Performance Optimization of an Image Transformation Software

DV1567 - Performance Optimization,

Group 4

1st Jonathan Olsson
Blekinge institute of technology
Karlskrona, Sweden
Joos21@student.bth.se
19950201

2nd Lokesh Kola
Blekinge institute of technology
Karlskrona, Sweden
lokl20@student.bth.se
20000717

***Index Terms*—Performance Optimization, C++, Valgrind, Cachegrind, Kcachegrind, scalability, execution time, cache misses.**

I. Introduction

In this report an investigation of an image transformation application was performed. The reason behind the investigation is to find areas of the source code that could be optimized to reduce the execution time required by the software. In addition, it is also desired to increase the scale-ability of the software to make the performance more feasible for larger images. If optimization techniques could be applied to achieve these desired tasks, it might have a positive impact for the users that use the application, since their waiting time reduces. In addition, by reducing the amount of resources that is required for the execution might also have an impact on how much power that is required to run the hardware. By reducing the power consumption, might contribute towards a healthier climate. As follows, an explanation of the profiling tools will be presented, which metrics that will be used for the evaluation will be discussed, the system under test will be revealed and the procedure for collecting the data will also be available. Then a short introduction about the software and the examined functions will be explained. In the results section, the initial benchmark test will be presented with an analysis regarding the data will be touched upon. Based upon this information, then an iterative performance optimization process will be conducted that tries to fix the potential hot-spots and the results of the changes will be presented. In the discussion section, a discussion about potential limitations and findings will be discussed. Then the derived conclusions will be presented in the conclusions section.

II. Background

A. Application

The application that is going to be investigated is an image transformation application that allows the user to either blur or apply a threshold filter to ppm images. The software contains eight different classes, where each class is responsible for different tasks. Out of these eight classes, there is two files that is dealing with importing images and then outputting the finalized image. Two files is dealing with matrices and additional four files is responsible for the transformation algorithms.

B. Tools

In this subsection a brief explanation of the used tool will be available. If the reader is interested to seek out more information about the used tools, then the user could find out more information by reaching out the references for each given tool.

1) Valgrind - Is a profiling software that enables the user to dynamically analyze Linux executable. The tools could capture information such as memory by using memcheck and cache misses with the help of the cachegrind tool. In addition, the tool could also collect information about how many times functions has been called in the software and how long time it took to execute the instructions. Valgrind will be used to acquire a better insight about the characteristics of the image transformation software. This information will then be used to find areas in the source code that might be possible to optimize to increase the performance and scale-ability. The software will be profiled with the callgrind and cachegrind tools.
2) Cg_annottate - Is an analysis tool that could process and view information collected from valgrind's cachegrind tool. The tool shows the amount of cache misses and where they occurred in the source code. This tool will be used to examine areas in the source code that could be improved to remove unnecessarily clock cycles that has to be spent fetching data into cache.
3) Kcachegrind - Is a profiling tool that enables a user to graphically analyze the data which was collected from the valgrind callgrind profiling tool. This is done by creating performance graphs which present how the software is interconnected with its respective functions. For each connection, it is possible to see the amount of

calls/callees and how much time that was spent in each function.

4) Linux Time command - Is a Linux terminal command stopwatch that allows the user to take measurements on how long time a certain task takes to complete. This tool will be used to measure the execution time of the image transformation software.
5) Python - Is a high-level language that will be used to work with the collected data. The data will be worked and presented from Jupyter notebook, which is built upon a python kernel.
6) Pandas - Is a scientific library used to import and clean data. In addition, it is possible to view and make changes to the data. Pandas will be used to import the collected data from the measurements, clean the data and then analyze it.
7) Seaborn - Is an enriched graphical library that is built upon Matplotlib. Seaborn will be used to visually present the correlation between metrics with the help of heat-map graphs.
8) Numpy - Is a mathematical library for python that enables a developer to perform arithmetic. Pandas will be used to calculate various statistical characteristics such as correlation tests and descriptive statistics.
9) Matplotlib - Is a graphical library that makes it possible to plot collected data in common graphs, such as line graph, histogram and pie charts.

C. Input files

To conduct an analysis of the performance of the image transformation software, there was a set of pictures included with the source code. The set of images had different motives where the color richness varied among the pictures. In addition, the pictures had different sizes. As follows, a short explanation of each image contained in the set will be revealed.

1) Img1 - This image has the size of 1Mb and the picture shows a girl holding a child. The image contains mostly colors from the brown palette, but some red is also present.
2) Img2 - This image has the size of 3Mb and presents a kid with a set of balloons. The picture contains a variety of colors.
3) Img3 - This image has the size of 5Mb and pictures a happy cat. The color of the cat is mostly containing colors from the gray-scale.
4) Img4 - This image has the size of 25,7Mb and pictures a set of neon color splashes. The image is colorfulness.

D. Metrics

In order to make an analysis of the provided software a set of metrics that could be used to evaluate the performance of the software will be selected. To do this, the following metrics has been chosen to enable a comparison between the versions:

1) Number of calls - The amount of function calls a function has recieved and called during runtime will be evaluated with the valgrind callgrind tool. The values will be captured with the help of Kcachegrind that is able to present the numbers both in a performance map and a list. This metric will be captured on an absolute scale.
2) Self - The percentage of the total execution time that was spend for a given function within the software. The metric will be collected with the help of valgrind cachegrind and then extracted with the kcachegrind tool. This metric will be captured on a ratio scale.
3) execution time - The amount of seconds that was required to execute the software with an arbitrarily input. The metric will be captured with the help of the Linux terminal command called time. The user and real metrics will be extracted and used for the comparison. This metric will be captured on a ratio scale.
4) size of image - The size of the input image that gets parsed to the system. This metric will be captured on a ratio scale.
5) Cache misses - Investigates if data could be directly read from the memory location in the cache. If the information is not stored in the cache a miss was found. This will cause the cache to allocate more space and copy the data from main memory. The metric is a set of metrics containing of I1, LLi ,D1, LLd, LL cache misses. This metric will be collected on an absolute scale. [2]

E. System under test

The system under test that was used for performing the optimization tests was run in a virtualization environment with the following configuration: The CPU that was used is an AMD Ryzen 7 5800U with Radeon Graphics. The CPU was run by using 64-bit registers with a byte order that used Little Endian. The CPU was configured to have 4 cores, where each core had one thread. The CPU was run with a clock speed of 1896Mhz and a bogoMIPS score of 3792.87. The size of L1d and L1i had a size of 128Kib, L2 a size of 2MiB, and L3 had a size of 65MiB. Hyper-Threading was disabled.

The amount of memory allocated for the system is 4Gb of RAM and 128KiB of memory for the BIOS.

The attached network card was an 82540EM Gigabit Ethernet Controller. However, the network card was not used for the performance tests.

The allocated size for the hard drive was 16Gb, with an I/O size of 512/512 bytes. Raid configuration was not used, since the system only used one hard drive.

The operating system that was used for this experiment was Ubuntu, Version 22.04.1 LTS.

F. Collection of data

Initially the software will be profiled with the profiling tool valgrind by using the tools cachegrind and callgrind. The profiling will capture information of the software when it was compiled with the makefile that came attached with the software. The profiling tool will capture information from four different images, which is described in the input section. The collected data will be extracted with the help of kcachegrind

and cg_annotate tool and then stored in an excel sheet. Once the data is saved, it will be cleaned so it could be used for the initial benchmark analysis.

The amount of seconds that is required to execute the application by running both transformation algorithms will be captured with the Linux time command. The software will be run for all the images contained in the set and the radius will be set to 15 for the blur algorithm. The software will be run five times for each image and the data will be stored in an excel sheet. Once the collection is complete, the average of the five runs will be used to hopefully get a more accurate measurement. The execution will be done without having any other application running by the user to limit the risk of having other applications competing for the same resources.

Once the collection of data is done, the software will initially get analyzed by the performance graph to seek out potential hot-spots in the source code. This might point to areas of the code that might be a bottleneck as the input increases. In addition, if it is possible to find hot-spots in the code might indicate towards where the most performance optimization techniques would yield the biggest boost. The metrics collected from the callgrind will be analyzed by using descriptive statistics and visually present the information in graphs.

The collected data from cachegrind will be analyzed in cg_annotate. First the information regarding cache misses will be analyzed to seek out areas in the code that produces cache misses and the amount of them. Hopefully by finding those spots, countermeasures could be taken into consideration to reduce the amount of clock cycles that is required to fetch data in cache. Second, a correlation test between the collected metrics will be conducted to seek out the relationship between the metrics. Once all the various aspects has been taken into consideration, the next step is to improve the found areas of the code that could yield in a performance increase. The tests will be done iterative by applying one performance optimization technique at once and then measure the performance changes. This step will be re-applied until the found areas has been fixed.

