



ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ
ΕΡΓΑΣΤΗΡΙΟ ΥΠΟΛΟΓΙΣΤΙΚΩΝ ΣΥΣΤΗΜΑΤΩΝ
www.cslab.ece.ntua.gr

ΣΥΣΤΗΜΑΤΑ ΠΑΡΑΛΛΗΛΗΣ ΕΠΕΞΕΡΓΑΣΙΑΣ 9ο εξάμηνο ΗΜΜΥ, ακαδημαϊκό έτος 2022-23

Εξαμηνιαία Άσκηση

1 Εξοικείωση με το περιβάλλον προγραμματισμού

1.1 Σκοπός της Άσκησης

Σκοπός της συγκεκριμένης άσκησης είναι η εξοικείωση με τις υποδομές του εργαστηρίου (πρόσβαση στα συστήματα, μεταγλώττιση προγραμμάτων, υποβολή εργασιών κλπ) μέσα από την παραλληλοποίηση ενός απλού προβλήματος σε αρχιτεκτονικές κοινής μνήμης.

1.2 Conway's Game of Life

Το *Παιχνίδι της Ζωής* (*Conway's Game of Life*) λαμβάνει χώρα σε ένα ταμπλό με κελιά δύο διαστάσεων. Το περιεχόμενο κάθε κελιού μπορεί να είναι γεμάτο (alive) ή κενό (dead), αντικατοπτρίζοντας την ύπαρξη ή όχι ζωντανού οργανισμού σε αυτό, και μπορεί να μεταβεί από τη μία κατάσταση στην άλλη μία φορά εντός συγκεκριμένου χρονικού διαστήματος. Σε κάθε βήμα (χρονικό διάστημα), κάθε κελί εξετάζει την κατάστασή του και αυτή των γειτόνων του (δεξιά, αριστερά, πάνω, κάτω και διαγώνια) και ακολουθεί τους παρακάτω κανόνες για να ενημερώσει την κατάστασή του:

- Αν ένα κελί είναι ζωντανό και έχει λιγότερους από 2 γείτονες πεθαίνει από μοναξιά.
- Αν ένα κελί είναι ζωντανό και έχει περισσότερους από 3 γείτονες πεθαίνει λόγω υπερπληθυσμού.
- Αν ένα κελί είναι ζωντανό και έχει 2 ή 3 γείτονες επιβιώνει μέχρι την επόμενη γενιά.
- Αν ένα κελί είναι νεκρό και έχει ακριβώς 3 γείτονες γίνεται ζωντανό (λόγω αναπαραγωγής).

1.3 Δεδομένα

Σας δίνεται η σειριακή υλοποίηση του παιχνιδιού της ζωής, στον scirouter, στο αρχείο `/home/parallel/pps/2022-2023/a1/Game_Of_Life.c`.

1.4 Ζητούμενα

1. Λάβετε έγκαιρα τον κωδικό πρόσβασης στα συστήματα και επιλύστε τυχόν προβλήματα σύνδεσης.
2. Εξοικειωθείτε με τον τρόπο μεταγλώττισης και υποβολής εργασιών στις συστοιχίες. Για οδηγίες σύνδεσης, μεταγλώττισης, εκτέλεσης κ.λ.π. των προγραμμάτων σας συμβουλευτείτε τις "ΟΔΗΓΙΕΣ ΕΡΓΑΣΤΗΡΙΟΥ", που είναι διαθέσιμες στο site του helios

3. Αναπτύξτε παράλληλο πρόγραμμα στο μοντέλο κοινού χώρου διευθύνσεων (shared address space) με τη χρήση του OpenMP.
4. Πραγματοποιείτε μετρήσεις επίδοσης σε έναν από τους κόμβους της συστοιχίας των clones για 1,2,4,6,8 πυρήνες και μεγέθη ταμπλώ 64×64 , 1024×1024 και 4096×4096 (σε όλες τις περιπτώσεις τρέξτε το παιχνίδι για 1000 γενιές).
5. Συγκεντρώστε τα αποτελέσματα, τις συγκρίσεις και τα σχόλιά σας στην Τελική Αναφορά.
6. **Προαιρετικά:** Αναζητήστε στο διαδίκτυο ειδικές αρχικοποιήσεις του ταμπλώ που οδηγούν σε ενδιαφέροντα οπτικά αποτελέσματα και δώστε την εξέλιξη των γενιών σε μορφή κινούμενης εικόνας.

