewitt

$$\mathbf{G}_x = \begin{bmatrix} +1 & +1 & +1 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \end{bmatrix} * \mathbf{A} \quad \text{and} \quad \mathbf{G}_y = \begin{bmatrix} +1 & 0 & -1 \\ +1 & 0 & -1 \\ +1 & 0 & -1 \end{bmatrix} * \mathbf{A}$$

Ο αλγόριθμος Prewitt χρησιμοποιεί δύο μάσκες 3×3 για τον υπολογισμό των παραγώγων κατά τις διευθύνσεις x (οριζόντια) και y (κατακόρυφη). Ύστερα, οι μάσκες G_x και G_y εφαρμόζονται ξεχωριστά μέσω συνέλιξης, και το συνολικό μέτρο του gradient υπολογίζεται ως:

$$\mathbf{G} = \sqrt{\mathbf{G}_x^2 + \mathbf{G}_y^2}$$

Τέλος, το αποτέλεσμα κλιμακώνεται ώστε να βρίσκεται στο εύρος τιμών $[0, 255]$.

2. Roberts Cross

$$\begin{bmatrix} +1 & 0 \\ 0 & -1 \end{bmatrix} \quad \text{and} \quad \begin{bmatrix} 0 & +1 \\ -1 & 0 \end{bmatrix}.$$

Ο αλγόριθμος Roberts Cross χρησιμοποιεί δύο μικρότερες μάσκες 2×2. Παρόμοια με τον Prewitt, υπολογίζει τις παραγώγους G_x και G_y , αλλά λόγω της μικρότερης μάσκας είναι πιο ευαίσθητος σε λεπτομέρειες. Ο υπολογισμός του gradient γίνεται με τον ίδιο τρόπο:

$$\mathbf{G} = \sqrt{\mathbf{G}_x^2 + \mathbf{G}_y^2}$$

Τα βασικά βήματα της υλοποίησης είναι τα εξής:

Αρχικά, πραγματοποιείται η φόρτωση της εικόνας. Η εικόνα εισόδου διαβάζεται σε μορφή YUV 4:4:4 και επεξεργάζεται μόνο το κανάλι φωτεινότητας Y . Έπειτα γίνεται η συνέλιξη με μάσκες. Δηλαδή, εφαρμόζονται οι μάσκες Gx και Gy μέσω συνελίξεων, λαμβάνοντας υπόψη την περιοχή του kernel. Μετέπειτα, υλοποιείται ο υπολογισμός Gradient. Υπολογίζεται το συνολικό μέτρο του gradient και τα αποτελέσματα του ομαλοποιούνται. Τέλος, υπάρχει η αποθήκευση των αποτελεσμάτων, δηλαδή το κανάλι Y με τις ανιχνευμένες ακμές αποθηκεύεται μαζί με τα κανάλια U και V , ώστε η έξοδος να είναι συμβατή με τη μορφή YUV.

Παρουσίαση Αρχικού Κώδικα

1. Prewitt

```
void apply_prewitt() {
    int x, y, gx, gy;
    for (i = 1; i < N - 1; i++) {
        for (j = 1; j < M - 1; j++) {
            gx = gy = 0;
            for (x = 0; x < 3; x++) {
                for (y = 0; y < 3; y++) {
                    gx += current_y[i + x - 1][j + y - 1] * Gx_Prewitt[x][y];
                    gy += current_y[i + x - 1][j + y - 1] * Gy_Prewitt[x][y];
                }
            }
            edge_prewitt[i][j] = (int)sqrt(gx * gx + gy * gy);
            if (edge_prewitt[i][j] > 255) edge_prewitt[i][j] = 255;
            else if (edge_prewitt[i][j] < 0) edge_prewitt[i][j] = 0;
        }
    }
}
```

Η συνάρτηση **apply_prewitt** υπολογίζει τις ακμές της εικόνας χρησιμοποιώντας τα δύο kernels του φίλτρου Prewitt, Gx και Gy .

Αρχικά, διατρέχεται το κάθε pixel της εικόνας (εκτός από τα όρια) με τις δύο εξωτερικές λούπες. Έπειτα, ορίζονται δύο τιμές, gx και gy , για την οριζόντια και την κατακόρυφη κλίση. Έπειτα, χρησιμοποιούνται δύο εσωτερικοί βρόχοι και πραγματοποιείται το convolution. Το συνολικό μέτρο της κλίσης υπολογίζεται ως $\sqrt{gx^2 + gy^2}$. Τέλος, το αποτέλεσμα περιορίζεται στις τιμές $[0, 255]$, που είναι το εύρος των τιμών για εικόνες 8-bit σε grayscale.

2. Roberts Cross

```
void apply_roberts() {
    int x, y, gx, gy;
    for (i = 0; i < N - 1; i++) {
        for (j = 0; j < M - 1; j++) {
            gx = gy = 0;
            for (x = 0; x < 2; x++) {
                for (y = 0; y < 2; y++) {
                    gx += current_y[i + x][j + y] * Gx_Roberts[x][y];
                    gy += current_y[i + x][j + y] * Gy_Roberts[x][y];
                }
            }
            edge_roberts[i][j] = (int)sqrt(gx * gx + gy * gy);
            if (edge_roberts[i][j] > 255) edge_roberts[i][j] = 255;
            else if (edge_roberts[i][j] < 0) edge_roberts[i][j] = 0;
        }
    }
}
```

Η συνάρτηση **apply_roberts** υλοποιεί το φίλτρο Roberts Cross, το οποίο είναι μια απλούστερη μέθοδος ανίχνευσης ακμών, με πυρήνες 2x2.

Οι εξωτερικοί βρόχοι (for (i, j)) διατρέχουν τα pixel της εικόνας, εξαιρώντας την τελευταία γραμμή και στήλη, λόγω του μεγέθους των πυρήνων. Χρησιμοποιούνται πυρήνες G_x και G_y μεγέθους 2x2 για τον υπολογισμό των τιμών gx και gy . Όπως και στον Prewitt, το μέτρο της κλίσης υπολογίζεται ως $\sqrt{gx^2 + gy^2}$. Τα αποτελέσματα περιορίζονται επίσης στις τιμές [0,255].

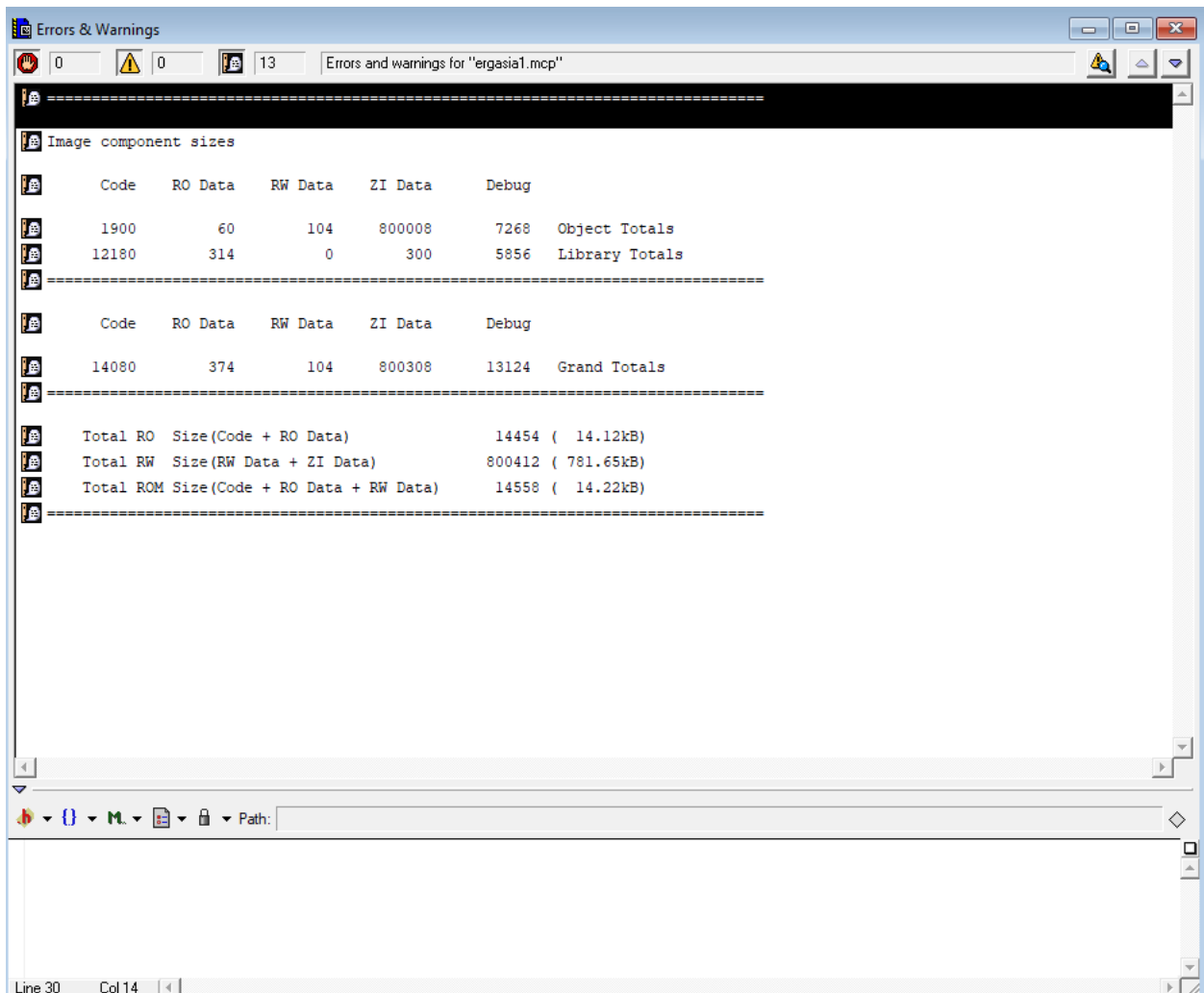


Εικόνα 2: Σύγκριση των μεθόδων Prewitt και Robert Cross.

