Programming Assignment 4 — Due May 16th 12:50PM

In this programming assignment you will use thrust, a template header library (like the C++ STL), to create one program that will encrypt a text using the Vigenère cipher and another that will solve a cipher text that has been encrypted with the Vigenère cipher.

Thrust provides a number of parallel primitives like scans, reductions, compactions, sorts, etc. that have high performance implementations for GPUs using CUDA and CPUs using OpenMP. These primitives can be chained together to do some very interesting things (quickly)!

There is a general introduction at `https://github.com/thrust/thrust/wiki/Quick-Start-Guide`. A more detailed list of the documentation relevant for the assignment is given at the end of the handout.

Vigenère cipher

Everyone is likely familiar with the Caesar cipher[1]. The idea behind this cipher is to use a constant shift (with modulo arithmetic), as suggested in Table 1. To solve this kind of cipher, we can use letter frequencies: indeed, by looking at the most frequent letter in the cipher text, we can find the mapping of the letter 'e', and then use this mapping to deduce the amount of the shift. Once we have the shift, it is easy to compute the plain text from the cipher text.

Table 1: Caesar Shift Cipher with a shift of two

Original Alphabet	Cipher Alphabet
A	C
B	D
C	E
...	...
Y	A
Z	B

In a poly-alphabetic cipher the shift is not constant for the entire text, it changes depending on the position of the characters within the text (see Figure below). A key is chosen which determines the shift of each letter (note that if the key has length one, we get back a Caesar cipher). For practicality, the key should be shorter than the message (usually much shorter) and it is repeated for the length of the message. The length of the key is called the period of the cipher. For a long time the key was chosen to be a word or phrase, but this was eventually found to be a weakness that could be exploited. It is best to choose the shifts at random. Although the shifts can be represented by numbers 0–25, the convention is usually to show the letter at that position in the alphabet.

This new cipher, called Vigenère cipher[2], defeats straightforward frequency analysis because each character in the plain text can become different characters in the cipher text. In the example below I $\rightarrow$ V and I $\rightarrow$ B. In the cipher text X appears twice and corresponds to K and E. How can we break this new cipher?

```
Plain text:    ILIKEMYTEACHER
KEY:           NOTNOTNOTNOTNO
=========================
Cipher text:   VZBXSFLHXNQARF
```

Part 1: Implement the Cipher

In this part, you should modify `create_cipher.cu` and fill in all the places marked with TODO. To do so, you will mainly write calls to thrust functions and fill in functor bodies.

[1] `http://en.wikipedia.org/wiki/Caesar_cipher`

[2] `http://en.wikipedia.org/wiki/Vigen%C3%A8re_cipher`

The `create_cipher` program takes two arguments from the command line: the name of a text file and the period that the cipher will use to encrypt this text. It will generate a random key of the specified period and then encrypt with the Vigenère cipher and output the encrypted text to a file `cipher_text.txt`.

Here are the steps you will need to complete:

1. Sanitize the input text to contain only lower case ASCII letters by converting uppercase letters to lowercase and removing all other symbols.
2. Compute the frequencies of letters in the clean text.
3. Apply the cipher.
4. Compute the frequencies of letters in the cipher text.

You will mainly write calls to thrust functions and fill in functor bodies. You should modify `create_cipher.cu` and fill in all the places marked with TODO.

Question 1

(10 points) Implement the functor `isnot_lowercase_alpha` that returns true if and only if the character is not a lower case character of the alphabet.

Question 2

(10 points) Implement the functor `upper_to_lower` that converts an uppercase character to a lowercase one.

Question 3

(10 points) Implement the functor `apply_shift` that shifts the input character by the shift amount. The way this functor works is as follows:

- Its constructor will take two arguments: a pointer to the beginning of the array containing the shift amounts, and the period (*i.e.* the length of the shift amounts array).
- The parentheses operator will take as arguments a character (the input character that will be shifted) and an integer (the position of this char in the input array).

