High Performance Computing Systems
Project 1

Tsogkas Panagiotis Nikas Ioannis Iason Tsiantos Dimitrios

October 2025

Contents

1 Introduction **3**

2 Common Optimizations **4**
2.1 Loop Interchange 4
2.2 Loop Unrolling 5
2.3 Loop Fusion 7
2.4 Function Inlining 7
2.5 Common Sub-expression Elimination &Loop Invariant Code Motion 7
2.6 Strength Reduction 8
2.7 Referencing 10

3 Mathematical Optimizations **10**
3.1 Baseline 10
3.2 Fast Convolution 11
3.3 Fast PSNR 12
3.4 Fast Min Function 12
3.5 Fast Square Root 13

4 Results **16**

1 Introduction

Starting with an initial implementation of the Sobel operator in the C programming language, we attempt to adapt the code to the system's characteristics to achieve the lowest possible execution time. To accomplish this, we can apply various optimization methods that work both on general-purpose processors and specifically on our system's microarchitecture.

Name	12th Gen Intel® Core™ i5-1235U × 12
Cache	12 MB Intel® Smart Cache
RAM	16GB DDR4
OS	Ubuntu 22.04.5 LTS
Kernel	6.8.0-85-generic
Compiler	Intel(R) oneAPI DPC++/C++ Compiler 2025.2.1
Compiler Version	2025.2.0.20250806

Table 1: Useful system characteristics.

To test the algorithm, we use an input image of size 4096 × 4096 pixels. We execute the algorithm 12 times and exclude the fastest and slowest executions from the average. All executions are performed on one Performance-core of the processor (4.4GHz max).

Executable	Speedup	Average (s)	Std Dev (s)	Min / Max outliers
01_sobel_orig	1.00×	1.75385	0.04514	1.690210 / 1.916120

Table 2: Execution time at `-O0`

Executable	Speedup	Average (s)	Std Dev (s)	Min / Max outliers
01_sobel_orig	1.00×	0.46939	0.01264	0.450155 / 0.518506

Table 3: Execution time at `-fast`

2 Common Optimizations

2.1 Loop Interchange

We observe that the initial implementation traverses the elements of each column. This access method is very slow as it does not exploit the processor's prefetching mechanism. During each memory access, the processor brings an entire line (cache line) into the cache, assuming that related data are stored in consecutive memory locations.

By reading the first element of the first row and then the first element of the second row, both the first and second rows are loaded into the cache without being used before they are evicted.

[Image showing column-major vs row-major memory access]

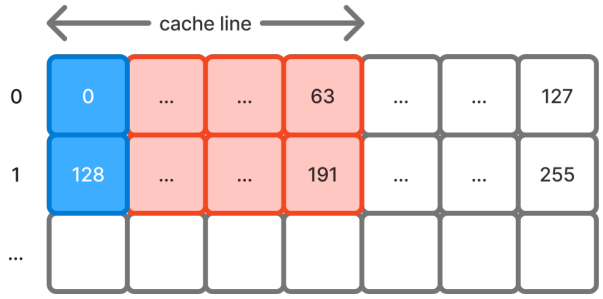


Figure 1: Accessing array elements by column. Red elements are placed in the cache but will not be accessed immediately by the processor.

The result is the creation of many cache misses and, consequently, stalls, significantly delaying program execution. By changing the access pattern to consecutive memory locations, execution time is significantly reduced as cache misses decrease.

```
/* For each pixel of the output image */
for (i=1; i<SIZE-1; i+=1 ) {
    for (j=1; j<SIZE-1; j+=1) {
        /* ... */
    }
}
```

Metric	Original	Interchange	Difference
Cache misses	57.7M	2.6M	22×
Cycles	7.4B	5.32B	39%
IPC (instr./cycle)	2.44	3.39	+0.95

Table 4: Comparison at `-O0`.

Metric	Original	Interchange	Difference
Cache misses	32.3M	2.65M	12.2×
Cycles	2.02B	0.854B	58%
IPC (instr./cycle)	0.76	1.80	+1.04

Table 5: Comparison at `-fast`.

Executable	Speedup	Average (s)	Std Dev (s)	Min / Max outliers
01_sobel_orig	1.00×	1.75385	0.04514	1.690210 / 1.916120
02_sobel_loop_interchange	1.44×	1.22102	0.01533	1.193020 / 1.259970

Table 6: Execution time at -O0

Executable	Speedup	Average (s)	Std Dev (s)	Min / Max outliers
01_sobel_orig	1.00×	0.46939	0.01264	0.450155 / 0.518506
02_sobel_loop_interchange	2.79×	0.16796	0.00047	0.167034 / 0.177655

