School of Computation, Information and Technology - Informatics

Technical University of Munich

Master's Thesis in Computational Science and Engineering

Statistical 3D Denoising on GPUs for Industrial CT

Yu Huang

School of Computation, Information and Technology - Informatics

Technical University of Munich

Master's Thesis in Computational Science and Engineering

Statistical 3D Denoising on GPUs for Industrial CT

Statistische 3D Entrauschung auf GPUs für industrielle CT

Author:	Yu Huang
Supervisor:	Prof. Dr. Michael Bader
1st advisor:	Dr. Alex Sawatzky
2nd advisor:	M.Sc. Mario Wille
Date:	August 23rd, 2023

I confirm that this Master's Thesis is my own work and I have documented all sources and material used.

Munich, August 23rd, 2023 Yu Huang

Acknowledgements

Throughout the writing of this thesis, I have received a great deal of support and assistance. First of all, I am grateful to Waygate Technologies and my manager Andreas Fischer for providing me the opportunity to work on this interesting project. Importantly, I would like to thank my advisor Dr. Alex Sawatzky at the company for his guidance, invaluable support, and patience throughout the research and implementation process.

I would also like to express my sincere thanks to my advisor, M.Sc. Mario Wille, for his valuable feedback and suggestions at all the various stages. A big thank you goes to Prof. Dr. Michael Bader for supporting me in completing my Master's thesis in an external company.

Finally, I would like to express my heartfelt gratitude to my family and friends for their love and encouragement throughout my masters. Vielen Dank!

Abstract

Industrial Computed Tomography (CT) scans increasingly demand for fast throughput time. It can be achieved by either reducing the number of projections or shortening the exposure time per projection. Both methods lead to noise degradation, resulting in insufficient reconstruction quality with classical Filtered Backprojection (FBP) based algorithms. One solution is statistical 3D post-denoising of reconstructions achieved with classical FBP-based algorithms. Total Variation (TV) regularization is a powerful denoising technique that preserves edges and details in images. To obtain desired speed and portability, OpenCL is suitable for implementing TV algorithms on GPUs.

In this work, we provide optimized 3D TV algorithms and a statistical 3D TV algorithm on multi-GPU systems for industrial CT data in OpenCL code. The numerical and GPU implementations of these algorithms are introduced and analyzed. The results show that the statistical denoising algorithm can improve the denoising performance both visually and quantitatively.

Contents

Part I.

Introduction and Background

1. Introduction

1.1. Motivation

A Computed Tomography (CT) scan is a powerful tool that generates high-resolution images of the body's internal structures used in the field of medical imaging. CT scans provide accurate and precise diagnostic information, such as the size, shape, and density of organs, tissues, bones, and blood vessels, enabling healthcare professionals to detect abnormalities and then make decisions regarding patient care. CT scanners use a rotating X-ray tube and a row of detectors placed in a gantry to measure X-ray attenuation by different tissues inside the scanned object. The multiple X-ray measurements taken from different angles are then processed on a computer using tomographic reconstruction algorithms to produce tomographic images of this object.

Based on the principles of medical CT scans, industrial CT scans visualize the internal structures of the scanned objects without causing any damage or altering their integrity. The Three-dimensional (3D) visualization enables the assessment of the quality, integrity, and dimensional accuracy of manufactured components and materials with non-destructive testing. CT scans have a broad range of applications in various industries, including materials science, electronics, automotive, aerospace, medical devices, and so on. Another advantage of industrial CT scans is the possibility of providing comprehensive quantitative data. By analyzing the CT scan data, measurements such as dimensions, distances, angles, and volumes can be accurately determined.

Despite its advantages, industrial CT scans are not without limitations. Unlike medical CT scans, which is typically applied to image smaller objects, such as the human body, industrial CT scans are used to image a wide range of objects, from small electronic components to larger aerospace components or even entire vehicles. Scanning such large or intricate objects may require a longer scan time, which has an impact on throughput and productivity in industries. Nowadays, the demand for faster throughput time in industrial CT scans is increasing due to the need for faster productivity and enhanced quality control. It can be achieved by either reducing the number of projections which may lead to under-sampling artifacts, or the X-ray dose. Lowering the X-ray dose leads to a reduced number of X-ray photons detected by the CT scanner. As a consequence, the Signal-to-noise Ratio (SNR) in the acquired CT images decreases, giving rise to an increase in image noise. This noise degradation results in insufficient reconstruction quality when employing classical Filtered Backprojection (FBP) [1] based algorithms. One strategy is iterative reconstruction incorporating statistical information to optimize projection data, but this is quite time-consuming for iterative reconstruction.

One alternative is statistical 3D post denoising of reconstruction achieved with classical FBP based algorithms. Total Variation (TV) regularization [2] is a powerful denoising technique while preserving edges and details in images. It imposes the prior assumption of a piecewise smooth image with occasional sharp edges by reducing the total amount of variation. Total Variation denoising stands out as one of the most popular methods for smoothing, manifesting in numerous distinct mathematical formulations and offering diverse numerical solutions. It is crucial to choose an algorithm regarding performance and computing efforts. Typically, a white Gaussian noise model based on a single estimated parameter is assumed for the images. However, due to variations in X-ray attenuation for different materials, the statistical nature of X-ray detector, etc., the noise levels in different parts of the image after reconstruction may vary [3]. Therefore, statistical information of noise level has to be taken into consideration for image denoising in this case.

In recent years, image sizes in tomographic applications have increased significantly, driven by higher resolution and more comprehensive imaging. Larger image sizes are enabled by the development of X-ray sources, detector technologies, and computational resources. This larger image size, unfortunately, results in higher storage requirements and longer processing time. Especially for industrial CT data, the volume size can reach 2k or 4k in each dimension, which leads to 32 GB and 256 GB respectively for the image size if stored in single float precision. These image sizes are a big challenge for compute resources. To overcome the challenge, powerful compute resources and efficient data management schemes are required.

Graphic Processing Unit (GPU) has been an essential part of providing processing power for high performance computing applications over recent years. General Purpose GPU (GPGPU) Programming is general purpose computing with the use of GPUs. This is done by applying a GPU together with a Central Processing Unit (CPU) to accelerate the computations in applications that are traditionally handled by only the CPU. However, each GPU has limited memory available to store and process data, especially for large industrial CT data, which may need memory-demanding computations. By multi-GPU configuration, several GPUs are combined together to offer more available GPU memory. Apart from memory, with more GPUs, the computation can leverage more parallelization among multi-GPU to achieve faster execution time.

With the aim of better utilizing the compute resources, GPU-accelerated frameworks are specifically designed for High Performance Computing (HPC). Compute Unified Device Architecture (CUDA) and Open Computing Language (OpenCL) are two widely used low-level GPU programming frameworks. For low-level languages, programmers are capable of straightforwardly optimizing specific applications depending on the sophistication of the GPU. CUDA gives fine-grained control over the GPU, yet it's tailored exclusively for NVIDIA GPUs, limiting the feasibility of transitioning the algorithms written in CUDA from NVIDIA GPUs to AMD GPUs or Intel GPUs. In contrast, OpenCL has the advantage in terms of portability when compared to CUDA.

In this work, we provide optimized statistical 3D denoising algorithms on multi-GPU for industrial CT data in OpenCL code. While obtaining desired denoising quality, these algorithms take much less time compared to iterative reconstruction.

1.2. Thesis Outline

The structure of the thesis is as follows: This Chapter introduces the research topic and outlines the objectives and significance of the study. The literature review and related work are presented in Chapter 2, which synthesizes previous research and identifies gaps that this thesis aims to address. Chapter 3 describes the numerical implementation of our total variation algorithms. The implementation of algorithms on GPU is presented in Chapter 4, followed by a comprehensive analysis and optimization scheme of performance on GPUs in Chapter 5. After that, Chapter 6 presents the numerical results and comparison of image quality. Finally, Chapter 7 states the conclusions, highlighting the contributions of this research and providing recommendations for future studies. Appendix A includes supplementary data and supporting materials for further reference.

2. Related Work

2.1. Problem Statement

Over the past few decades, the development of hardware has consistently elevated image quality, but the influence of numerous factors during the image acquisition process and subsequent processing introduces inevitable noise degradation. This causes a loss of image quality, impacting both visual clarity and image accuracy. The target of image denoising is to remove noise from a noisy image in order to restore the true image. Mathematically, the problem of image denoising can be modeled as follows:

$$y = x + n, \tag{2.1}$$

where y is the observed noisy image, x is the unknown true image, and n represents Additive White Gaussian Noise (AWGN) with standard deviation σ_n [4].

Distinguishing between noise, edges, and textures during the denoising process proves challenging, as they are high-frequency components. Due to the fact that solving the true image x from Equation (2.1) is an ill-posed problem, there exists no unique solution [5]. For the purpose of obtaining a good estimation of the unknown true image, image denoising has been studied for a long time in the field of image processing. So far, various methods for decreasing noise have been proposed. The classical denoising approach, operating within the spatial domain, involves noise reduction based on pixel correlations. This method has the distinct advantage of fast processing, such as Median Filtering [6], Gauss Filtering [7], Total Variation Regularization [2], etc.

Traditional denoising methods, such as Gaussian Filtering, tend to blur edges, but Total Variation Regularization can keep sharp edges, making it suitable for applications where edge preservation is essential. TV denoising algorithms are suitable for implementation on GPUs or other parallel computing architectures, so they can take advantage of hardware acceleration to process large images efficiently. The total variation norm is defined as the sum of the L2-norm of the directional gradients of the variable:

$$TV(u) = \sum_n \|\nabla u_n\|_2, \tag{2.2}$$

where n is the number of elements. In the context of 3D CT scans, the total variation norm encapsulates the total sum of total variation throughout the volume. A diminished total variation corresponds to a smoother volume, and this volume will have more homogeneous regions.

2.2. Total Variation Denoising

Paper [2] is one of the most influential papers in the field of image denoising, and in the paper, L. I. Rudin, S. Osher, and E. Fatemi proposed Rudin-Osher-Fatemi (ROF) model, which is now widely used in minimizing total variation. The target function consists of a data fidelity term and a total variation regularization term and solves an unconstrained problem:

$$u = \arg\min_u \frac{\alpha}{2}\|u - f\|^2 + TV(u), \tag{2.3}$$

where

- the first term $\frac{\alpha}{2}\|u - f\|^2$ is the data fidelity term which forces the denoised result to be close to the original image, and is a convex function. u is the variable to be optimized, and $u \in \mathbb{R}^N$. In our case, u is the total volume, and N is the total number of voxels. u has the same size as the original noisy volume f. The value of the hyperparameter α controls the strength of this regularization.
- the second term $TV(u)$ is the TV regularization term, as presented in Equation (2.2). It performs noise removal and is also a convex function [8].

Primal-dual (PDU) is a solution to the ROF problem and was presented in [9]. PDU converted the minimization problem to a saddle point problem by introducing a dual variable $p = (p^1, p^2, p^3)^T$ for the 3D case. By reformulating Equation (2.3) with the dual variable p, the target function can be rewritten after applying the Cauchy-Schwartz inequality [10]:

$$u = \arg\min_u \arg\max_{\|p\|<1} \|p\nabla u\| + \frac{\alpha}{2}\|u - f\|^2. \tag{2.4}$$

This saddle point problem can be solved by minimizing regarding u and maximizing regarding p.

The ROF model results from the assumption of the additive Gaussian noise model presented in equation 2.1. However, if the characteristic of noise varies across the image, the variance of η is multivariate, and the statistical modeling approach leads to the weighted least-squares fidelity term where the weight w is specified by the noise variances [11] at each element. The ROF model then is adapted to:

$$u = \arg\min_u \frac{\alpha}{2}\|u - f\|^2_{\frac{1}{w}} + TV(u), \tag{2.5}$$

where $\|x\|_z^2 = \langle zx, x\rangle$, and denotes a weighted scalar product with z being positive.

Estimating noise [12] is a crucial step in various image denoising algorithms as accurate noise estimation helps to choose the optimal denoising algorithm and parameters to effectively reduce noise while preserving image details. There are numerous methods to estimate noise, such as local variance [13], Block Matching and 3D Filtering [14], Wavelet-Based methods [15], etc. Paper [16] presents a methodology to obtain reliable pixel-wise noise estimates, which computes image variance by propagating the noise estimation in projections through the reconstruction pipeline, which enables the local noise statistics to be taken into account in the post-processing of the image. The noise in projections is estimated by using the local variances, but the non-stationarity is also taken into account. [16] provides a way for us to combine the noise level estimated element-wise with the traditional PDU algorithm.

Different from PDU, Projection onto Convex Sets (POCS) [17] is a constrained minimization problem and can be described as:

$$u = \arg\min_u TV(u) \ subject\ to \ \|u - f\| \leq \epsilon. \tag{2.6}$$

POCS directly performs minimization regarding the total variation norm, but is constrained by the predefined data fidelity term. In an unconstrained problem such as PDU, the balance between the data constraint and the regularization constraint can be tuned via a hyper-parameter, such as α in Equation (2.4). In contrast, for constrained problems like POCS, multiple solutions for the data fidelity term may exist. POCS consistently integrates with reconstruction algorithms, functioning as a regularization term, but it frequently encounters the challenge of excessive smoothing, which can obscure vital details within the image.

In [18], an Adaptive-weighted Total Variation (AW-TV) minimization algorithm was derived from POCS by considering the anisotropic edge property among neighboring image voxels. A higher weight is assigned when there is a minor alteration in voxel intensity. Conversely, a lower weight can be incorporated if there is a substantial change in voxel intensity, achieved through the utilization of an exponential function. Adaptive weights are applied locally regarding the intensity difference of neighboring voxels. The weights are smaller when dealing with edges with more pronounced changes in voxel intensity. This indicates that the structure of edge regions is then better preserved in the regions exhibiting weaker smoothing strength. Diverging from the statistical model outlined in Equation (2.5), the weights in this context are modulated by voxel intensity disparities rather than noise levels. Consequently, the adaptive-weighted minimization approach does not conform to the statistical denoising methodology.

Nowadays, bolstered by increasingly powerful computing resources, particularly GPUs, the image denoising process can be accelerated in terms of speed and efficiency. This acceleration is attributed to the fact that image denoising algorithms frequently involve computationally intensive operations, which can benefit from parallel processing capabilities. This is a key advantage of GPUs. Paper [19] presents the application of POCS minimization as an additional constraint to CT reconstruction problems on multi-GPUs written in CUDA programming languages, and it demonstrates the possibility of acceleration for total variation regularization on multi-GPUs. Inspired by this, we propose an optimized statistical 3D denoising algorithm in OpenCL for large data on multi-GPUs to achieve desired denoising quality.

Part II.

Implementation

3. Numerical Implementation

In our work, two total variation algorithms, PDU and Total Variation Gradient Descent (TV-GD) are implemented and analyzed on multiple GPUs in OpenCL code. Apart from that, PDU, in conjunction with statistical noise information, is also conducted and compared with the classical PDU approach. In this chapter, the numerical solutions derived from these algorithms are presented and analyzed.

3.1. Primal-dual

Primal-dual (PDU) is a solution to the ROF problem by introducing the dual variable, as shown in Equation (2.4), the minimization problem of ROF in Equation (2.3) is converted to a saddle point problem in PDU [9]. By differentiating the saddle point problem by u and p respectively, we can obtain the primal update and the dual update accordingly. Firstly after differentiating Equation (2.4) by the variable u, the primal update is then retrieved from the equation as follows:

$$-\nabla \cdot p + \alpha(u - f) = 0. \tag{3.1}$$

By applying the gradient descent update scheme, Equation (3.1) leads to

$$u^{k+1} = u^k(1 - \tau_p^k) + \tau_P^k(f + \frac{1}{\alpha}\nabla \cdot p), \tag{3.2}$$

where τ_P^k is the step size for the primal update in step k, and the choice of τ_P^k is crucial for the stability and convergence of the algorithm and will be discussed later, $\nabla \cdot p$ represents the divergence of the variable p.

Then after differentiating Equation (2.4) by p, the dual update is obtained from the equation:

$$-\nabla u + p\alpha = 0. \tag{3.3}$$

By applying the gradient descent update scheme again, Equation (3.3) leads to

$$p^{k+1} = \prod_{B_0}(p^k + \tau_D^k \nabla u), \tag{3.4}$$

with

$$\prod_{B_0}(q) = \frac{q}{\max\{1, \|q\|\}}, \tag{3.5}$$

where $\prod_{B_0}(q)$ denotes the unit ball centered at the origin and τ_D^k is the step size for the primal update in step k, ∇u represents the gradient of the variable u.

Optimal selections of τ_P^k and τ_D^k play a pivotal role in faster convergence in the PDU approach, similar to [20], the following step size choice performs well in our evaluation using testing data:

$$\tau_D^k = 0.3 + 0.02k, \tag{3.6}$$

and

$$\tau_P^k = \frac{1}{\tau_D^k}(\frac{1}{6} - \frac{5}{15+k}). \tag{3.7}$$

Algorithm 1 illustrates the update procedures for the PDU approach. The duality variable p is initialized as zeros, and the u is set as the original noisy volume itself. Line 2 pertains to the primal update in Equation (3.2), and line 3 corresponds to the dual update in Equation (3.4). In addition to step size selection, the discretization of the divergence and gradient operators is also vital for the numerical computation of the PDU algorithm. Ensuring the consistency of these two operators is essential to maintain algorithmic coherence. In our implementation, the gradient is computed by the forward difference, so the divergence should be computed by the backward difference, as the divergence is the adjoint of the gradient [21].

Algorithm 1: Numerical update approach for PDU

```
   Input: $f$, $u^0$, $p^0$, $\alpha$, $\tau_P^k$, $\tau_D^k$
1  for nIter and if stopping criteria is not met do
2      $u^{k+1} = u^k(1-\tau_P^k) + \tau_P^k(f + \frac{1}{\alpha}\nabla \cdot p^k)$
3      $p^{k+1} = \prod_{B_0}(p^k + \tau_D^k \nabla u^k)$
```

PDU yields a converged solution [9], and in order to stop after obtaining desired quality without wasting unnecessary iterations, the stopping criteria and the stability are also proposed and will be presented in Section 6.2.1.

3.2. Total Variation Gradient Descent

Unlike PDU, POCS is a constrained minimization solution that directly performs minimization regarding the total variation norm, but is constrained by the predefined data fidelity term. The POCS algorithm is often served as an additional regularization for reconstruction, such as Adaptive-steepest-descent POCS (ASD-POCS) [17] combined with the Algebraic Reconstruction Technique (ART).