The collected data will be available at Github [1], in addition the Jupyter notebook that was used for the analysis will also be uploaded to the same repository.

III. Optimization Techniques

A. Initial Benchmark/Baseline Performance

1) Blur software: As could be seen in Fig.1, the total amount of calls that is required to execute the blur software is steadily increasing as the size of the file is growing larger. In addition, it seems that the amount of calls that is required for an arbitrarily image size increases linearly. This might indicates towards that performance optimizations might have a larger impact on the execution time and the number of calls that is required as the image grows larger.

[1]https://github.com/JongeDev/OptimizationProjectAssignment

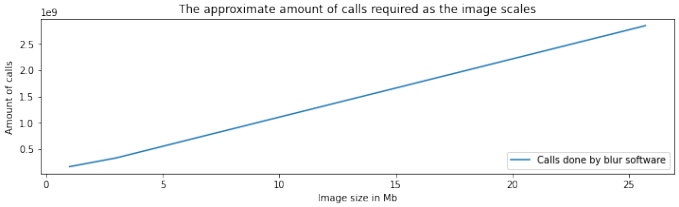


Fig. 1. Shows the approximated amount of calls with respect to image size.

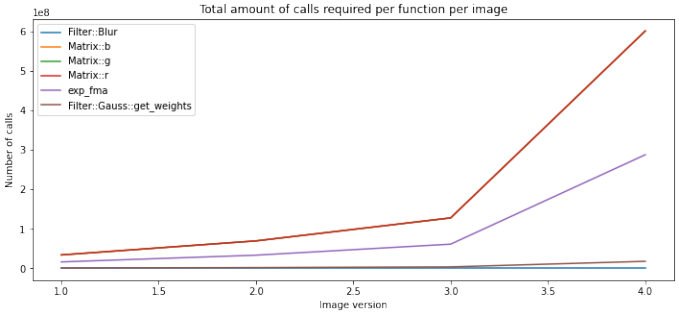


Fig. 2. Shows the amount of calls done by a function across different images.

As could be seen in Fig.2, is seems that the matrix computation for b, g and r is the most time consuming process. In addition, the function exp_fma also seems to be increasing its required functions calls at a 45 degree angle. These two functions seem to be have the largest impact on the total amount of calls required by the software. Each of the matrix action computations consumes approximately 9% of the total amount of execution time. Due to these reasons, it would be interesting to further investigate these functions to see if it is possible to optimize them. The Filter::Gauss::get_weights does not seem to increase that heavy with respect to the number of calls as the size of the image increases. However, the required time that was used by the functions was relatively high at 8,32 and 7,24 respectively. This might indicate towards that much work is done internally in the function. There might be a possibility to change how the computation is performed to reduce the amount of time that is required.

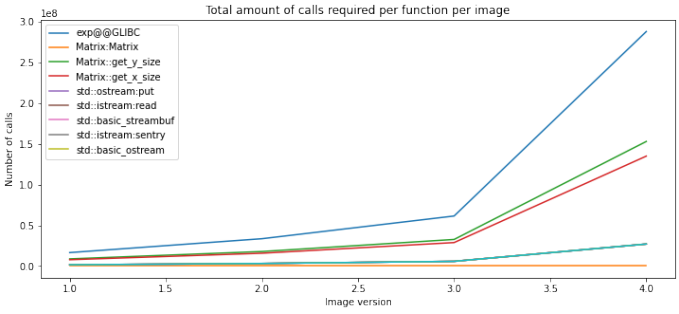


Fig. 3. Shows the amount of calls done by a function across different images.

The exp@@GLIBC function seems to have a steady increase in the amount of calls as the size of the image increases. By examine the amount of time that was required during

execution, it was measured to be approximately 3,86% across the different images. As the size of the image grows larger, this function would hypothetically have a bigger impact and might cause a future hot-spot. By either changing algorithm function to use a custom made exponential formula or by using another library could hopefully improve the performance. It might also be an idea to see if it is possible to reduce the amount of time the function gets called.

By examine the amount of time the software called the get_size of the x and y axis that is associated with the matrix, it is shown that these calls also increases quite rapidly. By investigating the required computation time for the functions was moderate at approximately 1,29% for the Matrix:get_x_size and 1,02 for the Matrix:get_y_size. By investigating the source code closer, it might be possible to see if there is a possibility of taking advantage of cache locality to reduce the amount of calls that is required. This change might provide a slight performance increase and also reduces the gradient of the curve, which would be make the computation more feasible as the size of the image increases.

The system calls regarding reading and writing does not seem to increase that much in comparison to the other functions, which could be seen in the tail colored line in Fig.3. The time required for performing the I/O computations ranges from [0,5%-0,84%]. Since there is five of these calls, the total amount of time in percentage is considered to be relatively moderate. It might be worth considering investigating if various optimization techniques could be used to increase the performance.

By comparing the different graphs with respect to the amount of calls and required computation time that was used, it seems that the matrix actions and exponential functions would provide the largest performance boost if there is room for improvement. After these functions, the get matrix size and I/O functions looks attractive. The rest of the functions that has not been mentioned so far, does not seem to be that interesting in regards to performance optimization at this stage. One of reason behind this is that the increase of calls is rather flat for the samples shown so far. In addition, the functions did not require that much time to complete their computation tasks.

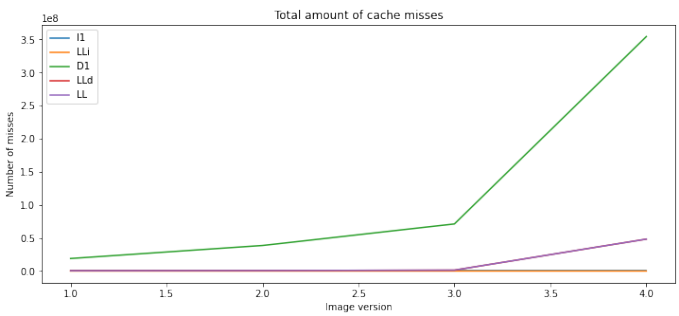


Fig. 4. Shows the amount of cache call misses for each image.

Fig.4 is presenting the total amount of cache misses that occurred when running the blur software. It seems that LLi, I1 read and write misses are relatively low in comparison to LL, LLd and D1 misses. In addition, the amount of misses seems to be fairly stable as the size of the image increases. However, LL and LLi seems to have a stable small increase before the software was run with the fourth image parsed as input. At the fourth image, it is possible to an gradient increase of the amount of cache misses.

However, this gradient increase might be due to that image three had a size of 5,5Mb compared to the fourth image which had an size of 25,7Mb. The D1 misses, seems to be the one that impacts the software the most, since the gradient is larger in comparison to the other metrics. It is expected that this trend will continue as the size of image increases. Due to this reason, it might be of interest to investigate further if there is any performance optimization that could reduce the amount of data cache misses occurring at layer one.

According to [8], a Layer one cache miss will approximately cost 10 CPU cycles, a missed branch around 10-30 CPU cycles and a LL miss could cost up to 200 instructions. Based upon this information, it might also be interesting to investigate if there is any obvious part of the source code that had such a high impact on the amount of LL misses between image three and four. By reducing the amount of cache misses will free up more CPU cycles that could be used for other computations. This optimization will hopefully reduce the execution time.

By investigating the information that was produced with valgrind's cachegrind tool in cg_annotate it was found out that multiple read and write operations was found in the matrix.cpp source code file. The calls was caused from multiple loops that called the functions get_x_size and get_y_size. The calls was initially measured at 35M disk reads and 17.53M disk writes for get_x_size and 30M disk reads and 15.47M disk writes for the get_y_size function for image one. The amount of read and writes was steadily increasing as the size of the image also increases.

The initial idea is to seek out if these loops could take advantages of cashing the values, instead of asking for the size at each iteration. in PPM.cpp's function operator it was found out that the standard library "endl" function was used a few times. When the software is calling "endl", the software ends the line and then clears the output buffer. This causes extra overhead to be performed and it might be worth investigating if "endl" could be replaced with "$\backslash n$", to skip unnecessarily buffer clears. In addition, it seems like the writer function is only printing one pixel color at the time. The reader fill function might also be worth to investigate to seek out if it is possible to change reading techniques to something that increase the performance, such as reading information in larger chunks.

Based upon information provided from the valgrind cachegrind and callgrind, it seems that the Gauss::get_weights function might also be of interest since this function was generating multiple calls, reads, writes and misses. The majority of calls was generated by the exponential function, by investigating if there is a better algorithm for computing the exponential might yield in a better performance. In addition,

the computation performed inside the for-loop on line 16, was also time consuming. Initially, it seems that it might be worth investigating if pre-computing $\frac{max_x}{n_i}$ outside the loop to avoid the costly division operation multiple times.

