High-Performance Computing for Block-Iterative Tomography Reconstructions

Mads Friis Hansen

Kongens Lyngby 2016

Technical University of Denmark
Department of Applied Mathematics and Computer Science
Richard Petersens Plads, building 324,
2800 Kongens Lyngby, Denmark
Phone +45 4525 3031
compute@compute.dtu.dk
www.compute.dtu.dk

Summary (English)

Modern equipment for X-ray tomography produces large amounts of data, and it is necessary to develop efficient high-performance algorithms and software for treating such problems. In this master thesis such algorithms that can take advantage of GPU accelerators are developed and implemented.

Specifically, central processing unit (CPU) and graphical processing unit (GPU) kernels are developed in C++, implementing the projection method introduced by Joseph, for use with large-scale tomographic reconstruction problems in an existing framework.

The implementation is compared to other projection methods both with regard to reconstruction quality and computation performance. This is specifically oriented towards block-sequential and block-parallel versions of the row-oriented Kaczmarz algorithm (also known as ART), that can use the CPU and/or GPU to compute the forward- and back-projections without explicitly forming and storing the so-called system matrix. Block-sequential and block-parallel versions of the reconstruction algorithm will be compared to highlight the specific advantages and disadvantages to the different approaches, and an implementation of the block-sequential method, proven to be superior for multicore computing, is tested and analysed for the best performance with domain decomposition.

The work focuses on implementation aspects, including issues of efficiency and portability. Background regarding the theoretical foundation of the algorithms is also studied. The software is tested on large-scale experimental data from DTU and has performance studies and comparison of the chosen projection methods conducted.

KEYWORDS: Block methods, block-sequential, block-parallel, ART methods,

SIRT methods, High-Performance Computing, tomography, tomographic image reconstruction, computed tomography, CT, projection methods, Joseph method, Line length method, backprojection method

Summary (Danish)

Moderne udstyr til X-ray tomografi producerer store mængder data, og det er nødvendigt at udvikle effektive high-performance algoritmer og software for at behandle sådanne problemer. I denne kandidat afhandling vil sådanne algoritmer, der kan udnytte GPU acceleratorer, blive udviklet og implementeret.

Specifikt bliver CPU og GPU kernels udvilket i C++, som implementerer projektionsmetoden introduceret af Joseph, til brug med stor-skala tomografisk rekonstruktionsproblemer i et eksisterende framework.

Implementationen er sammenlignet med øvrige projektionsmetoder både i relation til kvaliteten af rekonstruktion og beregningsmæssig ydeevne. Dette er specifikt beregnet mod blok-sekventielle og blok-parallelle versioner af den rækkeorienterede Kaczmarz algoritme (også kendt som ART), som kan bruge CPUen og/eller GPUen til at beregne forward- og back-projections uden eksplicit at konstruere og lagre den såkaldte system matrix.

Blok-sekventielle og blok-parallelle versioner af rekonstruktions algoritmen vil blive sammenlignet for at belyse de specifikke fordele og ulemper ved de forskellige fremgangsmåder, og en implementation af den blok-sekventielle metode, vist at være bedre ved multicore computing, er testet og analyseret for at få den bedste ydeevne med domæne opsplitning.

Arbejdet med projektet fokuserer på implementation aspekter, inkluderende områder om effektivitet og integration. Baggrund omkring det teoretiske grundlag for algoritmerne bliver ligeledes studeret. Softwaren bliver testet på stor-skala eksperimentel data fra DTU og ydelses-studier og sammenligninger for de valgte projektionsmetoder bliver udført.

NØGLEORD: Blok metoder, blok-sekventiel, blok-parallel, ART metoder, SIRT metoder, Høj-Performance Computing, tomografi, tomografisk billede gendannelse, computed tomografi, CT, Joseph metode, line length metode

Preface

This thesis was prepared at DTU Compute in fulfilment of the requirements for acquiring an M.Sc. in Engineering and the completion of the master degree in *Mathematical Modeling and Computing*.

It represents a workload of 35 ECTS points and has been prepared during a six month period from September 8 to March 8 which was extended by three months from April 25 to July 25. The study has been conducted under the supervision of Associate Professor Bernd Dammann, Hans Henrik Brandenborg Sørensen and Professor Per Christian Hansen.

Structuring

The structure of this thesis will be to regularly introduce the relevant theory as a support of obtained results, examples, implementation and discussion such that these two parts mutually support each other.

The goal of this is to have a continuous flow throughout the report with the focus and result of the project in mind the entire way through.

After introducing the underlying concepts and motivation in chapter 1, we will cover the conversion from the physical world described by tomography to the *compute* part of computed tomography in chapter 2. This will focus on modelling the physics involved as accurately as possible and how this can be implemented with different choice of projection methods.

How our modelled problem is solved is covered in chapter 3, where we look at large 3D problems where large amounts of data is an issue. We will especially compare how the quality of the reconstruction of our CT problem compares for

the implementation.

How the covered methods are implemented is covered in chapter 4 along with areas to be aware of when implementing the methods and where to optimize the code for performance.

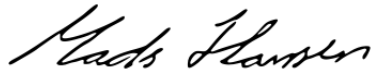
High-Performance Computing aspects for large-scale problems will be looked into in chapter 5 along with a performance study of our implementation for large-scale problems on state-of-the-art hardware before concluding the thesis.

Acknowledgements

I would like to thank all three of my supervisors, they have been a tremendous support during this project and incredibly helpful and patient. And I especially thank Hans Henrik Sørensen for putting time and effort into this project with me.

I dedicate this work to my Dad, I hope to be able to live up to his hopes and expectations of me.

Lyngby, 25-July-2016



Mads Friis Hansen

List of Symbols

Symbol	Description
A	System coefficient matrix: discrete to discrete forward operator, approximated projection matrix
A^T	Transpose of projection matrix A
A_i	Projection operator corresponding to the ith projection angle.
a_i	ith row of A
$a_{i,j}$	(i,j)th element of projection matrix A
b	Right-hand side: Measured data, vectorized version of projections
x	"Naive" solution, body source
m, n	Matrix dimensions
λ	Relaxation parameter
ϵ	Discretization error
I	Identity matrix
X	Solution volume
P_i	Projection plane i
P_x	Projection plane x-dimension
P_y	Projection plane y-dimension

List of Abbreviations

Symbol	Description
CT	Computed tomography
CAT	Computerized axial tomography
MRI	Magnetic resonance imaging
ERT	Electrical resistivity tomography
$SPECT$	Single-photon emission computed tomography
FBP	Filtered Back Projection
FP	Forward projection
BP	Back projection
ART	Algebraic Reconstruction Technique
$SIRT$	Simultaneous Iterative Reconstruction Technique
$SART$	Simultaneous Algebraic Reconstruction Technique
AIR	Algebraic Iterative Reconstruction
HPC	High-Performance Computing
GPU	Graphics processing unit
$GPGPU$	General-purpose (computing) graphics processing unit
CPU	Central processing unit
SP, SMP, SM, SMX	Stream multi-processor
$SIMD$	Single instruction, multiple data
ALU	Arithmetic logic unit

Contents

Summary (English) **i**

Summary (Danish) **iii**

Preface **v**

Acknowledgements **vii**

List of Symbols **ix**

List of Abbreviations **xi**

1 Introduction **1**
1.1 Tomography . . . 1
1.1.1 Computed tomography . . . 4
1.1.2 Computational complexity . . . 5
1.2 The Radon transform . . . 7
1.3 Filtered backprojection . . . 9
1.4 Algebraic methods . . . 10
1.4.1 SIRT . . . 11
1.4.2 ART . . . 12
1.5 Modern hardware architecture . . . 13
1.5.1 CPU architecture . . . 13
1.5.2 Multicore architecture . . . 14
1.5.3 Manycore architecture . . . 16
1.5.4 Domain Decomposition . . . 17
1.6 Test problem(s) . . . 19
1.6.1 Walnut . . . 19
1.6.2 Test phantom . . . 20
1.7 Test hardware . . . 21

