Performance Optimization - Assignment 2

1 Technical Specifications

Blur
AMD Ryzen 7 77730U with Radeon Graphics

- cores: 8
- logical processors: 16

16 GB RAM

Pearson
12th Gen Intel(R) Core(TM) i7-1255U

- cores: 10
- logical prozessors: 12

16 GB RAM

2 Blur

2.1 Baseline Measurements

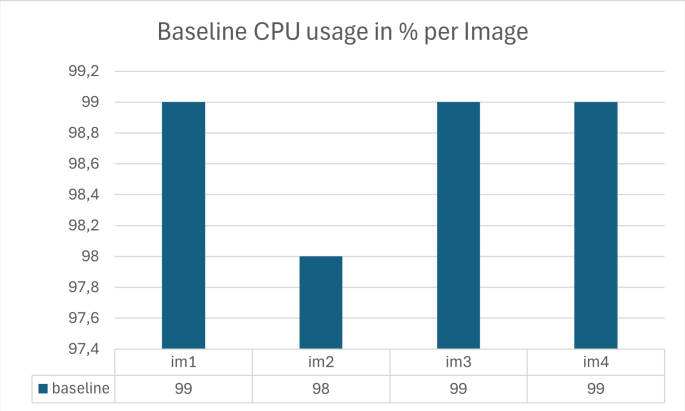


Figure 1: CPU usage in % for baseline

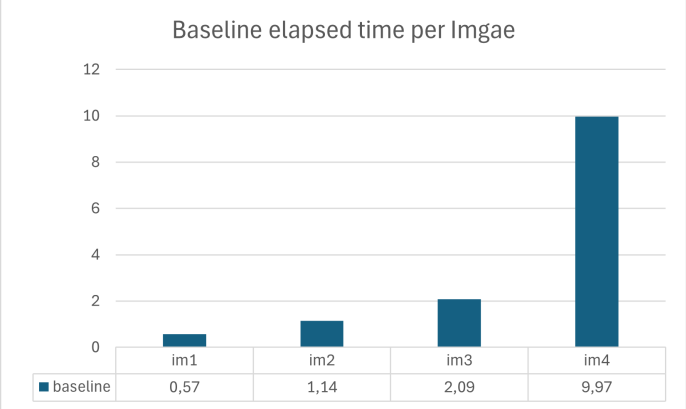


Figure 2: Elapsed time in seconds for baseline

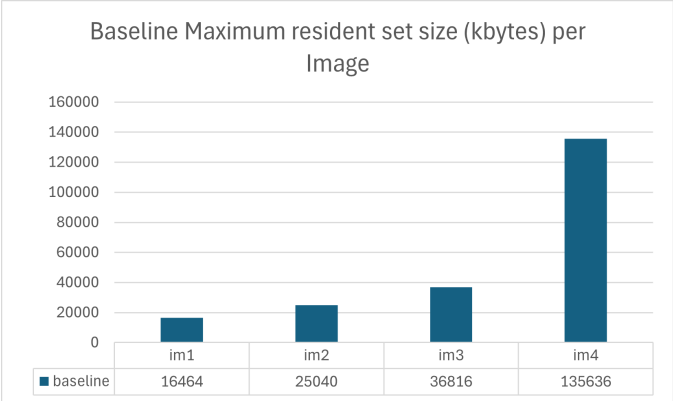


Figure 3: Maximum resident set size (kbytes) for baseline

The CPU usage recorded for the baseline version of blur was very high, between 98-99%. This led us to the conclusion that the blur program was CPU bound and that the CPU was the bottleneck. In order to understand what in the blur program was using most of the time of the CPU we analyzed the call-graphs (Figure 15-21). There we could see two problems that we explain in the Methods part.

2.2 Methods

2.2.1 Sequential Program

Optimization 1 - Loop invariant computation When analyzing the call-graph from valgrind callgrind of the baseline version im1.ppm, we found that Filters::blur(Matrix m, const int radius) was acquiring a significant amount of percent of the CPU time 42,4% (Figure 16). After further investigation we also found the line double w[Gauss::max_radius]{}; in blur() was getting 7,58% of the CPU time (Figure 17) and that blur() was calling
Gauss::get_weights(radius, w); 1 031 576 number of times that took 7,26% of the CPU (Figure 16).
We observed that w was reallocated and assigned a value multiple times even though it never needed to be reallocated or its value never changed during runtime. The reason for this is that the allocation and call to get_weight to assigning a value was placed in two nested for loops. By having a larger image the two lines would get called even more and take more time.

For this optimization we used loop invariant computation and moved calculations that did not change between iterations of loop and moved them outside the loop.

Following changes where done:

filters.cpp
Changed this code in two places from:

```
for (auto x{0}; x < dst.get_x_size(); x++)
{
    for (auto y{0}; y < dst.get_y_size(); y++)
    {
        double w[Gauss::max_radius]{};
        Gauss::get_weights(radius, w);
```

to one place:

```
double w[Gauss::max_radius]{};
Gauss::get_weights(radius, w);

for (auto x{0}; x < dst.get_x_size(); x++)
{
    for (auto y{0}; y < dst.get_y_size(); y++)
    {
```

This way the two lines only get called the number of times blur() gets called and reduces the time spent allocating and computation w.

