Name: _____

Directions: **Work only on this sheet** (on both sides, if needed); do not turn in any supplementary sheets of paper. There is actually plenty of room for your answers, as long as you organize yourself BEFORE starting writing.

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 **nnew** 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
- **nnew**, the number of new data points
- **clump**, matrix of the new data, with **nnew** 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.

```
1  #pragma omp parallel
2  {
3     int i,j,grpnum;
4     for (i = 0; i < nnew; i++) {
5        // function closest() (not shown), finds the index of
6        // the closest centroid to the new data point i
7        grpnum = closest(i);
8        for (j = 0; j < p; j++) {
9           tmp =  centroid[grpnum][j] _____________________________;
10          tmp += clump[i][j];
11          tmp /= _____________________________
12       }
13    }
14 }
```

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

```
1  __global__ void dct(float *dx,int n, int m, float *dd)
2  {  int j,k;
3     float pi = 3.14;
4     for (u = 0; u < n; u++)
5        for (v = 0; v < m; v++) {
6        }
7  }
```

```

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++) {
      // function closest() (not shown), finds the index of
      // the closest centroid to the new data point 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;
}
```