Εργασία 3η

Σταύρος Μαλακούδης 9368 samalako@ece.auth.gr

Τα αρχεία της εργασίας: https://github.com/Stavrenas/PDS_ex3

Σκοπός της 3ης εργασίας ήταν η αποθορυβοποίηση εικόνας με τη χρήση του αλγορίθμου non Local Means σε περιβάλλον Cuda-gpu. Διαθέτω την GTX1650 και επομένως το development-testing έγινε τοπικά, αλλά τα αποτελέσματα είναι από τις tesla P100 της συστοιχίας.

Καταρχάς, έγινε μια σειριακή υλοποίηση σε απλή γλώσσα C , ώστε να γίνει πιο ξεκάθαρη η δομή της υλοποίησης σε Cuda. Μάλιστα, παρουσιάζονται τα αποτελέσματα της σειριακής υλοποίησης ώστε να γίνει ξεκάθαρο το κέρδος της χρήσης GPU σε προβλήματα με μεγάλη πολυπλοκότητα, όπως το συγκεκριμένο.

Τόσο η υλοποίηση σε gpu όσο και η σειριακή, χρησιμοποιούν αρχεία .csv για την ανάγνωση αλλά και την εγγραφή τετράγωνων και ασπρόμαυρων εικόνων (για την Online μετατροπή από jpg σε csv προτείνω το https://products.aspose.app/words/conversion/jpg-to-csv). Η οπτικοποίηση του αποτελέσματος γίνεται με τα script show_imagesCPU.m και show_imagesCUDA.m αντίστοιχα, σε περιβάλλον octave ή MATLAB. Αφού γίνει η ανάγνωση της εικόνας, προστίθεται τυχαίος θόρυβος με γκαουσιανή συνάρτηση πυκνότητας πιθανότητας και έπειτα γίνεται η αποθορυβοποίηση. Ο χειρισμός της εικόνας γίνεται με τη χρήση μονοδιάστατου πίνακα. Στην δημιουργία των patches, δεν λαμβάνονται υπόψη τα pixel, αν αυτά βρίσκονται εκτός εικόνας. Για παράδειγμα, στο pixel με συντεταγμένες (0,0) με μέγεθος patch 3x3, λαμβάνονται υπόψη μόνο τα (0,0), (0,1) ,(1,0) και (1,1). Δεν γίνεται δηλαδή κάποιου είδους καθρεπτισμός της εικόνας.

Σύντομη περιγραφή της σειριακής υλοποίησης:

Αρχικά , δημιουργούνται τα ξεχωριστά patches, που σε αυτή την περίπτωση αποθηκεύονται σε δισδιάστατο πίνακα(πρακτικά, κάθε patch έχει τον δικό του pointer) και υπολογίζονται τα γκαουσιανά βάρη που θα χρησιμοποιηθούν στην εύρεση των αποστάσεων μεταξύ των patches κάθε pixel. Σειριακά, ο κώδικας υπολογίζει σε κάθε pixel τις αποστάσεις από όλα τα υπόλοιπα pixel της εικόνας (*πίνακας distances*) και ταυτόχρονα υπολογίζει το Z(i) (*normalFactor*). Αφού βρεθούν όλες οι αποστάσεις $e^{\frac{-\|f(N_i)-f(N_j)\|^2_{G(a)}}{\sigma^2}}$, διαρούνται με το Z(i) και έτσι προκύπτει το w(i,j) (*που αντιστοιχεί στον πίνακα distances πάλι*). Τέλος , για την εύρεση της τιμής του κάθε pixel της νέας εικόνας, δηλαδή την εύρεση του $\hat{f}(x)$, γίνεται η πράξη $\sum_{j=0}^{total\ pixels} w(i,j)\cdot f(j)$

Σύντομη περιγραφή της υλοποίησης σε CUDA:

Όμοια με πριν, γίνεται η εύρεση του patch που αντιστοιχεί σε κάθε pixel. Όμως, ο πίνακας αυτός αποθηκεύεται σε μορφή RowMajor, ώστε να είναι πιο εύκολη η διαχείρισή του σε περιβάλλον CUDA. Το kernel που κατασκευάστηκε εκμεταλλεύεται τη δυνατότητα της ταυτόχρονης εκκίνησης χιλιάδων νημάτων και έτσι “αναθέτει” την εύρεση του w(i,j) , του Z(i) και του $\hat{f}(i)$ σε ένα pixel. Δηλαδή , ένα thread αντιστοιχεί σε ενα pixel .

Σε γενικές γραμμές, δεν μπορεί να γίνει 1-1 αντιστοίχηση με το σειριακό κώδικα. Στο σειριακό , γίνεται δέσμευση ενός πίνακα distances, στον οποίο γίνεται η αποθήκευση των αποστάσεων των patches από το τρέχων. Ο πίνακας αυτός έχει τόσες θέσεις όσα τα pixel της εικόνας. Αυτό σημαίνει ότι , αν ακολουθούσαμε την ίδια λογική, κάθε thread θα χρειάζεται έναν τέτοιο πίνακα. Αυτό είναι απαγορευτικό, καθώς σε μια εικόνα με μέγεθος 256x256 θα έχουμε 65536 threads και κάθε ένα από αυτά θα δεσμεύσει πίνακα μεγέθους 65536 θέσεων. Στην πιο απλή περίπτωση όπου μια θέση αντιστοιχεί σε έναν float (4 bytes) , αυτό μεταφράζεται σε έναν πίνακα 4294967296 θέσεων μεγέθους 16GB. Για αυτό λοιπόν, αλλάζουμε τη σειρά των υπολογισμών, ώστε να μην χρειαστεί ο πίνακας. Δηλαδή, υπολογίζουμε το άθροισμα $\sum_{i \in \Omega} e^{\frac{-\|f(N_i)-f(N_j)\|^2_{G(a)}}{\sigma^2}} \cdot f(j)$ και ταυτόχρονα βρίσκουμε το Z(j), με το οποίο διαιρούμε την έκφραση, φτάνοντας στο ζητούμενο.

Με αρκετές δοκιμές, κατέληξα στο συμπέρασμα ότι οι λούπες που βρίσκονται στο kernel τρέχουν ταυτόχρονα στα threads. Δηλαδή , τα threads θα τρέξουν πρώτα όλα μαζί την πρώτη επανάληψη, μετά τη δεύτερη κτλ . Αυτό σημαίνει ότι θα μπορούσαμε να αποθηκεύσουμε στην shared μνήμη τα patches όλων των pixel στην τρέχουσα γραμμή για παράδειγμα, έτσι ώστε να μην χρειαστεί καθόλου ανάγνωση από την global memory. Ακόμα και με το καθολικό άνω φράγμα των 48KB είναι δυνατή η επεξεργασία εικόνων διάστασης μέχρι περίπου 1024x1024. Λόγω έλλειψης χρόνου, αυτό δεν κατέστη δυνατό (ο κώδικας υπάρχει σε σχόλια) αλλά γίνεται η χρήση της shared memory ώστε να αποθηκευτούν τα γκαουσιανά βάρη, τα οποία χρησιμοποιούνται σε κάθε επανάληψη μεν, αλλά έχουν μικρή επίδραση στο χρόνο .

Παρουσίαση αποτελεσμάτων και συμπεράσματα

Cuda execution time (seconds)

PatchSize	64x64	128x128	256x256
3	0,814	1,078	2,116
5	0,9	1,198	4,204
7	0,962	1,77	9,228

Serial execution time (seconds)

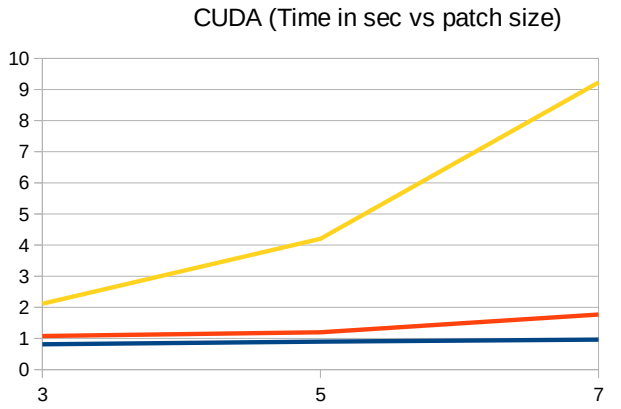
PatchSize	64x64	128x128	256x256
3	1,052	16,18	258
5	1,656	26,32	423
7	2,474	39,77	647

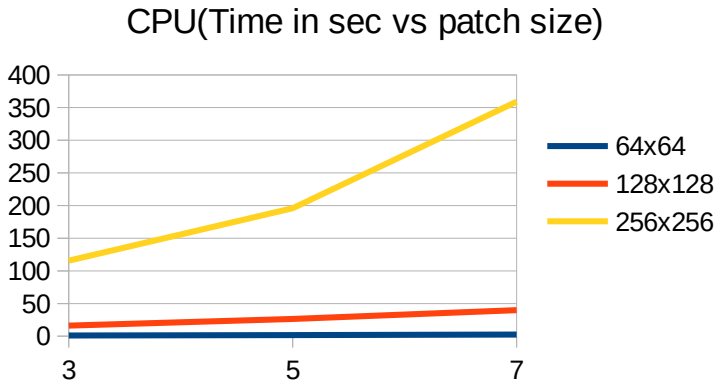
Cuda patch size – time correlation

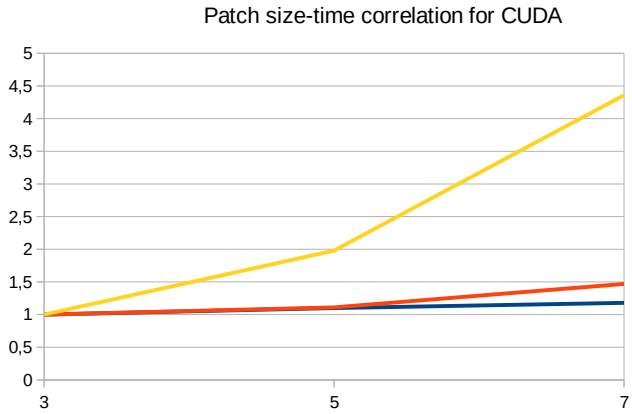
3	1,052	16,18	258
5	1,656	26,32	423
7	2,474	39,77	647

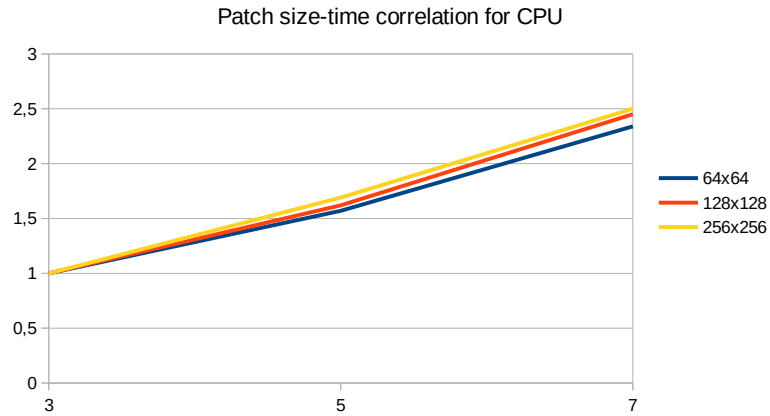
Serial patch size – time correlation

PatchSize	64x64	128x128	256x256
3	1	1	1
5	1,57	1,62	1,69
7	2,34	2,45	2,5









Συμπεράσματα:

- Σαφώς, η υλοποίηση με cuda προσφέρει πολύ μεγάλες επιταχύνσεις, ακόμα και δύο τάξεων μεγέθους , εφόσον το πρόβλημα είναι επαρκώς μεγάλο. Στην προκειμένη περίπτωση, έχουμε πρόβλημα σειριακής πολυπλοκότητας $O(N^4)$ που με κατάλληλο προγραμματισμό πέφτει στο $O(N^2)$. Σε μικρά μεγέθους προβλήματα (εικόνες <<256x256) δεν γίνεται πλήρης χρήση της GPU . Τα νήματα είναι πιο αδύναμα σε σχέση με αυτά της CPU. Σαν αποτέλεσμα, επειδή το συνολικό πλήθος των υπολογισμών είναι μικρό, η διαφορά που παρατηρείται είναι μικρή . Λόγω της φύσης του αλγορίθμου ($O(N^4)$) , παρατηρείται τεράστια διαφορά σε μεγαλύτερες εικόνες.
- Στην σειριακή υλοποίηση, το μέγεθος του patch έχει καθαρά γραμμική επίδραση, ανεξάρτητα με το μέγεθος της εικόνας. Αντιθέτως, η υλοποίηση με CUDA παρουσιάζει μη γραμμικότητα. Πιο συγκεκριμένα, η επίδραση σε μικρές εικόνες είναι μικρή, ενώ σε μεγαλύτερες αυξάνεται σημαντικά και φαίνεται να έχει τετραγωνική συσχέτιση