SCIT

School of Computing & Information Technology

CSCI376 – Multicore and GPU Programming
SIM Session 2 2022

Assignment 2

Task 1 (5 marks)

Write a program using OpenCL to do the following:

- In the host code, create two arrays as follows:
  - *vec1*: An 8-element array of ints that contains random values between 10 and 20.
  - *vec2*: A 16-element array of ints. Initialise the first half of the array with values from 1 to 8 (i.e. 1, 2, 3,…, 8) and the second half with values from -9 to -2 (i.e. -9, -8, -7,…, -2).

(1 mark)

- Write a kernel that
  - Accepts three parameters:
    - *input1*: an input array of type int4
    - *input2*: an input array of type int
    - *output*: an output array of type int

    When the kernel is enqueued by the host, the contents of *vec1* and *vec2* are to be passed to the kernel as *input1* and *input2*, respectively.
  - Reads the contents from *input1* and *input2* into local memory
    - Copy the contents of *input1* into an int8 vector called *v*
    - Copy the contents (using **vload*n***) of *input2* into two int8 vectors called *v1* and *v2*
  - Creates an int8 vector in private memory called *results*. The content of this vector should be filled as follows:
    - Check whether **any** of the elements in *v* are greater than 15
      - If there are, then for elements that are greater than 15, copy the corresponding elements from *v1* into *results*; for elements less than or equal to 15, copy the elements from *v2* into *results*. (Use **select**).
      - If not, fill the first 4 elements of *results* with the contents from the first 4 elements of *v1*; and fill the next 4 elements of *results* 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***)

Note that the host will only have to enqueue 1 work-item.

(3 marks)

- In the host code, check that the results are correct and display the content of the output array.

(1 mark)

<u>Task 2</u> (7 marks)

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: **QEB NRFZH YOLTK CLU GRJMP LSBO QEB IXWV ALD.**

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.

Leave anything that is not an alphabet as is (i.e. punctuations and spaces).

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

Task 2a

Write a normal C/C++ program (not using OpenCL) that reads the content from a text file called "plaintext.txt" (a test file is 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 "ciphertext.txt". To ensure that the encryption was performed correctly, your program must also decrypt the ciphertext into a file called "decrypted.txt" to check whether it matches the original plaintext (albeit in upper case).

(2 marks)

Task 2b

Write a program using OpenCL to perform the same functionality as Task 2a, but in parallel (i.e. parallel encryption, followed by parallel decryption). Note that it is more efficient to use OpenCL vector datatypes for processing in the kernel, as less work-items will be required.

(3 marks)

Task 2c

Write a program using OpenCL to perform parallel encryption, followed by parallel 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
G	X	S	Q	F	A	R	O	W	B	L	M	T	H	C	V	P	N	Z	U	I	E	Y	D	K	J

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

(2 marks)

<u>Task 3</u> (8 marks)

For this section, a test image – “peppers.bmp”, is provided.

Task 3a (Image Luminance)

Write a parallel program using OpenCL to convert the RGB values (i.e. Red, Green and Blue colour channels) of an image to luminance values (this converts a colour image into a greyscale image).

For each pixel, calculate:

Luminance = 0.299*R + 0.587*G + 0.114*B

Save the luminance image into a 24-bit BMP file. To do this, set the RGB values of each pixel to the luminance value.

Note that if you use the example code from the tutorial on image processing, the R, G, and B values range from 0 to 255 on the host (unsigned char), and 0.0 to 1.0 (float) on the device.

(1 mark)

Task 3b (Gaussian Blurring)

Gaussian blurring is a commonly used technique to image processing and graphics to create a smooth blurring effect using a Gaussian function. The weights of the filter depend on the size of the Gaussian filter window. The following are example weights for a 7x7 windows:

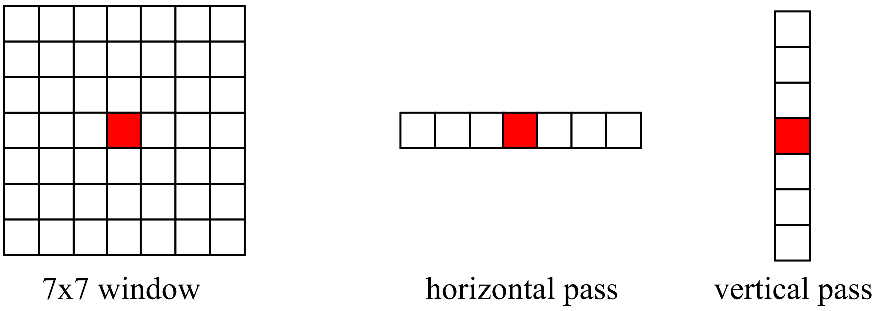
0.000036	0.000363	0.001446	0.002291	0.001446	0.000363	0.000036
0.000363	0.003676	0.014662	0.023226	0.014662	0.003676	0.000363
0.001446	0.014662	0.058488	0.092651	0.058488	0.014662	0.001446
0.002291	0.023226	0.092651	0.146768	0.092651	0.023226	0.002291
0.001446	0.014662	0.058488	0.092651	0.058488	0.014662	0.001446
0.000363	0.003676	0.014662	0.023226	0.014662	0.003676	0.000363
0.000036	0.000363	0.001446	0.002291	0.001446	0.000363	0.000036

i. Write a program using OpenCL that accepts a colour image and outputs a filtered image using Gaussian blurring based on the 7x7 window weights provided above.

