Μάθημα : Παράλληλα και Διανεμημζνα Συστήματα

Φοιτητής : Χωραφάς Χρήστος

ΑΝΑΦΟΡΑ 2ης ΕΡΓΑΣΙΑΣ

Υλοποίηση αλγορίθμου Mean Shift για τον εντοπισμό τοπικών μεγίστων.

Γενικά :

- Χρησιμοποιήθηκαν S-datasets με N=5000 2-d σημεία και 15 Gaussian-ά clusters από https://cs.joensuu.fi/sipu/datasets/ .
- Ο σειριακός αλγόριθμος δουλεύει πολύ αργά μπροστά στον παράλληλο.
- Ναι, δυστυχώς ο έλεγχος έγινε με μια printf, βλέποντας την τιμή της mean για κάθε στοιχείο. Κακό...Θεωρητικά ένας έλεγχος που θα μπορούσε να γίνει είναι να ελέγξω τον πίνακα με το διάνυσμα της mean για κάθε σημείο και να βρώ που συγκλίνουν(π.χ. αν κάποια σημεία έχουν παρόμοια mean-μικρή διαφορά, τότε για αυτά τα σημεία η mean θα είναι ο μέσος όρος τους), οπότε αν βρώ 15 means σημαίνει ότι είμαι σωστός.
- Για αλλαγή αριθμού σημείων και διαστάσεων -> *#define I* και *#define J* αντίστοιχα. Λόγω πίεσης χρόνου δεν έχω τεστάρει τον κώδικα πέρα από το προαναφερόμενο dataset αλλά πρέπει να δουλέψει.

Mean Shift

Τα βήματα που ακολουθούνται κατ' επανάληψη, για κάθε σημείο είναι :

i. Προσδιορίζεται με χρήση του αλγορίθμου kNN, τον οποίο πήρα από την πρώτη εργασία, το παράθυρο-bandwidth γύρω από το

σημείο. Ο kNN βρίσκει τους k κοντινότερους γείτονες και δημιουργεί μια περιοχή αναζήτησης γύρω από αυτό με βάση την απόσταση του από τον k-οστό γείτονα. Έτσι μπορώ να βρίσκω μια κατάλληλη περιοχή αναζήτησης κάθε φορά και να μην έχω μια σταθερά που μπορεί να μου χαλάει τις μετρήσεις. (Ο kNN χρησιμοποιεί την συνάρτηση βιβλιοθήκης qsort για την ταξινόμηση των γειτόνων.)

ii. Υπολογίζεται η mean των σημείων μέσα σε αυτό το παράθυρο.

iii. Τέλος μετατοπίζεται το παράθυρο, έχοντας νέο κέντρο την mean, μέχρι να συγκλίνει (*for(i = 0; i < c; i++),* όπου c ο αριθμός των επαναλήψεων για σύγκλιση που δίνεται από τον χρήστη).

Cuda

Χρησιμοποιώ :

i. *cudaMalloc()* για την δέσμευση μνήμης των απαραίτητων μεταβλητών στη κάρτα. Π.χ. `cudaMallocPitch(&d_x, I*J * sizeof(double));` ,όπου d_x ο pointer που δείχνει στην δεσμευμένο χώρο, μεγέθους `I*J * sizeof(double)`, στην κάρτα.

ii. *cudaMemcpy()* για την αντιγραφή των δεδομένων από την host μνήμη στην μνήμη της κάρτας. Π.χ. `cudaMemcpy(d_x,x,I*J*sizeof(double),cudaMemcpyHostToDevice);` όπου αντιγράφονται τα δεδομένα του πίνακα x, που περιέχει τα σημεία, από την host μνήμη στον δεσμευμένο χώρο από την 1) .

iii. *dim3 gridSize()*, *dim3 blockSize()*, από τις οποίες η πρώτη μου καθορίζει πόσα blocks θα έχω, ενώ η δεύτερη πόσα threads(θέτω *BLOCKSIZEx*=512, μετά απο testing νομίζω ότι στον διάδη τόσα είναι τα max threads per block) θα έχω σε κάθε block. Η πρώτη συνάρτηση ως όρισμα παίρνει την επιστρεφόμενη τιμή της *Blocks(),* η οποία αν *I*J > BLOCKSIZEx* μου επιστρέφει *i*J/BLOCKSIZEx* + 1 blocks αλλιώς 1 .

iv. `kernel<<<gridSize, blockSize>>>(d_c, d_k, d_x, d_y, d_m);`

Στον kernel space ακολουθείται η ίδια τακτική με τον σειριακό αλγόριθμο, απλα παράλληλα(duh).

