1. There are 3 files in the folder: **working_kdf.c** , **working_multi_kdf.c** and **epc_kdf_gpu.cu** which contain sequential, mutithreaded and GPU code respectively.
2. **working_kdf.c** and **epc_kdf_gpu.cu** include **"auc.h**" where as **working_multi_kdf.c** include "**auc_multithread.h**".
3. Some macros:
   1. NUM_REQUESTS(common to all files): number of user requests.
   2. NUM_THREADS (in working_multi_kdf.c): Number of parallel CPU threads.
   3. THREADS_PER_BLOCK (in epc_kdf_gpu.cu): number of threads in one block of CUDA. Block is analogous to OpenCL's "**workgroup**". So basically its number of "**workitems"** in a workgroup (in case of OpenCL).
4. How to compile:
   1. gcc working_kdf.c -lrt -o seq_kdf
   2. gcc working_multi_kdf.c -lrt -pthread -o multi_kdf
   3. nvcc -w epc_kdf_gpu.cu -o gpu_kdf
5. All the programs prints the runtime at the end in milliseconds. There is a loop (commented) at the end of the program which prints all the values.
6. Some structures:
   1. **typedef struct {**

      ```
        uint8_t opc_in[16];//128 bit
        uint64_t imsi_in;
        uint8_t key_in[16]; //128 bit
        uint8_t plmn_in[3];
        uint8_t sqn_in[6];
        auc_vector_t* auc_vector_in;
      } input_request;
      ```

   2. **typedef struct {**

      ```
        uint8_t rand[16];
        uint8_t rand_new;
        uint8_t xres[8];
        uint8_t autn[16];
        uint8_t kasme[32];
      } auc_vector_t;
      ```

      The above two structures are used to specify the input to programs.
   3. The functions **void init_input(input_request *input,int offset)** and **void**

init_auc_vector(auc_vector_t *auc_vector_in) are used to initialize these structures with some values. ***(values for testing purpose only wont be part of final implementation)***

4. **generate_vector(...)** function is the function which does the necessary computation. Whatever computation it does and be treated as a black box. The output of the function is store in the ***input_request*** and ***auc_vector_t*** structures. There is no need to pass separate output structure.
5. In case of multi-threaded program (**working_multi_kdf.c**) the data (NUM_REQUESTS) is divided equally among (NUM_THREADS) and each thread calls **generate_vector(...)** on its data-set.
6. In case of GPU code, threads equal to NUM_REQUESTS are created and each thread get one data unit. Then each thread executes **generate_vector(..)** on its data.
7. **Some points regarding GPU code (epc_kdf_gpu.cu):**
    1. Function defined with __global__ and __device__ keyword are GPU functions.
    2. Data variable defined with __constant__ and __device__ are GPU variables.
    3. The __device__ function, __contants__ variables and __device__variables cannot be accessed from CPU codes.
    4. Only __global__ function can be called from CPU code. In our case **generate_vector_gpu(...)** is only __global__ function. The other GPU funcitons are __device__ functions.
    5. cudaMalloc(..) is used to allocate memory on GPU.
    6. cudaMemcpy(..) is used to transfer data to and from CPU and GPU. The last parameter specifies the direction of transfer.
    7. cudaMemcpyToSymbol(..) is used to copy constants from CPU to GPU.
    8. **__global__ void generate_vector_gpu(input_request input[], auc_vector_t auc_vector_in [],int num_requests)** function has two structures as input and total number of requests.
    9. CudaEventCreate(..), cudaEventRecord(..), and cudaEventSynchronize(...) are used to create, record and synchronize events. These events are then used to get execution time of the GPU function. You can record an event before GPU kernel exection and record another event after kernel execution. You can measure the time elaspsed between those events by **cudaEventElapsedTime(..).** When you are converting this code to OpenCL these calls needs to be replaced by corresponding OpenCL calls which measures the runtime of a GPU kernel function.
    10. cudaFree(..) is used to free the GPU memory allocated via the cudaMalloc(..) call.