ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΣΥΣΤΗΜΑΤΑ ΠΑΡΑΛΛΗΛΗΣ ΕΠΕΞΕΡΓΑΣΙΑΣ

ΑΝΑΦΟΡΑ 4ης ΑΣΚΗΣΗΣ

Στοιχεία Ομάδας

- Αναγνωριστικό: parlab05
- Μέλος 1ο: Πέππας Μιχαήλ – Αθανάσιος, Α.Μ: 03121026
- Μέλος 2ο: Σαουνάτσος Ανδρέας, Α.Μ: 03121197
- Ημερομηνία Παράδοσης Αναφοράς: 20.10.2025

▪ Εισαγωγή

Σκοπός της άσκησης είναι η παραλληλοποίηση και η βελτιστοποίηση του αλγορίθμου K-means σε επεξεργαστές γραφικών NVIDIA, μέσω CUDA. Η υλοποίηση βασίζεται στο μοντέλο CPU-GPU: η CPU (host) προετοιμάζει τα δεδομένα, εκκινεί τα kernels στην GPU (device) και (ανάλογα το πρόγραμμα) εκτελεί μέρος του αλγορίθμου και διαχειρίζεται μεταφορές μνήμης. Η άσκηση συγκρίνει 4 διαδοχικές εκδόσεις του αλγορίθμου, οι οποίες στοχεύουν στην ανάδειξη και αξιολόγηση κλασικών παραγόντων επίδοσης GPU: προσπελάσεις global memory/coalescing, αξιοποίηση shared memory, κόστος atomic operations και overhead επικοινωνίας host-device.

Σύμφωνα με τα ζητούμενα της άσκησης, δουλέψαμε και παραγάγαμε 4 εκδόσεις του αλγορίθμου K-means:

1. Naive (cuda_kmeans_naive.cu): η GPU υπολογίζει μόνο την ανάθεση στο κοντινότερο cluster ανά αντικείμενο. Η ενημέρωση των κέντρων (update_centroids) γίνεται στην CPU, με μεταφορές host ↔ device ανά επανάληψη.
2. Transpose (cuda_kmeans_transpose.cu): αναδιάταξη δεδομένων σε column-major/transpose μορφή για βελτιωμένο memory coalescing στις προσπελάσεις της GPU (κοντινά threads διαβάζουν γειτονικές διευθύνσεις).
3. Shared (cuda_kmeans_shared.cu): επιπλέον φόρτωση των cluster centers στη shared memory ανά block, ώστε οι επαναλαμβανόμενες αναγνώσεις των clusters κατά τον υπολογισμό αποστάσεων να γίνονται από ταχύτερη on-chip μνήμη.
4. All-GPU (cuda_kmeans_all_gpu.cu): πλήρες offload και του update_centroids στη GPU. Η συσσώρευση (sums/counts) υλοποιείται με atomics, αναδεικνύοντας το κόστος συγχρονισμού/contention ως πιθανό bottleneck.

Οι παραπάνω αυτές εκδοχές είναι αυτές που θα αναλύσουμε και θα συγκρίνουμε στη συνέχεια.

▪ Ενότητα 3.1 – Naive Version

Α. Εισαγωγή

Η «naive» έκδοση μεταφέρει στη GPU μόνο το πιο υπολογιστικά βαρύ βήμα του K-means: την ανάθεση κάθε αντικειμένου στο κοντινότερο κέντρο ενός cluster. Η ενημέρωση των κέντρων (update_centroids: αθροίσματα/πλήθη/μέσοι όροι) παραμένει στην CPU. Έτσι, σε κάθε επανάληψη εκτελούνται:

- Host → Device: αντιγραφή των τρεχόντων cluster centers στη GPU,
- GPU kernel: υπολογισμός nearest cluster για κάθε object και ενημέρωση membership/delta,
- Device → Host: αντιγραφή membership και delta πίσω στην CPU,
- CPU: update_centroids, παραγωγή νέων cluster centers για το επόμενο loop.

Η αρχική αντιγραφή του συνόλου των objects (dataset) προς τη GPU γίνεται μία φορά πριν το while-loop (initialization) και δεν αποτελεί μέρος του «per-loop» breakdown.

Ο κώδικας του αρχείου μας (cuda_kmeans_naive.cu) παρατίθεται ακολούθως:

Β. Υλοποίηση και Ορθότητα

(α) Υπολογισμός απόστασης (euclid_dist_2)

Υλοποιείται η τετραγωνική Ευκλείδεια απόσταση σε μορφή row-major (naive έκδοση):

$$\mathrm{d}^2(x, c) = \sum_{i=1}^{n} (x_i - c_i)^2$$

με indexing objects[objectId*numCoords + i], clusters[clusterId*numCoords + i]. Αυτή η προσέγγιση ακολουθήθηκε για τη naive μορφή δεδομένων.

(β) Kernel find_nearest_cluster: αντιστοίχιση threads σε objects

Κάθε thread αντιστοιχεί σε ένα object μέσω global thread id:

tid = blockIdx.x * blockDim.x + threadIdx.x.

Ο αριθμός blocks ορίζεται ως ceil(numObjs / block_size), ώστε να καλύπτονται όλα τα objects, και γίνεται έλεγχος ορίων (tid < numObjs).

(γ) Υπολογισμός delta με atomics

Η μεταβλητή delta μετρά πόσα objects άλλαξαν cluster σε μία επανάληψη. Στο kernel, αν το νέο clusterId διαφέρει από το παλιό membership[tid], γίνεται atomicAdd(devdelta, 1).

Η επιλογή atomics είναι σωστή για αποφυγή race conditions, αλλά αποτελεί και κλασικό σημείο bottleneck (contention) όταν πολλά threads ενημερώνουν την ίδια global μεταβλητή. Δηλαδή, τα atomics επιτυγχάνουν ορθότητα, αλλά όχι απαραίτητα επίδοση, και συχνά αντικαθίστανται από reduction patterns.

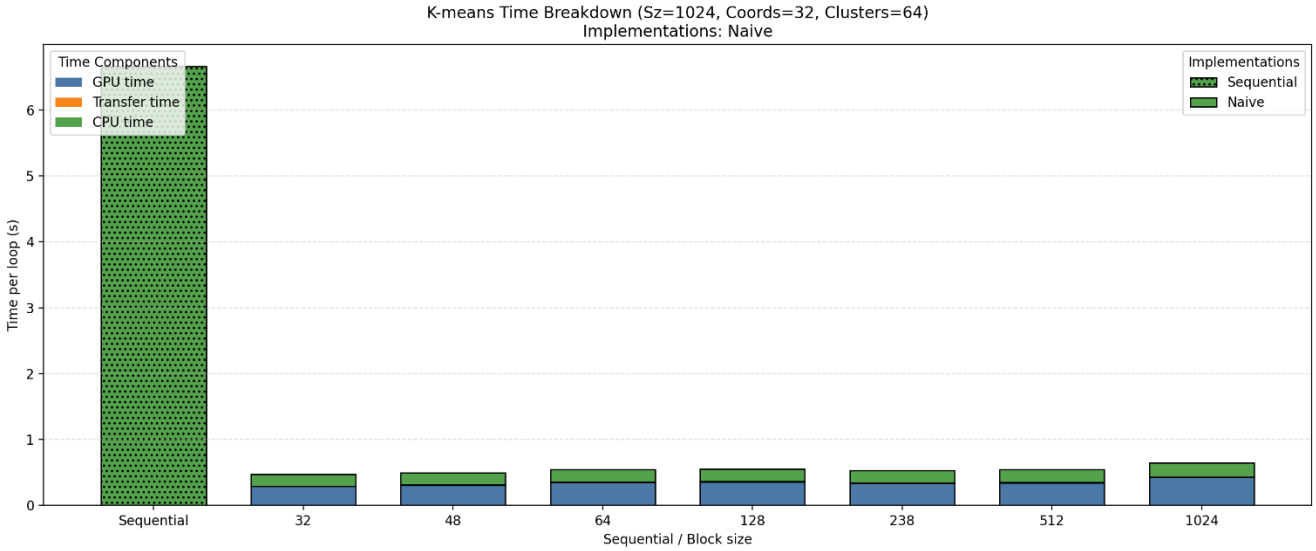
(δ) Timers (breakdown CPU / GPU / Transfers)

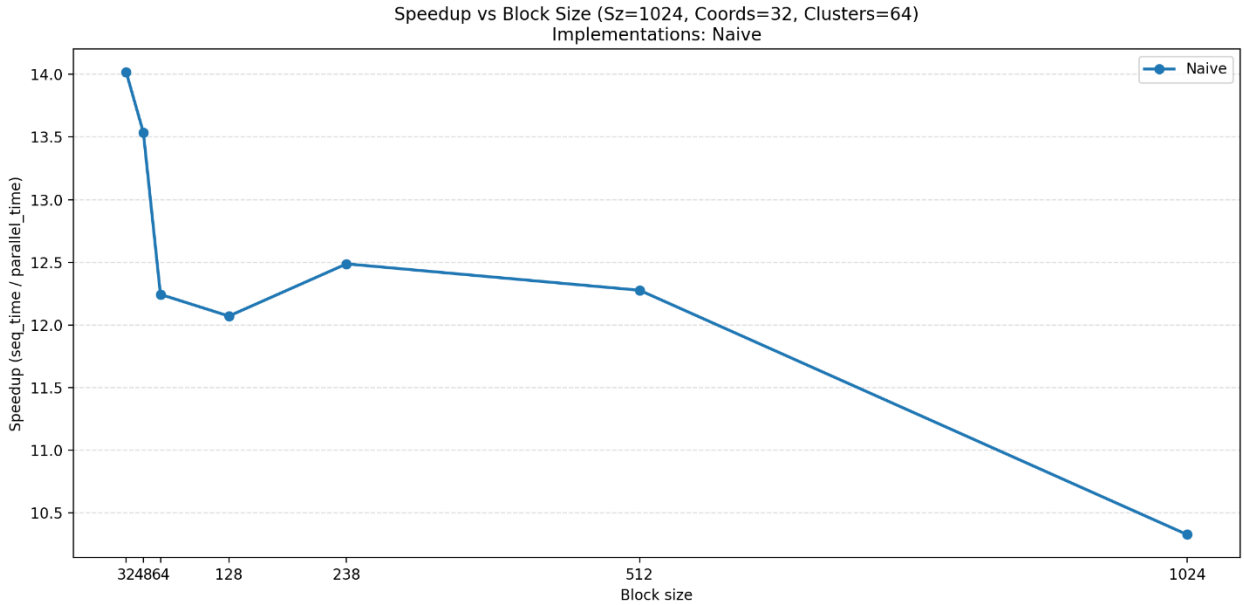
Στη naive έκδοση καταγράφονται τρεις συνιστώσες χρόνου ανά loop:

- GPU time: χρόνος εκτέλεσης του kernel,
- Transfers time: χρόνος αντιγραφών host↔device που γίνονται μέσα στο loop,
- CPU time: χρόνος update_centroids στην CPU.

