KIV/PPR

Classification of statistical distributions

Jakub Šilhavý
A21N0072P
silhavyj@students.zcu.cz

11. 11. 2022

Contents

1 Assignment description

The application takes as a parameter a path to a binary file which is interpreted as a collection of `double` values. The goal of the program is to determine whether the input data comes from the Normal, Uniform, Exponential, or Poisson distribution. As a result, the user is presented with the final answer as well as a short reasoning as to why the application came to this particular conclusion.

It was required that the execution does not exceed a memory limit of `1GB` or a time limit of 15 minutes.

The user can also specify in which mode the application should run. These modes affect how the program is executed. The application can be run in the `SMP` mode, which does not take advantage of any `OpenCL` devices. Instead, the entire code is executed on a symmetrical multiprocessor. Using the `OpenCL Dev` mode, the user can list out different `OpenCL` devices that should be involved in the program execution. Finally, the `ALL` mode uses all available resources found on the machine. Essentially, it combines the `SMP` mode and the `OpenCL Dev` mode.

2 Analysis

Before coding, it is important to have a great mental picture of how the problem is going to be tackled. As it turns out, there are several approaches to solve the problem. In this chapter, I am going to lay out a few options along with what solution I eventually decided to go with.

2.1 One-pass approach

Generally speaking, the input file should be processed as few times as possible, ideally once. As we can see on the Wikipedia page, each distribution can be distinguished by its corresponding Skewness [2] and Excess kurtosis [3]; they differ with every distribution. As pointed out by John D. Cook, PhD in his article [1], these statistics can be calculated in a single pass, giving us ultimately the fastest solution of the problem.

However, the algorithm does not work well with large numbers. If the input values are not within a reasonable range, the calculated data will most likely overflow due to limited `double` precision [1]. Therefore, it can be concluded that even though the algorithm is theoretically fast, it is not robust.

2.2 Two-pass approach

Another way to find out what distribution the input data comes from is to use statistical tests. Perhaps the biggest disadvantage of this kind of approach is that the input file has to be read at least two times, which negatively impacts the performance of the application.

On the other hand, it can be implemented in a way that is much more robust than the single-pass approach 2.1, meaning it produces correct results regardless of the input data looks like. In this section, I am going to describe the idea behind the Chi-Square goodness of fit test, which I decided to implement within this assignment [4].

[1] A single multiplication of `x` and `y` may overflow. For instance, when `x` = `-DBL_MAX` and `y` = `DBL_MAX`.

2.2.1 Chi-Square goodness of fit test

This principle of this method is to test whether measured data comes from a theoretical distribution by either accepting or rejecting the null hypothesis [5]. In other words, it calculates the difference between the measured data, in this case the data read from the input file, and a theoretical distribution with parameters derived off of the measured values.

Calculating basic statistics

Let us assume that the input data looks something like this:
`1.5, -5.55, 3.1, 100, -1.0, 0.015, 215.8, ...`

First, we need to calculate the following statistical values `min`, `max`, `mean`, and `variance`.

These values are used to both create a histogram and calculate the parameters of the statistical distribution. It is important to mention that not all input values may be valid `doubles`. This is why we need to read the input file twice. We can calculate the `min`, `max`, and `mean` in the first iteration without any major difficulty. However, to calculate the `variance` and construct the histogram, we already need to know what these values are. Hence, they must be calculated in the second iteration.

Histogram

As mentioned previously, we need to create a histogram that will represent how the input data is distributed. The histogram will be then used to calculate the difference between the theoretical distribution and the input data.

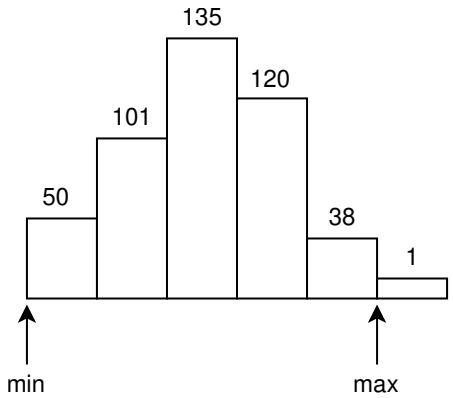

Figure 2.1: Example of a histogram

Before inserting the values into their corresponding bins. We need to decide how many bins the histogram will contain in total.

In this assignment, I took an inspiration from an online article [6] that recommends to set this value to $2n^{2/5}$, where n is the number of valid `doubles` in the input file.

As far as the memory constrain is concerned, let us say that the input file contains 13,421,772,800 valid `double` values (`100GB` of data). Using the formula above, we can see that the number of bins needed is $51,688$. Assuming we use `size_t` to represent a single bin, the overall size of the histogram on a `64-bit` machine is still less than `1MB`. Therefore, we do not have to worry about the histogram exceeding the given memory limit.

Based on the number of intervals, and the `min` and `max` values, we can work out the width of a single interval as
`(max - min) / number_of_intervals`. As we process the file the second time, we insert every valid `double` into its corresponding bin using the following formula `(value - min) / interval_size`. This formula causes the very last bin to have only one value in it, the `maximum` itself. This would introduce a significant error when it comes to calculating the Chi-Square value. Therefore, the last interval is omitted.

CDF function

Before running the statistical test itself, we need to implement a CDF function [7] for each and every theoretical distribution we want to compare the data to. CDF stands for a cumulative distribution function and it expresses the probability that a random variable χ will take a value less than or equal to x.

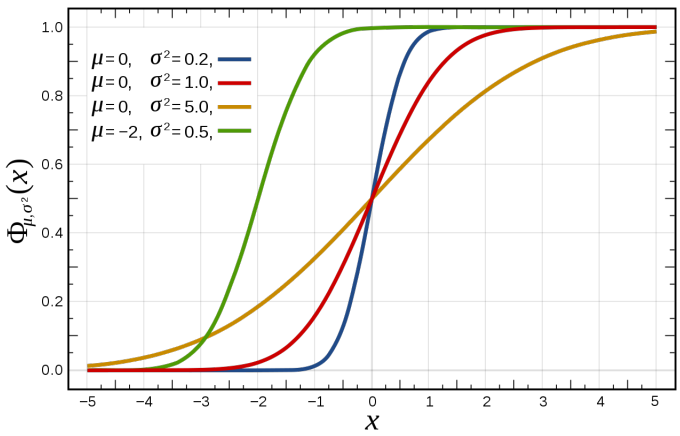

Figure 2.2: CDF function of the normal distribution [7]

Using the CDF function, we can calculate the expected number of values in each interval as `(CDF(x) - CDF(x_prev)) * count`, where `count` is the total number of values in the histogram.

Chi-Square test

When running the Chi-Square goodness of fit test, we need to sum up squared differences between the expected value E_i and the actual value O_i in every bin of the histogram.

$$\chi^2 = \sum_{i=1}^{n} \frac{(O_i - E_i)^2}{E_i}$$

The actual value for each bin is represented by the total number of values in it. To calculate the expected value, we use the CDF function of the corresponding distribution we are testing. The parameters of the CDF function were calculated in the first iteration 2.2.1.

