university of groningen

faculty of science and engineering

Efficient implementation of Non Local Means Algorithm

Bachelor Thesis

August 27, 2023
University of Groningen

Author:
Bob van der Vuurst

Primary supervisor:
Dr. A. Shahbahrami

Secondary supervisor:
Prof. dr. G.N. Gaydadjiev

Abstract

Image denoising is a widely used technique to improve image quality. In 2005 Buades et al. developed the non local means (NLM) algorithm which uses redundancy of similar patterns in images to average these patterns out and thus effectively reduce noise. This NLM algorithm preserves details of images and provides better denoising than classical denoising algorithms, such as Gaussian smoothing. A drawback of NLM is that it has a relatively large computational complexity, which result in large computational times. In 2018 Yamanappa et al. proposed a variation on NLM that uses the Shapiro-Wilk test to determine the similarity of patterns, which provides better denoising quality at the cost of an even larger computational complexity.

To reduce the computational time of the algorithm, in this bachelor project I propose a parallel version for GPUs implemented in CUDA for both the conventional NLM (CNLM) and Shapiro-Wilk NLM (SWNLM) algorithms. These CUDA versions perform well with achieved speedups of up to 45 and 145 for CNLM and SWNLM respectively, with virtually identical denoising quality.

Contents

1 Introduction

Noise in images is the random variations of intensity for each pixel, which lowers the visual quality of the image and makes image processing more difficult. The devices that create images, for example cameras and medical scanners, have imperfections, which make the presence of noise unavoidable. For simplicity, in this thesis I assume that the noise follows a Gaussian distribution with a mean of 0. Image denoising is the removal of this noise to increase the visual quality of the image and is an important part of preprocessing for various applications, such as AI pattern recognition and classification tasks. The goal of image denoising is to remove all noise, but preserve all the edges and small details in an image.

Several classical image denoising methods exist, such as Gaussian filtering [6], anisotropic diffusion [9] and wavelet transform denoising [1]. However, these denoising methods blur the image and thus fail to preserve the edges and details, which lowers its visual quality.

Buades et al. proposed a non local means (NLM) denoising method, which makes use of the redundancy of similar patches present in natural images [2]. For each pixel, the patch around the pixel is compared to the patches around other pixels and a similarity value is calculated using the Euclidean distance, where a higher similarity value indicates more similarity. These similarities are then used as weights for the pixels, where the denoised value is a weighted average of these pixels. Image denoising using NLM provides better visual quality than classical denoising methods and does not blur the image.

Several variations of the NLM algorithm have been proposed, which differ in the way the similarity measure is calculated. These variations aim to reduce the execution time or improve the denoising performance of the NLM algorithm. Yamanappa et al. proposed a variation on the NLM algorithm which uses the Shapiro-Wilk test to calculate the similarity measure [15]. The Shapiro-Wilk test is a statistical test to determine if a set of samples follow the Gaussian distribution and thus can be used to detect Gaussian noise. The Shapiro-Wilk NLM (SWNLM) algorithm provides better denoising quality than the original NLM, though the execution time is slower due to its larger computational complexity.

In this thesis I will research and develop an efficient implementation of both the original NLM and the SWNLM algorithms for Graphics Processing Units (GPUs) using the CUDA framework. These implementations are then compared against the sequential versions of the algorithms in both denoising quality and execution time.

The thesis is organized as follows: The workings of NLM and SWNLM and other background information are discussed in section 2, the methods are explained in section 3 and the results are presented in section 4. The conclusion and potential improvements are discussed in sections 5 and 6.

2 Related Work

Over the years many variants of the NLM algorithm have been proposed to improve both the denoising quality and the speed of the algorithm. In this chapter I will first explain the workings of NLM in detail and then discuss the SW-test, SWNLM and how it differs from NLM.

2.1 Gaussian filter

The Gaussian filter is a widely used tool in the image processing field for smoothing images [6]. It is based on the Gaussian function, which is defined in equation (1), where $a, b, c \in \mathbb{N}$ and $c \neq 0$.

$$f(x) = ae^{-\frac{(x-b)^2}{2c^2}} \tag{1}$$

When plotted, the Gaussian function produces a bell curve known as the Gaussian distribution or normal distribution, as can be seen in figure 1. a controls the amplitude of the curve, b its center and c its standard deviation. The Gaussian function is important in image processing because most cameras and medical scanners produce images with some noise that fits the Gaussian distribution. This noise is called Gaussian additive noise. For image denoising purposes the center is typically at 0 and the area of the curve is normalized to 1, which means that the amplitude is factored out. Thus, we can set $a = 1$ and $b = 0$ and simplify the equation with the typical notation $\sigma = c$.

$$f(x) = e^{-\frac{x^2}{2\sigma^2}} \tag{2}$$

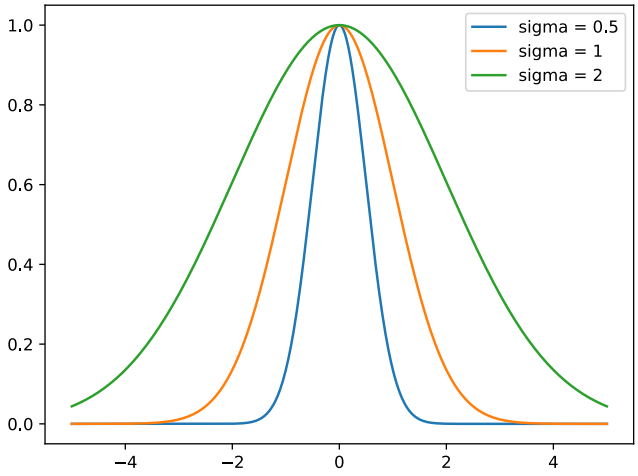
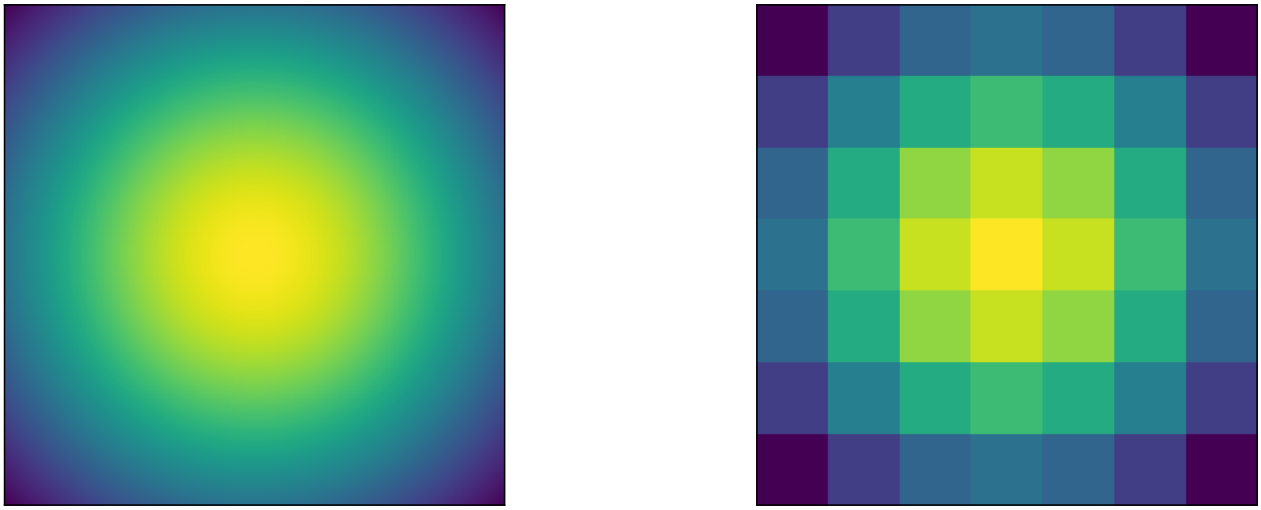


Figure 1: Gaussian function plotted with different values of σ.

For two dimensions, the Gaussian function is modified to use the squared euclidean distance, as is shown in equation 3. This equation is plotted in figure 2a. Typically in Gaussian filtering a discrete version is used called the Gaussian kernel, shown in figure 2b.

$$f(x, y) = e^{-\frac{x^2+y^2}{2\sigma^2}} \tag{3}$$



(a) (b)

Figure 2: 2D Gaussian function with $\sigma = 3$ plotted with $x, y \in [-3, 3]$. (a) is plotted as a continuous function, (b) is plotted as a discrete Gaussian kernel with a radius of 3.

Gaussian filtering works as follows: The Gaussian kernel is passed over every pixel i in the image with the center of the kernel at i. The neighborhood $\mathcal{N}_i$, which is the square of pixels around i with a given radius r, is then multiplied element-wise with the Gaussian kernel G. The resulting matrix $\mathcal{N}_i \odot G$ is summed up to get the new pixel value $d[i]$. In short, $d[i]$ is a weighted average of $\mathcal{N}_i$ where the weights are determined in G. The calculation for a pixel using Gaussian filtering is defined in equation 4.

$$d[i] = \sum \mathcal{N}_i \odot G \tag{4}$$

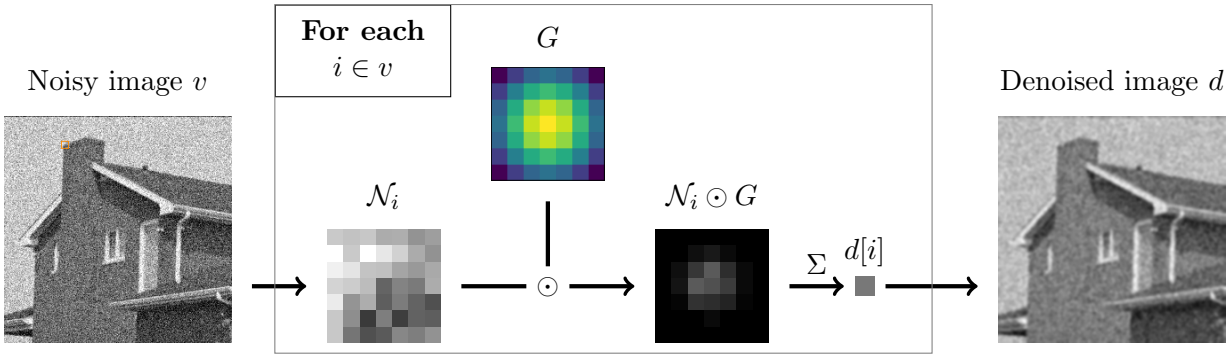


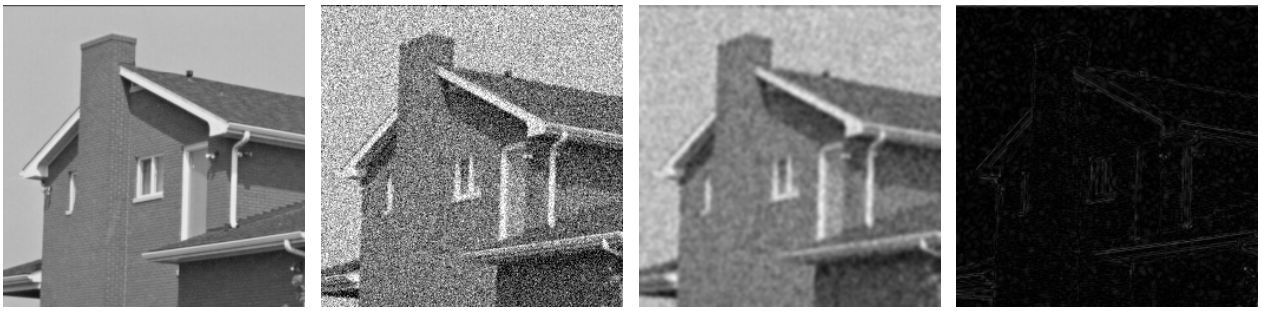
Figure 3: Example of the workings of the Gaussian filter. The neighborhood for the displayed example i is shown in the orange square on the left. The pixel values in $\mathcal{N}_i \odot G$ are multiplied by 5 for visual clarity.

Figure 4 shows an example of Gaussian filtering applied to the house image, where 4c is the denoised image. Though some noise has been removed, there is still a subjectively large amount left. Figure 4d shows the difference between the original and the denoised image. Ideally this figure should be black, indicating that there is no difference between the original and denoised image and thus all noise has been removed. The outline of the house can clearly be seen in the figure, which means that

Algorithm 1 The Gaussian filter algorithm.

Initialize output image D with 0's and same shape as input image I.
Initialize Gaussian kernel G with radius r, center $(0, 0)$ and standard deviation $\sigma \times k$.
Normalize G such that $sum(G) = 1$.
for all $I[x][y] \in I$ **do**
    for all $(u, v) \in [-r..r]^2$ **do**
        $D[x][y] = D[x][y] + G[u][v] \times I[x + u][y + v]$
    end for
end for
return D

these edges from the original image have been lost. These are the two major limitations of Gaussian filtering: The denoised images still have a noticeable amount of noise and the edges are blurred.



(a) Original House image (b) Noisy image (c) Denoised with Gaussian blur (d) Difference between original and denoised image

Figure 4: Gaussian filtering applied to the House image with $\sigma = 40$ and $k = 10$.

