

ΣΥΣΤΗΜΑΤΑ ΠΑΡΑΛΛΗΛΗΣ ΕΠΕΞΕΡΓΑΣΙΑΣ

ΑΝΑΦΟΡΑ 4^{ης} ΑΣΚΗΣΗΣ



Στοιχεία Ομάδας

- Αναγνωριστικό: parlab05
- Μέλος 1^ο: Πέππας Μιχαήλ – Αθανάσιος, Α.Μ: 03121026
- Μέλος 2^ο: 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 \leftrightarrow 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

A. 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:

a5/cuda_kmeans_naive.cu

```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4  #include "kmeans.h"
5  #include "alloc.h"
6  #include "error.h"
7
8  #ifdef __CUDACC__
9  inline void checkCuda(cudaError_t e) {
10     if (e != cudaSuccess) {
11         // cudaGetErrorString() isn't always very helpful. Look up the error
12         // number in the cudaError enum in driver_types.h in the CUDA includes
13         // directory for a better explanation.
14         error("CUDA Error %d: %s\n", e, cudaGetErrorString(e));
15     }
16 }
17
18 inline void checkLastCudaError() {
19     checkCuda(cudaGetLastError());
20 }
21 #endif
22
23 __device__ int get_tid() {
24     return blockIdx.x * blockDim.x + threadIdx.x;
25 }
26
27 /* square of Euclid distance between two multi-dimensional points */
28 __host__ __device__ inline static
29 double euclid_dist_2(int numCoords,
30                     int numObjs,
31                     int numClusters,
32                     double *objects,    // [numObjs][numCoords]
33                     double *clusters,   // [numClusters][numCoords]
34                     int objectId,
35                     int clusterId) {
36     int i;
37     double ans = 0.0;
38
39     /* TODO: Calculate the euclid_dist of elem=objectId of objects from elem=clusterId from
clusters*/
40     for (i = 0; i < numCoords; i++) {
41         double objectVal = objects[objectId * numCoords + i];
42         double clusterVal = clusters[clusterId * numCoords + i];
43
44         double diff = objectVal - clusterVal;
45         ans += diff * diff;
46     }
47
48     return (ans);
49 }
50
51 __global__ static

```

```

52 void find_nearest_cluster(int numCoords,
53                          int numObjs,
54                          int numClusters,
55                          double *objects,          // [numObjs][numCoords]
56                          double *deviceClusters,  // [numClusters][numCoords]
57                          int *deviceMembership,    // [numObjs]
58                          double *devdelta) {
59
60     /* Get the global ID of the thread. */
61     int tid = get_tid();
62
63     if (tid < numObjs) {
64         int index, i;
65         double dist, min_dist;
66
67         /* find the cluster id that has min distance to object */
68         index = 0;
69
70         min_dist = euclid_dist_2(numCoords, numObjs, numClusters,
71                                objects, deviceClusters,
72                                tid, index);
73
74         for (i = 1; i < numClusters; i++) {
75
76             dist = euclid_dist_2(numCoords, numObjs, numClusters,
77                                objects, deviceClusters,
78                                tid, i);
79             /* no need square root */
80             if (dist < min_dist) { /* find the min and its array index */
81                 min_dist = dist;
82                 index = i;
83             }
84         }
85
86         if (deviceMembership[tid] != index) {
87
88             atomicAdd(devdelta, 1.0);
89         }
90
91         /* assign the deviceMembership to object objectId */
92         deviceMembership[tid] = index;
93     }
94 }
95
96 //
97 // -----
98 // DATA LAYOUT
99 //
100 // objects          [numObjs][numCoords]
101 // clusters          [numClusters][numCoords]
102 // newClusters       [numClusters][numCoords]
103 // deviceObjects     [numObjs][numCoords]
104 // deviceClusters    [numClusters][numCoords]
105 // -----

```

```

106 //
107 /* return an array of cluster centers of size [numClusters][numCoords] */
108 void kmeans_gpu(double *objects, /* in: [numObjs][numCoords] */
109               int numCoords, /* no. features */
110               int numObjs, /* no. objects */
111               int numClusters, /* no. clusters */
112               double threshold, /* % objects change membership */
113               long loop_threshold, /* maximum number of iterations */
114               int *membership, /* out: [numObjs] */
115               double *clusters, /* out: [numClusters][numCoords] */
116               int blockSize) {
117     double timing = wtime(), timing_internal, timer_min = 1e42, timer_max = 0;
118     double timing_gpu, timing_cpu, timing_transfers, transfers_time = 0.0, cpu_time = 0.0,
119     gpu_time = 0.0;
120     int loop_iterations = 0;
121     int i, j, index, loop = 0;
122     int *newClusterSize; /* [numClusters]: no. objects assigned in each
123                          new cluster */
124     double delta = 0, *dev_delta_ptr; /* % of objects change their clusters */
125     double **newClusters = (double **) calloc_2d(numClusters, numCoords, sizeof(double));
126     double *deviceObjects;
127     double *deviceClusters;
128     int *deviceMembership;
129
130     printf("\n|-----Naive GPU Kmeans-----|\n\n");
131
132
133     /* initialize membership[] */
134     for (i = 0; i < numObjs; i++) membership[i] = -1;
135
136     /* need to initialize newClusterSize and newClusters[0] to all 0 */
137     newClusterSize = (int *) calloc(numClusters, sizeof(int));
138     assert(newClusterSize != NULL);
139
140     timing = wtime() - timing;
141     printf("t_alloc: %lf ms\n\n", 1000 * timing);
142     timing = wtime();
143
144     const unsigned int numThreadsPerClusterBlock = (numObjs > blockSize) ? blockSize :
145     numObjs;
146     const unsigned int numClusterBlocks = (numObjs + numThreadsPerClusterBlock - 1) /
147     numThreadsPerClusterBlock; /* TODO: Calculate Grid size, e.g. number of blocks. */
148     const unsigned int clusterBlockSharedDataSize = 0;
149
150     checkCuda(cudaMalloc(&deviceObjects, numObjs * numCoords * sizeof(double)));
151     checkCuda(cudaMalloc(&deviceClusters, numClusters * numCoords * sizeof(double)));
152     checkCuda(cudaMalloc(&deviceMembership, numObjs * sizeof(int)));
153     checkCuda(cudaMalloc(&dev_delta_ptr, sizeof(double)));
154
155     timing = wtime() - timing;
156     printf("t_alloc_gpu: %lf ms\n\n", 1000 * timing);
157     timing = wtime();
158

```

```
157     checkCuda(cudaMemcpy(deviceObjects, objects,
158                           numObjs * numCoords * sizeof(double), cudaMemcpyHostToDevice));
159     checkCuda(cudaMemcpy(deviceMembership, membership,
160                           numObjs * sizeof(int), cudaMemcpyHostToDevice));
161     timing = wtime() - timing;
162     printf("t_get_gpu: %lf ms\n\n", 1000 * timing);
163     timing = wtime();
164
165     do {
166         timing_internal = wtime();
167
168         /* GPU part: calculate new memberships */
169
170         timing_transfers = wtime();
171
172         checkCuda(cudaMemcpy(deviceClusters, clusters,
173                               numClusters * numCoords * sizeof(double),
174                               cudaMemcpyHostToDevice));
175
176         transfers_time += wtime() - timing_transfers;
177
178         checkCuda(cudaMemset(dev_delta_ptr, 0, sizeof(double)));
179
180         //printf("Launching find_nearest_cluster Kernel with grid_size = %d, block_size = %d,
181         shared_mem = %d KB\n", numClusterBlocks, numThreadsPerClusterBlock, clusterBlockSharedDa-
182         taSize/1000);
183         timing_gpu = wtime();
184         find_nearest_cluster
185         <<< numClusterBlocks, numThreadsPerClusterBlock, clusterBlockSharedDataSize >>>
186         (numCoords, numObjs, numClusters,
187          deviceObjects, deviceClusters, deviceMembership, dev_delta_ptr);
188
189         cudaDeviceSynchronize();
190         checkLastCudaError();
191         gpu_time += wtime() - timing_gpu;
192         //printf("Kernels complete for itter %d, updating data in CPU\n", loop);
193
194         timing_transfers = wtime();
195
196         checkCuda(cudaMemcpy(membership, deviceMembership,
197                               numObjs * sizeof(int),
198                               cudaMemcpyDeviceToHost));
199
200         checkCuda(cudaMemcpy(&delta, dev_delta_ptr,
201                               sizeof(double),
202                               cudaMemcpyDeviceToHost));
203
204         transfers_time += wtime() - timing_transfers;
205
206         /* CPU part: Update cluster centers*/
207         timing_cpu = wtime();
208         for (i = 0; i < numObjs; i++) {
209             /* find the array index of nestest cluster center */
210             index = membership[i];
```

```

209
210     /* update new cluster centers : sum of objects located within */
211     newClusterSize[index]++;
212     for (j = 0; j < numCoords; j++)
213         newClusters[index][j] += objects[i * numCoords + j];
214 }
215
216 /* average the sum and replace old cluster centers with newClusters */
217 for (i = 0; i < numClusters; i++) {
218     for (j = 0; j < numCoords; j++) {
219         if (newClusterSize[i] > 0)
220             clusters[i * numCoords + j] = newClusters[i][j] / newClusterSize[i];
221         newClusters[i][j] = 0.0; /* set back to 0 */
222     }
223     newClusterSize[i] = 0; /* set back to 0 */
224 }
225
226 delta /= numObjs;
227 //printf("delta is %f - ", delta);
228 loop++;
229 //printf("completed loop %d\n", loop);
230 cpu_time += wtime() - timing_cpu;
231
232 timing_internal = wtime() - timing_internal;
233 if (timing_internal < timer_min) timer_min = timing_internal;
234 if (timing_internal > timer_max) timer_max = timing_internal;
235 } while (delta > threshold && loop < loop_threshold);
236
237 timing = wtime() - timing;
238 printf("nloops = %d : total = %lf ms\n\t-> t_loop_avg = %lf ms\n\t-> t_loop_min = %lf
ms\n\t-> t_loop_max = %lf ms\n\t"
239     "-> t_cpu_avg = %lf ms\n\t-> t_gpu_avg = %lf ms\n\t-> t_transfers_avg = %lf
ms\n\n|-----|\n",
240     loop, 1000 * timing, 1000 * timing / loop, 1000 * timer_min, 1000 * timer_max,
241     1000 * cpu_time / loop, 1000 * gpu_time / loop, 1000 * transfers_time / loop);
242
243 char outfile_name[1024] = {0};
244 sprintf(outfile_name, "Execution_logs/silver1-V100_Sz-%lu_Coo-%d_Cl-%d.csv",
245     numObjs * numCoords * sizeof(double) / (1024 * 1024), numCoords, numClusters);
246 FILE *fp = fopen(outfile_name, "a+");
247 if (!fp) error("Filename %s did not open succesfully, no logging performed\n",
outfile_name);
248 fprintf(fp, "%s,%d,%lf,%lf,%lf\n", "Naive", blockSize, timing / loop, timer_min,
timer_max);
249 fclose(fp);
250 checkCuda(cudaFree(deviceObjects));
251 checkCuda(cudaFree(deviceClusters));
252 checkCuda(cudaFree(deviceMembership));
253
254 free(newClusters[0]);
255 free(newClusters);
256 free(newClusterSize);
257
258 return;

```



```
259 | }  
260  
261
```

B. Υλοποίηση και Ορθότητα

(α) Υπολογισμός απόστασης (euclid dist 2)

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

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

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 $\text{ceil}(\text{numObjs} / \text{block_size})$, ώστε να καλύπτονται όλα τα objects, και γίνεται έλεγχος ορίων ($tid < \text{numObjs}$).

