a5/cuda_kmeans_naive.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 */
__host__ __device__ inline static
double euclid_dist_2(int numCoords,
                     int numObjs,
                     int numClusters,
                     double *objects,     // [numObjs][numCoords]
                     double *clusters,    // [numClusters][numCoords]
                     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*/
  for (i = 0; i < numCoords; i++) {
    double objectVal = objects[objectId * numCoords + i];
    double clusterVal = clusters[clusterId * numCoords + i];

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

  return (ans);
}

__global__ static
```

```
void find_nearest_cluster(int numCoords,
                          int numObjs,
                          int numClusters,
                          double *objects,           //  [numObjs][numCoords]
                          double *deviceClusters,    //  [numClusters][numCoords]
                          int *deviceMembership,          //  [numObjs]
                          double *devdelta) {

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

  if (tid < numObjs) {
    int index, i;
    double dist, min_dist;

    /* find the cluster id that has min distance to object */
    index = 0;

    min_dist = euclid_dist_2(numCoords, numObjs, numClusters,
                             objects, deviceClusters,
                             tid, index);

    for (i = 1; i < numClusters; i++) {

      dist = euclid_dist_2(numCoords, numObjs, numClusters,
                           objects, deviceClusters,
                           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) {

      atomicAdd(devdelta, 1.0);
    }

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

//
//  ----------------------------------------
//  DATA LAYOUT
//
//  objects         [numObjs][numCoords]
//  clusters        [numClusters][numCoords]
//  newClusters     [numClusters][numCoords]
//  deviceObjects   [numObjs][numCoords]
//  deviceClusters  [numClusters][numCoords]
//  ----------------------------------------
```

```
//
/* 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;
  int *newClusterSize; /* [numClusters]: no. objects assigned in each
                                new cluster */
  double delta = 0, *dev_delta_ptr;          /* % of objects change their clusters */
  double **newClusters = (double **) calloc_2d(numClusters, numCoords, sizeof(double));

  double *deviceObjects;
  double *deviceClusters;
  int *deviceMembership;

  printf("\n|-------------Naive GPU Kmeans--------------|\n\n");


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

  /* need to initialize newClusterSize and newClusters[0] to all 0 */
  newClusterSize = (int *) calloc(numClusters, sizeof(int));
  assert(newClusterSize != NULL);

  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. */
  const unsigned int clusterBlockSharedDataSize = 0;

  checkCuda(cudaMalloc(&deviceObjects, numObjs * numCoords * sizeof(double)));
  checkCuda(cudaMalloc(&deviceClusters, numClusters * numCoords * sizeof(double)));
  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, objects,
                       numObjs * numCoords * sizeof(double), cudaMemcpyHostToDevice));
  checkCuda(cudaMemcpy(deviceMembership, membership,
                       numObjs * sizeof(int), cudaMemcpyHostToDevice));
  timing = wtime() - timing;
  printf("t_get_gpu: %lf ms\n\n", 1000 * timing);
  timing = wtime();

  do {
    timing_internal = wtime();

    /* GPU part: calculate new memberships */

    timing_transfers = wtime();

    checkCuda(cudaMemcpy(deviceClusters, clusters,
                 numClusters * numCoords * sizeof(double),
                 cudaMemcpyHostToDevice));

    transfers_time += wtime() - timing_transfers;

    checkCuda(cudaMemset(dev_delta_ptr, 0, sizeof(double)));

    //printf("Launching find_nearest_cluster Kernel with grid_size = %d, block_size = %d, shared_mem = %d KB\n", numClusterBlocks, numThreadsPerClusterBlock, clusterBlockSharedDataSize/1000);
    timing_gpu = wtime();
    find_nearest_cluster
    <<< numClusterBlocks, numThreadsPerClusterBlock, clusterBlockSharedDataSize >>>
            (numCoords, numObjs, numClusters,
             deviceObjects, 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();

    checkCuda(cudaMemcpy(membership, deviceMembership,
                     numObjs * sizeof(int),
                     cudaMemcpyDeviceToHost));

    checkCuda(cudaMemcpy(&delta, dev_delta_ptr,
                     sizeof(double),
                     cudaMemcpyDeviceToHost));

    transfers_time += wtime() - timing_transfers;

    /* CPU part: Update cluster centers*/
    timing_cpu = wtime();
    for (i = 0; i < numObjs; i++) {
      /* find the array index of nestest cluster center */
      index = membership[i];

      /* update new cluster centers : sum of objects located within */
      newClusterSize[index]++;
      for (j = 0; j < numCoords; j++)
        newClusters[index][j] += objects[i * numCoords + j];
    }

    /* average the sum and replace old cluster centers with newClusters */
    for (i = 0; i < numClusters; i++) {
      for (j = 0; j < numCoords; j++) {
        if (newClusterSize[i] > 0)
          clusters[i * numCoords + j] = newClusters[i][j] / newClusterSize[i];
        newClusters[i][j] = 0.0;   /* set back to 0 */
      }
      newClusterSize[i] = 0;   /* set back to 0 */
    }

    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);

  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", "Naive", blockSize, timing / loop, timer_min, timer_max);
  fclose(fp);
  checkCuda(cudaFree(deviceObjects));
  checkCuda(cudaFree(deviceClusters));
  checkCuda(cudaFree(deviceMembership));

  free(newClusters[0]);
  free(newClusters);
  free(newClusterSize);

  return;
```

```
}


```