a5/cuda_kmeans_all_gpu.cu

```
#include <stdio.h>
#include <stdlib.h>

#include "kmeans.h"
#include "alloc.h"
#include "error.h"

#ifdef __CUDACC__
inline void checkCuda(cudaError_t e)
{
  if (e != cudaSuccess)
  {
    // cudaGetErrorString() isn't always very helpful. Look up the error
    // number in the cudaError enum in driver_types.h in the CUDA includes
    // directory for a better explanation.
    error("CUDA Error %d: %s\n", e, cudaGetErrorString(e));
  }
}

inline void checkLastCudaError()
{
  checkCuda(cudaGetLastError());
}
#endif

__device__ int get_tid()
{
  return blockIdx.x * blockDim.x + threadIdx.x;
}

/* square of Euclid distance between two multi-dimensional points using column-base format */
__host__ __device__ inline static double euclid_dist_2_transpose(int numCoords,
                                                                 int numObjs,
                                                                 int numClusters,
                                                                 double *objects,  // [numCoords][numObjs]
                                                                 double *clusters, // [numCoords][numClusters]
                                                                 int objectId,
                                                                 int clusterId)
{
  int i;
  double ans = 0.0;

  /* TODO: Calculate the euclid_dist of elem=objectId of objects from elem=clusterId from clusters, but for column-base format!!! */
  for (i = 0; i < numCoords; i++)
  {
    double objectVal = objects[i * numObjs + objectId];
    double clusterVal = clusters[i * numClusters + clusterId];

    double diff = objectVal - clusterVal;
    ans += diff * diff;
  }

  return (ans);
}

__global__ static void find_nearest_cluster(int numCoords,
                                            int numObjs,
                                            int numClusters,
                                            double *deviceObjects, //  [numCoords][numObjs]
                                                                   /*
                                                                   TODO: If you choose to do (some of) the new centroid calculation here, you will need some extra parameters here (from "update_centroids").
                                                                   */
                                            int *devicenewClusterSize,
                                            double *devicenewClusters, //  [numCoords][numClusters]
                                            double *deviceClusters,    //  [numCoords][numClusters]
                                            int *deviceMembership,     //  [numObjs]
                                            double *devdelta)
{
  extern __shared__ double shmemClusters[];
  // TODO: Copy deviceClusters to shmemClusters so they can be accessed faster.
  int tid_in_block = threadIdx.x;                      // Το ID του νήματος μέσα στο Block
  int block_size = blockDim.x;                         // Πόσα νήματα έχει το Block
  int total_cluster_doubles = numClusters * numCoords; // Συνολικά νούμερα προς αντιγραφή

  // Κάθε νήμα αντιγράφει όσα στοιχεία του αναλογούν (με βήμα block_size)
  for (int k = tid_in_block; k < total_cluster_doubles; k += block_size)
  {
    shmemClusters[k] = deviceClusters[k];
  }

  /* Συγχρονισμός (BARRIER) */

  __syncthreads();

  /* Get the global ID of the thread. */
  int tid = get_tid();

  /* TODO: Maybe something is missing here... should all threads run this? */
  if (tid < numObjs)
  {
    int index, i;
    double dist, min_dist;

    /* find the cluster id that has min distance to object */
    index = 0;
    /* TODO: call min_dist = euclid_dist_2(...) with correct objectId/clusterId using clusters in shmem*/

    min_dist = euclid_dist_2_transpose(numCoords, numObjs, numClusters,
                                       deviceObjects, shmemClusters,
                                       tid, index);

    for (i = 1; i < numClusters; i++)
    {
      dist = euclid_dist_2_transpose(numCoords, numObjs, numClusters,
                                     deviceObjects, shmemClusters,
                                     tid, i);

      /* no need square root */
      if (dist < min_dist)
      { /* find the min and its array index */
        min_dist = dist;
        index = i;
      }
    }

    if (deviceMembership[tid] != index)
    {
      /* TODO: Maybe something is missing here... is this write safe? */
      atomicAdd(devdelta, 1.0);
    }

    /* assign the deviceMembership to object objectId */
    deviceMembership[tid] = index;

    /* TODO: additional steps for calculating new centroids in GPU? */

    atomicAdd(&devicenewClusterSize[index], 1);

    for (int j = 0; j < numCoords; j++)
    {
      // Διαβάζουμε την τιμή του αντικειμένου (Coordinate j, Object tid)
      double objVal = deviceObjects[j * numObjs + tid];

      // Προσθέτουμε στο άθροισμα (Coordinate j, Cluster index)
      atomicAdd(&devicenewClusters[j * numClusters + index], objVal);
    }
  }
}

__global__ static void update_centroids(int numCoords,
                                        int numClusters,
                                        int *devicenewClusterSize, //  [numClusters]
                                        double *devicenewClusters, //  [numCoords][numClusters]
                                        double *deviceClusters)    //  [numCoords][numClusters])
{
  /* Κάθε νήμα αναλαμβάνει ΜΙΑ τιμή (double) του πίνακα clusters.