Table 7: Execution time at -fast

2.2 Loop Unrolling

When we have a loop, its management adds overhead to the system, as a termination check condition and an index modification operation must be executed in each iteration. These operations can be reduced using unrolling.

```
for (i = 1; i < SIZE - 1; i++) {
    for (j = 1; j < SIZE - 1 - remainder; j += 4) {
        // pixel (i, j)
        p = ...;
        res = ...;
        output[i*SIZE + j] = ...;

        // pixel (i, j+1)
        p = ...;
        res = ...;
        output[i*SIZE + j+1] = ...;

        // pixel (i, j+2)
        p = ...;
        res = ...;
        output[i*SIZE + j+2] = ...;

        // pixel (i, j+3)
        p = ...;
        res = ...;
        output[i*SIZE + j+3] = ...;
    }
}
```

In this specific example, the unroll factor is 4. The number of branches and loop additions are quartered. From perf measurements, we observe that in the execution without compiler optimizations, the number of total branches decreased significantly. In the case of execution with -O3, the compiler optimizes sequential instructions within the loop, making execution even faster.

Metric	Interchange	Unrolled	Difference
Branch instructions	2B	987M	2.1×
Cycles	5.3B	4B	1.31×
IPC (instr./cycle)	3.41	3.1	-0.4

Table 8: Comparison at -O0.

Metric	Interchange	Unrolled	Difference
Branch instructions	42.5M	36.8M	1.16×
Cycles	854M	599M	1.43×
IPC (instr./cycle)	1.80	2.62	+0.82

Table 9: Comparison at `-fast`.

Executable	Speedup	Average (s)	Std Dev (s)	Min / Max outliers
02_sobel_loop_interchange	1.44×	1.22102	0.01533	1.193020 / 1.259970
03_sobel_loop_unrolling	1.94×	0.90287	0.00164	0.900786 / 0.942234

Table 10: Execution time at `-O0`

Executable	Speedup	Average (s)	Std Dev (s)	Min / Max outliers
02_sobel_loop_interchange	2.79×	0.16796	0.00047	0.167034 / 0.177655
03_sobel_loop_unrolling	4.25×	0.11050	0.00047	0.108389 / 0.112336

Table 11: Execution time at `-fast`

2.3 Loop Fusion

To calculate the PSNR of the result, the produced image and the correct (golden) image are traversed in a new loop. Instead of two double loops, we could incorporate all calculations into a single loop, reducing the total number of iterations and instructions.

```
for (i = 1; i < SIZE - 1; i++) {
    for (j = 1; j < SIZE - 1 - remainder; j += 4) {
        // pixel (i, j)
        p = ...
        res = ...
        output[i*SIZE + j] = ...
        t = pow((output[i*SIZE+j  ] - golden[i*SIZE+j  ]),2);
        PSNR += t;

        // pixel (i, j+1)
        p = ...
        res = ...
        output[i*SIZE + j+1] = ...
        t = pow((output[i*SIZE+j+1] - golden[i*SIZE+j+1]),2);
        PSNR += t;

        /* ... */
    }
}
```

Executable	Speedup	Average (s)	Std Dev (s)	Min / Max outliers
03_sobel_loop_unrolling	1.94×	0.90287	0.00164	0.900786 / 0.942234
04_sobel_loop_fusion	2.14×	0.82101	0.01090	0.810427 / 0.964667

Table 12: Execution time at `-O0`

Executable	Speedup	Average (s)	Std Dev (s)	Min / Max outliers
03_sobel_loop_unrolling	4.25×	0.11050	0.00047	0.108389 / 0.112336
04_sobel_loop_fusion	4.35×	0.10795	0.00071	0.107109 / 0.109979

Table 13: Execution time at `-fast`

2.4 Function Inlining

Calling functions in C is a process that has overhead from procedures such as stack allocation, argument storage, return address saving, local variables, their deletion at the end of the function, and the return mechanism. In our code, the `convolution2d` function is called repeatedly, making the call overhead noticeable. To solve this, we can change the function into a macro that the preprocessor will replace later where needed.

```
#define CONVOLUTION2D(posy, posx, input, operator)
```

2.5 Common Sub-expression Elimination & Loop Invariant Code Motion

Another important optimization technique is Common Subexpression Elimination, which aims to avoid repetitive calculations of the same mathematical expression within a loop or code segment.