Optimization 2 - Function inlining After the first optimization we observe that most of the CPU time was spent on calculations surrounding these types of lines:

```
r += wc * dst.r(x2, y);
g += wc * dst.g(x2, y);
b += wc * dst.b(x2, y);
n += wc;
```

The functions r(), g() and b() were called 34 212 436 times each and occupied 13,99 % of the CPU time each (Figure 19). We stated that this is normal because that is were a lot of the computation is done. However, when looking closer into how the r(), g(), b() functions work we could see that the computation for example B[y * x_size + x] took 7,77 % but also that the surrounding overhead of calling the function took 6,21 % in total (Figure 21). By having larger images the functions would get called even more and create more overhead.

For this optimization we used Function inlining to optimized the code by removing the overhead of calling many functions by combining them into one function instead.

Following changes where done:

matrix.hpp
Added rgb() function declaration:

```
void rgb(unsigned x, unsigned y, double wc,
    double &r, double &g, double &b);
```

matrix.cpp
Added rgb() function definition:

```
void Matrix::rgb(unsigned x, unsigned y, double wc,
    double &r, double &g, double &b)
{
    r += wc * R[y * x_size + x];
    g += wc * G[y * x_size + x];
    b += wc * B[y * x_size + x];
}
```

filters.cpp
Changed the code in two places from:

```
r += wc * dst.r(x2, y);
g += wc * dst.g(x2, y);
b += wc * dst.b(x2, y);
n += wc;
```

to:

```
dst.rgb(x2, y, wc, r, g, b);
n += wc;
```

Changed the code in two places from:

```
r += wc * scratch.r(x, y2);
g += wc * scratch.g(x, y2);
b += wc * scratch.b(x, y2);
n += wc;
```

to:

```
scratch.rgb(x, y2, wc, r, g, b);
n += wc;
```

2.2.2 Parallelized Program

The parallelization of the blur was done on top of the optimized version. For the parallelized version of blur we wanted the different threads to work on different parts of the data at the same time. The first step was to decide how to divide the matrix for the different threads. We decided to divide the matrix data block-wise by dividing the x axis to different threads. We first divided the size of the x axis with the number of threads to get the base, in order to handle if the x axis is odd we use the modular and add one if odd. The start and end of the x axis was passed to the threads through the struct ThreadData explained later. This code to the right is for dividing the x axis:

```
int mSizeX = m.get_x_size();
int base = mSizeX / nrOfThreads;
int rem = mSizeX % nrOfThreads;
int y_max = m.get_y_size();
int start;
int add;
int end;
int next_start = 0;
for (int i = 0; i < nrOfThreads; i++)
{
    start = next_start;
    add = base + (i < rem ? 1 : 0);
    end = start + add;
    next_start = end;
    ...
```

The main function in blur_par.cpp had the flowing changes:

- Added so possible to specify the number of threads blur should run
- Gave values to the different threads arguments
- Calculations for dividing the x axis to different threads
- Creating threads depending on thread amount given
- Joining of threads when completed to make into one
- Initialization of pthread barrier
- Destroy pthread barrier

In order to send specific data to the threads a struct ThreadData was created in filters.hpp. A array of ThreadData with the size of the number of threads was created and the values was assigned in blur.cpp main function. The unique array value was sent with the thread that called the function blur in filters.cpp.
This was the struct:

```
    struct ThreadData
{
    Matrix *m;
    Matrix *blurred;
    Matrix *scratch;
    int radius;
    int x_max;
    int x_min;
    int y_max;
};
```

The x_max gave the value of what x coordinate to start blurring the image and x_min gave the value of where to stop. The m Matrix was the original image Matrix that had the base data, the blurred Matrix was the return image and the scratch Matrix was the middle image during blurring. We made the Matrixes pointers so that for blurred and scratch the threads could work on the same Matrix and the m Matrix was made a pointer to avoid copying overhead for each thread.

For blur in filters.cpp and filters.hpp in order to parallelize this function we had to change the function form:

```
Matrix blur(Matrix m, const int radius)
```

to:

```
void *blur(void *args)
```

We also had to add this line for the arguments:

```
ThreadData *data = (ThreadData*)args;
```

Another change to filters.cpp was that the function needed to use the ThreadData values.
Example of change, from:

```
scratch.r(x, y) = r / n;
scratch.g(x, y) = g / n;
scratch.b(x, y) = b / n;
```

to:

```
data->scratch->r(x, y) = r / n;
data->scratch->g(x, y) = g / n;
data->scratch->b(x, y) = b / n;
```

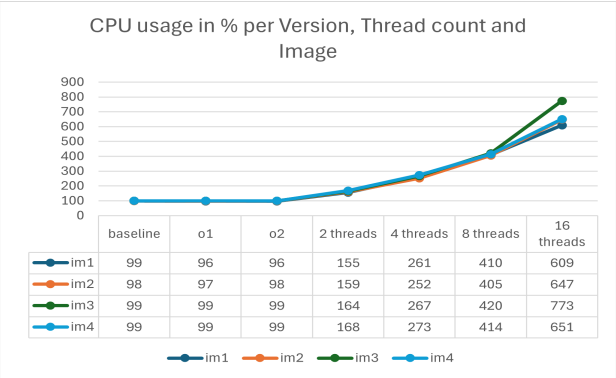
(Since the threads are not reading and writing to the same coordinates there is no need for mutex-locks here)

Because the blur function was done in two parts (two nested for-loops) that was dependent on the first data being done before starting the next, the threads had to wait for each other before continuing and not continue as normal. To make the threads wait for each other, a pthread_barrier was placed between the parts. The barrier was initialized with the value of the number of threads, which made the barrier block continue until all threads had reached the barrier.

```
pthread_barrier_wait(&barrier);
```