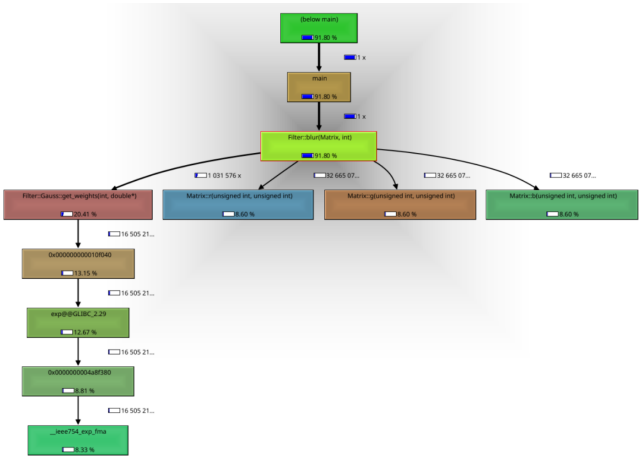


Fig. 5. Performance graph for the blur software.

By examine the performance graph generated by valgrind which is presented in Fig.5 it seems that the matrix actions is performed sequential. In addition, multiple of the calls was generated by the matrix function r, g and b. First, it is expected that the matrix actions is going to be a resource consuming task which was shown by the numbers in the analysis. However, it might be possible to exchange the algorithm or optimize the current one. By doing so, it might be possible to reduce the amount of calls done which might also reduce the execution time. In addition, by introducing parallelism within the software by for an example introducing threads could take advantages of multiple CPU cores, to better utilize the available resources.

In Fig.6 it is also possible to see similar data regarding the Gauss::get_weights that inline with what was shown with the cachegrind tool.

The execution time for the blur software started at approximate 0,9 seconds and increases steadily as the size of the image also increases. For the largest image that was parsed as an input, the execution time took approximately 16 seconds. It also seems that the time used by average real and user seems to increase in line with each other as the size of the image increases.

By investigating the correlation between the execution time and total amount of calls required, it seems that there is a strong relationship between the metrics. This indicate towards that the number of calls has a large impact on the amount of time that is required to execute the software. By reducing the amount of calls performed in the software, suggests that it also will reduce the total amount of time required. It also seems that the size of the image has an impact on both the execution time and amount of calls required with a correlation strength of 0.86. This could be shown in Fig.8.

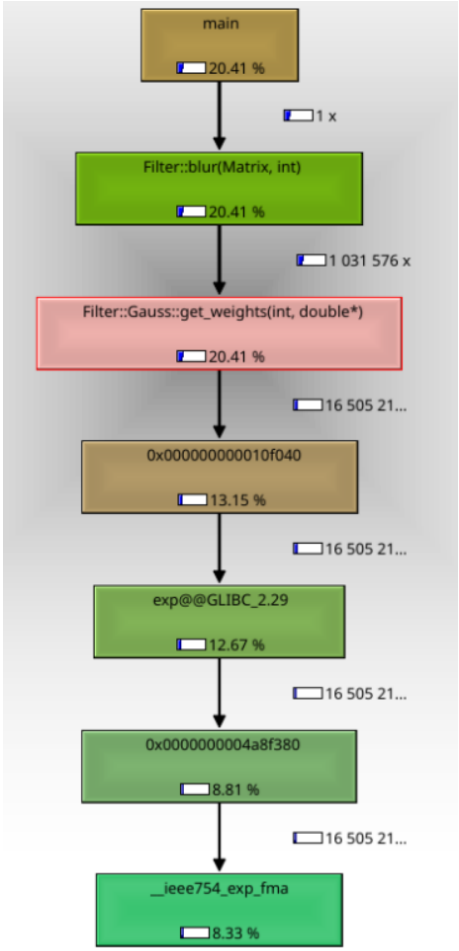


Fig. 6. Performance graph for the blur software.

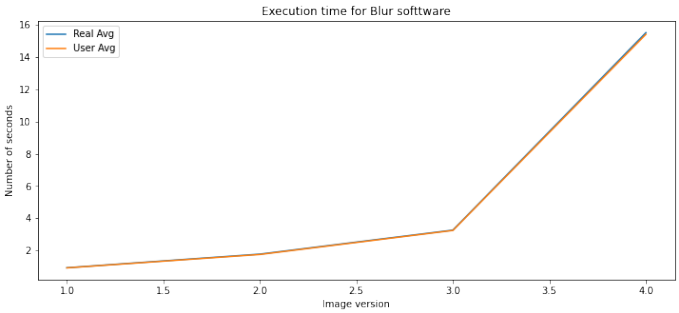


Fig. 7. Execution time for the blur application.

In Fig.9, the functions of the blur software has been ranked after the total amount of time in percentage that the software spent in each function. The breakdown presents numbers when the blur algorithm was used with the Img1 as parsed input. By taking the execution time and the amount of calls into consideration it is possible to find interesting areas in the source code where an analysis of the source code could be done to seek out if performance optimization could be applied.

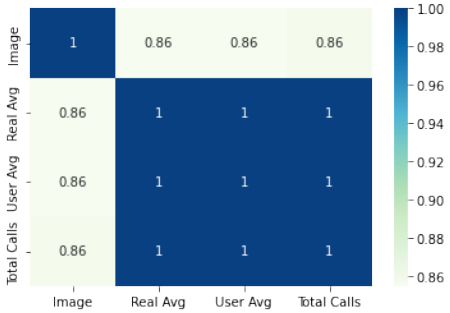


Fig. 8. Correlation matrix for the blur application.

Incl.	Self	Called	Function	Location
91.80	42.40	1	Filter::blur(Matrix, int)	blur: filters.cpp
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
8.33	8.33	16 505 216	__ieee754_exp_fma	libm.so.6: e_exp.c, math_config.h
20.41	7.26	1 031 576	Filter::Gauss::get_weights(i...	blur: filters.cpp
12.67	3.86	16 505 216	exp@@GLIBC_2.29	libm.so.6: w_exp_template.c
3.80	1.29	3	Matrix::Matrix(Matrix const...	blur: matrix.cpp
1.03	1.03	8 769 750	Matrix::get_y_size() const	blur: matrix.cpp
0.91	0.91	7 738 176	Matrix::get_x_size() const	blur: matrix.cpp
1.45	0.88	1 547 367	std::ostream::put(char)	libstdc++.so.6.0.30
2.65	0.84	1 547 364	std::istream::read(char*, lo...	libstdc++.so.6.0.30
1.13	0.72	1 547 364	std::basic_streambuf<char, ...	libstdc++.so.6.0.30
0.63	0.63	1 547 368	std::istream::sentry::sentry(...	libstdc++.so.6.0.30
0.52	0.52	1 547 378	std::ostream::sentry::sentry...	libstdc++.so.6.0.30
1.99	0.50	1 547 364	std::basic_ostream<char, st...	libstdc++.so.6.0.30

Fig. 9. Shows performance graph breakdown for the blur application.

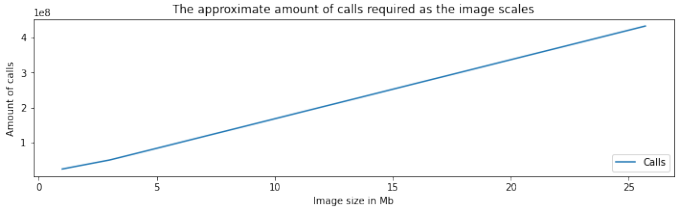


Fig. 10. Shows the approximated amount of calls with respect to image size.

2) Threshold software: In Fig.10, it is possible to see the approximated total amount of calls required during execution of the threshold software as the size of the image increases. As could be seen, the amount of calls is steadily increasing as the size of the image increases. By limiting the amount of calls that is required to execute the software might reduce the execution time. In addition, it might also contribute towards a better scale-ability of the software could be achieved.

It seems that the amount of calls Matrix::Matrix is stable at 3 calls per execution independent on the size of the image size, which could be seen in Fig.11. However, the number of calls done related to the matrix b, g and r seems to increase by a large amount as the the size of the image increases. This indicates towards that the matrix actions is resource demanding.

It seems that the amount of calls performed by the filter threshold function to be stable independent on the size of the

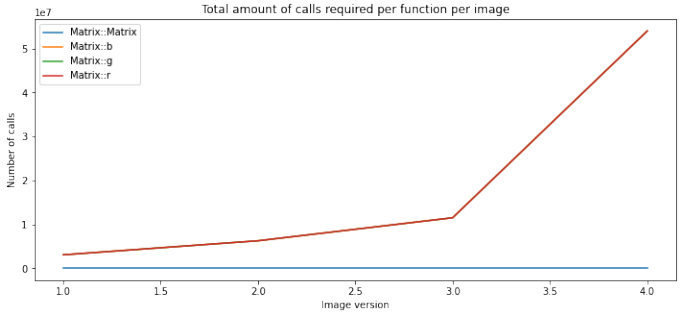


Fig. 11. Shows the amount of calls done by a function across different images.

