FAKULTA APLIKOVANÝCH VĚD
ZÁPADOČESKÉ UNIVERZITY
V PLZNI

KIV/PPR
Estimating the distribution of data using parallelized algorithm

Václav Honzík
A21N0045P
5. 1. 2023

Assignment

The program reads a specified file where each 64-bit sequence is interpreted as a 64-bit double. The application processes the data and estimates which of the following distribution are the most similar - Gaussian, Poisson, exponential, or uniform. The application should be run as follows:

```
pprsolver.exe <FILE_PATH> <PROCESSOR>
```

Where:

- *FILE_PATH* specifies either absolute or relative path to the processed file
- *PROCESSOR* - defines which processing units are used, these can be:
    - ALL - uses CPU and all available GPUs
    - SMP - multithreaded processing on CPU
    - Listing of desired OpenCL devices

The tested file can be several GB in size, however, the process is limited to 1 GB of RAM. The program execution should not exceed 15 minutes on Intel Core i7 Skylake CPU.

Analysis

All four mentioned distributions (i.e. Gauss, Poisson, exponential, and uniform) are an example of a probability distribution. Probability distribution could be described as a mathematical function that assigns a likelihood to given values of a measured random variable. These values can be both discrete or continuous, depending on the distribution. Knowing the distribution of processed data is crucial, as it can e.g. help detect outliers, or use specialized methods, depending on the task.

While generating values from a given distribution is simple (many easy-to-implement methods exist, e.g. rejection sampling), the reverse operation is usually not. The first issue is that we need large amounts of data points to ensure our estimation is not skewed (for example by some low-probability samples). Thankfully, in this work, the processed files will be gigabytes in size, which should contain enough data to make the estimation meaningful.

A much bigger challenge is the fact, that the computations are performed on machines that do not operate with infinite floating-point precision and instead use 32/64-bit formats. Most algorithms to estimate the distribution use common statistical values such as mean, standard deviation, skewness, etc., which are computed by a combination of the data points (i.e. addition, multiplication, division).

Since we operate with finite precision, computations of these values can easily produce an incorrect result (e.g. an overflown or underflown value) which invalidates the entire solution. Therefore, we need to use numerically stable algorithms, that are designed around finite precision, such as running mean, running standard deviation, etc., rather than conventional mathematical formulas.

Next, we require the entire estimation process to be done in parallel. This inevitably means that the dataset will have to be split into several chunks, where each chunk is used by a single processing unit to output a sub-result. The term processing unit can be interpreted differently, here it simply means either a CPU thread or an OpenCL work item. Finally, if the algorithm works well, merging outputs from each processing unit should return a result that is very close to one obtained by serialized computation.

Chi-square goodness of fit test

One way to estimate the given distribution is to perform the so-called Chi-square goodness of fit. This is a statistical test to determine whether a given population (i.e. our dataset) follows a certain distribution (this is the null hypothesis) or not (the alternative hypothesis).

While this test is used only for categorical data - i.e. outputs of a discrete random variable, it can be easily adapted for continuous values as well. The general idea is to discretize the data into K intervals (bins) where each continuous value is put into one of the bins. This effectively produces a histogram that is then used to compute the χ^2 value. The value is computed as follows:

$$\chi^2 = \sum \frac{(O_i - E_i)^2}{E_i}$$

Figure 1 - Formula to compute Chi Square value

Where O_i is the i-th observed value (i.e. i-th bin) and E_i is the expected value from the given distribution. The computed Chi Square value is then used to measure the fit. Depending on the degrees of freedom of the distribution and significance level alpha we compute the critical value *p*. If the Chi Square value is less than the critical value we do not reject the null hypothesis - i. e. the data follow the given distribution. Visualization of the test for three degrees of freedom can be seen in Fig 2.

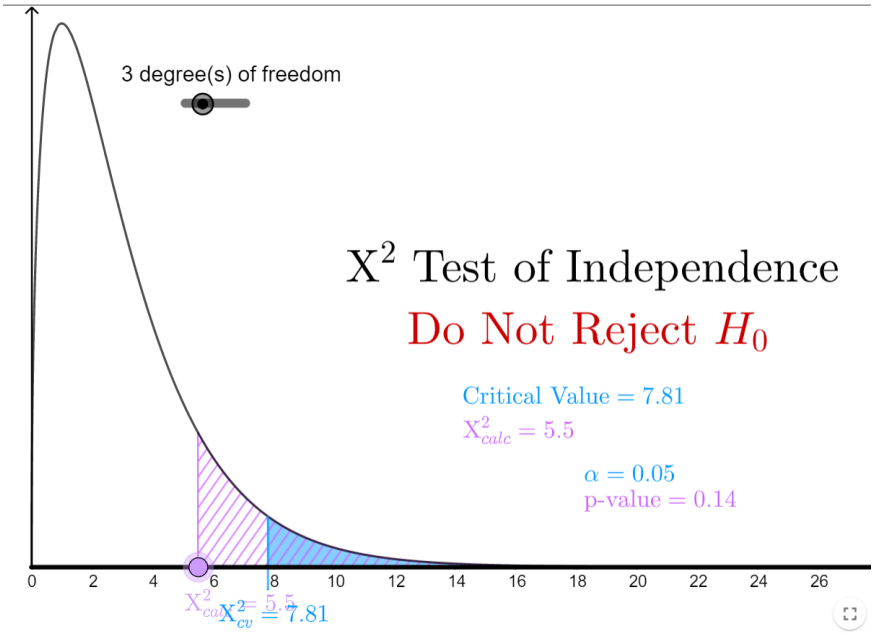

Figure 2 - Visualization of Chi Square Goodness of Fit test.
Source: https://www.geogebra.org/m/smhy8cxz

