**Multicore Programming**

**Project OpenCL**

**Bilal Laaroussi El Alami**

**0547571**

**Overview**

In the application we port a sequential version that identifies stars to a naïve parallel version, furthermore we have made three optimizations of the parallel program. We have tested for correctness, and executed some experiments.

The files for the naïve version are *solution.py* and *kernels.cl*

The files for the first optimisation are *first\_optimisation.py* and *first\_optimised\_kernel.cl*

The files for the second optimisation are *second\_optimisation.py* and *second\_optimised\_kernel.cl*

The files for the third optimisation *are third\_optimisation.py* and *third\_optimised\_kernel.cl*

The file *generate\_image.py* generates input that can be used to test the solutions

**Porting the sequential algorithm to the naïve parallel version**

The parallel version is composed of three steps. The first one is calculating the greyscale values of the image, the second one is calculating the average brightness of a greyscaled image, the third step is identifying stars.  
  
Step 1: Calculating greyscale values

First, in the host program, the r,g,b values are extracted from the image. Each, a separate numpy array, those arrays are converted to C arrays which will serve as input to the greyscale kernel.

The greyscale kernel takes as input: N: the length of the image array (= the number of pixels) ;float \*R an array of red values; float \*G an array of green values; float \*B an array of blue values and float \*L. In the array L will be stored the greyscale values of each pixel by calculating the weighted sum of the r,g,b values.

Step2: Calculating average brightness

The sum kernel will calculate the sum of a row i of the image and put the result of that sum in Res[i]. The host program will group one work item per row, this by having the global\_work\_size = (HEIGHT,) with HEIGHT the number of rows in the image.

The host program will give the kernel the (previously calculated) greyscaled array as input and an array d\_stars that should store the sums per row. When the kernel is finished executing, the results are copied to a numpy array h\_stars. Finally, we can calculate the average brightness of the image by taking the sum of the partial sums and dividing it by the number of pixels of the image.

Step 3: Identifying stars

The identifyStars kernel takes as input, a threshold, window size, an array \*L of greyscale values, and an array \*IsStars.

A pixel is a star if IsStars[i] = 1 and is not a star if IsStars[i] = 0.

The host program is going to provide these arguments, and in the naïve version each work item will be 1 pixel/index of the image. When the kernel finished executing, the results will be copied to a numpy array h\_stars and the sums of stars will be calculated. Additionally an image will be generated from h\_stars that shows the identified stars.

We used mirroring as edge handling technique. The reason is that if we would take a random pixel of an image in general a small region left, right, up and down of that pixel all look similar to each other, so for the edges it is likely that the same holds true. The functions *correct\_row\_index* and *correct\_col\_index* return the mirrored row or column if the row or column are outside the image and otherwise it just returns the original row or column. The edge handling technique is ported back to the sequential version.

**2.Optimizations:**

**First optimization**

In the first optimization we made a work item bigger, namely a (1x1024) subset of the image.

The index space looks like this

A graph paper with a drawing of a flag

Description automatically generated

Figure 1 rectangular work items

The goal is to have more FLOPS per memory access per work item. When having bigger work items, a lot of memory accesses can be ‘shared’ by the cells of the work item.  
A drawing on a graph paper

Description automatically generated

Figure 2 memory rectangular work item

In this image the green memory area would be accessed once for all cells in the work item so there would be less memory accesses than in the naïve version.

However, in our implementation the green area still gets accessed multiple times per cell in the work item as we simply loop over the cells in the work item, and do the same as in the naïve version, so an efficiency gain is not expected.

**Second optimization**

Having rectangular work items is not optimal because a lot of memory accesses are repeated between work items, this is illustrated by following image.

A graph paper with writing on it

Description automatically generated

Figure 3 memory overlap rectangular WI's

The green area is the memory that is accessed by work item(p,q) etc. (also imagine the coloured lines going through the rectangles)

We can see that there is a big overlap between the green, red and blue area.

This means that all the work items do memory accesses that other work items also do.

Having square work items would significantly reduce the overlap. We have chosen to define work items of 256 x 256 cells.

A graph paper with writing on it

Description automatically generated

Figure 4 memory overlap square work items

We see that for a work item

(Overlapping memory accesses) ➗ (non overlapping memory accesses)

Is a lot smaller than if we would use rectangular work items.

