1. (50) The OpenMP code below implements what might be considered a variant of k-means clustering. It is assumed that once a data point is placed into a cluster, it stays with that cluster even as new data points are added. The number of clusters is fixed, but the centroids and counts of cluster members are updated each time a new data point is acquired.

Assume that new data arrives in clumps. The code below takes a clump of new data points and updates the cluster centroids and counts. (Which must be updated once for each new data point.)

Globals:

- $k$, the number of clusters
- $p$, the dimensionality of the space
- $n$, the total number of data points recorded in clusters (will grow by the amount of $n_{new}$ below)
- centroids, a matrix of the current centroids, $k$ rows, $p$ columns
- clstrcounts, an array, length $k$, recording how many data points are in each cluster
- grps, an array listing group membership, so that for example grps[88] = 3 means that data point number 88 is in cluster 3; length is assumed as large as $n$ will ever get
- $n_{new}$, the number of new data points
- clump, matrix of the new data, with $n_{new}$ rows, $p$ columns

Fill in the blanks, and add any lines necessary. For the latter action, write something like, “Place the following code between lines 8 and 9.” Do NOT delete or change lines.

```c
#pragma omp parallel
{
    int i,j,grpnum;
    for (i = 0; i < nnew; i++) {
        grpnum = closest(i);
        for (j = 0; j < p; j++) {
            tmp = centroid[grpnum][j] + clump[i][j];
            tmp /= clump[i][j];
        }
    }
}
```

2. (50) The CUDA code below computes the discrete cosine transform of an image, p.135. Assume there is only one block, with that block consisting of $n$ rows and $m$ columns of threads. Each thread handles a single pixel, making a local copy. Shared memory is not used. The arguments to dct are:

- $n$, the number of rows in the image and the transform
- $m$, the number of columns in the image and the transform
- $dx$, the image data on the device, $n$ rows, $m$ columns
- $dd$, the transform data on the device, $n$ rows, $m$ columns, initially all 0.0

```c
__global__ void dct(float *dx, int n, int m, float *dd)
{
    int j,k;
    float pi = 3.14;
    for (u = 0; u < n; u++)
        for (v = 0; v < m; v++) {
        }
}
```
float y(int q) {
    if (q == 0) return 0.71;
    else return 1.0;
}

Solutions:

1.
#pragma omp parallel
{
    int i,j,grpnum;
    #pragma omp for
    for (i = 0; i < nnew; i++) {
        grpnum = closest(i);
        #pragma omp critical
        { for (j = 0; j < p; j++) {
            tmp = clstrcounts[grpnum] * centroid[grpnum][j];
            tmp += clump[i][j];
            tmp /= (clstrcounts[grpnum]+1);
            centroids[grpnum][j] = tmp;
        }
        clstrcounts[grpnum]++;
        n++;
        grps[n] = grpnum;
    }
}

2.
__global__ void dct(float *dx, int n, int m, float *dd)
{
    int u,v;
    int j = threadIdx.x;
    int k = threadIdx.y;
    float pi = 3.14, myx, tmp;
    myx = dx[n*j+k];
    for (u = 0; j < n; u++)
        for (v = 0; k < m; v++) {
            tmp = myx * cos((2*j+1)*u*pi/(2*n)) + cos((2*k+1)*v*pi/(2*m));
            tmp /= y(u) * y(v) * 2 / sqrt(m*n);
            atomicAdd(&dd[n*u+v],tmp);
        }
}

float y(int q) {
    if (q == 0) return 0.71;
    else return 1.0;
}