Μέθοδοι Βελτιστοποίησης

Αρχική υλοποίηση

Τρέχοντας το πρόγραμμα για πρώτη φορά στο ARMulator λαμβάνονται τα αποτελέσματα με τα μεγέθη των πινάκων δεδομένων καθώς και ο αριθμός των κύκλων όπως φαίνονται παρακάτω:



The screenshot shows the 'Errors & Warnings' window of the ARMulator. The window title is 'Errors & Warnings' and the subtitle is 'Errors and warnings for "ergasia1.mcp"'. The main content area displays the following text:

```
=====
Image component sizes
=====
Code    RO Data  RW Data  ZI Data  Debug
-----
1900    60        104     800008   7268   Object Totals
12180   314       0       300     5856   Library Totals
=====

Code    RO Data  RW Data  ZI Data  Debug
-----
14080   374      104     800308   13124  Grand Totals
=====

Total RO  Size(Code + RO Data)          14454 ( 14.12kB)
Total RW  Size(RW Data + ZI Data)      800412 ( 781.65kB)
Total ROM Size(Code + RO Data + RW Data)  14558 ( 14.22kB)
=====
```

Below the main content area, there is a path field and a status bar showing 'Line 30 Col 14'. At the bottom of the window, there is a table with the following data:

Reference Points	Instructions	Core_Cycles	S_Cycles	N_Cycles	I_Cycles	C_Cycles	Total
\$statistics	124825576	202541476	141262357	47033740	17361904	0	205658001

Εικόνα 3: Αποτελέσματα αρχικής υλοποίησης (Μέγεθος πίνακα δεδομένων και αριθμός κύκλων)

Σε αυτό το σημείο, είναι απαραίτητο να δοθεί εξήγηση της σημασίας των πεδίων στη δεύτερη εικόνα, τα οποία παρουσιάζουν τα αποτελέσματα της αρχικής υλοποίησης:

1. Instructions: Συνολικός αριθμός των εντολών που εκτελέστηκαν κατά τη διάρκεια της προσομοίωσης. Οι ARM επεξεργαστές, ως αρχιτεκτονικές RISC, εκτελούν πολλές εντολές γρήγορα λόγω της απλοποιημένης σχεδίασής τους.

2. Core Cycles: Αριθμός των κύκλων του επεξεργαστή για την εκτέλεση όλων των εντολών. Περιλαμβάνει όλους τους τύπους κύκλων (stall, normal, idle).

3. Stall (S) Cycles: Κύκλοι των καθυστερήσεων, όπως αναμονές για μνήμη ή εξαρτήσεις δεδομένων.

4. Normal (N) Cycles: Κύκλοι της κανονικής λειτουργίας χωρίς καθυστερήσεις.

5. Idle (I) Cycles: Κύκλοι όπου ο επεξεργαστής είναι αδρανής, π.χ., λόγω απουσίας εντολών ή αναμονής εξωτερικών γεγονότων.

6. Coprocessor Cycles (C): Κύκλοι εκτέλεσης σε συνεπεξεργαστή, που στα πλαίσια του πρώτου μέρους της εργασίας απουσιάζουν.

Για τη βελτιστοποίηση του αλγορίθμου, εξετάζονται διάφορες τεχνικές βελτιστοποίησης με μετασχηματισμό βρόχων, οι οποίες εφαρμόστηκαν στις λειτουργίες επεξεργασίας εικόνας για τη μείωση του χρόνου εκτέλεσης και την αποτελεσματικότητα του κώδικα. Παρακάτω θα γίνει ανάλυση όλων των τεχνικών καθώς και των επιπτώσεών τους στον κώδικα.

1. Loop unrolling

Η τεχνική **loop unrolling** μπορεί να χρησιμοποιηθεί για τη βελτιστοποίηση των επαναληπτικών βρόχων αυξάνοντας την απόδοση του κώδικα με τη μείωση του κόστους επανάληψης (π.χ. του ελέγχου συνθηκών και των ενημερώσεων). Έτσι έχουμε μείωση του overhead από τον έλεγχο των συνθηκών και την ενημέρωση του δείκτη επανάληψης σε κάθε βήμα του loop. Αυτή η μέθοδος επιτρέπει την εκτέλεση πολλαπλών επαναλήψεων σε μία μόνο βηματική διαδικασία. Παρακάτω εφαρμόζουμε **loop unrolling** στους πιο κρίσιμους βρόχους του κώδικα, συγκεκριμένα στις συναρτήσεις **read**, **apply_prewitt** και **apply_roberts**.

Για παράδειγμα, στην αντιγραφή των byte της εικόνας εισόδου (input image bytes) στον προσωρινό buffer input, τροποποιήθηκε το loop ως εξής:

```
for (i = 0; i < N; i++)
{
    for (j = 0; j < M; j += 4) // Unrolling by 4
    {
        current_y[i][j] = fgetc(frame_c);
        current_y[i][j + 1] = fgetc(frame_c);
        current_y[i][j + 2] = fgetc(frame_c);
        current_y[i][j + 3] = fgetc(frame_c);
    }
}
```


Με αυτόν τον τρόπο μειώνεται το overhead της διαχείρισης του loop (όπως τον έλεγχο συνθηκών και την ενημέρωση των μετρητών που αναγκάζονται να εκτελούνται σε κάθε επανάληψη της for loop), επιταχύνοντας την εκτέλεση, αυξάνοντας ωστόσο δραματικά τις απαιτήσεις μνήμης του προγράμματος.

Όσον αφορά τις συναρτήσεις **apply_prewitt** και **apply_roberts**, στον αρχικό κώδικα οι βρόχοι εξετάζουν κάθε pixel της εικόνας, από το δεύτερο ($i = 1, j = 1$) μέχρι το προτελευταίο ($i = N-2, j = M-2$), για την εφαρμογή του φίλτρου Prewitt σε κάθε pixel της εικόνας.

```
for (i = 1; i < N - 1; i++)
{
    for (j = 1; j < M - 1; j++)
    {
        // Επεξεργασία κάθε pixel
    }
}
```

Τροποποιώντας τον βρόχο, τώρα το j αυξάνεται με βήμα 2 (δηλαδή, $j += 2$):

```
for (i = 1; i < N - 1; i++)
{
    for (j = 1; j < M - 1; j += 2) // Unrolling by a factor of 2
    {
        // First pixel
        gx = 0;
        gy = 0;

        for (x = 0; x < 3; x++)
        {
            for (y = 0; y < 3; y++)
            {
                gx += current_y[i + x - 1][j + y - 1] * Gx_Prewitt[x][y];
                gy += current_y[i + x - 1][j + y - 1] * Gy_Prewitt[x][y];
            }
        }

        edge_prewitt[i][j] = (int)sqrt(gx * gx + gy * gy);
        if (edge_prewitt[i][j] > 255)
            edge_prewitt[i][j] = 255;

        // Second pixel
        if (j + 1 < M - 1) // Ensure within bounds
        {
            gx = 0;
            gy = 0;
```

```

        for (x = 0; x < 3; x++)
        {
            for (y = 0; y < 3; y++)
            {
                gx += current_y[i + x - 1][j + y] * Gx_Prewitt[x][y];
                gy += current_y[i + x - 1][j + y] * Gy_Prewitt[x][y];
            }
            edge_prewitt[i][j + 1] = (int)sqrt(gx * gx + gy * gy);
            if (edge_prewitt[i][j + 1] > 255)
                edge_prewitt[i][j + 1] = 255;
        }
    }
}

```

Με το **unrolling** του βρόχου επεξεργάζονται δύο γειτονικά pixels ταυτόχρονα μέσα σε έναν βρόχο, αντί να επεξεργάζεται κάθε pixel σε ξεχωριστό γύρο. Έτσι μειώνεται το κόστος των επαναλήψεων, όπως οι έλεγχοι συνθηκών του βρόχου ($j < M - 1$) και η αύξηση του δείκτη j , με το να εκτελείται περισσότερο έργο σε κάθε γύρο του βρόχου.

Referen...	Instruct...	Core_Cycles	S_Cycles	N_Cycles	I_Cycles	C_Cycles	Total
\$statistics	122974862	199356803	139214234	46172806	17026986	0	202414026

Εικόνα 4: Αποτελέσματα loop unrolling

2. Loop fusion

Η τεχνική **loop fusion** συνδυάζει πολλούς βρόχους που εκτελούν παρόμοιες ή σχετικές εργασίες σε έναν ενιαίο βρόχο, μειώνοντας τον αριθμό των επαναλήψεων και της διαχείρισης του ελέγχου, καθώς και βελτιώνοντας τη συνολική αποδοτικότητα του προγράμματος. Έτσι εφαρμόζεται ενοποίηση στους βρόχους ανάγνωσης και εγγραφής, με τον εξής τρόπο:

```

for (int i = 0; i < N; i++)
{
    for (int j = 0; j < M; j++)
    {
        current_y[i][j] = fgetc(frame_c);
        current_u[i][j] = fgetc(frame_c);
        current_v[i][j] = fgetc(frame_c);
    }
}

```

Αρχικά υπήρχαν τρεις ξεχωριστοί βρόχοι για την ανάγνωση των τριών καναλιών της εικόνας (Y, U, V). Με τη συγχώνευση, οι τρεις αναγνώσεις συμβαίνουν μέσα σε έναν βρόχο, που κάνει την διαδικασία πιο αποτελεσματική. Το ίδιο εφαρμόστηκε και για την εγγραφή της εικόνας. Με τη συγχώνευση των βρόχων, μειώνονται οι αριθμοί των βρόχων και η πολυπλοκότητα του κώδικα. Η ενοποίηση των βρόχων μειώνει το κόστος του overhead που συνδέεται με την επαναλαμβανόμενη επεξεργασία των ίδιων δεικτών, και μπορεί να οδηγήσει σε καλύτερη εκμετάλλευση της μνήμης cache και της επεξεργαστικής ισχύος.

Referen...	Instruc...	Core_Cy...	S_Cycles	N_Cycles	I_Cycles	C_Cycles	Total
\$statistics	123938976	201289498	140313968	46730530	17360777	0	204405275

Εικόνα 5: Αποτελέσματα *loop fusion*

3. Loop fission

Το **loop fission** είναι μια τεχνική βελτιστοποίησης του κώδικα, όπου ένας μεγάλος βρόχος (loop) διαχωρίζεται σε δύο ή περισσότερους μικρότερους βρόχους. Αυτό μπορεί να έχει ως στόχο τη βελτίωση της απόδοσης μέσω της μείωσης της αλληλεξάρτησης μεταξύ των βρόχων, της καλύτερης εκμετάλλευσης της μνήμης cache και της παράλληλης εκτέλεσης.

Στον αρχικό κώδικα, ο υπολογισμός των παραγώγων gx και gy για το φίλτρο Prewitt ή Roberts γινόταν σε έναν ενιαίο βρόχο. Στην νέα έκδοση, διαχωρίζουμε τον υπολογισμό των παραγώγων gx και gy σε δύο ξεχωριστούς βρόχους και τους αποθηκεύουμε σε ενδιάμεσους πίνακες (gx_temp και gy_temp). Μετά, σε έναν τρίτο βρόχο υπολογίζουμε το μέτρο της τιμής της έντασης και κάνουμε την κανονικοποίηση:

```
// Loop Fission: Υπολογισμός gx
for (int i = 1; i < N - 1; i++)
    for (int j = 1; j < M - 1; j++)
    {
        int gx = 0;
        for (int x = 0; x < 3; x++)
            for (int y = 0; y < 3; y++)
                gx += current_y[i + x - 1][j + y - 1] * Gx_Prewitt[x][y];
        gx_temp[i][j] = gx;
    }

// Loop Fission: Υπολογισμός gy
for (int i = 1; i < N - 1; i++)
    for (int j = 1; j < M - 1; j++)
    {
```

```

int gy = 0;
for (int x = 0; x < 3; x++)
    for (int y = 0; y < 3; y++)
        gy += current_y[i + x - 1][j + y - 1] * Gy_Prewitt[x][y];
gy_temp[i][j] = gy;
}

// Loop Fission: Υπολογισμός μέτρου και κανονικοποίηση
for (int i = 1; i < N - 1; i++)
    for (int j = 1; j < M - 1; j++)
    {
        edge_prewitt[i][j] = (int)sqrt(gx_temp[i][j] * gx_temp[i][j] +
gy_temp[i][j] * gy_temp[i][j]);
        if (edge_prewitt[i][j] > 255)
            edge_prewitt[i][j] = 255;
        else if (edge_prewitt[i][j] < 0)
            edge_prewitt[i][j] = 0;
    }

```

Αυτό βοηθάει στην απομόνωση και διαχείριση διαφορετικών υπολογιστικών εργασιών σε ξεχωριστούς βρόχους. Με αυτόν τον τρόπο, η εκτέλεση γίνεται πιο οργανωμένη και λιγότερο εξαρτημένη από την κατάσταση των δεδομένων σε άλλους βρόχους.