Executable	Speedup	Average (s)	Std Dev (s)	Min / Max outliers
04_sobel_loop_fusion	2.14×	0.82101	0.01090	0.810427 / 0.964667
05_sobel_function_inlining	2.28×	0.76785	0.01184	0.753455 / 0.798266

Table 14: Execution time at -O0

Executable	Speedup	Average (s)	Std Dev (s)	Min / Max outliers
04_sobel_loop_fusion	4.35×	0.10795	0.00071	0.107109 / 0.109979
05_sobel_function_inlining	4.38×	0.10729	0.00080	0.106468 / 0.119504

Table 15: Execution time at -fast

A similar function is achieved by Loop Invariant Code Motion, where a piece of code that does not depend on loop variables is moved outside the loop and calculated once.

```
for (i = 1; i < SIZE - 1; i++) {
    /* Calculate expression once, refer to it by variable i_times_SIZE */
    i_times_SIZE = i * SIZE;
    for (j = 1; j < SIZE - 1 - remainder; j += unroll_factor) {
        // pixel (i, j)
        p = ...;
        res = ...;

        /* Refer by variable to avoid redundant additions*/
        i_times_SIZE_plus_j = i_times_SIZE + j;
        output[i_times_SIZE_plus_j] = ...;

        /* Store intermediate result in t to prevent recalculating */
        t = pow((output[i_times_SIZE_plus_j] - golden[i_times_SIZE_plus_j]), 2);
        PSNR += t;

        // pixel (i, j+1)
        ...

        /* ... */
    }
}
```

Executable	Speedup	Average (s)	Std Dev (s)	Min / Max outliers
05_sobel_function_inlining	2.28×	0.76785	0.01184	0.753455 / 0.798266
06_sobel_common_subexpression_elimination	1.95×	0.89913	0.01290	0.890251 / 0.958620

Table 16: Execution time at -O0

As we observe, the execution is slower compared to the previous version. Nevertheless, implementing these two optimizations creates the foundation for further performance improvement at a later stage of the implementation.

2.6 Strength Reduction

The use of slow operators can significantly burden program performance, especially when they are used inside the main loop. For this reason, replacing complex operations with simpler operators or more efficient implementations can noticeably reduce the program's computational load.

Executable	Speedup	Average (s)	Std Dev (s)	Min / Max outliers
05_sobel_function_inlining	4.38×	0.10729	0.00080	0.106468 / 0.119504
06_sobel_common_subexpression_elimination	4.12×	0.11386	0.00643	0.105915 / 0.162000

Table 17: Execution time at `-fast`

```
for (i = 1; i < SIZE - 1; i++) {
    for (j = 1; j < SIZE - 1 - remainder; j += unroll_factor) {
        // pixel (i, j)
        pixel_horizontal = ...;
        pixel_vertical = ...;
        /* Remove pow use multiplication instead */
        res = sqrt(pixel_horizontal * pixel_horizontal + pixel_vertical * pixel_vertical);
        i_times_SIZE_plus_j = ...;
        output[i_times_SIZE_plus_j] = ...;
        t = ...;
        PSNR += ...;

        // pixel (i, j+1)
        /* ... */
    }
    /* Eliminate multiplication use addition instead */
    i_times_SIZE += SIZE;
}
```

Executable	Speedup	Average (s)	Std Dev (s)	Min / Max outliers
06_sobel_common_subexpression_elimination	1.95×	0.89913	0.01290	0.890251 / 0.958620
07_sobel_strength_reduction	8.30×	0.21133	0.00381	0.208410 / 0.226750

Table 18: Execution time at `-O0`

Executable	Speedup	Average (s)	Std Dev (s)	Min / Max outliers
06_sobel_common_subexpression_elimination	4.12×	0.11386	0.00643	0.105915 / 0.162000
07_sobel_strength_reduction	5.02×	0.09359	0.00079	0.092326 / 0.095376

Table 19: Execution time at `-fast`

2.7 Referencing

Referencing is used to avoid redundant pointer operations with constant values, which can be avoided by redefining the pointer with the appropriate offset. Additionally, this technique facilitates the use of pointer arithmetic; however, it was observed to have a negative impact on execution time as it did not allow the use of the `const` keyword. For this reason, this specific approach was not included in the final optimizations.

```
for (i = 1; i < SIZE - 1; i++) {
    /* Referencing array offset address */
    const unsigned char *top_row = &input[dec_i_times_SIZE];
    const unsigned char *bottom_row = &input[inc_i_times_SIZE];
    const unsigned char *mid_row = &input[i_times_SIZE];
    unsigned char *out_row = &output[i_times_SIZE];
    unsigned char *gold_row = &golden[i_times_SIZE];
    for (j = 1; j < SIZE - 1 - remainder; j += unroll_factor) {
        /* Refer to convolution inputs by pointers avoid redundant operations */
        pixel_horizontal = CONVOLUTION2D(j, top_row, mid_row, bottom_row, horiz_operator);
        pixel_vertical = CONVOLUTION2D(j, top_row, mid_row, bottom_row, vert_operator);
        res = ...;
        out_row[j] = ...;
        t = (out_row[j] - gold_row[j]);
        PSNR += t * t;

        /* ... */
    }
    dec_i_times_SIZE += SIZE;
    inc_i_times_SIZE += SIZE;
    i_times_SIZE += SIZE;
}
```

3 Mathematical Optimizations

3.1 Baseline

As a base version, we selected the fastest single-core implementation with infinite PSNR, but we removed the loop fusion step, as we noticed that the program was particularly memory-bound in later versions of the code. This was due to increased performance from computational optimizations and alternating accesses to the `golden` and `output` arrays. An additional choice was the removal of loop unrolling, which allowed for easier isolation of future issues using V-Tune, generally facilitating analysis and specifically the analysis of memory behavior. The removal of these two optimizations later led to better execution times compared to other base versions.