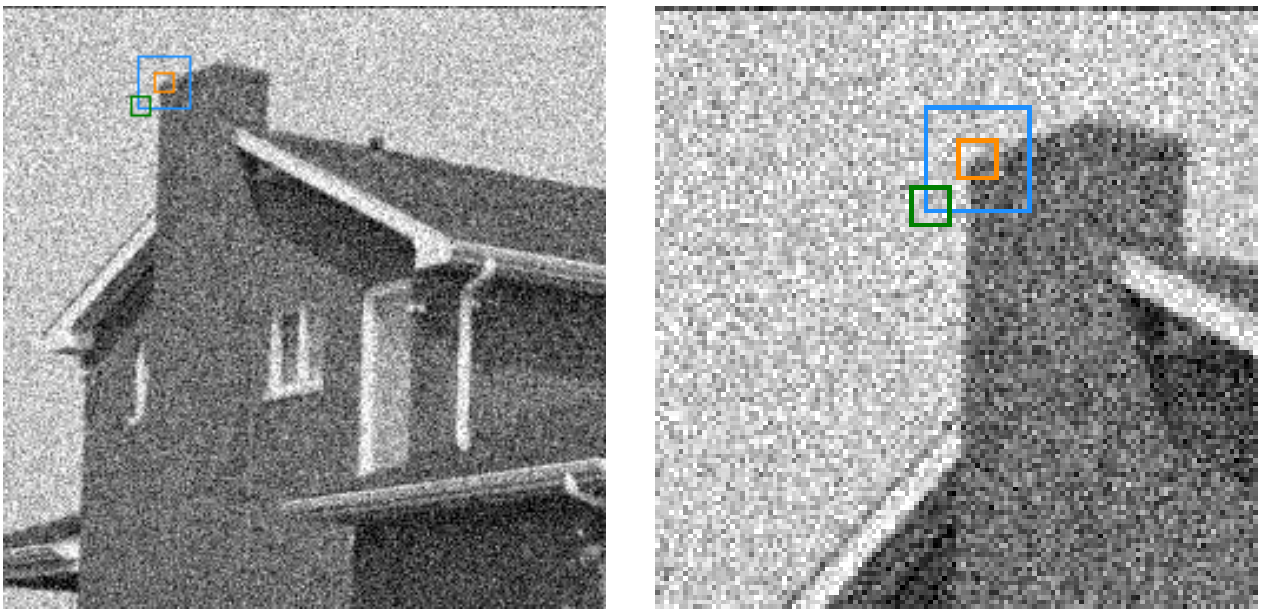
2.2 Non Local Means

The conventional NLM (CNLM) algorithm proposed by Buades et al. works as follows [2]: For each pixel i in a noisy image v, the denoised value $NL[v](i)$ is a weighted average of all other pixels in the image:

$$NL[v](i) = \sum_{j \in v} w(i, j)v(j) \tag{5}$$

The weights are given by the similarity between the pixels i and j, which is determined using the neighborhoods of the pixels. The neighborhood $\mathcal{N}_i$ of a pixel i is defined as all the pixels surrounding the pixel in a square area with a given radius $r_{\mathcal{N}}$. In the original paper, Buades et al. defined a radius of 3, which gives a 7×7 square matrix neighborhood $\mathcal{N}_i$ centered at each pixel i. The weight $w(i, j)$ is determined by first computing the element-wise squared difference between $\mathcal{N}_i$ and $\mathcal{N}_j$. Then a Gaussian kernel is used to compute $w(i, j)$. Equation (6) shows the calculation of $w(i, j)$, where $Z(i)$ is the sum of all the weights of i.

$$w(i, j) = \frac{1}{Z(i)} e^{\frac{||v(\mathcal{N}_i) - v(\mathcal{N}_j)||}{h^2}} \tag{6}$$



(a) Original size. (b) Cropped.

Figure 5: Example of a neighborhood $\mathcal{N}_i$, the search window S_i and a neighborhood $\mathcal{N}_j$ in that search window. $\mathcal{N}_i$ is displayed in orange, S_i in blue and $\mathcal{N}_j$ in green.

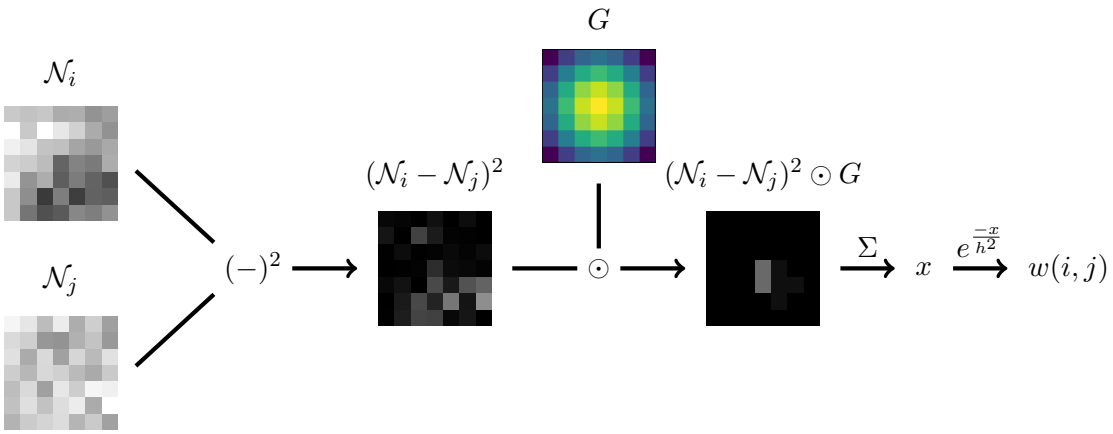


Figure 6: Overview of the weight computation $w(i,j)$ for CNLM. The neighborhoods are outlined in figure 5. The pixel values in $(\mathcal{N}_i - \mathcal{N}_j)^2 \odot G$ are multiplied by 15 for visual clarity.

A downside of CNLM is that it is computationally expensive compared to other denoising algorithms such as the Gaussian filter. This is mainly because for each pixel i in the image its neighborhood $\mathcal{N}_i$ has to be compared with every other neighborhood $\mathcal{N}_j$. Given a neighborhood size 7×7 and number of pixels N, this gives a total computational complexity of $\mathcal{O}(N^2 \times 49)$. Buades et al. mentioned this and proposed to use a search window S_i centered at i, where only the pixels j inside S_i are used for calculating the weighted average in equation 5. With a search radius r_S of 10, this gives a square 21×21 search window and thus $21^2 = 441$ neighborhood comparisons for each pixel in the image. This gives a final complexity of $\mathcal{O}(N \times 441 \times 49)$. For comparison, Gaussian filtering has a computational complexity of $\mathcal{O}(N \times 49)$ with $r_{\mathcal{N}} = 3$. Thus, CNLM is over two orders of magnitude slower than Gaussian filtering, which can become a problem when denoising large images with millions of pixels.

Algorithm 2 The basis of the CNLM algorithm.

```
Initialize output image D with 0's and same shape as input image I
for all I[x][y] ∈ I do
    Z ← 0
    for all I[u][v] ∈ searchWindow(I[x][y]) do
        Calculate neighborhood similarity of (I[x][y], I[u][v])
        Z ← Z + similarity
        D[x][y] ← D[x][y] + similarity * p_{x,y}
    end for
    D[x][y] ← D[x][y]/Z
end for
return D
```

2.3 SWNLM

The SWNLM algorithm by Yamanappa et al. is a variation of the NLM algorithm, where the weights are calculated using the Shapiro-Wilk test instead of the difference between two neighborhoods [15]. I will first explain what the Shapiro-Wilk test is, then I will explain how it is used to improve the NLM algorithm.

2.3.1 Shapiro-Wilk test

The Shapiro-Wilk (SW) test was proposed by Shapiro and Wilk in 1965 and tests whether a given collection of samples X is normally distributed [11]. This is done by first sorting X and then comparing each sample of X with an expected value from the normal distribution. The SW test then returns two values: W and Wp. Wp can be used to determine if X is normally distributed using a t test. A t test is a widely used statistical test that can be used for testing a null hypothesis H_0 and if it can be rejected. If t is below some threshold α (usually set to 0.05), then we reject the null hypothesis. Otherwise if $t \geq \alpha$ we fail to reject the null hypothesis. Note that we never explicitly accept the null hypothesis, because it could still be rejected if X had more samples.

In our case for the SW test, we have the hypothesis H_0: X is normally distributed. Thus, X is not normally distributed for $t < \alpha$ and X is statistically likely to be normally distributed for $t \geq \alpha$. If we fail to reject the null hypothesis, the value W specifies how normally distributed X is, where $W = 0$ means that X is not normally distributed at all and $W = 1$ means that X follows the normal distribution perfectly.

Unfortunately the SW test is arguably quite complicated, so I will not explain how it works here. A full explanation can be found in the original paper [11]. However, there are a few important limitations that should be mentioned: To begin with, the SW test only tests one tail of the normal distribution, i.e. the distribution of X is assumed to be symmetrical. Thus, if X is heavily skewed it might still pass the SW test, even though it does not follow the normal distribution. Secondly, the SW test uses an approximation of the normal distribution, which is not suitable for sample sizes higher than 5000 due to lack of precision [10].

The SW test requires that X is sorted first, and then all the elements are accessed once to calculate W. Given that most sorting algorithms have computational complexity $\mathcal{O}(n \log n)$ and X has n elements, the computational complexity of the SW test becomes $\mathcal{O}(n \log n + n) \approx \mathcal{O}(n \log n)$.

2.3.2 SWNLM

SWNLM has the same overall structure as the NLM algorithm and only differs in the weight calculation. Instead of calculating the weights based on the differences between $\mathcal{N}_i$ and each $\mathcal{N}_j$ directly, SWNLM applies the SW test on $F_{i,j} = (\mathcal{N}_i - \mathcal{N}_j)/(2\sigma)$ and the weight is determined by the outcome of the test [15]. If it passes the test, then w becomes the weight of the pixel j for i. If it does not pass the test, then the weight of j for i is 0. The reasoning behind this is intuitive: If $F_{i,j}$ follows the normal distribution, then it is likely to only contain Gaussian noise. When given enough pixels j with similar neighborhoods to i, this Gaussian noise should average to 0 and give only the denoised image.

Because the neighborhoods provide small sample sizes, the authors of the SWNLM algorithm have also implemented the standard error SE to check if $F_{i,j}$ has any inaccuracies due to its small sample size. The standard error is defined as

$$\sigma_{F_{i,j}}/\sqrt{n} \tag{7}$$

where $\sigma_{F_{i,j}}$ is the standard deviation of $F_{i,j}$ and n the sample size.

Algorithm 3 Pseudocode for the SWNLM algorithm

Initialize output image d with 0's and same shape as input image v
for all $i \in v$ **do**
    $Z \leftarrow 0$
    $Wmax \leftarrow 0$
    set $\mathcal{N}_i$ to neighborhood of i
    for all $j \in$ searchWindow(i) where $i \neq j$ **do**
        set $\mathcal{N}_j$ to neighborhood of j
        $F_{i,j} \leftarrow (\mathcal{N}_i - \mathcal{N}_j)/(2\sigma)$
        $(w, t) =$ Shapiro-Wilk($F_{i,j}$)
        Calculate mean $\mu_{F_{i,j}}$ and standard deviation $\sigma_{F_{i,j}}$
        Calculate standard error SE
        if $t >= 0.05$ and $-SE < \mu_F < SE$ and $1 - SE < \sigma_F < 1 + SE$ **then**
            $Z \leftarrow Z + w$
            $Wmax \leftarrow \max(Wmax, w)$
            $d[i] \leftarrow d[i] + w * v[j]$
        end if
    end for
    if $Wmax == 0$ **then**
        $d[i] \leftarrow v[i]$
    else
        $d[i] \leftarrow d[i] + v[i] * Wmax$
        $Z \leftarrow Z + Wmax$
        $D[i] \leftarrow D[i]/Z$
    end if
end for
return D

The focus of SWNLM is to improve denoising quality, which is in contrast to most other NLM variations, such as Multi Resolution NLM by Karnati et al. [5] and FFT NLM by Wang et al. [12]. These variations use approximations of the original NLM algorithm to gain lower execution times, but with slightly lower denoising quality. Instead of relying on these approximations to decrease

execution times for the NLM algorithm, I will use GPU's with SWNLM to both decrease execution times and improve denoising quality over the sequential CNLM algorithm.

2.4 GPU computing

A GPU is a specialized computing chip designed for performing calculations on a large amount of data in parallel, with a focus on computer graphics. Developing efficient algorithms and programs for GPUs requires a different way of thinking than developing sequential algorithms for CPUs because of different design decisions and limitations for GPUs. In this report I will focus on GPU computing using the CUDA programming language and Nvidia hardware. In this section I will first discuss the architecture of a GPU, and then discuss the basics and some best practices for developing GPU algorithms.

2.4.1 GPU architecture

In contrast to a CPU with typically fewer than 20 cores, a GPU can have thousands of CUDA cores, which makes it very efficient at parallel computing. However, a CPU core is not equal to a CUDA core: A typical CPU core is very fast and can schedule and execute computations independently, whereas a CUDA core is relatively slow and grouped with other cores and this group is driven by a single scheduler, which means that each core in the group executes the same instructions at the same time. The group of threads running on these cores is a warp, where the size of a warp is fixed to 32 threads.

When running a CUDA program, each warp is assigned to a streaming multiprocessor (SM). An SM can have multiple warps assigned to it and schedules them to maximize the core usage. For example, if a warp has to wait for accessing memory, another warp can be scheduled to do some computations. A GPU can have multiple SMs to increase the amount of simultaneous active warps and thus computations at the same time.

A GPU has three types of memory: Global memory, shared memory and constant memory. Global memory has the largest memory capacity in the order of gigabytes and is accessible by all SMs. Global memory is used by default when allocating memory on the GPU. Global memory is the slowest type of memory, because it is located on its own physical chips apart from the GPU. Shared memory is part of an SM and a few orders of magnitude faster than global memory, but it can only be used by the SM in which it is located and it is limited in size to only tens of kilobytes, depending on the GPU. Constant memory is read-only memory that is copied to each SM's cache at the start when a CUDA program is run and has similar access speeds to shared memory. Constant memory is also limited in size to tens of kilobytes.

2.4.2 CUDA programming

When some code needs to be run on a GPU, it can be done via a kernel. A kernel is a function with the __global__ keyword. When a kernel is launched, two special arguments are given: The number of blocks and the number of threads per block. A block is a group of threads that all share the same shared memory and is assigned to a single SM. When the kernel is launched, then all threads in all blocks will run the same code. Each block and thread has its own block index and thread index respectively, which allows the division of work based on these indexes.

Because of the warp scheduling, the rule of thumb is that all threads in a CUDA function should run the same instructions whenever possible. If one thread in a warp has to execute a different set of instructions, then all other threads have to wait for it and do nothing. This is called warp divergence. An example of where this can happen is a conditional statement.

The amount of blocks per SM is limited by the amount of resources each block requires. Each SM has limited amounts of registers and shared memory, which need to be divided between the blocks.

