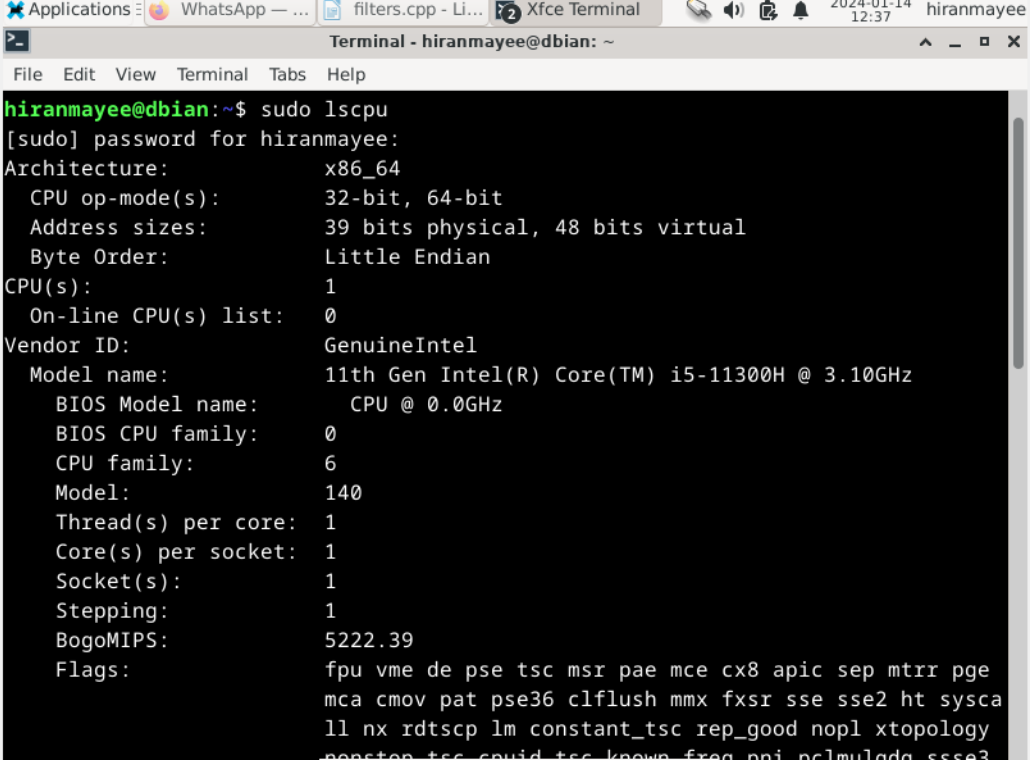


## 1. INTRODUCTION:

This report explains different techniques used to optimize the code of the given transformation application. There are two simple transformation algorithms implemented, Blurring and thresholding which operate on ppm images. We used different techniques mentioned in software\_optimization\_cookbook provided and valgrind and Kcachegrind tool for profiling and seeing the optimization of the code.

Initially, we provide the System under testing used for this project:



```
hiranmayee@dbian:~$ sudo lscpu
[sudo] password for hiranmayee:
Architecture:             x86_64
  CPU op-mode(s):         32-bit, 64-bit
  Address sizes:          39 bits physical, 48 bits virtual
  Byte Order:              Little Endian
CPU(s):                    1
  On-line CPU(s) list:    0
Vendor ID:                 GenuineIntel
Model name:                11th Gen Intel(R) Core(TM) i5-11300H @ 3.10GHz
  BIOS Model name:        CPU @ 0.0GHz
  BIOS CPU family:        0
  CPU family:             6
  Model:                  140
  Thread(s) per core:     1
  Core(s) per socket:     1
  Socket(s):              1
  Stepping:               1
  Bogomips:               5222.39
  Flags:                  fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge
                        mca cmov pat pse36 clflush mmx fxsr sse sse2 ht sysca
                        ll nx rdtscp lm constant_tsc rep_good nopl xtopology
                        noester tsc_cquid tsc_known_freq npi noplulada ssse3
```

Fig 1: SUT

Different commands used for the project are given below:

1. **make** is used to run the project make file to get compiler commands to execute the code.

```
g++ -std=c++17 -g -Wunused -Wall -Wunused blur.cpp matrix.o ppm.o filters.o -o
blur
g++ -std=c++17 -g -Wunused -Wall -Wunused threshold.cpp matrix.o ppm.o filters
.o -o threshold
```

- 2.

3. Valgrind --tool=callgrind ./blur 15 data/im1.ppm bluroutputim1.ppm
4. Valgrind --tool=callgrind ./threshold data/im1.ppm thresoutputim1.ppm
5. Kcachegrind callgrind.out.pid – common for all pid's
6. Time ./function 15 input.ppm output.ppm – can give time taken to execute the function.

```

filters.cpp:103:26: warning: comparison of integer expressions of different si
gnedness: 'int' and 'unsigned int' [-Wsign-compare]
  103 |         for (auto i { 0 }; i < nump; i++) {
      |                             ~~~~~
g++ -std=c++17 -g -Wunused -Wall -Wunused blur.cpp matrix.o ppm.o filters.o -o
blur
g++ -std=c++17 -g -Wunused -Wall -Wunused threshold.cpp matrix.o ppm.o filters
.o -o threshold
hiranmayee@dbian:~/Desktop/filters_org/filters$ ^[[200~g++ -std=c++17 -g -Wunu
sed -Wall -Wunused blur.cpp matrix.o ppm.o filters.o -o blur
bash: $'\E[200~g++': command not found
hiranmayee@dbian:~/Desktop/filters_org/filters$ ~
bash: /home/hiranmayee: Is a directory
hiranmayee@dbian:~/Desktop/filters_org/filters$ g++ -std=c++17 -g -Wunused -Wa
ll -Wunused blur.cpp matrix.o ppm.o filters.o -o blur
hiranmayee@dbian:~/Desktop/filters_org/filters$ g++ -std=c++17 -g -Wunused -Wa
ll -Wunused threshold.cpp matrix.o ppm.o filters.o -o threshold
hiranmayee@dbian:~/Desktop/filters_org/filters$ ls
blur      filters.cpp  Makefile    matrix.o    ppm.o       verify.sh
blur.cpp  filters.hpp  matrix.cpp  ppm.cpp     threshold
data      filters.o   matrix.hpp  ppm.hpp    threshold.cpp
hiranmayee@dbian:~/Desktop/filters_org/filters$