3.2 Fast Convolution

The convolution operation is one of the most computationally demanding operations. However, by exploiting the properties of this specific kernel, it is possible to simplify the convolution through appropriate mathematical transformations, thus reducing the total number of operations and improving performance.

Furthermore, the order of execution of operations is of particular importance, as organizing the calculations in a way that follows a predictable memory access pattern contributes to improving cache locality. In this way, all necessary operations are performed per row of the kernel, minimizing cache misses and evictions of useful data, leading to faster execution.

```
for (i = 1; i < SIZE - 1; i++) {
    const unsigned char *top_row = &input[dec_i_times_SIZE];
    const unsigned char *bot_row = &input[inc_i_times_SIZE];
    const unsigned char *mid_row = &input[i_times_SIZE];
    unsigned char *out_row = &output[i_times_SIZE];

    for (j = 1; j < SIZE - 1; j++) {
        /* Convolution computations row by row */
        pixel_horizontal = -top_row[j - 1] + top_row[j + 1];
        pixel_vertical = top_row[j - 1] + (top_row[j] << 1) + top_row[j + 1];
        pixel_horizontal += -(mid_row[j - 1] << 1) + (mid_row[j + 1] << 1) -
                            bot_row[j - 1] + bot_row[j + 1];
        pixel_vertical += -bot_row[j - 1] - (bot_row[j] << 1) -
                          bot_row[j + 1];

        /* ... */
    }
    dec_i_times_SIZE += SIZE;
    inc_i_times_SIZE += SIZE;
    i_times_SIZE += SIZE;
}
```

This stems from:

$$H = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}, \quad V = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

$$\begin{aligned} \text{pixel_horizontal} = {} & (-1) \cdot \text{top_row}[j-1] + 0 \cdot \text{top_row}[j] + 1 \cdot \text{top_row}[j+1] \\ & + (-2) \cdot \text{mid_row}[j-1] + 0 \cdot \text{mid_row}[j] + 2 \cdot \text{mid_row}[j+1] \\ & + (-1) \cdot \text{bot_row}[j-1] + 0 \cdot \text{bot_row}[j] + 1 \cdot \text{bot_row}[j+1] \end{aligned}$$

Removing the terms multiplied by zero:

$$\text{pixel_horizontal} = -\text{top_row}[j-1] + \text{top_row}[j+1] - 2 \cdot \text{mid_row}[j-1] + 2 \cdot \text{mid_row}[j+1] - \text{bot_row}[j-1] + \text{bot_row}[j+1]$$

In the same way, we derive `pixel_vertical`.

Executable	Speedup	Average (s)	Std Dev (s)	Min / Max outliers
09_sobel_no_unroll_no_fusion	12.08×	0.14522	0.00156	0.143731 / 0.151505
10_sobel_fast_convolution	14.53×	0.12072	0.00102	0.119427 / 0.126503

Table 20: Execution time at -O0

Executable	Speedup	Average (s)	Std Dev (s)	Min / Max outliers
09_sobel_no_unroll_no_fusion	6.11×	0.07683	0.00069	0.075577 / 0.079137
10_sobel_fast_convolution	17.30×	0.02714	0.00057	0.026333 / 0.028785

Table 21: Execution time at `-fast`

3.3 Fast PSNR

If m and n are the image dimensions, I the algorithm's result, and K the ideal image, then the MSE is calculated as:

$$MSE = \frac{1}{mn} \sum_{i=0}^{m-1} \sum_{j=0}^{n-1} [I_{i,j} - K_{i,j}]^2$$

To perform this calculation, we would need to execute the operation $[I_{i,j} - K_{i,j}]^2$ either in the main loop or a second loop. If this operation occurs in the main loop, we would be bringing elements of array K into the cache, filling it, whereas if it occurs in a separate loop, there would be significant overhead. To avoid this, we could transform the MSE formula to access only array I in the main loop.

$$MSE = \frac{1}{mn} \sum_{i=0}^{m-1} \sum_{j=0}^{n-1} [I_{i,j}^2 - 2I_{i,j}K_{i,j} + K_{i,j}^2] \;=\; \frac{1}{mn}(I^2 - 2IK + K^2)$$

Now we can calculate the I^2 term in one variable and then calculate the terms $-2IK + K^2$ in a second loop.

