Lab 5 Introduction to OpenCL programming

Diego Bielsa Monterde (776564), Alejandro Báscones Gállego (801469)

December 7, 2023

1 Introduction

In this lab, an OpenCL programming introduction is carried out. In order to achieve the goal of this lab, a set of steps is followed, from the basics (not included in this report according to the lab script petition) to the usage of a device given an OpenCl code.
So as to get our measurements without noise (which could happen if a connection with ssh to central.cps.unizar.es is done in our own laptop, since we might not be the only ones connected) a computer of the lab has been used. This computer has OpenCL [1] installed and contains 3 platforms with one device each:

- Intel(R) FPGA Emulation Platform for OpenCL(TM):
  - Intel(R) FPGA Emulation Device (accelerator):
    - Max compute units: 4.
    - Max clock frequency: 3700MHz.
    - Address bits: 64, Little-Endian.
    - Global memory size: 16670388224 (15.53GiB).
    - Max memory allocation: 8335194112 (7.763GiB).
    - Global Memory cache size: 262144 (256KiB).

    This field-programmable gate array has 4 compute units which allows it to execute 4 instruction in parallel. The frequency value means that 3700 millions of instructions can be executed in one second (assuming that 1 clock = 1 instruction); this value is given per compute unit, so in an ideal case $4 * 3700 = 14.800$ millions of instructions per second could be executed. It has 64 address bits that allows the access to 16EiB in which the values are stored in Little-Endian (least significant byte is stored at the lowest memory address). The total memory size of this FPGA is 15.53GiB; each kernel can have a maximum of 7.763GiB of memory allocated; the cache size of the global memory is size 256KiB.

- Intel(R) OpenCL:
  - Intel(R) Core(TM) i3-6100 CPU @ 3.70GHz (CPU):
    - Max compute units: 4.
    - Max clock frequency: 3700MHz.
    - Address bits: 64, Little-Endian.
    - Global memory size: 16670388224 (15.53GiB).
    - Max memory allocation: 8335194112 (7.763GiB).
    - Global Memory cache size: 262144 (256KiB).
    - Max work group size: 8192.

    This central processing unit has the same main attributes as the FPGA described before adding a maximum work-group size of 8192 meaning that the maximum number of work-items in a work-group is 8192.

- Intel(R) OpenCL HD Graphics:
  - Intel(R) HD Graphics 530 [0x1912] (GPU):
    - Max compute units: 23.

* Max clock frequency: 1050MHz.
* Address bits: 64, Little-Endian.
* Global memory size: 13336309760 (12.42GiB).
* Max memory allocation: 4294959104 (4GiB).
* Global Memory cache size: 524288 (512KiB).
* Max work group size: 256.

Regarding the GPU the number of computing units is significantly bigger than the previous cases, being this 23, this is because the GPU aims to get the most possible parallelization. However, the frequency decreases a lot (as the GPU has 23 computer units, does not need that high frequency), getting a top of 1050MHz, which assuming the same conditions as in the previous cases, in an ideal case $23 * 1050 = 24.150$ millions of instructions per second would be executed; the result doubles the previous number of instruction per second. As the data stored by a GPU is usually lower than in a GPU, the global memory and the max memory allocation are smaller; in the other hand, the cache size is bigger due to the need for faster accesses.

The selected device to perform the execution of our kernel has been the GPU. This choice is based on the task that each work-item has, which is just flipping a pixel computing its mirror one. As this task can be fully parallelized we have chosen the GPU.

2 Basic creation of a kernel: image_flip

2.1 Program Properties

In this section the way of programming the image flip is going to be explained, having as a priority the main features. First of all, the whole program is divided in two parts, the host and the device ones:

- Host code (.c): the host is responsible for setting up the OpenCl environment which in this case is going to be composed of one context; this is an instance of a platform that can contain several devices (only one device is going to be used). As just mentioned, only a context with one device has been created. To begin with the implementation of the host a device must be chosen, creating a context and a command queue for the device, in this case the GPU; after that the image and the kernel code are loaded, then the kernel is compiled and built.
  Now is time of creating input and output arrays, since the flip operation is in-place only one array is created. After thinking about how this array should be (int*, uchar3*, unsigned char*), the decision taken has been to send a char* array passing the image per rows in which a pixel is formed by 3 chars (red, green and blue). The int* approach wasted 1B per pixel, causing this a huge total waste of memory; the uchar3* possibility was the first one taken, but this type of values caused error while passing it; eventually, the unsigned char* approach was the chosen one due to the fact that memory is not wasted and that it can be easily passed.
  Finally the data is copied from host to device and passed via function arguments; subsequently the kernel is launched with a global work size equals to the number of pixels (total amount of work) and a local work size equals to NULL (a NULL makes the local size be established by OpenCL, this is done by dividing the total amount of data by the number of compute units). After that, the result is read and the flipped image stored.

- Device code (.cl): in this file, the device code is implemented, meaning this that each work-item is going to execute the code in this file. In this case this .cl file *kernel_flip.cl* implements a pixel flip with its mirror one. As previously exposed, it receives a char array that represents the pixels per row flattened; knowing that a pixel occupies 3B (one per spectrum channel RGB) an arithmetic operation to perform the flip is done.

2.2 Measurements

Several measurements have been taken and described in detail so that the questions in the lab scripts are answered properly. Note that 50 experiments have been done, therefore the results shown are the average and standard deviation.

2.2.1 Execution time (program and kernel)

The first time measure taken in order to describe the performance of the code is the overall execution time, by using the clock function (time.h library). Knowing the number of clock cycles we obtain the time in seconds dividing by the number of clock cycles per second:

$$Time = \frac{end\_cycle - init\_cycle}{CLOCKS\_PER\_SEC(n\_clockcycles/s)}s$$

After several executions we have obtained the following metrics:

- Mean: 0,332578 s.
- Standard deviation: 0,001637 s.

To obtain the kernel execution time avoiding overhead times, we have measured times on the device by using events in the function *clEnqueueNDRangeKernel*, which launches the kernel. The following metrics were obtained derived from various executions using this profiling technique:

- Mean: 0,000179 s.
- Standard deviation: 0,000004 s.

It can be seen that the overall execution time is quite bigger than the kernel time itself. This is because of the computational cost of the overall program with respect to the kernel execution; the function *printf* (which raises the time noticeably) and the OpenCL environment creation by the host is more expensive than the image flip itself, which as it can be seen is pretty fast.

2.2.2 Bandwidth

Regarding the bandwidth there are two types to take into account: **from memory to kernel** and **from kernel to memory**. The formula to compute bandwidth is the following :

$$Bandwidth = \frac{amount\_of\_data\_passed(B)}{time\_spent(s)}B/s$$

- **From memory to kernel**: this bandwidth describes the number of bytes per second that have been transmitted from the host memory to the kernel. It is specifically represented by the function *clEnqueueWriteBuffer* which enqueues commands to write to a buffer object from host memory. Hence the time measurements have also been taken with profiling, using events to avoid overhead time of OpenCL runtime.
  - Mean: 19.740.055.552 B/s = 19,74 GB/s.
  - Standard deviation: 344.180.752 B/s = 0,344 GB/s.

  The resulting bandwidth are quite reasonable, taking the values of 20GB/s approximately. The standard deviation given is quite normal regarding the operation performed, as sometimes the data is passed faster or slower.

- **From kernel to memory**: this bandwidth describes the number of bytes per second that have been transmitted from the kernel to the host memory. It is specifically represented by the function *clEnqueueReadBuffer* which enqueues commands to read from a buffer object to host memory. Hence the time measurements have also been taken with profiling, using events to avoid overhead time of OpenCL runtime.
  - Mean: 24.177.506.304 B/s = 24,177 GB/s.
  - Standard deviation: 427.849.481 B/s = 0,427 GB/s.

  The resulting bandwidth are quite reasonable, taking the values of 25GB/s approximately. The standard deviation given is quite normal regarding the operation performed, as sometimes the data is passed faster or slower.

2.2.3 Throughput of the kernel

The throughput of a program is given by the division of the amount of work by the execution time of the program. Since the kernel job is flipping pixels, the amount of job is the same as the number of pixels. Therefore, the throughput equation is the following:

$$Throughput = \frac{number\_of\_pixels(pixels)}{execution\_time(s)} pixels/s$$

- Mean: 2.365.400.064 pixels/s = 2.365,4 Mpx/s.
- Standard deviation: 391.600 pixels/s.

This throughput means that each pixel is flipped in 0,00000426s which is quite acceptable since the only task performed by the kernel is a few lines of code that switches some values. Moreover, assuming that 1MB is 1.000.000B, an image with 1MB size could contain 333.333 pixels, and making the division of $\frac{2.365.400.064}{333.333} = 7.096$ images of 1MB could be flipped in one second.

2.2.4 Memory footprint (program and kernel)

As memory footprint of the program, the memory used for the flip has been taken into account. There are 3 main footprints to compute, these are the following:

- Host: the memory used by the host to perform the flip is the one that allows the kernel to perform the flip operation, therefore it is formed by the image as a *unsigned char** and two *int* values to pass the width and height of the image. The operation to compute it can be seen down here:

  $$sizeof(unsigned\_char) * img.size() + sizeof(unsigned\_int) * 2$$

  In this case the result is 1.145.780B which is 1MB approximately.

- Kernel: on the other hand the memory used by the kernel is given by the arguments that the function has, which are the image as an *unsigned char** and the width and height of the image as an *int* each. The operation to compute it can be seen down here:

  $$sizeof(unsigned\_char) * img.size() + sizeof(unsigned\_int) * 2$$

  In this case the result is 1.145.780B which is 1MB approximately.

- Program: note that for the whole program we need the input/output host array and another input/output buffer for the device, adding to this both integers representing the size. In conclusion the operation to calculate is the following:

  $$(sizeof(unsigned\_char) * img.size() + sizeof(unsigned\_int) * 2) * 2$$

  The result of this operation is 2.291.560B which is 2MB approximately.

References

[1] *OpenCL*. OpenCL. URL: `https://www.khronos.org/opencl/`. (accessed: 03.12.2023).