While this test is often a good approach it has many downsides. The main one is that the processing needs to be done in two passes. In the first pass the algorithm computes statistics to obtain the expected value for the classified distribution - i.e.

mean, standard deviation, minimum, maximum, etc., while in the second pass the actual test is performed. Additionally, this is difficult to parallelize as we need some way to merge the sub results from each processing unit which will require some form of synchronization, slowing down the entire process.

Skewness and Kurtosis

An alternative to Chi Square test is to compute higher central moments that can be used to compute skewness and kurtosis. Skewness describes the asymmetry of the distribution while kurtosis describes how heavy-tailed the distribution is (in relation to Gaussian distribution). The visualization of skewness can be seen in Fig. 3 while Fig. 4 shows kurtosis

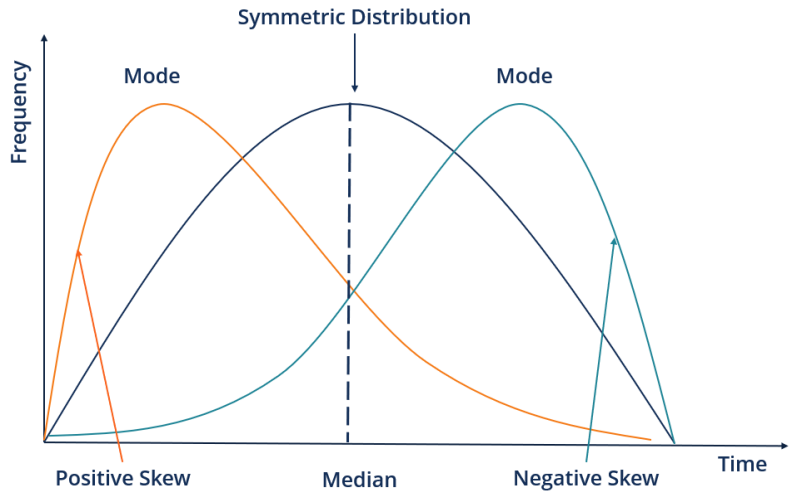

Figure 3 - Visualization of skewness
Source: https://corporatefinanceinstitute.com/resources/data-science/skewness/

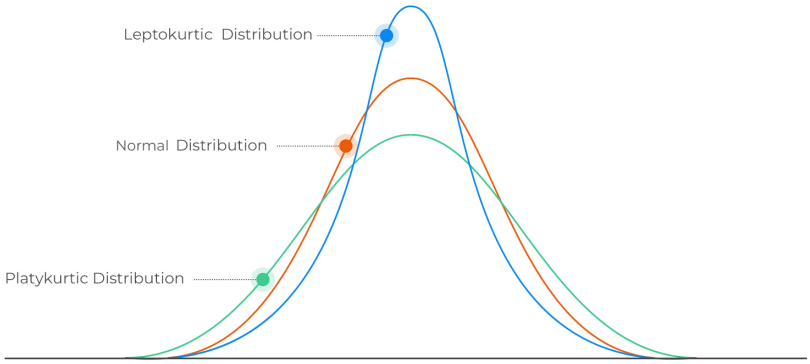

Figure 4 - Visualization of Kurtosis - normal distribution has kurtosis equal to zero, while leptokurtic has kurtosis more than zero and platykurtic has kurtosis less than zero

Source: https://analystprep.com/cfa-level-1-exam/quantitative-methods/kurtosis/

The main idea behind this approach is that both kurtosis and skewness differ between each distribution and are (mostly) independent of parameters - i.e. Gaussian distribution will have the same skewness regardless of its mean or standard deviation.

Essentially, we can take each distribution and project it to 2D Euclidean space where one dimension is skewness while the other is kurtosis (see Fig. 5). The same works for the processed data as it must follow also follow some distribution (though it may not be any of the four we are trying to classify). To determine the similarity between the data and particular distribution we can use a simple distance metric such as Euclidean distance.

There is, however, one exception which is the Poisson distribution that has skewness both defined by its parameter Lambda and for large Lambda has the same skewness and kurtosis as Gaussian distribution. Fortunately, this issue can be easily circumvented by simply keeping a flag that indicates whether all processed data samples were integers. If such a flag is set to true we can then classify the distribution as Poisson.

The main advantage of this method, over the previously mentioned Chi Square test, is that this it can be done in one pass and can be easily parallelized as there exist algorithms to compute both skewness and kurtosis in one pass.