2 Παραλληλοποίηση και βελτιστοποίηση αλγορίθμων σε αρχιτεκτονικές κοινής μνήμης

2.1 Παραλληλοποίηση και βελτιστοποίηση του αλγορίθμου K-means

Σκοπός της άσκησης

Στόχος της άσκησης είναι η ανάπτυξη δυο παράλληλων εκδόσεων του αλγορίθμου K-means στο προγραμματιστικό μοντέλο του κοινού χώρου διευθύνσεων με τη χρήση του προγραμματιστικού εργαλείου OpenMP. Προς αυτό το σκοπό, θα δοκιμάσετε κάποιες συνήθεις βελτιώσεις υλοποίησης παράλληλου κώδικα και θα αξιολογήσετε την τελική επίδοση του παράλληλου προγράμματος.

Ο αλγόριθμος K-means

Σκοπός του αλγορίθμου είναι ο διαχωρισμός N αντικειμένων σε k μη επικαλυπτόμενες ομάδες (συστάδες - clusters). Ο αλγόριθμος λειτουργεί ως εξής:

```
until convergence (or fixed loops)
  for each object
    find nearest cluster
  for each cluster
    calculate new cluster center coordinates.
```

Οι εκδόσεις που καλείστε να αναπτύξετε είναι δύο: μια βασική έκδοση που παραλληλοποιεί τον σειριακό αλγόριθμο με προσθήκη καταλλήλων εντολών συγχρονισμού ώστε να γίνει ορθά η παράλληλη ενημέρωση του πίνακα `newClusters` με τις νέες υπολογιζόμενες συντεταγμένες των κέντρων των συστάδων, και μια πιο εξελιγμένη έκδοση που χρησιμοποιεί τοπικούς (copied) πίνακες για κάθε νήμα, ώστε να μπορεί να γίνεται η εγγραφή χωρίς συγχρονισμό. Στη δεύτερη έκδοση μετά τους τοπικούς υπολογισμούς τα αποτελέσματα των τοπικών αυτών πινάκων θα πρέπει να συνδυάζονται (reduction) στον πίνακα `newClusters`.

Δεδομένα

Για την υλοποίηση του αλγορίθμου σας δίνεται ο σειριακός αλγόριθμος, καθώς και σκελετοί και για τις δύο παράλληλες υλοποιήσεις στο `/home/parallel/pps/2022-2023/a2/kmeans/`, στους οποίους καλείστε να συμπληρώσετε τα κενά. Η κλήση όλων των υλοποιήσεων γίνεται μέσω της `main.c`, η οποία είναι υπεύθυνη για την δημιουργία ενός ζητούμενου μεγέθους dataset (καλώντας την `file_io.c`) και την κλήση του κατάλληλου αλγορίθμου K-means.

Ζητούμενα

Για τις 2 παράλληλες εκδόσεις που σας δίνονται:

shared clusters

1. Παραλληλοποιήστε αυτή την έκδοση του αλγορίθμου, προσέχοντας το διαμοιρασμό των δεδομένων μεταξύ των threads, χρησιμοποιώντας το OpenMP και πραγματοποιήστε μετρήσεις για το configuration `{Size, Coords, Clusters, Loops} = {256, 16, 16, 10}` για threads = {1, 2, 4, 8, 16, 32, 64} στο μηχανήμα *sandman*. Δημιουργήστε το barplot διάγραμμα χρόνου εκτέλεσης που προκύπτει (x-axis = sequential και threads, y-axis = time), και το αντιστοίχο speedup plot (x-axis = sequential και threads, y-axis = seq_time/time). Πώς μπορείτε να εξηγήσετε την επίδοση της παράλληλης έκδοσης σε σχέση με την σειριακή;
2. Αναζητήστε πληροφορίες για τη χρήση της μεταβλητής περιβάλλοντος `GOMP_CPU_AFFINITY` (και χρησιμοποιήστε τη σε όλα τα υπόλοιπα ερωτήματα), η οποία "προσδένει" τα threads σε συγκεκριμένους πυρήνες για όλη την εκτέλεση (thread binding) και επαναλάβετε τις μετρήσεις και τα διαγράμματα. Τι διαφορά βλέπετε και πώς τη δικαιολογείτε;

