

CMDA 3634 Spring 2018 Homework 05

Vincent Zhao

April 30, 2018

You must complete the following task by 5pm on Tuesday 04/24/18.

Your write up for this homework should be presented in a L^AT_EX formatted PDF document. You may copy the L^AT_EX used to prepare this report as follows

1. Click on this [link](#)
2. Click on Menu/Copy Project.
3. Modify the HW05.tex document to respond to the following questions.
4. Remember: click the Recompile button to rebuild the document when you have made edits.
5. Remember: Change the author.

Each student must individually upload the following files to the CMDA 3634 Canvas page at <https://canvas.vt.edu>

1. `firstnameLastnameHW05.tex` L^AT_EX file.
2. Any figure files to be included by `firstnameLastnameHW05.tex` file.
3. `firstnameLastnameHW05.pdf` PDF file.

In addition, all source code must be submitted to an online git repository as follows:

1. While on the webpage for your git repository, go to Settings → Collaborators.
2. Add nchalmer@vt.edu and zjiaqi@vt.edu as collaborators.
3. Make a folder named HW05 in you repository and store all relevant source files to this assignment in this folder. Ensure your assignment files compile with `make`.

You must complete this assignment on your own.

100 points will be awarded for a successful completion.
Extra credit will be awarded as appropriate.

ElGamal Public-key Cryptography

Through the previous assignments, we've managed to develop a functional program which performs encryption and decryption of string messages using an n bit ElGamal cryptographic system. We can set up the cryptographic system by finding the values of the public key (p, g, h) . We can cast characters in strings into elements of \mathbb{Z}_p and encrypt them. Finally, we can decrypt elements of \mathbb{Z}_p and cast them back to characters if we know the secret key x .

In the case where we imagined that we do not have access to the secret key x , we have used MPI and OpenMP to crack our ElGamal encryption in parallel using many CPU cores. Even with that level of parallelism, it is easy to see that our program can use n bit encryption where n is large enough that we still cannot find the secret key in a practical amount of time.

In this final assignment, let's turn our previous monolithic programs into something a bit more useful by separating the basic operations into their own programs which run and save their results in files. Once we've accomplished that, let's tackle our final hurdle and use GPU acceleration to enable us to break even the strongest encryption our program can produce, namely $n = 31$ bit cryptographic systems.

Part 1: Setup, Encrypt, and Decrypt

In the HW05 folder of the instructor's github the `makefile` has been altered so that four separate programs will be compiled when you type `make` or `make all`. The names of the programs are `setup`, `encrypt`, `decrypt`, and `cudaDecrypt`. They can be compiled separately by typing `make setup`, `make encrypt`, etc. Note that the function `cudaDecrypt` requires CUDA's `nvcc` compiler, and therefore the New River cluster, to compile.

Q1(10 points) The program in `setup.c` is meant to perform the ElGamal cryptosystem setup routines and write the public key information to a file named `public_key.txt`. Complete the program so that, after querying the user for a bit length n , it write the bit length n , prime p , generator g , and number h , to a file named `public_key.txt`. For example, after running `setup`, the file `public_key.txt` could look something like,

```
25
30080879
28192225
28973487
```

Which means $n = 25$, $p = 30080879$, $g = 28192225$, and $h = 28973487$.

ANSWER:

See attached code on Canvas and GitHub.

Q2(20 points) The program in `encrypt.c` is meant to input a message string from the user and use the public key information in the file `public_key.txt` to encrypt the message. Edit the program so that it after reading the message as a string from the user the program reads the public key information from `public_key.txt` and encrypts the message. Afterwards, write the the number of cyphertext pairs, and each cyphertext pair (\hat{m}, a) , into a file called `message.txt`. The contents of `message.txt` should look something like,

```
14
28395666 9350445
3562739 5794092
10156865 16818778
2858293 6363325
29107964 21484289
3274137 24856781
19487449 15592980
18341691 30012651
15930624 18293246
```

11025926 2833780
3862999 23130298
16935347 23161724
21230787 9009798
25866655 2102881

which indicates that there are 14 cyphertext entries, $(\hat{m}, a)_1 = (28395666, 9350445)$, $(\hat{m}, a)_2 = (3562739, 5794092)$, etc.

ANSWER:

See attached code on Canvas and GitHub.

