UOW AUSTRALIA

SCIT

School of Computing & Information Technology

CSCI376 – Multicore and GPU Programming
Spring 2018

Assignment 2

Due on Wednesday, 19th September 2018 at 23:55

Part 1: Basic Kernel Programming

Task:

Write a program that does the following:

- In the host program, create two arrays as follows:
  - Array 1: An 8 element array of ints with random values between 0 and 9.
  - Array 2: A 16 element array of ints. Initialise the first half of the array with values from 1 to 8 and the second half with values from -8 to -1.

(0.5 marks)

- Write a kernel that
  - Accepts array 1 as an array of int4s, array 2 and an output array
  - Reads the contents from array 1 and 2 into local memory
    - Copy the contents of array 1 into an int8 vector called v
    - Copy the contents of array 2 into two int8 vectors called *v1* and *v2* (using **vload*n***)
  - Creates an int8 vector in private memory called *results*. The contents of this vector should be filled as follows:
    - Check whether **any** of the elements in *v* are greater than 5
      - If there are, then for elements that are greater than 5, copy the corresponding elements from *v1*; for elements less than 5, copy the elements from *v2*. (Using **select**).
      - If not, fill the first 4 elements with the contents from the first 4 elements of *v1* and the next 4 elements with contents from the first 4 elements of *v2*
  - Stores the contents of *v*, *v1*, *v2* and *results* in the output array (using **vstore*n***)

(2 marks)

- In the host program, check that the results are correct and display the contents of the output array (0.5 marks)

Part 2: Shift Cipher

A shift cipher (a.k.a. Caesar's cipher) is a simple substitution cipher in which each letter in the plaintext is replaced with another letter that is located a certain number, *n*, positions away in the alphabet. The value of *n* can be positive or negative. For positive values, replace letters with letters located *n* places on its right (i.e. 'shifted' by *n* positions to the right). For negative values, replace letters with letters located *n* places on its left. If it reaches the end/start of the alphabets, wrap around to the start/end.

For example:

If *n* = -3, each letter in the plaintext is replaced with a letter 3 positions before that letter in the alphabet list.

Plaintext: **The quick brown fox jumps over the lazy dog.**

Ciphertext: **QEBNR FZHYO LTKCL UGRJM PLSBO QEBIX WV**

Note that in the example above, c → Z, since 3 positions before 'c' wraps around to the end of the alphabet list and continues from 'Z'. Similarly, a → X and b → Y.

Also note that everything in the ciphertext is converted to upper case letters that are organised in groups of five-letter blocks and anything that is not a letter is removed.

Decrypting the ciphertext is simply a matter of reversing the shift.

Task:

a. Write a normal program that reads the plaintext from a file (a test file called "plaintext" has been provided), removes everything that is not a letter, prompts the user to input a valid *n* value and encrypts the plaintext using the shift cipher method described above.

(2 marks)

b. Extend the program to be an OpenCL program that uses a kernel to encrypt and decrypt the plaintext in parallel. Check the result using the code that you wrote in (a). Display the ciphertext and decrypted text on screen.

(3 marks)

c. Extend your program so that the user can perform parallel encryption and decryption by substituting characters based on the following lookup table:

a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z
C	I	S	Q	V	N	F	O	W	A	X	M	T	G	U	H	P	B	K	L	R	E	Y	D	Z	J

Based on the table above, for encryption the letter a (or A) will be replaced by C, b (or B) will be replaced by I, c (or C) will be replaced by S, etc.

(2 marks)

Part 3: Brute-force Plaintext Password Cracking

Given the following information about a password:

- It is stored as plaintext[1]
- The password contains 4-6 characters
- It only consists of letters (from a to z)
- It is not case-sensitive

Design and implement an efficient parallel program in OpenCL to perform brute-force plaintext password cracking. Note that your program design must attempt to match the **entire** password (i.e. you are NOT allowed to try to find the 1st character, then try to find the 2nd character, etc.)

Task:

- Store the *password* as a constant global variable in the .cl file
- Write a function (not a kernel) called *checkPassword* in the .cl file
    - The function should accept an **entire** password *attempt*
        - NOTE: This function is to check whether all characters in the *attempt* match the stored *password*
    - Check whether this attempt matches the stored *password*
    - Returns *true* if it matches, or *false* otherwise

(1 mark)

- Your program should search through the space of possible passwords to find a match with the stored plaintext password
    - Write an OpenCL kernel that will call the *checkPassword* function
    - Once a match is found, the search should terminate as soon as possible
    - Display information about the work-item that found the password (i.e. its global id, work-group id and id within the work-group)

(4 marks)

[1] Note that proper systems do not store passwords as plaintext.

Instructions and Assessment

Organise your solutions into 3 folders (named part1, part2 and part3) and put your **source files** (.c or .cpp, .cl and .h (if you use any) files) into the respective folders.

Zip the 3 folders a single file and submit this via Moodle by the due date and time (do **NOT** zip entire visual studio project files as this can be very large). Assignments that are not submitted on Moodle will not be marked.

You will have to demonstrate your working program during the lab in Week 9. You must be ready for this lab task to be assessed at the start of this lab. Do not try to fix your code during this lab, otherwise late penalties may apply. Your program will be marked in the lab so it must work on the computers in the lab (or you must demonstrate it in the lab on your own laptop).

The assignment must be your own work. If asked, you must be able to explain what you did and how you did it. Marks will be deducted if you cannot correctly explain your code.

NOTE: The marking allocations shown above are merely a guide. Marks will be awarded based on the overall quality of your work. Marks may be deducted for other reasons, e.g. if your code is too messy or inefficient, if you cannot correctly explain your code, etc.

For code that does not compile, does not work or for programs that crash, the most you can get is half the marks.

