

```

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>
#include <iostream>
#include "timer.h"
#include <ctype.h>
using namespace std;

```

/* Utility function, use to do error checking.

Use this function like this:

```
checkCudaCall(cudaMalloc((void **) &deviceRGB, imgS * sizeof(color_t)));
```

And to check the result of a kernel invocation:

```

checkCudaCall(cudaGetLastError());
*/
static void checkCudaCall(cudaError_t result) {
    if (result != cudaSuccess) {
        cerr << "cuda error: " << cudaGetErrorString(result) << endl;
        exit(1);
    }
}

__global__ void encryptKernel(char* deviceDataIn, char* deviceDataOut)
{
    const int key = 1;
    int dummy;
    unsigned index = blockIdx.x * blockDim.x + threadIdx.x;
    for(index = 0; deviceDataIn[index] != '\0'; ++index)
    {
        deviceDataOut[index] = deviceDataIn[index] + key;
    }
}

__global__ void decryptKernel(char* deviceDataIn, char* deviceDataOut) {
    const int key = 1;
    unsigned index = blockIdx.x * blockDim.x + threadIdx.x;
    for(index = 0; deviceDataIn[index] != '\0'; ++index)
    {
        deviceDataOut[index] = deviceDataIn[index] - key;
    }
}

```

```
}
```

```
int fileSize() {  
    int size;  
  
    ifstream file ("original.data", ios::in|ios::binary|ios::ate);  
    if (file.is_open())  
    {  
        size = file.tellg();  
        file.close();  
    }  
    else {  
        cout << "Unable to open file";  
        size = -1;  
    }  
    return size;  
}
```

```
int readData(char *fileName, char *data) {  
  
    streampos size;  
  
    ifstream file (fileName, ios::in|ios::binary|ios::ate);  
    if (file.is_open())  
    {  
        size = file.tellg();  
        file.seekg (0, ios::beg);  
        file.read (data, size);  
        file.close();  
  
        cout << "The entire file content is in memory." << endl;  
    }  
    else cout << "Unable to open file" << endl;  
    return 0;  
}
```

```
int writeData(int size, char *fileName, char *data) {  
    ofstream file (fileName, ios::out|ios::binary|ios::trunc);  
    if (file.is_open())  
    {  
        file.write (data, size);  
        file.close();  
  
        cout << "The entire file content was written to file." << endl;
```

```

    return 0;
}
else cout << "Unable to open file";

return -1;
}
int EncryptSeq (int n, char* data_in, char* data_out)
{
    int i;
    int key = 1;
    timer sequentialTime = timer("Sequential encryption");

    sequentialTime.start();
    for(i=0; (i<n && data_in[i] != '\0'); ++i)
    {
        data_out[i] = data_in[i] + key;
    }
    sequentialTime.stop();
    cout << fixed << setprecision(6);
    cout << "Encryption (sequential): \t\t" << sequentialTime.getElapsed() << "
seconds." << endl;

    return 0;
}

int DecryptSeq (int n, char* data_in, char* data_out)
{
    int i;
    int key = 1;
    timer sequentialTime = timer("Sequential decryption");

    sequentialTime.start();
    for(i = 0; data_in[i] != '\0'; i++)
    {
        data_out[i] = data_in[i] - key;
    }

    sequentialTime.stop();
    cout << fixed << setprecision(6);
    cout << "Decryption (sequential): \t\t" << sequentialTime.getElapsed() << "
seconds." << endl;

    return 0;
}