Q3.1(20 points) The program in `decrypt.c` is meant to read the contents of `public_key.txt` and `message.txt` and decrypt the message. After reading the files, the program will ask the user for the secret key. Complete the program so that if the user inputs $x = 0$ or an incorrect secret key, the program will attempt to break the encryption by searching for the correct secret key x . Once the correct secret key is found, decrypt the contents of `message.txt` and print the message to the terminal.

ANSWER:

See attached code on Canvas and GitHub.

Q3.2(5 points) Using the `decrypt` function or one of your programs from previous assignments, give a rough estimate of how long you would expect a serial code to break an $n = 31$ bit encryption. Be sure to explain how you arrived at your estimate.

ANSWER:

Using the `decrypt` function on an $n = 31$ bit encryption means brute-forcing a possible 2^{31} values for the secret key. I tried to crack an $n = 31$ bit encryption on my own computer and left my program running for over an hour with no results to show. Testing the values that approach it, I saw that the runtime increased at an exponential rate from $n = 20$ to $n = 27$, with $n = 27$ taking 1098 seconds to run (or 18 minutes). The throughput also decreased as the number of bits increased, with $n = 27$ having a throughput of 90000 values per second. A very rough estimate for how long the serial code would take to crack an $n = 31$ bit encryption is $\frac{2^{31}}{90000} = 23,860$ seconds, or 6.6 hours.

Part 2: GPU Accelerated Decryption In this part, we will use Nvidia's CUDA programming language to add GPU acceleration to the `decrypt` function. In the HW05 folder is the CUDA file `cudaDecrypt.cu`. Begin this section by copying the contents of the `main` function in your complete `decrypt.c` file to `cudaDecrypt.cu`

Q4.1(10 points) Examine the loop where we search for the secret key. Describe in words how/why we could perform this section of the program in a GPU kernel. What would the kernel require as input? How much output will the kernel produce, and how do we access it on the host? How much memory must we allocate on the device? How many threads per block would be reasonable? Why?

ANSWER:

The loop for finding the secret key takes on a brute force approach and tries every value from 0 to $p - 1$ until it finds the correct value. For large values of p , this approach will take a very long time to complete. We use a GPU kernel for this section of code in order to speed up the process.

First we allocate memory on both the host (the local CPU) and the device (the remote GPU). Then the data is initialized on the host. We transfer the data from the host over to the device, and then execute our kernels on the GPU which will run our program in parallel. Once we get results, we transfer that data over to the host and finish our program.

GPU kernels can help us save time by finding the result faster (since it can run in parallel). The kernel needs two pieces of information to execute: first the number of thread-blocks in the grid, and second the number of threads in each thread-block. The output can be accessed on the respective variables on the host

after copying the memory from the device.

Q4.2(35 points) Edit the `cudaDecrypt.cu` file to implement the kernel you described in Q4.1, i.e. add a GPU kernel function which can find the secret key x . If any `__device__` functions are required add them to `cudaDecrypt.cu` as well (you may need to rename them to not conflict with functions in `functions.c`). Be sure to `cudaMalloc` any arrays the kernel will need, and use `cudaMemcpy` to transfer memory between host and device.

ANSWER:

See attached code on Canvas and GitHub.

Q4.3(5 points) Experiment with decrypting several different n -bit encryptions. In general, how much faster (in time and throughput) is the GPU accelerated version of your decryption program? Assuming you could scale perfectly in parallel with MPI or OpenMP, how many CPU cores would you need in order to achieve the same performance as one of the P100 GPUs on the New River cluster?

Bonus:(20 points) In the HW05 folder of the instructor's github there is a file named `bonus_public_key.txt` and `bonus_message.txt`. Decrypt the message and include a screenshot of the output here.

ANSWER:

Bonus message is attached below! Really sad I missed out on the meeting!

```
nr153: vi cudaDecrypt.cu
nr153: make cudaDecrypt
nvcc cudaDecrypt.cu -I./ -g -O3 -arch=sm_60 -o cudaDecrypt -ln
nr153: ./cudaDecrypt
Enter the secret key (0 if unknown): 0
Reading file.
Retrieving cyphertext pairs.
Finding the secret key...
Searching all keys took 19.77 seconds, throughput was 7.06051e+07 values tested per second.
Decrypted Message = "Congratulations! If you are reading this then you successfully implemented a GPU accelerated ElGamal decryption and deciphered the message include
d in HW05. This is no small feat. Keep this message's contents to yourself and meet in McBryde 316 on Friday April 27 at 5:00pm. "
nr153: ==> PBS: job killed: walltime 1217 exceeded limit 1200
```