However, to maintain the fast speed of the denoising algorithm, iterative reconstruction is not considered in our algorithm. Instead, only the part of the TV Gradient Descent is extracted from the complete algorithm. Algorithm 2 presents the update procedures of the TV-GD approach. The variable u is initialized as the original noisy image. First, the numerical approximation of the gradient of the total variation norm is computed in line 2 of the TV update algorithm. Then the gradient of the TV norm is normalized in line 3, and the image will be updated accordingly with a certain step size denoted by λ in line 4 of the algorithm.

Algorithm 2: Numerical update approach for TV-GD

```
  Input: $u^0 = f, \lambda$
1 for nIter and if stopping criteria is not met do
2     $\overrightarrow{du^k} = \nabla_u \|u^k\|_{TV}$
3     $\widehat{dx^k} = \overrightarrow{du^k} / \|\overrightarrow{du^k}\|_{TV}$
4     $\overrightarrow{u^{k+1}} = \overrightarrow{u^k} - \lambda \widehat{dx^k}$
```

The total variation norm is not differentiable in the general case. However, in CT Scans, the $\overrightarrow{u}$ can be described as u_{ijk}, which is a discretized 3D mesh with indices i, j, and k in x, y and z directions respectively. According to the transformation of the total variation norm in [21], the numerical approximation of the gradient of the total variation norm can be computed as:

$$
\begin{aligned}
(\nabla_u \|u\|_{TV})_{i,j,k} = {} & \frac{\partial_x u_{i,j,k} + \partial_y u_{i,j,k} + \partial_z u_{i,j,k}}{\sqrt{\sum_\beta (\partial_\beta u_{i,j,k})^2}} \\
& - \frac{(1-\delta_{i,i_{\max}})\partial_z u_{i+1,j,k}}{\sqrt{\sum_\beta (\partial_\beta u_{i+1,j,k})^2}} - \frac{(1-\delta_{j,j_{\max}})\partial_y u_{i,j+1,k}}{\sqrt{\sum_\beta (\partial_\beta u_{i,j+1,k})^2}} - \frac{(1-\delta_{k,k_{\max}})\partial_z u_{i,j,k+1}}{\sqrt{\sum_\beta (\partial_\beta u_{i,j,k+1})^2}},
\end{aligned}
\tag{3.8}
$$

which describes the scalar field of the same size as the image. The δ denotes the Kronecker deltas [22] for the Cartesian axis, and $i_{\max}$, $j_{\max}$, $k_{\max}$ represent the maximum indices in each direction. $\delta_{i,j}$ is equal to 1 if $i = j$ and $\delta_{i,j}$ is equal to 0 if $i \neq j$. The introduction of Kronecker deltas is to make sure that boundary conditions are set to zero to meet the requirements of the Neumann boundary conditions [23]. The operator ∂_β is the gradient with an additional Cartesian index β.

Adaptive-weighted POCS (AW-POCS) is similar to POCS, and the only difference is the computation of the gradient of the TV norm. In the implementation of AW-POCS, each gradient element is weighted in Equation (3.8) by an exponential function:

$$
f(z) = \exp\left[-\left(\frac{z}{\gamma}\right)^2\right] \cdot z, \tag{3.9}
$$

where γ is a scale factor which controls the strength of the diffusion. From the equation, a larger weight is assigned if there is a small change in voxel intensity, and vice versa, a smaller weight is added if there is a large change of voxel intensity. The numerical behavior of AW-POCS is almost the same as POCS [18]. Although the TV-GD algorithm from POCS implemented in our work will not converge to the optimal solution, but there is still a way to find the proper stopping criteria (cf. Section 6.2.2).

3.3. Statistical PDU

The ROF problem results from the assumption of additive Gaussian noise model with Gaussian-distributed noise as a scalar variance. Paper [11] proposes a statistical modeling approach to the ROF problem in the Equation (3.10), which leads to a weighted least squares data fidelity term with the weight as the corresponding noise variance element-wise. The element-wise noise is estimated through the method proposed in [16], which enables reliable element-wise noise estimation of 3D images through propagating the noise estimated in projections by the reconstruction pipeline. The numerical implementation of Statistical PDU (S-PDU) is similar to PDU, but as there are new weights attached to data fidelity, and the primal update should be changed accordingly. After differentiating Equation (2.5) by u, the primal update from the equation becomes

$$-w\nabla \cdot p + \alpha(u - f) = 0. \tag{3.10}$$

With the same gradient descent update scheme, Equation (3.10) leads to a similar update step:

$$u^{k+1} = u^k(1 - \tau_P^k) + \tau_P^k(f + \frac{w}{\alpha}\nabla \cdot p), \tag{3.11}$$

where the only difference from Equation (3.2) is that the hyperparameter α is element-wise weighted by the weights. The weights are directly obtained by noise estimation. The value of α needs to be tuned accordingly based on the range of noise variances. The stopping criteria of S-PDU is similar to PDU. Algorithm 3 shows the update scheme for statistical PDU. Similar to the update algorithm of PDU in Algorithm 1, u is initialized as the original noisy image, and the dual variable p is initialized as zeros. Line 2 is then changed according to the new primal update scheme shown in Equation (3.10). The step sizes τ_p^k and τ_D^k are the same as those in the PDU algorithm.

Algorithm 3: Numerical update approach for S-PDU

```
  Input: $f$, $w$, $u^0$, $p^0$, $\alpha$, $\tau_P^k$, $\tau_D^k$
1 for nIter and if stopping criteria is not met do
2     $u^{k+1} = u^k(1 - \tau_p^k) + \tau_P^k(f + \frac{w}{\alpha}\nabla \cdot p^k)$
3     $p^{k+1} = \prod_{B_0}(p^k + \tau_D^k \nabla u^k)$
```

4. GPU Implementation

With the growing scale of image sizes in industrial CT, the demand for robust compute resources has escalated. In contrast to CPUs, which usually feature a limited number of cores optimized for sequential processing tasks, GPUs are designed with hundreds or even thousands of cores, facilitating the simultaneous execution of multiple computations. This inherent parallel processing capability renders GPUs remarkably efficient for applications that can be parallelized, such as image processing, machine learning, and scientific computation. Furthermore, GPUs provide significantly higher bandwidth than CPU, which allows them to access data in a more efficient parallel manner. Nevertheless, individual GPUs possess finite memory capacity, posing a challenge for handling substantial industrial CT data, which often requires memory-intensive computations. To address this challenge, multi-GPU configurations merge multiple GPUs, augmenting the available GPU memory.

To utilize the high bandwidth of GPUs efficiently, the data partitioning and memory management scheme are crucial parts of GPU implementation to enable data parallelism. In the context of industrial CT data, the volume size can be 2k or 4k in each dimension, leading to 32GB or 256 GB for the whole volume size stored in single float precision. Even for multi-GPU setups, the GPU memory is infeasible to keep the whole volume. Thus, a suitable image splitting scheme becomes indispensable to facilitate algorithm execution within the constraints of limited GPU resources. Under some conditions, the image needs to be split into several regions first and then distributed each region as individual parts evenly among GPUs.

4.1. Overview of OpenCL

Our implementation of these denoising algorithms is based on OpenCL, and this programming language will be introduced first in the following sections.

Open Computing Language (OpenCL) [24] is a parallel programming standard across heterogeneous platforms consisting of Central Processing Units (CPUs), Graphics Processing Units (GPUs), Digital Signal Processors (DSPs), Field-programmable Gate Arrays (FPGAs) and other processors or hardware accelerators. OpenCL was initially created by Apple Inc, and the first version, OpenCL 1.0, was released in 2009. Then OpenCL has been maintained by Khronos Group. Now the latest specification, OpenCL 3.0, was released in 2020.

4.1.1. Advantages of OpenCL

The Khronos Group also developed SYCL, which is a higher-level programming model for OpenCL to improve programming productivity [25]. SYCL developers can write code using C++ and apply modern C++ features, such as templates, lambda functions, and classes. SYCL abstracts lots of low-level details and thus reduces the learning effort and puts more focus on parallel programming. Typically, the application written in SYCL requires fewer lines of code to implement the compute functions and needs less calls to Application Programming Interface (API) functions. As for portability, SYCL is built based on OpenCL and inherits its portability across devices and vendors. SYCL code can also run on the devices that support OpenCL.

However, compared to the higher-level programming models, the low-level programming models OpenCL and CUDA continue to provide the highest possible performance [26].

Compared to CUDA, OpenCL provides an advantage in terms of portability. OpenCL focuses on data-parallelism and task-parallelism, and provides abstractions for memory management, thread scheduling, and synchronization, so it has a higher level than CUDA. Although some papers [27] and [28] point out CUDA is about 30% faster than OpenCL by comparing CUDA programs with OpenCL on NVIDIA GPUs, CUDA only supports NVIDIA computing resources and is not available on AMD and Intel hardware.

4.1.2. OpenCL Architecture

At its core, the OpenCL architecture is structured around several key components [29]:

- Platform and Compute Units: in the OpenCL architecture, the platform groups all hardware capable of executing an OpenCL program. It typically includes a host and one or more compute devices, such as GPUs. The host is responsible for running the main application and creating and managing the OpenCL context, which provides the interface between the host and the OpenCL devices. The host is where the OpenCL code is executed, and it interacts with the underlying devices to initiate and control parallel computations. The compute device is a processing unit capable of executing parallel computations. Each device is further divided into a set of compute units. The number of compute units depends on the target hardware. A compute unit is further subdivided into processing elements. A processing element is the fundamental computation engine in the compute unit, which is responsible for executing the operations of one work-item.
- Work-item and Work-group: in OpenCL, computations are divided into work-items, which are individual threads executing the same code on different data elements. Work-items are grouped into work-groups, which are collections of work-items that can share data and synchronize their execution.
- Kernel and Command Queue: a kernel is a function written in the OpenCL C programming language that defines the computation to be executed by each work-item. Kernels are compiled at runtime and executed in parallel across multiple work-items. To execute OpenCL kernels on a device, the host submits commands to the device through a command queue. The command queue ensures that kernels are executed in the correct order and handles data transfers between the host and devices.
- Memory Hierarchy: the OpenCL memory hierarchy includes different levels of memory available in the computing system and how data is stored and accessed at each level during the execution of parallel computations. It includes global memory, local memory, and private memory. The global (constant) memory is shared by all

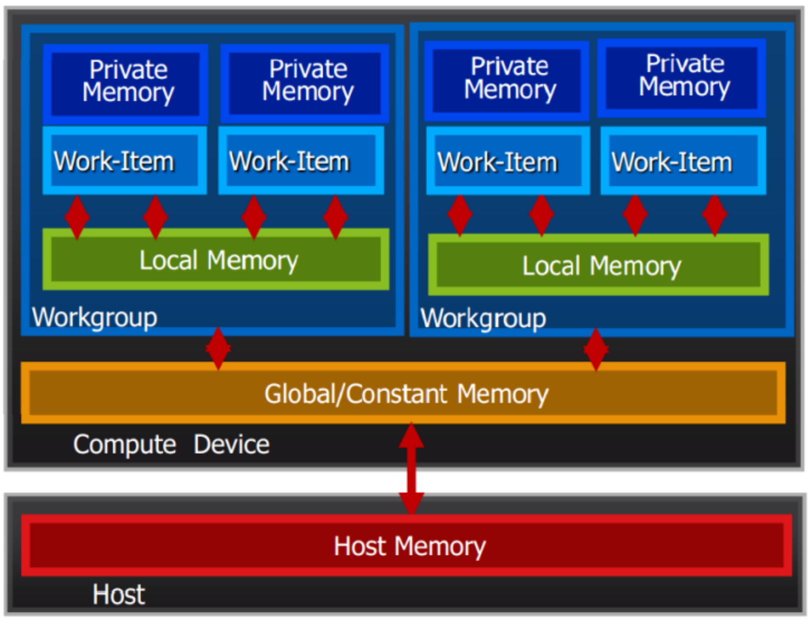


Figure 4.1.: OpenCL memory model (taken from [24]).

work-items, and all compute units in the device, and access to this memory is the slowest. Global memory is used to store data that needs to be shared among different work-groups, allowing communication and data exchange between work-groups. Local memory is shared by work-items within a work-group. Each work-group has its own dedicated local memory space. Local memory is used for sharing data between work-items within the same work-group, enabling efficient data exchange and cooperation during parallel computations. Private memory is owned only by the work-item itself. It is used to store private variables and intermediate results that are only relevant to a single work-item. Each work-item has its own private memory space, and data stored in private memory is not visible to other work-items, as shown in Figure 4.1.

Overall, the OpenCL architecture enables developers to write parallel code that can take advantage of the computational power of various devices. By dividing computations into work-items and work-groups, and utilizing the memory hierarchy efficiently, developers can achieve significant performance gains in their parallel applications.

4.1.3. OpenCL Environment

Creating an OpenCL environment is essential to execute OpenCL code, and this involves several steps, including setting up the host code, initializing OpenCL, discovering available platforms and devices, creating a context and command queues, loading and building kernels, and managing memory. Below is a general outline of how to create an OpenCL environment [30]:

1. Include OpenCL Headers: start by including the necessary OpenCL headers, which

contain the definitions of OpenCL functions and data types in the host code.

2. Discover Platforms and Devices: discover available platforms and devices. We can obtain the corresponding information on the system using OpenCL API call $clGetPlatformIDs()$, $clGetDeviceIDs()$, $clGetDeviceInfo()$ and so on.
3. Create a Context and Command Queues: after discovering platforms and devices, the context needs to be created, which serves as the interface between the host and the devices using $clCreateContext()$. Then create one or more command queues associated with the context using $clCreateCommandQueue()$.
4. Build Program and Create Kernels: write the parallel computation tasks as OpenCL kernels in the OpenCL C programming language. Load these kernels from source files or strings using $clBuildProgram()$ to build a program executable and then create kernels at runtime using $clCreateKernel()$.
5. Create Buffers and Transfer Data: allocate and manage memory on both the host and devices using OpenCL buffer objects by $clCreateBuffer()$. Transfer data between the host and devices as needed for the computations using corresponding OpenCL API calls.
6. Set Kernel Arguments and Enqueue Kernels for Execution: set the arguments of OpenCL kernels using $clSetKernelArg()$. Use the command queues to enqueue kernels for execution on the devices by $clEnqueueNDRangeKernel()$.
7. Execute Kernels and Synchronize: execute the enqueued kernels on the devices using the command queues. Properly synchronize the host and devices to ensure correct execution and avoid data hazards.
8. Environment Cleanup: release any OpenCL resources, such as buffers, kernels, command queues, and the context, to free up memory and avoid memory leaks using appropriate OpenCL API calls.

4.2. Image Splitting and Distribution Scheme

3D images inherently possess three dimensions in total. Our image splitting scheme adopts that the image is split among a predefined dimension, and the other two dimensions remain untouched. Then, the total image is divided into a collection of 2D image slices regarding the specific dimension. In our case, we denote the specific dimension as the z direction by default.

When implementing the TV denoising algorithms, we have to consider the discrete nature of the data. To update a voxel, the information of neighboring voxels are also needed according to the update scheme in Algorithm 1 and 2. Since the image has to be split into multiple parts, the additional neighboring data (often referred to as buffers) needs to be added for each image part. Because we use backward differences only for the computation of the gradients in both x, y, and z directions, and the neighboring information can be easily accessed from image slices in the x and y directions. Only the buffers from neighboring slices at the left side in the z direction are necessary.

Here we consider two conditions:

- Adequate GPU Memory: in cases where the requisite memory fits entirely within the GPUs, image splitting is unnecessary. The image can be directly distributed evenly across GPUs with appropriate buffers. Here we apply the Single-split Multi-GPU scheme presented in Section 4.2.1.
- Limited GPU Memory: when GPU memory is insufficient, the image is divided into distinct regions. Each region is then evenly allocated across GPUs. In this case, Multiple-splits Multi-GPU scheme is used, which is demonstrated in Section 4.2.2.

The memory requirements and the number of divisions are determined by factors like image size, number of copies, buffer length, and auxiliary memory.

Splitting images across multiple GPUs may introduce challenges such as communication overhead, synchronization issues, etc. The problem will be discussed next in Chapter 5. Efficient data splitting also involves load balancing, where efforts are made to distribute the workload evenly among GPUs to avoid any one GPU being idle while others are overloaded.

4.2.1. Single-split Multi-GPU Scheme

If the memory needed can be fully kept on GPUs, then splitting is not necessary, and the image is distributed evenly into GPUs. For each part on the GPU after distribution, the buffers should also be added alongside the parts. Buffers are taken from the neighboring part of the volume. Due to the characteristics of the algorithms we chose, only the left buffers are required. Figure 4.2 shows the distribution of an image into four GPUs. Suppose there are a total of four available GPUs, and the image is evenly distributed into four distinct parts, each indicated by a different color. Each of these parts is then assigned to a dedicated GPU for computation with corresponding buffers taken from the left neighboring part of the image. In addition, each of the buffers has the same size.

The length of the buffer also corresponds to the number of iterations that could be run independently on GPUs. Following each set of independent iterations, a synchronization step is required to update the buffers, ensuring their alignment for subsequent iterations.

4.2.2. Multiple-splits Multi-GPU Scheme

The second condition is that the GPU memory is not enough or the memory needed is too large, so the memory needed can not be kept on GPUs. In that case, the image should be first split into several regions according to the image size and GPU memory size. Figure 4.3 shows the image splitting and distribution scheme in this condition. Given the availability of four GPUs, the image is strategically divided into three distinct regions in the first row based on the image size and the GPU memory size, each represented by a different color. Subsequently, each of these regions is further distributed into four equal parts, as depicted in the third row, each distinguished by a unique color. These divided parts are individually assigned to a separate GPU for computation purposes with corresponding buffers taken from the left neighboring part of the image. Each of the buffers has the same size.