       Συνολικά νήματα = numCoords * numClusters
    */
  int tid = get_tid();
  int total_elements = numCoords * numClusters;

  if (tid < total_elements)
  {
    // Αποκωδικοποίηση του 1D tid σε 2D (Coordinate, Cluster)
    // Layout: [numCoords][numClusters] --> index = coord * numClusters + cluster
    int clusterId = tid % numClusters;
    // int coordId = tid / numClusters; // Δεν το χρειαζόμαστε άμεσα για τον υπολογισμό, αλλά για το reset

    int count = devicenewClusterSize[clusterId];

    // Υπολόγισε το νέο κέντρο (Average)
    if (count > 0)
    {
      double sum = devicenewClusters[tid];
      deviceClusters[tid] = sum / count;
    }
    // Αν count == 0, κρατάμε την παλιά τιμή (ή δεν κάνουμε τίποτα), όπως και στον CPU κώδικα

    //  RESET για τον επόμενο γύρο (Πολύ σημαντικό!)
    // Μηδενίζουμε το άθροισμα που μόλις χρησιμοποιήσαμε
    devicenewClusters[tid] = 0.0;
  }
}

//
//  ----------------------------------------
//  DATA LAYOUT
//
//  objects         [numObjs][numCoords]
//  clusters        [numClusters][numCoords]
//  dimObjects      [numCoords][numObjs]
//  dimClusters     [numCoords][numClusters]
//  newClusters     [numCoords][numClusters]
//  deviceObjects   [numCoords][numObjs]
//  deviceClusters  [numCoords][numClusters]
//  ----------------------------------------
//
/* return an array of cluster centers of size [numClusters][numCoords]       */
void kmeans_gpu(double *objects,     /* in: [numObjs][numCoords] */
                int numCoords,       /* no. features */
                int numObjs,         /* no. objects */
                int numClusters,     /* no. clusters */
                double threshold,    /* % objects change membership */
                long loop_threshold, /* maximum number of iterations */
                int *membership,     /* out: [numObjs] */
                double *clusters,    /* out: [numClusters][numCoords] */
                int blockSize)
{
  double timing = wtime(), timing_internal, timer_min = 1e42, timer_max = 0;
  double timing_gpu, timing_cpu, timing_transfers, transfers_time = 0.0, cpu_time = 0.0, gpu_time = 0.0;
  int loop_iterations = 0;
  int i, j, index, loop = 0;
  double delta = 0, *dev_delta_ptr; /* % of objects change their clusters */
  /* TODO: Copy me from transpose version*/
  double **dimObjects = (double **)calloc_2d(numCoords, numObjs, sizeof(double));      // calloc_2d(...) -> [numCoords][numObjs]
  double **dimClusters = (double **)calloc_2d(numCoords, numClusters, sizeof(double)); // calloc_2d(...) -> [numCoords][numClusters]
  double **newClusters = (double **)calloc_2d(numCoords, numClusters, sizeof(double));

  printf("\n|-----------Full-offload GPU Kmeans------------|\n\n");

  /* TODO: Copy me from transpose version*/
  for (i = 0; i < numObjs; i++)
  {
    for (j = 0; j < numCoords; j++)
    {
      dimObjects[j][i] = objects[i * numCoords + j];
    }
  }

  double *deviceObjects;
  double *deviceClusters, *devicenewClusters;
  int *deviceMembership;
  int *devicenewClusterSize; /* [numClusters]: no. objects assigned in each new cluster */

  /* pick first numClusters elements of objects[] as initial cluster centers*/