copied clusters and reduce

1. Παραλληλοποιήστε αυτή την έκδοση του αλγορίθμου (αγνοώντας τα *"hints για false-sharing"*) και επαναλάβετε τις προαναφερόμενες μετρήσεις και διαγράμματα. Συγκρίνετε την επίδοση και τον παραλληλισμό σε σχέση με την προηγούμενη έκδοση. Επιτυγχάνει η συγκεκριμένη έκδοση ικανοποιητική επίδοση;
2. Δοκιμάστε το ακόλουθο configuration $\{\text{Size, Coords, Clusters, Loops}\} = \{256, 1, 4, 10\}$, και συγκρίνετε το scalability με το προηγούμενο $\{\text{Size, Coords, Clusters, Loops}\} = \{256, 16, 16, 10\}$. Τι διαφορά παρατηρείτε στα scalability plots; Αναζητήστε πληροφορίες για την πολιτική **first-touch** του Linux και για τα φαινόμενα **false-sharing**. Με βάση αυτά, τι πρόβλημα δημιουργείται στην πραγματική τοποθέτηση των `local_newClusters` δεδομένων στη μνήμη; Προσπαθήστε να ξεπεράσετε αυτό το πρόβλημα (Hint: οι memory allocators προσπαθούν να αποφύγουν τον κατακερματισμό της μνήμης). Αναφέρετε τον καλύτερο χρόνο που επιτύχατε!
3. **Προαιρετικά (bonus)** - NUMA aware allocation: Στο configuration $\{\text{Size, Coords, Clusters, Loops}\} = \{256, 1, 4, 10\}$ του προηγούμενου ερωτήματος δοκιμάστε να βελτιώσετε την επίδοση του προγράμματος, λαμβάνοντας υπόψη τα NUMA χαρακτηριστικά του μηχανήματος και το διαμοιρασμό του πίνακα *objects* στα memory nodes, ο οποίος αρχικοποιείται στο αρχείο `"file_io.c"` (Hint: αξιοποιήστε την πολιτική first-touch των Linux που είδαμε πιο πάνω). Δοκιμάστε και το προηγούμενο configuration $\{\text{Size, Coords, Clusters, Loops}\} = \{256, 16, 16, 10\}$ με την νέα έκδοσή. Τι παρατηρείτε; Ποιό είναι το κυρίαρχο bottleneck σε κάθε configuration;

Αμοιβαίος Αποκλεισμός - Κλειδώματα

Στο ερώτημα αυτό καλείστε να αξιολογήσετε διαφορετικούς τρόπους υλοποίησης κλειδωμάτων για αμοιβαίο αποκλεισμό. Το κρίσιμο τμήμα που θα χρειαστεί να προστατευτεί είναι η ενημέρωση του πίνακα `newClusters` στην `"shared-clusters"` υλοποίηση του Kmeans.

Συγκεκριμένα, θα μελετήσετε τα παρακάτω είδη κλειδωμάτων:

- **nosync_lock**: Η συγκεκριμένη υλοποίηση δεν παρέχει αμοιβαίο αποκλεισμό οπότε δεν παράγει και σωστά αποτελέσματα. Ωστόσο, θα χρησιμοποιηθεί ως άνω όριο για την αξιολόγηση της επίδοσης των υπόλοιπων κλειδωμάτων.
- **pthread_mutex_lock**: Το `pthread_mutex_t` κλειδωμά που παρέχει η βιβλιοθήκη Pthreads.
- **pthread_spin_lock**: Το `pthread_spinlock_t` κλειδωμά που παρέχει η βιβλιοθήκη Pthreads.
- **tas_lock**: Το *test-and-set* κλειδωμά όπως περιγράφεται στις διαφάνειες του μαθήματος.
- **ttas_lock**: Το *test-and-test-and-set* κλειδωμά όπως περιγράφεται στις διαφάνειες του μαθήματος.
- **array_lock**: Το array-based κλειδωμά όπως περιγράφεται στις διαφάνειες του μαθήματος.
- **clh_lock**: Ένα είδος κλειδώματος που στηρίζεται στη χρήση μίας συνδεδεμένης λίστας. Αναλυτικές πληροφορίες μπορείτε να βρείτε στο Κεφάλαιο 7 του βιβλίου *"The Art of Multiprocessor Programming"*.