```

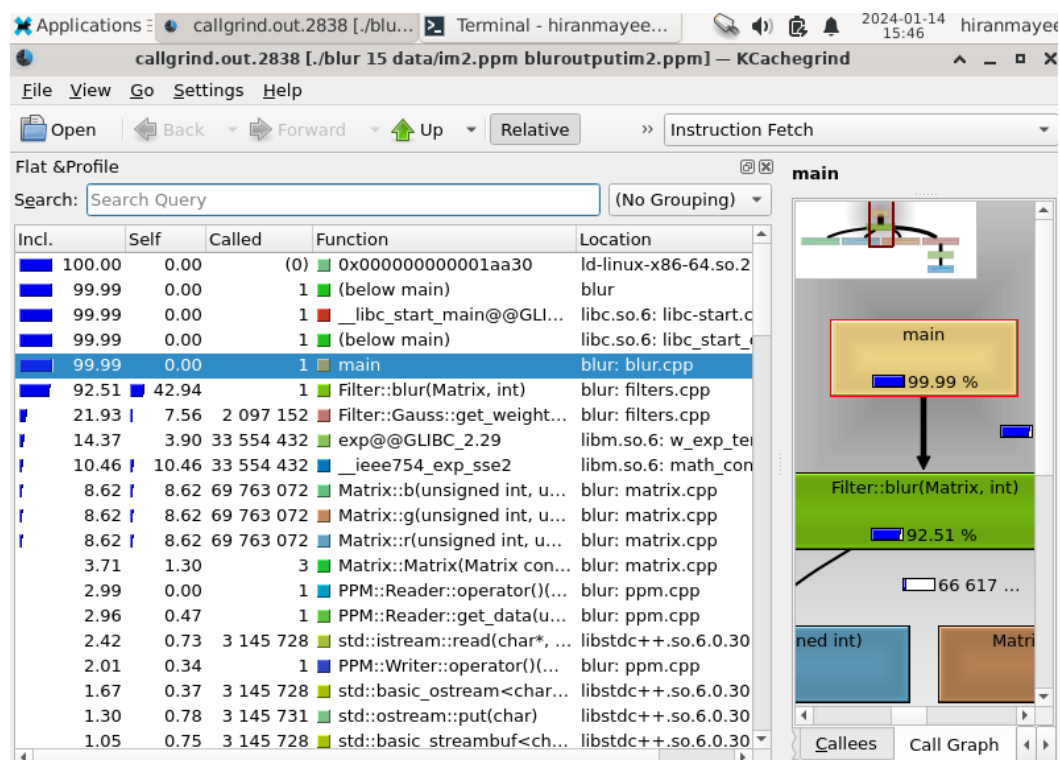
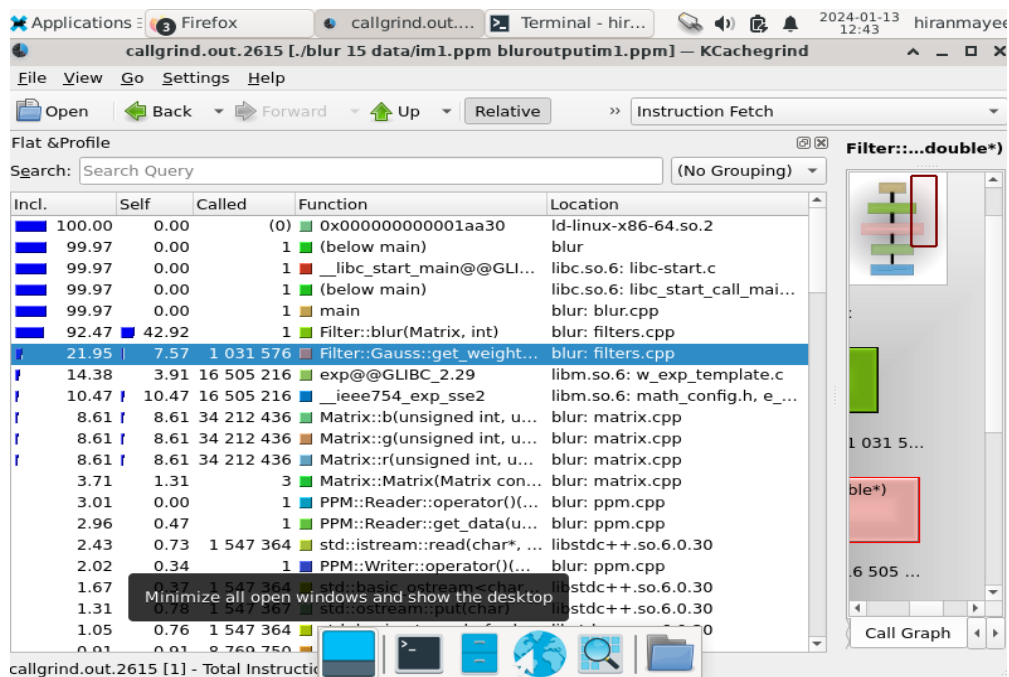
Fig 2: Make file execution.

