"Σχεδιασμός Ενσωματωμένων Συστημάτων"

*Δημοκρίτειο Πανεπιστήμιο Θράκης*

ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ

ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

*«Ανιχνευτής ακμών κατά Prewitt και Robert cross*

*στην επεξεργασία εικόνας»*

*Μέρος 1ο*

**ΟΜΑΔΑ 48**

*Φαίδρα Μπογιάνογλου Βραχνού ΑΜ:58422*

***-    Ξάνθη, 2024    -***

*Μέρος 1ο*

**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 ΥUV 4:4:4, όπου θα επεξεργαστούμε το κανάλι φωτεινότητας (Y). Οι συγκεκριμένοι αλγόριθμοι εφαρμόζουν διαφορικές 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.

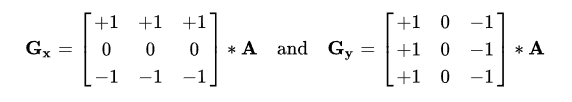


**Εικόνα 1:** Σύγκριση των μεθόδων Prewitt και Robert Cross.

**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**

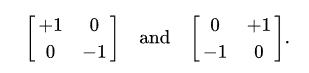


* + Ο αλγόριθμος Prewitt χρησιμοποιεί δύο μάσκες 3×3 για τον υπολογισμό των παραγώγων κατά τις διευθύνσεις (οριζόντια) και (κατακόρυφη).
  + Οι μάσκες ​ και ​ εφαρμόζονται ξεχωριστά μέσω συνέλιξης, και το συνολικό μέτρο του gradient υπολογίζεται ως:



* + Το αποτέλεσμα κλιμακώνεται ώστε να βρίσκεται στο εύρος τιμών [0, 255].

1. **Roberts Cross**

****

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

**Βασικά Βήματα Υλοποίησης**

1. **Φόρτωση Εικόνας:** Η εικόνα εισόδου διαβάζεται σε μορφή YUV 4:4:4 και επεξεργάζεται το κανάλι φωτεινότητας .
2. **Συνέλιξη με Μάσκες:** Οι μάσκες ​ και ​ εφαρμόζονται μέσω συνελίξεων, λαμβάνοντας υπόψη την περιοχή του kernel.
3. **Υπολογισμός Gradient:** Το συνολικό μέτρο του gradient υπολογίζεται και τα αποτελέσματα ομαλοποιούνται.
4. **Αποθήκευση Αποτελεσμάτων:** Το κανάλι με τις ανιχνευμένες ακμές αποθηκεύεται μαζί με τα κανάλια και , ώστε η έξοδος να είναι συμβατή με τη μορφή YUV.

**Συνοπτική Σύγκριση**

|  |  |  |
| --- | --- | --- |
| **Χαρακτηριστικά** | **Prewitt** | **Robert Cross** |
| **Μέγεθος Kernel** | 3x3 | 2x2 |
| **Πολυπλοκότητα** | Περισσοτεροι υπολογισμοί λόγο μεγαλύτερης μάσκας | Λιγότεροι υπολογισμοί |
| **Ευαισθησία** | Λιγότερο ευαίσθητο στις λεπτομέρεις | Περισσότερο ευαίσθητο στις λεπτομέρεις |

**Πίνακας 1:** Σύγκριση μεταξύ των χαρακτηριστικών των δύο φίλτρων

**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, και ​.

* Αρχικά διατρέχουμε το κάθε pixel της εικόνας (εκτός από τα όρια) με τις δύο εξωτερικές λούπες.
* Ορίζουμε δύο τιμές, και , για την οριζόντια και την κατακόρυφη κλίση.
* Χρησιμοποιούμε δύο εσωτερικούς βρόχους και κάνουμε convolution.
* Το συνολικό μέτρο της κλίσης υπολογίζεται ως
* Το αποτέλεσμα περιορίζεται στις τιμές , που είναι το εύρος των τιμών για εικόνες 8-bit σε grayscale.

1. **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 2×2.