2 Discretization **23**
2.1 Projection Models . . . 25
2.1.1 $0-1$ model . . . 25
2.1.2 Line length . . . 26
2.1.3 Strip area . . . 27
2.1.4 Joseph's method . . . 28
2.1.5 Backprojection method . . . 30
2.2 Comparing the Projection Methods . . . 32
2.2.1 Reconstructions . . . 32
2.2.2 Performance of the forward- and backprojections . . . 37

3 Block-methods **41**
3.1 Block-Iteration method . . . 42
3.1.1 BLOCK-IT . . . 43
3.1.2 SART . . . 44
3.2 Advantages of block-sequential methods . . . 45
3.2.1 Underdetermined system . . . 46
3.2.2 Overdetermined system . . . 48
3.3 Block reconstruction software . . . 50
3.3.1 General structure . . . 50
3.3.2 The block loop . . . 52

4 Implementing the *Joseph* method **55**
4.1 Building the kernel . . . 55
4.2 *Joseph* method using domain decomposition . . . 59
4.2.1 Explicit zero-padding . . . 60
4.2.2 Virtual zero-padding . . . 62
4.2.3 Domain decomposition on multiple devices . . . 62

5 Large-scale performance results **65**
5.1 Blocking and domain decomposition . . . 65
5.2 GPU cluster performance . . . 68
5.2.1 Scaling of the implementation . . . 68
5.2.2 Barriers and communication . . . 72

6 Conclusion and Future Work **75**
6.1 Future Work . . . 76

A Code **77**
A.1 Joseph GPU Kernel, main loop body . . . 77
A.2 Full Joseph GPU Kernel . . . 80

Bibliography **91**

Chapter 1

Introduction

In this thesis the concept, and the use and relevance of tomography and computed tomography (CT) will first be introduced. This includes the fundamentals of CT such as the physical set-up. Following this is a run-down of some integral methods used in CT, which is the Radon transform, Filtered backprojection (FBP) and algebraic methods, where the latter is the main focus point of this thesis. This will be tied to the aspect of High-Performance Computing (HPC) and why this is highly relevant for computed tomography, and the motivation of this project. Especially the performance of the examined projection model, called the *Joseph* method, compared to other widely-used methods, is of importance in this study. We wish to show that it is viable for High-Performance Computing on large-scale problems, with a variety of advantages over other methods.

The *Joseph* projection method has been implemented for both CPU and GPU computation with tomographic reconstruction software, to accomodate for a wider range of hardware and usability.

1.1 Tomography

The concept of tomography refers to imaging by sectioning or slicing. The word "tomography" is derived from Ancient Greek *tomos*, meaning "slice" or "section" and *graphō*, meaning "to write".

Modern tomography is usually referring to tomographic reconstruction. It has many variations of gathering projection data by projecting beams through an object at multiple angles and applying the data to a tomographic reconstruction software algorithm. Said algorithm will then output the desired reconstructed image of the object interior in accordance with the methods involved with the software algorithm.

Tomography has a lot of important applications, a major one being in medical imaging where computed tomography is used as a tool to diagnose or screening of discease and to supplement X-rays and medical ultrasonography. In this effect it can be used to plan an incision or treatment method prior to actually operating with the ability to observe the interior of an object strictly from the exterior without any intrusion, except for in the form of radiation.

(a) X-ray CT **(b)** MRI **(c)** SPECT **(d)** PET

Figure 1.1: In figure **(a)** is a X-ray driven computed tomography (CT) scanner, while figure **(b)** shows a magnetic resonance imaging (MRI) scanner [18]. Figure **(c)** displays a Single-photon emission computed tomography (SPECT) scanner, whereas a Positron emission tomography (PET) scanner can be seen in **(d)** [22]. All of these four scanners used for medical imaging look fairly identical. This is not much of a surprise as the underlying mathematical problem is the same seen in all four types of medical imaging scanners, with the main difference being the physical phenomena used for signal acquisition, be it X-rays, gamma rays, magneticism or something else.

Figure 1.1 shows four types of medical imaging scanners based on computed tomography. A so-called CT or CAT scanner can be seen in figure 1.1a. This type of medical imaging scanner is based on X-ray radiation for constructing measured projection data, while the MRI scanner shown in figure 1.1b uses magnetic resonance to construct projections of the target volume. The SPECT (Single-photon emission computed tomography) scanner in figure 1.1c and PET (positron emission tomography) scanner in figure 1.1d both rely on detecting radiation from gamma-emitting radioisotope, injected into the patient prior to the scan [22].

The first Computer Tomograph was invented by Sir Godfred Newbold Hounsfield in 1969, and his original prototype, that was ready in 1971, worked by acquiring 160 parallel readings from 180 angles with a separation of 1 degree between each angle. A single scan could take up to 5 minutes and processing the data from the first scan and constructing a reconstructed image took 2.5 hours of work from the EMI computer center (at EMI Central Research Laboratories in Hayes, UK) by Algebraic Reconstruction Technique (ART) [11].

The scanners used for Computed Tomography went through a technical evolution after the first CT scanner that was mainly used for brain CT. After the EMI head scanner began being used actively in medicine in 1971, the competition to the Computed Tomography scanner started introducing faster scanners to the market. To improve the signal acquisition time of over 5 minutes, the next generation of scanners began measuring multiple readings during a scan instead of just a single one as before. This was accomplished by changing the geometry of the beam from a single ray to a so-called fan-beam, utilizing a detector array instead of a single point for detection. This beam set-up can be seen to the right in figure 1.5 on page 8.

This second generation of CT scanners had the fan-beam geometry set-up for signal acquisition as the distinguishing feature, but were still using the step and shoot approach when scanning an object. This is where the ray source and detector plane are both rotated in between each measurement angle, in contrast to measuring under a continuous rotation. The introduction of the fan-beam did achieve an improvement in performance even if the first scanners with this technology only used three detector elements per fan. But this number was quickly increased.

The first whole-body CT scanner was introduced by Hounsfield in 1975 and brought to the market in 1976. This device made it possible to reduce the signal acquisition time from 5 minutes per scan as for the first generation of CT scanners, to an astonishing 18 seconds per scan by using a 20 elements detector. The reduction time of a single scan to below 20 seconds was considered a breakthrough for the technology, as it allowed a scan through the thorax with next to no movement with the patient holding his or her breath during the scan.

The next evolutionary step for the Computed Tomography scanner happened in the period between 1974-1977. Here, several companies were developing the next generation of scanners that would only require rotary motion using broader fan-beam than seen previously. This was accomplished by integrating the radiation source and detector system into what later became known as a "gantry", where the source and detector mechanism rotates around the stationary patient and thus allows for a whole-body Computed Tomography examination. The modern design of this gantry can be seen for four different kind of Computed Tomogra-

phy in figure 1.1 on page 2, where the patient is located on the stationary bench and the measurement device is then rotated around the patient.

1.1.1 Computed tomography

In medicine, the terms computed tomography (CT) or computerized axial tomography (CAT) are mostly referring to the X-ray CT or CAT scans as it the most common form of CT. This is a bit of a misconception though, as computed tomography covers a wide range of techniques using different physical phenomena for the signal acquisition to construct the tomographic image, such as magnetic resonance imaging (MRI) that uses radio-frequency waves, electrical resistivity tomography (ERT) with the use of electrical resistance, Single-photon emission computed tomography (SPECT) where gamma rays are used, or Positron emission tomography (PET) with gamma rays [23]. The underlying mathematical problem is the same for all of these techniques with the main difference being the equipment and type of radiation used for the signal acquisition.

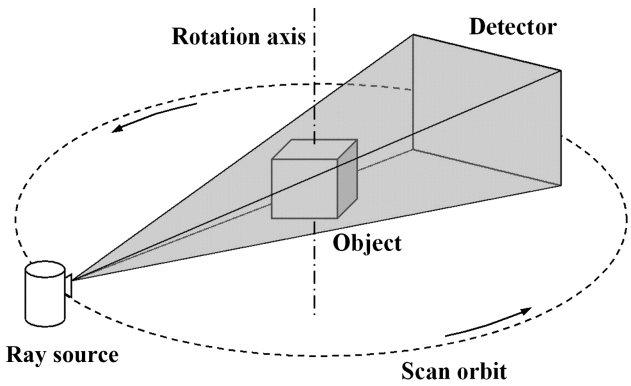


Figure 1.2: General physical set-up of a tomographic image reconstuction problem with a cone-beam [4].