Executable	Speedup	Average (s)	Std Dev (s)	Min / Max outliers
10_sobel_fast_convolution	14.53×	0.12072	0.00102	0.119427 / 0.126503
11_sobel_fast_psnr	16.30×	0.10762	0.00150	0.104974 / 0.111273

Table 22: Execution time at `-O0`

Executable	Speedup	Average (s)	Std Dev (s)	Min / Max outliers
10_sobel_fast_convolution	17.30×	0.02714	0.00057	0.026333 / 0.028785
11_sobel_fast_psnr	13.18×	0.03562	0.00044	0.034291 / 0.037935

Table 23: Execution time at `-fast`

3.4 Fast Min Function

Inside the main loop, the algorithm takes the square root result and, if it is greater than 255, sets it to 255; otherwise, it keeps the square root return value. This is the logic of the `min` function, as it finds the minimum between 255 and the square root return.

$$output_{i,j} = min(255,\ (int)sqrt(pixel))$$

The initial implementation of `min` uses an `if-else` statement, increasing branch instructions in the program. To avoid this, we used a version of the `min` function that has no branch instructions.

$$diff = pixel - 255$$
$$output_{i,j} = 255 + diff\ \&\ (diff >> 31)$$

In this function, when the pixel is less than 255, the `diff` term is negative, and then when shifted right by 31 bits, the mask becomes `0xFFFFFF` due to sign extension. Thus, when we subtract the `diff` from 255, we result in the original pixel value.

$$diff = x - 255, \quad x < 255$$
$$output_{i,j} = 255 + diff \ \& \ 0xFFFFFF$$

From the operation $diff \ \& \ 0xFFFFFF$ we get $diff$, so

$$output_{i,j} = 255 + diff \ = \ 255 + pixel - 255 \ = \ pixel$$

Conversely, when the pixel is greater than 255, then `diff` is positive and the resulting mask is 0, so only the 255 term remains in the result.

$$diff = x - 255, \quad x \geq 255$$
$$output_{i,j} = 255 + diff \ \& \ 0 \ = \ 255$$

Executable	Speedup	Average (s)	Std Dev (s)	Min / Max outliers
11_sobel_fast_psnr	16.30×	0.10762	0.00150	0.104974 / 0.111273
12_sobel_fast_min	16.41×	0.10687	0.00084	0.105859 / 0.109175

Table 24: Execution time at `-O0`

Executable	Speedup	Average (s)	Std Dev (s)	Min / Max outliers
11_sobel_fast_psnr	13.18×	0.03562	0.00044	0.034291 / 0.037935
12_sobel_fast_min	11.69×	0.04017	0.00153	0.038610 / 0.044180

Table 25: Execution time at `-fast`

3.5 Fast Square Root

The square root operation is a complex and slow arithmetic operation. Based on the algorithm for approximating the fast inverse square root, we created a corresponding version for calculating the square root.

We consider a floating-point number x with mantissa M and exponent E. We will now find the binary logarithm of x.

Now we must use an approximation of the function $\log_2(1+x) \approx x + \mu$, which is where the error in the square root calculation comes from. The choice of μ is arbitrary, and in our case, we chose $\mu = 0.043$ to have the smallest error. Continuing, we have:

$$(1) \implies \frac{M}{2^{23}} + E - 127 + \mu = \frac{1}{2^{23}}\left(M + 2^{23} \cdot E\right) + \mu - 127$$

$$\log_2 x = \log_2\left[\left(1 + \frac{M}{2^{23}}\right) \cdot 2^{E-127}\right] = \log_2\left(1 + \frac{M}{2^{23}}\right) + E - 127 \tag{1}$$

Thus, we have arrived at a relationship that equates the number x with its representation as a floating-point number $x_f = M + 2^{23} \cdot E$.

$$\log_2 x = \frac{1}{2^{23}} x_f + \mu - 127 \tag{2}$$

We consider a number y for which $y = \sqrt{x}$ holds. Applying the logarithm, we have:

$$y = \sqrt{x} \implies \log_2 y = \frac{1}{2}\log_2 x \overset{(2)}{\implies}$$

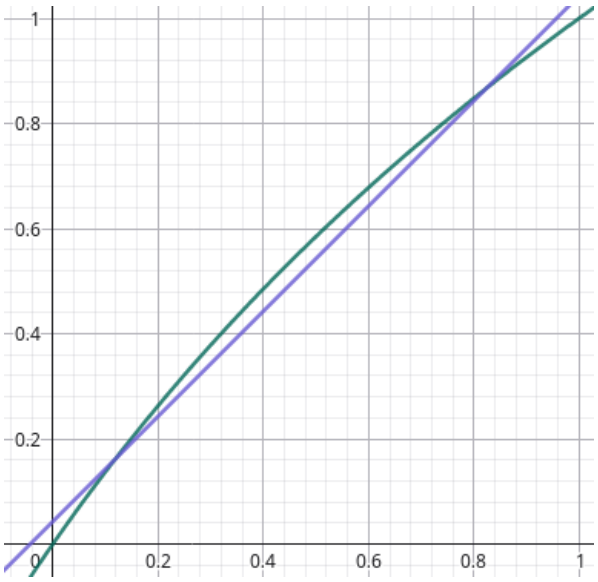


Figure 2: Plot of $y = x + \mu, \quad \mu = 0.043$ and $y = log_2(1 + x)$

$$\frac{1}{2^{23}} y_f + \mu - 127 = \frac{1}{2^{24}} x_f + \frac{1}{2} \cdot (\mu - 127) \implies y_f = \frac{1}{2} x_f - 2^{22} \cdot (\mu - 127)$$