Although optimizing CUDA programs can become very complex, some simple best practices for CUDA Programming are:

- Use block sizes that are multiples of the warp size to avoid warp divergence.
- Avoid large conditional statements that do not apply to all threads in a warp to avoid warp divergence.
- Copy repeatedly used global memory in a block to shared memory if possible.
- Profile and test the code often for finding bottlenecks.

3 Methods

3.1 Used tools

In this section I will discuss the used tools for this project, both hardware and software. These tools are used for both the development and the testing of all the software

3.1.1 Hardware

My personal desktop computer will be used for developing and testing the NLM algorithms. This computer has an AMD Ryzen 5 1600 processor with a clock speed of 3.6 GHz, 16 gigabytes (GB) of ram and an Nvidia GTX 1080 GPU. The GTX 1080 has 8 GB of ram, 20 SMs and 2560 CUDA cores in total [8]. The CUDA cores have a clock speed of up to 1733 MHz.

3.1.2 Software

All software development will be done using a Github repository, which allows for version control, backups and online access to the program. When the thesis is handed in, this Github repository will be made public and the thesis will be added to it. The Github repository can then be accessed with the following link: `https://github.com/Bob-vdV/NonLocalMeans`.

C++ will be used as the programming language for development and testing. C++ is a high level object oriented language with many of the functionalities of other high level languages, but with minimal overhead. This makes it very suitable for developing fast algorithms. The C++ 17 standard will be used to develop the software.

CUDA is a framework for developing algorithms for Nvidia GPUs. CUDA can be used as an extension to C++, which should provide an easy integration with the rest of the code base. CUDA has been chosen instead of other parallel programming frameworks, because it is well documented and also the fastest for Nvidia hardware. There exists a different CUDA compute capability version for each generation of GPU hardware, where each newer compute capability version adds new functionality supported by the hardware. Because I have a GTX 1080, its latest supported compute capability version will be used, which is version 6.1.

OpenCV is an open source C++ library for image processing, which provides an abstraction for handling images, including reading and writing to and from files and displaying them. OpenCV also includes many algorithms, such as Gaussian filtering and also some variants of NLM.

CMake will be used for building the software. CMake can manage libraries, languages and dependencies for the NLM algorithms and allows easy creation of various executables and tests.

The Nvidia visual profiler (NVVP) is a profiling tool for CUDA software. NVVP offers many measurement tools, including the amount of registers used per thread, memory efficiency and warp divergence. This can be used to determine the bottlenecks for a kernel and find potential improvements.

3.2 Development

In this section I will cover the strategy that was followed for developing the NLM algorithms. The order of implementation is as follows:

1. Sequential CNLM
2. Sequential SWNLM
3. SWNLM-CUDA

4. CNLM-CUDA

The sequential CNLM was first implemented as a baseline for the other algorithms, because it was the easiest to implement. This was then copied and modified for the sequential SWNLM implementation. Because the main goal of this project is to create an efficient SWNLM-CUDA implementation, it was implemented first based on the sequential SWNLM. Afterwards, the SWNLM-CUDA was implemented based on the SWNLM-CUDA implementation. The development of the sequential and CUDA algorithms are discussed in detail in their own subsection below.

The search windows and neighborhoods for pixels at the edges of images are not well-defined by the algorithm: They require that some pixels exist beyond the edge of the image. This can be solved in three ways: Adding special cases for the search windows and neighborhoods at edges with boundary checking, only including the pixels with well-defined search windows and neighborhoods in the denoised image (and thus losing the outer pixels), or padding the image beforehand with extra pixels along the edges, such that all search windows and neighborhoods of the pixels from the original image are well-defined. The last method is chosen because it is relatively simple to implement and the resolution of the denoised image is the same as the original. OpenCV offers a padding function with several padding types. Some experimentation showed that reflecting the outer pixels gives the best denoised images overall, thus that is is applied to the image for each NLM algorithm.

3.2.1 Sequential NLM algorithms

The development of the sequential NLM algorithms was relatively straightforward, because CNLM could be implemented based on algorithm 2 and SWNLM based on algorithm 3. No optimizations or approximations were made for the sequential algorithms, because they should function as a baseline for comparison against the CUDA algorithms.

At first, OpenCV was used extensively for doing the various matrix operations, such as element-wise multiplication and summing. However, this code could not easily be adapted for the CUDA versions of the algorithms and it was also relatively slow. Instead, in the final version OpenCV is only used for padding the images and storing them in memory. The pixels are directly accessed and modified through 1D arrays, because that gives the best performance for both the sequential and CUDA versions of the NLM algorithm.

Applying the SW test was more difficult than imagined beforehand. This is because there seems to be essentially only one implementation of the SW test, which is algorithm R94 by Royston written with Fortran from 1995 [10]. This implementation uses an approximation of the normal distribution using a large polynomial equation, which makes the code arguably hard to interpret. Fortunately, this code has been ported line by line to more modern languages, such as javascript, and is available on Github [7]. I have chosen to port the javascript version to C++ for the SWNLM algorithm, because the javascript version is easier to understand and translate to C++ than the Fortran version.

3.2.2 CUDA NLM algorithms

For creating the CUDA NLM algorithms, an iterative development cycle was used, which is the Assess, Parallelize, Optimize, Deploy (APOD) cycle shown in figure 7. First, the program to be parallelized is assessed for which parts are the most computationally intensive and are suitable for running on the GPU. Then these parts are parallelized and implemented in CUDA. The next step is to optimize them, for example by using shared memory and reducing warp divergence. The last step is to deploy the modified program and to test if it produces the same output as the previous version of the program. If more speedup is required, then the cycle can be repeated. This cycle allows for

small iterative speedup increments, where each potential optimization can be tested individually for the impact it has on the program's performance.

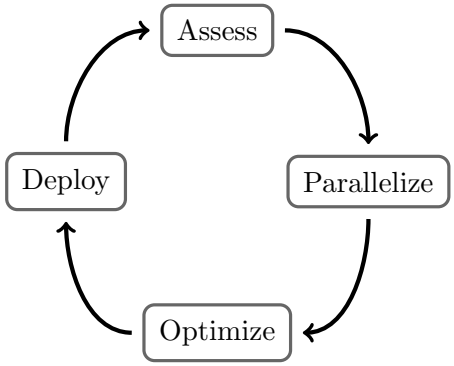


Figure 7: The APOD development cycle for CUDA programming.

First iteration.
The first iteration of the SWNLM-CUDA implementation is rather straightforward: Each pixel i in the image has its own block, where each thread in the block is the pixel j in the search window of i. Each thread then allocates memory for and computes $F_{i,j}$, applies the SW test to it and either accepts or rejects the result as described in algorithm 3. If it is accepted, then the pixel value and the weights are written to two variables in shared memory. When the whole block is finished executing, the final pixel value $d[i]$ is calculated and written to the output image by the first thread.

This approach was tested and was roughly 2 times faster than the sequential version with an execution time of 51 seconds on the Mandril image (shown in figure 9f). All other timings in this section are also based on the execution time on the Mandril image. Profiling the program with NVVP showed a large warp divergence and that the threads were inactive 80% of the time. Thus, more optimizations were needed. Three potential bottlenecks were found:

- The reads from global memory are too slow.
- The sorting algorithm creates a large warp divergence.
- Each thread needs to allocate memory for $F_{i,j}$.
- The block sizes are not multiples of the warp size because search windows have odd sizes.

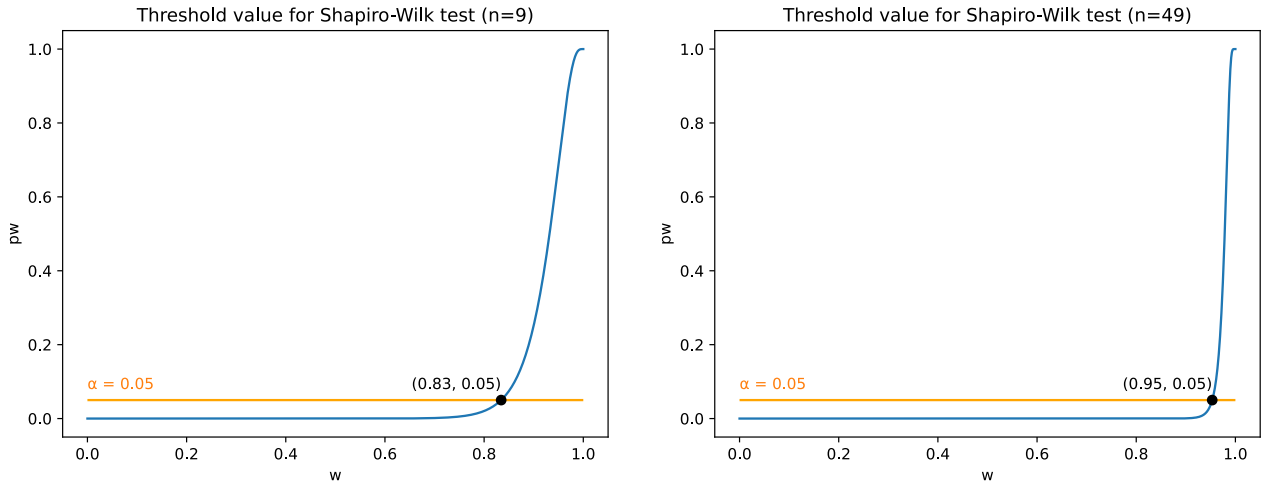
The optimizations in table 1 were applied to the algorithm, though most had little to no effect on the runtime. The numbers, blocks and threads were not changed (yet), because that would require refactoring a large part of the program.

Copying the search window to shared memory had no effect on runtime. This is probably because $F_{i,j}$ was allocated in global memory and thus the algorithm was bound by writing to the slower global memory. The problem is that the search window and $F_{i,j}$ for each thread cannot both be allocated to shared memory, due to shared memory limits per block. Allocating the shared memory for $F_{i,j}$ also gave no measurable speedup, probably due to slow reads from the search window in global memory.

Some testing showed that $Wmax$ is almost always greater than 0.95. Removing the calculations of $Wmax$ and setting the weight of the pixel i to 1 should theoretically speed up the program with a very small visual difference compared to the sequential SWNLM. However, some testing showed no measurable speedup for the algorithm.

Changing the sorting algorithm to a less warp-divergent version is the biggest optimization to the first iteration of SWNLM-CUDA. At first, a simple bubble sort was used for testing purposes. This algorithm is not known for its fast execution times with the average case computational complexity of $\mathcal{O}(N^2)$. It performs a large number of redundant comparisons and swaps, which all create warp divergence. When replacing it with heap sort, the execution time of the program decreased by 5 seconds. Heap sort was chosen because it has a worst case complexity of $\mathcal{O}(N \log N)$ and it requires no extra memory. Some other sorting algorithms were tested, such as bitonic sort, insertion sort and quick sort, but they were all slower than heap sort. To eliminate the warp divergence completely, a non-comparative sorting algorithm such as radix sort was also considered. However, this could not be implemented because radix sort requires auxiliary memory that exceeds the amount of maximum memory per block.

Profiling the SWNLM-CUDA algorithm with NVVP showed that the calculations of pw created a significant amount of warp divergence, because the value for pw is calculated using a conditional statement based on w and several polynomial equations. Fortunately, the SWNLM algorithm does not need an exact pw value, because it is only used to check if $pw \geq \alpha$. Because the value α is constant, the threshold of w to test the null hypothesis is also constant. The curve of pw and the threshold is visualized in figure 8. The threshold of w can be precomputed in the sequential part of the program, and then in the kernel w is directly compared to the threshold to test the null hypothesis. This optimization was implemented and reduced the execution time of the program by 1 second.



(a) $r_{\mathcal{N}} = 1$ (b) $r_{\mathcal{N}} = 3$

Figure 8: pw values for each w and their threshold for the null hypothesis with $\alpha = 0.05$. The threshold depends on α and $r_{\mathcal{N}}$, where a higher $r_{\mathcal{N}}$ gives a steeper curve and thus larger threshold.

Optimization	Achieved speedup
Copy the search window to shared memory.	No difference
Allocate space for each $F_{i,j}$ in shared memory.	No difference
Remove the calculation of $Wmax$ and replace it with 1.	No difference
Replace the sorting algorithm with a less warp-divergent version.	51 $\rightarrow$ 46 seconds
Precompute the minimum value of w that passes the SW test with $\alpha = 0.05$.	46 $\rightarrow$ 45 seconds

Table 1: Optimizations applied to the first iteration of the SWNLM-CUDA algorithm.

Final version
For the final version of the SWNLM-CUDA algorithm, the block sizes were refactored to not directly correlate with the given image resolution and search window size. Instead, the block sizes are fixed to some multiple of the warp size, which avoids the problem of large warp divergence caused by the odd-numbered search windows. Each thread computes a unique thread number based on its thread index and block index. Then this thread number is mapped to the pixels i and j that the thread needs to compute the weights for. Refactoring the block sizes reduced the algorithm's runtime from 45 to 7 seconds. Various multiples of the warp size were tested for the block size, where a block size of 32 was the fastest.

A disadvantage of the fixed block sizes is that there is no guarantee that all threads in a block belong to the same search window, which means that the search window cannot be copied to shared memory for faster computations. Instead, the shared memory was allocated for $F_{i,j}$ for each thread. Using shared memory gives another speedup of 1 second, reducing the total runtime down to 6 seconds for SWNLM-CUDA.

Up to this point SWNLM-CUDA was applied to images with double precision floating points (doubles), because it is the most versatile: It can be used to store most pixel representations, including standard 8-bit integers, 10-bit integers for high dynamic range, and also native floating point representations. However, using doubles requires 4 times the amount of memory space and memory bandwidth as 8-bit integers. Because of this, the program was rewritten using C++ templates to natively support images of any type of number format for both flexibility and memory efficiency. Using this template version with 8-bit images reduced the execution time from 6 to 4.5 seconds.

Although the images could be stored in any number format, $F_{i,j}$ and the SW test were still performed using doubles because it requires decimal numbers. However, 8-bit integers do not require the SW test to have double precision, because 8-bit integers themselves have very low precision and thus any extra precision is lost when the final result is stored in the 8-bit denoised image. Due to this reason, the program was modified with an extra template, where the precision of the floating points used for $F_{i,j}$ and the SW test is determined by the precision of the input image type: When given low precision type, a single precision float is used. Otherwise, a double precision is used. This optimization with 8-bit images reduced the execution time further from 4.5 to 1.5 seconds.