(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.

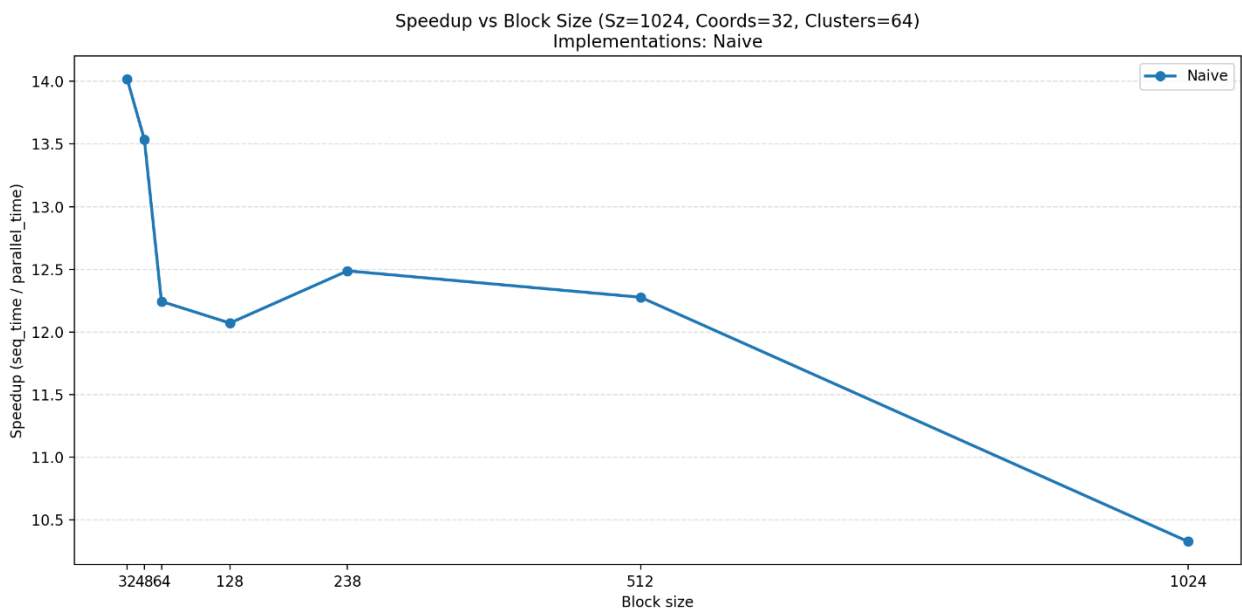
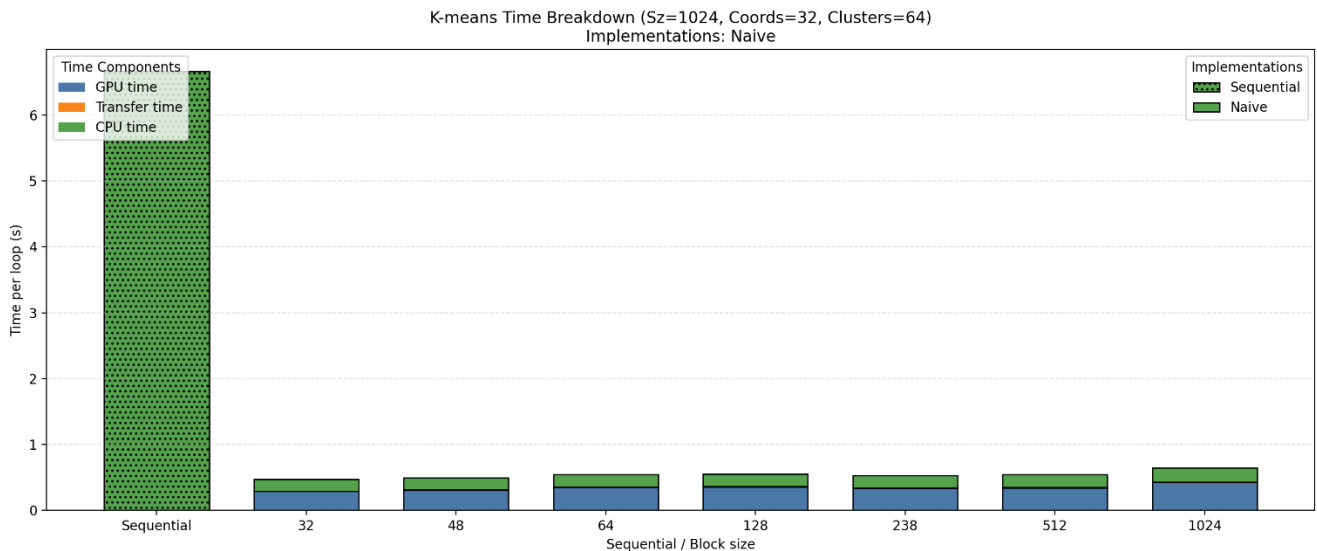
(δ) Timers (breakdown CPU / GPU / Transfers)

Στη naive έκδοση καταγράφονται τρεις συνιστώσες χρόνου ανά loop:

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

Επισημαίνεται ότι το «μεγάλο» H \rightarrow 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

A. Εισαγωγή

Η έκδοση Transpose στοχεύει αποκλειστικά στη βελτιστοποίηση των προσπελάσεων της global μνήμης στην GPU. Στην naïve έκδοση, τα threads ενός warp που επεξεργάζονται διαδοχικά objects προσπελούν 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:

a5/cuda_kmeans_transpose.cu

```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4  #include "kmeans.h"
5  #include "alloc.h"
6  #include "error.h"
7
8  #ifdef __CUDACC__
9  inline void checkCuda(cudaError_t e) {
10     if (e != cudaSuccess) {
11         // cudaGetErrorString() isn't always very helpful. Look up the error
12         // number in the cudaError enum in driver_types.h in the CUDA includes
13         // directory for a better explanation.
14         error("CUDA Error %d: %s\n", e, cudaGetErrorString(e));
15     }
16 }
17
18 inline void checkLastCudaError() {
19     checkCuda(cudaGetLastError());
20 }
21 #endif
22
23 __device__ int get_tid() {
24     return blockIdx.x * blockDim.x + threadIdx.x;
25 }
26
27 /* square of Euclid distance between two multi-dimensional points using column-base format
28 */
29 __host__ __device__ inline static
30 double euclid_dist_2_transpose(int numCoords,
31                                int numObjs,
32                                int numClusters,
33                                double *objects,    // [numCoords][numObjs]
34                                double *clusters,    // [numCoords][numClusters]
35                                int objectId,
36                                int clusterId) {
37     int i;
38     double ans = 0.0;
39
40     /* TODO: Calculate the euclid_dist of elem=objectId of objects from elem=clusterId from
41     clusters, but for column-base format!!! */
42     for (i = 0; i < numCoords; i++) {
43         double objectVal = objects[i * numObjs + objectId];
44         double clusterVal = clusters[i * numClusters + clusterId];
45
46         double diff = objectVal - clusterVal;
47         ans += diff * diff;
48     }
49
50     return (ans);
51 }

```

```

51 __global__ static
52 void find_nearest_cluster(int numCoords,
53                           int numObjs,
54                           int numClusters,
55                           double *objects,          // [numCoords][numObjs]
56                           double *deviceClusters,   // [numCoords][numClusters]
57                           int *membership,          // [numObjs]
58                           double *devdelta) {
59     /* Get the global ID of the thread. */
60     int tid = get_tid();
61
62     if (tid < numObjs) {
63         int index, i;
64         double dist, min_dist;
65
66         /* find the cluster id that has min distance to object */
67         index = 0;
68
69         min_dist = euclid_dist_2_transpose(numCoords, numObjs, numClusters,
70                                           objects, deviceClusters,
71                                           tid, index);
72
73         for (i = 1; i < numClusters; i++) {
74
75             dist = euclid_dist_2_transpose(numCoords, numObjs, numClusters,
76                                           objects, deviceClusters,
77                                           tid, i);
78             /* no need square root */
79             if (dist < min_dist) { /* find the min and its array index */
80                 min_dist = dist;
81                 index = i;
82             }
83         }
84
85         if (membership[tid] != index) {
86
87             atomicAdd(devdelta, 1.0);
88         }
89
90         /* assign the deviceMembership to object objectId */
91         membership[tid] = index;
92     }
93 }
94
95 //
96 // -----
97 // DATA LAYOUT
98 //
99 // objects          [numObjs][numCoords]
100 // clusters          [numClusters][numCoords]
101 // dimObjects        [numCoords][numObjs]
102 // dimClusters       [numCoords][numClusters]
103 // newClusters       [numCoords][numClusters]
104 // deviceObjects     [numCoords][numObjs]

```



```

105 // deviceClusters [numCoords][numClusters]
106 // -----
107 //
108 /* return an array of cluster centers of size [numClusters][numCoords] */
109 void kmeans_gpu(double *objects, /* in: [numObjs][numCoords] */
110                int numCoords, /* no. features */
111                int numObjs, /* no. objects */
112                int numClusters, /* no. clusters */
113                double threshold, /* % objects change membership */
114                long loop_threshold, /* maximum number of iterations */
115                int *membership, /* out: [numObjs] */
116                double *clusters, /* out: [numClusters][numCoords] */
117                int blockSize) {
118     double timing = wtime(), timing_internal, timer_min = 1e42, timer_max = 0;
119     double timing_gpu, timing_cpu, timing_transfers, transfers_time = 0.0, cpu_time = 0.0,
gpu_time = 0.0;
120     int loop_iterations = 0;
121     int i, j, index, loop = 0;
122     int *newClusterSize; /* [numClusters]: no. objects assigned in each
123                          new cluster */
124     double delta = 0, *dev_delta_ptr; /* % of objects change their clusters */
125
126     /* TODO: Transpose dims */
127     double **dimObjects = (double **) calloc_2d(numCoords, numObjs, sizeof(double));
128     //calloc_2d(...) -> [numCoords][numObjs]
129     double **dimClusters = (double **) calloc_2d(numCoords, numClusters, sizeof(double));
130     //calloc_2d(...) -> [numCoords][numClusters]
131     double **newClusters = (double **) calloc_2d(numCoords, numClusters, sizeof(double));
132     //calloc_2d(...) -> [numCoords][numClusters]
133
134     double *deviceObjects;
135     double *deviceClusters;
136     int *deviceMembership;
137
138     printf("\n|-----Transpose GPU Kmeans-----|\n\n");
139
140     // TODO: Copy objects given in [numObjs][numCoords] layout to new
141     // [numCoords][numObjs] layout
142     for (i=0 ; i < numObjs; i++){
143         for (j=0; j<numCoords; j++){
144             dimObjects[j][i]=objects[i*numCoords + j];
145         }
146     }
147
148     /* pick first numClusters elements of objects[] as initial cluster centers*/
149     for (i = 0; i < numCoords; i++) {
150         for (j = 0; j < numClusters; j++) {
151             dimClusters[i][j] = dimObjects[i][j];
152         }
153     }
154
155     /* initialize membership[] */
156     for (i = 0; i < numObjs; i++) membership[i] = -1;
157