Figure 1.2 shows a general physical set-up of a Computed Tomgraphy scan with a cone-beam, as section 1.1 has described as third generation CT scanners. The radiation source projects a beam through the object and onto a detector plane while circling the scanned object around the rotational axis. The main difference of the physical set-up between a fan-beam and a cone-beam is how the cone-beam will cover a greater area of the object at any one projection angle and that a detector plane is used compared to a line of detectors.

Classical Computed Tomography has a wide range of applications. Many of these medical applications are based on perfusion measurements, which can be perfusion inside tumors or livers or to measure cerebral blood flow and volume in brain perfusions. For these methods to reach the required medical standard, application specific contrast agent often has to be used in combination with CT scanners to accurately measure the desired perfusion.

CT is also an important technology as a planning tool, and many kinds of surgeries and radiotherapies are planned from the patient information attained non-invasively out of a classical CT system.

While the classical CT system and its medical applications are exemplary in their use to conveying the history and the basics of the technology, time has seen the innovation that is Computed Tomography make its way into many different specialized diagnostic machines such as Breast CT systems, Dental CT or Micro CT and even into quality assurance and testing of materials and products in industry. This is commonly referred to as Non-Destructive Testing (NDT) [11].

1.1.2 Computational complexity

When applying tomographic reconstruction, the tomography problems will in most cases become very big and computationally demanding. This is because we want the reconstructed solution to be of as high precision and resolution as possible, which in turn causes the number of angles that rays are projected through the object to increase. This is done with the goal of increasing the precision of the solution but will also increase the number of projection planes. Another reason for the problems becoming very large is how the solution domain is sectioned into smaller and smaller parts during the discretization of the domain. This is done, for one, to increase the resolution and physical correctness of the reconstructed image and also to accommodate the available hardware as much as possible to get the most performance out of the implementation as possible. But this will be discussed in more detail in subsection 1.5.4 and chapter 2.

The increasing complexity in CT can be divided into two underlying causes, both of which advocates the significance of High-Performance Computing in Computed Tomography. On one hand the complexity in CT is associated with the complexity of the used reconstruction algorithm and on the other the complexity is also determined by the amount of data used for the image reconstruction. An increase in the number of rays and projection angles will have a significant impact on the complexity of the CT problem that is to be solved. When considering time critical imaging, as is seen in the use of CT in medicine,

the amount of data per time is especially important.

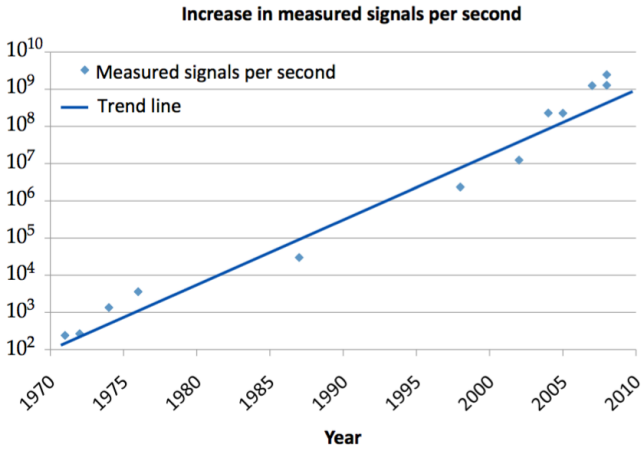


Figure 1.3: Increase in measured signals per second for CT systems through the decades since the introduction of the technology [11].

Since its introduction, when Sir Godfred Newbold Hounsfield and Allan McLeod Cormack was awarded the Nobel Price in Medicine in 1979 for their invention of Computed Tomography, the technical complexity of Computed Tomography has been on the rise. Especially the number of measured signals per time during a tomographic scan has been on an exponential rise. In figure 1.3 can be seen the development in measured signals per second for CT systems from the 1970s until recent times. The exponential increase in problem sizes in Computed Tomography put a large demand on the implementation of the applied image reconstruction algorithms.

The original EMI head scanner introduced by Hounsfield used a just 80^2 pixel images, while most modern scanners use at least 512^2 up to and beyond 1024^2 image matrices [11].

Boyd et al. [3] stated a recommendation for CT fan-beam scanners using an image resolution of $n \times n$ per image slice to have $2n$ as the number of angular samples, and the number of samples in a projection as $1.4n$. The resolution improves with an increase in n, but so does the total number of attenuation measurements and with $O(n^2)$ at that. A higher resolution is always desirable from an image reconstruction perspective, but there has to be struck a balance between higher resolution and number of feasible measurements. In practise this balance often leads to a smaller number of projection angles, making some reconstruction methods perform significantly worse.

1.2 The Radon transform

The basics of computed tomography is the problem of reconstructing the internal structure of an object from multiple projections of that object without actually entering the object, also called non-invasive three-dimensional (3D) imaging [2]. By rotating a beam around the object at different angles, sending out rays from a known source location to a known detector plane, 2D projections are acquired from the known amount of energy that has been absorbed through the object from the radiation by measurements from the detector plane. It can be expressed that the absorbed amount of energy and the projections are acquired through a transformation, called the Radon transform, that maps a line (a ray) into a real number on the projected plane [8]. Such a transform can be described for any ray travelling through the domain with the scanned object. The Radon transform represents the projection data obtained as the output of a tomographic scan

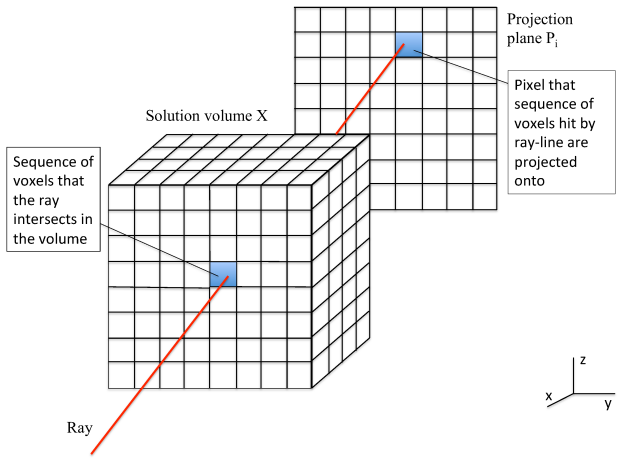


Figure 1.4: The Radon transform maps a line going through the volume (a sequence of voxels intersected by the ray in the discrete case) onto a number on the projection plane (a pixel value).

If the Radon transform maps a line L contained in the domain Ω into a real number, $L \to \mathbb{R}$, which is to say a mapping of the picture X to its sinogram, and if the density at a point $x \in \Omega$ is denoted by $d(x)$, a physical model for the continuous case can be described with the Lambert-Beer's law for any line $L \in \Omega$ as:

$$R(L) = \int_{x \in L} d(x)ds. \tag{1.1}$$

By sending rays through an object from multiple angles and creating a 2D projection of the object for each of these angles, one can then compute a 3D reconstruction of the scanned object by reconstructing the original density of the object from the inverse Radon transform or form a tomographic reconstruction algorithm.

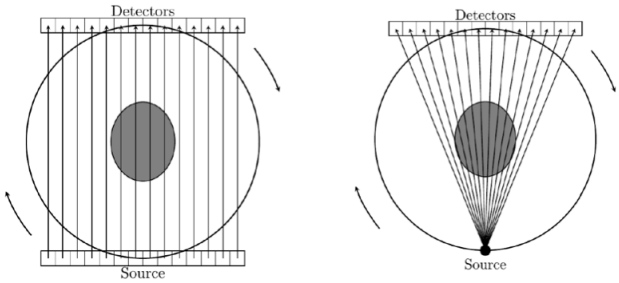


Figure 1.5: Different beam setup. To the left is seen a circular parallel beam and to the right a fan-beam setup.