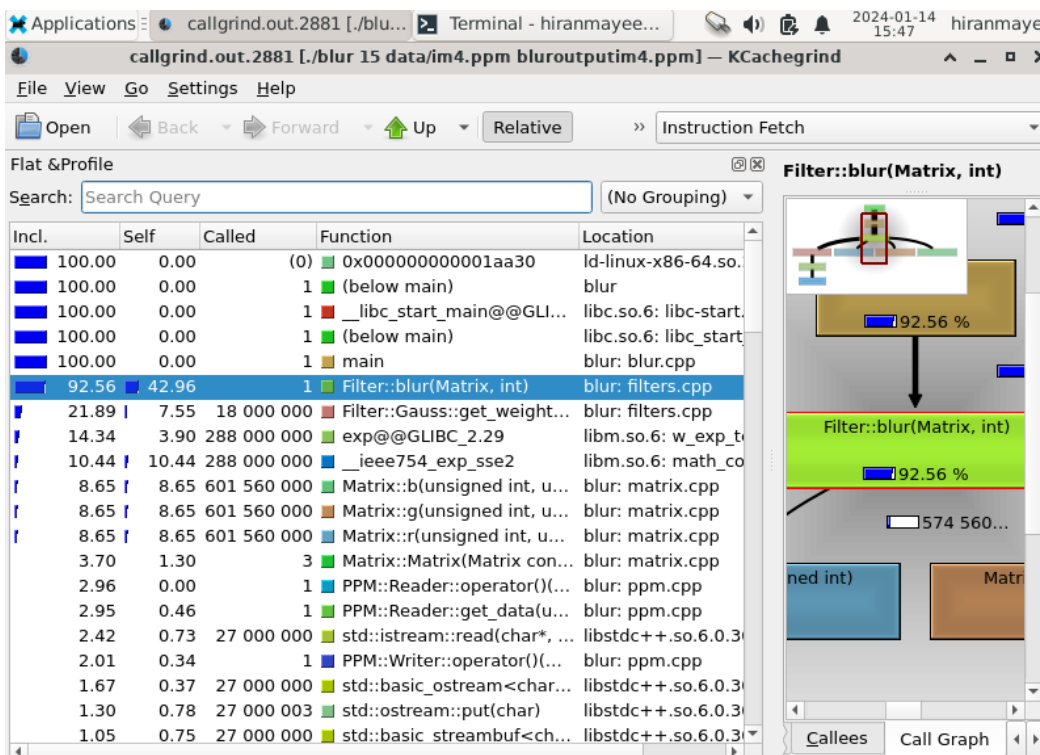
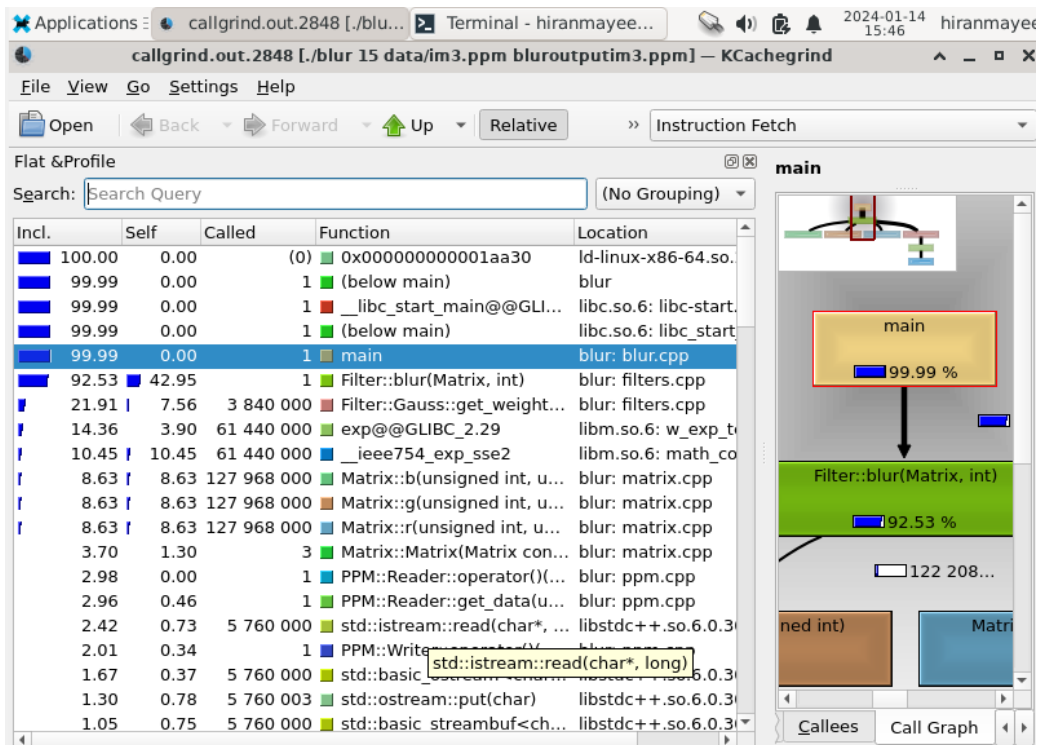
## 2. Baseline Performance:

Identifying Hotspots in the application is very important in code optimization of any application which shows us the areas of code which are resource intensive or inefficient. Using profiling tools like Valgrind and Kcachegrind helps us get profiling data and visualize it in terms of call graph.

Below are the call graphs and execution of using profiling tools for blur and threshold functions:

### Blur Function: 4 ppm files execution for blur function

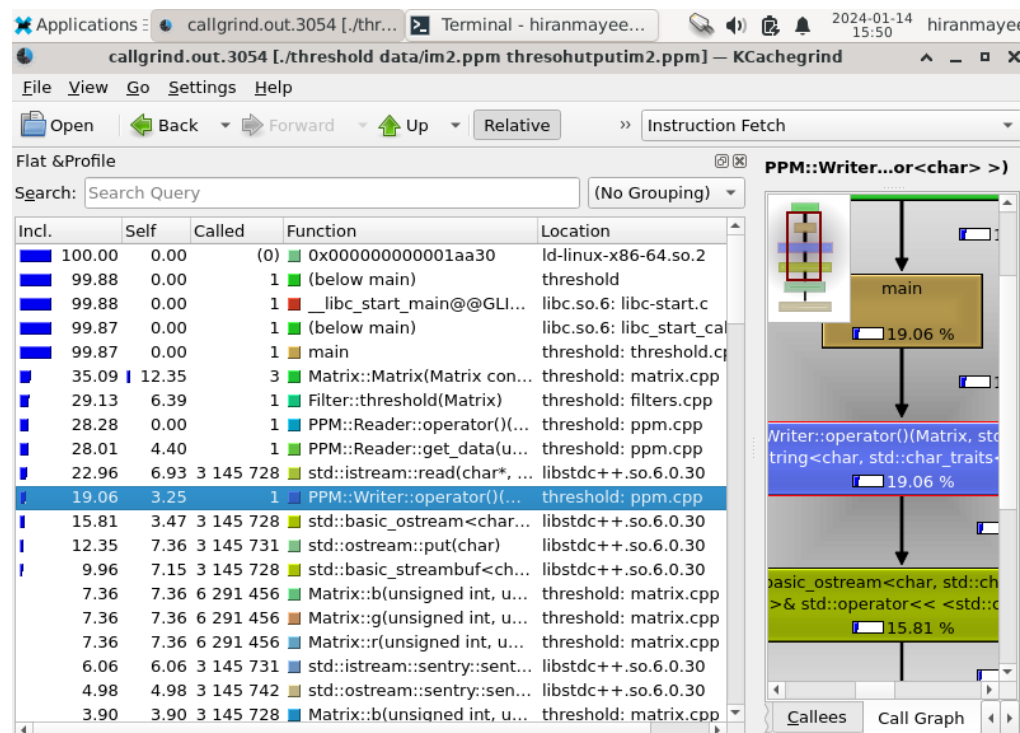
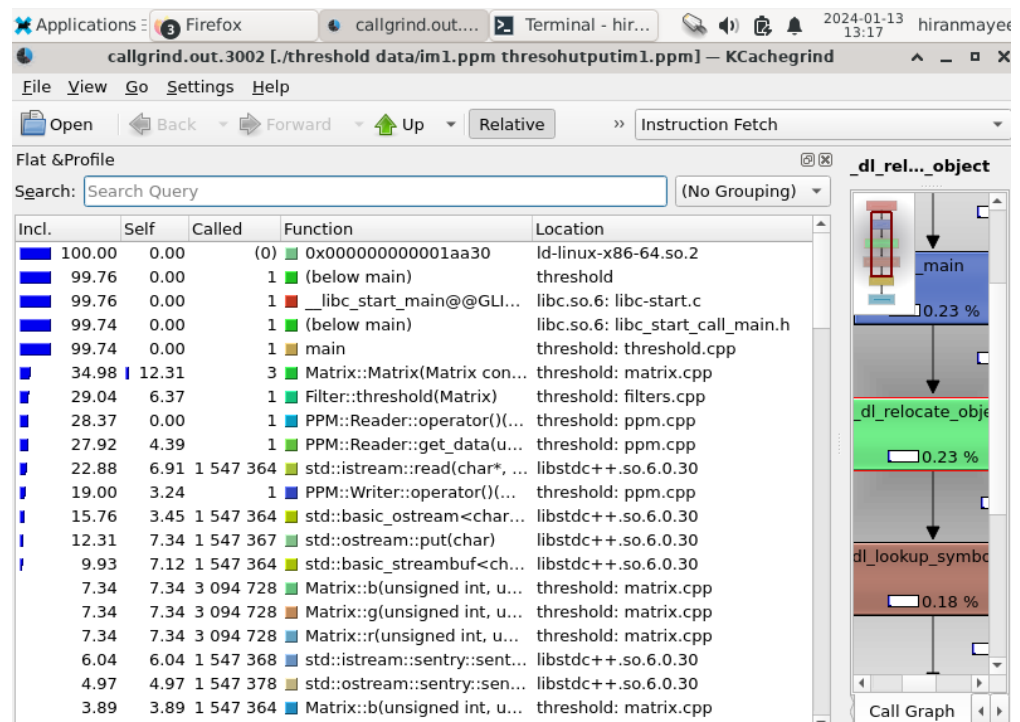