Therefore, by dividing the number x_f in its form as a floating-point number by 2 and adding a constant, we get a good estimate of the square root of x. Its implementation is done by typecasting the floating-point number to an integer without performing an integer conversion; that is, we simply keep the bits of the number. The number `0x1FBD3F7D` results from the constant $-2^{22} \cdot (\mu - 127) = 532496252.928 \approx 532496253$.

```
float fast_sqrt(float a) {
    int i = * ( int * ) &a;
    i = (i >> 1) + 0x1FBD3F7D;
    return * ( float * ) &i;
}
```

In the version for the inverse square root, an iteration of the Newton-Raphson method is applied, which in our case requires division, so we decided not to apply it. This leads to larger errors which, however, remain relatively small for the range of numbers we are dealing with, as shown below.

We observe that the error remains below 5%, which leads to relatively accurate results with a PSNR close to 50.

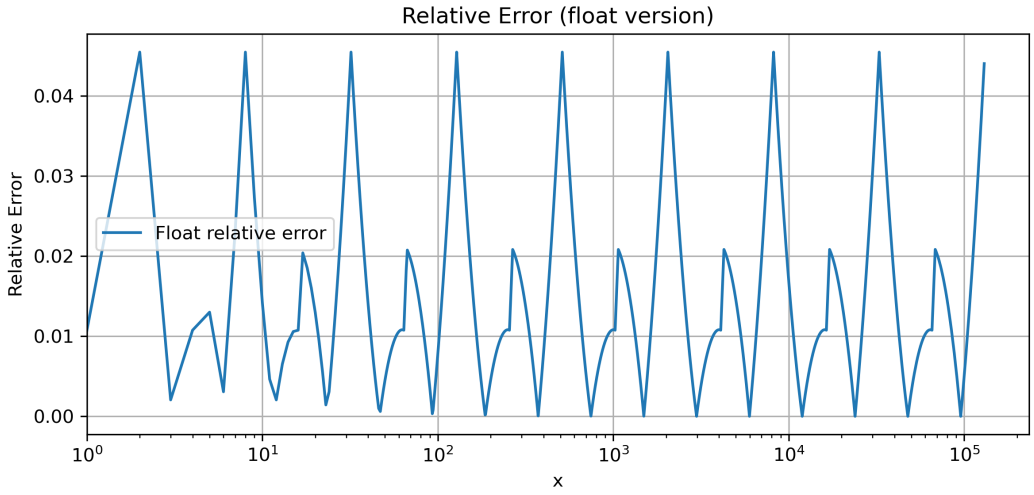


Figure 3: Relative error for numbers from 1 to 10^5

Executable	Speedup	Average (s)	Std Dev (s)	Min / Max outliers
12_sobel_fast_min	16.41×	0.10687	0.00084	0.105859 / 0.109175
13_sobel_fast_sqrt	18.56×	0.09451	0.00074	0.093438 / 0.096139
14_sobel_fast_conv_sqrt	17.26×	0.10160	0.00156	0.099090 / 0.114298

Table 26: Execution time at `-O0`

Executable	Speedup	Average (s)	Std Dev (s)	Min / Max outliers
12_sobel_fast_min	11.69×	0.04017	0.00153	0.038610 / 0.044180
13_sobel_fast_sqrt	13.55×	0.03465	0.00113	0.032232 / 0.036176
14_sobel_fast_conv_sqrt	21.49×	0.02184	0.00074	0.020820 / 0.022988

Table 27: Execution time at `-fast`

4 Results

Executable	Speedup	Average (s)	Std Dev (s)	Min / Max outliers
01_sobel_orig	1.00×	1.75385	0.04514	1.690210 / 1.916120
02_sobel_loop_interchange	1.44×	1.22102	0.01533	1.193020 / 1.259970
03_sobel_loop_unrolling	1.94×	0.90287	0.00164	0.900786 / 0.942234
04_sobel_loop_fusion	2.14×	0.82101	0.01090	0.810427 / 0.964667
05_sobel_function_inlining	2.28×	0.76785	0.01184	0.753455 / 0.798266
06_sobel_common_subexpression_elimination	1.95×	0.89913	0.01290	0.890251 / 0.958620
07_sobel_strength_reduction	8.30×	0.21133	0.00381	0.208410 / 0.226750
08_sobel_referencing	8.33×	0.21048	0.00281	0.208129 / 0.418608
09_sobel_no_unroll_no_fusion	12.08×	0.14522	0.00156	0.143731 / 0.151505
10_sobel_fast_convolution	14.53×	0.12072	0.00102	0.119427 / 0.126503
11_sobel_fast_psnr	16.30×	0.10762	0.00150	0.104974 / 0.111273
12_sobel_fast_min	16.41×	0.10687	0.00084	0.105859 / 0.109175
13_sobel_fast_sqrt	18.56×	0.09451	0.00074	0.093438 / 0.096139
14_sobel_fast_conv_sqrt	17.26×	0.10160	0.00156	0.099090 / 0.114298

