Laboratory 6: Heterogeneous Programming in OpenCL

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

December 31, 2023

1 Introduction

This laboratory session continues the work done on the previous one. However, instead of just sending an image to a device this lab is based on sending a large set of images to two devices (two separated GPUs). The main idea is to design a concurrent application that takes advantage of all possible parallelism, both the host one and the device one. Besides, the workload of each device has to be balanced in order to leverage the whole performance of both devices, spending the least time possible to execute the code. For sure some metrics and tables are going to be shown throughout this document.

The machine used is *berlin.unizar.cps* which has two GPUs with the same characteristics (due to this, only one is going to be explained). These GPUs are inside the same platform but in different devices and have the following characteristics:

- **NVIDIA CUDA**:
  - NVIDIA A10 (GPUs):
    - Max compute units: 72.
    - Max clock frequency: 1695MHz.
    - Address bits: 64, Little-Endian.
    - Global memory size: 23677698048 (22.05GiB).
    - Max memory allocation: 5919424512 (5.513GiB).
    - Global Memory cache size: 2064384 (1.969MiB).
    - Max work group size: 1024.

    Regarding the GPU the number of computing units is significantly large, being this 72, this is because the GPU aims to get the most possible parallelization. On the other hand, the frequency gets a top of 1695MHz, which in an ideal case $72 * 1695 = 122.040$ millions of instructions per second would 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 GPU is 22.05GiB; each kernel can have a maximum of 5.513GiB of memory allocated; and the cache size of the global memory is size 1.969KiB.

Regarding the forced (the devices themselves have the same hardware but duplicated) heterogeneity it is mandatory to realize that each device might have different performance time given the tasks even though they got the same hardware specifications. This is the reason of the necessary workload balance, otherwise a large time gap between devices could appear.

It is important to note that both devices to use are in the same platform, making it possible to create an extra *command_queue* in the same context to send and retrieve data from devices.

2 Device Code Properties

To implement the kernel code is necessary to create a *.cl* file. In this file, the device code is implemented, this means that each work-item is going to execute the code written in it. In this case this .cl file *kernel_flip.cl* implements a pixel flip with its mirror one. As in the previous lab, 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. The code is based on the following algorithm:

```
image_flip(char* image, int width, int height){
    x <- pixel_to_flip()
    IF x is on the left column part of the image THEN:
        mirror_index <- get_mirror_index()
        flip(x*3, mirror_index*3)
        flip(x*3+1, mirror_index*3+1)
        flip(x*3+2, mirror_index*3+2)
}
```

Thanks to this kernel code and to the nature of the algorithm, it is possible to parallelize efficiently the kernel code since it could be possible to split the image in several chunks. However, this approach is not the taken one due to the small images used (lab machines don't allow us to use big images), the approach followed is going to be explained in the following sections.

3 Host Code Properties

According to this laboratory script, the main part to take into account is going to be described in the following sections where the parallelism achieved and the different program decisions to face the problem are exposed.

The whole code is based on the one coded on the laboratory 5, and it follows this path:

- **Creating the context**
- **Creating the command queues**
- **Loading the image**
- **Creating and building the program**
- **Creating kernel with the binary**
- **Copying data from host to device**
- **Launching kernels**
- **Retrieving the data**
- **Measurements**

Having this code structure several issues to face appears, such as how to parallelize this path in order to launch both GPUs in parallel for better performance; how to divide the set of images in order to be able to get the best possible performance; how to manage the memory to avoid getting a segmentation fault (keep in mind that the space allowed to use in lab machines is small); and how to make the code as much adaptive as possible. Furthermore the bottleneck of the heterogeneous execution is determined and a comparison of the execution time with the non-heterogeneous alternative is done.

3.1 Launching Kernels in both GPUs

The first issue to face while coding the solution to the problem is how to launch the kernel in both GPUs given in the machine *berlin.unizar.cps*. The first step is identifying the common issues in the path mentioned in the Section 3:

- **Creating the context - Common**: as the devices are in the same platform, the context is the same.
- **Creating the command queues - NOT Common**: each device must have its own command queue.
- **Loading the image - Common**: since the image is the same, only one has to be loaded.
- **Creating and building the program - Common**: creating and building the program is done for both devices at the same time in the same variables. This is due to the fact that the binary is the same.
- **Creating kernel with the binary - Common**: with the binary in the previous step, only one kernel is created.
- **Copying data from host to device - NOT Common**: for each image a device memory buffer is created and the command of writing it to a buffer object from host memory is enqueued. As the variables are different the task is not common.
- **Launching kernels - NOT Common**: at this step each kernel must be launched. As we launch one kernel per image, each launch is independent, making the task not common.
- **Retrieving the data - NOT Common**: as both previous task, retrieving the data is independent since the data is read once per kernel.
- **Measurements - NOT Common**: given all this task within the command queue, the measurements of the not common data is also not common.

Given this detailed path, a pattern can be seen: the first 5 steps are mostly commons to each device (except the command queue creation, but the following tasks depend on this creation which makes impossible to parallelize it) and the rest are not common. This situation opens the posibility of going in parallel from copying data to device until retrieving it and adding some metrics. This parallelization has been done and it can be seen as creating the environment and then working with each device in parallel, which is one of the best approaches to parallelize the host code in addition to the devices pixel parallelization itself.

For sure, after all this parallelism is quite important to wait for the threads termination and to free the variables needed.

3.2 Workload Balancing

Once the kernels are launched in each device, it is the moment of balancing the workload for each GPU. For this load balancing several aspects must be taken into account.

The first one is the way of dividing the work where two approaches can be followed: issue each image of the stream to a device, or split each image among both devices. For simplicity and lack of time the first approach has been chosen, however, the second one would be slightly better in this case due to the same GPU hardware. Note that the images are quite small, therefore, changing the approach would not cause a big change of performance of the program.

Next aspect to handle given the approach chosen is the amount of images that have to be sent to each device. For the purpose of solving this current problem of image division two alternatives have been implemented:

- **Half approach**: this solution is quite simple, half of images to each device. In this case the performance of each device is not taken into account since they are assumed to perform the same. For this reason this approach is pretty bad with quite different devices as the slowest device will keep executing code meanwhile the fastest will end much earlier. In the Figure 1 the execution times of each device are exposed; a set of *5.000* images have been used, each device taking half *2.500*. Note that the hardware of each GPU is the same, thus, the execution times are similar.

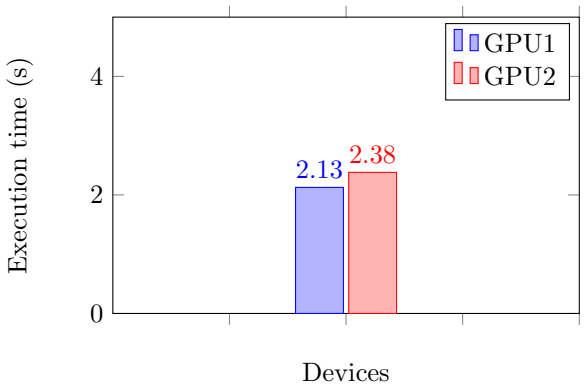

Figure 1: Half approach execution times

- **Chunk approach**: using chunks is the most efficient approach we have seen during the course. The way it has been programmed is the following: first each device runs only one kernel for the same image and gets the execution time; then with each execution time, we compute the chunk division for each device.

  For example, if GPU1 lasts 1s and GPU2 lasts 0.5s for performing the kernel operation, GPU2 is clearly twice faster than GPU1, then GPU1 chunk-size would be 1 and GPU2 chunk size would be $\frac{1}{0.5} = 2$. Given this example GPU2 would execute twice operations than GPU1; having then *5.000* images, if this equation is solved $2\#GPU1 + \#GPU1 = 5.000$ we get the amount of images to send to GPU1; eventually the quantity of images to send to GPU2 will be $\#GPU2 = 5.000 - \#GPU1$.

  In the Figure 2 the execution times of each device are exposed: A set of *5.000* images has been used, then the chunk-size is dynamically computed and sent to each device. Note that the hardware of each GPU is the same, thus, the chunk-size is half for each one. However, this approach will properly work with different devices.

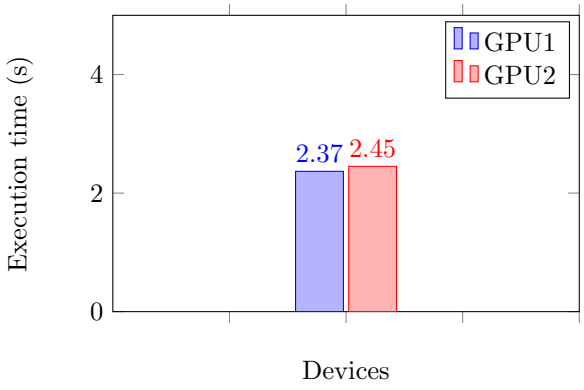

Figure 2: Chunk approach execution times

Despite of having the same hardware conditions, GPU2 seems to be always spending a bit more time, this may be caused for being the one launched the last and having less schedule priority then (regarding the parallelization in host code).

So as to measure the execution times, *std::chrono* has been used. Section 3.1 mentions that there is a parallel part in which each device is managed; the times obtained are the ones of that parallel part, so they provide information from copying the data to the device until its retrieval. It has been done that way with the aim of taking into account all device memory accesses. This is the structure of the measurement code:

```
auto start = std::chrono::steady_clock::now();

for each_image do:
    clCreateBuffer();
    clEnqueueWriteBuffer();
    clSetKernelArgs();
    clEnqueueNDRangeKernel();
    clEnqueueReadBuffer();

auto end = std::chrono::steady_clock::now();
auto execution_time = end - start;
```

3.3 Memory Management

Concerning the memory management a main problem given by the space limitation in the DIIS department appears. This space limitation avoid the user to store more than a given space of memory, preventing any further storage and creating a problem while storing a large amount of images.

So as to avoid this space limitations, the only way of carrying it out is not storing the images, but replicating them in the code itself. It is important to note that storing the images in a vector of the program will also cause a *segmentation fault* as the space that is being used is the same as the used one while storing things locally (empirically proved).

```
unsigned char image_data[N_images][img.size()];

ERROR: Segmentation fault (core dumped); the acces to that amount of memory is not allowed
```

Then, the only way of working with this large set of images is using the memory in each device thanks an approach based on creating a buffer per replicated image to send to each device; causing this the device to have a large amount of buffers with data waiting to be executed, but all the space used will belong to the device itself. In order to set the parameters properly, each buffer (remember that a buffer contains only one image) is passed as an argument immediately before of launching a kernel that will use these arguments set. The last step is the information retrieval, in this case each image is retrieved but only the first one and the second one is stored locally for its properly visualization. In the following list, these steps are explained in detail:

- **Storing the image in host memory**: the first step of this path is storing the image into a *unsigned char** which is the way we are passing it. It has been mentioned before that if we store a large number of images, a segmentation fault will appear, this is the reason of storing only one image, the loaded one with *1.145.772B* size. The algorithm is based on the following pseudo-code:

  ```
  unsigned char image_data[img.size()]; # only one array

  # storing the image in the image_data array
  initArray(image_data, img);
  ```

- **Creating device stream of images**: the second step is creating the memory buffers in device memory and enqueue a command to write on them so as to fill them with the image information

desired. The image used is the one used for the previous lab, which is *1.145.772B* size; as the full image set to use has a size of *5.000* images, the total size (sum of both devices) will be $1.145.772B * 5.000 = 5728,86MB$. Thanks to the information given in the Section 3.2, it is already known each device will compute half of images *2.500*, thus each device will need 2864.43MB of storage capability to handle this stream. As mentioned in the Section 1 the global memory size of each device is 22.05GiB which is large enough to solve the situation without struggling. The algorithm is based on the following pseudo-code:

```
cl_mem img_buffers[N_images]; # N_images buffers
for each_image do:
    # one buffer per image
    img_buffers[i] = clCreateBuffer(img.size());

    # enqueue write host to device command
    clEnqueueWriteBuffer(img_buffers[i], image_data);
```

- **Setting arguments and launching kernels**: once the device memory write commands have been enqueued it is time of setting the arguments and launching the kernels with their respective arguments. Each kernel is going to work with a device buffer, therefore, each kernel is solving one unique image. As the command queue allows to enqueue several commands, all launch commands are enqueued sequentially without waiting for the execution of the other kernels. Hence, once the buffers have been created and the writing and launching commands have been enqueued, the rest of the work lays on the device itself. Since the device parallelization granularity is pixel size and the command queues are in order (no need for out of order queues), the execution paradigm is based on flipping one image, then preparing the flipped image to be read by the host, keep working with the next image and so on. The algorithm is based on the following pseudo-code:

```
for each_image do:
    # setting arguments, one buffer per kernel
    clSetKernelArg(img_buffers[i]);

    # launching the kernel that will compute those arguments
    clEnqueueNDRangeKernel(kernel);
```

- **Data recovering**: last but not least, the data (flipped images) must be recover. Here the same problem as in the first step of this path appears: it is not possible to retrieve all the data and store it locally due to the space restriction. However, it would not be either fair either proper just retrieving the images that fit. Therefore, the approach followed for the data recovering is based on recovering all the images from the device but replacing each in the same array; then the only locally stored are the first and the last. This approach guarantees the correction of the algorithm. The algorithm is based on the following pseudo-code:

```
for each_image do:
    # each buffer modified is read in the same array
    clEnqueueReadBuffer(img_buffers[i], image_data);

    # only storing two images locally
    if image.isFirst() OR image.isLast() then:
        store(image);
```

3.4 Bottleneck

Bottlenecks are specific points in a program where the performance is significantly limited or restricted. They can be found at many application levels. In this case we can focus in two mainly areas where bottlenecks are quite notable:

- **Two devices**: if we aim the case of two devices working cooperatively, the possibility of one of them performing much slower than the other one is quite high (usually they have not the same hardware conditions), and here is where this kind of bottleneck is placed. Having a smart workload balance is important so as to decrease this bottleneck as much as possible. In the Figure 3 a poor workload balance is shown (it has been simulated passing one GPU twice work to do). As Figure 3 exposes, the bottleneck is clearly found in the execution time of GPU2, since it lasts pretty much longer than GPU1, thus, the execution time is hardly conditioned by GPU2 performance.

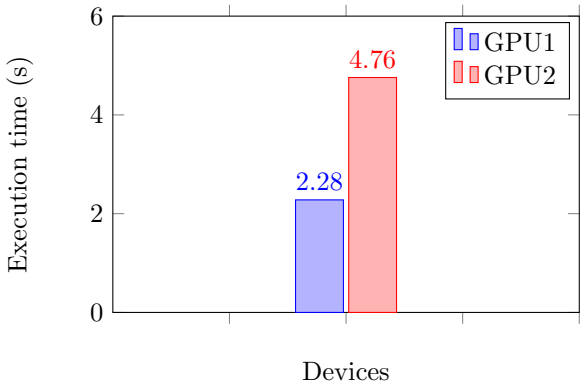

Figure 3: Poor workload balancing meaning big bottleneck; specifically GPU1 got 2.500 images whereas GPU2 got 7.500 images (time measurement like in Section 3.2)

Nevertheless, when a nice workload approach is applied, the bottleneck decreases such a lot. This is the case of the Figure 4 in which the workload balance manages to almost vanish the bottleneck given by the two devices working cooperatively.

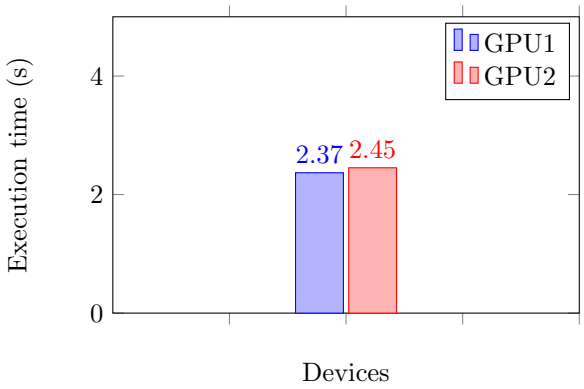

Figure 4: Nice workload balancing meaning small bottleneck; specifically both GPUs got 2.500 images (time measurement like in Section 3.2)

- **One single device**: in the case of a single device the bottleneck is found elsewhere. Specifically, it is necessary to focus on what happens when apllying the kernel to a single image, measuring the computation and communication time. With both these measurements, since passing more images is proportional, the bottleneck will be placed.

  In our case, the data passed to the device (only one image) is *1.145.772B* and the operation is flipping the image. Figure 6 shows the bottleneck with one only image, it can be seen that this

bottleneck is found in communication, hence, the communication will be the bottleneck of our program.

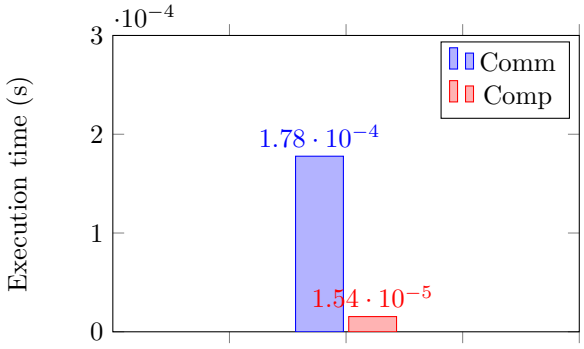

Figure 5: Bottleneck in one single device (one single image)

Figure 6 shows the bottleneck with *5.000* images which is the same as with one single image but with proportional values.

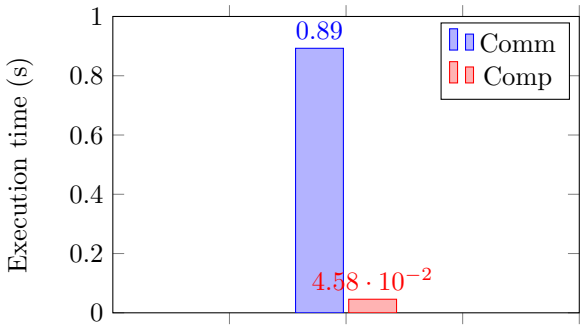

Figure 6: Bottleneck in one single device (5.000 images)

Note that at this moment two new time measurements are needed: communication time measurement and computation time measurement. The pseudo-code of the time measurements approach is the following:

```
float comumnication_time = 0;
float computation_time = 0;

for each_image do: # it can be either one image or 5.000 images
    clCreateBuffer();
    comumnication_time += clEnqueueWriteBuffer().execute().getTime();
    clSetKernelArgs();
    computation_time += clEnqueueNDRangeKernel().execute().getTime();
    comumnication_time += clEnqueueReadBuffer().execute().getTime();

print(comunication_time);
print(computation_time);
```

3.5 Adaptability

The adaptability of the code programmed is quite high. There are 3 main scenarios to test ours:

- **New execution environments**: with respect to the creation of new execution environments, our code lets the programmer using *berlin.unizar.es* described in Section 1 as he/she pleases. As the machine just mentioned only has a platform with two devices, our adaptability is based on either using only one device or both. I also allows setting the number of images to process and choosing the balance approach.

```
# platform: allows choosing the platform in wich the devices are located
# device1: allows choosing the first device
# device2: allows choosing the second device
# add_second_device: 1 ==> device2 is used; 0 ==> device2 is not used
# N_images: total number of images to modify
# balance approach: 0 ==> half approach; 1 ==> chunck approach
Usage: <platform> <device1> <device2> <add_second_device> <N_images> <balance_approach>
```

  However, this code has been programmed taking into account the characteristics given by *berlin.unizar.es*. This means that no more devices can be added and no more than one platform can be used. In order to do that, instead of using current values of contexts and command queues, arrays could be the solution. To sum up, this approach is perfect for *berlin.unizar.es* but might struggle with other machines. (We have not programmed it as scalar as possible for all machines, just for the current one).

- **Different set of input images**: the size of the input images set can be modified just changing the program argument *N_images*, thus, with respect to the size of the image set the adaptability is huge. However, our approach performs the operations only in one image, and it must be named *image.jpg*. This last drawback can be fixed with another program argument named *image_name* for example.

```
# current
CImg<unsigned char> img("image.jpg");

# potential fix
CImg<unsigned char> img(image_name);
```

  Another small problem appears when applying the algorithm to different images as this is not possible as this moment. A vector of images could solve this problem.

- **Different transformations**: to apply different transformations (diferent kernel) to our set of images the only thing to do is to set the file name to *kernel_flip.cl* and the name of the function to *image_flip*. Again this could be solved by passing the name as parameters.

```
# current
FILE *fileHandler = fopen("kernel_flip.cl", "r");
kernel = clCreateKernel(program, "image_flip", &err);

# potential fix
FILE *fileHandler = fopen(kernel_file, "r");
kernel = clCreateKernel(program, kernel_name, &err);
```

  Note that this would be the only change to be made cause the memory management and the pixel operation would be the same.

  If the programmer would like to have several kernels instead of only one, probably a vector of them will be enough. Of course if pixel granularity is kept.

Note that the program is quite adaptive to the machine *berlin.unizar.es* so as to be able to execute the programmed kernel on it in each device and with each approach without recompiling it. Nevertheless, the lack of time prevented us from go even further as we wanted to.

3.6 Comparison with non-heterogeneous alternative

Thanks to our code adaptability described in the previous Section 3.5, running the non-heterogeneous alternative is easy since the only mandatory change is found in the program arguments. In order to get a correct comparison 11.000 images have been flipped in both approaches: two devices (chunk workload balance) and one device.

Figure 7 shows how non-heterogeneous approach tends to spend more time executing the same amount of images. Ideally, the time spent with the heterogeneous approach (assuming same devices characteristics) would be half; however this amount of data to work on is quite low to achieve the full performance of the parallel host-code approach, as the overhead of initializing and joining threads is larger enough to not to be ignored (we have tried to use more than 11.000 images, but these GPUs do not allow to do that).

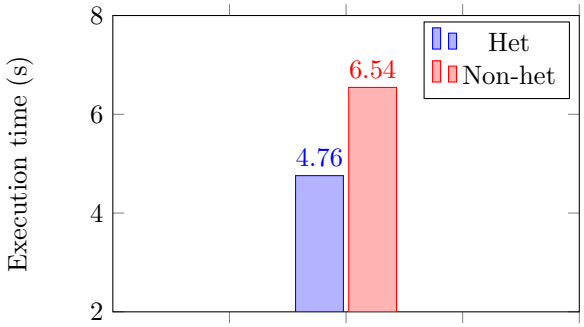

Figure 7: Heterogeneous and non-heterogeneous comparison (5.000 images)

There is a case in which the heterogeneous approach would be even better. In this case a GPU and a significantly slower CPU would be used in the heterogeneous approach; on the other hand the non-heterogeneous approach would be executed by the slower CPU, then, if a chunk balance is used, the non-heterogeneous approach would be enven much slower.

Note that now the execution time does not depend on each GPU but in the overall program, so as to achieve this new constriction the following pseudo-code has been the base of this new time measurement:

```
auto start = std::chrono::steady_clock::now();

... execute parallel kernels ... # as many threads as devices

auto end = std::chrono::steady_clock::now();
float overall_execution_time = end - start;
```

4 Conclusion

Being aware of the devices that a programmer is going to use is critical for a correct usage of the accelerators; it will allow the programmer coding a well workload-balanced program so as to achieve the best performance possible given an heterogeneous environment. Adaptability is also crucial for programming OpenCl [1] code if it is going to be used in several machines with different devices and possible environments.

In this specific laboratory, two same devices have been used, causing the workload balance to lose the importance it really has. However, forced metrics in Section 3.4 show how a good balance decreases the execution time such in a notable way. Communication must be carefully taken since it can induce such a large undesired bottleneck.

Section 7 exposes how a well used heterogeneous approach can beat the non-heterogeneous widely, nevertheless, it must be well programmed for getting the best possible performance.

References

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