Different from the first condition, the GPUs can only process one region at one time, and the GPU memory will be rewritten if the GPUs start to iterate a new region, so the updated data on GPUs will be eliminated and needs to be consolidated in CPU memory for later

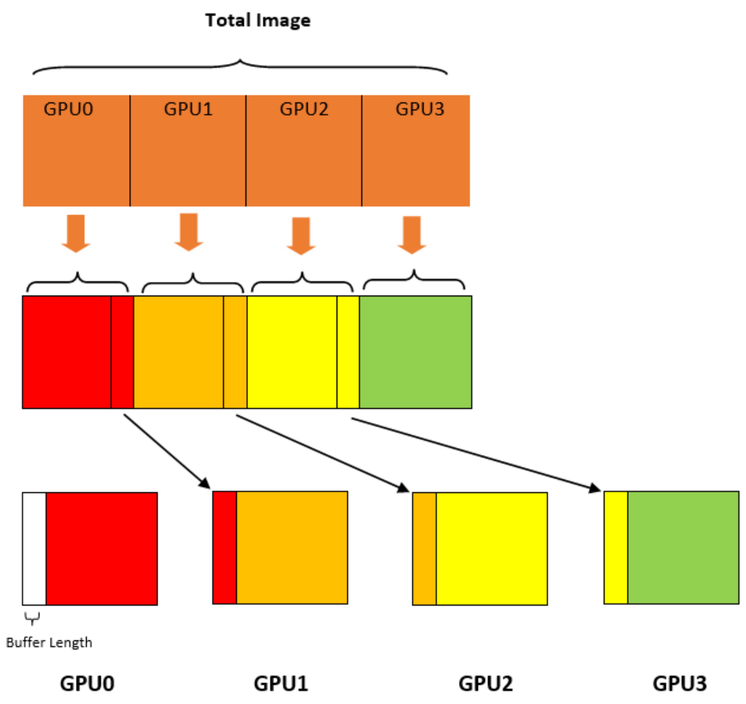


Figure 4.2.: Image splitting scheme for the case of the Single-split Multi-GPU scheme.

iterations. After independent iterations on GPUs for one region, the central part, excluding the buffers, needs to be copied back to the CPU, and then the GPUs can continue to process the next region. After iterating the whole image, all the regions are combined again in the CPU, and a new cycle will begin again from the first region with corresponding data and information if there are still remaining iterations.

The length of the buffer also corresponds to the number of iterations that could be run independently on GPUs. Different from the condition of as the Single-split Multi-GPU scheme, after the independent number of iterations, the data are merged again in CPU memory, and the details will be explained in Section 4.2.3.

4.2.3. Buffer Synchronization Scheme

Each time after independent iterations, the buffers are needed to be synchronized between image parts and updated to ensure the correctness of the computation. In the case of the Single-split Multi-GPU scheme, the data are fully kept on GPU. There is no need to store the data back to the CPU. The buffers can be directly synchronized between GPUs which saves lots of time for memory transfer operations. We only consider the buffers on the left side, so only forward buffer synchronization is performed, as shown in Figure 4.4. The corresponding part of the updated image is then written to the buffer on the right side for each GPU. When employing the Multiple-splits Multi-GPU scheme, it is important to consider if there are still remaining iterations. If yes, consequently, the buffers are then not needed anymore. The parts without buffers need to be transferred back to the CPU, which may take much longer time than synchronization between GPUs.

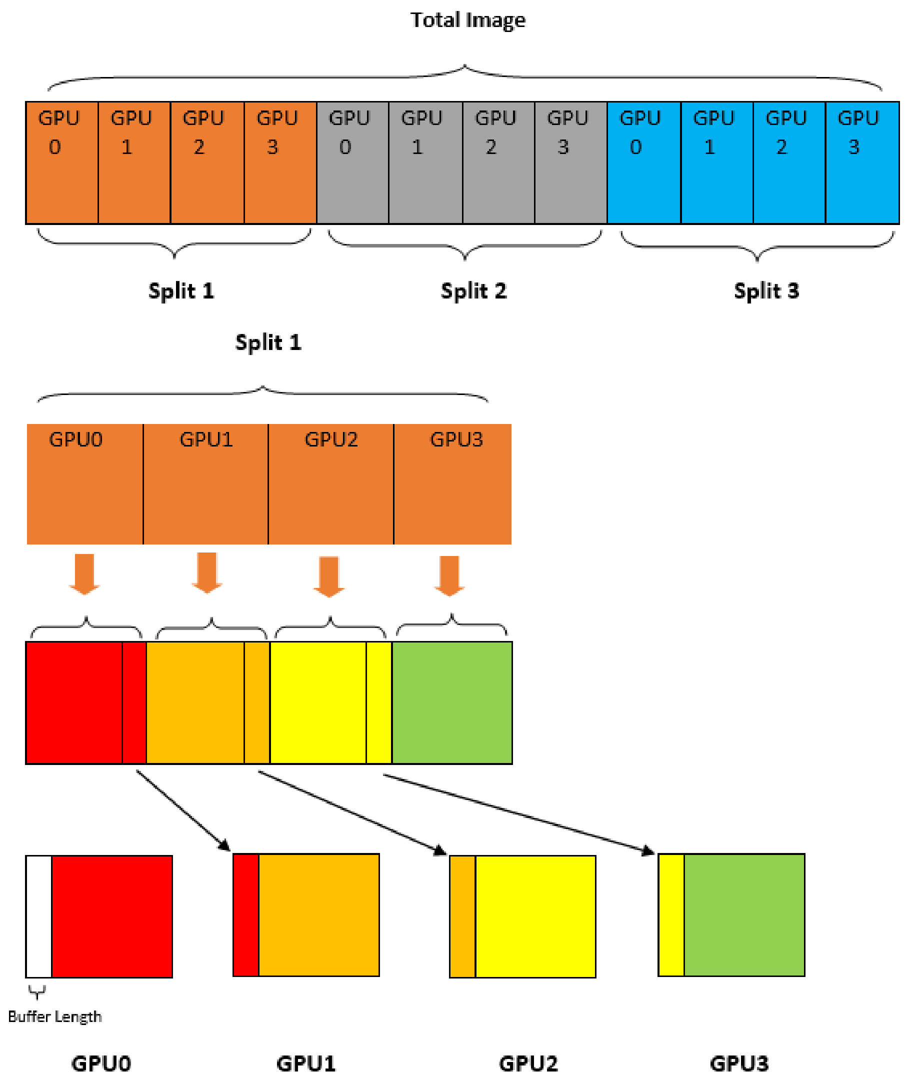


Figure 4.3.: Image splitting scheme for the case of the Multiple-splits Multi-GPU scheme.

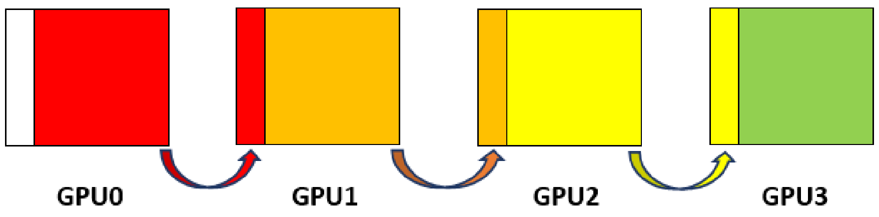


Figure 4.4.: Forward buffer synchronization scheme.

The length of the buffer corresponds to the number of iterations that could be run independently on GPUs. With a larger buffer length, the frequency of memory transfer is reduced, but the time for each independent set of iterations and the size of memory transfer is increased. A good choice of buffer length is crucial to balance the memory transfer time and the computation time. The minimum length we have set in our implementations is 10, and the maximum length we have set is 60. The final choice of the buffer length depends on the free memory after considering the memory needed for the central parts of the image.

4.3. Workflow on GPUs

Algorithm 4 and Algorithm 5 describe the workflow on CPU and GPUs for PDU and TV-GD respectively. In both algorithms, the input variable h_{src} represents the original image. The variable h_{dst} is initialized as an array of zeros, serving as a container for the updated image. The value of $nIter$ represents the predetermined number of iterations, while the associated parameters α and λ also play a crucial role in the calculations.

In the context of OpenCL, the environment of OpenCL should be first created, and the resources should be freed in the end. The data and memory management scheme is similar for both algorithms. There are two main differences between the implementation of PDU and TV-GD:

- PDU needs five copies in total for both CPU and GPU memory, while TV-GD needs only one copy for CPU memory and two copies for GPU memory. When performing the image splitting and distribution scheme, the number of copies also needs to be taken into account in the computation of needed memory. For example, the memory needed is five times the memory of image size in PDU and twice the memory of image size in TV-GD.
- The kernels used in PDU and TV-GD are different. For PDU, there are only two kernels called $Update\_U$ and $Update\_P$ which correspond with the primal update and the dual update step respectively. For TV-GD, there exist four kernels in total which are called $GradientTV$, $ReduceNorm$, $ReduceSum$, and $UpdateImg$. The kernel $GradientTV$ corresponds to the computation of the gradient of the TV norm in line 3 of Algorithm 2. According to line 4 of the TV update algorithm, the kernels $ReduceNorm$ and $ReduceSum$ are reduction operations to obtain the sum of the L2-norm of the gradient of the total variation norm from GPUs. We have to mention that synchronization is needed before summing up the value in CPU. After obtaining the sum of the L2-norm

of the gradient of the total variation norm, the kernel, $UpdateImg$, performs the operation in line 5 of the TV update algorithm.

Algorithm 4: Workflow for implementation of PDU on GPUs

```
   Input: $h\_src$, $h\_dst$, $nIter$, $\alpha$
1  Check platforms and GPUs
2  Determine splits and bufferLen
3  Create an OpenCL environment
4  for nIter/bufferLen do
5      for nSplits do
6          if the first iteration then
7              Write original image $CPU \rightarrow GPUs$
8              Copy original image to modified image $GPUs \rightarrow GPUs$
9              Synchronize()
10         if not first iteration and nSplits>1 then
11             Write modified image $CPU \rightarrow GPUs$
12             Write px, py, pz $CPU \rightarrow GPUs$
13             Synchronize()
14         for bufferLen do
15             Update_U $\lll Launch \ggg$
16             Update_P $\lll Launch \ggg$
17         Synchronize()
18         if nSplits=1 then
19             if has remaining iterations and nGPUs>1 then
20                 Forward buffer synchronization $GPUs \leftrightarrow CPU$
21         else
22             Read modified image $GPUs \rightarrow CPU$
23             if has remaining iterations then
24                 Read px, py, pz $GPUs \rightarrow CPU$
25 if nSplits=1 then
26     Read modified image $GPUs \rightarrow CPU$
27 Free GPU resources
   Output: h_dst
```

Algorithm 5: Workflow for implementation of TV-GD on GPUs

```
   Input: $h\_src$, $h\_dst$, $nIter$, $\lambda$
1  Check platforms and GPUs
2  Determine splits and bufferLen
3  Create an OpenCL environment
4  for nIter/bufferLen do
5      for nSplits do
6          if the first iteration then
7              Write original image $CPU \rightarrow GPUs$
8          if not first iteration and nSplits>1 then
9              Write modified image $CPU \rightarrow GPUs$
10         for bufferLen do
11             GradientTV$\lll Launch \ggg$
12             ReduceNorm2$\lll Launch \ggg$
13             ReduceSum$\lll Launch \ggg$
14             Synchronize()
15             <CPU Code >
16             UpdateImg$\lll Launch \ggg$
17         Synchronize()
18         if nSplits=1 then
19             if has remaining iterations and nGPUs>1 then
20                 Forward buffer synchronization $GPUs \leftrightarrow CPU$
21         else
22             Read modified image $GPUs \rightarrow CPU$
23 if nSplits=1 then
24     Read modified image $GPUs \rightarrow CPU$
25 Free GPU resources
   Output: h_dst
```

Part III.

Experiments and Conclusions

5. GPU Performance

To test the performance of the OpenCL code on multi-GPU environment, our experiments are conducted with varying image sizes on up to four GPUs. Workstations with different types of GPUs are utilized to demonstrate the case for GPU portability of the OpenCL code. One workstation with four AMD FirePro W8100 GPUs is coupled with an Intel Xeon E5 2650 v2 CPU. Each AMD FirePro W8100 GPU has a memory size of 8GB, a theoretical bandwidth of 320 GB/s, and a theoretical single float point precision compute power of 4.21 Tera Floating Point Operations Per Second (TFLOPS). Complementing this, the Intel CPU is accompanied by 128GB of Random-access Memory (RAM). On another workstation, four NVIDIA RTX 3080 TURBO GPUs are connected with an AMD EPYC 7402 CPU. Each NVIDIA RTX 3080 TURBO GPU has a memory size of 10GB, a theoretical bandwidth of 760.3 GB/s, and a theoretical single float point precision compute power of 29.77 TFLOPS. The AMD CPU is equipped with 256 GB of RAM.

To thoroughly examine the GPU performance, the utilization of profiling tools becomes essential, enabling the tracing of hardware usage. For CUDA-based programs, profiling is conveniently achieved through NVIDIA's profiling and tracing tools, such as the NVIDIA Visual Profiler. The NVIDIA Nsight Visual Studio has emerged as a more comprehensive option compared to the NVIDIA Visual Profiler, offering enhanced profiling capabilities. When dealing with OpenCL-coded programs, several profiling tools are available, including AMD Radeon GPU Profiler, AMD CodeXL, and CLtracer. Nonetheless, these tools may not match the robustness of NVIDIA's profiling tools. They might lack the ability to provide detailed hardware insights during the process and might not generate critical reports like the Roofline model [31], which assesses performance bottlenecks.

In order to estimate the performance of our algorithms on multi-GPU setups, the performance of the compute kernels is analyzed by the Roofline model, and GPU scaling behavior is compared across different numbers of GPUs and varying data sizes. In addition, GPU memory access patterns, parallel execution, and synchronization are explored.

5.1. Kernel Analysis

As stated, each AMD FirePro W8100 GPU on the first workstation has a theoretical bandwidth of 320 GB/s and a theoretical single float point precision compute power of 4.21 TFLOPS. Each NVIDIA RTX 3080 TURBO GPU on the second workstation has a theoretical bandwidth of 760.3 GB/s and a theoretical single float point precision compute power of 29.77 TFLOPS. These theoretical indicators are essential for analyzing the efficiency of the kernels.

As shown in the GPU pseudo code of PDU in Algorithm 4 and TV-GD in Algorithm 5, there are several computation functions in the inner loop. The functions are also called kernels which are intended to be executed in parallel on GPUs. Kernels are the fundamental

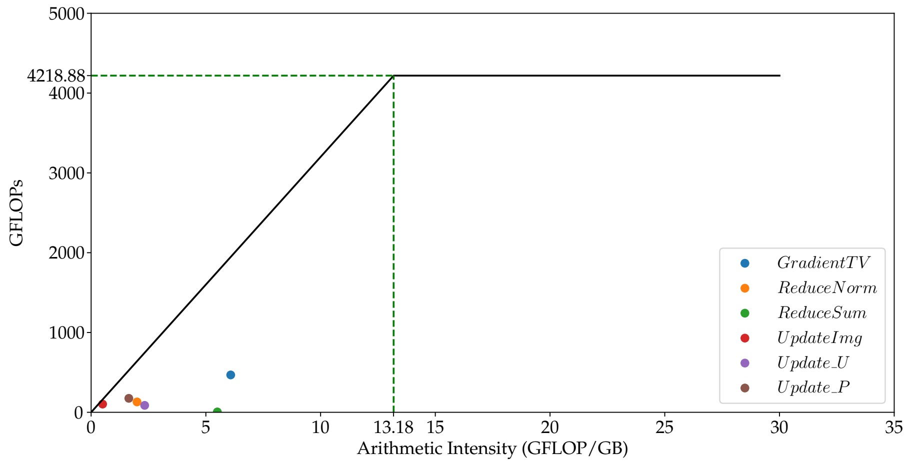


Figure 5.1.: Roofline model for the kernels in both PDU and TV-GD approaches tested on the AMD FirePro W8100 GPU.

units of computation in OpenCL. A kernel is distinguished from the normal C or C++ function by adding the $\_\_kernel$ specifier in the front.

It is crucial to write a computation function to gain good hardware resource utilization. A kernel could be either memory-bound or compute-bound. Memory-bound refers to a situation in which the time to complete a given computational problem is decided primarily by the amount of free memory required to hold the working data. It also means that most of kernel time is spent in executing memory instructions. This is in contrast to a kernel that is compute-bound, where the number of elementary computation steps is the deciding factor. GPUs have high memory and compute bandwidth and can be suitable for both categories. The terms are used for categorization and to indicate which optimization techniques may improve the performance of application significantly. To discriminate whether a function is memory-bound or compute-bound, the Roofline model is used together with theoretical peak bandwidth, peak performance, and arithmetic intensity.

Floating Point Operations Per Second (FLOPS) is an important index to measure the compute power or performance of a processor. The specification of GPU always provides the theoretical maximum single float point compute power or double float point compute power for reference.

A given kernel is characterized by a point given by its arithmetic intensity I. The arithmetic intensity is defined as the ratio of the number of FLOPS per byte of memory traffic. If the arithmetic intensity of a kernel is smaller than the division of peak performance and peak bandwidth, then this kernel is characterized to be memory-bound, and vice versa, the kernel is characterized to be compute-bound. Figure 5.1 plots the Roofline model for all the kernels in PDU and TV-GD. The result shows that all the kernels are memory-bound.

For compute-bound kernels, FLOPS is the right metric that can reflect the distance to GPU peak performance. In addition, bandwidth is chosen as the metric for memory-bound kernels to check GPU efficiency. Table 5.1 documents the GPU bandwidth efficiency for kernels in both PDU and TV-GD approaches, CPU time, and theoretical time for specific

	GradientTV	*ReduceNorm*	*ReduceSum*	*UpdateImg*	*Update_U*	*Update_P*
Data size	3.801 GB	3.675 GB	0.0287GB	3.801 GB	1.489GB	1.489GB
Theoretical time	0.0356 s	0.0115 s	8.97E-05 s	0.0356 s	0.0279 s	0.0326 s
CPU time	0.148 s	0.0566 s	0.0336 s	0.0554 s	0.0686 s	0.0724 s
Theoretical time/voxel	1.25E-11 s	1.25E-11 s	1.25E-11 s	1.25E-11 s	1.25E-11 s	1.25E-11 s
Measured time/voxel	5.19E-11 s	6.09E-11 s	4.68E-09 s	1.94E-11 s	3.07E-11 s	2.78E-11 s
Efficiency	24.1%	20.5%	0.267%	64.4%	40.7%	45.0%

Table 5.1.: GPU bandwidth efficiency for kernels in both PDU and TV-GD approaches tested on the AMD FirePro W8100 GPU.

test data size on the AMD FirePro W8100 GPU. The theoretical time is obtained based on the theoretical GPU bandwidth. The GPU bandwidth is calculated by the division of theoretical time for updating a voxel and measured time for updating a voxel. The results show that the kernel $ReduceSum$ reaches very low efficiency, and it may be the performance bottleneck for the whole algorithm. Depending on the type of the kernel, different kinds of optimization methods can be used, which will be explained in the next section.

5.2. Kernel Optimization

Kernel optimization aims to maximize the utilization of GPU compute power and memory bandwidth in order to make computations faster and more efficient. Efficient kernel optimization is essential to fully utilize the capabilities of the GPU and achieve significant speedup over traditional CPU-based computations. Optimizing compute-bound and memory-bound kernels in OpenCL needs to consider the performance bottlenecks of each type of kernel [32].