Στο path της άσκησης θα βρείτε updated το Makefile για την παραγωγή των εκτελέσιμων της μορφής `kmeans_omp_<lock-name>_lock`, για κάθε κλειδωμά ξεχωριστά, καθώς και τα αρχεία:

- `omp_lock_kmeans.c` : χρησιμοποιούνται τα διάφορα κλειδώματα για την προστασία του κρίσιμου τμήματος.
- `omp_critical_kmeans.c` : χρησιμοποιείται το OpenMP directive `#pragma omp critical` για την προστασία του κρίσιμου τμήματος.

Στο subdirectory `locks` θα βρείτε έτοιμες τις υλοποιήσεις των κλειδωμάτων.

1. Συγκεντρώστε μετρήσεις για κάθε είδος lock με το configuration $\{\text{Size, Coords, Clusters, Loops}\} = \{16, 16, 16, 10\}$ για `threads = \{1, 2, 4, 8, 16, 32, 64\}` στο μηχανήμα `sandman` (εφαρμόζοντας `thread-binding`). Συγκρίνετε τους χρόνους εκτέλεσης (τόσο των κλειδωμάτων, όσο και του `omp critical`) με την `shared-clusters` υλοποίησή σας από την προηγούμενη άσκηση όπου

για την προστασία του κρίσιμου τμήματος χρησιμοποιήσατε τα OpenMP directives `#pragma omp critical` ή/και `#pragma omp atomic`. Εξηγήστε τη λειτουργία του κάθε κλειδώματος. Πώς αυτή επηρεάζει την επίδοση του;

2.2 Παραλληλοποίηση του αλγορίθμου Floyd-Warshall

Σκοπός της άσκησης

Στόχος της άσκησης είναι να εξοικειωθείτε με τη χρήση των OpenMP tasks παράλληλοποιώντας τον αλγόριθμο Floyd-Warshall για αρχιτεκτονικές κοινής μνήμης.

Ο αλγόριθμος Floyd-Warshall

Ο αλγόριθμος των Floyd-Warshall (FW) υπολογίζει τα ελάχιστα μονοπάτια ανάμεσα σε όλα τα ζεύγη των N κόμβων ενός γράφου (all-pairs shortest path). Θεωρώντας το γράφο αποθηκευμένο στον πίνακα γειτνίασης A , ο αλγόριθμος έχει ως εξής:

```
for (k=0; k<N; k++)
  for (i=0; i<N; i++)
    for (j=0; j<N; j++)
      A[i][j] = min(A[i][j], A[i][k]+A[k][j]);
```

Εκτός από την *standard* έκδοση του αλγορίθμου, έχουν προταθεί άλλες δύο εκδόσεις, μία *αναδρομική (recursive)* και μία *tiled*. Σε αυτή την άσκηση καλείστε να παραλληλοποιήσετε την **recursive** υλοποίηση, η οποία δεν μπορεί να παραλληλοποιηθεί με `parallel for` αλλά απαιτείται για αυτή την περίπτωση η χρήση `tasks`.

Δεδομένα

Η σειριακή έκδοση του αλγορίθμου Floyd-Warshall (3 versions) σας δίνεται στον scirouter στο φάκελο `/home/parallel/pps/2022-2023/a2/FW/`.

Ζητούμενα

Υλοποιήστε μια παράλληλη έκδοση του **recursive** αλγορίθμου Floyd-Warshall χρησιμοποιώντας OpenMP tasks και πραγματοποιήστε μετρήσεις για μεγέθη πινάκων 1024×1024 , 2048×2048 και 4096×4096 για `threads = {1, 2, 4, 8, 16, 32, 64}` στο μηχάνημα *sandman*. Δημιουργήστε τα 3 barplot διαγράμματα χρόνου εκτέλεσης που προκύπτουν (x-axis = sequential και threads, y-axis = time).

