FAKULTA APLIKOVANÝCH VĚD
ZÁPADOČESKÉ UNIVERZITY
V PLZNI

KIV/PPR
Estimating the distribution of data using parallelized algorithm

Václav Honzík
A21N0045P
29. 11. 2022

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 subresult. 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.

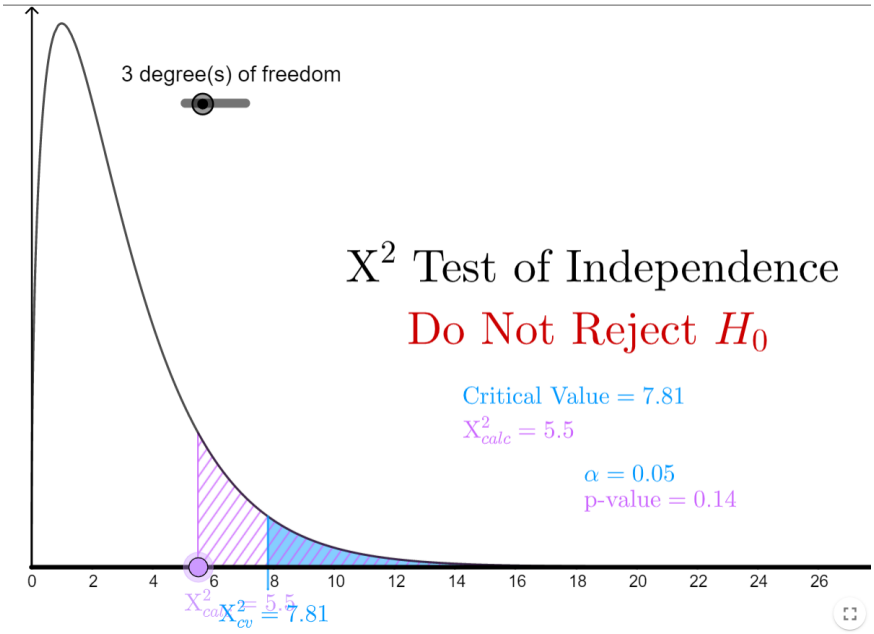

Figure 2 - Visualization of Chi Square Goodness of Fit test.
Source: https://www.geogebra.org/m/smhy8cxz

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

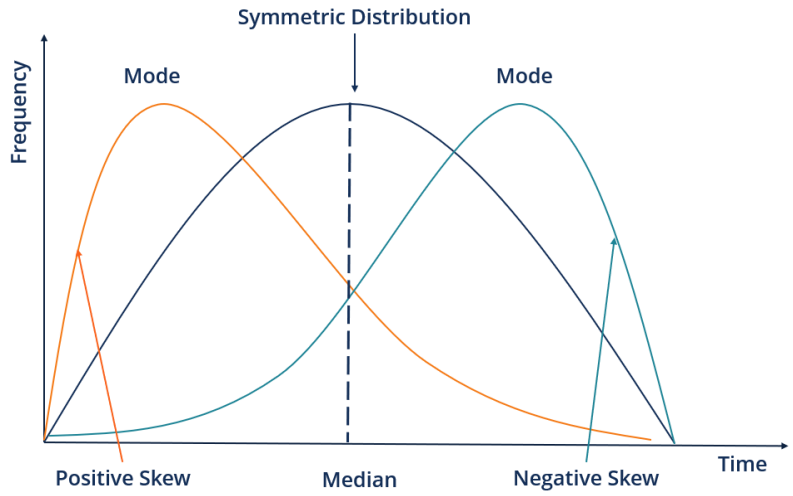

Figure 3 - Visualization of skewness
Source: https://corporatefinanceinstitute.com/resources/data-science/skewness/