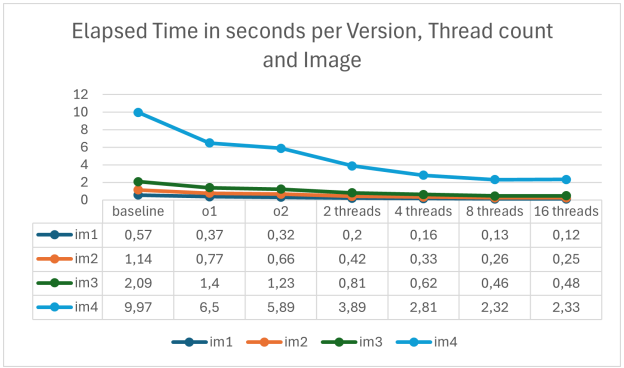
(Note: we found for most of the time for the thread counts we tried the threads were synchronized anyways, however to be sure no accuracy problems would occur we made the threads wait for each other.)

2.3 Results



	baseline	o1	o2	2 threads	4 threads	8 threads	16 threads
im1	99	96	96	155	261	410	609
im2	98	97	98	159	252	405	647
im3	99	99	99	164	267	420	773
im4	99	99	99	168	273	414	651

Figure 4: CPU usage in % for every blur version and image size



	baseline	o1	o2	2 threads	4 threads	8 threads	16 threads
im1	0,57	0,37	0,32	0,2	0,16	0,13	0,12
im2	1,14	0,77	0,66	0,42	0,33	0,26	0,25
im3	2,09	1,4	1,23	0,81	0,62	0,46	0,48
im4	9,97	6,5	5,89	3,89	2,81	2,32	2,33

Figure 5: Elapsed time in seconds for every blur version and image size

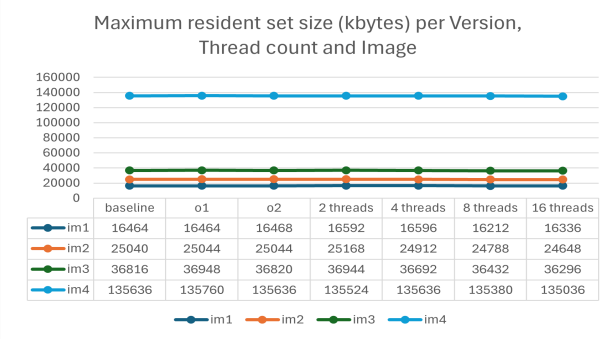


Figure 6: Maximum resident set size (kbytes) for every blur version and image size

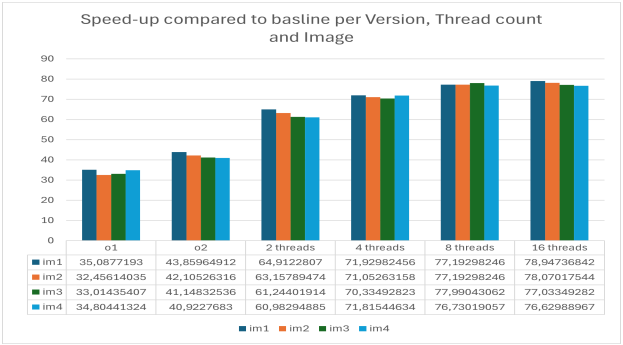


Figure 7: Speed-up for every version, thread count and Image in %

2.3.1 Sequential Program

Optimization 1

The first optimization achieved a speed-up of 35,09% that was 1,540540541 times faster than the baseline of blur for image 1. For image 4 the speedup was 1,533846154 times faster with the speed up in percent being 34,8% compared to the baseline version. The lowest speed up was for image 2 with 32,5% speed-up.

Optimzation 2

The second optimization were a bit smaller then the first one but did still improve the Elapsed time. The highest speed-up for optimization 1 and 2 combined compared to the baseline was image 1 that was 1,78125 times faster than the baseline with a speed-up of 43,9%. The lowest speed-up recorded was image 4 with 1,692699491 times speed-up that is equivalent to 41% speed-up.

The CPU percentage stayed high at 96-99% for all sequential versions (baseline, o1 and o2). For The first optimization one can see that the CPU % stayed pretty much the same compared to the baseline with 2-3 % decrease for images 1 and 2. The Maximum resident set size did not change a lot for any of the optimizations, with the largest change being image 3 with a change of -128 kbytes from optimization 1 to 2.

2.3.2 Parallelized Program

The parallelized version of blur was built on top of the optimized version and the maximum of speed-up measured was 4,75 times faster than the baseline version equivalent to 79% speed-up (blurring image 1 with 16 threads). For all image sizes one can see that the Elapsed time decreases with higher thread count up to 16 threads for image 1-2 and up to 8 threads for image 3-4. When increasing the number of threads, the CPU percentage also increased due to the cores worked simultaneously but in a shorter time period. One can see the highest CPU % of 773 % was blurring Image 3 with 16 threads. The Maximum resident set size in kbytes stayed next until the same for all threads with a slight decrease and increase depending on the Image.

2.4 Additional optimizations steps

One additional optimization step would be to move do the weight calculations into main() that creates the threads instead of having it in every blur() thread. In this case the w variable would be a part of the data struct ThreadData and get sent that way to every thread. This would increase the performance when running blur with multiple threads.