The shape of the beam can cause the rays to travel in different configurations, such as a *parallel beam*, where the rays travel in parallel spread over the same area from the source to the detector, or in a *fan beam*, where the rays originate from the same point and spread out through one plane in the solution volume to hit the detector plane at multiple points. These two different type of beams can be seen in figure 1.5.

An advantage of the fan-beam when compared to a parallel beam is in how the rays are more densely packed when entering the volume close to the ray source. When the rays are spread out with a larger distance between each ray, it introduces a larger risk of the rays not intersecting a voxel and thus providing no knowledge of this voxel. This can result in a lack of information being transmitted about interior parts of the scanned object or more severely, potentially produce "black spots" in the reconstructed image, especially if the number of projection angles in the measured data is not sufficiently large. The rays being transmitted from a single point and spread out in a fan will result in the entire outer layer of the volume to be especially well-lit when scanning at a 360° rotation, while the innermost interior will be equally lit as for the parallel beam.

Another type of widely-used beam is the cone-beam. This beam is shaped like a cone, spreading out from the ray source and will cover the solution volume in three dimensions compared to the two dimensions of both the parallel- and fan-beam. This also means that the cone-beam will be projected onto a detector plane compared to a line of detectors as for the other two beam set-ups.

The fan-beam, and especially cone-beam, perform very well when working with a limited number of projections as they tend to cover more of the volume compared to the parallel beam.

1.3 Filtered backprojection

To solve this reconstruction problem algorithmically, it is necessary to discretize the image volume consisting of our physical model into a discretized model consisting of a system of linear equations

$$Ax = b, \tag{1.2}$$

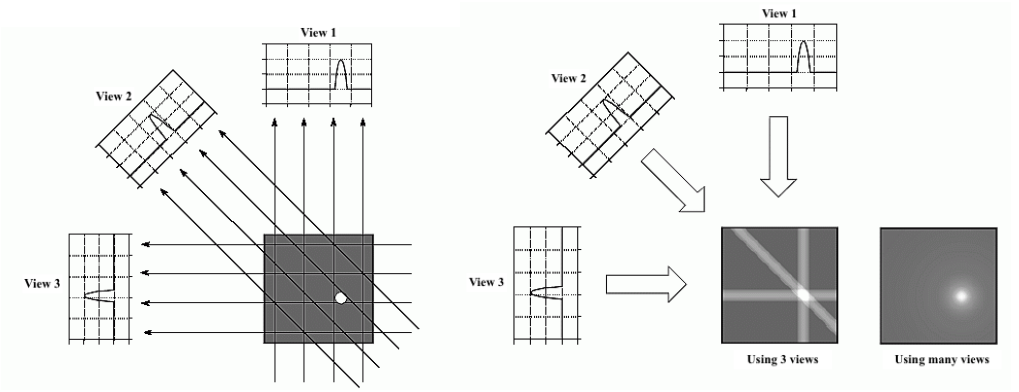
where A is an $m \times n$ system matrix, or discrete forward operator, x is a given image and b represents the measured projection data. How to construct the discrete model will be covered in chapter 2.

One such tomographic reconstruction algorithm is the Filtered backprojection (FBP).

Filtered backprojection (FBP) is a method to correct the blurring of the image encountered in a simple backprojection by filtering it.

First, A backprojection is done by setting all voxel values lying along a ray pointing to a detector plane to the same value. That way each projection is projected back along the original route of the projected rays and the final back-projected image is aquired as the sum of all backprojected views. An illustration of this can be seen in figure 1.6 on the following page. The forward and back projections will be covered further in chapter 3.

With a FBP, the projections are passed through a filter before the backprojection to counteract the blurring of the final backprojected image. A high-



(a) Forward projection **(b)** Backprojection

Figure 1.6: Forward projection and backprojection illustrations.

frequency filter reduces noise and makes the image appear "smoother", while a low-frequency filter enhances edges in the image and makes it appear "sharper".

Using the FBP for image reconstruction requires a large number of projections. If there are not enough, the quality of the reconstructed image suffers. Added to this, the reconstruction technique works only on a limited number of geometries. If the FBP is applied on a problem where the filter doesn't work well, the reconstruction will not be good.

FBP is a fast method with good reconstruction results, but only in the case where plenty of data is available, eg. where the measured data b is equal to or larger than the object x in equation 1.2 on the previous page. But in the case where x is larger than b and we are dealing with an undetermined, ill-posed, sparse system, which often arise in tomographic image reconstruction, this method is not desirable. In that case we look to the algebraic iterative methods.

1.4 Algebraic methods

Algebraic methods such as ART can give better reconstructions for undetermined problems with more limited data and are more flexible methods compared to the FBP. Furthermore, when working with medical Computed Tomography an iterative method is essential to be able to work with smaller doses of radioactive contrast agent.

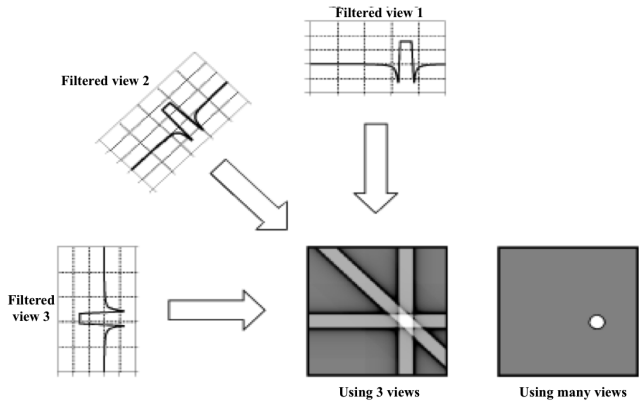


Figure 1.7: Filtered backprojection

It is possible to incorporate prior knowledge of the problem and Total Variation regularization into algebraic methods with little effort. This is not the case with the filtered back-projection.

There are a multitude of methods available, but we have chosen to look more closely at the Algebraic Reconstruction Technique which we have found most suitable for our needs with the biggest error reduction per iteration [19].

1.4.1 SIRT

Simultaneous Iterative Reconstruction Techniques (SIRT) seen in algorithm 1 is an algebraic reconstruction technique used in computed tomography. The main part where the method differs from the Algebraic Reconstruction Technique (ART) discussed in subsection 1.4.2 is that it involves the simultaneous use of every row in the system matrix A for each iteration and thus involve a matrix-vector product.

We write the SIRT algorithm in a general form as:

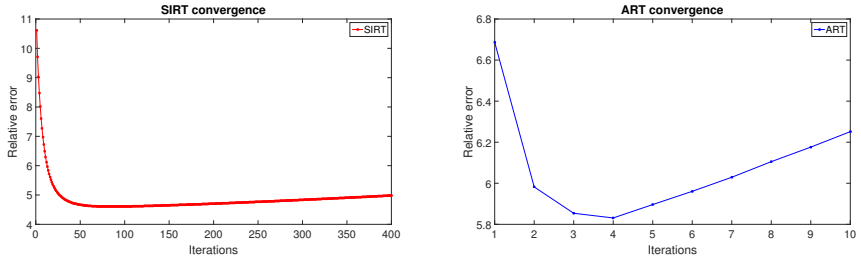
Algorithm 1 SIRT

$$x^k = P_C\left(x^{k-1} + \lambda T A^T M\left(b - Ax^{k-1}\right)\right)$$

where T and M are positive definite matrices, λ is a relaxation parameter and it is required that $0 \leq \lambda \leq 2\|T^{-\frac{1}{2}} A M^{\frac{1}{2}}\|_2^2$ for asymptotic convergence [19]. How T and M are chosen will determine the specific SIRT method in use, for

example defining $T = I$ and $M = I$ will correspond to the Landweber method [12], while setting $T = I$ and $M = diag\left(\frac{1}{\|a_i\|_2^2}\right)$ gives the Cimmino method and corresponds to normalizing the rows of A [6].

P_C defines the constraints used with the algorithm and can for example be a non-negativity or box constraint. In this thesis, P_C will be defined as a projection of all negative values to zero.



