Sure! This code implements the **K-Means clustering algorithm** using **CUDA** to leverage the GPU for parallel processing. The dataset used here is assumed to be the **Iris dataset**, which has 150 samples with 4 features each, grouped into 3 clusters.

🧠 Concept: K-Means Clustering

K-Means is an **unsupervised machine learning algorithm** used to group data into `K` clusters based on feature similarity.

Steps of K-Means:

1. **Initialize** `K` centroids randomly from the dataset.
2. **Assign** each data point to the **nearest centroid** (based on Euclidean distance).
3. **Update** the centroids by calculating the **mean** of all points assigned to each centroid.
4. **Repeat** steps 2–3 until convergence (or for a fixed number of iterations).

🧠 CUDA Parallelization

- We use **GPU kernels** to speed up:
  - Assigning clusters
  - Updating centroids

📦 Code Breakdown

Headers and Macros

```
#define N 150       // Number of data points
#define D 4         // Number of features (dimensions)
#define K 3         // Number of clusters
#define MAX_ITER 100
```

🧮 Device Kernels

1. `assign_clusters`

Each thread processes one data point, calculates its distance to each centroid, and assigns it to the closest one.

```
__global__ void assign_clusters(float *data, float *centroids, int *labels) {
    int idx = threadIdx.x + blockIdx.x * blockDim.x;
    ...
}
```

2. `update_centroids`

This updates centroids by averaging all points in each cluster.

- Shared memory is used to accumulate sums.
- `atomicAdd` ensures correctness when multiple threads update shared data.

```
__global__ void update_centroids(float *data, float *centroids, int *labels) {
    __shared__ float centroid_sums[K][D];
    __shared__ int counts[K];
    ...
}
```

📥 Data Loading: `load_data()`

Loads the first 150 rows of the `IRIS.csv` file and randomly initializes centroids.

```
void load_data(const std::string &filename) {
    ...
    std::ifstream file(filename);
    ...
    for (int k = 0; k < K; ++k) {
        int rand_idx = rand() % N;
        ...
    }
}
```

🚀 Main Function: `int main()`

Memory Allocation

```
cudaMalloc(&d_data, N * D * sizeof(float));
cudaMalloc(&d_centroids, K * D * sizeof(float));
cudaMalloc(&d_labels, N * sizeof(int));
```

Data Copy to GPU

```
cudaMemcpy(d_data, data, N * D * sizeof(float), cudaMemcpyHostToDevice);
```

cudaMemcpy(d_centroids, centroids, K * D * sizeof(float), cudaMemcpyHostToDevice);

K-Means Iterations

```
for (int i = 0; i < MAX_ITER; i++) {
    assign_clusters<<<1, 256>>>(d_data, d_centroids, d_labels);
    update_centroids<<<1, 256>>>(d_data, d_centroids, d_labels);
    cudaDeviceSynchronize();
}
```

Execution Time Calculation

```
std::clock_t start = std::clock();
...
std::clock_t end = std::clock();
double elapsed_time = double(end - start) / CLOCKS_PER_SEC * 1000;
```

Output Final Centroids

```
std::cout << "Final centroids:\n";
...
```

Cleanup

```
delete[] data;
delete[] centroids;
delete[] labels;
cudaFree(d_data);
cudaFree(d_centroids);
cudaFree(d_labels);
```

📈 Summary of Output

- **Final centroids** after clustering
- **Execution time** of the algorithm on GPU

🔍 Potential Improvements

- Use **dynamic grid size** to support larger datasets.
- Use **CUDA streams** for overlapping computation and memory copy.
- Add **convergence checking** instead of fixed iteration.

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)?