3 Pearson

3.1 Baseline Program

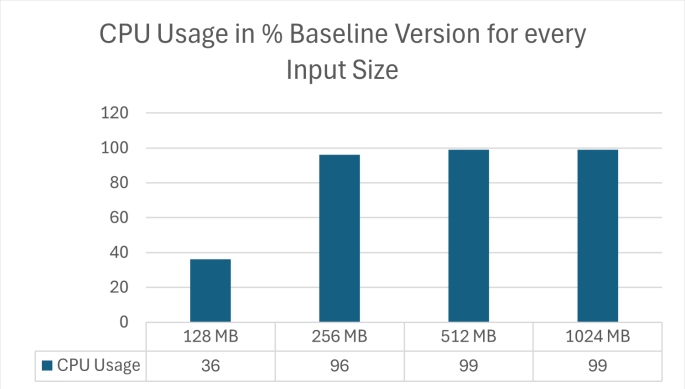


Figure 8: CPU usage in % for the basic Pearson version and different file sizes

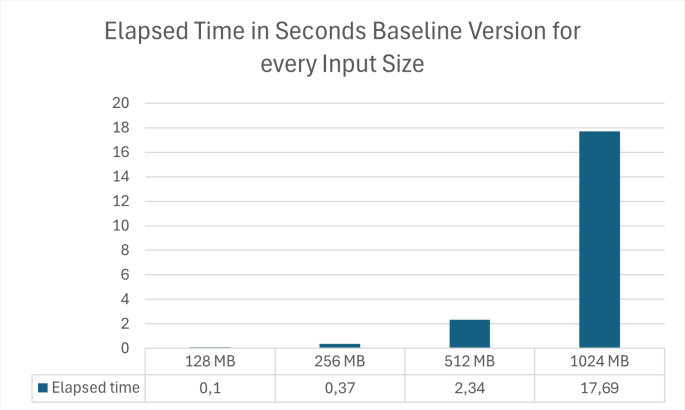


Figure 9: Elapsed time in seconds for the basic Pearson version and every file size

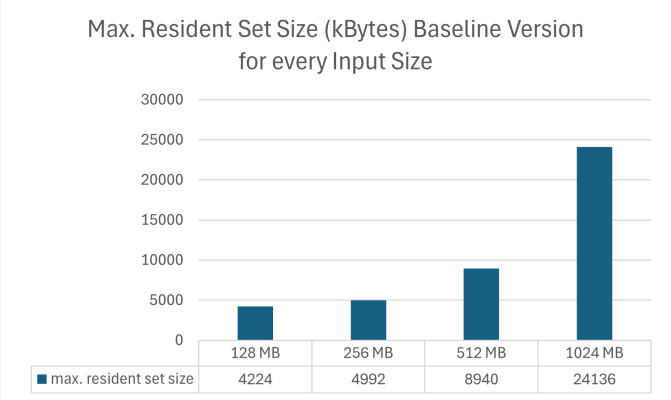


Figure 10: Maximum resident set size (kbytes) for the basic Pearson version and every file size

The basic Pearson version is very computationally heavy. This can be seen in Figure 8. With increasing workload, Pearson reaches a 96% of CPU usage up to 99%. This leads us to believe that the bottleneck in Pearson is the CPU; therefore, our Optimizations have to goal to reduce some of the algorithmic complexity of Pearson to make it more CPU-friendly and therefore faster.

3.2 Methods

3.2.1 Sequential Program

Optimization 1 - working on references instead of objects
After analysing the valgrind call-graph of the baseline version we saw that a significant amount of the time was spent in the copy constructor of the new vector class. Roughly 35.94% of the runtime. In total it was called 8.128 times. When taking a closer look at the code of the baseline version we see that every function that is implemented, e.g. correlation_coefficients() and pearson() two of the main functions, works with vector objects. This has the effect that the input vectors get copied in the runtime process several time leading to an increase in calls to the copy constructor. In this optimization we focused on handing over references to functions, in order to decrease the call frequency to the copy construction.

Following changes where done:

analysis.hpp
Changed function declaration to:

```
std::vector<double> correlation_coefficients(std::
    vector<Vector>& datasets);
```

Changed function declaration to:

```
double pearson(Vector& vec1, Vector& vec2);
```

analysis.cpp
Changed function definition to:

```
std::vector<double> correlation_coefficients(std::
    vector<Vector>& datasets);
```

Changed function definition to:

```
double pearson(Vector& vec1, Vector& vec2);
```

Optimization 2 - faster computations Looking closer into the code and the amount of calls to the copy constructor lead us to following lines of code:

```
double pearson(Vector& vec1, Vector& vec2)
{
    auto x_mean { vec1.mean() };
    auto y_mean { vec2.mean() };
    auto x_mm { vec1 - x_mean };
    auto y_mm { vec2 - y_mean };
    auto x_mag { x_mm.magnitude() };
    auto y_mag { y_mm.magnitude() };
    auto x_mm_over_x_mag { x_mm / x_mag };
    auto y_mm_over_y_mag { y_mm / y_mag };
    ...
}
```

We see, that for the normalization and the division with the scalar product we create new vectors. Which needs copying, what we want to do less of. Our changes had the goal to do parts of the calculation in place to reduce calls to the copy constructor.