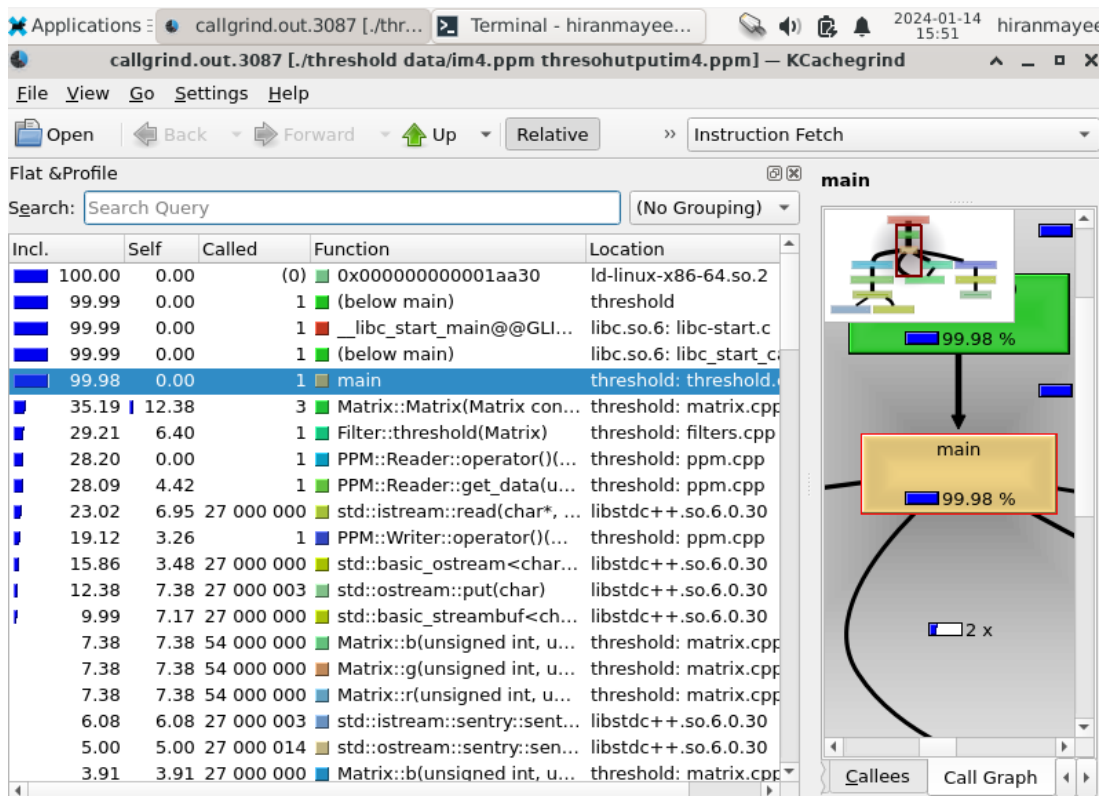
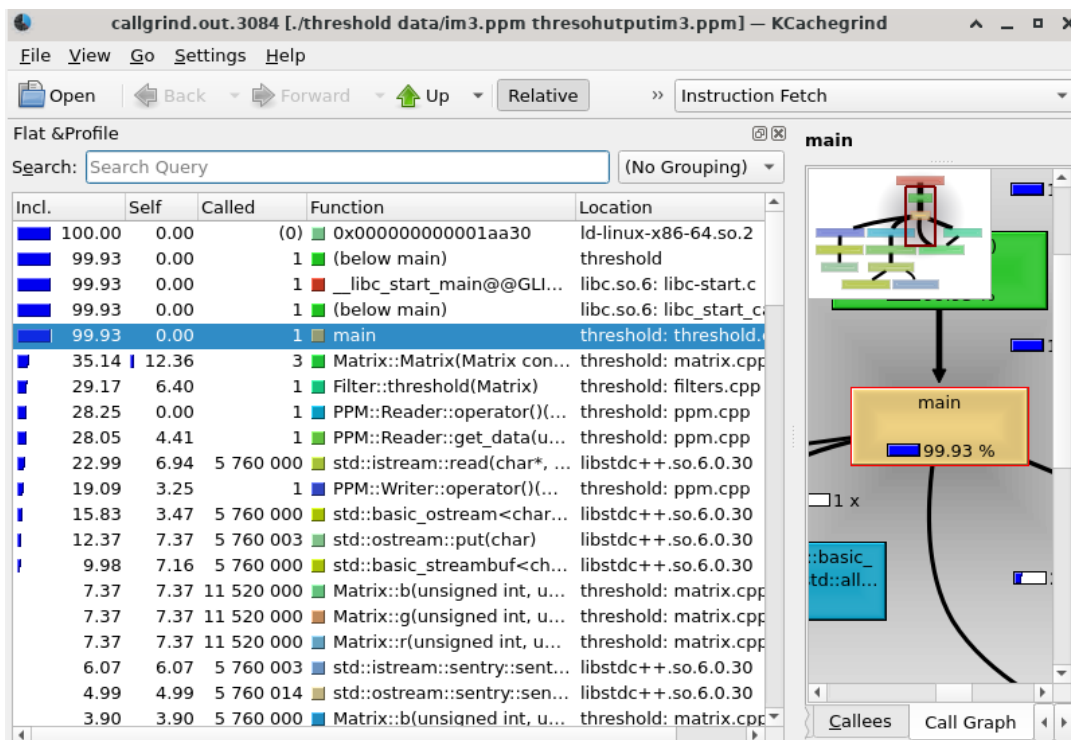


