Programming Assignment 3

Due May 9, 2012 12:50 PM

In this programming assignment you will learn about thrust, a library which enables programming easability, performance, and portability by implementing a suite of library functions similar to the c++ STL, and adding functionality needed by gpus, such as sorting and reductions. Thrust provides a general introduction at http://code.google.com/p/thrust/wiki/QuickStartGuide

You will also learn about functors, a way of creating parameterizable functions.

1 Code Implementation

In the first programming assignment, you implemented a simple Caesar cipher using CUDA. In this assignment, you will implement a Vigenere cipher and a Vigenere solver. Caesar used a mono-alphabetic cipher, shifting every letter by the same shift value. Vigenere ciphers instead use a combination of values to shift multiple letters at the same time.

Creating a Vigenere cipher

As discovered by Al-Kindi in the 9th century frequency analysis can be used to break codes of this fashion. In English the most common letter is E, and in a large enough ciphertext, the most common symbol is then very likely to correspond to E. If a caesar shift was used, then no further work needs to be done to break the cipher. If a general mono-alphabetic substitution cipher was used, then we can continue to the next most frequent letter T and so on. Determining the more uncommon letters through pure frequency analysis may be unsuccesful and additional information might be needed (bigrams or entire words).

These ciphers were state-of-the-art until the 15th century when the first poly-alphabetic ciphers were created. In a poly-alphabetic cipher the shift isn't constant for the entire text, but changes depending on the position of the characters within the text. A KEY is chosen which determines the shift of each letter. For practicality, the key should be shorter than the message (usually much shorter) and it is repeated. 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 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. 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
```

Note that a period of 1 implies a mono-alphabetic cipher. Also, it would be possible to use a more complicated transposition than a pure shift at each position, but this doesn't really make it any harder to break the cipher, just more tedious.

Your task for the first part of the assignment is to implement a Vigenere cipher. You may only use thrust functions and iterators.

Note: Because we are going to be deciphering the text with statistical analysis, we do want to wrap the characters around, so make sure to compensate if you shift a character outside of the alphabet.

Cracking a cipher

Starting in the mid 19th century some clever cryptologists such as Charles Babbage were able to break Vigenere ciphers by realizing that once the length of the key (N) was known; the problem was reduced to solving N separate mono-alphabetic substitution ciphers. Solving each of these ciphers was more difficult than normal because in a (relatively) short message there wouldn't be many characters from each alphabet, making frequency analysis difficult. Furthermore, because letters from each alphabet are not consecutive, using larger context information such as bi-graphs and words is difficult.

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.

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 1. For English texts this number is ~1.73 due to the uneven distribution of letters in English.

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.

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

```
* MYTEACHERISAWESOME
* ILOVETHRUSTCODING
* ==================
* 00000010000000000
*
* IOC = 1 / (17 / 26) = 1.53
```

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

```
* ITWASTHEWORSTOFTIMESITWASTHEBESTOFTIMES
*  ITWASTHEWORSTOFTIMESITWASTHEBESTOFTIMES
* ========================================
*  00000000000000000000000000000000000000
*
* IOC: 0
```

In English letters tend not to follow themselves, so a shift of one corresponds to an IOC < 1, there are less matches than would be expected by pure chance.

(The phrase is backwards...Dickens wrote "It was the best of times, it was the worst of times...")

```
* ITWASTHEWORSTOFTIMESITWASTHEBESTOFTIMES
*        ITWASTHEWORSTOFTIMESITWASTHEBESTOFTIMES
* ==========================================
*        00001100000000000000000000000000
*
* IOC: 2 / (32 / 26) = 1.625
```

After we've shifted the text by more than about 3 spaces, the IOC returns to ~1.73, implying there is no correlation between letters more the 3 spaces distant.

Your task is to determine the key length, decode the ciphered text, and output the plain text. Once again, you may only use thrust algorithms.

What we give you

- `create_cipher.cu` - This contains the starter code for implementing the Vigenere cipher. You do not need to change the variable names of function headers in this file, but you must implement the `TODO` sections using the thrust library. You should modify this file.
- `solve_cipher.cu` - This contains the starter code for implemented the Vigenere solver. You must generate the frequency table for the cipher text, determine the key length, and decode the text. You must use the thrust library. You should modify this file.
- `gnu-license.txt` - This contains the GNU license in plain text. You should use it to test your file. You do not need to modify this file.

Note that we are not providing you with a `Makefile`. You must construct your own Makefile. A good reference point is the Makefiles from the previous assignments. Include this Makefile in your submission.

You can read the first few sections in `http://www.opussoftware.com/tutorial/TutMakefile.htm` to help you understand how Makefiles are constructed. You only need a basic understanding for the homework. None of the advanced functionalities are required.

2 Analysis

Type your responses and submit them as a pdf file. These questions are to encourage you to understand what you are implementing and how you are implementing it. We additionally want to ensure you can create and modify a makefile, as well as implement basic timing functionality.

1. How do you allocate memory that can be used by thrust?
2. State which algorithms from the thrust library you used.
3. Modify your `Makefile` so that it generates a release version with `-O3` or a development version with the debugging flags `-g -G`. Time the entire `solve_cipher` code. Run the code several times and average the results. Is this what you expected? Can you explain your observations?

   Hint: to compile different options using the same flag, you can check for variable definitions. For example in your `Makefile`:

   ```
   ifdef DEBUG
     NVCCFLAGS= ...
   else
     NVCCFLAGS= ...
   endif
   ```

   Then at the shell prompt, use `make` for the release version or `make DEBUG=1` for the development version.
4. One way to attempt to beat your solver, without making it harder to decipher the message, is to add text repeated several times over, perhaps even making it larger than your message. Explain how to thwart your solver using this idea, and why your solver then fails.

   Good examples to break your solver might be:

   ```
   The quick brown fox jumps over the lazy dog.
   ```

   or

   ```
   How quickly can you find out what is unusual about this paragraph?  It looks so ordinary that
   you would think that nothing was wrong with it at all, and in fact, nothing is.  But it is unusual.
   Why?  If you study it and think about it you may find out, but I am not going to assist you
   in any way.  You must do it without coaching.  No doubt if you work at it for long, it will
   dawn on you.  I don't know.  Now, go to work and try your luck.
   ```

3 Submission instructions

You should submit the complete source code for your working solution and a brief pdf file (no more than one or two pages excluding plots) containg your analysis.

To submit:

- Create a directory, for example `pset3`, on your Leland account.
- Copy the files you would like to submit in the directory `pset3`. Do not include auxiliary files that are not needed.
- From within the directory `pset3`, run the shell command
  ```
  corn:> /afs/ir/clas/cme213/bin/submit pa3
  ```
  This will send the entire content of the directory to us and will label it `pa3`.

A timestamp is also included. We will use your last submission before the deadline for grading. Please send a message to the course staff through piazza if you encounter a problem.

4 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.

To learn more about these machines and how to use them, visit
`http://icme.stanford.edu/Computer Resources/gpu.php`
The tutorial is very helpful if you are unsure how to proceed.

Compiling

To use nvcc on the icme cluster, you must copy and paste these lines to your .bashrc file:

```
module add open64
module add openmpi
module add cuda40
module add torque
```

You can now use nvcc to compile CUDA code. We have provided a makefile which will compile both the by typing `make`.

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 nodes=# :ppn=# :gpus=#
```

where # correspond to the number of nodes, processors, and gpus respectively. For this assignment, you can use 1 for every value.

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

```
./create_cipher gnu-license.txt 4
```

as you would on your personal computer.