<u>Παρατηρήσεις για CUDA</u> :

a) Για σωστά αποτελέσματα, ελέγχω αν `το indexi = blockIdx.x*blockDim+threadIdx.x*J` είναι μικρότερο από των αριθμό των στοιχείων I*J του πίνακα. Ο λόγος που πολ/άζω το *threadId.x* με το J είναι γιατί θέλω κάθε thread να δείχνει στην πρώτη συντεταγμένη ενός σημείου(στην ουσία να δείχνει το τάδε σημείο) . Π.χ. έστω ότι J=2 και *blockId.x*=0. Τότε για *threadId.x*=0, το thread δείχνει στο 0 σημείο. Για *threadId.x=1*, το thread δείχνει στο 2^{o} σημείο του πίνακα, για *threadId.x=2* δείχνει στο 4^{o} κ.ο.κ. (Σκέψου μετατροπή 2d πίνακα σε 1d).

b) Για κάθε σημείο λοιπόν του πίνακα X βρίσκω με kNN τους k πλησιέστερους γείτονες (εδώ ο kNN χρησιμποιεί την insertion sort, γιατί η *qsort()* δεν υπάρχει στις βιβλιοθήκες της cuda. Παρόλο που δεν έχει τον καλύτερο χρόνο, είναι ένας αξιόλογος αλγόριθμος και εφόσον δεν μας νοιάζει ο χρόνος ταξινόμησης δεν υπάρχει πρόβλημα).

c) Ο *kernel()* βλέπει την συνάρτηση kNN() και ο kNN() με την σειρά του βλέπει την sort(), καθώς τις έχω ορίσει ως "`__device__`" συναρτήσεις. Το ίδιο και με την *gaussian()*.

d) Φορτώνω στον πίνακα M το διάνυσμα της mean για κάθε σημείο.

e) Αφού επιστρέψω στην cpu, αντιγράφω από την device memory στην host τα δεδομένα του M στον πίνακα m και εκτυπώνω το περιεχόμενό του.

f) Καλώ την συνάρτηση *cudaMalloc()* για την αποδέσμευση μνήμης για κάθε device pointer.

Χρόνοι εκτέλεσης στον διάδη:

- Για k(neighbors) = 1, c(converge) = 15, I(σημεία) = 5000 σε 2d ο χρόνος είναι 0.28028
- k = 2 , c = 15,…, ο χρόνος είναι 0.2805
- k = 2, c = 20,…, ο χρόνος είναι 0.3651
- k = 5, c = 20,…, ο χρόνος είναι 0.367 -> κακό αποτέλεσμα
- k = 2, c = 40,…, ο χρόνος είναι 0.87843 -> πολύ καλό αποτέλεσμα
- Πολύ καλύτερος από τον χρόνο του matlab (16.89 sec για λιγότερα στοιχεία)

Συμπεράσματα :

Ο χρόνος και τα αποτελέσματα εξαρτώνται από τον αριθμό γειτόνων και σύγκλισης που θα δώσει ο χρήστης. Τα αποτελέσματα φαίνεται να είναι πιο σωστά όταν ο αριθμός γειτόνων είναι 2 (μετά απο πειράματα και πάντα για 2d σημεία.Για 3d κλπ πιθανότατα αλλάζει). Παρατηρώ ότι όταν αυξάνω τον αριθμό αυτό, η διάκριση για διαφορετικές mean γίνεται πιο δύσκολη, καθώς όσο αυξάνω τους γείτονες, αυξάνω και το bandwidth και τα αποτελέσματα χάνουν ακρίβεια. Από την άλλη αύξηση του αριθμού σύγκλισης αντιστοιχεί σε μεγαλύτερο χρόνο αλλά και καλύτερη ακρίβεια. Τελικά, φαίνεται ότι ο χρόνος επηρρεάζεται κυρίως από τον αριθμό σύγκλισης, γιατί ο αριθμός γειτόνων δεν συμβάλλει ιδιαίτερα και εκτός αυτού έχει ένα peak, που αν το ξεπεράσω δίνει ανακριβή αποτελέσματα.