On observing the above Callgraph, we can understand, function named “Filter::Gauss::get\_weight” in Filter::Blur takes a lot of execution time and has high number of self-calls,

Then secondly, most number of self-calls and execution time is in Matrix::b, Matrix::g and Matrix::r functions.

## Threshold Function: 4 PPM file execution for threshold function.





Looking at the details and Callgraph for the above threshold functions, we can observe, Matrix::Matrix(Matrix const) has highest allocated time along with Filter::threshold, PPM::Reader::Operator and PPM::Reader::get data operator.



With the help of this hotspots, we have performed several optimization techniques inspired by the materials provided in the course. Below is the time execution of blur and threshold files before optimization:

```
hiranmayee@dbian:~/Desktop/filters_org/filters$ time ./blur 15 data/im4.ppm bl
uroutputim4.ppm

real    0m17.332s
user    0m10.813s
sys     0m5.842s
```

Figure 3: Time execution for blur function.

```
hiranmayee@dbian:~/Desktop/filters_org/filters$ time ./threshold data/im3.ppm
threshoutputim3.ppm

real    0m0.390s
user    0m0.247s
sys     0m0.114s
```

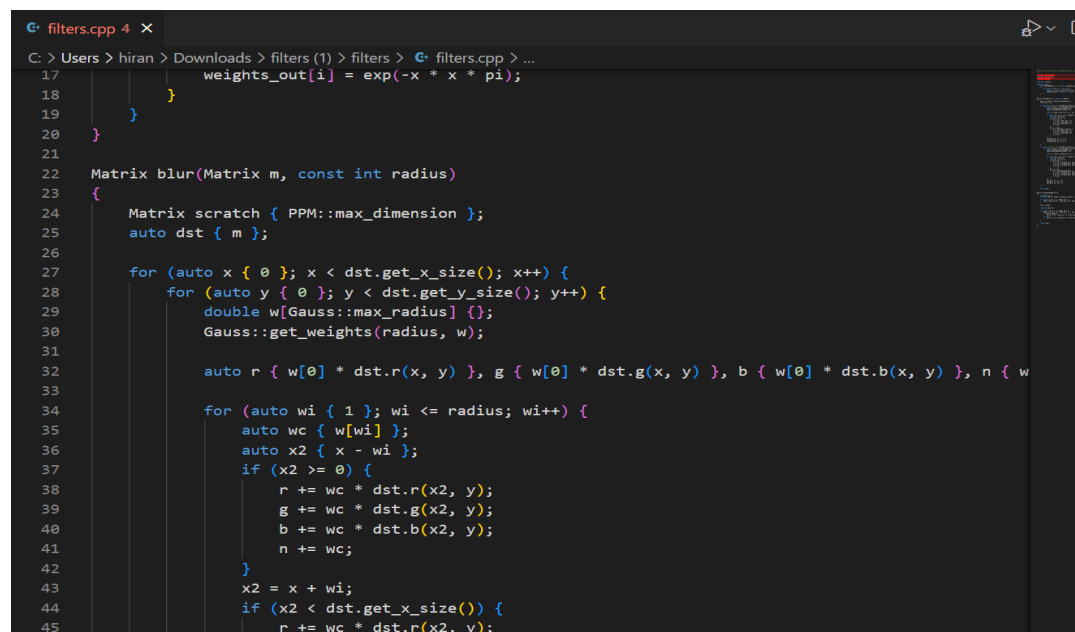
Figure 3: Time execution for threshold function.

### 3.OPTIMIZATION TECHNIQUES:

#### 1. Loop Invariant:

Motivation: Calculations that don't change between loop iterations are called loop invariant computations. Few branches make it harder for the compilers to perform optimizations. So, these branches can be moved outside the loop.

Approach: In blur function: we have found that the gaussian weights are recalculated for every pixel calling the loop and making it inefficient. Moving the gaussian weights outside the loop using Loop invariant technique can help reduce the loop calls and execution time.



```
filters.cpp 4 X
C: > Users > hiran > Downloads > filters (1) > filters > filters.cpp > ...
17         weights_out[i] = exp(-x * x * pi);
18     }
19 }
20
21
22 Matrix blur(Matrix m, const int radius)
23 {
24     Matrix scratch { PPM::max_dimension };
25     auto dst { m };
26
27     for (auto x { 0 }; x < dst.get_x_size(); x++) {
28         for (auto y { 0 }; y < dst.get_y_size(); y++) {
29             double w[Gauss::max_radius] {};
30             Gauss::get_weights(radius, w);
31
32             auto r { w[0] * dst.r(x, y) }, g { w[0] * dst.g(x, y) }, b { w[0] * dst.b(x, y) }, n { w
33
34             for (auto wi { 1 }; wi <= radius; wi++) {
35                 auto wc { w[wi] };
36                 auto x2 { x - wi };
37                 if (x2 >= 0) {
38                     r += wc * dst.r(x2, y);
39                     g += wc * dst.g(x2, y);
40                     b += wc * dst.b(x2, y);
41                     n += wc;
42                 }
43                 x2 = x + wi;
44                 if (x2 < dst.get_x_size()) {
45                     r += wc * dst.r(x2, y);
```

Figure 4: Before optimization.

```

C: > Users > hiran > Downloads > filters_optimizedcode > filters_optimizedcode > filters > filters.cpp > ...
21
22 Matrix blur(Matrix m, const int radius)
23 {
24     Matrix scratch { PPM::max_dimension };
25     auto dst { m };
26
27     double w[Gauss::max_radius] {};
28     Gauss::get_weights(radius, w);
29
30     for (auto y { 0 }; y < dst.get_y_size(); y++) {
31         for (auto x { 0 }; x < dst.get_x_size(); x++) {
32
33             auto r { w[0] * dst.r(x, y) }, g { w[0] * dst.g(x, y) }
34
35             for (auto wi { 1 }; wi <= radius; wi++) {
36                 auto wc { w[wi] };
37                 auto x2 { x - wi };
38                 if (x2 >= 0) {
39                     r += wc * dst.r(x2, y);
40                     g += wc * dst.g(x2, y);
41                     b += wc * dst.b(x2, y);
42                     n += wc;

```

Figure 5: Optimized code for loop invariant.