  for (i = 0; i < numCoords; i++)
  {
    for (j = 0; j < numClusters; j++)
    {
      dimClusters[i][j] = dimObjects[i][j];
    }
  }

  /* initialize membership[] */
  for (i = 0; i < numObjs; i++)
    membership[i] = -1;

  timing = wtime() - timing;
  printf("t_alloc: %lf ms\n\n", 1000 * timing);
  timing = wtime();
  const unsigned int numThreadsPerClusterBlock = (numObjs > blockSize) ? blockSize : numObjs;
  const unsigned int numClusterBlocks = (numObjs + numThreadsPerClusterBlock - 1) / numThreadsPerClusterBlock; /* TODO: Calculate Grid size, e.g. number of blocks. */

  /*    Define the shared memory needed per block.
      - BEWARE: We can overrun our shared memory here if there are too many
      clusters or too many coordinates!
      - This can lead to occupancy problems or even inability to run.
      - Your exercise implementation is not requested to account for that (e.g. always assume deviceClusters fit in shmemClusters */
  const unsigned int clusterBlockSharedDataSize = numClusters * numCoords * sizeof(double);

  cudaDeviceProp deviceProp;
  int deviceNum;
  cudaGetDevice(&deviceNum);
  cudaGetDeviceProperties(&deviceProp, deviceNum);

  if (clusterBlockSharedDataSize > deviceProp.sharedMemPerBlock)
  {
    error("Your CUDA hardware has insufficient block shared memory to hold all cluster centroids\n");
  }

  checkCuda(cudaMalloc(&deviceObjects, numObjs * numCoords * sizeof(double)));
  checkCuda(cudaMalloc(&deviceClusters, numClusters * numCoords * sizeof(double)));
  checkCuda(cudaMalloc(&devicenewClusters, numClusters * numCoords * sizeof(double)));
  checkCuda(cudaMalloc(&devicenewClusterSize, numClusters * sizeof(int)));
  checkCuda(cudaMalloc(&deviceMembership, numObjs * sizeof(int)));
  checkCuda(cudaMalloc(&dev_delta_ptr, sizeof(double)));

  timing = wtime() - timing;
  printf("t_alloc_gpu: %lf ms\n\n", 1000 * timing);
  timing = wtime();

  checkCuda(cudaMemcpy(deviceObjects, dimObjects[0],
                       numObjs * numCoords * sizeof(double), cudaMemcpyHostToDevice));
  checkCuda(cudaMemcpy(deviceMembership, membership,
                       numObjs * sizeof(int), cudaMemcpyHostToDevice));
  checkCuda(cudaMemcpy(deviceClusters, dimClusters[0],
                       numClusters * numCoords * sizeof(double), cudaMemcpyHostToDevice));
  checkCuda(cudaMemset(devicenewClusterSize, 0, numClusters * sizeof(int)));
  free(dimObjects[0]);

  timing = wtime() - timing;
  printf("t_get_gpu: %lf ms\n\n", 1000 * timing);
  timing = wtime();

  do
  {
    timing_internal = wtime();
    checkCuda(cudaMemset(dev_delta_ptr, 0, sizeof(double)));
    checkCuda(cudaMemset(devicenewClusterSize, 0, numClusters * sizeof(int)));
    timing_gpu = wtime();
    // printf("Launching find_nearest_cluster Kernel with grid_size = %d, block_size = %d, shared_mem = %d KB\n", numClusterBlocks, numThreadsPerClusterBlock, clusterBlockSharedDataSize/1000);
    //  TODO: change invocation if extra parameters needed
    find_nearest_cluster<<<numClusterBlocks, numThreadsPerClusterBlock, clusterBlockSharedDataSize>>>(numCoords, numObjs, numClusters,
