Θεόφιλος Σπύρου ΑΕΜ: 8583

# ΠΑΡΑΛΛΗΛΟΠΟΙΗΣΗ ΑΛΓΟΡΙΘΜΟΥ MEANSHIFT

*Παράλληλα και διανεμημένα συστήματα – 7ο Εξάμηνο*

## Ανάλυση προβλήματος

Στην τρίτη εργασία ασχολούμαστε με την παραλληλοποίηση του αλγορίθμου Mean shift στην GPU με CUDA. Ο αλγόριθμος παίρνει σαν είσοδο ένα σύνολο διανυσμάτων (σημείων) **x** διάστασης d και στόχος είναι η εύρεση των διανυσμάτων **y** από κάθε σημείο προς το μέγιστο για την ομαδοποίηση των δεδομένων. Λειτουργεί επαναληπτικά και σε κάθε βήμα υπολογίζονται τα νέα, πιο ακριβή **y**, έως ότου το σφάλμα (μέτρο της διαφοράς των **y** του τρέχοντος βήματος από το προηγούμενο) γίνει μικρότερο από μία μικρή σταθερά ε.

## Πρώτες σκέψεις για παραλληλοποίηση

Ο Mean shift ενδείκνυται για παραλληλοποίηση σε δύο επίπεδα:

1) Ο αριθμητής και ο παρονομαστής του τύπου υπολογισμού των νέων διανυσμάτων **y** μπορούν να υπολογιστούν ανεξάρτητα ο ένας από τον άλλον

$$\boldsymbol{y}_i^{[k+1]} = \frac{\sum_{j=1}^{N} k\left(\left\|\boldsymbol{y}_i^{[k]} - \boldsymbol{x}_j\right\|_2^2\right)\boldsymbol{x}_j}{\sum_{j=1}^{N} k\left(\left\|\boldsymbol{y}_i^{[k]} - \boldsymbol{x}_j\right\|_2^2\right)}$$

2) Αφού υπολογιστούν όλοι οι αριθμητές και παρονομαστές, τότε μπορούμε να συνεχίσουμε με τον υπολογισμό του νέου **y** οπότε και του διανύσματος mean shift $\boldsymbol{m}_i^{[k]} = \boldsymbol{y}_i^{[k+1]} - \boldsymbol{y}_i^{[k]}$.

Τα παραπάνω γίνονται ακόμα πιο κατανοητά αν δούμε όλα τα διανύσματα σαν πίνακες διάστασης $N \times d$, δηλαδή $X, Y \in \mathbb{R}^{N\times d}$. Έτσι, ο παραπάνω τύπος απλοποιείται στη μορφή:

$$Y^{[k+1]} = k(Z\ .\hat{}\ 2) \cdot X\ ./\ k(Z\ .\hat{}\ 2) \cdot \vec{1}$$

Όπου $Z \in \mathbb{R}^{N\times N}$, $Z_{ij} = \begin{cases} \left\|\boldsymbol{y}_i^{[k]} - \boldsymbol{x}_j\right\|_2, & \alpha\nu\ \left\|\boldsymbol{y}_i^{[k]} - \boldsymbol{x}_j\right\|_2 > \sigma^2 \\ 0, & \alpha\lambda\lambda\iota\acute{\omega}\varsigma \end{cases}$

και $\vec{1} = [1 \quad \dots \quad 1]^T \in \mathbb{R}^d$. Οι τελεστές .^ και ./ είναι αυτοί του MATLAB, οι οποίοι πραγματοποιούν ύψωση σε δύναμη και διαίρεση αντίστοιχα ανά στοιχείο.

## Παραδοχές & Σημειώσεις

Καθώς η ανάγνωση δεδομένων από αρχείο mat είναι πολύπλοκη και απαιτεί βιβλιοθήκες που ενδεχομένως να μην είναι διαθέσιμες σε κάποια συστήματα, τα δεδομένα των

δύο αρχείων της άσκησης αποθηκεύτηκαν σε μορφή απλού κειμένου txt · ένα για κάθε μεταβλητή MATLAB στην πρώτη σειρά των οποίων αναγράφονται οι διαστάσεις του πίνακα που περιέχουν. Ακόμη οι σταθερές ε και σ ορίζονται στην αρχή του προγράμματος, οπότε για να αλλαχθούν χρειάζεται εκ νέου compile.