Table 28: Execution time at `-O0`

Executable	Speedup	Average (s)	Std Dev (s)	Min / Max outliers
01_sobel_orig	1.00×	0.46939	0.01264	0.450155 / 0.518506
02_sobel_loop_interchange	2.79×	0.16796	0.00047	0.167034 / 0.177655
03_sobel_loop_unrolling	4.25×	0.11050	0.00047	0.108389 / 0.112336
04_sobel_loop_fusion	4.35×	0.10795	0.00071	0.107109 / 0.109979
05_sobel_function_inlining	4.38×	0.10729	0.00080	0.106468 / 0.119504
06_sobel_common_subexpression_elimination	4.12×	0.11386	0.00643	0.105915 / 0.162000
07_sobel_strength_reduction	5.02×	0.09359	0.00079	0.092326 / 0.095376
08_sobel_referencing	5.05×	0.09298	0.00166	0.091319 / 0.113244
09_sobel_no_unroll_no_fusion	6.11×	0.07683	0.00069	0.075577 / 0.079137
10_sobel_fast_convolution	17.30×	0.02714	0.00057	0.026333 / 0.028785
11_sobel_fast_psnr	13.18×	0.03562	0.00044	0.034291 / 0.037935
12_sobel_fast_min	11.69×	0.04017	0.00153	0.038610 / 0.044180
13_sobel_fast_sqrt	13.55×	0.03465	0.00113	0.032232 / 0.036176
14_sobel_fast_conv_sqrt	21.49×	0.02184	0.00074	0.020820 / 0.022988