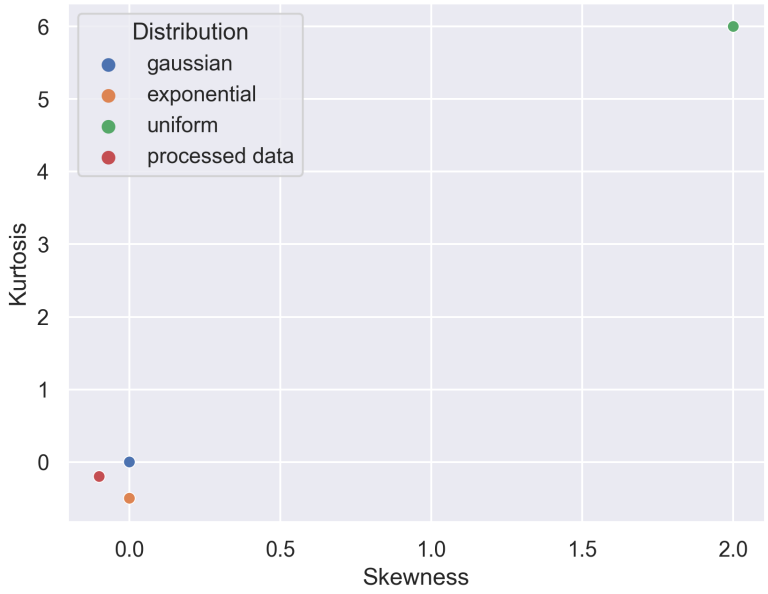

Figure 5 - Visualization of Gaussian, exponential and uniform distributions in 2D space. "Processed data" is a point computed from the processed file that we then assign to the closest point

Implementation

The implementation itself uses the second mentioned approach in the previous section - i. e. classification via kurtosis and skewness. The program itself is written in C++ 17 and compiles using MSVC compiler. Additionally, the application utilizes OpenCL and OneTBB frameworks for parallelization. The entire application comprises several classes/components, where each one serves one specific purpose. A high-level overview of the application is depicted in the dataflow diagram in Fig. 6.

Job Scheduler

The first core component is called *JobScheduler*. As the name suggests job scheduler is used to distribute work among workers in a Farmer-Worker fashion. Each job comprises a start and end index in the file. The class is constructed using *ProcessingConfig* which contains all the necessary information to start the computation (e.g. whether to compute on SMP, vector of OpenCL devices, the maximum amount of memory, etc.).

This component is also responsible for creating (in its constructor) and terminating threads for the watchdog and workers (in its destructor). Its most important method is *run()* which starts the computation and returns computed statistics or throws *std::runtime_error* should the computation fail. The class first assigns a job to OpenCL devices (if they are available) and then gives it to SMP. If there is no worker available it waits (via counting semaphore) for any worker to finish.

Device Coordinator

Device coordinator can be thought of as a worker that wraps functionality over a specific device. In this context, there can be three types of devices - CPU (or rather SMP CPU, that calls TBB library), OpenCL device, and AVX2 vectorized CPU (again SMP CPU but additionally uses AVX2 instructions). The class itself is abstract and has an *onProcessJob* method that needs to be implemented by a specific device type. Therefore, there exist additional three classes each for the given device type - *CpuDeviceCoordinator, Avx2CpuDeviceCoordinator,* and *ClDeviceCoordinator* which are discussed later.

Each device coordinator has a separate thread that runs a loop where it waits until it is assigned a job or terminated. To synchronize with the JobScheduler, the coordinator uses a counting semaphore, that is initialized to 0 to block the coordinator thread until it is given a job. To avoid circular dependency, each coordinator is given three callback functions that implicitly communicate with JobScheduler - *jobFinishedCallback*, *notifyWatchdogCallback,* and *errCallback*.

Device coordinators load data directly - via the *DataLoader* object which opens a file and reads data to the host or device buffer (depending on the coordinator type). This implies that multiple threads read from the file at the same time.

Watchdog

Another important component is the watchdog. This is an attempt to check whether the program is running correctly (though we would need an actual HW for correct implementation). Internally, the watchdog has a counter that symbolizes the amount of work processed since the last update (in bytes).

The class periodically checks this variable and logs results to the standard output. If the counter value is greater than zero it interprets it as the number of bytes processed, whereas zero is interpreted as “no data processed”. Potentially, zero could also signal a deadlock. Each device coordinator needs to call the watchdog either when they finish the job or process part of it (via *notifyWatchdogCallback*). To avoid logging before any job is scheduled, the watchdog is firstly blocked on a counting semaphore and later notified by the scheduler after it has started handing jobs.

StatsAccumulator

Stats accumulator contains logic for computing (and accumulating) running statistics in the program. Each instance comprises the number of processed items, the first four central moments, the minimum value, and a flag *isIntegerDistribution*. The data is added via the *push()* method that takes one double, checks if it is classified as *FP_NORMAL* or *FP_ZERO,* and updates the accumulator’s state. The implementation of the push method is adapted from John Cook’s implementation:
https://www.johndcook.com/blog/skewness_kurtosis/