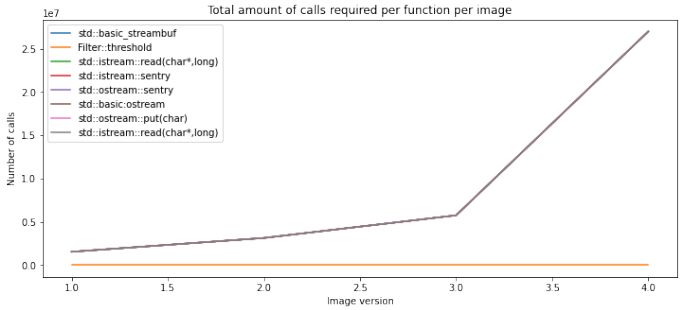


Fig. 12. Shows the amount of calls done by a function across different images.

image, which is presented in Fig.12. In addition, it seems that there is a correlation between the calls related to input and output calls, since the metrics overlap each other. As the size of the image increases, the amount of calls performed required by I/O calls also increases. It might be worth investigating if it is possible to reduce the amount of I/O calls to speed up the read and write process since it is impacting multiple areas.

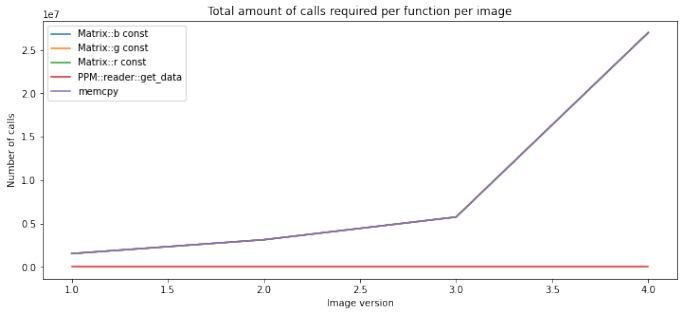


Fig. 13. Shows the amount of calls done by a function across different images.

By taking a look at Fig.13, it is possible to see that the matrix b, g, r const and memcpy functions gets called multiple times and the amount of calls increases steadily as the size of the image increases. This makes it an interesting point to further investigate if performance optimization techniques could be applied. The amount of calls PPM's reader get_data function seems to be stable and independent on the size of the image. Due to this reason, the focus will initially be targeted to other areas of the code.

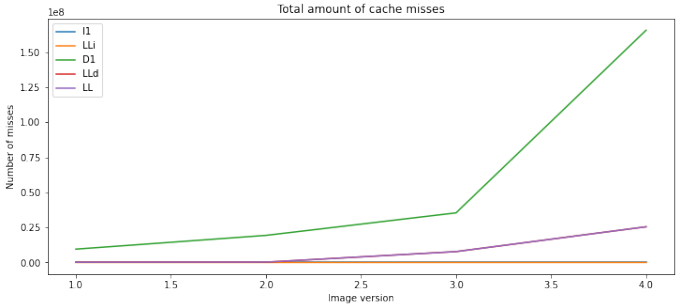


Fig. 14. Shows the amount of cache misses found in the threshold software.

By examine the amount of cache misses produced by the software across different image sizes, presented in Fig.14, it is possible to see that the amount of D1 cache misses has the largest gradient for larger image sizes. In addition, LL cache misses seems to also have slight positive gradient which seems to have a larger impact after the second image. LLi and LLd cache misses seems to be stable independent on the image size.

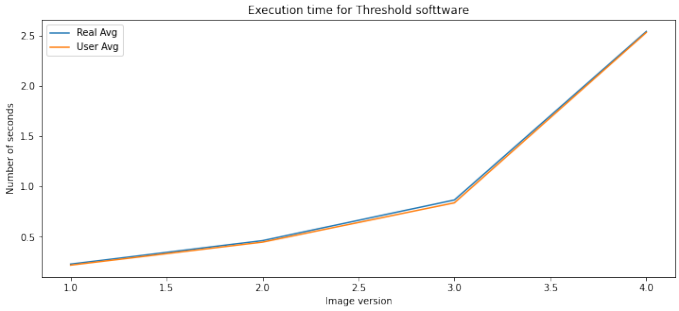


Fig. 15. Shows the execution time for the threshold application.

The execution time required for the threshold software to process images is increasing as the size of the image scales. It is possible to see in Fig.15 that the average amount of time for real and user overlap with each other. The results indicate towards that, as the size of the image increases, the amount of data that has to be processes by the filter also increases. Since the amount of data increases, the amount of calls also increases, which has an impact on the final execution time. As could be seen for the largest image four, it took approximate 2.6 seconds to execute the software

By investigating the correlation between the amount of calls and execution time, it is possible to see that there is a strong positive relationship between the total amount of time required for execution and the amount of calls. This means that each extra call takes some extra time to compute and by reducing the amount of calls could also reduce the amount of time that is required for execution. In addition, it seems that the size of the image also has a strong positive relationship at 0.86 in relation to the execution time and the amount of calls. This exploration strengthens the hypothesis presented earlier that as the size of the image increases, the more data has to be processed.

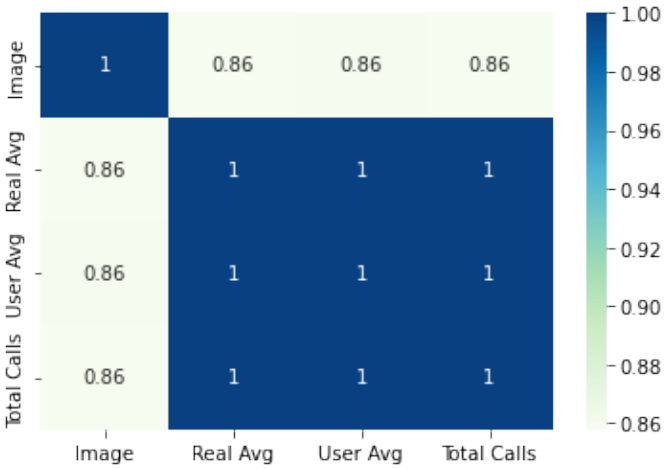


Fig. 16. Shows the correlation between features in the threshold application

This results in that the amount of calls increases which takes more time to compute, resulting in that the execution time also increases. However, to which degree a function call is expensive has to be explored. This information could be seen in Fig.16. In Fig.17, the functions of the threshold software has

Incl.	Self	Called	Function
33.46	11.35	3	Matrix::Matrix(Matrix const&)
12.75	7.77	1 547 367	std::ostream::put(char)
23.30	7.37	1 547 364	std::istream::read(char*, long)
7.17	7.17	3 094 728	Matrix::b(unsigned int, unsigned int)
7.17	7.17	3 094 728	Matrix::g(unsigned int, unsigned int)
7.17	7.17	3 094 728	Matrix::r(unsigned int, unsigned int)
9.96	6.37	1 547 364	std::basic_streambuf<char, std::char_traits<char...
27.78	5.88	1	Filter::threshold(Matrix)
5.58	5.58	1 547 368	std::istream::sentry::sentry(std::istream&, bool)
4.58	4.58	1 547 378	std::ostream::sentry::sentry(std::ostream&)
17.53	4.38	1 547 364	std::basic_ostream<char, std::char_traits<char> >...
3.78	3.78	1 547 364	Matrix::b(unsigned int, unsigned int) const
3.78	3.78	1 547 364	Matrix::g(unsigned int, unsigned int) const
3.78	3.78	1 547 364	Matrix::r(unsigned int, unsigned int) const
28.35	3.65	1	PPM::Reader::get_data(unsigned int, unsigned int)
3.36	3.36	1 548 334	__memcpy_avx_unaligned_erms

Fig. 17. Shows performance graph breakdown for the threshold application.

been ranked after the total amount of time in percentage that the software spent in each function. The breakdown presents numbers when the threshold algorithm was used with the Img1 as parsed input. By taking the execution time and the amount of calls into consideration it is possible to find interesting areas in the source code where an analysis of the source code could be done to seek out if performance optimization could be applied.

