Homework 4 — Due May 23 11PM

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 crack a text that has been encrypted with the Vigenère cipher.

Thrust provides a number of parallel primitives such as 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

You might be 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; 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 shift amount. Once we have the shift, it is easy to compute the plain text from the cipher text.

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

Table 1: Caesar Shift Cipher with a shift of two

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, it reduces to a Caesar cipher). To be practical, the key should be shorter than the message (usually much shorter). It is then 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 to represent it by the letter in the alphabet at that position.

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 → V and I → B. Conversely, in the cipher text X appears twice and corresponds to K the first time and E the second time. 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.
`$ make create_cipher`
will only make this code.

[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, encrypt the plaintext 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. Uppercase letters must be converted to lowercase and all other symbols must be removed.
2. Compute the frequency of each letter in the clean text.
3. Apply the cipher.
4. Compute the frequency of each letter 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`, which returns `true` if and only if the character is not a lower case alphabetic character.

Question 2

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

Question 3

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

- The 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 argument a `char` (the input character to be shifted) and an integer (the position of this `char` in the input array). You may assume that the input `char` is in the range a–z.

Question 4

(20 points) Implement `getLetterFrequencyGpu`, which calculates the letter frequency in the plain text and cipher text, prints the top 5 letters along with their frequencies (out of 1.0, not as a percentage), and returns an `std::vector` with the frequency values.

Note: make sure that the function can handle the case when there are less than 5 distinct letters in the text, in which case you should print out however many letters there happen to be.

The output should look similar to this:

```
Before ciphering!

Top 5 Letter Frequencies
-------------
? 0.XXXXXXX
...

After ciphering!

Top 5 Letter Frequencies
-------------
? 0.XXXXXXX
...
```

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`.

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 individually is more difficult than solving one large Caesar cipher, 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 that the text is long enough for the letter 'e' to be the most common symbol in each alphabet (up to a cipher 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:

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

where N is the length of the overlap between texts 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-correlation between two unrelated texts:

```
MYTEACHERISAWESOME
ILOVETHRUSTCODING
==================
00000010000000000

IOC = 1 / (17 / 26) = 1.53
```

With such short samples, the actual numerical values don't work out as precisely 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 key length, 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 key length 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 you've found the right key length, you can check if the same IOC spike occurs at $2k$. If it does, then k is definitely the key length.

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.

```
$ make solve_cipher
```

will only make for this part.

The program `solve_cipher` takes as input a cipher text, 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. Use thrust to compute `numMatches`.

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. Use Thrust to calculate the shifts and to transform the cipher text back.

To ensure that your implementation is correct you should check the following:

- The output of `getLetterFrequencyGpu` matches the output of `getLetterFrequencyCpu` (the test should pass).
- The `ioc` values are in a similar range to those suggested above.
- The key length in `solve_cipher` is the same as the one you passed as input to `create_cipher`.
- The encryption key computed by `solve_cipher` is the same as the one used by `create_cipher`.
- The `plain_text.txt` file, which `solve_cipher` generates contains readable English text (with the exception of spaces and punctuation).

Total number of points: 100

A Submission instructions

To submit:

(a) The homework should be submitted using a submission script on `cardinal`. The submission script must be run on `cardinal.stanford.edu`.

(b) Copy your submission files to `cardinal.stanford.edu`. You can use the following command in your terminal:

```
scp <your submission file(s)> <your SUNetID>@cardinal.stanford.edu:
```

The script will copy the files below to a directory accessible to the CME 213 staff. Only these files will be copied. Make sure these files exist and that no other files other than those provided in the starter code are required to compile and run your code. In particular, do not use external libraries, additional header files etc, that would prevent the teaching staff from compiling the code successfully. Here is the list of files we are expecting and that will be copied:

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

The script will fail if one of these files does not exist.

(c) Make sure your code compiles on `cme213-cluster` and runs. To grade your homework, we will perform the following steps:

- `make`
- `sbatch hw4.sh`

(d) Type:

```
/usr/bin/python /usr/class/cme213/WWW/script/submit.py hw4 <directory with your submission files>
```

(e) You can submit at most 10 times before the deadline; each submission will replace the previous one.

(f) There will be a 10% penalty per 24 hours for late submission. We will not accept submissions that are submitted past two days.

B Documentation

- ASCII Upper Case Letters: ‘A’ = 65 – ‘Z’ = 90
- ASCII Lower Case Letters: ‘a’ = 97 – ‘z’ = 122
- NVIDIA Thrust documentation: `http://docs.nvidia.com/cuda/thrust`
- Thrust homepage: `http://thrust.github.io`
- Quick start guide: `https://github.com/thrust/thrust/wiki/Quick-Start-Guide`

- Thrust API (Doxygen): `http://thrust.github.io/doc/modules.html`
- Some examples that use thrust: `https://github.com/thrust/thrust/tree/master/examples`
- Check `piazza` (General Resources) for information (slides and tutorial papers).
- 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 first keyword on each line is clickable and will send you to the corresponding github API documentation page:

- `binary_function`
- `constant_iterator`
- `counting_iterator`
- `device_ptr`
- `device_vector`
- `Functional`. Various thrust functions like plus, minus, equal, greater, maximum, unary and binary functions, etc.
- `host_vector`
- `inner_product`
- `make_transform_iterator`
- `max`
- `Memory management`. `raw_pointer_cast` and other functions.
- `permutation_iterator`
- `reduce_by_key`
- `remove_copy_if`
- `sort`
- `sort_by_key`
- `transform`
- `transform_iterator`
- `unary_function`
- `uniform_int_distribution`