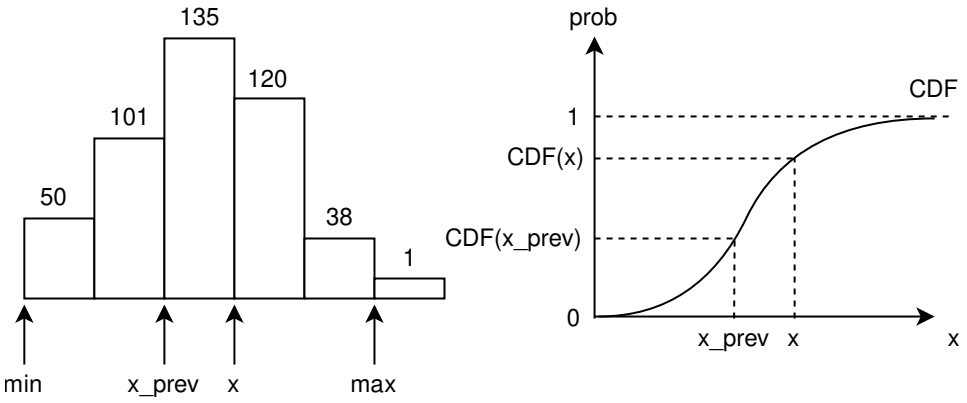

Figure 2.3: Chi-Square goodness of fit test

It should be noted that if the expected value E_i is less than 5, the current bin needs to be repetitively merged with the next one until the condition is satisfied. This is something that is required by the statistical test itself.

At the end, we end up with Chi-Square value which represents the total error of how much the data differs from the theoretical distribution.

P-value

As the intervals of the histogram are merged differently depending on the expected value E_i, we cannot make any definitive conclusions as to what distribution the data comes from. To put it simply, the histogram does not

look the same for each of the tests, hence we cannot make a unified conclusion as the input may vary with each test.

The P-value [8] represents the probability of accepting the null hypothesis. It accounts for degrees of freedom (DoF) and the already calculated Chi-Square error. The DoF value can be expressed using the following formula `DoF = number_of_intervals - estimated_parameters - 1`. The number of estimated parameters depends on the type of distribution. For instance, the normal distribution has two parameters μ and σ^2.

If the P-value is less than a critical P-value, we accept the null hypothesis. Otherwise, we reject it. Ideally, there should be only one test accepted and the rest should be rejected.

3 Implementation

The entire application was implemented in `Visual Studio 2022` using `C++17`. In this chapter, I will go over some challenges I was facing during implementation. I will also explain how the application implements parallelism. For more details of how the algorithms are implemented, please see the commented source code.

3.1 File reader

The input of the program is handled by the `CFile_Reader<double>` class which provides a unified thread-safe access to the input file. It is used by worker threads when processing the input file. The template parameter determines how the input data is interpreted.

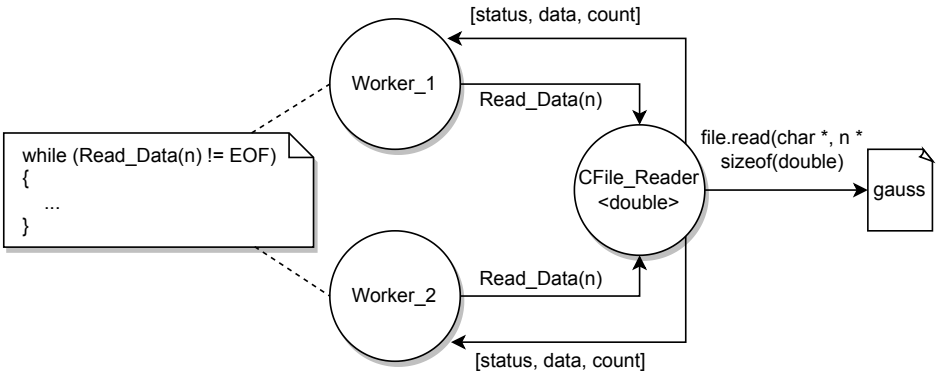

Figure 3.1: Context of `CFile_Reader<double>`

The return value of the `Read_Data` function is the following structure. It is important to point out that due to the size of a data block read off the disk, the values are purposely stored on the heap.

```
struct TData_Block
{
    NRead_Status status;
    size_t count;
    std::shared_ptr<T[]> data;
};
```

3.2 Resource manager

The purpose of the `CResource_Manager` class is to hand out resources to individual worker threads to process the data.

Whenever a worker thread is created, it asks `CResource_Manager` if there is an `OpenCL` device available, assuming the program was run in the `All` or `OpenCL dev` mode. If there is no device available and program was run in the `All`, the data is processed on the `CPU`. Otherwise, the worker thread finishes as the user wishes to use only `OpenCL` devices [1].

3.2.1 Resource guard

The `CResource_Guard` class is just a wrapper that frees an `OpenCL` device upon destruction of the object, so the workers do not have to release the device manually when they are done.

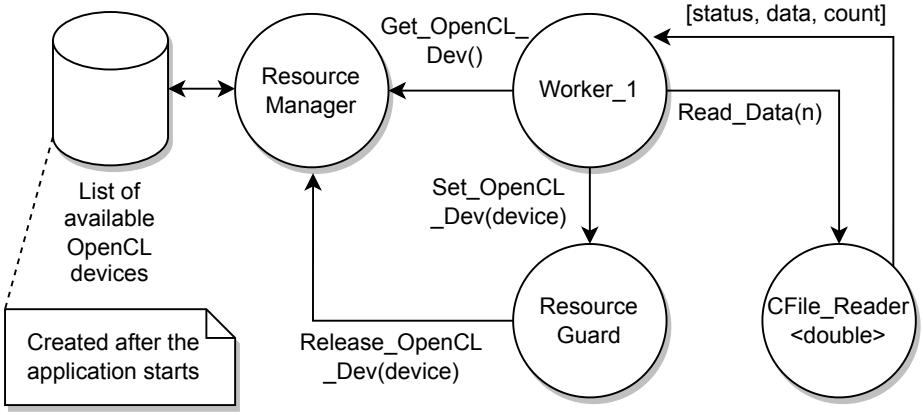

Figure 3.2: Context of `CResource_Manager` and `CResource_Guard`

3.3 Watchdog

The `CWatchog` class is there to check whether or not the program is running correctly. Each worker thread is supposed to call its method `Kick(size_t count)`, which takes as a parameter how many values the worker thread has processed since the last call of the method. This value is added to the watchdog counter. Eventually, this counter value should equal to the size of the input file.

[1] The program supports only those `OpenCL` devices which support double precision.

The watchdog is a separate thread that periodically checks if the counter has changed. If it has, it goes to sleep for a number of seconds which can be specified using the `-w` parameter 5.3. If it has not, the watchdog will print out a warning message saying that the program may not be running correctly [2].