B. Iteration one

By investigating the information provided from the profiler in Fig.3, it was possible to see that multiple calls was per-

formed to get the x and y size of the matrix. By investigating where the calls was located from, it was possible to see that the method blur had caused the calls. The calls was present in two different nested for-loops where for each iteration, the size the x or y-axis was collected. Since the matrix size is independent on what was happening inside the loop, it is possible to cache the value outside of the loop. By doing so, the amount of calls gets reduced which also reduces the overhead. [9] In addition, it was also shown that the creation of the array called w was independent of the nested loops. This made it possible to also apply the same technique to this variable. In Fig.18, it

```
23  Matrix blur(Matrix m, const int radius)
24  {
25      Matrix scratch { PPM::max_dimension };
26      auto dst { m };
27      float xSize = dst.get_x_size(); //change for itteration one.
28      float ySize = dst.get_y_size(); //change for itteration one.
29      double w[Gauss::max_radius] {}; //change for itteration one.
30
31      for (auto x { 0 }; x < xSize; x++) {
```

Fig. 18. Shows the changes for iteration one.

is possible to see the source code changes for iteration one. For image four it took 153006002 calls to get y size and 135006004 calls to get x size. All the calls to get y size took 1.02% of the execution time and get x size took 0.9%. By applying the cache technique, the calls got reduced to two calls and an insignificant impact on the execution time.

C. Iteration two

By investigating the information produced by valgrinds tool cachegrind it was possible to see that over two million cache misses was produced in the blur function. The cache misses was produced when the software tried to load the RGB colors from each pixel in the picture. This is shown in Fig.19. According to Valgrinds documentation [8] A cache miss often

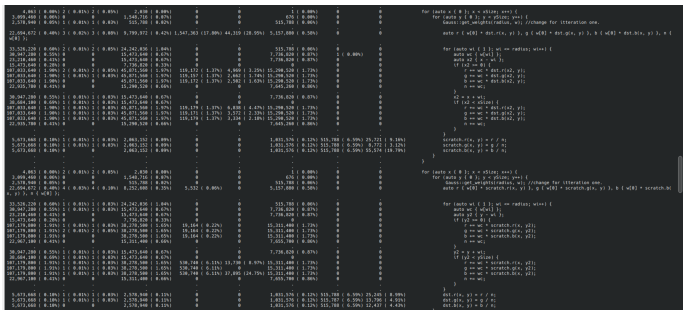


Fig. 19. Shows the cache misses for iteration two prior to the changes.

occurs when the CPU needs a certain data which is unavailable in the cache block memory. A cache miss results in that the CPU might have to fetch the asked data into cache from memory and doing so could cause 10-200 extra clock cycles depending on what type of cache miss that was produced. A cache miss is usually occurring when the CPU tries to get data from another cache block. If the software generates multiple cache misses then there might be performance benefits of reducing these so that more clock cycles could be spent on other tasks. For this case, it was found out that by traversing the matrix in the Y-axis caused the cache misses. By changing the traversing order by using a loop interchange technique it was possible to reduce the amount of cache misses. [4] As

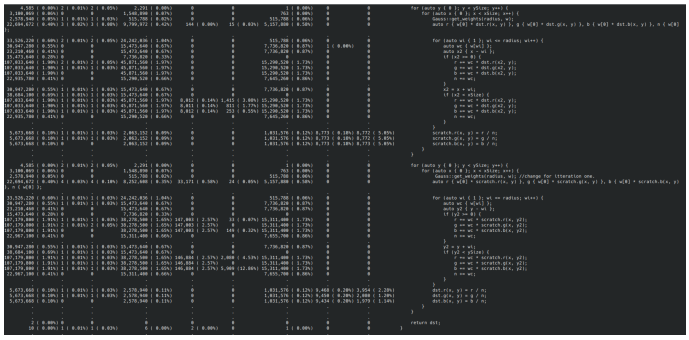


Fig. 20. Shows the cache changes for iteration two.

could be seen in Fig.20, the amount of D1mr cache misses got reduced to approximate 900K from 2.1M and the amount of DLmr got reduced from 134k down to 27k. The amount of D1mw got reduced from 7,73M down to 4,7M and DLmw got reduced from 198K to 91K. The total difference between iteration one and two is shown in Fig.21 where the terminal above is after applying the technique. The loop interchange

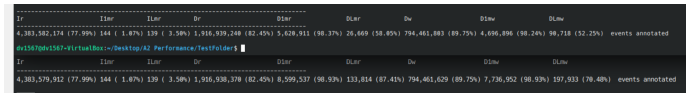


Fig. 21. Shows the difference in cache misses in iteration two.

technique got applied to both of the nested for-loops that was present in the blur function in filters.cpp. In Fig.22 a showcase on how the technique was applied for the first nested for-loop, where the traversing of the matrix instead moves by each row. The reduction of cache misses, enables the CPU to spend extra

```
29
30      for (auto y { 0 }; y < ySize; y++) {
31          for (auto x { 0 }; x < xSize; x++) {
32              Gauss::get_weights(radius, w);
```

Fig. 22. Shows the source code changes for iteration two.

clock cycles on other tasks.

D. Iteration three

In Fig.9, that presented data from the benchmark tests, it was possible to see that the Filger:gauss:get_weights function got called just above 1M times, which caused the software to spend approximate 7.26% of the runtime there. Unfortunately, inside the function, the expensive mathematical exponential call was performed approximate 33M times by two different functions exp_fma and exp@@glibc, which occupied the system by an additional 8,33% and 3.86%. By an investigation in the valgrind callgrind profiler tool it was possible to see that the calls have also been caused from the blur function inside

filers.cpp. The function call was placed inside the nested for-loop twice, which is why there was such a high amount of calls. By investigating the source code, it was possible to see that the function call get_weights was placed at an invariant place, which caused the software to make redundant computations since the work done after each iteration remained the same. [6]

By moving out the the function call outside of the inner for-loop it was possible to see that the amount of calls of exp_fma got reduced to 24416 calls and the self% was measured to 0.02%. The Filter:gauss:get_weights got reduced to 1526 calls and the self was measured to 0.02. The exp@@glibc got reduced to 24416 calls and a self percentage of 0.01. The results is presented in Fig.23.

0.05	0.03	2 374	_dl_lookup_symbol_x	ld-linux-x86-64.so.2: dl-lookup.c
0.02	0.02	24 416	__ieee754_exp_fma	libm.so.6: e_exp.c, math_config.h
0.05	0.02	1 526	Filter::Gauss::get_weights(i...	blur: filters.cpp
0.02	0.02	2 374	do_lookup_x	ld-linux-x86-64.so.2: dl-lookup.c, dl-protecte...
0.05	0.01	7	_dl_relocate_object	ld-linux-x86-64.so.2: dl-reloc.c, dl-machine.h,...
0.03	0.01	24 416	exp@@GLIBC_2.29	libm.so.6: w_exp_template.c

Fig. 23. Shows the impact of calls for iteration three.

In addition to these changes it was also possible to see that the cache locality was improved in the function, since the amount of cache misses got reduced by approximate 100k. This is shown in Fig.24, where iteration two is presented in the back terminal and iteration three is presented in the front.

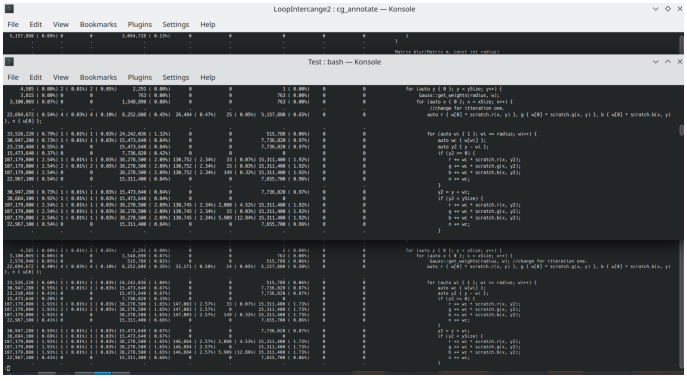


Fig. 24. Shows the reduction of cache misses for iteration three.

Fig.25 presents where one of the changes for iteration three took place. This technique was also applied for the second nested for-loop.

```
31    for (auto y { 0 }; y < ySize; y++) {
32        Gauss::get_weights(radius, w);
33        for (auto x { 0 }; x < xSize; x++) {
```

Fig. 25. Shows source code change for iteration three.

E. Iteration four

As shown in the performance graph breakdown in Fig.9, it is possible to see that much time was spent by calling to get the color of the pixels in the matrix. Each of the pixel colors have been called for 34212436 times. Unfortunately, no feasible change of the blur algorithm was found that could reduce the amount of calls that occurred in the function. However, according to information published at Microsoft [7], it is possible to declare functions that is used frequently with the inline function. By doing so, the developer asks the compiler to optimize the function call by removing unnecessarily overhead. Microsoft think it is appropriate to use inline functions when accessing private data from small objects when the function is simply returning information about the object. As in this case, the matrix class simply return the color of a pixel at a given coordinate. Unfortunately, this inline function is just a request so it may be the case that the compiler does not want to accept the request. To test this out, the inline command got added to each of the Matrix::R,G,B functions and then let the application run for five times with and without the change. By taking the average between these five executions it was shown that by including the inline command, the execution time got reduces by 0.5 seconds. However, it is worth mentioning that the time measurement had some overlap with their data points so it is not known to what degree the probabilistic nature had to the measured result. The inline change will persist until iteration ten, when it gets removed for a more convenient compiler optimization.