Optimization	Achieved speedup
Remove the calculation of $Wmax$ and replace it with 1.	No difference
Replace the sorting algorithm with a less warp-divergent version.	$51 \rightarrow 46$ seconds
Precompute the minimum value of w that passes the SW test for given α.	$46 \rightarrow 45$ seconds
Set the block sizes to a fixed number	$45 \rightarrow 7$ seconds
Allocate space for each $F_{i,j}$ in shared memory.	$7 \rightarrow 6$ seconds
Store images using 8-bit integers instead of doubles	$6 \rightarrow 4.5$ seconds
Do SW test with less precise floats instead of doubles	$4.5 \rightarrow 1.5$ seconds

Table 2: Optimizations applied to the final iteration of the SWNLM-CUDA algorithm.

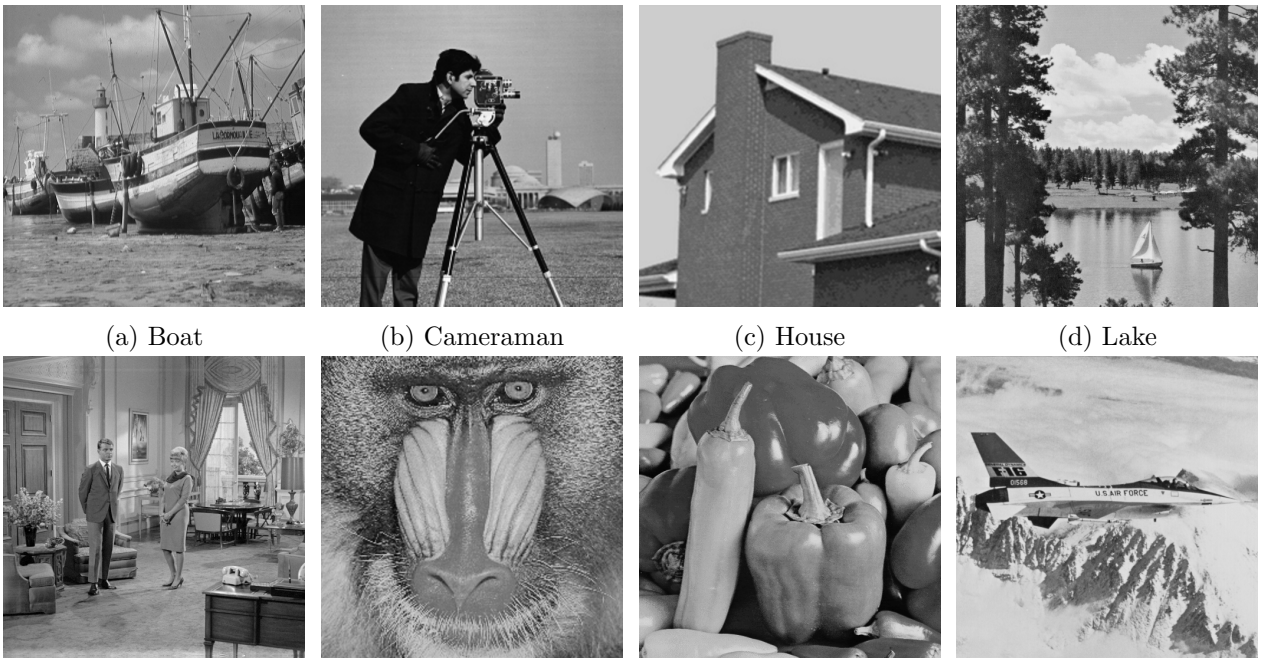
CNLM-CUDA
For developing the CNLM-CUDA program, all applicable SWNLM CUDA optimizations were also used. Thus, the program uses a fixed block size and C++ templates for the input image type and precision of intermediate calculations. This gave a baseline execution time of 0.7 seconds.

Profiling the program gave an interesting result: The whole algorithm was slowed down by a single pow function, which was used for squaring the difference between $\mathcal{N}_i$ and $\mathcal{N}_j$. Replacing the pow function with a simpler square function reduced the program execution from 0.7 to 0.1 seconds.

3.3 Testing

In this section I will describe the testing methodology that will be used to evaluate the quality of the denoising algorithms. I will first discuss the images and then the performance metrics that will be used in the tests.

3.3.1 Images



(a) Boat (b) Cameraman (c) House (d) Lake

(e) Livingroom (f) Mandril (g) Peppers (h) Plane

Figure 9: Images from the standard images dataset

(a) Book (b) Toy (c) Water

Figure 10: Images from the Poly U dataset

The images the algorithms will be tested on are displayed in figure 9 and are very commonly used in image processing papers [4]. These images have been chosen specifically because of their wide variety: Some images contain a lot of detail, while others have more even surfaces. Some pictures with faces are also included for testing because face quality is often a subjectively important part in photographs.

The NLM algorithms will also be tested on three images from the Poly U dataset [14]. This dataset offers images with noisy and ground truth variants, where the noise in the images are created by

cameras instead of some software that adds Gaussian noise. However, these noisy images will not be used for the tests and the noise will be added in the same way as with the standard testing images. The images from the Poly U dataset are instead selected because of their large size. These large images can be downscaled to various resolutions, which allows testing for time consumption based on the image sizes. The resolutions are listed in table 3.

Resolution	Number of pixels
2592×1728	4 478 976
1296×864	1 119 744
648×432	279 936
324×216	69 984

Table 3: The different testing resolutions for the images from the Poly U dataset

All images are in grayscale format, because the tested NLM algorithms can only denoise one color channel at a time. For simplicity and easier comparison all images have been converted to grayscale. This should make the difference between the NLM algorithms clearer, because there is only one color channel.

3.3.2 Performance metrics

The denoising algorithms will be evaluated on two properties: image denoising quality and speed. The image denoising quality will be evaluated using the Peak Signal to Noise Ratio (PSNR) and Mean Structural Similarity Index Measure (MSSIM). PSNR is chosen because it is used in many other image processing papers and thus it provides a frame of reference for denoising quality. However, PSNR does not represent well how a human perceives image quality. MSSIM is used because it does represent the human's visual system. The calculations of PSNR and MSSIM are explained below. Additionally, the denoised images will be evaluated subjectively and checked for any visual artifacts.

PSNR

The PSNR is a measure for quantifying the amount of noise compared to the actual data for a given signal. A high PSNR value indicates a low amount of noise in a signal and a low value indicates a high amount of noise. When there is no noise in a signal, the PSNR value is infinite. PSNR is calculated using the Mean Square Error (MSE), which is the sum of the squared difference between the original and the denoised image. The PSNR is defined as

$$\text{PSNR} = 10 \log_{10}(\frac{MAX^2}{\text{MSE}}) \tag{8}$$

where MAX is the maximum value a signal can have. Typically $MAX = 255$ for images with 8 bits per color channel. The MSE between two images x and y of size N is defined as

$$\text{MSE} = \frac{1}{N} \sum_{i=0}^{N-1} (x(i) - y(i))^2 \tag{9}$$

Though PSNR can give a mathematical indication of how similar two images are, it is not a good representative of how a person perceives the similarity between two images [3]. For example, an image with a slight change in brightness will give a low PSNR value, but will visually look almost identical to the original.

MSSIM

For a better representation of the human visual system, we will use MSSIM [13]. MSSIM is the mean of SSIM values applied to small patches of an image. I will first explain SSIM and then discuss why MSSIM is used. SSIM is calculated as a combination of three components: luminance, contrast, and structure. Luminance is defined as the overall brightness of an image, which is identical to the mean. The luminance comparison is defined in equation 10. C_1 is a small constant to avoid division by 0. C_1 defined as $(K_1 MAX)^2$, where typically $K_1 = 0.01$.

$$l(x, y) = \frac{2\mu_x \mu_y + C_1}{\mu_x^2 + \mu_y^2 + C_1} \tag{10}$$

The contrast is defined using the standard deviation σ_x, σ_y variance σ_x^2, σ_y^2 of the images and is calculated by equation 11. Similar to luminance, the contrast calculation has a constant $C_2 = (K_2 MAX)^2$ with typically $K_2 = 0.03$.

$$c(x, y) = \frac{2\sigma_x \sigma_y + C_2}{\sigma_x^2 \sigma_y^2 + C_2} \tag{11}$$

The structure is given by equation 12, where σ_{xy} is the cross-correlation of x and y. C_3 is defined as $\frac{C_2}{2}$.

$$s(x, y) = \frac{\sigma_{xy} + C_3}{\sigma_x \sigma_y + C_3} \tag{12}$$

SSIM is a weighted combination of the luminance, contrast and structure defined in equation 13. The parameters α, β, γ control the weights of each individual component. For testing the denoising algorithms we set $\alpha = \beta = \gamma = 1$ for simplicity. SSIM values range from 0 to 1, where 0 indicates two entirely dissimilar images and 1 two identical images. Thus, a number close to 1 is desirable for the denoised image.

$$\text{SSIM} = l(x, y)^\alpha \times c(x, y)^\beta \times s(x, y)^\gamma \tag{13}$$

A drawback of SSIM is that it is computed on the image as a whole, which means that images with many local dissimilarities could still have a high SSIM value. To prevent this, the SSIM values are calculated locally for overlapping 8×8 square patches of the image. The average of these SSIM values is then calculated to get the MSSIM.

Time Measurements

Each algorithm's execution time will also be measured. For each test, the algorithm is run 3 times and the minimum execution time is saved in the results. This is done because the operating system might do some tasks in the background and use up some of the hardware resources, which could slow down the program.

3.3.3 Parameters

The algorithms will be tested with several parameters, which are the amount of noise σ, the search radius r_S and the neighbor radius $r_{\mathcal{N}}$. The tested values for each parameter are listed in table 4. Each permutation of parameters will be tested, thus each combination of image and NLM algorithm will have $6 \times 3 \times 3 = 54$ testing results. These results will then be compared to determine which combinations of parameters give the best denoising quality relative to the execution time.

Parameter	Values
σ	$\{10, 20, 30, 40, 50, 60\}$
r_S	$\{5, 8, 10\}$
$r_{\mathcal{N}}$	$\{1, 2, 3\}$

Table 4: The parameters the algorithms will be tested with.

4 Results

In this section I will discuss the results for the denoising algorithm tests. Due to the amount of data points, I will only discuss the average values over all standard images. All individual values for each image can be found in the Github repository discussed in subsection 3.1.2.

The averages for the standard images of all combinations of denoising algorithm, search radius and neighbor radius can be found in appendix A.

4.1 Optimal search radius and neighbor radius

Table 5 shows the r_S and $r_\mathcal{N}$ combinations with the highest MSSIM score for each NLM algorithm. Overall, the combination $r_S = 10$ and $r_\mathcal{N} = 3$ give the highest MSSIM scores for each NLM algorithm, with some outliers at the lower and higher noise levels. It should be noted though that the differences in scores between best and second best combination can be very small. With the exception of $\sigma = 10$, SWNLM-CUDA consistently gives the highest MSSIM values, closely followed by SWNLM. The PSNR and MSSIM values for CNLM and CNLM-CUDA are identical for all noise levels.

sigma	algorithm	search radius	neighbor radius	avg PSNR	avg MSSIM	avg execution time (s)
10	cnlm	10	3	**33.1131**	**0.8272**	6.9164
10	cnlm-cuda	10	3	**33.1131**	**0.8272**	0.1478
10	swnlm	5	2	32.6393	0.8187	19.7478
10	swnlm-cuda	5	2	32.6470	0.8188	**0.1179**
20	cnlm	10	3	29.3295	0.6342	6.7559
20	cnlm-cuda	10	3	29.3295	0.6342	**0.1481**
20	swnlm	10	2	29.8178	0.7005	72.5798
20	swnlm-cuda	10	2	**29.8234**	**0.7006**	0.3890
30	cnlm	10	3	26.9291	0.4933	6.7469
30	cnlm-cuda	10	3	26.9291	0.4933	**0.1478**
30	swnlm	10	3	27.4474	0.5969	169.6403
30	swnlm-cuda	10	3	**27.4509**	**0.5970**	1.1845
40	cnlm	10	3	25.1879	0.3996	6.7478
40	cnlm-cuda	10	3	25.1879	0.3996	**0.1481**
40	swnlm	10	3	25.9478	0.4987	169.9409
40	swnlm-cuda	10	3	**25.9504**	**0.4988**	1.1823
50	cnlm	8	3	23.8182	0.3329	4.4209
50	cnlm-cuda	8	3	23.8182	0.3329	**0.0988**
50	swnlm	10	3	24.2764	0.4102	170.1825
50	swnlm-cuda	10	3	**24.2792**	**0.4103**	1.1834
60	cnlm	5	3	22.5459	0.2857	1.8588
60	cnlm-cuda	5	3	22.5459	0.2857	**0.0443**
60	swnlm	10	2	23.1724	0.3385	73.3346
60	swnlm-cuda	10	2	**23.1782**	**0.3387**	0.3878

Table 5: The combinations of r_S and $r_\mathcal{N}$ with the highest MSSIM values for each denoising algorithm. The best value for each performance metric is shown in bold.

Figure 11 shows the results of CNLM-CUDA applied to the boat image for each combination of r_S and $r_\mathcal{N}$. For all images, there is still a visually noticeable amount of noise present. The results for

$r_S = 5$ have the lowest MSSIM scores and also show some visual artifacts where the noise is grouped in small patches, whereas the noise for $r_S = 8$ and $r_S = 10$ is less noticeable. For this image $r_S = 8$ gives slightly higher MSSIM scores, though there is no visual difference compared to $r_S = 10$. For all tested r_S, $r_\mathcal{N} = 3$ gives the highest MSSIM scores with $r_\mathcal{N} = 2$ as a close second with no visual differences. $r_\mathcal{N} = 1$ gives the lowest performance with more noise in the images.