We made following changes:

mean_normalize()
We added a new function called `mean_normalize()` that computes the mean, subtracts it from every element, computes the magnitude of the new values and divides the new vector by the computed magnitude. This function does these operations in place, which safes copy instructions.

```
void Vector::mean_normalize() {
    auto m {this->mean()};
    for (int i{0}; i < size; ++i) {
        data[i] -= m;
    }
    auto mag {this->magnitude()};
    for (int i{0}; i < size; ++i) {
        data[i] /= mag;
    }
}
```

using mean_normalize() Now that we have this new function we used it right before the pearson computation.

```
std::vector<double> correlation_coefficients(std::
    vector<Vector>& datasets)
{
    ...
    for (auto& vec : datasets)
        vec.mean_normalize();

    for (auto sample1 { 0 }; sample1 < datasets.
        size() - 1; sample1++) {
        ...
    }
    ...
```

change in dot product
We also changed the dot product slightly to be able to work on a reference and not on an object.
The new function looks like this:

```
double Vector::dot(const Vector& rhs) const
{
    double result{0.0};
    for (size_t i = 0; i < size; ++i) {
        result += data[i] * rhs[i];
    }
    return result;
```

```
    }
```

We also had to change the magnitude function to not const.

3.2.2 Parallelized Program

For the parallelized version of Pearson the goal was to divide the data in a way that each thread can work separately from other threads on given data without tempering or relying on data/computations from the other threads. The approach was based on splitting the pairwise computations between the threads. We made following changes:

pearson.cpp

The main function in pearson.cpp became the thread management function. Essentially this function:

- normalizes the data beforehand

```
for (auto& vec : datasets)
    vec.mean_normalize();
```

- makes calculations on how many pairs per thread they are

```
int total_pairs = size_dataset * (size_dataset -
     1)/2;
int base = total_pairs / n_threads;
int rem = total_pairs % n_threads;
```

- calculates the beginning and the end of each threads range and makes sure they don't overlap

```
for(int t = 0; t < n_threads; ++t){
    int start = next_start;
    int add = base + (t < rem ? 1 : 0);
    int end = start + add;
    next_start = end;
    ...
```

- fills the thread data struct and keeps the threads organized in an array

```
std::vector<pthread_t> threads(n_threads);
std::vector<Analysis::ThreadData> thread_data(
    n_threads);
...
for(int t = 0; t < n_threads; ++t){
    ...
    thread_data[t] = {&datasets, start, end, {}};
    ...
```

- creates the threads

```
    ...
    pthread_create(&threads[t], nullptr, Analysis
        ::correlation_coefficients, &thread_data[
        t]);
}
```

- joins the thread results into one

```
std::vector<double> corrs;
for(int t = 0; t < n_threads; ++t){
    pthread_join(threads[t], nullptr);
    corrs.insert(corrs.end(),
    thread_data[t].results.begin(),
    thread_data[t].results.end());
}
```

Now the main function creates the threads that run the correlation_coefficients function with the thread data given.

analysis.cpp

The new correlation_coefficients function does the following things:

- takes in void* pointer as arguments

```
void* correlation_coefficients(void* arg)
```

- casts and saves the input arguments

```
auto thread_data =
static_cast<ThreadData*>(arg);
auto& datasets = *thread_data->datasets;
std::vector<double>&
results = thread_data->results;
int size_dataset =
static_cast<int>(datasets.size());
int start = thread_data->start;
int end = thread_data->end;
```

- for every pairwise combination it looks if it's part of the thread's range to compute, if yes, it does so, if not, it does nothing (this utilizes the input data)

```
for (auto sample1 { 0 }; sample1 < size_dataset
    - 1; sample1++) {
   for (auto sample2 { sample1 + 1 }; sample2 <
       size_dataset; sample2++) {
      if(k >= start &&k < end){
         auto corr { pearson(datasets[sample1],
              datasets[sample2]) };
         results.push_back(corr);
      }
```

```
        ++k; if(k >= end) break;
    }
    if(k >= end) break;
}
```

Here ,there is a possibility for more optimization by playing around with the nesting of the loops and the if check.

analysis.hpp

Due to the changes in analysis.cpp following changes were made to analysis.hpp:

- changed the correlation_coefficients() function interface

```
void* correlation_coefficients(void* thread_data);
```

- added the thread data struct

```
struct ThreadData{
    std::vector<Vector>* datasets;
    int start;
    int end;
    std::vector<double> results;
};
```

