Project OpenCL: Counting cells

Multicore Programming 2025

Janwillem Swalens (janwillem.swalens@vub.be)

For this project, you will implement and evaluate a program in OpenCL that counts cells in microscopic images. The program loads a photo of a blood sample and counts the number of cells 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 For regular students, the deadline is Thursday, **8 May** 2025 at 23:59.
For students official registered as working student for this course, the deadline is Wednesday, **28 May** 2025 at 23:59, although you are recommended to submit earlier.

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.

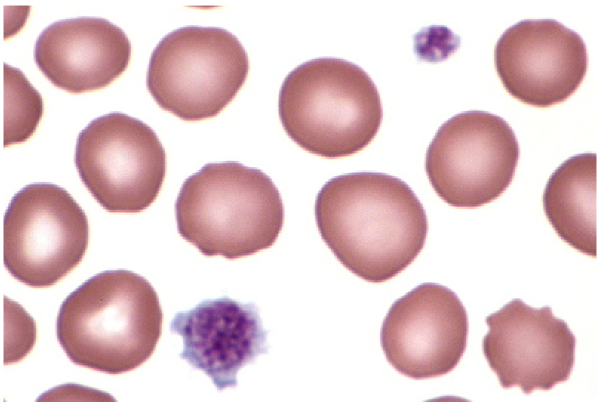
Counting cells

Mammalian blood contains several types of cells, such as red blood cells, white blood cells, and platelets. Platelets (or "thrombocytes") are small cell fragments in the blood that help form blood clots. The number of platelets in the blood can be an important indicator of health.

During a blood test, a sample of blood is taken, as shown in Figure 1a, and analyzed to determine the platelet concentration. This can be done manually, but that is time-consuming and error-prone. An automated system can help to speed up the process and reduce errors. The goal of this project is to implement a cell counting algorithm using OpenCL, leading to a result as shown in Figure 1b.[1]

To convert a microscopic image as shown in Figure 1a to a count of cells, we need to process it to an image as shown in Figure 1b. This consists of a few steps. First, we iterate over all pixels in the image and convert

[1]While our project is inspired by the problem of counting platelets in blood, the algorithm is not an actual algorithm used in practice. It is a simplified version for the purpose of this project.


(a) Blood platelets captured by a microscope.

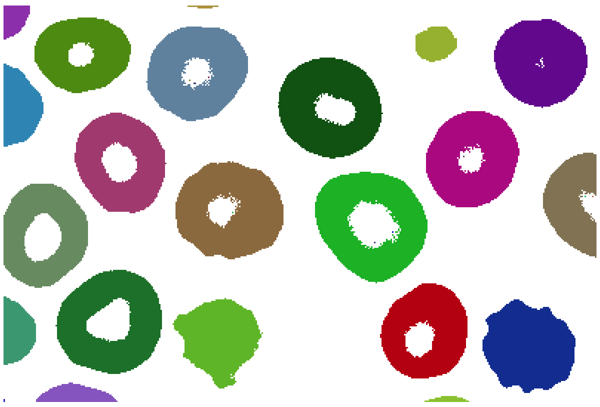
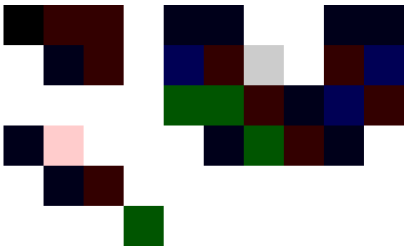
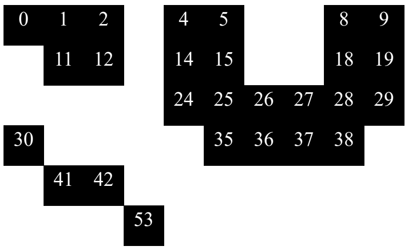

(b) The different platelets/cells, after processing.

Figure 1: Blood platelets.

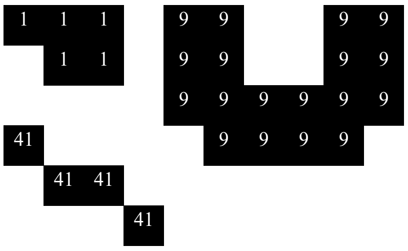
it to a binary image: if the pixel is darker than a certain threshold, a "black" pixel, we consider it part of a cell; otherwise is a "background" pixel. Next, we identify connected regions of "black" pixels. Each separate region of black pixels corresponds to a cell. Finally, we count the number of cells.

The problem of identifying connected regions of "black" pixels is a classic problem in image processing, referred to as "connected component labeling". In this project, we will use a simple algorithm to solve this problem, based on the "union-find" algorithm.


(a) Input image



(b) Mask matrix



(c) Cell numbers

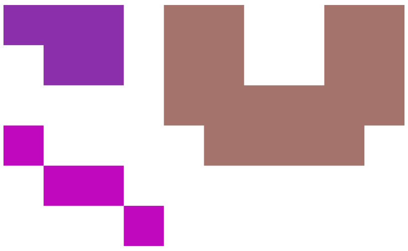

(d) Output image, with colored cells

Figure 2: The union-find algorithm applied to a small image of 'cells'. Figure 2b and Figure 2c were generated using the `matrix_to_svg` function in the Python code.