```

```

155  /* need to initialize newClusterSize and newClusters[0] to all 0 */
156  newClusterSize = (int *) calloc(numClusters, sizeof(int));
157  assert(newClusterSize != NULL);
158
159  timing = wtime() - timing;
160  printf("t_alloc: %lf ms\n\n", 1000 * timing);
161  timing = wtime();
162
163  const unsigned int numThreadsPerClusterBlock = (numObjs > blockSize) ? blockSize :
numObjs;
164  const unsigned int numClusterBlocks = (numObjs + numThreadsPerClusterBlock - 1) /
numThreadsPerClusterBlock;
165  const unsigned int clusterBlockSharedDataSize = 0;
166
167  checkCuda(cudaMalloc(&deviceObjects, numObjs * numCoords * sizeof(double)));
168  checkCuda(cudaMalloc(&deviceClusters, numClusters * numCoords * sizeof(double)));
169  checkCuda(cudaMalloc(&deviceMembership, numObjs * sizeof(int)));
170  checkCuda(cudaMalloc(&dev_delta_ptr, sizeof(double)));
171  timing = wtime() - timing;
172  printf("t_alloc_gpu: %lf ms\n\n", 1000 * timing);
173  timing = wtime();
174
175  checkCuda(cudaMemcpy(deviceObjects, dimObjects[0],
176                      numObjs * numCoords * sizeof(double), cudaMemcpyHostToDevice));
177  checkCuda(cudaMemcpy(deviceMembership, membership,
178                      numObjs * sizeof(int), cudaMemcpyHostToDevice));
179  timing = wtime() - timing;
180  printf("t_get_gpu: %lf ms\n\n", 1000 * timing);
181  timing = wtime();
182
183  do {
184      timing_internal = wtime();
185
186      /* GPU part: calculate new memberships */
187
188      timing_transfers = wtime();
189      // TODO: Copy clusters to deviceClusters
190      checkCuda(cudaMemcpy(deviceClusters, dimClusters[0],
191                          numClusters * numCoords * sizeof(double),
192                          cudaMemcpyHostToDevice));
193
194      transfers_time += wtime() - timing_transfers;
195
196      checkCuda(cudaMemset(dev_delta_ptr, 0, sizeof(double)));
197
198      //printf("Launching find_nearest_cluster Kernel with grid_size = %d, block_size = %d,
shared_mem = %d KB\n", numClusterBlocks, numThreadsPerClusterBlock, clusterBlockSharedDa-
taSize/1000);
199      timing_gpu = wtime();
200      find_nearest_cluster
201      <<< numClusterBlocks, numThreadsPerClusterBlock, clusterBlockSharedDataSize >>>
202          (numCoords, numObjs, numClusters,
203          deviceObjects, deviceClusters, deviceMembership, dev_delta_ptr);
204

```

```

205     cudaDeviceSynchronize();
206     checkLastCudaError();
207     gpu_time += wtime() - timing_gpu;
208     //printf("Kernels complete for itter %d, updating data in CPU\n", loop);
209
210     timing_transfers = wtime();
211
212     checkCuda(cudaMemcpy(membership, deviceMembership,
213                          numObjs * sizeof(int),
214                          cudaMemcpyDeviceToHost));
215
216     checkCuda(cudaMemcpy(&delta, dev_delta_ptr,
217                          sizeof(double),
218                          cudaMemcpyDeviceToHost));
219     transfers_time += wtime() - timing_transfers;
220
221     /* CPU part: Update cluster centers*/
222
223     timing_cpu = wtime();
224     for (i = 0; i < numObjs; i++) {
225         /* find the array index of nestest cluster center */
226         index = membership[i];
227
228         /* update new cluster centers : sum of objects located within */
229         newClusterSize[index]++;
230         for (j = 0; j < numCoords; j++)
231             newClusters[j][index] += objects[i * numCoords + j];
232     }
233
234     /* average the sum and replace old cluster centers with newClusters */
235     for (i = 0; i < numClusters; i++) {
236         for (j = 0; j < numCoords; j++) {
237             if (newClusterSize[i] > 0)
238                 dimClusters[j][i] = newClusters[j][i] / newClusterSize[i];
239             newClusters[j][i] = 0.0; /* set back to 0 */
240         }
241         newClusterSize[i] = 0; /* set back to 0 */
242     }
243
244     delta /= numObjs;
245     //printf("delta is %f - ", delta);
246     loop++;
247     //printf("completed loop %d\n", loop);
248     cpu_time += wtime() - timing_cpu;
249
250     timing_internal = wtime() - timing_internal;
251     if (timing_internal < timer_min) timer_min = timing_internal;
252     if (timing_internal > timer_max) timer_max = timing_internal;
253 } while (delta > threshold && loop < loop_threshold);
254
255 /*TODO: Update clusters using dimClusters. Be carefull of layout!!!
clusters[numClusters][numCoords] vs dimClusters[numCoords][numClusters] */
256 for (i = 0; i < numClusters; i++) {
257     for (j = 0; j < numCoords; j++) {

```

```

258     clusters[i * numCoords + j] = dimClusters[j][i];
259 }
260 }
261
262 timing = wtime() - timing;
263 printf("nloops = %d : total = %lf ms\n\t-> t_loop_avg = %lf ms\n\t-> t_loop_min = %lf
ms\n\t-> t_loop_max = %lf ms\n\t"
264        "-> t_cpu_avg = %lf ms\n\t-> t_gpu_avg = %lf ms\n\t-> t_transfers_avg = %lf
ms\n\n|-----|\n",
265        loop, 1000 * timing, 1000 * timing / loop, 1000 * timer_min, 1000 * timer_max,
266        1000 * cpu_time / loop, 1000 * gpu_time / loop, 1000 * transfers_time / loop);
267
268 char outfile_name[1024] = {0};
269 sprintf(outfile_name, "Execution_logs/silver1-V100_Sz-%lu_Coo-%d_Cl-%d.csv",
270         numObjs * numCoords * sizeof(double) / (1024 * 1024), numCoords, numClusters);
271 FILE *fp = fopen(outfile_name, "a+");
272 if (!fp) error("Filename %s did not open succesfully, no logging performed\n",
outfile_name);
273 fprintf(fp, "%s,%d,%lf,%lf,%lf\n", "Transpose", blockSize, timing / loop, timer_min,
timer_max);
274 fclose(fp);
275
276 checkCuda(cudaFree(deviceObjects));
277 checkCuda(cudaFree(deviceClusters));
278 checkCuda(cudaFree(deviceMembership));
279
280 free(dimObjects[0]);
281 free(dimObjects);
282 free(dimClusters[0]);
283 free(dimClusters);
284 free(newClusters[0]);
285 free(newClusters);
286 free(newClusterSize);
287
288 return;
289 }
290
291

```

B. 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

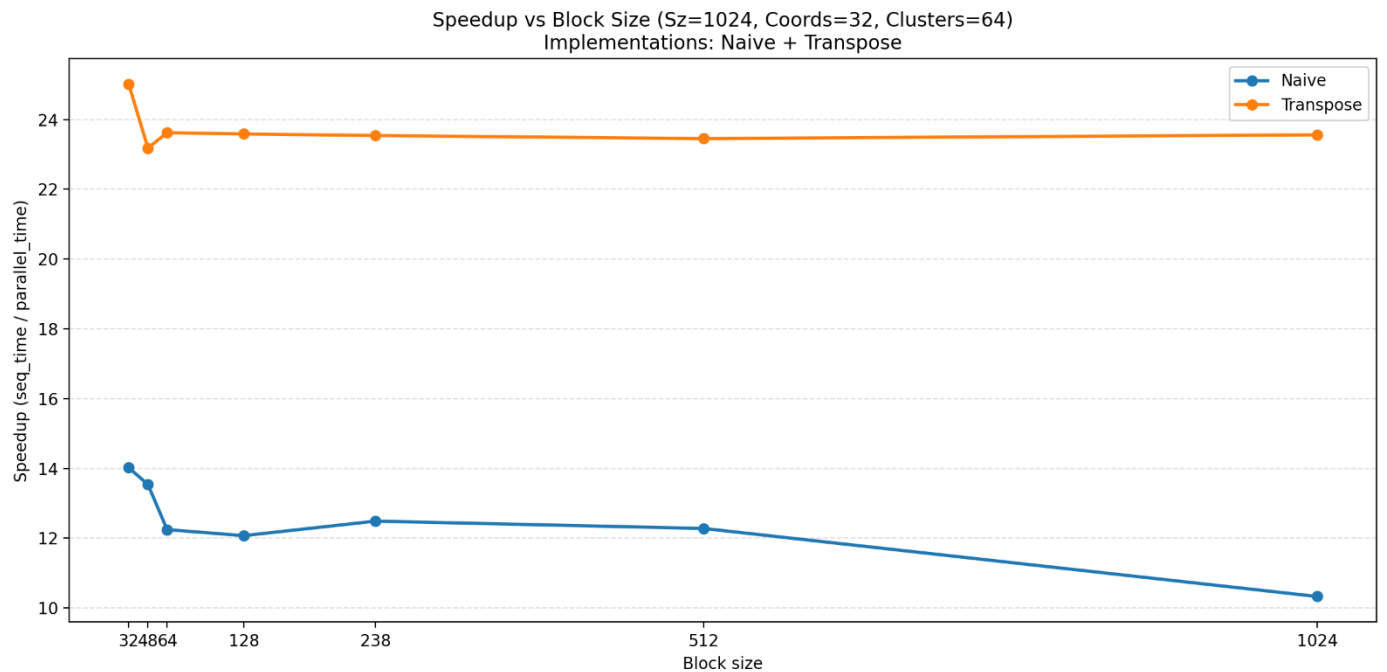
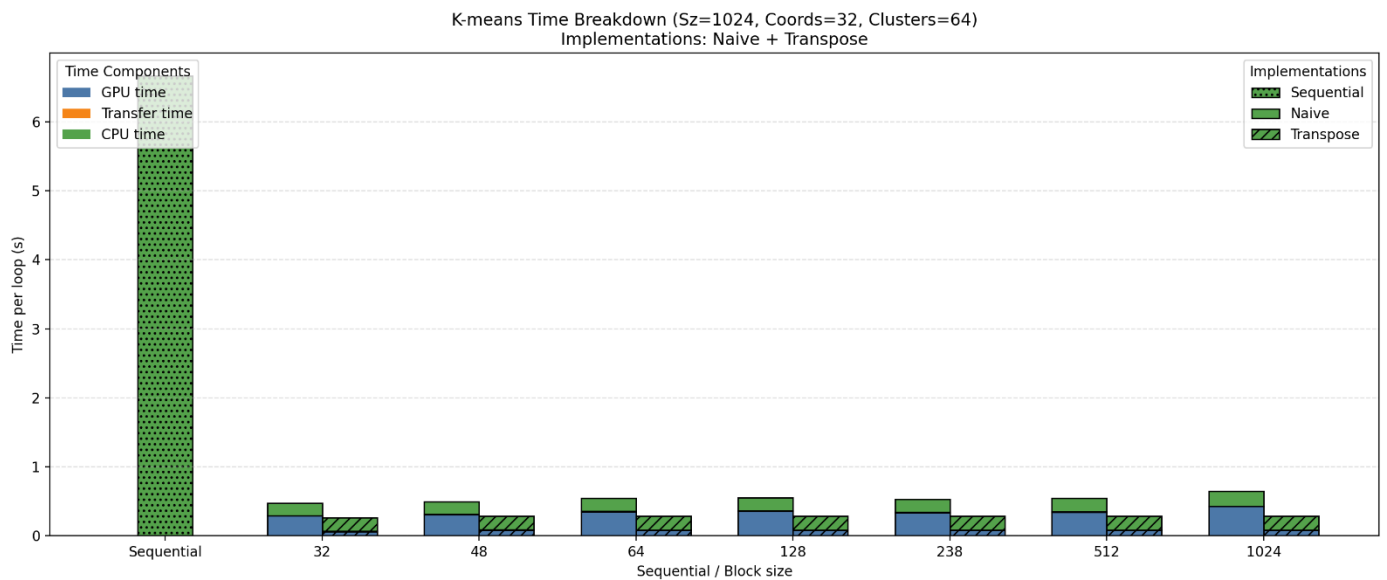
Πριν ξεκινήσει το loop, ο host κατασκευάζει τον dimObjects πίνακα από το αρχικό row-major objects. Αυτό είναι κόστος προεπεξεργασίας (εκτός loop) και δεν επηρεάζει το per-loop breakdown.