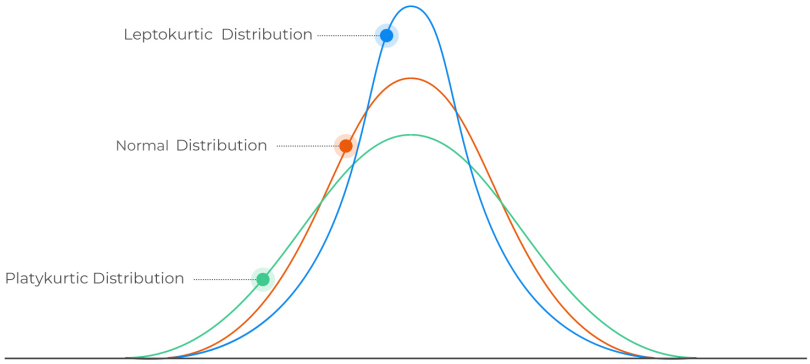

Figure 4 - Visualization of Kurtosis
Source: https://analystprep.com/cfa-level-1-exam/quantitative-methods/kurtosis/

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, though, 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.

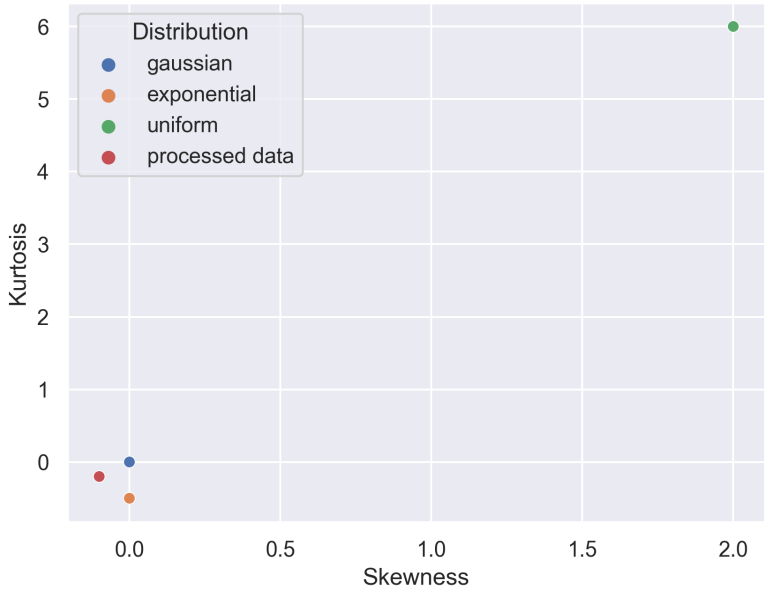

Figure 5 - Visualization of Gaussian, exponential and uniform distributions in 2D space. "Processed data" is a point computed from the processed file that we then assign to the closest point

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 data- 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 which 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.

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, 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 here:
https://www.johndcook.com/blog/skewness_kurtosis/