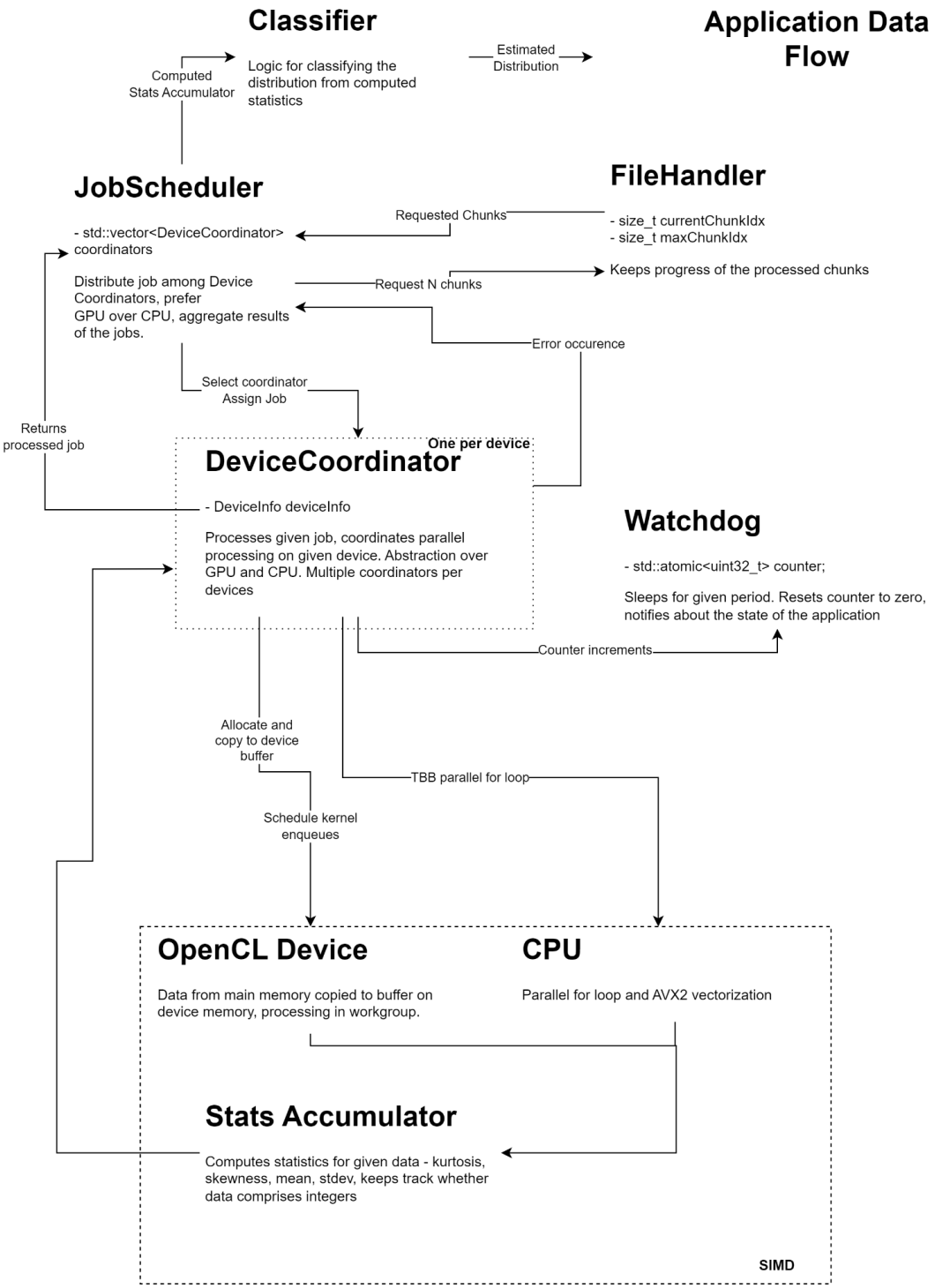

Figure 6 - Dataflow diagram of the application

Avx2StatsAccumulator

There is also an implementation of the accumulator logic using Intel's AVX2 instructions called *Avx2StatsAccumulator*. The vectorization is done manually, using functions from the *immintrin.h* header, as it was much easier than attempting to vectorize it automatically via MSVC.

Similarly to the scalar stats accumulator, the vector implementation filters out invalid values (see Fig. 7). This could also be done beforehand but would most likely result in severe performance degradation. Since the vectorized code cannot use if statements we are forced to mask out the values. This, however, does not change the code by much as most values are updated by simply summing their value with a newly computed one. Therefore, we can mask out the computed value by multiplying it by zero or one. Figure 8 depicts the corresponding C++ code (the method *pushWithFiltering*) and contains a vectorized update of the first and fourth moment in the accumulator. Figure 9 shows part of the generated assembly from the compiler.

```
inline auto valuesValid(const __m256d& x) -> __m256i {
    const auto bits: __m256i = _mm256_castpd_si256(x); // convert x to integer vector
    auto exponent: __m256i = _mm256_and_si256(bits, EXPONENT_MASK); // extract exponent
    exponent = _mm256_srli_epi64(exponent, 52); // shift by 52 bits to get mantissa

    // if exponent == 0
    const auto expEqualsZero: __m256i = _mm256_cmpeq_epi64(exponent, _mm256_setzero_si256());

    // AND bits & MANTISSA_MASK > 0
    auto bitsAndMantissa: __m256i = _mm256_and_si256(bits, MANTISSA_MASK);
    bitsAndMantissa = _mm256_cmpgt_epi64(bitsAndMantissa, _mm256_setzero_si256());

    // If exponent == 0 && bits & MANTISSA_MASK set to true
    const auto invalid1: __m256i = _mm256_and_si256(expEqualsZero, bitsAndMantissa);

    // Similarly if exponent == 0x7ff it is invalid as well
    const auto invalid2: __m256i = _mm256_cmpeq_epi64(exponent, _mm256_set1_epi64x(0x7ff));

    // Combine both with OR operation
    // This will return for each element in the vector if they are invalid
    const auto invalid: __m256i = _mm256_or_si256(invalid1, invalid2);

    // Negate - i.e. XOR with 0xFFFFFFFFFFFFFFFF == int64_t(-1) == UINT64_MAX
    return _mm256_xor_si256(invalid, _mm256_set1_epi64x(UINT64_MAX));
}
```