For memory-bound kernels, there are various possible ways to overcome the bottlenecks:

- Memory Access Coalescing: access memory in a contiguous and aligned manner to enable GPUs to efficiently transfer data between global memory and processing units [33]. Exploit shared memory or local memory to cache frequently accessed data or intermediate results within the kernel. Minimize redundant memory accesses by reusing data stored in faster memory spaces [34]. Memory bandwidth can be better utilised by choosing the size of work items and workgroups wisely.
- Vectorization: use SIMD instructions to process multiple data elements in a single instruction, reducing memory access overhead.
- Loop Unrolling: unroll loops within the kernel to reduce loop overhead and increase instruction-level parallelism. Unrolling eliminates loop control flow and enables the compiler to better schedule instructions for execution. Kernel fusion is similar to loop fusion, but it aims to optimize the execution of multiple kernels.

For compute-bound kernels, there are also some methods to improve performance:

- Parallelism Maximization: ensure that the kernel effectively utilizes parallelism to leverage the computational power of the device. This includes optimizing work-item or work-group sizes, maximizing occupancy, minimizing thread divergence, and exploiting Single Instruction, Multiple Data (SIMD) capabilities [35].
- Memory Localization: consider employing data or loop transformations to enhance data locality, reducing memory access overhead. Techniques such as loop tiling [36] or blocking can improve data reuse and reduce memory access latency.

Based on the results of kernel analysis and the kernel optimization schemes, we will introduce the details and examples of optimizing the kernels in the following sections.

5.2.1. Choice of Work-item and Work-group

In OpenCL, a work-item refers to an individual task or operation that is executed in parallel. Work-items are executed by GPU cores, and each core can handle multiple work-items simultaneously. Each work-item is identified by a unique global ID. Work-items are organized into work-groups. Each work-group contains a set of work-items. Work-items within a work-group can synchronize and communicate with each other by shared memory. Work-groups may have up to three dimensions which can be advantageous when the workload naturally maps to a 2D or 3D domain [37], such as the image processing task.

To reduce the overhead of maintaining a work-group, the size of work-groups should be as large as possible. Each GPU has the upper bound for work-group size, which is typically specified by the GPU manufacturer and can be obtained by $clGetDeviceInfo()$ function with the $CL\_DEVICE\_MAX\_WORK\_GROUP\_SIZE$ parameter. On our NVIDIA RTX 3080 TURBO GPUs, the maximum allowed amount of work-items in one work-group is 1024, whereas the limit is 256 on AMD FirePro W8100 GPUs. When neighboring work-items access consecutive memory locations in one work-group, it results in a memory access coalescing, and this allows a chunk of work-items access contiguous , which may maximize memory bandwidth.

In our code, the 3D image is stored as a flattened 1D array, and the linear index of the voxel is calculated based on the coordinates (x, y, z) using the formula $z \times rows \times cols + y \times cols + x$, where $rows$, $cols$, and $depth$ represent the sizes of the 3D image volume in the x, y, and z directions respectively. The coordinates are represented by the global ID of work-items in each direction.

In order to maximize memory bandwidth, the decision of work-group sizes should be matched to the layout of the data in order to improve memory coalescing and cache utilization [38]. Take the kernel $UpdateImg$ as an example, and fix the maximum work-group size as 256 (which is the limit of our AMD FirePro W8100 GPUs), the kernel execution time with different work-group sizes tested on the NVIDIA RTX 3080 TURBO GPUs are documented in Table 5.2. The experiments were tested on the dataset with a size of 3.6GB. As shown, a well-chosen work-group size can result in a performance improvement of up to about 2.48x compared to other choices. If there are more work-items in the z direction, the execution will take longer as the linear index in the work-group will be sparse in the layout of the data, and neighboring work-items access non-consecutive memory. Data locality is then lost, and memory latency becomes higher. In our implementation, we have chosen

Execution time	Work-group size in x	Work-group size in y	Work-group size in z
0.07736 s	2	2	64
0.03529 s	8	8	4
0.03234 s	16	16	1
0.03121 s	256	1	1

Table 5.2.: Example of kernel execution time using different 3D work-group sizes on the NVIDIA RTX 3080 TURBO GPU.

work-group sizes as (16, 16, 1) along the x, y, and z directions. This selection has proven to be effective and suitable for all the test data we have employed.

5.2.2. Vectorization Scheme

Vectorization in OpenCL refers to the utilization of vector data types and operations to perform SIMD processing on multiple elements simultaneously using a single instruction [39]. OpenCL has vector versions for each fundamental data type, such as float2, float4, float8, etc., which can hold multiple elements in the same data type. For example, by using float4, four float elements can be processed in parallel, which reduces the overall execution time. There are also lots of built-in vector operations in OpenCL, like load and store operations. These operations, such as vload2, vload4, vstore2, vstore4, etc., can load and store multiple elements from or to memory for vector data types. This makes memory access between memory and vector registers efficient. The availability of vectorization depends on hardware support or compiler optimizations for SIMD instructions. Figure 5.2 shows an example of an OpenCL kernel before and after applying SIMD by utilizing vector data types and operations. As in both of our workstations, the vectorization is tested to be auto-implemented by our experiments. In the real implementation, the OpenCL code can benefit from implicit vectorization without writing vector operations. The performance benefit from the vectorization might be lower for the kernels that include a complex control flow.

5.2.3. Kernel Fusion

Kernel fusion [40] is analogous to loop fusion. However, there is a crucial difference between kernels and loops since kernels also allow synchronization at the device memory level. If two threads within a thread block need to synchronize, they can use shared memory and a barrier to do so. When two threads in different thread blocks need to synchronize, this is only possible with different kernel launches acting as a barrier. Combining multiple kernel operations into a single kernel reduces the need to store intermediate results in memory. This minimizes memory reads and writes and reduces data transfer between the CPU and GPU or between different memory levels, which can significantly improve overall performance. Kernel Fusion also reduces the number of kernel launches, leading to lower overhead and improved efficiency.

For instance, in our implementation of TV-GD, the kernel $UpdateImg$ performs the

```
__kernel void updateImg(__global float *f, __global const float *u, float sum,
    float lambda, int depth, int rows, int cols){
  int x = get_global_id(0);
  int y = get_global_id(1);
  int z = get_global_id(2);
  int idx = z * rows * cols + y * cols + x;
  if (x >= cols || y >= rows || z >= depth)
    return;

  f[idx] = f[idx] - u[idx] * lambda / sum;
  return;
}
```

(a) OpenCL kernel example

```
__kernel void updateImg(__global float *f, __global const float *u, float sum,
    float lambda, int depth, int rows, int cols){
  int x = get_global_id(0);
  int y = get_global_id(1);
  int z = get_global_id(2);
  int simd_x = 4 * x;
  int idx = z * rows * cols + y * cols + simd_x;
  if (simd_x >= cols || y >= rows || z >= depth)
    return;

  float4 f_val = vload4(idx, f);
  float4 u_val = vload4(idx, u);

  f_val = f_val - u_val * lambda_vec / sum_vec;

  vstore4(f_val, idx, f);
}
```

(b) OpenCL kernel example after adding vectorization

Figure 5.2.: OpenCL kernel example before and after adding vectorization.

Kernel name	Execution time
$Division$	0.02476 s
$Multiplication$	0.02504 s
$Subtraction$	0.03225 s
$UpdateImg$	0.03234 s

Table 5.3.: Example of kernel execution time before and after kernel fusion on the NVIDIA RTX 3080 TURBO.

operations in both line 3 and line 4 of the algorithm shown in Algorithm 2. The division, multiplication, and subtraction are fused in the kernel, as the operations are done on the same data. We compare the execution time of the kernels $Division$, $Multiplication$, and $Subtraction$ before applying kernel fusion and of the kernel $UpdateImg$ on the dataset with a size of 3.6GB, as documented in Table 5.3. The experiments were conducted on the NVIDIA RTX 3080 TURBO GPU. The total execution time of the kernels $Division$, $Multiplication$, and $Subtraction$ is 0.08214 s, and the kernel $UpdateImg$ achieves a speedup of about 2.55x after kernel fusion.

5.2.4. Synchronization and Shared Memory Scheme

Synchronization in OpenCL can only occur: between work-items in a single work-group or among commands enqueued to command queues in a single context. Here we only focus on the first type of synchronization as it relates to the performance of kernels [30]. Synchronization with a work-group means the coordination of work-items to ensure correct execution of parallel computations. Excessive or unnecessary synchronization can lead to performance bottlenecks and reduced parallelism. Work-items will wait at the synchronization point until all of them complete their tasks, so some of working-items just waste time waiting and thus stall execution.

Synchronization can be performed via either a memory fence or barrier function. The difference between these two is that a barrier forces that all the work-items stop at the barrier while a memory fence only requires that loads and stores before the memory fence are committed. With careful use of the memory fence, performance can be greatly increased as this synchronization keeps work-items active as long as possible.

Shared memory is shared among work-items within a work-group and provides a high bandwidth and low latency memory that can be used to accelerate data sharing and communication between work-items. Especially, the shared memory can be used for parallel reduction operations, where work-items produce partial results to a shared memory buffer that is later combined to produce a final result.

Reduction in OpenCL is used to efficiently compute the sum, minimum, maximum, or any other associative operation of a large array or buffer of data. The reduction process involves iteratively combining elements of the input data to produce a single result. The reduction algorithm divides the input data into smaller chunks and performs partial reductions on each chunk in parallel. The partial results are then combined in subsequent iterations until the final reduction result is obtained. The process continues until the data is reduced to a single value.

In our implementation, the reduction is performed in two kernels of TV-GD named $ReduceNorm$ and $ReduceSum$, which is also presented in line 2 and line 3 of Algorithm 2. The gradient of the TV norm is computed element-wise by the equation (3.8), and then reduced to compute the sum of the L2-norm of the gradient of the TV norm across the whole volume. To obtain good performance, we optimize the reduction kernel regarding memory coalescing and also loop unrolling, as shown in Figures 5.3 and 5.4. In the inner loop of the original reduction kernel of Figure 5.3, the index is computed as $2 * s * tid$. As s is increased, the addressing will then be interleaved, so sequential addressing is needed to make sure that memory is coalescing. Since all warps execute every iteration of the for loop and if statement, applying loop unrolling for the last warp can save useless work in all warps, which is shown in Figure 5.4.

5.3. GPU Scaling

It is important that the hardware can provide greater compute power when the amount of resources is increased. This characteristic is also called scaling. There are two kinds of scaling, depending on the application itself. Strong scaling [41] means the number of processors is increased while the problem size remains constant, while in the case of weak scaling, both the number of processors and the problem size are increased. Lots of factors may influence GPU scalability:

- Workload Parallelism: sufficient parallelism make sure the workload is effectively distributed among multiple GPUs.
- Communication and Synchronization Overhead: communication and synchronization between multiple GPUs can hinder scalability by introducing overhead. To obtain good scalability, the frequency of communication and synchronization between GPUs should be minimized.
- Data Partitioning and Memory Management: data should be appropriately split and distributed among GPUs, and the frequency of transferring memory should be minimized to maximize memory locality.

In OpenCL, a context represents the execution environment that encapsulates devices, memory objects, and other resources necessary for executing OpenCL kernels. It manages the sharing of resources, memory allocation, and synchronization across devices. The command queue schedules commands for execution on a device. Each device in the context can have one or more associated command queues. Commands enqueued in a command queue are executed in the order they are enqueued.

5.3.1. Multiple Command Queues within a Single Device

Multiple command queues associated with a single device allow for parallel execution of commands and efficient utilization of the available resources [42]. This feature enables asynchronous execution, which can overlap data transfers, computation, and other operations, resulting in better overall performance. But it also has some limitations, such as increased complexity and increased overhead. Take an example in PDU, in the forward

```
unsigned int tid = get_local_id(0);
unsigned int i = get_global_id(0);
sdata[tid] = (i < n) ? g_idata[i] : 0;
barrier(CLK_LOCAL_MEM_FENCE);

for(unsigned int s=1; s < get_local_size(0); s *= 2)
{
    int index = 2 * s * tid;
    if (index < get_local_size(0))
    {
        sdata[index] += sdata[index + s];
    }
    barrier(CLK_LOCAL_MEM_FENCE);
}
if (tid == 0) g_odata[get_group_id(0)] = sdata[0];
```

(a) Original reduction kernel

```
unsigned int tid = get_local_id(0);
unsigned int i = get_group_id(0)*(get_local_size(0)*2) + get_local_id(0);
sdata[tid] = (i < n) ? g_idata[i] : 0;
if (i + get_local_size(0) < n)
    sdata[tid] += g_idata[i+get_local_size(0)];
barrier(CLK_LOCAL_MEM_FENCE);

for(unsigned int s=get_local_size(0)/2; s>0; s>>=1)
{
    if (tid < s)
    {
        sdata[tid] += sdata[tid + s];
    }
    barrier(CLK_LOCAL_MEM_FENCE);
}

if (tid == 0) g_odata[get_group_id(0)] = sdata[0];
```

(b) Reduction kernel with memory coalescing

Figure 5.3.: OpenCL reduction kernel with memory coalescing.

```
unsigned int tid = get_local_id(0);
unsigned int i = get_group_id(0)*(get_local_size(0)*2) + get_local_id(0);
sdata[tid] = (i < n) ? g_idata[i] : 0;
if (i + blockSize < n)
    sdata[tid] += g_idata[i+blockSize];
barrier(CLK_LOCAL_MEM_FENCE);

if (blockSize >= 512) { if (tid < 256) { sdata[tid] += sdata[tid + 256]; }
    barrier(CLK_LOCAL_MEM_FENCE); }
if (blockSize >= 256) { if (tid < 128) { sdata[tid] += sdata[tid + 128]; }
    barrier(CLK_LOCAL_MEM_FENCE); }
if (blockSize >= 128) { if (tid <  64) { sdata[tid] += sdata[tid +  64]; }
    barrier(CLK_LOCAL_MEM_FENCE); }
if (tid < 32)
{
    if (blockSize >=  64) { sdata[tid] += sdata[tid + 32]; }
    if (blockSize >=  32) { sdata[tid] += sdata[tid + 16]; }
    if (blockSize >=  16) { sdata[tid] += sdata[tid +  8]; }
    if (blockSize >=   8) { sdata[tid] += sdata[tid +  4]; }
    if (blockSize >=   4) { sdata[tid] += sdata[tid +  2]; }
    if (blockSize >=   2) { sdata[tid] += sdata[tid +  1]; }
}

if (tid == 0) g_odata[get_group_id(0)] = sdata[0];
```

Figure 5.4.: OpenCL reduction kernel with unrolling.

buffer synchronization part, not only the variable u denoting updated image, but also the dual variables px, py, and pz have to be synchronized. By utilizing multiple command queues here, the data transfers between GPUs can be overlapped, as shown in Figure 5.5. In this case, the data transfer time is reduced, and the parallelism is enhanced by utilizing multiple command queues.

5.3.2. Synchronization between Command Queues

Synchronization between command queues is necessary when there are conditions such as data dependency, resource sharing, and execution correctness. One of the mechanisms in OpenCL to perform synchronization is the $clFlush()$ and $clFinish()$ functions, which provide a brute force capability to flush or wait for all queued commands to complete. $clFinish()$ is a blocking function, and when the function is called, the host program will be blocked until all the previous OpenCL commands in the command queue have completed execution on the device before allowing the program to proceed further. So, $clFinish()$ provides a global synchronization point. On the other hand, $clFlush()$ is a non-blocking function that only enqueues commands in the command queue for execution on the device without waiting for their completion. When $clFlush()$ is called, it will queue all the commands from the specified command queue for execution, but it does not block the host program's execution. The commands are merely sent to the device for processing, but the host program can continue with other tasks immediately. It does not guarantee that the commands have completed execution.

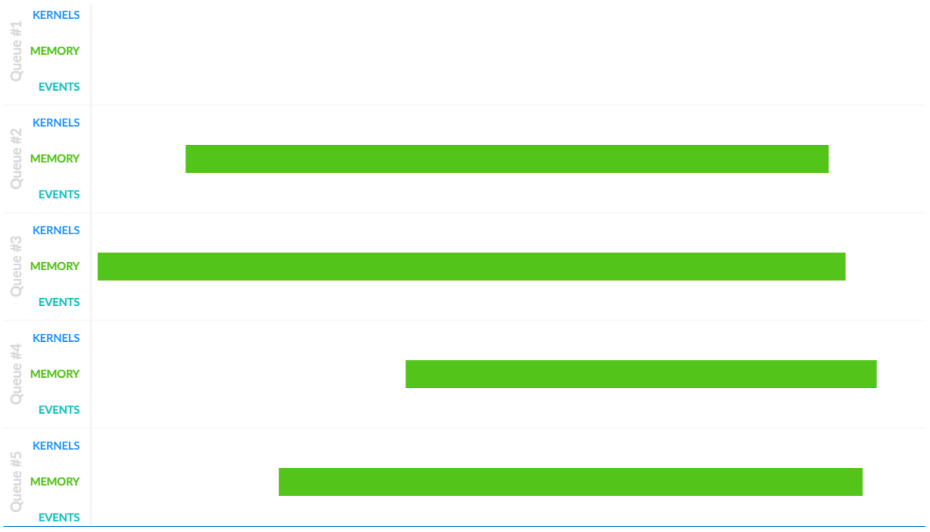
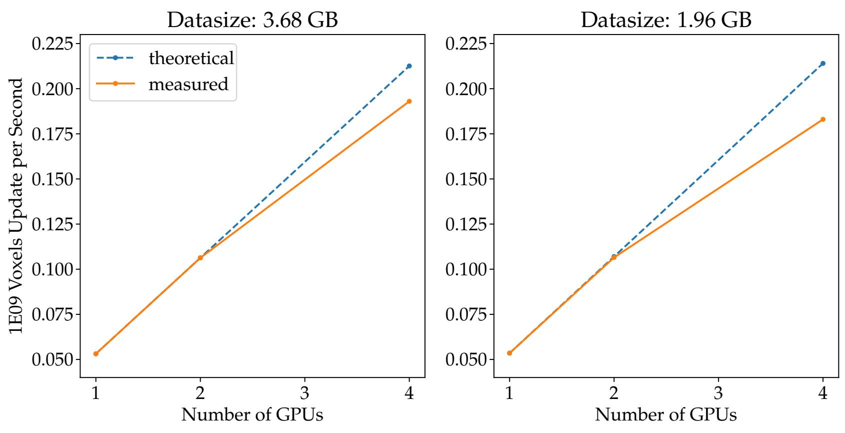


Figure 5.5.: Example of multiple command queues in one device.

