ΔΗΜΟΚΡΙΤΕΙΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΡΑΚΗΣ

ΠΟΛΥΤΕΧΝΙΚΗ ΣΧΟΛΗ

ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΣΧΕΔΙΑΣΜΟΣ ΕΝΣΩΜΑΤΩΜΕΝΩΝ ΣΥΣΤΗΜΑΤΩΝ

Φίλτρο Μέσης Τιμής Για Την Επεξεργασία Εικόνας
3ο Μέρος

Διδάσκων

Συρακούλης Γεώργιος

Ομάδα 24

Λαζαρίδου Νίνα ΑΜ:57260

Μούσλεχ Στυλιανός Γεώργιος ΑΜ:57382

Εισαγωγή

Στο πρώτο μέρος της εργασίας υλοποιήσαμε ένα συγκεκριμένο αλγόριθμο επεξεργασίας εικόνας και τον βελτιστοποιήσαμε αυξάνοντας την απόδοση του αλγορίθμου. Θεωρήσαμε άπειρη μνήμη με άμεση πρόσβαση σε αυτή.

Στο δεύτερο μέρος της εργασίας, εισαγάγαμε εμείς τα είδη, τα μεγέθη και την ταχύτητα της μνήμης, δημιουργώντας δικές μας ιεραρχίες. Αναλύσαμε διαφορετικές περιπτώσεις και τις συγκρίναμε ως προς το μέγεθος και τα συνολικά cycles του επεξεργαστή.

Στο τρίτο μέρος της εργασίας, λοιπόν, σκοπός είναι να εξετάσουμε την χρησιμότητα της επαναχρησιμοποίησης δεδομένων στη μνήμη. Αρχικά, αναλύουμε την μέθοδο της επαναχρησιμοποίησης δεδομένων και την εφαρμογή της στον κώδικά μας και εξηγούμε την ιεραρχία μνήμης που ακολουθούμε. Έπειτα, περιγράφουμε τις υλοποιήσεις που κάναμε. Εν συντομία δηλαδή, κρατώντας την καλύτερη έκδοση του προηγούμενου μέρους, και θεωρώντας τον κώδικα πλέον παγιωμένο προσθέτουμε άλλο ένα επίπεδο πολύ γρήγορης μνήμης, η οποία παίζει το ρόλο μνήμης cache και εφαρμόζουμε τεχνικές επαναχρησιμοποίησης όπως η χρήση buffers και η αποθήκευση μεταβλητών που καλούνται πολλαπλές φορές.

Αφού παρουσιάσουμε τα αποτελέσματα των υλοποιήσεων, τα αναλύουμε και εξάγουμε τα ανάλογα συμπεράσματα.

Τέλος, ανακεφαλαιώνουμε τη συνολικής πορείας της Εργασίας μας και κάνουμε ένα γενικό σχολιασμό.

Οι μετρήσεις έγιναν με την εικόνα dog_440x330.yuv και RADIUS=2.

Ανάλυση Επαναχρησιμοποίησης Δεδομένων στον Αλγόριθμο μας

Για να κάνουμε μια αρχική εκτίμηση της χρησιμότητας και των χαρακτηριστικών της εισαγωγής πολύ μικρής μνήμης για επαναχρησιμοποίηση δεδομένων, πρέπει να μελετήσουμε πρώτα τον τρόπο που λειτουργεί ο αλγόριθμος μας και το τρόπο που χρησιμοποιεί τους πίνακες δεδομένων.

Θέλουμε να εισάγουμε buffers στα οποία να αποθηκεύουμε μέρη του πίνακα κατά περίπτωση ώστε να έχουμε πρόσβαση σε αυτά πιο γρήγορα. Για να έχουμε κέρδος στην απόδοση, αυτά τα μέρη θα πρέπει να τα χρησιμοποιεί ο αλγόριθμος πάνω από μία φορά (επαναχρησιμοποίηση) καθώς σε διαφορετική περίπτωση απλά θα έχουμε το overhead της φόρτωσης τους στην πιο γρήγορη μνήμη για την προσπέλασή τους μόνο 1 φορά από εκεί. (Σε μια τέτοια περίπτωση θα ήταν καλύτερο να τα προσπελάσουμε κατευθείαν από τη αρχική μνήμη).

Παρατηρώντας τον βελτιστοποιημένο αλγόριθμο που χρησιμοποιούμε, έχουμε δημιουργήσει τον integral image, με σκοπό να κάνουμε όσες λιγότερες προσπελάσεις στην μνήμη γίνεται και να έχουμε πολύ λίγες επαναλαμβανόμενες.