(a) Original

(b) Noisy
MSSIM = 0.1529

(c) $r_S = 5$, $r_\mathcal{N} = 1$
MSSIM = 0.3950

(d) $r_S = 5$, $r_\mathcal{N} = 2$
MSSIM = 0.4254

(e) $r_S = 5$, $r_\mathcal{N} = 3$
MSSIM = 0.4275

(f) $r_S = 8$, $r_\mathcal{N} = 1$
MSSIM = 0.4016

(g) $r_S = 8$, $r_\mathcal{N} = 2$
MSSIM = 0.4329

(h) $r_S = 8$, $r_\mathcal{N} = 3$
MSSIM = 0.4350

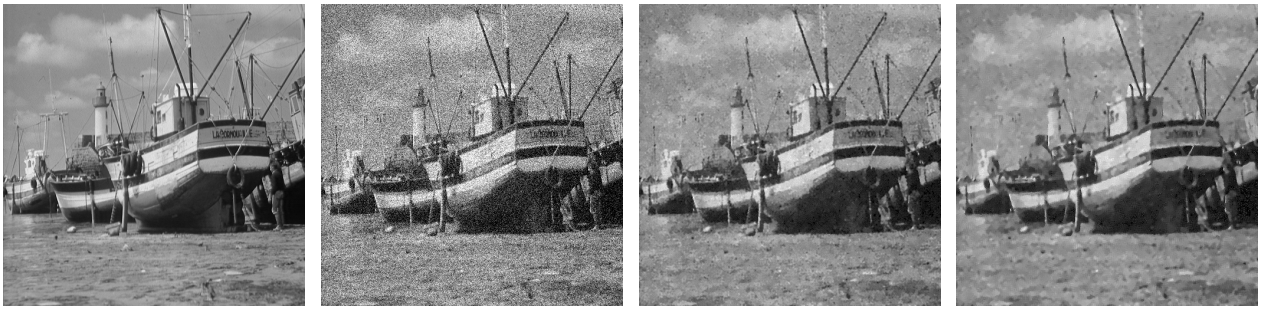
(i) $r_S = 10$, $r_\mathcal{N} = 1$
MSSIM = 0.4010

(j) $r_S = 10$, $r_\mathcal{N} = 2$
MSSIM = 0.4321

(k) $r_S = 10$, $r_\mathcal{N} = 3$
MSSIM = 0.4342

Figure 11: Noisy boat image ($\sigma = 40$) denoised with CNLM-CUDA for each combination of r_S and $r_\mathcal{N}$.

The results of each r_S and $r_\mathcal{N}$ for SWNLM-CUDA are shown in figure 12. $r_S = 5$ gives the lowest MSSIM scores and also gives some visual artifacts with patches of noise. $r_S = 8$ and $r_S = 10$ have similar MSSIM scores and are visually identical. Visually, $r_\mathcal{N} = 3$ gives the least noise overall, though the MSSIM scores for $r_S = 8$ and $r_S = 10$ with $r_S = 3$ are lower than the scores with $r_S = 2$. Figure 12k shows some unaltered noise on the dark underside of the boats at the center right, which could explain the lower MSSIM score. The text at the back of the boat is also more blurry at $r_\mathcal{N} = 3$ than at $r_\mathcal{N} = 1$ and $r_\mathcal{N} = 2$, which could indicate a loss of details.



(a) Original

(b) Noisy ($\sigma = 40$)
MSSIM = 0.1529

(c) $r_S = 5$, $r_{\mathcal{N}} = 1$
MSSIM = 0.4467

(d) $r_S = 5$, $r_{\mathcal{N}} = 2$
MSSIM = 0.5114

(e) $r_S = 5$, $r_{\mathcal{N}} = 3$
MSSIM = 0.4964

(f) $r_S = 8$, $r_{\mathcal{N}} = 1$
MSSIM = 0.4500

(g) $r_S = 8$, $r_{\mathcal{N}} = 2$
MSSIM = 0.5195

(h) $r_S = 8$, $r_{\mathcal{N}} = 3$
MSSIM = 0.4950

(i) $r_S = 10$, $r_{\mathcal{N}} = 1$
MSSIM = 0.4485

(j) $r_S = 10$, $r_{\mathcal{N}} = 2$
MSSIM = 0.5199

(k) $r_S = 10$, $r_{\mathcal{N}} = 3$
MSSIM = 0.4911

Figure 12: Noisy boat image ($\sigma = 40$) denoised with SWNLM-CUDA for each combination of r_S and $r_{\mathcal{N}}$.

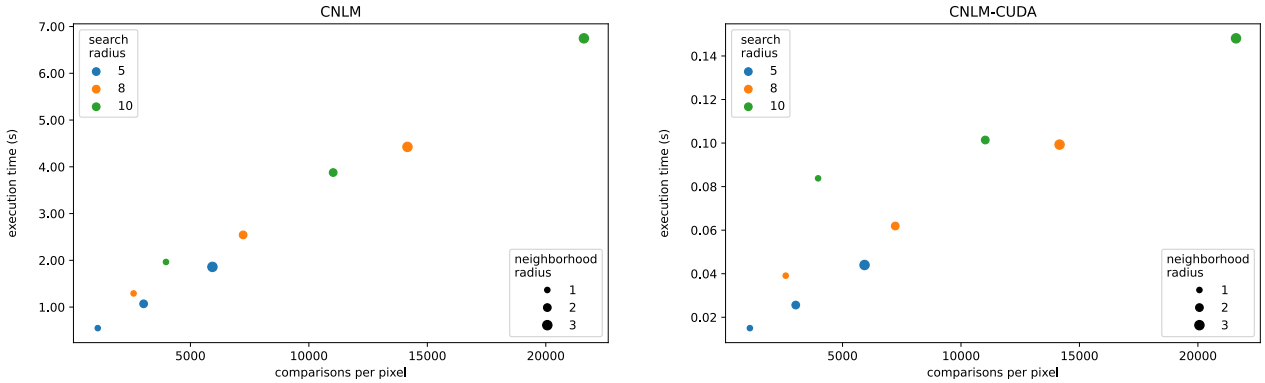
4.1.1 Execution times

The number of comparisons that need to be performed per pixel are dependent on r_S and $r_{\mathcal{N}}$ and is computed with the equation $(2r_S + 1)^2 \times (2r_{\mathcal{N}} + 1)^2$. Table 6 shows the number of comparisons that are performed for each combination of r_S and $r_{\mathcal{N}}$.

Figure 13 shows the execution times for the algorithms plotted against the comparisons per pixel for each tested r_S and $r_{\mathcal{N}}$. It shows that the execution times of the algorithms tend to scale linearly with the number of comparisons per pixel. Figure 13b shows one outlier for CNLM-CUDA with $r_S = 10$ and $r_{\mathcal{N}} = 1$, where the execution time is relatively large.

r_S	$r_\mathcal{N}$	size of S	size of $\mathcal{N}$	Comparisons per pixel
5	1	121	9	1089
5	2	121	25	3025
5	3	121	49	5929
8	1	289	9	2601
8	2	289	25	7225
8	3	289	49	14161
10	1	441	9	3969
10	2	441	25	11025
10	3	441	49	21609

Table 6: The number of comparisons per pixel for each tested combination of r_S and $r_\mathcal{N}$.



(a) CNLM (b) CNLM-CUDA

(c) SWNLM (d) SWNLM-CUDA

Figure 13: Execution times of all combinations of r_S and $r_\mathcal{N}$ for each NLM algorithm.

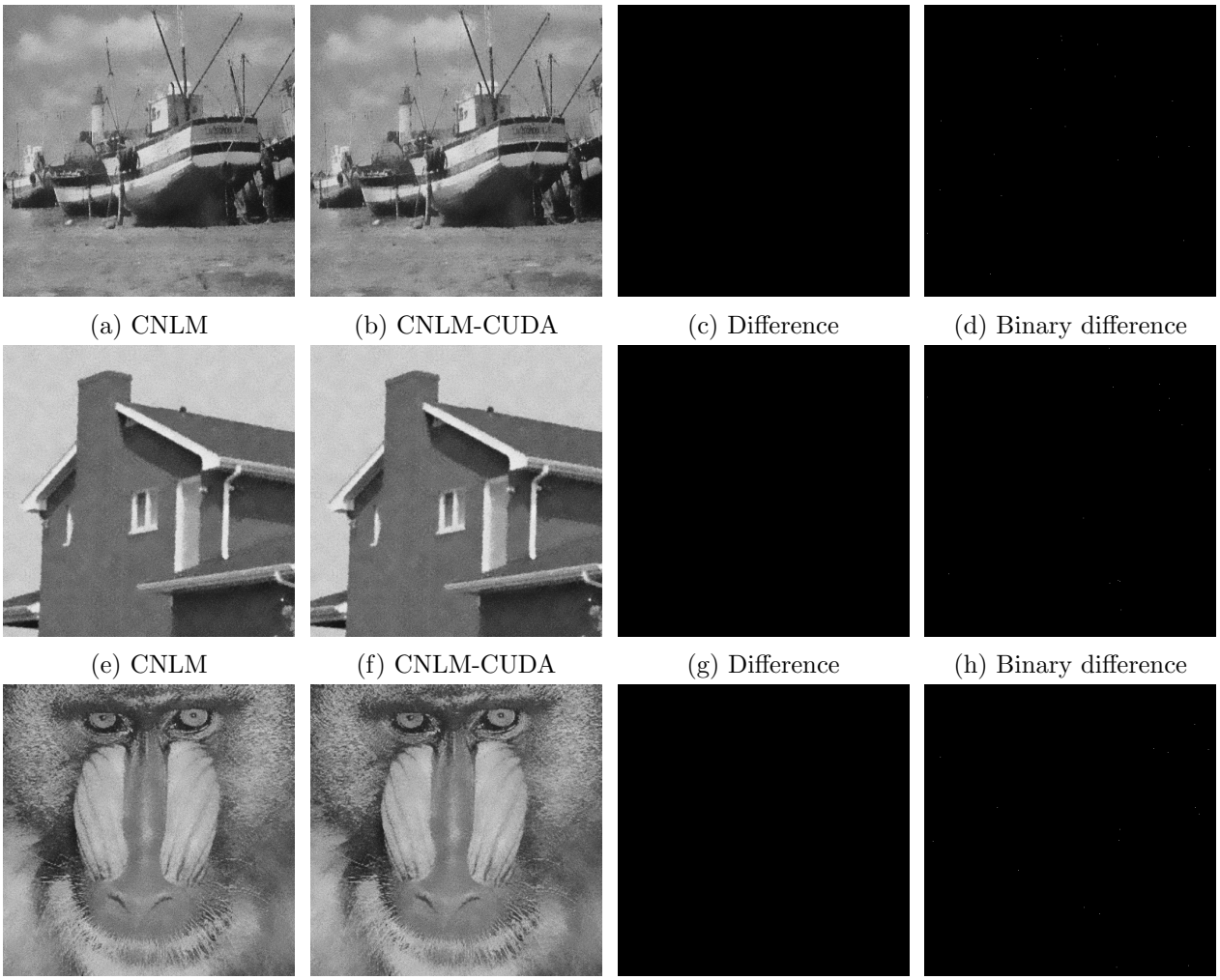
4.2 Differences of CUDA implementations

In this section I will compare the sequential and CUDA versions of the NLM algorithms and highlight any differences.

CNLM

Figure 14 shows the differences between CNLM and CNLM-CUDA. The third images from the left are the actual differences and the fourth images are binary representations of these differences, where any non-zero value is mapped to white. The denoised images are visually identical: the difference images are almost completely black and the binary differences show that only a small

amount of pixels have different values for CNLM and CNLM-CUDA. These small differences are likely caused by imprecisions in floating point calculations.



(a) CNLM (b) CNLM-CUDA (c) Difference (d) Binary difference

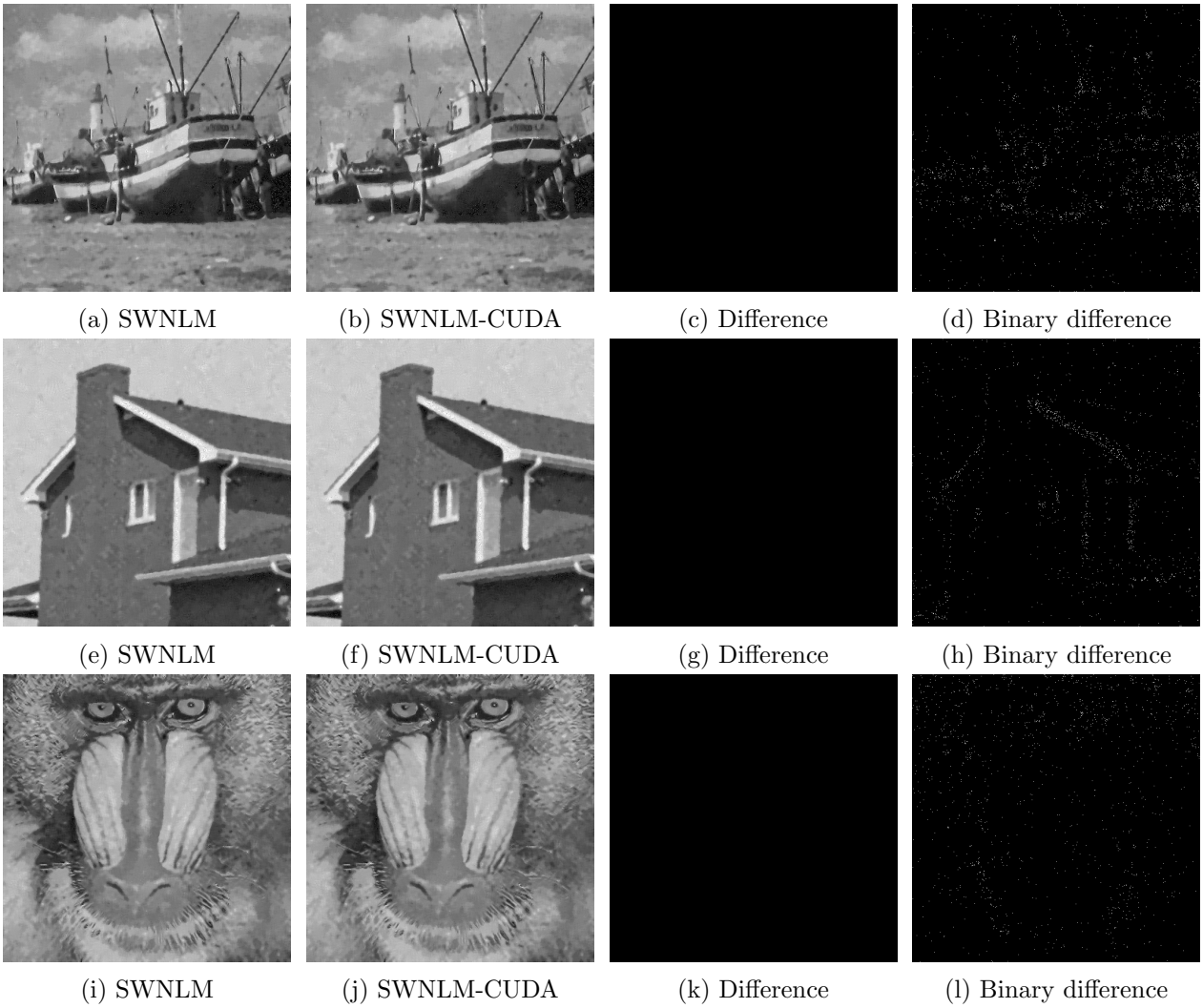
(e) CNLM (f) CNLM-CUDA (g) Difference (h) Binary difference

(i) CNLM (j) CNLM-CUDA (k) Difference (l) Binary difference

Figure 14: Differences between the sequential and CUDA versions of CNLM for the Boat, House and Mandril images.