Figure 7 - Implementation of custom std::fpclassify to return whether double is valid

```cpp
void Avx2StatsAccumulator::pushWithFiltering(const __m256d x) {
    // AVX2 uses 256-bit registers which can hold 4 doubles
    // so we need to process 4 doubles at a time

    // Check whether values are valid
    const auto validMask:__m256i = VectorizationUtils::valuesValid(x);

    // If the values are valid check if they are integers
    // To update isIntegerDistribution we have two boolean variables: valid and isInteger
    // We must satisfy that if valid is false we always return true and if valid is true we return isInteger
    // Boolean formula for this is: !valid || isInteger
    const auto isInteger:__m256i = VectorizationUtils::valuesInteger(x);
    const auto validNegation:__m256i = _mm256_xor_si256(validMask, _mm256_set1_epi64x(UINT64_MAX));
    const auto isIntegerDistributionUpdate:__m256i = _mm256_or_si256(validNegation, isInteger);

    // Update isIntegerDistribution = isIntegerDistribution && !valid || isInteger
    isIntegerDistribution = _mm256_and_si256(isIntegerDistribution, isIntegerDistributionUpdate);

    // Update the minimum
    // If the value is valid we keep the value of xi, if the value is invalid we set the value of xi to INFINITY
    const auto minMask:__m256d = VectorizationUtils::maskDouble4(
        value:_mm256_set1_pd(std::numeric_limits<double>::infinity()), validNegation);
    minVal = _mm256_min_pd(minVal, _mm256_add_pd(x, minMask));

    const auto n1:__m256d = convertInt4ToDouble4(n); // n1 = n, we convert to double to avoid casting later
    n = _mm256_add_epi64(n, _mm256_and_si256(_mm256_set1_epi64x(1), validMask)); // n += valid * 1

    const auto nDouble:__m256d = convertInt4ToDouble4(n); // nDouble = n, we convert to double to avoid casting later
    const auto delta:__m256d = _mm256_sub_pd(x, m1); // delta = x - m1
    const auto deltaN:__m256d = _mm256_div_pd(delta, nDouble); // deltaN = delta / n
    const auto deltaNSquared:__m256d = _mm256_mul_pd(deltaN, deltaN); // deltaNSquared = deltaN * deltaN
    const auto term1:__m256d = _mm256_mul_pd(delta, _mm256_mul_pd(deltaN, n1)); // term1 = delta * deltaN * n1

    // m1 = m1 + validMask * deltaN
    m1 = _mm256_add_pd(m1, VectorizationUtils::maskDouble4(deltaN, validMask));

    // m4 += (term1 * deltaNSquared) * (n * n - 3 * n + 3) + 6 * deltaNSquared * m2 - 4 * deltaN * m3
    // m4a1 = term1 * deltaNSquared
    // m4a2 = (((n * n) - (3 * n)) + 3)
    // m4a = m4a1 * m4a2
    const auto m4a1:__m256d = _mm256_mul_pd(term1, deltaNSquared);
    const auto m4a2:__m256d = _mm256_add_pd(
        _mm256_set1_pd(3), _mm256_sub_pd(_mm256_mul_pd(nDouble, nDouble), _mm256_mul_pd(_mm256_set1_pd(3), nDouble)));
    const auto m4a:__m256d = _mm256_mul_pd(m4a1, m4a2);
```