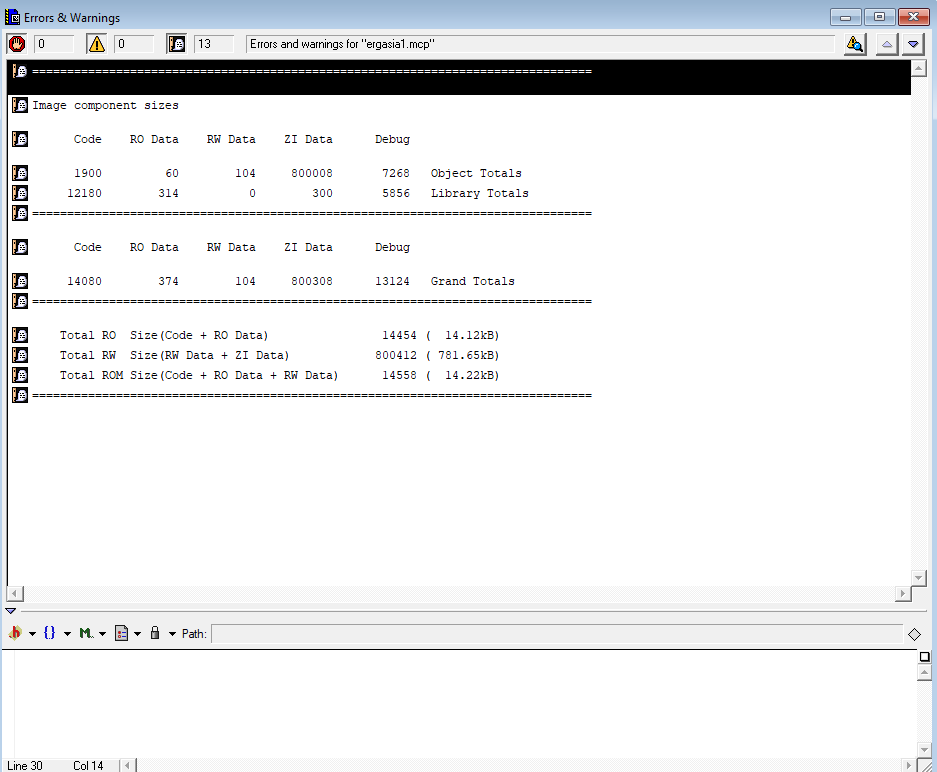
* Οι εξωτερικοί βρόχοι (for (i, j)) διατρέχουν τα pixel της εικόνας, εξαιρώντας την τελευταία γραμμή και στήλη, λόγω του μεγέθους των πυρήνων.
* Χρησιμοποιούνται πυρήνες ​ και ​ μεγέθους 2×2 για τον υπολογισμό των τιμών και .
* Όπως και στον Prewitt, το μέτρο της κλίσης υπολογίζεται ως
* Τα αποτελέσματα περιορίζονται επίσης στις τιμές

****

**Εικόνα 2:** Σύγκριση των μεθόδων Prewitt και Robert Cross.

**Μέθοδοι Βελτιστοποίησης**

**Αρχική υλοποίηση**

****

****

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

Σε αυτό το σημείο, είναι απαραίτητο να εξηγήσουμε τη σημασία 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, εξετάσαμε 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 αναλύσουμε τις όλες αυτές τεχνικές και τις επιπτώσεις 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, τροποποιήσαμε 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 μειώνουμε 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*      }  } |

Tροποποιώντας τον βρόχο μας, 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 επεξεργαζόμαστε δύο γειτονικά pixels ταυτόχρονα μέσα σε έναν βρόχο, αντί να επεξεργαζόμαστε κάθε pixel σε ξεχωριστό γύρο. Έτσι μειώνουμε το κόστος των επαναλήψεων, όπως οι έλεγχοι συνθηκών του βρόχου (j < M - 1) και η αύξηση του δείκτη j, με το να εκτελούμε περισσότερο έργο σε κάθε γύρο του βρόχου.