Προαιρετικά (bonus): Υλοποιήστε παράλληλη έκδοση του **tiled** αλγορίθμου Floyd-Warshall και συγκεντρώστε μετρήσεις για όμοια μεγέθη πινάκων. Συγκρίνετε επιδόσεις με την recursive έκδοση. Τι παρατηρείτε;

2.3 Ταυτόχρονες Δομές Δεδομένων

Σκοπός της άσκησης

Σκοπός του συγκεκριμένου ερωτήματος της άσκησης είναι η εξοικείωση με την εκτέλεση εφαρμογών σε σύγχρονα πολυπύρρηνα συστήματα και η αξιολόγηση της επίδοσής τους. Συγκεκριμένα, σας δίνονται διάφορες ταυτόχρονες υλοποιήσεις μίας απλά συνδεδεμένης ταξινομημένης λίστας και καλείστε να εκτελέσετε κάποια πειράματα με αυτές με στόχο την αξιολόγηση της επίδοσής τους κάτω από διαφορετικές συνθήκες.

Δεδομένα

Στον φάκελο `/home/parallel/pps/2022-2023/a2/conc_11/` σας δίνονται έτοιμες οι παρακάτω ταυτόχρονες υλοποιήσεις μίας απλά συνδεδεμένης ταξινομημένης λίστας:

- Coarse-grain locking
- Fine-grain locking
- Optimistic synchronization
- Lazy synchronization
- Non-blocking synchronization

Αφού αντιγράψετε τα περιεχόμενα του φακέλου σε κάποια τοποθεσία στο home directory σας, εκτελώντας `make` δημιουργούνται τα έξι εκτελέσιμα, συμπεριλαμβανομένης και της σειριακής έκδοσης. Όλα τα εκτελέσιμα χρησιμοποιούνται με ακριβώς τον ίδιο τρόπο αφού έχουν παραχθεί χρησιμοποιώντας την ίδια `main` συνάρτηση η οποία ορίζεται στο αρχείο `main.c`. Κάθε εκτελέσιμο παίρνει 4 ορίσματα, το εύρος των κλειδιών που θα χρησιμοποιηθούν από τα νήματα και το ποσοστό των λειτουργιών που θα είναι αναζητήσεις (`contains()`), εισαγωγές (`add()`) και διαγραφές (`remove()`). Επίσης, ο αριθμός των ταυτόχρονων νημάτων που θα εκτελεστούν ρυθμίζεται με χρήση της μεταβλητής περιβάλλοντος `MT_CONF`. Η ίδια μεταβλητή καθορίζει και σε ποιόν πυρήνα του μηχανήματος θα τρέξει το κάθε νήμα. Για παράδειγμα αν ορίσουμε `MT_CONF=0,2,4,8` θα εκτελεστούν 4 νήματα τα οποία θα καταλαμβάνουν τους πυρήνες 0, 2, 4 και 8 αντίστοιχα του μηχανήματος.

Ζητούμενα

Εκτελέστε πειράματα χρησιμοποιώντας τις παρακάτω τιμές για τις παραμέτρους των εκτελέσιμων:

- Αριθμός νημάτων: 1, 2, 4, 8, 16, 32, 64, 128
- Μέγεθος λίστας: 1024, 8192
- Ποσοστό λειτουργιών: 100-0-0, 80-10-10, 20-40-40, 0-50-50. Ο πρώτος αριθμός υποδηλώνει το ποσοστό αναζητήσεων και οι επόμενοι δύο το ποσοστό εισαγωγών και διαγραφών αντίστοιχα.

Εξηγήστε τη συμπεριφορά των διαφορετικών ταυτόχρονων υλοποιήσεων δίνοντας κατάλληλα διαγράμματα. Τα διαγράμματα πρέπει να είναι ξεκάθαρα και μπορούν να περιλαμβάνουν απόλυτα νούμερα `throughput` (δηλαδή λειτουργίες ανά μονάδα χρόνου), κανονικοποιημένες τιμές ή ότι άλλο θεωρήσετε απαραίτητο.

Σημείωση: σε όλες τις εκτελέσεις θα θέσετε κατάλληλα την μεταβλητή περιβάλλοντος `MT_CONF` ώστε τα νήματα να καταλαμβάνουν διαδοχικούς πυρήνες, π.χ. τα 16 νήματα εκτελούνται στους πυρήνες 0-15. Στις εκτελέσεις με 128 νήματα, κάθε πυρήνας θα εκτελεί δύο νήματα.

