VRIJE
UNIVERSITEIT
BRUSSEL

PROJECT ERLANG: COUNTING CELLS

Multicore Programming

Daan Hensmans
Daan.Hensmans@vub.be
0564284

Submitted: 4 May 2025

Docent: Prof. dr. Janwillem Swalens

Science and Bio-Engineering Sciences

Contents

1 Introduction

This is the second project for the multicore programming course. The objective of this project was to construct two versions of an union algorithm in OpenCL. The first version is the naive one, and the second one has optimisations that the naive one doesn't have. The subsequent section presents an overview of the components and their respective functions, and information regarding the experiments. Subsequently, Section 3 presents a high-level overview of the architecture implemented. Section 4 will provide a detailed discussion and analysis of the experiments conducted as part of the project.

2 Overview

The openCL versions will be a direct translation of the sequential algorithm, with the exception of the removal of the ranks. A variety of optimisation techniques are employed with the objective of reducing execution time. The following changes have been made: the method of assigning array elements to workgroups has been modified, the optimal workgroup size has been measured, and some of the variables' data types have been changed to smaller ones.

These optimisations are measured separately and compared to the naive implementation to see if they have significantly reduced the execution time. The speedups over the sequential algorithm and the naive implementation are also discussed.

3 Implementation

The most significant difference from the sequential implementation is that the unionize by ranks algorithm is modified by removing the ranks. Instead of choosing the root by looking at the rank, it will now be chosen by looking which root has the biggest global index. This change was chosen to circumvent race conditions in the ranking and to minimize memory allocation by eliminating the need for an additional array to be copied. The absence of ranks gives rise to a problem, namely that the trees in the parent array are not balanced. The solution to this issue is to do path compression subsequent to the execution of the unionize kernel.

The initial part of this section will address the architecture of the naive implementation, followed by a discussion of the optimizations in the optimized version in Section 4.1.

3.1 Architecture

The naive version is comprised of three distinct files. The subsequent list offers a concise overview of each file:

- **main-naive.py:** This file only includes one function, and this is to call the unionize algorithm with a set of arguments.
- **algorithm-naive.py:** Will call the algorithm by first creating all the necessary kernel components, allocating and copying memory to the GPU, and then calling the kernels in the right order and parameters.
- **algorithm.cl:** Includes all the kernels for the algorithm.

The file *algorithm.cl* includes the following (kernel) functions that will be used for the unionize algorithm.

- **convert-to-black-and-white:** Converts the pixels into black and white by taking the average of all the RGB(a) values.
- **give-pixels-unique-numbers:** Creates an unique number for all the pixels, if the mask value is under a certain threshold.
- **unionize-cells:** Unionizes a cell and one of its neighbor cells. If there is a race condition, try again until there is none.
- **compress-parent:** Compresses the trees in the parent array by finding the

root for every array element, and giving that as a new value for the element.

- **update-cell-numbers:** Give all the elements in the same set the same number. The result is put in the already-used mask array.

These kernels are called inside the file *algorithm-naive.py*. The sequential steps that the Python part of the algorithm will take are the following:

1. Allocation of the host memory.
2. Allocation of the device memory.
3. Creating the callable kernels.
4. Calling the *convert-to-black-and-white* kernel.
5. Calling the *give-pixels-unique-numbers* kernel.
6. A loop that will iterate over all the possible neighbor locations of a cell. In the body of the loop the *unionize-cells* kernel will be called, and after that the *compress-parent* kernel.
7. Calling the *update-cell-numbers* kernel.
8. Copying the result array in the host memory.

In total three different memory arrays will be allocated to make the program work. The first array is the input source of the image, wherein all the RGB values are specified for each individual pixel. The second one is an array to hold the parent locations of every cell. The final array is the mask array, which contains the mask of the image and will also contain the result of the algorithm at the conclusion of the program.