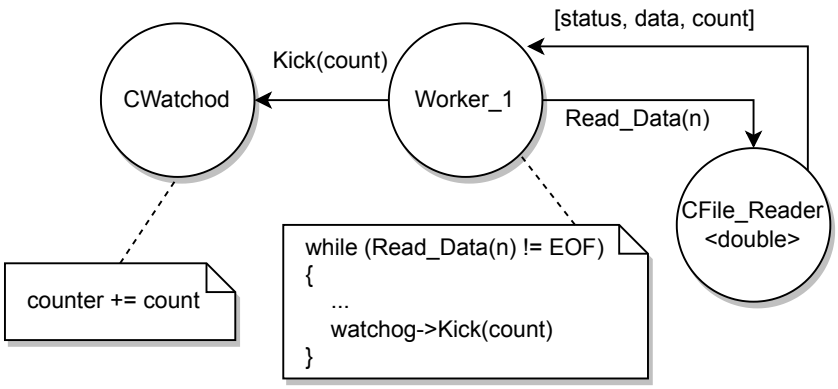

Figure 3.3: Context of `CWatchdog` from a point of a single worker thread

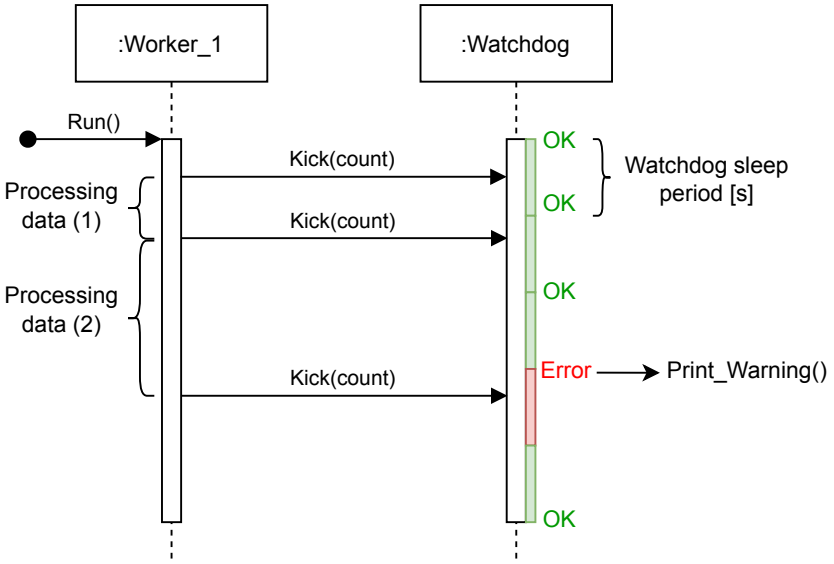

Figure 3.4: Sequence diagram of "Kicking" the watchdog

[2]For example, the OS may not be scheduling the threads of the application fast enough as there might be processes with a higher priority.

3.4 File Processor

As mentioned in section 2.2, the program uses a two-pass approach. In each of the interactions, it creates a number of worker threads to process the input file. Depending on the mode, each thread either processes its designated block of data on a `CPU` or on a `GPU`. When the block processed, it reports the statistics to the farmer for their final aggregation.

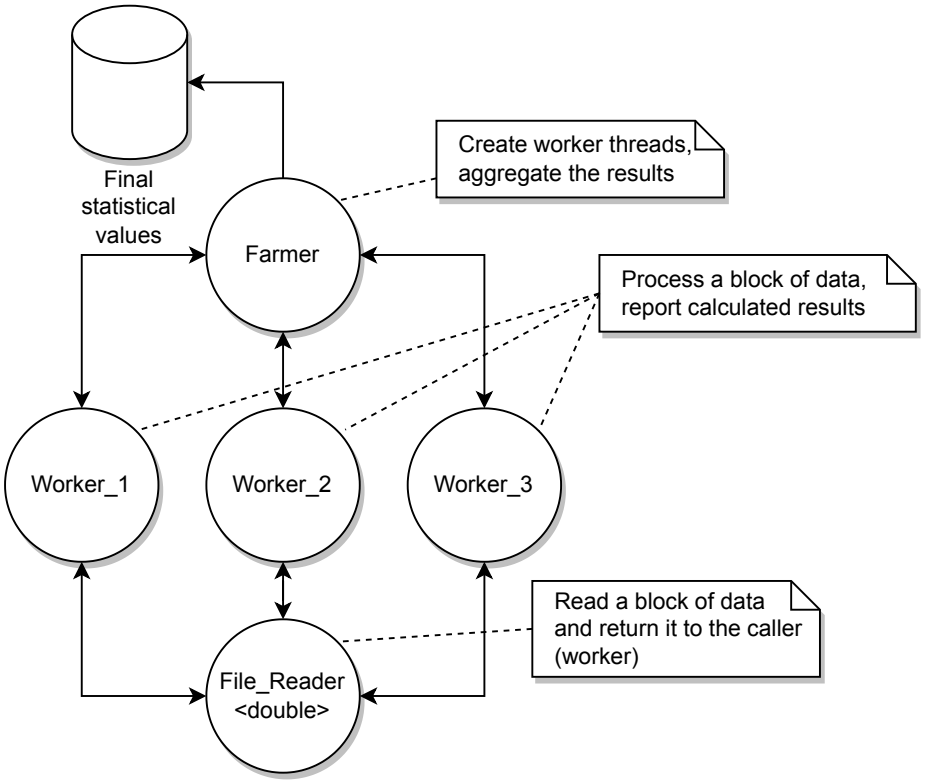

Figure 3.5: Farmer-worker schema for both iterations

The farmer-worker schema follows the same pattern in both iterations. The only difference is what the workers are calculating. The output of the first iteration is the `minimum`, `maximum`, `mean`, `count` (number of valid `doubles`), and `all_ints` (whether or not all values are integers). The output of the second iteration is the `variance`, `sd` (standard deviation), and the histogram.

The final statistical values are held in a class called `CFile_Stats` which fires off both iterations.

