Parallel cryptography in CUDA

There are many cryptography algorithms that can be accelerated using parallel processing. The simplest one of them is Cesar's code. In this symmetric encryption/decryption algorithm, one needs to set a numerical key (1 number, typically between 1 and 255) that will be added to every character in the text to be encoded.

For example:
Input : ABCDE
Key: 1
Encrypted output: BCDEF

In this assignment you are requested to build a parallel encryption/decryption of a given text file. The starting code for this example can be found in crypto.zip. To compile, use the provided Makefile. To execute the code, use the same structure as for the vector-add example:

prun -v -np 1 -native '-l gpu=GTX480' ./encrypt

TODO List:

1. Implement a correct encryption and decryption sequential versions and CUDA kernels (replacing the *dummy* ones already there) and test them on at least 5 different files of different sizes (from small (few KB) to very large (MB or tens of MB); you can use any (text) file as an input for the algorithm, and you can get very large text files online). Report speed-up per file per operation, and compare all these speed-ups in a graphical manner. Is there a correlation (to be seen) between the size of the files and the performance of the application for the sequential and the GPU versions? Explain.

Note that the file names are hardcoded: original.data is the file to be encrypted, sequential.data is the reference result for the CPU encryption, and cuda.data is the result of the GPU encryption. You are recommended to use recovered.data for the decryption, which should be identical with original.data. Do not submit any input files with your solution, but make sure you state, in your report, the size (in bytes) of each input file.

To test whether two files are identical, use the diff command: diff file1 file2 > differences

If the file differences is empty, the files are identical. If the file is not empty, it will contain the differences between the two files, marked by their position in the original file(s).

2. An extension of this encryption algorithm is to use a larger key - i.e., a set of values, applied to consecutive characters.

For example:
Input: ABCDE
Key : [1,2]
Output: BDDFF

Implement this encryption/decryption algorithm as an extension to the original version (1). You can assume the key is already known (fixed, constant), and choose its length (or, better, test with multiple lengths). Optimize your code as you see necessary for this extended version, and test the extended version with the same files as for point 4.2.1. Compare the results against the single-value key, report the comparison in a graphical form, and comment on your findings.

3 (optional!!!). Implement a kernel to efficiently calculate the checksum of a file in CUDA (use a simple, additive checksum). For doing this efficiently in parallel, the information presented in class is sufficient. Report the performance of the kernel for each file (in a graphical form, too) and comment on the correlation (if any) between performance and the file size. Apply this kernel to both the original and encrypted files, and compare the output. Comment on your findings.