## Παραλληλοποίηση με CUDA

Στην αρχή εισάγονται τα δεδομένα από το αρχείο που ορίσθηκε κατά την εκτέλεσή του προγράμματος. Αφού γίνουν οι απαραίτητες κατανομές μνήμης για κάθε πίνακα που χρειάζεται στο Device (cudaMalloc), ξεκινάει η μεταφορά του πίνακα X στην GPU και αμέσως μετά αντιγράφονται τα δεδομένα του στον Y. Εδώ είναι σημαντικό να τονισθεί ότι όλοι οι πίνακες αποθηκεύονται σε «ξετυλιγμένη» (unrolled) μορφή στη μνήμη της κάρτας γραφικών, δηλαδή σαν διανύσματα με διάσταση το γινόμενο των διαστάσεων του πίνακα από τον οποίο προέκυψαν, και ανά σειρά.

Έπειτα ξεκινάει η επαναληπτική διαδικασία με την αρχικοποίηση του σφάλμτος, του αριθμητή και του παρονομαστή στο μηδέν. Εκτελείται το πρώτο kernel (cudaMeanshift1) με blocks διαστάσεων BLOCK_SIZE x BLOCK_SIZE[1]. Κάθε thread αντιστοιχεί σε ένα συγκεκριμένο i (rowY) και j (rowX), τα οποία δίνουν και τους δείκτες για τον πίνακα Z (colZ & rowZ). Έτσι, αφού σιγουρευτούμε ότι οι δείκτες που προέκυψαν είναι έγκυροι για τους πίνακες που έχουμε[2], υπολογίζουμε το τετράγωνο της νόρμας των διανυσμάτων $\mathbf{y}_i$ και $\mathbf{x}_j$ (difYiXj). Αν λοιπόν η ρίζα του difYiXj είναι μικρότερη του $\sigma^2$, τότε υπολογίζεται η γκαουσιανή της και προστίθεται[3] στη σειρά i του παρονομαστή, ενώ για τον αριθμητή υπολογίζονται τα ανά διάσταση γινόμενα της γκαουσιανής με την τιμή του $\mathbf{x}_j$ στην αντίστοιχη διάσταση και προστίθενται στη σειρά i και αντίστοιχη διάσταση του αριθμητή. Με άλλα λόγια, πραγματοποιούνται οι πράξεις του τύπου που σημειώθηκε παραπάνω και συμπληρώνονται οι πίνακες denom και numer διαστάσεων N x 1 και N x d αντίστοιχα. Στην περίπτωση που η απόσταση των δύο σημείων $\mathbf{y}_i$ και $\mathbf{x}_j$ είναι μεγαλύτερη από $\sigma^2$, τότε απλά δε συμμετέχουν στους υπολογισμούς.

Στο σημείο αυτό και προτού συνεχίσουμε στον υπολογισμό των νέων τιμών του Y, είναι απαραίτητο να έχουν τελειώσει όλα τα threads του προηγούμενου kernel, ώστε οι πίνακες του αριθμητή και του παρονομαστή να έχουν πάρει τις τελικές τιμές τους. Αυτό εξασφαλίζεται με την εισαγωγή ενός barrier στον Host μέσω της συνάρτησης cudaDeviceSynchronize.

Πλέον είμαστε έτοιμοι να εκτελέσουμε το δεύτερο και τελευταίο kernel του αλγορίθμου. Αυτή τη φορά εκκινούμε block πλήθους που εξαρτάται από τις διαστάσεις του Y και όχι του Z όπως προηγουμένως. Σε κάθε thread βρίσκουμε τις τιμές των row και col που υποδεικνύουν ποιο στοιχείο του πίνακα των νέων Y θα υπολογιστεί στο τρέχον thread. Στη συνέχεια ελέγχουμε την εγκυρότητα των δεικτών σύμφωνα με τις διαστάσεις του Y, όπως και προηγουμένως, και αν είναι εντάξει αποθηκεύουμε στην αντίστοιχη με το thread θέση του Y το νέο πηλίκο των στοιχείων του αριθμητή και του παρονομαστή

που του αναλογούν. Στην περίπτωση όπου ο παρονομαστής έχει κάποιο μηδενικό, τότε και το αντίστοιχο διάνυσμα του αριθμητή θα είναι το μηδενικό, καθώς ο αριθμητής εμπεριέχει μέσα του πολλαπλασιασμό με ίδιο όρο με τον παρονομαστή. Έτσι για να αποφευχθεί διαίρεση με το μηδέν, αντικαθιστούμε τον συγκεκριμένο παρονομαστή με τη μονάδα και στη θέση αυτή του νέου Y προκύπτει μηδέν χωρίς να δημιουργηθούν προβλήματα.