SWNLM

The differences between SWNLM and SWNLM-CUDA are shown in figure 15. Similar to CNLM, the difference images are almost completely black, indicating that there is visually no difference between images denoised by SWNLM and SWNLM-CUDA. However, the binary difference show a substantial amount of white pixels, indicating that SWNLM-CUDA produces slightly different images than SWNLM. This can be explained by the fact that $Wmax$ is not computed in SWNLM-CUDA and simply substituted by 1, whereas it is computed in the sequential SWNLM algorithm.



(a) SWNLM (b) SWNLM-CUDA (c) Difference (d) Binary difference

(e) SWNLM (f) SWNLM-CUDA (g) Difference (h) Binary difference

(i) SWNLM (j) SWNLM-CUDA (k) Difference (l) Binary difference

Figure 15: Differences between the sequential and CUDA versions of SWNLM for Boat, House and Mandril.

4.3 Denoising quality

In this subsection I will analyze and compare the denoising quality of CNLM-CUDA and SWNLM-CUDA. The sequential versions will not be evaluated, because they are visually identical to their CUDA versions. All images in this subsection are made with the noise level $\sigma = 40$ and parameters $r_S = 10$, $r_{\mathcal{N}} = 3$.

Table 7 shows the PSNR and MSSIM values for CNLM-CUDA and SWNLM-CUDA for the standard images and the results are displayed in figure 16 and figure 17. For the most images SWNLM-CUDA has higher PSNR and MSSIM values than CNLM-CUDA. A subjective visual inspection confirms this, because the SWNLM-CUDA results contain less noise than CNLM-CUDA. However, it should be mentioned that the SWNLM-CUDA results have some artifacts, which somewhat resemble a watercolor filter. This is most noticeable in the shadows on the wall in figure 17d. These artifacts can also explain SWNLM-CUDA's lower MSSIM value for Mandril, because all the small details in the mandril's hairs are lost.

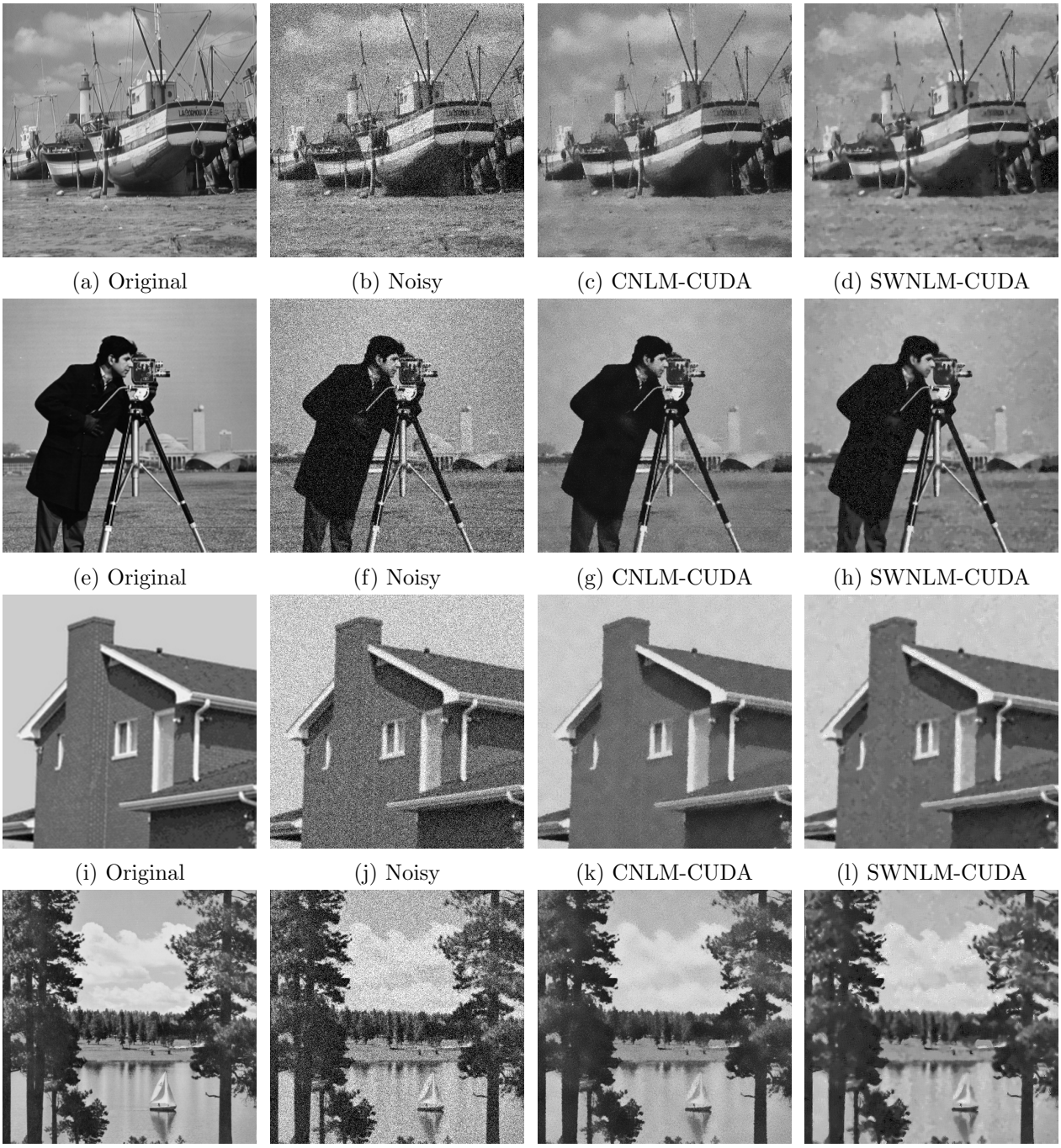
SWNLM-CUDA also has a lower PSNR value than CNLM-CUDA for the Cameraman image, which can be explained by figure 16h. It shows that the cameraman's jacket is still noisy, which is because

of the threshold used by SWNLM-CUDA and the method of adding noise. When the noise is added to the image, it is clipped to the value range of the image, which means that any value below 0 is set to 0. As a result, the noise in the black areas of the image is not normally distributed, which is why the black areas fail to be denoised by SWNLM-CUDA.

For the noisy images, the PSNR values are relatively constant across the images. This is as expected, because there should be roughly equal amounts of noise in all images. However, there is a considerable difference in the MSSIM values across the noisy images, with the lowest value for House and the highest value for Mandril. This can be explained by the way the MSSIM is calculated: A part of the MSSIM calculation is the difference in contrast. The original House image does not have much contrast due to its smooth textures, which means that the noisy image adds a relatively large amount of contrast, which gives a lower MSSIM value. On the other hand, the original Mandril image already has large amounts of contrast due to the hairs, and thus the added noise has relatively low effect on the contrast, which gives a higher MSSIM value. These MSSIM values also represent the human visual system: The noise on the smooth textures in House is subjectively more noticeable than on the many hairs in Mandril.

Image	PSNR			MSSIM		
	Noisy	CNLM-CUDA	SWNLM-CUDA	Noisy	CNLM-CUDA	SWNLM-CUDA
Boat	16.3804	24.6654	**25.4696**	0.1529	0.4342	**0.4911**
Cameraman	16.6392	**25.9880**	25.1043	0.1192	0.3217	**0.4719**
House	16.5133	26.7664	**29.7184**	0.0778	0.2648	**0.5006**
Lake	16.6491	24.8156	**25.0256**	0.1901	0.4739	**0.5340**
Livingroom	16.3487	24.2110	**24.9051**	0.1637	0.4348	**0.4741**
Mandril	16.2563	23.4073	**23.5753**	0.2120	**0.4962**	0.4405
Peppers	16.4532	25.7017	**27.1050**	0.1105	0.4091	**0.5667**
Plane	16.7011	25.9475	**26.7001**	0.1364	0.3619	**0.5111**

Table 7: PSNR and MSSIM values for standard images for CNLM-CUDA and SWNLM-CUDA. The best values are shown in bold.



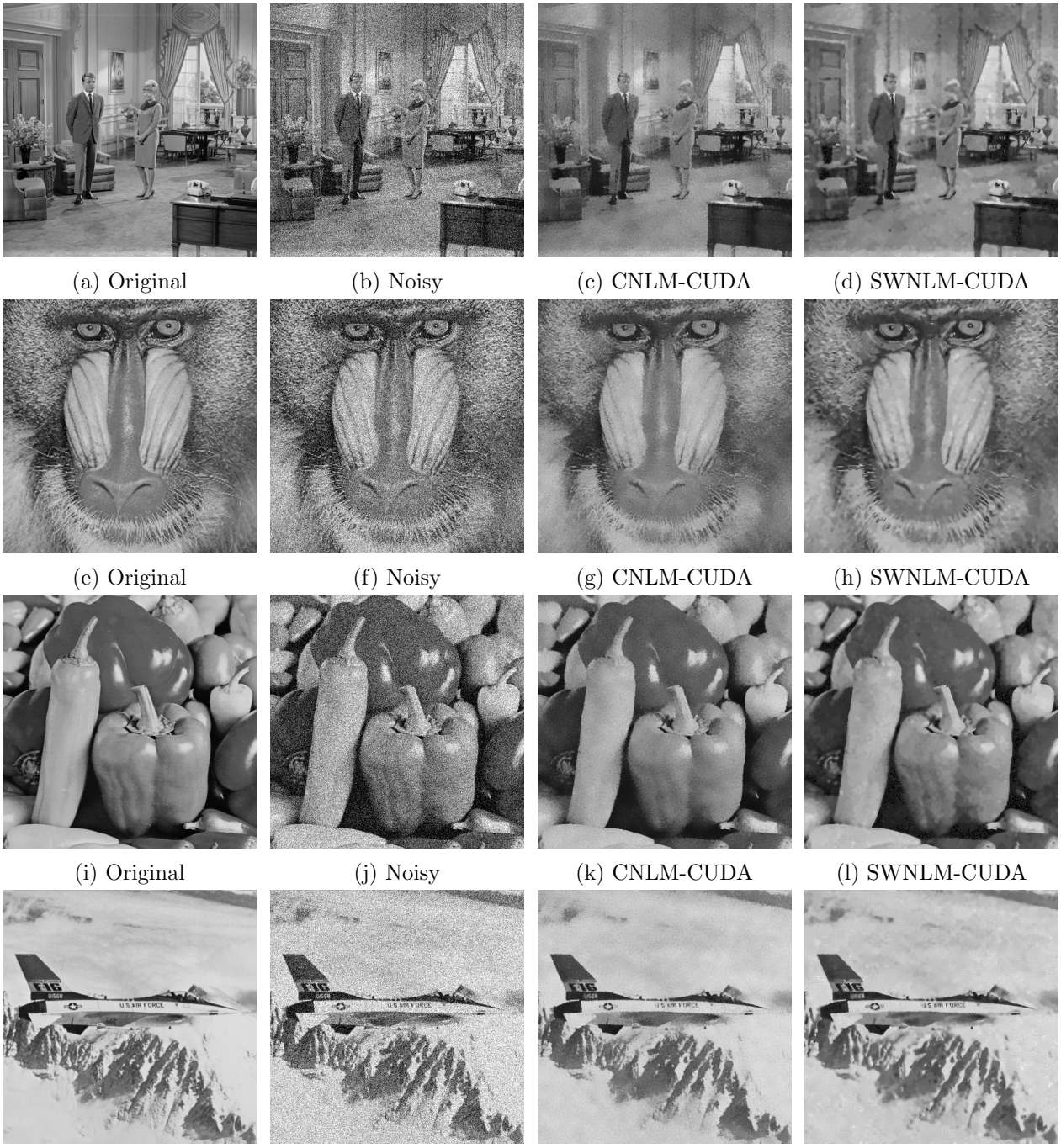
(a) Original (b) Noisy (c) CNLM-CUDA (d) SWNLM-CUDA

(e) Original (f) Noisy (g) CNLM-CUDA (h) SWNLM-CUDA

(i) Original (j) Noisy (k) CNLM-CUDA (l) SWNLM-CUDA

(m) Original (n) Noisy (o) CNLM-CUDA (p) SWNLM-CUDA

Figure 16: Denoised results for Boat, Cameraman, House and Lake.



(a) Original (b) Noisy (c) CNLM-CUDA (d) SWNLM-CUDA

(e) Original (f) Noisy (g) CNLM-CUDA (h) SWNLM-CUDA

(i) Original (j) Noisy (k) CNLM-CUDA (l) SWNLM-CUDA

(m) Original (n) Noisy (o) CNLM-CUDA (p) SWNLM-CUDA

Figure 17: Denoised results for Livingroom, Mandril, Peppers and Plane.

4.4 Execution times

In this section I will discuss the execution times of the algorithm. I will explore several factors that could influence execution times, which are the types of images, different noise levels and resolutions.

