Project OpenCL: Counting Stars

Multicore Programming 2023

Janwillem Swalens (janwillem.swalens@vub.be)

For this project, you will implement and evaluate a program in OpenCL that counts stars. The program loads a photo of the night sky and counts the number of stars using a simple algorithm. We provide a sequential implementation in Python that you should port to OpenCL. Then, you will try to optimize it using the techniques seen in class. Finally, you should evaluate your program with a set of benchmark images.

Overview

This project consists of three parts: an **implementation** in OpenCL, an **evaluation** of this system using benchmarks, and a **report** that describes the implementation and evaluation.

- **Deadline**: Sunday, **14th of May** 2023 at 23:59.
- **Submission**: Package the implementation, your benchmark code, and the report as a PDF into a single ZIP file. Submit the ZIP file on the Canvas page of the course.
- **Grading**: This project accounts for one third of your final grade. It will be graded based on the *submitted code*, the accompanying *report and its evaluation*, and the *project defense* at the end of the year. If you hand in late, two points will be deducted per day you were late. If you are more than four days late, you'll get an absent grade for the course.
- **Academic honesty**: All projects are individual. You are required to do your own work and will only be evaluated on the part of the work you did yourself. We check for plagiarism. More information about our plagiarism policy can be found on the course website.

Counting stars

The Hubble and James Webb space telescopes have taken many photos of the night sky. These photos can be used to estimate the number of stars in the sky. That will be the goal of this project.

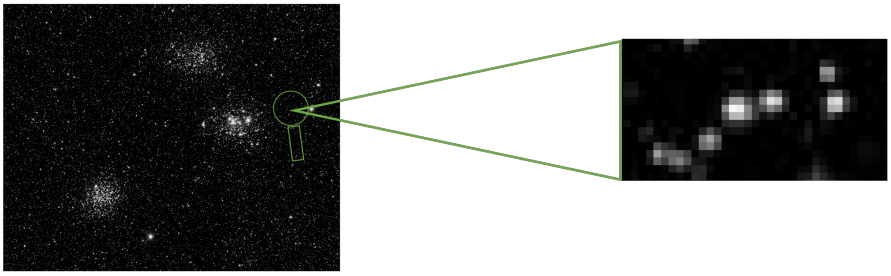

Figure 1: Close up of a part of the night sky.

The algorithm we will use is based on examining the brightness of the pixels in the image. It works as follows:

1. Convert the image to grayscale, using the formula `L = 0.299 * R + 0.587 * G +`

`0.114 * B` for each pixel, where `R`, `G`, and `B` are the red, green, and blue components of the pixel, and `L` is the resulting grayscale value.
2. Compute the average brightness of the image, i.e. the average `L` value of all pixels. We then calculate a *threshold* value, which is 2 times the average brightness.
3. Iterate over all pixels in the image. For each pixel:
    1. Check whether it is brighter than (or equal to) the threshold.
    2. If so, check whether it is brighter than (or equal to) the maximum brightness of its neighbors, in a 7x7 window.
    3. If so, count it as a star.
4. The sum of all matching pixels is the number of stars in the image.

Implementation

We provide a sequential implementation of the algorithm in Python. The program reads in an image from disk, applies the algorithm, and prints the estimated number of stars. It uses the Pillow library to convert images to an array of pixels.

First, you should port this sequential algorithm to OpenCL, to create a "naive" implementation in OpenCL. You should make sure the algorithm produces results that are identical to the sequential algorithm. Make sure to use an appropriate edge handling technique, and port those changes back to the sequential algorithm as well, so that both implementations produce the same results.

Then, you should try to optimize the algorithm. You can create several "optimized" implementations, using the techniques we have seen in class, such as: using private or local memory, tweaking work group sizes, vectorization, combining or splitting multiple kernels, or re-ordering/combining steps in the algorithm. You can try several optimizations and compare them with each other and with the sequential implementation. If performance of one of the "optimizations" is worse, that's okay, the goal is for you to show that you grasp the concepts of GPU programming and how an algorithm can be optimized.

> Tip: to calculate the average brightness, it may be useful to calculate partial averages on the GPU (one per work group), and combine them on the CPU. Similarly, computing the final number of stars can happen on the GPU, CPU, or by computing partial sums on the GPU and combining them on the CPU.

Correctness

You should validate the correctness of your implementations by comparing their results with the sequential implementation. Make sure to apply the same edge handling technique in all implementations.