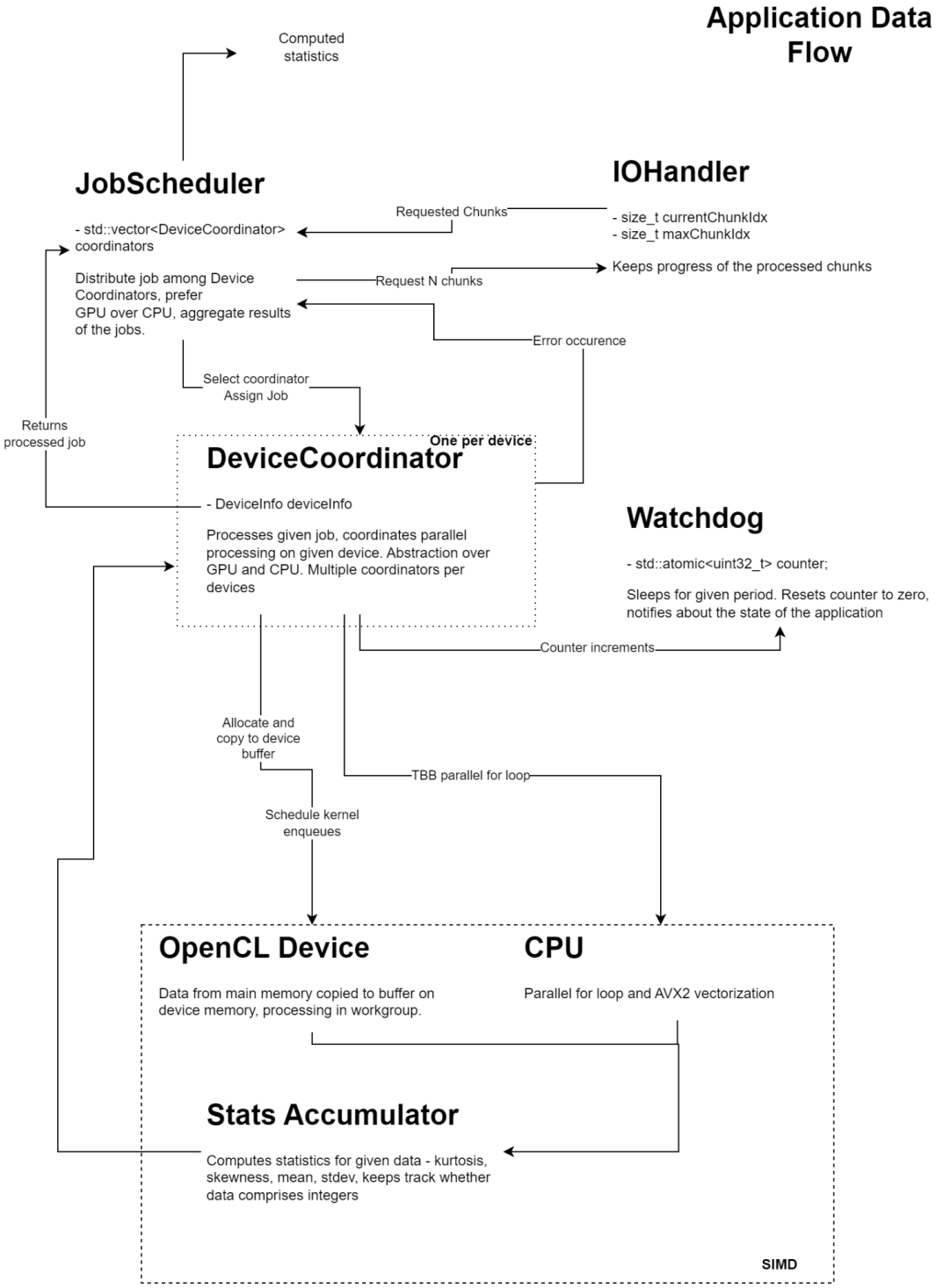

Figure 6 - Data flow diagram of the application

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. To make the code human-readable the most used AVX2 functions' names are aliased in the *VectorizationUtils.h* header.

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. The code for updating the first moment can be seen in Figure 8. Figure 9 shows part of the generated assembly.

```
inline auto valuesValid(const double4& x)->__m256i {
    const auto bits:__m256i = castDouble4ToInt4(x); // convert x to integer vector
    auto exponent:__m256i = int4And(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 = int4CompareEquals(exponent, _mm256_setzero_si256());

    // AND bits & MANTISSA_MASK > 0
    auto bitsAndMantissa:__m256i = int4And(bits, MANTISSA_MASK);
    bitsAndMantissa = int4CompareGreaterThan(bitsAndMantissa, _mm256_setzero_si256());

    // If exponent == 0 && bits & MANTISSA_MASK set to true
    const auto invalid1:__m256i = int4And(expEqualsZero, bitsAndMantissa);

    // Similarly if exponent == 0x7ff it is invalid as well
    const auto invalid2:__m256i = int4CompareEquals(exponent, int4Set(0x7ff));

    // Combine both with OR operation
    // This will return for each element in the vector if they are invalid
    const auto invalid:__m256i = int4Or(invalid1, invalid2);

    // Negate - i.e. XOR with 0xFFFFFFFFFFFFFFFF == int64_t(-1) == UINT64_MAX
    return int4Xor(invalid, int4Set(UINT64_MAX));
}
```