F. Iteration five

By further investigating the software with the valgrind cachegrind tool and then analysing the results in cg_annotate, it was found out that the constant functions that returns the RBG color of a matrix pixel, produced for almost 1,55M cache misses for each color. The measurement is shown in Fig.26. By looking at the performance graph produced by valgrinds

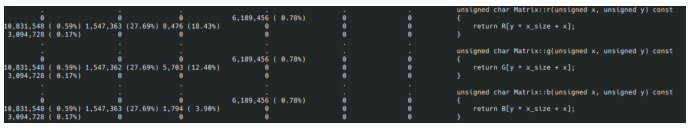


Fig. 26. Shows cache information for iteration five.

cachegrind, it was shown that the calls originated from the matrix class in the Matrix::Matrix function. By applying the same loop interchange technique that was used in iteration two by changing how the nested for-loop iterates through the matrix, it was possible to reduce the amount of cache misses to approximate 24k for each color. This data is shown from the measurement available in Fig.27. The source code change

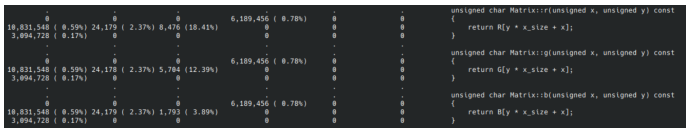


Fig. 27. Shows cache information for iteration five after applying loop interchange.

is presented in Fig.28, within the nested for-loop.

```
Matrix::Matrix(const Matrix& other)
    : R { new unsigned char[other.x_size * other.y_size] }
    , G { new unsigned char[other.x_size * other.y_size] }
    , B { new unsigned char[other.x_size * other.y_size] }
    , x_size { other.x_size }
    , y_size { other.y_size }
    , color_max { other.color_max }
{
    for (auto y { 0 }; y < y_size; y++) {
        for (auto x{ 0 }; x < x_size; x++) {
            auto &r_val { r(x, y) }, &g_val { g(x, y) }, &b_val { b(x, y) };
            auto other_r_val { other.r(x, y) }, other_g_val { other.g(x, y) }, other_b_val { other.b(x, y) };

            r_val = other_r_val;
            g_val = other_g_val;
            b_val = other_b_val;
        }
    }
}
```

Fig. 28. Shows source code change for iteration five after applying loop interchange.

In addition to this change, a minor change took place in the input buffer of the ppm class. According to Kurt Guntheroth in his published book [9], he suggests that by increasing the size of the input buffer, allows the software to read in data from a binary file in larger chunks. By applying this technique, the author found a performance increase of approximate 5% in his application. By applying his technique for this application, the changes could be found in Fig.29

```
void Reader::fill(std::string filename)
{
    std::ifstream f {};

    f.open(filename);

    if (!f) {
        stream.setstate(std::ios::failbit);
        return;
    }
    char buf[8192];
    f.rdbuf()->pubsetbuf(buf,sizeof(buf));
    std::copy(
        std::istreambuf_iterator<char> { f.rdbuf() },
        std::istreambuf_iterator<char> {},
        std::ostreambuf_iterator<char> { stream });
```

Fig. 29. Shows source code change for iteration five after increasing the size of the input buffer.

G. Iteration six

For this iteration a minor change was done in the output buffer which caused an improvement of approximate 1 second when the tests were run five times. The first change was to replace the ”endl” statements that occurred three times in the matrix writer function with ”$\backslash n$”. The reason to apply this technique is that when ”endl” is called the software also clears the output buffer, which takes additional time [9]. For this application where the commands only was performed three times the performance benefits is probably insignificant, but was applied for best practices. Speaking of how the function is printing data, it was found out that the ”<<” operand makes additional controls and then it is using the put function. This creates additional overhead compared to just directly run the put command. According to, [1] the ”<<” is not well suited for working with binary files and as could be seen in the source code of ppm, this operand is occurring three times per iteration. By instead directly using the put command to bypass the extra overhead, there was an improvement of performance by approximate 1 second when the test was run five times. The source code change is available in Fig.30.

```
while (it_R < R + size && it_G < G + size && it_B < B + size) {
    //f << *it_R++
    //  << *it_G++
    //  << *it_B++;
    f.put(*it_R++);
    f.put(*it_G++);
    f.put(*it_B++);
}
```

Fig. 30. Shows source code change for iteration size after changing to put operand.

H. Iteration seven

By investigating the new breakdown of calls and self percentage, it is possible to see that approximate 1547364 calls has been done from three different places that was related to the input stream for image one. This caused a total amount of approximate 4,5M calls and a total amount of self 3.5. In addition, 1,5M copy calls was used and a similar amount of gcount() calls was done. This took 0.62 and 0.11 respectively. This information is available in Fig.31.

Incl.	Self	Called	Function	Location
87.48	43.21	1	Filter::blur(Matrix, int)	blur: filters.cpp
14.72	14.72	34 212 436	Matrix::b(unsigned int, unsi...	blur: matrix.cpp
14.72	14.72	34 212 436	Matrix::g(unsigned int, unsi...	blur: matrix.cpp
14.72	14.72	34 212 436	Matrix::r(unsigned int, unsi...	blur: matrix.cpp
6.21	2.11	3	Matrix::Matrix(Matrix const...	blur: matrix.cpp
2.37	1.44	1 547 367	std::ostream::put(char)	libstdc++.so.6.0.30
4.33	1.37	1 547 364	std::istream::read(char*, lo...	libstdc++.so.6.0.30
1.85	1.18	1 547 364	std::basic_streambuf<char, ...	libstdc++.so.6.0.30
1.04	1.04	1 547 368	std::istream::sentry::sentry(...	libstdc++.so.6.0.30
0.85	0.85	1 547 375	std::ostream::sentry::sentry...	libstdc++.so.6.0.30
0.70	0.70	1 547 364	Matrix::b(unsigned int, unsi...	blur: matrix.cpp
0.70	0.70	1 547 364	Matrix::g(unsigned int, unsi...	blur: matrix.cpp
0.70	0.70	1 547 364	Matrix::r(unsigned int, unsi...	blur: matrix.cpp
5.26	0.68	1	PPM::Reader::get_data(unsi...	blur: ppm.cpp
0.62	0.62	1 548 335	__memcpy_avx_unaligned_...	libc.so.6: memmove-vec-unaligned-erms.S
2.96	0.52	1	PPM::Writer::operator()(Ma...	blur: ppm.cpp
0.11	0.11	1 547 364	std::istream::gcount() const	libstdc++.so.6.0.30
0.70	0.07	1 547 595	0x00000000048f9120	(unknown)

Fig. 31. Shows the initial performance call breakdown.

By investigating the ppm class that deals with inputs it was possible to see that the current reader function reads one pixel color at a time and then makes sure that the pixel has been read and stored. Once this procedure has been done for each color in the pixel, then it checks that it actually read the information, else the loop breaks. If everything went well, the loop starts to work on the next pixel.

In Fig.32 it is possible to see the breakdown of how the reading is working, where for each Pixel P iteration it reads the three colors and then moves to the next pixel. This procedure continues either until the matrix has been traversed or an error occurred. To speed up the process, the idea was to read the whole contents of all the pixels that contained in the image and then store the information in an array instead of having a matrix structure. The array structure could be shown in Fig.33,