                                                                                     deviceObjects, devicenewClusterSize, devicenewClusters, deviceClusters, deviceMembership, dev_delta_ptr);

    cudaDeviceSynchronize();
    checkLastCudaError();

    gpu_time += wtime() - timing_gpu;

    // printf("Kernels complete for itter %d, updating data in CPU\n", loop);

    timing_transfers = wtime();
    // TODO: Copy dev_delta_ptr to &delta
    checkCuda(cudaMemcpy(&delta, dev_delta_ptr, sizeof(double), cudaMemcpyDeviceToHost));
    transfers_time += wtime() - timing_transfers;

    const unsigned int update_centroids_block_sz = (numCoords * numClusters > blockSize) ? blockSize : numCoords * numClusters;         /* TODO: can use different blocksize here if deemed better */
    const unsigned int update_centroids_dim_sz = (numCoords * numClusters + update_centroids_block_sz - 1) / update_centroids_block_sz; /* TODO: calculate dim for "update_centroids" */
    timing_gpu = wtime();
    // TODO: use dim for "update_centroids" and fire it
    update_centroids<<<update_centroids_dim_sz, update_centroids_block_sz, 0>>>(numCoords, numClusters, devicenewClusterSize, devicenewClusters, deviceClusters);
    cudaDeviceSynchronize();
    checkLastCudaError();
    gpu_time += wtime() - timing_gpu;

    timing_cpu = wtime();
    delta /= numObjs;
    // printf("delta is %f - ", delta);
    loop++;
    // printf("completed loop %d\n", loop);
    cpu_time += wtime() - timing_cpu;

    timing_internal = wtime() - timing_internal;
    if (timing_internal < timer_min)
      timer_min = timing_internal;
    if (timing_internal > timer_max)
      timer_max = timing_internal;
  } while (delta > threshold && loop < loop_threshold);

  checkCuda(cudaMemcpy(membership, deviceMembership,
                       numObjs * sizeof(int), cudaMemcpyDeviceToHost));
  checkCuda(cudaMemcpy(dimClusters[0], deviceClusters,
                       numClusters * numCoords * sizeof(double), cudaMemcpyDeviceToHost));

  for (i = 0; i < numClusters; i++)
  {
    for (j = 0; j < numCoords; j++)
    {
      clusters[i * numCoords + j] = dimClusters[j][i];
    }
  }

  timing = wtime() - timing;
  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"
         "-> t_cpu_avg = %lf ms\n\t-> t_gpu_avg = %lf ms\n\t-> t_transfers_avg = %lf ms\n\n|-------------------------------------------|\n",
         loop, 1000 * timing, 1000 * timing / loop, 1000 * timer_min, 1000 * timer_max,
         1000 * cpu_time / loop, 1000 * gpu_time / loop, 1000 * transfers_time / loop);

  char outfile_name[1024] = {0};
  sprintf(outfile_name, "Execution_logs/silver1-V100_Sz-%lu_Coo-%d_Cl-%d.csv",
          numObjs * numCoords * sizeof(double) / (1024 * 1024), numCoords, numClusters);
  FILE *fp = fopen(outfile_name, "a+");
  if (!fp)
    error("Filename %s did not open succesfully, no logging performed\n", outfile_name);
  fprintf(fp, "%s,%d,%lf,%lf,%lf\n", "All_GPU", blockSize, timing / loop, timer_min, timer_max);
  fclose(fp);

  checkCuda(cudaFree(deviceObjects));
  checkCuda(cudaFree(deviceClusters));
  checkCuda(cudaFree(devicenewClusters));
  checkCuda(cudaFree(devicenewClusterSize));
  checkCuda(cudaFree(deviceMembership));

  return;
}

```