Referen...	Instruc...	Core_Cy...	S_Cycles	N_Cycles	I_Cycles	C_Cycles	Total
\$statistics	72876396	109648524	78111744	24485273	10324445	0	112921462

Εικόνα 6: Αποτελέσματα loop fission

4. Loop interchange

Το **loop interchange** αναφέρεται στην αλλαγή της σειράς με την οποία εκτελούνται οι δύο (ή περισσότεροι) εμφωλευμένοι βρόχοι. Στο πρόγραμμα μας, οι εξωτερικοί και εσωτερικοί βρόχοι άλλαξαν θέσεις:

```

for (j = 0; j < M; j++) {
    for (i = 0; i < N; i++) {
        current_y[i][j] = fgetc(frame_c);
    }
}

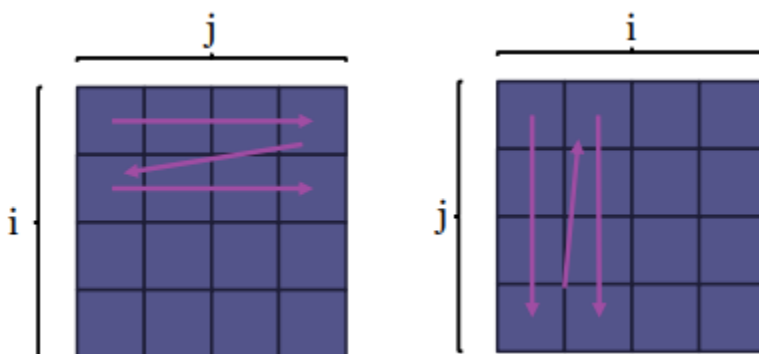
```

Αντί για πρώτα γραμμές (i), μετά στήλες (j), έχουμε πρώτα στήλες (j), μετά γραμμές (i).

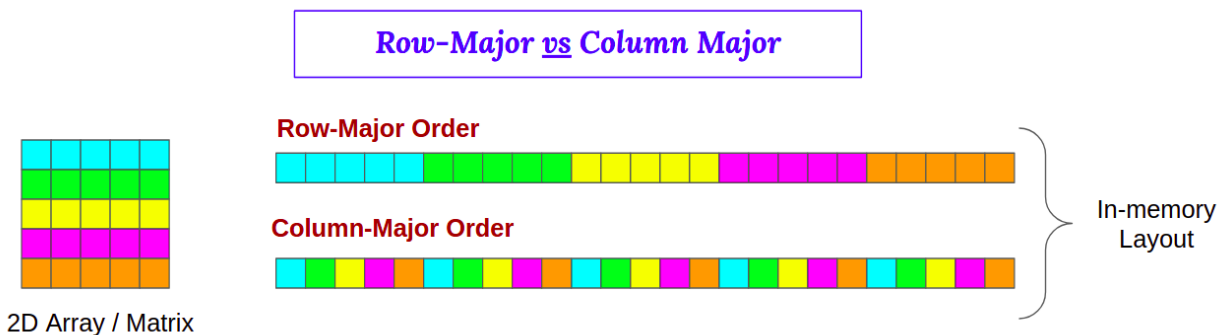
Το ίδιο πραγματοποιήθηκε και στην εφαρμογή φίλτρων Prewitt και Roberts:

```
for (j = 1; j < M - 1; j++) { // Εξωτερικός βρόχος για τις στήλες
    for (i = 1; i < N - 1; i++) { // Εσωτερικός βρόχος για τις γραμμές
        // Υπολογισμοί gx και gy
    }
}
```

Με το loop interchange πετυγχάνεται η βελτίωση προσπέλασης της μνήμης (memory access), καθώς οι σύγχρονες CPUs λειτουργούν αποδοτικότερα όταν τα δεδομένα που χρησιμοποιούνται βρίσκονται διαδοχικά στη μνήμη (spatial locality). Επίσης, τα δεδομένα του πίνακα αποθηκεύονται στη μνήμη κατά γραμμές, πράγμα που συμφέρει ιδιαίτερα την C αφού είναι **row-major order** γλώσσα. Με το interchange, οι εσωτερικοί βρόχοι επεξεργάζονται συνεχόμενα στοιχεία στη μνήμη, βελτιώνοντας την απόδοση (cache efficiency).



Εικόνα 7: Σχηματική αποτύπωση του loop interchange



Εικόνα 8: Row Major vs Column Major στον χώρο της μνήμης

Internal Variables		Statistics					
Referen...	Instructions	Core_Cycles	S_Cycles	N_Cycles	I_Cycles	C_Cy...	Total
\$statistics	64917115	93341866	71926179	16844751	7881885	0	96652815

Εικόνα 9: Αποτελέσματα *loop interchange*

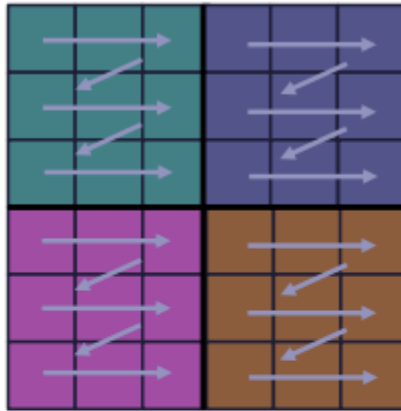
5. Loop tiling

Η τεχνική **loop tiling** βελτιστοποιεί τη χρήση της ιεραρχίας της μνήμης χωρίζοντας τους βρόχους σε μικρότερα κομμάτια (πλακίδια ή "tiles") που ταιριάζουν καλύτερα στη cache. Αυτή η προσέγγιση επιτρέπει στον αλγόριθμο να εκτελείται πιο αποδοτικά, καθώς μικρότερα τμήματα δεδομένων παραμένουν στη μνήμη cache κατά την επεξεργασία. Αντί να επεξεργάζεται ολόκληρος ο πίνακας σειριακά, ο υπολογισμός περιορίζεται σε μικρότερα blocks, που χωρούν πιο αποδοτικά στις cache μνήμες, μειώνοντας έτσι την ανάγκη για πρόσβαση στην κύρια μνήμη. Με τον τρόπο αυτό γλιτώνονται τις συχνές προσκομίσεις του CPU στην RAM. Στην περιπτωσή μας θα εφαρμόσουμε loop tiling στις συναρτήσεις `apply_prewitt()` και `apply_roberts()`.

```
void apply_prewitt()
{
    int x, y, gx, gy;
    int tile_size = 16; // Μέγεθος πλακιδίων (μπορεί να προσαρμοστεί για καλύτερη
// απόδοση)

    for (int ii = 1; ii < N - 1; ii += tile_size) // Εξωτερικός βρόχος για τα
// tiles (γραμμές)
    {
        for (int jj = 1; jj < M - 1; jj += tile_size) // Εξωτερικός βρόχος για τα
// tiles (στήλες)
        {
            for (i = ii; i < ii + tile_size && i < N - 1; i++) // Εσωτερικός βρόχος
// για επεξεργασία πλακιδίου
            {
                for (j = jj; j < jj + tile_size && j < M - 1; j++)
                {
                    //εφαρμογή φίλτρου
                }
            }
        }
    }
}
```

Το μέγεθος του πλακιδίου `tile_size` έχει οριστεί σε 16, αλλά μπορεί να προσαρμοστεί ανάλογα με την cache του συστήματος. Επίσης, οι εξωτερικοί βρόχοι (`ii` και `jj`) διατρέχουν τα πλακίδια του πίνακα, ενώ οι εσωτερικοί βρόχοι (`i` και `j`) υπολογίζουν τις τιμές μέσα σε κάθε πλακίδιο.



Loop tiling divides loops into smaller blocks, improving data locality and reducing cache misses

Εικόνα 10: Σχηματική αποτύπωση του loop tiling

Με τον τρόπο αυτό πετυχαίνουμε αλύτερη χρήση της cache, καθώς κάθε πλακίδιο χωρά στη μνήμη, μειώνοντας τα cache misses.

Referen...	Instruc...	Core_Cy...	S_Cycles	N_Cycles	I_Cycles	C_Cycles	Total
\$statistics	125432972	203806410	141875571	47377896	17669468	0	206922935

Εικόνα 11: Αποτελέσματα loop tiling

6. Loop collapsing

Το **loop collapsing** είναι μία τεχνική που συνδυάζει πολλαπλούς βρόχους (nested loops) σε έναν ενιαίο βρόχο, προκειμένου να βελτιστοποιηθεί η αποδοτικότητα ή να απλοποιηθεί ο

κώδικας. Στον παρακάτω κώδικα, η επεξεργασία για το Prewitt και Roberts Operators τροποποιείται ώστε να χρησιμοποιηθεί ενιαίος βρόχος μέσω της τεχνικής collapsing.

```
void apply_roberts()
{
    int gx, gy;

    for (int idx = 0; idx < (N - 1) * (M - 1); idx++) // Ενιαίος βρόχος
    {
        int i = idx / (M - 1); // Μετατροπή μονοδιάστατης συντεταγμένης σε γραμμή
        int j = idx % (M - 1); // Μετατροπή μονοδιάστατης συντεταγμένης σε στήλη

        gx = 0;
        gy = 0;

        // Εφαρμογή του Roberts φίλτρου
        for (int x = 0; x < 2; x++)
        {
            for (int y = 0; y < 2; y++)
            {
                gx += current_y[i + x][j + y] * Gx_Roberts[x][y];
                gy += current_y[i + x][j + y] * Gy_Roberts[x][y];
            }
        }

        edge_roberts[i][j] = (int)sqrt(gx * gx + gy * gy);

        if (edge_roberts[i][j] > 255)
            edge_roberts[i][j] = 255;
        else if (edge_roberts[i][j] < 0)
            edge_roberts[i][j] = 0;
    }
}
```

Ο διπλός βρόχος for (i) και for (j) αντικαθίσταται από έναν βρόχο for (int idx) που διατρέχει τα στοιχεία με μονοδιάστατη σειρά. Επιπλέον, οι συντεταγμένες (i, j) υπολογίζονται από τον δείκτη idx, ως $i = idx / (M - 1)$ (γραμμή) και $j = idx \% (M - 1)$ (στήλη).