(a) SIRT Cimmino semi-convergence. **(b)** ART Kaczmarz semi-convergence.

Figure 1.8: Semi-convergence of the SIRT Cimmino and ART Kaczmarz algorithms on a simple 50×50 tomography example problem with 36 projection angles from a 180 degrees parallel beam. The SIRT algorithm reaches semi-convergence after around 70 iterations while the ART algorithm does after only 4 iterations.

This group of algorithms is easily made to run in parallel which is great for a HPC implementation, but the downside is that the convergence rate might be rather slow causing the iteration count to climb up as can be seen in figure 1.8a. Because of this, the inherent suitability of the algorithm to be run in parallel might not account for the cost of having to run an increased number of iterations for the same reconstruction result.

1.4.2 ART

The Algebraic Reconstruction Technique shown in Algorithm 2 below is an essentially sequential method in that it only uses a single row a_i of the system matrix A at a time in a predefined order in each update of the volume x. Note that we here use ART as a modification of the classical Kaczmarz method with a projection P_C and a sweep over the rows of A at the kth iteration.

Algorithm 2 ART

$x^{k,0} = x^{k-1}$
for i **in** $1 \rightarrow m$ **do**
$$x^{k,i} = P_C\left(x^{k,i-1} + \lambda \frac{b_i - a_i^T x^{k,i-1}}{\|a_i\|_2^2} a_i\right)$$
end for
$x^k = x^{k,m}$

where a_i is the i^{th} row in the matrix A, λ is a relaxation parameter, and it is required that $0 \leq \lambda \leq 2$ for asymptotic convergence. As mentioned, this method has a very fast convergence rate, as seen in figure 1.8b on the preceding page above, but is required to run in serial.

We know that ART has a fast rate of convergence and as such is low in iterations, but we would like a method that works better with and can take advantage of multi-core architecture. Sørensen and Hansen [19] show that a blocking approach with a partitioning of A where the rows in each block are selected to be structurally orthogonal coupled with the ART-sweep allows for an efficient parallel implementation while still retaining the iterate from ART with a fast convergence.

In this thesis we focus on a Block-ART algorithm which is a fast converging "hybrid" of the SIRT and ART algorithms. This method is introduced in section 3.1.

1.5 Modern hardware architecture

When solving an ill-posed sparse linear system in large-scale problems such as in tomographic image reconstruction it is necessary to do so with the help of high-performance computing. The choice of method can have a large impact on the performance of the implementation, and having a method that works well in parallel, is fast, and has a low cost in iterations is very beneficial when solving these types of problems in large scale.

1.5.1 CPU architecture

A computer's central processing unit (CPU), found in all modern computers, is the electric circuitry within the computer that handles all of the instructions of

a computer program by carrying out basic logical, control, input/output (I/O) and arithmetic operations as specified by the instructions from the program.

During the late 1960's and early 1970's the microprocessor was introduced, which is a computer processor incorporating the functions of the CPU on a single, or at most a few, integrated circuits. The integration of the functionality of a CPU onto a single chip made it possible to reduce the cost of processing power tremendously and thus, by being able to produce in large numbers by automated processes, reduce the per unit costs.

As the technology for integrated circuits advanced, it became possible to manufacture single chips with more and more complex processors on. As the size of data objects became larger, it allowed more transistors on a single chip and thus allowing word sizes to increase from 8-bit to 8-bit words and all the way up to today's 64-bits.

After having the technology to put a large number of transistors on a single chip, it then became possible to integrate memory on the same die as the processor in the form of a CPU cache. The CPU cache has the distinct advantage of allowing the CPU a much faster access to data than the off-chip main memory, and this can increase the processing speed of the system if handled well by the application. As the processor clock frequency in general has increased much more rapidly than the speed of external memory, the cache memory is necessary for the processor not to be delayed further by slower external memory.

This makes it necessary for computer programs to take advantage of the structure of the hardware to execute at a faster speed. This is especially the case when large data processing is introduced and to make high performance computing possible.

1.5.2 Multicore architecture

Multicore processor refers to a single chip, or circuit die, with two or more independent processing units, also refered to as cores, integrated. The advantage of a multi-core processor over a single-core processor is in the ability to execute multiple program tasks simultaneously. While there is no increase in single-thread performance compared to a single-core processor, the multiprocessing potential of multi-core CPUs allow for a large overall increase in speed for programs that are designed to take advantage of parallel computing.

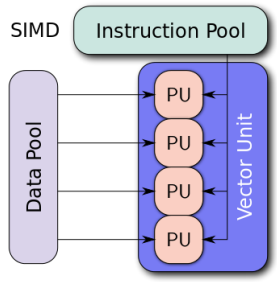


Figure 1.9: Single Instruction, Multiple Data (SIMD) processing unit architecture illustration.

The prevalent architecture for modern multi-core processors, as well as for many manycore processors, are the SIMD processors, which stand for single instruction, multiple data. This describes, as illustrated in figure 1.9, processing units with multiple processing elements that perform the same instruction, or operation, on multiple data elements in parallel - that is, a processing unit with a single instruction stream but multiple data streams. This type of operations are especially advantageous with vectorized programs such as real-time graphics and ray-tracing.

The first foreshadowing of what would later evolve into the multicore architecture of today, was when Rockwell International in the mid 1980s manufactured versions of the Mos Technology 6502 8-bit microprocessor with two 6502 cores on a single chip, sharing the chip's pins on alternate clock-phases.

But it took close to two decades before the first multicore processors as we know them was developed by Intel and AMD among others in the early 2000's. The multicore architecture has quickly become the norm for all modern central processing units (CPUs). Modern multicore processors can have from two up to 22 processing units, or cores, on a single chip.

The general architecture of a multicore SIMD processor is shown in figure 1.10 on the next page. Typically, each processing unit, also called arithmetic logic unit (ALU), will have its own local L1 cache memory while all ALUs will be connected to a shared on-chip memory, called the L2 cache, larger than the L1 cache and that all ALUs have access to. Depending on the architecture in question the multicore CPU might also be equipped with a shared L3 cache.

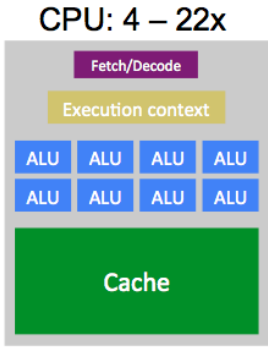


Figure 1.10: Generalized SIMD unit architecture of a central processing unit (CPU). Shown is a single CPU core with modern multiprocessors having between 4 to 22 of such cores.

1.5.3 Manycore architecture

Compared to multicore processors, manycore processors are designed with a higher degree of explicit parallel processing and a higher throughput in mind. This comes at the expense of latency and a lower single thread performance compared to the standard multicore processor.

Manycore processors, or hardware accelerators, are build from a large number of independent processor cores with a lower individual performance than the typical CPU core, which explains the lower single thread performance. Because of this, manycore processors are not very ideal for running serial code but are instead highly efficient at running parallel code.

A typical use for manycore processors, in the form of the GPU, has been in applications for graphical processing because of the tendency for these applications to demand the same operation done many times over for different objects, such as the handling of textures and image pixels in graphical applications. This is also a feature for many HPC implementations, such as in tomographic image reconstruction.

Multicore processors typically has a limited scaling of the executed code as the amounts of data increase due to the issue of cache coherency, which is the inconsistency of shared data across local caches. This is less of an issue for manycore processors, and together with the high parallel performance makes these devices highly useful in High-Performance Computing.

Devices such as graphics processing units (GPUs) and coprocessors like the Intel

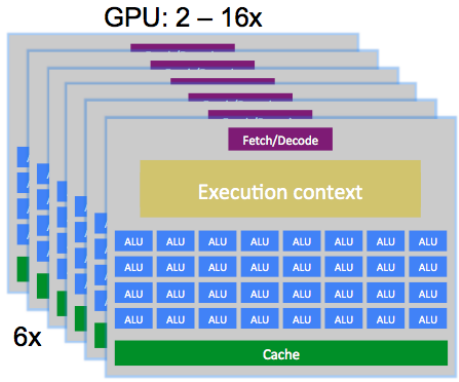


Figure 1.11: SIMD units of a generalized graphics processing unit (GPU) architecture. One layer equals one SIMD unit on a stream multiprocessor (SMX) with typically 6 SIMD units per SMX and between 2 to 16 stream multi-processors on a GPU. That gives 192 ALUs, or CUDA cores, per SMX and up to 3,072 per GPU.