We can say the technique is quite successful because of the reduced execution time.

```

hiranmayee@dbian:~/Downloads/filters_loopinvariant/filters$ time ./blur 15 data/im4.ppm bluroutputim4.ppm
real    0m13.792s
user    0m7.993s
sys     0m5.334s

```

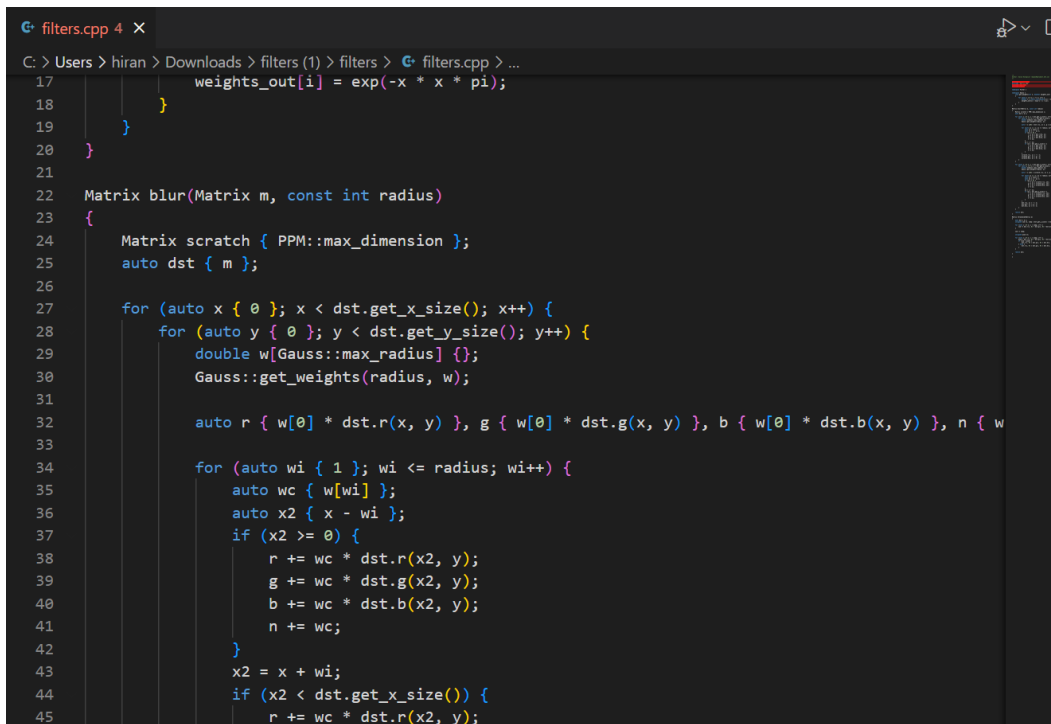
2. **Optimization attempt:** Loop Unrolling (In blur function) is the combination of two or more loop iterations together. In unrolled version, the code is little bit increased but reduces overhead. Because there is a overhead in the Matix blur function, so we use loop unrolling that iterates over the radius (wi) for blurring. The method inturn slightly increased the execution time, so it's not used.

### 3. Loop Nest Optimization(Loop Interchanging):

Motivation: The transformation technique switches the position of one loop tightly nested within another loop. In the code, there are nested loops for x and y and within these loops there is another loop wi. Applying loop interchange between x and y can potentially improve cache locality as data accessed within the inner loop (e.g dst.r(x,y) , dst.g(x,y)) remains in cache for longer time.

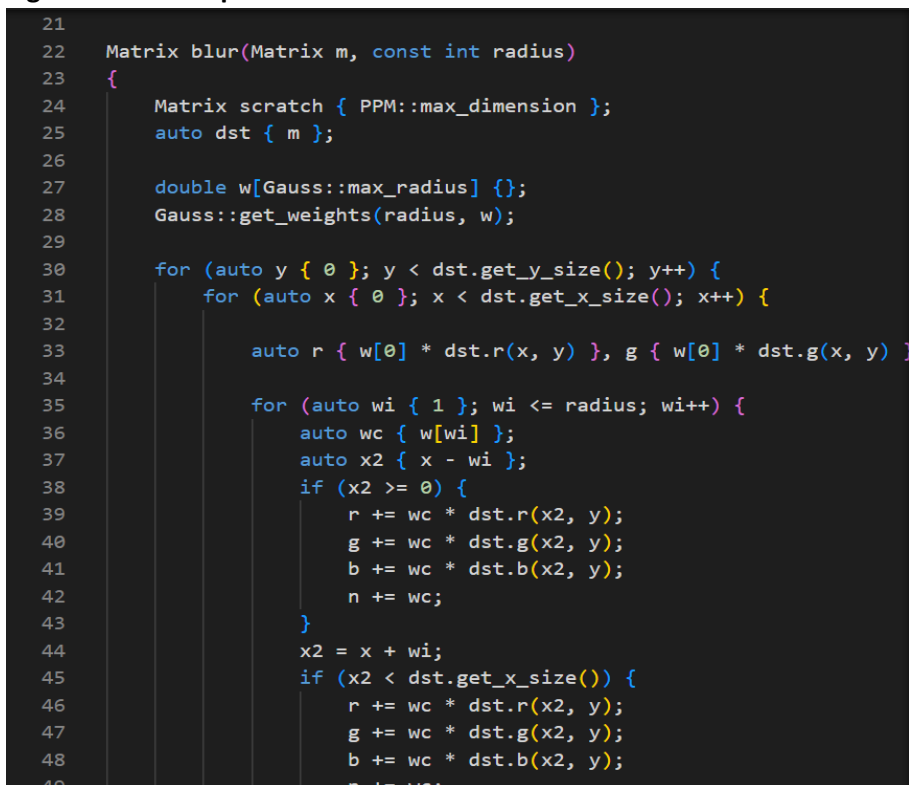


Approach: This attempt slightly reduced the execution time.



```
filters.cpp 4 X
C: > Users > hiran > Downloads > filters (1) > filters > filters.cpp > ...
17     weights_out[i] = exp(-x * x * pi);
18 }
19 }
20 }
21
22 Matrix blur(Matrix m, const int radius)
23 {
24     Matrix scratch { PPM::max_dimension };
25     auto dst { m };
26
27     for (auto x { 0 }; x < dst.get_x_size(); x++) {
28         for (auto y { 0 }; y < dst.get_y_size(); y++) {
29             double w[Gauss::max_radius] {};
30             Gauss::get_weights(radius, w);
31
32             auto r { w[0] * dst.r(x, y) }, g { w[0] * dst.g(x, y) }, b { w[0] * dst.b(x, y) }, n { w
33
34             for (auto wi { 1 }; wi <= radius; wi++) {
35                 auto wc { w[wi] };
36                 auto x2 { x - wi };
37                 if (x2 >= 0) {
38                     r += wc * dst.r(x2, y);
39                     g += wc * dst.g(x2, y);
40                     b += wc * dst.b(x2, y);
41                     n += wc;
42                 }
43                 x2 = x + wi;
44                 if (x2 < dst.get_x_size()) {
45                     r += wc * dst.r(x2, y);
```