(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 $\sim 1.7\times-2\times$ 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 \rightarrow συνεχόμενες διευθύνσεις \rightarrow 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

A. 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:

a5/cuda_kmeans_shared.cu

```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4  #include "kmeans.h"
5  #include "alloc.h"
6  #include "error.h"
7
8  #ifdef __CUDACC__
9  inline void checkCuda(cudaError_t e) {
10     if (e != cudaSuccess) {
11         // cudaGetErrorString() isn't always very helpful. Look up the error
12         // number in the cudaError enum in driver_types.h in the CUDA includes
13         // directory for a better explanation.
14         error("CUDA Error %d: %s\n", e, cudaGetErrorString(e));
15     }
16 }
17
18 inline void checkLastCudaError() {
19     checkCuda(cudaGetLastError());
20 }
21 #endif
22
23 __device__ int get_tid() {
24     return blockIdx.x * blockDim.x + threadIdx.x;
25 }
26
27 /* square of Euclid distance between two multi-dimensional points using column-base format
28 */
29 __host__ __device__ inline static
30 double euclid_dist_2_transpose(int numCoords,
31                                int numObjs,
32                                int numClusters,
33                                double *objects,    // [numCoords][numObjs]
34                                double *clusters,    // [numCoords][numClusters]
35                                int objectId,
36                                int clusterId) {
37     int i;
38     double ans = 0.0;
39
40     /* TODO: Calculate the euclid_dist of elem=objectId of objects from elem=clusterId from
41     clusters, but for column-base format!!! */
42     for (i = 0; i < numCoords; i++) {
43         double objectVal = objects[i * numObjs + objectId];
44         double clusterVal = clusters[i * numClusters + clusterId];
45
46         double diff = objectVal - clusterVal;
47         ans += diff * diff;
48     }
49
50     return (ans);
51 }

```

```
51 __global__ static
52 void find_nearest_cluster(int numCoords,
53                           int numObjs,
54                           int numClusters,
55                           double *objects,          // [numCoords][numObjs]
56                           double *deviceClusters,    // [numCoords][numClusters]
57                           int *deviceMembership,      // [numObjs]
58                           double *devdelta) {
59     extern __shared__ double shmemClusters[];
60
61     // TODO: Copy deviceClusters to shmemClusters so they can be accessed faster.
62     int tid_in_block = threadIdx.x;          // Το ID του νήματος μέσα στο Block
63     int block_size = blockDim.x;             // Πόσα νήματα έχει το Block
64     int total_cluster_doubles = numClusters * numCoords; // Συνολικά νούμερα προς αντιγραφή
65
66     // Κάθε νήμα αντιγράφει όσα στοιχεία του αναλογούν (με βήμα block_size)
67     for (int k = tid_in_block; k < total_cluster_doubles; k += block_size) {
68         shmemClusters[k] = deviceClusters[k];
69     }
70
71     /* Συγχρονισμός (BARRIER) */
72
73     __syncthreads();
74
75     /* Get the global ID of the thread. */
76     int tid = get_tid();
77
78     /* TODO: Maybe something is missing here... should all threads run this? */
79     if (tid < numObjs) {
80         int index, i;
81         double dist, min_dist;
82
83         /* find the cluster id that has min distance to object */
84         index = 0;
85         /* TODO: call min_dist = euclid_dist_2(...) with correct objectId/clusterId using
clusters in shmem*/
86
87
88         min_dist = euclid_dist_2_transpose(numCoords, numObjs, numClusters,
89                                           objects, shmemClusters,
90                                           tid, index);
91
92         for (i = 1; i < numClusters; i++) {
93             dist = euclid_dist_2_transpose(numCoords, numObjs, numClusters,
94                                           objects, shmemClusters,
95                                           tid, i);
96
97             /* no need square root */
98             if (dist < min_dist) { /* find the min and its array index */
99                 min_dist = dist;
100                 index = i;
101             }
102         }
103     }
```

```

104     if (deviceMembership[tid] != index) {
105         /* TODO: Maybe something is missing here... is this write safe? */
106         atomicAdd(&devdelta, 1.0);
107     }
108
109     /* assign the deviceMembership to object objectId */
110     deviceMembership[tid] = index;
111 }
112 }
113
114 //
115 // -----
116 // DATA LAYOUT
117 //
118 // objects      [numObjs][numCoords]
119 // clusters      [numClusters][numCoords]
120 // dimObjects     [numCoords][numObjs]
121 // dimClusters    [numCoords][numClusters]
122 // newClusters    [numCoords][numClusters]
123 // deviceObjects  [numCoords][numObjs]
124 // deviceClusters [numCoords][numClusters]
125 // -----
126 //
127 /* return an array of cluster centers of size [numClusters][numCoords] */
128 void kmeans_gpu(double *objects, /* in: [numObjs][numCoords] */
129                int numCoords, /* no. features */
130                int numObjs, /* no. objects */
131                int numClusters, /* no. clusters */
132                double threshold, /* % objects change membership */
133                long loop_threshold, /* maximum number of iterations */
134                int *membership, /* out: [numObjs] */
135                double *clusters, /* out: [numClusters][numCoords] */
136                int blockSize) {
137     double timing = wtime(), timing_internal, timer_min = 1e42, timer_max = 0;
138     double timing_gpu, timing_cpu, timing_transfers, transfers_time = 0.0, cpu_time = 0.0,
139     gpu_time = 0.0;
140     int loop_iterations = 0;
141     int i, j, index, loop = 0;
142     int *newClusterSize; /* [numClusters]: no. objects assigned in each
143                          new cluster */
144     double delta = 0, *dev_delta_ptr; /* % of objects change their clusters */
145     /* TODO: Copy me from transpose version */
146     double **dimObjects = (double **) calloc_2d(numCoords, numObjs, sizeof(double));
147     //calloc_2d(...) -> [numCoords][numObjs]
148     double **dimClusters = (double **) calloc_2d(numCoords, numClusters, sizeof(double));
149     //calloc_2d(...) -> [numCoords][numClusters]
150     double **newClusters = (double **) calloc_2d(numCoords, numClusters, sizeof(double));
151     //calloc_2d(...) -> [numCoords][numClusters]
152
153     double *deviceObjects;
154     double *deviceClusters;
155     int *deviceMembership;
156
157     printf("\n|-----Shared GPU Kmeans-----|\n\n");

```

```
154
155 /* TODO: Copy me from transpose version*/
156 for (i=0 ; i < numObjs; i++){
157     for (j=0; j<numCoords; j++){
158         dimObjects[j][i]=objects[i*numCoords + j];
159     }
160 }
161
162 /* pick first numClusters elements of objects[] as initial cluster centers*/
163 for (i = 0; i < numCoords; i++) {
164     for (j = 0; j < numClusters; j++) {
165         dimClusters[i][j] = dimObjects[i][j];
166     }
167 }
168
169 /* initialize membership[] */
170 for (i = 0; i < numObjs; i++) membership[i] = -1;
171
172 /* need to initialize newClusterSize and newClusters[0] to all 0 */
173 newClusterSize = (int *) calloc(numClusters, sizeof(int));
174 assert(newClusterSize != NULL);
175
176 timing = wtime() - timing;
177 printf("t_alloc: %lf ms\n\n", 1000 * timing);
178 timing = wtime();
179 const unsigned int numThreadsPerClusterBlock = (numObjs > blockSize) ? blockSize :
numObjs;
180 const unsigned int numClusterBlocks = (numObjs + numThreadsPerClusterBlock - 1) /
numThreadsPerClusterBlock; /* TODO: Calculate Grid size, e.g. number of blocks. */
181
182 /* Define the shared memory needed per block.
183 - BEWARE: We can overrun our shared memory here if there are too many
184 clusters or too many coordinates!
185 - This can lead to occupancy problems or even inability to run.
186 - Your exercise implementation is not requested to account for that (e.g. always
assume deviceClusters fit in shmemClusters */
187 const unsigned int clusterBlockSharedDataSize = numClusters*numCoords*sizeof(double);
188
189 cudaDeviceProp deviceProp;
190 int deviceNum;
191 cudaGetDevice(&deviceNum);
192 cudaGetDeviceProperties(&deviceProp, deviceNum);
193
194 if (clusterBlockSharedDataSize > deviceProp.sharedMemPerBlock) {
195     error("Your CUDA hardware has insufficient block shared memory to hold all cluster
centroids\n");
196 }
197
198 checkCuda(cudaMalloc(&deviceObjects, numObjs * numCoords * sizeof(double)));
199 checkCuda(cudaMalloc(&deviceClusters, numClusters * numCoords * sizeof(double)));
200 checkCuda(cudaMalloc(&deviceMembership, numObjs * sizeof(int)));
201 checkCuda(cudaMalloc(&dev_delta_ptr, sizeof(double)));
202
203 timing = wtime() - timing;
```

```
204     printf("t_alloc_gpu: %lf ms\n\n", 1000 * timing);
205     timing = wtime();
206
207     checkCuda(cudaMemcpy(deviceObjects, dimObjects[0],
208                          numObjs * numCoords * sizeof(double), cudaMemcpyHostToDevice));
209     checkCuda(cudaMemcpy(deviceMembership, membership,
210                          numObjs * sizeof(int), cudaMemcpyHostToDevice));
211     timing = wtime() - timing;
212     printf("t_get_gpu: %lf ms\n\n", 1000 * timing);
213     timing = wtime();
214
215     do {
216         timing_internal = wtime();
217
218         /* GPU part: calculate new memberships */
219
220         timing_transfers = wtime();
221         // TODO: Copy clusters to deviceClusters
222         checkCuda(cudaMemcpy(deviceClusters, dimClusters[0],
223                              numClusters * numCoords * sizeof(double),
224                              cudaMemcpyHostToDevice));
225
226         transfers_time += wtime() - timing_transfers;
227
228         checkCuda(cudaMemset(dev_delta_ptr, 0, sizeof(double)));
229
230         timing_gpu = wtime();
231         //printf("Launching find_nearest_cluster Kernel with grid_size = %d, block_size = %d,
232         //shared_mem = %d KB\n", numClusterBlocks, numThreadsPerClusterBlock, clusterBlockSharedDa-
233         //taSize/1000);
234         find_nearest_cluster
235         <<< numClusterBlocks, numThreadsPerClusterBlock, clusterBlockSharedDataSize >>>
236         (numCoords, numObjs, numClusters,
237         deviceObjects, deviceClusters, deviceMembership, dev_delta_ptr);
238
239         cudaDeviceSynchronize();
240         checkLastCudaError();
241         gpu_time += wtime() - timing_gpu;
242         //printf("Kernels complete for itter %d, updating data in CPU\n", loop);
243
244         timing_transfers = wtime();
245
246         checkCuda(cudaMemcpy(membership, deviceMembership,
247                              numObjs * sizeof(int),
248                              cudaMemcpyDeviceToHost));
249
250         checkCuda(cudaMemcpy(&delta, dev_delta_ptr,
251                              sizeof(double),
252                              cudaMemcpyDeviceToHost));
253
254         transfers_time += wtime() - timing_transfers;
255
256         /* CPU part: Update cluster centers*/
```

```

256
257     timing_cpu = wtime();
258     for (i = 0; i < numObjs; i++) {
259         /* find the array index of nestest cluster center */
260         index = membership[i];
261
262         /* update new cluster centers : sum of objects located within */
263         newClusterSize[index]++;
264         for (j = 0; j < numCoords; j++)
265             newClusters[j][index] += objects[i * numCoords + j];
266     }
267
268     /* average the sum and replace old cluster centers with newClusters */
269     for (i = 0; i < numClusters; i++) {
270         for (j = 0; j < numCoords; j++) {
271             if (newClusterSize[i] > 0)
272                 dimClusters[j][i] = newClusters[j][i] / newClusterSize[i];
273             newClusters[j][i] = 0.0;    /* set back to 0 */
274         }
275         newClusterSize[i] = 0;    /* set back to 0 */
276     }
277
278     delta /= numObjs;
279     //printf("delta is %f - ", delta);
280     loop++;
281     //printf("completed loop %d\n", loop);
282     cpu_time += wtime() - timing_cpu;
283
284     timing_internal = wtime() - timing_internal;
285     if (timing_internal < timer_min) timer_min = timing_internal;
286     if (timing_internal > timer_max) timer_max = timing_internal;
287 } while (delta > threshold && loop < loop_threshold);
288
289 /*TODO: Update clusters using dimClusters. Be carefull of layout!!!
clusters[numClusters][numCoords] vs dimClusters[numCoords][numClusters] */
290 for (i = 0; i < numClusters; i++) {
291     for (j = 0; j < numCoords; j++) {
292         clusters[i * numCoords + j] = dimClusters[j][i];
293     }
294 }
295
296 timing = wtime() - timing;
297 printf("nloops = %d : total = %lf ms\n\t-> t_loop_avg = %lf ms\n\t-> t_loop_min = %lf
ms\n\t-> t_loop_max = %lf ms\n\t"
298     "-> t_cpu_avg = %lf ms\n\t-> t_gpu_avg = %lf ms\n\t-> t_transfers_avg = %lf
ms\n\n|-----|\n",
299     loop, 1000 * timing, 1000 * timing / loop, 1000 * timer_min, 1000 * timer_max,
300     1000 * cpu_time / loop, 1000 * gpu_time / loop, 1000 * transfers_time / loop);
301
302 char outfile_name[1024] = {0};
303 sprintf(outfile_name, "Execution_logs/silver1-V100-Sz-%lu_Coo-%d_Cl-%d.csv",
304     numObjs * numCoords * sizeof(double) / (1024 * 1024), numCoords, numClusters);
305 FILE *fp = fopen(outfile_name, "a+");