3.3 Results

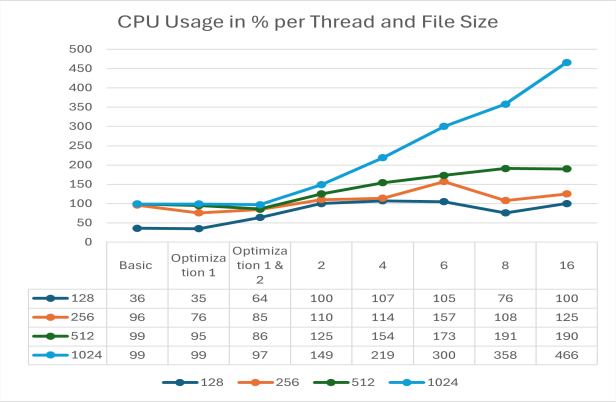


Figure 11: CPU usage in % for every pearson version and input size

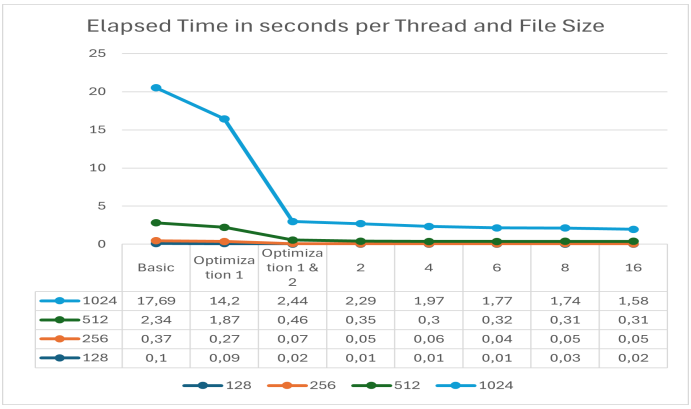


Figure 12: Elapsed time in seconds for every pearson version and input size

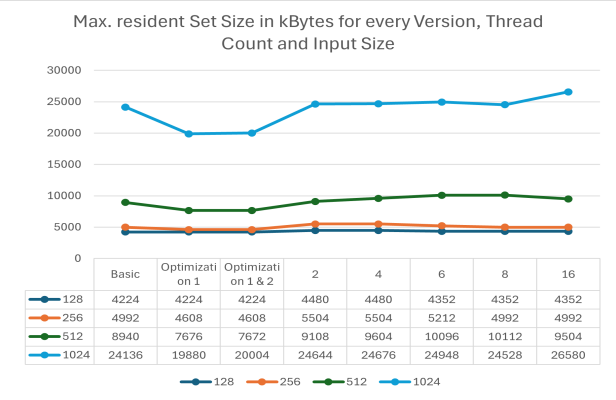


Figure 13: Maximum resident set size (kBytes) for every version, thread count, and input file size

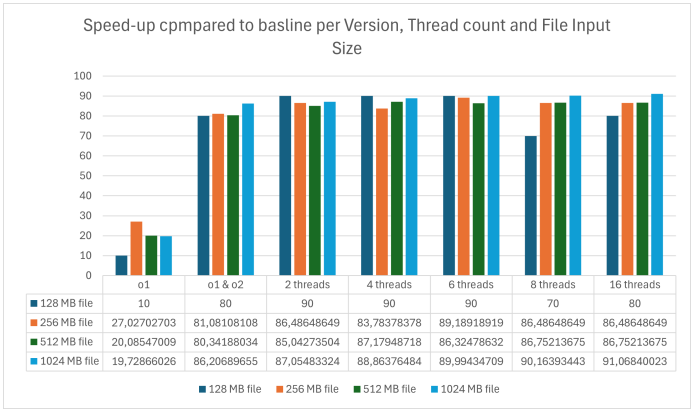


Figure 14: Speed-up for every version, thread count ,and file size in %

3.3.1 Sequential Program

Optimization 1

We can see that the first optimization already achieved a small speed-up of running 1.111111111x faster than the basic Pearson implementation, increasing its speed by 10% (using the smallest possible file size). Using the biggest file size, Pearson runs 1.245774648x faster than the basic version, increasing its speed by 19.72866026%. The fastest increase can be reached using the 256MB file as input. There, the Pearson implementation runs 1,37037037 times faster, with a speed increase of 27.02702703%.
In figure 25 we can even see that the amount of time that was spend in the copy constructor was reduced to 20.43%.

Optimization 1 & 2

Using both optimizations for the sequential version using the smallest filesize, we achieve a speed increase of 80%, meaning it runs 5 times faster than the basic Pearson version. With the largest file size, we get the highest speed increase of 86.20689655%, meaning this version runs 7.25 times faster than the basic version.

In both optimized versions, we can see an increase in the CPU usage, with the difference that in the version with both optimizations, the total difference between the minimum and the maximum is smaller. The version with the first optimization has a difference of 99% - 35% = 64%. The second version with both optimizations has, on average, a higher CPU usage with 83% vs. 76.25%, but the difference between the file sizes is smaller. 97% - 64% = 33%. Meaning that the version with both optimizations scales a bit better than the one with only one.
We can also see a slight decrease in the memory space needed for both optimized sequential version of Pearson.

3.3.2 Parallelized Program

If we compare the speedups of Pearson running with the smallest input file, we can see that compared to the baseline version, we achieved a speedup of a minimum of 70% for 8 threads and up to 90% for 2, 4, and 6 threads. This makes the parallelized version with the smallest file size up to 10 times faster (speed-up for 2, 4, and 6 threads) than the original. If we look at the biggest file size as input, this trend continues. The speed-up achieved lies between 87.05% and 91.07% (for 2 and up to 16 threads). This makes the parallelized Pearson version up to 11.19620253 times faster than the original (using the speed-up for 16 threads).
If we compare the average speed-up in % per thread, we see that the 8-thread version has the lowest average speed-up with 69,41% and the 6-thread version has the highest with 88,88%.
If we take a look at the CPU usage, one can clearly see that the more threads we are using, the higher the % gets. Especially for higher workloads (e.g. the 1024MB sized input file), this scales really badly.
Lastly, looking at the maximum resident set size the versions got, we can see a relatively stable amount of kBytes with increasing values regarding the input file size.