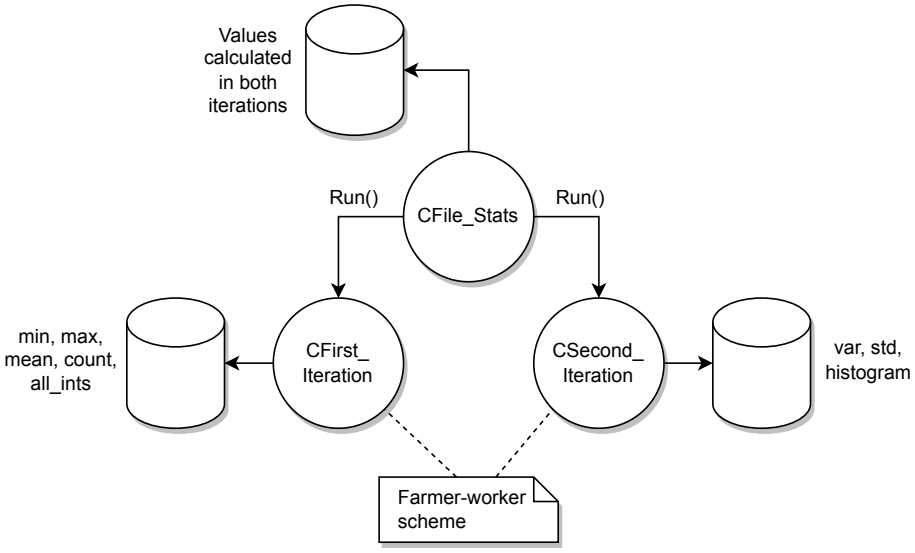

Figure 3.6: Farmer-worker schema for both iterations

3.4.1 First Iteration

Statistical values calculated in the first iteration are held in the following data structure.

```
struct TValues
{
    double min = std::numeric_limits<double>::max();
    double max = std::numeric_limits<double>::lowest();
    double mean = 0.0;
    size_t count = 0;
    bool all_ints = true;
};
```

We can effortlessly calculate all of the statistical values except for the `mean`. If the input make up only `double` values, it is obvious that the `mean` must be a `double` values as well. However, if we apply the following algorithm, the final result may overflow.

```
TValues local_values{};

for (size_t i = 0; i < count; ++i)
{
    const double value = data[i];

    ++local_values.count;
    const double delta = value - local_values.mean;
    local_values.mean += delta / local_values.count;
}
```

In general, if we have two `double` values `x = DBL_MAX` and `y = -DBL_MAX`, we are unable to calculate the difference of these two values as the result will not fill into a `double`. The solution is to scale the input down by dividing all values by 2.

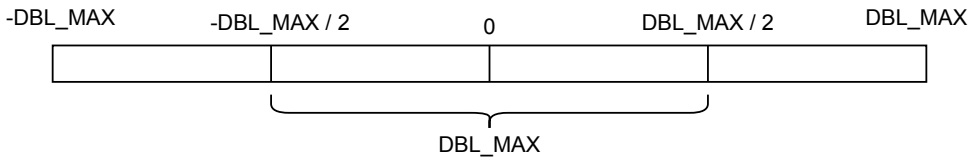

Figure 3.7: Scaling the values down to avoid overflowing

Scaling the numbers down will not change how they are distributed. However, it will affect the result of the Chi-Square goodness of fit test for the Poisson distribution. The solution, in this case, is to multiply the calculated statistical values back up by 2. We do not have to worry about overflowing values as the Poisson distribution does not contain any negative numbers. The general rule here is therefore
`if (min >= 0) { Scale_Values_Up(2); }`.

`OpenCL`

When it comes to `OpenCL`, I decided to implement a numerical reduction algorithm which will be executed by every work item within a work group. The disadvantage of this approach is that we will need a separate local memory for every statistical value we want to calculate (`min`, `max`, ...). It is possible that due to the total local memory size, we will not be able to run the kernel as it would exceed the maximum local memory size of the `OpenCL` device.

This was tacked by repetitively reducing the size of a work group in half until the condition was satisfied. To ensure correct execution of the numerical reduce algorithm, the work group size needs to be a power of 2, hence dividing the work group size by 2.

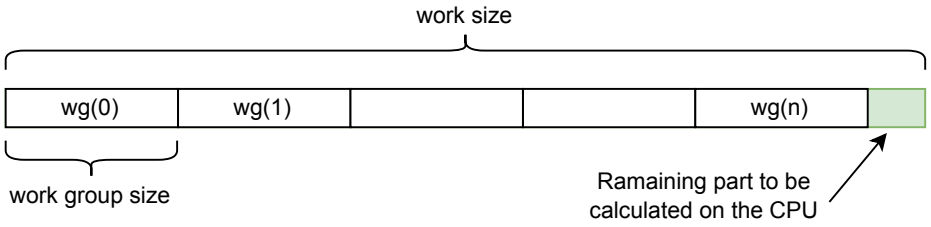

Figure 3.8: Dividing work among multiple work groups (wg)

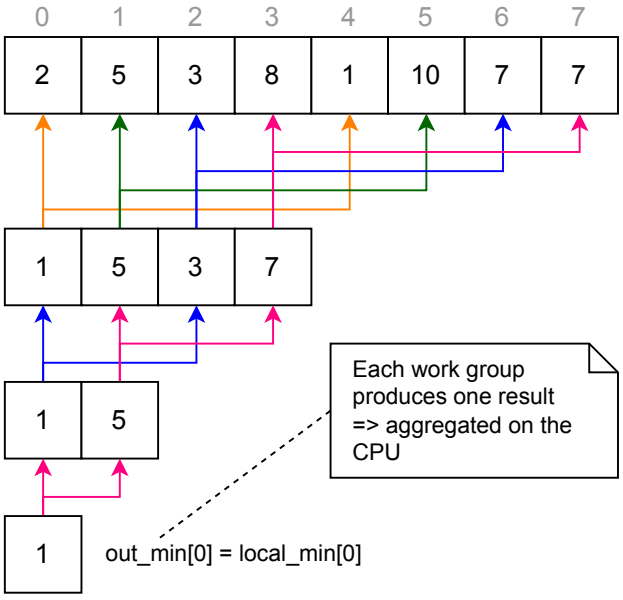

Figure 3.9: Numerical reduce algorithm (calculating the `minimum`)

The source code of the kernel functions is held in the following file `src/processing/gpu_kernels.h`.

3.4.2 Second Iteration

The second iteration works essentially the same as the first iteration. It just uses different formulas to calculate different statistical values. One thing I

would like to highlight though, is the way a histogram is created using an `OpenCL` device.

3.4.3 Creating a histogram on an `OpenCL` device

The issue is that not every `OpenCL` device supports the `atom_inc` operation [10]. This function would be used to atomically increment the value in the bin the current value would fall into (each bin is represented using the `size_t` datatype).

One solution would be to use `uint32_t` instead of `size_t`, and use the `atomic_inc` function, which takes a `uint32_t`. However, we would limit ourselves as to how many values we would be able to store in the histogram. Instead, I decided to represent each bin (`size_t`) as two `uint32_t` numbers.

```
size_t slot_id = (size_t)((value - min) / interval_size);

uint old_value = atomic_inc(&histogram[2 * slot_id]);
uint carry = old_value == 0xFFFFFFFF;
atomic_add(&histogram[2 * slot_id + 1], carry);
```

The final value in each bin is then calculated as `histogram[2 * i] + (histogram[2 * i + 1] * sizeof(uint))`.

3.5 Chi-Square goodness of fit test

The Chi-Square goodness of fit test is implemented as described in subsection 2.2.1. The `CTest_Runner` class executes all tests in parallel. Also, we do not have to run a test for every single distribution. For example, if the `min < 0`, we do not have run the tests for the Poisson distribution or the Exponential distribution.

