```
#include <stdio.h>
#include <stdlib.h>

/*
 * The maximum and minimum integer values of the range of printable characters
 * in the ASCII alphabet. Used by encrypt kernel to wrap adjust values to that
 * ciphertext is always printable.
 */
#define MAX_PRINTABLE 64
#define MIN_PRINTABLE 128
#define NUM_ALPHA MAX_PRINTABLE - MIN_PRINTABLE

__global__ void encrypt(unsigned int *text, unsigned int *key, unsigned int *result) {
 /* Calculate the current index */
    const unsigned int idx = (blockIdx.x * blockDim.x) + threadIdx.x;

     /*
      * Adjust value of text and key to be based at 0
      * Printable ASCII starts at MIN_PRINTABLE, but 0 start is easier to work with
      */
    char adjusted_text = text[idx] - MIN_PRINTABLE;
    char adjusted_key = key[idx] - MIN_PRINTABLE;

     /* The cipher character is the text char added to the key char modulo the number
of chars in the alphabet*/
    char cipherchar = (adjusted_text + adjusted_key) % (NUM_ALPHA);

     /* adjust back to normal ascii (starting at MIN_PRINTABLE) and save to result */
    result[idx] = (unsigned int) cipherchar + MIN_PRINTABLE ;
}

void pageable_transfer_execution(int array_size, int threads_per_block, FILE *input_fp
, FILE *key_fp) {
    /* Calculate the size of the array*/
     int array_size_in_bytes = (sizeof(unsigned int) * (array_size)); int i = 0;

     unsigned int *cpu_text = (unsigned int *) malloc(array_size_in_bytes);
```

```
    unsigned int *cpu_key = (unsigned int *) malloc(array_size_in_bytes);
    unsigned int *cpu_result = (unsigned int *) malloc(array_size_in_bytes);

    /* Read characters from the input and key files into the text and key arrays resp
ectively */
    // Code left out for brevity sake

    cudaMalloc((void **)&gpu_text, array_size_in_bytes);
    cudaMalloc((void **)&gpu_key, array_size_in_bytes);
    cudaMalloc((void **)&gpu_result, array_size_in_bytes);

    /* Copy the CPU memory to the GPU memory */
    cudaMemcpy( gpu_text, cpu_text, array_size_in_bytes, cudaMemcpyHostToDevice);
    cudaMemcpy( gpu_key, cpu_key, array_size_in_bytes, cudaMemcpyHostToDevice);

    /* Designate the number of blocks and threads */
    const unsigned int num_blocks = array_size/threads_per_block;
    const unsigned int num_threads = array_size/num_blocks;

    /* Execute the encryption kernel and keep track of start and end time for duratio
n */
    float duration = 0;
    cudaEvent_t start_time = get_time();

    encrypt<<<num_blocks, num_threads>>>(gpu_text, gpu_key, gpu_result);

    cudaEvent_t end_time = get_time();
    cudaEventSynchronize(end_time);
    cudaEventElapsedTime(&duration, start_time, end_time);

    /* Copy the changed GPU memory back to the CPU */
    cudaMemcpy( cpu_result, gpu_result, array_size_in_bytes, cudaMemcpyDeviceToHost);

    printf("Pageable Transfer- Duration: %fmsn\n", duration);
    print_encryption_results(cpu_text, cpu_key, cpu_result, array_size);
```

```
    /* Free the GPU memory */
    // INSERT CODE HERE
    cudaFree(gpu_text);
    cudaFree(gpu_key);
    cudaFree(gpu_result);

    /* Free the CPU memory */
    // INSERT CODE HERE
    free(cpu_text);
    free(cpu_key);
    free(gpu_result);

}

void pinned_transfer_execution(int array_size, int threads_per_block, FILE *input_fp,
FILE *key_fp) { // Code left out for brevity sake

    //pin it
    cudaMallocHost((void **)&cpu_text_pinned, array_size_in_bytes);
    cudaMallocHost((void **)&cpu_key_pinned, array_size_in_bytes);
    cudaMallocHost((void **)&cpu_result_pinned, array_size_in_bytes);

    /* Copy the memory over */
    // INSERT CODE HERE
    // Read data from files into pinned buffers
    read(input_fp, cpu_text_pinned, array_size_in_bytes);
    read(key_fp, cpu_key_pinned, array_size_in_bytes);

    /* Declare and allocate pointers for GPU based parameters */
    unsigned int *gpu_text;
    unsigned int *gpu_key;
    unsigned int *gpu_result;

    cudaMalloc((void **)&gpu_text, array_size_in_bytes);
    cudaMalloc((void **)&gpu_key, array_size_in_bytes);
    cudaMalloc((void **)&gpu_result, array_size_in_bytes);
