Absolutely! Let's walk through both CUDA kernel functions: `assign_clusters` and `update_centroids`, **line by line**, to explain exactly what each piece does and why it's needed for the **K-Means clustering algorithm**.

🔹 `assign_clusters` — Assigns each data point to the nearest centroid

```
__global__ void assign_clusters(float *data, float *centroids, int *labels)
```

- This is a **CUDA kernel** that runs in parallel.
- Each thread is responsible for one data point (from the dataset `data`).
- `labels` will store which cluster (0, 1, or 2) each point belongs to.

```
int idx = threadIdx.x + blockIdx.x * blockDim.x;
```

- `idx` is the **global index** of the thread. It maps each thread to a unique data point.
- For example, if we launch 256 threads per block, the second block's first thread will have `idx = 256`.

```
if (idx < N)
```

- Ensures that the thread only processes valid indices (up to `N = 150`).

```
float min_dist = FLT_MAX;
 int closest = -1;
```

- `min_dist`: initialized to the largest possible float value (to find the minimum distance).
- `closest`: will eventually hold the index of the closest centroid.

```
for (int k = 0; k < K; k++)
```

- Loop over each of the `K` centroids (3 clusters in this case).

```
float dist = 0.0f;
```

```
for (int d = 0; d < D; d++) {
    float diff = data[idx * D + d] - centroids[k * D + d];
    dist += diff * diff;
}
```

- Calculate the **Euclidean distance squared** between the current data point (`data[idx]`) and the `k-th` centroid.
- `data` and `centroids` are 1D arrays, so we use `idx * D + d` to access features correctly.

```
if (dist < min_dist) {
    min_dist = dist;
    closest = k;
}
```

- Update `min_dist` and `closest` if we find a closer centroid.

```
labels[idx] = closest;
```

- After finding the closest centroid, we assign its index to the `labels` array for that data point.

🔹 `update_centroids` — Recomputes centroids based on cluster assignments

```
__global__ void update_centroids(float *data, float *centroids, int *labels)
```

- This kernel updates the `centroids` array based on the average of the assigned data points.

```
__shared__ float centroid_sums[K][D];
__shared__ int counts[K];
```

- These are **shared memory arrays**, accessible by all threads in the block.
- `centroid_sums` accumulates the sum of all vectors for each cluster.
- `counts` keeps track of how many points were assigned to each cluster.

```
int tid = threadIdx.x + blockIdx.x * blockDim.x;
```

- Same as before: global thread index.

Step 1: Initialization by the first thread in the block

```
if (threadIdx.x == 0) {
    for (int k = 0; k < K; k++) {
        counts[k] = 0;
        for (int d = 0; d < D; d++) {
            centroid_sums[k][d] = 0.0f;
        }
    }
}
__syncthreads();
```

- Thread 0 initializes the shared memory arrays.
- `__syncthreads()` ensures **all threads wait** until initialization is complete.

Step 2: Accumulate sums of assigned data points

```
if (tid < N) {
    int label = labels[tid];
    for (int d = 0; d < D; d++) {
        atomicAdd(&centroid_sums[label][d], data[tid * D + d]);
    }
    atomicAdd(&counts[label], 1);
}
```

- Each thread (representing a data point) adds its values to the centroid it was assigned to.
- We use `atomicAdd()` to **safely update** shared memory from multiple threads (avoids race conditions).

Step 3: Compute the mean to get new centroids

```
__syncthreads();
```

- Wait for all threads to finish updating the sums.

```
if (threadIdx.x == 0) {
    for (int k = 0; k < K; k++) {
```

```
        if (counts[k] > 0) {
            for (int d = 0; d < D; d++) {
                centroids[k * D + d] = centroid_sums[k][d] / counts[k];
            }
        }
    }
}
```

- Only thread 0 now computes the **average** to update the actual centroids.
- For each cluster `k`, we divide the accumulated sum by the number of points assigned to that cluster.

✅ Summary

Kernel Function	Responsibility	Key CUDA Feature Used
`assign_clusters`	Assign each data point to the closest centroid	Thread-wise parallelism
`update_centroids`	Compute new centroid positions	Shared memory + atomicAdd

Would you like me to visualize the flow or help you with enhancements like multi-block/grid support or dynamic cluster count (K)?

Great question! Let's break down how the formula **π / 4** comes into play when estimating π using the **Monte Carlo method**.

🎯 The Idea Behind the π/4 Formula

We use **geometry** to estimate π by comparing the area of a **quarter circle** with the area of a **square** that contains it.

🟦 The Square

We consider a **unit square**:

- Its sides go from 0 to 1 on both axes.

- So its **area is**:
 Asquare=1×1=1A_{\text{square}} = 1 \times 1 = 1

🟠 The Quarter Circle

We inscribe a **quarter of a unit circle** inside this square:

- It's part of a circle of **radius 1** centered at the origin `(0,0)`.

- The full area of a circle is:
 Acircle=πr2=π⋅12=πA_{\text{circle}} = \pi r^2 = \pi \cdot 1^2 = \pi
- Since we're only using a **quarter** of the circle (from x = 0 to 1 and y = 0 to 1), its area is:
 Aquarter-circle=14⋅π=π4A_{\text{quarter-circle}} = \frac{1}{4} \cdot \pi = \frac{\pi}{4}

🧪 Monte Carlo Estimation

Now we generate **random points (x, y)** inside the square, where `x` and `y` are random numbers between 0 and 1.

We check if a point lies inside the quarter circle using:

x2+y2≤1x^2 + y^2 \leq 1

Let:

- `N` = total number of random points

- `M` = number of points inside the quarter circle

Because the points are uniformly distributed, the **ratio** of points inside the quarter circle to the total points should approach the ratio of their areas:

$$\frac{M}{N} \approx \frac{\text{Area of quarter-circle}}{\text{Area of square}} = \frac{\pi}{4}$$

So,

$$\boxed{\pi \approx 4 \cdot \frac{M}{N}}$$

That's where the formula comes from!

✅ Summary

Concept	Value
Area of full circle	$\pi r^2 = \pi$
Area of quarter circle	$\frac{\pi}{4}$
Area of square	1
Ratio (quarter circle to square)	$\frac{\pi}{4}$
Estimate π	$\pi \approx 4 \cdot \frac{\text{Points in circle}}{\text{Total points}}$

Would you like a visual diagram for this?