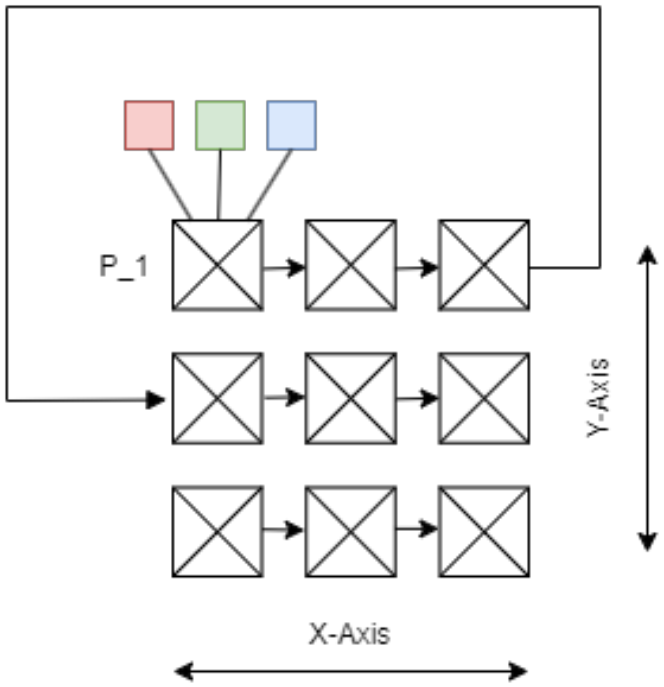


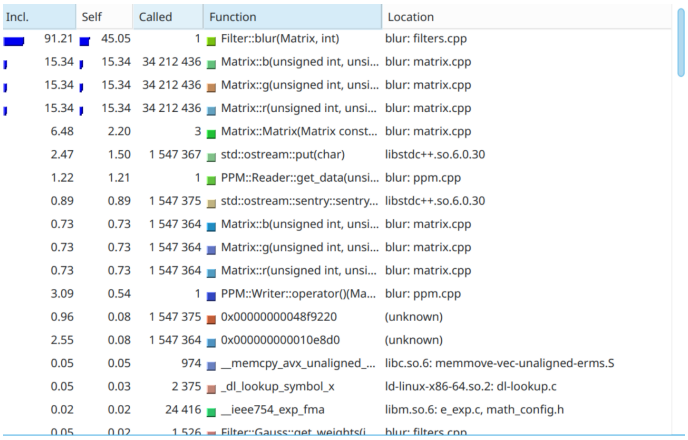
Fig. 32. Shows the concept for iteration 7.

Fig. 33. Shows the concept for iteration 7.

where each of the colors gets after each other. To access each of the pixels it is just simply to iterate through the array and then use the mod 3 to find out which color that is being read for the pixel. By applying this technique that increased the amount of data that was being read and changing the structure a little bit, it was possible to see that the gcount calls got reduced to zero. The extra if statement that got checked for each iteration got removed. The amount of copying went down to 974 from 1.55M calls. The istream read, stream buffer and istream sentry also got reduced to an insignificant amount of calls an self percentage. The new call graph is available in Fig.34 and the source code changes for this iteration is available in Fig.35.

I. Iteration eight

Since the discoveries in iteration seven seems to work well, the same idea got applied to the output writer function. The source code change could be seen in Fig.36, where first the buffer technique is used. In addition, the while loop has been replaced by a loop unrolling technique [5]. The unrolling of the loop made it possible to read each of the colors of a given pixel at each iteration.



Incl.	Self	Called	Function	Location
91.21	45.05	1	Filter::blur(Matrix, int)	blur: filters.cpp
15.34	15.34	34 212 436	Matrix::b(unsigned int, unsi...	blur: matrix.cpp
15.34	15.34	34 212 436	Matrix::g(unsigned int, unsi...	blur: matrix.cpp
15.34	15.34	34 212 436	Matrix::r(unsigned int, unsi...	blur: matrix.cpp
6.48	2.20	3	Matrix::Matrix(Matrix const...	blur: matrix.cpp
2.47	1.50	1 547 367	std::ostream::put(char)	libstdc++.so.6.0.30
1.22	1.21	1	PPM::Reader::get_data(unsi...	blur: ppm.cpp
0.89	0.89	1 547 375	std::ostream::sentry::sentry...	libstdc++.so.6.0.30
0.73	0.73	1 547 364	Matrix::b(unsigned int, unsi...	blur: matrix.cpp
0.73	0.73	1 547 364	Matrix::g(unsigned int, unsi...	blur: matrix.cpp
0.73	0.73	1 547 364	Matrix::r(unsigned int, unsi...	blur: matrix.cpp
3.09	0.54	1	PPM::Writer::operator()(Ma...	blur: ppm.cpp
0.96	0.08	1 547 375	0x00000000048f9220	(unknown)
2.55	0.08	1 547 364	0x000000000010e8d0	(unknown)
0.05	0.05	974	__memcpy_avx_unaligned_...	libc.so.6: memmove-vec-unaligned-erms.S
0.05	0.03	2 375	_dl_lookup_symbol_x	ld-linux-x86-64.so.2: dl-lookup.c
0.02	0.02	24 416	__ieee754_exp_fma	libm.so.6: e_exp.c, math_config.h
0.05	0.02	1 526	Filter::Gauss::get_weights(i...	blur: filters.cpp

Fig. 34. Shows the new call graph for iteration 7.

```
79  std::tuple<unsigned char*, unsigned char*, unsigned char*> Reader::get_data(unsigned x_size, unsigned y_size)
80  {
81      auto size { x_size * y_size };
82      auto R { new char[size] }, G { new char[size] }, B { new char[size] };
83      unsigned int temp = size*3;
84      char * buffer = new char [temp];
85      stream.read(buffer,temp);
86      auto pointR{0},pointG{0},pointB{0};
87      for(unsigned int i{0};i<temp;i++){
88        switch(i%3){
89            case 0: R[pointR] = buffer[i];pointR++;
90            break;
91            case 1: G[pointG] = buffer[i];pointG++;
92             break;
93            case 2: B[pointB] = buffer[i];pointB++;
94             break;
95
96        }
97
98      }
99      return { reinterpret_cast<unsigned char*>(R), reinterpret_cast<unsigned char*>(G), reinterpret_cast<unsigned char*>(B) };
100 }
```

Fig. 35. Shows the source code change for iteration 7.

```
    //auto size { m.get_x_size() * m.get_y_size() };
    auto R { m.get_R() }, G { m.get_G() }, B { m.get_B() };
    auto it_R { R }, it_G { G }, it_B { B };
     unsigned int temp = m.get_x_size() * m.get_y_size()*3;
     char * buffer = new char [temp];
     for(auto i{0};i<temp;i+=3){
         buffer[i] = *it_R++;
         buffer[i+1] = *it_G++;
         buffer[i+2] = *it_B++;
     }
    /* while (it_R < R + size && it_G < G + size && it_B < B + size) {
         //f << *it_R++
         //  << *it_G++
         //  << *it_B++;
         f.put(*it_R++);
         f.put(*it_G++);
         f.put(*it_B++);
     }
*/
    f.write(buffer,temp);
    f.close();
} catch (std::runtime_error e) {
    error("writing", e.what());
}
```

Fig. 36. Shows the source code change for iteration 8.

By investigating the new call graph breakdown, which is shown in Fig.37 it is possible to see that the amount of self percentage used by the put calls got reduced from 1.44 down to ≤ 0.01 and the output sentry got reduced to ≤ 0.01 from 0.85. In addition, the calls only gets performed an insignificant amount of times.

J. Iteration nine

By investigating the information in Fig.37 once again, it is possible to see that the get data function still had a significant impact of execution time. By investigating this further, it was

Incl.	Self	Called	Function	Location
93.70	46.28	1	Filter::blur(Matrix, int)	blur: filters.cpp
15.76	15.76	34 212 436	Matrix::b(unsigned int, unsi...	blur: matrix.cpp
15.76	15.76	34 212 436	Matrix::g(unsigned int, unsi...	blur: matrix.cpp
15.76	15.76	34 212 436	Matrix::r(unsigned int, unsi...	blur: matrix.cpp
6.65	2.26	3	Matrix::Matrix(Matrix const...	blur: matrix.cpp
1.26	1.24	1	PPM::Reader::get_data(unsi...	blur: ppm.cpp
0.75	0.75	1 547 364	Matrix::b(unsigned int, unsi...	blur: matrix.cpp
0.75	0.75	1 547 364	Matrix::g(unsigned int, unsi...	blur: matrix.cpp
0.75	0.75	1 547 364	Matrix::r(unsigned int, unsi...	blur: matrix.cpp
0.44	0.44	1	PPM::Writer::operator()(Ma...	blur: ppm.cpp
0.05	0.05	974	__memcpy_avx_unaligned_...	libc.so.6: memmove-vec-unaligned-erms.S
0.05	0.03	2 374	_dl_lookup_symbol_x	ld-linux-x86-64.so.2: dl-lookup.c
0.02	0.02	24 416	__ieee754_exp_fma	libm.so.6: e_exp.c, math_config.h
0.05	0.02	1 526	Filter::Gauss::get_weights(i...	blur: filters.cpp
0.02	0.02	2 374	do_lookup_x	ld-linux-x86-64.so.2: dl-lookup.c, dl-protected.h, ld...
0.06	0.01	7	_dl_relocate_object	ld-linux-x86-64.so.2: dl-reloc.c, dl-machine.h, do-re...
0.03	0.01	24 416	exp@@GLIBC_2.29	libm.so.6: w_exp_template.c

Fig. 37. Shows the call graph for iteration 8.

possible to see that the modulus operation that was used for reading the inputs and finding out which color it belongs to was costly. By instead applying a loop unrolling technique, it was possible to exchange the expensive modulus operation to instead collect the colors for each pixel at each iteration. The source code change is available in Fig.38.

```
std::tuple<unsigned char*, unsigned char*, unsigned char*> Reader::get_data(unsigned x_size, unsigned y_size)
{
    auto size { x_size * y_size };
    auto R { new char[size] }, G { new char[size] }, B { new char[size] };
    unsigned int temp = size*3;
    char * buffer = new char [temp];
    stream.read(buffer,temp);
    auto poss{0};
    for(unsigned int i{0};i<temp;i+=3){
            R[poss] = buffer[i];
            G[poss] = buffer[i+1];
            B[poss] = buffer[i+2];
            poss++;
        }
    return { reinterpret_cast<unsigned char*>(R), reinterpret_cast<unsigned char*>(G), reinterpret_cast<unsigned char*>(B) };
}
```

Fig. 38. Shows the source code changes for iteration 9.