2.4 Περιβάλλον εκτέλεσης

- Για την άσκηση αυτή, θα χρησιμοποιήσετε το μηχανήμα *sandman*, που ανήκει στην ουρά *serial* (βλ. παρουσίαση άσκησης). Για να υποβάλλετε ένα script, έστω `script.sh`, στο μηχανήμα, δίνετε την εντολή `qsub` ως εξής:

```
$ qsub -q serial -l nodes=sandman:ppn=64 script.sh
```
- ΠΡΟΣΟΧΗ: Μπορείτε να χρησιμοποιείτε τα μηχανήματα της συστοιχίας `parlab` για την ανάπτυξη και τον έλεγχο παράλληλων προγραμμάτων.
- Για να ρυθμίσετε το περιβάλλον μεταγλώττισης και εκτέλεσης για OpenMP θα χρησιμοποιήσετε τα `modules`. Για το OpenMP φορτώνετε το αντίστοιχο module με `module load openmp`. Λεπτομέρειες για τη χρήση των `modules` μπορείτε να βρείτε εδώ:
<https://trac.cs.lab.ece.ntua.gr/wiki/EnvModulesUsage>.

3 Παραλληλοποίηση και βελτιστοποίηση αλγορίθμων σε επεξεργαστές γραφικών **NEW!**

3.1 Άλλη μια παραλληλοποίηση και βελτιστοποίηση του αλγορίθμου K-means

Σκοπός της άσκησης

Στόχος της άσκησης είναι η υλοποίηση και βελτιστοποίηση του αλγορίθμου K-means σε μια κάρτα γραφικών NVIDIA με τη χρήση του εργαλείου CUDA. Προς αυτό το σκοπό, θα φτιάξετε μια αρχική υλοποίηση που να λειτουργεί σωστά, και στη συνέχεια με βάση αυτή θα δοκιμάσετε κάποιες συνήθεις βελτιώσεις επίδοσης για κάρτες γραφικών καθώς και θα αξιολογήσετε την τελική επίδοση του παράλληλου προγράμματος.

Δεδομένα

Για την υλοποίηση του αλγορίθμου σας δίνεται ο σειριακός αλγόριθμος, καθώς και 5 (!) σκελετοί για τις διάφορες εκδόσεις στο `/home/parallel/pps/2022-2023/a3/kmeans/`, στους οποίους καλείστε να συμπληρώσετε τα κενά. Η κλήση όλων των υλοποιήσεων γίνεται μέσω της `main.c` (`main_seq.c` for CPU/C++, `main_gpu.cu` for GPU/nvcc), η οποία είναι υπεύθυνη για την δημιουργία ενός ζητούμενου μεγέθους dataset (καλώντας την `file_io.c`) και την κλήση του κατάλληλου αλγορίθμου K-means, όμοια με τον σκελετό του OpenMP. Το compilation γίνεται μέσω Makefile (e.g. 'make') το οποίο σας δίνεται και δεν χρειάζεται να αλλάξετε (εκτός αν θέλετε εσείς να προσθέσετε κάποιο optimization flag etc).

Ζητούμενα

Στα περιθώρια αυτής της άσκησης καλείστε να δουλέψετε πάνω στους σκελετούς που σας δίνονται με την εξής σειρά:

Naive version

Σε αυτή την έκδοση θα αναθέσετε στην κάρτα γραφικών το πιο υπολογιστικά βαρύ κομμάτι του αλγορίθμου: τον υπολογισμό των nearest clusters σε κάθε iteration. Συμπληρώστε αυτή την αρχική έκδοση του αλγορίθμου `cuda_kmeans_naive.cu` υλοποιώντας όλα τα **TODO**:

1. Υλοποιήστε τον πυρήνα `find_nearest_cluster` (και τις υπο-ρουτίνες του `euclid_dist_2`, `get_tid`) για εκτέλεση στην GPU. ΠΡΟΣΟΧΗ: Φροντίστε για το σωστό διαμοιρασμό των δεδομένων σε threads θέτοντας το κατάλληλο μέγεθος `dim`.
2. Πραγματοποιήστε τις ζητούμενες μεταφορές δεδομένων σε κάθε iteration του αλγορίθμου, ώστε να γίνει δυνατή η εκτέλεση του υπόλοιπου kmeans (νέα cluster centers) στην CPU μετά από κάθε GPU kernel invocation.