4 Appendix

4.1 Blur baseline (image 1)

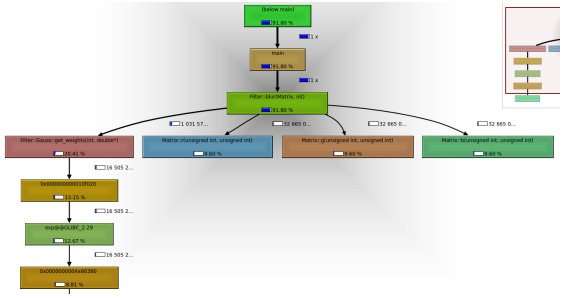


Figure 15: Blur baseline call-graph

Incl.	Self	Called	Function	Location
100.00	0.00	(0)	0x0000000000020290	ld-linux-x86-64.so.2
99.97	0.00	1	(below main)	blur
99.97	0.00	1	__libc_start_main@@GLIBC...	libc.so.6: libc-start.c
99.96	0.00	1	(below main)	libc.so.6: libc_start_call_main.h
99.96	0.00	1	main	blur: blur.cpp
91.80	42.40	1	Filter::blur(Matrix, int)	blur: filters.cpp
20.41	7.26	1 031 576	Filter::Gauss::get_weights(i...	blur: filters.cpp
13.15	0.48	16 505 216	0x000000000010f020	(unknown)
12.67	3.86	16 505 216	exp@@GLIBC_2.29	libm.so.6: w_exp_template.c
9.00	9.00	34 212 436	Matrix::b(unsigned int, unsi...	blur: matrix.cpp
9.00	9.00	34 212 436	Matrix::g(unsigned int, unsi...	blur: matrix.cpp
9.00	9.00	34 212 436	Matrix::r(unsigned int, unsi...	blur: matrix.cpp
8.81	0.48	16 505 216	0x0000000004a86380	(unknown)
8.33	8.33	16 505 216	__ieee754_exp_fma	libm.so.6: e_exp.c, math_confi...

Figure 16: Blur baseline table

```
33                  {
34    7.58              double w[Gauss::max_radius]{};
35    0.04              Gauss::get_weights(radius, w);
      10.20 515788 call(s) to 'Filter::Gauss::get_weights(int, double*)' (blur: filters.cpp)
36
```

Figure 17: Blur baseline percent of code lines

4.2 Blur after Optimization 1 (image 1)

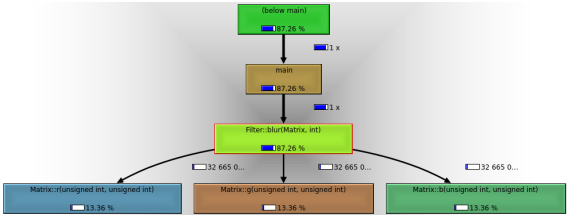


Figure 18: Blur after Optimization 1 call-graph

Incl.	Self	Called	Function	Location
100.00	0.00	(0)	0x0000000000020290	ld-linux-x86-64.so.2
99.95	0.00	1	(below main)	blur
99.95	0.00	1	__libc_start_main@@GLIBC...	libc.so.6: libc-start.c
99.94	0.00	1	(below main)	libc.so.6: libc_start_call_main.h
99.94	0.00	1	main	blur: blur.cpp
87.26	42.21	1	Filter::blur(Matrix, int)	blur: filters.cpp
13.99	13.99	34 212 436	Matrix::b(unsigned int, unsi...	blur: matrix.cpp
13.99	13.99	34 212 436	Matrix::g(unsigned int, unsi...	blur: matrix.cpp
13.99	13.99	34 212 436	Matrix::r(unsigned int, unsi...	blur: matrix.cpp
5.91	2.00	3	Matrix::Matrix(Matrix const&)	blur: matrix.cpp

Figure 19: Blur after Optimization 1 table

```
2.43                r += wc * dst.r(x2, y);
3.13  7645260 call(s) to 'Matrix::r(unsigned int, unsigned int)' (blur: matrix.cpp)
2.43                g += wc * dst.g(x2, y);
3.13  7645260 call(s) to 'Matrix::g(unsigned int, unsigned int)' (blur: matrix.cpp)
2.43                b += wc * dst.b(x2, y);
3.13  7645260 call(s) to 'Matrix::b(unsigned int, unsigned int)' (blur: matrix.cpp)
0.52                n += wc;
```

Figure 20: Blur after Optimization 1 percent of code lines in function blur()

```
        unsigned char& Matrix::b(unsigned x, unsigned y)
  4.66 {
  7.77     return B[y * x_size + x];
  1.55 }
```

Figure 21: Blur after Optimization 1 percent of code lines in function b()

