UOW AUSTRALIA

SCIT

School of Computing & Information Technology

CSCI376 – Multicore and GPU Programming
Spring 2022

Assignment 2

Due on Friday, 23rd Sep 2022 at 17:00 AEST

Task 1 – Basic Kernel Programming

Task 1a

Write a program using OpenCL that uses a kernel to fill in the content of an array of 512 numbers in parallel. The program is to prompt the user to enter a number between 2 and 99 (inclusive). The program is to check whether the user entered a valid number, if not, the program should quit. If a valid number was entered, enqueue a kernel (using the enqueueNDRangeKernel function) that accepts the number and an empty array as its arguments, and fills in the content of the array using the number (and a work-item's global ID) as follows:

- If the user enters 2, the resulting contents of the array should be: 3, 5, 7, 9, 11, … until 1025
- If the user enters 3, the resulting contents of the array should be: 3, 6, 9, 12, 15, … until 1536
- If the user enters 4, the resulting contents of the array should be: 3, 7, 11, 15, 19, … until 2047
- …
- If the user enters 99, the resulting contents of the array should be: 3, 102, 201, 300, 399, … until 50592

After the kernel has completed its execution, display the content of the output array on the screen.

(2 marks)

Task 1b

Write a program that uses OpenCL to do the following:

- Host code.
    - Create two C++ Standard Template Library (STL) vectors, named *vector1* and *vector2*. Initialise their contents as follows:
        - *vector1*: Stores 32 integers. Initialise the integers with random values between 1 and 9 (inclusive). The set of random values is to be different each time the program runs.

- *vector2*: Stores 16 elements. Initialise the first half of the vector with integer values from 2 to 9 in sequence (i.e., 2, 3, 4, …, 9), and the second half with negative integer values from -9 to -2 in sequence (i.e., -9, -8, -7, …, -2).

  - Create and initialise the necessary buffer objects to associate *vector1* and *vector2* as arguments to the kernel and another buffer object for the kernel output (refer to the kernel code section below).
  - Enqueue the kernel such that each work-item will access 8 elements from *vector1* in the following manner:
    - Work-item 1 will access elements 0-7, work-item 2 will access elements 8-15, etc.
  - After kernel execution, obtain the kernel output and display the results on the screen.

(2 marks)

- Kernel code.
  - The kernel is to have the following parameters:

```
__kernel void task1b(__global int4* input1,
                     __global int* input2,
                     __global int* output)
{
    // your code goes in here
}
```

  - When the kernel is enqueued by the host, the content of *vector1* is to be associated with *input1*, while the content of *vector2* is to be associated with *input2*.
  - In the kernel, copy the content from *input1* and *input2* into three private OpenCL vectors of type **int8**:
    - *v*: is to contain 8 elements obtained from *input1*. The content accessed by *v* will depend on the work-item, such that
      - Work-item 1: *v* will contain vectors 0-1 from *input1*

        (Note: the content from two int4 vectors gives a total of 8 elements)
      - Work-item 2: *v* will contain vectors 2-3 from *input1*
      - etc.
    - *v1*: is to contain elements 0-7 from *input2* (use the **vload*n*** function). The content in *v1* will be the same for all work-items.
    - *v2*: is to contain elements 8-15 from *input2* (use the **vload*n*** function). The content in *v2* will be the same for all work-items.
  - Create an **int8** vector, named *results*. The content of this vector should be filled based on the following:
    - Check whether any of the elements in *v* are greater than 7

- If any elements in *v* are greater than 7, then for elements in *v* that are greater than 7, copy the corresponding elements from *v1* into *results*; for elements less than or equal to 7, copy the corresponding elements from *v2* into *results.* Use the **select** function to do this.
- Otherwise, fill the first 4 elements of *results* with the content from the first 4 elements of *v1,* and the next 4 elements of *results* with content from the first 4 elements of *v2.*

The example output provided on the next page shows an example of how *results* should be filled. Note that the values in *v* are random.

- Store the content of *v*, *v1*, *v2* and *results* in the output array (use the **vstore*n*** function). Note that each work-item will have to store its results in its respective location in the output array.