Figure 6: Before optimization.



```
21
22 Matrix blur(Matrix m, const int radius)
23 {
24     Matrix scratch { PPM::max_dimension };
25     auto dst { m };
26
27     double w[Gauss::max_radius] {};
28     Gauss::get_weights(radius, w);
29
30     for (auto y { 0 }; y < dst.get_y_size(); y++) {
31         for (auto x { 0 }; x < dst.get_x_size(); x++) {
32             auto r { w[0] * dst.r(x, y) }, g { w[0] * dst.g(x, y) }
33
34             for (auto wi { 1 }; wi <= radius; wi++) {
35                 auto wc { w[wi] };
36                 auto x2 { x - wi };
37                 if (x2 >= 0) {
38                     r += wc * dst.r(x2, y);
39                     g += wc * dst.g(x2, y);
40                     b += wc * dst.b(x2, y);
41                     n += wc;
42                 }
43                 x2 = x + wi;
44                 if (x2 < dst.get_x_size()) {
45                     r += wc * dst.r(x2, y);
46                     g += wc * dst.g(x2, y);
47                     b += wc * dst.b(x2, y);
48                     n += wc;
49                 }
```

Figure 7: Optimized code for loop interchanging.

```

hiranmayee@dbian:~/Downloads/filters_threshold/filters$ time ./blur 15 data/im
4.ppm thresoutputim4.ppm

real    0m12.517s
user    0m8.327s
sys     0m4.087s

```

Figure 8: Time for execution.

#### 4. Optimization in Makefile:

Motivation: To enable the compiler to apply more aggressive optimizations for code, we enable '-O3' flag in the make file. This significantly reduces the execution time.

```

C: > Users > hiran > Downloads > filters_optimizedcode > filters_optimizedcode > filters > M
1  # Author: David Holmqvist <daae19@student.bth.se>
2
3  CXX=g++
4  CXXFLAGS=-std=c++17 -g -O3 -Wunused -Wall -Wunused
5
6  all: blur threshold
7
8  threshold: matrix ppm filters threshold.cpp
9      $(CXX) $(CXXFLAGS) threshold.cpp matrix.o ppm.o filters.o -o threshold
10
11 blur: matrix ppm filters blur.cpp
12     $(CXX) $(CXXFLAGS) blur.cpp matrix.o ppm.o filters.o -o blur
13
14 filters: matrix filters.hpp filters.cpp
15     $(CXX) $(CXXFLAGS) -c filters.cpp -o filters.o
16
17 matrix: matrix.hpp matrix.cpp
18     $(CXX) $(CXXFLAGS) -c matrix.cpp -o matrix.o
19
20 ppm: ppm.hpp ppm.cpp
21     $(CXX) $(CXXFLAGS) -c ppm.cpp -o ppm.o
22
23 clean:
24     rm -rf blur threshold *.ppm *.o *.dSYM 2> /dev/null
25

```

Figure 9: Optimized make file.

```

hiranmayee@dbian:~/Downloads/filters_threshold/filters$ time ./blur 15 data/im
4.ppm bluroutputim4.ppm

real    0m11.017s
user    0m9.979s
sys     0m0.801s
hiranmayee@dbian:~/Downloads/filters_threshold/filters$ time ./threshold data
/im3.ppm thresoutputim3.ppm

real    0m0.337s
user    0m0.266s
sys     0m0.050s

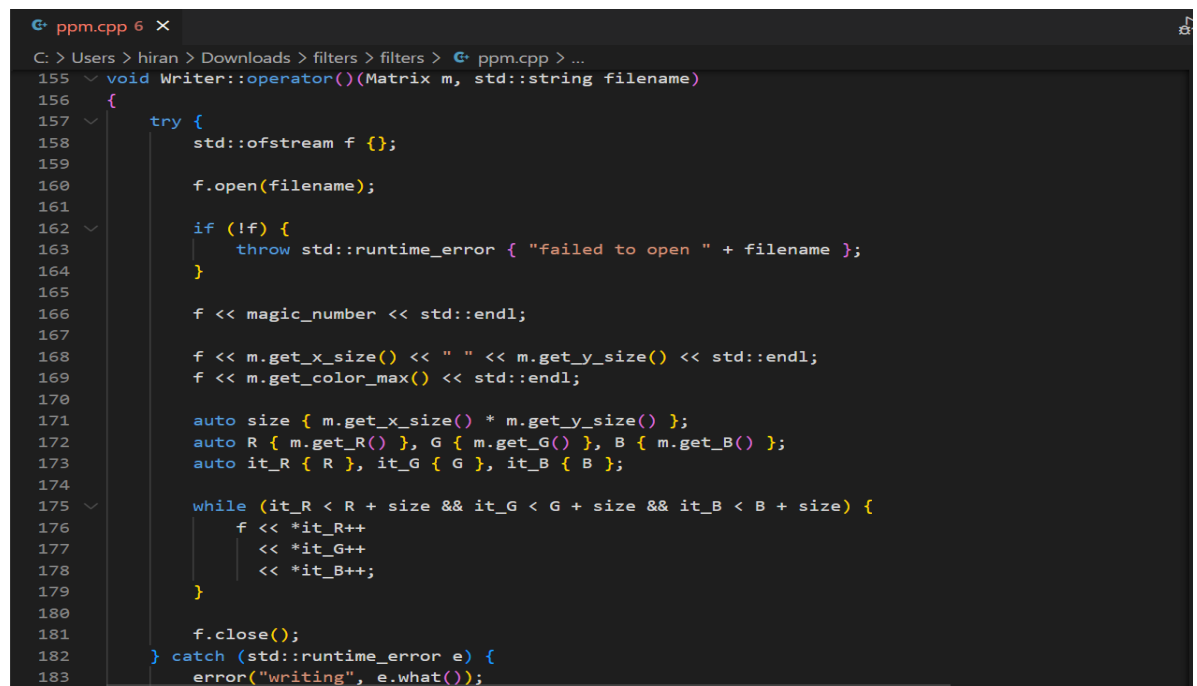
```

Figure 10: Time execution improved.