****

**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 εφαρμόζουμε ενοποίηση στους βρόχους ανάγνωσης και εγγραφης, 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.



**Εικόνα 4:** Αποτελέσματα loop fusion

1. 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.



**Εικόνα 4:** Αποτελέσματα loop fission

1. 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:

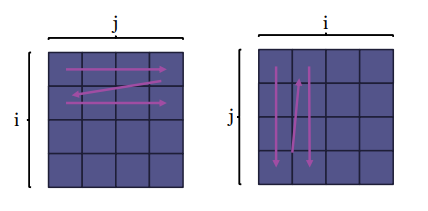
|  |
| --- |
| for (j **=** 0; j **<** M; j**++**) {    for (i **=** 0; i **<** N; i**++**) {      current\_y[i][j] **=** fgetc(frame\_c);    }  } |

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

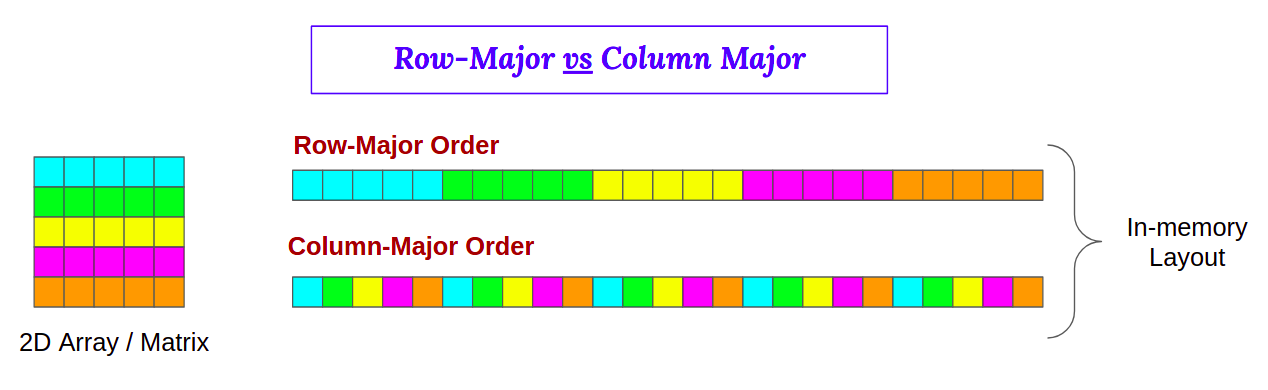
Το ίδιο πραγματοποιήθηκε και στην εφαρμογή φίλτρων Prewitt και Roberts:

|  |
| --- |
| 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*    }  } |

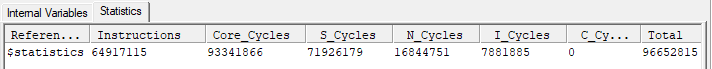
Με το loop interchange πετυγχαίνουμε τη βελτίωση προσπέλασης της μνήμης (memory access), καθώς oι σύγχρονες CPUs λειτουργούν αποδοτικότερα όταν τα δεδομένα που χρησιμοποιούνται βρίσκονται διαδοχικά στη μνήμη (spatial locality). Επίσης, τα δεδομένα του πίνακα αποθηκεύονται στη μνήμη κατά γραμμές, πράγμα 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 5:** Σχηματική αποτύπωση του loop interchange



**Εικόνα 5:** Row Major vs Column Major στον χώρο της μνήμης



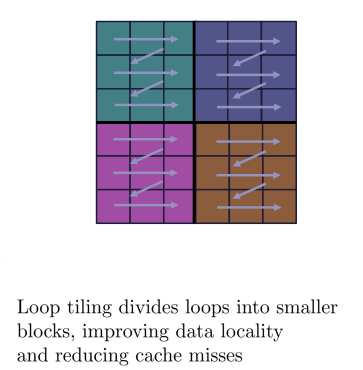
**Εικόνα 6:** Αποτελέσματα loop interchange

1. Loop tilling

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 επεξεργαζόμαστε ολόκληρο τον πίνακα 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 γλιτώνουμε 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**++**)          {  *//εφαρμογη φιλτρου*          }        }      }    }  } |

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.