Performance evaluation

To evaluate performance, you should run your implementations on a set of benchmark images. Several images have been provided as part of this project assignment, but you can add others if you want.

First, you should compare the execution time of all your optimized implementations with the

naive implementation on the GPU, for varying work group sizes.

Next, you should perform at least one of these experiments:

- You can compare the GPU with the CPU: both the Python implementation and the OpenCL implementation(s) running on the CPU.
- You can measure and report the time it takes to transfer data to and from the GPU (and examine the trade-offs for varying image sizes).
- You can compare the performance of your implementations on different images (different sizes, different number of stars, different average brightness).
- You can choose different values for the parameters of the algorithm, i.e. the brightness threshold and the window size.
- You can compare different hardware: Dragonfly vs. your machine. (Especially if you have a good GPU, or a chip like the Apple M1 or M2, a comparison with the server will be interesting. If you have an M1 or M2, also compare the time to transfer the data.)
- If you like, you are also allowed to port your implementation to CUDA (for NVIDIA GPUs) or Metal (for Apple chips), and compare with those implementations. CUDA and Metal are not covered in class, so you will have to do some research on your own, but both have a programming model similar to OpenCL.
- Or you can come up with your own experiments.

You are free to choose whether to focus more on implementing extra optimizations or on doing more performance experiments. Next to the naive implementation and required benchmark above, you can choose to either implement *one* more optimization and do *at least two* performance experiments, or implement *two* optimizations and do *at least one* performance experiment. You can of course do more than that.

Note that you should always optimize for execution on the GPU (and not the CPU). The most important part of the evaluation is that you can explain and reason why your optimizations are effective (or ineffective).

Hardware

For this project, you should run your experiments on "Dragonfly", a server with a high-end GPU available at the SOFT lab. Its specifications are in figure 2. Before running your experiments on the server, you should first run them on your machine or one of the machines in the computer room. Make sure your experiments run correctly and provide the expected results on your own hardware before you run them on the server, as you only have limited time available on the server.

Your report can consist of experiments that you ran on your local machine or the machines in the computer rooms as well as those that ran on the server (see list of experiments above). Especially if you have a good GPU, or a chip like the Apple M1 or M2, a comparison with the server will be interesting.

Include the specifications of the machines you used in your report, as in figure 2, even if you used Dragonfly. To get information about your GPU, you can use the `device_info.py` script from the first exercise session.

Hardware	
CPU	AMD Ryzen 9 7950X (16 cores / 32 threads, at 4.5 GHz base, 5.7 GHz boost)
RAM	128 GB
GPU	NVIDIA GeForce RTX 4090 (16384 cores at 2.625 GHz max; 24 GB memory)
Software	
OS	Ubuntu 22.04.2, running Linux kernel 5.15.0-67-generic
OpenCL	OpenCL 3.0 CUDA 12.0.147

Figure 2: Specifications of the machine Dragonfly

Report

Finally, you should write a report about your implementation and evaluation. Please follow the outline below:

1. **Overview**: Briefly summarize your overall implementation approach and the experiments you performed. (1 or 2 paragraphs)

2. **Implementation**:

    1. **Naive version**: Describe how you ported the sequential algorithm to OpenCL. What does a work item do? How are work items grouped into work groups? What edge handling technique did you use?
    2. **Optimizations**: Describe each optimization. What did you change? How do you expect this to affect performance?

3. **Evaluation**:

    - How did you verify **correctness**?
    - Describe your **experimental set-up**, including all relevant details (use tables where useful):
        - Which hardware, platform, versions of software, etc. you used (even if you used Dragonfly).
        - All details that are necessary to put the results into context.
    - Describe your **experimental methodology**:
        - How often did you repeat your experiments?
    - For your **experiments**, describe:
        - What (dependent) *variable(s)* did you measure? For example: execution time, memory transfer time.
        - What (independent) *parameter* did you vary? For example: the optimization you used, work group size, image size, hardware.
        - *Report* results appropriately: use diagrams (e.g. box plots), report averages or medians and measurement errors. Graphical representations are preferred over large tables with many numbers. Describe what we see in the graph in your report text.
        - *Interpret* the results: explain why we see the results we see. Relate the results back to the changes you made. What is the optimal value for the measured

parameter (e.g. optimal work group size)? If the results contradict your intuition, explain what caused this.