Figure 8 - Update of the moment M1 and M4 in the accumulator

```
    const auto deltaN = _mm256_div_pd(delta, nDouble); // deltaN = delta / n
00007FF68936F354  vmovupd     ymm0,ymmword ptr [delta]
00007FF68936F35C  vdivpd      ymm0,ymm0,ymmword ptr [nDouble]
00007FF68936F364  vmovupd     ymmword ptr [rbp+0DA0h],ymm0
00007FF68936F36C  vmovupd     ymm0,ymmword ptr [rbp+0DA0h]
00007FF68936F374  vmovupd     ymmword ptr [deltaN],ymm0
    const auto deltaNSquared = _mm256_mul_pd(deltaN, deltaN); // deltaNSquared = deltaN * deltaN
00007FF68936F37C  vmovupd     ymm0,ymmword ptr [deltaN]
00007FF68936F384  vmulpd      ymm0,ymm0,ymmword ptr [deltaN]
00007FF68936F38C  vmovupd     ymmword ptr [rbp+0DE0h],ymm0
00007FF68936F394  vmovupd     ymm0,ymmword ptr [rbp+0DE0h]
00007FF68936F39C  vmovupd     ymmword ptr [deltaNSquared],ymm0
    const auto term1 = _mm256_mul_pd(delta, _mm256_mul_pd(deltaN, n1)); // term1 = delta * deltaN * n1
00007FF68936F3A4  vmovupd     ymm0,ymmword ptr [deltaN]
00007FF68936F3AC  vmulpd      ymm0,ymm0,ymmword ptr [n1]
00007FF68936F3B4  vmovupd     ymmword ptr [rbp+0E60h],ymm0
00007FF68936F3BC  vmovupd     ymm0,ymmword ptr [delta]
00007FF68936F3C4  vmulpd      ymm0,ymm0,ymmword ptr [rbp+0E60h]
00007FF68936F3CC  vmovupd     ymmword ptr [rbp+0E20h],ymm0
00007FF68936F3D4  vmovupd     ymm0,ymmword ptr [rbp+0E20h]
00007FF68936F3DC  vmovupd     ymmword ptr [term1],ymm0

    // m1 = m1 + validMask * deltaN
    m1 = _mm256_add_pd(m1, VectorizationUtils::maskDouble4(deltaN, validMask));
00007FF68936F3E4  vmovdqu     ymm0,ymmword ptr [validMask]
00007FF68936F3E9  vmovdqu     ymmword ptr [rbp+17E0h],ymm0
00007FF68936F3F1  vmovupd     ymm0,ymmword ptr [deltaN]
00007FF68936F3F9  vmovupd     ymmword ptr [rbp+17A0h],ymm0
00007FF68936F401  lea         rdx,[rbp+17E0h]
00007FF68936F408  lea         rcx,[rbp+17A0h]
00007FF68936F40F  call        VectorizationUtils::maskDouble4 (07FF68929491Fh)
00007FF68936F414  vmovupd     ymmword ptr [rbp+0EE0h],ymm0
00007FF68936F41C  mov         rax,qword ptr [this]
00007FF68936F423  vmovupd     ymm0,ymmword ptr [rax]
00007FF68936F427  vaddpd      ymm0,ymm0,ymmword ptr [rbp+0EE0h]
00007FF68936F42F  vmovupd     ymmword ptr [rbp+0EA0h],ymm0
00007FF68936F437  mov         rax,qword ptr [this]
00007FF68936F43E  vmovupd     ymm0,ymmword ptr [rbp+0EA0h]
00007FF68936F446  vmovupd     ymmword ptr [rax],ymm0
```

Figure 9 - Disassembly of part of the code shown in Fig 8 - update of the first moment M1

CpuDeviceCoordinator and Avx2CpuDeviceCoordinator

Depending on the target architecture there are two variants of processing data on SMP / CPU. The first one is called *CpuDeviceCoordinator* which does not use AVX2 vectorization (i.e. it uses *StatsAccumulator*) and it can be either used by configuration in the command line or by default if the program is not compiled with AVX2 vectorization.

The *Avx2CpuDeviceCoordinator* uses the *pushWithFiltering* method of the vectorized variant of the accumulator. This method is used in a for-loop where we take four doubles and push them to the accumulator. Effectively, this makes the loop vectorized since the accumulator itself performs the operation on the entire 256-bit vector of doubles at once. This is depicted in Fig 10.

```
    for (auto accumulatorId = r.begin(); accumulatorId < r.end(); accumulatorId += 1) {
        // Since we are using AVX2 in each step we process 4 doubles at once, therefore the indices must
        // be scaled by 1/4th
        const auto jobStart :unsigned long long = (accumulatorId * doublesPerAccumulator) / 4;
        const auto jobEnd :unsigned long long = jobStart + (doublesPerAccumulator) / 4;
        for (auto i :unsigned long long = jobStart; i < jobEnd; i += 1) {
            accumulators[accumulatorId].pushWithFiltering(x: {
                .m256d_f64[0]: buffer[i * 4],
                .m256d_f64[1]: buffer[i * 4 + 1],
                .m256d_f64[2]: buffer[i * 4 + 2],
                .m256d_f64[3]: buffer[i * 4 + 3],
                });
        }
    }
});
```

Figure 10 - Vectorized for loop in the Avx2CpuDeviceCoordinator

To process the data the coordinator uses the TBB function *parallel_for* where it creates N tasks, depending on the number of accumulators it is processing (i.e. one task per accumulator), and processes each task in one thread. The number of accumulators is determined by the available memory which is configured in the job scheduler. By default, one accumulator processes around 4 MB of data (or less if the file is smaller).

ClDeviceCoordinator

The last important component in the system is called *ClDeviceCoordinator* which is responsible for coordinating work on an OpenCL device. At the start, the coordinator sets up communication with the device using OpenCL C++ functions. This comprises several steps:

- Creation of OpenCL context and command queue
- Compilation of OpenCL kernel for the device
- Estimation of optimal work group size

To estimate the optimal work group size the class uses the compiled kernel to get the value of *CL_KERNEL_PREFERRED_WORK_GROUP_SIZE_MULTIPLE*. This value was, however, very small in the testing environment, and increasing it helped performance dramatically. Therefore, for high-performance Nvidia GPUs (RTX and GTX series), this value is set to 1/8th of the maximum workgroup size.

To process the data, the device is run with a single kernel that emulates the behavior of *StatsAccumulator::push* method, where each work item computes values for exactly one instance of StatsAccumulator. This kernel is invoked several times to ensure enough data is processed by each accumulator. By default, each work item processes around 4 MB of doubles (or less if the file is smaller).

