Outline:

* My program uses a BodiesProblem struct which includes all input data as given in the file. Point and Cluster are also structs with additional Points struct for better performance.
* General types are used in the program such as **real\_t** in order to switch easily between Double and Float and **size\_t** to avoid overflow.
* **KMeans.c** file includes initialization of cluster centers and **kMeans** function which initializes different buffers and has a loop which iterates **LIMIT** times: first iteration performs initialization of cluster centers and further iterations recalculate the cluster centers using OpenMP.
* The general flow of the program: File is read via argv[] (input and output paths are required along with exe file path and "-v=CUDA" which is my final version of the program), constants are saved in BodiesProblem variable, buffers are allocated and then a for loop of **t** iterates over time interval **[0, T]** and runs as long as qualityMeasure is higher than **QM.** Inside the loop **KMeans** function is called (inner loop of **LIMIT**), the coordinates of points are change due to velocities and quality measure is calculated (serialy).

Parallel implementations:

Inside **KMeans.c** 🡪 **recalculateClusterCentersOmp** function has a loop which is parallelized using **#pragma omp parallel for schedule(guided).** This function matches points to clusters by distances and so it better be parallelized.I chose (guided) scheduling policy in order to deal well with large scales of N. In this policy, the scheduling begins with big chunks (number of iterations for thread) but then adjusts to smaller chunk sizes if the workload is imbalanced.

Inside **Kmeans.c** 🡪 **findClosestClustersCuda** function uses GPU via **findClosestClustersKernel**. This function indicates if points have to move from their previous cluster.

At the beginning, it allocates GPU buffers and copies points into GPU 🡪 this is done inside an extern "C" block (This is done to avoid linker problems with C++ compiler. Note that cudaMalloc, cudaMemCpy and cudaFree are called with the help of "check" macro which is written in C++ style). Points will not be changed until a next call of KMeans

In the loop, it copies centers into GPU, runs a kernel, waits until it's done, copies a clusterization from GPU

There are 2 kernels: cached and not cached

**Not cached** **kernel** is straight, algo is the same as for OpenMP. The only problem is when reading centers. **Threads in a warp** read the same center simultaneously but the ideal state is when adjacent threads read adjacent values.

In **cached kernel** every **block** reads centers into its **shared** memory, it gives a speedup about 1.5-3x compared to not cached version (but shared memory has a limit: depends on macro named CUDA\_USED\_SHARED\_MEMORY\_SIZE).

Note: Very big values will downgrade the performance, because shared memory is also an automatic cache for other readings

At the end, it frees allocated GPU buffers