4.4.1 Execution times for different images

I will first discuss the relation between the type of image and the execution times for the algorithms. Table8 shows the tested algorithms' execution times for the standard images with $\sigma = 40$, $r_S = 10$

and $r_{\mathcal{N}} = 3$. For all algorithms, the execution times between the images are very consistent and vary in only a few percentage points. The small execution time differences are likely caused by external factors, such as the operating system performing background tasks and thus using resources.

image	CNLM	CNLM-CUDA	SWNLM	SWNLM-CUDA
boat	6.724	0.148	170.049	1.174
cameraman	6.751	0.148	169.671	1.188
house	6.743	0.148	170.261	1.181
lake	6.739	0.149	169.971	1.189
livingroom	6.749	0.147	170.044	1.173
mandril	6.802	0.150	169.844	1.178
peppers	6.736	0.148	169.825	1.186
plane	6.738	0.147	169.862	1.189

Table 8: Execution times (s) of the algorithms for standard images with $\sigma = 40$, $r_S = 10$ and $r_{\mathcal{N}} = 3$.

4.4.2 Impact of noise level on execution time

Table 9 shows the average execution times of the algorithms for the standard images with $r_S = 10$ and $r_{\mathcal{N}} = 3$. Overall, the execution times are consistent for all noise levels, indicating that the amount of noise has (almost) no impact on the execution time of the algorithms. However, for SWNLM there seems to be a small correlation where a higher noise level results in higher execution times by a few percent. This could potentially be caused by variances in sensitivity of the SW-test for different noise levels, where a lower noise level leads to more rejections, which in turn requires fewer additions of weights. However, this is a speculation and requires further research.

sigma	CNLM	CNLM-CUDA	SWNLM	SWNLM-CUDA
10	6.916	0.148	166.352	1.207
20	6.756	0.148	168.836	1.191
30	6.747	0.148	169.640	1.185
40	6.748	0.148	169.941	1.182
50	6.742	0.147	170.183	1.183
60	6.744	0.147	170.232	1.188

Table 9: Execution times (s) for the tested algorithms at different noise levels. The execution times are the averages of all standard images with $r_S = 10$ and $r_{\mathcal{N}} = 3$.

4.4.3 Resolution scaling

In this subsection I will discuss the relative speedup that is achieved by the CUDA versions of the NLM algorithms. The table with all execution times for the Poly U images can be found in appendix B. The averages for all images are plotted in figure 18, which shows that all tested NLM algorithms scale linearly with the image resolution.

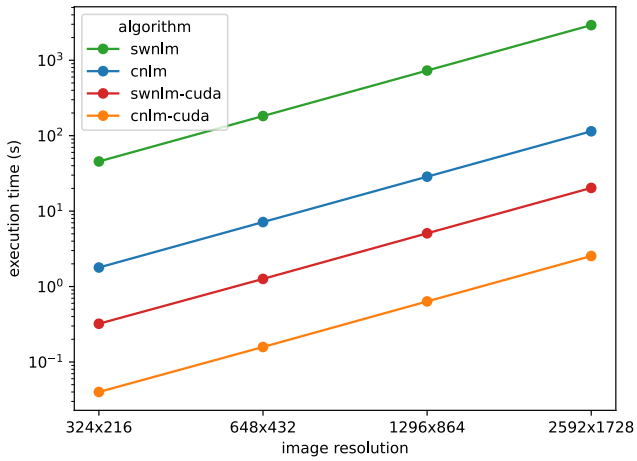


Figure 18: Average execution times for the tested Poly U images at different resolutions. Note that both axes have logarithmic scales.

Figure 19 shows the relative speedup achieved for the CUDA versions of CNLM and SWNLM. CNLM-CUDA is roughly 45 times faster than CNLM and SWNLM-CUDA is up to 144 times faster than SWNLM. The smaller speedup for CNLM-CUDA is likely due to memory bandwidth constraints by the GPU and the fact that the CUDA cores can compute the denoised result faster than the GPU can read and write to memory. SWNLM-CUDA does not have this problem because there is relatively more compute per memory access, which makes it more suitable for a GPU implementation. Both CUDA algorithms have a relatively smaller speedup at the lowest resolution, which can be explained by a relatively larger sequential overhead compared to the small workloads.

On average, SWNLM is roughly 25 times slower than CNLM, which is due to the relatively larger computational complexity of SWNLM. However, SWNLM-CUDA is only 8 times slower than CNLM-CUDA because of the larger speedup achieved by SWNLM-CUDA.

resolution	avg sequential time (s)	avg CUDA time (s)	achieved speedup
324×216	1.7873	0.0400	44.6833
648×432	7.1577	0.1580	45.3017
1296×864	28.6027	0.6350	45.0436
2592×1728	114.4240	2.5407	45.0370

(a) CNLM

resolution	avg sequential time (s)	avg CUDA time (s)	achieved speedup
324×216	45.5367	0.3213	141.7116
648×432	182.1727	1.2627	144.2761
1296×864	729.9890	5.0773	143.7741
2592×1728	2921.9523	20.3050	143.9031

(b) SWNLM

Figure 19: Average execution times for the Poly U images and relative speedup achieved by the CUDA versions of the NLM algorithms.

5 Conclusion

The goal of this bachelor thesis was to create fast CUDA implementations of the CNLM and SWNLM algorithms. CNLM-CUDA is up to 45 times faster than CNLM and SWNLM-CUDA is up to 145 times faster than SWNLM, thus I believe that this goal has been achieved well. The results given by CNLM-CUDA and SWNLM-CUDA are visually identical to their sequential versions, which means that the CUDA versions are implemented correctly.

The iterative approach for developing the CUDA versions has worked well because it allowed for quick testing on both correctness and performance for each potential improvement. The overviews in section 3.2 with all performance improvements and their respective speedup should also be useful for a future CUDA NLM implementation based on some different variation.

It was shown that overall SWNLM-CUDA has better denoising quality than CNLM-CUDA, with the exception of a few cases where SWNLM-CUDA can produce some noticeable artifacts. However, CNLM-CUDA is 8 times faster on average, which means that there is no clear best algorithm for all use cases: If speed or throughput is more important than visual quality, then CNLM-CUDA should be considered. Otherwise, if an image is denoised only occasionally or time consumption is non-critical, such as in photo editing software or in post processing for medical scans, then SWNLM-CUDA would be the better choice, given that the images do not contain large black areas or very fine details.

6 Future Work

There are a few things that could be improved to the NLM algorithms and there are also topics that need some further investigation:

- The results show that $r_S = 10$ and $r_\mathcal{N} = 3$ give the best denoising quality on average for the tested NLM algorithms. These are the largest values tested, which suggests the possibility that larger values of r_S and $r_\mathcal{N}$ could further improve denoising quality. This could be explored in further research.
- In this paper I have mentioned the Poly U dataset with real-world noisy images, though I have not tested the NLM algorithms on these noisy images. Some tests on real-world noisy images could show the impact on denoising quality for SWNLM when the noisy is not artificially normally distributed.
- One disadvantage of SWNLM is that it lacks a denoising method when there are no similar pixels in a pixel's search window. Instead, the noisy pixel is copied as-is to the output image, which can lead to visually distracting patches of noise in the output image. A future improvement could be to use some backup denoising algorithm, such as NLM or a Gaussian filter to reduce the noise in these pixels and make them fit in more with the rest of the denoised image. Alternatively, the value of α could be decreased iteratively until some neighborhoods pass the minimum threshold for similarity.
- For the CUDA NLM versions there are some potential improvements that could explored. For example, the implementations for this paper copy the whole image to the GPU first, then denoise the image and lastly copy the denoised image back to the CPU. This approach does not fully utilize both the cores and bandwidth at the same time, which could be a potential bottleneck. A future version could split the image into chunks, where each chunk could be copied and denoised asynchronously, increasing GPU utilization.
- Though not strictly necessary for SWNLM, an improved implementation and explanation of the SW-test could be beneficial for statistics in general, because the original implementation is arguably hard to follow and it relies heavily on tabulated coefficients. Given the speed of modern hardware, it should be possible to calculate the W and Wp values exactly without approximations. An exact implementation would also remove the upper limit of 5000 for the SW-test, increasing its versatility.

References

[1] A. Antoniadis, G. Oppenheim, and F. Franco-Belgian Meeting of Statisticians (15th : Villard-de Lans. *Wavelets and statistics*. Lecture notes in statistics ; 103. Springer-Verlag, New York, 1994.

[2] A. Buades, B. Coll, and J. Morel. Image Denoising By Non-Local Averaging. In *Proceedings. (ICASSP '05). IEEE International Conference on Acoustics, Speech, and Signal Processing, 2005.*, volume 2, pages 25–28, Philadelphia, Pennsylvania, USA, 2005. IEEE.

[3] A. Eskicioglu and P. Fisher. Image quality measures and their performance. *IEEE Transactions on Communications*, 43(12):2959–2965, Dec. 1995.

[4] Gonzalez, Woods, and Eddins. Imageprocessingplace image databases. `https://imageprocessingplace.com/root_files_V3/image_databases.htm`.

[5] V. Karnati, M. Uliyar, and S. Dey. Fast Non-Local algorithm for image denoising. In *2009 16th IEEE International Conference on Image Processing (ICIP)*, pages 3873–3876, Cairo, Egypt, Nov. 2009. IEEE.

[6] M. Lindenbaum, M. Fischer, and A. Bruckstein. On gabor's contribution to image enhancement. *Pattern Recognition*, 27(1):1–8, 1994.

[7] R. Niwa. shapiro-wilk. `https://github.com/rniwa/js-shapiro-wilk`.

[8] Nvidia. Geforce GTX 1080 whitepaper, 2016.

[9] P. Perona and J. Malik. Scale-space and edge detection using anisotropic diffusion. *IEEE Transactions on Pattern Analysis & Machine Intelligence*, 12(7):629–639, 1990.

[10] P. Royston. Remark AS R94: A Remark on Algorithm AS 181: The W-test for Normality. *Applied Statistics*, 44(4):547, 1995.

[11] S. S. Shapiro and M. B. Wilk. An Analysis of Variance Test for Normality (Complete Samples). *Biometrika*, 52(3/4):591, Dec. 1965.

[12] J. Wang, Y. Guo, Y. Ying, Y. Liu, and Q. Peng. Fast Non-Local Algorithm for Image Denoising. In *2006 International Conference on Image Processing*, pages 1429–1432, Atlanta, GA, Oct. 2006. IEEE.

[13] Z. Wang, A. Bovik, H. Sheikh, and E. Simoncelli. Image Quality Assessment: From Error Visibility to Structural Similarity. *IEEE Transactions on Image Processing*, 13(4):600–612, Apr. 2004.

[14] J. Xu, H. Li, Z. Liang, D. C. Zhang, and L. Zhang. Real-world noisy image denoising: A new benchmark. *ArXiv*, abs/1804.02603, 2018.

[15] W. Yamanappa, P. V. Sudeep, M. K. Sabu, and J. Rajan. Non-Local Means Image Denoising Using Shapiro-Wilk Similarity Measure. *IEEE Access*, 6:66914–66922, 2018.

A Average results for standard images

The table below shows the average results of all standard images for each combination of denoising algorithm, search radius and neighbor radius.

sigma	algorithm	search radius	neighbor radius	avg psnr	avg MSSIM	avg execution time (s)
0	original				1.0000	
10	noisy			28.1623	0.5167	
10	cnlm	5	1	32.5937	0.7954	0.5656
10	cnlm	5	2	32.6590	0.8140	1.0830
10	cnlm	5	3	32.6491	0.8151	1.8975
10	cnlm	8	1	32.9383	0.8089	1.3533
10	cnlm	8	2	33.0117	0.8249	2.5939
10	cnlm	8	3	32.9990	0.8256	4.5270
10	cnlm	10	1	33.0451	0.8115	2.0771
10	cnlm	10	2	33.1263	0.8266	3.9694
10	cnlm	10	3	33.1131	0.8272	6.9164
10	cnlm-cuda	5	1	32.5937	0.7954	0.0151
10	cnlm-cuda	5	2	32.6590	0.8140	0.0254
10	cnlm-cuda	5	3	32.6491	0.8151	0.0446
10	cnlm-cuda	8	1	32.9383	0.8089	0.0391
10	cnlm-cuda	8	2	33.0117	0.8249	0.0619
10	cnlm-cuda	8	3	32.9990	0.8256	0.0994
10	cnlm-cuda	10	1	33.0451	0.8115	0.0833
10	cnlm-cuda	10	2	33.1263	0.8266	0.1010
10	cnlm-cuda	10	3	33.1131	0.8272	0.1478
10	swnlm	5	1	33.0576	0.8120	5.2474
10	swnlm	5	2	32.6393	0.8187	19.7478
10	swnlm	5	3	31.9731	0.8085	45.7859
10	swnlm	8	1	33.2200	0.8149	12.4174
10	swnlm	8	2	32.8718	0.8147	46.9651
10	swnlm	8	3	32.1737	0.8010	109.1304
10	swnlm	10	1	33.2651	0.8155	18.8693
10	swnlm	10	2	32.9605	0.8136	71.4739
10	swnlm	10	3	32.2527	0.7982	166.3518
10	swnlm-cuda	5	1	33.0739	0.8122	0.0473
10	swnlm-cuda	5	2	32.6470	0.8188	0.1179
10	swnlm-cuda	5	3	31.9778	0.8086	0.3340
10	swnlm-cuda	8	1	33.2323	0.8151	0.1005
10	swnlm-cuda	8	2	32.8790	0.8149	0.2620
10	swnlm-cuda	8	3	32.1781	0.8011	0.7958
10	swnlm-cuda	10	1	33.2754	0.8156	0.1471
10	swnlm-cuda	10	2	32.9675	0.8137	0.3985
10	swnlm-cuda	10	3	32.2570	0.7983	1.2068
20	noisy			22.2097	0.3089	
20	cnlm	5	1	28.3995	0.5813	0.5514
20	cnlm	5	2	28.7504	0.6120	1.0690
20	cnlm	5	3	28.7687	0.6142	1.8600
20	cnlm	8	1	28.7954	0.5963	1.2939