Xeon Phi are both considered manycore processors. Most modern manycore processors use the SIMD processing paradigm with a general architectural layout illustrated in figure 1.11. For the general layout of a NVIDIA Kepler GPU shown here, one "layer" refers to a processing core with each core consisting of a large number of ALUs, or CUDA cores in NVIDIA language. A stream multiprocessor (SMX) consists of 6 cores each with 32 ALUs/CUDA cores, resulting in 192 ALUs/CUDA cores per SMX and between 2 to 16 stream multi-processors per GPU, giving 384 to 3,072 ALUs/CUDA cores.

The stream multi-processor may also be referred to by SMP or simply SM. The denotation SMX originates from NVIDIA for their Kepler hardware line before which they used SM for their stream multi-processors.

1.5.4 Domain Decomposition

A way to take advantage of the underlying hardware to attain faster computation speeds is to split up parts of the executed program to have each part be computed on individual processing units on the modern multi- and manycore processors. In the case of the tomography problem it is then desirable to split up the solution domain into smaller subdomains with each subdomain being assigned to a processor. This is the meaning of domain decompostion, to split up the domain into smaller parts which can then be computed individually and

gathered after parallel computations are done. In this thesis we will denote these subdomains as processor-domains.

On figure 1.12 is an illustration of how the solution volume is split into 8 processor-domains (one along the x-direction, two along y and four along z) with the projection plane on the right side in the figure.

After having a subdomain assigned to a processor it is then beneficial to further split up the subdomain into smaller parts to fit the on-board memory of the microprocessor. By having a block of the subdomain be loaded to the cache at a time it is possible to finish computations for each block before moving on to the next and not waste time and resources by continually loading data that won't yet be used and flushing the cache more often than needed to load in a new part to memory. We call these subdomains cachedomains.

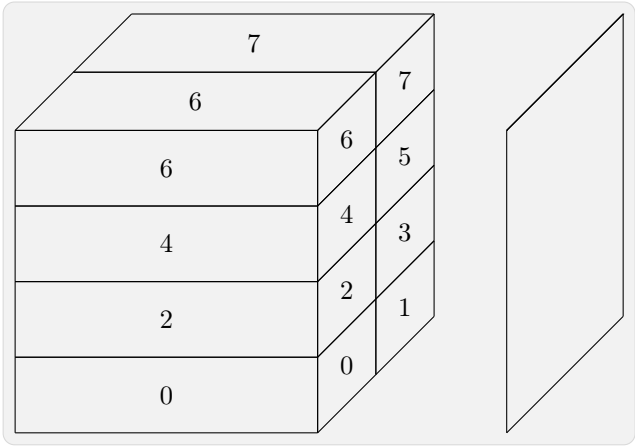


Figure 1.12: Illustrating a domain decomposition with domains $(1, 2, 4)$ along the (x, y, z) directions. The image domain is shown split into 8 domains with a projection plane of equal size along x- and y-axis as the image domain shown on the right. Each of these domains 0 through 7 can be run in parallel on independant devices accordint to the desired distribution.

This validates the decomposition of the domain into parts for each device and processor to handle and to further block the subdomains into cacheblocks that fit with the on-board memory of the device to take full advantage of the computational power of the hardware.

While this is describing the reasoning behind domain decomposition on a CPU,

the same underlying principles apply when doing parallel computations with a domain decomposition on a graphics processing unit (GPU).

1.6 Test problem(s)

When conducting computations and introducing results throughout the report, certain data and test problems is needed to facilitate the computations. This section will serve to introduce the used problems and their setup in advance.

The problem used for the numerical experimentations will consist of a real data sample of a walnut's interior scan. Argument could be made to use a test problem with a known solution for conducting the numerical experiments, but for the purpose of this project a real data sample is of little difference to a test case and further grounds the project in reality and provide a good indication to the usability and application of the implemented methods outside of a strictly academic purpose.

A test example could have been added beside the walnut data, but due to time constraints a working version with an implemented Shepp Logan test phantom was not finished in time.

Unless specified otherwise, all problems of a volume size of for example $N_x \times N_y \times N_z = N^3$ will have projections of equal size $P_x \times P_y = P^2 = N^2$.

1.6.1 Walnut

The Walnut problem consists of real measurement data from a tomographic scan with X-rays of a walnut. The measurement data has a size of 1600×1024^2 and contain 1600 cone beam projections in a uniform distribution around the object domain.

The specific resolution and number of projection angles in use as well as the number of solution interations will be specified whenever new results using the measurement data is introduced.

Figure 1.13: The walnut data reconstructed solution at 1024×1024 resolution with 1600 projection angles, 0.25 relaxation parameter and three solution iterations.

1.6.2 Test phantom

As real measurement data is not ideal in all circumstances when testing an implementation, a test case, or in the case of the tomographic image reconstruction problem a test phantom, with knowledge of the true solution is needed to determine the margin of error of the implementation. For this the Shepp Logan phantom as shown in figure 1.14 on the facing page is widely used with tomographic reconstruction techniques. Due to time constraints it was not possible to finish a working implementation of this phantom and as such it will not be used in this report.

This phantom can be generated at a chosen resolution with any number of projections at both parallel- or cone beam configuration. The parameters chosen to generate the problem will be specified at each new introduction of results using the test phantom.

That said, in consideration to the beam configuration of the walnut measurement data as mentioned in subsection 1.6.1, a cone beam configuration will be prevalent in the use of the test phantom.

Figure 1.14: Shepp Logan test phantom at 512×512 resolution.

1.7 Test hardware

All numerical performance tests are made on the GPU nodes of University of Southern Denmark's (SDU) DeiC Abacus Cluster. The cluster holds in total 72 GPU nodes distributed on 4 switches with 18 nodes on each. Each node consists of two NVIDIA K40 GPU cards.

The node configuration can be seen in table 1.1. Each GPU node has the base node configuration and GPU configuration installed.

Table 1.1: DeiC Abacus Cluster Configuration

Base node configuration	GPU configuration
IBM/Lenovo NeXtScale nx360 m5 Two Intel E5-2680v3 CPUs each with 12 CPU cores and a theoretical performance of 480 GFlop/s in single-precision. 64 GB RAM 200 GB local SSD local storage One high speed InfiniBand uplink connection for communication with the other nodes	2 NVIDIA K40 GPU cards per node, each with 2880 CUDA cores and 12 GB RAM. Theoretical performance for each K40 is 1.43 TFlop/s in double-precision and 4.29 TFlops/s in single-precision.

Figure 1.15 on the following page show the bandwidth of non-blocking MPI send and receive operations as a function of number of nodes for 1KB and 1MB

message sizes, and as a function of the message size for 2 and max number of nodes.

The full curves are for all-to-all communication and dashed curves are for neighbor-to-neighbor communication in a 1D ring topology. The curves are the average values over 1000 independent trials; the minimum and maximum values envelop the shaded regions.

It should be noted that the obtained performance depends heavily on the current load on the cluster apart from the particular settings, wiring, and make of the infiniband cards [17].