**Εικόνα 7:** Σχηματική αποτύπωση του loop tilling

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



**Εικόνα 8:** Αποτελέσματα loop tilling

1. 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).



**Εικόνα 9:** Αποτελέσματα loop collapsing

1. 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 μας, μπορούμε να εφαρμόσουμε **loop inversion** αλλάζοντας τον τρόπο που γίνεται ο έλεγχος των συνθηκών στα for loops.

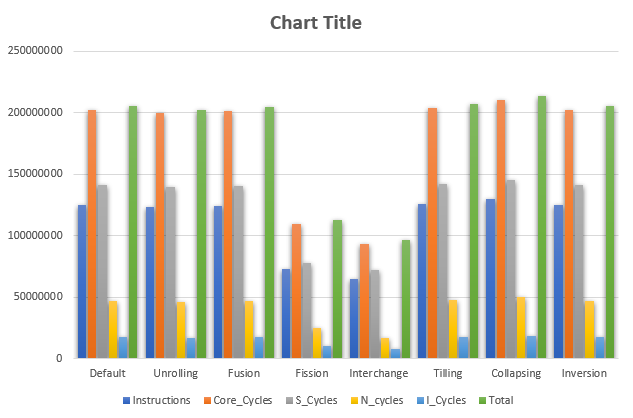
|  |
| --- |
| void apply\_prewitt()  {    int x, y, gx, gy;    i **=** 1; *// Αρχικοποίηση εκτός του βρόχου*    while (i **<** N **-** 1) *// Αντί του `for (i = 1; i < N - 1; i++)`*    {      j **=** 1; *// Αρχικοποίηση εκτός του εσωτερικού βρόχου*      while (j **<** M **-** 1) *// Αντί του `for (j = 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 εφαρμόζουμε 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, κάνουμε τις αρχικοποιήσεις (i = 1, j = 1) πριν από τον αντίστοιχο βρόχο και μεταφέρουμε την ενημέρωση (i++, j++) στο τέλος του σώματος του βρόχου. Με τον τρόπο αυτό χρησιμοποιούμε σαφή ροή, αφού ο βρόχος δεν εξαρτάται από ενσωματωμένες δομές for.



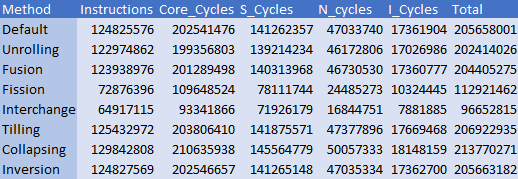
**Εικόνα 10:** Αποτελέσματα loop inversion

**Σύγκριση διαφορετικών τεχνικών βελτιστοποίησης**

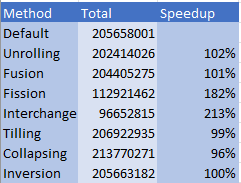


**Εικόνα 11:** Διάγραμμα σύγκρισης των μεθόδων βελτιστοποίησης.

Στο παραπάνω διάγραμμα φαίνονται συγκεντρωτικά τα αποτελέσματα όλων των τεχνικών βελτιστοποίησης που υλοποιήσαμε. Πιο συγκεκριμένα, η πιο αποτελεσματική μέθοδος βελτιστοποίησης του κώδικα μας είναι μέσω της τεχνικής του Loop Interchange, η οποία, με την προσπέλασης της μνήμης, επιτρέπει ταχύτερη εκτέλεση των εντολών, μειώνοντας τα cache misses. Ιδιαίτερα αποδοτική στη περιπτωσή μας 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.