Question 4

(20 points) Implement `printLetterFrequency` that calculates the letter frequency in the plain text and cipher text and print the top 5 letters along with their frequencies (out of 1.0, not as a percentage).

A few remarks on this question:

- Make sure that the letter frequency is robust to the case of less than 5 distinct letters in the text.
- Print out however many letters there happen to be if there are less than 5 respectively.

Question 5

(15 points) Implement the `main` function. You only need to fill in the parts marked `TODO`. For this question, please use a seed of 123 for the random number generator.
To test your code, you can use `mobydick.txt` that is provided in the foler common.

Part 2: Cracking the Cipher

Index of Coincidence Starting in the mid 19th century some cryptologists such as Charles Babbage were able to break Vigenère ciphers by realizing that once the length of the key (N) was known; the problem was reduced to solving N separate Caesar shift ciphers. Solving each of these ciphers is more difficult than normal because there are fewer symbols in each alphabet making frequency analysis more difficult. Furthermore, because letters

from each alphabet are not consecutive, using larger context information such as bigrams and words is also more challenging.

Despite these difficulties, the cracking process was usually possible, just tedious and required a fair bit of trial and error. In this homework, we avoid the tedious part by giving you a very long cipher text such that a pure frequency analysis based attack on each alphabet once the key length is known will work. That is we guarantee the text is long enough that the letter 'e' will be the most common symbol in each alphabet (up to a period of about 200).

How do you figure out the key length? The most general method is essentially by using the auto-correlation of the cipher text. We define an Index of Coincidence (ioc) between two texts A and B as:

$$\frac{1}{(N/26)} \sum (A_i = B_i)$$

where N is the length of the overlap between A and B. In texts where the letters are chosen uniformly at random, this number should be 1. For English texts this number is ∼1.73 due to the uneven distribution of letters in English.

Here is an example of a cross-corelation between two unrelated texts:

```
MYTEACHERISAWESOME
ILOVETHRUSTCODING
==================
0000001000000000

IOC = 26 * 1 / 17 = 1.53
```

With such short samples of texts, the actual numerical value doesn't work out exactly as we would expect with longer samples...

If in the auto-correlation we have a shift of 0, then clearly we will get an IOC of 26, this isn't particularly useful. If we shift the text by 1, then the IOC goes down below 1 (around .6) because in English letters tend to not follow themselves - there are less matches than would be expected by pure chance.

```
ITWASTHEBESTOFTIMESITWASTHEWORSTOFTIMES
 ITWASTHEBESTOFTIMESITWASTHEWORSTOFTIMES
========================================
00000000000000000000000000000000000000

IOC: 0
```

It turns out that after we shift a text by four or more (nearly) all correlation is lost and the IOC returns to 1.73 — as if we were comparing two completely distinct texts.

```
ITWASTHEBESTOFTIMESITWASTHEWORSTOFTIMES
       ITWASTHEBESTOFTIMESITWASTHEWORSTOFTIMES
========================================
       00000000000000000000000110000000

IOC: 2 / (32 / 26) = 1.625
```

Using the IOC to crack the cipher If we shift a cipher text by k and calculate the IOC with an unshifted version of itself, and if the shift matches the keylength, then the IOC will be close to 1.73 because each pair of letters will have been encoded using the same alphabet! If the shift doesn't match the keylength it will be as if we are comparing two randomly generated sets of characters and the IOC will be close to 1. This works as long as the period is greater than 3, otherwise the results are muddled by the correlation of the plaintext. For this assignment we won't worry about handling the case where the period is less than 4.

Here is a visual demonstration of what is happening — each alphabet is shown as a different color. This is for demonstration purposes, don't worry that the shifts are less than 4 here.