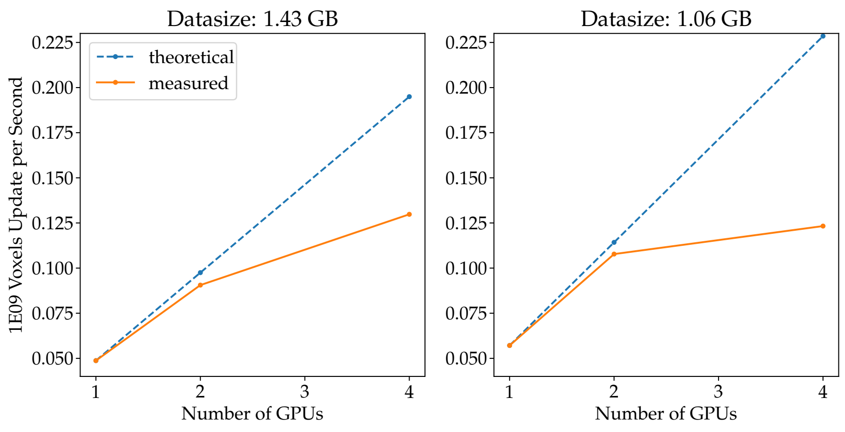
Synchronization can also be enabled by coordinating events [43]. An event in OpenCL serves as a synchronization point and provides information about the status and completion of commands enqueued in a command queue. Using the $clWaitForEvents()$ function enables the command to wait for the completion of specific events before proceeding. This function is non-blocking and synchronizes specific commands or kernel executions in the command queue without blocking the entire program, which allows for fine-grained synchronization. By waiting for specific events to complete before launching subsequent commands, the order of execution can be controlled, and data dependencies are properly handled. It also allows the host program to continue with other tasks while waiting for the specified events to complete. This asynchronous behavior can lead to improved overall program performance, as the host can continue to perform useful work during the synchronization process and thus reduce the synchronization overhead.

In our implementation, we have tried to avoid using $clFlush()$ and $clFinish()$ functions and replaced them with events to perform synchronization. For instance, in TV-GD implementation shown in Algorithm 5, the $ReducedNorm$ kernel is executed after the $GradientTV$ kernel. Here, the synchronization between these two kernels can be completely avoided, since they are queued in the same command queue, and the kernels will be executed in order. In the case that before entering the inner loop shown in line 10, memory transfer operations have to be taken, and in this case, the execution of kernels needs to wait until the completion of memory transfer. Waiting events from memory transfer can be assigned to the $GradientTV$ kernel, and this reduces the synchronization overhead and makes our algorithms faster.

Figure 5.6 plots the strong scaling performance of TV-GD and PDU regarding varying data sizes and on up to four GPUs. The experiment was tested on the workstation with four AMD FirePro W8100 GPUs. The y-axis represents the number of elements updated per second. Duplicating the amount of GPUs should also double the number of elements updated per second in theory. To analyze strong scaling performance, the memory needed should be fitted into the memory size of a single GPU. In Figure 5.6a, two data sizes are



(a) Strong scaling performance for TV-GD on up to 4 GPUs

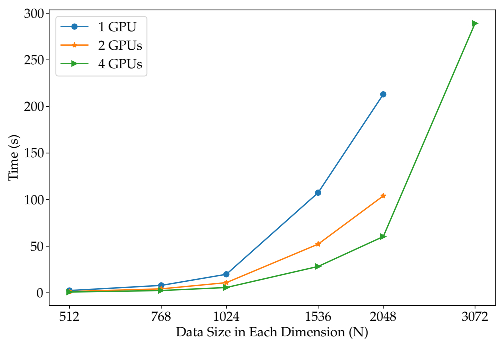


(b) Strong scaling performance for PDU on up to 4 GPUs

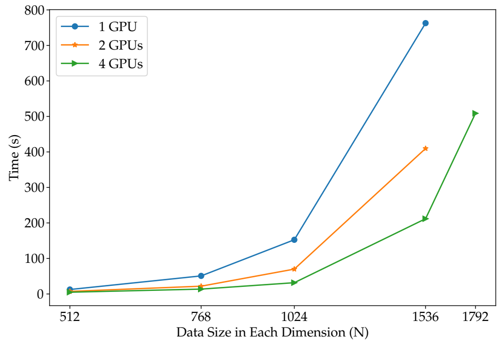
Figure 5.6.: Strong scaling performance for TV-GD and PDU algorithms on up to 4 GPUs.

individually compared on up to four GPUs, and the larger data size we have set to is 3.68 GB for TV-GD, which satisfies 8 GB of the GPU memory size on the AMD FirePro W8100 GPU, as TV-GD needs two copies of the original image. The scaling efficiency tested on larger data size is about 99.98% on two GPUs and 90.79% on four GPUs, and this efficiency shows a good scaling behavior of TV-GD on multi-GPUs. The smaller data size we have set to is 1.96 GB, almost half of the larger one, the scaling efficiency tested on this small data size is about 99.52% on two GPUs and 85.51% on four GPUs, which is a little worse than the larger one, and this is due to the fact that synchronization time is relatively larger for less execution time.

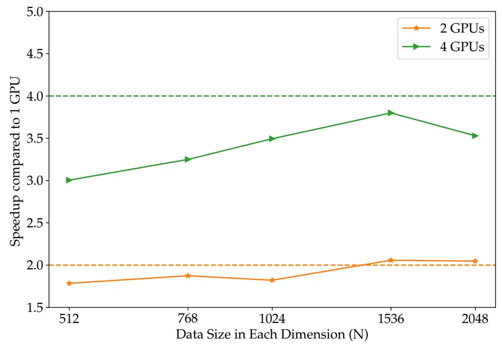
In Figure 5.6b, two different data sizes are set for PDU, and the larger data size is 1.43 GB, which also satisfies the GPU memory size, as the PDU approach needs five copies of the original image. The scaling manner is worse for PDU compared to TV-GD, especially on four GPUs, and the scaling efficiency declines to 66.57% from 92.92% on two GPUs. The reason for this is that the computational load per GPU is too small compared to the required memory transfers, which is the case for the small sizes in this experiment.



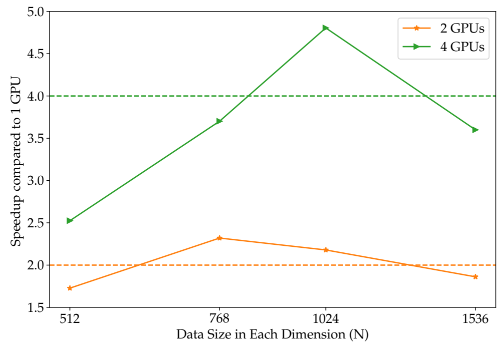
(a) The speeds of TV-GD at different sizes on different number of GPUs.



(b) The speeds of PDU at different sizes on different number of GPUs.



(c) The speedup of TV-GD compared to the 1 GPU execution time at different sizes on different number of GPUs.



(d) The speedup of PDU compared to the 1 GPU execution time at different sizes on different number of GPUs.

Figure 5.7.: The speeds and speedups of TV-GD and PDU at different sizes on different number of GPUs.

Besides the analysis of the GPU scaling performance, Figure 5.7 presents the total time for different sizes (N) and the number of GPUs tested on the workstation with four AMD FirePro W8100 GPUs. The experiments used N^3 image volumes with N data size for each dimension, and the total time includes computation time and memory transfer time. We run TV-GD and PDU with a fixed number of iterations, 50 and 200 respectively, and the reason for choosing these values will be explained in the next chapter. The missing points are caused by insufficient GPU memory for the 4 GPU workstations to satisfy the minimal buffer length we have set.

The kernel is fast enough in the case of small problem sizes so that time is dominated by GPU property checks and memory transfers. The computational kernel is small enough to mask any improvements that multiple GPUs could bring. The computational cost of both operations increases as N increases, so the measured speedup ratios are close to theoretical (2x and 4x for 2 and 4 GPUs, respectively). In general, adding more GPUs brings the speedup ratio closer to the theoretical limit until there are enough GPUs and the computational load of each GPU is too small compared to the required memory transfers, as was the case for the smallest size in this experiment.

However, if the image size is too large, the number of splits increases, and the buffer length decreases, leading to an increase in the frequency of memory transfers. Memory transfers take up more of our execution time. Therefore, the speedup ratio drops again for the largest size.

6. Numerical Results

6.1. Dataset Description

All the datasets employed in our testing consist of 3D images, often referred to as volumes. These volumes are stored in the single precision floating-point format, with each voxel represented as a 32-bit floating-point binary value, equating to four bytes of storage per voxel. For instance, in a typical 2k volume with 2000 pixels in each dimension, the volume size amounts to 32 GB.

These datasets are generated through CT scans followed by CT reconstruction. Different datasets are used for various tests, varying not only in size but also in noise patterns. The initial noisy data results from the reconstruction of projections with limited acquisition time. The SNR of acquired projection data decreases, and this can result in increased noise in the reconstructed image. Depending on whether the original noisy data has corresponding ground truth data [44], the dataset can be split into two main parts:

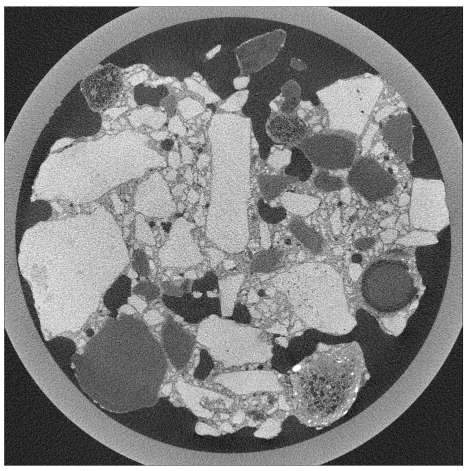
- Full-reference (FR): in full-reference image quality assessment, the quality of a processed image is compared to a high-quality reference image. This reference image is considered to be a distortion-free version of the original image.
- No-reference (NR): in no-reference image quality assessment, the quality of a processed image is evaluated without using a reference image. This is typically achieved by analyzing statistical features or image properties that are affected by the presence of noise or distortion.

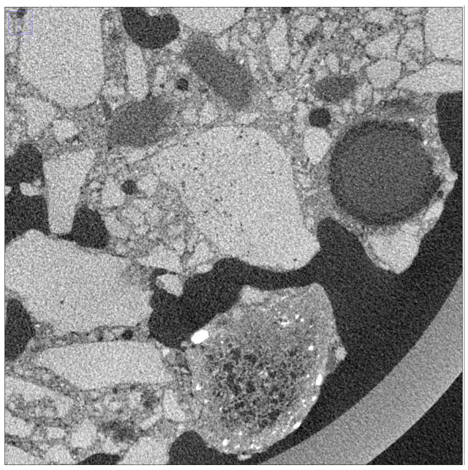
In the case of FR, the ground truth reference is reconstructed from projections with a longer acquisition time, as shown in Figure 6.1, exhibiting less noise and higher quality. These ground truth datasets provide a means of validation and testing. Additionally, there are datasets with varying noise levels across different regions designed for experiments involving statistical denoising.

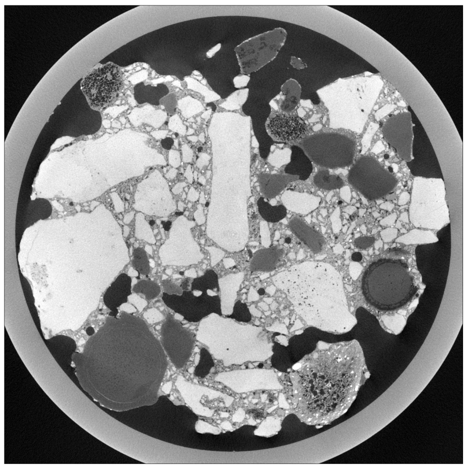
6.2. Convergence Analysis

The objective of convergence analysis is to comprehend how a denoising algorithm approaches the optimal denoised image and the extent to which parameters influence the convergence rate and accuracy of the algorithm.

- Convergence: in certain scenarios, denoising algorithms eventually converge to the optimal solution under specific conditions, while in others, they might not. Convergence criteria are then needed to define when the denoising algorithm should stop iterating.


(a) Noisy image with low acquisition time


(b) Noisy image with magnification


(c) Reference image from a long acquisition time

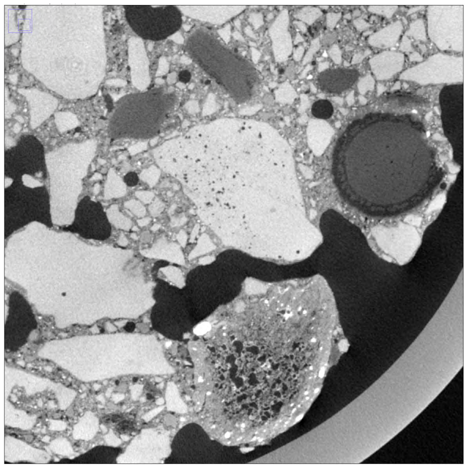

(d) Reference image with magnification

Figure 6.1.: Stone data with noisy image from a low acquisition time and reference image from a long acquisition time.

- Stability: stability is another factor to consider, ensuring that the algorithm consistently produces the same denoised outcome given the same noisy input image and parameter settings.
- Robustness: robustness refers to the ability to generate satisfactory denoised results across diverse noise conditions, encompassing varying noise levels.

In the forthcoming sections, we will delve into the convergence aspects and discuss the stopping criteria applied to PDU and TV-GD algorithms individually.

6.2.1. Convergence of PDU

From the numerical analysis in [9], the PDU algorithm has a converged solution, and the algorithm has to be stopped to avoid unnecessary iterations. It is difficult to decide when to stop as it is hard to determine whether the approximate solution has high quality or not if the ground truth data is not available, so the stopping criteria are hard to choose. [9] proposed reliable and easily calculated criteria named duality gap:

$$\frac{G(u^k, p^k)}{D(p^k)} < TOL, \tag{6.1}$$

where $G(u^k, p^k) = P(u^k) - D(p^k)$ and k denotes the number of iterations. TOL is the pre-specified tolerance threshold. $P(u)$ and $D(p)$ are cost functions for the primal update and the dual update respectively. The functions to compute the energy of primal-dual updates are defined as:

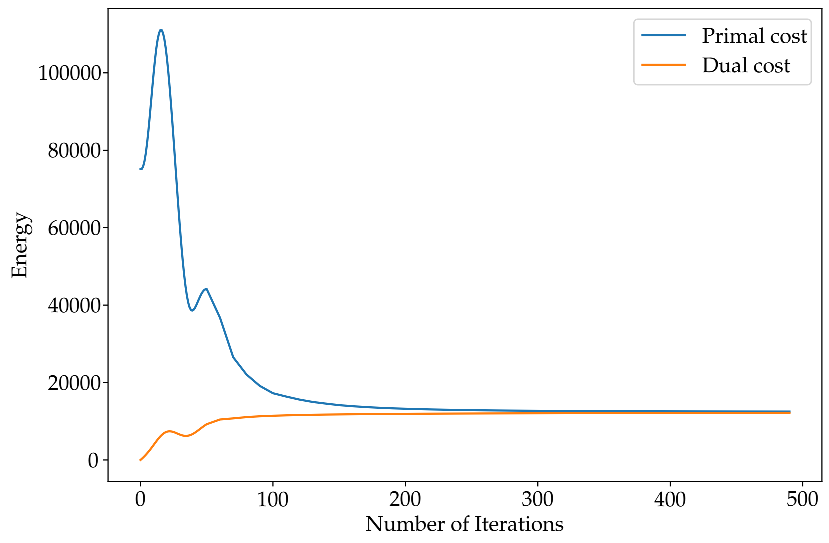
$$P(u) = \frac{1}{2}\|u - f\|^2 + \frac{1}{\alpha}\|\nabla u\|, \tag{6.2}$$

and

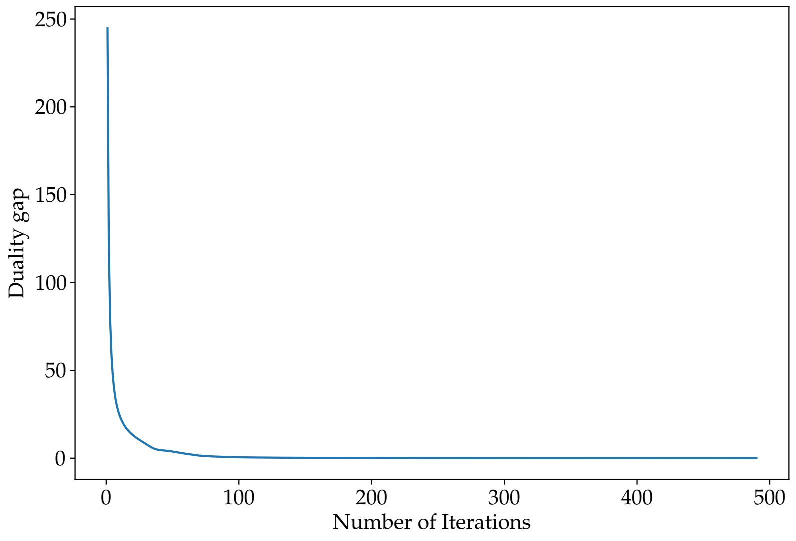
$$D(p) = \frac{1}{2}(\|f\|^2 - \|\frac{1}{\alpha}\nabla \cdot p + f\|^2). \tag{6.3}$$

The duality gap is a measure of the closeness of the primal-dual pair to the primal-dual solution. As the optimization scheme consists of a minimization and a maximization problem, $P(u)$ presents an upper bound, and $D(p)$ presents a lower bound for the true minimum of the ROF model. Figure 6.2 shows a representative example of the convergence behavior for the duality gap with the number of iterations. The energy of the primal update and the dual update is approaching the same, and the duality gap is exponentially decreasing. This figure also indicates that iterating after a certain number of iterations is not necessary. For instance, the duality gap after 200 iterations will decrease exponentially, and denoised results are already good before this point.

For cases where ground truth data exists as a reference, ground truth data can be utilized to validate the convergence, and Figure 6.3 shows the error of the updated data to the ground truth data with the number of iterations. The error of the updated data to the ground truth data is computed as the L2 Norm. The error initially increases as the dual variables are initialized as zeros, which are far from the optimal solution. Subsequently, the error starts decreasing and eventually converges to a specific value. The outcomes from the analysis affirm that PDU consistently converges after a certain number of iterations. This validates the convergence of the PDU algorithm with the chosen step sizes.



(a) Evolution of the primal and dual cost regarding the number of iterations.



(b) Evolution of the duality gap regarding the number of iterations.

Figure 6.2.: Evolution of the primal cost, the dual cost and the duality gap regarding the number of iterations.

