Lab 6 Report

Programming and Architecture of Computing Systems

César Borja and Nerea Gallego

January 5, 2024

1 Introduction

This report delves into the realm of OpenCL programming [1], a framework for unlocking the potential of heterogeneous computing architectures. The aim of this work is to create an execution engine that uses two acceleration devices to solve a problem.

As part of this work, it was necessary to deal with the scheduling of a heterogeneous system and try to balance the workload between the devices.

2 Problem description

The key challenge is to manage image processing that accommodates the computational capabilities of two different acceleration devices housed in the lab infrastructure: a multi-core processor and a graphics processing unit (GPU). The goal is clear: to efficiently distribute the workload between the two devices, improving processing performance while applying an image transformation across an entire image dataset.

This task aims to explore and implement an optimal strategy for the concurrent use of the available computing resources. By applying transformations to the image stream, exploiting the capabilities of both the multi-core processor and the GPU, the goal is to uncover the most efficient and effective approach to workload distribution while achieving accurate transformation results.

3 Main decisions taken

We have decided to issue each image of the stream to a device. Although each task is greater than splitting each frame between the two devices, and could cause imbalance if the images are of different sizes, it is easier at this point to recover the final image.

In terms of the transformation applied, we have decided to use the histogram kernel as it is more complex in terms of computation time, so it would be easier to measure different metrics between the devices.

To achieve the goal of having the most efficient and effective approach to workload distribution, we have done the following approximation: first we send the first task (image) to each of the devices and measure the kernel time. With this first measurement, we assign probabilities to each device according to the execution time. Later, we enqueue all the images to be processed on each device with the probability assigned in the previous step.

To measure the kernel time we have decided to use *cl_events*. Using OpenCL events provides a robust mechanism for accurately measuring the time taken by a kernel execution or any OpenCL operation. It allows you to precisely capture the execution time and allows detailed profiling of individual commands, aiding in identifying bottlenecks and optimizing performance.

4 Analysis of results

This results have been obtained using a 5000 images dataset, all images being the same (Lena, Fig. 1). An example of the histogram created for the Lena image is shown in Figure 2.

Figure 1: Lena test image.

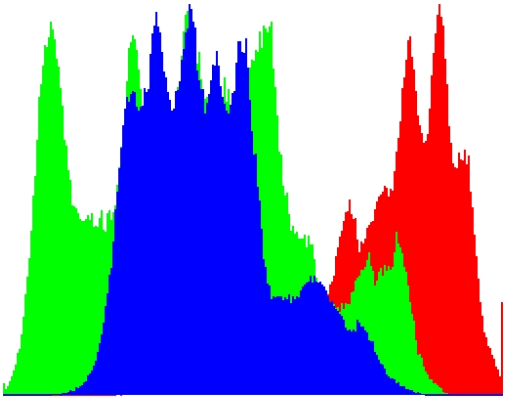

Figure 2: Lena test image histogram.

4.1 Execution times and performance

Figure 3 shows a comparison of total kernel time and overall time between the non-heterogeneous and heterogeneous alternatives.

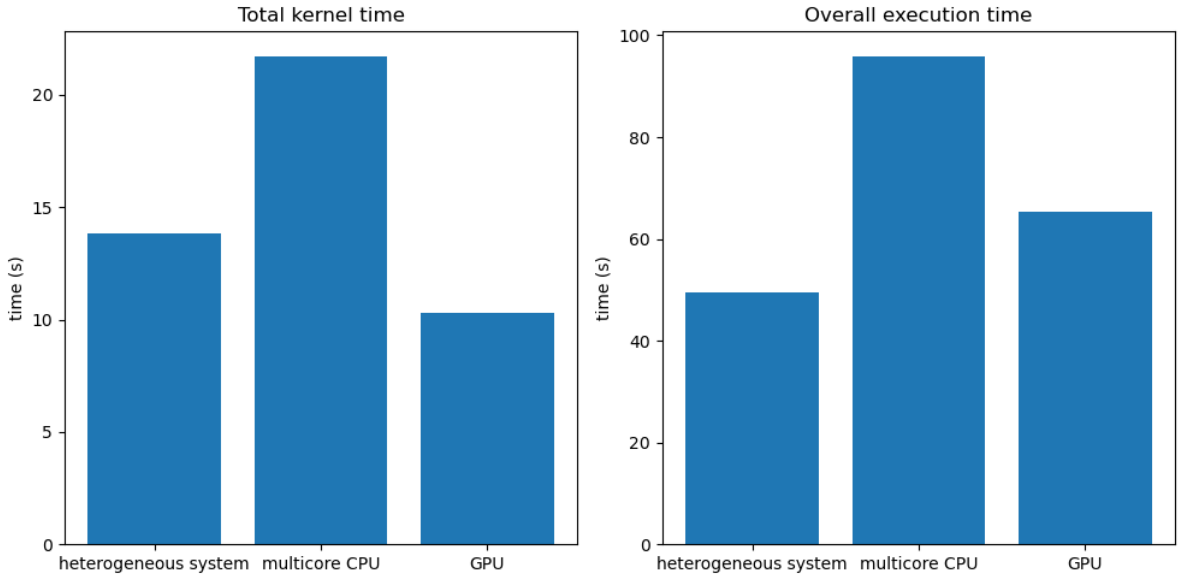

Figure 3: Total kernel time and overall time comparisons. We compare the heterogeneous approach with the non-heterogeneous approaches

.

The first observation we can make is that there is a bottleneck between the CPU and the GPU in terms of kernel computation. The GPU performs significantly faster than the multi-core CPU in kernel computation. We will look at this bottleneck in more detail later in this section.

Focusing on the heterogeneous system, we can see that the overall execution time is the lowest, so the main objective of using two devices to improve performance is achieved. However, the kernel time is slightly higher than the only-GPU version. The reason of this is that the kernel time of the heterogeneous system is the sum of all kernel executions for both devices, but many of these executions occurs in parallel, so some of this time is being measured twice.

Other performance metrics In addition, we have measured other performance metrics that appear in Table 1. Here is a brief explanation of these metrics:

- **Throughput**: Number of pixels processed per second.
- **Writing bandwidth**: Bandwidth from host to kernel.
- **Reading bandwidth**: Bandwidth from kernel to host.
- **Kernel bandwidth**: Bytes processed per second by the kernel.

	multi-core CPU	**GPU**	**heterogeneous**
Throughput [Gpixels/s]	0.060	0.128	0.108
Writing bandwidth [GBytes/s]	3.249	15.091	11.134
Reading bandwidth [GBytes/s]	0.613	0.836	0.821
Kernel bandwidth [GBytes/s]	0.181	0.382	0.285

Table 1: Comparison of other performance metrics over the different system approaches.

By observing these metrics it is clear that the GPU is much faster than CPU in both computation and communication. As expected, the values obtained from the heterogeneous version are close to the GPU, but them remain below as it uses both.

It is important to mention that the writing bandwidth is one order of magnitude higher than reading bandwidth, it is specially higher for the GPU and heterogeneous approaches.

4.2 Workload unbalance

As we have seen previously, the multi-core CPU and GPU do not perform equally. The GPU is overall faster than the CPU. Thus, if we would give both devices the same number of tasks to perform, the GPU will finished much before the CPU will do, producing an important workload unbalance in the application. As we want both devices to be executing tasks as much as possible at the same time, we have to give the GPU more tasks than the CPU.

In section 3 we talked about our balance approach. Basically we give each task to one of the two devices with a probability based on how fast the devices can execute an specific task, assuming that all the tasks are equal and have the same computational cost. This means that if the GPU device can get the histogram of an image 10 times faster than the multi-core CPU device, the next task will be assigned to the GPU with a probability 10 times higher.

The results obtained in the workload balance are assigning the **68.36%** of the images (3418 images) to the GPU and the **31.64%** (1582 images) remaining to the multi-core CPU. This tasks are translated into a **49.90%** of all the kernel execution time corresponding to the GPU and a **50.10%** of the kernel execution time corresponding to the multi-core CPU. That is why we can say that all the devices finish their tasks roughly at the same time.

4.3 Bottleneck of the heterogeneous execution

Bottleneck can be found at different application levels. In the case of our heterogeneous system, there are two different devices working cooperatively, and as we have seen in the table 1, the multi-core CPU performs worse.

Also, we can find bottleneck in a part of our system that could be limiting the most the total performance. Again, if we compare the different bandwidth measured in the heterogeneous system (Tab. 1), we observe how the kernel bandwidth (computation) is significantly lower than the communication measurements. This means that the system have the capability of transfer data between host and the device too much faster than it can process.

4.4 Capacity to adapt to new environments

In this subsection we will talk about two new types of environments: a different set of input images and different transformations to be carried out.

Different images dataset By a "different images dataset" we assume that the images of the dataset are not equal, so they can have different sizes and RGB values. In this case, our workload balancing method can work reasonably well because we set the probabilities by measuring the kernel time for the same image, so the probability of choosing the GPU for a task remains higher. Of course there will be some problems if the application gives the CPU very large images. To try to solve this problem, we might also take into account the image size for setting the probabilities so the bigger the image is, the more probable is to give it to the fastest device.

Different transformations By using different transformations the execution time of the kernel will vary. However, if this kernel time is very low it may difficult the workload balance (two different executions of the application might have very different workloads because of the sensitivity of the first measurement that is responsible of the probabilities assignment).

We have particularly try the **image flip** kernel and it had this issue. In that case, the kernel time for CPU and GPU for the first image were $0.000499s$ and $0.000434s$ respectively so the number of images assigned to each device was roughly a half.

References

[1] *OpenCL*. `https://man.opencl.org/`.