The work groups are grouped by the global index. Consequently, the global index will be used to do the operations on the arrays at that index. Therefore, the work groups are grouped over the 2D matrix in the same row or in neighboring rows, as illustrated in Figure 1.

3.2 Optimizations

The optimised version incorporates a series of modifications with the objective of reducing the total execution time.

1. The first optimization is to change some of the kernel arguments to a different data type. The first argument data type that has undergone modification is the source array of the image. It is notable that this can only include values between 0 and 255; consequently, the data type has been altered to uint8. This reduces the values of the array from 32 bytes to 8 bytes. The aim of this modification is to accelerate the processes of copying and performing operations on the array. It should be noted that other arguments have been converted to 8 bytes, including the colour threshold and the offsets of the cells that will be used to find the neighbour.

2. The second optimization involves the combination of two kernels into a single one. In particular, the functions for converting images to black and white, and for assigning unique numbers to pixels, have been combined. It is possible to execute this process within a single kernel, which has the potential to decrease the overall execution time by reducing the number of kernels with which the host must communicate with the device.

3. While naive version will automatically select a group size, the optimized version will have its optimal group size measured and then applied. The objective is to achieve a group size that is superior to the one selected by the program. The group size of the optimized version will be a square of a number (due to the third optimization).

4. The last optimization focuses on the method by which groups are grouped. In the naïve version, the grouping will be based on the global index. This configuration of the work group results in its resemblance to a line in a two-dimensional matrix. As demonstrated in Figure **??**, in the optimized version, the work groups will be arranged in a way that they appear in the 2D matrix as a square, as illustrated in Figure 2. The objective of this is to improve the effectiveness of the caching. It is evident that cells belonging to the same set are likely to share a common root in the parent array. It is more probable that these sets will contain a blob-like form within the image, as opposed to a linear configuration. It is therefore more intriguing that the work groups also possess a form that is capable of encompassing a significant proportion of this

set. Should a large proportion of a set reside within a work group, it can be expected that a significant number of parents will be cached, thereby reducing the time required to access them.

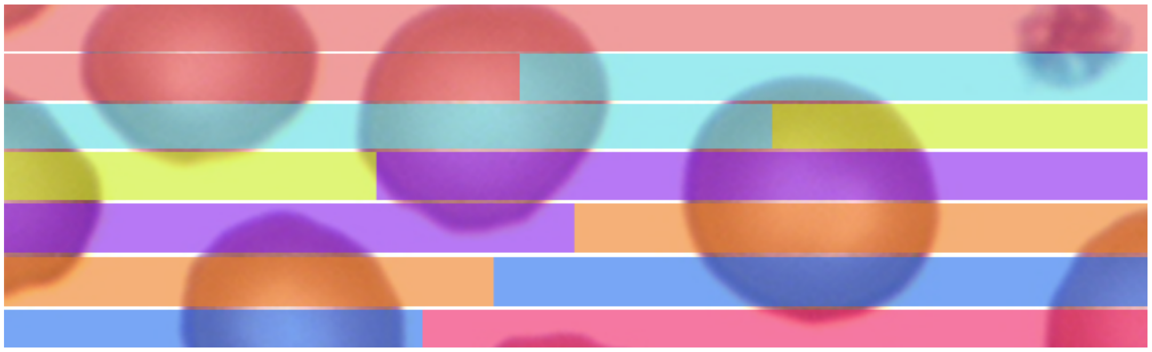

Figure 1: Workgroup layout of the naive version.