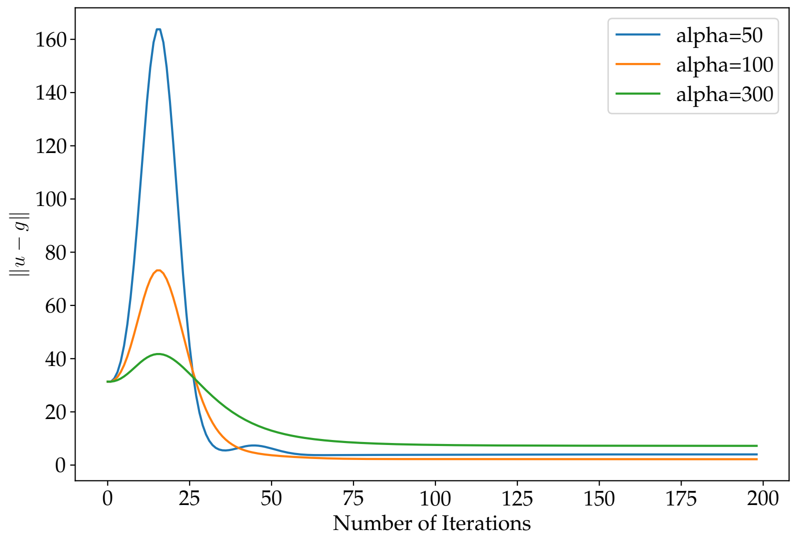


Figure 6.3.: Error to the ground truth data regarding the number of iterations for different parameters in PDU.

6.2.2. Convergence of TV-GD

When ground truth data is available as a reference, the error of updated data to the ground truth data can be used to investigate the numerical behavior of the algorithm. Figure 6.4 shows the L2-norm error of updated data to the ground truth data with the number of iterations. The error will first go down to an optimal minimum error and then go up as the image will be overly smoothed without the data fidelity constraint. After this point, TV regularization is too strong. This diverging behavior corresponds to the Algorithm 2, with a very large of iterations, the norm of TV will be minimized to a point, which means the amount of total variation is minimized, and the image will become too flat and far away from the true image.

However, in general cases, there is no reference data, so it is hard to find the optimal solution for TV-GD. In addition, it is also complicated to predefine a tolerance for the data fidelity term as it depends on volume size, noise level, desired image quality, etc.

The problem of TV-GD is that this algorithm can easily over smooth the noisy image with strong TV minimization. In order to ensure that the result is an optimal estimate, similar to that in the ASD-POCS implementation [17], the associative Karush-Kuhn-Tucke (KKT) condition should be satisfied. The KKT condition is tested with an indicator, which is defined as:

$$C_\alpha = \frac{\overrightarrow{d_{TV}} \cdot \overrightarrow{d_{data}}}{\|\overrightarrow{d_{TV}}\| \cdot \|\overrightarrow{d_{data}}\|}, \tag{6.4}$$

where $\overrightarrow{d_{TV}}$ is a vector of the derivative of the TV term, and $\overrightarrow{d_{data}}$ is a vector of the derivative of the data constraints using a Lagrangian multiplier. C_α represents the cosine of the angle

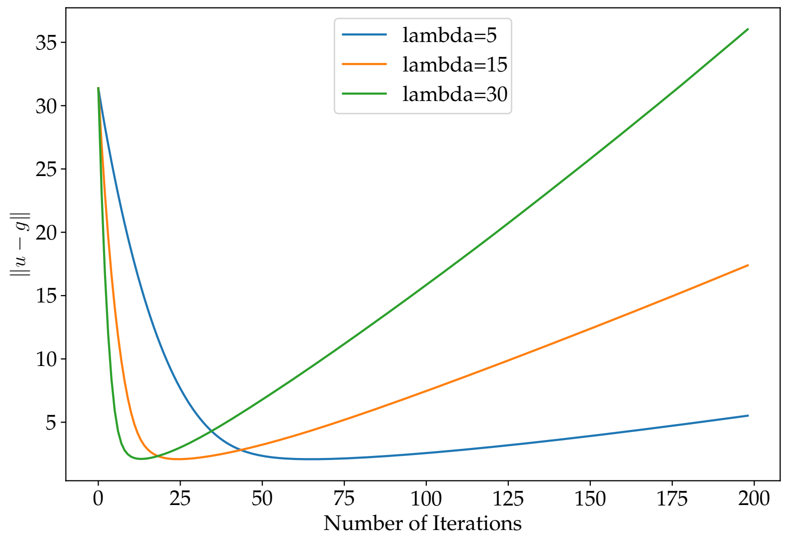


Figure 6.4.: Error to the ground truth data regarding the number of iterations for different parameters in TV-GD.

between these two vectors. As reported in [17], $C_\alpha = -1.0$ is a necessary condition for TV minimization to reach an optimal solution with sufficient data constraints. These two vectors point in exactly opposite directions, but this condition requires a large number of iterations and may not be practical. Typically, the stopping criteria are chosen as $C_\alpha < -0.5$ or $C_\alpha < -0.6$ in real applications shown in [18]. In our experiments, we found that the stopping criterion of the ASD-POCS algorithm does not work in TV-GD because the constraint of the data fidelity term is missing, and the convergence endpoint is not an optimal solution.

The TV-GD algorithm will also converge to a specific limit in the end with a larger number of iterations, but this algorithm is not converging to the optimal solution, as shown in Figure 6.4. There still exists an optimal solution before over-smoothing. It is important to analyze the smoothing property, which is related to not only the number of iterations but also the predefined parameters, to avoid under-smoothing and over-smoothing. The parameter selection scheme will be introduced in the next section.

6.3. Parameter Selection Scheme

Over-smoothing and under-smoothing are two common issues that can arise when applying denoising algorithms. Both situations have distinct effects on the denoised image, and finding the right balance is crucial for achieving optimal denoising results. Over-smoothing occurs when the denoising algorithm removes not only the noise but also some of the finer image details and edges. The denoised image appears overly blurry or excessively smoothed, resulting in a loss of important information. This is often caused by using denoising algorithms with aggressive filtering or large smoothing kernels. In some cases, over-smoothing can even introduce some artifacts, such as halos and rings. Under-smoothing occurs when the denoising algorithm does not effectively remove the noise, resulting in a denoised image that still contains noticeable noise artifacts [45].

As shown in Figure 6.5, the TV-GD algorithm is tested on the $Stone$ dataset, and the right column is obtained by magnifying the lower right part of the volume from the left column. The $Stone$ dataset contains lots of small structures and consists of various materials with different gray values. Figures 6.5a and 6.5b in the first row are the denoised images with weak smoothing strength, and from the figures, visible noise still can be noticed in the volume, and the volume is under-smoothing. Figures 6.5e and 6.5f in the third row are obtained with strong smoothing strength, where the noise is almost removed, but the volume is overly blurry and loses some important details. For instance, the sand between the stones can not be distinguished anymore, the region becomes flat, and the volume is over-smoothing. In the middle row, Figures 6.5c and 6.5d show the denoised image with median smoothing strength, and in this case, the noise can be effectively removed, and the vital details can be kept.

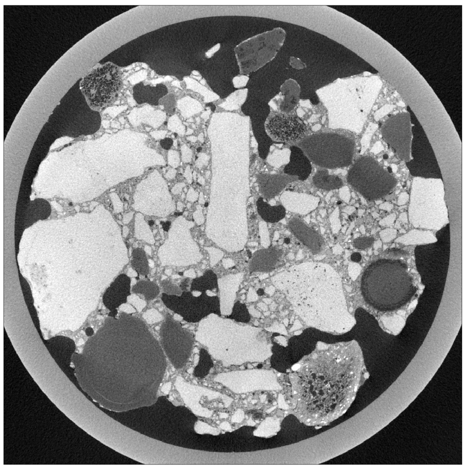
The key to effective image denoising is finding the right balance between over-smoothing and under-smoothing. Different denoising algorithms have various parameters that control the level of smoothing. By tuning these parameters and selecting appropriate denoising methods, it is possible to achieve an optimal denoising result that preserves important image details while effectively reducing noise [46].

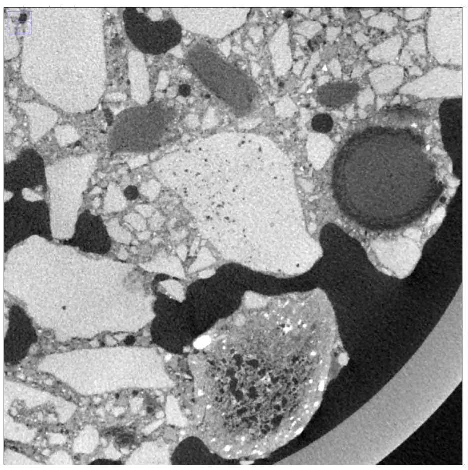
6.3.1. Smoothing Strength of PDU

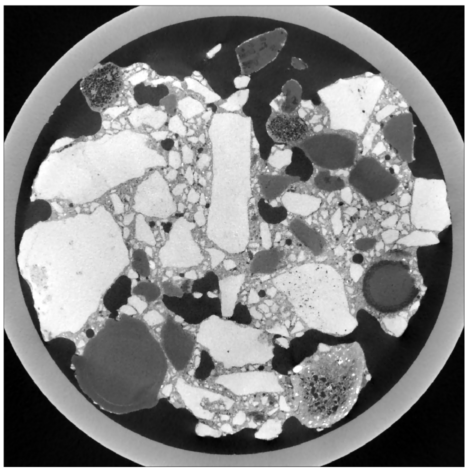
Smoothing strength in the task of image processing refers to the level of smoothing applied to an image during a denoising or filtering process. It represents how much the noise in the image is suppressed or reduced to obtain a smoother version of the original data. Smoothing strength is often controlled by adjusting parameters that control the amount of noise removal or blurring. In the context of denoising, increasing the smoothing strength results in a more aggressive noise reduction, which may lead to loss of fine details or edges in the data, and the image is over-smoothing. On the other hand, reducing the smoothing strength may retain more details but may also allow some noise to persist in the denoised result, and the result is under-smoothing.

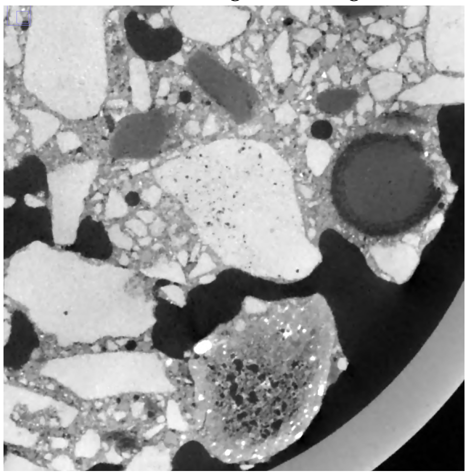
For PDU, the parameter α controls the amount of noise removal and the smoothing strength. With different parameters, the convergence rate will differ, and the converging point will change as well accordingly, as shown in Figure 6.6. Here, the L2-norm of the data fidelity term is chosen as a reference with the number of iterations for different values of α. A larger L2-norm of the data fidelity term means there is more difference between the original image and the updated image.

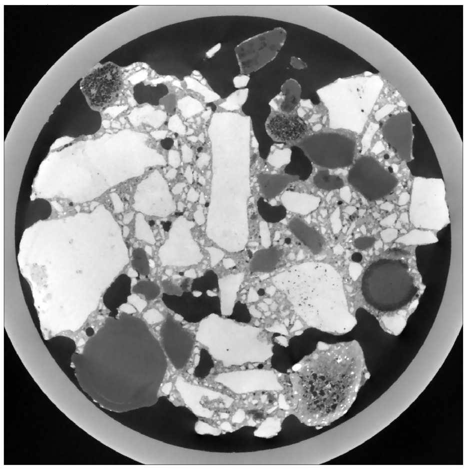
According to Equation (2.4), the larger the alpha is, the more weight the data fidelity term has. Consequently, the difference between updated volume and noisy volume will be smaller at the end after convergence. It also means less weight the TV regularization has, the weaker TV regularization will be after convergence, and the algorithm has weak smoothing strength. Vice versa, with a smaller alpha, the algorithm has stronger smoothing strength. The smoothing strength is determined by the parameter α. In PDU, the selection of τ_P^k and τ_D^k determines the step size of the primal update and the duality update, controlling the convergence rate and smoothing rate of the algorithm. In our implementation, the


(a) Under-smoothing


(b) Under-smoothing with magnification


(c) Optimal-smoothing


(d) Optimal-smoothing with magnification


(e) Over-smoothing

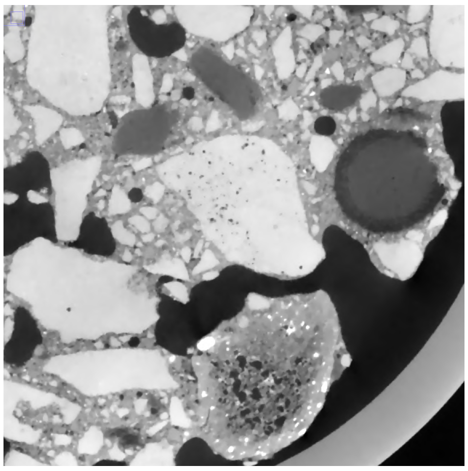

(f) Over-smoothing with magnification

Figure 6.5.: Under and over-smoothing tested on the $Stone$ dataset.

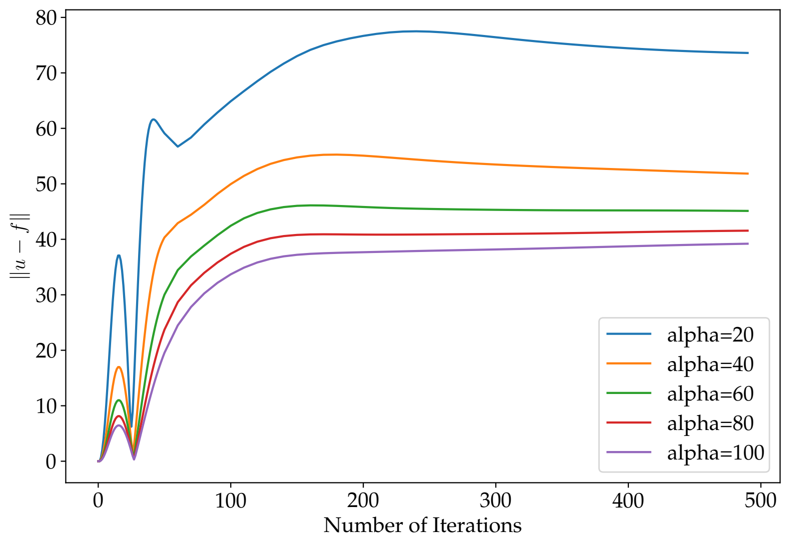


Figure 6.6.: L2-norm of data fidelity regarding the number of iterations with different parameters in PDU.

number of iterations is set as a fixed number between 200 and 500 which has shown good experimental results. The α is the only parameter that has to be tuned for various datasets.

Concerning the variant of PDU known as S-PDU, the sole distinction lies in the data fidelity term. In statistical PDU, this term is weighted element-wise based on the noise level at each voxel. Through our experiments, we have observed that the convergence and smoothing behavior of statistical PDU closely align with those of the original PDU. Therefore, we have opted not to present a separate discussion of numerical results for statistical PDU in this context.

6.3.2. Smoothing Strength of TV-GD

For TV-GD, the number of iterations controls the smoothing strength. However, there exists no data fidelity constraint for TV-GD, so TV-GD will smooth the volume continuously until the volume reaches the minimization of TV regularization and become flat. The corresponding data fidelity term will grow up endlessly, as shown in Figure 6.7. With a larger λ, the smoothing procedure will be faster, and the λ controls the step size of the smoothing procedure shown in Algorithm 2. As the number of iterations increases, the volume will first become under-smoothing and then reach an optimal-smoothing point where the important details of volumes are preserved while removing the noise. Afterward, the volume will be over-smoothing with more iterations, and the important details may be lost.

The smoothing speed is determined by the parameter λ, and a larger λ means a faster smoothing rate. Here, we have to be careful that the $lambda$ can not be too high, as the algorithm will then have unstable behavior. In detail, there will be a high oscillation at the initial iterations. The smoothing strength is decided by the number of iterations, and

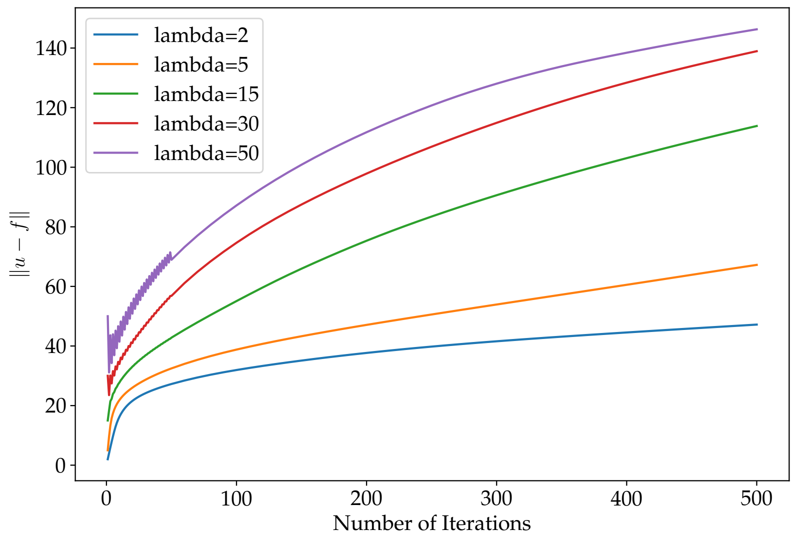


Figure 6.7.: L2-norm of data fidelity regarding the number of iterations with different parameters in TV-GD.

with more number of iterations, the update image will first be under-smoothing and then be over-smoothing. In our real implementation, the number of iterations is set as a fixed number, and the optimal solution can be obtained by tuning the λ. The number between 50 and 80 presents a satisfactory result in our experiments for the iterations.

An additional variant of POCS known as AW-POCS introduces a subtle alteration to the computation of the gradient element. In AW-POCS, the gradient of the TV norm is weighted by an exponential function. Our experimental findings correspondingly indicate that AW-POCS shares similar convergence behavior and smoothing characteristics with the standard POCS algorithm. The discussion of numerical results for AW-POCS is not presented here.

6.4. Image Quality

Apart from analyzing the numerical behavior of the algorithms, the quality of the denoised image is essential to be assessed to judge if the algorithm performs well or not. In the next section, the means and metrics of evaluating image quality will be introduced, and the algorithms will be compared together regarding the denoising performance.