Στο εσωτερικό του ίδιου kernel πραγματοποιείται και ο υπολογισμός του σφάλματος της τρέχουσας επανάληψης. Αντί λοιπόν να αποθηκεύσουμε ολόκληρο τον πίνακα m, εφ' όσον δε χρειάζεται κάπου αλλού απλά προσθέτουμε το τετράγωνο της διαφοράς του νέου με το παλιό στοιχείο του Y που αναλογεί στο τρέχον thread. Το άθροισμα αυτό είναι το εσωτερικό της νόρμας (και όχι η νόρμα). Συνεχίζοντας, αντικαθίσταται η παλιά τιμή του Y για το συγκεκριμένο thread με την νέα της.

Ολοκληρώνοντας το βήμα της επανάληψης, και πάλι περιμένουμε να συγχρονιστούν όλα τα thread της GPU, μιας και πρέπει να ολοκληρωθεί ο υπολογισμός του mean shift πριν πάρουμε τη ρίζα του η οποία και αντιστοιχεί στη νόρμα του πίνακα m.

- **Υλοποίηση με shared memory:**

Η υλοποίηση με τη χρήση της κοινής μνήμης στο εσωτερικό κάθε block εισάγει μια επιπρόσθετη πολυπλοκότητα στον αλγόριθμο με σκοπό τη μείωση χρήσης της global memory. Πιο αναλυτικά, πρέπει κάθε thread στην αρχή της εκτέλεσής του να μεταφέρει τα δεδομένα που χρειάζεται από την global στη shared memory, να κάνει όποιες πράξεις στα δεδομένα αυτά (με τις φθηνότερες κλήσεις read/write στη shared memory) και αν αλλάξει κάτι που αφορά και threads εκτός του block του πρέπει να τοποθετήσει τις νέες τιμές στην global.

Η χρησιμότητα της shared memory προκύπτει (πέραν της αλληλεπίδρασης των threads ενός block μεταξύ τους που εδώ δεν συμβαίνει γιατί το κάθε ένα χρησιμοποιεί ξεχωριστό κομμάτι από την κοινή μνήμη) όταν κάθε thread χρησιμοποιεί πολύ συχνά κάποια θέση μνήμης, καθώς αν τη χρησιμοποιεί ελάχιστα χειροτερεύει ο χρόνος εκτέλεσης αντί να καλυτερεύει αφού πραγματοποιείται τουλάχιστον μία επιπλέον κλήση της global μνήμης (π.χ. $t_{rg} + t_{ws} + t_{rs} + t_{wg} > t_{rg} + t_{wg}$). Έτσι στο δεύτερο kernel είναι προφανές πως η χρήση της shared μνήμης είναι απαγορευτική για τους πίνακες denom και numer, ενώ για τον Y δεν κερδίζουμε κάτι ουσιαστικό αφού πραγματοποιούνται μονάχα δύο αναγνώσεις ανά thread.

Με παρόμοιο σκεπτικό, στο πρώτο kernel χρησιμοποιούμε shared memory μόνο για τον πίνακα $X^4$, γεγονός που ωφελεί κατά ένα μικρό βαθμό λόγω μικρότερου κόστους πράξεων. Στην πράξη όμως (για μικρά d) ο χρόνος χειροτερεύει, αφού γίνονται περισσότερες πράξεις για να βρούμε τους σχετικούς δείκτες για τους shared πίνακες και απαιτείται χρόνος για την καταχώρηση και μεταφορά της μνήμης. Για μεγάλα d τα πράγματα είναι καλύτερα, διότι αντικαθίστανται περισσότερες προσπελάσεις της global μνήμης.