The results are then sorted primarily by their corresponding P-values and secondly by their Chi-Square error. The final answer is at the 0^{th} position.

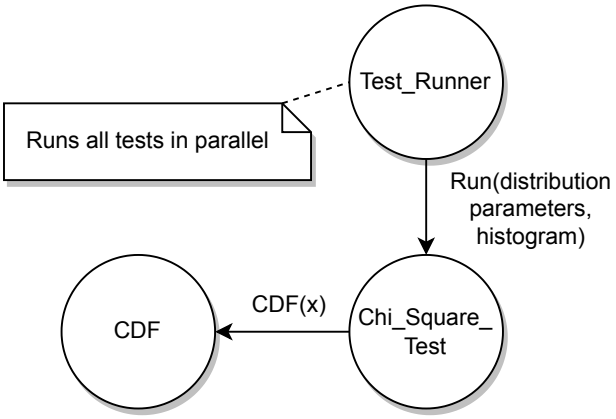

Figure 3.10: Running the Chi-Square goodness of fit test

3.5.1 P-Value

Calculating the P-value involves a lot of math. In this assignment, I decided to implement the `ACM 299` algorithm [11] that calculates the P-value.

3.5.2 CDF Poisson

All CDF functions can be implemented just by writing down the corresponding formula, except for the Poisson distribution.

$$f(k, \lambda) = P(0 \leq X \leq k) = \sum_{x=1}^{k} \frac{\lambda^x}{x!}$$

The issue is that we are unable to use this formula for large values of k. Luckily, there is an article that explains how it can be implemented using the Ramanujan's factorial approximation [12].

3.6 Vectorization

As an attempt to further improve the performance of the program, I decided to use manual vectorization. This technique was applied in the second iteration when the program calculates the `variance` [3]. Using `__m256d`, the variance is calculated as four double values in parallel. After each block of data is processed, these values are summed up into the final result.

[3]It was chosen due to its data dependability.

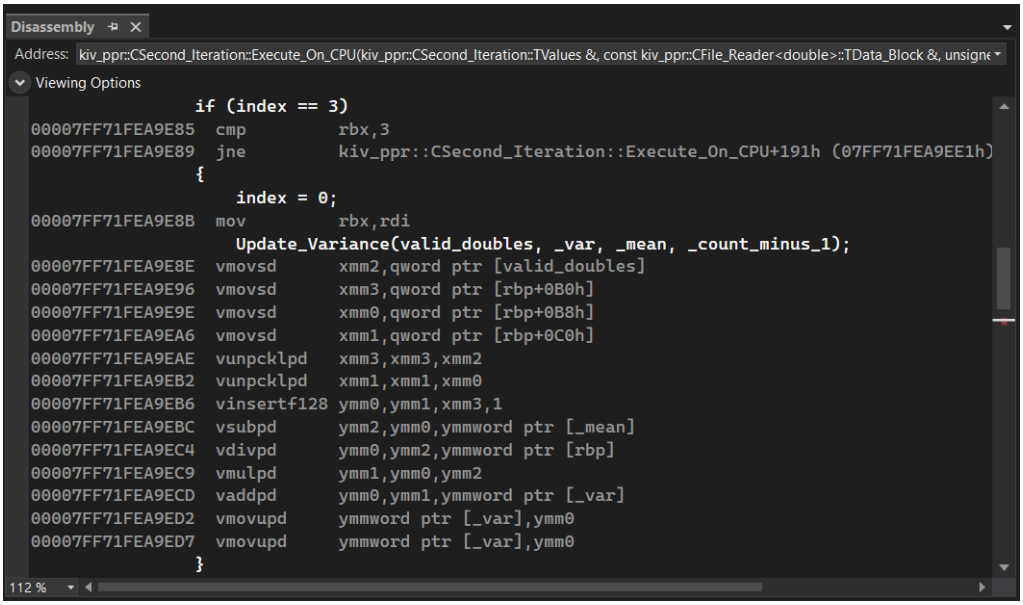

Figure 3.11: SIMD instructions used for variance calculation

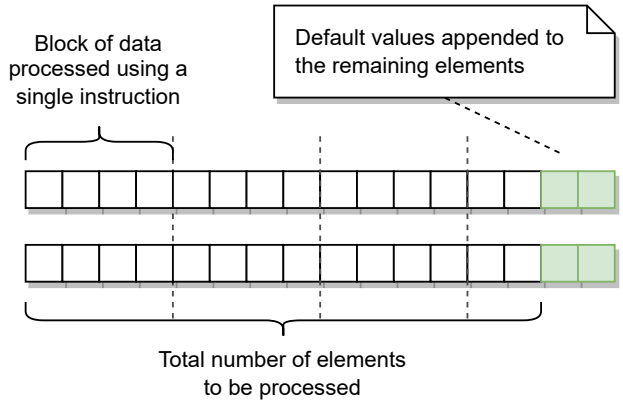

Figure 3.12: Processing a data block using vectorization

If the input chunk of data is not a multiple of four, we need to append default values to the end, so the processor is able to process it using an SIMD instruction. We need to make sure they do not affect the final result. In the case of `variance`, we need to append the value of the `mean` to the end, so the difference is zero, and final result is not affected. If we calculate the `min`, for instance, we need to append `std::numeric_limits<double>::max()`.

```
__m256d _delta = _mm256_sub_pd(_vals, _mean);
const __m256d _tmp_value = _delta;
_delta = _mm256_div_pd(_delta, _count_minus_1);
_delta = _mm256_mul_pd(_delta, _tmp_value);
_var = _mm256_add_pd(_var, _delta);
```

The process of vectorization also takes place in the first iteration when the program calculates the `minimum` and the `maximum`. In particular, it is used both in the `SMP` mode as well as when the program calculates the remaining part of a data block that an `OpenCL` device was not able to calculate.

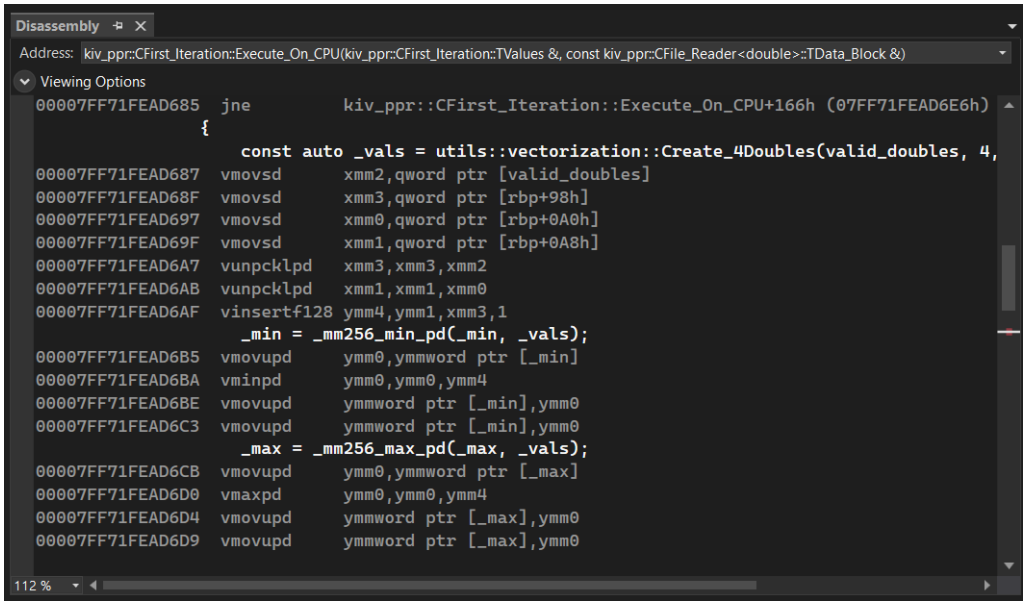

Figure 3.13: `SIMD` instructions used for the `min, max` calculation

4 Experiments

4.1 Performance of different modes

As a first experiment, I wanted to find out how much the execution time differs with each mode. I generated a `16GB` file to test out the performance of the application.

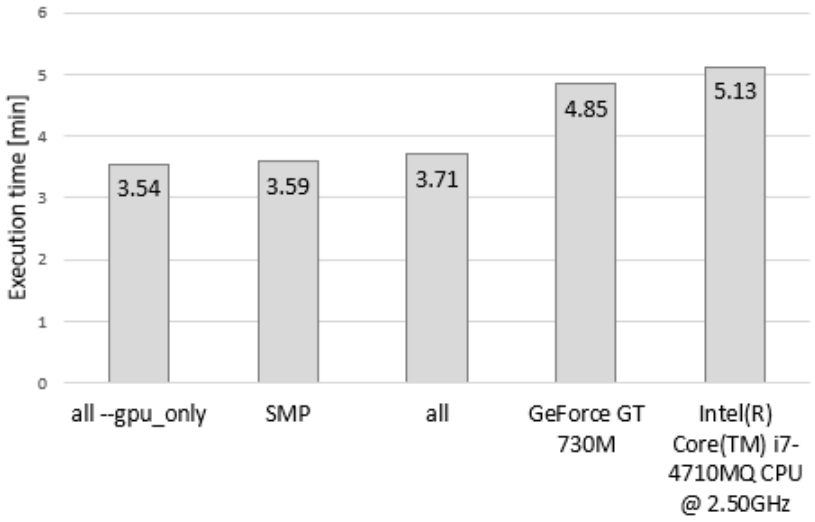

Figure 4.1: Mode performance comparison

[1] The program was run in each of the modes 10 times. The final value was calculated as the average of all the values with the extremes taken away.

As an outcome of this experiment, I decided to implement an option `--gpu_only` switch 5.3 that prevents the use of a `CPU OpenCL` device. As it turns out, using a `CPU` as an `OpenCL` device negatively impacts the performance of the application [2]. The measured values can be found in the attachments 7.2.

[1]Parameters of the machine used for testing: `LenovoT540p`; `Intel(R) Core(TM) i7-4710MQ CPU @ 2.50GHz 2.49 GHz`, installed `RAM 8.00 GB`, `GPU` used `GeForce GT 730M`

[2]The numerical reduction algorithm used in this application performs better on a `GPU`

4.2 Performance after vectorization

In order to find out how much vectorization improves the overall performance of the application, I used the `Ubuntu ISO` image which can be downloaded from their website [13]. The program was run five times for each data block size [3].

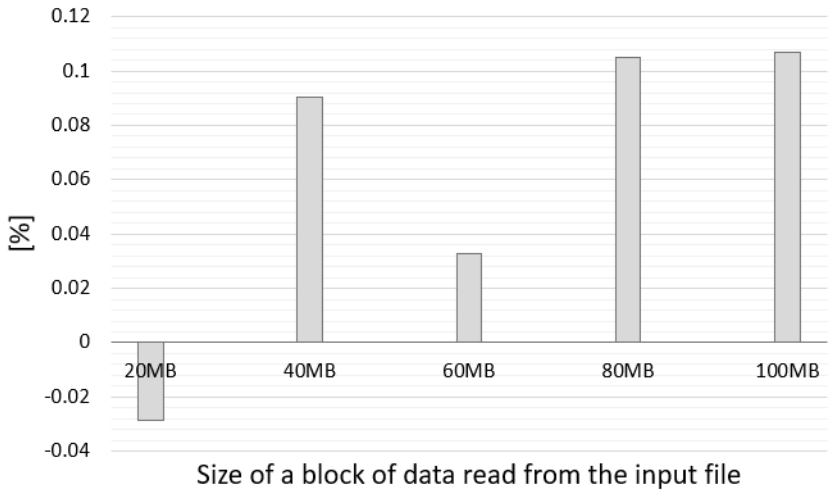

Figure 4.2: Performance improvement of the program run in the `SMP` mode

The highest improvement out of all can be seen in the chart shown above. It is caused by the absence of `OpenCL` devices. All work is done only by the `CPU`, so the portion of vectorized code is executed more often than if there were other devices the work would be split among. However, it is important to mention that the overall application of the program was found almost identical. Results of the other two modes can be seen further down below. [4].

[3]Each value was calculated as a ratio of the averaged measured time of the program execution.

[4]The tests were carried out on a `Lenovo ThinkPad T490s` with the following parameters: `Intel Core i7-8565U`, `16 GB RAM`, `Intel(R) UHD Graphics 620`

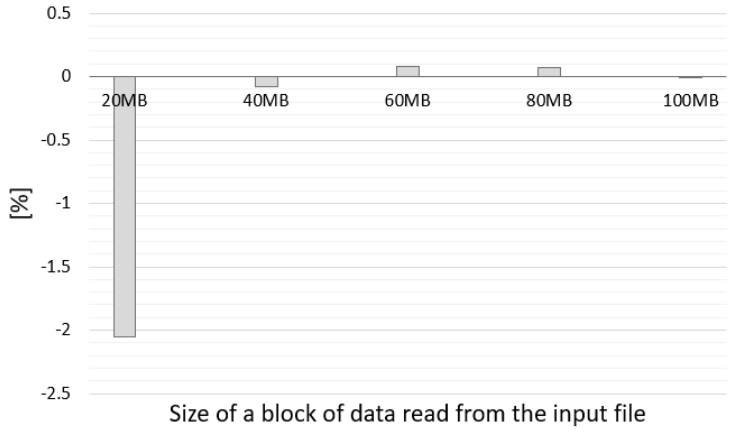

Figure 4.3: Performance improvement of the program run in the `ALL` mode

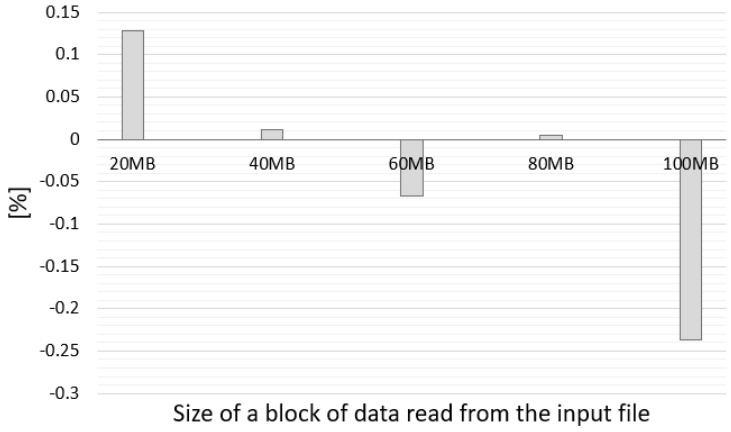

Figure 4.4: Performance improvement of the program run in the `OpenCL dev` mode

The raw data of each of these measurements can be found in the attachments 7.1.

4.3 Bottleneck

The bottleneck of the application was found out to be reading data from the disk. When the program reads an exceptionally big chunk of data at once, it can be observed that the `GPU` does not have any performance issues when processing the data. Instead, it periodically waits for the `CPU` to finish reading data from the disk.

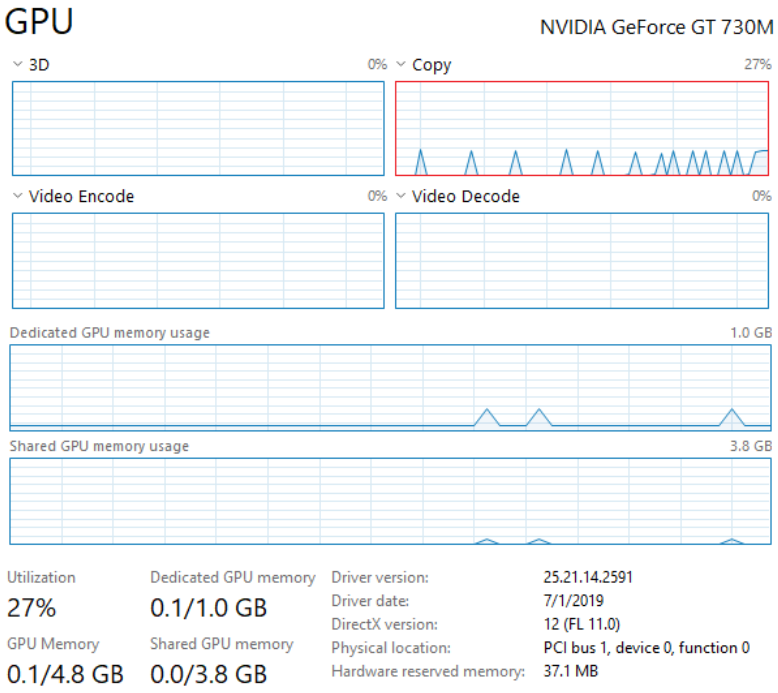

Figure 4.5: Usage of `GeForce GT 730M`

The program was executed with and additional option `-b 209715200` 5.3 to make the application read `200MB` from the input file at a time.

5 User manual

The program should run on both Linux and Windows. However, it was primarily tested on the Windows platform.

5.1 Dependencies

The application uses the `OpenCL.lib` library along with the `cl.hpp` file. To parse the input arguments, it takes advantage of the `cxxopts` header-only library [9].

5.2 Compulsory parameters

There are two compulsory parameters the user must enter when running the program. The first parameter is a path to the input file, and the second one is the mode in which the program will execute (`SMP`, `ALL`, or a list of `OpenCL` devices to be used). After the compulsory parameters, the user can enter optional parameters.

5.3 Help

To see what parameters can be passed into the program, all the user has to do is to run the `pprsolver.exe --help` command. All parameters can be used either in their short version (`-p 0.01`) or the more descriptive version (`--p_critical=0.01`). Handling input parameters is done with the use of `cxxopts`, which is an open-source header-only library [9].

```
C:\dev> pprsolver.exe --help
KIV/PPR Semester project - Classification of statistical
   distributions (Chi-Square Goodness of Fit Test)
Usage:
  pprsolver.exe <filename> <all | SMP | "dev1" "dev2" "dev3"
   ...> [OPTION...]

  -p, --p_critical arg       Critical p value used in the Chi-
   square test
                             (default: 0.050000)
```

```
  -b, --block_size arg       Number of bytes read from the
   input file at a
                             time (block size) (default:
   10485760)
  -w, --watchdog_period arg  How often the watchdog checks if
   the program
                             is working correctly [s] (default:
    10)
  -t, --thread_count arg     Number of threads created by the
   application
                             (default: 8)
  -g, --gpu_only             Do not use an OpenCL device which
   is not a GPU
  -h, --help                 Print out this help menu
```

The `-g` switch is there to leave out the use of an `OpenCL` devices which are not `GPUs`. As far as performance is concerned, a `CPU OpenCL` device may not be optimal to execute the numerical reduction algorithm design within this project.

5.4 Invalid path

If the user enters an invalid path or the program fails to open up the input file, they will be presented with the following error message.

```
c:\dev> pprsolver.exe gauss all
The program is running in 'all' mode
Block size per read = 1310720 [B]
Watchdog checkup period = 10s
Number of threads = 8

Available OpenCL devices supporting double precision:
GeForce GT 730M
Intel(R) Core(TM) i7-4710MQ CPU @ 2.50GHz

Failed to open the input file (gauss)
```

5.5 Execution examples

5.5.1 Run (1)

```
c:\dev> pprsolver.exe gauss smp
The program is running in 'SMP' mode
Block size per read = 1310720 [B]
Watchdog checkup period = 10s
Number of threads = 8

Processing file gauss [16000 B]

Calculated statistics (parameters):
min   = -4.3361
max   = 3.6349
mean  = 0.026487
count = 2000
var   = 0.5156
sd    = 1.0155

Distribution   Chi Square     DF        P-value     Accepted
------------   ----------     --        -------     --------
Normal         19.612         26        0.80958     YES
Uniform        2468.5         38        0           NO
-------------------------------------------------------------
Level of significance = 5%

The date correlates the most to the Normal distribution.

Time of execution: 0 sec
```

In this example, the program was run the in `SMP` mode. In the first section of the output, the program prints out settings to be used for the program execution (mode, number of threads, etc.). After that, the program prints out the name of the input file along with statistical values calculated within the first as well as the second iteration. Lastly, the program prints out the results of the Chi-Square goodness of fit test along with the final answer based on these results [1].

[1] `DF` stands for degrees of freedom.

5.5.2 Run (2)

```
c:\dev>pprsolver.exe ubuntu-22.04.1-desktop-amd64.iso all --
   gpu_only
The program is running in 'all' mode
Block size per read = 1310720 [B]
Watchdog checkup period = 10s
Number of threads = 8

Available OpenCL devices supporting double precision:
GeForce GT 730M

Processing file ubuntu-22.04.1-desktop-amd64.iso [3826831360 B]

Calculated statistics (parameters):
min   = -1.7977e+308
max   = 1.7977e+308
mean  = -2.9636e+303
count = 477109960
var   = inf
sd    = inf

Distribution   Chi Square     DF        P-value     Accepted
------------   ----------     --        -------     --------
Uniform        2.7837e+12     5919      0           NO
Normal         2.3855e+08     -2        0           NO
-------------------------------------------------------------
Level of significance = 5%

Statistically, none of the tests has been accepted.

However, based on the Chi-Square error (2.7837e+12) and the
   degrees of freedom (5919), the data seems to correlate the
   most to the Uniform distribution though it is STRONGLY
   recommended to double verify the answer.

Time of execution: 11 sec
```

In the example, none of the Chi-Square goodness of fit tests was accepted. However, the program still gives the user an answer based on the Chi-Square

error. It also recommends the user to double verify the answer as it may not be correct [2].

5.5.3 Run (3)

```
c:\dev>pprsolver.exe test_data.dat "GeForce GT 730M"
The program is running in 'OpenCL devs' mode
Block size per read = 1310720 [B]
Watchdog checkup period = 10s
Number of threads = 8

Checking availability of the listed devices:
Name: GeForce GT 730M (OK)

Processing file test_data.dat [80 B]

Calculated statistics (parameters):
min   = 3.6187
max   = 6.7795
mean  = 4.8955
count = 10
var   = 1.3727
sd    = 1.1716

Distribution   Chi Square     DF        P-value     Accepted
------------   ----------     --        -------     --------
Normal         0.02245        -2        0           NO
Uniform        0.1            -2        0           NO
Exponential    0.30159        -1        0           NO
-------------------------------------------------------------
Level of significance = 5%

Statistically, none of the tests has been accepted.
Judging by all degrees of freedom being less than 0, you may
   need to input more data into the program.

Time of execution: 0 sec
```

[2]The `var` and `sd` values are `inf` because they are too big to fit into a `double`.

In this example, the program came to the conclusion that the user did not provide enough data and therefore, it cannot answer the question what distribution the data comes from. It is based on the the fact that the histogram is made of only a very few bins ($\texttt{DF} < 0$).

6 Conclusion

The program correctly classifies all of the test files that were provided with the assignment. The program was tested on an Intel `CPU`, Intel `GPU`, Nvidia `GPU`, and AMD `GPU`. It was concluded that using `OpenCL` devices does not out perform the `SMP` mode as much due to the bottleneck found in reading data from the disk. Also, as you can see in figure 4.4, using a `CPU` as an `OpenCL` devices slows down the overall performance of the application.

Bibliography

[1] John D. Cook, PhD, President, Computing skewness and kurtosis in one pass, https://www.johndcook.com/blog/skewness_kurtosis/

[2] Skewness, https://en.wikipedia.org/wiki/Skewness

[3] Excess kurtosis, https://en.wikipedia.org/wiki/Kurtosis#Excess_kurtosis

[4] Goodness of fit test, https://en.wikipedia.org/wiki/Goodness_of_fit

[5] Null hypothesis https://en.wikipedia.org/wiki/Null_hypothesis

[6] Keith A. Baggerly, Probability binning and testing agreement between multivariate immunofluorescence histograms: Extending the chi-squared test, https://onlinelibrary.wiley.com/doi/full/10.1002/1097-0320(20011001)45:2%3C141::AID-CYTO1156%3E3.0.CO;2-M#bib11

[7] Cumulative distribution function, https://en.wikipedia.org/wiki/Cumulative_distribution_function

[8] P-value, https://en.wikipedia.org/wiki/P-value

[9] cxxopts library, https://github.com/jarro2783/cxxopts

[10] atom_inc function, https://registry.khronos.org/OpenCL/sdk/1.2/docs/man/xhtml/atom_inc.html

[11] Algorithm ACM 299 https://dl.acm.org/doi/pdf/10.1145/363242.363274

[12] oisson CDF for large lambdas https://www.codeproject.com/Tips/1216237/Csharp-Poisson-Cumulative-Distribution-for-large-L

[13] Ubuntu ISO image https://ubuntu.com/download/desktop

7 Attachments

7.1 Performance after vectorization

ALL Mode										
	20MB		40MB		60MB		80MB		100MB	
	N	V	N	V	N	V	N	V	N	V
1	10080	10083	10073	10070	10132	10119	10113	10129	10100	10096
2	10098	10106	10072	10098	10127	10105	10115	10113	10118	10106
3	10071	11107	10091	10086	10100	10120	10123	10115	10117	10103
4	10085	10095	10069	10083	10110	10101	10130	10114	10100	10120
5	10094	10092	10075	10085	10132	10117	10113	10086	10097	10113

Table 7.1: Time of the program execution [ms] in the `ALL` mode with different block sizes read from the input file

[1]

`OpenCL dev` Mode										
	20MB		40MB		60MB		80MB		100MB	
	N	V	N	V	N	V	N	V	N	V
1	15350	15331	15471	15470	15443	15478	15437	15435	15391	15451
2	15427	15390	15459	15439	15487	15493	15432	15445	15418	15456
3	15459	15427	15472	15497	15449	15431	15456	15465	15437	15491
4	15371	15363	15479	15470	15468	15454	15481	15467	15466	15477
5	15371	15365	15448	15444	15474	15517	15439	15429	15458	15478

Table 7.2: Time of the program execution [ms] in the `OpenCL Dev` mode with different block sizes read from the input file

[1] N means not vectorized and V means vectorized.

	SMP Mode									
	20MB		40MB		60MB		80MB		100MB	
	N	V	N	V	N	V	N	V	N	V
1	9066	9064	9072	9061	9112	9105	10113	10086	10130	10120
2	9068	9096	9090	9077	9109	9107	10109	10101	10108	10105
3	9093	9076	9087	9090	9104	9103	10091	10103	10139	10199
4	9061	9056	9077	9072	9114	9115	10114	10107	10126	10109
5	9059	9068	9067	9052	9107	9101	10099	10076	10111	10107

Table 7.3: Time of the program execution [ms] in the SMP mode with different block sizes read from the input file

7.2 Performance of different modes

all –gpu_only	all	SMP	GeForce GT 730M	Intel(R) Core(TM) i7-4710MQ CPU @ 2.50GHz
4.82	3.52	2.85	5.63	3.48
4.18	3.23	5.28	4.52	6.05
3.37	2.8	3.1	4.63	4.73
3.13	3.43	4.47	4.95	4.8
3.2	3.1	3.27	8.75	5.25
3.12	5.03	3.48	4.85	5.53
2.98	4.02	2.85	4.17	5
4.5	3.68	3.63	4.6	5.1
3.8	5.42	3.72	4.38	7.15
2.58	3.63	4.18	5.22	4.57

Table 7.4: Times of execution [min] of different modes (tested on a 16GB file).