In order to distinguish whether the image denoising is good or not, subjective evaluation by visual inspection is a common method to assess image quality. Experts visually examine the 3D volume and provide qualitative feedback based on factors like clarity, artifacts, noise, and overall image fidelity. This is not a good way to compare the performance among different datasets, algorithms, and image processing methods. Therefore, a quantitative evaluation is needed. In order to get quantitative results, several measurements can be chosen regarding specific requirements and settings. In the case of FR, the ground truth data is provided for measuring the image quality, offering a better estimation. In our case, ground truth data is retrieved from the reconstruction of projections after a long time scanning with high quality.

There are various methods to measure the similarity between estimated images and ground truth in the case of FR, which focuses on the assessment of the quality of a test image in comparison with a reference image [47].

- Mean Square Error (MSE) is one of the most traditional estimators. MSE measures the average of the squares of the difference between the estimated values and the actual value [48]. Root Mean Square Error (RMSE) is the square root of MSE and provides a measure of the average difference between denoised data and ground truth. MSE can be meaningless if the peak intensity is uncertain.
- Peak Signal to Noise Ratio (PSNR) is also a popular measure of image quality. PSNR is an expression for the ratio between the maximum possible power of a signal and the power of corrupting noise that affects the quality of its representation. PSNR has the advantage of enabling to compare results on images with different peak intensities compared to MSE. Generally speaking, When the PSNR value is high, it implies a better similarity and higher image quality between the denoised images and ground truth. A high PSNR value indicates that the denoised image has less distortion compared to the ground truth. PSNR is usually defined as [49]:

$$PSNR = 10\log_{10}\frac{MAX^2}{MSE}, \tag{6.5}$$

  where MAX represents the max voxel value of this volume.
- Structural Similarity Index (SSIM) [50] provides a more meaningful evaluation by taking human visual characteristics into account. It measures the structural similarity between two images by comparing luminance, contrast, and structural information. The SSIM is computed as:

$$\begin{aligned} SSIM(x,y) &= Luminance\ function + Contrast\ function + Structure\ function \\ &= [l(x,y)]^{\alpha} + [c(x,y)]^{\beta} + [s(x,y)]^{\gamma} \\ &= \frac{2\mu_x\mu_y + C_1}{\mu_x^2\mu_y^2 + C_1} + \frac{2\sigma_x\sigma_y + C_2}{\sigma_x^2\sigma_y^2 + C_2} + \frac{\sigma_{xy} + C_3}{\sigma_x\sigma_y + C_3} \\ &= \frac{(2\mu_x\mu_y + C_1)(2\sigma_{xy} + C_2)}{(\mu_x^2 + \mu_y^2 + C_1)(\sigma_x^2\sigma_y^2 + C_2)}, \end{aligned} \tag{6.6}$$

  where x and y represent two images, and the operator μ is the pixel sample mean. σ is the standard deviation, whereas σ_{xy} is the cross-correlation of x and y. C is a constant influenced by the dynamic range of pixel values. In our work, we have set $C_1 = 0.01^2$ and $C_2 = 0.03^2$, and the image values are normalized between 0 and 1 before evaluation. The SSIM index ranges between 0 and 1, with higher values indicating better similarity. The drawback of SSIM is that the computation is more complex and takes a longer time compared to MSE and PSNR.

However, the problem is that the alignment between ground truth data and noisy data is not perfect even after lots of alignment operations. In this case, quantitative results for measuring image quality are hard to achieve good results as the metrics are very sensitive to geometrical issues such as alignment and scaling. Besides, ground truth data still contains noises itself, but the noise level is much lower compared to the noisy data. Thus, the computation of quantitative results will be affected more or less. In addition, there are no ground truth data available as a reference sometimes. SNR is an option to measure the image quality in this case of NR, which means no reference image is used, and the metrics focus on the assessment of the quality of a test image only. SNR can be calculated using different formulas, and it depends on how signal and noise are determined. One way to define SNR is [51]:

$$SNR = \frac{\mu}{\sigma}, \tag{6.7}$$

where μ is the mean of the signal and σ is the standard deviation of the noise, and SNR is usually calculated as the ratio of the mean pixel value to the standard deviation of the pixel values over a given neighborhood. The definition is only valid if the variables are always non-negative. In our real computation, values of volume may need an offset to make sure that the values of each voxel are non-negative.

In the case that the ground truth is available, the MSE, PSNR, and SSIM indexes can be utilized as a quantitative interpretation of image quality. SNR can be used as the index in the case that the ground truth is not available. In the following sections, the PDU and TV-GD, PDU and S-PDU are compared separately regarding the quality of the denoised image.

6.5. Comparison of PDU and TV-GD

It is difficult to compare PDU and TV-GD in a fair way when there exists no ground truth reference data, as two algorithms solve different mathematical problems. PDU solves the unconstrained problem, and TV-GD solves the constrained problem. It is meaningless to directly compare the number of iterations or parameters. The data fidelity term can always be easily obtained even if the ground truth is not available and indicates how far the update volume from the noisy volume is. As shown in Figure 6.6, PDU will always converge to a stable data fidelity term. This certain value could also be used as a reference for TV-GD. So, we propose a comparison scheme for the NR condition as follows:

1. Fix the parameter α in PDU, and first run PDU until it converges to a stable data fidelity term.
2. Set this certain data fidelity term as a reference for TV-GD, and run TV-GD until the data fidelity term reaches the predefined value.
3. Compare the updated images obtained from PDU and TV-GD.

Figure 6.8 shows a representative example of how we compare TV-GD and PDU, and the data fidelity term is obtained from PDU after convergence. TV-GD runs until it reaches the predefined data fidelity term. The runtime and also the corresponding image quality are

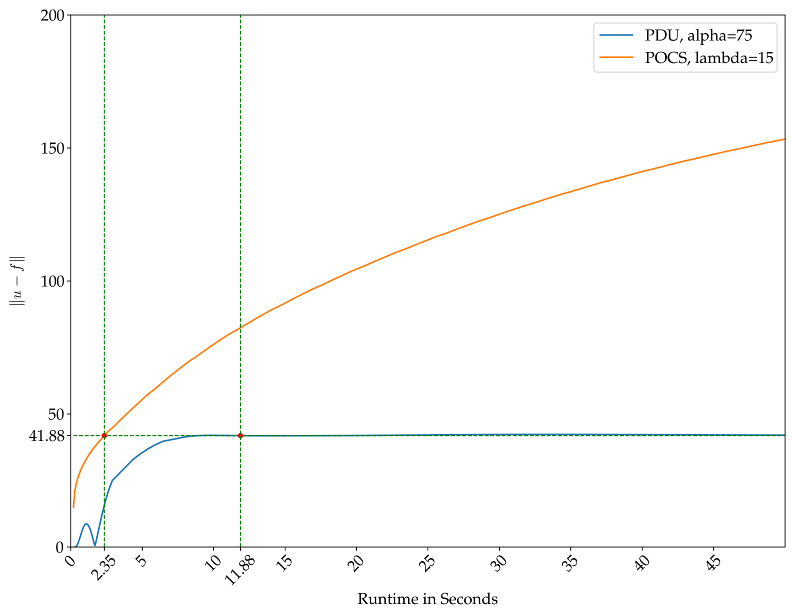


Figure 6.8.: Example of runtime (including computation and memory transfer time) for TV-GD and PDU respectively to reach the same predefined data fidelity term.

Test image	SSIM	MSE	PSNR
Noisy image	0.6460	0.00524	22.808 dB
Denoised image	0.8731	0.00027	35.693 dB

Table 6.1.: Comparison of image quality metrics between the updated image from TV-GD and the noisy image.

compared. As shown, the runtime required to achieve the same predefined data fidelity term in TV-GD is about 2.35s, which is much less than 11.88s in PDU. The experiment was conducted on one of the *Stone* datasets with a size of about 1.8GB. After applying the comparison procedure, the scheme was validated by the ground truth reference data, and the L2-norm of the error to the ground truth at this point in TV-GD is about 2.09136, while the value comes to 2.19174 in PDU. The updated images obtained from PDU and TV-GD are both close to the ground truth data, as they have almost the same L2-norm of the error to the ground truth. Therefore, we have the conclusion that TV-GD are able to reach almost the same image quality as PDU, while TV-GD needs less time but is harder to adjust to produce an optimal solution.

In addition, good denoising performance was obtained for the updated image from TV-GD according to the image quality metrics, as shown in Table 6.1. The metrics of the updated image from PDU are not presented here, as this updated image is almost the same as the updated image from TV-GD.

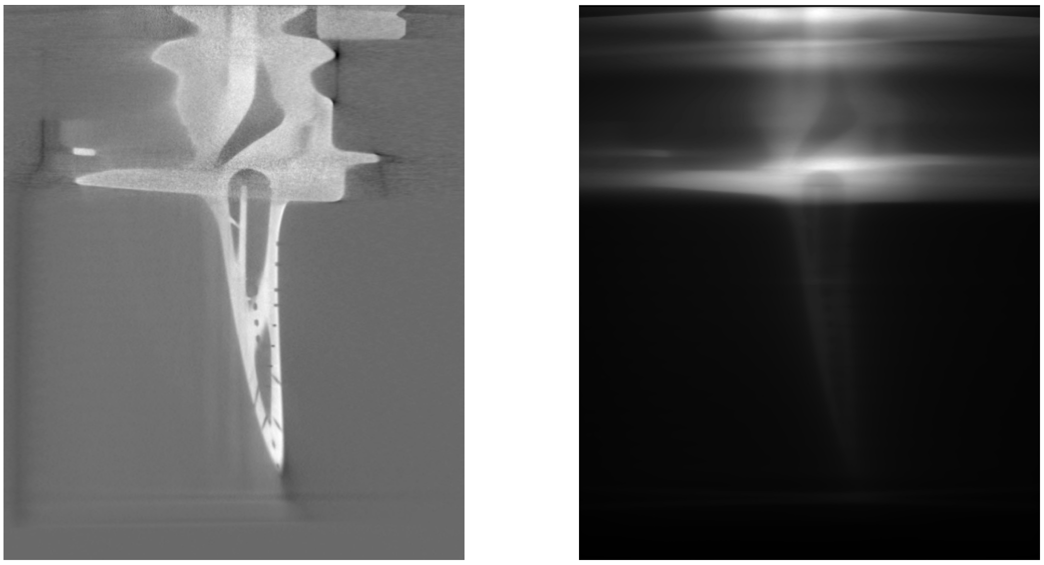

(a) Original image (b) Noise level estimation

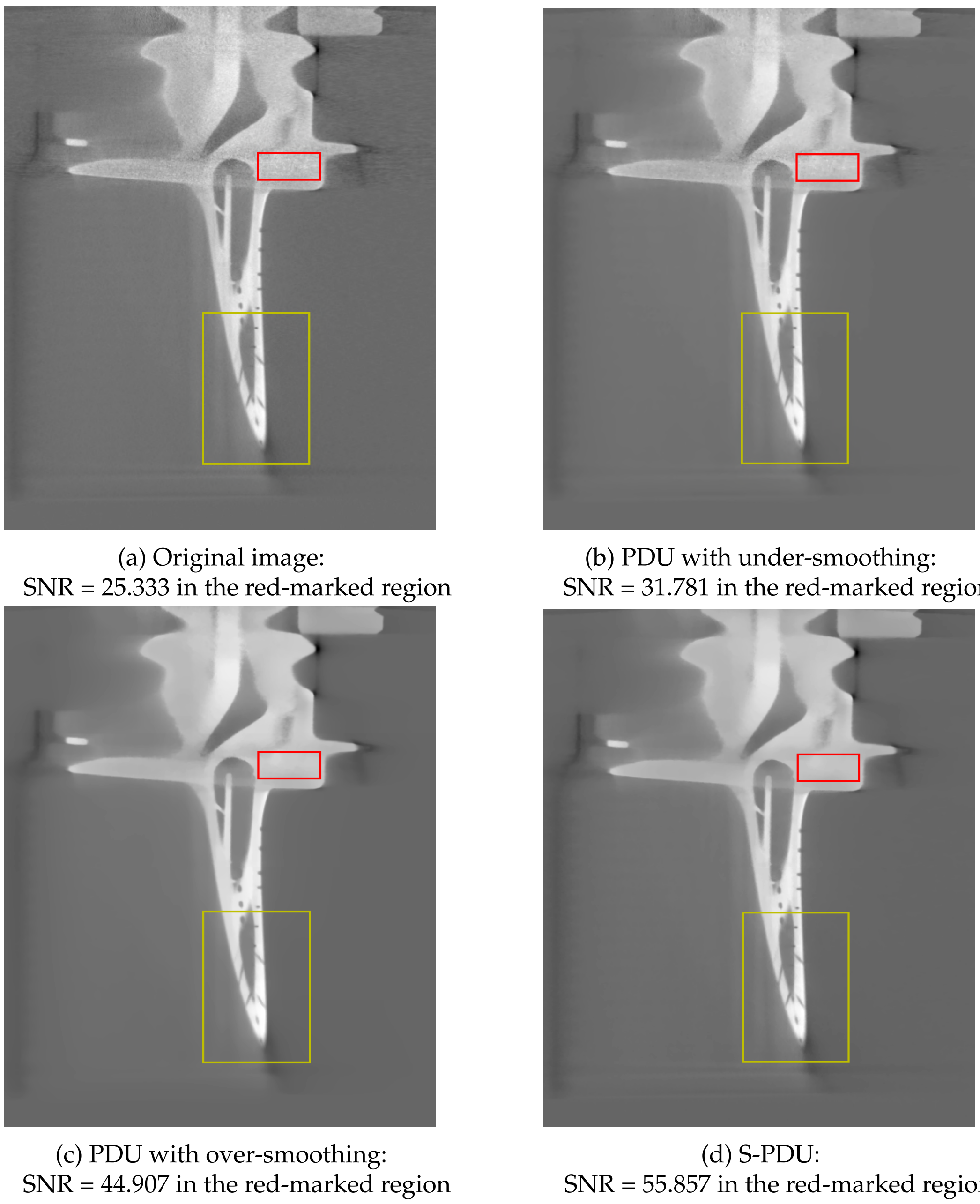
Figure 6.9.: Example of noise level estimation from the noisy image.

6.6. Comparison of PDU and S-PDU

The PDU solution results from the ROF problem, which assumes that the additive noise in the original image is a Gaussian-distributed random variable with a scalar variance. If the noise level varies across the image, and PDU applies the same smoothing strength to the whole image, then some parts of the image may become over-smoothing or under-smoothing depending on the smoothing strength.

The statistical PDU considers multivariate variance of noise across the volume and introduces the weighted least square data fidelity term. The weights are directly represented by the noise level at the corresponding voxel. The smoothing strength is correspondingly weighted voxel-wise, as shown in Figure 6.9. This figure illustrates the estimation of noise level from the original noisy image for one image slice from the $Blade$ dataset. As we can see, the noise levels differ at each voxel, and the noise level at the upper part is higher compared to the lower part in this volume.

Figure 6.10 compares the image quality of the updated image after applying PDU and S-PDU for the original noisy $Blade$ data in Figure 6.10a. The noise level in the upper part is visibly much higher than in the lower part. In order to preserve the small structures in the lower part, Figure 6.10b is obtained after applying PDU with weak smoothing strength, but the noise on the upper part is still obvious. If stronger smoothing strength is chosen, like in Figure 6.10c, most of the noise is eliminated, but the lower part is over-smoothing. Small structures are destroyed, and important details are lost in that area. Figure 6.10d shows the denoised image after applying statistical denoising, and the noise is removed on the upper part while still preserving the small structures with weaker smoothing strength compared to the upper part. The SNR obtained from the red-marked homogeneous region is 55.857, which is much higher than the original image and under-smoothing PDU. Statistical PDU shows a much better denoising result both visually and quantitatively over PDU.

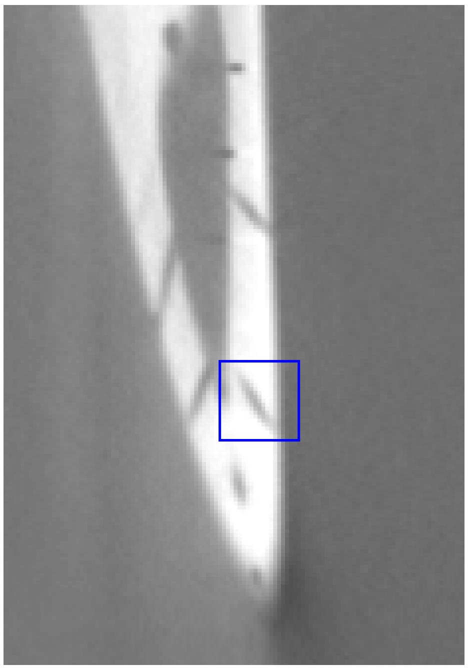

(a) Original image:
SNR = 25.333 in the red-marked region

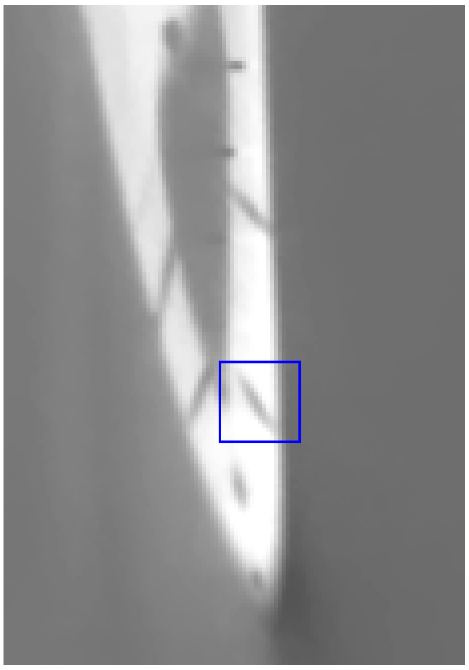
(b) PDU with under-smoothing:
SNR = 31.781 in the red-marked region

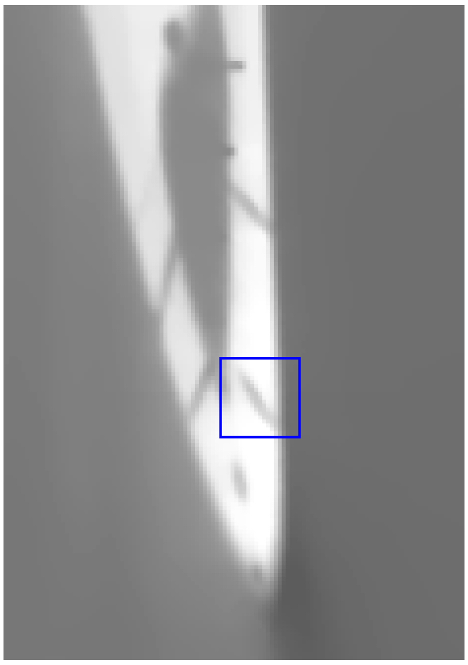
(c) PDU with over-smoothing:
SNR = 44.907 in the red-marked region

(d) S-PDU:
SNR = 55.857 in the red-marked region

Figure 6.10.: Comparison of PDU and S-PDU on the *Blade* dataset with different noise levels across the image, and SNR values are computed in the same red-marked regions.