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

```
// 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 = int4Xor(validMask, int4Set(UINT64_MAX));
const auto isIntegerDistributionUpdate:__m256i = int4Or(validNegation, isInteger);

// Update isIntegerDistribution = isIntegerDistribution && !valid || isInteger
isIntegerDistribution = int4And(isIntegerDistribution, isIntegerDistributionUpdate);

const auto n1:__m256d = convertInt4ToDouble4(n); // n1 = n, we convert to double to avoid casting later
n = int4Add(n, int4And(int4Set(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 = double4Sub(x, m1); // delta = x - m1
const auto deltaN:__m256d = double4Div(delta, nDouble); // deltaN = delta / n
const auto deltaNSquared:__m256d = double4Mul(deltaN, deltaN); // deltaNSquared = deltaN * deltaN
const auto term1:__m256d = double4Mul(delta, double4Mul(deltaN, n1)); // term1 = delta * deltaN * n1

// m1 = m1 + validMask * deltaN
m1 = double4Add(m1, VectorizationUtils::maskDouble4(deltaN, validMask));
```

Figure 8 - Update of the first moment in the accumulator

```
    // m1 = m1 + validMask * deltaN
    m1 = double4Add(m1, VectorizationUtils::maskDouble4(deltaN, validMask));
00007FF67F0C2666  vmovdqu     ymm0,ymmword ptr [validMask]
00007FF67F0C266B  vmovdqu     ymmword ptr [rbp+1660h],ymm0
00007FF67F0C2673  vmovupd     ymm0,ymmword ptr [deltaN]
00007FF67F0C267B  vmovupd     ymmword ptr [rbp+1620h],ymm0
00007FF67F0C2683  lea         rdx,[rbp+1660h]
00007FF67F0C268A  lea         rcx,[rbp+1620h]
00007FF67F0C2691  call        VectorizationUtils::maskDouble4 (07FF67EFF08E3h)
00007FF67F0C2696  vmovupd     ymmword ptr [rbp+0DE0h],ymm0
00007FF67F0C269E  mov         rax,qword ptr [this]
00007FF67F0C26A5  vmovupd     ymm0,ymmword ptr [rax]
00007FF67F0C26A9  vaddpd      ymm0,ymm0,ymmword ptr [rbp+0DE0h]
00007FF67F0C26B1  vmovupd     ymmword ptr [rbp+0DA0h],ymm0
00007FF67F0C26B9  mov         rax,qword ptr [this]
00007FF67F0C26C0  vmovupd     ymm0,ymmword ptr [rbp+0DA0h]
00007FF67F0C26C8  vmovupd     ymmword ptr [rax],ymm0
```

Figure 9 - Update of M1 - disassembly

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 command line or by default if the program is not compiled with AVX2 vectorization.

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 16 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 32 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.

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 23.8 GB large file (5,568,477,184 bytes) 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 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
- 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. 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 10.

Configuration	Average time [s]	Best time [s]	Worst time [s]	% base performance
Single Threaded	60.603	57.678	63.193	100.000
Single Threaded with AVX2 vectorization	33.187	30.651	34.624	182.610
SMP	40.334	39.278	41.838	150.253
SMP with AVX2 vectorization	30.904	30.051	31.921	196.101
OpenCL	37.736	37.348	38.403	160.597
ALL	44.613	41.725	47.156	135.841
ALL with AVX2 vectorization	29.417	28921	29.835	206.0135