Referen...	Instruc...	Core_Cy...	S_Cycles	N_Cycles	I_Cycles	C_Cycles	Total
\$statistics	129842808	210635938	145564779	50057333	18148159	0	213770271

Εικόνα 12: Αποτελέσματα loop collapsing

7. Loop inversion

Το **loop inversion** είναι μία τεχνική βελτιστοποίησης, όπου ένας βρόχος μετασχηματίζεται ώστε να χρησιμοποιεί διαφορετική λογική εξόδου (π.χ., προϋποθέσεις εξόδου αντί για είσοδο). Στον κώδικα εδώ, μπορούμε να εφαρμόσουμε **loop inversion** αλλάζοντας τον τρόπο που γίνεται ο έλεγχος των συνθηκών στα for loops.

```
void apply_prewitt()
{
    int x, y, gx, gy;

    i = 1; // Αρχικοποίηση εκτός του βρόχου
    while (i < N - 1) // Αντί του `for (i = 1; i < N - 1; i++)`
    {
        j = 1; // Αρχικοποίηση εκτός του εσωτερικού βρόχου
        while (j < M - 1) // Αντί του `for (j = 1; j < M - 1; j++)`
        {
            gx = 0;
            gy = 0;

            for (x = 0; x < 3; x++) // Παραμένει ίδιος
            {
                for (y = 0; y < 3; y++)
                {
                    gx += current_y[i + x - 1][j + y - 1] * Gx_Prewitt[x][y];
                    gy += current_y[i + x - 1][j + y - 1] * Gy_Prewitt[x][y];
                }
            }

            edge_prewitt[i][j] = (int)sqrt(gx * gx + gy * gy);

            if (edge_prewitt[i][j] > 255)
                edge_prewitt[i][j] = 255;
            else if (edge_prewitt[i][j] < 0)
                edge_prewitt[i][j] = 0;

            j++; // Ενημέρωση του εσωτερικού δείκτη
        }
        i++; // Ενημέρωση του εξωτερικού δείκτη
    }
}
```

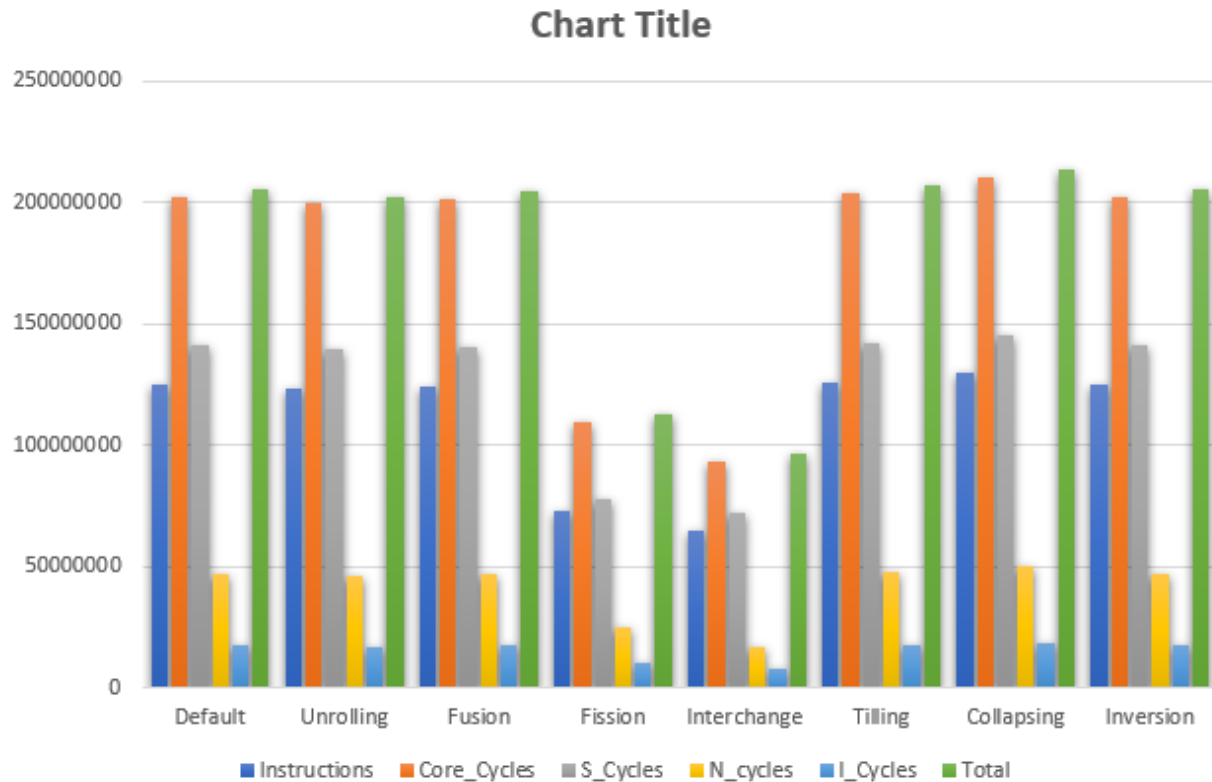
Αρχικά εφαρμόζεται αντικατάσταση των for με while. Δηλαδή τα for ($i = 1; i < N - 1; i++$) και for ($j = 1; j < M - 1; j++$) αντικαθίστανται από while loops. Εφαρμόζοντας αυτό, γίνονται

οι αρχικοποιήσεις ($i = 1, j = 1$) πριν από τον αντίστοιχο βρόχο και μεταφέρεται η ενημέρωση ($i++$, $j++$) στο τέλος του σώματος του βρόχου. Με τον τρόπο αυτό χρησιμοποιείται πιο σαφή ροή, αφού ο βρόχος δεν εξαρτάται από ενσωματωμένες δομές for.

Referen...	Instruc...	Core_Cy...	S_Cycles	N_Cycles	I_Cycles	C_Cycles	Total
\$statistics	124827569	202546657	141265148	47035334	17362700	0	205663182

Εικόνα 13: Αποτελέσματα *loop inversion*

Σύγκριση διαφορετικών τεχνικών βελτιστοποίησης



Εικόνα 14: Διάγραμμα σύγκρισης των μεθόδων βελτιστοποίησης.

Στο παραπάνω διάγραμμα φαίνονται συγκεντρωτικά τα αποτελέσματα όλων των τεχνικών βελτιστοποίησης που υλοποιήσαμε. Πιο συγκεκριμένα, η πιο αποτελεσματική μέθοδος βελτιστοποίησης του κώδικα είναι μέσω της τεχνικής του Loop Interchange, η οποία, με την προσπέλαση της μνήμης, επιτρέπει ταχύτερη εκτέλεση των εντολών, μειώνοντας τα cache misses. Ιδιαίτερα αποδοτική στη περιπτωσή αυτή είναι και η Loop Fission, που με τον διαχωρισμό μεγάλων βρόχων σε μικρότερους, τα δεδομένα που χρησιμοποιούνται από έναν βρόχο μπορούν να παραμείνουν πιο κοντά στην κρυφή μνήμη (cache), μειώνοντας τις δαπανηρές προσπελάσεις στη RAM. Εντούτοις, παρατηρείται σημαντική αύξηση στον αριθμό των κύκλων (cycles) για την εκτέλεση, ιδιαίτερα στις τεχνικές που σχετίζονται με την τμηματοποίηση των δεδομένων στη μνήμη (Loop Tiling), καθώς η καλύτερη διαχείριση της cache συνοδεύεται από αυξημένο υπολογιστικό κόστος για τη διαχείριση των τμημάτων (tiles). Το ίδιο ισχύει και για τις Collapsing και Inversion ο οπολιες αυξάνουν τη πολυπλοκότητα της μνήμης και δεν παρέχουν ουσιαστικά οφέλη σε μικρούς βρόχους.

Method	Instructions	Core_Cycles	S_Cycles	N_cycles	I_Cycles	Total
Default	124825576	202541476	141262357	47033740	17361904	205658001
Unrolling	122974862	199356803	139214234	46172806	17026986	202414026
Fusion	123938976	201289498	140313968	46730530	17360777	204405275
Fission	72876396	109648524	78111744	24485273	10324445	112921462
Interchange	64917115	93341866	71926179	16844751	7881885	96652815
Tiling	125432972	203806410	141875571	47377896	17669468	206922935
Collapsing	129842808	210635938	145564779	50057333	18148159	213770271
Inversion	124827569	202546657	141265148	47035334	17362700	205663182

Πίνακας 2: Αποτελέσματα τεχνικών βελτιστοποίησης.

Method	Total	Speedup
Default	205658001	
Unrolling	202414026	102%
Fusion	204405275	101%
Fission	112921462	182%
Interchange	96652815	213%
Tiling	206922935	99%
Collapsing	213770271	96%
Inversion	205663182	100%

Πίνακας 3: Speedup.

Ο παραπάνω πίνακας παρουσιάζει τη συνολική απόδοση και το ποσοστό επιτάχυνσης (speedup) για τις διάφορες τεχνικές βελτιστοποίησης που εφαρμόστηκαν στον αρχικό αλγόριθμο. Οι τεχνικές του Unrolling και Fusion βελτίωσαν ελαφρώς την αποδοσή μας. Το loop Fission και Interchange βελτίωσαν σημαντικά την τοπικότητα των δεδομένων και την απόδοση της μνήμης, δίνοντας περίπου διπλάσιο speedup. Οι τεχνικές όπως Tiling και Collapsing δεν πέτυχαν τις αναμενόμενες βελτιώσεις, δίνοντας αρνητικό speedup, υποδεικνύοντας ότι η φύση των δεδομένων και ο αλγόριθμος δεν ευνοούν πάντα αυτές τις μεθόδους.

Μέρος 2^ο

Εισαγωγή

Στα πλαίσια του μαθήματος "Σχεδιασμός Ενσωματωμένων Συστημάτων", το παρόν project επικεντρώνεται στη βελτιστοποίηση της απόδοσης των αλγορίθμων ανίχνευσης ακμών όπως οι μέθοδοι Prewitt και Robert Cross, μέσω προσεκτικού σχεδιασμού της ιεραρχίας μνήμης και του κώδικα υλοποίησής τους.

Συγκεκριμένα, η εργασία χωρίζεται σε δύο βασικά μέρη:

Πρώτο μέρος: Υιοθετείται μια ιεραρχία μνήμης δύο επιπέδων (ROM, RAM, και Cache), με στόχο τη διερεύνηση της επίδρασης των χαρακτηριστικών της μνήμης (π.χ., χρόνοι πρόσβασης) στην απόδοση του συστήματος. Στο πλαίσιο αυτό, υλοποιούνται πειραματισμοί και μετρήσεις χρησιμοποιώντας τον προσομοιωτή ARMulator.