```

```
306     if (!fp) error("Filename %s did not open succesfully, no logging performed\n",
outfile_name);
307     fprintf(fp, "%s,%d,%lf,%lf,%lf\n", "Shmem", blockSize, timing / loop, timer_min,
timer_max);
308     fclose(fp);
309
310     checkCuda(cudaFree(deviceObjects));
311     checkCuda(cudaFree(deviceClusters));
312     checkCuda(cudaFree(deviceMembership));
313
314     free(dimObjects[0]);
315     free(dimObjects);
316     free(dimClusters[0]);
317     free(dimClusters);
318     free(newClusters[0]);
319     free(newClusters);
320     free(newClusterSize);
321
322     return;
323 }
324
325
```


B. 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)

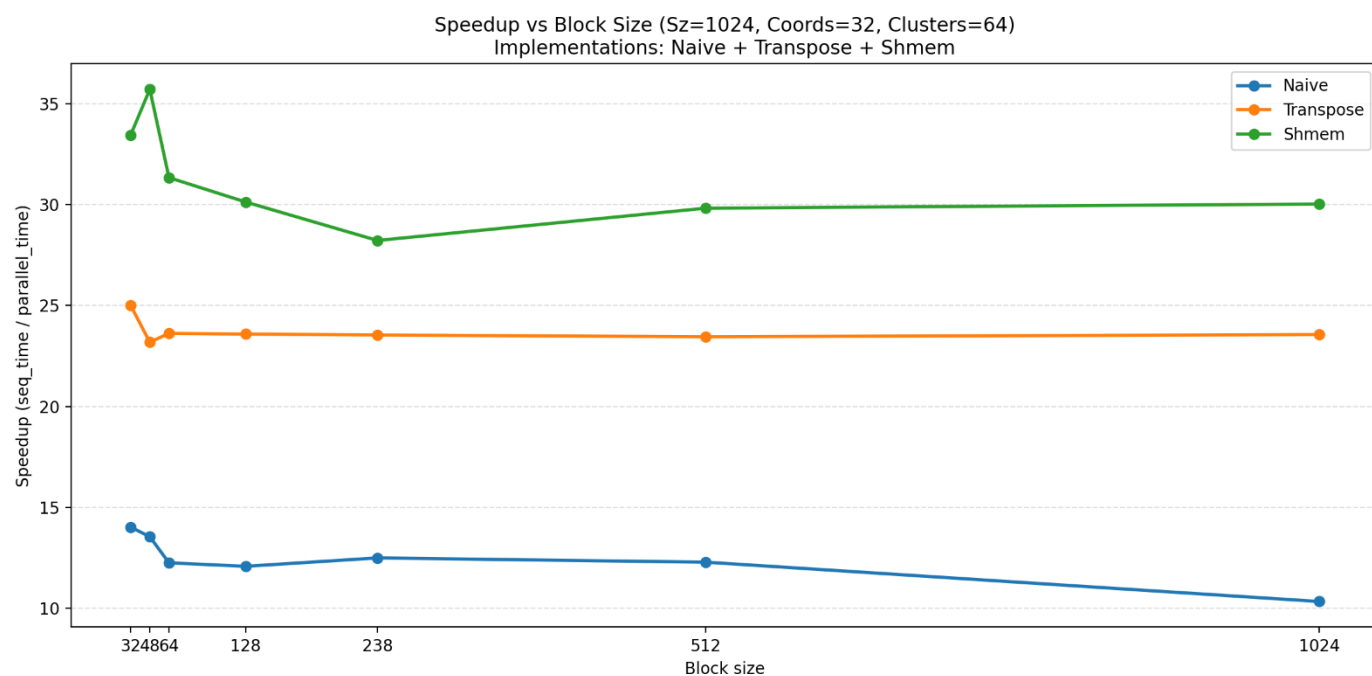
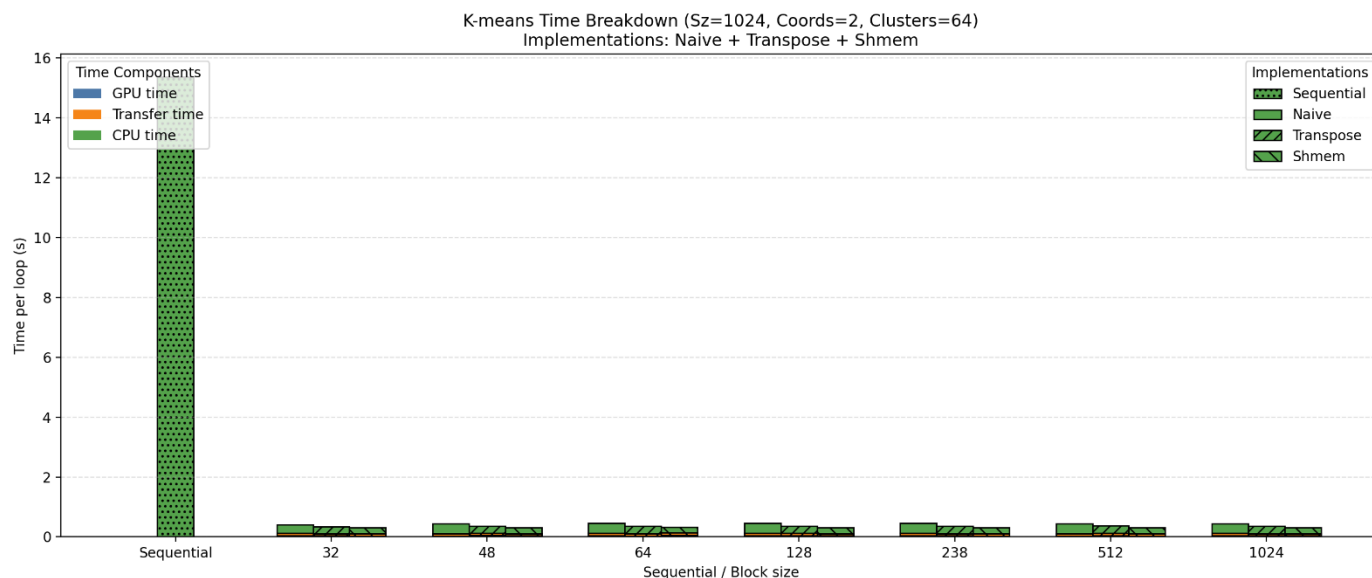
Μετά τη φόρτωση στη shared, γίνεται __syncthreads() ώστε να εξασφαλιστεί ότι όλα τα threads του block βλέπουν πλήρως γραμμένα τα δεδομένα πριν ξεκινήσουν τους υπολογισμούς αποστάσεων. Αυτό είναι απαραίτητο για ορθότητα (διαφορετικά κάποια threads θα διάβαζαν μη αρχικοποιημένες τιμές).

(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 $H \rightarrow D$, membership+delta $D \rightarrow H$) 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: $\text{numClusters} * \text{numCoords} * \text{sizeof(double)}$), 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

A. 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 \leftrightarrow GPU / CPU update). Υπενθυμίζουμε ότι τα transfers που μετράμε εδώ αφορούν τις αντιγραφές μέσα στο loop (π.χ. clusters $H \rightarrow D$ και membership+delta $D \rightarrow H$) και όχι την αρχική μεταφορά του dataset προς τη GPU, η οποία γίνεται μία φορά πριν ξεκινήσουν οι επαναλήψεις.