Memory allocation

To ensure that the application does not exceed 1 GB of RAM it needs to keep information about the memory it allocates. The algorithm used here is very simple, where we use part of the memory for runtime structures (i.e. job scheduler, device coordinators, accumulators, etc.) and the rest is allocated for buffers that hold data loaded from the file. Depending on the processing mode (i.e. ALL, OpenCL devices, or SMP) we give each device coordinator according to the amount of memory that it can allocate. If there are more OpenCL devices, the memory for them is split evenly.

Classification

Functionality for the classification of the processed data is implemented in the header *distributionClassification.h* where the stats accumulator object is examined and assigned to the closest distribution. To classify the distribution, the application uses several checks to ensure the classification is correct:

- Check *isInteger* flag to disable Poisson classification for non-integer data
- Check if any numerical errors occurred while merging accumulators from threads / OpenCL devices
- Check minimum value, if it is negative the data cannot come from exponential distribution

To produce a single stats accumulator (that is passed to the classifier), the sub-results are combined together from left to right (i.e. the first one is combined with the second one and the result is added to the third one, and so forth...). If some numerical errors occur during this process, the merging algorithm sets a corresponding flag that is always checked in the classifier and the user is notified with a message.

Performance and Benchmark

Next part of the work was to estimate how much the given parallelization / vectorization techniques improve (or hurt) performance. To do so, we generated a 22.3 GB large file (3 billion doubles) which was processed by each configuration. The list of variants is as follows:

- Single Threaded - TBB constrained to one thread via *tbb::global_control::max_allowed_parallelism*
- Single Threaded with AVX2 vectorization
- SMP
- SMP with AVX2 vectorization
- OpenCL
- ALL
- ALL with AVX2 vectorization

All tests were performed on the following machine:

- CPU - Intel Core i7 9700f 8-Core 8-Thread @ 3 GHz / 4.7 GHz Turbo boost
- RAM - 32 GB (4 × 8 GB) Kingston HyperX Fury DDR4 @ 2666 MHz
- GPU - GIGABYTE Nvidia RTX 3060 Ti 8 GB VRAM Vision OC (Connected via PCIe3.0)
- Disk - WD Blue SN550 NVMe SSD 500GB 2400 MB/s Read
    - CrystalDiskMark shows around 1924 MB/s Read in single-threaded mode
- OS - Windows 11 Pro, 22H2, 22621.819

Unfortunately, the CPU does not have integrated GPU, and thus the only OpenCL device in the system is the Nvidia GPU (the CPU is not detected by the framework either). We used the OpenCL runtime provided in CUDA 11.8.

The benchmark itself is implemented in *Benchmark.h* file. It runs the computation several times (specified by the user) and then computes the best, worst, and average runtime. The output is printed to the console or an output file. Each configuration was run 10 times. The measured results are in Table 1 and visualized in Figure 11.

The executable of the app was compiled in MSVC Release mode with O2 optimization (favor speed) enabled.

Configuration	Average time [s]	Best time [s]	Worst time [s]	% Base performance
Single Threaded	60.60	57.68	63.19	100.00
Single Threaded with AVX2 vectorization	37.55	37.15	37.99	161.40
SMP	39.94	35.92	42.57	151.73
SMP with AVX2 vectorization	30.43	29.28	34.45	199.14
OpenCL	61.11	60.42	62.39	99.17
ALL	32.88	30.56	34.25	184.30
ALL with AVX2 vectorization	26.77	26.28	27.14	226.43