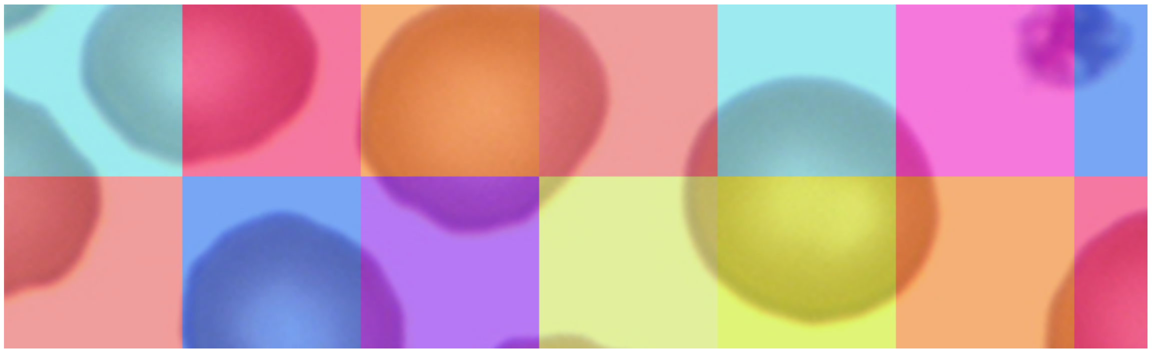

Figure 2: Workgroup layout of the optimized version.

4 Evaluation

In Section 4.1 the context of the experiments and the platform on which they were executed is provided. Subsequently, Section 4.2 will discuss what the optimal work-group size is. Finally, the experiments conducted are examined in Section 4.3.

4.1 Experiment Context

Firstly, it is necessary to provide some context about the experiments. It is to be noted that all experiments conducted on the OpenCL implementations will be executed 10,000 times each, while the sequential algorithm will comprise of 30 runs for each experiment. All experiments will be conducted on all the provided images.

Image Name	**Dimension**	**Pixel Count**	**Cell Count**
Thrombocytes	172x320	165,120	6
Plaquette geante	267x400	320,400	31
Plaquettes normales	400x600	720,000	71
Anisocytose plaquettaire	400x600	720,000	54
Neutrophils monocyte	858x858	2,208,492	1567
Eosinophil blood smear	1443x1584	6,857,136	199
Platelets2	1920x2560	14,745,600	4,411
Patology of platelet	2056x2464	15,197,952	27,787
Thrombocytosis	2048x2560	15,728,640	3,730
RBCs Platelets WBC in PBS 2	2250x4000	27,000,000	2,588

Table 1: Attributes of the images used in the experiments.

It was determined that conducting the experiments on Dragonfly was unsuccessful due to issues with the OpenCL version on the server. Therefore, it was agreed upon with the professor that the experiments would be carried out at home. The computer specifications can be observed in Table 2 and Table 3.

Type	
CPU	13th Gen Intel(R) Core(TM) i7-13700KF, 3400 Mhz, 16 Core(s), 24 Logical Processor(s)
RAM	32 GB
GPU	NVIDIA GeForce RTX 4070 Ti (7680 cores at 2310 MHz; 12 GB memory)

Table 2: Hardware specifications of the PC.

Type	
OS	Microsoft Windows 11 Pro (Version 10.0.26100 Build 26100)
OpenCL version	OpenCL 3.0 CUDA

Table 3: Software specifications of the PC.

Each experiment will have the runtime as the dependent variable. The elapsed time is measured at various points throughout the program. For instance, the times between kernel calls, the times to allocate memory, the times between transferring memory from host to device, and vice versa are all measured. The measurement of elapsed time is determined by the Python timer function, which quantifies the monotonic time in microseconds. For every graph, the median will be used of those 10,000 (or 30 if sequential) runs.

To ensure the soundness of both the naive and optimized versions, the cell count of the result is measured at the completion of each run. In the event that the total number of runs is equivalent to the sequential count, the program is determined to be valid for that particular image, and consequently, it will be stored within the designated file that it has the same cell count. Following a review of the experimental results, it was determined that both the naive and optimized versions would yield equivalent outcomes to the sequential version.