```

```

int EncryptCuda (int n, char* data_in, char* data_out) {
    int threadBlockSize = 512;

    // allocate the vectors on the GPU
    char* deviceDataIn = NULL;
    checkCudaCall(cudaMalloc((void **) &deviceDataIn, n * sizeof(char)));
    if (deviceDataIn == NULL) {
        cout << "could not allocate memory!" << endl;
        return -1;
    }
    char* deviceDataOut = NULL;
    checkCudaCall(cudaMalloc((void **) &deviceDataOut, n * sizeof(char)));
    if (deviceDataOut == NULL) {
        checkCudaCall(cudaFree(deviceDataIn));
        cout << "could not allocate memory!" << endl;
        return -1;
    }

    timer kernelTime1 = timer("kernelTime");
    timer memoryTime = timer("memoryTime");

    // copy the original vectors to the GPU
    memoryTime.start();
    checkCudaCall(cudaMemcpy(deviceDataIn, data_in, n*sizeof(char),
    cudaMemcpyHostToDevice));
    memoryTime.stop();

    // execute kernel
    kernelTime1.start();
    encryptKernel<<<n/threadBlockSize, threadBlockSize>>>(deviceDataIn,
    deviceDataOut);
    cudaDeviceSynchronize();
    kernelTime1.stop();

    // check whether the kernel invocation was successful
    checkCudaCall(cudaGetLastError());

    // copy result back
    memoryTime.start();
    checkCudaCall(cudaMemcpy(data_out, deviceDataOut, n * sizeof(char),
    cudaMemcpyDeviceToHost));
    memoryTime.stop();

    checkCudaCall(cudaFree(deviceDataIn));
    checkCudaCall(cudaFree(deviceDataOut));
}

```

```

    cout << fixed << setprecision(6);
    cout << "Encrypt (kernel): \t\t" << kernelTime1.getElapsed() << " seconds."
<< endl;
    cout << "Encrypt (memory): \t\t" << memoryTime.getElapsed() << "
seconds." << endl;

    return 0;
}

int DecryptCuda (int n, char* data_in, char* data_out) {
    int threadBlockSize = 512;

    // allocate the vectors on the GPU
    char* deviceDataIn = NULL;
    checkCudaCall(cudaMalloc((void **) &deviceDataIn, n * sizeof(char)));
    if (deviceDataIn == NULL) {
        cout << "could not allocate memory!" << endl;
        return -1;
    }
    char* deviceDataOut = NULL;
    checkCudaCall(cudaMalloc((void **) &deviceDataOut, n * sizeof(char)));
    if (deviceDataOut == NULL) {
        checkCudaCall(cudaFree(deviceDataIn));
        cout << "could not allocate memory!" << endl;
        return -1;
    }

    timer kernelTime1 = timer("kernelTime");
    timer memoryTime = timer("memoryTime");

    // copy the original vectors to the GPU
    memoryTime.start();
    checkCudaCall(cudaMemcpy(deviceDataIn, data_in, n*sizeof(char),
cudaMemcpyHostToDevice));
    memoryTime.stop();

    // execute kernel
    kernelTime1.start();

    decryptKernel<<<n/threadBlockSize, threadBlockSize>>>(deviceDataIn,
deviceDataOut);
    cudaDeviceSynchronize();
    kernelTime1.stop();

    // check whether the kernel invocation was successful
    checkCudaCall(cudaGetLastError());

```

```

    // copy result back
    memoryTime.start();
    checkCudaCall(cudaMemcpy(data_out, deviceDataOut, n * sizeof(char),
    cudaMemcpyDeviceToHost));
    memoryTime.stop();

    checkCudaCall(cudaFree(deviceDataIn));
    checkCudaCall(cudaFree(deviceDataOut));

    cout << fixed << setprecision(6);
    cout << "Decrypt (kernel): \t\t" << kernelTime1.getElapsed() << " seconds."
    << endl;
    cout << "Decrypt (memory): \t\t" << memoryTime.getElapsed() << "
    seconds." << endl;

    return 0;
}
int main(int argc, char* argv[]) {
    int n;
    int key = 3;
    n = fileSize();
    if (n == -1) {
        cout << "File not found! Exiting ... " << endl;
        exit(0);
    }

    char* data_in = new char[n];
    char* data_out = new char[n];
    readData("original.data", data_in);

    cout << "Encrypting a file of " << n << " characters." << endl;

    EncryptSeq(n, data_in, data_out);
    writeData(n, "sequential.data", data_out);
    EncryptCuda(n, data_in, data_out);
    writeData(n, "cuda.data", data_out);

    readData("cuda.data", data_in);
    // readData("sequential.data", data_in);
    cout << "Decrypting a file of " << n << " characters" << endl;
    DecryptSeq(n, data_in, data_out);
    writeData(n, "sequential_decrypted.data", data_out);
    DecryptCuda(n, data_in, data_out);
    writeData(n, "recovered.data", data_out);

```

```
delete[] data_in;  
delete[] data_out;  
  
return 0;  
}
```