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

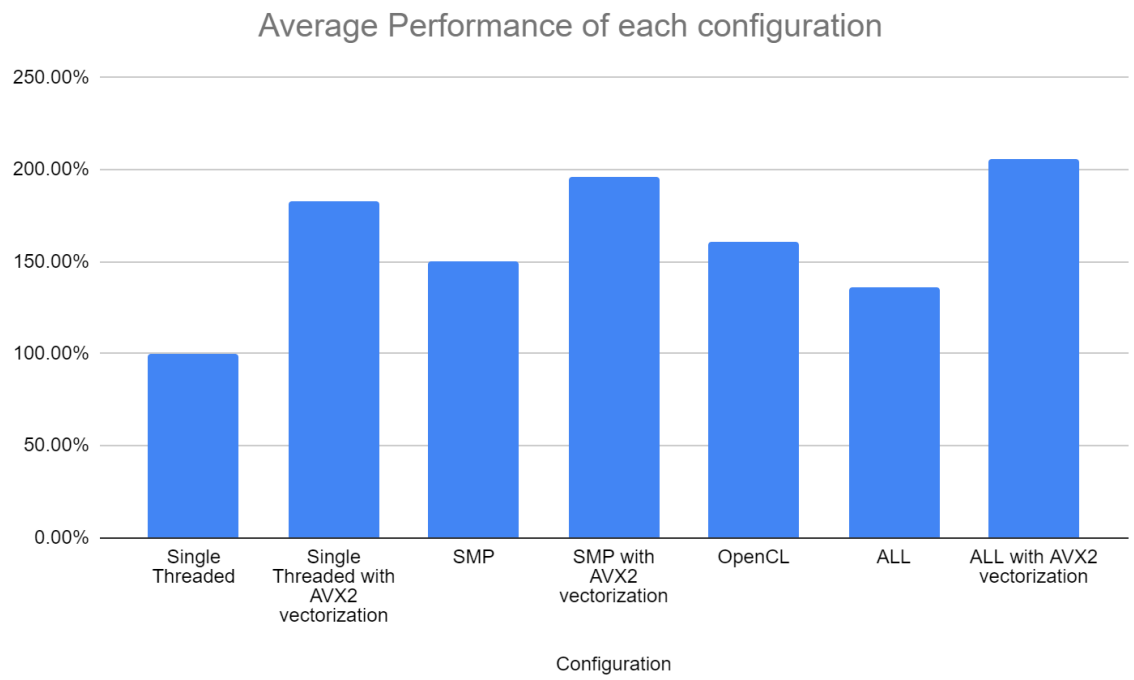

Figure 10 - Average performance of each configuration on 24GB of data

As expected, the single-threaded variant without any further enhancements performed the worst. This might not be the case for smaller sizes where we would waste more time distributing the work among CPU threads but it shows how much performance can be gained if we have large enough data.

What is more surprising, however, is the fact that single-threaded configuration with vectorized code performs much better than using SMP without vectorization. The single-threaded code with the AVX2 instructions shows about 82% improvement over the baseline and is 32% better than the pure SMP variant. This is also the case when we test ALL devices - where ALL without vectorization performs only 35% better than the baseline variant whereas with vectorization enabled we achieve almost double the performance.

The openCL variant shows about 60% improvement compared to the baseline which is disappointing as it does not perform better than its SMP counterpart. On the other hand, if we combine both using the ALL mode (with vectorization), we achieve the best result which is around 106% improvement. The performance of the OpenCL could be explained by a relatively low amount of memory and the amount of data that we need to write data from RAM to the GPU's VRAM for each calculation which could be reduced if we allowed the application to use more RAM.

During the development, there was also an issue with the recommended work group size obtained from *CL_KERNEL_PREFERRED_WORK_GROUP_SIZE_MULTIPLE* which returns 32 for this GPU. Unfortunately, this workgroup size is very small and the card is not 100% utilized. Therefore, for Nvidia GPUs similar to this (i.e. RTX and GTX series) the workgroup size is increased to 1/8th of the maximum (i.e. workgroup size of 128). Increasing this value further did not improve performance.

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 (especially 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 number of runs performed by the benchmark, default value is 10
- `-o, --output_file`
  - Specifies output file name
- `--disable_avx2`
  - Disables AVX2 vectorization
- `-t, --watchdog_timeout`
  - Specify watchdog timeout
- `-h, --help`
  - Prints help to the console

Conclusion

The application can correctly estimate distribution on the provided testing data. The fastest variant for processing large files is the ALL variant, which performed 106% better than the single-threaded variant without AVX2 vectorization. The application was tested on the Windows platform with Intel CPU and Nvidia GPU.