Δεύτερο μέρος: Διερευνώνται τεχνικές βελτίωσης της απόδοσης μέσω της χρήσης πινάκων προσωρινής αποθήκευσης (buffer) και κατευθυντήριων οδηγιών (#pragma directives), ώστε να μεγιστοποιηθεί η αποδοτική χρήση των ταχύτερων επιπέδων μνήμης (Cache).













Η εργασία αυτή προσφέρει την ευκαιρία να συνδυαστούν θεωρητικές γνώσεις για την αρχιτεκτονική των συστημάτων με πρακτικές υλοποιήσεις, αναδεικνύοντας την πολυπλοκότητα και τις προκλήσεις του σχεδιασμού σε ενσωματωμένα συστήματα. Μέσω της υλοποίησης και ανάλυσης των παραπάνω βημάτων, αναμένεται να αποκτηθεί βαθύτερη κατανόηση της σχέσης μεταξύ της αρχιτεκτονικής της μνήμης και της απόδοσης των αλγορίθμων επεξεργασίας εικόνας.

Αρχιτεκτονικές μνήμης

Οι αρχιτεκτονικές μνήμης που παρουσιάζονται στη παρούσα εργασία εφαρμόζονται στον βέλτιστο κώδικα του 1^{ου} μέρους όπου έγινε η τεχνική του loop interchange. Με βάση τις ανάγκες του προγράμματός και τα δεδομένα που χρειάζεται, σκοπός είναι να σχεδιαστεί μία **ιεραρχία μνήμης δύο επιπέδων** με **ROM** για τον κώδικα και **RAM + Cache** για τα δεδομένα.

- Αρχιτεκτονική 1 – ROM/RAM BUS = 2

Αρχικά, θα δοκιμάσουμε μια πολύ απλή αρχιτεκτονική μνήμης με μία μνήμη **ROM** και μία μνήμη **RAM**, την οποία στη συνέχεια θα χρησιμοποιήσουμε και ως βάση για να υπολογίσουμε τις βελτιώσεις που θα επιφέρουν οι όποιες αλλαγές μας. Στην **ROM** θα αποθηκεύεται ο κώδικας του προγράμματος (Read-Only Memory). Ύστερα, στη **RAM** θα πραγματοποιείται η αποθήκευση των δεδομένων (ZI Data + RW). Με βάση τα στατιστικά στοιχεία του βέλτιστου κώδικα μας από την εντολή Make μπορούμε να υπολογίσουμε τα μεγέθη και να υλοποιήσουμε την ιεραρχία των μνημών.

	Image component sizes					
	Code	RO Data	RW Data	ZI Data	Debug	
	1644	60	104	800008	8240	Object Totals
	12112	314	0	300	5780	Library Totals
	=====					
	Code	RO Data	RW Data	ZI Data	Debug	
	13756	374	104	800308	14020	Grand Totals
	=====					
	Total RO	Size(Code + RO Data)			14130	(13.80kB)
	Total RW	Size(RW Data + ZI Data)			800412	(781.65kB)
	Total ROM	Size(Code + RO Data + RW Data)			14234	(13.90kB)
	=====					

Εικόνα 15: Μέγεθος δεδομένων του προγράμματός.

Χρειάζεται, επομένως, μια μνήμη **ROM** για τον κώδικα και τα RO data (14130 bytes) αλλά και τα RW data όταν βρίσκεται σε Load Mode (14234 bytes total). Επίσης, καθίσταται απαραίτητη μια μνήμη **RAM** που σε execution mode περιέχει τα RW και ZI Data (800412 bytes). Όσον αφορά τις τιμές προσπέλασης της μνήμης, πραγματοποιήθηκε μια αναζήτηση τυπικών τιμών ταχυτήτων μνήμης και σε αυτή τη περίπτωση επιλέχθηκε μία πιο αργή παρεκδοχή. Με βάση τα παραπάνω ορίστηκαν τα αρχεία **scatter.txt** και **memory.map** ως εξής:

scatter.txt	memory.map
ROM 0x0 0x379C	00000000 0000379C ROM 4 R 1/1 1/1
{	0000379C 000C6DD4 RAM 4 RW 250/50 250/50
ROM 0x0 0x379C	
{	
*.o (+RO)	
}	
RAM 0x379C 0xC369C	
{	
* (+RW, +ZI)	
}	
}	

Στη συνέχεια, πρέπει να δηλωθούν σωστά αυτές οι δομές και στο αρχείο **stack.c** ώστε να μην επικαλύπτονται με τα υπόλοιπα δεδομένα της RAM.

```
#include <rt_misc.h>

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

    struct __initial_stackheap config;

    //config.heap_limit = 0x00724B40;

    //config.stack_limit = 0x00004000;
    //config.stack_limit = 0x00100000;

    /* works */
    config.heap_base = 0x000C6E38;
    config.stack_base = 0x0018A4D4;

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

Εικόνα 16: Δήλωση των stack & heap

Τρέχοντας το πρόγραμμα, λαμβάνονται τα εξής δεδομένα:

Debugger Internals									
Internal Variables					Statistics				
Referenc...	Instruct...	Core_Cycles	S_Cycles	N_Cycles	I_Cycles	C_Cycles	Wait_States	Total	True_Idl...
\$statistics	64917202	93342074	71926266	16844850	7881907	0	67711640	164364663	1261876

Εικόνα 17: Αποτελέσματα Αρχιτεκτονικής 1

Τα δεδομένα που μας ενδιαφέρουν είναι ο συνολικός αριθμός κύκλων (Total_Cycles) και οι καταστάσεις αναμονής για προσπέλαση της μνήμης (Wait_States).

- Αρχιτεκτονική 2 – ROM/RAM BUS = 2

Σε αυτή την αρχιτεκτονική θα πραγματοποιηθεί προσπάθεια μείωσης του μεγέθους του BUS στο μισό. Έτσι προκειμένου ο επεξεργαστής να φέρει έναν ακέραιο στη μνήμη απαιτεί 2 κύκλους ρολογιού και όχι έναν όπως στην αρχική περίπτωση. Με βάση αυτό το γεγονός, αναμένεται να υπάρξει μεγαλύτερος αριθμός Wait Cycles.

Το memory.map θα τροποποιηθεί ως εξής, καθώς το scatter.txt παραμένει το ίδιο:

```
00000000 0000379C ROM 2 R 1/1 1/1
0000379C 000C6DD4 RAM 2 RW 250/50 250/50
```

Παρακάτω φαίνονται τα αποτελέσματα της προσομοίωσης, όπου τα Wait States έχουν γίνει παραπάνω από υπερδιπλάσια της αρχικής αρχιτεκτονικής, πράγμα το οποίο επιφέρει αύξηση στον συνολικό αριθμό κύκλων.

Debugger Internals									
Internal Variables					Statistics				
Referenc...	Instruct...	Core_Cycles	S_Cycles	N_Cycles	I_Cycles	C_Cycles	Wait_States	Total	True_Idl...
\$statistics	64917202	93342074	71926266	16844850	7881907	0	165067206	261720229	1261876

Εικόνα 18: Αποτελέσματα Αρχιτεκτονικής 2

- Αρχιτεκτονική 3 – ROM/RAM FAST

Στη συνέχεια θα χρησιμοποιηθεί μια γρηγορότερη μνήμη RAM, μειώνοντας τους χρόνους ανάγνωσης/εγγραφής τόσο για τους sequential (S) όσο και για τους Non-sequential (N) κύκλους. Η αλλαγή αυτή θα γίνει μέσω του αρχείου memory.map καθώς το scatter.txt αρχείο παραμένει το ίδιο. Στην περίπτωση αυτή αναμένεται μείωση των Wait Cycles.

```
00000000 0000379C ROM 4 R 1/1 1/1
0000379C 000C6DD4 RAM 4 RW 120/40 120/40
```

Εικόνα 19: Νέο memory.map

Τα αποτελέσματα που λαμβάνονται απο τη προσομοίωση είναι τα παρακάτω:

Debugger Internals									
Internal Variables		Statistics							
Referenc...	Instruct...	Core_Cycles	S_Cycles	N_Cycles	I_Cycles	C_Cycles	Wait_States	Total	True_Idl...
\$statistics	64917202	93342074	71926266	16844850	7881907	0	28226341	124879364	1261876

Εικόνα 20: Αποτελέσματα Αρχιτεκτονικής 3

Παρατηρείται αρκετά μεγάλη μείωση στον αριθμό των καταστάσεων αναμονής, πράγμα που συμβάλει στην μείωση των συνολικών αριθμών κύκλων.

• Αρχιτεκτονική 4 – ROM/RAM + CACHE

Θα δημιουργηθεί μια ακόμη περιοχή μνήμης (με χρόνους ανάγνωσης 1/1) με σκοπό να γίνει προσομοίωση της πολύ γρήγορης cache που βρίσκεται κοντά στον επεξεργαστή. Τα αρχεία για τη δήλωση της περιοχής αυτής φαίνονται παρακάτω:

scatter.txt	memory.map
ROM 0x0 0x37A4	00000000 000037A4 ROM 4 R 1/1 1/1
{	000037A4 000C6DD4 RAM 4 RW 250/50 250/50
ROM 0x0 0x37A4	000c6e40 0004E200 CACHE 4 RW 1/1 1/1
{	
* (+RO)	
}	
RAM 0x37A4 0xc36a4	
{	
* (ram)	
* (+RW, +ZI)	
}	
CACHE 0xc6e40 0x4e200	
{	
* (cache)	
}	
}]	

Προκειμένου να καθοδηγηθεί ο compiler να τοποθετήσει τα δεδομένα του προγράμματος αποκλειστικά στη μνήμη RAM και CACHE γίνεται χρήση της δήλωσης **#pragma**, δηλώνοντας το επιθυμητό όνομα του section, το οποίο στη συνέχεια θα περαστεί στα αρχεία που περιγράφουν τη μνήμη:

```
/* Frame data */
#pragma arm section zidata="ram"
int current_y[N][M];
int current_u[N][M];
int current_v[N][M];
#pragma arm section

#pragma arm section zidata="cache"
int edge_prewitt[N][M];
int edge_roberts[N][M];
#pragma arm section
```

Τα αποτελέσματα που λαμβάνονται είναι αρκετά ικανοποιητικά, καθώς παρατηρείται μείωση των Wait_States, πράγμα που επιφέρει μείωση και στον αριθμό των συνολικών κύκλων. Το γεγονός αυτό οφείλεται στην πολύ γρήγορη μνήμη CACHE στην οποία τοποθετήσαμε τα πιο συχνά χρησιμοποιούμενα δεδομένα στον κώδικα:

Debugger Internals									
Internal Variables		Statistics							
Referenc...	Instruct...	Core_Cycles	S_Cycles	N_Cycles	I_Cycles	C_Cycles	Wait_States	Total	True_Idl...
\$statistics	64917229	93342110	71926297	16844853	7881909	0	43194452	139847511	1261876

Εικόνα 21: Αποτελέσματα Αρχιτεκτονικής 4

• Αρχιτεκτονική 5 – ROM/RAM FAST + CACHE

Τέλος, θα τοποθετηθεί η ίδια CACHE μαζί με μία όμως πιο γρήγορη RAM, όπως έγινε στην Αρχιτεκτονική 3. Αναμένονται καλύτερα αποτελέσματα με αυτήν την τεχνική.

Το νέο memory.map θα είναι το εξής:

```
00000000 000037A4 ROM 4 R 1/1 1/1
000037A4 000C6DD4 RAM 4 RW 120/40 120/40
000c6e40 0004E200 CACHE 4 RW 1/1 1/1
```

Debugger Internals									
Internal Variables		Statistics							
Referenc...	Instruct...	Core_Cycles	S_Cycles	N_Cycles	I_Cycles	C_Cycles	Wait_States	Total	True_Idl...
\$statistics	64917229	93342110	71926297	16844853	7881909	0	18010838	114663897	1261876

Εικόνα 22: Αποτελέσματα Αρχιτεκτονικής 5

Έτσι, είναι εμφανές ότι τόσο οι κύκλοι αναμονής όσο και οι συνολικοί κύκλοι παρουσιάζουν μείωση σε σχέση με τα προηγούμενα πειράματα/προτάσεις. Ο συνδιασμός με την γρήγορη RAM μαζί με την CACHE δίνει τα καλύτερα αποτελέσματα.

Χρήση Buffer στη συνέλιξη του αλγορίθμου

Για την βελτίωση της αποδοτικότητας του αλγορίθμου και τη χρησιμοποίηση buffer (προσωρινής μνήμης) για τον περιορισμό της πρόσβασης στα δεδομένα του κύριου μεγάλου πίνακα, είναι δυνατό να διαβάζεται ένα τμήμα της εικόνας (π.χ., μία σειρά ή ένα μπλοκ δεδομένων) κάθε φορά.

Ο παρακάτω κώδικας υλοποιεί μια βελτιωμένη προσέγγιση για την επεξεργασία εικόνων μέσω της χρήσης προσωρινών πινάκων (buffer) για την αποδοτικότερη εκτέλεση του αλγορίθμου σε συστήματα με ιεραρχία μνήμης. Συγκεκριμένα, η εικόνα διαιρείται σε μικρότερα τμήματα που επεξεργάζονται τοπικά μέσα σε έναν μικρό buffer μεγέθους BUFFER_SIZE. Ο buffer φορτώνει τμηματικά τις απαραίτητες γραμμές της εικόνας στην προσωρινή μνήμη, επιτρέποντας στον αλγόριθμο να μειώσει τις πολλαπλές και δαπανηρές προσπελάσεις στην κύρια μνήμη. Έτσι, επιτυγχάνεται καλύτερη αξιοποίηση της τοπικότητας των δεδομένων και της cache του συστήματος.

Στη συνάρτηση `apply_prewitt_with_buffer`, η χρήση buffer επιτρέπει την τοπική εφαρμογή των Prewitt μασκών (Gx και Gy) για την ανίχνευση ακμών. Το πρόγραμμα φορτώνει τμηματικά BUFFER_SIZE γραμμές της εικόνας στον προσωρινό πίνακα και εκτελεί τους υπολογισμούς για τον εντοπισμό ακμών σε μικρότερες περιοχές. Για κάθε pixel του buffer, υπολογίζονται οι τιμές gx και gy μέσω των Prewitt μασκών και κατόπιν εξάγεται το συνολικό μέτρο ακμής μέσω του υπολογισμού του μεγέθους του διανύσματος $\sqrt{gx^2 + gy^2}$. Αντίστοιχα, στη συνάρτηση `apply_roberts_with_buffer` εφαρμόζονται οι Roberts μάσκες σε τμήματα της εικόνας που φορτώνονται τοπικά στον buffer, ακολουθώντας παρόμοια λογική.

Η χρήση του buffer παρέχει σημαντικά πλεονεκτήματα, καθώς μειώνει τη συχνότητα πρόσβασης στην κύρια μνήμη και εκμεταλλεύεται καλύτερα την ιεραρχία μνήμης και την cache-friendly σχεδίαση. Η τοπική επεξεργασία εντός του buffer εξασφαλίζει ότι κάθε τμήμα της εικόνας διαβάζεται μία φορά, ενώ οι επαναλαμβανόμενες προσπελάσεις πραγματοποιούνται στον ταχύτερο buffer αντί για την κύρια μνήμη. Με τον τρόπο αυτό, ο προτεινόμενος αλγόριθμος είναι αποδοτικότερος και ιδανικός για την επεξεργασία μεγάλων εικόνων σε συστήματα με περιορισμένο μέγεθος cache.

```

/* Apply Prewitt Operator using Buffer */
void apply_prewitt_with_buffer()
{
    int gx, gy;

    for ( j = 1; j < M - 1; j += BUFFER_SIZE - 2)
    {
        /* Load BUFFER_SIZE rows into buffer */
        for ( row = 0; row < BUFFER_SIZE && (j + row - 1) < M; row++)
        {
            memcpy(buffer[row], current_y[j + row - 1], M * sizeof(int));
        }

        for ( row = 1; row < BUFFER_SIZE - 1 && (j + row) < M - 1; row++)
        {
            for ( i = 1; i < N - 1; i++)
            {
                gx = 0;
                gy = 0;

                for ( y = 0; y < 3; y++)
                {
                    for ( x = 0; x < 3; x++)
                    {
                        gx += buffer[row + x - 1][i + y - 1] * Gx_Prewitt[x][y];
                        gy += buffer[row + x - 1][i + y - 1] * Gy_Prewitt[x][y];
                    }
                }

                edge_prewitt[j + row][i] = (int)sqrt(gx * gx + gy * gy);

                if (edge_prewitt[j + row][i] > 255)
                    edge_prewitt[j + row][i] = 255;
                else if (edge_prewitt[j + row][i] < 0)
                    edge_prewitt[j + row][i] = 0;
            }
        }
    }
}

```

Εικόνα 23: Χρήση buffer στην εφαρμογή του Prewitt

Προκειμένου να επιλεχτεί η περιοχή της μνήμης στην οποία θα αποθηκευτεί ο buffer γίνεται χρήση της δήλωσης **#pragma**, δηλώνοντας το αρχείο του buffer μέσα στη cache:

```

#pragma arm section zidata="ram"
int current_y[N][M];
int current_u[N][M];
int current_v[N][M];
#pragma arm section

#pragma arm section zidata="cache"
int edge_prewitt[N][M];
int edge_roberts[N][M];
int buffer[BUFFER_SIZE][M];
#pragma arm section

```

Τα συνολικά στατιστικά για τα μεγέθη των μνημών που παράγονται από το Make μετά την χρήση των **buffers** φαίνονται παρακάτω:

Image component sizes						
	Code	RO Data	RW Data	ZI Data	Debug	
	2036	60	104	812820	8440	Object Totals
	12220	314	0	300	5864	Library Totals
=====						
	Code	RO Data	RW Data	ZI Data	Debug	
	14256	374	104	813120	14304	Grand Totals
=====						
	Total RO	Size(Code + RO Data)			14630	(14.29kB)
	Total RW	Size(RW Data + ZI Data)			813224	(794.16kB)
	Total ROM	Size(Code + RO Data + RW Data)			14734	(14.39kB)
=====						

Εικόνα 24: Μέγεθος δεδομένων με χρήση *buffer*

Τροποποιούνται τώρα κατάλληλα τα αρχεία `scatter.txt`, `memory.map` και `stack.c` για να δηλωθεί η νέα μνήμη **CACHE** με καλύτερους χρόνους ανάγνωσης και εγγραφής:

scatter.txt	memory.map
ROM 0x0 0x3994	00000000 00003994 ROM 4 R 1/1 1/1
{	00003994 000CA23C RAM 4 RW 250/50 250/50
ROM 0x0 0x3994	000CA23C 00051400 CACHE 4 RW 1/1 1/1
{	
*(+RO)	
}	
RAM 0x3994 0xCA23C	
{	
*(ram)	
*(+RW, +ZI)	
}	
CACHE 0xCA23C 0x51400	
{	
*(cache)	
}	
}	

```

#include <rt_misc.h>

__value_in_regs struct __initial_stackheap __user_initial_stackheap(
unsigned R0, unsigned SP, unsigned R2, unsigned SI){

struct __initial_stackheap config;

//config.heap_limit = 0x00724B40;

//config.stack_limit = 0x00004000;
//config.stack_limit = 0x00100000;

/* works */
config.heap_base = 0x0011B638;
config.stack_base = 0x001E1EE0;