4.2 Optimal Workgroup Size

The subsequent experiment will determine the optimal workgroup size, with the objective of minimizing execution time for subsequent experiments. The experiment will be conducted in two steps. First, the execution time of various image sizes in the optimized implementation will be measured. Second, the experiment will determine the optimal workgroup size. The results of this experiment are displayed in Figure 3. The dependent variables of the experiment are workgroup size and the image used. The independent variable is the measured time between calling the kernel functions and the completion of said functions. In the graph this will be displayed as the speed-up compared to the result of the experiment with workgroup size one. Due to the implementation of square workgroup optimization, the workgroup sizes are constrained to be a square of a specific number. The potential dimensions are listed as follows: $1^2, 2^2, 4^2, 8^2, ..., 32^2$ and so forth, up to 31^2. The process is terminated at 31^2 because the maximum workgroup size of Dragonsfly's GPU is 1028. To decide the optimal workgroup size for all the images, the median is calculated of the iterations from all the experiments, and then the mean is taken from those medians. The result of this can be seen in Figure 3. The analysis of these times revealed that the average optimal workgroup size out of all the results is 16. This value will be established as a constant for the subsequent experiments of the optimal variant.

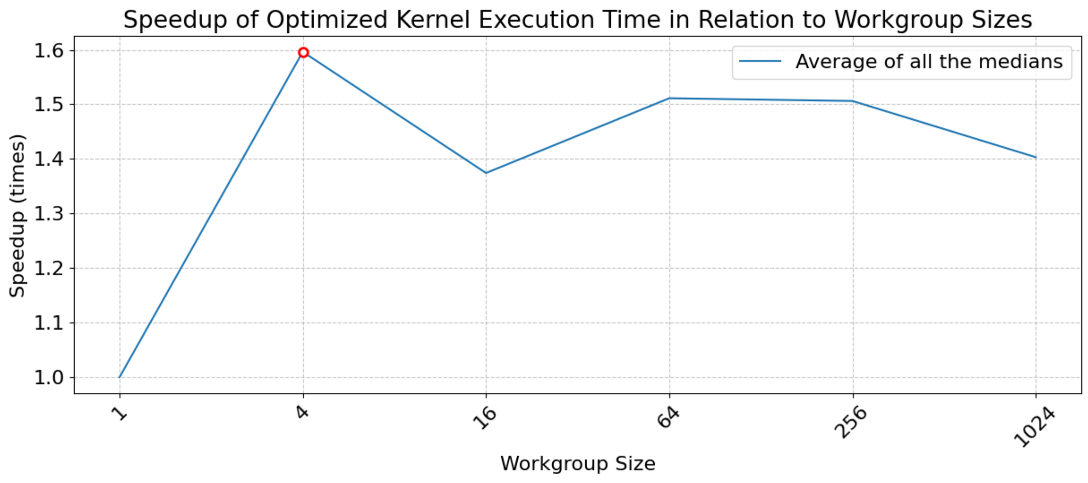

Figure 3: Average speed-up of kernel execution time in relation with the size of the workgroup size.

4.3 Experiments

All of the following experiments will have as independent variable the image, and as dependent variable the measured time between a step, or the full algorithm. What sort of time measured will be specified in the paper and title of the graph. The following list will discuss the results of the experiments that look at individual (or two in the third experiment its case) optimizations in comparison with the naive version.

Initially, a comparison is made between the time between allocating and transferring data for the naive and optimized versions. This is done to ascertain whether modifying certain data types has any impact on the transfer time. As illustrated in Figure 4. The images have been arranged in descending order of size. The outcomes of these two groups are comparable, with a speed improvement ranging from 1.60 to 2.50. The sole outlier is the image titled ”Neutrophils monocyte 16694967012.” One potential explanation for this phenomenon is the possibility that an unrelated application was in active use during the measurement of the naive version for the given image. The results indicate that modifying the data type resulted in a significant impact on the rate of data transfer into the device memory.