**Πίνακας 2: Αποτελέσματα τεχνικών βελτιστοποίησης**

****

**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 2ο*

**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.

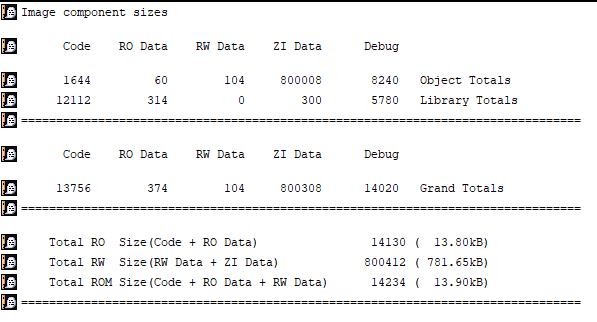
**Δεύτερο μέρος**: Διερευνώνται τεχνικές βελτίωσης της απόδοσης μέσω της χρήσης πινάκων προσωρινής αποθήκευσης (buffer) και κατευθυντήριων οδηγιών (#pragma directives), ώστε να μεγιστοποιηθεί η αποδοτική χρήση των ταχύτερων επιπέδων μνήμης (Cache).

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 1ου μέρους όπου έγινε η τεχνική του loop interchange. Με βάση τις ανάγκες του προγράμματός μας 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.

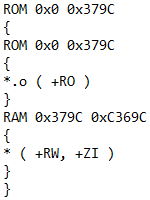
* Αρχιτεκτονική 1 – ROM/RAM BUS = 2

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.  
  


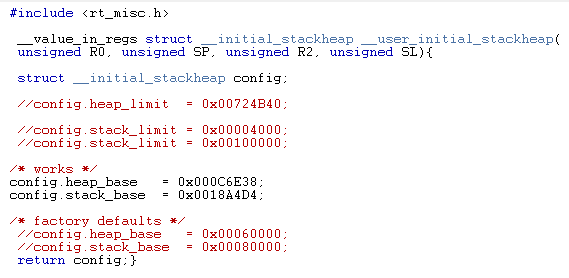
**Εικόνα 12:** Μέγεθος δεδομένων του προγράμματός μας

Χρειαζόμαστε επομένως μια μνήμη **ROM** για τον κώδικα και τα RO data (14130 bytes) αλλά και τα RW data όταν είμαστε σε Load Mode (14234 bytes total). Επίσης χρειαζόμαστε 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

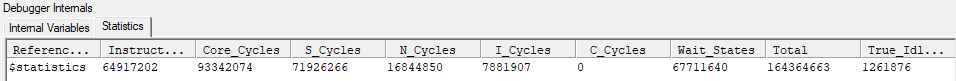


Στη συνέχεια, πρέπει να δηλωθούν σωστά αυτές οι δομές και στο αρχείο **stack.c** ώστε να μην επικαλύπτονται με τα υπόλοιπα δεδομένα της RAM.



**Εικόνα 13:** Δήλωση των stack & heap

Τρέχοντας το πρόγραμμα, λαμβάνουμε τα εξής δεδομένα:



**Εικόνα 14:** 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).

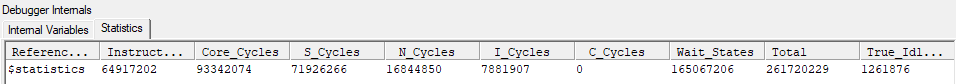
* Αρχιτεκτονική 2 – ROM/RAM BUS = 2

Σε αυτή την αρχιτεκτονική θα πραγματοποιηθεί προσπάθεια μείωσης του μεγέθος 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:



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



**Εικόνα 15:** Αποτελέσματα Αρχιτεκτονικής 2

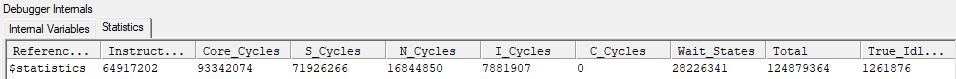
* Αρχιτεκτονική 3 – ROM/RAM FAST

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. H 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.