(a) Original image: magnification for details


(b) PDU with under-smoothing: magnification for details


(c) PDU with over-smoothing: magnification for details

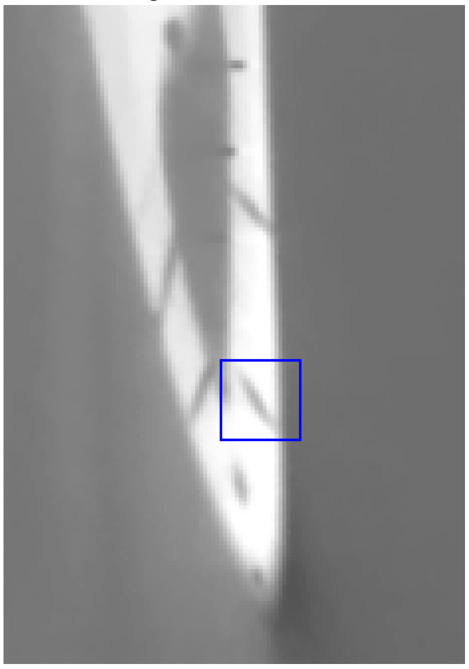

(d) S-PDU: magnification for details

Figure 6.11.: Comparison of PDU and S-PDU on the *Blade* dataset with different noise levels across the image, and details in the lower part are magnified.

7. Discussion and Conclusion

The work presented in this thesis can be divided into two main components. The first part involves the multi-GPU implementation of two 3D total variation algorithms: PDU and TV-GD, both implemented in OpenCL. The second part encompasses the multi-GPU implementation of a statistical 3D denoising algorithm derived from PDU in OpenCL. The chapters of the thesis have presented both the numerical and GPU implementation of these algorithms. Detailed discussions about GPU performance and numerical analysis are also included.

PDU addresses the unconstrained problem of the ROF model by introducing a dual variable, converting the minimization problem into a saddle point problem. In contrast, POCS represents a constrained minimization approach that directly minimizes the total variation norm, but POCS is constrained by a predefined data fidelity term. TV-GD is the part of the TV Gradient Descent taken from the complete POCS algorithm. The statistical PDU applies a weighted least squares data fidelity term, utilizing element-wise noise variance as weights. The element-wise noise is estimated by propagating the noise estimation in projections through the reconstruction pipeline.

The algorithms are implemented on multi-GPU setups using OpenCL programming language, offering hardware portability across different vendors. Optimized image splitting and distribution schemes enable accommodating large datasets across multiple GPUs. Compute kernels are analyzed through the Roofline model, leading to tailored optimizations based on whether they are compute-bound or memory-bound. Appropriate work-item and vectorization schemes are adopted to maximize parallelism and memory access coherence. In addition, detailed discussions on reduction operations are provided to avoid unnecessary synchronization and better utilize shared memory. GPU scaling is also analyzed and optimized through the use of multiple command queues within a single device and minimizing synchronization between command queues.

Convergence evaluation for TV-GD and PDU employs individual stopping criteria and the L2-norm error of updated images in relation to ground truth data when available. The results reveal that PDU converges to an optimal solution eventually with chosen step sizes, while TV-GD needs to be halted before the updated image becomes over-smoothing. In order to avoid both under-smoothing and over-smoothing, the smoothing strength and rate are investigated by tuning parameters and iteration counts in both algorithms. In PDU, the smoothing strength is controlled by the parameter α, and the smoothing rate is influenced by the selection of step sizes for the primal and duality updates. In contrast, the smoothing strength of TV-GD is tied to the number of iterations, while the parameter λ governs the step size for the smoothing procedure.

To check the image quality in a quantitative way across different datasets and algorithms, metrics like MSE, PSNR, SSIM, and SNR are introduced based on the availability of ground truth data. Utilizing these indicators, a comparison scheme is devised to compare PDU and TV-GD in terms of runtime and denoised image quality. The experiments demonstrate that

TV-GD achieves the same predefined data fidelity term with significantly less runtime than PDU, but PDU retains an advantage in robustness.

A comparison between PDU and S-PDU is conducted on the $Blade$ data where the noise level on the upper part is visibly much higher than the lower part. The results show the superiority of statistical PDU in applying variable smoothing strengths at different voxels based on statistical information. Statistical PDU can efficiently prevent under-smoothing and over-smoothing in specific regions, leading to enhanced denoising results both visually and quantitatively compared to PDU.

While the presented structure and code are comprehensive, there remain potential areas for improvement. Enhancements could include efficient memory transfer and computation overlapping schemes [52], allowing the multi-GPU code to increase parallelism and reduce overhead, especially for PDU, which involves numerous memory transfer steps. In traditional data transfer, the host waits for the data transfer between the host memory and the device memory to complete before proceeding with other tasks. This waiting time can lead to inefficient resource utilization and decreased performance. In order to ensure overlapping results are correct, a more elaborate synchronization mechanism and data management scheme are needed.

Further improvements could be made to the numerical implementation of TV-GD. Despite its good GPU performance and scalability in comparison to PDU and its lower CPU and GPU memory requirements, TV-GD is less stable and robust due to the absence of a data fidelity term constraint. The algorithm is harder to fine-tune for optimal denoised solutions and is prone to producing over-smoothing updated images. Additionally, it would be intriguing to explore the application of statistical information using the TV-GD framework.

Bibliography

[1] Dan E Dudgeon and Russell M Mersereau. *Multidimensional digital signal processing*. Prentice-hall, 1984.

[2] Leonid I Rudin, Stanley Osher, and Emad Fatemi. Nonlinear total variation based noise removal algorithms. *Physica D: nonlinear phenomena*, 60(1-4):259–268, 1992.

[3] Gengsheng L Zeng. Nonuniform noise propagation by using the ramp filter in fan-beam computed tomography. *IEEE Transactions on Medical Imaging*, 23(6):690–695, 2004.

[4] David R Pauluzzi and Norman C Beaulieu. A comparison of snr estimation techniques for the awgn channel. *IEEE Transactions on communications*, 48(10):1681–1691, 2000.

[5] Andrei Nikolaevich Tikhonov, AV Goncharskii, VV Stepanov, and Igor Viktorovich Kochikov. Ill-posed problems of image processing. In *Akademiia Nauk SSSR Doklady*, volume 294, pages 832–837, 1987.

[6] Thomas Huang, GJTGY Yang, and Greory Tang. A fast two-dimensional median filtering algorithm. *IEEE transactions on acoustics, speech, and signal processing*, 27(1):13–18, 1979.

[7] Richard A Haddad, Ali N Akansu, et al. A class of fast gaussian binomial filters for speech and image processing. *IEEE Transactions on Signal Processing*, 39(3):723–727, 1991.

[8] Bastian Goldluecke. *Total variation*, pages 1266–1269. Springer International Publishing, Cham, 2021.

[9] Mingqiang Zhu and Tony Chan. An efficient primal-dual hybrid gradient algorithm for total variation image restoration. *Ucla Cam Report*, 34:8–34, 2008.

[10] Rajendra Bhatia and Chandler Davis. A cauchy-schwarz inequality for operators with applications. *Linear algebra and its applications*, 223:119–129, 1995.

[11] Alex Sawatzky. Performance of first-order algorithms for TV penalized weighted least-squares denoising problem. In *Image and Signal Processing: 6th International Conference, ICISP 2014, Cherbourg, France, June 30–July 2, 2014. Proceedings 6*, pages 340–349. Springer, 2014.

[12] Soeren I. Olsen. Estimation of noise in images: An evaluation. *CVGIP: Graphical Models and Image Processing*, 55(4):319–323, 1993.

[13] Avinash C Kak and Malcolm Slaney. *Principles of computerized tomographic imaging*. SIAM, 2001.

[14] Kostadin Dabov, Alessandro Foi, Vladimir Katkovnik, and Karen Egiazarian. Image denoising with block-matching and 3d filtering. In *Image processing: algorithms and systems, neural networks, and machine learning*, volume 6064, pages 354–365. SPIE, 2006.

[15] Mário AT Figueiredo and Robert D Nowak. An em algorithm for wavelet-based image restoration. *IEEE Transactions on Image Processing*, 12(8):906–916, 2003.

[16] Anja Borsdorf. *Adaptive filtering for noise reduction in X-Ray computed tomography*. PhD thesis, Citeseer, 2009.

[17] Emil Y Sidky and Xiaochuan Pan. Image reconstruction in circular cone-beam computed tomography by constrained, total-variation minimization. *Physics in Medicine & Biology*, 53(17):4777, 2008.

[18] Yan Liu, Jianhua Ma, Yi Fan, and Zhengrong Liang. Adaptive-weighted total variation minimization for sparse data toward low-dose x-ray computed tomography image reconstruction. *Physics in Medicine & Biology*, 57(23):7923, 2012.

[19] Ander Biguria, Reuben Lindroosa, Robert Bryllb, Hossein Towsyfyana, Hans Deyhlea, Richard Boardmand, Mark Mavrogordatod, Manjit Dosanjhc, Steven Hancockc, and Thomas Blumensatha. Arbitrarily large iterative tomographic reconstruction on multiple gpus using the tigre toolbox. *arXiv preprint arXiv:1905.03748*, 2019.

[20] Florian Knoll, Markus Unger, Clemens Diwoky, Christian Clason, Thomas Pock, and Rudolf Stollberger. Magnetic resonance materials in physics, biology and medicine fast reduction of undersampling artifacts in radial mr angiography with 3d total variation on graphics hardware. *Magma (New York, NY)*, 23(2):103, 2010.

[21] Ander Biguri. *Iterative reconstruction and motion compensation in computed tomography on GPUs*. PhD thesis, University of Bath, 2018.

[22] Ramon Carbó and Emili Besalú. Definition, mathematical examples and quantum chemical applications of nested summation symbols and logical kronecker deltas. *Computers & Chemistry*, 18(2):117–126, 1994.

[23] Alexander H-D Cheng and Daisy T Cheng. Heritage and early history of the boundary element method. *Engineering analysis with boundary elements*, 29(3):268–302, 2005.

[24] John E Stone, David Gohara, and Guochun Shi. Opencl: A parallel programming standard for heterogeneous computing systems. *Computing in science & engineering*, 12(3):66, 2010.

[25] Hercules Cardoso Da Silva, Flavia Pisani, and Edson Borin. A comparative study of sycl, opencl, and openmp. In *2016 International Symposium on Computer Architecture and High Performance Computing Workshops (SBAC-PADW)*, pages 61–66. IEEE, 2016.

[26] Goutham Kalikrishna Reddy Kuncham, Rahul Vaidya, and Mahesh Barve. Performance study of gpu applications using sycl and cuda on tesla v100 gpu. In *2021 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7. IEEE, 2021.

[27] Suejb Memeti, Lu Li, Sabri Pllana, Joanna Kołodziej, and Christoph Kessler. Benchmarking opencl, openacc, openmp, and cuda: programming productivity, performance, and energy consumption. In *Proceedings of the 2017 Workshop on Adaptive Resource Management and Scheduling for Cloud Computing*, pages 1–6, 2017.

[28] Marcel Breyer, Alexander Van Craen, and Dirk Pflüger. A comparison of sycl, opencl, cuda, and openmp for massively parallel support vector machine classification on multi-vendor hardware. In *International Workshop on OpenCL*, pages 1–12, 2022.

[29] Benedict Gaster, Lee Howes, David R Kaeli, Perhaad Mistry, and Dana Schaa. *Heterogeneous computing with openCL: revised openCL 1.* Newnes, 2012.

[30] Aaftab Munshi, Benedict Gaster, Timothy G Mattson, and Dan Ginsburg. *OpenCL programming guide*. Pearson Education, 2011.

[31] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65–76, 2009.

[32] Pieter Hijma, Stijn Heldens, Alessio Sclocco, Ben Van Werkhoven, and Henri E Bal. Optimization techniques for gpu programming. *ACM Computing Surveys*, 55(11):1–81, 2023.

[33] Gert-Jan van den Braak, Bart Mesman, and Henk Corporaal. Compile-time gpu memory access optimizations. In *2010 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation*, pages 200–207. IEEE, 2010.

[34] Andrew Davidson, Sean Baxter, Michael Garland, and John D Owens. Work-efficient parallel gpu methods for single-source shortest paths. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, pages 349–359. IEEE, 2014.

[35] Chao-Hung Hsu, I-Wei Wu, and Jean Jyh-Jiun Shann. Dynamic memory optimization and parallelism management for opencl. In *2014 International Conference on Information Science, Electronics and Electrical Engineering*, volume 2, pages 776–780. IEEE, 2014.

[36] Nicolas Delbosc, Jon L Summers, AI Khan, Nikil Kapur, and Catherine J Noakes. Optimized implementation of the lattice boltzmann method on a graphics processing unit towards real-time fluid simulation. *Computers & Mathematics with Applications*, 67(2):462–475, 2014.

[37] M Lefebvre, P Guillen, J-M Le Gouez, and C Basdevant. Optimizing 2d and 3d structured euler cfd solvers on graphical processing units. *Computers & Fluids*, 70:136–147, 2012.

[38] Vasily Volkov and James W Demmel. Benchmarking gpus to tune dense linear algebra. In *SC'08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–11. IEEE, 2008.

[39] Yi Yang and Huiyang Zhou. The implementation of a high performance gpgpu compiler. *International Journal of Parallel Programming*, 41:768–781, 2013.

[40] Santonu Sarkar, Sayantan Mitra, and Ashok Srinivasan. Reuse and refactoring of gpu kernels to design complex applications. In *2012 IEEE 10th International Symposium on Parallel and Distributed Processing with Applications*, pages 134–141. IEEE, 2012.

[41] Jens Glaser, Trung Dac Nguyen, Joshua A Anderson, Pak Lui, Filippo Spiga, Jaime A Millan, David C Morse, and Sharon C Glotzer. Strong scaling of general-purpose molecular dynamics simulations on gpus. *Computer Physics Communications*, 192:97–107, 2015.

[42] Pekka Jääskeläinen, Ville Korhonen, Matias Koskela, Jarmo Takala, Karen Egiazarian, Aram Danielyan, Cristóvão Cruz, James Price, and Simon McIntosh-Smith. Exploiting task parallelism with opencl: a case study. *Journal of Signal Processing Systems*, 91:33–46, 2019.

[43] Aaftab Munshi. The opencl specification. In *2009 IEEE Hot Chips 21 Symposium (HCS)*, pages 1–314. IEEE, 2009.

[44] Mahdi Khosravy, Nilesh Patel, Neeraj Gupta, and Ishwar K Sethi. Image quality assessment: A review to full reference indexes. *Recent Trends in Communication, Computing, and Electronics: Select Proceedings of IC3E 2018*, pages 279–288, 2019.

[45] Earl W Duncan and Kerrie L Mengersen. Comparing bayesian spatial models: Goodness-of-smoothing criteria for assessing under-and over-smoothing. *PloS one*, 15(5):e0233019, 2020.

[46] Clifford M Hurvich, Jeffrey S Simonoff, and Chih-Ling Tsai. Smoothing parameter selection in nonparametric regression using an improved akaike information criterion. *Journal of the Royal Statistical Society Series B: Statistical Methodology*, 60(2):271–293, 1998.

[47] Umme Sara, Morium Akter, and Mohammad Shorif Uddin. Image quality assessment through fsim, ssim, mse and psnr—a comparative study. *Journal of Computer and Communications*, 7(3):8–18, 2019.

[48] Jacob Søgaard, Lukáš Krasula, Muhammad Shahid, Dogancan Temel, Kjell Brunnström, and Manzoor Razaak. Applicability of existing objective metrics of perceptual quality for adaptive video streaming. In *Electronic Imaging, Image Quality and System Performance XIII*, 2016.

[49] Renuka G Deshpande, Lata L Ragha, and Satyendra Kumar Sharma. Video quality assessment through psnr estimation for different compression standards. *Indonesian Journal of Electrical Engineering and Computer Science*, 11(3):918–924, 2018.

[50] Zhou Wang, Alan C Bovik, Hamid R Sheikh, and Eero P Simoncelli. Image quality assessment: from error visibility to structural similarity. *IEEE transactions on image processing*, 13(4):600–612, 2004.

[51] Jerrold T Bushberg and John M Boone. *The essential physics of medical imaging*. Lippincott Williams & Wilkins, 2011.

[52] Timo Zinsser and Benjamin Keck. Systematic performance optimization of cone-beam back-projection on the kepler architecture. *Proceedings of the 12th Fully Three-Dimensional Image Reconstruction in Radiology and Nuclear Medicine*, pages 225–228, 2013.

List of Figures

List of Tables

List of Algorithms

A. Appendix

A.1. Abbreviations

CT	Computed Tomography	2
3D	Three-dimensional	2
FBP	Filtered Backprojection	2
SNR	Signal-to-noise Ratio	2
TV	Total Variation	2
GPU	Graphic Processing Unit	3
GPGPU	General Purpose GPU	3
CPU	Central Processing Unit	3
HPC	High Performance Computing	3
CUDA	Compute Unified Device Architecture	3
OpenCL	Open Computing Language	3
AWGN	Additive White Gaussian Noise	5
ROF	Rudin-Osher-Fatemi	6
PDU	Primal-dual	6

POCS	Projection onto Convex Sets	7
AW-TV	Adaptive-weighted Total Variation	7
TV-GD	Total Variation Gradient Descent	9
AW-POCS	Adaptive-weighted POCS	11
ASD-POCS	Adaptive-steepest-descent POCS	10
ART	Algebraic Reconstruction Technique	10
S-PDU	Statistical PDU	12
TFLOPS	Tera Floating Point Operations Per Second	24
RAM	Random-access Memory	24
DSPs	Digital Signal Processors	13
FPGAs	Field-programmable Gate Arrays	13
API	Application Programming Interface	14
FLOPS	Floating Point Operations Per Second	25
SIMD	Single Instruction, Multiple Data	27
FR	Full-reference	38
NR	No-reference	38
KKT	Karush-Kuhn-Tucke	42

MSE	Mean Square Error	48
RMSE	Root Mean Square Error	48
PSNR	Peak Signal to Noise Ratio	48
SSIM	Structural Similarity Index	48