B. 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 \rightarrow Transpose: η κύρια μείωση χρόνου έρχεται από το GPU time, επειδή το transpose βελτιώνει το memory coalescing (λιγότερα global memory transactions ανά warp).
- Transpose \rightarrow 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:

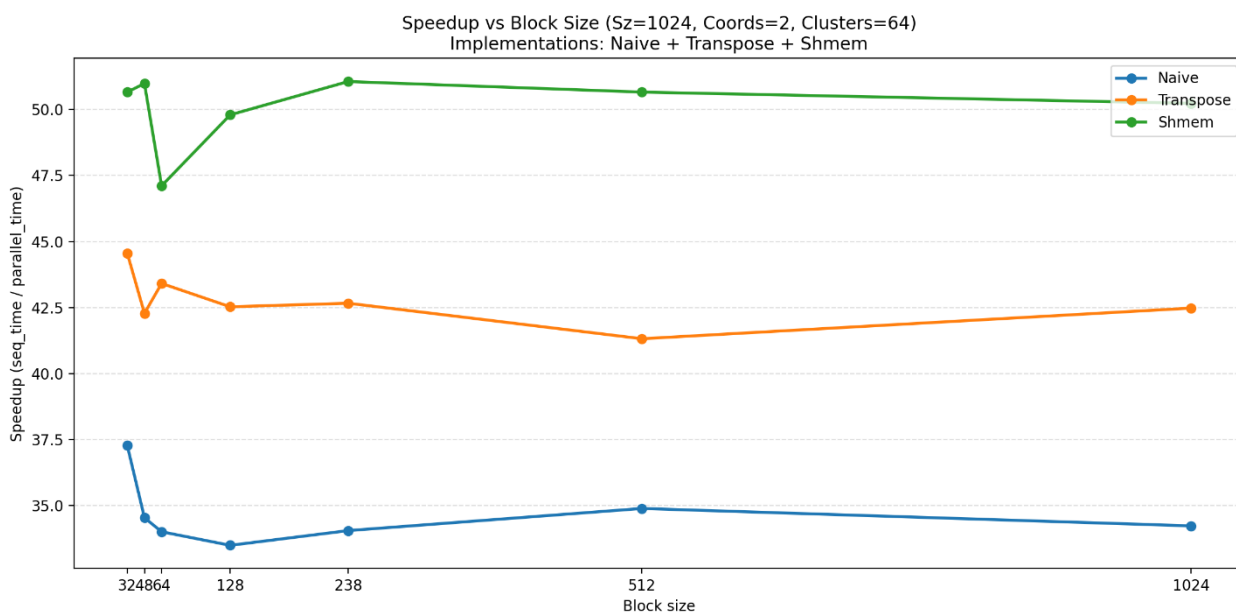
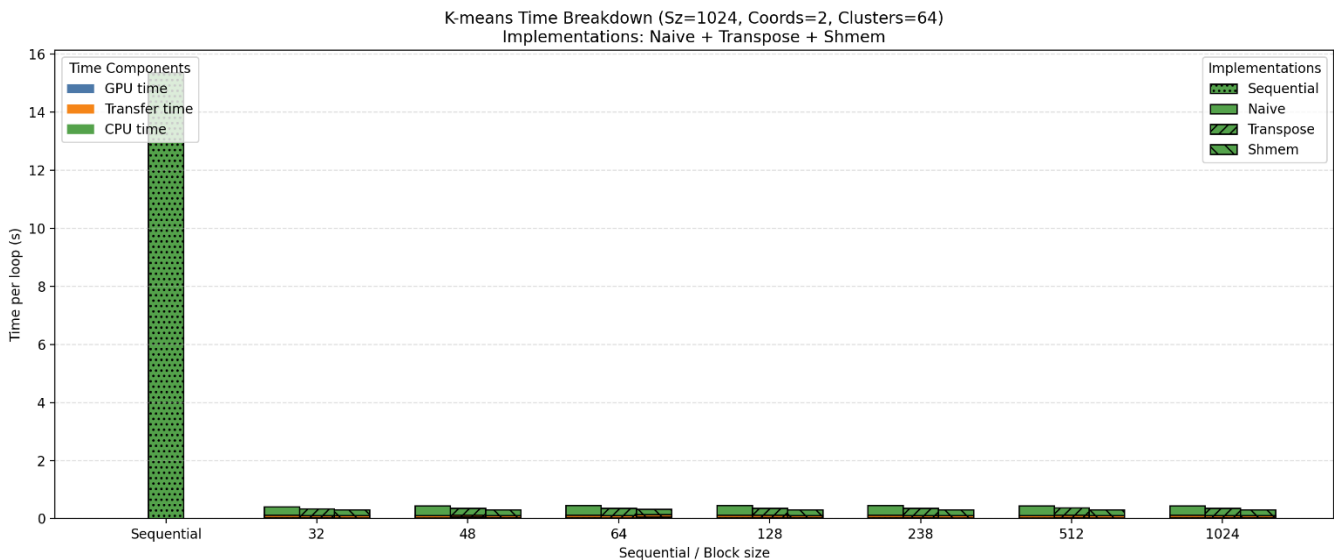
- CPU time (update_centroids): παραμένει σειριακό στην CPU στις τρεις εκδόσεις και θέτει όριο βάσει του Amdahl (όσο μικραίνει το GPU kernel, τόσο μεγαλύτερο ποσοστό του συνολικού χρόνου καταλαμβάνει το CPU update).

- 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 $16\times$ ($\text{numObjs} \propto 1/\text{numCoords}$). Αυτό έχει δύο κρίσιμες συνέπειες:

1. Αυξάνεται δραστικά το μέγεθος του membership που πρέπει να επιστρέφει στη CPU σε κάθε επανάληψη ($D \rightarrow H$), άρα ο χρόνος transfers γίνεται πολύ πιο σημαντικός σε σχέση με το $\text{Coords}=32$.
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 $\text{Coords}=2$ επιβεβαιώνουν αυτή τη μετατόπιση: ενώ η Shared παραμένει η ταχύτερη υλοποίηση ($\text{Shared} > \text{Transpose} > \text{Naive}$), 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 $(\text{numClusters} \times \text{numCoords})$ 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 $\text{Coords}=2$, όπου αυξάνεται έντονα το communication overhead (membership $D \rightarrow H$) 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 \leftrightarrow 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:

a5/cuda_kmeans_all_gpu.cu

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  #include "kmeans.h"
5  #include "alloc.h"
6  #include "error.h"
7
8  #ifdef __CUDACC__
9  inline void checkCuda(cudaError_t e)
10 {
11     if (e != cudaSuccess)
12     {
13         // cudaGetErrorString() isn't always very helpful. Look up the error
14         // number in the cudaError enum in driver_types.h in the CUDA includes
15         // directory for a better explanation.
16         error("CUDA Error %d: %s\n", e, cudaGetErrorString(e));
17     }
18 }
19
20 inline void checkLastCudaError()
21 {
22     checkCuda(cudaGetLastError());
23 }
24 #endif
25
26 __device__ int get_tid()
27 {
28     return blockIdx.x * blockDim.x + threadIdx.x;
29 }
30
31 /* square of Euclid distance between two multi-dimensional points using column-base format
32 */
33 __host__ __device__ inline static double euclid_dist_2_transpose(int numCoords,
34                                                                    int numObjs,
35                                                                    int numClusters,
36                                                                    double *objects, //
37                                                                    [numCoords][numObjs]
38                                                                    double *clusters, //
39                                                                    [numCoords][numClusters]
40                                                                    int objectId,
41                                                                    int clusterId)
42 {
43     int i;
44     double ans = 0.0;
45
46     /* TODO: Calculate the euclid_dist of elem=objectId of objects from elem=clusterId from
47     clusters, but for column-base format!!! */
48     for (i = 0; i < numCoords; i++)
49     {
50         double objectVal = objects[i * numObjs + objectId];
51         double clusterVal = clusters[i * numClusters + clusterId];
```

```

49     double diff = objectVal - clusterVal;
50     ans += diff * diff;
51 }
52
53 return (ans);
54 }
55
56 __global__ static void find_nearest_cluster(int numCoords,
57                                           int numObjs,
58                                           int numClusters,
59                                           double *deviceObjects, // [numCoords]
60                                           [numObjs]
61                                           /*
62 TODO: If you choose to do (some of) the new centroid calculation here, you will need some
63 extra parameters here (from "update_centroids").
64                                           */
65                                           int *devicenewClusterSize,
66                                           double *devicenewClusters, // [numCoords]
67                                           [numClusters]
68                                           double *deviceClusters, // [numCoords]
69                                           [numClusters]
70                                           int *deviceMembership, // [numObjs]
71                                           double *devdelta)
72 {
73     extern __shared__ double shmemClusters[];
74     // TODO: Copy deviceClusters to shmemClusters so they can be accessed faster.
75     int tid_in_block = threadIdx.x; // Το ID του νήματος μέσα στο Block
76     int block_size = blockDim.x; // Πόσα νήματα έχει το Block
77     int total_cluster_doubles = numClusters * numCoords; // Συνολικά νούμερα προς αντιγραφή
78
79     // Κάθε νήμα αντιγράφει όσα στοιχεία του αναλογούν (με βήμα block_size)
80     for (int k = tid_in_block; k < total_cluster_doubles; k += block_size)
81     {
82         shmemClusters[k] = deviceClusters[k];
83     }
84
85     /* Συγχρονισμός (BARRIER) */
86     __syncthreads();
87
88     /* Get the global ID of the thread. */
89     int tid = get_tid();
90
91     /* TODO: Maybe something is missing here... should all threads run this? */
92     if (tid < numObjs)
93     {
94         int index, i;
95         double dist, min_dist;
96
97         /* find the cluster id that has min distance to object */
98         index = 0;
99         /* TODO: call min_dist = euclid_dist_2(...) with correct objectId/clusterId using
100 clusters in shmem*/

```

```

97
98     min_dist = euclid_dist_2_transpose(numCoords, numObjs, numClusters,
99                                         deviceObjects, shmemClusters,
100                                         tid, index);
101
102     for (i = 1; i < numClusters; i++)
103     {
104         dist = euclid_dist_2_transpose(numCoords, numObjs, numClusters,
105                                         deviceObjects, shmemClusters,
106                                         tid, i);
107
108         /* no need square root */
109         if (dist < min_dist)
110         { /* find the min and its array index */
111             min_dist = dist;
112             index = i;
113         }
114     }
115
116     if (deviceMembership[tid] != index)
117     {
118         /* TODO: Maybe something is missing here... is this write safe? */
119         atomicAdd(&devdelta, 1.0);
120     }
121
122     /* assign the deviceMembership to object objectId */
123     deviceMembership[tid] = index;
124
125     /* TODO: additional steps for calculating new centroids in GPU? */
126
127     atomicAdd(&devicenewClusterSize[index], 1);
128
129     for (int j = 0; j < numCoords; j++)
130     {
131         // Διαβάζουμε την τιμή του αντικειμένου (Coordinate j, Object tid)
132         double objVal = deviceObjects[j * numObjs + tid];
133
134         // Προσθέτουμε στο άθροισμα (Coordinate j, Cluster index)
135         atomicAdd(&devicenewClusters[j * numClusters + index], objVal);
136     }
137 }
138 }
139
140 __global__ static void update_centroids(int numCoords,
141                                         int numClusters,
142                                         int *devicenewClusterSize, // [numClusters]
143                                         double *devicenewClusters, // [numCoords]
144                                         [numClusters]
145                                         double *deviceClusters) // [numCoords]
146 {
147     /* Κάθε νήμα αναλαμβάνει ΜΙΑ τιμή (double) του πίνακα clusters.
148     Συνολικά νήματα = numCoords * numClusters
149     */

```