(3 marks)

Example output

```
Work-item 1
v        :  7  2  6  3  6  4  5  5
v1       :  2  3  4  5  6  7  8  9
v2       : -9 -8 -7 -6 -5 -4 -3 -2
results  :  2  3  4  5 -9 -8 -7 -6

Work-item 2
v        :  7  5  5  9  2  9  5  8
v1       :  2  3  4  5  6  7  8  9
v2       : -9 -8 -7 -6 -5 -4 -3 -2
results  : -9 -8 -7  5 -5  7 -3  9

Work-item 3
...
...
```


Task 2 – Substitution Ciphers

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

For example: If *n* = -3, each letter in the plaintext is replaced with an uppercase letter 3 positions before that letter in the alphabet list.

Plaintext:	**The 'quick' brown fox jumps over the "lazy" dog.**
Ciphertext:	**QEB 'NRFZH' YOLTK CLU GRJMP LSBO QEB "IXWV" ALD.**
Decrypted:	**THE 'QUICK' BROWN FOX JUMPS OVER THE "LAZY" DOG.**

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

Leave anything that is not an alphabet as is (i.e., punctuations and spaces should remain in the encrypted/decrypted text without alteration). The encrypted/decrypted text is to be in uppercase.

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

IMPORTANT: For the task in this section, read the content from the input file as is. Do **NOT** modify the content when reading in the text from the input file. For example, do not read in characters as uppercase letters, this conversion is to be performed when encrypting the characters.

Task 2a

Write a normal C/C++ program (not using OpenCL) that reads the content of a text file called "plaintext.txt" (a test file has been provided). The program should prompt the user to input a valid *n* value, then encrypt the plaintext using the shift cipher method described above and output the ciphertext into an output text file called "ciphertext2a.txt". To ensure that the encryption was performed correctly, your program must also decrypt the ciphertext and check that it matches the original plaintext (albeit in uppercase). Output the decrypted text into a file called "decrypted2a.txt".

(2 marks)

Task 2b

Write a program that uses OpenCL to perform parallel encryption using the shift cipher method. After encrypting the plaintext, the program is to perform parallel decryption on the ciphertext and checks that it matches the original plaintext (albeit in uppercase) on the host.

The program is to

- Read plaintext from a file called "plaintext.txt" (a test file has been provided). Note that your program is to read the plaintext as is. Your program should **NOT** pre-process/modify the plaintext before it is passed to the kernel (e.g., do not convert letters to uppercase on the host, do not change or remove letters/punctuations/spaces on the host). The text that is passed to the kernel must be the same as the text in the input file.

- Prompt the user to input a valid *n* value.
- Encrypt letters from the plaintext in parallel using an OpenCL kernel (i.e., multiple letters in the plaintext are to be encrypted in parallel). Note that it is more efficient to use OpenCL vector data types for processing in the kernel (fewer work-items and parallel vector operations).
- After completing the encryption, enqueue the kernel again to perform parallel decryption (i.e., multiple letters in the ciphertext are to be decrypted in parallel). Note that encryption and decryption can be done using the same kernel.
- Check whether the decrypted text matches the original plaintext (albeit in uppercase) on the host.
- Output the resulting ciphertext into an output text file called "ciphertext2b.txt" and the decrypted text into an output text file called "decrypted2b.txt".

(4 marks)

Task 2c

Write a program that uses OpenCL to perform parallel encryption, followed by parallel decryption, 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. Decryption is performed by simply reversing this. As with Task 2b, it is more efficient to use OpenCL vector data types for processing in the kernel, and encryption and decryption can be done using the same kernel.

Output the resulting ciphertext into an output text file called "ciphertext2c.txt" and the decrypted text into an output text file called "decrypted2c.txt".

(2 marks)

Screenshots

For **each task**, include screenshots with your submission to demonstrate that the programs work on your computer. The screenshots should capture user input (if any) and what your program displays on screen. Please use a common image format, i.e., bmp/jpeg/png. Marks will be deducted if screenshots are not provided.

Instructions and Assessment

Organise your solutions into 2 folders, named Task1 and Task2 (and subfolders as required), and put all your **source files** (i.e., all source files required to compile and run your program, e.g., .cpp, .h and .cl files) into the respective folders.

Zip the 2 folders into a single file and submit it via Moodle by the due date and time (please do **NOT** use the .rar format, and do **NOT** zip entire visual studio project folders as this can be very large). Assignments that are not submitted on Moodle will not be marked.

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. 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 allocated marks for each part. It is better to comment out sections of code that do not work and leave a note for the marker.