## 5. Newline optimization technique:

Motivation: Since, we haven't optimized any threshold hotspots, we focused on using the newline optimization technique in PPM::writer function which is a hotspot in threshold function.

Generally, in many C++ programming languages, endl is used to represent end-of-line character and flush the output buffer. Flushing the output buffer is expensive and stores it in memory. So, using '\n' doesn't trigger this output buffer and reduces the flush operations and performance of the code.

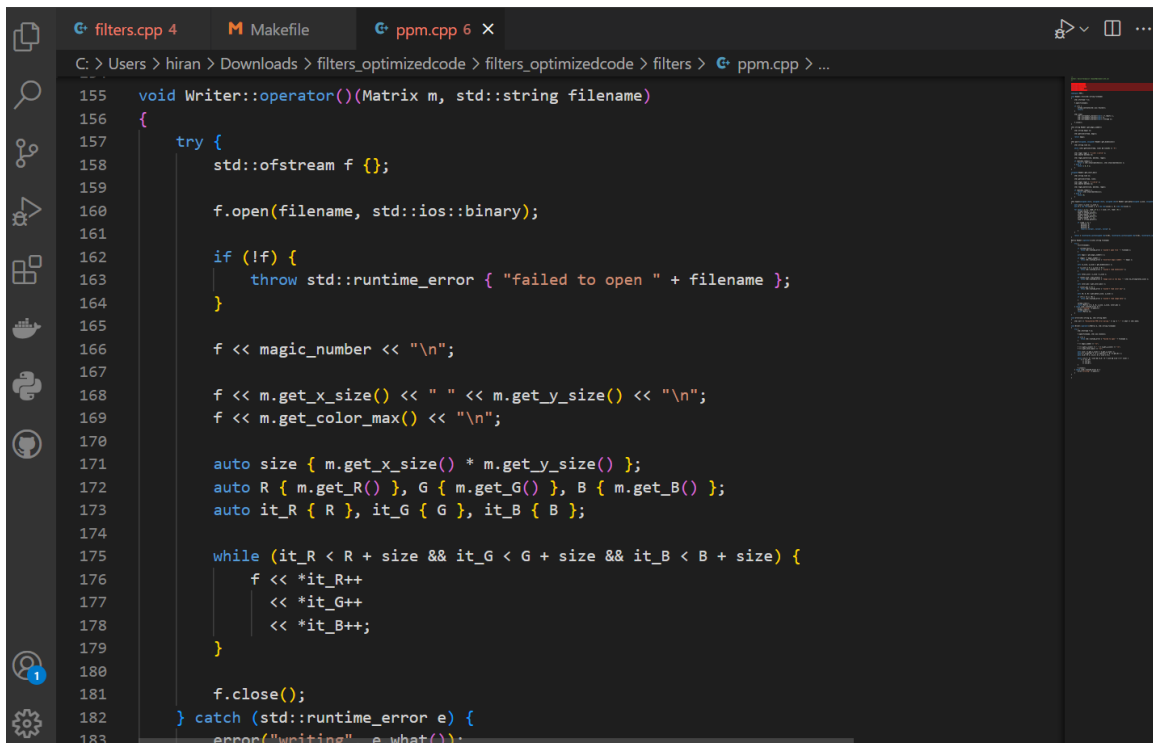


```

ppm.cpp 6 X
C: > Users > hiran > Downloads > filters > filters > ppm.cpp > ...
155 void Writer::operator()(Matrix m, std::string filename)
156 {
157     try {
158         std::ofstream f {};
159
160         f.open(filename);
161
162         if (!f) {
163             throw std::runtime_error { "failed to open " + filename };
164         }
165
166         f << magic_number << std::endl;
167
168         f << m.get_x_size() << " " << m.get_y_size() << std::endl;
169         f << m.get_color_max() << std::endl;
170
171         auto size { m.get_x_size() * m.get_y_size() };
172         auto R { m.get_R() }, G { m.get_G() }, B { m.get_B() };
173         auto it_R { R }, it_G { G }, it_B { B };
174
175         while (it_R < R + size && it_G < G + size && it_B < B + size) {
176             f << *it_R++
177               << *it_G++
178               << *it_B++;
179         }
180
181         f.close();
182     } catch (std::runtime_error e) {
183         error("writing", e.what());

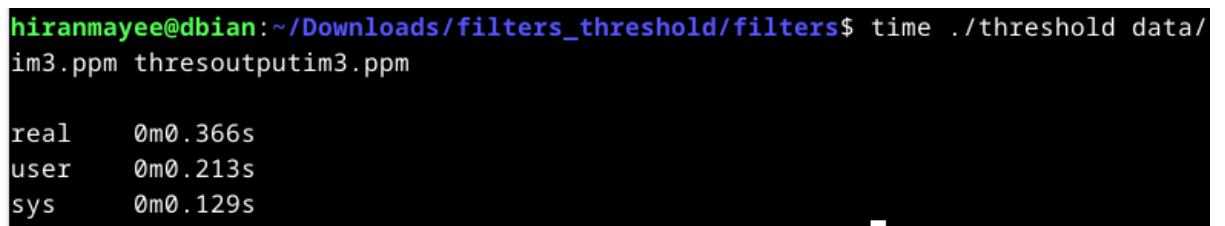
```

Figure 11: Before optimization writer::operator



```
155 void Writer::operator()(Matrix m, std::string filename)
156 {
157     try {
158         std::ofstream f {};
159
160         f.open(filename, std::ios::binary);
161
162         if (!f) {
163             throw std::runtime_error { "failed to open " + filename };
164         }
165
166         f << magic_number << "\n";
167
168         f << m.get_x_size() << " " << m.get_y_size() << "\n";
169         f << m.get_color_max() << "\n";
170
171         auto size { m.get_x_size() * m.get_y_size() };
172         auto R { m.get_R() }, G { m.get_G() }, B { m.get_B() };
173         auto it_R { R }, it_G { G }, it_B { B };
174
175         while (it_R < R + size && it_G < G + size && it_B < B + size) {
176             f << *it_R++
177               << *it_G++
178               << *it_B++;
179         }
180
181         f.close();
182     } catch (std::runtime_error e) {
183         error("writing", e.what());
184     }
```

Figure 12: Optimized Writer::operator.



```
hiranmayee@dbian:~/Downloads/filters_threshold/filters$ time ./threshold data/
im3.ppm thresoutputim3.ppm

real    0m0.366s
user    0m0.213s
sys     0m0.129s
```

Figure 13: Time execution improved: