Efficient Implementation of edge-preserving Denoising Filters on GPU

A Project Report Submitted
in Partial Fulfillment of the Requirements
for the Degree of

BACHELOR OF TECHNOLOGY

in

COMPUTER SCIENCE AND ENGINEERING

by

Kushagra Indurkhya: CS19B1017
Yashwanth Vallabhu: CS19B1030

ಭಾರತೀಯ ಮಾಹಿತಿ ತಂತ್ರಜ್ಞಾನ ಸಂಸ್ಥೆ ರಾಯಚೂರು
भारतीय सूचना प्रौद्योगिकी संस्थान रायचूर
Indian Institute of Information Technology Raichur

To

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
INDIAN INSTITUTE OF INFORMATION TECHNOLOGY
RAICHUR - 584135,
INDIA

May 2023

DECLARATION

We, **Kushagra Indurkhya** (**Roll No: CS19B1017**),**Yashwanth Vallabhu(Roll No: CS19B1030)** , hereby declare that this report entitled **"Efficient Implementation of edge-preserving Denoising Filters on GPU"** submitted to the Indian Institute of Information Technology Raichur towards the partial requirement of **Bachelor of Technology** in **Computer Science and Engineering** is an original work carried out by us in a group of two under the supervision of **Dr. Debasish Mukherjee** and has not formed the basis for the award of any degree or diploma, in this or any other institution or university. We have sincerely tried to uphold academic ethics and honesty. Whenever an external information or statement or result is used, that has been duly acknowledged and cited.

[Kushagra Indurkhya]

Raichur-584135

May 2023

[Yashwanth Vallabhu]

Raichur-584135

May 2023

CERTIFICATE

This is to certify that the work contained in this project report entitled **"Efficient Implementation of edge-preserving Denoising Filters on GPU"** submitted by **Kushagra Indurkhya (Roll No: CS19B1017)** to the Indian Institute of Information Technology Raichur towards the partial requirement of **Bachelor of Technology** in **Computer Science and Engineering** has been carried out by him under my supervision and that it has not been submitted elsewhere for the award of any degree.

Raichur-584135
May 2023

Dr. Debasish Mukherjee
Project Supervisor(s)

ABSTRACT

Image denoising is a critical step in computer vision and image processing that has a wide range of applications, from medical imaging to autonomous driving. Nonlinear edge-preserving filters have proven to be effective in removing noise while retaining edge information, but they can be computationally intensive. To address this, our project implements three advanced algorithms - bilateral filter, adaptive bilateral filter, and non-local means bilateral filter - on CUDA GPUs to achieve efficient performance for large-scale images, the project also contains an in-depth analysis of the current state-of-the-art techniques, Based on this analysis, some variations,algorithmic designs were employed to keep the algorithms simple yet effective.

We employed several techniques to optimize the GPU implementation for improved performance. One of the primary optimizations we used is shared memory utilization, which enables faster data access and reduces global memory access latency, the other techniques include precomputing and caching expensive arithmetic operations. We also parallelized the algorithms to leverage the immense parallelism offered by GPU architecture. Memory coalescing was used to ensure that threads access data from consecutive memory locations, which improves data transfer and access performance.

To evaluate our implementation, we tested it on various images and compared its performance to a CPU-based approach. Our results show that the GPU implementation achieved significant speedup compared to the CPU-based approach, making it a viable solution for real-time image-denoising applications. Our project contributes to the field of computer vision and image processing by providing efficient implementations of advanced image-denoising algorithms on GPUs, which can facilitate further research and enable real-time applications.

List of Figures

List of Tables

List of Algorithms

Contents

Chapter 1

Introduction

Image processing is a broad and complicated field with applications in anything from picture enhancement to medical imaging. Denoising, which is taking out noise from a picture, is one of the most often performed jobs in image processing. Film grain, sensor noise, and electrical noise are just a few examples of the many things that can generate noise. Denoising is crucial for enhancing an image's quality since it might make recognizing objects and perceiving details simpler.

Several different denoising algorithms exist, each with unique benefits and drawbacks. The Gaussian filtering algorithm, which computes the weighted average of nearby pixels and is one of the most widely used denoising algorithms, has some drawbacks. A bilateral filter, an improved algorithm, is used to get around these drawbacks. Bilateral filter considers both the spatial and intensity information of nearby pixels because of this, it works well for maintaining edges and other crucial details in an image. But the computational cost of the bilateral filter can be high, especially for large images. This is due to the bilateral filter's requirement for several calculations to be made for each image pixel.

Adobe Photoshop and GIMP are two popular image editing software that implement bilateral filters in their respective tools. In Adobe Photoshop, the bilateral filter is implemented in the Surface Blur tool, while in GIMP, it is implemented in the Filters $\rightarrow$ Blur tools and is called Selective Gaussian Blur. Additionally, the free G'MIC plugin for GIMP adds more control to the bilateral filter through its Repair $\rightarrow$ Smooth tool.

In recent years, GPUs and CUDA have become increasingly popular for image processing. This is due to the fact that GPUs are highly parallel devices that are well-suited for performing computationally intensive tasks such as image processing.

GPUs are made up of thousands of smaller processors called cores. These cores can work independently of each other, which allows for high levels of parallelism. This makes GPUs well-suited for tasks that require a large number of calculations to be performed, such as image processing.

CUDA, or Compute Unified Device Architecture, is a parallel computing platform de-

veloped by Nvidia that allows developers to program GPUs for general-purpose computing. CUDA has become a popular choice for image processing, as it allows developers to accelerate image processing algorithms by taking advantage of the parallel processing capabilities of GPUs. GPUs can be used to accelerate the bilateral filter, by performing the calculations in parallel. This can significantly improve the performance of the bilateral filter, making it suitable for applications that require real-time processing, such as video conferencing and medical imaging, and time-critical applications like self-driving cars.

In this project, we have implemented several image processing filters on a GPU using the CUDA programming language. We will focus on the bilateral filter, the non-local means filter, and the adaptive bilateral filter. These filters are all well-suited for smoothing images while preserving edges, we have implemented their sequential and parallelized CPU implementations and implemented them on GPU as well along with several optimization techniques some of which are novel, providing significant performance speedup to the algorithms.

1.1 Problem Statement

This project aims to work on the following problem statements:

- Identify edge-preserving effective denoising algorithms, and their variations and design some tweaks in the algorithms which can be parallelized to efficiently provide edge-preserving denoising for real-time applications like medical imaging, video processing, and computer vision.
- Implement the algorithms efficiently using CUDA programming to harness the massively parallel power of modern GPUs
- Ideate and implement the optimizations that can be done to GPU implementation for more efficient usage of the cores, and memory to reduce the latencies.
- Compare the performance of the optimized algorithm against the traditional CPU implementations using sequential and parallel paradigms.

1.2 Outline of this report

- The Introduction section includes a Problem Statement and an Outline of the report.
- The Image Denoising section defines noise and describes the different types of noise. It also explains the concept of image denoising and outlines the features of a good denoising system. The section also covers traditional denoising methods, such as Box Filter, Gaussian Filter, and Median Filter.
- The Leveraging the Power of GPU section explains why parallelization is necessary, describes the different types of parallelization and discusses the advantages and challenges of parallel processing. It introduces the concept of Graphics Processing Unit (GPU) and provides an overview of CUDA, a parallel computing platform and programming model developed by NVIDIA.
- The Algorithms and Related Work section covers three algorithms used for image denoising: Bilateral Filter and Adaptive Bilateral Filter, nonlocal means bilateral. It explains the idea behind each algorithm, its design, variations, tweaks made, and related works. The section also highlights the applications of Bilateral Filters in image denoising.
- The Methodology section describes the experimental setup used to evaluate the performance of the Bilateral Filter and Adaptive Bilateral Filter on CPU and GPU. It provides details on the implementation and design levels. The section also explains the optimization techniques used to improve the performance of the algorithms
- The Experiments and Results section presents the results of the experiments conducted to compare the performance of the algorithms on CPU and GPU. It provides a detailed analysis of the results obtained, such as execution time, speedup, and efficiency. The section also includes graphs and tables to illustrate the performance of the algorithms under different conditions, such as image size, and filter size.
- The Conclusions and future works section presents an analysis and summary of the performances of the algorithms and their GPU implementations and discusses how the work of this project can be extended.
- In Appendix A, The My Contribution section Highlights my individual contribution along with the associated timelines for each task. This provides a comprehensive overview of my involvement in the project, emphasizing the technical aspects of my work and the milestones achieved.

Chapter 2

Image Denoising

2.1 What is noise?

Noise is defined as a random or patterned variation or disturbance in the image signal. Image noise is a random variation of brightness or color information in images, and is usually an aspect of electronic noise. It can be produced by the image sensor and circuitry of a scanner or digital camera. Image noise can also originate in film grain and in the unavoidable shot noise of an ideal photon detector. Noise can be caused by a number of factors, including poor lighting conditions, high ISO settings, long exposure times, and heat.

Figure 2.1: Black and white image of IIITR Campus as base image to demonstrate the effect of different types of noises

2.2 Types of noise

To explain the different types of noise let's take a look at the effect of different types of noises when programmatically simulated and applied to the Figure 2.1

- **Salt-and-Pepper Noise**: Salt-and-pepper noise, also known as impulse noise, is a type of random noise that occurs as isolated pixels with very high or very low-intensity values compared to the surrounding pixels. It appears as scattered bright and dark pixels in the image, resembling salt and pepper sprinkled on it. Salt-and-pepper noise can be caused by sensor malfunctions, data transmission errors, or other sources of random signal corruption.

Figure 2.2: Salt and pepper (impulse) noise

- **Gaussian Noise**: Gaussian noise is an additive noise that follows a Gaussian or normal distribution. It is characterized by random fluctuations in intensity values around the mean value, resulting in a smooth and symmetric distribution of noise. Gaussian noise is often encountered in images due to electronic sensor noise in digital cameras, thermal noise in imaging sensors, or noise introduced during image transmission and storage.

Figure 2.3: Gaussian noise

- **Quantization Noise**: Quantization noise is a type of noise that occurs during the digitization process of continuous analog signals, such as in image acquisition or compression. It is caused by the finite precision of digital representation, and it appears as discrete steps

or rounding errors in the quantized image. Quantization noise can result in the loss of image details and reduced image quality.

It can be simulated by reducing the number of intensity levels (or bit depth) in a grayscale image, resulting in a loss of detail and an increase in noise.

Figure 2.4: Quantization noise

- **Speckle Noise**: Speckle noise is a type of noise that is characterized by the presence of randomly distributed bright and dark spots in an image. It is often caused by the scattering of light waves as they pass through a medium, such as air or water. Speckle noise can be a significant problem in a variety of applications, such as medical imaging, remote sensing, and astronomy.

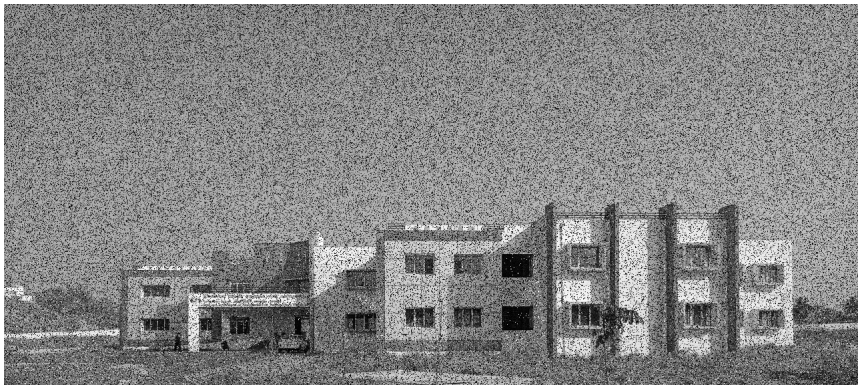


Figure 2.5: Speckle Noise

2.3 Image denoising

Image denoising is a critical step in image processing that removes noise from digital images.

Image denoising is a fundamental problem in image processing and computer vision. It has numerous applications in various fields such as medical imaging, surveillance, and astronomy. In medical imaging, for example, the quality of an image can impact the accuracy of diagnosis and treatment. In surveillance, removing noise from a video feed can improve the ability to detect and identify objects. In astronomy, image denoising can help reveal faint details of astronomical objects that would otherwise be obscured by noise.

There are various approaches to image denoising, including traditional methods such as filtering and wavelet analysis, as well as deep learning-based methods. Traditional methods are based on mathematical algorithms that analyze the image data and remove noise based on specific criteria. Filtering, for example, involves applying a mathematical filter to the image to remove noise. Wavelet analysis, on the other hand, involves decomposing the image into different frequency bands and selectively removing noise from each band.

2.4 Features of a good denoising system

- **Noise reduction**: Smoothing helps to attenuate or eliminate high-frequency noise components in an image, which are typically responsible for the grainy or speckled appearance of noisy images. By reducing the noise, smoothing can improve the visual quality of the denoised image and enhance the accuracy of subsequent image analysis or processing tasks.
- **Preserve Image Features**: One of the most important goals of a good denoising system is maintaining the integrity of edges, lines, and other structures in the image, preventing them from being overly blurred or distorted during the denoising process. This is particularly important in applications where preserving the fine details or boundaries in the image is critical, such as in medical imaging or computer vision tasks.
- **Speed**: The speed of a denoising system is measured by the time it takes to denoise an image. There are a number of factors that can affect the speed of a denoising system, including the type of denoising algorithm, the size of the image, and the hardware platform. In general, simple denoising algorithms, such as the bilateral filter, are faster than more complex algorithms, such as non-local means (NLM). However, simple denoising algorithms may not be as effective as more complex algorithms. Some applications where the speed of the algorithm is of utmost importance are:
  * **Denoise medical image**: Medical images are often noisy, which can make it difficult to diagnose diseases. The bilateral filter can be used to remove noise

from medical images, making it easier for doctors to see the details of the image.

* **Improve the quality of video conferencing**: Video conferencing is a time-sensitive application, as it requires real-time processing of video images. The bilateral filter can be used to improve the quality of video conferencing by removing noise from the images, making them sharper and more clear.
* **Enhance the quality of surveillance footage**: Surveillance footage is often noisy, which can make it difficult to identify people or objects. The bilateral filter can be used to improve the quality of surveillance footage by removing noise from the images, making it easier to identify people and objects.

2.5 Some traditional denoising methods

2.5.1 Basics and Notation

2.5.1.1 Image

An image is a two-dimensional array of pixels, where each pixel represents a single point in the image. Each pixel is typically represented by a set of values, such as the intensity of the red, green, and blue color channels in a color image, or a single intensity value in a black-and-white image.

In the case of a black and white image, each pixel is represented by a single 8-bit value that indicates its intensity, where 0 represents pure black and 255 represents pure white. This value can be stored in an array, where the position of each value corresponds to the position of the corresponding pixel in the image. Therefore, a black-and-white image can be represented as an array of 8-bit intensity values. For all further references in this report and our project image is taken as a 2d array of 8-bit intensity values since we will be dealing with black and white images.

2.5.1.2 Convolution

In image processing, convolution is a mathematical operation that is commonly used for filtering an image. The operation involves multiplying a small matrix, known as the kernel or filter, with the corresponding pixel values in the image, and summing the results. The result of this operation is a new image with the filtered pixel values.

The kernel is centered over the pixel and the product of the kernel values and corresponding pixel values in the image are summed up. The resulting value is then placed in the

output image at the same pixel location (x,y). This process is repeated for every pixel in the image.

The values in the convolution kernel determine the specific operation being performed. For example, a Gaussian kernel can be used to smooth the image and reduce noise, while an edge detection kernel can be used to enhance edges in the image.

Mathematically, the convolution of an image $f(x,y)$ with a kernel $h(x,y)$ is defined as:

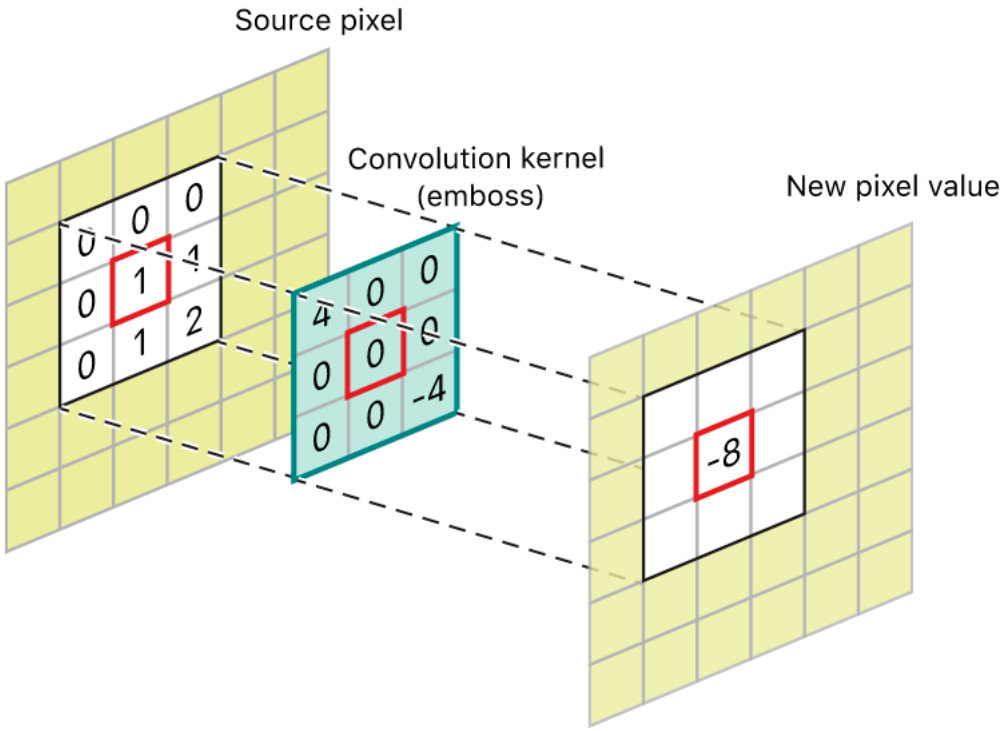
$$(f * h)(x,y) = \sum_{i=-\infty}^{\infty} \sum_{j=-\infty}^{\infty} f(i,j)h(x-i,y-j)$$

where (x,y) are the coordinates of the output pixel, i, j are the coordinates of the kernel, and $*$ denotes the convolution operation.

The kernel is typically a small, square matrix with odd dimensions (e.g., 3x3, 5x5, etc.), where the center element of the matrix corresponds to the output pixel. The kernel is also known as a filter or mask and contains values that define the filtering operation to be performed. The size of the kernel also affects the operation. A larger kernel size will have a more pronounced effect on the output image, while a smaller kernel size will have a more subtle effect. However, using larger kernel sizes can also increase computation time and may cause blurring or other unwanted effects if not carefully selected.

During convolution, the kernel is slid over the image, and at each location, the corresponding pixel values in the image are multiplied by the corresponding kernel values, and the resulting products are summed. The resulting value is then placed in the output image at the corresponding location.

Convolution can be used for a variety of image processing operations, such as blurring, sharpening, edge detection, and noise reduction. It is a fundamental operation in many computer vision and image processing applications.



Source:
https://developer.apple.com/documentation/accelerate/blurring_an_image

Figure 2.6: Example of convolving a 3*3 kernel

2.6 Some Conventional denoising algorithms

2.6.1 Box Filter

Box filters are commonly used for simple image smoothing tasks, such as noise reduction and blurring. They are computationally efficient and easy to implement but can cause significant loss of image detail and edges. It works by replacing each pixel value with the average value of its neighboring pixels, which are located within a square or rectangular region centered around the pixel. The size of this region is specified by a kernel or window, which determines the amount of smoothing or blurring applied to the image. The kernel for a box filter is a square matrix of identical values, typically with odd dimensions, such as 3x3, 5x5, or 7x7. The center pixel of the kernel corresponds to the pixel being filtered, and the other values in the kernel represent the neighboring pixels. Here's an example of a 3*3 kernel

$$\begin{bmatrix} \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \\ \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \\ \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \end{bmatrix}$$

2.6.2 Gaussian Filter

Gaussian filter is a linear smoothing filter that is widely used in image processing and computer vision. It is a type of low-pass filter that can effectively remove high-frequency noise from an image while preserving the edges and structures. The filter works by convolving the image with a Gaussian kernel, which is a 2D distribution with a bell-shaped curve. The Gaussian kernel assigns a weight to each pixel in the image, based on its distance from the center of the kernel. The closer a pixel is to the center, the higher its weight, and the farther away it is, the lower its weight.

The formula for the Gaussian kernel is:

$$G(x,y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}$$

where (x,y) are the coordinates of the kernel center, and σ is the standard deviation of the Gaussian distribution. The size of the kernel is determined by the value of σ, with larger values resulting in larger kernels. The Gaussian filter is especially effective at removing gaussian noise. To apply the Gaussian filter to an image, the image is first convolved with the Gaussian kernel using the 2D convolution operation. The resulting image is a smoothed version of the original image, with the high-frequency noise suppressed.

Gaussian filter only considers the spatial proximity of pixels using a single Gaussian kernel. It blurs an image uniformly, without considering the intensity differences between neighboring pixels. As a result, the Gaussian filter can blur edges and details in an image, as it smooths the

intensity values without distinguishing between edges and non-edges. The degree of blurring in a Gaussian filter is determined by the width or standard deviation of the Gaussian kernel, with larger values resulting in more blurring.

2.6.3 Median Filter

The median filter is a nonlinear image filtering technique used for removing noise from an image. The median filter replaces the pixel value at each position in the image with the median value of its neighboring pixels. Unlike linear filters such as the box filter or Gaussian filter, the median filter does not use a weighted average of neighboring pixel values but instead relies on the statistical properties of the data.

The median filter is especially effective in removing impulse noise or salt and pepper noise, which randomly adds white or black pixels to the image. The filter works by first selecting a window of neighboring pixels centered around the pixel being filtered. The pixel values within the window are then sorted and the median value is selected as the new value for the pixel being filtered.

The drawback of the median filter is that it causes blurring of edges and other fine details in the image. It is also computationally expensive for larger window sizes, making it less practical for real-time applications on high-resolution images.

Chapter 3

Leveraging the Power of GPU

3.1 Why Parallelization?

Over the years, as technology advanced, the demand for computational power, and speed increased. As the number of cores of a processor increased (Moore's law), parallelism was introduced to execute multiple instructions simultaneously. Traditionally, the processing took place sequentially, where instructions are executed one at a time. Parallel processing helps in faster execution of the program as we can divide the task into multiple sub-tasks which can be executed concurrently by utilizing the cores which are available. Parallel processing is significantly used in our daily life as it is widely used for real-time applications, medical applications, etc. which require faster analysis of data. One of the main applications of parallel processing is in AI, machine learning, and deep learning as they involve working with huge datasets, and complex mathematical computations, which can be executed efficiently, and faster when computed in parallel.

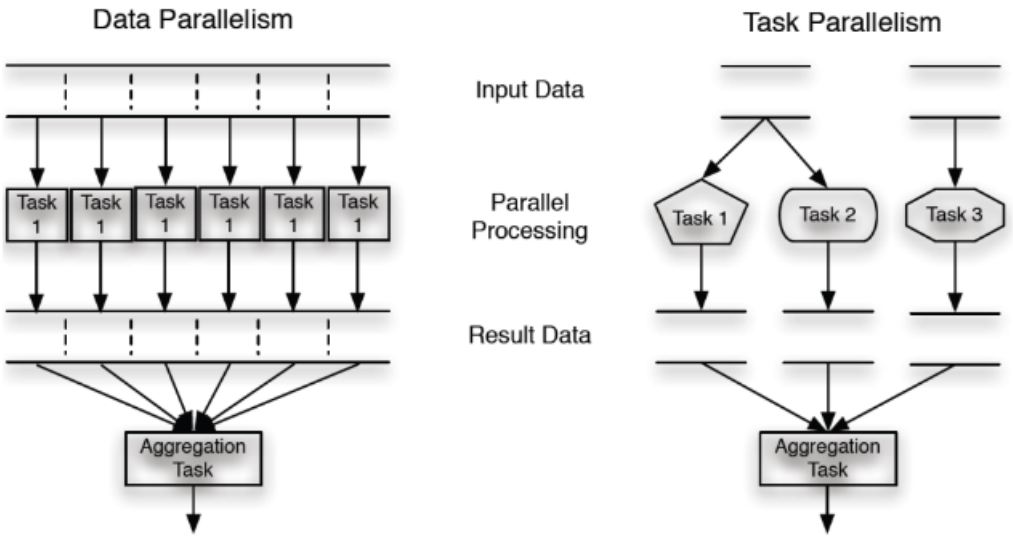
3.2 Types of Parallelization

There are two main types of parallelization:

- **Task Parallelism**: It involves dividing the program into smaller independent sub-tasks which can be executed concurrently on multiple processors. One of the examples where we can use this type of parallelism is while testing in software development, where many tests if divided into multiple independent tests, can be executed concurrently which helps reduce the time taken for testing.

- **Data Parallelism**: It involves dividing a large dataset into smaller datasets, so that computations can be performed concurrently. One of the examples where this can be used is

in applications like image, video processing where we need to perform repetitive computations on some large data, data parallelism helps in reducing the time taken to process.



Source: `https://osm.hpi.de/parProg/2012/05_progmodels.pdf`

Figure 3.1: Data parallelism and Task parallelism

Other types of parallelism include:

- **Pipeline Parallelism**: It involves dividing a task into multiple stages, where each stage is processed simultaneously on different processors. An analogy for this type of parallelism would be the assembly line of a car, where each stage does its processing independently, after that uses the result of the previous stage if necessary. One of the main challenges of pipeline parallelism is that its performance is dependent on the slowest sub-task as a sub-task uses the result of another sub-task.
- **Bit-level Parallelism**: It involves processing multiple bits of data simultaneously. It is very useful in tasks which involve large amounts of data which can be processed in parallel like image/video processing.

3.3 Advantages and applications of parallel processing:

As the tasks are executed concurrently, parallel processing saves the time taken for the execution of the program. It also helps in saving costs as the resources are used efficiently. Almost every application that we see in our daily life tries to use parallel processing as it helps provide real-time experience to the user in contrast serial processing takes a significant amount of time

which results in a bad user experience. One of the main applications of parallel processing can be observed in the medical industry, where it helps in generating faster, accurate diagnosis of reports which improves patient healthcare, and reduces costs. Parallel processing also plays a crucial role in the space industry where it is used for simulating, and designing optimal trajectories, for faster communication between satellites and the ground stations, processing and analyzing the data being sent by the satellites, etc.

3.4 Challenges of parallel processing:

Despite its many benefits, parallel processing also has many pitfalls as applications should be carefully designed so that they work correctly. There are many problems to consider such as synchronization, deadlocks, race conditions, etc.

- **Race condition**: This occurs when multiple threads try to access and modify the same data (ex: variable), resulting in unpredictable/inconsistent behavior. To avoid this, we need to use locks so that mutual exclusion is ensured whenever multiple threads try to modify the same data.

- **Deadlock**: This occurs when two or more threads are waiting for each other to acquire a shared resource which results in a state where no thread proceeds further. They cause applications to become unresponsive. These are algorithms that detect these deadlock situations (like using a resource dependency graph and detecting a cycle in it).

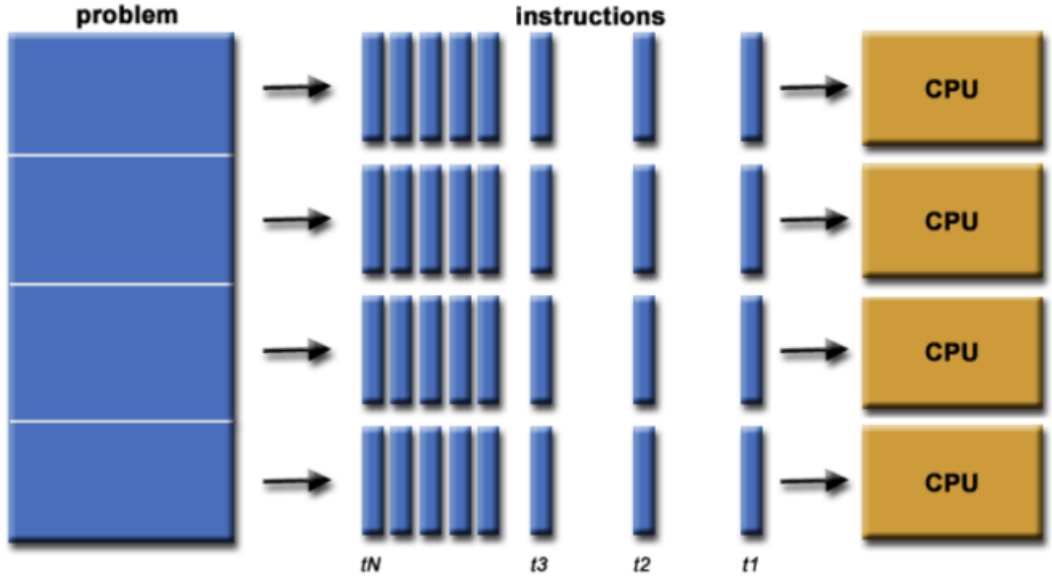
- **Load balancing**: It is important to distribute the workload evenly between all the threads because overloading some threads would not be a very efficient as they take more time to finish their job, whereas in the meantime other threads would finish their job leaving other processors idle, which results in underutilization of the available resources.

- Synchronization across threads needs to be carefully considered because it affects the application's overall performance. It is one of the most crucial components that need to be attended to because it may increase overhead and decrease parallelism. Overusing synchronization causes contention, which causes the program to run more like serial processing.

- Debugging a parallel program is way harder as we need to take into account the race condition, deadlock and synchronization issues which are complex and difficult to trace as they vary each time the program is executed.

3.5 Parallelization in CPUs:

Parallelism in CPUs typically works through the use of multiple cores. Each core can execute a separate thread of instructions simultaneously, allowing multiple tasks to be processed in parallel. The operating system (OS) schedules tasks and assigns them to different cores, allowing them to run simultaneously. The OS manages the sharing of resources such as memory, input/output devices, and other system resources to ensure that each core has access to the necessary resources to complete its assigned tasks.



Source: `https://docs.huihoo.com/hpc-cluster/parallel-computing/index.html#Whatis`

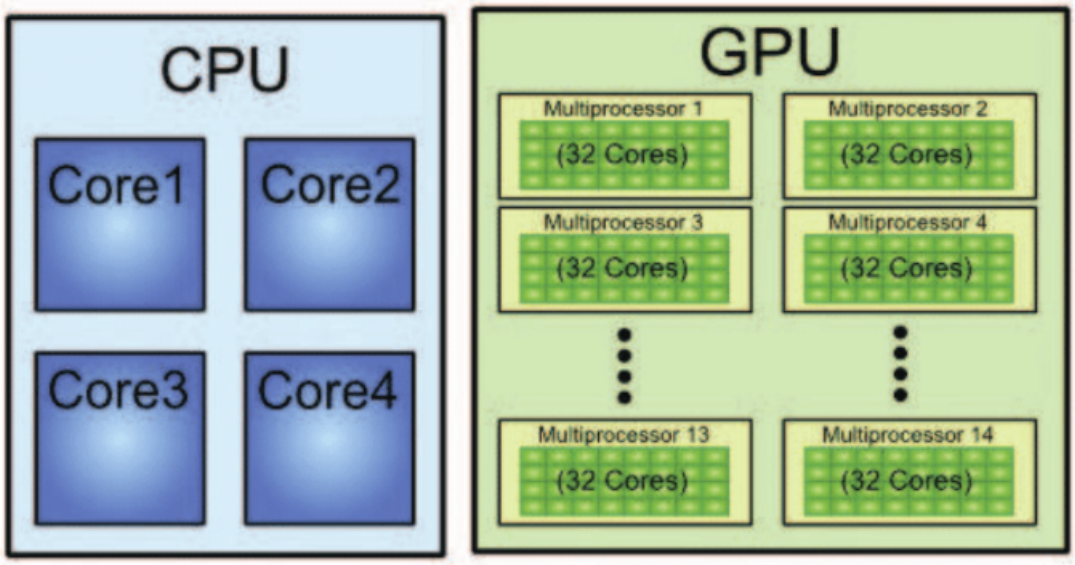
Figure 3.2: Parallelism in CPU

3.6 What is a GPU?

GPU is a specialized processor which was initially developed to accelerate the rendering of graphics. It is used in many different applications, including AI, machine learning, deep learning, video rendering, and gaming. GPUs have a high memory throughput and make use of numerous lightweight computing cores to benefit from data parallelism. Due to their extensive parallel structure, they are excellent for carrying out parallel computational tasks.

A single GPU device consists of various processing clusters, with each having their dedicated cache layers and cores associated with them.

In the above figure, we can observe the architectures of CPU and GPU. While there are only a few cores in the CPU, GPUs have thousands of cores which makes them suitable for designing parallel computation-intensive applications. One of the main reasons to choose a



Source: https://www.researchgate.net/publication/319468789_Mobile_devices'_GPUs_in_cloth_dynamics_simulation

Figure 3.3: CPU and GPU architectures

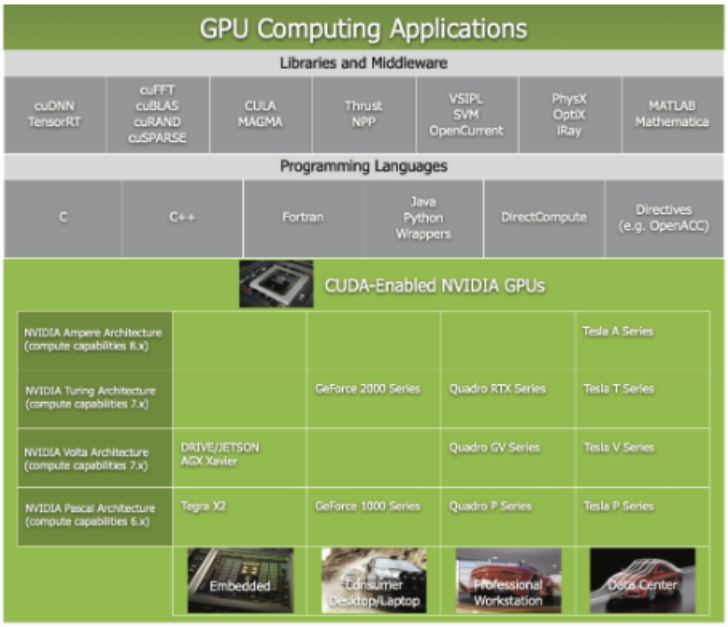
GPU for parallel processing is that it is able to divide complex tasks into thousands of smaller tasks, which it works on simultaneously unlike a CPU, since it has only fewer cores available, having to perform thousands of tasks makes it slow (almost similar to serial processing).

Each multiprocessor on a GPU uses a unique architecture called SIMT (Single Instruction, Multiple Thread), which is similar to SIMD (Single Instruction, Multiple Data) i.e; same instruction being performed on multiple data, which makes the GPU more suitable for the tasks which require the same process to be performed on numerous data items.This architecture allows programmers to write code for thread-level parallelism and data-parallel code.

In the following section, we go through the overview of CUDA, its architecture, and some advantages of using it.

3.7 Overview of CUDA

CUDA (Compute Unified Device Architecture) is a general-purpose parallel computing platform, developed by Nvidia in November 2006 to use on their GPUs. It is a programming model which allows the software to perform general-purpose computations on GPUs, that are traditionally performed on CPUs, generally, known as GPGPU (General Purpose computation on Graphics Processing Units). CUDA comes with various environments which allow developers to use it with different high-level languages such as C, C++, Java, Python, and Fortran, which makes it less difficult for programmers to use it.



GPU Computing Applications. CUDA is designed to support various languages and application programming interfaces.

Source: `https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html`

Figure 3.4: GPU computing applications

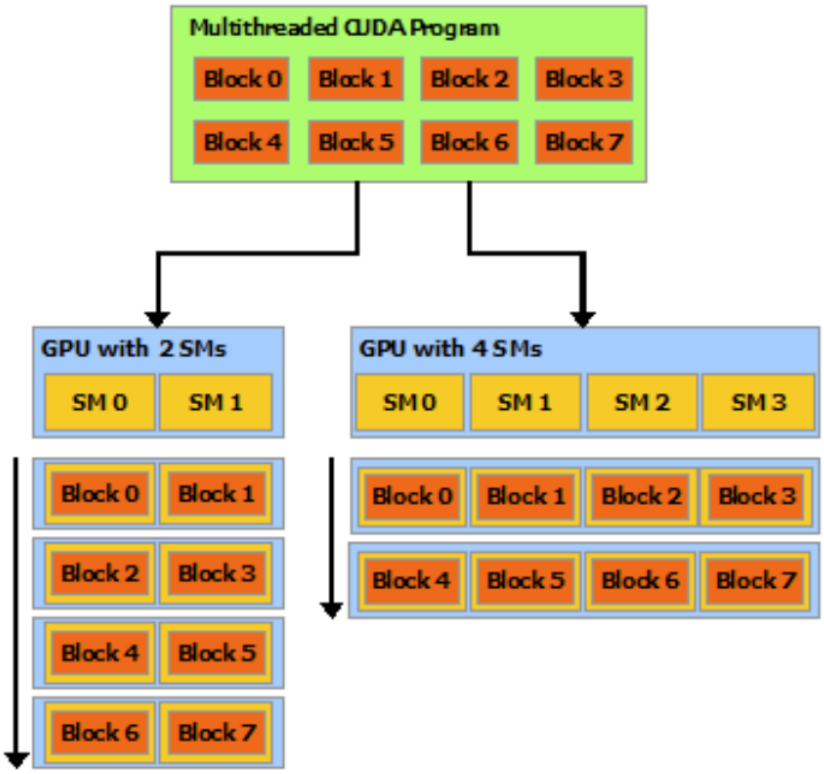
3.7.1 CUDA Architecture

The GPUs are built around a scalable array of multi-threaded Streaming Processors (SMs). When a CUDA kernel is launched from the host, various thread blocks are distributed between these processors so that they can be executed concurrently. CUDA is a scalable programming model which allows the software to scale its parallelization to the number of GPUs available. This makes the software adaptable to the change in processing cores i.e; the more cores available, the software leverages them to increase its parallelism. This nature allows software with more thread blocks to execute faster on the GPU with more SMs than with fewer SMs (automatic scalability).

3.7.1.1 Thread Hierarchy:

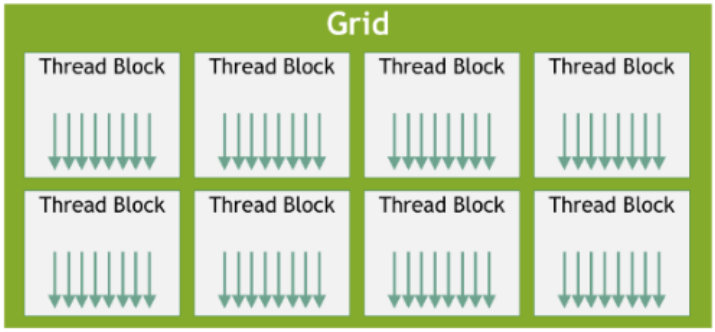
The thread block consists of a group of threads that can be launched by the SMs. These threads can either be 1, 2 or 3-dimensional blocks of threads. A thread block can contain up to 1024 threads. Furthermore, these blocks can be organized into 1, 2, or 3 dimensional blocks to form a grid.

Each thread block must be independent of one another so that it can execute in any order, which allows them to scale as the processors increase/decrease. Threads within a block are guaranteed to be scheduled on the same SM. They can be synchronized with one another using *__syncthreads()* which acts as a barrier lock. They can also cooperate between themselves using



Source: `https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html`

Figure 3.5: Automatic scalability of thread blocks

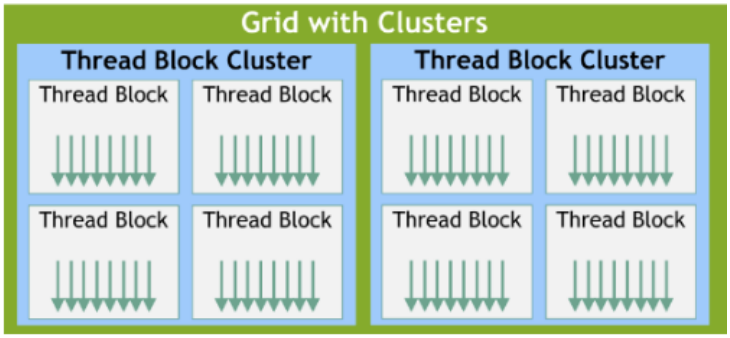


Source: `https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html`

Figure 3.6: Grid of thread blocks

shared memory (which is accessible to all the threads within the block).

An optional level of hierarchy called Thread Block Cluster was introduced which is made up of thread blocks. All the thread blocks within a cluster are guaranteed to be co-scheduled on the GPU processing cluster GPC. The thread blocks in a cluster can be organized into 1, 2, or 3-dimensional blocks.



Source: `https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html`

Figure 3.7: Grid of Thread Block Clusterss

3.7.1.2 Memory Hierarchy:

Threads have access to various memory spaces throughout their execution; there are several memory hierarchies available that offer different trade-offs between memory capacity, latency, and bandwidth. Understanding the different memory hierarchies in CUDA and choosing the appropriate memory space for storing data based on their access patterns, size, and usage requirements is essential for optimizing CUDA kernels for maximum performance on NVIDIA GPUs. The different memory hierarchies in CUDA include:

- **Global Memory**: Global Memory is the largest memory space available in CUDA, and it is accessible by all threads in a CUDA kernel. It is typically used for storing data that needs to be shared across multiple threads or blocks, and it has relatively high latency and lower bandwidth compared to other memory hierarchies.
- **Shared Memory**: Shared memory is a small, fast, on-chip memory that is shared among threads within a thread block. It has much lower latency and higher bandwidth compared to global memory, making it suitable for inter-thread communication and synchronization. Shared memory is physically located on the GPU chip and is managed by the programmer, but it is limited in size (typically a few kilobytes per thread block).

Read-only memory:

- **Constant Memory**: Constant memory is a read-only memory space that is cached and optimized for read-heavy operations. It is typically used for storing constant data that is accessed by multiple threads, such as lookup tables or constants used in mathematical computations.
- **Texture Memory**: Texture memory is a cached read-only memory space optimized for accessing 2D or 3D data with spatial locality, such as images or volume data. It provides special memory access features, such as automatic interpolation and filtering, making it suitable for texture mapping or image processing operations.

Low-Level Memory:

- **Register File**: The register file is a set of small, fast, on-chip registers that are used for storing thread-specific data. Registers are the fastest memory hierarchy in CUDA, with very low latency and high bandwidth. However, the number of registers available per thread is limited, and excessive use of registers can result in register spills to local memory, which can degrade performance.

- **Local Memory**: Local memory is the private memory space allocated to each thread within a thread block. It is used for storing thread-specific data, and it has similar characteristics to global memory in terms of latency and bandwidth. However, local memory is typically not used directly, as it is managed automatically by the CUDA compiler and runtime system.

- **L1 and L2 Cache**: The L1 and L2 caches are on-chip caches that are used for caching data from global memory, local memory, and texture memory. They are managed automatically by the GPU hardware and provide low-latency access to frequently used data, helping to hide the memory latency and improve performance.

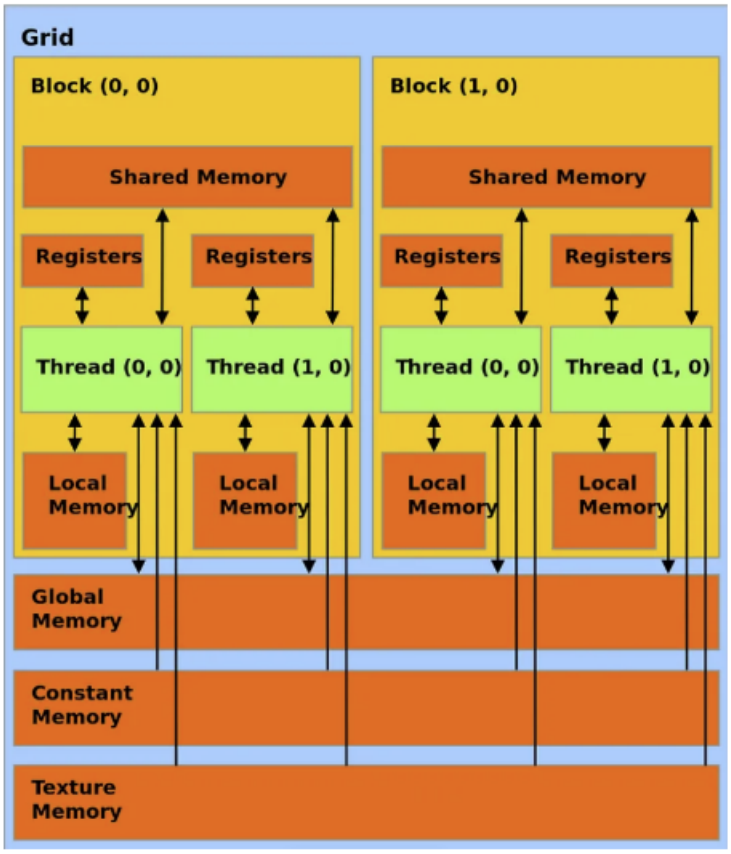
Shared memory has lower latency to access data compared to global, local memory, so in our optimizations, we try to use shared memory whenever possible so that the overall time taken by the algorithm is optimal. However, one limitation of shared memory is that its size is limited i.e: 1024 KB.

3.7.2 Heterogeneous programming:

CUDA assumes that the CUDA threads are executed on a separate device that acts as a co-processor to the host. "Host" refers to the CPU, and its memory is called host memory, while "Device" refers to GPU, and its memory is called device memory. CUDA assumes that both host and device have separate memory spaces in DRAM, because of which before we launch a kernel call from the host, we need to copy all the host memory which is needed by the kernel to the device memory, as CUDA threads cannot access host memory. Here are the three main steps while executing any CUDA program:

- **Allocate memory to the device**:
  Allocate memory on the device and copy the input data from host memory to device memory (Host-to-Device data transfer) so that the CUDA kernel can access the data to perform computations.

- **Launch GPU kernel**:
  Launch multiple GPU kernels so that they can perform computations on the data copied onto the device memory in parallel. The result is stored in the device's memory.



Source: `https://bodunhu.medium.com/pascal-gpu-memory-and-cache-hierarchy-28439cdee0fd`

Figure 3.8: Memory Hierarchy in GPU

MEMORY	ACCESS	SCOPE	LIFETIME	SPEED	NOTE
Global	RW	All threads and CPU	All	Slow, cached	Large
Constant	R	All threads and CPU	All	Slow, cached	Read same address
Local	RW	Per thread	Thread	Slow, cached	Register Spilling
Shared	RW	Per block	Block	Fast	Fast communication between threads
Registers	RW	Per thread	Thread	Fast	Don't use too many

Source: `https://bodunhu.medium.com/pascal-gpu-memory-and-cache-hierarchy-28439cdee0fd`

Figure 3.9: Memory Hierarchy in GPU

- **Copying data from device to host**: After the GPU kernels finish their execution, the resultant output is stored on device memory. To make it accessible to the host, we need to

copy the data back from device memory to host memory (Device-to-Host data transfer), and then deallocate all the device memory used.

Since CUDA kernels can access only device memory, the time taken for the memory transfers for Host-to-Device and Device-to-Host should also be taken care of as it is one of the factors which determine the efficiency of the program. There are various methods which are offered by CUDA to do these memory transfers.

- *cudaMalloc* is used to allocate memory in the device. We use this at the beginning so that after the memory is allocated to the device, we can transfer the data from the host to this location.
- *cudaMemcpy* is used to transfer memory from the device to the host and vice versa. It takes either one of the arguments, *cudaMemcpyHostToDevice* (transfer from host to device)or *cudaMemcpyDeviceToHost* (transfer from device to host).
- *cudaMemcpyToSymbol*, which is used to transfer memory from the host to constant memory in the device. As discussed in the memory hierarchy, constant memory is a read-only memory space that is optimized for read-heavy operations.

CUDA also gives us the ability to make these data transfers asynchronously with the use of the following methods such as cudaMemcpyAsync so that these tasks can be performed concurrently. Please note that the concurrency of these methods depends on the device we perform these on. We present the results we observed on making these tasks asynchronous (results might not be consistent as their behavior varies with the hardware used).

When the tasks are performed asynchronously, we use cudaDeviceSynchronize() to synchronize all the threads. To deallocate the memory used in the device, we use cudaFree(). It is helpful in removing unwanted data stored in the device

Chapter 4

Algorithms and related work

4.1 Bilateral Filter

4.1.1 Idea

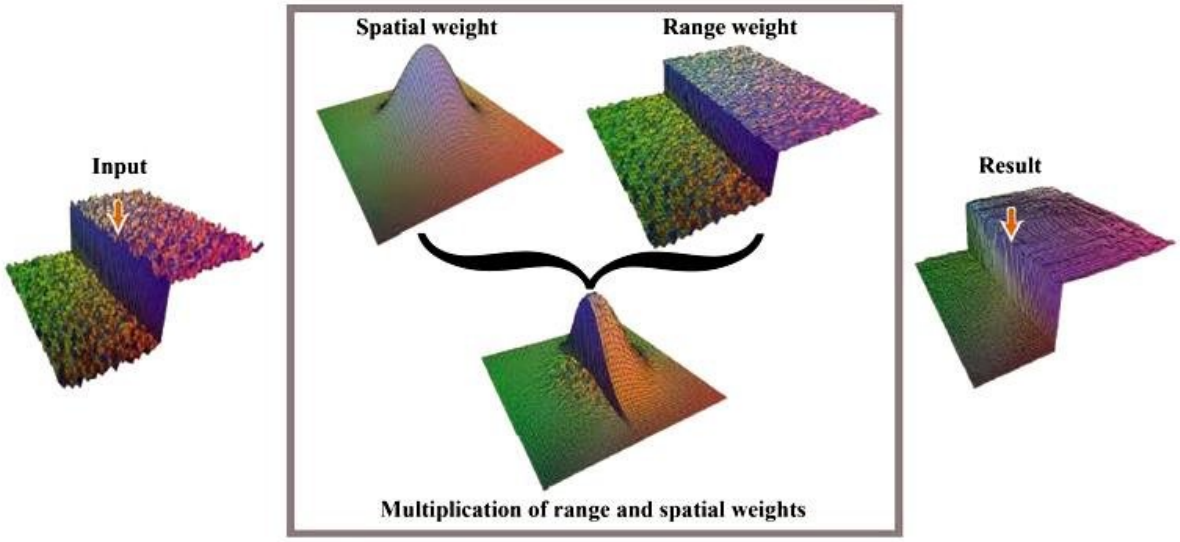
Bilateral filtering is a non-linear image filtering technique that aims to smooth images while preserving edges and details. It considers both spatial proximity and intensity similarity in the filtering process, making it suitable for a wide range of image processing applications.

Bilateral filtering is widely used in various image processing tasks, such as denoising, edge-preserving smoothing, and tone mapping. It has been shown to effectively remove noise while preserving image details and edges, making it suitable for applications where both the smoothing and preservation of fine structures are desired.

The bilateral filter came out as an improvement on the Gaussian filter with the goal of preserving edges by preventing edge blurring while removing noise. It is a non-linear filter that considers both the spatial proximity and intensity similarity between pixels in an image. It uses two separate Gaussian kernels: one for spatial filtering and another for intensity filtering. The spatial Gaussian kernel determines the weighting based on the distance between neighboring pixels, while the intensity Gaussian kernel determines the weighting based on the similarity of intensity values between neighboring pixels. By combining these two kernels, the bilateral filter takes into account both the spatial structure and the intensity contrast of the image, and it selectively smooths the image while preserving edges or boundaries with high intensity differences. This makes the bilateral filter particularly effective in preserving edges and fine details in an image, even during the smoothing process.

The bilateral filter is defined by two key components:

- The spatial domain kernel spatial(i, j)
- The range domain kernel range(I, I')



Bilateral filter weights at the central pixel

Source: http://doi.acm.org/10.1145/566570.566574

Figure 4.1: The bilateral filter smooths an input image while preserving its edges. Each pixel is replaced by a weighted average of its neighbors. Each neighbor is weighted by a spatial component that penalizes distant pixels and range component that penalizes pixels with a different intensity. The combination of both components ensures that only nearby similar pixels contribute to the final result. The weights shown apply to the central pixel (under the arrow)

where i and j denote the pixel coordinates in the image, I and I' denote the intensity values of two pixels, and σ_s and σ_r are the respective spatial and range domain standard deviations.

The spatial domain kernel represents the spatial proximity between pixels and is typically a Gaussian function centered at the target pixel, with a standard deviation σ_s that determines the extent of the spatial neighborhood considered. The range domain kernel, on the other hand, measures the intensity similarity between pixels and is typically a Gaussian function centered at the intensity value of the target pixel, with a standard deviation σ_r that controls the strength of the filter.

The bilateral filter operation is defined as follows:

$$\text{BilateralFilter}(I) = \frac{1}{W_p} \sum_{i,j} I(i,j) \cdot \text{spatial}(i,j) \cdot \text{range}(I(i,j), I') \tag{4.1}$$

where $I(i,j)$ is the intensity value of the pixel at position (i,j) in the image, and W_p is the normalization factor, computed as the sum of the spatial domain kernel multiplied by the range domain kernel.

Algorithm 1 Bilateral Filtering Algorithm

Require: Image I, Spatial standard deviation σ_s, Range standard deviation σ_r

Ensure: Filtered image I_{filtered}

for $i = 1$ to M **do**

for $j = 1$ to N **do** ▷ Initialize the normalization factor $W_p = 0$

$W_p \leftarrow 0$ ▷ Initialize the filtered intensity $I_{\text{filtered}}(i,j) = 0$

$I_{\text{filtered}}(i,j) \leftarrow 0$

for $k = 1$ to $diameter$ **do**

for $l = 1$ to $diameter$ **do**

$\text{spatial}(k,l) \leftarrow \exp\left(-\frac{(k-i)^2+(l-j)^2}{2\cdot\sigma_s^2}\right)$ ▷ Compute the spatial domain kernel $\text{spatial}(k,l)$ using the Gaussian function

$\text{range}(I(i,j),I(k,l)) \leftarrow \exp\left(-\frac{(I(i,j)-I(k,l))^2}{2\cdot\sigma_r^2}\right)$ ▷ Compute the range domain kernel $\text{range}(I(i,j),I(k,l))$ using the Gaussian function

$I_{\text{filtered}}(i,j) \leftarrow I_{\text{filtered}}(i,j) + I(k,l)\cdot\text{spatial}(k,l)\cdot\text{range}(I(i,j),I(k,l))$ ▷ Update the filtered intensity

$W_p \leftarrow W_p + \text{spatial}(k,l)\cdot\text{range}(I(i,j),I(k,l))$ ▷ Update the Normalization Factor

end for

end for

$I_{\text{filtered}}(i,j) \leftarrow \frac{I_{\text{filtered}}(i,j)}{W_p}$ ▷ Normalize the filtered intensity

end for

end for

4.1.2 Applications

The versatility of bilateral filter and their ability to preserve edges make them a valuable tool in many image processing and computer vision tasks.

- **Image Denoising**: The bilateral filter is commonly used for reducing noise in images while preserving edges and details. It can effectively remove noise from images without blurring edges, making it suitable for applications such as medical imaging, photography, video processing, and computer vision.
- **Image Enhancement**: The bilateral filter can be used to enhance images by improving contrast, sharpness, and details. It can selectively enhance image regions while preserving the overall image structure, making it useful for tasks such as image retouching, image editing, and image restoration.
- **Tone Mapping**: The bilateral filter can be used in High Dynamic Range (HDR) imaging and tone mapping to enhance the visual appearance of images with a wide dynamic range of intensity values. It can effectively compress the intensity range of an HDR image while preserving details and avoiding artifacts.
- **Stereo Vision**: The bilateral filter is commonly used in stereo vision and depth map estimation to smooth depth maps while preserving sharp depth discontinuities or edges. It helps to reduce noise in depth maps and improve the accuracy of 3D scene reconstruction.
- **Image Guided Surgery**: The bilateral filter can be used in image-guided surgery to filter and enhance medical images, such as CT scans or MRI scans, for better visualization of anatomical structures, tumor detection, and surgical planning.
- **Computer Graphics**: The bilateral filter is widely used in computer graphics for tasks such as texture filtering, anti-aliasing, and image-based rendering. It helps to smooth textures while preserving fine details and edges, resulting in improved image quality and visual realism.
- **Video Processing**: The bilateral filter is used in video processing applications, such as video denoising, video compression, and video stabilization, to reduce noise, blur, and artifacts in video sequences while preserving details and motion boundaries.

In addition to the above mentioned applications, the bilateral filter has been used in various other areas, such as image segmentation, image registration, image fusion, and image inpainting, among others.

4.1.3 Related Works

4.1.3.1 Bilateral Filter

1. The bilateral filter was first proposed by authors Tomasi, Carlo and Manduchi, Roberto in their paper [13], where they proposed a simple nonlinear edge-preserving filter that combines gray levels or colors based on both their geometric closeness and their photometric similarity prefer near values to distant values in both domain and range and is able to smooth colors and preserve edges in a way that is tuned to human perception.

2. Authors in [11] have provided a graphical, intuitive introduction to bilateral filtering, a practical guide for efficient implementation and an overview of its numerous applications, as well as mathematical analysis.

3. Authors in [2] have reviewed the maturity of fast algorithms in bilateral filtering and also presented the gross categories of the approaches different stream of researchers chosen through faster algorithm propositions along with comparative results in terms of both RMSE and computation efficiency.

4.1.3.2 Bilateral Filter on GPU

1. Authors in [3] explore the performance impact of using different memory access patterns, of using different types of device/on-chip memories, of using strictly aligned and unaligned memory, and of varying the size/shape of thread blocks to achieve optimal performance on GPUs.They achieved a 200x as compared to a single-threaded CPU implementation.

2. Authors in [19] have showed that advanced acceleration techniques (such as pre-computation, prefetching) can further speedup straightforward GPU implementations of bilateral filter and nearest neighborhood filters to approximately 1.4 and 4 times respectively.

3. Authors in [6] optimized the code to run more efficiently on the GPU, with the final result running at more than a thousand times faster than the CPU sequential implementation.

4.1.3.3 Bilateral Filter Misc

1. The authors in [9]present a new data structure the bilateral grid, that enables fast edge-aware image processing and demonstrates its usage on modern GPUs.

4.2 Adaptive Bilateral Filter

4.2.1 Idea

As an improvement to the bilateral filters' denoising capabilities, some literature works on the idea of adaptive bilateral filters, taking some key ideas from the publicly available sources we came up with a simple yet effective technique to come up with a better PSNR value in the results,

The adaptive bilateral filter typically provides control over various parameters, such as the local characteristics used for adaptivity, the strength of the adaptivity, and the size of the local neighborhood. This allows users to fine-tune the filter's behavior based on their specific needs and requirements, the key ideas this improved bilateral algorithm works on are the following:

- **Adaptivity**: Bilateral filter as described in Section 4.1 uses fixed width kernel for spatial and range kernels with fixed parameters σ_s and σ_r, this constricts the algorithm to use a one size fit all technique for every pixel and its neighborhood, The efficient performance of such algorithm relies on the fine-tuning of parameters using experiments or some approximations based on a study of results.

  The range sigma is a parameter that controls the strength of the intensity difference weight in the bilateral filter, with larger values resulting in more smoothing across a wider range of intensities.

  The filter doesn't dynamically adapt its filtering strength to characteristics of the locality of the pixel like image content, preserving edges, details, and textures.

  To resolve this adaptive bilateral filter comes with an adaptive range sigma and adaptive width in some techniques to facilitate better denoising in the window it's acting on, this indeed adds an extra computation step but improves the quality of the results and gives better edge preserving.

- **Localization properties**: Local characteristics of the range sigma play a crucial role in determining the strength of the intensity difference weight used for filtering at each pixel in an image.

  The local characteristics of the range sigma refer to the values or properties of the image intensities within a local neighborhood around each pixel, which can be used to adaptively adjust the range sigma for more localized filtering.

  The local characteristics are typically they are based on the statistics or properties of the image intensities within the neighborhood. Hence some of the metrics which can be used for locally adapting range sigma can be:

- Local intensity variance or standard deviation: The variance or standard deviation of the image intensities within a local neighborhood can provide an indication of the local intensity variation. Higher variance or standard deviation values indicate regions with greater intensity variation, such as edges or textures, and a smaller range sigma can be used to preserve these details.
- Local gradient magnitude: The gradient magnitude of the image intensities, which represents the rate of change of intensity values, can indicate regions with sharp intensity transitions or edges. Higher gradient magnitude values suggest areas with stronger edges or details, and a smaller range sigma can be used to preserve these features.
- Local image structure or texture: The local image structure or texture features, such as local patterns or textures, can be used to adapt the range sigma. For example, regions with complex or fine textures may require a smaller range sigma to preserve the texture details, while regions with smoother or simpler textures may benefit from a larger range sigma for more effective noise reduction.

4.2.2 Design of the algorithm used

Based on the ideas described in 4.2.1 and some references to the idea to test out the effectiveness of implementing an adaptive bilateral filter on CUDA GPU. We went with variance as the parameter for adaptivity of the weights of the range because of the following reasons:

- **Intensity variation indicator**: Variance measures how much the intensity values within a neighborhood deviate from the mean intensity value. A higher variance indicates higher intensity variation or contrast, while a lower variance indicates lower intensity variation or a more uniform intensity region. By using variance as a metric, an adaptive bilateral filter can effectively capture the local characteristics of the image content, such as edges, textures, and details, which often exhibit higher intensity variations. This allows the filter to adaptively adjust its filtering strength based on the local intensity variation, resulting in more localized filtering.

- **Sensitivity to local changes**: Variance is sensitive to local changes in the image content. When there are intensity transitions or changes within a neighborhood, the variance value increases, indicating a change in the local image characteristics. This sensitivity to local changes makes variance a suitable metric for localization in adaptive bilateral filters, as it can capture the presence of edges, textures, or other local features that require localized filtering.

- **Robustness to noise**: Variance is relatively robust to noise compared to other metrics, such as gradient magnitude or image structure, which can be affected by noise. Since variance is based on the squared differences between intensity values and the mean intensity, it tends to amplify larger differences while reducing the impact of small differences due to noise. This makes variance a robust metric for localization in adaptive bilateral filters, as it can effectively capture the local intensity variation even in the presence of noise.

- **Easy to compute**: Variance is a computationally efficient metric to compute, as it only requires computationally simple mathematical operations, and can be calculated using local neighborhood statistics. This makes it practical for real-time or large-scale image processing applications.

Local Intensity variance:

$$V(x,y) = \frac{1}{W^2} \sum_{i=-\frac{W-1}{2}}^{\frac{W-1}{2}} \sum_{j=-\frac{W-1}{2}}^{\frac{W-1}{2}} (I(x+i,y+j) - \mu)^2$$

where:

- $V(x,y)$ represents the local intensity variance at pixel (x, y).
- $I(x+i,y+j)$ denotes the intensity of the pixel at coordinates (x+i, y+j) within the window or patch centered at (x, y).
- μ is the mean intensity of the pixels within the window or patch, calculated as: $\mu = \frac{1}{W^2}\sum_{i=-\frac{W-1}{2}}^{\frac{W-1}{2}}\sum_{j=-\frac{W-1}{2}}^{\frac{W-1}{2}} I(x+i,y+j)$.
- W is the window size, typically an odd integer value, which determines the size of the local neighborhood or patches around each pixel for variance calculation.
- $\sum$ denotes the summation symbol, which indicates the sum of the values over the specified range.

Another tweak that we made in our adaptive bilateral filter was first applying a basic bilateral filter to a neighborhood before calculating its variance, instead of directly calculating the variance of the intensities, this was done because for the following reasons:

- **Noise reduction**: Applying a basic bilateral filter to the neighborhood before calculating variance helps in reducing noise in the image. The bilateral filter considers both the intensity differences and spatial distances between pixels and applies a weighted averaging operation. This can help to smooth out small intensity variations caused by noise while preserving larger intensity differences associated with edges or other image structures.
- **Edge preservation**: The bilateral filter is known for its ability to preserve edges in the image, which is important for maintaining image details and sharpness. By first applying a basic bilateral filter to the neighborhood, the edges are preserved to some extent, and the subsequent calculation of variance is performed on a smoother version of the image. This can help to reduce the impact of noise on the calculated variance, leading to a more accurate estimate of the local image statistics.
- **Computational efficiency**: Calculating variance directly on the original intensities in an adaptive bilateral filter can be computationally expensive, as it requires considering the intensity differences between pixels in the neighborhood for each pixel in the image. On the other hand, applying a basic bilateral filter to the neighborhood first reduces the complexity of the subsequent variance calculation, as it smoothens the image and reduces the number of intensity differences to consider. This can result in faster processing times and more efficient computations.

The above algorithm showed erroneous results in regions with low variance , on analysis it was found that some regions had low variance leading to new sigma intensity as zero, to fix this we used regularization in the variance calculation :

new_sigma_intensity = sqrt(max(variance + 0.01, 0.f)) / 2

Here, the value of 0.01 is a parameter that can be fine-tuned according to the specific requirements of the application.

It represents the minimum value of the new sigma intensity in cases where the variance in the result window is zero. By implementing this regularization technique, we were able to improve the algorithm's performance in regions with low variance and produce more accurate results overall.

Algorithm 2 Adaptive Bilateral Filtering Algorithm

Require: Image I, Spatial standard deviation σ_s, Range standard deviation σ_r

Ensure: Filtered image I_{filtered}

for $i = 1$ to M **do**

for $j = 1$ to N **do**

$W_p \leftarrow 0$ ▷ Initialize the normalization factor $W_p = 0$

$I_{\text{filtered}}(i,j) \leftarrow 0$ ▷ Initialize the filtered intensity $I_{\text{filtered}}(i,j) = 0$

$VarSum \leftarrow 0$ ▷ Initialize the Variance Sum $VarSum = 0$

for $k = 1$ to $diameter$ **do**

for $l = 1$ to $diameter$ **do**

$\text{spatial}(k,l) \leftarrow \exp\left(-\frac{(k-i)^2+(l-j)^2}{2\cdot\sigma_s^2}\right)$ ▷ Compute the spatial domain kernel $\text{spatial}(k,l)$ using the Gaussian function

$\text{range}(I(i,j),I(k,l)) \leftarrow \exp\left(-\frac{(I(i,j)-I(k,l))^2}{2\cdot\sigma_r^2}\right)$ ▷ Compute the range domain kernel $\text{range}(I(i,j),I(k,l))$ using the Gaussian function

$I_{\text{filtered}}(i,j) \leftarrow I_{\text{filtered}}(i,j) + I(k,l)\cdot\text{spatial}(k,l)\cdot\text{range}(I(i,j),I(k,l))$ ▷ Update the filtered intensity

$W_p \leftarrow W_p + \text{spatial}(k,l)\cdot\text{range}(I(i,j),I(k,l))$ ▷ Update the Normalization Factor

$VarSum \leftarrow VarSum + W_p\cdot(I(i,j)-I(k,l))^2$ ▷ Update the Variance sum

end for

end for

$Variance \leftarrow \frac{VarSum}{W_p}$ ▷ Calculate Variance

$NewSigmaRange \leftarrow max(\sqrt{Variance}+0.01,0),/2$ ▷ Calculate New Sigma

for $k = 1$ to $diameter$ **do**

for $l = 1$ to $diameter$ **do**

$\text{spatial}(k,l) \leftarrow \exp\left(-\frac{(k-i)^2+(l-j)^2}{2\cdot\sigma_s^2}\right)$ ▷ Compute the spatial domain kernel $\text{spatial}(k,l)$ using the Gaussian function

$\text{range}(I(i,j),I(k,l)) \leftarrow \exp\left(-\frac{(I(i,j)-I(k,l))^2}{2\cdot\ NewSigmaRange^2}\right)$ ▷ Compute the range domain kernel $\text{range}(I(i,j),I(k,l))$ using the Gaussian function

$I_{\text{filtered}}(i,j) \leftarrow I_{\text{filtered}}(i,j) + I(k,l)\cdot\text{spatial}(k,l)\cdot\text{range}(I(i,j),I(k,l))$ ▷ Update the filtered intensity

$W_p \leftarrow W_p + \text{spatial}(k,l)\cdot\text{range}(I(i,j),I(k,l))$ ▷ Update the Normalization Factor

end for

end for

$I_{\text{filtered}}(i,j) \leftarrow \frac{I_{\text{filtered}}(i,j)}{W_p}$ ▷ Normalize the filtered intensity

end for

end for

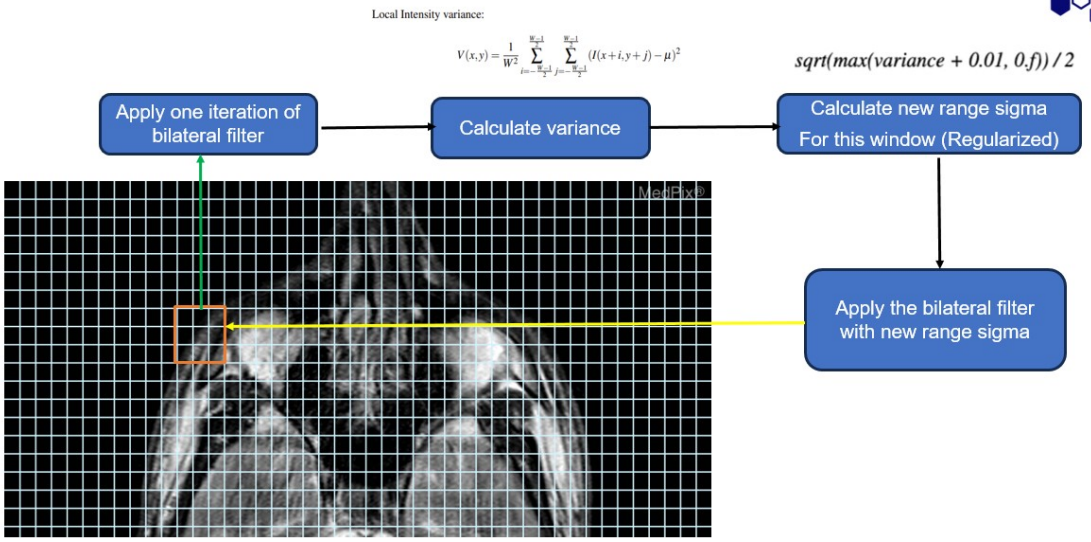


Figure 4.2:

Adaptive Bilateral Filter

4.2.3 Related Works

1. The authors in [10] present an idea for an adaptive bilateral filter that filters the high-frequency noise of the image and simultaneously maintains the high-frequency edge information of an image based on image brightness variations proposed. The filtering method optimizes the weight coefficient of the Gaussian filter into the product form of the Gaussian function and image brightness information. The optimized weight coefficient will be applied to the convolution of the image. In this way, image brightness will be considered and the image edge should be simultaneously maintained as possible during filtering image noise. Since the bilateral filtering method can make the weight coefficient of the filter vary with image brightness, the adaptive filtering effect can be attained during the filtering process.

2. The authors in [17] proposed a method that exploits the local phase characteristics of an image signal to perform bilateral filtering in an adaptive manner. The proposed method takes advantage of the human perception system to preserve perceptually significant signal detail while suppressing perceptually significant noise in the image signal

3. The authors in [12] presented a method that exploits the local Gaussian gradient information of the processing image and applies a bilateral filter by changing the range filter parameter σ_r adaptively. The proposed adaptive bilateral filter could preserve more de-

tails of the image compared with the original bilateral filter in edge or texture regions

4. The authors in [1] presented The concept of the Total variation model added to the bilateral filter. Based on analyzing force distribution rules of variance, the standard deviation is managed to distinguish the degree of degradation. The optimization solution of the total variation is gained by tracking minimum change channels and keeping maximum edges.

5. The authors in [5] proposed a novel Gaussian-adaptive bilateral kernel that can adaptively acquire a clean Gaussian range kernel for different noise filtering inputs. On the basis of the Gaussian-adaptive bilateral kernel, we proposed a GABF. It is a simple yet efficient edge-preserving image-smoothing filter that enables the use of a noise-free bilateral kernel to effectively perform edge-preserving image smoothing for noise-filtering inputs

4.3 NL Means + Bilateral Filter

4.3.1 Idea

The Non-Local Means bilateral filter improves upon the traditional bilateral filter by considering patch-based similarity, allowing for non-local information exchange, providing flexibility in tuning parameters and being versatile for different image processing tasks. These improvements make NLM bilateral filter more effective in denoising images with complex structures and textures, and more adaptable to different image characteristics and applications.

Non-local means (NLM) is an image-denoising algorithm that aims to remove noise from an image while preserving its details and structures introduced in [4]. The basic idea behind the NLM algorithm is to exploit the redundancy of information in natural images. It assumes that similar patches (small image regions) in an image have similar intensity patterns and that the denoised pixel value at a particular location can be estimated by averaging the pixel values of similar patches from the entire image.

The NLM algorithm consists of the following steps:

- **Patch Extraction**: For each pixel in the image, a patch (a small square or rectangular region) centered around that pixel is extracted. The size of the patch is typically chosen to be large enough to capture local structures and small enough to ensure computational efficiency.

- **Patch Similarity Calculation**: The similarity between patches is computed using a similarity metric, such as SSD (Sum of Squared Differences), SSIM (Structural Similarity Index), or other similarity measures. The similarity metric measures the difference between the intensities of corresponding pixels in the patches.

- **Weight Calculation**: Based on the patch similarity values, weights are calculated for each patch. Similar patches are assigned higher weights, indicating that they contribute more to the denoised pixel value at the center of the patch.

- **Denoised Pixel Calculation**: The denoised pixel value is computed as a weighted average of the pixel values in the similar patches, where the weights are obtained from the patch similarity values. This weighted averaging process takes into account the contributions of similar patches to estimate the denoised pixel value.

- **Iterative Process**: The denoising process is typically performed iteratively, with the denoised pixel values updated at each iteration. This helps to refine the denoised image gradually and improve its quality.

Algorithm 3 Non-Local Means (NLM) Algorithm

Require: Input noisy image I, patch window size w, search window size h, noise variance σ^2

Ensure: Denoised output image I_{denoised}

for each pixel p in I **do**
    Extract patch P_p centered at p from I
    Initialize $I_{\text{denoised}}(p) = 0$ and $Z_p = 0$
    for each pixel q in search window W_p around p **do**
        Extract patch P_q centered at q from I
        Compute patch similarity $S(p,q)$ between P_p and P_q using a similarity metric (detailed in next section)
        Compute weight $w(p,q)$ for P_q as: $w(p,q) = \exp\left(-\frac{S(p,q)}{h^2}\right)$
        Update $I_{\text{denoised}}(p)$ as: $I_{\text{denoised}}(p) = I_{\text{denoised}}(p) + w(p,q) \cdot I(q)$
        Update Z_p as: $Z_p = Z_p + w(p,q)$
    end for
    Normalize $I_{\text{denoised}}(p)$ by dividing by Z_p: $I_{\text{denoised}}(p) = \frac{I_{\text{denoised}}(p)}{Z_p}$
end for
return I_{denoised}

4.3.1.1 Limitations

The time complexity of the NLM algorithm can be relatively high, especially for large images and large patch window sizes. The time complexity of the NLM algorithm can be approximated as $O(N \times M \times w^2 \times f(w,h))$, where $f(w,h)$ represents the time complexity of the similarity metric, weight calculation, and denoising steps, the size of the input image is $N \times M$, where N is the height and M is the width of the image. The patch window size is denoted as w, and the search window size as h

4.3.2 NLM+Bilateral Filter

The Non-Local Means (NLM) + Bilateral Filter algorithm is a hybrid denoising algorithm that combines the strengths of both the NLM and bilateral filter approaches to achieve improved image denoising performance.

The NLM part of the algorithm is responsible for capturing the non-local similarity information from the entire image in order to denoise each pixel. It computes the weighted average of intensities from similar patches in the image, where the weights are determined based on the similarity between patches. The NLM part helps in preserving fine details and structures in the image while reducing noise.

The Bilateral Filter part of the algorithm is used to further refine the denoised image obtained from the NLM part. The bilateral filter is a spatial domain filter that smoothens the image while preserving edges and structures. It achieves this by applying a weighted average of pixel intensities, where the weights are determined based on both the spatial distance and the intensity similarity between neighboring pixels. The bilateral filter helps in reducing noise and smoothing the image while preserving edges and structures.

The combination of NLM and bilateral filter in this algorithm helps in achieving a denoised image that has reduced noise, preserved edges and structures, and enhance fine details. The NLM part captures non-local similarities, while the bilateral filter further refines the denoised image in the spatial domain. The algorithm can be implemented with appropriate parameter settings for the NLM and bilateral filter components to achieve the desired denoising performance for a particular image or application.

The Non-Local Means (NLM) bilateral filter is an improvement over the traditional bilateral filter in several key aspects:

- **Patch-based Similarity**: While traditional bilateral filters consider only the local neighborhood of each pixel for computing the weighted average, NLM bilateral filter considers the similarity of entire patches in the image. This means that instead of relying solely on the intensity values of nearby pixels, NLM considers the structure and texture of larger patches, which can capture more complex patterns and structures in the image. This patch-based similarity measure in NLM allows for better noise reduction performance and preservation of edges and fine details, making it more effective in denoising images with intricate textures or repetitive patterns. Some of the metrics which can be used for calculating the similarity between patches are:

  - **Euclidean Distance**: This is the most straightforward metric, which calculates the Euclidean distance between the intensities of corresponding pixels in the two patches. It is defined as:

$$D(p,q) = \sqrt{\sum_{k=1}^{N} (p_k - q_k)^2}$$

where p and q are the intensities of corresponding pixels in the two patches, N is the number of pixels in the patch, and p_k and q_k are the intensity values of the k-th pixel in the patches.

– **Manhattan Distance**: Also known as the L1 distance or city-block distance, this metric calculates the sum of the absolute differences between the intensities of corresponding pixels in the two patches. It is defined as:

$$D(p,q) = \sum_{k=1}^{N} |p_k - q_k|$$

– **Squared Euclidean Distance**: This metric calculates the squared Euclidean distance between the intensities of corresponding pixels in the two patches. It is defined as:

$$D(p,q) = \sum_{k=1}^{N} (p_k - q_k)^2$$

– **Cosine Similarity**: This metric measures the cosine of the angle between the intensity vectors of the two patches. It is defined as:

$$D(p,q) = 1 - \frac{\sum_{k=1}^{N} p_k \cdot q_k}{\sqrt{\sum_{k=1}^{N} p_k^2} \cdot \sqrt{\sum_{k=1}^{N} q_k^2}}$$

where p_k and q_k are the intensity values of the k-th pixel in the patches.

– **Histogram-based Similarity**: Instead of comparing pixel intensities directly, this metric compares the histograms of pixel intensities in the patches. Histogram-based similarity metrics, such as histogram intersection or histogram chi-squared distance, can capture the statistical distribution of intensities in the patches, which can be useful for capturing texture or structural similarities.

– **SSD (Sum of Squared Differences)** SSD calculates the squared differences between corresponding pixel intensities of two patches and then sums up these squared differences to obtain the overall similarity measure. SSD is a commonly used similarity metric in image processing and computer vision tasks, such as image alignment, object recognition, and stereo vision. It is a simple and fast metric to compute pixel-wise similarity metric for comparing two patches x and y. The formula for SSD is as follows:

$$SSD(x,y) = \sum_i (x[i] - y[i])^2$$

where:

* x, y: the two patches being compared, represented as arrays of pixel intensities.
* x[i], y[i] :the pixel intensities at corresponding positions in x and y respectively.

– **SSIM (Structural Similarity Index)**: SSIM is a widely used image quality assessment metric that measures the structural similarity between two images. It takes into account not only the mean intensity differences but also the variances and covariances of the patches being compared, making it more robust to changes in lighting conditions, contrast, and texture. SSIM values range from -1 to 1, with 1 indicating perfect similarity and -1 indicating complete dissimilarity.

$$\text{SSIM}(x,y) = \frac{(2\mu_x\mu_y + c_1)(2\sigma_{xy} + c_2)}{(\mu_x^2 + \mu_y^2 + c_1)(\sigma_x^2 + \sigma_y^2 + c_2)}$$

where:

x, y : the two patches being compared, represented as arrays of pixel intensities.
μ_x, μ_y : the mean intensities of patches x and y, respectively.
σ_x^2, σ_y^2 : the variances of patches x and y, respectively.
σ_{xy} : the covariance of patches x and y.
c_1, c_2 : constants to avoid division by zero

– **NCC (Normalized Cross-Correlation)**is a measure of similarity between two signals or images. It is commonly used in image processing and computer vision tasks for tasks such as image registration, template matching, and feature matching.

The NCC between two signals or images is computed as the dot product of the two signals normalized by the product of their magnitudes. The formula for NCC is as follows:

$$\text{NCC}(x,y) = \frac{\sum_i (x[i] - \mu_x)(y[i] - \mu_y)}{\sqrt{\sum_i (x[i] - \mu_x)^2 \sum_i (y[i] - \mu_y)^2}}$$

where:

* x and y are the two signals or images being compared, represented as arrays of values.
* $x[i]$ and $y[i]$ are the values at corresponding positions in x and y, respectively.
* μ_x and μ_y are the means of x and y, respectively.
* The summation $\sum_i$ denotes the sum over all elements in x and y.

NCC produces a value between -1 and 1, where 1 indicates perfect similarity, -1 indicates perfect dissimilarity, and 0 indicates no correlation. NCC is commonly used in image-matching tasks because it is invariant to changes in intensity scale and offsets, making it robust to changes in lighting conditions and contrast. It is a useful similarity measure for aligning and comparing images, particularly when the images have varying illumination conditions or intensity scales.

- **Non-Local Information Exchange**: Unlike traditional bilateral filters that only consider local neighborhoods, NLM bilateral filter allows for non-local information exchange between distant patches in the image. This means that the filter can capture similarity across the entire image, enabling it to capture global image structures and textures. This is particularly useful in denoising images with large-scale structures or when the noise is not confined to local regions. The non-local information exchange property of NLM makes it more versatile and adaptable to different image characteristics.

- **Flexibility in Tuning Parameters**: NLM bilateral filter provides more flexibility in tuning its parameters compared to traditional bilateral filters. For example, the window size, which determines the size of the patches used for patch comparison, can be adjusted to capture different levels of image structures and textures. Additionally, the dissimilarity metric used for patch comparison in NLM can be chosen based on the specific characteristics of the image or noise, allowing for finer control over the denoising process. This flexibility in parameter tuning makes NLM more adaptable to different types of images or noise levels.

- **Versatility for Different Applications**: Due to its ability to preserve image structures and textures while denoising, NLM bilateral filter has been widely used in various image processing tasks beyond denoising, such as image restoration, image fusion, and image inpainting. The patch-based similarity measure and non-local information exchange property of NLM make it versatile for different image processing applications where capturing global image context is important.

4.3.3 Design of the algorithm used

Building upon the ideas presented in [4] and [15],the variation of NL means bilateral algorithm which we implemented was crafted with the following tweaks:

- **Keep the whole image as the search window**: We have kept the search window large (the whole image), which is the most compute expensive search window possible to analyze the speedup correctly and also to achieve the following benefits:

- **Improved Denoising Performance:** Using a larger search window allows for a more extensive search space, which can result in better denoising performance. It enables the filter to consider a wider range of pixels in the image when calculating the weighted average, leading to a more effective removal of noise while preserving image details.
- **Better Preservation of Image Features:** A larger search window in bilateral filtering allows for better preservation of image features, as it considers a larger neighborhood around each pixel. This can help in retaining sharp edges, fine textures, and other important image details that may be lost with a smaller search window.
- **Robustness to Local Image Variations:** In images with local variations in intensity or color, using a larger search window can help in capturing the local image characteristics more effectively. This can lead to improved filtering results, as the filter can adapt to varying local image structures and textures.
- **Reduced Blocking Artifacts:** Bilateral filtering can sometimes introduce blocking artifacts, which are visible as sharp transitions between filtered and non-filtered regions. Using a larger search window can help in reducing such artifacts, as it allows for a smoother transition between filtered and non-filtered regions, resulting in a more visually pleasing output.
- **Flexibility in Parameter Selection:** With a larger search window, bilateral filtering can be less sensitive to the choice of parameters such as the filter kernel size or the intensity/color similarity threshold. This can provide more flexibility in parameter selection, making it easier to achieve desired filtering results across a wider range of images.

- **Use SSD as metric for patch similarity index** We chose SSD as the metric to calculate patch similarity for its simplicity and effectiveness, the specific reasons for choosing SSD are listed below:

  - **Intensity-based Similarity:** SSD calculates the squared differences between pixel intensities in the patches being compared. It measures the dissimilarity in intensity values between the reference patch and the candidate patches. In non-local means bilateral filtering, the intensity similarity is an important criterion for determining the weight of each candidate patch in the weighted averaging process. SSD can effectively capture the differences in intensity values, making it suitable for this purpose.
  - **Easy to Compute:** SSD is computationally efficient and straightforward to calculate. It involves simple arithmetic operations, such as subtraction and squaring,

making it computationally efficient and suitable for real-time image processing applications.

  - **Sensitivity to Pixel Differences:** SSD is sensitive to small pixel differences, which allows for precise discrimination between similar and dissimilar patches. It can capture fine details and subtle differences in patches, making it suitable for preserving image details and textures during denoising or other image-processing tasks.
  - **Widely Used in Literature:** SSD has been widely used in various image processing and computer vision applications, including non-local means bilateral filtering. It has been extensively studied in the literature and has shown promising results in many practical scenarios.

- **Use intensity difference Gaussian in weight**

  - **Improved Filtering Accuracy:** By incorporating both intensity difference and range difference in patch similarity, we can better capture the structural information of the image. Intensity difference helps to capture local contrast and texture information, while range difference considers the spatial distance between pixels. Combining both of these factors can lead to improved filtering accuracy, resulting in better denoising or image enhancement results.
  - **Robustness to Local Image Variations:** Intensity difference and range difference together provide a more comprehensive measure of patch similarity, making the filter more robust to local image variations. In images with varying intensities or colors, using both intensity and range differences allows the filter to adapt to different local structures and textures, leading to more accurate filtering results.
  - **Enhanced Preservation of Image Details:** Intensity difference is effective in capturing fine details and subtle differences in patches, while range difference considers the spatial distance between pixels. This combination can help in better-preserving image details such as edges, textures, and small structures, which may be lost with only one of these factors. As a result, the non-local means the bilateral filter can better preserve image details while reducing noise or enhancing image quality.
  - **Flexibility in Parameter Selection:** Incorporating both intensity difference and range difference in patch similarity provides more flexibility in parameter selection. The weights assigned to these factors can be adjusted to achieve the desired filtering results for different types of images or noise levels. This allows for fine-tuning of the filter's behavior, making it more adaptable to different image processing scenarios.
  - **Improved Performance on Images with Large Intensity Variations:** In images with large intensity variations, using only range difference in patch similarity may

not be sufficient to accurately capture the local structures. Intensity difference can help in better discriminating between similar and dissimilar patches, even in the presence of large intensity variations. This can lead to improved filtering performance on images with diverse intensity ranges, making the non-local means bilateral filter more versatile in handling a wide range of images.

Algorithm 4 NLMeansBilateralFilter

Require: image, h, *hForSSD*, *patchSize*, *windowSize*

Ensure: filteredImage

function NLMEANSBILATERALFILTER(image, h, *hForSSD*, *patchSize*, *windowSize*)
    Initialize $filteredImage$ with same size as *image*
    Initialize *weightSum* with same size as *image*
    for each pixel (x,y) in *image* **do**
        $weightedSum \leftarrow 0$
        $weightSum \leftarrow 0$
        for each pixel (i,j) in *windowSize* around (x,y) **do**
            if (i,j) is within image bounds **then**
                SSD $= \sum_{k,l} (\text{image}(x+k,y+l) - \text{image}(i+k,j+l))^2$, where $k,l \in [-\frac{\text{patchSize}}{2}, \frac{\text{patchSize}}{2}]$ ▷ Calculate Sum of Squared Differences (SSD) between patch centered at (x,y) and patch centered at (i,j)
                intensityDifferenceGaussian $= \exp\left(-\frac{0.5*\text{image}(x,y)-\text{image}(i,j)^2}{\sigma^2_{intensity}}\right)$
                spatialDistanceGaussian $= \exp\left(-\frac{0.5*(x-i)^2+(y-j)^2}{\sigma^2_{spatial}}\right)$
                rangeWeight $= \exp\left(\frac{0.5\cdot \text{SSD}^2}{\sigma_h^2 \cdot \text{patch\_size}}\right) \cdot$ intensityDifferenceGaussian $\cdot$ spatialDistanceGaussian
                Add weighted pixel value: $weightedPixelValue = \text{rangeWeight} \cdot \text{image}(i,j)$ to *weightedSum*
                Add rangeWeight to *weightSum*
            end if
        end for
        Set $filteredImage(x,y) \leftarrow \frac{weightedSum}{weightSum}$
    end forreturn $filteredImage$
end function

4.3.4 Related Works

4.3.4.1 NL Means

1. In [4] the authors presented a new algorithm, the nonlocal means (NL-means), based on a nonlocal averaging of all pixels in the image The NL means not only compares the grey level in a single point but the geometrical configuration in a whole neighborhood. This fact allows a more robust comparison than neighborhood filters. Natural images also have enough redundancy to be restored by NL means. Flat zones present a huge number of similar configurations lying inside the same object They prove that the NL-means algorithm corrects the noisy image rather than trying to separate the noise (oscillatory) from the true image (smooth).
2. In [7] the authors proposed and advocated a unified approach to finding similarity measures in the context of NLM denoising using subtractive similarity measure (SSM) and rational similarity measure (RSM)
3. In [14] the authors propose an improved non-local means (INLM) filter for color image denoising. The algorithm combines the advantage of the NLM and the BILF. It calculates a new weight by measuring the spatial similarity, the pixel similarity, and the mean of differences
4. In [20] The authors propose an improved version of NLM by using weak textured patches based single image noise estimation and two-stage NLM with adaptive smoothing parameter, their method first applies weak textured patches-based noise estimation to achieve the noise level of input noisy image. Then relying on the estimated noise level, we apply the first stage NLM with adaptive smoothing parameter to attain a basic denoised image. After that, the basic denoised image is refined by the second stage of NLM with smaller smoothing strength
5. [16] Goes through the effectiveness of existing NL means filters
6. Authors in [18] used the Visible Human's Human Head test image (size 256*256) to evaluate the performance of the different filters (Bilateral,NLM) and establish the general strengths and weaknesses of these filters in terms of quality within an iterative reconstruction framework

4.3.4.2 NL Means+Bilateral

1. In [15] the authors proposed a non-local bilateral filter algorithm for image denoising based on the neighborhoods' gray value and the corresponding neighborhoods' Gaussian curvature

4.3.4.3 NL Means on GPU

1. In [8] The authors tried implementing Non-local means on a GPU. The experimental results show that, on the aspect of the denoising effect, the GPU-based NLM denoising algorithm does not change the accuracy of the original algorithm, that is, effectively maintains the image edges and small structures, at the same time removes the noises. On the aspect of computing speed, the implementation of GPU-based NLM denoising algorithm is up to 45 times the speedup

Chapter 5

Methodologies used

5.1 Bilateral Filter

5.1.1 CPU Implementation

5.1.1.1 Sequential

The sequential CPU implementation of the bilateral filter (bilateralFilterSeq) follows a straightforward approach where the filter is applied to each pixel in the source image sequentially using nested loops, features of this implementation are:

- **Iterating over Pixels**: Nested loops apply the bilateral filter to each pixel in the source image. The outer loop iterates over the rows of the image, and the inner loop iterates over the columns of the image. This ensures that each pixel in the image is processed.
- **applyBilateralFilter Function**: The applyBilateralFilter function is called for each pixel, passing the source image, filteredImage, and the current pixel's coordinates (i, j) as parameters. Inside the applyBilateralFilter function, the bilateral filter is applied to the current pixel using the given parameters such as diameter, sigmaI, and sigmaS. The specific implementation of the bilateral filter algorithm may vary depending on the chosen approach, but it typically involves computing weighted averages of pixel values within a local neighborhood around the current pixel.
- **FilteredImage Initialization**: The filteredImage, which is the output of the bilateral filter, is initialized as a Mat (matrix) of zeros with the same size as the source image. This creates a blank canvas for storing the filtered pixel values.

– **Sequential Processing**: The bilateral filter is applied to each pixel in the source image sequentially, from top to bottom and left to right, following the order of the nested loops. This means that the filter is applied to the first pixel, then the second pixel, and so on, until the last pixel in the image is processed. This sequential processing ensures that each pixel's filtered value is updated in the filtered image matrix according to the bilateral filter algorithm.

Algorithm 5 applyBilateralFilter CPU procedure

function APPLYBILATERALFILTER(source, filteredImage, x, y, diameter, sigmaI, sigmaS)
    $iFiltered \leftarrow 0$
    $wP \leftarrow 0$
    $neighbor\_x \leftarrow 0$
    $neighbor\_y \leftarrow 0$
    $half \leftarrow \frac{diameter}{2}$
    for $i \leftarrow 0$ to $diameter - 1$ **do**
        for $j \leftarrow 0$ to $diameter - 1$ **do**
            $neighbor\_x \leftarrow x - (half - i)$
            $neighbor\_y \leftarrow y - (half - j)$
            $gi \leftarrow \text{gaussian}(source(neighbor\_x, neighbor\_y) - source(x,y), sigmaI)$
            $gs \leftarrow \text{gaussian}(\text{distance}(x,y,neighbor\_x,neighbor\_y), sigmaS)$
            $w \leftarrow gi \cdot gs$
            $iFiltered \leftarrow iFiltered + source(neighbor\_x, neighbor\_y) \cdot w$
            $wP \leftarrow wP + w$
        end for
    end for
    $iFiltered \leftarrow \frac{iFiltered}{wP}$
    $filteredImage(x,y) \leftarrow \text{cast\_to\_unsigned\_char}(iFiltered)$
end function

Algorithm 6 Bilateral Filter Sequential on CPU

```
1: function BILATERALFILTERSEQ(source, diameter, sigmaI, sigmaS)
2:     filteredImage ← Mat::zeros(source.rows, source.cols, CV_64F)
3:     radius ← 0
4:     width ← source.cols
5:     height ← source.rows
6:     for i ← radius to height − radius − 1 do
7:         for j ← radius to width − radius − 1 do
8:             applyBilateralFilter(source, filteredImage, i, j, diameter, sigmaI, sigmaS)
9:         end for
10:    end for
11:    return filteredImage
12: end function
```

5.1.1.2 Parallel

The parallel CPU implementation of the bilateral filter takes advantage of multiple threads to process different rows of the source image concurrently, allowing for potential speedup in processing time compared to a sequential implementation. While implementing the parallel version we ensured proper synchronization and handling of shared resources, such as the filteredImage, to avoid race conditions or other concurrency-related issues. Features of this implementation are:

- **Thread Creation:** A vector of threads is created to store the threads that will be spawned for parallel processing. The number of threads is determined by the numThreads parameter, which specifies how many threads will be used to process the source image concurrently.
- **Row Division:** The source image is divided into rows to distribute the workload among the threads. The height of the image (the number of rows) is divided by the number of threads (numThreads) to determine the number of rows each thread will be responsible for. The rowsPerThread variable stores the number of rows that each thread will process.
- **Thread Processing:** The threads are then created and started using a loop. Each thread is assigned a subset of rows to process concurrently. The start and end variables are used to determine the range of rows that each thread will process. The lambda function inside the thread captures these variables by reference to ensure that each thread operates on its assigned rows independently.
- **applyBilateralFilter Function:** Inside the lambda function, the applyBilateralFil-

ter function is called for each pixel in the assigned rows, passing the source image, filteredImage, and the current pixel's coordinates (i, j) as parameters. This allows each thread to independently apply the bilateral filter to its assigned rows using the given parameters such as diameter, sigmaI, and sigmaS.

- **FilteredImage Update:** As each thread processes its assigned rows, the filteredImage is updated independently. Since each thread is working on a separate subset of rows, there should be no conflicts when updating the filteredImage concurrently.
- **Thread Joining:** After all threads have completed processing their assigned rows, the main thread waits for all the threads to finish using the join() function, which ensures that all threads have completed their processing before proceeding.
- **FilteredImage Return:** Once all threads have completed and joined, the filteredImage is returned, which now contains the final result of the bilateral filter applied to the entire source image.

Algorithm 7 Bilateral Filter Parallel On CPU

```
1:  function BILATERALFILTERPARALLEL(source, diameter, sigmaI, sigmaS, numThreads)
2:      filteredImage ← Mat::zeros(source.rows, source.cols)
3:      radius ← 0
4:      width ← source.cols
5:      height ← source.rows
6:      threads ← empty vector of threads
7:      rowsPerThread ← height / numThreads
8:      start ← radius
9:      end ← start + rowsPerThread
10:     for t ← 0 to numThreads − 1 do
11:         if t == numThreads − 1 then
12:             end ← height - radius
13:         end if
14:         threads.push_back(thread([start, end]()
15:         for i ← start to end do
16:             for j ← radius to width − radius − 1 do
17:                 applyBilateralFilter(source, filteredImage, i, j, diameter, sigmaI, sigmaS)
18:             end for
19:         end for))
20:             start ← end
21:             end ← start + rowsPerThread
22:         end for
23:         for t in threads do
24:             t.join()
25:         end for
26:         return filteredImage
27:     end function
```

5.1.2 GPU Implementation

5.1.2.1 Naive

The kernel function bilateralFilterKernel is parallelized for pixels using CUDA grid and block dimensions. Each thread in the grid is responsible for processing a single pixel in the input image.

The input image is divided into a grid of blocks, with each block containing a number of threads specified by the threadsPerBlock parameter. The number of blocks required is calculated based on the size of the input image and the number of threads per block using the numBlocks parameter.

Each thread in the kernel function accesses the pixel data in the input image and calculates the weighted sum of intensities for the pixels within the filter window centered at the current pixel. The weights for each pixel are calculated based on the distance between pixels in the spatial and intensity domains, using the provided sigma values.

Once the weighted sum and weight values have been calculated for all pixels in the filter window, the filtered output intensity value is computed by dividing the intensity sum by the weighted sum. This value is then written to the output image.

By parallelizing the kernel function for pixels, the bilateral filter is applied efficiently to large images using CUDA-enabled GPUs, resulting in significant performance gains over CPU-based implementations.

5.1.2.2 Kernel Call Function from CPU

The function bilateralFilterCUDA is the CPU function that calls the kernel function bilateralFilterKernel on the GPU. This function takes in the source image, diameter of the filter, sigma values for spatial and intensity domains as input parameters.

Inside the function, it first creates a destination image with the same dimensions as the source image, and initializes it to 0 using the **setTo(0)** function. It then allocates memory on the device for the source and destination images using the **cudaMalloc** function, which returns a pointer to the allocated memory.

The function then copies the source image data from the host to the device memory using cudaMemcpy function. It does the same for the destination image data by copying it from the host memory to the device memory. The function then calculates the number of blocks and threads per block needed for the kernel using the **dim3** data type. The **numBlocks** parameter is calculated by dividing the total number of pixels in the image by the number of threads per block, which is set to **16 x 16**. This ensures that the kernel

function is launched with enough threads to handle all pixels in the image. Once the number of blocks and threads per block are determined, the kernel function bilateralFilterKernel is launched with these parameters using the <<< >>> syntax. This invokes the kernel on the GPU to process each pixel in parallel. After the kernel function completes, the resulting filtered image is copied back to the host memory using the **cudaMemcpy** function. Finally, the function frees memory allocated on the device using the **cudaFree** function and returns the filtered image to the calling function.

Algorithm 8 Bilateral filter CUDA

1: **function** BILATERALFILTERCUDA($src, diameter, sigma_space, sigma_intensity$)
2: $filteredImage \leftarrow$ new Mat with size $src.rows \times src.cols \times$ CV_8UC1
3: $filteredImage.setTo(0)$
4: $dev\_src \leftarrow$ nullptr
5: $dev\_dst \leftarrow$ nullptr
6: $rows \leftarrow src.rows$
7: $cols \leftarrow src.cols$
8: cudaMalloc((void**)dev_src, $rows \times cols \times$ sizeof(unsigned char))
9: cudaMalloc((void**)dev_dst, $rows \times cols \times$ sizeof(unsigned char))
10: cudaMemcpy(dev_src, src.data, $rows \times cols \times$ sizeof(unsigned char), cudaMemcpyHostToDevice)
11: cudaMemcpy(dev_dst, filteredImage.data, $rows \times cols \times$ sizeof(unsigned char), cudaMemcpyHostToDevice)
12: $threadsPerBlock \leftarrow$ dim3(16, 16)
13: $numBlocks \leftarrow$ dim3(($cols + threadsPerBlock.x - 1$) / threadsPerBlock.x, ($rows + threadsPerBlock.y - 1$) / threadsPerBlock.y)
14: bilateralFilterKernel¡¡¡$numBlocks$, $threadsPerBlock$¿¿¿(dev_src, dev_dst, $rows$, $cols$, $diameter$, $sigma\_space$, $sigma\_intensity$)
15: cudaMemcpy(filteredImage.data, dev$_dst$,rows $\times cols \times$ sizeof(unsigned char), cudaMemcpyDeviceToHost)
16: cudaFree(dev_src)
17: cudaFree(dev_dst)
18: **return** $filteredImage$
19: **end function**

5.1.2.3 Kernel Function

The function bilateralFilterKernel is the GPU kernel function that is called by the CPU function bilateralFilterCUDA to perform the actual bilateral filtering on the input image.

This kernel function takes in the source image, the diameter of the filter, sigma values for spatial and intensity domains, and the dimensions of the image as input parameters.

The kernel function is designed to operate on a single pixel at a time. It first calculates the row and column index of the current pixel based on the thread and block indices using the formula:

$$row = threadIdx.x + blockIdx.x * blockDim.x$$

$$col = threadIdx.y + blockIdx.y * blockDim.y$$

- `threadIdx.x` and `threadIdx.y` represent the thread index in the block in the x and y directions, respectively.
- `blockIdx.x` and `blockIdx.y` represent the block index in the grid in the x and y directions, respectively.
- `blockDim.x` and `blockDim.y` represent the number of threads per block in the x and y directions, respectively.
- By multiplying the `blockIdx` by `blockDim`, we can get the starting index of the current block. Adding `threadIdx` gives the current index of the pixel being processed by the current thread.
- The resulting `col` and `row` values represent the current pixel position in the image, which can be used to access the intensity value of the pixel in the source image.

It then checks if the current pixel is within the bounds of the image. If it is, the function applies the bilateral filter to the pixel.

To apply the filter, the function calculates the weighted average of the pixel intensities in a window centered around the current pixel. The size of the window is determined by the diameter of the filter. The function calculates the weight for each pixel in the window based on its spatial distance from the current pixel and its intensity difference with the current pixel. These weights are multiplied with the corresponding pixel intensities and added to obtain the weighted sum.

Finally, the function divides the weighted sum by the sum of the weights to obtain the filtered pixel intensity value, and writes it to the destination image.

Once all pixels have been processed, the kernel function completes and returns control to the CPU function bilateralFilterCUDA.

The use of parallel threads in the kernel function allows the filtering operation to be performed on multiple pixels simultaneously, greatly improving the performance of the bilateral filtering algorithm.

Algorithm 9 Bilateral Filter CUDA

```
1: function BILATERALFILTERKERNEL(src, dst, rows, cols, diameter, sigma_space, sigma_intensity)
2:     x ← blockIdx.x * blockDim.x + threadIdx.x
3:     y ← blockIdx.y * blockDim.y + threadIdx.y
4:     if x ≥ cols or y ≥ rows then
5:         return
6:     end if
7:     intensity_sum ← 0
8:     weight_sum ← 0
9:     for j = −diameter to diameter do
10:        for i = −diameter to diameter do
11:            neighbor_x ← x + i
12:            neighbor_y ← y + j
13:            if neighbor_x < 0 or neighbor_y < 0 or neighbor_x ≥ cols or neighbor_y ≥ rows then
14:                continue
15:            end if
16:            src_index ← neighbor_y * cols + neighbor_x
17:            dst_index ← y * cols + x
18:            intensity_diff ← src[src_index] − src[dst_index]
19:            spatial_distance ← √(i * i + j * j)
20:            weight ← gaussianIntensity(intensity_diff, sigma_intensity) * gaussianSpace(spatial_distance, sigma_space)
21:            intensity_sum ← intensity_sum + src[src_index] * weight
22:            weight_sum ← weight_sum + weight
23:        end for
24:    end for
25:    dst[y * cols + x] ← (unsignedchar)(intensity_sum/weight_sum)
26: end function
```

5.1.2.4 Optimization-1: Precomputing Gaussian Spatial kernel and caching it

An observation can be made that the same spatial kernel is used repeatedly for each pixel during the filtering process. Computing the kernel values for each pixel is a very computationally expensive operation, especially when processing large images, or large kernel sizes. One way to optimize this operation is to precompute the Gaussian spatial kernel and cache the values in memory. We implemented it using a spatial_lookup table

which is a one-dimensional array of size (2*radius+1) x (2*radius+1), where the radius is the half diameter of the filter kernel. The lookup table stores pre-computed values of the Gaussian spatial kernel, which are computed using the spatial distance between pixel locations and the standard deviation of the Gaussian function (sigma_space).

The lookup table is computed using a nested for loop that iterates over all pixel locations within the diameter of the filter kernel. For each pixel location, the spatial distance between that location and the center pixel is computed using the Pythagorean theorem. The spatial distance is then used to compute the Gaussian value using the formula $\exp(-0.5f * spatial\_distance * spatial\_distance / (sigma\_space * sigma\_space))$

The computed values are stored in the lookup table in row-major order. That is, the first (2*radius+1) values represent the first row of the lookup table, the second (2*radius+1) values represent the second row, and so on.

The lookup table is returned as a one-dimensional array of float values, which can be passed to the GPU for use in the bilateral filtering algorithm.

```
float* computeSpatialLookup(int diameter, float sigma_space) {
    int max_distance = 2*radius+1;
    float* spatial_lookup = new float[max_distance*max_distance];
    float spatial_distance;
    int t_i, t_j;

    for (int i = 0; i <= max_distance; i++) {
      for (int j=0; j<=max_distance; j++) {
        t_i = i-radius;
        t_j = j-radius;
        spatial_distance = sqrt((float)(t_i * t_i + t_j * t_j));
        spatial_lookup[i*diameter + j] =
        expf(-0.5f *
        spatial_distance* spatial_distance
        / (sigma_space * sigma_space));
      }
    }

    return spatial_lookup;
}
```

Inside the kernel, The expression spatial_lookup[(i+radius)*(2 * radius + 1)+(j+radius)] is used to access an element in the spatial_lookup table.

First, the i and j values are added to radius to shift them to positive indices, since the spatial_lookup table starts from index 0. This gives the values i+radius and j+radius.

Next, the expression (i+radius)*(2 * radius + 1) calculates the index of the row in which the element is located. The radius is multiplied by (2 * radius + 1) to account for the fact that each row has a (2 * radius + 1) number of elements.

Hence (i+radius)*(2 * radius + 1)+(j+radius) calculates the index of the element in the 1D array spatial_lookup.

There are different memory types available on GPUs, and the choice of memory type for caching the kernel can have a significant impact on performance. We experimented with the following methods:

- **Global Memory**: Global memory is a large, shared memory space that can be accessed by all threads in the grid. However, global memory is slower to access than other memory types, such as shared memory or constant memory. Thus by using global memory, we got the optimization of caching preventing recomputations but with memory access latency that can be improved.
- **Shared Memory**: Shared memory is a small, fast memory space that is shared among threads in a thread block. The spatial kernel values can be loaded into shared memory by a single thread in each block, and all other threads in the block can then access the cached values. This leads to reduced memory access latency and improved performance.
- **Constant Memory**: Constant Memory is a small, read-only memory space that is cached on the GPU. Constant memory has a lower latency and higher bandwidth than global memory, making it faster to access. However, the amount of memory available in constant memory is limited, and the kernel size may need to be reduced to fit in constant memory.

On experimenting with each memory type we concluded the following:

- The shared and constant memory were giving better performances by some fraction of milliseconds than global memory.
- All memories have higher latency than registers and L1 cache. Therefore, accessing the spatial kernel from shared or constant memory may still be slower than computing the values on the fly if the kernel size is small.
- The amount of shared memory available on a GPU is usually much smaller than that of global memory, and it is shared among all threads in a block. Therefore, if the kernel size is too large, it may not be possible to cache the entire kernel in shared memory, which can limit the potential performance gains.

- Another issue with shared memory usage is that it requires careful management to avoid bank conflicts. Shared memory is divided into multiple memory banks, and if multiple threads access the same bank simultaneously, it can cause a performance penalty. Therefore, when using shared memory to cache the spatial kernel, it is important to ensure each thread accesses a different bank to avoid conflicts.
- Constant memory has a larger capacity than shared memory but is typically smaller than global memory, so it may not be able to cache very large kernels.

5.1.2.5 Optimization-2: Precomputing Gaussian Range Differences and caching it

Since the range kernel is computed dynamically at every pixel it cant be precomputed, but the optimization that can be used to speed up the computation of the range kernel is to cache the intensity difference values for each pixel in the image which is an integer between 0 to 255 since every pixel value is an 8-bit value. This can be done by precomputing the intensity lookup table using the computeIntensityLookup() function,

```
float* computeIntensityLookup(float sigma_intensity) {

    float* intensity_lookup = new float[256];
    for (int i = 0; i < 256; i++) {
        intensity_lookup[i] = expf(-0.5f * i*i /
        (sigma_intensity * sigma_intensity));
    }

    return intensity_lookup;
}
```

Once the intensity lookup table has been computed, the range kernel can be computed by simply looking up the precomputed intensity difference values for each pair of pixels in the image. This can significantly reduce the amount of computation required to compute the range kernel, particularly for large images.

Caching the intensity difference values in this also helped to reduce memory bandwidth requirements, as the intensity lookup table can be stored in fast on-chip memory, such as the shared memory of a GPU. This can help to minimize the number of memory accesses required during the computation of the range kernel, which can be a bottleneck for performance in some cases.

As discussed in the above section the same reasoning holds caching in global, shared, constant memory.

5.1.2.6 Optimization-3: Using shared memory to access source image

In this optimization technique we have loaded the image data into shared memory which is done by each thread block, which is a portion of the image that will be processed together by the threads in that block. The image data is loaded into shared memory in a tiled fashion, with each thread block loading a subset of the image into its shared memory allowing for faster access to the image data during computation, since the data is stored in the fast on-chip memory of the GPU.

The shared memory is allocated as a two-dimensional array

`shared_src`[`THREADS_PER_BLOCK_Y` $+ 2 *$ `Radius`][`THREADS_PER_BLOCK_X` $+ 2 *$ `Radius`].

Here, `THREADS_PER_BLOCK_Y` and `THREADS_PER_BLOCK_X` are the number of threads in the y and x directions respectively, and `Radius` is the radius of the filter kernel. The extra $2 *$ `Radius` space in each dimension is for loading the border pixels from neighboring blocks.

Each thread loads its own pixel from the global memory to shared memory, with the index `shared_src`[`shared_y`][`shared_x`] = `src`[$y *$ `cols` $+ x$];. The indices `shared_x` and `shared_y` represent the location of the thread in shared memory, while x and y represent the location of the thread in global memory.

To load the border pixels from neighboring blocks, the threads in the first `Radius` columns and rows of each block load the corresponding pixels from the global memory, using the ternary operator to check if the pixel is within the bounds of the image.

For example, `shared_src`[`shared_y`][`shared_x` $-$ `Radius`] $= (x -$ `Radius` $\geq 0)$?`src`[$y *$ `cols` $+$ $x -$ `Radius`] : 0; loads the pixel to the left of the current pixel, if it exists, or sets it to 0 otherwise.

Once the image data is loaded into shared memory, the threads can access it efficiently using the indices `neighbor_shared_x` $=$ `shared_x` $+ i$ and `neighbor_shared_y` $=$ `shared_y` $+ j$, which represent the coordinates of the neighboring pixels in shared memory relative to the current pixel.During the computation of the bilateral filter, the intensity difference is calculated using the neighboring pixels in shared memory using the following line of code:

```
intensity_diff = abs(shared_src[neighbor_shared_y]
[neighbor_shared_x] - shared_src[shared_y][shared_x]);
```

This is a fast operation since the data is already in shared memory and does not need to be loaded from global memory. Along with this shared memory optimization, we have optimized it further with the precompute and caching techniques described in 5.1.2.4,5.1.2.5 we cached the Gaussian spatial kernel, and the Gaussian values at all the possible intensity differences, and we kept all the caches in global memory for saving the limited shared memory for loading images.

Algorithm 10 Bilateral Filter CUDA Shared Memory Kernel

function BILATERALFILTERKERNEL(src, dst, rows, cols, radius, σ_{space}, $\sigma_{\text{intensity}}$)

$x \leftarrow \text{blockIdx.x} \times \text{blockDim.x} + \text{threadIdx.x}$

$y \leftarrow \text{blockIdx.y} \times \text{blockDim.y} + \text{threadIdx.y}$

if $x \geq \text{cols}$ **or** $y \geq \text{rows}$ **then**

return

end if

Allocate shared memory for a subset of the source image

__shared unsigned char shared_src[THREADS_PER_BLOCK_Y + 2 × radius][THREADS_PER_BLOCK_X + 2 × radius]

Compute the indices for the shared memory subset

shared_x $\leftarrow$ threadIdx.x + radius

shared_y $\leftarrow$ threadIdx.y + radius

Load the subset of the source image into shared memory

shared_src[shared_y][shared_x] $\leftarrow$ src[$y \times \text{cols} + x$]

if threadIdx.x $<$ radius **then**

shared_src[shared_y][shared_x − radius] $\leftarrow$ ($x - \text{radius} \geq 0$) **?** src[$y \times \text{cols} + x -$ radius] **:** 0

shared_src[shared_y][shared_x + THREADS_PER_BLOCK_X] $\leftarrow$ (x + THREADS_PER_BLOCK_X $<$ cols) **?** src[$y \times \text{cols} + x +$ THREADS_PER_BLOCK_X] **:** 0

end if

if threadIdx.y $<$ radius **then**

shared_src[shared_y - radius][shared_x] $\leftarrow$ (y - radius $\geq$ 0) ? src[(y - radius) * cols + x] : 0

shared_src[shared_y + THREADS_PER_BLOCK_Y][shared_x] $\leftarrow$ (y + THREADS_PER_BLOCK_Y $<$ rows) ? src[(y + THREADS_PER_BLOCK_Y) * cols + x] : 0

end if

if threadIdx.x $<$ radius & threadIdx.y $<$ radius **then**

shared_src[shared_y - radius][shared_x - radius] $\leftarrow$ (x - radius $\geq$ 0 & y - radius $\geq$ 0) ? src[(y - radius) * cols + x - radius] : 0

shared_src[shared_y - radius][shared_x + THREADS_PER_BLOCK_X] $\leftarrow$ (x + THREADS_PER_BLOCK_X $<$ cols & y - radius $\geq$ 0) ? src[(y - radius) * cols + x + THREADS_PER_BLOCK_X] : 0

shared_src[shared_y + THREADS_PER_BLOCK_Y][shared_x - radius] $\leftarrow$ (x - radius $\geq$ 0 & y + THREADS_PER_BLOCK_Y $<$ rows) ? src[(y + THREADS_PER_BLOCK_Y) * cols + x - radius] : 0

shared_src[shared_y + THREADS_PER_BLOCK_Y][shared_x + THREADS_PER_BLOCK_X] $\leftarrow$ (x + THREADS_PER_BLOCK_X $<$ cols & y + THREADS_PER_BLOCK_Y $<$ rows) ? src[(y + THREADS_PER_BLOCK_Y) * cols + x + THREADS_PER_BLOCK_X] : 0

end if

synchronize threads

$intensity\_sum \leftarrow 0$

$weight\_sum \leftarrow 0$

for $j \leftarrow -radius$ **to** $radius$ **do**

  for $i \leftarrow -radius$ **to** $radius$ **do**

    $neighbor\_shared\_x \leftarrow shared\_x + i$

    $neighbor\_shared\_y \leftarrow shared\_y + j$

    $src\_index \leftarrow (y + j) * cols + (x + i)$

    $dst\_index \leftarrow y * cols + x$

    $intensity\_diff \leftarrow |, shared_src[neighbor\_shared\_y][neighbor\_shared\_x] - shared_src[shared\_y][shared\_x], |$

    $spatial\_weight \leftarrow gaussianSpace(\sqrt{i^2 + j^2}, \sigma\_space)$

    $intensity\_weight \leftarrow gaussianIntensity(intensity\_diff, \sigma\_intensity)$

    $weight \leftarrow intensity_weight * spatial_weight$

    $intensity\_sum \leftarrow intensity\_sum + shared\_src[neighbor\_shared\_y][neighbor\_shared\_x] * weight$

    $weight\_sum \leftarrow weight_sum + weight$

  end for

end for

$dst[y * cols + x] \leftarrow (intensity\_sum/(weight\_sum))$

=0

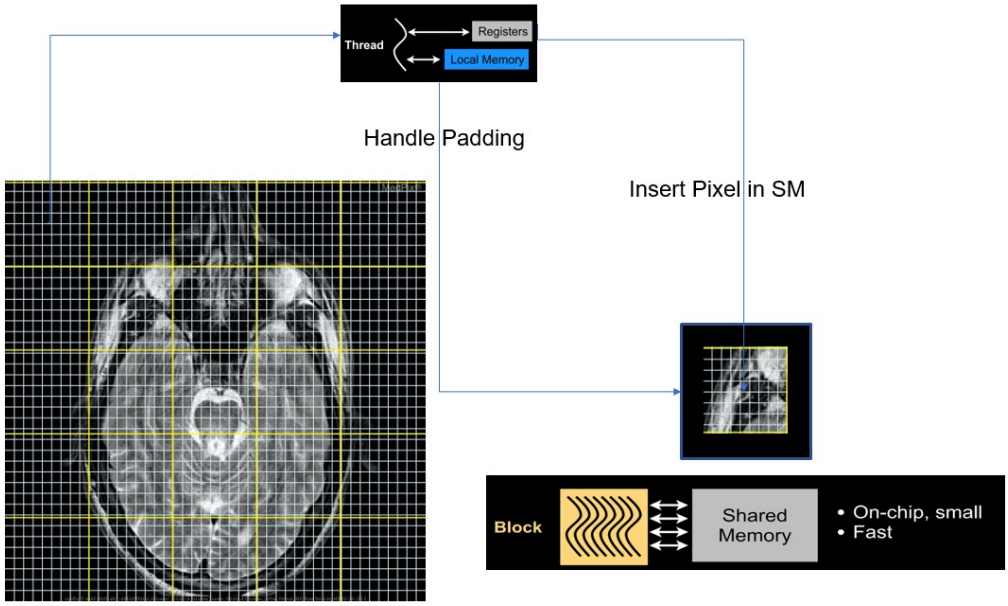


Figure 5.1:

Tiling Source Image in shared memory

5.2 Adaptive Bilateral Filter

5.2.1 CPU Implementation

The adaptive bilateral filter shares many similarities with the bilateral filter in terms of its execution. However, there is an additional step in the adaptive bilateral filter where a new sigma value is calculated for each pixel based on the local image statistics. Despite this difference, the methodology for implementing the adaptive bilateral filter is similar to that of the bilateral filter.

Both the sequential and parallel implementations of the adaptive bilateral filter involve the same steps as those of the bilateral filter. The main difference lies in the calculation of the new sigma value for each pixel, which is based on the local image statistics. However, the overall implementation details are similar to those explained in the previous section for the bilateral filter in 5.1.1.1,5.1.1.2.

5.2.2 GPU Implementation

The GPU implementation of the adaptive bilateral filter is similar to the bilateral filter implementation discussed earlier. The naive implementation involves launching a kernel for each

pixel, which can result in significant overhead due to thread creation and memory access patterns. To address this issue, we can take advantage of the massive parallelism offered by the GPU architecture by launching a grid of blocks, where each block processes a subset of pixels.

Looking at the optimizations discussed in the previous section we can conclude the following for Adaptive bilateral filter

- We can Cache the Gaussian spatial kernel allowing for faster access to the kernel and reducing compute latency(sec 5.1.2.4)
- We can use shared memory to load tiles of images that need to be processed by the threads in a block.
- We cannot precompute and cache the intensity difference values as a new sigma is calculated for every neighborhood

.

5.3 Non Local Means + Bilateral Filter

5.3.1 CPU Implementation

The Non-local means (explained in sec 4.3) plus bilateral filter is a more complex algorithm compared to the bilateral and adaptive bilateral filters. However, it also shares some similarities in its execution and implementation methodology.

Like the bilateral filter, the non-local means plus the bilateral filter uses a spatial filter and a range filter to filter the image. However, instead of using a fixed window size, the non-local means plus bilateral filter employs a non-local means filter to estimate the similarity between pixels. This additional step helps to preserve more details in the filtered image.

In terms of implementation details, the non-local means plus bilateral filter can be implemented in a similar way to the bilateral and adaptive bilateral filters as the sequential and parallel implementations of the non-local means plus bilateral filter follow similar steps to those of the bilateral. The main difference lies in the non-local means filtering step, which involves comparing every pixel in the image to every other pixel, making it computationally very expensive requiring more memory resources and computation power

5.3.2 GPU Implementation

The implementation of the Non-local means plus bilateral filter on GPU also shares similarities with the implementations of the bilateral and adaptive bilateral filters. The naive implemen-

tation would involve launching a kernel for each pixel, leading to significant overheads. To improve the performance, the GPU implementation can be parallelized by launching a grid of blocks, with each block processing a subset of pixels. Looking at the optimizations discussed in the previous section we can conclude the following for Adaptive bilateral filter

- There is no need of employing a Gaussian spatial kernel as discussed in 4.3.3 so Gaussian kernel optimization has no merit for this algorithm
- We cannot use shared memory to load source image since the search window is the whole image,for the large images each block getting the whole image in shared memory is practically impossible because of its limited size , hence the image needs to be in global memory
- We cannot precompute and cache the intensity difference values as a new sigma is calculated for every neighborhood

.

5.4 Summary of methodologies used and experimented with

Implementation			
	Global	Shared	Const
Bilateral GPU Naive	-	-	-
Bilateral GPU Kernel Cached	-	✓	-
Bilateral Shared Memory	-	-	-
Bilateral Shared Memory+ Cached Kernels	✓	-	-
Adaptive Bilateral Naive	-	-	-
Adaptive Bilateral Optimized	✓	-	-
NL Means Bilateral Naive	-	-	-
NL Means Bilateral Optimized	-	-	-

Table 5.1: Gaussian Spatial Kernel

Implementation			
	Global	Shared	Const
Bilateral GPU Naive	-	-	-
Bilateral GPU Kernel Cached	-	✓	-
Bilateral Shared Memory	-	-	-
Bilateral Shared Memory+ Cached Kernels	✓	-	-
Adaptive Bilateral Naive	-	-	-
Adaptive Bilateral Optimized	-	-	-
NL Means Bilateral Naive	-	-	-
NL Means Bilateral Optimized	-	✓	-

Table 5.2: Gaussian of Intensity difference

Implementation			
	Global	Shared	Const
Bilateral GPU Naive	✓	-	-
Bilateral GPU Kernel Cached	✓	-	-
Bilateral Shared Memory	-	✓	-
Bilateral Shared Memory+ Cached Kernels	-	✓	-
Adaptive Bilateral Naive	✓	-	-
Adaptive Bilateral Optimized	-	✓	-
NL Means Bilateral Naive	✓	-	-
NL Means Bilateral Optimized	✓	-	-

Table 5.3: Source Image

Chapter 6

Results and Experiments

6.1 Setup

6.1.1 Hardware Used

For executing sequential and parallel implementations, we used the CPUs provided by Google Colab : Intel(R) Xeon(R) CPU @ 2.20GHz:

Specification	**Value**
Base Clock Speed	2.20 GHz
Number of Cores	2
Number of Threads	4
Cache	4 MB SmartCache
Memory Support	Up to 768 GB DDR4
TDP	65 watts

Table 6.1: Hardware Specifications of Intel(R) Xeon(R) CPU @ 2.20GHz

For executing the GPU implementations, we used the GPUs provided by Google Colab, which is the Nvidia Tesla K80 GPU which is a powerful GPU designed for use in high-performance computing and machine learning applications that require massive parallel processing capabilities and a large amount of memory. Its advanced architecture, large number of CUDA cores, and high memory bandwidth make it well-suited for these types of workloads. It is a high-performance computing accelerator developed by NVIDIA for large data processing applications such as machine learning, deep learning, and scientific computing.

Source : `https://www.nvidia.com/en-gb/data-center/tesla-k80/`

Specification	Value
GPU Architecture	Kepler
Number of CUDA Cores	4992
Base Clock Speed	562 MHz
Boost Clock Speed	875 MHz
Memory Support	24 GB GDDR5, 480 GB/s
Power Consumption	300 watts

Table 6.2: Hardware Specifications of Tesla K80

6.1.2 Images Used

We have used the following three images for conducting experiments:

(a) Car image **(b)** Cameraman image **(c)** Brain image

Figure 6.1: Images used to demonstrate results.

The reasons for choosing these images are:

- The car image was used by us during testing because of evident edges (near the brand logo and headlights) which were being blurred by traditional denoising algorithms, hence we were able to asses the performance of our algorithm in terms of perception visual quality.
- The cameraman image was used because it is a standard image that is used by many research papers to demonstrate their results.
- The brain image was used to demonstrate how useful bilateral filter is in medical image applications.

6.1.3 Noise Used

We added noise to the image before applying the bilateral filter to observe how effectively it works. Specifically, we have added Gaussian noise to the images as it is often encountered in images due to electronic sensor noise in digital cameras, thermal noise in imaging sensors, or noise introduced during image transmission and storage.

6.1.4 Metrics Measured

We have used the following metrics to compare the results of various implementations:

- Time Measure
- PSNR
- GPU occupancy

6.1.4.1 Time Measure

As one of the main aims of this project was to reduce the execution time of bilateral filters, we have decided to compare the execution times of various implementations of bilateral filters. **Note:** For GPU-based implementations we haven't taken into account the time taken for allocating memory on the device and transferring data from host to device.

6.1.4.2 PSNR

Peak-Signal-to-Noise-Ratio, or PSNR, is a metric used to compare the original and reconstructed images' quality. When compressing images and videos, it's common practice to compare their quality to the original input with PSNR as a metric. It is important to keep in mind that this is not a perfect comparison metric because it often does not reflect how humans perceive quality.

$$PSNR = 10 \cdot \log_{10} \left(\frac{MAX_I^2}{MSE} \right) \tag{6.1}$$

6.1.4.3 GPU Occupancy

GPU occupancy denotes how much if the available GPU resources are utilized by the application. It is the percentage of SMs (Streaming multi processors) that are processing data at a given time. The higher the GPU occupancy, the more is the performance as higher occupancy denotes better utilization of the available resources.

6.2 Experiments with Car image:

6.2.1 Comparison between Sequential, Parallel and GPU Naive

6.2.1.1 Experiment

In this experiment, we have varied the size of the filter and observed the time taken by different implementations of bilateral filter i.e.: Sequential, Parallel implementation on CPU, GPU naive implementation on car image.

6.2.1.2 Results

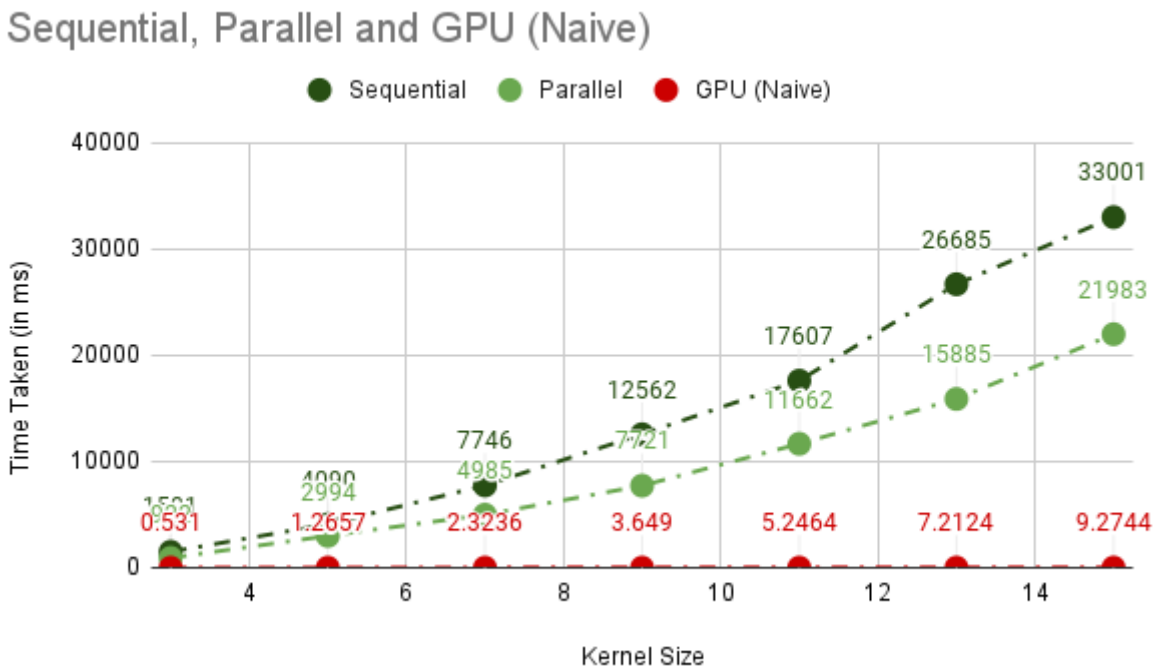


Figure 6.2: Comparison of execution times (in milliseconds (ms)) between Sequential, Parallel, and GPU Naive with varying filter size (Car image)

Filter Radius	Sequential	Parallel	GPU (Naive)
3	1501	922	0.531
5	4090	2994	1.2657
7	7746	4985	2.3236
9	12562	7721	3.649
11	17607	11662	5.2464
13	26685	15885	7.2124
15	33001	21983	9.2744

Table 6.3: Comparison of execution times (in milliseconds (ms)) between Sequential, Parallel and GPU Naive with varying filter size (Car image)

6.2.1.3 Conclusion

As the results suggest, the time taken by GPU Naive is significantly less compared to the parallel implementation on CPU which is comparatively faster than sequential. This is because, as GPUs have thousands of cores, all the pixels of the output image are computed simultaneously. In parallel implementation on the CPU, the cores available on the CPU are not many (only 2), so it takes more time to compute the output image than in GPU Naive implementation. In sequential, as the output pixels are calculated one after the other, the time taken is significantly more compared to other implementations.

The parallel implementation didn't speed up much when compared to sequential, this is because even if we deployed multiple threads to compute the output image, the hardware on which we ran our experiments had on 2 processors, so due to hardware limitation, the speedup wasn't as much as we would expect it to be.

As the size of the filter increases, we can observe that the time taken by each implementation also increases. This is because as the kernel size increases, the number of pixels taken to compute an output pixel also increases (since we calculate the output pixel based on all the neighbouring pixels within the window size), so the time taken increases.

The **speedup** achieved by parallel is $\approx \mathbf{1.5}$, whereas the speedup achieved by GPU Naive implementation is $\approx \mathbf{3558.28}$ when compared to sequential on kernel radius of 15.

6.2.2 Comparison between Naive, Kernel Caching, Shared memory, Shared Memory Optimization+Kernel Caching

6.2.2.1 Experiment

In this experiment, we have varied the size of the filter and observed the time taken by different GPU implementations of bilateral filter i.e. Naive, Kernel Caching, Shared memory, Shared Memory Optimization+Kernel Caching on the car image.

6.2.2.2 Results

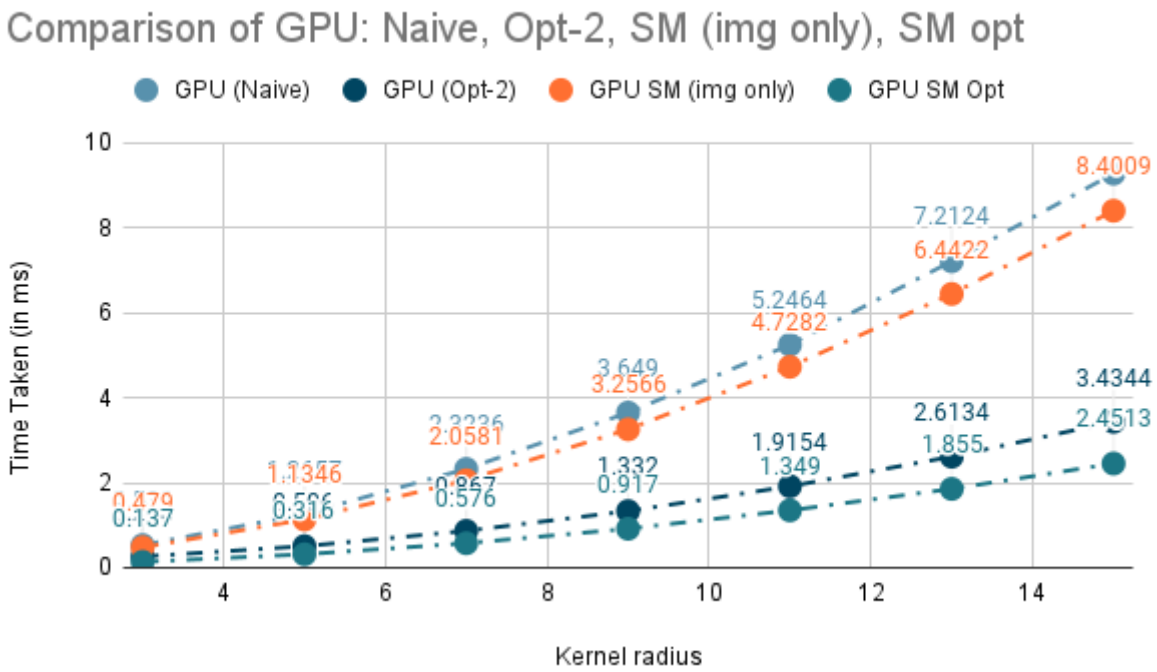


Figure 6.3: Comparison of execution times (in milliseconds (ms)) between GPU's Naive, Kernel Caching, Shared memory, Shared Memory Optimization+Kernel Caching (Car image)

Filter Radius	Naive	Kernel Caching	Shared memory	Shared Memory Optimization+Kernel Caching
3	0.531	0.263	0.479	0.137
5	1.2657	0.506	1.1346	0.316
7	2.3236	0.867	2.0581	0.576
9	3.649	1.332	3.2566	0.917
11	5.2464	1.9154	4.7282	1.349
13	7.2124	2.6134	6.4422	1.855
15	9.2744	3.4344	8.4009	2.4513

Table 6.4: Comparison of execution times (in milliseconds (ms)) between GPU's Naive, Kernel Caching, Shared memory, Shared Memory Optimization+Kernel Caching (Car image)

6.2.2.3 Conclusion

As the results suggest, Naive implementation is slower than all the other implementations. After that, the GPU Shared memory Optimization in which image data is kept in shared memory works slightly better than the naive implementation. This is because, since image data is stored in shared memory, the time taken to access it faster (discussed in memory hierarchy section), which makes the overall execution time faster.

GPU Kernel Caching Optimization is the next best variation after GPU shared memory Optimization implementation. In this implementation, we have optimized naive implementation to precompute spatial kernel and range kernel at the beginning instead of computing it for every pixel. Also, these precomputed kernels are stored in shared memory for faster read access. Since the number of computations is reduced and those are stored in shared memory, it outperformed both naive and the shared memory Optimization implementations.

Shared Memory Optimization+Kernel Caching implementation outperforms all the variations. It is a combination of GPU shared memory and GPU Kernel Caching Optimization implementations. In this variation, we keep the image data in shared memory and also precomputed both the spatial and range kernels, and stored them in global memory. Since computations are less and image data is faster to access, this is the most optimal of all the other variations.

The **speedup** achieved by Shared memory is $\approx \mathbf{3928.26}$, Kernel Caching is $\approx \mathbf{9608.95}$, and Shared Memory+Kernel Caching is $\approx \mathbf{13462.65}$ on kernel radius of 15 when compared to sequential.

6.2.3 Comparison between Sequential. Parallel and GPU Naive of Adaptive bilateral filter

6.2.3.1 Experiment

In this experiment, we have varied the size of the filter and observed the time taken by different implementations of adaptive bilateral filter i.e. Sequential, Parallel implementation on CPU, GPU naive implementation on car image.

6.2.3.2 Results

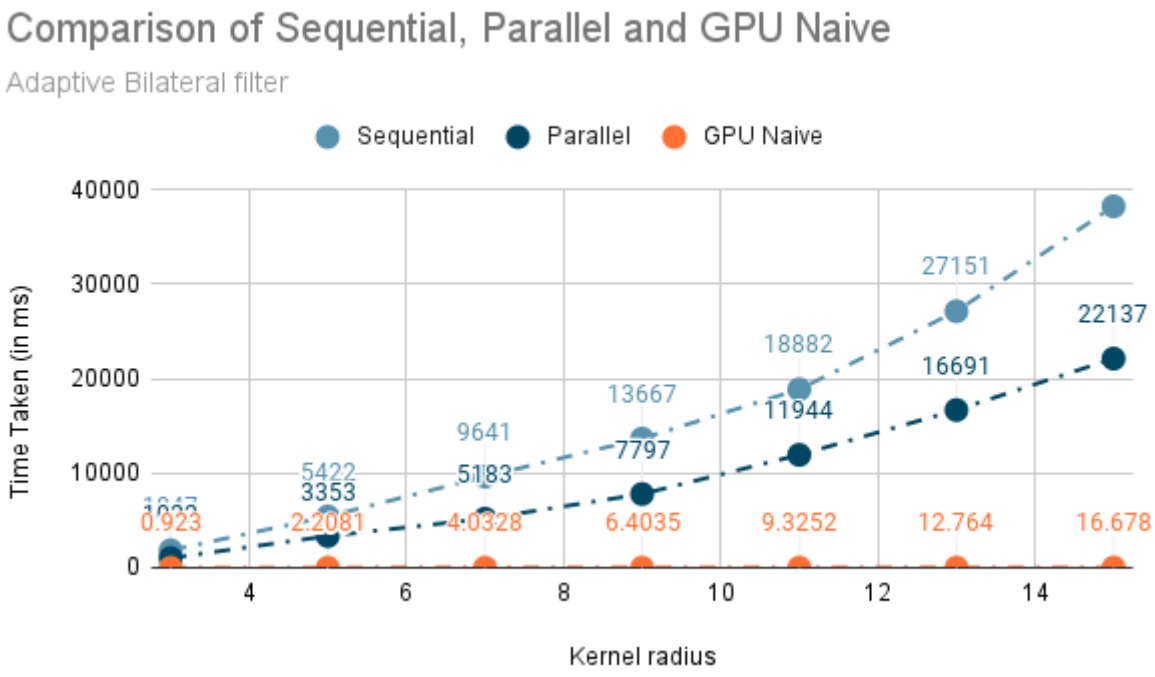


Figure 6.4: Comparison of execution times (in milliseconds (ms)) of adaptive bilateral filter between Sequential, Parallel and GPU Naive with varying filter size (Car image)

Filter Radius	Sequential	Parallel	GPU (Naive)
3	1847	1022	0.923
5	5422	3353	2.2081
7	9641	5183	4.0328
9	13667	7797	6.4035
11	18882	11944	9.3252
13	27151	16691	12.764
15	38256	22137	16.678

Table 6.5: Comparison of execution times (in milliseconds (ms)) between Sequential, Parallel and GPU Naive of the adaptive bilateral filter with varying filter size (Car image)

6.2.3.3 Conclusion

The results observed are as expected which are explained in the bilateral filter experiments. Moreover, the time taken by each implementation is higher compared to the bilateral filter as in the adaptive bilateral filter, we first calculate sigma based on the variance and then apply it, so it requires more computations than bilateral.

6.2.4 Comparison between GPU Naive and Shared memory Optimization implementations of Adaptive bilateral filter

6.2.4.1 Experiment

In this experiment, we have varied the size of the filter and observed the time taken by different GPU implementations of adaptive bilateral filter i.e. Naive and Shared memory Optimization on the car image.

6.2.4.2 Results

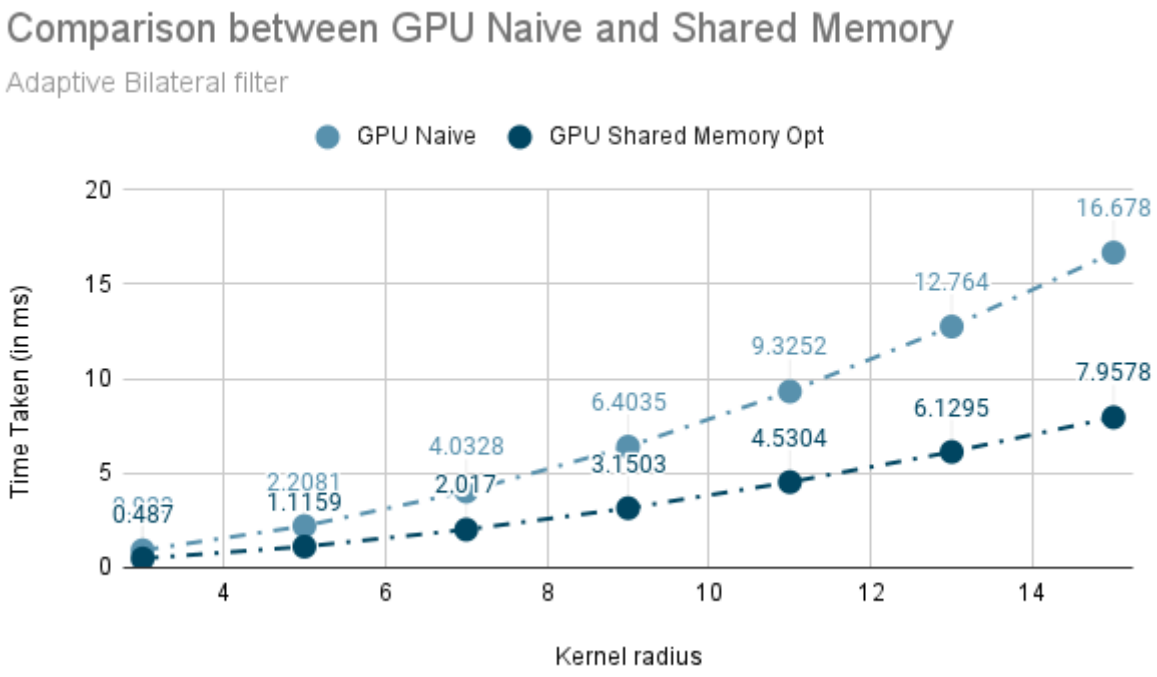


Figure 6.5: Comparison of execution times (in milliseconds (ms)) between GPU Naive and GPU Shared memory + Kernel Caching of the adaptive bilateral filter with varying filter size (Car image)

Filter Radius	GPU (Naive)	GPU Shared memory + Kernel Caching
3	0.923	0.487
5	2.2081	1.1159
7	4.0328	2.017
9	6.4035	3.1503
11	9.3252	4.5304
13	12.764	6.1295
15	16.678	7.9578

Table 6.6: Comparison of execution times (in milliseconds (ms)) between GPU Naive and GPU Shared memory + Kernel Caching of the adaptive bilateral filter with varying filter size (Car image)

6.2.4.3 Conclusion

As the results suggest, the time taken by GPU Shared memory optimization is less compared to the naive implementation as in the shared memory optimization, we have precomputed both the kernels, also placed the image data in shared memory to reduce the data access time.

6.2.5 Comparison between NL-means GPU-Kernel Optimization in global Vs shared memory

6.2.5.1 Experiment

In this experiment, we have varied the size of the filter and observed the time taken by different GPU implementations of NL-means bilateral filter i.e. kernel in global memory Vs shared memory on car image.

6.2.5.2 Results

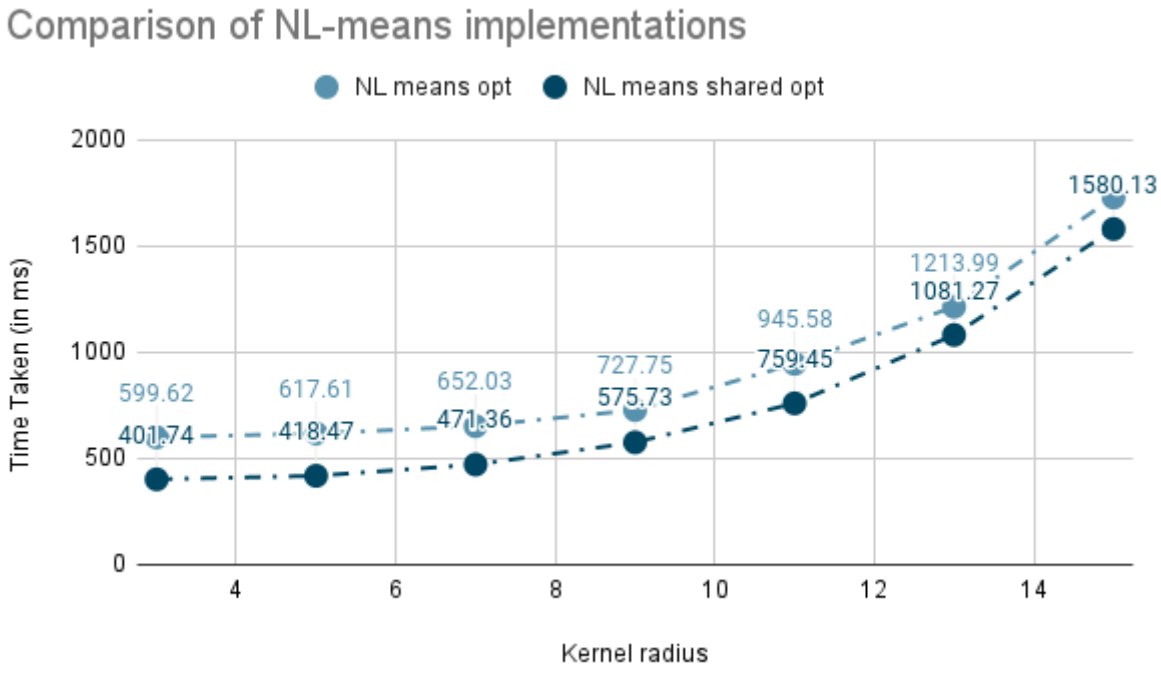


Figure 6.6: Comparison of execution times (in milliseconds (ms)) between different GPU implementations of NL-means bilateral filter i.e. kernel in global memory Vs shared memory with varying kernel size (Car image)

Filter Radius	Kernel (in global memory)	Kernel (in shared memory)
3	599.62	401.74
5	617.61	418.47
7	652.03	471.36
9	727.75	575.73
11	945.58	759.45
13	1213.99	1081.27
15	1728.28	1580.13

Table 6.7: Comparison of execution times (in milliseconds (ms)) between different GPU implementations of NL-means bilateral filter i.e. kernel in global memory Vs shared memory with varying kernel size (Car image)

6.2.5.3 Conclusion

As the above results suggest, the time taken when the kernel is placed in shared memory is less compared to when they are placed in global memory as the time taken to access the kernel data decreased since it is placed in shared memory.

6.3 Experiments with Cameraman image:

6.3.1 Comparison between Sequential, Parallel and GPU Naive

6.3.1.1 Experiment

In this experiment, we have varied the size of the filter and observed the time taken by different implementations of bilateral filter i.e. Sequential, Parallel implementation on CPU, GPU naive implementation on cameraman image.

6.3.1.2 Results

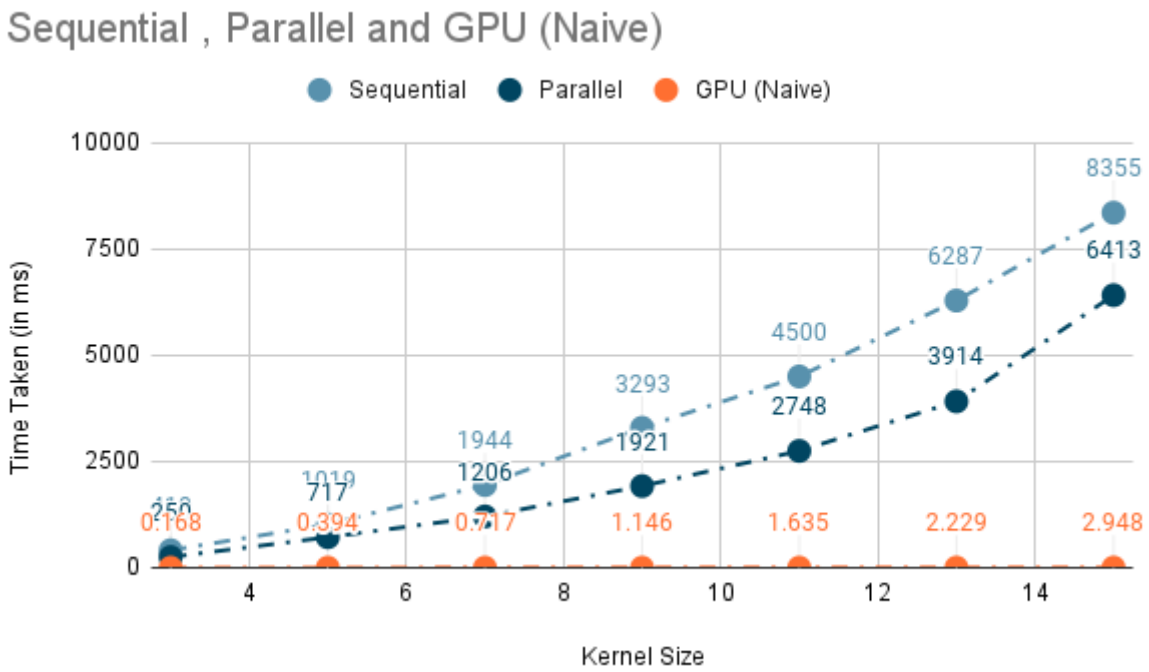


Figure 6.7: Comparison of execution times (in milliseconds (ms)) between Sequential, Parallel and GPU Naive with varying filter size (Cameraman image)

Filter Radius	Sequential	Parallel	GPU (Naive)
3	410	250	0.168
5	1019	717	0.394
7	1944	1206	0.717
9	3293	1921	1.146
11	4500	2748	1.635
13	6287	3914	2.229
15	8355	6413	2.948

Table 6.8: Comparison of execution times (in milliseconds (ms)) between Sequential, Parallel and GPU Naive with varying filter size (Cameraman image)

6.3.1.3 Conclusion

The results observed are as expected which are explained in the experiments of car image section. We can also observe that the time taken for each implementation is less when compared to car image, this is because of the difference in the size of the images. The size of cameraman image is less when compared to car image, so the time taken is less.

The **speedup** achieved by parallel is $\approx \mathbf{1.302}$, whereas the speedup achieved by GPU Naive implementation is $\approx \mathbf{2834.12}$ when compared to sequential on kernel radius of 15.

6.3.2 Comparison between GPU Naive, Kernel Caching Optimization, Shared memory Optimization and Shared Memory Optimization+Kernel Caching

6.3.2.1 Experiment

In this experiment, we have varied the size of the filter and observed the time taken by different GPU implementations of bilateral filter i.e: Naive, Kernel Caching Optimization, Shared memory Optimization and Shared Memory Optimization+Kernel Caching on cameraman image.

6.3.2.2 Results

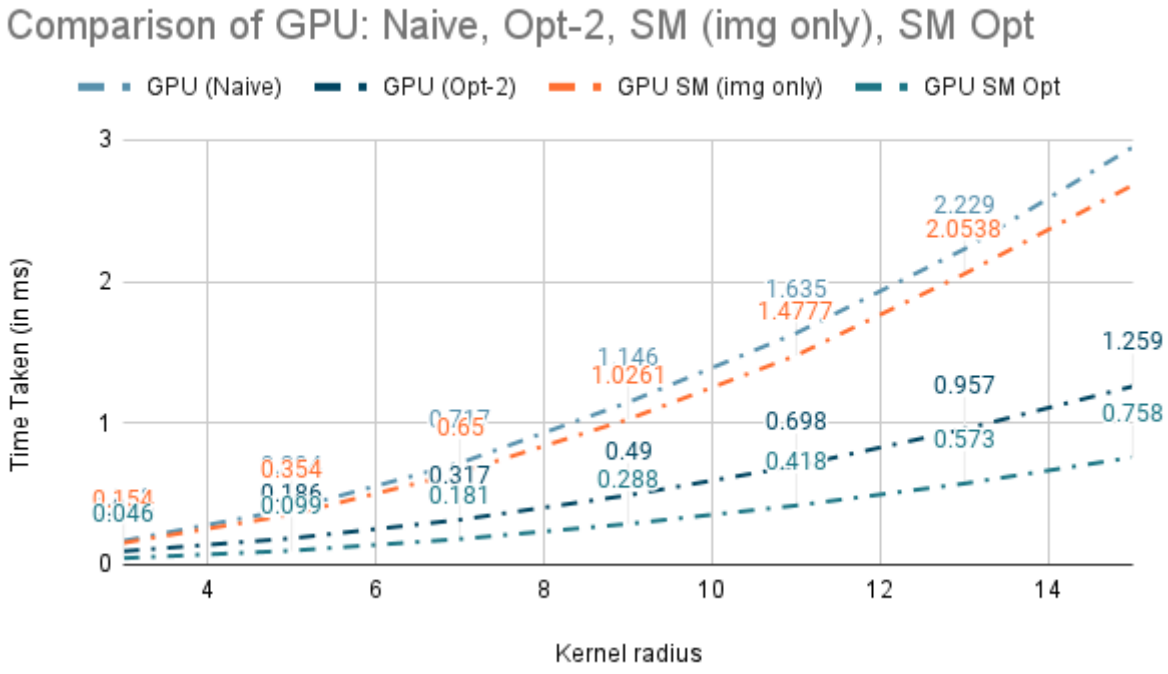


Figure 6.8: Comparison of execution times (in milliseconds (ms)) between GPU's Naive, Kernel Caching Optimization, Shared memory Optimization and Shared Memory Optimization+Kernel Caching (Cameraman image)

Filter Radius	Naive	Kernel Caching Optimization	Shared Memory Optimization	Shared Memory Optimization+Kernel Caching
3	0.168	0.095	0.154	0.046
5	0.394	0.186	0.354	0.099
7	0.717	0.317	0.65	0.181
9	1.146	0.49	1.0261	0.288
11	1.635	0.698	1.4777	0.418
13	2.229	0.957	2.0538	0.573
15	2.948	1.259	2.6802	0.758

Table 6.9: Comparison of execution times (in milliseconds (ms)) between GPU's Naive, Kernel Caching Optimization, Shared memory Optimization and Shared Memory Optimization+Kernel Caching (Cameraman image)

6.3.2.3 Conclusion

The results observed are as expected, which is explained in car image experiments, with Shared Memory Optimization+Kernel Caching being the fastest among all the implementations followed by Kernel Caching Optimization variation, The next fast variation is GPU Shared memory Optimization(img only), then GPU Naive which is the slowest of all.

The **speedup** achieved by Shared memory is $\approx \mathbf{3117.3}$, Optimization-2 is $\approx \mathbf{6636.21}$, Shared memory optimization is $\approx \mathbf{11022.42}$ on kernel radius of 15 when compared to sequential.

6.3.3 Comparison between Sequential. Parallel and GPU Naive of Adaptive bilateral filter

6.3.3.1 Experiment

In this experiment, we have varied the size of the filter and observed the time taken by different implementations of adaptive bilateral filter i.e. Sequential, Parallel implementation on CPU, GPU naive implementation on cameraman image.

6.3.3.2 Results

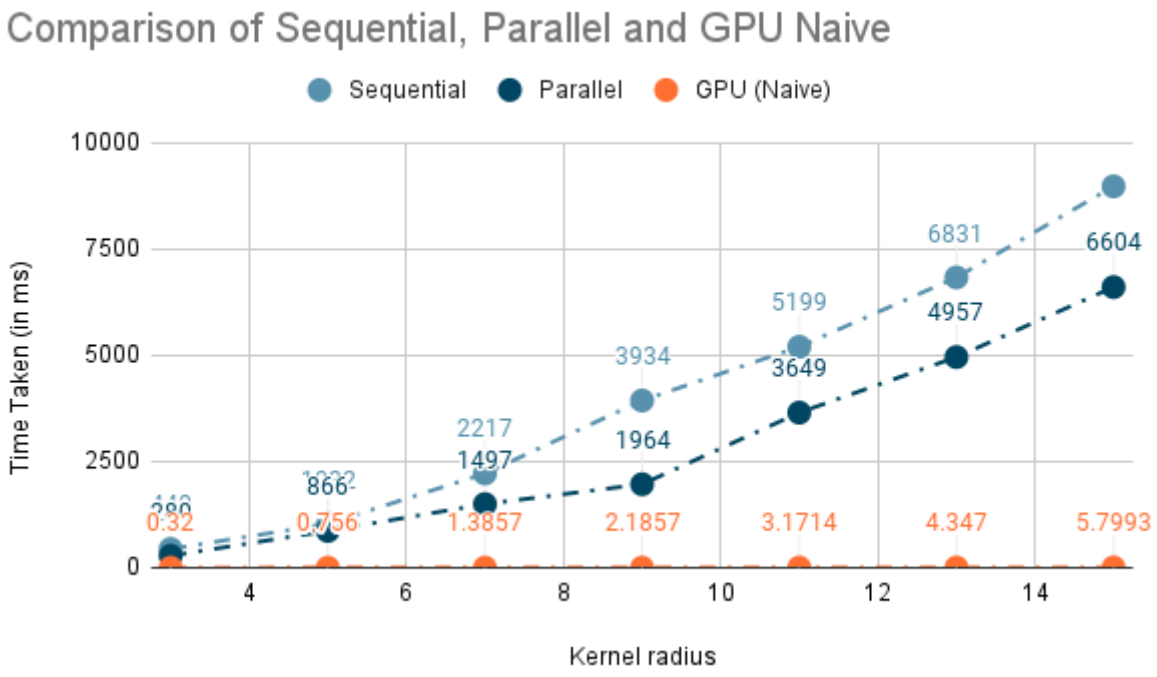


Figure 6.9: Comparison of execution times (in milliseconds (ms)) of adaptive bilateral filter between Sequential, Parallel and GPU Naive with varying filter size (Cameraman image)

Filter Radius	Sequential	Parallel	GPU (Naive)
3	440	280	0.32
5	1032	866	0.756
7	2217	1497	1.3857
9	3934	1964	2.1857
11	5199	3649	3.1714
13	6831	4957	4.347
15	8975	6604	5.7993

Table 6.10: Comparison of execution times (in milliseconds (ms)) of adaptive bilateral filter between Sequential, Parallel and GPU Naive with varying filter size (Cameraman image)

6.3.3.3 Conclusion

The results observed are as expected which are explained in the bilateral filter experiments. Moreover, the time taken by each implementation is higher compared to the bilateral filter as in the adaptive bilateral filter, we first calculate sigma based on the variance and then apply it, so it requires more computations than bilateral.

6.3.4 Comparison between GPU Naive and Shared memory Optimization implementations of Adaptive bilateral filter

6.3.4.1 Experiment

In this experiment, we have varied the size of the filter and observed the time taken by different GPU implementations of adaptive bilateral filter i.e. Naive and Shared memory Optimization on the cameraman image.

6.3.4.2 Results

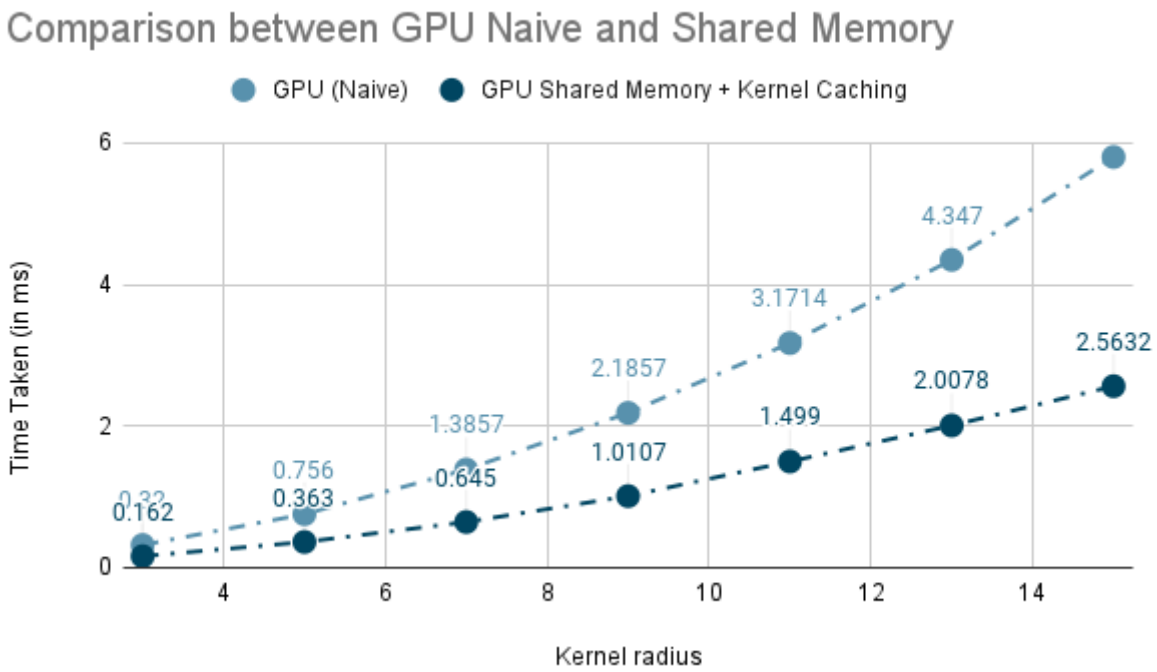


Figure 6.10: Comparison of execution times (in milliseconds (ms)) between GPU Naive and GPU Shared memory optimization of the adaptive bilateral filter with varying filter size (Cameraman image)

Filter Radius	GPU (Naive)	GPU Shared memory + Kernel Caching
3	0.32	0.162
5	0.756	0.363
7	1.3857	0.645
9	2.1857	1.0107
11	3.1714	1.499
13	4.347	2.0078
15	5.7993	2.5632

Table 6.11: Comparison of execution times (in milliseconds (ms)) between GPU Naive and GPU Shared memory optimization + Kernel Caching of the adaptive bilateral filter with varying filter size (Cameraman image)

6.3.4.3 Conclusion

As the results suggest, the time taken by GPU Shared memory optimization + Kernel Caching is less compared to the naive implementation as in the shared memory optimization, we have precomputed both the kernels, also placed the image data in shared memory to reduce the data access time.

6.3.5 Comparison between NL-means GPU-Kernel Optimization in global Vs shared memory

6.3.5.1 Experiment

In this experiment, we have varied the size of the filter and observed the time taken by different GPU implementations of NL-means bilateral filter i.e. kernel in global memory Vs shared memory on cameraman image.

6.3.5.2 Results

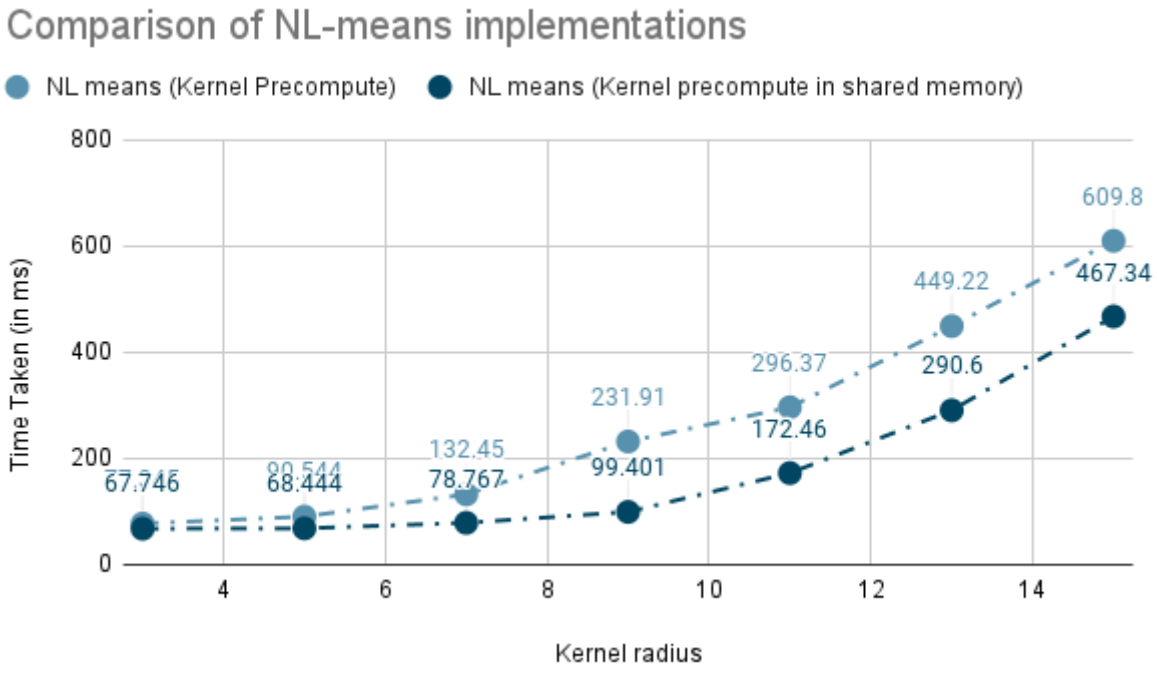


Figure 6.11: Comparison of execution times (in milliseconds (ms)) between different GPU implementations of NL-means bilateral filter i.e. kernel in global memory Vs shared memory with varying kernel size (Cameraman image)

Filter Radius	Kernel (in global memory)	Kernel (in shared memory)
3	77.345	67.746
5	90.544	68.444
7	132.45	78.767
9	231.91	99.401
11	296.37	172.46
13	449.22	290.6
15	609.8	467.34

Table 6.12: Comparison of execution times (in milliseconds (ms)) between different GPU implementations of NL-means bilateral filter i.e. kernel in global memory Vs shared memory with varying kernel size (Cameraman image)

6.3.5.3 Conclusion

As the above results suggest, the time taken when the kernel is placed in shared memory is less compared to when they are placed in global memory as the time taken to access the kernel data decreased since it is placed in shared memory.

6.4 Experiments with Brain image:

6.4.1 Comparison between Sequential, Parallel and GPU Naive

6.4.1.1 Experiment

In this experiment, we have varied the size of the filter and observed the time taken by different implementations of bilateral filter i.e: Sequential, Parallel implementation on CPU, GPU naive implementation on brain image.

6.4.1.2 Results

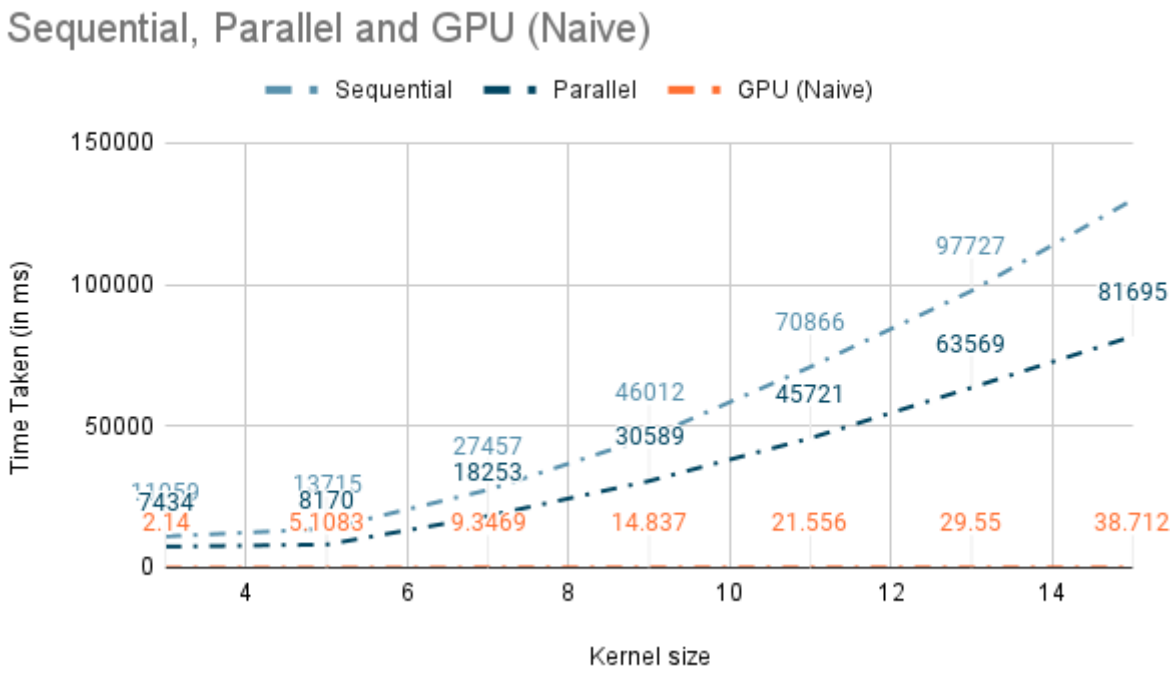


Figure 6.12: Comparison of execution times (in milliseconds (ms)) between Sequential, Parallel and GPU Naive with varying filter size (Brain image)

Filter Radius	Sequential	Parallel	GPU (Naive)
3	11059	7434	2.14
5	13715	8170	5.1083
7	27457	18253	9.3469
9	46012	30589	14.837
11	70866	45721	21.556
13	97727	63569	29.55
15	129900	81695	38.712

Table 6.13: Comparison of execution times (in milliseconds (ms)) between Sequential, Parallel and GPU Naive with varying filter size (Brain image)

6.4.1.3 Conclusion

The results observed are as expected which are explained in the experiments of car image section. We can also observe that the time taken for each implementation is more when compared to car, cameraman images, this is because of the difference in the size of the image. The size of brain image is much larger when compared to the other images, so the time taken is high.

The **speedup** achieved by parallel is $\approx \mathbf{1.59}$, whereas the speedup achieved by GPU Naive implementation is $\approx \mathbf{3355.54}$ when compared to sequential on kernel radius of 15.

6.4.2 Comparison between GPU Naive, Kernel Caching Optimization, Shared memory Optimization, Shared Memory Optimization+Kernel Caching

6.4.2.1 Experiment

In this experiment, we have varied the size of the filter and observed the time taken by different GPU implementations of bilateral filter i.e: Naive, Kernel Caching Optimization, Shared memory Optimization, Shared Memory Optimization+Kernel Caching (Brain image)

6.4.2.2 Results

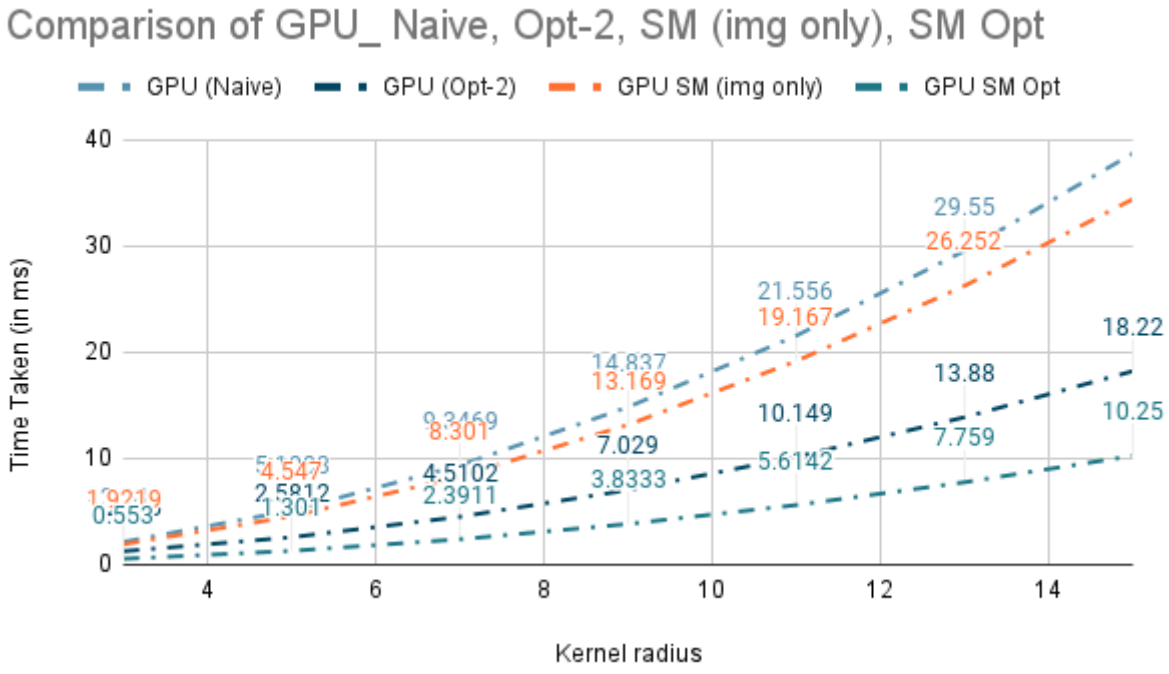


Figure 6.13: Comparison of execution times (in milliseconds (ms)) between GPUs Naive, Kernel Caching Optimization, Shared memory Optimization, Shared Memory Optimization+Kernel Caching (Brain image)

Filter Radius	Naive	Kernel Caching Optimization	Shared memory Optimization	Shared Memory Optimization+Kernel Caching
3	2.14	1.2646	1.9219	0.553
5	5.1083	2.5812	4.547	1.301
7	9.3469	4.5102	8.301	2.3911
9	14.837	7.029	13.169	3.8333
11	21.556	10.149	19.167	5.6142
13	29.55	13.88	26.252	7.759
15	38.712	18.22	34.415	10.25

Table 6.14: Comparison of execution times (in milliseconds (ms)) between GPUs Naive, Kernel Caching Optimization, Shared memory Optimization, Shared Memory Optimization+Kernel Caching (Brain image)

6.4.2.3 Conclusion

The results observed are as expected, which is explained in car image experiments, with Shared Memory Optimization+Kernel Caching being the fastest among all the implementations followed by Kernel Caching Optimization variation, The next fast variation is GPU Shared memory Optimization(img only), then GPU Naive which is the slowest of all.

The **speedup** achieved by Shared memory is $\approx \mathbf{3774.62}$, Optimization-2 is $\approx \mathbf{7129.52}$, Shared memory optimization is $\approx \mathbf{12673}$ on kernel radius of 15 when compared to sequential.

6.4.3 Comparison between Sequential. Parallel and GPU Naive of Adaptive bilateral filter

6.4.3.1 Experiment

In this experiment, we have varied the size of the filter and observed the time taken by different implementations of adaptive bilateral filter i.e. Sequential, Parallel implementation on CPU, GPU naive implementation on brain image.

6.4.3.2 Results

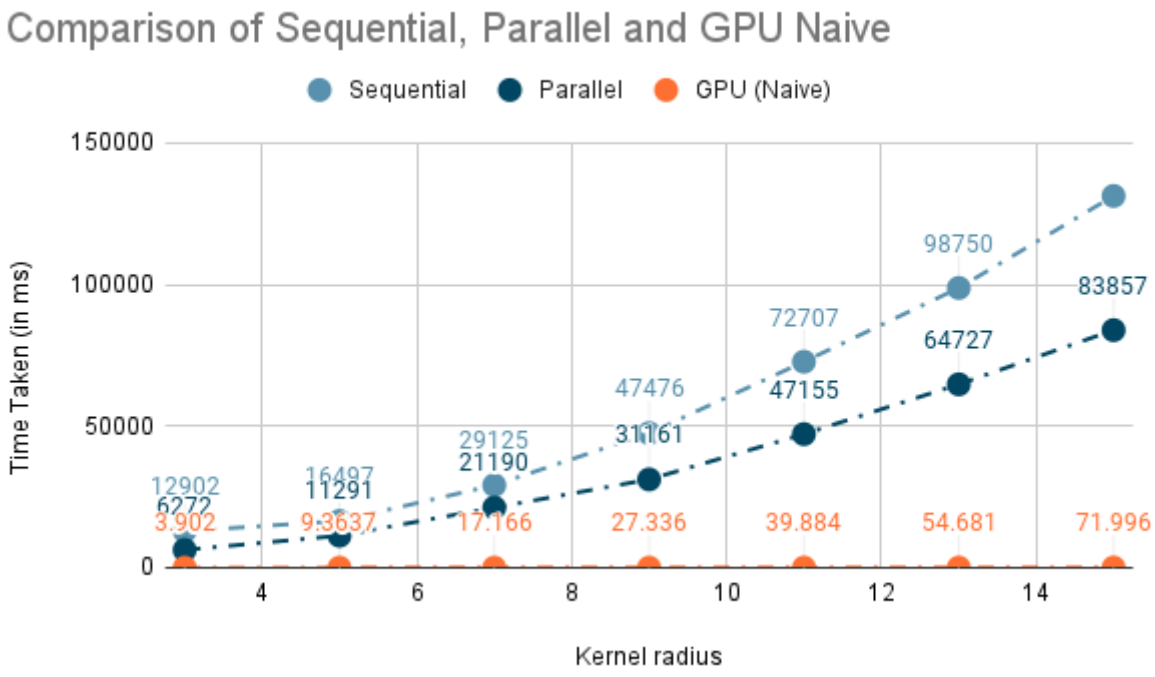


Figure 6.14: Comparison of execution times (in milliseconds (ms)) of adaptive bilateral filter between Sequential, Parallel and GPU Naive with varying filter size (Brain image)

Filter Radius	Sequential	Parallel	GPU (Naive)
3	12902	6272	3.902
5	16497	11291	9.3637
7	29125	21190	17.166
9	47476	31161	27.336
11	72707	47155	39.884
13	98750	64727	54.681
15	131237	83857	71.996

Table 6.15: Comparison of execution times (in milliseconds (ms)) of adaptive bilateral filter between Sequential, Parallel and GPU Naive with varying filter size (Brain image)

6.4.3.3 Conclusion

The results observed are as expected which are explained in the bilateral filter experiments. Moreover, the time taken by each implementation is higher compared to the bilateral filter as in the adaptive bilateral filter, we first calculate sigma based on the variance and then apply it, so it requires more computations than bilateral.

6.4.4 Comparison between GPU Naive and Shared memory Optimization implementations of Adaptive bilateral filter

6.4.4.1 Experiment

In this experiment, we have varied the size of the filter and observed the time taken by different GPU implementations of adaptive bilateral filter i.e. Naive and Shared memory Optimization on the brain image.

6.4.4.2 Results

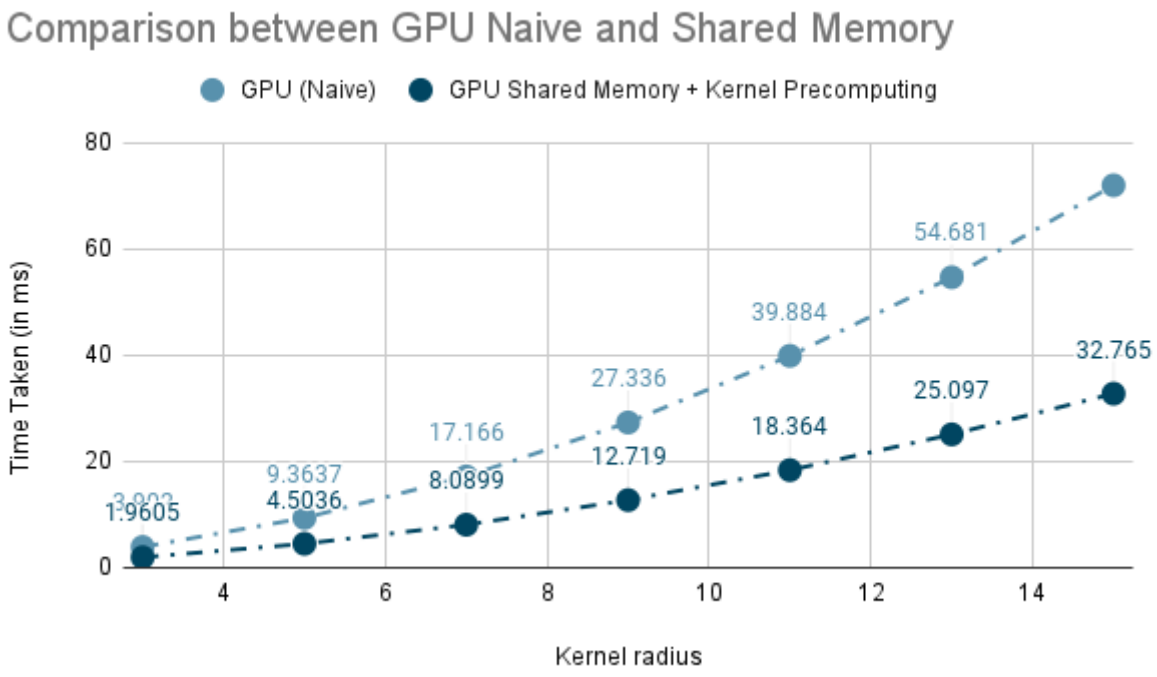


Figure 6.15: Comparison of execution times (in milliseconds (ms)) between GPU Naive and GPU Shared memory optimization of the adaptive bilateral filter with varying filter size (Brain image)

Filter Radius	GPU (Naive)	GPU Shared memory + Kernel Caching
3	3.902	1.9605
5	9.3637	4.5036
7	17.166	8.0899
9	27.336	12.719
11	39.884	18.364
13	54.681	25.097
15	71.996	32.765

Table 6.16: Comparison of execution times (in milliseconds (ms)) between GPU Naive and GPU Shared memory optimization + Kernel Caching of the adaptive bilateral filter with varying filter size (Brain image)

6.4.4.3 Conclusion

As the results suggest, the time taken by GPU Shared memory optimization + Kernel Caching is less compared to the naive implementation as in the shared memory optimization, we have precomputed both the kernels, also placed the image data in shared memory to reduce the data access time.

6.4.5 Comparison between NL-means GPU-Kernel Optimization in global Vs shared memory

6.4.5.1 Experiment

In this experiment, we have varied the size of the filter and observed the time taken by different GPU implementations of NL-means bilateral filter i.e. kernel in global memory Vs shared memory on brain image.

6.4.5.2 Results

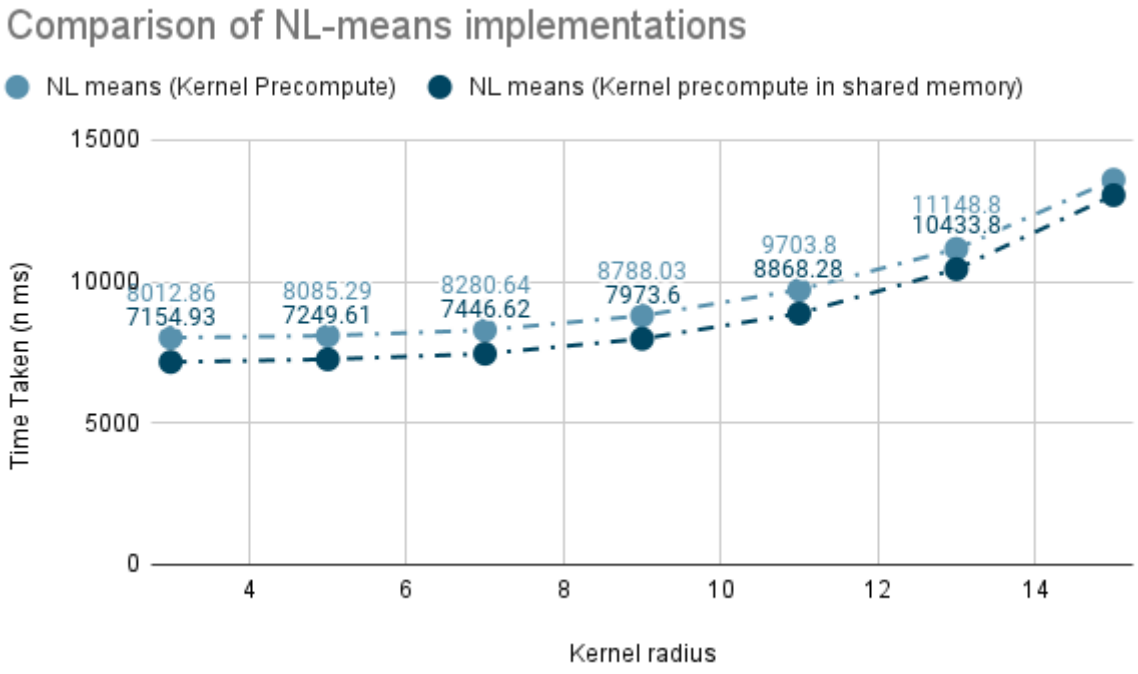


Figure 6.16: Comparison of execution times (in milliseconds (ms)) between different GPU implementations of NL-means bilateral filter i.e. kernel in global memory Vs shared memory with varying kernel size (Brain image)

Filter Radius	Kernel (in global memory)	Kernel (in shared memory)
3	8012.86	7154.93
5	8085.29	7249.61
7	8280.64	7446.62
9	8788.03	7973.6
11	9703.8	8868.28
13	11148.8	10433.8
15	13589	13045.3

Table 6.17: Comparison of execution times (in milliseconds (ms)) between different GPU implementations of NL-means bilateral filter i.e. kernel in global memory Vs shared memory with varying kernel size (Brain image)

6.4.5.3 Conclusion

As the above results suggest, the time taken when the kernel is placed in shared memory is less compared to when they are placed in global memory as the time taken to access the kernel data decreased since it is placed in shared memory.

6.5 Comparison between Bilateral, Adaptive bilateral and NL-means bilateral

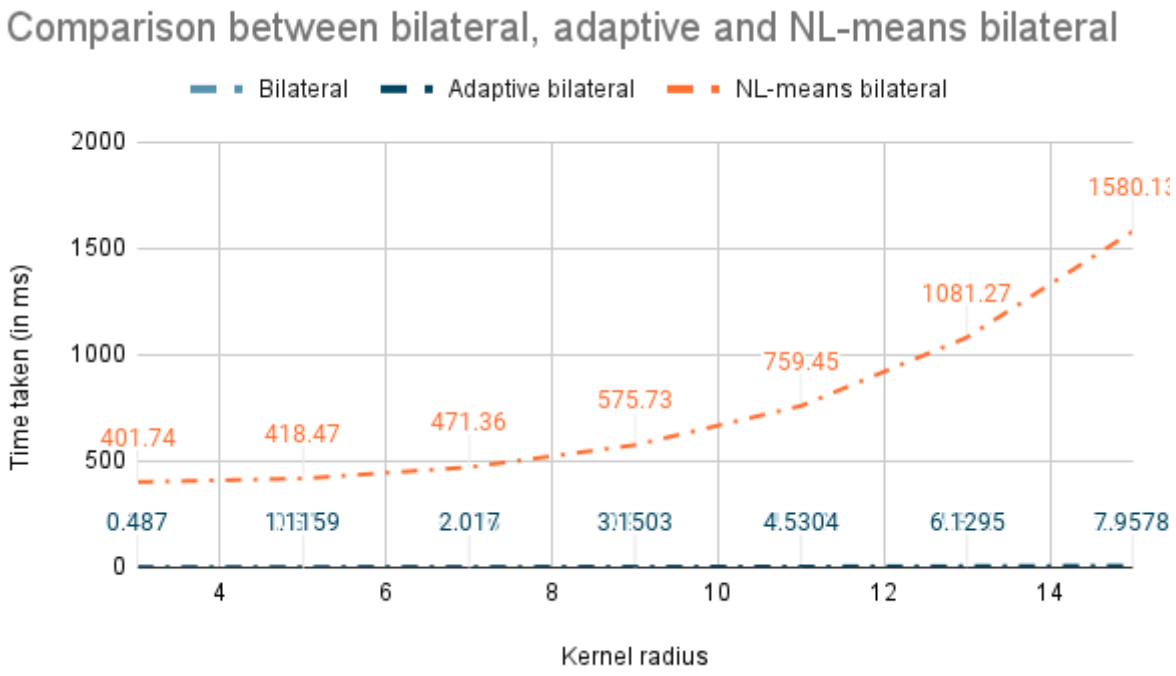


Figure 6.17: Comparison of execution times (in milliseconds (ms)) between Bilateral, Adaptive bilateral and NL-means bilateral filters with varying filter size

Filter Radius	Bilateral	Adaptive Bilateral	NL-means Bilateral
3	0.137	0.487	401.74
5	0.316	1.1159	418.47
7	0.576	2.017	471.36
9	0.917	3.1503	575.73
11	1.349	4.5304	759.45
13	1.855	6.1295	1081.27
15	2.4513	7.9578	1580.13

Table 6.18: Comparison of execution times (in milliseconds (ms)) between Bilateral, Adaptive bilateral and NL-means bilateral filters with varying filter size

6.5.1 Conclusion

As the results suggest, the time taken by bilateral filter is less compared to adaptive filter. NL-means bilateral filter has the highest execution time among all the others. This is because, in adaptive bilateral filter, there are extra computations required to calculate new sigma, and in NL-means bilateral filter, we iterate over all the pixels of the image to find a similar patch. So, as the number of computations required increased, the execution time also increased. As the size of the filter increases, the computation srequired also increases, hence the execution time of every algorithm also increased.

6.6 Comparitive Visual Quality Analysis

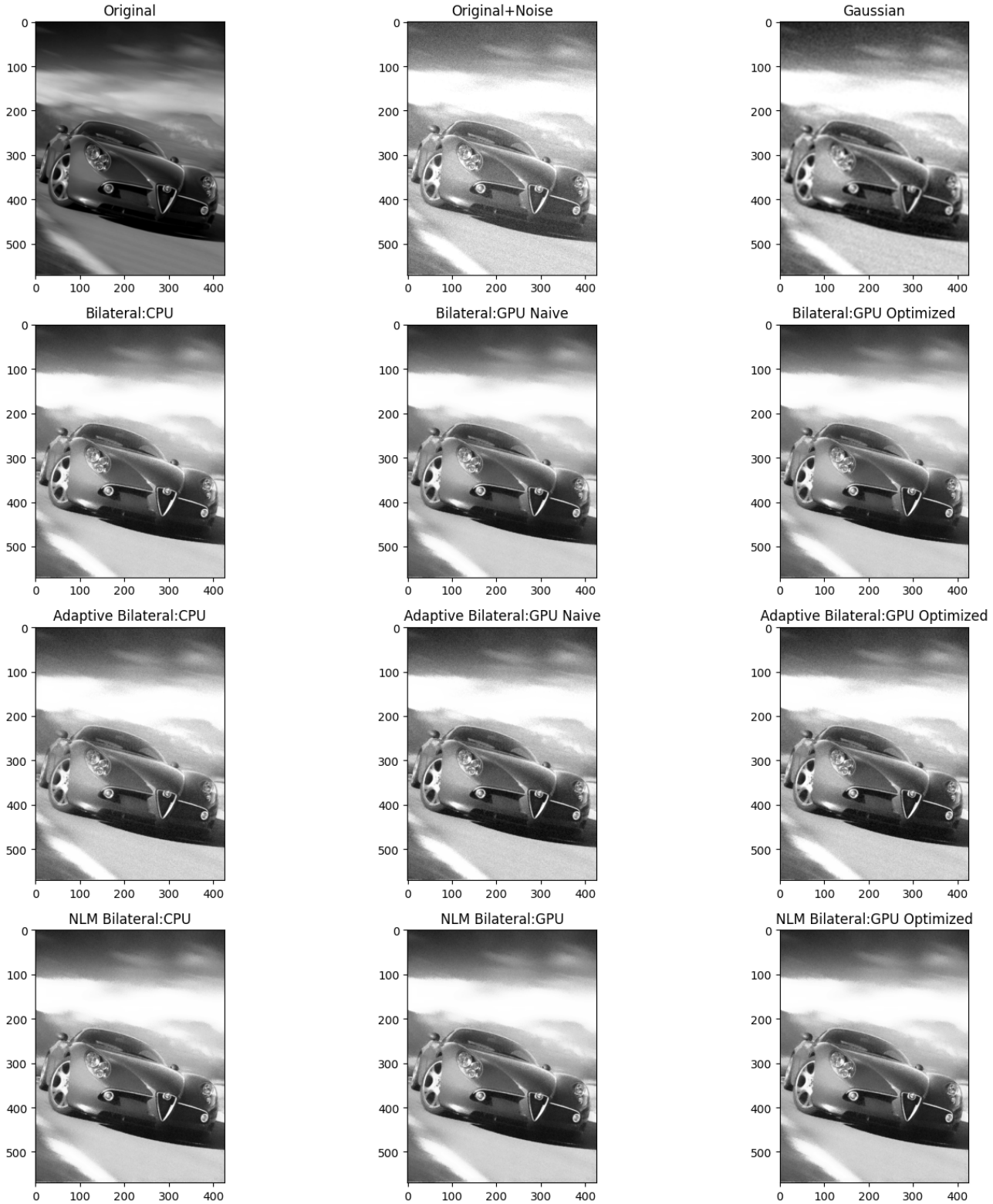


Figure 6.18: Comparison of Effect on Visual Quality of various Filters on the Car Image

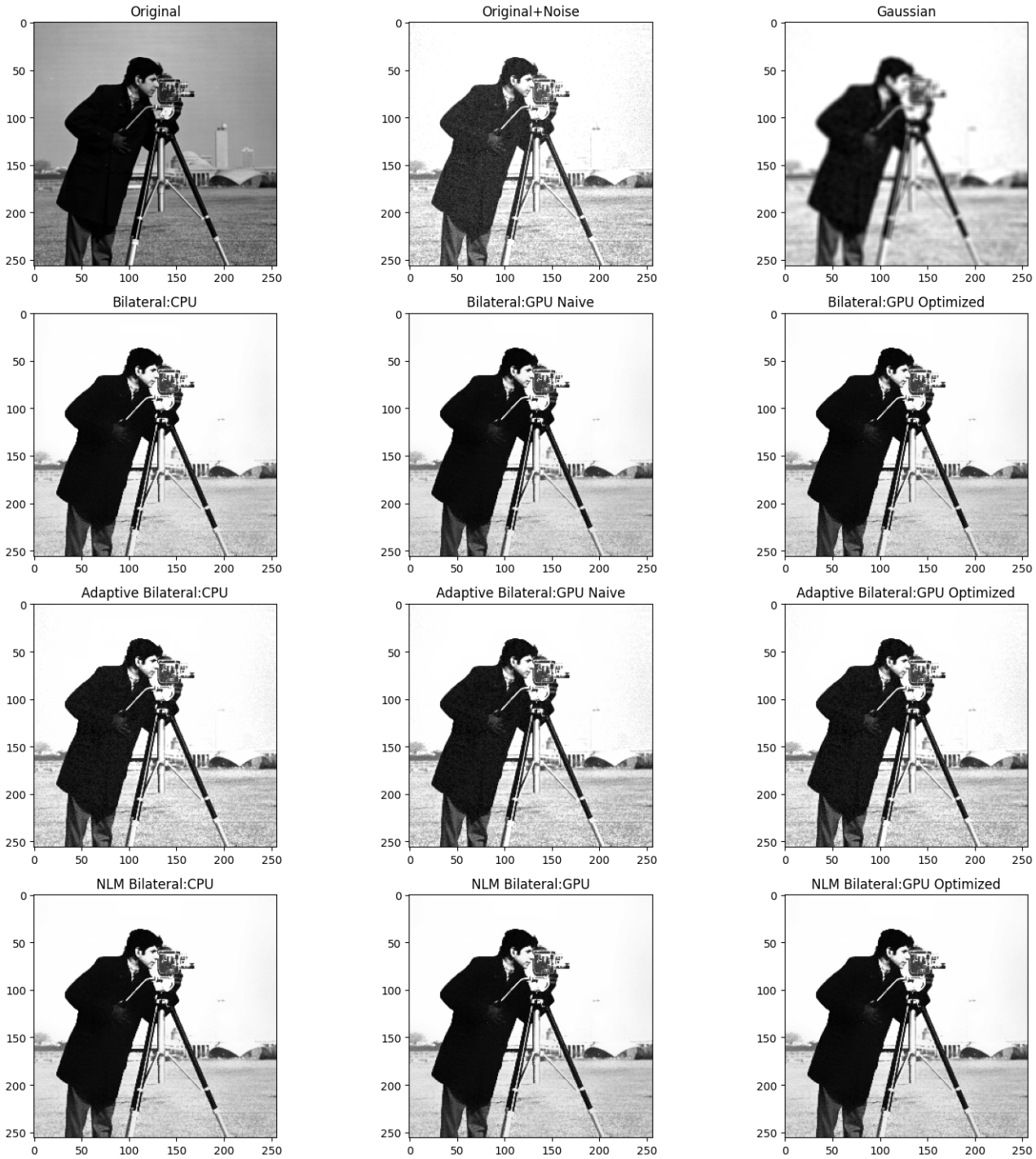


Figure 6.19: Comparison of Effect on Visual Quality of various Filters on the Cameraman Image

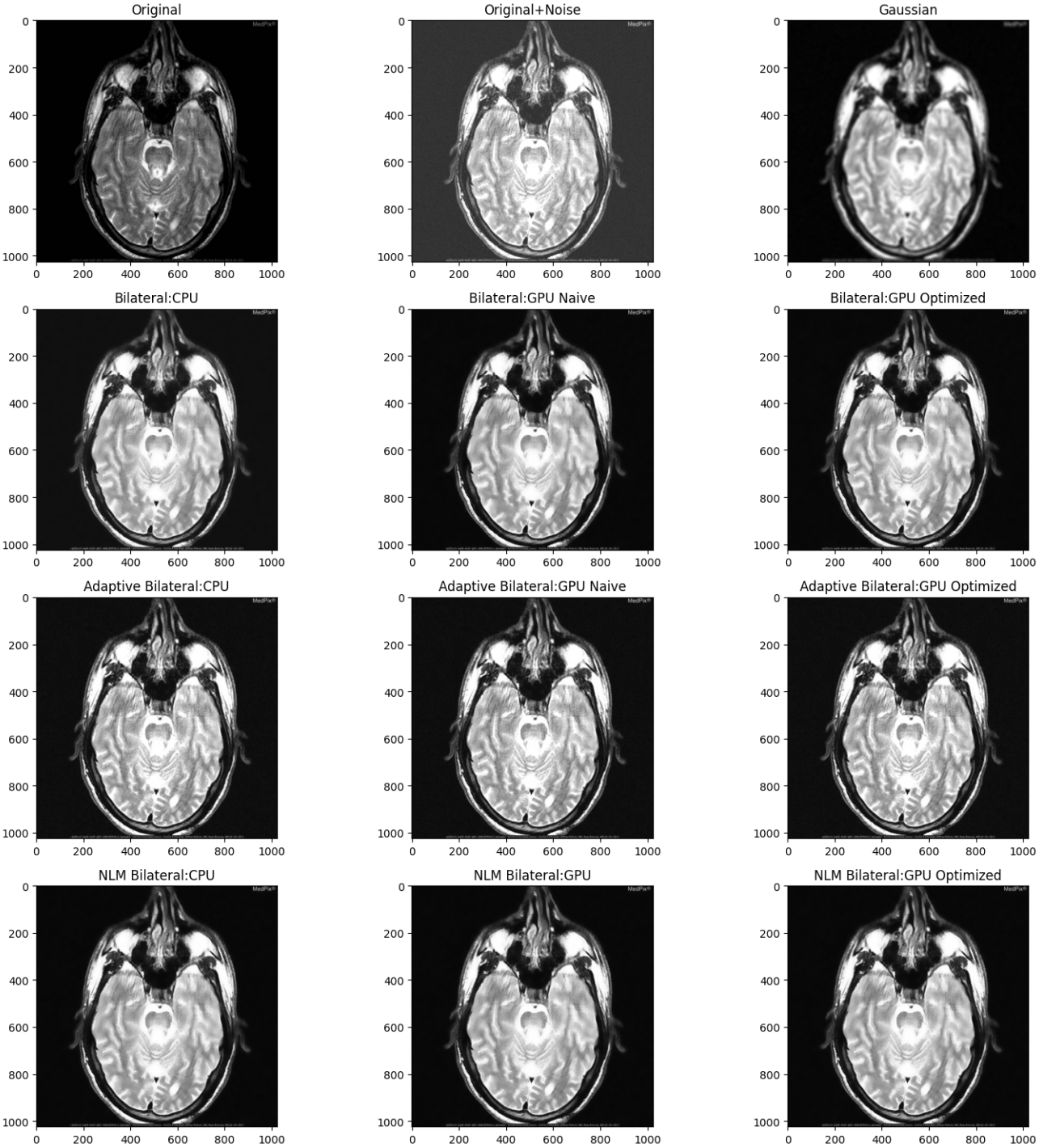


Figure 6.20: Comparison of Effect on Visual Quality of various Filters on the brain Image

6.6.1 Conclusion

Visual quality is a more important metric for evaluating edge preservation than PSNR values,Visual quality refers to the subjective perception of image quality by a human observer, and takes into account factors such as sharpness, clarity, and edge preservation.

In the context of filters that we have used, visual quality is particularly important because the goal of the filter is to preserve edges while still reducing noise and smoothing the image. A filter that achieves a high PSNR value but does not preserve edges well may not be as visually pleasing as a filter that achieves a lower PSNR value but preserves edges more effectively.

As apparent from the above results from our GPU implementations the speedups achieved are in the magnitude of 1000s than the CPU implementation for all the images while keeping the effectiveness of the underlying algorithms intact in terms of visual quality.

In terms of the algorithms themselves the quality of bilateral results is much better than Gaussian, while there's a subtle difference acheived by bilateral and nl means bilateral filters although the nl means filter has been shown to give comparitively much better results like presented in [18],but due to limitations of the kind and intensity of noise we used in our work the difference is subtle.

Chapter 7

Conclusions and future works

The preceding section of the report presents experimental results indicating that the implementation of the algorithm using GPUs leads to a significant reduction in processing time. The utilization of parallel processing leads to a speedup ranging from 1.367 to 1.681 compared to the sequential implementation. However, the employment of GPU Naive implementations yields a much greater reduction in processing time, resulting in speedups ranging from 2826.94 to 3696.75 compared to the sequential implementation which is much more than the 200x speedup achieved in [3], and speedup of 1000x achieved in [6].

For instance, the sequential implementation of the algorithm required 33001 ms to process an image of dimensions 570 x 426. However, the implementation using GPU Naive techniques achieved a reduction in processing time to 9.2744 ms for the same image dimensions. This reduction in processing time highlights the efficacy of GPU-based implementations in reducing the computational burden and enhancing the overall performance of the algorithm.
Advanced GPU techniques, such as shared memory, have been utilized to further optimize the processing time of the GPU Naive implementation. The shared memory technique reduces memory latency and improves the performance of the algorithm. Additionally, experiments have been conducted using precomputed spatial and intensity kernels to enhance the processing speed.

The efficient implementation utilizing these advanced GPU techniques has achieved a speedup of approximately 3.78 compared to the Naive implementation. This improvement in processing speed highlights the effectiveness of utilizing advanced GPU techniques in optimizing the computational performance of the algorithm. This speedup is more than the 3x the speedup of 1.2 achieved in [19] for similar-sized images and kernel size.
We have also experimented with two variations of bilateral filter namely Adaptive bilateral filter and NL-means bilateral filter on GPU and presented their results. The GPU implementations of those algorithms also outperformed their sequential and parallel (CPU-based) implementations

by a large factor.

The GPU Occupancy(a metric that measures the percentage of time that a GPU is executing a kernel achieved by our experiments was very high (approximately 94 percent) suggesting that our implementation utilized the GPU very efficiently and that a balanced workload is being processed.

Among all the variations, the bilateral filter takes less time than the adaptive bilateral filter which is faster than NL-means bilateral filter. It is because, NL-means bilateral requires more computations than the adaptive bilateral filter, which has more computations than the bilateral filter.

The adaptive bilateral filter incorporates an adaptive range parameter, which adjusts the filter weights based on the local image statistics. This allows the filter to adapt to the local variations in the image and achieve better results than the standard bilateral filter. However, the additional computations required to calculate the adaptive range parameter result in longer computation times than the standard bilateral filter.

The NL-means bilateral filter further extends the bilateral filter by incorporating a non-local means approach to image denoising. This involves computing the similarity between patches of pixels across the entire image, resulting in a more accurate denoising process. However, the increased complexity of this approach leads to even longer computation times than the adaptive bilateral filter.

Overall, our results demonstrate the effectiveness of GPU-based implementations of these image-processing algorithms, and highlight the trade-offs between computational complexity and denoising accuracy in different variations of the bilateral filter.