In these two cases, the shifts don't line up and the IOC is ∼ 1:

```
VZBXSFLHXNQARF
 VZBXSFLHXNQARF

VZBXSFLHXNQARF
  VZBXSFLHXNQARF
```

When the shifts line up suddenly the IOC jumps to ~ 1.73 because now at each position both characters have been shifted by the same amount!:

```
VZBXSFLHXNQARF
   VZBXSFLHXNQARF
```

To be absolutely sure we've found the right keylength, if the same IOC spike occurs at $2k$, then k is definitely the keylength.

Implement the Solver In this part, you should modify `solve_cipher.cu` and fill in all the places marked with TODO. You will mainly use thrust functions and fill in functor bodies.

The program `solve_cipher` takes as input a cipher text (given at the command line), outputs the key length and writes the decrypted text to a file named `plain_text.txt`.

The implementation is decomposed in two steps:

1. Determine the key length (which can be between 4 and 200) using the IOC.
2. Decrypt the cipher text.

Question 6

(5 points) Fill in the `apply_shift` functor. Your implementation can be the same as in Part 1.

Question 7

(10 points) In the `main` function, fill in the part at the beginning of the while loop in order to get the value of the IOC. After having done this, you will know the key length.

Question 8

(20 points) Fill in the rest of the `main` function (places marked `TODO`) to transform the cipher text back to the plain text. Remember that, to compute the plain text from the cipher text, you will be solving `keyLength` independent Caesar ciphers.

Total number of points: 100

A Submission instructions

You should submit one zip file containing a folder named `LastName_FirstName_SUNetID_PA4`. Make sure to have a folder with this name otherwise your files will get mixed up with other students when we unzip your file. This folder should contain:

- `create_cipher.cu`
- `solve_cipher.cu`

The zip file should be named `PA4.zip`.

To run (and grade) your code, we will copy your source file in our directory and type:

- `make`

- `./create_cipher plaintext.txt period ; ./solve_cipher_solution cipher_text.txt`
- `./create_cipher_solution plaintext.txt period ; ./solve_cipher cipher_text.txt`

This way, we will test each of the program independently (if only one of them contains a mistake, the other one will run fine).

B Hardware available for this class

Machines

We will be using the icme-gpu teaching cluster, which you can access with ssh:

`ssh sunet@icme-gpu1.stanford.edu`

We have only provided accounts for those enrolled in the course. If you are auditing the course, we may consider a special request for an account.

Compiling

To use nvcc on the icme cluster, you must copy and paste these lines to your `.bashrc` file (so they will be loaded when you connect on the cluster).

```
module add open64
module add cuda50
```

For the first time, after you changed the `.bashrc`, you need to logout and re-login (so that the changes are taken into account) or source the `.bashrc`.

You can now use `nvcc` to compile CUDA code.

Running

The cluster uses MOAB job control. The easiest way to run an executable is by using interactive job submission. First enter the command

`msub -I -l nodes=1:gpus=1`

You can then run any executable in the canonical fashion. For example

`./create_cipher` or `./solve_cipher`

as you would on your personal computer.

C Thrust documentation and additional information

- ASCII Upper Case Letters: 'A' = 65 : 'Z' = 90
- ASCII Lower Case Letters: 'a' = 97 : 'z' = 122
- Thrust homepage: `http://thrust.github.io`
- Quick start guide on thrust: `https://github.com/thrust/thrust/wiki/Quick-Start-Guide`
- A list of all thrust functionalities: `http://thrust.github.io/doc/modules.html`

- Some examples that use thrust: `https://github.com/thrust/thrust/tree/master/examples`
- Make sure to test your code. For instance, you can try to encrypt something meaningful, like Mobydick and decrypt it to see if the result is consistent with what you expected.

The specific parts of the documentation you will need for the assignment are listed below. The keywords below are clickable and will send you to the github documentation pages:

- binary_function
- unary_function
- uniform_int_distribution
- device_vector
- host_vector
- device_ptr
- sort
- sort_by_key
- remove_copy_if
- reduce_by_key
- constant_iterator
- counting_iterator
- permutation_iterator
- max
- transform
- device_pointer_cast
- inner_product
- For random number engines, refer to the slides of Lecture 11.