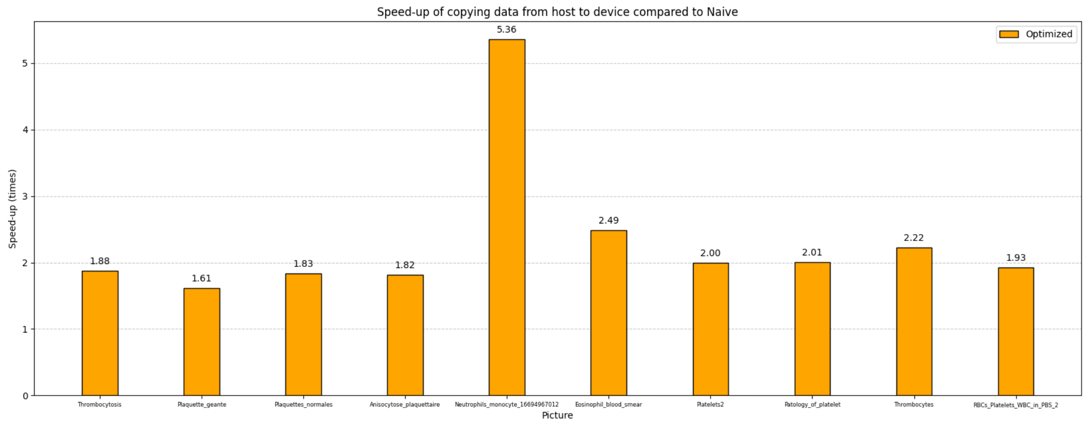

Figure 4: Speed-up of copying the data from host to device, compared to the Naive implementation.

The second experiment will examine whether combining two kernels into one results in a reduction in execution time. The premise underlying this approach is that it would facilitate a reduction in the amount of communication required between the host and the device, thereby increasing efficiency. As illustrated in Figure 5, the speed-ups of the optimized version can be observed in comparison with the naive one. The dependent variable will be the execution time of the kernel functions ”convert-to-black-and-white-kernel” and ”give-pixels-unique-numbers-kernel.” The speed-ups will range from 1.64 to 5.69. Therefore, it can be concluded that the combination of the first two kernel functions into a single kernel function resulted in a substantial increase in speed.

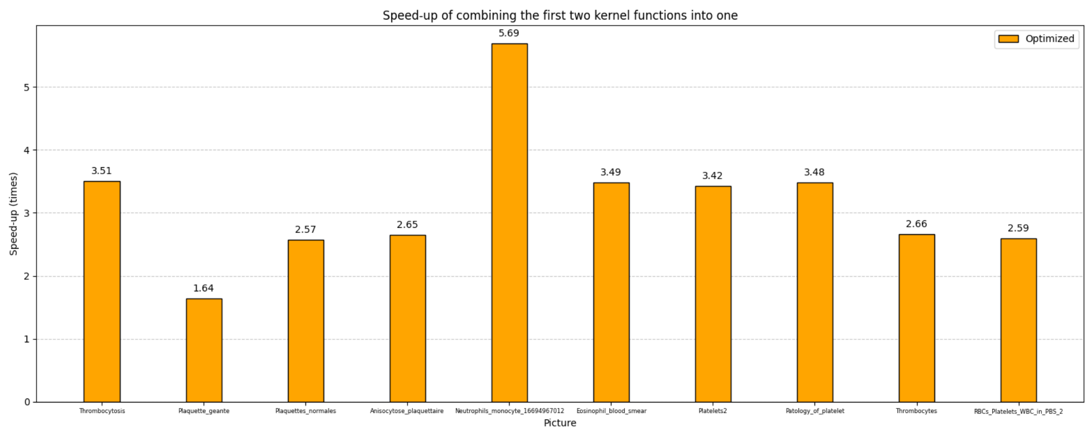

Figure 5: Speed-up of copying the data from host to device, compared to the Naive implementation.