7.1 Future Works

7.1.1 Async Memory Transfers

During our experiments, we observed that a significant amount of time is being consumed by the memory allocation and data transfer steps in our GPU implementations. These steps are crucial for efficient GPU processing, as they involve transferring data from the host to the device and allocating memory on the device for processing. Therefore, efforts can be made towards exploring ways to optimize these steps and reduce the time taken for memory allocation and data transfer.

One potential approach to reducing the time taken for memory allocation and data transfer is to use asynchronous memory transfers. This involves overlapping the memory transfer and computation steps so that the GPU can begin processing data while it is still being transferred

from the host to the device. This can significantly reduce the overall processing time, as it allows the GPU to begin processing data as soon as possible.

7.1.2 Data compression

Data compression can be used to optimize the transfer of data to and from the GPU in bilateral filters on CUDA GPU. By compressing the data before transferring it to the GPU, it is possible to reduce the amount of data that needs to be transferred, which can improve memory bandwidth and reduce the overall processing time. One approach to data compression is to use lossless compression techniques, such as run-length encoding or Huffman coding. These techniques can be used to compress the data without losing any information, which is important for applications such as image processing where data fidelity is critical. Another approach to data compression is to use lossy compression techniques, such as JPEG or MPEG compression. These techniques can be used to compress the data more aggressively but at the cost of some loss of information. In the context of bilateral filters on CUDA GPU, lossy compression may be acceptable if the loss of information does not significantly impact the quality of the output.

7.1.3 Dynamic Parallelism

Dynamic parallelism is another technique that can be used to optimize the processing of bilateral filters on CUDA GPU. By using nested parallelism, it is possible to create new threads and launch new kernels dynamically, which can improve the efficiency of GPU processing. One approach to dynamic parallelism is to use a recursive implementation of the bilateral filter, where each thread launches a new kernel to process a smaller subset of the data. This approach can be used to distribute the workload across multiple threads and GPUs, which can significantly reduce the processing time and improve the overall efficiency of the system. Another approach to dynamic parallelism is to use a task-based parallelism model, where tasks are dynamically created and assigned to threads as needed. This approach can be used to optimize the use of GPU resources and improve the overall efficiency of the system.

7.1.4 Realtime applications

Real-time applications, such as video processing or real-time image enhancement, require high-performance image filtering algorithms that can process large amounts of data in real time. Therefore, it is important to investigate the potential of the implemented filters for use in such applications. This could involve testing the filters on live video streams or other real-time image sources and measuring their performance in terms of speed and accuracy.

7.1.5 Signal Processing

In addition to image processing applications, the implemented filters also have the potential for use in other domains beyond image processing. For example, the filters could potentially be used in signal processing applications or in other areas where noise reduction or smoothing is required. To explore the potential for using the implemented filters in other domains, it may be necessary to adapt the filters to work with different types of data, such as audio signals or time-series data. This could involve modifying the filters to work with different data formats or developing new algorithms that are specifically designed for these types of data.

Appendix A

Contributions

Our project is done in a group of two this chapter highlights my individual contribution along with the associated timelines for each task. This provides a comprehensive overview of individual involvement in the project, emphasizing the technical aspects of our work and the milestones achieved.

A.1 Individual Contribution:Kushagra Indurkhya

My individual contribution can be broadly classified into the following sections:

A.1.1 Implementations

The following implementations were implemented by me end to end:

- **Bilateral Filter**
  - **CPU Implementations(Sequential and Parallel)**: I implemented and tested the Bilateral Algorithm on CPU using sequential and parallel paradigms where the image input was taken using OpenCV C++ API and the calculation was done in OpenCV MAT format as described in 5.1.1
  - **GPU Naive**: I Implemented a naive GPU Implementation on CUDA for the bilateral algorithm keeping the source image in global memory, each thread getting one pixel as described in 5.2.1
  - **Spatial Kernel Cache in Global**: Ideated and Cached the precomputed Gaussian spatial kernel as described in global memory sec 5.1.2.4
  - **Intensity Gaussian PreComputed in Global**: Ideated and implemented the intensity Gaussian value caching in global memory as described in sec 5.1.2.5

    - **Source image tiled in shared memory**: Ideated and Implemented the optimization of loading tiles of the source image in shared memory reducing the memory access time by a large factor as described in sec 5.1.2.5

- **Adaptive and Non-Local Means Bilateral Filter GPU Implementations**: Taking inspiration from the Optimized GPU Implementation of the bilateral filter directly wrote the GPU implementation of the Adaptive and Non-Local Means Bilateral Filter, experimented extensively with the tweaks for the algorithms directly in the GPU implementation, As described in sec 5.2.2 and sec 5.3.2 respectively ,the CPU implementations were modelled later on the same.

A.1.2 Algorithm Designs

Worked extensively on the design of the various algorithms we were using, choosing the trade-offs carefully, the following variations/tweaks were ideated by me

- **Adaptive Bilateral Filter**: The following tweaks and variations were conceptualized and incorporated by me after careful analysis as described in 4.2.2
    - Variance as the parameter for adaptivity of the weights of the range.
    - First applying a basic bilateral filter to a neighborhood before calculating its variance, instead of directly calculating the variance of the intensities
    - Regularization of variance for low variance regions
- **Non-Local Means Bilateral Filter (NLMBF)**: The following tweaks and variations were conceptualized and incorporated by me after careful analysis as described in 4.3.3
    - Keep the whole image as the search window
    - Use SSD as metric for patch similarity index
    - Use intensity difference Gaussian in weight

A.1.3 Documentation

Chapters 2,4,5 of this report were written end to end by me :

- **Chapter 2 (Image Denoising)**: In this chapter, I defined what noise is and presented the different types of noise that can affect images. I then went on to explain the concept of image denoising and what features a good denoising system should have. This chapter also delved into the traditional denoising methods, such as the Box Filter, Gaussian Filter, and Median Filter.

- **Chapter 4(Algorithms and Related Work)**: In this chapter, I covered three specific algorithms that are commonly used for image denoising, namely the Bilateral Filter and Adaptive Bilateral Filter, as well as the nonlocal, which means bilateral algorithm. I meticulously explained the concept behind each algorithm, its design, and the variations and tweaks that can be made to improve its effectiveness. I also presented related works that have been done in this area to give readers a comprehensive view of the state-of-the-art algorithms.
- **Chapter 5(Methodology)**: I discussed the methodology we used for evaluating the performance of the Bilateral Filter and Adaptive Bilateral Filter on both CPU and GPU. This chapter goes into the details of the implementation and design levels, explaining the optimization techniques we utilized to enhance the performance of these algorithms.

A.1.4 Prerequisites and setup

- **Cuda Basics and Matrix Multiplication**: Read the extensive documentation and GPU architecture documentation of NVIDIA CUDA.On the suggestion of our mentor to get a preliminary idea of GPU programming and NVIDIA Cuda APIs, as a prerequisite, I implemented matrix multiplication on Google collab using CUDA along with its shared memory implementation.
- **Implemented A Demoable Testing Framework**: Once the algorithm was identified and finalized I started working on the implementation details starting with a Python script running on Google Colab which downloads the image, adds Gaussian noise to it, runs the appropriate filters(C++/Cuda) and displays comparative results in a grid.

A.2 Summary

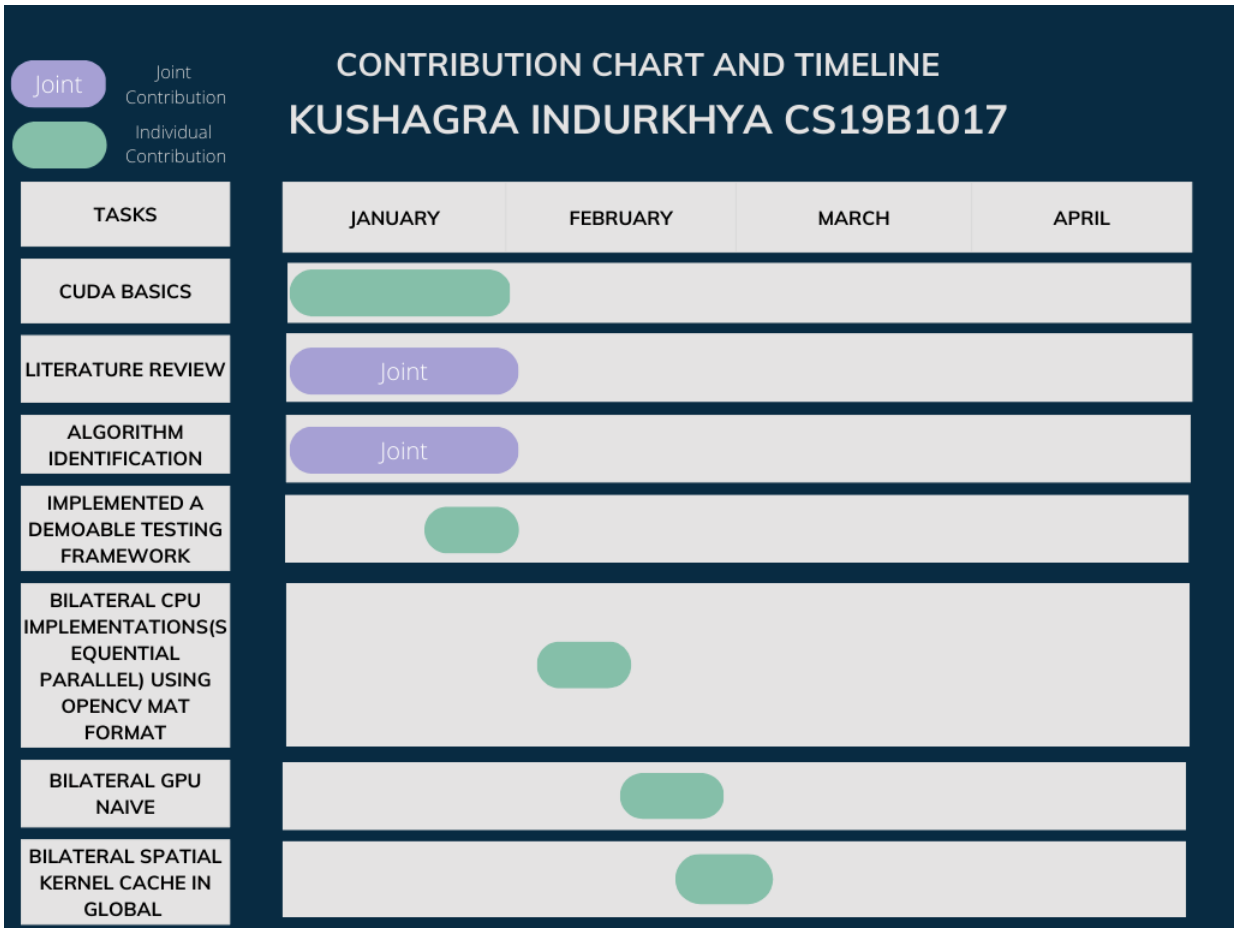


Figure A.1: A brief summary of my contribution:CS19B1017 (a)

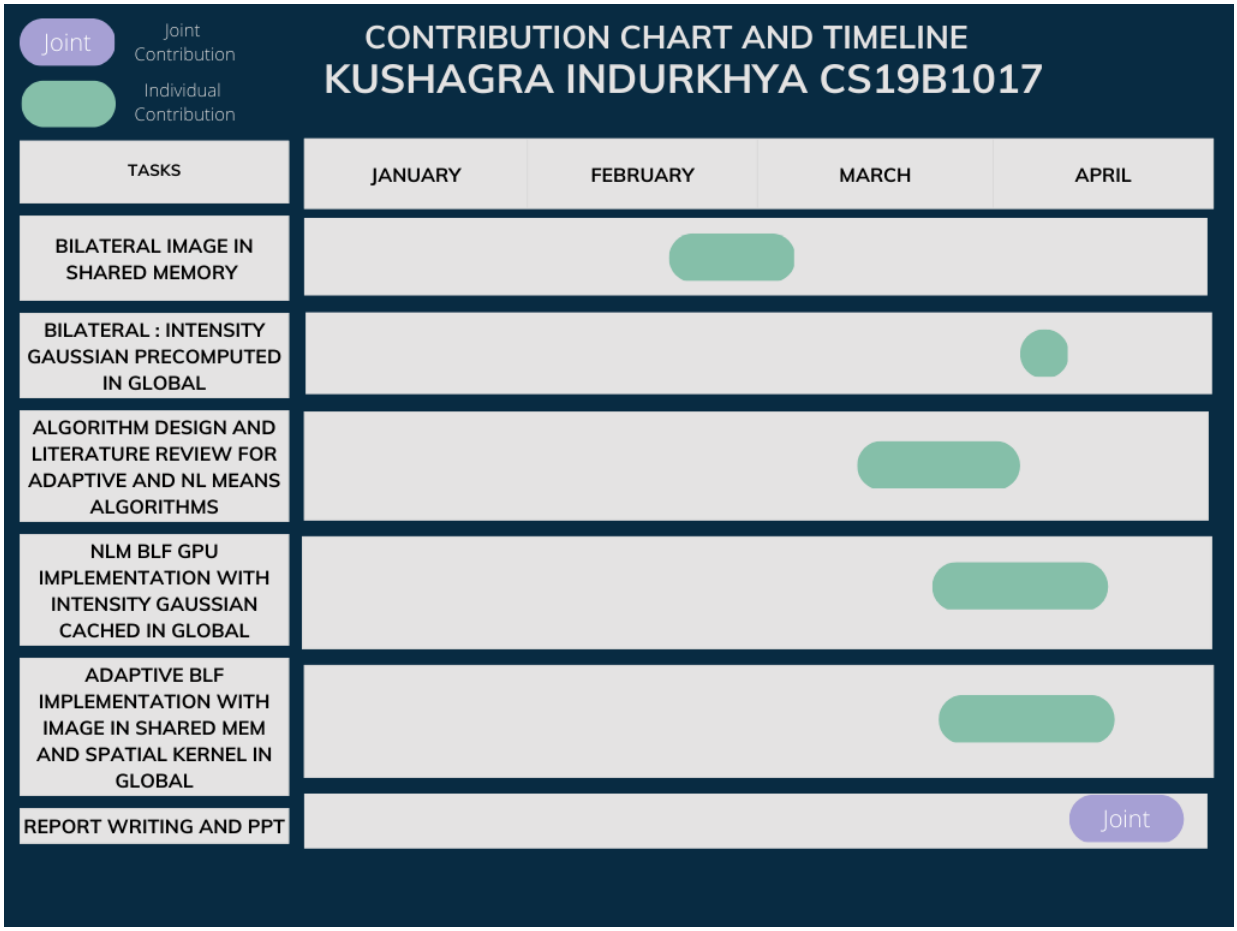


Figure A.2: A brief summary of my contribution:CS19B1017 (b)

A.3 Individual Contribution: Yashwanth Vallabhu

My individual contribution can be broadly classified in the following sections:

A.3.1 Implementations

- **Bilateral Filter:** I have worked on the naive implementation of the bilateral filter, which as suggested by our mentor, I have implemented a procedure that reads the image and stores its pixel values in a .txt file, and the GPU algorithm processes it. But due to some complications, we have decided to shift to the implementation which uses the OpenCV library to read the image, which was implemented by my teammate Kushagra. Also, I have also worked on the precomputation of both spatial and intensity kernels as mentioned in sections 5.1.2.4 and 5.1.2.5. I have implemented the following experiments with the precomputed kernels:
  * Stored them in shared memory.
  * Stored them in constant memory.
  * One in shared and the other in constant memory.
- **Adaptive Bilateral Filter:** I have worked on Sequential and Parallel CPU implementations of Adaptive bilateral filters as mentioned in detail in sections 5.2.1. I have implemented the caching of kernels similar to the above as mentioned in detail in section 5.2.2.

- **NL-means Bilateral Filter:** I have worked on Sequential and Parallel CPU implementations of NL-means bilateral filters as mentioned in detail in sections 5.3.1. I have also implemented the caching of kernels into the shared memory as mentioned in detail in section 5.3.2.

A.3.2 CUDA Basics and Matrix Multiplication

As suggested by our mentor, before starting GPU implementation of image processing algorithms, I first learned about GPU programming, when through CUDA (programming model for GPU). Then as part of the demonstration implemented a sequential, GPU-naive version of matrix multiplication. After that used one of the advanced GPU techniques such as shared memory to further bring down the execution time. After getting satisfactory results, proceeded to image processing algorithms.

A.3.3 Documentation

Chapters: 3,6 of this report were solely written by me :

- **Chapter 3 (Leveraging the Power of GPU):** In this chapter, I have covered why parallelization is needed, various types of it, and some advantages and challenges of parallelization. I have also explained how it works in CPU and described why GPU is important for parallel processing and also introduced CUDA, which is a programming model that allows programmers to harness the computational power of the GPUs. I have in detail described its architecture, its thread, and memory hierarchies.

- **Chapter 6 (Results and Experiments):** This is one of the most important chapters as it depicts the performance gain achieved by us by using GPUs to parallelize the algorithms. In this chapter, I have covered mentioned the hardware setup that was used by us to conduct the experiments and also mentioned what noise we used on the images, and what metrics we used to compare the results of different variations. I have used three images and conducted five experiments on each of them to compare the execution times of various algorithms. While collecting data, I ran the experiment multiple times and took the average of all the execution times I observed so as to make sure the results depicted are correct. For every experiment, I have plotted a graph, shown a table depicting the experiment results, and explained why the results are the way they are.

A.4 Summary

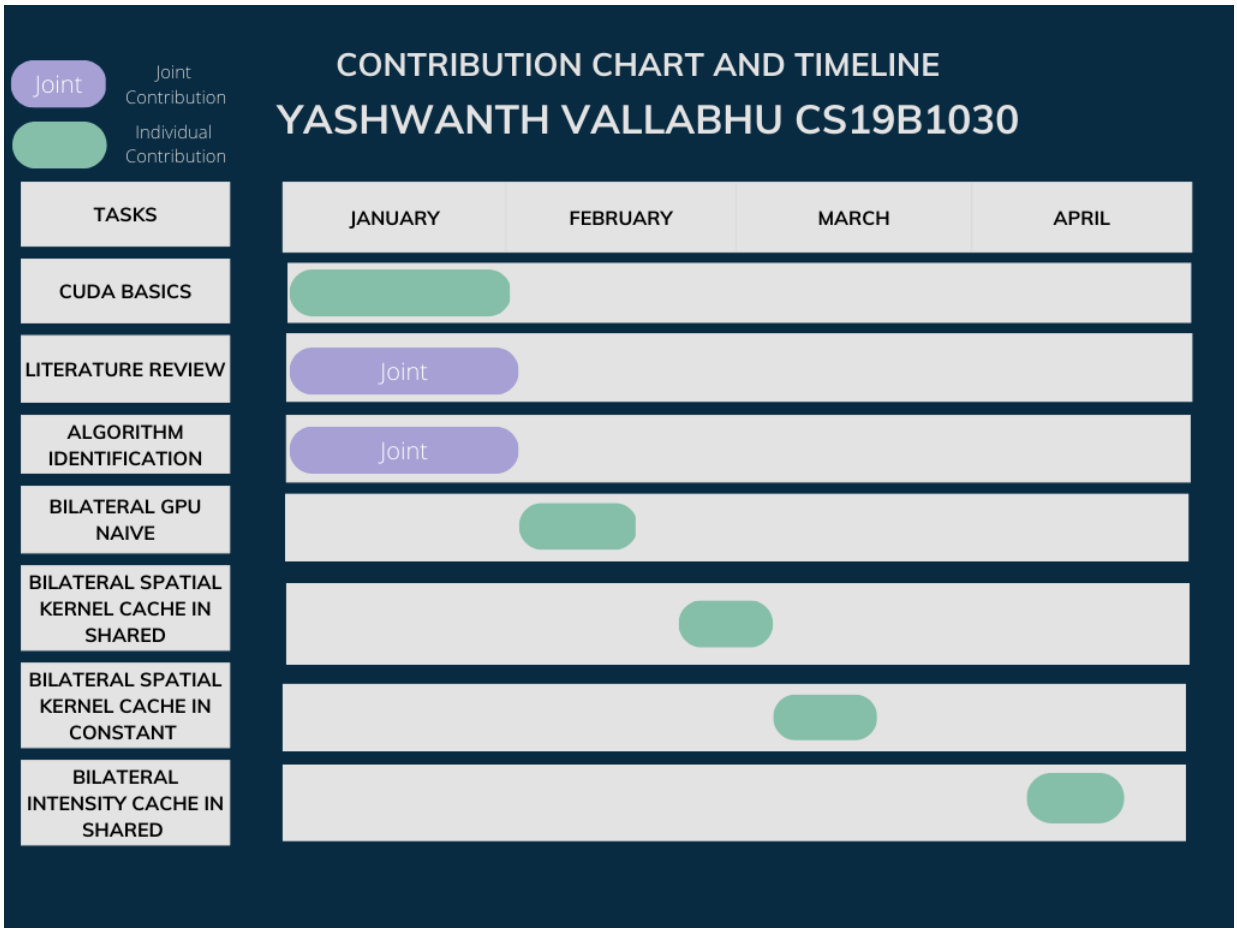


Figure A.3: A brief summary of my contribution CS19B1030 (a)

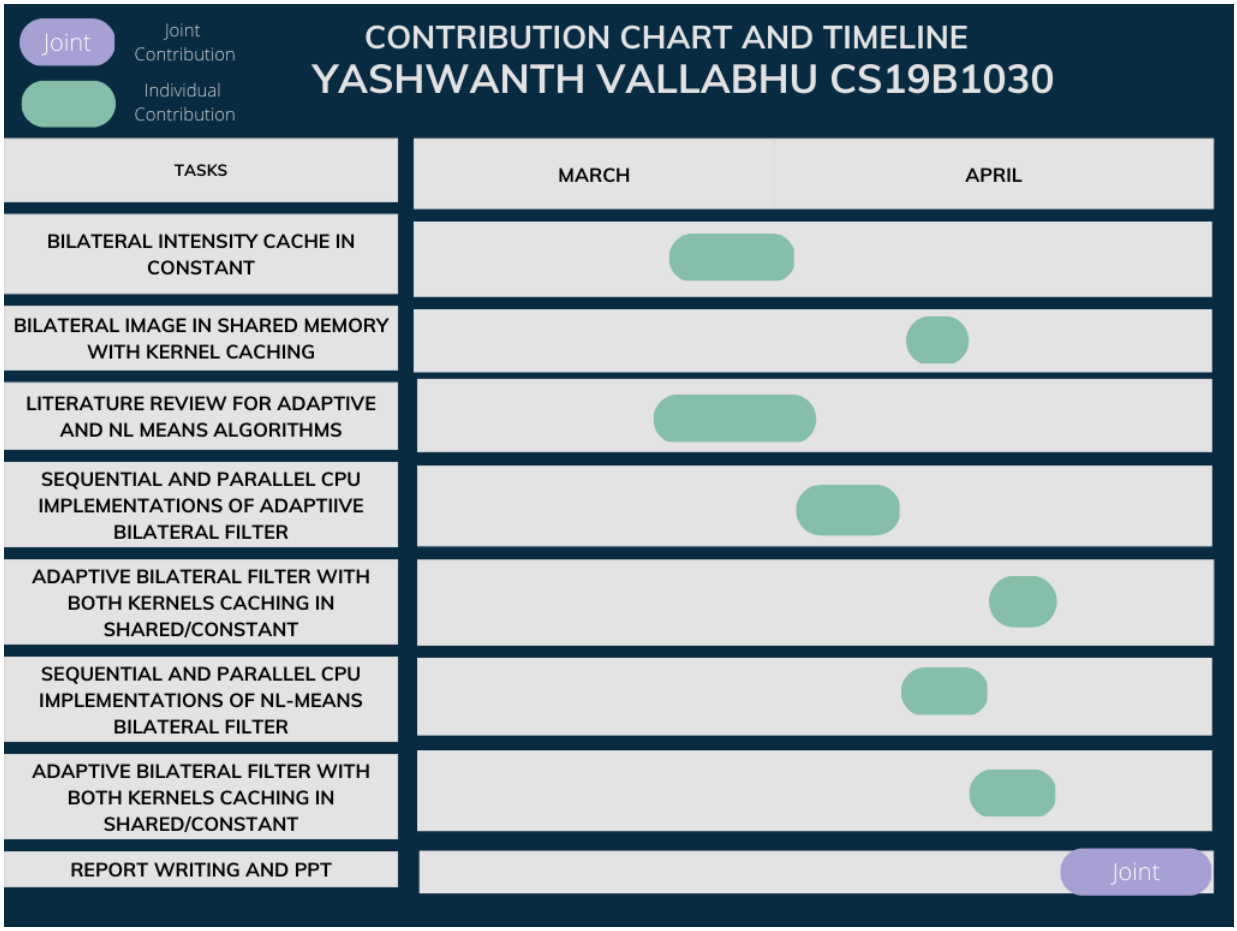


Figure A.4: A brief summary of my contribution CS19B1030 (b)

A.5 Join ContrSibution

The parts of the project done jointly were:

- **Algorithm Identification**: At the beginning of the project we set out to choose the algorithms which have applications in real-time image processing and can utilize the power of GPU by parallelism, Going through various papers and after having insightful discussions with our mentor we chose bilateral filter as the base algorithm for our project
- **Literature Review**: After the algorithm was identified we went through various papers and existing works related to the bilateral filter and came across the adaptive bilateral filter and nonlocal means bilateral filter which we incorporated in our project as well.
- **Work Documentation**: Chapters 1 and 7 of this report were written jointly as they emphasize the project as a whole.
- **Presentation**: The presentation which is one of the deliverables for this project was worked on jointly by both members.

Bibliography

[1] Dao Nam Anh. Local adaptive bilateral filter with variation for deblurring. 2014.

[2] Shylaja S S Kunal N. Chaudhury Apurba Das, Pravin Nair. A concise review of fast bilateral filtering. *Fourth International Conference on Image Information Processing (ICIIP)*.

[3] E. Bethel. Exploration of optimization options for increasing performance of a gpu implementation of a three-dimensional bilateral filter. Technical report, Lawrence Berkeley National Laboratory, 2012.

[4] A. Buades, B. Coll, and J.-M. Morel. A non-local algorithm for image denoising. In *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05)*, volume 2, pages 60–65 vol. 2, 2005.

[5] Bo-Hao Chen, Yi-Syuan Tseng, and Jia-Li Yin. Gaussian-adaptive bilateral filter. *IEEE Signal Processing Letters*, 27:1670–1674, 2020.

[6] A A Deshmukh and Sanjay L. Badjate. "implementation of bilateral filtering on cuda". 2015.

[7] Sudipto Dolui, Iván C. Salgado Patarroyo, and Oleg V. Michailovich. Generalized non-local means filtering for image denoising. In Karen O. Egiazarian, Sos S. Agaian, and Atanas P. Gotchev, editors, *Image Processing: Algorithms and Systems XII*. SPIE, feb 25 2014.

[8] Kuidong Huang, Dinghua Zhang, and Kai Wang. Non-local means denoising algorithm accelerated by GPU. In Faxiong Zhang and Faxiong Zhang, editors, *MIPPR 2009: Medical Imaging, Parallel Processing of Images, and Optimization Techniques*, volume 7497, page 749711. International Society for Optics and Photonics, SPIE, 2009.

[9] Fredo Durand Jiawen Chen, Sylvain Paris. Real-time edge-aware image processing with the bilateral grid.

[10] Jin Ming. An adaptive bilateral filtering method for image processing. *Optoelectronic Engineering*, 2004.

[11] Sylvain Paris, Pierre Kornprobst, Jack Tumblin, and Frédéric Durand. Bilateral filtering: Theory and applications. *Foundations and Trends in Computer Graphics and Vision*, 4(1):1–73, 2006.

[12] Lin Sun, Oscar C. Au, Ruobing Zou, Wei Dai, Xing Wen, Sijin Li, and Jiali Li. Adaptive bilateral filter considering local characteristics. *2011 Sixth International Conference on Image and Graphics*, pages 187–192, 2011.

[13] Carlo Tomasi and Roberto Manduchi. Bilateral filtering for gray and color images. *International Conference on Computer Vision (ICCV)*, pages 839–846, 1998.

[14] Gaihua Wang, Yang Liu, Wei Xiong, and Yan Li. An improved non-local means filter for color image denoising. *Optik*, 173:157–173, 11 2018.

[15] Wen-Qiang Feng, Shu-Min Li, and Ke-Long Zheng. A non-local bilateral filter for image denoising. In *The 2010 International Conference on Apperceiving Computing and Intelligence Analysis Proceeding*. IEEE, 12 2010.

[16] Beshiba Wilson and Dr. Julia Punitha Malar Dhas. A survey of non-local means based filters for image denoising. *International journal of engineering research and technology*, 2, 2013.

[17] Alexander Wong. Adaptive bilateral filtering of image signals using local phase characteristics. *Signal Processing*, 88(6):1615–1619, 2008.

[18] Wenyuan Xu and Klaus Mueller. Evaluating popular non-linear image processing filters for their use in regularized iterative ct. In *Conference Record IEEE Medical Imaging Conference (MIC)*, pages 1352–1355. IEEE, 2010.

[19] Ziyi Zheng, Wei Xu, and Klaus Mueller. Performance tuning for cuda-accelerated neighborhood denoising filters. 01 2011.

[20] Shujin Zhu, Yuehua Li, and Yuanjiang Li. Two-stage non-local means filtering with adaptive smoothing parameter. *Optik*, 125(23):7040–7044, 2014.