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

```

Εικόνα 25: Δήλωση των stack & heap

Λαμβάνοντας τα αποτελέσματα του Debugger Internals, παρατηρείται μια σαφή βελτίωση με την χρήση buffers σε σχέση με την πρώτη υλοποίηση που πραγματοποιήθηκε:

Debugger Internals									
Internal Variables Statistics									
Referenc...	Instruct...	Core_Cycles	S_Cycles	N_Cycles	I_Cycles	C_Cycles	Wait_States	Total	True_Idl...
\$statistics	66545368	97273994	72559492	19537637	8478795	0	59403114	159979038	1257717

Με την χρήση μιας ακόμα πιο γρήγορης RAM (120/40) υπάρχουν ακόμη καλύτερα αποτελέσματα:

Debugger Internals									
Internal Variables Statistics									
Referenc...	Instruct...	Core_Cycles	S_Cycles	N_Cycles	I_Cycles	C_Cycles	Wait_States	Total	True_Idl...
\$statistics	66545368	97273994	72559492	19537637	8478795	0	24762338	125338262	1257717

Έτσι, συμπεραίνεται πλέον ότι τόσο οι κύκλοι αναμονής όσο και οι συνολικοί κύκλοι παρουσιάζουν εμφανέστατη μείωση σε σχέση με τα προηγούμενα πειράματα/προτάσεις.

Συνολικές Μετρικές των μεθόδων αρχιτεκτονικής μνήμης

Παρακάτω παρουσιάζεται μια σύντομη αποτίμηση των παραπάνω προτάσεων:

	Wait_States	Βελτίωση	Total Cycles	Βελτίωση
Αρχιτεκτονική 1	67711640		164364663	
Αρχιτεκτονική 2	165067206	-144%	261720229	-59%
Αρχιτεκτονική 3	28226341	58%	124879364	24%
Αρχιτεκτονική 4	43194452	36%	139847511	15%
Αρχιτεκτονική 5	18010838	73%	114663897	30%
Buffer 1	59403114	12%	159979038	3%
Buffer 2	24762338	63%	125338262	24%

Πίνακας 3: Αποτελέσματα αρχιτεκτονικών μνήμης

Συμπερασματικά αν προκληθεί μείωση του μεγέθους του διαύλου BUS, υπάρχει αρνητική επίδραση στην απόδοση του κώδικα, ενώ μια γρηγορότερη μνήμη βελτιώνει τα τελικά αποτελέσματα αναφορικά με τον αριθμό των Wait States αλλά και τον συνολικό αριθμό κύκλων του προγράμματος. Η προσθήκη buffer προκαλεί επίσης βελτίωση στην απόδοση, αλλά για καλύτερα αποτελέσματα θα πρέπει να συνδιαστεί και με τη χρήση μίας γρηγορότερης RAM.

Μέρος 3^ο

Εισαγωγή

Το τρίτο μέρος της εργασίας αποσκοπεί στην εξέταση της βελτίωσης της απόδοσης του αλγορίθμου ανιχνευσης ακμών κατά Prewitt και Robert Cross, μέσω της χρήσης πινάκων προσωρινής αποθήκευσης (buffer) και της ιεραρχίας μνήμης σε συστήματα υπολογιστών. Ο αλγόριθμος που θα εξετάσουμε έχει ήδη υλοποιηθεί στο πλαίσιο προηγούμενου μέρους του project, και ο στόχος μας είναι να εξετάσουμε την αποδοτικότητα του αλγορίθμου με τη χρήση διαφορετικών στρατηγικών αποθήκευσης δεδομένων στη μνήμη, όπως η αποθήκευση στους buffer και οι διάφορες ακολουθίες αντιγραφής δεδομένων.

Στην παρούσα εργασία, προτείνεται η εισαγωγή πινάκων buffer για τη βελτίωση της εκτέλεσης του αλγορίθμου, καθώς και η αναλυτική περιγραφή των πιθανών ακολουθιών αντιγραφής δεδομένων που μπορούν να εφαρμοστούν. Πιο συγκεκριμένα, θα εξετάσουμε δύο διαφορετικές στρατηγικές για την αντιγραφή των δεδομένων από τη μνήμη RAM στους buffer, συγκρίνοντας τις επιπτώσεις τους στην ταχύτητα εκτέλεσης και τον αριθμό των προσπελάσεων στους πίνακες δεδομένων.

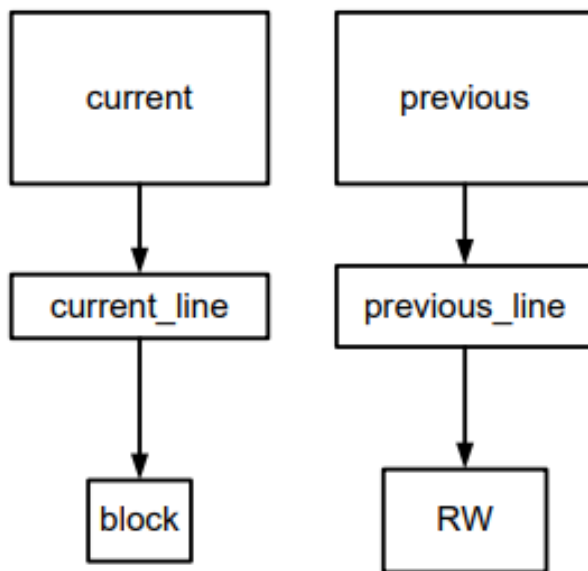
Επιπλέον, προτείνεται μια ιεραρχία μνήμης για την αποδοτικότερη εκτέλεση του προγράμματος, η οποία περιλαμβάνει μνήμη ROM για τον κώδικα και μνήμη δεδομένων τριών επιπέδων (RAM και δύο επίπεδα Cache). Οι buffer θα τοποθετηθούν στους κατάλληλους χώρους μνήμης, ώστε να μεγιστοποιηθεί η απόδοση του συστήματος, με την L1Cache να φιλοξενεί τους μικρότερους και ταχύτερους buffer, ενώ η L2Cache τους μεγαλύτερους buffer.

Η σύγκριση θα γίνει μεταξύ τριών εκδόσεων του αλγορίθμου: (i) αυτής που δεν κάνει χρήση buffer και αποθηκεύει τα δεδομένα αποκλειστικά στη RAM, (ii) αυτής που χρησιμοποιεί buffer με μία από τις δύο στρατηγικές αντιγραφής δεδομένων, και (iii) αυτής που χρησιμοποιεί την άλλη στρατηγική. Η ανάλυση της απόδοσης θα γίνει μέσω μετρήσεων που αφορούν την ταχύτητα εκτέλεσης του αλγορίθμου και τον αριθμό προσπελάσεων σε κάθε πίνακα.

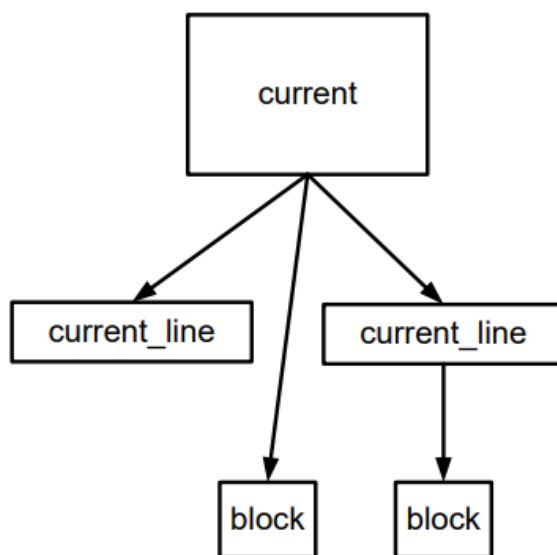
Η εργασία αυτή, μέσω της εφαρμογής αυτών των στρατηγικών, επιδιώκει να προσφέρει μια βαθύτερη κατανόηση του τρόπου με τον οποίο η διαχείριση της μνήμης μπορεί να επηρεάσει την απόδοση ενός αλγορίθμου, με στόχο τη βελτιστοποίηση της εκτέλεσης σε περιβάλλοντα με περιορισμένους πόρους, όπως αυτά των ενσωματωμένων συστημάτων.

Επαναχρησιμοποίηση Δεδομένων

Η εισαγωγή των buffers στον αλγόριθμό μας εξυπηρετεί την επαναχρησιμοποίηση δεδομένων, μειώνοντας τις προσπελάσεις στη μνήμη RAM και βελτιστοποιώντας την απόδοση, καθώς τα δεδομένα μπορούν να ανακτηθούν τοπικά από τους buffers, οι οποίοι βρίσκονται σε ταχύτερες μνήμες (π.χ. L1Cache ή L2Cache). Στο σημείο αυτό θα προτείνουμε δύο διαφορετικές ακολουθίες αντιγραφής δεδομένων.



Σχήμα 5: Στο σχήμα αυτό, χρησιμοποιείται ένας buffer γραμμής και ένας buffer για μικρά μπλοκ δεδομένων. Οι γραμμές δεδομένων φορτώνονται σταδιακά και ανακυκλώνονται σε κάθε βήμα του αλγορίθμου.



Σχήμα 6: Σε αυτό το σχήμα, δύο διαδοχικές γραμμές δεδομένων αποθηκεύονται σε buffers και χρησιμοποιούνται για τον υπολογισμό. Το σχέδιο αυτό ενισχύει την τοπικότητα δεδομένων και βελτιώνει την πρόσβαση στη μνήμη.

Σε αυτό το κομμάτι θα πραγματοποιηθεί υλοποίηση των δύο ακολουθιών πάνω στον αλγόριθμο ανίχνευσης ακμών.

Παρακάτω φαίνεται η εισαγωγή των buffers στον κώδικα για τον αλγόριθμο Prewitt, όπως φαίνεται στη συνάρτηση `apply_prewitt_with_buffers`, η οποία βασίζεται στο Σχήμα 5.

```
/* Apply Prewitt Operator using buffers */
void apply_prewitt_with_buffers()
{
    int x, y, gx, gy;

    for (j = 1; j < M - 1; j++)
    {
        // Load one line of current_y into buffer_current_line
        for (i = 0; i < N; i++)
        {
            buffer_current_line[i] = current_y[i][j];
        }

        for (i = 1; i < N - 1; i++)
        {
            gx = 0;
            gy = 0;

            // Load 3x3 block into buffer_block
            for (y = 0; y < 3; y++)
            {
                for (x = 0; x < 3; x++)
                {
                    buffer_block[x][y] = current_y[i + x - 1][j + y - 1];
                }
            }

            for (y = 0; y < 3; y++)
            {
                for (x = 0; x < 3; x++)
                {
                    gx += buffer_block[x][y] * Gx_Prewitt[x][y];
                    gy += buffer_block[x][y] * Gy_Prewitt[x][y];
                }
            }

            edge_prewitt[i][j] = (int)sqrt(gx * gx + gy * gy);

            if (edge_prewitt[i][j] > 255)
                edge_prewitt[i][j] = 255;
            else if (edge_prewitt[i][j] < 0)
                edge_prewitt[i][j] = 0;
        }
    }
}
```

Αρχικά τα δεδομένα ανακτώνται από τον buffer μίας γραμμής (`buffer_current_line`) και χρησιμοποιείται για την αποθήκευση μιας μόνο γραμμής από τον πίνακα `current_y`. Δηλαδή, για κάθε γραμμή `j` του πίνακα `current_y`, όλα τα στοιχεία της γραμμής αποθηκεύονται στον `buffer_current_line`. Έτσι, η γραμμή γίνεται προσβάσιμη από τη γρηγορότερη μνήμη (π.χ., L2Cache), μειώνοντας τις προσπελάσεις στη RAM. Ύστερα για κάθε στοιχείο `current_y[i][j]` που επεξεργάζεται ο αλγόριθμος, τα 9 γειτονικά στοιχεία (σχηματίζοντας μπλοκ 3x3) φορτώνονται σε μία τοπική δομή, τον `buffer_block`. Μετέπειτα εκτελείται το convolution και

αποθηκεύονται τα αποτελέσματα. Αντίστοιχα, η υλοποίηση του Roberts Operator με χρήση buffer `buffer_block_roberts` λειτουργεί με παρόμοιο τρόπο, αλλά το μέγεθος του μπλοκ είναι 2x2 αντί για 3x3.

Προκειμένου να επιλεχτεί η περιοχή της μνήμης στην οποία θα αποθηκευτούν οι buffers γίνεται χρήση της δήλωσης **#pragma**, δηλώνοντας το αρχείο του buffer current line μέσα στη `cache_L2` και το buffer block στη `cache_L1`:

```
/* Frame data */
#pragma arm section zidata="ram"
int current_y[N][M];
int current_u[N][M];
int current_v[N][M];
int edge_prewitt[N][M];
int edge_roberts[N][M];
#pragma arm section

int i, j;

/* Buffers */
#pragma arm section zidata="cache_L2"
int buffer_current_line[M]; // Buffer for one line of current frame
#pragma arm section

#pragma arm section zidata="cache_L1"
int buffer_block[3][3]; // Buffer for a 3x3 block (Prewitt)
int buffer_block_roberts[2][2]; // Buffer for a 2x2 block (Roberts)
#pragma arm section
```

Τροποποιούνται τώρα κατάλληλα τα αρχεία `scatter.txt`, `memory.map` και `stack.c` για να δηλωθεί η νέα μνήμη **CACHE** με καλύτερους χρόνους ανάγνωσης και εγγραφής:

scatter.txt	memory.map
ROM 0x0 0x3994	00000000 00003994 ROM 4 R 1/1 1/1
{	00003994 000CA23C RAM 4 RW 250/50 250/50
ROM 0x0 0x3994	000CA23C 00002000 CACHE 4 RW 30/10 30/10
{	000CC23C 000000F0 CACHE2 4 RW 1/1 1/1
*(+R0)	
}	
RAM 0x3994 0xCA23C	
{	
*(ram)	
*(+RW, +ZI)	
}	
CACHE 0xCA23C 0x2000	
{	
*(cache_L2)	
}	
CACHE2 0xCC23C 0xF0	
{	
*(cache_L1)	
}	
}	

Παρακάτω φαίνεται η εισαγωγή των buffers στον κώδικα για τον αλγόριθμο Prewitt, όπως φαίνεται στη συνάρτηση apply_prewitt, η οποία βασίζεται στο Σχήμα 6.