Code Link: https://github.com/TheoSpyrou/Parallila/tree/master/Ergasia%203

Τέλος, στην shared υλοποίηση ο μέγιστος αριθμός των threads ανά block μειώνεται δραστικά, καθώς η shared μνήμη για κάθε block είναι μικρή (~10KB) γεγονός που περιορίζει ακόμα περισσότερο το μέγεθος των εισαγόμενων δεδομένων.

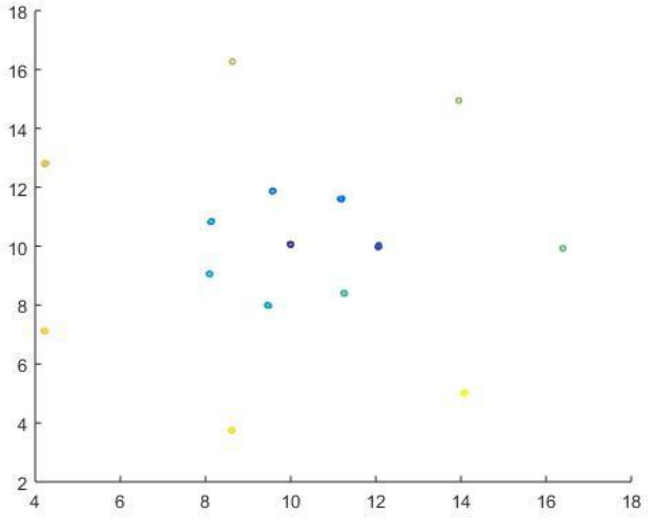
## Μετρήσεις χρόνων εκτέλεσης και αποτελέσματα (Διάδης)

| | | MATLAB | Σειριακός -O0 | Σειριακός -O3 | CUDA (global) | CUDA (shared) |
|---|---|---|---|---|---|---|
| **a** | **Χρόνος** | 0,62 secs | 0,25 secs | 0.069 secs | 0,0028 secs | 0,0031 secs |
| | **Επιτάχυνση** | 1x | 2.5x | 8x | **214x** | 194x |
| **b** | **Χρόνος** | 0,27 secs | 3,4 secs | 0,72 secs | 0,076 secs | 0,098 secs |
| | **Επιτάχυνση** | 13x | 1x | 5x | **45x** | 35x |

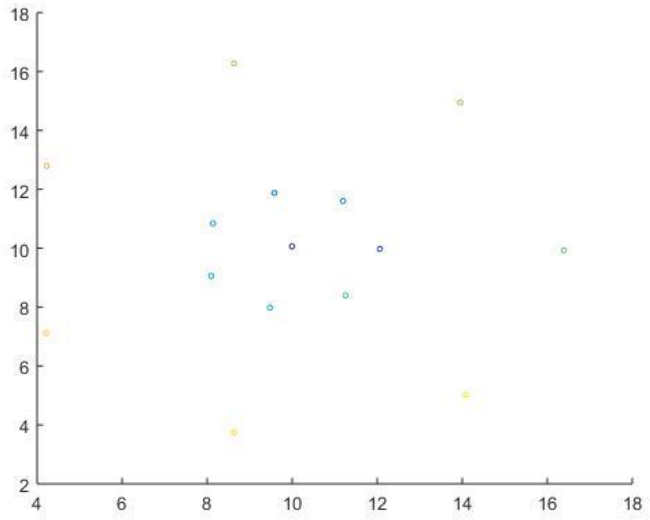
Οι παραπάνω χρόνοι αφορούν μόνο τη διάρκεια εκτέλεσης του αλγορίθμου στην GPU χωρίς τις καθυστερήσεις από μεταφορά δεδομένων από και προς την Device. Για τις μετρήσεις χρησιμοποιήθηκαν το data set διαστάσεων 600 x 2 που δόθηκε με την εκφώνηση της εργασίας (a) και ένα ακόμα διαστάσεων 1024 x 32 από την κατηγορία "DIM-sets (high)" της σελίδας https://cs.joensuu.fi/sipu/datasets (b). Σημειώνεται πως για την εκτέλεση με το (a) χρησιμοποιήθηκαν ε = 0.0001 και BLOCK_SIZE = 20 και με το (b) ε = 0.00015 και BLOCK_SIZE = 10.