The subsequent experiment will examine the total time required for all kernels to execute. This measurement is taken for both implementations and then compared to examine the differences. The objective of this study is to ascertain whether modifying the workgroups as squares in the two-dimensional matrix has a positive impact on the program's performance in comparison with the naive approach. Furthermore, the determined workgroup size, as established in a prior experiment, and the combination of the two kernel functions into a single kernel will also influence the execution time. As illustrated in Figure 1, the speed-ups are expected to range from 1.13 to 1.45. The image titled "RBCs-Platelets-WBC-in-PBS-2" is presented as an outline with a speed of less than 1. The reason for this phenomenon remains unclear. It can be concluded from these results that the combination of the previous three kernel optimizations had a positive effect on the speed-up.

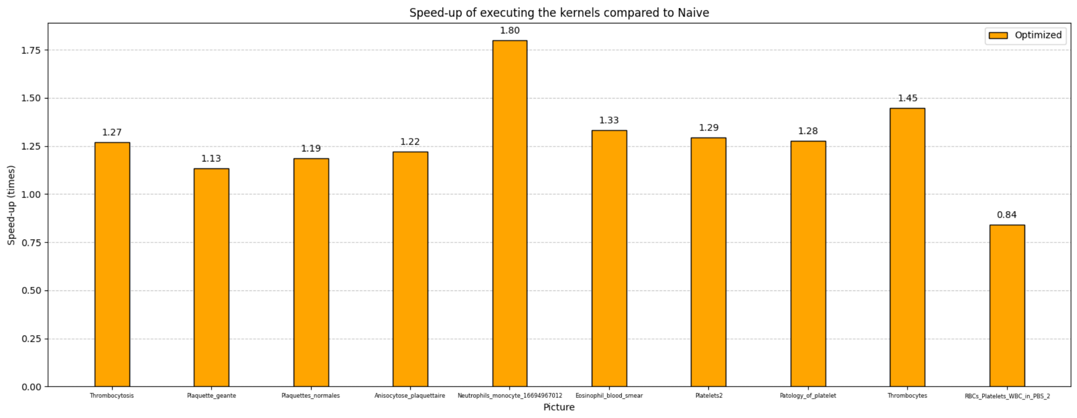

Figure 6: Speed-up of executing all the kernels, compared to the naive implementation.

The final experiment is conducted to determine the extent to which the optimal and naive versions outperform the sequential one, and to quantify the impact in execution time when the optimizations are applied. The duration of the program is measured from the moment of initiation to the moment of completion. As illustrated in Figure 7, the boxplots of the execution time for the naive and optimized versions of the program are presented. As illustrated in the image entitled "RBCs-Platelets-WBC-in-PBS-2," there is a considerable amount of variation in the measured time. This variation can account for the conflicting results observed in previous experiments. A possible explanation for the variation is because the image is the smallest. As illustrated in Figure 8, a speed-up can be seen from the naive and optimized implementation in comparison with the sequential one. The speed-up of the naive will range from 100.73 to 546.97. The optimized speed-up will range from 131.71 to 987.24. As illustrated in Figure 9, the speed-up of the optimized compared to the naive is shown. The speed-up will range from 1.18 to 1.79.

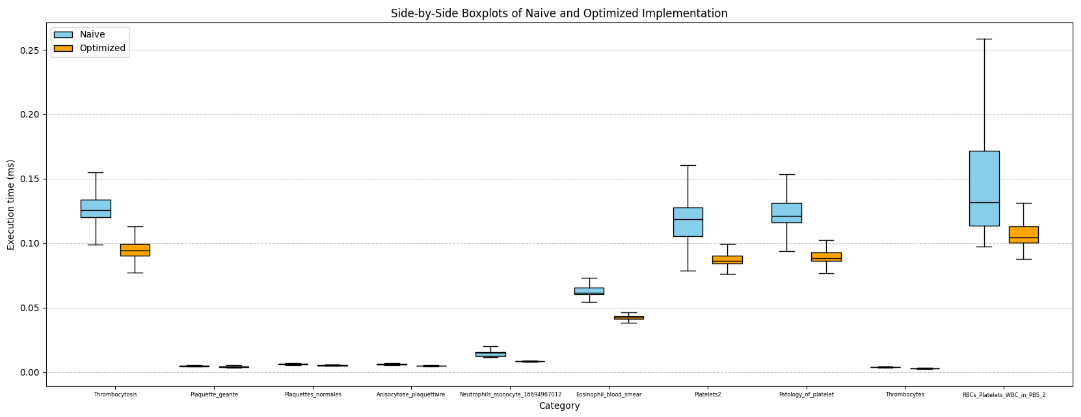

Figure 7: Side by side box-plot comparison between the measured execution time of the naive and optimized version.