The memory accesses per FLOPS decrease so a higher speedup is expected.

We expect that the size of the work item and the speedup are positively correlated until a certain point where the speedup will start to reduce.

In our implementation each memory area still gets accessed multiple times as we again just loop through the cells of the work item and do the same as in the naïve version. So an addition gain in speedup is not expected.

**Optimization 3**

For the third optimization we started from the second optimization. We first copied the (majority of the) memory that a work item will use into private memory (the array called *lum\_priv*).

A work item then mostly accesses private memory.

The area marked in black are the private accesses, but there are still global accesses, namely the area marked in green, but the global accesses are much smaller than in the previous optimizations.

A drawing on a graph paper

Description automatically generated

Figure 5 private memory global memory

Unfortunately, I couldn’t compile this optimization due to -despite efforts- an unresolved bug.

**Verifying correctness**

I first generated small random images (through *generate\_image.py*) that served as input for the different versions of the program. Then I compared the resulting images that show the stars of the different versions, and compared them through the site <https://www.diffchecker.com/image-compare/> which allows to see the images side by side (*split* feature), seeing the difference between 2 pictures (*difference* feature) and to see the difference through with the *slider* feature which was especially useful. Using this approach I was able to find and solve some bugs.

Then I did the same thing but for the bigger input images that were already provided (like “behemoth” etc.)

Comparison sequential and naïve version.  
The sequential version counts: 22709 stars while the naïve OpenCL solution counts 20429

The resulting image of the sequential version is stored in *results/Sequentialbehemoth-black-hole.png* and the resulting image for the naïve OpenCL version is stored in *results/naïve\_behemoth-black-hole.png*. With the naked eye it is hard to see a difference between the images. But if we take the difference of the images, we see the ~2000 stars that are identified in the sequential version which were not identified in the parallel version

A black square with white dots

Description automatically generated

Figure 6 difference sequential and naive version

Comparison first optimisation and naïve version.

The naïve version and first optimisation produce identical results.

Comparison first\_optimisation and second\_optimisation

Comparing images of the first optimisation version (*results/FirstOptimisedbehemoth-black-hole.png*) and the second optimisation version *(results/SecondOptimisedParallelbehemoth-black-hole.png*) we see a little difference.   
The difference.

A black square with white specks

Description automatically generated

Figure 7 difference first and second optimisation

The white spots show the stars that are identified in the second optimisation that weren’t identified in the first optimisation. What happens is that around the area where stars are and should be identified relative to the first optimisation, a sphere of extra stars are identified around the correctly identified stars. Unfortunately, I have not been able to identify the problem in the code.

**Experiments**

Hardware

* **Apple M1 chip**  
  8-core CPU with 4 perform­ance cores and 4 efficiency cores  
  7-core GPU, 8-core GPU  
  16-core Neural Engine
* Memory: 8GB unified memory
* OS: macOS Monterey 12.2.1

In the first experiment we compared the execution time of the naïve version with the execution times of the parallel versions input image: *behemoth-black-hole*. We repeated each execution of each version five times. Results:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| version | sequential | naive | Optimized 1 | Optimized 2 |
| 1st measure | 13.4045 | 0.0534 | 0.0378 | 0.3406 |
| 2nd measure | 13.2118 | 0.0591 | 0.0574 | 0.3409 |
| 3rd measure | 13.1434 | 0.0418 | 0.0379 | 0.3322 |
| 4th measure | 13.7924 | 0.0529 | 0.0419 | 0.3460 |
| 5th measure | 13.3211 | 0.0481 | 0.0503 | 0.3232 |
| Average | 13.3746 | 0.0511 | 0.0451 | 0.3366 |

As expected, we see that the naïve version is much faster than the sequential version, this because of the parallel on the GPU.

We also see that the first optimized version is slightly faster than the naïve version. This is due to the fact that there are fewer work items in the first optimized version

in comparison to the naïve version where each cell is a work item which will cause a bigger overhead.

We see that the second optimized version is significantly slower than the naïve and first optimisation versions. This is probably because the work items are too big. We made work items of 256x256 = 65536. The behemoth image has dimension 2219x2243 = 4977217 pixels

So there are only 4977217 / 65536 = 75 work items. Which is very low. We should make the work items smaller to have a better speedup in this optimisation.