20	cnlm	8	2	29.2026	0.6292	2.5435
20	cnlm	8	3	29.2233	0.6315	4.4333
20	cnlm	10	1	28.8806	0.5986	1.9661
20	cnlm	10	2	29.3077	0.6319	3.8739
20	cnlm	10	3	29.3295	0.6342	6.7559
20	cnlm-cuda	5	1	28.3995	0.5813	0.0153
20	cnlm-cuda	5	2	28.7504	0.6120	0.0251
20	cnlm-cuda	5	3	28.7687	0.6142	0.0441
20	cnlm-cuda	8	1	28.7954	0.5963	0.0389
20	cnlm-cuda	8	2	29.2026	0.6292	0.0614
20	cnlm-cuda	8	3	29.2233	0.6315	0.0990
20	cnlm-cuda	10	1	28.8806	0.5986	0.0830
20	cnlm-cuda	10	2	29.3077	0.6319	0.1005
20	cnlm-cuda	10	3	29.3295	0.6342	0.1481
20	swnlm	5	1	29.3860	0.6370	5.2893
20	swnlm	5	2	29.3407	0.6900	20.0019
20	swnlm	5	3	28.6353	0.6834	46.2548
20	swnlm	8	1	29.5799	0.6428	12.5395
20	swnlm	8	2	29.6986	0.6984	47.6614
20	swnlm	8	3	28.9904	0.6861	110.6253
20	swnlm	10	1	29.6175	0.6426	19.0508
20	swnlm	10	2	29.8178	0.7005	72.5798
20	swnlm	10	3	29.1165	0.6858	168.8358
20	swnlm-cuda	5	1	29.3946	0.6373	0.0450
20	swnlm-cuda	5	2	29.3484	0.6902	0.1113
20	swnlm-cuda	5	3	28.6397	0.6835	0.3270
20	swnlm-cuda	8	1	29.5844	0.6430	0.0984
20	swnlm-cuda	8	2	29.7051	0.6985	0.2555
20	swnlm-cuda	8	3	28.9945	0.6862	0.7810
20	swnlm-cuda	10	1	29.6209	0.6427	0.1435
20	swnlm-cuda	10	2	29.8234	0.7006	0.3890
20	swnlm-cuda	10	3	29.1206	0.6859	1.1914
30	noisy			18.8050	0.2054	
30	cnlm	5	1	25.8679	0.4467	0.5514
30	cnlm	5	2	26.4066	0.4769	1.0696
30	cnlm	5	3	26.4430	0.4791	1.8594
30	cnlm	8	1	26.2318	0.4570	1.2935
30	cnlm	8	2	26.8243	0.4894	2.5410
30	cnlm	8	3	26.8623	0.4918	4.4243
30	cnlm	10	1	26.2888	0.4582	1.9651
30	cnlm	10	2	26.8910	0.4910	3.8766
30	cnlm	10	3	26.9291	0.4933	6.7469
30	cnlm-cuda	5	1	25.8679	0.4467	0.0155
30	cnlm-cuda	5	2	26.4066	0.4769	0.0251
30	cnlm-cuda	5	3	26.4430	0.4791	0.0444
30	cnlm-cuda	8	1	26.2318	0.4570	0.0391
30	cnlm-cuda	8	2	26.8243	0.4894	0.0615
30	cnlm-cuda	8	3	26.8622	0.4918	0.0988
30	cnlm-cuda	10	1	26.2888	0.4582	0.0831
30	cnlm-cuda	10	2	26.8910	0.4910	0.1009

30	cnlm-cuda	10	3	26.9291	0.4933	0.1478
30	swnlm	5	1	26.9337	0.5055	5.2983
30	swnlm	5	2	27.3897	0.5757	20.0903
30	swnlm	5	3	26.8358	0.5802	46.4013
30	swnlm	8	1	27.1280	0.5106	12.5935
30	swnlm	8	2	27.7670	0.5892	47.9163
30	swnlm	8	3	27.2936	0.5950	111.1893
30	swnlm	10	1	27.1642	0.5103	19.1386
30	swnlm	10	2	27.8758	0.5920	73.0161
30	swnlm	10	3	27.4474	0.5969	169.6403
30	swnlm-cuda	5	1	26.9409	0.5058	0.0448
30	swnlm-cuda	5	2	27.3960	0.5759	0.1125
30	swnlm-cuda	5	3	26.8401	0.5803	0.3268
30	swnlm-cuda	8	1	27.1318	0.5107	0.0980
30	swnlm-cuda	8	2	27.7717	0.5894	0.2548
30	swnlm-cuda	8	3	27.2974	0.5951	0.7773
30	swnlm-cuda	10	1	27.1671	0.5104	0.1405
30	swnlm-cuda	10	2	27.8797	0.5921	0.3853
30	swnlm-cuda	10	3	27.4509	0.5970	1.1845
40	noisy			16.4927	0.1453	
40	cnlm	5	1	24.1120	0.3629	0.5496
40	cnlm	5	2	24.7465	0.3900	1.0699
40	cnlm	5	3	24.7908	0.3920	1.8595
40	cnlm	8	1	24.4328	0.3685	1.2920
40	cnlm	8	2	25.1099	0.3972	2.5434
40	cnlm	8	3	25.1545	0.3993	4.4259
40	cnlm	10	1	24.4634	0.3687	1.9646
40	cnlm	10	2	25.1439	0.3975	3.8773
40	cnlm	10	3	25.1879	0.3996	6.7478
40	cnlm-cuda	5	1	24.1120	0.3629	0.0150
40	cnlm-cuda	5	2	24.7465	0.3900	0.0256
40	cnlm-cuda	5	3	24.7908	0.3920	0.0440
40	cnlm-cuda	8	1	24.4328	0.3685	0.0391
40	cnlm-cuda	8	2	25.1099	0.3972	0.0619
40	cnlm-cuda	8	3	25.1545	0.3993	0.0993
40	cnlm-cuda	10	1	24.4635	0.3687	0.0838
40	cnlm-cuda	10	2	25.1439	0.3975	0.1014
40	cnlm-cuda	10	3	25.1879	0.3996	0.1481
40	swnlm	5	1	24.9877	0.4105	5.3056
40	swnlm	5	2	25.7223	0.4749	20.1380
40	swnlm	5	3	25.3016	0.4748	46.5406
40	swnlm	8	1	25.1686	0.4142	12.5936
40	swnlm	8	2	26.1092	0.4907	48.0044
40	swnlm	8	3	25.8110	0.4958	111.3663
40	swnlm	10	1	25.1952	0.4136	19.1606
40	swnlm	10	2	26.2068	0.4935	73.1481
40	swnlm	10	3	25.9478	0.4987	169.9409
40	swnlm-cuda	5	1	24.9959	0.4108	0.0428
40	swnlm-cuda	5	2	25.7282	0.4752	0.1123
40	swnlm-cuda	5	3	25.3055	0.4750	0.3241

40	swnlm-cuda	8	1	25.1734	0.4144	0.0981
40	swnlm-cuda	8	2	26.1136	0.4909	0.2534
40	swnlm-cuda	8	3	25.8140	0.4959	0.7775
40	swnlm-cuda	10	1	25.1990	0.4137	0.1420
40	swnlm-cuda	10	2	26.2105	0.4936	0.3835
40	swnlm-cuda	10	3	25.9504	0.4988	1.1823
50	noisy			14.7639	0.1070	
50	cnlm	5	1	22.7952	0.3046	0.5514
50	cnlm	5	2	23.4779	0.3289	1.0690
50	cnlm	5	3	23.5255	0.3307	1.8559
50	cnlm	8	1	23.0622	0.3061	1.2928
50	cnlm	8	2	23.7716	0.3311	2.5448
50	cnlm	8	3	23.8182	0.3329	4.4209
50	cnlm	10	1	23.0660	0.3052	1.9618
50	cnlm	10	2	23.7738	0.3303	3.8774
50	cnlm	10	3	23.8190	0.3320	6.7420
50	cnlm-cuda	5	1	22.7952	0.3046	0.0150
50	cnlm-cuda	5	2	23.4779	0.3289	0.0253
50	cnlm-cuda	5	3	23.5255	0.3307	0.0441
50	cnlm-cuda	8	1	23.0622	0.3061	0.0389
50	cnlm-cuda	8	2	23.7716	0.3311	0.0610
50	cnlm-cuda	8	3	23.8182	0.3329	0.0988
50	cnlm-cuda	10	1	23.0660	0.3052	0.0821
50	cnlm-cuda	10	2	23.7738	0.3303	0.1009
50	cnlm-cuda	10	3	23.8190	0.3320	0.1474
50	swnlm	5	1	23.3195	0.3372	5.3106
50	swnlm	5	2	24.1447	0.3909	20.1486
50	swnlm	5	3	23.5996	0.3824	46.5210
50	swnlm	8	1	23.4829	0.3395	12.6084
50	swnlm	8	2	24.5361	0.4064	48.1021
50	swnlm	8	3	24.1260	0.4053	111.4158
50	swnlm	10	1	23.5049	0.3391	19.1789
50	swnlm	10	2	24.6388	0.4096	73.3171
50	swnlm	10	3	24.2764	0.4102	170.1825
50	swnlm-cuda	5	1	23.3299	0.3376	0.0445
50	swnlm-cuda	5	2	24.1516	0.3912	0.1108
50	swnlm-cuda	5	3	23.6036	0.3826	0.3241
50	swnlm-cuda	8	1	23.4893	0.3397	0.0979
50	swnlm-cuda	8	2	24.5413	0.4066	0.2515
50	swnlm-cuda	8	3	24.1293	0.4055	0.7764
50	swnlm-cuda	10	1	23.5100	0.3393	0.1410
50	swnlm-cuda	10	2	24.6434	0.4097	0.3839
50	swnlm-cuda	10	3	24.2792	0.4103	1.1834
60	noisy			13.4277	0.0819	
60	cnlm	5	1	21.7982	0.2625	0.5486
60	cnlm	5	2	22.4977	0.2841	1.0704
60	cnlm	5	3	22.5459	0.2857	1.8588
60	cnlm	8	1	22.0064	0.2602	1.2923
60	cnlm	8	2	22.7179	0.2818	2.5428
60	cnlm	8	3	22.7642	0.2834	4.4220

60	cnlm	10	1	21.9838	0.2583	1.9633
60	cnlm	10	2	22.6890	0.2799	3.8764
60	cnlm	10	3	22.7333	0.2813	6.7444
60	cnlm-cuda	5	1	21.7982	0.2625	0.0151
60	cnlm-cuda	5	2	22.4977	0.2841	0.0254
60	cnlm-cuda	5	3	22.5459	0.2857	0.0443
60	cnlm-cuda	8	1	22.0064	0.2602	0.0386
60	cnlm-cuda	8	2	22.7179	0.2818	0.0608
60	cnlm-cuda	8	3	22.7642	0.2834	0.0994
60	cnlm-cuda	10	1	21.9838	0.2583	0.0830
60	cnlm-cuda	10	2	22.6890	0.2799	0.1008
60	cnlm-cuda	10	3	22.7333	0.2813	0.1473
60	swnlm	5	1	21.8537	0.2811	5.3124
60	swnlm	5	2	22.6468	0.3220	20.1609
60	swnlm	5	3	21.8215	0.3031	46.5449
60	swnlm	8	1	21.9954	0.2823	12.6109
60	swnlm	8	2	23.0597	0.3359	48.1309
60	swnlm	8	3	22.4434	0.3264	111.5169
60	swnlm	10	1	22.0049	0.2816	19.1984
60	swnlm	10	2	23.1724	0.3385	73.3346
60	swnlm	10	3	22.6327	0.3318	170.2324
60	swnlm-cuda	5	1	21.8670	0.2815	0.0440
60	swnlm-cuda	5	2	22.6552	0.3223	0.1121
60	swnlm-cuda	5	3	21.8259	0.3033	0.3250
60	swnlm-cuda	8	1	22.0040	0.2825	0.0985
60	swnlm-cuda	8	2	23.0662	0.3361	0.2523
60	swnlm-cuda	8	3	22.4472	0.3266	0.7823
60	swnlm-cuda	10	1	22.0120	0.2818	0.1434
60	swnlm-cuda	10	2	23.1782	0.3387	0.3878
60	swnlm-cuda	10	3	22.6362	0.3319	1.1883

B Execution times for Poly U images

image	resolution	algorithm	execution time (s)
book	324x216	swnlm	45.538
book	324x216	cnlm	1.782
book	324x216	swnlm-cuda	0.32
book	324x216	cnlm-cuda	0.04
book	648x432	swnlm	182.21
book	648x432	cnlm	7.154
book	648x432	swnlm-cuda	1.272
book	648x432	cnlm-cuda	0.158
book	1296x864	swnlm	729.619
book	1296x864	cnlm	28.614
book	1296x864	swnlm-cuda	5.09
book	1296x864	cnlm-cuda	0.623
book	2592x1728	swnlm	2920.597
book	2592x1728	cnlm	114.4
book	2592x1728	swnlm-cuda	20.407

book	2592x1728	cnlm-cuda	2.538
toy	324x216	swnlm	45.515
toy	324x216	cnlm	1.791
toy	324x216	swnlm-cuda	0.33
toy	324x216	cnlm-cuda	0.041
toy	648x432	swnlm	182.014
toy	648x432	cnlm	7.159
toy	648x432	swnlm-cuda	1.262
toy	648x432	cnlm-cuda	0.159
toy	1296x864	swnlm	729.812
toy	1296x864	cnlm	28.604
toy	1296x864	swnlm-cuda	5.145
toy	1296x864	cnlm-cuda	0.66
toy	2592x1728	swnlm	2922.723
toy	2592x1728	cnlm	114.444
toy	2592x1728	swnlm-cuda	20.307
toy	2592x1728	cnlm-cuda	2.547
water	324x216	swnlm	45.557
water	324x216	cnlm	1.789
water	324x216	swnlm-cuda	0.314
water	324x216	cnlm-cuda	0.039
water	648x432	swnlm	182.294
water	648x432	cnlm	7.16
water	648x432	swnlm-cuda	1.254
water	648x432	cnlm-cuda	0.157
water	1296x864	swnlm	730.536
water	1296x864	cnlm	28.59
water	1296x864	swnlm-cuda	4.997
water	1296x864	cnlm-cuda	0.622
water	2592x1728	swnlm	2922.537
water	2592x1728	cnlm	114.428
water	2592x1728	swnlm-cuda	20.201
water	2592x1728	cnlm-cuda	2.537