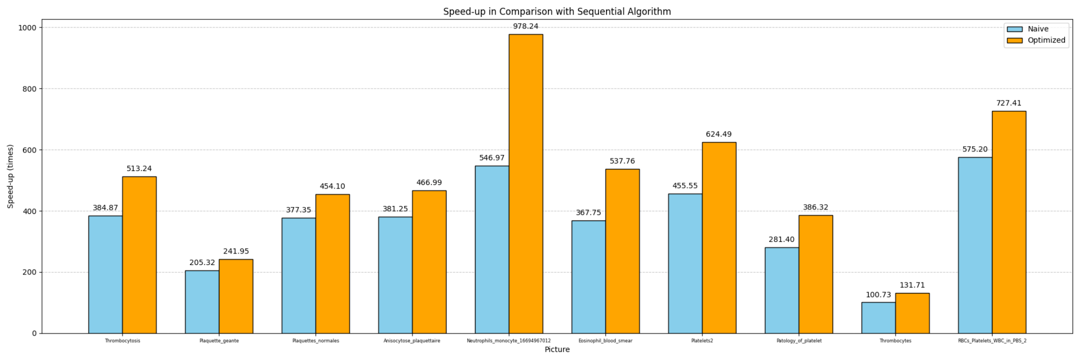

Figure 8: Speed-up of executing the program of the naive and optimized version, compared to the sequential implementation.

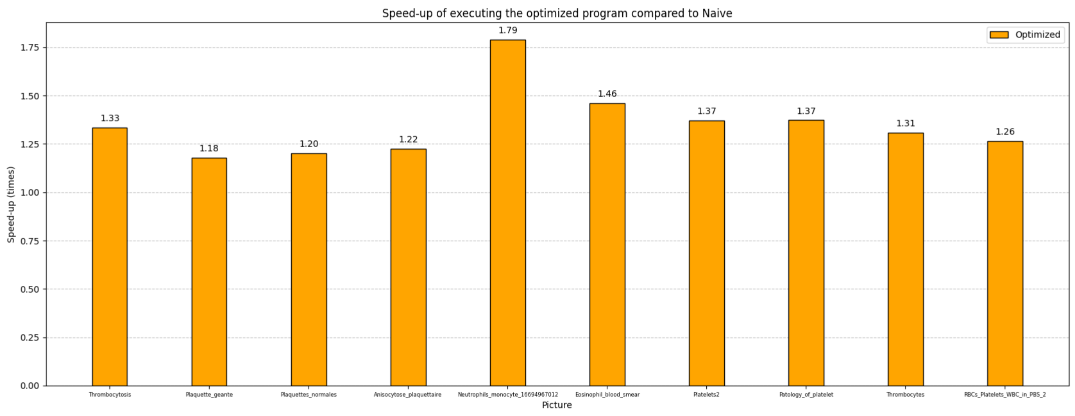

Figure 9: Speed-up of executing the optimized program, compared to the naive implementation.

5 Conclusion

Because of OpenCL the naive version is considerably faster than the sequential one. Around 100.73 and 546.97 times fast. After all the experiments, it can be noticed that each optimization has a noticeable effect on increasing the speed of the naive implementation. Therefore, it can be concluded that the accumulation of all implemented optimizations will increase the speed of the naive version. The speed-up because of the optimizations on the naive implementation will range from 28 percent up to 79 percent.