The algorithm works as follows: (This corresponds to the provided Python code.)

1. We assign a unique label to each pixel in the image. For background pixels (pixels that are not part of a cell), we use the value -1. For pixels that are part of a cell, we use the pixel's 'index' (the row-major index of the pixel in the image). We refer to the matrix of labels as the "mask matrix". This is shown in Figure 2b (white pixels have the value -1, not shown).
2. We create a "disjoint-set" data structure, where each label is a set. Initially, each pixel is a set by itself. Hence, we create a matrix of "parent" values, where each pixel points to itself. We also create a matrix of "rank" values, which is initially 0 for all pixels.

3. We iterate over all pixels in the image. For each pixel:
    1. If the pixel is a background pixel (-1), we skip it.
    2. Otherwise, we check the pixel's previous neighbors (top-left, top, top-right, left). If a neighbor is a background pixel, we skip it. Otherwise, we merge the neighbor's label with the label of the current pixel. We do this by finding the "root" of each set (the pixel with the smallest index in the set) and setting the parent of one root to the other root. If the two roots have the same rank, we increment the rank of the new root by one. This uses a `find` and `union` function from the disjoint-set data structure (described in more detail on Wikipedia and brilliant.org).
4. This algorithm has merged all pixels per connected region. Now, we create a matrix mapping pixels to a single number, by iterating over all pixels and finding their 'root'. The result is shown in Figure 2c. Note that cell numbers are not necessarily contiguous.
5. Finally, we count the number of unique cell numbers.
6. For debugging, we assign a color to each cell number and render the result, as shown in Figure 2d.

> ℹ Note
>
> An alternative approach is to use an algorithm based on the "flood fill" technique, which is also used in the "paint bucket" tool in image editing software. However, this algorithm is much more difficult to parallelize, as it relies on a shared queue. An implementation using this algorithm is provided as a reference in the given Python code, in the function `count_cells_flood_fill`.

Implementation

As part of this assignment, we have provided 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 cells. It uses the Pillow library to convert images to an array of pixels. This library supports a wide range of image formats, including PNG, JPEG, and BMP.

For this project, you should first port this sequential algorithm to OpenCL. We refer to this as the "naive" implementation in OpenCL. You should make sure the algorithm produces results that are identical to the sequential algorithm.

> Caution
>
> When porting the union-find algorithm to OpenCL, be careful to avoid data races in the `find` and `union` operations. You may need to use atomic operations or other synchronization mechanisms to ensure correctness.

> Tip
>
> When porting the algorithm, you can choose how to divide the algorithm in steps and corresponding kernels. You can also handle some steps on the CPU using Python (e.g. step 4 above), if this is more efficient. Be sure to motivate these choices in your report.

Next, you should optimize your naive implementation. You can create several optimized implementations using the techniques we have seen in class, such as:

- using private or local memory,
- optimizing based on the work group size,

- how the work is divided among work items and work groups (e.g. row or column-based, using tiles, using a “checkerboard” pattern),
- how data is loaded into memory (e.g. data types),
- optimizing memory access patterns,
- vectorization,
- changes to the algorithm, including combining or splitting multiple kernels, or re-ordering/combining steps in the algorithm. (Possibly implementing executing some steps on the CPU and others on the GPU).

You are expected to implement at least one optimization.

If performance of an “optimization” 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. In the evaluation, explain why the optimization did not work as expected.

Note that you should always optimize for execution on the GPU (and not the CPU).

Correctness

You should validate the correctness of your implementations by comparing their results with the sequential implementation. If you modify a part of the algorithm, make the same changes to the Python code.

Performance evaluation

To evaluate performance, you should compare the execution time of your naive and optimized implementations, using different images. Several images have been provided as part of this project assignment.

Optionally, you can also measure the time it takes to transfer data between the host and the device; compare with the Python implementations (also the flood fill one); or compare with different hardware (your own GPU, or the OpenCL implementations running on the CPU).

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 Table 1. 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.

Further, you can also include results from running on your local machine or the machines in the computer rooms. Especially if you have a good GPU, or a chip like the Apple M-series, a comparison with the server can be interesting.

Include the specifications of the machines you used in your report, as in Table 1, even for Dragonfly. To get information about your GPU, you can use the `device_info.py` script from the first exercise session.

Table 1: Specifications of the machine “Dragonfly”

Hardware	
CPU	AMD Ryzen Threadripper 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 (Linux kernel 5.15.0-67-generic)
OpenCL	OpenCL 3.0 CUDA 12.0.147

Report

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

1. **Overview** (1 or 2 paragraphs): Briefly summarize your overall implementation approach and the experiments you performed.
2. **Implementation** (up to 2 pages excluding figures):
    1. **Naive version**: Describe how you ported the sequential algorithm to OpenCL. How is it split into kernels? What does a work item do? How are work items grouped into work groups?
    2. **Optimizations**: Describe each optimization. What did you change? How do you expect this to affect performance?
3. **Evaluation** (up to 2 pages excluding figures):
    - 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 for 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 **experiment**, 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, 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.

Remember that shorter is better: edit your text to make it clear and concise.