```

149     int tid = get_tid();
150     int total_elements = numCoords * numClusters;
151
152     if (tid < total_elements)
153     {
154         // Αποκωδικοποίηση του 1D tid σε 2D (Coordinate, Cluster)
155         // Layout: [numCoords][numClusters] --> index = coord * numClusters + cluster
156         int clusterId = tid % numClusters;
157         // int coordId = tid / numClusters; // Δεν το χρειαζόμαστε άμεσα για τον υπολογισμό,
        αλλά για το reset
158
159         int count = devicenewClusterSize[clusterId];
160
161         // Υπολόγισε το νέο κέντρο (Average)
162         if (count > 0)
163         {
164             double sum = devicenewClusters[tid];
165             deviceClusters[tid] = sum / count;
166         }
167         // Αν count == 0, κρατάμε την παλιά τιμή (ή δεν κάνουμε τίποτα), όπως και στον CPU
        κώδικα
168
169         // RESET για τον επόμενο γύρο (Πολύ σημαντικό!)
170         // Μηδενίζουμε το άθροισμα που μόλις χρησιμοποιήσαμε
171         devicenewClusters[tid] = 0.0;
172     }
173 }
174
175 //
176 // -----
177 // DATA LAYOUT
178 //
179 // objects      [numObjs][numCoords]
180 // clusters      [numClusters][numCoords]
181 // dimObjects    [numCoords][numObjs]
182 // dimClusters   [numCoords][numClusters]
183 // newClusters   [numCoords][numClusters]
184 // deviceObjects [numCoords][numObjs]
185 // deviceClusters [numCoords][numClusters]
186 // -----
187 //
188 /* return an array of cluster centers of size [numClusters][numCoords] */
189 void kmeans_gpu(double *objects, /* in: [numObjs][numCoords] */
190                int numCoords, /* no. features */
191                int numObjs, /* no. objects */
192                int numClusters, /* no. clusters */
193                double threshold, /* % objects change membership */
194                long loop_threshold, /* maximum number of iterations */
195                int *membership, /* out: [numObjs] */
196                double *clusters, /* out: [numClusters][numCoords] */
197                int blockSize)
198 {
199     double timing = wtime(), timing_internal, timer_min = 1e42, timer_max = 0;

```

```

200     double timing_gpu, timing_cpu, timing_transfers, transfers_time = 0.0, cpu_time = 0.0,
    gpu_time = 0.0;
201     int loop_iterations = 0;
202     int i, j, index, loop = 0;
203     double delta = 0, *dev_delta_ptr; /* % of objects change their clusters */
204     /* TODO: Copy me from transpose version*/
205     double **dimObjects = (double **)calloc_2d(numCoords, numObjs, sizeof(double)); //
    calloc_2d(...) -> [numCoords][numObjs]
206     double **dimClusters = (double **)calloc_2d(numCoords, numClusters, sizeof(double)); //
    calloc_2d(...) -> [numCoords][numClusters]
207     double **newClusters = (double **)calloc_2d(numCoords, numClusters, sizeof(double));
208
209     printf("\n|-----Full-offload GPU Kmeans-----|\n\n");
210
211     /* TODO: Copy me from transpose version*/
212     for (i = 0; i < numObjs; i++)
213     {
214         for (j = 0; j < numCoords; j++)
215         {
216             dimObjects[j][i] = objects[i * numCoords + j];
217         }
218     }
219
220     double *deviceObjects;
221     double *deviceClusters, *devicenewClusters;
222     int *deviceMembership;
223     int *devicenewClusterSize; /* [numClusters]: no. objects assigned in each new cluster */
224
225     /* pick first numClusters elements of objects[] as initial cluster centers*/
226     for (i = 0; i < numCoords; i++)
227     {
228         for (j = 0; j < numClusters; j++)
229         {
230             dimClusters[i][j] = dimObjects[i][j];
231         }
232     }
233
234     /* initialize membership[] */
235     for (i = 0; i < numObjs; i++)
236         membership[i] = -1;
237
238     timing = wtime() - timing;
239     printf("t_alloc: %lf ms\n\n", 1000 * timing);
240     timing = wtime();
241     const unsigned int numThreadsPerClusterBlock = (numObjs > blockSize) ? blockSize :
    numObjs;
242     const unsigned int numClusterBlocks = (numObjs + numThreadsPerClusterBlock - 1) /
    numThreadsPerClusterBlock; /* TODO: Calculate Grid size, e.g. number of blocks. */
243
244     /* Define the shared memory needed per block.
245     - BEWARE: We can overrun our shared memory here if there are too many
246     clusters or too many coordinates!
247     - This can lead to occupancy problems or even inability to run.

```

```

248     - Your exercise implementation is not requested to account for that (e.g. always
assume deviceClusters fit in shmemClusters */
249     const unsigned int clusterBlockSharedDataSize = numClusters * numCoords *
sizeof(double);
250
251     cudaDeviceProp deviceProp;
252     int deviceNum;
253     cudaGetDevice(&deviceNum);
254     cudaGetDeviceProperties(&deviceProp, deviceNum);
255
256     if (clusterBlockSharedDataSize > deviceProp.sharedMemPerBlock)
257     {
258         error("Your CUDA hardware has insufficient block shared memory to hold all cluster
centroids\n");
259     }
260
261     checkCuda(cudaMalloc(&deviceObjects, numObjs * numCoords * sizeof(double)));
262     checkCuda(cudaMalloc(&deviceClusters, numClusters * numCoords * sizeof(double)));
263     checkCuda(cudaMalloc(&devicenewClusters, numClusters * numCoords * sizeof(double)));
264     checkCuda(cudaMalloc(&devicenewClusterSize, numClusters * sizeof(int)));
265     checkCuda(cudaMalloc(&deviceMembership, numObjs * sizeof(int)));
266     checkCuda(cudaMalloc(&dev_delta_ptr, sizeof(double)));
267
268     timing = wtime() - timing;
269     printf("t_alloc_gpu: %lf ms\n\n", 1000 * timing);
270     timing = wtime();
271
272     checkCuda(cudaMemcpy(deviceObjects, dimObjects[0],
numObjs * numCoords * sizeof(double), cudaMemcpyHostToDevice));
273     checkCuda(cudaMemcpy(deviceMembership, membership,
numObjs * sizeof(int), cudaMemcpyHostToDevice));
274     checkCuda(cudaMemcpy(deviceClusters, dimClusters[0],
numClusters * numCoords * sizeof(double), cudaMemcpyHostToDevice));
275     checkCuda(cudaMemcpy(deviceClusters, dimClusters[0],
numClusters * numCoords * sizeof(double), cudaMemcpyHostToDevice));
276     checkCuda(cudaMemset(devicenewClusterSize, 0, numClusters * sizeof(int)));
277     free(dimObjects[0]);
278
279     timing = wtime() - timing;
280     printf("t_get_gpu: %lf ms\n\n", 1000 * timing);
281     timing = wtime();
282
283     do
284     {
285         timing_internal = wtime();
286         checkCuda(cudaMemset(dev_delta_ptr, 0, sizeof(double)));
287         checkCuda(cudaMemset(devicenewClusterSize, 0, numClusters * sizeof(int)));
288         timing_gpu = wtime();
289         // printf("Launching find_nearest_cluster Kernel with grid_size = %d, block_size = %d,
shared_mem = %d KB\n", numClusterBlocks, numThreadsPerClusterBlock, clusterBlockSharedDa-
taSize/1000);
290         // TODO: change invocation if extra parameters needed
291         find_nearest_cluster<<<numClusterBlocks, numThreadsPerClusterBlock,
clusterBlockSharedDataSize>>>(numCoords, numObjs, numClusters,
292         deviceObjects, devicenewClusterSize, devicenewClusters, deviceClusters, deviceMembership,
dev_delta_ptr);

```

```

295
296     cudaDeviceSynchronize();
297     checkLastCudaError();
298
299     gpu_time += wtime() - timing_gpu;
300
301     // printf("Kernels complete for itter %d, updating data in CPU\n", loop);
302
303     timing_transfers = wtime();
304     // TODO: Copy dev_delta_ptr to &delta
305     checkCuda(cudaMemcpy(&delta, dev_delta_ptr, sizeof(double), cudaMemcpyDeviceToHost));
306     transfers_time += wtime() - timing_transfers;
307
308     const unsigned int update_centroids_block_sz = (numCoords * numClusters > blockSize) ?
blockSize : numCoords * numClusters; /* TODO: can use different blocksize here if
deemed better */
309     const unsigned int update_centroids_dim_sz = (numCoords * numClusters +
update_centroids_block_sz - 1) / update_centroids_block_sz; /* TODO: calculate dim for
"update_centroids" */
310     timing_gpu = wtime();
311     // TODO: use dim for "update_centroids" and fire it
312     update_centroids<<<update_centroids_dim_sz, update_centroids_block_sz, 0>>>(numCoords,
numClusters, devicenewClusterSize, devicenewClusters, deviceClusters);
313     cudaDeviceSynchronize();
314     checkLastCudaError();
315     gpu_time += wtime() - timing_gpu;
316
317     timing_cpu = wtime();
318     delta /= numObjs;
319     // printf("delta is %f - ", delta);
320     loop++;
321     // printf("completed loop %d\n", loop);
322     cpu_time += wtime() - timing_cpu;
323
324     timing_internal = wtime() - timing_internal;
325     if (timing_internal < timer_min)
326         timer_min = timing_internal;
327     if (timing_internal > timer_max)
328         timer_max = timing_internal;
329 } while (delta > threshold && loop < loop_threshold);
330
331 checkCuda(cudaMemcpy(membership, deviceMembership,
332                     numObjs * sizeof(int), cudaMemcpyDeviceToHost));
333 checkCuda(cudaMemcpy(dimClusters[0], deviceClusters,
334                     numClusters * numCoords * sizeof(double), cudaMemcpyDeviceToHost));
335
336 for (i = 0; i < numClusters; i++)
337 {
338     for (j = 0; j < numCoords; j++)
339     {
340         clusters[i * numCoords + j] = dimClusters[j][i];
341     }
342 }
343

```