Στη συνέχεια αξιολογήστε την επίδοση σας:

1. Πραγματοποιήστε μετρήσεις για την *naive version* και την σειριακή έκδοση που σας δίνεται για το configuration `{Size, Coords, Clusters, Loops} = {256, 2, 16, 10}` και για `block_size = {32, 64, 128, 256, 512, 1024}`. Δημιουργήστε το barplot διάγραμμα χρόνου εκτελεσης που προκύπτει (x-axis = sequential και `block_size`, y-axis = time), και το αντιστοιχο speedup plot χωρίς την σειριακή (x-axis = `block_size`, y-axis = `seq_time/time`). Σχολιάστε την επίδοση της παράλληλης έκδοσης σε σχέση με την σειριακή.
2. Με βάση τα ίδια plot, σχολιάστε και την συμπεριφορά της επίδοσης σε σχέση με το `block_size`. Πως δικαιολογείτε την συμπεριφορά που βλέπετε για τη συγκεκριμένη έκδοση?

Transpose version

Σε αυτή την έκδοση καλείστε να αλλάξετε απλά την δομή των δεδομένων που έχουν διαστάσεις (e.g. `objects & clusters`) από row-based σε column-based indexing. Αυτό σημαίνει πως πρέπει

όλοι οι σχετικοί πίνακες να γίνουν transpose με τη χρήση buffers όπου αναγράφεται. Συμπληρώστε την έκδοση του αλγορίθμου `cuda_kmeans_transpose.cu` υλοποιώντας όλα τα νέα **TODO** και χρησιμοποιώντας τα κομμάτια από την *naive version* για ό,τι έχετε ήδη υλοποιήσει:

1. Υλοποιήστε τον πυρήνα `euclid_dist_2_transpose` για τη νέα δομή των δεδομένων.
2. Φροντίστε για τη σωστή αρχικοποίηση και μετατροπή δεδομένων στη νέα μορφή όπου αναγράφεται.

Στη συνέχεια αξιολογήστε την επίδοση σας:

1. Επαναλάβετε τις παραπάνω μετρήσεις για την *transpose version* και σχολιάστε την επίδοση της παράλληλης έκδοσης σε σχέση με την *naive*, τοποθετώντας τις στο ίδιο προαναφερόμενο διάγραμμα speedup. Παίζει κάποιο διαφορετικό ρόλο το `block_size`?
2. Πως δικαιολογείτε την διαφορά στην επίδοση μεταξύ των δύο φαινομενικά αντίστοιχων εκδόσεων με απλά άλλη δομή δεδομένων? HINT: Σκεφτείτε πώς γίνονται οι προσβάσεις στη μνήμη από 'κοντινά' threads για κάθε μια από τις δύο περιπτώσεις - όταν οποιοδήποτε thread διαβάζει ένα στοιχείο, 1) ποιο στοιχείο διαβάζει το ακριβώς επόμενο και 2) ποιο είναι δίπλα του στη μνήμη?

Shared version

Σε αυτή την έκδοση καλείστε να τοποθετήσετε τα `clusters` στην διαμοιραζόμενη μνήμη της GPU ώστε να μπορούν τα στοιχεία κάθε block να τα κάνουν access πιο γρήγορα. Συμπληρώστε την έκδοση του αλγορίθμου `cuda_kmeans_shared.cu` υλοποιώντας όλα τα νέα **TODO** και χρησιμοποιώντας τα κομμάτια από την *transpose version* για ό,τι έχετε ήδη υλοποιήσει:

1. Ορίστε το μέγεθος της διαμοιραζόμενης μνήμης που χρειάζεται η υλοποίησή σας.
2. Προσθέστε στον πυρήνα `find_nearest_cluster` την μεταφορά των `clusters` στην διαμοιραζόμενη μνήμη. ΠΡΟΣΟΧΗ: προσέξτε ποιος βλέπει κάθε μνήμη και φροντίστε να μην γίνουν άσκοπες extra μεταφορές από threads.

Στη συνέχεια αξιολογήστε την επίδοσή σας:

1. Επαναλάβετε τις παραπάνω μετρήσεις για την *shared version* και σχολιάστε την επίδοση της παράλληλης έκδοσης σε σχέση με την *naive* και την *transpose* τοποθετώντας τις στο ίδιο προαναφερόμενο διάγραμμα speedup. Παίζει κάποιο διαφορετικό ρόλο το `block_size` και γιατί?