```

```
    /* Copy the CPU memory to the GPU memory */
    cudaMemcpy( gpu_text, cpu_text_pinned, array_size_in_bytes, cudaMemcpyHostToDevice
);
    cudaMemcpy( gpu_key, cpu_key_pinned, array_size_in_bytes, cudaMemcpyHostToDevice);

     /* Designate the number of blocks and threads */
    const unsigned int num_blocks = array_size/threads_per_block;
    const unsigned int num_threads = array_size/num_blocks;

     /* Execute the encryption kernel and keep track of start and end time for duratio
n */
    float duration = 0;
    cudaEvent_t start_time = get_time();

     encrypt<<<num_blocks, num_threads>>>(gpu_text, gpu_key, gpu_result);

     cudaEvent_t end_time = get_time();
     cudaEventSynchronize(end_time);
     cudaEventElapsedTime(&duration, start_time, end_time);

     /* Copy the changed GPU memory back to the CPU */
     cudaMemcpy( cpu_result_pinned, gpu_result, array_size_in_bytes, cudaMemcpyDeviceT
oHost);

     printf("Pinned Transfer- Duration: %fmsn\n", duration);
     print_encryption_results(cpu_text_pinned, cpu_key_pinned, cpu_result_pinned, arra
y_size);

     /* Free the GPU memory */
    cudaFree(gpu_text);
    cudaFree(gpu_key);
    cudaFree(gpu_result);

     /* Free the pinned CPU memory */
    cudaFreeHost(cpu_text_pinned);
    cudaFreeHost(cpu_key_pinned);
```

```
    cudaFreeHost(cpu_result_pinned);


     /* Free the pageable CPU memory */

    // INSERT CODE HERE

    // Use the CUDA library call to free up pinned memory

    cudaFreeHost(cpu_text_pinned);

    cudaFreeHost(cpu_key_pinned);

    cudaFreeHost(cpu_result_pinned);

}


/** * Prints the correct usage of this file * @name is the name of the executable (arg
v[0]) */

void print_usage(char *name) {

    printf("Usage: %s <total_num_threads> <threads_per_block> <input_file> <key_file>\
n", name);

}


/**

  * Performs simple setup functions before calling the pageable_transfer_execution()

  * function. * Makes sure the files are valid, handles opening and closing of file po
inters.

  */ void pageable_transfer(int num_threads, int threads_per_block, char *input_file,
char *key_file) {


    // Code left out for brevity sake


    /* Perform the pageable transfer */

    pageable_transfer_execution(num_threads, threads_per_block, input_fp, key_fp);


    fclose(input_fp); fclose(key_fp);

}


/**

  * Performs setup functions before calling the pageable_transfer_execution()

  * function.

  * Makes sure the files are valid, handles opening and closing of file pointers.

  */ void pinned_transfer(int num_threads, int threads_per_block, char *input_file, ch
ar *key_file) {
```

```
    // Code left out for brevity sake

    /* Perform the pageable transfer */
    pinned_transfer_execution(num_threads, threads_per_block, input_fp, key_fp);

    fclose(input_fp); fclose(key_fp);
}

    /**
      * Entry point for excution. Checks command line arguments and
      * opens input files, then passes execution to subordinate main_sub()
      */
int main(int argc, char *argv[]) {
    /* Check the number of arguments, print usage if wrong
    */
    if(argc != 5) {
        printf("Error: Incorrect number of command line arguments\n");
        print_usage(argv[0]); exit(-1);
    }

    /* Check the values for num_threads and threads_per_block */
    int num_threads = atoi(argv[1]);
    int threads_per_block = atoi(argv[2]);
    if(num_threads <= 0 || threads_per_block <= 0) {
        printf("Error: num_threads and threads_per_block must be integer > 0");
        print_usage(argv[0]); exit(-1);
    }

    if(threads_per_block > num_threads) {
        printf("Error: threads per block is greater than number of threads\n");
        print_usage(argv[0]);
        exit(-1);
    }

    printf("\n");
```

```
    /* Perform the pageable transfer */
    pageable_transfer(num_threads, threads_per_block, argv[3], argv[4]);

    printf("------------------------------------------------------------------\n");

    /* Perform the pinned transfer */
    pinned_transfer(num_threads, threads_per_block, argv[3], argv[4]);

    return EXIT_SUCCESS;
}
```