Δηλαδή πλέον για τον υπολογισμό του αθροίσματος των τιμών σε ένα kernel χρησιμοποιούμε μόνο 4 «ακριανές» τιμές ενός προϋπολογισμένου πίνακα, όπως για παράδειγμα φαίνεται παρακάτω όπου θέλουμε να βρούμε το άθροισμα του πράσινου τμήματος του input table χρησιμοποιώντας μόνο τις τιμές των κόκκινων pixels του summed-area table:

input table

1	3	0	2	1	2
3	2	4	3	6	0
0	5	1	1	5	3
2	2	3	3	7	2
4	2	8	4	2	5

summed-area table

1	4	4	6	7	9
4 D	9	13	18	25 C	27
4	14	19	25	37	42
6 B	18	26	35	54 A	61
10	24	40	53	74	86

Για να έχουμε επαναχρησιμοποίηση δεδομένων σε ικανοποιητικό βαθμό πρέπει να έχουμε στους buffers δεδομένα που θα «προλάβουν» να χρησιμοποιηθούν ως τα κάτω pixel του kernel και καθώς αυτό κυλάει προς τα κάτω, να επαναχρησιμοποιηθούν ως τα πάνω pixel. Φυσικά το ίδιο ισχύει και για τα αριστερά και δεξιά pixel αυτής της κινούμενης τετράδας αλλά εφόσον φορτώνουμε ολόκληρες γραμμές στους buffers εξασφαλίζεται η επαναχρησιμοποίηση τους.

Έτσι, χρησιμοποιώντας radius 2 (kernel 5x5) θέλουμε από τον πίνακα integralImg τιμές [5+1] γραμμών όπου σε κάθε υπολογισμό της τιμής του φίλτρου θα χρειαζόμαστε δεδομένα της

πρώτης και της τελευταίας γραμμής. Οπότε χρειαζόμαστε 6 buffers όπου αποθηκεύονται 6 διαδοχικές γραμμές του πίνακα για να έχουμε καλή επαναχρησιμοποίηση δεδομένων.

Τέλος, αλγοριθμικά έχουμε και αρκετές μεταβλητές που επαναχρησιμοποιούνται αρκετά, τις οποίες στην συνέχεια προσθέτουμε στην γρήγορη αυτή μνήμη.

Ιεραρχία Μνήμης

Όπως αναφέραμε θα βασιστούμε πάνω στην καλύτερη υλοποίηση που προτείναμε στο δεύτερο μέρος της εργασίας (έκδοση V5).

Θα εισάγουμε 6 buffers μήκους [M+(2*RADIUS)+1] θέσεων τύπου int. Χρειαζόμαστε λοιπόν 6*445*4=10.680 Byte ή 10.43 kB. Χρειαζόμαστε και χώρο για μερικές μεταβλητές. Θα εισάγουμε λοιπόν μια πολύ γρήγορη μνήμη με όνομα “CACHE” και μέγεθος 11kB.

Η ιεραρχία μνήμης μας λοιπόν είναι η εξής:

```
00000000 00080000 ROM   4 R  1/1 1/1
00080000 08000000 DRAM  4 RW 250/50 250/50
08080000 00200000 SRAM  4 RW 30/15 30/15
08280000 00002C00 CACHE 4 RW 1/1 1/1
```

Φυσικά, σε περίπτωση που δεν μπορούμε να έχουμε την επιθυμητή μνήμη , ή θέλουμε μια πιο scalable λύση (ως προς το μέγεθος της εικόνας), θα εφαρμόζαμε τους buffers με λογική «σπασίματος» της εικόνας σε μικρότερα τετράγωνα κομμάτια (έτσι ώστε να έχουμε 6 γραμμές της υποεικόνας στο buffer). Έτσι με τη χρήση loop tiling θα μπορούμε να δουλεύουμε σε κάθε υποεικόνα με τους buffers allocated σωστά. Ουσιαστικά προσφέρουμε ένα είδος data locality για την δικιά μας μνήμη που λειτουργεί σαν cache.

Περιγραφή Υλοποιήσεων

Version 5.1: withBuffers

Σε αυτήν την έκδοση εισάγουμε για πρώτη φορά τους 6 buffers. Η λογική για την αρχικοποίηση τους και το “sliding” των γραμμών του πίνακα σε αυτούς γίνεται ακριβώς με τον ίδιο τρόπο με αυτόν που είδαμε στο 3^{ο} εργαστήριο του μαθήματος.

Version 5.2: withBuffers, Vars

Σε αυτή την έκδοση βάζουμε, εκτός από τους buffers, και τις μεταβλητές του namespace “vars” και “othervars” στην cache. Ο κώδικας δεν διαφοροποιείται από το version 5.1, αλλάζει μόνο το αρχείο scatter. Το μέγεθος μνήμης που απαιτούν αυτές οι μεταβλητές είναι πολύ μικρό με αποτέλεσμα να μην χρειάζεται να μεταβάλλουμε το μέγεθος της cache που διαθέτουμε.

Version 5.3: withBuffPointers

Σε αυτή την υλοποίηση για να μειώσουμε το overhead που δημιουργεί η κυλιόμενη εναλλαγή των τιμών των buffers εισάγουμε δείκτες. Δημιουργούμε 7 δείκτες (δηλαδή όσοι οι buffers +1 βοηθητικό) και ουσιαστικά κυλάμε τους δείκτες ώστε να δείχνουν διαφορετικούς buffers. Έτσι σε κάθε επανάληψη πλέον ενημερώνονται οι τιμές ενός μόνο buffer αντί για όλων.

Version 5.4: withUnroll

Έχοντας μειώσει το overhead αλγοριθμικά με την χρήση pointers, θέλουμε να το μειώσουμε ακόμα περισσότερο κάνοντας unroll στις for loops των buffers

Version 5.5: Refined

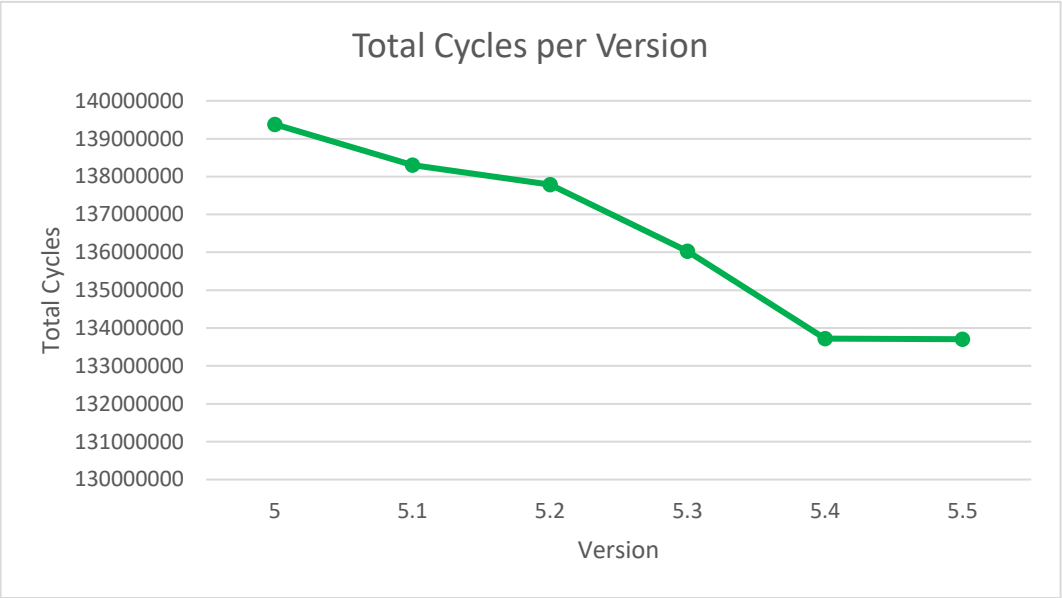
Έχοντας τελειώσει με τις αλλαγές που δοκιμάσαμε, κάναμε μια προσεκτική μελέτη του συνολικού κώδικα ,καθώς είναι το τελευταίο μέρος της εργασίας, και αφαιρέσαμε πράξεις που δεν χρειάζονται.

Αποτελέσματα Υλοποιήσεων - Ανάλυση

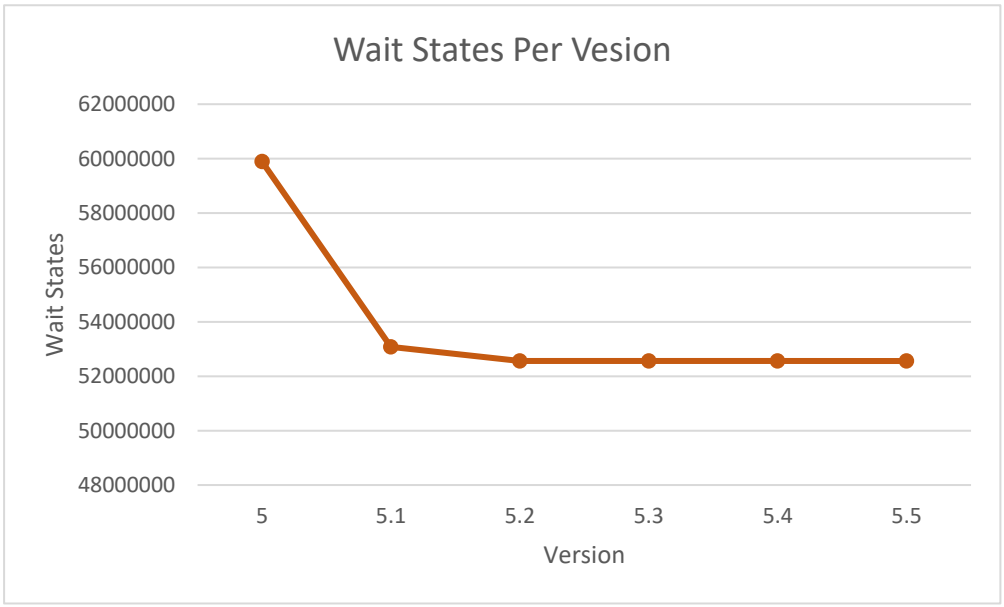
Τα αποτελέσματα των παραπάνω υλοποιήσεων που περιγράψαμε συνοψίζονται ως εξής (συμπεριλαμβάνουμε στην αρχή και την καλύτερη εκτέλεση του δεύτερου μέρους της εργασίας):

Version	Name	Instructions	Core Cycles	S Cycles	N Cycles	I Cycles	C Cycles	Wait States	Total Cycles	True idle Cycles
5	3arrays	51849100	73243320	54769345	14343007	10377605	0	59888873	139378830	2180260
5.1	withBuffers	54401760	78978362	56731638	17043700	11449252	0	53075276	138299866	2180182
5.2	withBuffers,Vars	54401760	78978362	56731638	17043700	11449252	0	52560123	137784713	2180182
5.3	withBuffPointers	54073021	77043935	56551164	16018798	10900021	0	52560083	136030066	2213926
5.4	withUnroll	52907037	74822179	55385839	15326575	10445903	0	52560083	133718400	2197054
5.5	Refined	52900474	74810275	553799598	15322243	10444158	0	52559332	133705331	2196975

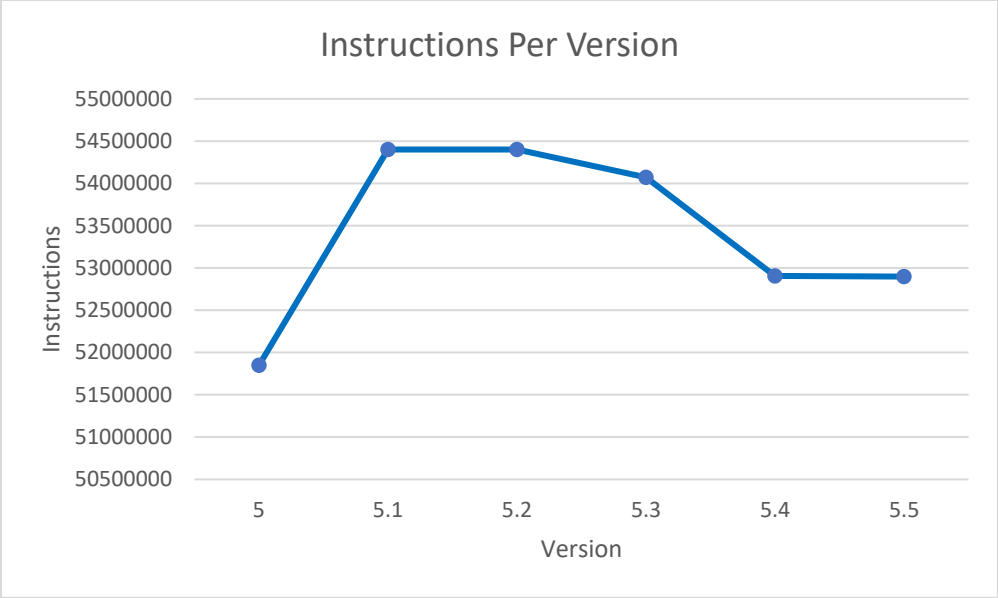
- Αρχικά παρατηρούμε ότι τα αποτελέσματα αναφορικά με τα total cycles είναι τα αναμενόμενα . Δηλαδή, εισάγοντας τους buffers έχουμε μείωση των συνολικών κύκλων, η οποία συνεχίζεται καθώς εμείς μειώνουμε το overhead και βελτιστοποιούμε την υλοποίηση τους.



- Στη συνέχεια, παρατηρούμε ότι ο αριθμός των wait states μειώθηκε σε μεγάλο βαθμό με την εισαγωγή των buffers (έκδοση 5.1). Βάζοντας και τις μεταβλητές που επαναχρησιμοποιούνται αρκετά παρατηρούμε μια επιπλέον μείωση(έκδοση 5.2). Αυτό είναι λογικό , καθώς πλέον χρησιμοποιούμε λιγότερο τις πιο αργές μνήμες που προσέθεταν περισσότερους κύκλους καθυστέρησης κατά την προσπέλαση. Μετά από αυτό , οι υπόλοιπες βελτιστοποιήσεις δεν μειώνουν ιδιαίτερα τα wait states κάτι αναμενόμενο αφορούν περισσότερο τη μείωση του overhead των buffers χωρίς να αυξάνεται το data reusability.



- Τέλος, βλέποντας τον αριθμό των instructions ανά έκδοση, μπορούμε αρχικά να δούμε μια μεγάλη αύξηση αυτών με την εισαγωγή των buffers. Αυτή η αύξηση προέρχεται από το overhead της αλγοριθμικής υλοποίησης των buffers. Στην έκδοση 5.3 που μειώνουμε το overhead με την χρήση δεικτών , έχουμε αναμενόμενη μείωση των instructions. Το ίδιο συμβαίνει και στην 5.4 , καθώς όπως είδαμε και στο πρώτο μέρος της εργασίας το Loop Unrolling μειώνει σημαντικά τον αριθμό των instructions.



Ανακεφαλαίωση – Γενικός Σχολιασμός Εργασίας

Συνοψίζοντας, στο 1ο μέρος του project κληθήκαμε να υλοποιήσουμε τον αλγόριθμο του φίλτρου μέσης τιμής για την επεξεργασία εικόνας. Εφαρμόσαμε μετασχηματισμούς βρόχων και αλγοριθμικές βελτιστοποιήσεις για την βελτίωση της απόδοσης θεωρώντας ιδανική και άπειρη μνήμη. Εξετάσαμε την κάθε περίπτωση ξεχωριστά για να βρούμε τις καλύτερες συνδυαστικές βελτιστοποιήσεις. Με την καλύτερη μας έκδοση είχαμε μια πτώση των συνολικών κύκλων κατά **79.16%** σε σχέση με την αρχική.

Στο 2ο μέρος, η μνήμη έγινε πεπερασμένη και μη ιδανική. Εδώ κληθήκαμε να προτείνουμε μία κατάλληλη ιεραρχία μνήμης για την καλύτερη έκδοση κώδικα που καταφέραμε στο 1ο μέρος και παρατηρήσαμε το trade-off του μεγέθους και είδους της μνήμης με τους συνολικούς κύκλους.

Στο 3ο και τελευταίο μέρος, έχοντας ως σημείο αναφοράς μια ιεραρχία μνήμης και έναν παγιωμένο κώδικα, κληθήκαμε να αυξήσουμε την απόδοση του προγράμματος εκμεταλλευόμενοι την επαναχρησιμοποίηση των δεδομένων, προσθέτοντας άλλο ένα επίπεδο πάρα πολύ γρήγορης και μικρής μνήμης. Σε αυτό το μέρος πετύχαμε πτώση κύκλων κατά **4%**.

Τις περισσότερες φορές, στα ενσωματωμένα συστήματα έχουμε περιορισμένη διαθεσιμότητα μνήμης καθώς και ανάγκη για χαμηλή κατανάλωση ισχύος και για γρήγορη εκτέλεση του αλγορίθμου.

Όλα τα παραπάνω στάδια μελέτης και βελτιστοποίησης συμβάλλουν στην όσο το δυνατό καλύτερη εκμετάλλευση των πόρων που διαθέτει ένα ενσωματωμένο.

Φυσικά, όπως έχουμε δει στο μάθημα, η αρχιτεκτονική του μικροεπεξεργαστή επηρεάζει το ποσοστό της βελτίωσης του προγράμματος για διαφορετικούς μετασχηματισμούς βρόγχων καθώς και για άλλες αλγοριθμικές βελτιστοποιήσεις. Για αυτό το λόγο είναι σημαντικό να αξιολογούμε κάθε αλλαγή στον κώδικα με βάση τις προσομοιώσεις που κάνουμε.