4.3 Pearson baseline (128.data)

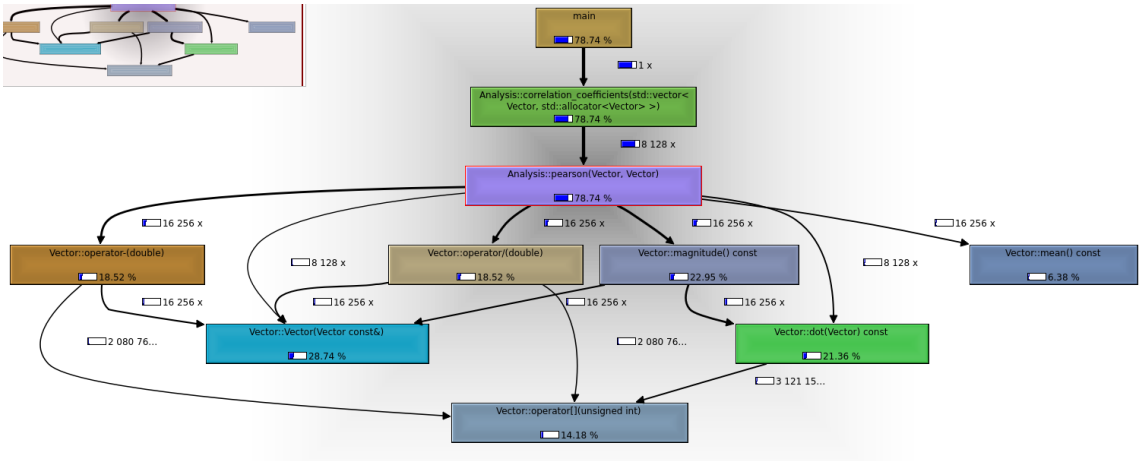


Figure 22: Pearson baseline call-graph

Incl.	Self	Called	Function	Location
100.00	0.00	(0)	0x0000000000020290	ld-linux-x86-64.so.2
99.58	0.00	1	(below main)	pearson
99.58	0.00	1	__libc_start_main@@GLIBC...	libc.so.6: libc-start.c
99.56	0.00	1	(below main)	libc.so.6: libc_start_call_main.h
99.56	0.00	1	main	pearson: pearson.cpp
87.61	0.08	1	Analysis::correlation_coeffi...	pearson: analysis.cpp
78.74	0.17	8 128	Analysis::pearson(Vector, V...	pearson: analysis.cpp
37.16	35.94	73 535	Vector::Vector(Vector const&)	pearson: vector.cpp
22.95	0.10	16 256	Vector::magnitude() const	pearson: vector.cpp
21.36	14.27	24 384	Vector::dot(Vector) const	pearson: vector.cpp
18.52	5.58	16 256	Vector::operator-(double)	pearson: vector.cpp
18.52	5.58	16 256	Vector::operator/(double)	pearson: vector.cpp
16.54	16.54	7 282 688	Vector::operator[](unsigne...	pearson: vector.cpp
6.38	6.38	16 256	Vector::mean() const	pearson: vector.cpp

Figure 23: Pearson baseline table

4.4 Pearson after Optimization 1 (128.data)

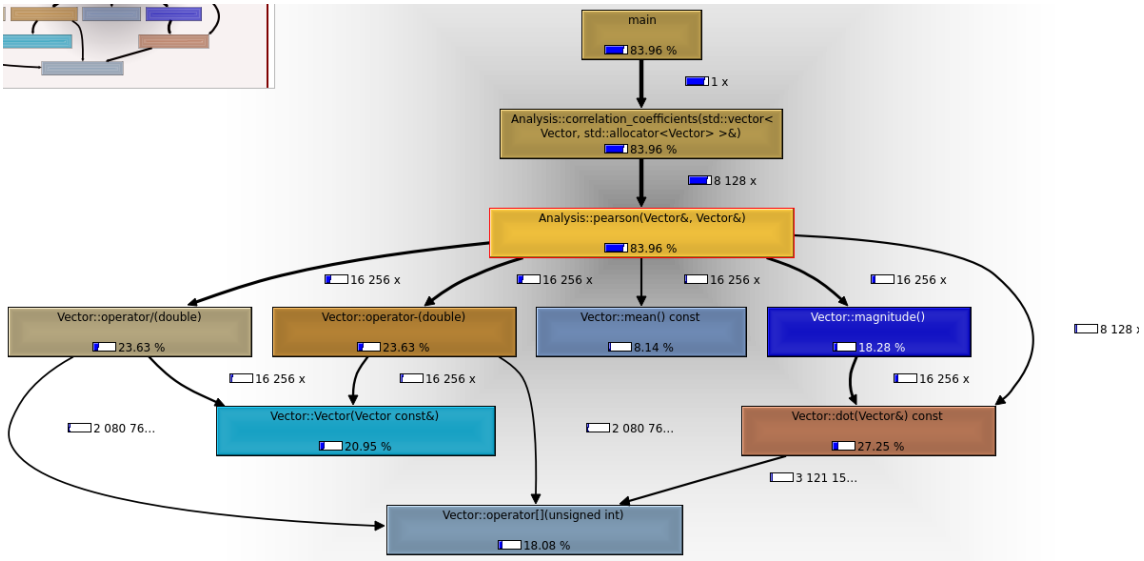


Figure 24: Pearson after optimization 1 call-graph

Incl.	Self	Called	Function	Location
100.00	0.00	(0)	0x0000000000020290	ld-linux-x86-64.so.2
99.46	0.00	1	(below main)	pearson
99.46	0.00	1	__libc_start_main@@GLIBC...	libc.so.6: libc-start.c
99.44	0.00	1	(below main)	libc.so.6: libc_start_call_main.h
99.44	0.00	1	main	pearson: pearson.cpp
84.29	0.07	1	Analysis::correlation_coeffi...	pearson: analysis.cpp
83.96	0.20	8 128	Analysis::pearson(Vector&, ...	pearson: analysis.cpp
27.25	18.20	24 384	Vector::dot(Vector&) const	pearson: vector.cpp
23.63	7.12	16 256	Vector::operator/(double)	pearson: vector.cpp
23.63	7.12	16 256	Vector::operator-(double)	pearson: vector.cpp
21.13	20.43	32 767	Vector::Vector(Vector const&)	pearson: vector.cpp
21.10	21.10	7 282 688	Vector::operator[](unsigne...	pearson: vector.cpp
18.28	0.07	16 256	Vector::magnitude()	pearson: vector.cpp
8.14	8.14	16 256	Vector::mean() const	pearson: vector.cpp
7.73	0.06	1	Dataset::write(std::vector<...	pearson: dataset.cpp
7.39	0.00	1	Dataset::read(std::__cxx11:...	pearson: dataset.cpp

Figure 25: Pearson after optimization 1 table