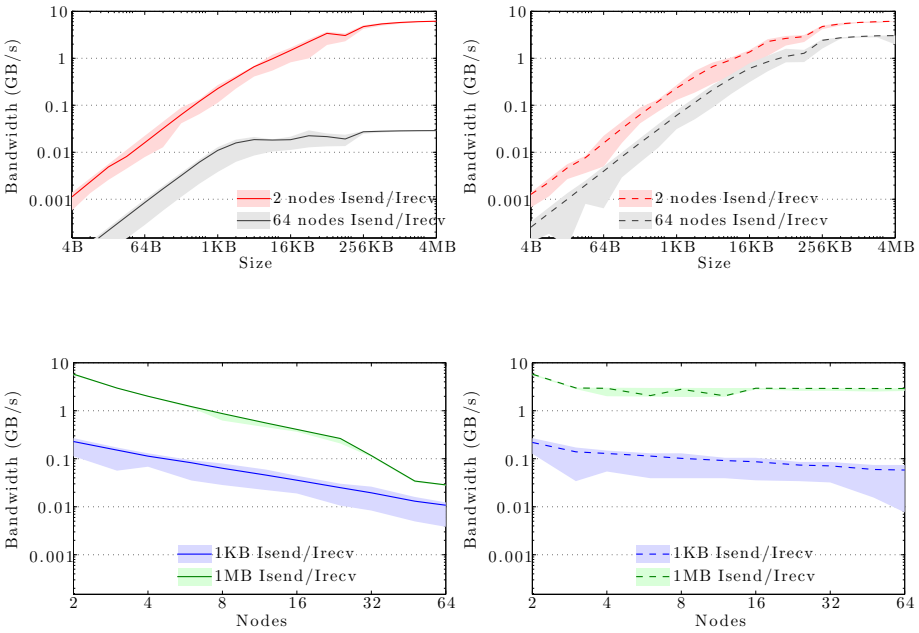


Figure 1.15: MPI benchmark of the SDU DeiC Abacus Cluster [5].

Chapter 2

Discretization

When we wish to solve a tomographic reconstruction problem, the mathematical formulation of the physics, using variables defined on the real axis, is not directly applicable for numerical computations. We are required to discretize the variables such that they model, or approximate, the physical model as closely as possible.

Looking at a discrete domain Ω, we refer to the elements in the projection array b as pixels and the elements in the volume x as voxels, arranged in a sequential manner. This way, the element, or voxel, at location $i, j, k \in \Omega$ in a volume of size $N_x \times N_y \times N_z$ is given by

$$x(i + j \cdot N_x + k \cdot N_x \cdot N_y) = X(i, j, k)$$

and the pixels in projections of size $P_x \times P_y$, corresponding to number of projection angles, are similarly stored in an array b.

Instead of using the Radon transform, we introduce a discretized model in the form of a linearised approximation A, called the projection matrix. Here, $A = (a_{ij})$ is a $m \times n$ matrix where the value of each element a_{ij} holds the contribution of the image pixel j $(1 \leq j \leq n)$ to the detector value i $(1 \leq i \leq m)$.

If our model consists of k projections, A can be expressed as $A = (A_1 \cdots A_k)$ where each matrix $A_i (i = 1 \cdots k)$ corresponds to the projection operator for the ith projection angle. The physical tomographic reconstruction problem can

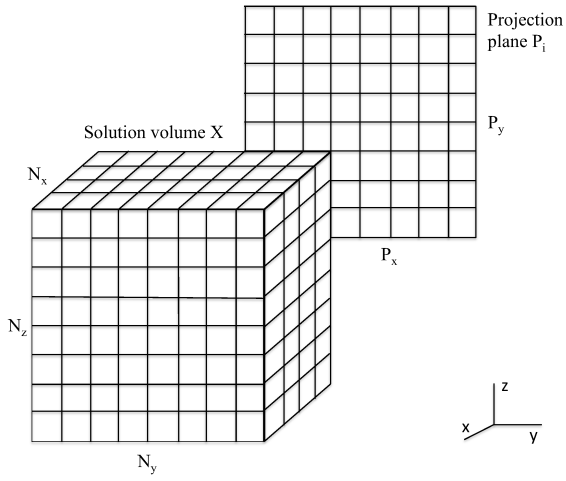


Figure 2.1: The solution volume X with the dimensions N_x, N_y, N_z and a projection plane with dimensions P_x, P_y. The i^{th} projection plane P_i is of a different coordinate system than the solution volume. As such, the x- and y-directions of the three-dimensional solution volume are not the same as for the two-dimensional projection plane.

thus be approximated by the discrete linear system, same as in equation 1.2 on page 9:

$$Ax = b, \quad A \in \mathbb{R}^{m \times n}, \quad x \in \mathbb{R}^n, \quad b \in \mathbb{R}^m, \tag{2.1}$$

where n represents the size of the image domain $N_x \times N_y \times N_z$, m the number of projections times the projection domain of size $\cdot P_x \times P_y$ and the vector b represents the measured projection data. For a parallel beam configuration A from equation 2.1 can be considered as the discretized version of the Radon transform as also mentioned previously [2].

Real applications have errors introduced in the measurements by the nature of physical instruments, which we must also add a small error to the projections in our model to be a more accurate approximation, so we get the system

$$Ax' = b + \epsilon, \tag{2.2}$$

where x' is the solution to the reconstruction problem.

Problems in CT are ill-posed by nature which means that our system will be highly influenced by noise and we as such will need a regularized solution to get a usable tomographic reconstruction. As mentioned, there are multiple reconstruction techniques available for this which will be covered more in chapter 3.

2.1 Projection Models

When constructing the discretized model for the physical tomography problem, a wide range of projection models are available for building the projection matrix A. The choice of projection model can have a large impact on not only performance but on the accuracy of the reconstructed solution as well. The goal here is to select a projection model that models the specific physical system in mind well and has a desirable run-time performance for the application.

Each projection model has its advantages and disadvantages. While one might model a particular setup very well, it could be much worse when used in a different setup. Some methods have a very fast computation speed which is very important when used in applications that require results in real-time, but these models typically make a compromise with the physical accuracy of the model. Important to note is that most methods can be optimized to compute a lot faster by taking full advantage of the available hardware, for instance by domain decomposition and incorporating priori knowledge of the problem, and by this reduce the disadvantage of a model with high physical accuracy considerably.

Below follows an introduction to some of the most well-known and widely used projection methods, among which is the *Joseph* method that this thesis puts a special focus on.

2.1.1 $0-1$ model

The $0-1$ projection model is increadibly simple in what it does. As the name suggests, the model works by simply distributing a weight of either 1 if the

tracked ray has intersected the current voxel, or 0 if it has not. This is illustrated in figure 2.2.

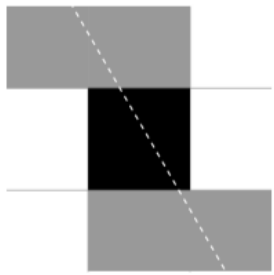


Figure 2.2: Illustration of the $0-1$ model for constructing the projection matrix. An element $a_{i,j}$ is weighted with either 1 if a ray intersects the voxel, or a 0 otherwise [2].

This is a very crude way to model the physical system as no notice is put to how much of an impact the ray has on the current and neighboring voxel. A ray might only intersect the very corner of a voxel and it will still be distributed with full weight while neighboring voxels that the ray just misses will get no weight whatsoever.

The advantage of this however, is that the model is very fast as the only computation that is needed, is to determine whether a ray hits a voxel or not.

2.1.2 Line length

This projection model, or approximation scheme, which we will refer to as the *line length* method, is quite simple in that an element $a_{i,j}$ in A is weighted as the line length of a ray intersecting that voxel, that is the length of the line that a given ray traverses a voxel from entrance to exit of the voxel. This method is illustrated in figure 2.3 on the facing page.

The disadvantage of this method is that it requires quite a bit of branching in order to determine which voxels are hit by a ray and the entrance and exit points of that voxel. Another disadvantage is that the method does not in any way consider how close to the center of the voxel the line intersects or how close to the edges, but only how far the line traverses the voxel. This can be considered a fault in our model approximation as the ray might intersect a voxel in a negligible distance from a neighbor voxel and due to floating point errors this can easily lead to one voxel being weighted 0 and the other 1. This can

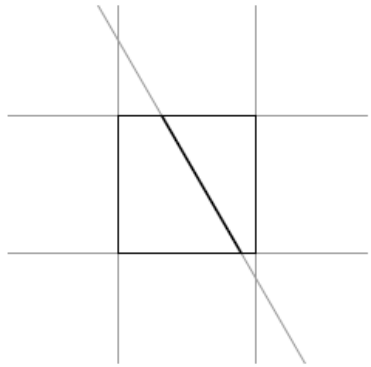


Figure 2.3: Illustration of the Line length method for approximating the projection matrix. The element $a_{i,j}$ at voxel (i, j) is weighted by the length that the ray traverses the voxel from entrance to exit.[2]

especially be a problem for rays travelling along a domain axis. As we will see in section 2.2, this causes the method to be prone to artefacts in the reconstructed images.