```

/* Apply Prewitt Operator using buffers (Schema 6) */
void apply_prewitt_schema_6()
{
    int x, y, gx, gy;

    for (j = 1; j < M - 1; j += 2)
    {
        // Load two rows into row buffers
        for (i = 0; i < N; i++)
        {
            buffer_row_1[i] = current_y[i][j - 1];
            buffer_row_2[i] = current_y[i][j];
        }

        for (i = 1; i < N - 1; i++)
        {
            gx = 0;
            gy = 0;

            // Load 2x3 block into buffer_block_2x3
            for (y = 0; y < 2; y++)
            {
                for (x = 0; x < 3; x++)
                {
                    if (y == 0)
                        buffer_block_2x3[y][x] = buffer_row_1[i + x - 1];
                    else
                        buffer_block_2x3[y][x] = buffer_row_2[i + x - 1];
                }
            }

            for (y = 0; y < 2; y++)
            {
                for (x = 0; x < 3; x++)
                {
                    gx += buffer_block_2x3[y][x] * Gx_Prewitt[x][y];
                    gy += buffer_block_2x3[y][x] * Gy_Prewitt[x][y];
                }
            }

            edge_prewitt[i][j] = (int)sqrt(gx * gx + gy * gy);

            if (edge_prewitt[i][j] > 255)
                edge_prewitt[i][j] = 255;
            else if (edge_prewitt[i][j] < 0)
                edge_prewitt[i][j] = 0;
        }
    }
}

```

Οι γραμμές δεδομένων αποθηκεύονται σε δύο buffers, τον buffer_row_1 και buffer_row_2. Ο buffer_row_1 αποθηκεύει τις γραμμές j-1 του πίνακα current_y και ο buffer_row_2 τις γραμμές j. Αυτό επιτρέπει τη φόρτωση δύο διαδοχικών γραμμών από τη RAM στη μνήμη cache, βελτιώνοντας την τοπικότητα των δεδομένων. Μετέπειτα, το μπλοκ 2x3 φορτώνεται στον buffer buffer_block_2x3 από τους buffers γραμμών. Τα δεδομένα της πρώτης γραμμής (buffer_row_1) αντιστοιχούν στην πρώτη σειρά του μπλοκ, ενώ της δεύτερης γραμμής (buffer_row_2) στη δεύτερη σειρά. Το μπλοκ αυτό χρησιμοποιείται για τους υπολογισμούς εντός του παραθύρου του Prewitt Operator.

Η ίδια λογική εφαρμόζεται στη συνάρτηση `apply_roberts`, όπου οι δύο διαδοχικές γραμμές χρησιμοποιούνται για να γεμίσουν το μπλοκ 2x2 (`buffer_block_2x2`).

Με τη χρήση της δήλωσης `#pragma`, δηλώνονται το αρχείο του `buffer_row` μέσα στη `cache_L2` και τα `buffer block` στη `cache_L2`:

```
/* Frame data */
#pragma arm section zidata="ram"
int current_y[N][M];
int current_u[N][M];
int current_v[N][M];
int edge_prewitt[N][M];
int edge_roberts[N][M];
#pragma arm section

int i, j;

/* Buffers */
#pragma arm section zidata="cache_L2"
int buffer_row_1[M]; // Buffer for the first row of a 2-row window
int buffer_row_2[M]; // Buffer for the second row of a 2-row window
#pragma arm section

#pragma arm section zidata="cache_L1"
int buffer_block_2x3[2][3]; // Buffer for a 2x3 block (Prewitt)
int buffer_block_2x2[2][2]; // Buffer for a 2x2 block (Roberts)
#pragma arm section
```

Μετρικές

Για να γίνει σύγκριση των δύο ακολουθιών αντιγραφής δεδομένων θα χρησιμοποιηθεί ο αλγόριθμος που υλοποιήθηκε στο πρώτο μέρος της εργασίας στον οποίο δεν γίνεται χρήση buffer και όλα τα δεδομένα τοποθετούνται αποκλειστικά στη μνήμη RAM.

Τα αποτελέσματα απο το τρέξιμο του κώδικα χωρίς buffers είναι τα εξής:

Debugger Internals									
Internal Variables					Statistics				
Referenc...	Instruct...	Core_Cycles	S_Cycles	N_Cycles	I_Cycles	C_Cycles	Wait_States	Total	True_Idl...
\$statistics	64917202	93342074	71926266	16844850	7881907	0	45948068	142601091	1261876

Τα αποτελέσματα με τη χρήση buffer που βασίζεται στην ακολουθία αντιγραφής δεδομένων του σχήματος 5 είναι τα παρακάτω:

Debugger Internals									
Internal Variables					Statistics				
Referenc...	Instruct...	Core_Cycles	S_Cycles	N_Cycles	I_Cycles	C_Cycles	Wait_States	Total	True_Idl...
\$statistics	70079023	100992261	76929096	19095005	8318908	0	62041112	166384121	1261876

Με την χρήση counters γίνεται μέτρηση του αριθμού των προσπελάσεων που γίνονται σε καθέναν από τους πίνακες σας των buffer

```
Memory access counts:  
buffer_current_line_access: 79400  
buffer_block_3x3_access: 352836  
buffer_block_2x2_access: 158404
```

L2: 79400 προσπελάσεις L1: 511240 προσπελάσεις

Στη συνέχεια ακολουθούν τα δεδομένα που βασίζονται στο σχήμα 6:

Debugger Internals									
Internal Variables					Statistics				
Referenc...	Instruct...	Core_Cycles	S_Cycles	N_Cycles	I_Cycles	C_Cycles	Wait_States	Total	True_Idl...
\$statistics	51440243	75361606	56487219	15084332	5740627	0	44415608	121727786	744160

Πάλι χρησιμοποιώντας μετρητές, υπολογίζονται οι προσπελάσεις κάθε φορά που γίνεται ανάγνωση απο τα περιεχόμενα των πινάκων buffer.

```
Memory access counts:  
buffer_row_1_access: 78606  
buffer_row_2_access: 78606  
buffer_block_2x3_access: 117612  
buffer_block_2x2_access: 158404
```

L2: 157212 προσπελάσεις L1: 276016

Συμπεράσματα

Η σύγκριση των αποτελεσμάτων που προέκυψαν από τις τρεις διαφορετικές υλοποιήσεις αναδεικνύει τη σημασία της χρήσης buffers στη βελτίωση της απόδοσης, καθώς και τις προκλήσεις που συνδέονται με τη βέλτιστη διαχείριση της μνήμης. Στην πρώτη περίπτωση, όπου δεν χρησιμοποιήθηκαν καθόλου buffers, ο αριθμός των κύκλων επεξεργασίας ήταν αρκετά υψηλός υψηλότερος (142601091 cycles). Η έλλειψη buffers οδηγεί σε συνεχείς προσπελάσεις στη μνήμη για κάθε υπολογισμό, γεγονός που προκαλεί αυξημένο κόστος σε χρόνους καθυστέρησης (latency). Οι συνολικές προσπελάσεις ήταν ιδιαίτερα αυξημένες, καθώς κάθε στοιχείο των πινάκων αναγνώστηκε ή γράφτηκε απευθείας από τη μνήμη.

Η δεύτερη περίπτωση βασίζεται στη χρήση buffers σύμφωνα με το σχήμα 5, όπου δεν παρατηρήθηκε μείωση στον αριθμό των κύκλων επεξεργασίας (163987961 cycles). Το αυξημένο κόστος λόγω της συχνής πρόσβασης στα τοπικά buffers (511240 προσπελάσεις L1 και 79400 προσπελάσεις στο L2) φαίνεται να εξουδετέρωσε το όφελος. Αυτό υποδεικνύει ότι το σχεδιαστικό αυτό μοντέλο δεν εκμεταλλεύεται πλήρως τη δυνατότητα βελτίωσης μέσω τοπικής αποθήκευσης.

Η τρίτη περίπτωση, που βασίστηκε στο σχήμα 6, αποδείχθηκε η πλέον αποδοτική, με τον συνολικό αριθμό κύκλων επεξεργασίας να μειώνεται στις 121727786 cycles, παρουσιάζοντας βελτίωση 117% σε σχέση με την αρχική υλοποίηση. Η χρήση περισσότερων και πιο εξειδικευμένων buffers, όπως τα `buffer_row_1` και `buffer_row_2`, σε συνδυασμό με την οργανωμένη φόρτωση δεδομένων σε μπλοκ, οδήγησε σε μείωση των προσπελάσεων L2 σε 157212 και L1 σε 276016. Η προσέγγιση αυτή εκμεταλλεύεται αποτελεσματικά την τοπικότητα δεδομένων και ελαχιστοποιεί τις δαπανηρές προσβάσεις στη μνήμη.

Συμπερασματικά, η ανάλυση καταδεικνύει ότι η χρήση buffers είναι αποτελεσματική μόνο όταν σχεδιάζεται με τρόπο που να ελαχιστοποιεί όχι μόνο τις προσπελάσεις στη μνήμη υψηλής καθυστέρησης αλλά και το κόστος πρόσβασης στα ίδια τα buffers. Παρά την περιορισμένη αποδοτικότητα της δεύτερης υλοποίησης, η υλοποίηση βάσει του σχήματος 6 ξεχωρίζει ως η πιο αποδοτική προσέγγιση, προσφέροντας μια ισορροπία μεταξύ χαμηλών προσπελάσεων μνήμης και ταχύτητας εκτέλεσης.

Βιβλιογραφία

- [1] https://en.wikipedia.org/wiki/Prewitt_operator
- [2] https://en.wikipedia.org/wiki/Roberts_cross
- [3] https://www.tutorialspoint.com/dip/prewitt_operator.htm
- [4] <https://homepages.inf.ed.ac.uk/rbf/HIPR2/roberts.htm>
- [5] <https://levelup.gitconnected.com/c-programming-hacks-4-matrix-multiplication-are-we-doing-it-right-21a9f1cbf53>
- [6] Διαφάνειες εργαστηριακών διαλέξεων του μαθήματος.