```
344     timing = wtime() - timing;
345     printf("nloops = %d : total = %lf ms\n\t-> t_loop_avg = %lf ms\n\t-> t_loop_min = %lf
ms\n\t-> t_loop_max = %lf ms\n\t"
346         "-> t_cpu_avg = %lf ms\n\t-> t_gpu_avg = %lf ms\n\t-> t_transfers_avg = %lf
ms\n\n|-----|\n",
347         loop, 1000 * timing, 1000 * timing / loop, 1000 * timer_min, 1000 * timer_max,
348         1000 * cpu_time / loop, 1000 * gpu_time / loop, 1000 * transfers_time / loop);
349
350     char outfile_name[1024] = {0};
351     sprintf(outfile_name, "Execution_logs/silver1-V100_Sz-%lu_Coo-%d_Cl-%d.csv",
352         numObjs * numCoords * sizeof(double) / (1024 * 1024), numCoords, numClusters);
353     FILE *fp = fopen(outfile_name, "a+");
354     if (!fp)
355         error("Filename %s did not open succesfully, no logging performed\n", outfile_name);
356     fprintf(fp, "%s,%d,%lf,%lf,%lf\n", "All_GPU", blockSize, timing / loop, timer_min,
timer_max);
357     fclose(fp);
358
359     checkCuda(cudaFree(deviceObjects));
360     checkCuda(cudaFree(deviceClusters));
361     checkCuda(cudaFree(devicenewClusters));
362     checkCuda(cudaFree(devicenewClusterSize));
363     checkCuda(cudaFree(deviceMembership));
364
365     return;
366 }
367
```


B. Υλοποίηση και Ορθότητα

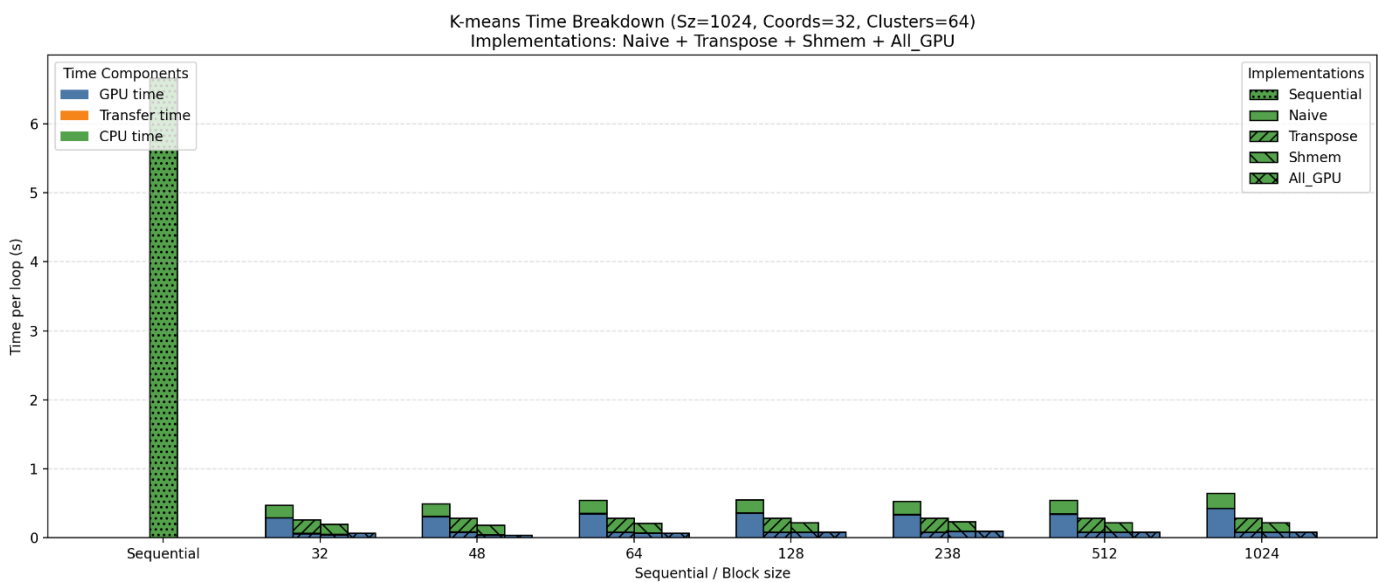
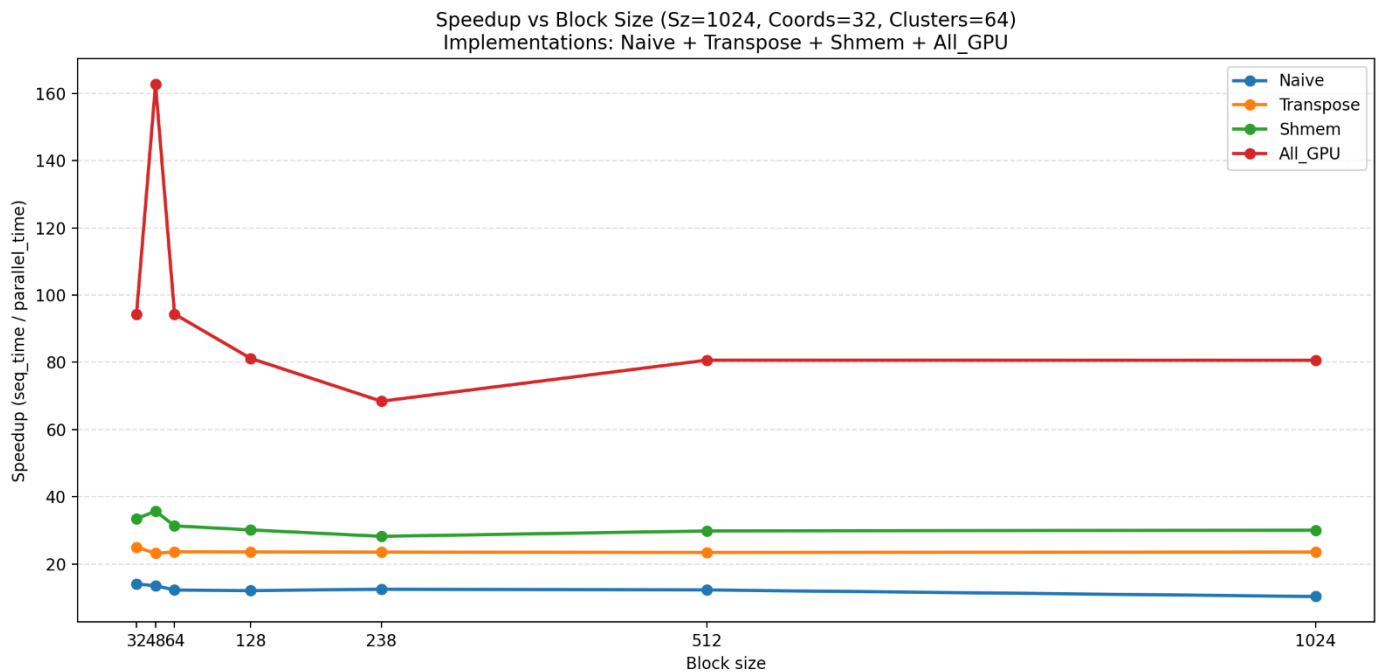
Η λογική της All-GPU υλοποίησης είναι να σπάσει το `update_centroids` σε βήματα που μπορούν να γίνουν ασφαλώς στη GPU χωρίς καθολικό `barrier` μέσα σε ένα `kernel`:

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.

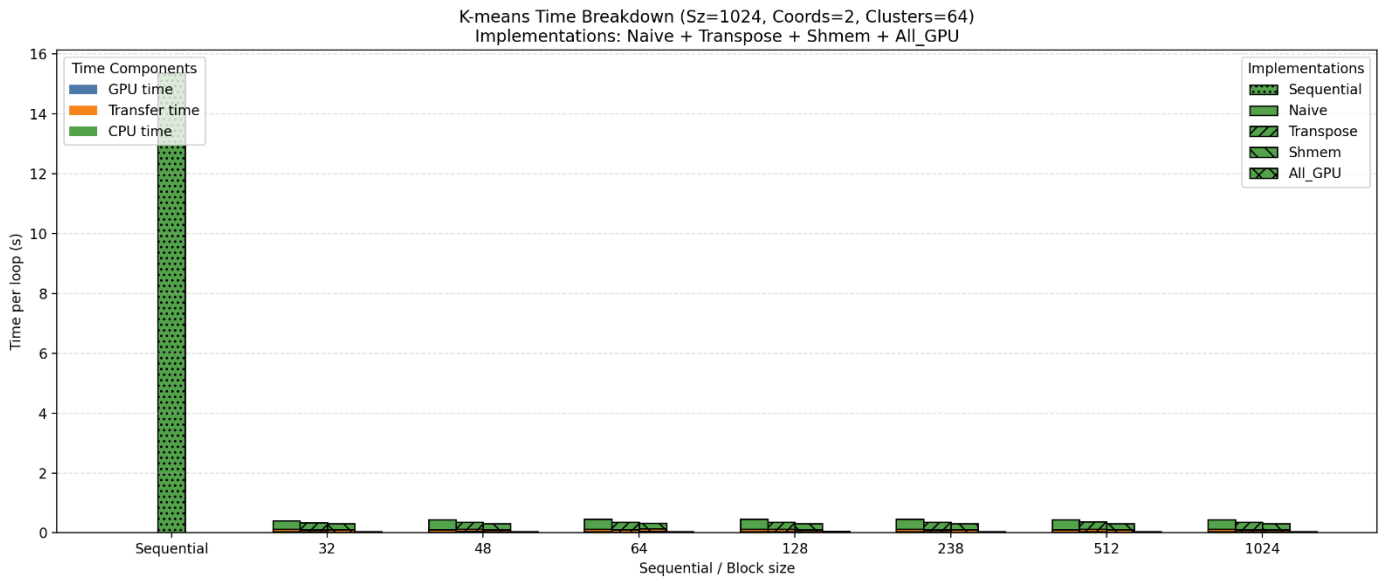
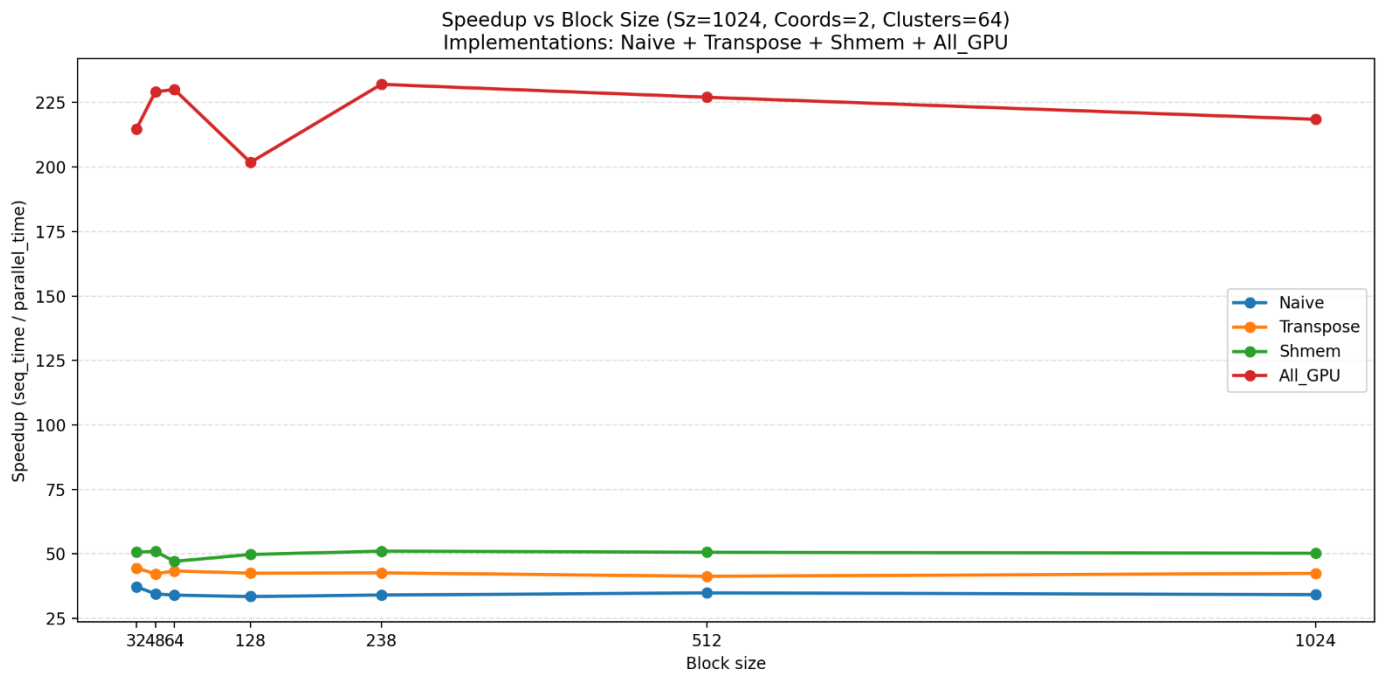
Η χρήση `atomics` εξασφαλίζει το σωστό αποτέλεσμα στα αθροίσματα, παρά το ταυτόχρονο `update` από πολλά `threads`. Ο απαιτούμενος καθολικός συγχρονισμός επιτυγχάνεται φυσικά με τη διάσπαση σε πολλαπλά `kernels` (τα `kernels` εκτελούνται σειριακά ως προς τη σειρά κλήσης τους), κάτι που αντικαθιστά την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

Z. 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).

■ Γενικά Συμπεράσματα

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