This change caused the self percentage to drop down from 1.24 to 0.48. As could be seen in the newly generated call graph in Fig.38.

Incl.	Self	Called	Function	Location
94.43	46.64	1	Filter::blur(Matrix, int)	blur: filters.cpp
15.88	15.88	34 212 436	Matrix::b(unsigned int, unsi...	blur: matrix.cpp
15.88	15.88	34 212 436	Matrix::g(unsigned int, unsi...	blur: matrix.cpp
15.88	15.88	34 212 436	Matrix::r(unsigned int, unsi...	blur: matrix.cpp
6.71	2.28	3	Matrix::Matrix(Matrix const...	blur: matrix.cpp
0.76	0.76	1 547 364	Matrix::b(unsigned int, unsi...	blur: matrix.cpp
0.76	0.76	1 547 364	Matrix::g(unsigned int, unsi...	blur: matrix.cpp
0.76	0.76	1 547 364	Matrix::r(unsigned int, unsi...	blur: matrix.cpp
0.49	0.48	1	PPM::Reader::get_data(unsi...	blur: ppm.cpp
0.44	0.44	1	PPM::Writer::operator()(Ma...	blur: ppm.cpp
0.05	0.05	974	__memcpy_avx_unaligned_...	libc.so.6: memmove-vec-unaligned-erms.S
0.05	0.03	2 374	_dl_lookup_symbol_x	ld-linux-x86-64.so.2: dl-lookup.c

Fig. 39. Shows the call graph for iteration 9.

K. Iteration ten

For the last iteration ten, no further improvements was found on a software level. However, by investigating the compiler flags that was used for compiling the software it was possible to see that multiple flags related to debugging was used, such a the ”-g” flag. By using debugging flags, it tells the compiler to optimize for debugging and finding errors in the code rather than optimizing the code for performance. By investigating the compiler flags provided by the documentation of gcc [3], it was possible to see that there is an option called ”-O3” which enables all optimization techniques related to execution time. The trade-off is that the amount of memory used might increase and the compilation might take longer time. A complete breakdown of the different optimization techniques that is enabled by the compiler is available on the documentation page cited above. Worth mentioning was that the inline change that was applied in iteration four had to be removed in order to enable the new compiler optimization.

L. Execution time between iterations

As could be seen in Fig.40, the execution time for the blur software has been reduced from 16 seconds down to 2,6 seconds. This performance optimization has made the software over six times faster than it originally was. By taking a closer look on the graph, it is possible to see that the optimizations regarding reducing loop invariant calls and optimizing the compiler, had the most impact on the execution time. This results goes inline with the theory that the time complexity has a large impact on the execution time, this is shown in the jump between iteration two to iteration three. In addition, it seems that for running software in production, the optimization flags should also be set to increase the performance. Fine-tuning algorithms seems to have an impact on the overall performance, but it is not as large as changing the algorithm itself or limiting redundant function calls.

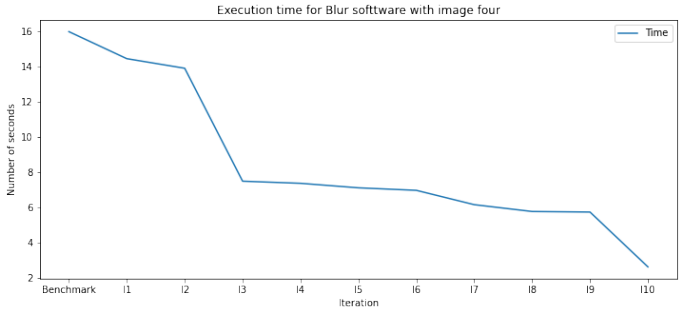


Fig. 40. Shows the execution time for each iteration.

For the threshold software the benchmark tests was run at 2.55 seconds and it got reduced down to 0.26 second for the final version. This is an improvement of almost ten times. As could be seen for the first iterations, the changes had no impact on the threshold software and the little fluctuation that was shown is most likely due to fluctuation by the measurement tool. However, the I/O changes seems to have the largest impact on the execution time. In addition, the compiler optimization also had a big impact. This information is Presented in Fig.41

IV. Discussion

During the investigation, it was found out that there was fluctuation in the time measurement tool. This fluctuation was tried to get reduced by making multiple measurements and

Fig. 41. Shows the execution time for each iteration.

then computing the average between the execution runs. In addition, when performing minor fine-tuning adjustments it is crucial that the measurement tool is able to being precise when data is captured to ensure that the real characteristics of the software is presented. Multiple minor fine-tuning could have a large impact on the over all performance of the software and missing out on those by an insufficient measurement tool might be costly. In short, if the collected data is of bad quality, then the analysis will be harder to make and might cause inappropriate conclusions. Due to this reason, it seems to be of value to have a good measurement tool.

One interesting thought that got brought up was that depending on when an optimization technique gets implemented might generate different results. For an example to reduce the execution time of a software that takes 100 seconds by 20%, will result in a reduction of 20 seconds. As the execution time will become shorter, a reduction of 20% in execution time when the overall execution time is only 1 second, will only produce a decrease of 0.2 seconds. the change still yielded in a great improvement by speaking in terms of percentage, but taking the time into consideration it might not be worth spending too much time to really squeeze out the last few percentages of an application if it is not required and if it compromise other things such as maintainability, scalability, security or understandability. It is up to the engineer to evaluate the application and what task it tries to solve in order to make good design decisions related to performance.

Another finding that is worth bringing up is that it might be possible to make changes that will have an improvement, but might not solve the underlying problem. To provide an example, lets say that you have a function that contains expensive arithmetic operations and that the function gets called multiple times within the software. The developer might came up with an idea that make is possible to exchange the expensive arithmetic operations inside the function with applying pre-computing techniques and since the function gets called so many times, this will have a significant effect on the execution time. However, if the developer was searching deeper, the root cause might have been that the function was called at times when it was not needed. By solving the root problem, might make the pre-computing techniques that was applied earlier insignificant. Also by taking this into consideration, the order of when a certain change gets implemented might view different results.

V. Conclusions

Before the investigation of the image transformation software took place, the execution speed was approximate 16 seconds for the blur algorithm and 2.6 seconds for the threshold algorithm. With the help of various profiling tools, it was possible to spot inefficient code design that not only caused unnecessarily function calls, also generated cache misses. By applying various optimization techniques and reconstruction of the code, it was possible to reduce the amount of time that the blur software required to execute starting from 16 seconds down to 2.6 seconds. This improvement made the software over six times faster. In addition, the the execution time for the threshold software got reduced from 2.55 seconds down to 0.26 seconds, which is almost ten times as fast.

VI. Acknowledgment

We would like to thank David Holmqvist for engaging in interesting discussion related to performance optimization.

References

[1] “C++ Documentation:.” [Online]. Available: https://cplusplus.com/doc/tutorial/files/
[2] “Cache memory in computer organization:.” [Online]. Available: https://www.geeksforgeeks.org/cache-memory-in-computer-organization/
[3] “GCC documentation:.” [Online]. Available: https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html#Optimize-Options
[4] “Loop interchange:.” [Online]. Available: https://en.wikipedia.org/wiki/Loop_interchange
[5] “Loop Unrolling:.” [Online]. Available: https://en.wikipedia.org/wiki/Loop_unrolling
[6] “LoopInvarient:.” [Online]. Available: https://en.wikipedia.org/wiki/Loop_invariant
[7] “Microsoft Inline functions:.” [Online]. Available: https://learn.microsoft.com/en-us/cpp/cpp/inline-functions-cpp?view=msvc-170
[8] “Valgrind cg_annotate:.” [Online]. Available: https://valgrind.org/docs/manual/cg-manual.html
[9] K. Guntheroth, *Optimized C++*. O’Reilly Media, 2016.