**Εικόνα 15:** Νέο memory.map

Τα αποτελέμστα που λαμβάνονται απο τη προσομοίωση είναι τα παρακάτω:



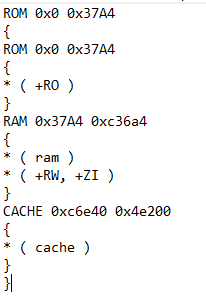
**Εικόνα 16:** Αποτελέσματα Αρχιτεκτονικής 3

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

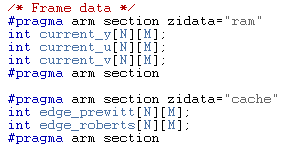
* Αρχιτεκτονική 4 – ROM/RAM + CACHE

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

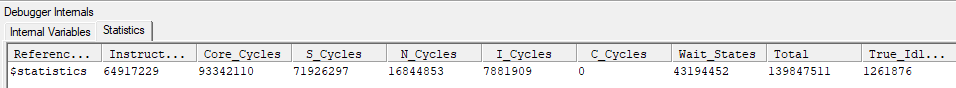
scatter.txt | memory.map



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



Tα αποτελέσματα που λαμβανουμε είναι αρκετα 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:

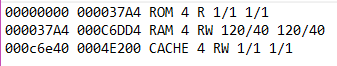


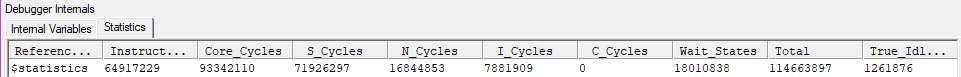
**Εικόνα 18:** Αποτελέσματα Αρχιτεκτονικής 4

* 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:





**Εικόνα 19:** Αποτελέσματα Αρχιτεκτονικής 4

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 μας δίνει τα καλύτερα αποτελέσματα.

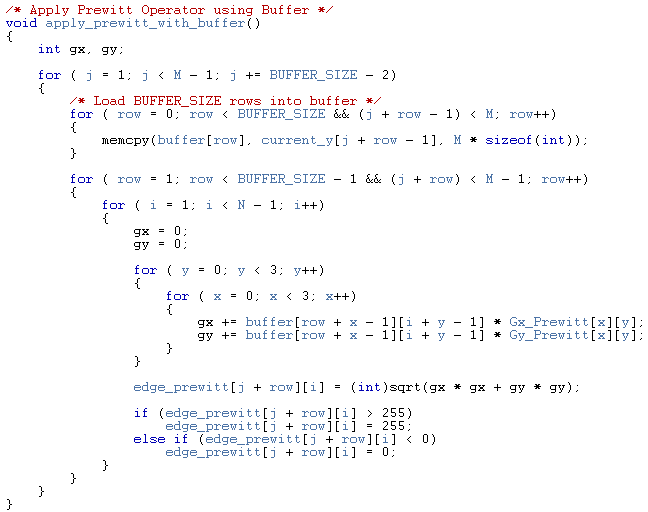
**Χρήση Βuffer στη συνέλιξη του αλγορίθμου**

Για την βελτίωση της αποδοτικότητα του αλγορίθμου και να χρησιμοποιήση buffer (προσωρινής μνήμης) για τον περιορισμό της πρόσβασης στα δεδομένα του κύριου μεγάλου πίνακα, μπορούμε να διαβάζουμε 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.

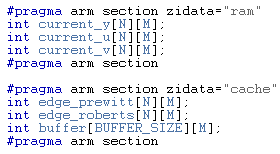
Στη συνάρτηση apply\_prewitt\_with\_buffer, η χρήση buffer επιτρέπει την τοπική εφαρμογή των Prewitt μασκών (Gx και Gy) για την ανίχνευση ακμών. Το πρόγραμμα φορτώνει τμηματικά BUFFER\_SIZE γραμμές της εικόνας στον προσωρινό πίνακα και εκτελεί τους υπολογισμούς για τον εντοπισμό ακμών σε μικρότερες περιοχές. 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 . 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.