Επισημαίνεται ότι το «μεγάλο» H→D copy του dataset (objects) γίνεται πριν την επανάληψη και μετριέται ξεχωριστά (άρα δεν εμφανίζεται στο per-loop transfers_avg).

Γ. Παρουσίαση Διαγραμμάτων





Δ. Ερμηνεία Διαγραμμάτων

(1) Speedup vs Block Size

Παρατηρείται μέγιστο speedup για μικρά block sizes (π.χ. 32–48) και σταδιακή υποβάθμιση για πολύ μεγάλα block sizes (έως 1024). Η συμπεριφορά αυτή είναι αναμενόμενη βάσει θεωρίας:

- Με μικρότερα blocks, ο scheduler μπορεί να διατηρεί περισσότερα resident blocks/warps ανά SM, αυξάνοντας την ικανότητα απόκρυψης latency (occupancy/latency hiding).
- Με πολύ μεγάλα blocks, μειώνεται ο αριθμός blocks που χωρούν ταυτόχρονα σε ένα SM (λόγω ορίων threads/SM ή πόρων όπως registers), άρα μειώνονται τα ενεργά warps και η GPU δυσκολεύεται να κρύψει memory latency. Επιπλέον, η naive πρόσβαση σε global μνήμη (objects/clusters) κάνει την επίδοση πιο ευαίσθητη σε occupancy.

(2) Time Breakdown

Το breakdown δείχνει ότι:

- Ο συνολικός χρόνος ανά loop της naive έκδοσης είναι πολύ μικρότερος από το sequential baseline, άρα επιτυγχάνεται σημαντικό speedup.
- Το GPU time είναι η κυρίαρχη συνιστώσα (όπως αναμενόταν, αφού το assignment είναι το κύριο υπολογιστικό μέρος).
- Τα transfer times φαίνονται μηδαμινά για Coords=32 και αυτό είναι λογικό: μέσα στο loop μεταφέρονται κυρίως (i) τα clusters (πολύ μικρά, ~KB) και (ii) το membership (μεγαλύτερο, αλλά όχι συγκρίσιμο με το 1GB dataset). Αντίθετα, η αρχική αντιγραφή του dataset προς τη GPU (1GB) γίνεται εκτός loop και δεν συμπεριλαμβάνεται στο transfers_avg του breakdown. Ωστόσο, το membership είναι O(N) ανά επανάληψη (Device→Host) και μπορεί να γίνει σημαντικό όταν το πλήθος objects N μεγαλώνει, ιδιαίτερα στο Coords=2 όπου για ίδιο Size προκύπτει πολύ μεγαλύτερο N. Άρα, το «μικρά transfers» ισχύει εδώ, για Coords=32, και όχι γενικά.

Ε. Συμπεράσματα

Η naive παραλληλοποίηση επιβεβαιώνει ότι το «assignment step» είναι κατάλληλο για GPU (data-parallel, ανεξάρτητος υπολογισμός ανά object), προσφέροντας υψηλό speedup. Ωστόσο, παραμένουν δύο εγγενή όρια:

- Επικοινωνία και CPU work ανά iteration (clusters/membership transfers + update_centroids στην CPU),
- Atomics για το delta (πιθανό contention).

Το K-means δεν είναι «ιδανικός» πυρήνας GPU ως συνολικός αλγόριθμος, αλλά περιέχει ένα τμήμα που είναι ιδιαίτερα κατάλληλο. Συγκεκριμένα, το βήμα ανάθεσης (assignment: για κάθε object υπολογισμός απόστασης από όλα τα clusters και επιλογή του ελάχιστου) είναι έντονα data-parallel, με ανεξάρτητη εργασία ανά object και μεγάλη παραλληλία, άρα ταιριάζει πολύ καλά στο SIMT μοντέλο των GPUs. Ωστόσο, η συνολική δομή του K-means είναι επαναληπτική και απαιτεί συγχρονισμό μεταξύ επαναλήψεων, ενώ το update των κέντρων είναι reduction/accumulation (sums & counts) και συχνά επιβαρύνεται από atomics και μη ευνοϊκές προσπελάσεις μνήμης. Στη naive υλοποίησή μας, επιπλέον, μέρος του κόστους παραμένει εκτός GPU (CPU update + μεταφορές membership/centroids ανά loop), άρα η επίδοση δεν εξαρτάται μόνο από το kernel αλλά και από επικοινωνία/overhead.

Αυτά αποτελούν και το κίνητρο για τις επόμενες εκδόσεις: βελτίωση προσπελάσεων global μνήμης (transpose/coalescing), επαναχρησιμοποίηση δεδομένων μέσω shared memory, και στη συνέχεια πλήρες offload (all-gpu) για μείωση CPU/transfer overhead.

▪ Ενότητα 3.2 – Transpose Version

Α. Εισαγωγή

Η έκδοση Transpose στοχεύει αποκλειστικά στη βελτιστοποίηση των προσπελάσεων της global μνήμης στην GPU. Στην naive έκδοση, τα threads ενός warp που επεξεργάζονται διαδοχικά objects προσπελαύνουν τα δεδομένα με «stride» ως προς τις συντεταγμένες (row-major layout), οδηγώντας σε μη coalesced accesses και αυξημένο αριθμό memory transactions. Η Transpose έκδοση αλλάζει τη διάταξη των δεδομένων σε column-based (transpose) μορφή, έτσι ώστε για κάθε συντεταγμένη i, τα 32 threads ενός warp να διαβάζουν συνεχόμενες διευθύνσεις μνήμης (coalescing), μειώνοντας δραστικά το κόστος πρόσβασης στη global memory και άρα τον χρόνο του kernel.

Ο κώδικας του αρχείου μας (cuda_kmeans_transpose.cu) παρατίθεται ακολούθως:

Β. Υλοποίηση και Ορθότητα

Η λογική του αλγορίθμου παραμένει ίδια: η GPU εκτελεί το assignment (membership) και η CPU εκτελεί το update_centroids. Η αλλαγή είναι καθαρά στη δομή δεδομένων:

- Αντί για objects[object][coord], δημιουργείται dimObjects[coord][object].
- Αντί για clusters[cluster][coord], δημιουργείται dimClusters[coord][cluster].

Επισημαίνουμε τα εξής σημεία:

(α) Νέα συνάρτηση απόστασης euclid_dist_2_transpose

Υπολογίζεται η ίδια Ευκλείδεια απόσταση, αλλά με indexing που ευνοεί coalescing:

objects[i*numObjs + objectId] και clusters[i*numClusters + clusterId].

Έτσι, για σταθερό i, τα threads του warp διαβάζουν συνεχόμενα objects (objectId διαδοχικά), άρα οι αναγνώσεις είναι coalesced.

(β) Μετασχηματισμός των δεδομένων (transpose) πριν την επανάληψη

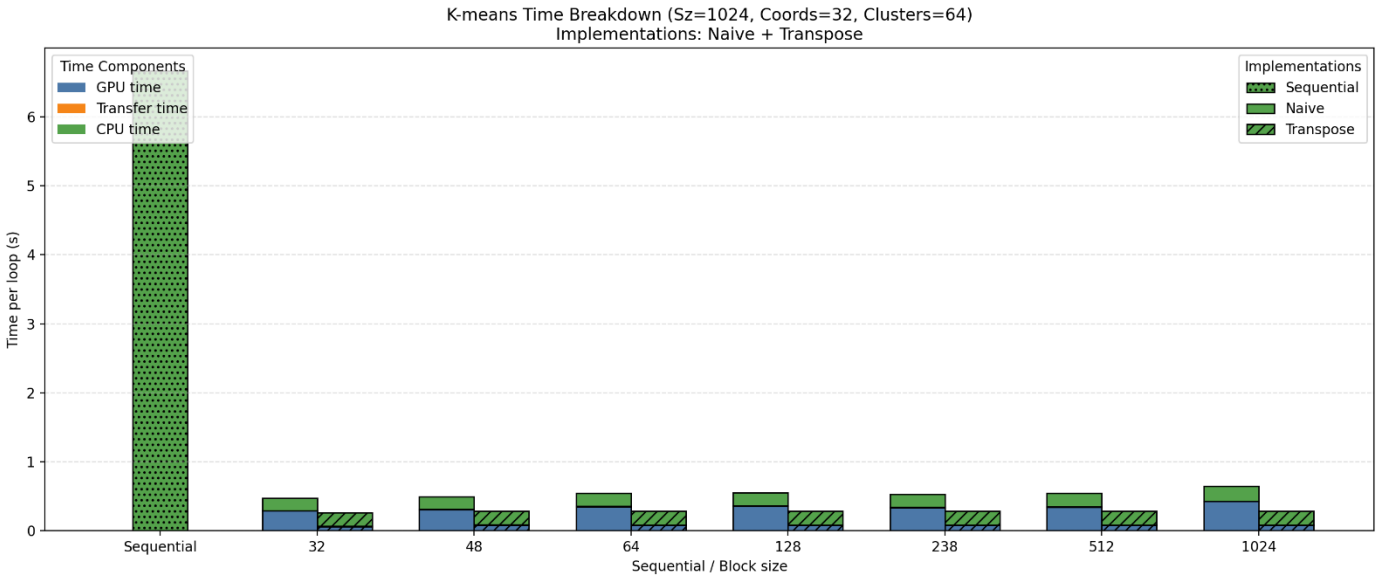
Πριν ξεκινήσει το loop, ο host κατασκευάζει τον dimObjects πίνακα από το αρχικό row-major objects. Αυτό είναι κόστος προεπεξεργασίας (εκτός loop) και δεν επηρεάζει το per-loop breakdown.