(1 mark)

ii. Instead of using the 7x7 window (the naïve approach), an alternate approach is to run the filter in 2-passes. The first pass will perform blurring in the horizontal direction; the result will then undergo a second pass to blur it in the vertical direction (enqueue the kernel twice to perform blurring in each direction). The result will be similar to the single window approach, but the amount of computation will be different.

For example, using a 7x7 window approach, each pixel will have to perform a weighted sum on 49 pixels. In the 2-pass approach, each pixel will have to perform a weighted sum on 7 pixels in each pass, processing a total of 14 pixels. This is illustrated below:



Your task is to implement the parallel 2-pass approach. For this, use the following weights for the horizontal pass as well as the vertical pass:

0.00598	0.060626	0.241843	0.383103	0.241843	0.060626	0.00598

(3 marks)

Task 3c (Bloom Effect)

Bloom effects are commonly used in graphics, movies, video games, etc. This part combines the work from Tasks 3a and 3b. The basic steps to create an image with a bloom effect are illustrated as follows:

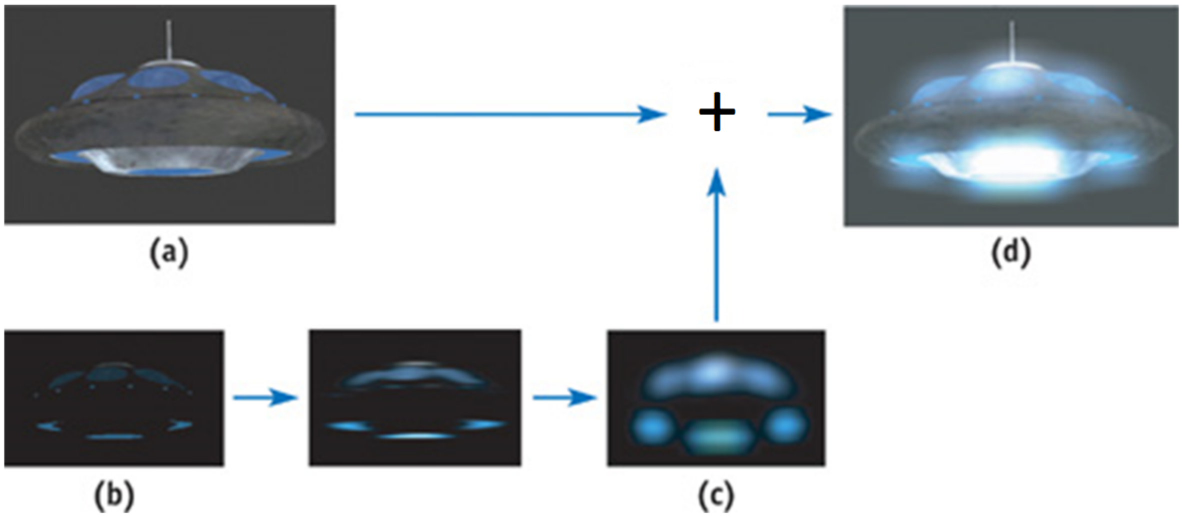


Figure 1: Bloom effect steps.

1. The image in Fig. 1(a) shows the original image.
2. The image in Fig. 1(b) shows an image where the glowing pixels are kept, while the rest are set to black. For this assignment, allow the user to input a valid threshold luminance value. Pixels above the threshold luminance value are kept, while pixels below this luminance value are set to black. The luminance value in this step is related to Task 3a.
3. The image in Fig. 1(b) undergoes a horizontal blur pass, then a vertical blur pass to obtain the image depicted in Fig. 1(c). This step is related to the horizontal and vertical blur passes in Task 3b.

4. Finally, the pixel values in the images shown in Fig. 1(a) and Fig. 1(c) are added together to form the final image shown in Fig. 1(d). Note that the values above the maximum colour value should be clamped to the maximum value.

Write a parallel program using OpenCL to perform the bloom effect on an input image. For the threshold value (in step 2), allow the user to enter a valid threshold value.

Your program should output the following images:

- an image after step 2 (i.e. image showing the glowing pixels)
- an image after the horizontal blur pass
- an image after the vertical blur pass
- the final image with the bloom effect

(3 marks)

Write a description of what your program does. You may draw diagrams if it helps with your explanation. Include this with your submission.

For **ALL** tasks, include **screenshots** with your submission. The screenshots are to demonstrate that the programs work on your computer.

For Task 3, include examples of output images obtained from your program.

Please follow the tutor's instructions.

Instructions and Assessment

Submit your tasks as one zip file with three folders, named Task1, Task2 and Task3, and include all the required files (e.g., .cpp, .h, .cl) in your submission.

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, is not well commented, 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 assessment marks or less.

References

The images were sourced from

- http://http.developer.nvidia.com/GPUGems/gpugems_ch21.html