Table 29: Execution time at `-fast`

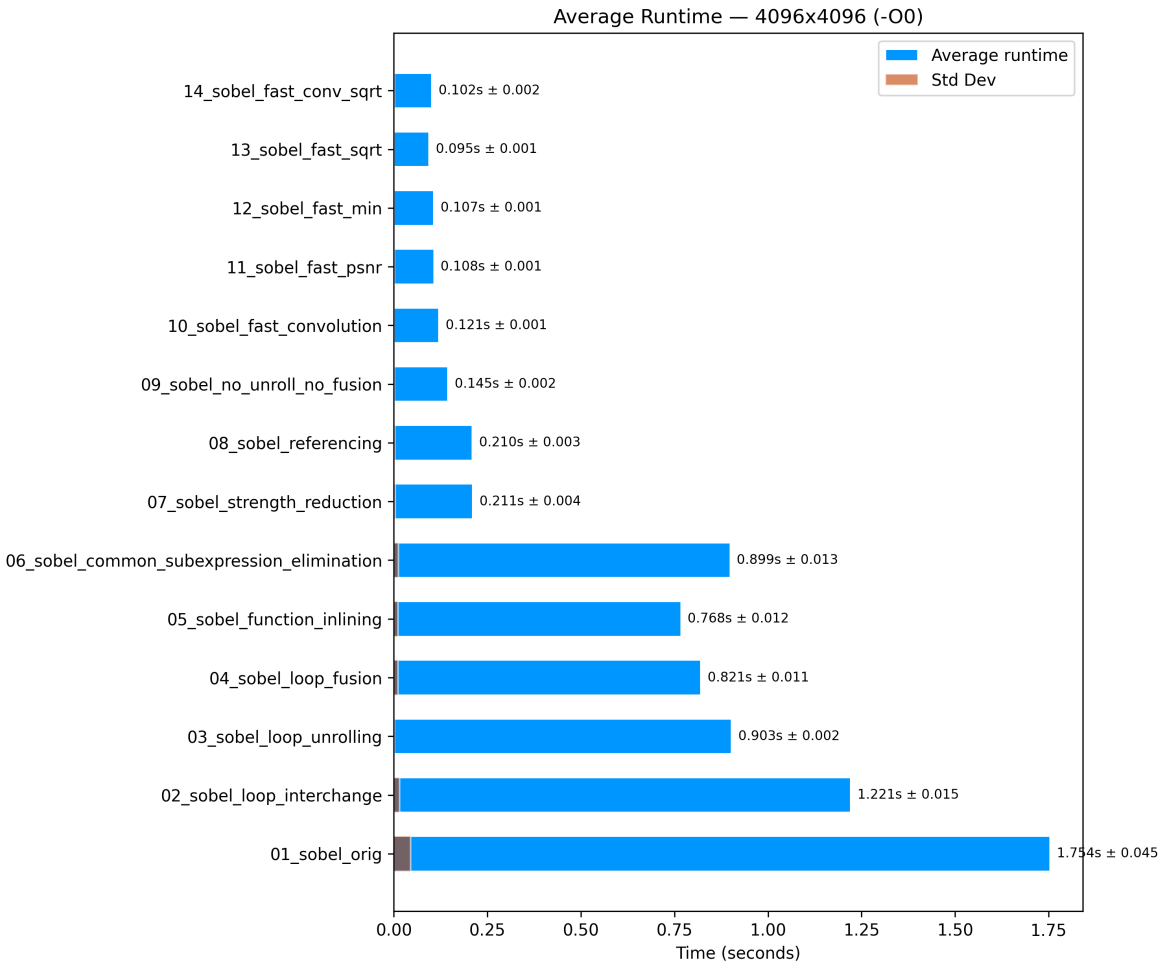


Figure 4: All executions at -O0

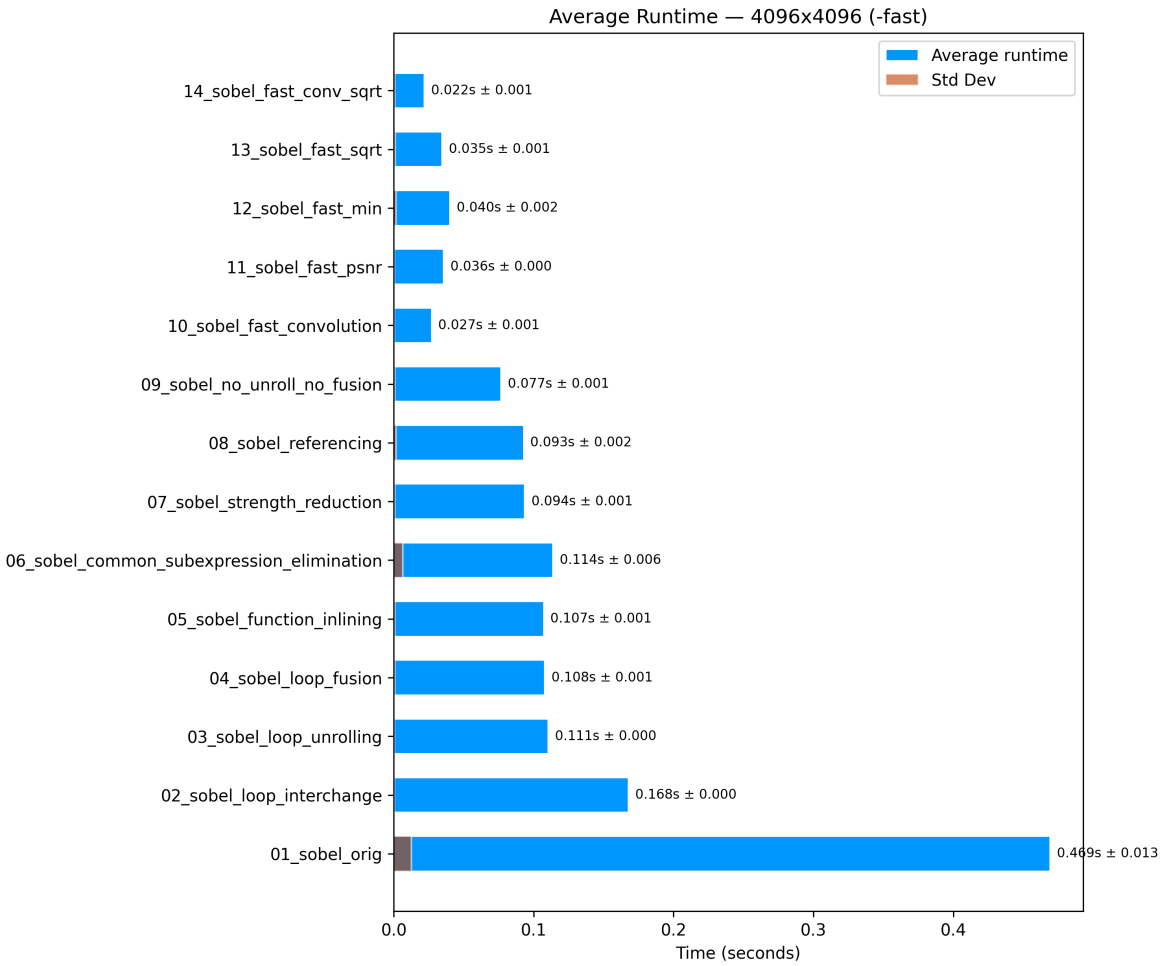


Figure 5: All executions at `-fast`