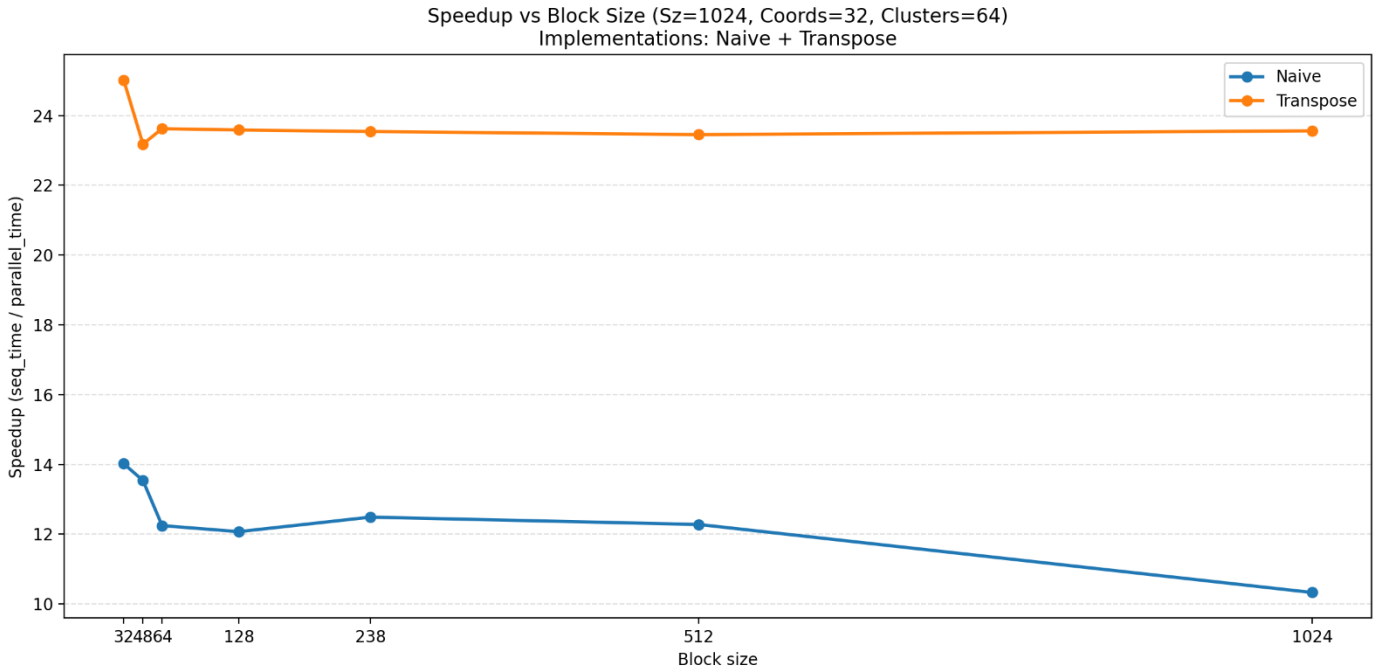
(γ) Ενημέρωση clusters με transpose μορφή

Στο CPU update_centroids, τα αθροίσματα/μέσοι όροι ενημερώνονται στη μορφή dimClusters[coord][cluster] ώστε το επόμενο H→D copy να διατηρεί τη coalesced διάταξη. Στο τέλος γίνεται back-transform σε clusters[cluster][coord] μόνο για λόγους συμβατότητας/εκτύπωσης.

Γενικά, η Transpose έκδοση είναι αριθμητικά ισοδύναμη με τη Naive (ίδια μετρική απόστασης, ίδια διαδικασία ανάθεσης/ενημέρωσης), αλλά με διαφορετική διάταξη στη μνήμη. Επομένως, αναμένουμε τα ίδια clusters (εντός floating-point διαφορών) και το ίδιο κριτήριο σύγκλισης· η διαφορά αφορά αποκλειστικά την επίδοση λόγω memory access pattern.

Γ. Παρουσίαση Διαγραμμάτων





Δ. Ερμηνεία Διαγραμμάτων

(1) Speedup vs Block Size

Η Transpose έκδοση παρουσιάζει σημαντικά υψηλότερο speedup από τη Naive (περίπου 23–25 έναντι ~10–14), και μάλιστα με σχετικά «επίπεδη» συμπεριφορά ως προς το block size. Αυτό είναι αναμενόμενο, καθώς:

- Με coalesced προσπελάσεις, μειώνονται τα global memory transactions ανά warp, άρα αυξάνεται το effective bandwidth και μειώνεται ο χρόνος kernel.
- Όταν το κύριο bottleneck είναι οι προσπελάσεις μνήμης, η βελτίωση στο memory access pattern έχει μεγαλύτερη επίδραση από μικρο-βελτιστοποιήσεις scheduling/occupancy μέσω block size, με αποτέλεσμα πιο σταθερή καμπύλη.

Στη Transpose έκδοση το block_size παίζει σαφώς μικρότερο ρόλο σε σχέση με τη Naive, όπως φαίνεται από τη σχεδόν επίπεδη καμπύλη speedup. Ο λόγος είναι ότι με το transpose πετυχαίνουμε coalesced προσπελάσεις στη global μνήμη (τα threads ενός warp διαβάζουν συνεχόμενες διευθύνσεις για κάθε συντεταγμένη), άρα μειώνεται δραστικά το κόστος memory transactions και το kernel γίνεται λιγότερο ευαίσθητο σε αλλαγές occupancy/scheduling που προκαλεί το block_size. Εφόσον το block_size είναι πολλαπλάσιο του 32 (warp size) και διατηρεί επαρκή ενεργά warps ανά SM, η απόδοση παραμένει σχεδόν σταθερή. Μόνο σε ακραία μεγέθη blocks ενδέχεται να εμφανιστεί μικρή πτώση (π.χ. λόγω μειωμένων resident blocks/warps ανά SM ή αυξημένων απαιτήσεων πόρων), αλλά συνολικά το κυρίαρχο κέρδος στη Transpose προέρχεται από το βελτιωμένο memory access pattern και όχι από την επιλογή block_size.

(2) Time Breakdown (GPU / Transfers / CPU)

Το breakdown δείχνει ότι η κύρια μείωση χρόνου προέρχεται από το GPU time (kernel). Αυτό είναι ακριβώς το αναμενόμενο αποτέλεσμα της βελτιστοποίησης coalescing: δεν αλλάζουμε τις μεταφορές ανά loop ούτε το CPU update_centroids, αλλά μειώνουμε δραστικά τον χρόνο της φάσης assignment στη GPU.

Τα transfer times παραμένουν χαμηλά στο συγκεκριμένο σενάριο (Coords=32), διότι εντός loop μεταφέρεται:

- Host→Device: clusters (μικρό μέγεθος),
- Device→Host: membership + delta.

Η αρχική αντιγραφή των objects (1GB) γίνεται εκτός loop και δεν περιλαμβάνεται στο per-loop transfer χρόνο.

Ε. Σύγκριση Αποτελεσμάτων (με Naive)

1. Κύριο εύρημα: Η Transpose έκδοση επιτυγχάνει ~1.7×–2× καλύτερο speedup από τη Naive, παρότι ο αλγόριθμος παραμένει ο ίδιος.
2. Η βελτίωση δεν οφείλεται σε περισσότερους υπολογισμούς στη GPU, αλλά σε καθαρά αρχιτεκτονικό λόγο: καλύτερη αξιοποίηση του memory subsystem μέσω coalescing (32-thread warps → συνεχόμενες διευθύνσεις → λιγότερα transactions).
3. Η συνιστώσα CPU (update_centroids) παραμένει πρακτικά η ίδια, άρα το συνολικό κέρδος έρχεται από τη μείωση του kernel time.
4. Η εξάρτηση από block size είναι μικρότερη σε σχέση με τη Naive, επειδή η Transpose μειώνει το memory overhead και σταθεροποιεί την απόδοση.

ΣΤ. Συμπεράσματα

Η Transpose έκδοση επιβεβαιώνει ότι η διάταξη των δεδομένων στη μνήμη μπορεί να είναι καθοριστική για την απόδοση σε GPU. Με την αναδιάταξη σε column-based μορφή πετυχαίνουμε coalesced global memory accesses κατά τον υπολογισμό αποστάσεων, μειώνοντας αισθητά τον χρόνο του kernel και αυξάνοντας το speedup. Αυτό αποτελεί το φυσικό επόμενο βήμα μετά τη Naive προσέγγιση και δημιουργεί τη βάση για την επόμενη βελτιστοποίηση (Shared), όπου στοχεύουμε επιπλέον στη μείωση των επαναλαμβανόμενων αναγνώσεων clusters μέσω shared memory (on-chip reuse).

- **Ενότητα 3.3 – Shared Version**

Α. Εισαγωγή

Η έκδοση Shared επεκτείνει την βελτιστοποίηση που εισαγάγαμε στην Transpose και στοχεύει στη μείωση των επαναλαμβανόμενων αναγνώσεων των cluster centers από την global μνήμη. Στο K-means, για κάθε object υπολογίζονται αποστάσεις από όλα τα clusters. Άρα, τα ίδια cluster centers επαναχρησιμοποιούνται πολλές φορές από τα threads ενός block. Με τη φόρτωσή τους στη shared memory (on-chip), μειώνουμε σημαντικά το global memory traffic και επιταχύνουμε το assignment kernel.

Ο κώδικας του αρχείου μας (cuda_kmeans_shared.cu) παρατίθεται ακολούθως:

Β. Υλοποίηση και Ορθότητα

Η δομή δεδομένων παραμένει transpose (dimObjects[coord][obj], dimClusters[coord][cluster]) για coalescing. Η βασική αλλαγή είναι ότι στον kernel:

- Τα cluster centers αντιγράφονται μια φορά ανά block από global σε shared memory.
- Όλοι οι υπολογισμοί απόστασης χρησιμοποιούν πλέον τη shared μνήμη για τα clusters.

Επισημαίνουμε τα εξής σημεία:

(α) Δυναμική shared memory και αντιγραφή clusters

Χρησιμοποιείται extern __shared__ double shmemClusters[] και αντιγράφεται ολόκληρος ο πίνακας dimClusters (numCoords*numClusters στοιχεία) στη shared memory, με «striding» ως προς threadIdx (k += blockDim.x). Έτσι η φόρτωση μοιράζεται σε threads και γίνεται μία φορά ανά block.

(β) Συγχρονισμός (__syncthreads)

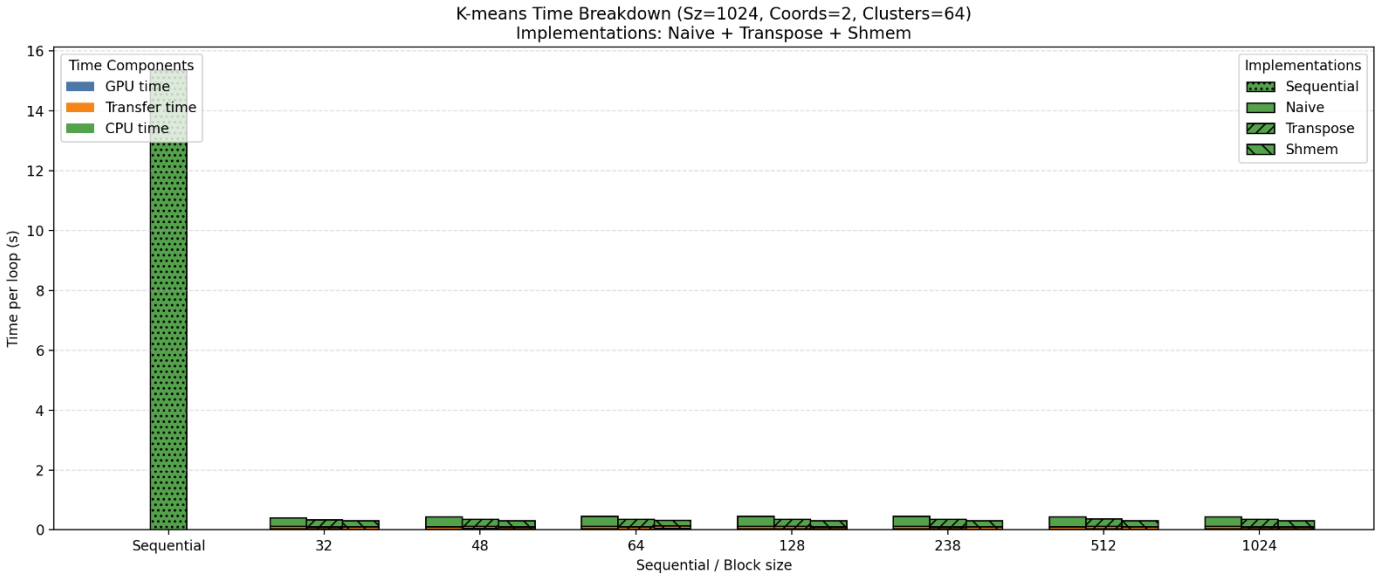
Μετά τη φόρτωση στη shared, γίνεται __syncthreads() ώστε να εξασφαλιστεί ότι όλα τα threads του block βλέπουν πλήρως γραμμένα τα δεδομένα πριν ξεκινήσουν τους υπολογισμούς αποστάσεων. Αυτό είναι απαραίτητο για ορθότητα (διαφορετικά κάποια threads θα διάβαζαν μη αρχικοποιημένες τιμές).

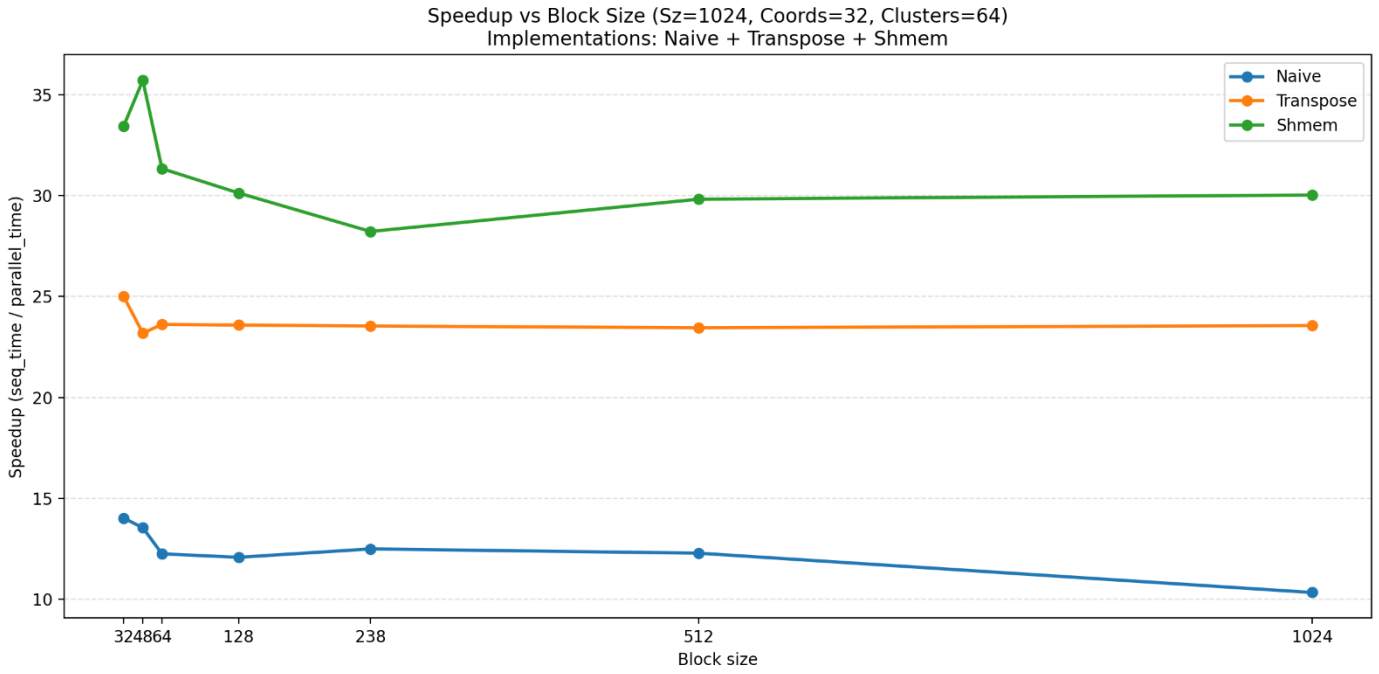
(γ) Έλεγχος διαθέσιμης shared μνήμης ανά block

Στον host υπολογίζεται το απαιτούμενο shared size = numClusters * numCoords * sizeof(double) και ελέγχεται έναντι της ιδιότητας sharedMemPerBlock της συσκευής. Αν το όριο ξεπεραστεί, η έκδοση δεν μπορεί να τρέξει (σωστό safeguard, καθώς το shared memory είναι περιορισμένος πόρος).

Η έκδοση Shared είναι αριθμητικά ισοδύναμη με Transpose/Naive: δεν αλλάζει ο ορισμός απόστασης ούτε το κριτήριο ανάθεσης. Αλλάζει μόνο η τοποθέτηση των cluster centers (shared αντί global), άρα αναμένουμε ίδια αποτελέσματα σύγκλισης (εντός floating-point διαφορών), με χαμηλότερο χρόνο kernel.

Γ. Παρουσίαση Διαγραμμάτων





Δ. Ερμηνεία Διαγραμμάτων

(1) Speedup vs Block Size

Η Shared υπερέχει σαφώς, με speedup περίπου 28–36, έναντι ~23–25 της Transpose και ~10–14 της Naive. Η βελτίωση είναι αναμενόμενη: ενώ η Transpose μειώνει τα global transactions μέσω coalescing, η Shared μειώνει και το πλήθος των global loads για τα clusters, αφού κάθε block φέρνει τα clusters μία φορά και τα επαναχρησιμοποιεί σε όλους τους distance υπολογισμούς.

(2) Time Breakdown (GPU / Transfers / CPU)

Το breakdown δείχνει ότι το κύριο κέρδος της Shared προέρχεται από περαιτέρω μείωση του GPU time (kernel). Οι μεταφορές (clusters H→D, membership+delta D→H) και ο CPU χρόνος (update_centroids) παραμένουν ουσιαστικά παρόμοια με Transpose/Naive, άρα η επιτάχυνση οφείλεται σχεδόν αποκλειστικά στη βελτίωση του memory access/reuse εντός του kernel.

Ρόλος του block_size στη Shared

Σε αντίθεση με τη Transpose (όπου η εξάρτηση από block_size ήταν μικρή), στη Shared το block_size μπορεί να επηρεάζει περισσότερο την απόδοση, επειδή η shared memory εισάγει πρόσθετους περιορισμούς στους resident πόρους ανά SM:

- Κάθε block δεσμεύει σταθερό shared size (εδώ: numClusters*numCoords*sizeof(double)), άρα ο μέγιστος αριθμός blocks/SM μπορεί να περιοριστεί από τη διαθέσιμη shared μνήμη, μειώνοντας occupancy (active warps/SM).
- Με πολύ μεγάλα blocks, περιοριζόμαστε επιπλέον από το όριο threads/SM, άρα μπορεί να μειωθούν ταυτόχρονα resident blocks και warps, και να αυξηθεί η ευαισθησία σε latency.

Συνεπώς, παρατηρείται συνήθως ένα ιδανικό σημείο (small block sizes) όπου συνδυάζονται αρκετά active warps και χαμηλό global traffic, ενώ σε ακραία μεγέθη blocks η απόδοση μπορεί να σταθεροποιείται ή να πέφτει.

Ε. Σύγκριση Αποτελεσμάτων (με Naive + Transpose)

1. Από Naive → Transpose: μεγάλο κέρδος λόγω coalescing (μείωση global memory transactions).
2. Από Transpose → Shared: επιπλέον μεγάλο κέρδος, διότι μειώνουμε τις επαναλαμβανόμενες αναγνώσεις clusters από global (on-chip reuse). Το κέρδος εμφανίζεται κυρίως ως περαιτέρω μείωση του GPU time, ενώ CPU και transfers παραμένουν περίπου σταθερά.
3. Η Shared εμφανίζει μεγαλύτερη (αλλά λογική) εξάρτηση από block_size σε σχέση με Transpose, λόγω των πόρων shared memory/occupancy.

ΣΤ. Συμπεράσματα

Η Shared έκδοση επιβεβαιώνει τη βασική αρχή βελτιστοποίησης GPU: πέρα από το coalescing, η επαναχρησιμοποίηση «hot» δεδομένων στη shared memory μπορεί να μειώσει δραστικά το global memory traffic και να επιταχύνει σημαντικά memory-bound kernels όπως το assignment του K-means. Για το συγκεκριμένο σενάριο (Coords=32, Clusters=64), η Shared είναι η καλύτερη από τις τρεις εκδόσεις, με το κέρδος να προέρχεται κυρίως από τη μείωση του χρόνου kernel και δευτερευόντως από επιλογές block_size που επηρεάζουν occupancy.

▪ Ενότητα 3.4 – Σύγκριση Υλοποιήσεων/Bottleneck Analysis

Α. Εισαγωγή

Στο σημείο αυτό συγκρίνουμε τις τρεις υλοποιήσεις (Naive, Transpose, Shared) και εντοπίζουμε το bottleneck χρησιμοποιώντας τα δεδομένα χρόνου ανά επανάληψη (GPU kernel / transfers CPU↔GPU / CPU update). Υπενθυμίζουμε ότι τα transfers που μετράμε εδώ αφορούν τις αντιγραφές μέσα στο loop (π.χ. clusters H→D και membership+delta D→H) και όχι την αρχική μεταφορά του dataset προς τη GPU, η οποία γίνεται μία φορά πριν ξεκινήσουν οι επαναλήψεις.

Β. Σύγκριση Υλοποιήσεων/Bottleneck Analysis

1. Ποιο bottleneck περιορίζει την επίδοση (Sz=1024MB, Coords=32, Clusters=64);

Από τα διαγράμματα για Coords=32, το αποτέλεσμα είναι ξεκάθαρο:

- Naive → Transpose: η κύρια μείωση χρόνου έρχεται από το GPU time, επειδή το transpose βελτιώνει το memory coalescing (λιγότερα global memory transactions ανά warp).
- Transpose → Shared: το GPU time μειώνεται περαιτέρω, επειδή τα cluster centers επαναχρησιμοποιούνται από shared memory (on-chip) αντί για επαναλαμβανόμενες αναγνώσεις από global.

Μετά τις δύο αυτές βελτιστοποιήσεις, όμως, παρατηρείται ότι το συνολικό κέρδος δεν αυξάνεται αναλογικά (ιδανικά, δηλαδή δεν κλιμακώνει τέλεια) με τη μείωση του GPU time. Ο λόγος είναι ότι πλέον αρχίζουν να κυριαρχούν/να γίνονται συγκρίσιμα τα μη-επιταχυνόμενα τμήματα:

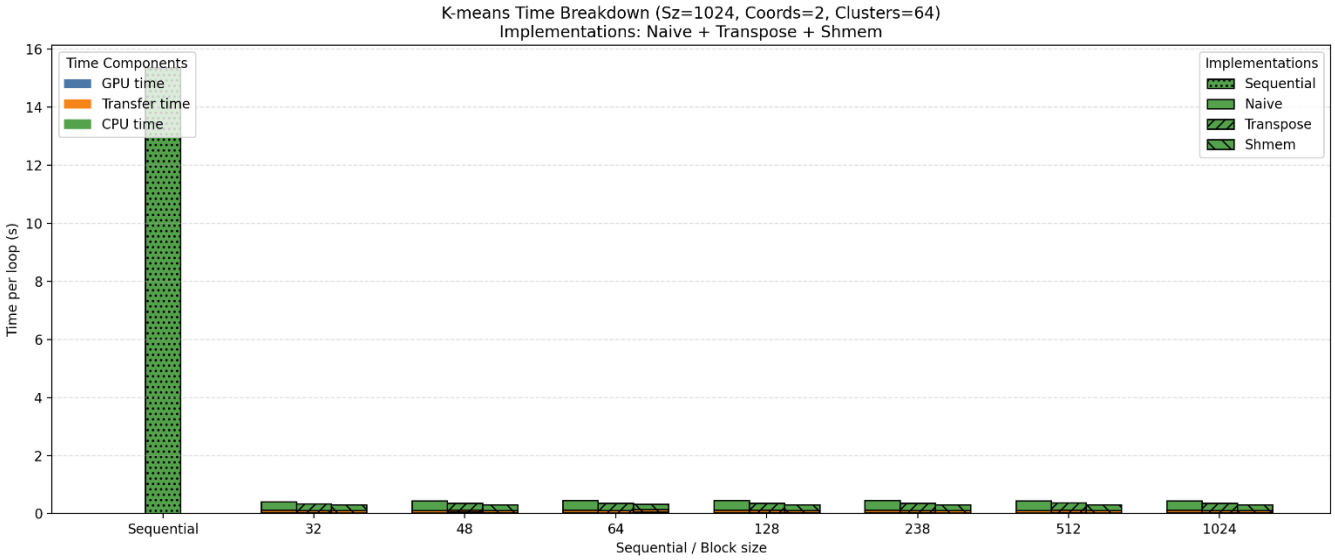
- CPU time (update_centroids): παραμένει σειριακό στην CPU στις τρεις εκδόσεις και θέτει όριο βάσει του Amdahl (όσο μικραίνει το GPU kernel, τόσο μεγαλύτερο ποσοστό του συνολικού χρόνου καταλαμβάνει το CPU update).

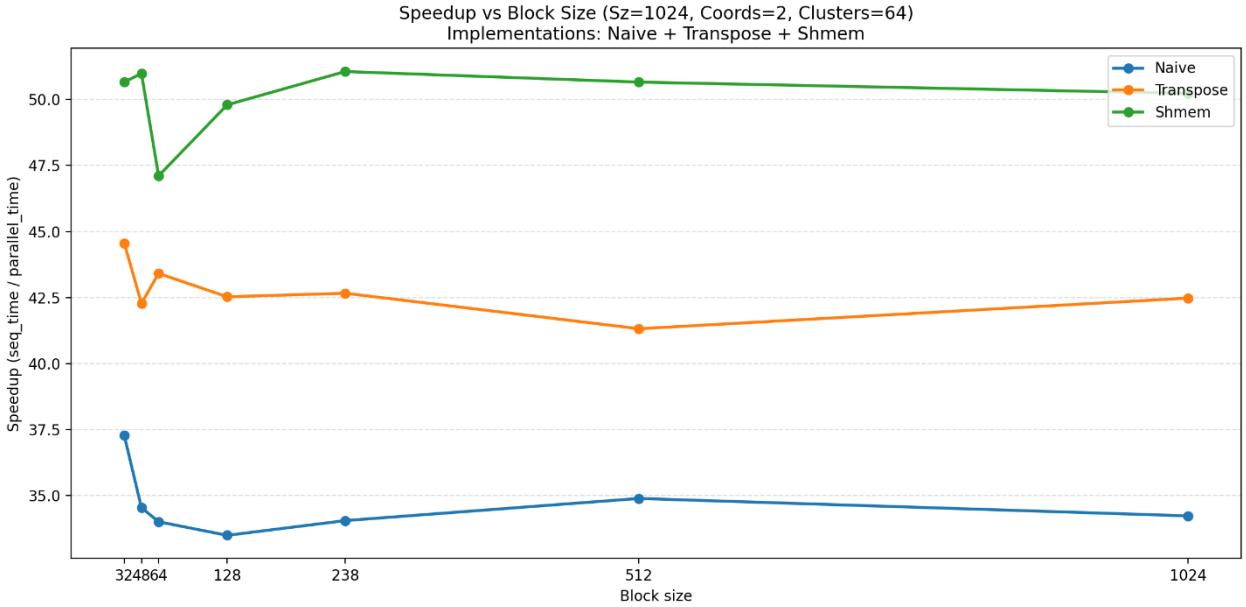
- Transfers time: για Coords=32 είναι σχετικά μικρό, αλλά είναι σταθερό overhead ανά επανάληψη (ιδίως η επιστροφή του membership), το οποίο δεν μειώνεται από τις βελτιστοποιήσεις μέσα στον kernel.

Συνεπώς, για Coords=32 το bottleneck «μετατοπίζεται» σταδιακά: αρχικά είναι κυρίως η απόδοση του GPU kernel (Naive), ενώ στις βελτιστοποιημένες εκδόσεις (Transpose/Shared) το όριο τίθεται ολοένα περισσότερο από το CPU update και το σταθερό κόστος επικοινωνίας ανά loop.

2. Τι αλλάζει για Coords=2 και είναι η προσέγγιση Shared κατάλληλη για arbitrary configs;

Αρχικά, παρουσιάζουμε τα διαγράμματα με 2 συντεταγμένες ακολούθως:





Με σταθερό dataset size (1024MB), όταν μειώνουμε τις διαστάσεις από 32 σε 2, ο αριθμός objects αυξάνεται περίπου κατά 16× (numObjs ∝ 1/numCoords). Αυτό έχει δύο κρίσιμες συνέπειες:

1. Αυξάνεται δραστικά το μέγεθος του membership που πρέπει να επιστρέφει στη CPU σε κάθε επανάληψη (D→H), άρα ο χρόνος transfers γίνεται πολύ πιο σημαντικός σε σχέση με το Coords=32.
2. Ταυτόχρονα, ο υπολογισμός απόστασης ανά object γίνεται ελαφρύτερος (μόνο 2 συντεταγμένες), άρα το GPU kernel έχει μικρότερο arithmetic work ανά element και η συνολική εκτέλεση τείνει να γίνεται λιγότερο compute-bound και πιο overhead/communication sensitive.

Τα διαγράμματα για Coords=2 επιβεβαιώνουν αυτή τη μετατόπιση: ενώ η Shared παραμένει η ταχύτερη υλοποίηση (Shared > Transpose > Naive), η διαφορά μεταξύ των GPU εκδόσεων προκύπτει πλέον κυρίως από σχετικά μικρότερες βελτιώσεις στο GPU time, επειδή ένα μεγαλύτερο ποσοστό του συνολικού χρόνου ανά loop ανήκει στις μεταφορές (και στο CPU update). Με άλλα λόγια, όταν το bottleneck είναι η επικοινωνία (membership transfer) και το σειριακό update, οι βελτιστοποιήσεις εντός του kernel έχουν περιορισμένο χώρο να αποδώσουν.

Γ. Συμπέρασμα για arbitrary configs

Η τεχνική shared memory για τα clusters είναι γενικά αποδοτική όταν:

- το (numClusters × numCoords) χωράει σε shared ανά block, και
- υπάρχει αρκετή επαναχρησιμοποίηση/υπολογιστικό έργο ανά φόρτωση (ώστε το κόστος φόρτωσης + __syncthreads να αποσβεστεί).

Ωστόσο, δεν είναι καθολική αλήθεια για όλα τα configs: σε περιπτώσεις όπως Coords=2, όπου αυξάνεται έντονα το communication overhead (membership D→H) και μειώνεται το arithmetic intensity του distance computation, το συνολικό bottleneck μετακινείται εκτός kernel. Τότε η Shared εξακολουθεί να βοηθά (μειώνει το GPU time), αλλά το συνολικό speedup περιορίζεται κυρίως από transfers και CPU update, δηλαδή από τμήματα που η Shared δεν μπορεί να βελτιώσει.

- **Full-Offload (All-GPU) Version**

A. Εισαγωγή

Στην έκδοση Full-Offload (All-GPU) μεταφέρουμε ολόκληρο το iterative μέρος του K-means στη GPU: όχι μόνο το assignment (εύρεση κοντινότερου cluster για κάθε object), αλλά και το update των centroids (συσσώρευση sums/counts και υπολογισμός νέων κέντρων). Στόχος είναι να εξαλειφθούν (i) το CPU load ανά επανάληψη (update_centroids στην CPU) και (ii) οι μεγάλες μεταφορές CPU↔GPU μέσα στο loop (ιδίως το D2H membership), ώστε το bottleneck να περιοριστεί στον καθαρό GPU υπολογισμό.

Ο κώδικας του αρχείου μας (cuda_kmeans_all_gpu.cu) παρατίθεται ακολούθως:

Β. Υλοποίηση και Ορθότητα

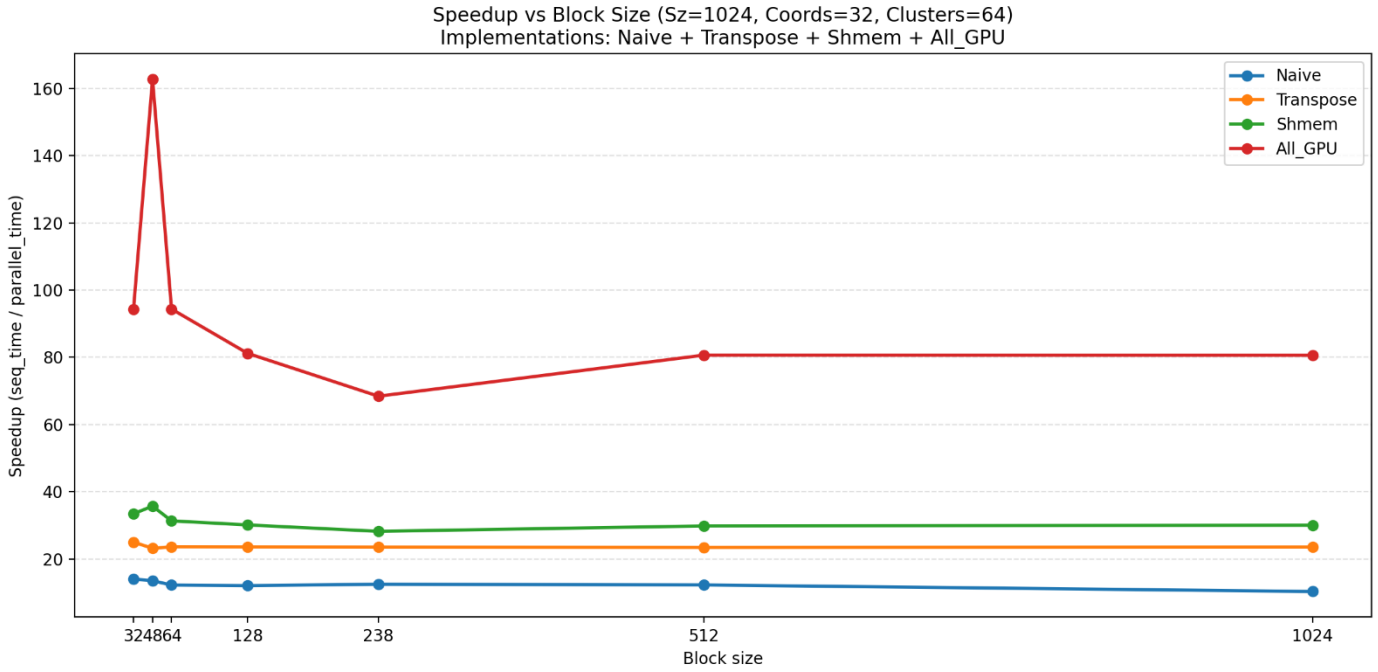
Η λογική της All-GPU υλοποίησης είναι να σπάσει το update_centroids σε βήματα που μπορούν να γίνουν ασφαλώς στη GPU χωρίς καθολικό barrier μέσα σε ένα kernel:

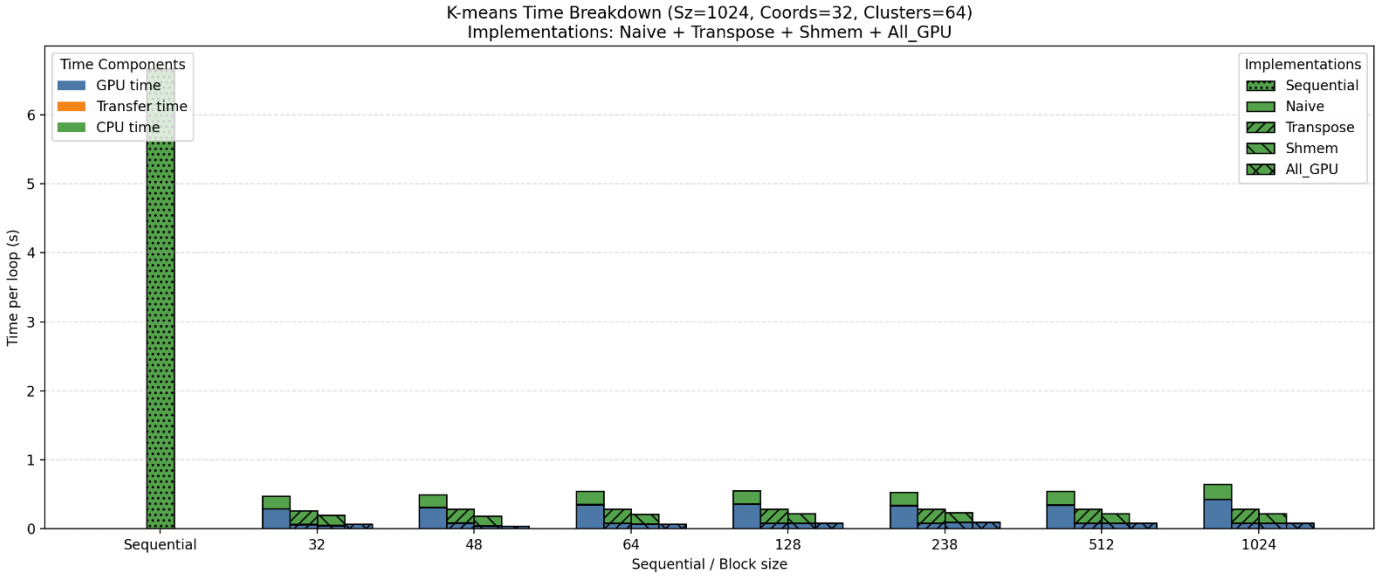
1. Μηδενισμός/αρχικοποίηση device arrays για newClusters_sums και newClusters_counts (και ό,τι άλλο χρειάζεται).
2. Kernel ανάθεσης (find_nearest_cluster): κάθε thread επεξεργάζεται ένα object, υπολογίζει αποστάσεις προς όλα τα clusters, ενημερώνει το membership και ταυτόχρονα συσσωρεύει τη συνεισφορά του object στο cluster που ανήκει.
    - Η συσσώρευση sums/counts γίνεται με atomics σε global μνήμη (atomicAdd σε counts και σε κάθε διάσταση του sum), ώστε να αποφευχθούν race conditions.
    - Τα cluster centers μπορούν να φορτωθούν ανά block στη shared memory (όπως στη shared έκδοση) ώστε οι επαναλαμβανόμενες αναγνώσεις κατά τον υπολογισμό αποστάσεων να γίνονται από on-chip μνήμη.
3. Kernel τελικοποίησης centroids: για κάθε cluster (και διάσταση) υπολογίζεται ο μέσος όρος (sum/count) και παράγονται τα νέα centers για το επόμενο iteration.
4. Για τον τερματισμό του while-loop, στον host επιστρέφει μόνο το delta (ή/και ελάχιστη μετα-πληροφορία). Έτσι, οι μεταφορές μέσα στο loop ελαχιστοποιούνται.

Η χρήση atomics εξασφαλίζει το σωστό αποτέλεσμα στα αθροίσματα, παρά το ταυτόχρονο update από πολλά threads. Ο απαιτούμενος καθολικός συγχρονισμός επιτυγχάνεται φυσικά με τη διάσπαση σε πολλαπλά kernels (τα kernels εκτελούνται σειριακά ως προς τη σειρά κλήσης τους), κάτι που αντικαθιστά την απουσία global barrier μέσα σε έναν kernel.

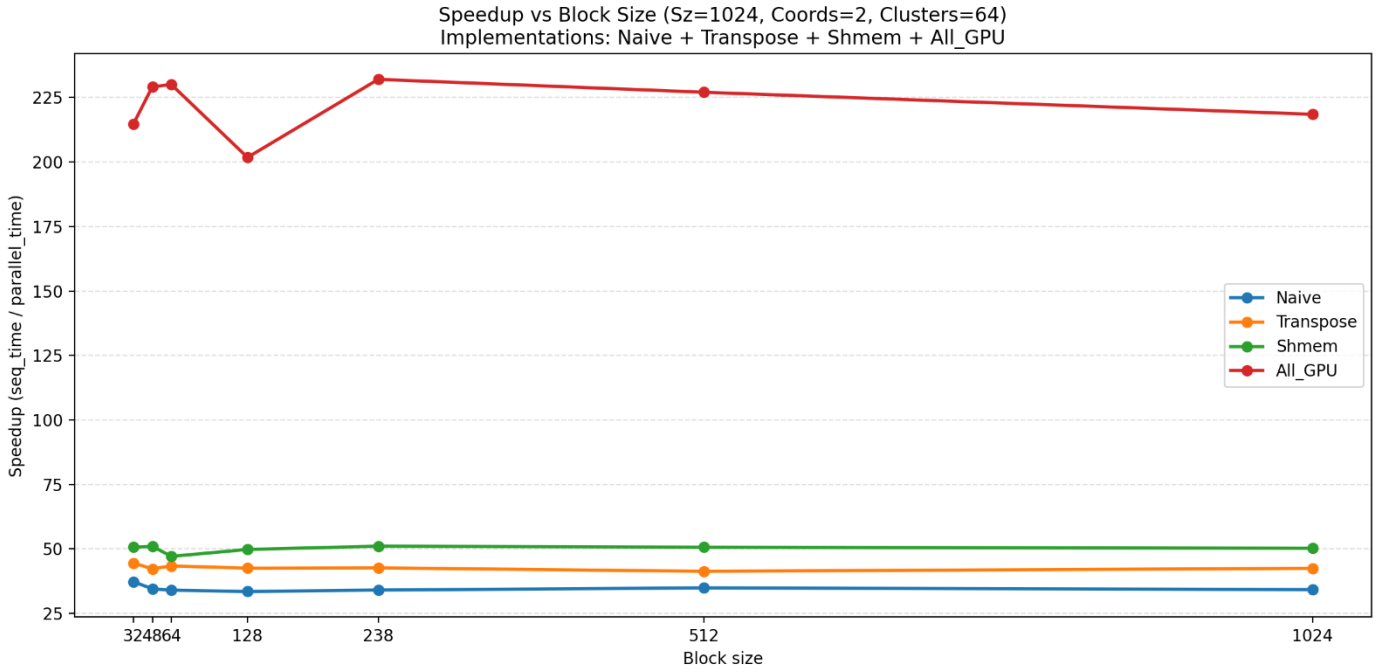
Γ. Παρουσίαση Διαγραμμάτων

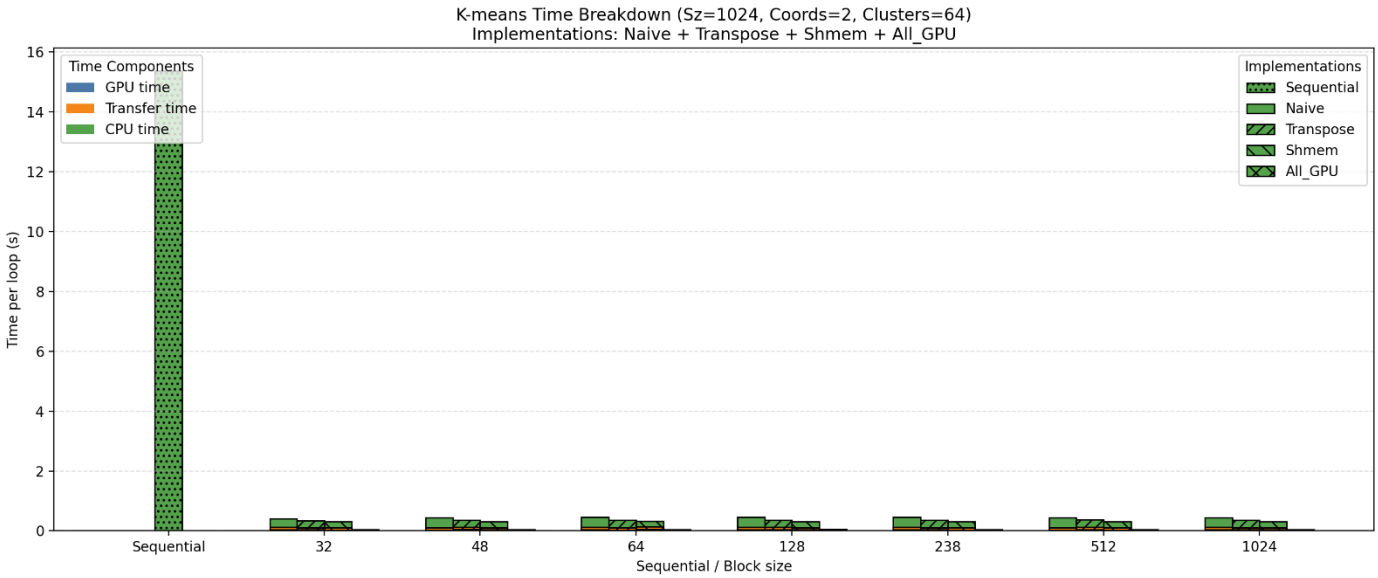
(α) Configuration {1024,32,64,10}





(β) Configuration {1024,2,64,10}





Δ. Ερμηνεία Διαγραμμάτων – Απάντηση Ερωτημάτων (1) και (2)

(1) Επίδοση All-GPU σε σχέση με naive/transpose/shared (και για τα δύο configurations)

Τα διαγράμματα δείχνουν ότι η All-GPU υπερέχει σημαντικά έναντι όλων των προηγούμενων εκδόσεων και στα δύο configurations. Αυτό είναι αναμενόμενο, καθώς:

- Το CPU time μέσα στο loop (update_centroids στην CPU) πρακτικά μηδενίζεται.
- Το transfer time μέσα στο loop μειώνεται δραστικά, επειδή δεν απαιτείται πλέον αντιγραφή του membership πίσω στην CPU σε κάθε επανάληψη. Στον host επιστρέφει μόνο το delta, άρα οι per-iteration μεταφορές περιορίζονται σε πολύ μικρά δεδομένα (σε αντίθεση με τις naive/transpose/shared όπου υπάρχει O(N) Device→Host membership).

Άρα, το iterative μέρος παύει να είναι υβριδικό (GPU assignment + CPU update + transfers) και γίνεται σχεδόν αποκλειστικά GPU workload, το οποίο ταιριάζει καλύτερα στη φιλοσοφία throughput της GPU. Το νέο bottleneck μεταφέρεται κυρίως στον GPU χρόνο (distance computations + atomics για sums/counts).

Σημείωση: Η εκτέλεση έχει σχετικά μικρό warp divergence (κυρίως bounds checks και η απλή ενημέρωση membership), άρα το bottleneck προέρχεται κυρίως από global memory traffic και atomic contention, όχι από branching

(2) Παίζει διαφορετικό ρόλο το block_size και γιατί;

Ναι, στην All-GPU έκδοση το block_size επηρεάζει έντονα την επίδοση και αυτό φαίνεται καθαρά στα διαγράμματα (ιδίως στο Coords=32, όπου υπάρχει πολύ μεγάλη διακύμανση speedup ανά block size). Ο λόγος είναι ότι, αφού σχεδόν μηδενίζονται τα per-loop transfers και το CPU update, το συνολικό runtime καθορίζεται σχεδόν αποκλειστικά από καθαρά GPU φαινόμενα, τα οποία εξαρτώνται άμεσα από το block_size:

1. Occupancy / latency hiding: Το block_size καθορίζει πόσα blocks/warps μπορούν να είναι resident ανά SM. Με μεγαλύτερα blocks αυξάνονται οι απαιτήσεις σε threads/SM (και σε registers ανά block), άρα συχνά μειώνονται

τα ταυτόχρονα resident blocks/warps. Όταν μειωθούν τα active warps, η GPU κρύβει χειρότερα τη latency της global μνήμης και η επίδοση πέφτει.

2. Πίεση σε registers και shared: Στον assignment kernel κάθε thread κάνει σχετικά βαριά δουλειά (loop σε numClusters και numCoords). Αυτό τείνει να αυξάνει τα registers/thread. Όσο μεγαλώνει το block_size, το συνολικό register footprint/block μεγαλώνει και μπορεί να περιορίσει τα blocks/SM. Αν χρησιμοποιείται και shared caching για τα clusters, το shared ανά block είναι σταθερό, αλλά σε συνδυασμό με τα registers/threads μπορεί να κλειδώσει το occupancy.
3. Atomic contention στο update_centroids: Η All-GPU κάνει συσσώρευση sums/counts με atomics. Το block_size επηρεάζει πόσα threads πηγαίνουν ταυτόχρονα τους ίδιους counters/αθροίσματα (ιδίως όταν πολλά objects καταλήγουν στα ίδια clusters). Μεγαλύτερη ταυτόχρονη πίεση σε atomics οδηγεί σε serialization και απώλεια throughput, άρα μπορεί να εμφανίζεται ισχυρό sweet spot σε συγκεκριμένα block sizes. Θεωρητικά, για να μειωθεί το contention, μια κλασική τεχνική είναι block-level partial sums/counts σε shared memory (με reduction) και στη συνέχεια ένα μόνο atomicAdd ανά (block, cluster, coord) προς global μνήμη
4. Warp efficiency / μη ιδανικά block sizes: Επειδή η εκτέλεση γίνεται σε warps των 32 threads, block sizes που δεν είναι πολλαπλάσια του 32 δημιουργούν μερικώς γεμάτα warps (wasted lanes). Αυτό μπορεί να επιδεινώσει την αποδοτικότητα και να αλλάξει το ισοζύγιο occupancy–contention.

Συμπέρασμα: Σε All-GPU, το block_size δεν είναι δευτερεύον όπως μπορεί να φαινόταν σε Transpose-only σενάρια. Αντίθετα καθορίζει άμεσα το occupancy και το atomic contention (και άρα τον GPU χρόνο), οπότε εμφανίζονται έντονα βέλτιστα σημεία και απότομες μεταβολές στην επίδοση, ειδικά στο Coords=32 όπου αυξάνεται το έργο/νήμα και το πλήθος atomicAdds ανά object. Συνολικά, το block_size καθορίζει ένα trade-off ανάμεσα σε occupancy/latency hiding και σε contention/πόρους (registers/shared), οπότε εμφανίζεται φυσιολογικά sweet spot.

Ε. Είναι το update_centroids κατάλληλο για GPUs; Και γιατί η All-GPU διαφέρει τόσο σε επίδοση;

Το update_centroids δεν είναι ιδανικό GPU kernel με την έννοια του τέλειου, ανεξάρτητου per-thread υπολογισμού: απαιτεί συνάθροιση (reduction) πολλών contributions σε κοινά arrays (sums/counts), άρα:

- Εισάγει συγχρονισμό μέσω atomics και contention (πολλά threads ενημερώνουν τα ίδια clusters), που μπορεί να περιορίσει το scaling.
- Περιλαμβάνει στάδια που απαιτούν καθολικό συγχρονισμό (π.χ. πρώτα να ολοκληρωθούν όλα τα sums/counts πριν γίνει η διαίρεση για τα νέα centroids), κάτι που μας αναγκάζει να το σπάσουμε σε πολλαπλά kernels.

Παρόλα αυτά, η All-GPU είναι πολύ ταχύτερη συνολικά, επειδή αφαιρεί τα προηγούμενα dominant bottlenecks:

- Δεν πληρώνουμε πλέον CPU χρόνο ανά iteration για update_centroids.
- Δεν πληρώνουμε πλέον μεγάλο D2H transfer του membership ανά iteration (ούτε το H2D των clusters σε κάθε γύρο).

Άρα, ακόμη κι αν το update_centroids στη GPU “δεν είναι τέλειο” και έχει atomic overhead, το συνολικό κέρδος από την εξάλειψη CPU+PCIe κόστους είναι πολύ μεγαλύτερο, με αποτέλεσμα την εντυπωσιακή αύξηση speedup έναντι naive/transpose/shared.

ΣΤ. Τι διαφέρει μεταξύ των δύο configurations και πώς αιτιολογείται η διαφορά επίδοσης;

Το κρίσιμο σημείο είναι ότι το “Size=1024” αντιστοιχεί σε σταθερό συνολικό μέγεθος dataset, άρα αλλάζει ο αριθμός των objects όταν αλλάζει το Coords:

- Με Coords=2, κάθε object έχει πολύ λιγότερα bytes → έχουμε πολύ περισσότερα objects.
- Με Coords=32, κάθε object είναι “βαρύτερο” → έχουμε πολύ λιγότερα objects.

Αυτό επηρεάζει και τη σειριακή και την παράλληλη εκτέλεση, αλλά και το είδος bottleneck:

- Coords=2: τεράστιος αριθμός objects → πολύ υψηλός παραλληλισμός (η GPU γεμίζει εύκολα), και στις παλιές εκδόσεις υπήρχαν πολύ μεγάλα per-iteration transfers (membership), τα οποία η All-GPU εξαφανίζει. Έτσι βλέπουμε πολύ υψηλό speedup.
- Coords=32: λιγότερα objects → λιγότερος παραλληλισμός και μεγαλύτερη σημασία στα σταθερά/overhead κόστη (kernel launches, reset/finalize kernels). Επιπλέον, στην All-GPU αυξάνεται η δουλειά ανά object (περισσότερες διαστάσεις σε distance + περισσότερα atomicAdds ανά object για sums), οπότε ο GPU χρόνος ανεβαίνει και το speedup περιορίζεται σε σχέση με το Coords=2.

Ζ. Συμπεράσματα

Η Full-Offload (All-GPU) εκδοχή επιβεβαιώνει ότι η μεγαλύτερη πηγή απώλειας στις προηγούμενες υλοποιήσεις ήταν το CPU work + PCIe transfers μέσα στο iterative loop. Με το πλήρες offload, το πρόγραμμα γίνεται πραγματικά GPU-centric και το bottleneck μεταφέρεται κυρίως στον GPU χρόνο και ειδικά στο κόστος των atomics του update_centroids. Παρ' όλα αυτά, η συνολική επίδοση βελτιώνεται θεαματικά και η All-GPU αποτελεί το φυσικό επόμενο βήμα μετά τις βελτιώσεις πρόσβασης μνήμης (transpose) και επαναχρησιμοποίησης δεδομένων (shared).

- **<u>Γενικά Συμπεράσματα</u>**

Η συνολική εικόνα που προκύπτει είναι ότι η επίδοση δεν καθορίζεται μόνο από το πόσο γρήγορο είναι το kernel, αλλά από το πού βρίσκεται κάθε φορά το bottleneck (PCIe transfers, CPU τμήμα, global memory traffic, atomics). Έτσι, όσον αφορά τις 4 εκδόσεις (προγράμματα) που υλοποιήσαμε:

1. Στη naive εκδοχή, το βασικό κέρδος προκύπτει από τη μεταφορά του assignment step στη GPU, όμως η συνολική επιτάχυνση περιορίζεται από το ότι παραμένουν σημαντικά κόστη εκτός GPU: (i) οι μεταφορές δεδομένων (ιδίως η μεταφορά του membership προς τον host σε κάθε επανάληψη, που είναι O(N)) και (ii) το update_centroids στην CPU. Έτσι, ακόμη κι αν το kernel βελτιωθεί, το speedup περιορίζεται. λόγω Amdahl (μη παραλληλοποιημένο/μη offloaded μέρος).
2. Η transpose εκδοχή δείχνει καθαρά τη σημασία της διάταξης δεδομένων και της συν-αξιοποίησης της μνήμης (coalescing). Με το transposed layout, οι προσπελάσεις γίνονται πιο συνεκτικές (coalesced) και μειώνεται η σπατάλη bandwidth, με αποτέλεσμα αισθητή βελτίωση σε σχέση με τη naive, ειδικά όταν το workload είναι memory-bound. Παράλληλα, ο ρόλος του block_size γίνεται πιο επηρεάζει περισσότερο τη GPU (occupancy/latency hiding), καθώς η διαφορά από transfers/CPU αρχίζει να μειώνεται.
3. Στη shared εκδοχή, η μεταφορά των centroids σε shared memory λειτουργεί ως user-managed cache και μειώνει περαιτέρω τα global reads, οδηγώντας σε επιπλέον επιτάχυνση όταν το configuration το επιτρέπει. Ωστόσο, η τεχνική δεν έρχεται χωρίς κόστος: περιορίζεται από τη διαθέσιμη shared memory και μπορεί να επηρεάσει το occupancy, άρα υπάρχει πρακτικό όριο ως προς τα K·Coords και το block_size. Το συμπέρασμα είναι ότι η shared memory δίνει κέρδος όταν υπάρχει επανάχρηση δεδομένων ανά block, αλλά απαιτεί προσεκτικό διάβασμα των resource constraints.
4. Η all-GPU εκδοχή επιβεβαιώνει ότι το μεγαλύτερο κέρδος έρχεται όταν εξαλειφθούν τα υβριδικά κομμάτια μέσα στο iterative loop. Αφαιρώντας το per-iteration Device→Host membership και μεταφέροντας και το update_centroids στη GPU, μειώνεται δραστικά το transfer/CPU bottleneck, και ο συνολικός χρόνος κυριαρχείται πλέον από καθαρά GPU κόστη. Σε αυτήν τη φάση, το block_size επηρεάζει κυρίως μέσω occupancy/latency hiding, πίεσης σε registers και (κυρίως στο update_centroids) μέσω atomic contention. Ειδικά στο update_centroids, τα atomics μπορούν να αποτελέσουν σημαντικό περιορισμό, κάτι που εξηγεί γιατί το

all-GPU δεν κλιμακώνει πάντα όσο ιδανικά θα περιμέναμε χωρίς πρόσθετες τεχνικές μείωσης contention (π.χ. block-level partial sums σε shared και λιγότερα atomics προς global).

Όσον αφορά τις συντεταγμένες και το block size:

1. Η σύγκριση Coords=32 με Coords=2 δείχνει ότι το ίδιο «Size» δεν συνεπάγεται ίδιο υπολογιστικό κόστος: με μικρότερο Coords προκύπτει πολύ μεγαλύτερο πλήθος points N, άρα αυξάνει έντονα το workload του assignment και το μέγεθος του membership (O(N)). Αυτό μεταβάλλει το bottleneck: στο Coords=2 είναι πολύ πιο εύκολο να κυριαρχήσουν bandwidth/atomics ή ακόμη και οι μεταφορές membership (στις υβριδικές εκδοχές), ενώ στο Coords=32 το προφίλ είναι πιο ισορροπημένο και οι per-iteration μεταφορές centroids είναι πράγματι αμελητέες.
2. Τέλος, από τη μελέτη του block_size προκύπτει ότι δεν υπάρχει μία σωστή τιμή: η βέλτιστη επιλογή είναι αποτέλεσμα trade-off ανάμεσα σε occupancy, latency hiding, register/shared pressure και contention. Γι’ αυτό βλέπουμε sweet spots και όχι μονοτονικές τάσεις, ενώ οι πολύ μικρές ή πολύ μεγάλες τιμές μπορούν να υποβαθμίσουν την επίδοση (είτε λόγω χαμηλής αξιοποίησης είτε λόγω περιορισμού πόρων).

Σ.Η.Μ.Μ.Υ. Ε.Μ.Π.
Ιανουάριος 2026