Table 1 - Performance of each configuration on 24GB of data

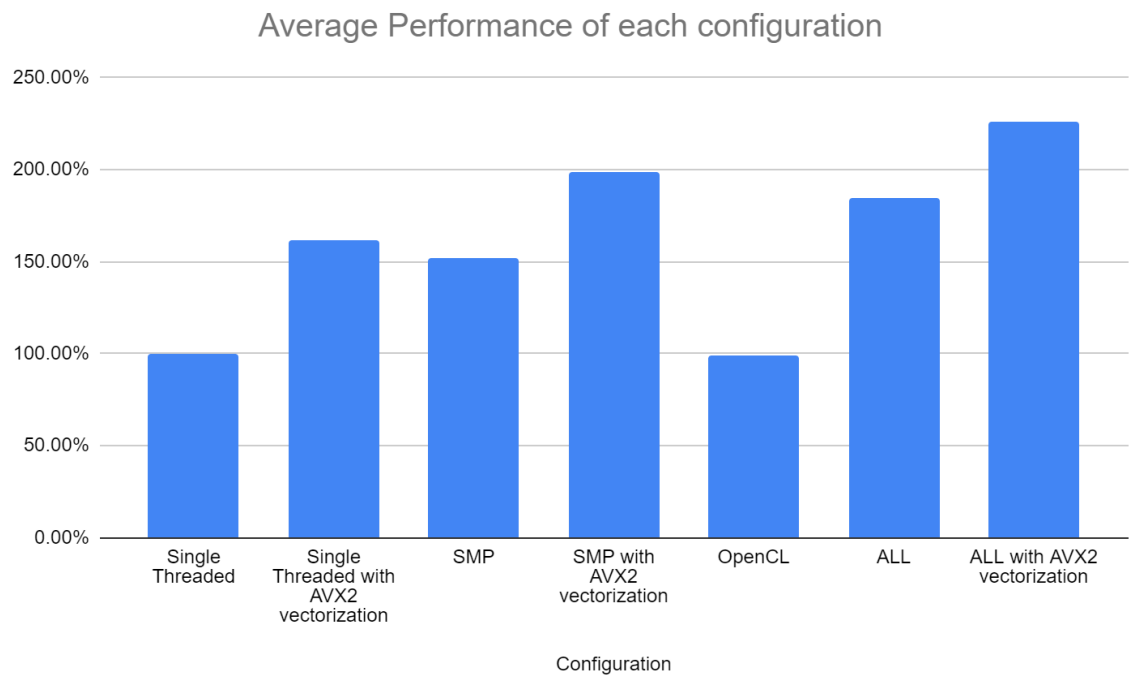

Figure 11 - Average performance of each configuration on 24GB of data

It is reasonable to expect the single-threaded implementation (without AVX2) to perform the worst, however, this was not the case and OpenCL-only run was about 1% worse than this baseline. Such poor performance can be caused by many factors, e.g. by an inefficient copy of the data from the host to the device or by a suboptimal implementation of the algorithm. The issue could also be the limited amount of RAM required for the application (i.e. 1 GB for the entire process) since OpenCL's runtime takes a large amount of memory from the actual code (this is tested later in this section).

From the benchmarked values it is visible that any other form of parallelization or vectorization increases performance dramatically. We can improve performance by 61% with only AVX2 vectorization which is then further improved to almost 100% increase when paired with parallel execution. Surprisingly, though, SMP code without vectorization offers less additional performance (51% improvement) than executing the same code on a single core with vectorization enabled.

While OpenCL itself was very slow it performed very well when used alongside SMP and on average it was 116 and 84% better than the single-threaded variant with and without vectorization respectively.

Improving OpenCL performance

We also attempted to improve the performance of the OpenCL-only execution as it was much slower than any other parallelized variants. The first idea was to increase the amount of memory available to the process. This is available to the user directly from the command line via the -x switch which configures the amount of memory available. We ran the benchmark with 1, 2, and 4 GB of the maximum amount of memory allocated to the process. The results are available in Table 2.

Max amount of available process memory [GB]	Average time [s]	Best time [s]	Worst time [s]	% Base performance
1 (Default)	61.11	60.42	62.39	99.17
2	72.81	66.73	78.66	83.24
4	74.29	69.52	79.38	81.58

Table 2 - Effect of memory increase on performance

Surprisingly, increasing the amount of available memory did not improve the OpenCL performance as twice the memory available increased the average execution time to 72.81 seconds and four times the memory yielded even slightly worse 74.29 seconds execution time.

Additionally, we also tried different workgroup sizes (the default for this GPU is 128) - 64, and 256. The results are available in Table 3. Again, the default values seem to perform the best and increasing the number of workgroup items to 256 performed about 5.2 seconds worse than the original 128 while 64 performed 6.4 seconds worse. Therefore, we kept the default values intact.

Workgroup size	Average time [s]	Best time [s]	Worst time [s]	% Base performance
64	67.51	65.58	69.24	89.77
128 (Default)	61.11	60.42	62.39	99.17
256	66.31	64.48	68.69	91.40

Table 3 - Performance of the GPU on different workgroup sizes

User guide

The program was developed and tested only on the Windows platform and would most likely require additional changes to port it to Linux or macOS (mainly, since MSVC was used for compilation). The program can be compiled using *checker.exe* with proper paths set to TBB and OpenCL libraries.

To run the application open the command line of your choice and run:

```
pprsolver.exe <FILE_PATH> <MODE> <DEVICES?...>
```

Where *FILE_PATH* specifies the absolute or relative path to the file we want to process, *MODE* is either *SMP*, *SINGLE_THREAD*, *OPENCL_DEVICES*, or *ALL*. *OPENCL_DEVICES* mode can be omitted. *DEVICES* specifies a list of devices - each device name must be specified in quotes, e.g.:

```
pprsolver.exe myfile.bin “Nvidia GTX 1060”
```

Optional arguments

The user can also run the application with additional arguments:

- `-l, --list_cl_devices`
  - Lists all available OpenCL devices.
- `-x, --memory_limit`
  - Sets memory limit in MB, this value can be between 1024 to 4096
- `-b, --benchmark`
  - Runs the application in benchmark mode performing N runs
- `--benchmark_runs`
  - Specifies the number of runs performed by the benchmark, the default value is 10
- `-o, --output_file`
  - Specifies output file name, this officially only works with files in the same directory
- `--disable_avx2`
  - Disables AVX2 vectorization
- `-t, --watchdog_timeout`
  - Specifies watchdog timeout in seconds
- `-h, --help`
  - Prints help to the console.

Conclusion

The application can estimate distribution on the provided testing data. The fastest variant for processing large files is the ALL variant, which performed 126% better than the single-threaded variant without AVX2 vectorization. The program also contains additional features to simplify its use such as a list of available OpenCL devices, a built-in benchmark, and output to a file alongside standard output.

Surprisingly, the slowest variant is the OpenCL-only execution. This could, however, be further improved, e.g., by improving the code coordinating the OpenCL execution or improving the kernel code. Alternatively, we could also rewrite the code for CUDA, which often performs faster, but also constraints the application to the Nvidia platform.