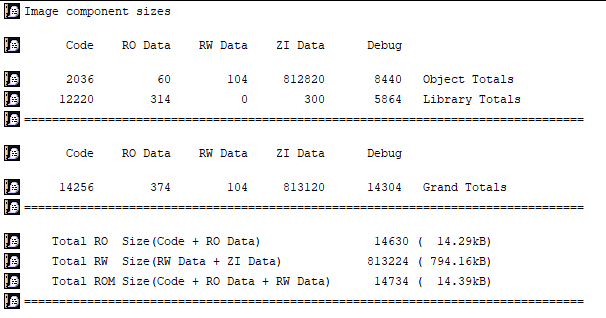


**Εικόνα 20:** Χρήση buffer στην εφαρμογή του Prewitt

Προκειμένου να επιλέξουμε την περιοχή της μνήμης στην οποία θα αποθηκευτεί ο buffer κάνουμε χρήση της δήλωσης **#pragma**, δηλώνοντας το αρχεία του buffer μεσα στη cache:



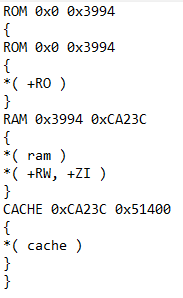
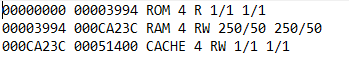
Τα συνολικά στατιστικά για τα μεγέθη των μνημών που παράγονται από το Make μετά την χρήση των **buffers** φαίνονται παρακάτω:

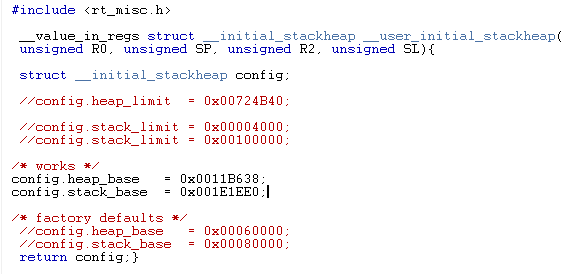


**Εικόνα 21:** Μέγεθος δεδομένων με χρήση buffer

Τροποποιούμε τώρα κατάλληλα τα αρχεία scatter.txt, memory.map και stack.c για να δηλώσουμε τη νέα μνήμη **CACHE** με καλύτερους χρόνους ανάγνωσης και εγγραφής:

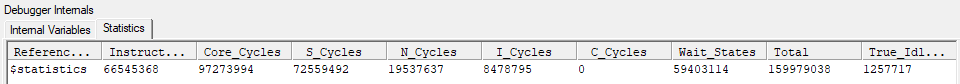
scatter.txt | memory.map



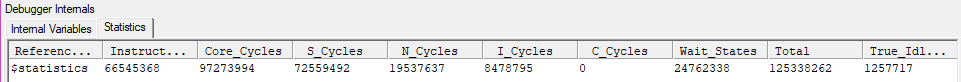


**Εικόνα 22:** Δήλωση των stack & heap

Τώρα μπορούμε να δούμε τα αποτελέσματα του Debugger Internals, που δείχνουν μια σαφή βελτίωση με την χρήση buffers σε σχέση με την πρώτη υλοποίηση που πραγματοποιήσαμε:



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



Έτσι, μπορούμε να δούμε 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.

**Bιβλιογραφία**

[1] [**https://en.wikipedia.org/wiki/Prewitt\_operator**](https://en.wikipedia.org/wiki/Prewitt_operator)

[2] [**https://en.wikipedia.org/wiki/Roberts\_cross**](https://en.wikipedia.org/wiki/Roberts_cross)

[3] [**https://www.tutorialspoint.com/dip/prewitt\_operator.htm**](https://www.tutorialspoint.com/dip/prewitt_operator.htm)

[4] [**https://homepages.inf.ed.ac.uk/rbf/HIPR2/roberts.htm**](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**](https://levelup.gitconnected.com/c-programming-hacks-4-matrix-multiplication-are-we-doing-it-right-21a9f1cbf53)

[6] Διαφάνειες εργαστηριακών διαλέξεων του μαθήματος.