## Παρατηρήσεις και συμπεράσματα

Παρατηρούμε πως και στις δύο περιπτώσεις η υλοποίηση CUDA (global) είναι η καλύτερη με αρκετά μεγάλη διαφορά. Η shared είναι πιο αργή σύμφωνα με τα όσα περιγράφθηκαν παραπάνω, ενώ με την αύξηση του d βλέπουμε να πλησιάζει την global, όπως αναμέναμε.




***Σχήμα 1.*** *Αποτελέσματα MATLAB (600 x 2)*




***Σχήμα 2.*** *Αποτελέσματα σειριακού και παράλληλου αλγορίθμου (600 x2)*

Code Link: https://github.com/TheoSpyrou/Parallila/tree/master/Ergasia%203

Τέλος, αν συγκρίνουμε προσεκτικά τα αποτελέσματα του κώδικα στο MATLAB με αυτά του σειριακού και του παράλληλου αλγορίθμου, θα δούμε ότι αυτά στο σχήμα 2 είναι ίδια ανά περιοχή. Αυτό συμβαίνει λόγω του ότι ο αλγόριθμος στο MATLAB είναι πιο αναλυτικός σε σχέση με τον απλό κλασματικό τύπο της εκφώνησης, οπότε και έχει πολύ μεγαλύτερη ακρίβεια. Το σφάλμα βέβαια είναι πολύ μικρό ανά σημείο (δεν έχει σχέση με το ε και το πλήθος των επαναλήψεων, αλλά με την πορεία εύρεσης του αποτελέσματος και γι' αυτό θα δούμε ότι οι τελευταίοι αλγόριθμοι ολοκληρώνονται με ελαφρώς διαφορετικά σφάλματα επαναλήψεων ανά εκτέλεση σε αντίθεση με αυτόν του MATLAB).

## Υποσημειώσεις

**1** Η σταθερά BLOCK_SIZE μπορεί να πάρει οποιαδήποτε ακέραια τιμή, αρκεί το $BLOCK\_SIZE^2$ να είναι μικρότερο από το μέγιστο πλήθος threads ανά block που επιτρέπει η κάρτα γραφικών που χρησιμοποιείται. Ένας επιπλέον περιορισμός αφορά το πλήθος registers ανά thread, οπότε και στις παράλληλες υλοποιήσεις χρησιμοποιήθηκαν οι μικρότεροι (από άποψη μνήμης) δυνατοί τύποι μεταβλητών (π.χ. float, unsigned short κτλ) για μέγιστη απόδοση.

**2** Ο έλεγχος της εγκυρότητας των δεικτών σε κάθε thread είναι απαραίτητος, καθώς ενδέχεται να μην ταιριάζει πάντα ακριβώς το πλήθος των threads με τις διαστάσεις των δεδομένων. Έτσι, μερικά threads αναπόφευκτα θα παραμείνουν απραγή. Γενικά η χρήση εντολών υπό συνθήκη (if, for κα) είναι κακή πρακτική στην CUDA. Στη συγκεκριμένη περίπτωση όμως δε δημιουργείται πρόβλημα, καθώς i) οι χρόνοι εκτέλεσης των threads είναι παραπλήσιοι (για οποιαδήποτε συνθήκη) και ii) τα for εκτελούνται για τον ίδιο αριθμό επαναλήψεων για όλα τα threads.

**3** Για την αποφυγή κάποιου πιθανού race condition λόγω της ταυτόχρονης εγγραφής σε ίδιες θέσεις μνήμης από πολλά threads, χρησιμοποιείται η συνάρτηση atomicAdd, η οποία με αριθμητή κινητής υποδιαστολής υποστηρίζεται για computing capability της GPU ≥ 2.0.

**4** Δοκιμάσθηκε και υλοποίηση με τους πίνακες Y, denom και numer του πρώτου kernel στη shared memory (επισυνάπτεται με σχόλια στον κώδικα του cudaMeanshift1 της shared υλοποίησης), αλλά απορρίφθηκε λόγω της επιπλέον καθυστέρησης που εισάγει μιας και χρειάζεται σε ελάχιστες κλήσεις ανά thread.