Σύγκριση υλοποιήσεων / bottleneck Analysis

Εφόσον ολοκληρώσετε τις 3 υλοποιήσεις, κάντε μια πιο σε-βάθος ανάλυση της επίδοσης τους και πιθανών προβλημάτων.

1. Με την προσθήκη κατάλληλων timer (e.g. GPU part, CPU-GPU transfers, CPU part) εντός του while loop, αξιολογήστε τι μπορεί να εμποδίζει την συνολική επίδοση του iterative μέρους (αγνοήστε τα εξωτερικά κοστη μεταφορών/αρχικοποιήσεων).
2. Δοκιμάστε και το configuration `{Size, Coords, Clusters, Loops} = {256, 16, 16, 10}` για `block_size = {32, 64, 128, 256, 512, 1024}` για όλες τις υλοποιήσεις και σχολιάστε/αιτιολογήστε τι διαφέρει. Πιστεύετε είναι η παρούσα *shared* υλοποίηση κατάλληλη για την επίλυση του kmeans για arbitrary configurations?

Bonus: Full-Offload (All-GPU) version

Σε αυτή την έκδοση καλείστε να υλοποιήσετε ολόκληρο τον kmeans στην GPU, δηλαδή στην προηγούμενη *shared* να προσθέσετε και τον υπολογισμό των νέων cluster centroid (`update_centroids`) στην GPU. Συμπληρώστε την έκδοση του αλγορίθμου `cuda_kmeans_all_gpu.cu` υλοποιώντας όλα τα νέα **TODO** και χρησιμοποιώντας τα κομμάτια από την *shared version* για ό,τι έχετε ήδη υλοποιήσει:

1. Υπάρχουν πολλοί τρόποι να γίνει η υλοποίηση της `update_centroids` σε συνδυασμό με προσθήκες στην `find_nearest_cluster` η/και ακόμα με παραπάνω `kernels`. Διαλέξτε κάτι με στόχο α) να δουλεύει και β) να βολεύει. HINT: Κυρίως σκεφτείτε α) ποιά κομμάτια χρειάζονται καθολικό συγχρονισμό και β) πόσα `thread` θα δουλεύουν σε κάθε κομμάτι/kernel και πόσα θα καταλήγουν ανενεργά ανάλογα με την τοποθέτησή σας.

Στη συνέχεια αξιολογήστε την επίδοσή σας:

1. Επαναλάβετε τις παραπάνω μετρήσεις για την *all-gpu version* και σχολιάστε την επίδοση της παράλληλης έκδοσης σε σχέση με την *naive*, την *transpose* και την *shared version* μελετώντας για τα δύο προαναφερόμενα `configuration` (`{256, 2, 16, 10}` και `{256, 16, 16, 10}`) τα διαγράμματα `speedup`.
2. Φαίνεται να παίζει κάποιο διαφορετικό ρόλο το `block_size` και γιατί?
3. Είναι το κομμάτι `update_centroids` κατάλληλο για GPUs? Με βάση την απάντησή σας και την προηγούμενη ανάλυση των `bottleneck`, πώς αιτιολογείτε την διαφορά επίδοσης μεταξύ των παλιών εκδόσεων και της *all-gpu version*?
4. Τι διαφέρει μεταξύ των δύο `configuration` και πώς αιτιολογείτε την διαφορά επίδοσης?

3.2 Περιβάλλον εκτέλεσης

- Για την άσκηση αυτή, θα χρησιμοποιήσετε το μηχάνημα *dungani*, που ανήκει στην ουρά *serial*. Για να υποβάλλετε ένα `script`, έστω `script.sh`, στο μηχάνημα, δίνετε την εντολή `qsub` ως εξής:

```
$ qsub -q serial -l nodes=dungani:ppn=8 script.sh
```
- ΠΡΟΣΟΧΗ: ΔΕΝ μπορείτε να χρησιμοποιείτε τα μηχανήματα της συστοιχίας *parlab* για την ανάπτυξη και τον έλεγχο παράλληλων προγραμμάτων, καθώς χρειάζεστε τον `compiler` της NVIDIA `nvc` και κάποια NVIDIA GPU για να εκτελεστεί. Επομένως να είστε επιπλέον προσεκτικοί με τα `resources` και τα χρονικά όρια που βάζετε όταν τρέχετε στο μηχάνημα!