2.1.3 Strip area

The *Strip area* projection method weights a given voxel by the area that is spanned between two parallel rays going through the voxel. If only one of the rays actually intersects the voxel then the weight will be the area of the voxel that is between the intersecting ray and the second ray. A way to calculate the area between the rays used for the weighting of the voxel is by the triangle substraction technique which divides the area covered by the strip into smaller triangles whos areas are more easily calculated and then calculates the area of the strip by the sum of the triangle areal composing the strip. This method is covered by Nguyen and Lee [14].

The *Strip area* method is a good approximation to the physical model as the strip being projected onto a pixel on the projection plane will provide a fair weighting of all voxels within the strip. If the entirety of a voxel is covered by the strip it will provide full weight for the projection pixel while a low weight will be provided if only a corner of a voxel is covered. The disadvantage of the method is how it is dependent on a parallel beam setup for the rays within a beam to be in parallel.

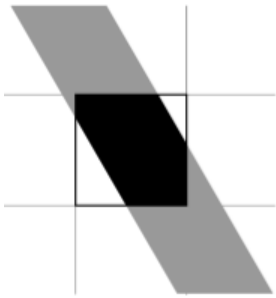


Figure 2.4: Illustration of the *Strip area* method for approximating the projection matrix. An element $a_{i,j}$ is weighted from the area inside of the voxel which lie between two parallel rays going through the voxel [2].

A voxel being intersected by two parallel rays and the spanned area used as weight is shown in figure 2.4.

2.1.4 Joseph's method

Another method that we will examine in this thesis was introduced by Joseph [10] and will be refered to as the *Joseph* method.

The *Joseph* method will contribute the measurement data from a ray going through an element $x_{i,j}$ with the contribution $w_{i,j}$ as the interpolation coefficients obtained when tracing a ray row by row (or column by column depending on the projection angle) and applying linear interpolation between the centers of the two adjacent voxels.

This way, a neighboring voxel will be weighted by the distance from the voxel center to the ray subtracted from 1, as long as a ray intersection is between this voxel and the intersected voxel's center. This is illustrated in figure 2.5 on the facing page.

The *Joseph* method is a so-called ray-driven projection method, meaning that for a given projection angle all voxels intersected by a given ray will be weighted before moving on to the next ray until all rays of the projection beam have been covered. A different approach to this is with voxel-driven methods. Here the contribution of a single voxel to the projection plane will be calculated one at a time for all voxels covered by the projection beam. An example of a voxel-driven method is the so-called *backprojection method* that is covered in subsection 2.1.5.

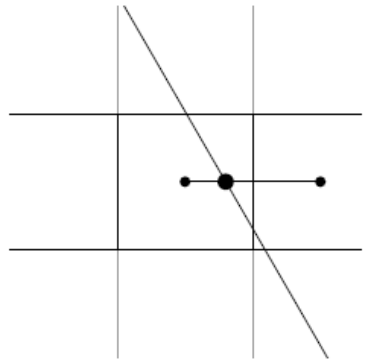


Figure 2.5: Illustration of the Joseph method for approximating the projection matrix. An element $x_{i,j}$ is contributed from the interpolation coefficients from tracing a ray row by row or column by column according to projection angle of that ray, and then applying linear interpolation between the centers of the two adjacent voxels. That means that as a ray traverses a voxel, that voxel as well as the closest neighboring voxel to the ray is contributed by (1 - ray distance to voxel center)[2].

The standard way to implement the *Joseph* method in 3D is to use bi-linear interpolation in the plane perpendicular to the dominant direction (this is faster than full trilinear interpolation and does not affect the accuracy significantly). This is illustrated for a dominant direction x in figure 2.6 on the next page.

The weights from bilinear interpolation are typically obtained in general form as

$$\begin{aligned} w_{i,j} = \quad & (1-d_x) \quad (1-d_y)x_{i,j} + d_x(1-d_y)x_{i+1,j} + \\ & (1-d_x) \quad d_y x_{i,j+1} + d_x d_y x_{i+1,j+1}, \end{aligned}$$

where the distances d_x and d_y are the composants of the distance from ray to the center of the intersected voxel in the x and y directions and are determined by the dominant direction of the ray and $x_{i,j}$ is the $(i,j)^{th}$ element of the solution matrix X for the current iteration. As for the example in figure 2.6 on the following page, the weights would then be calculated from d_y and d_z. We note that the compiler interprets this in terms of FMA (Fused-Multiply-Add) as

$$
\begin{aligned}
u &= x_{i,j} + d_x(x_{i+1,j} - x_{i,j}) \\
v &= x_{i,j+1} + d_x(x_{i+1,j+1} - x_{i,j+1}) \\
w_{i,j} &= u + d_y(v - u).
\end{aligned}
$$

Here $w_{i,j}$ is the calculated weight for the $(i,j)^{th}$ element of X.

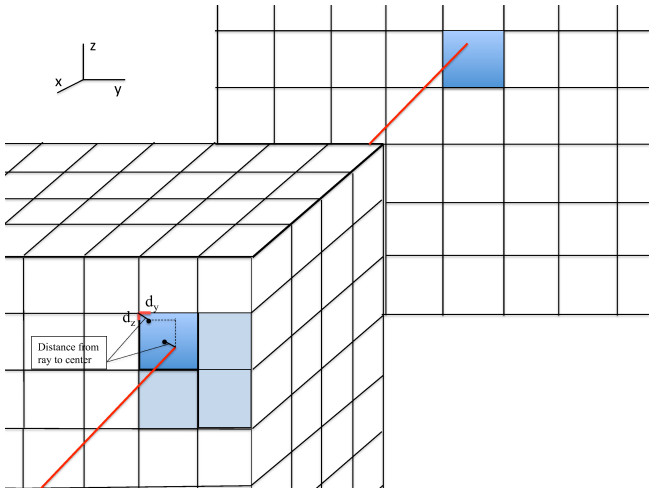


Figure 2.6: Illustration of the Joseph method and the voxels to be weighted with bilinear interpolation for a ray in 3D.

By using the weighting scheme from the *Joseph* method, the intersecting rays will "bleed" onto the neighboring pixels which will result in a less sharp and more rounded reconstruction. But it will also increase the accuracy of the approximation and decrease the chances of dark spots when encountering voxels with no intersecting rays.

2.1.5 Backprojection method

The backprojection method for construction the projection matrix has a lot of similarities with the *Joseph* method, with the main difference being that while the *Joseph* method applies bilinear interpolation for weighting a voxel and its neighbors in the solution volume, the *backprojection method's* weights are applied from the projection plane instead.

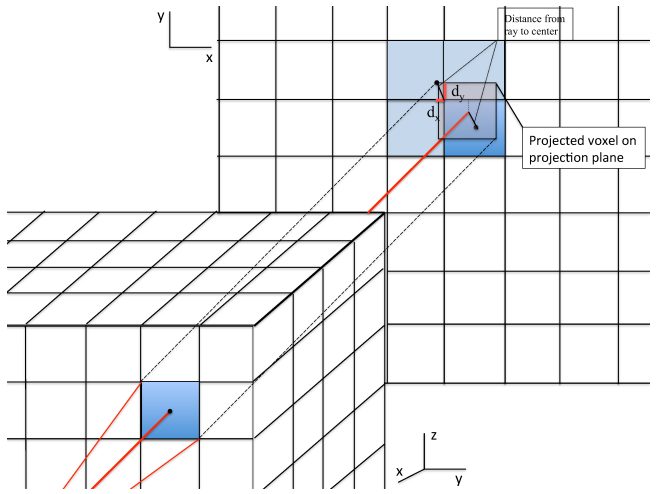


Figure 2.7: Voxel-driven Backprojection method for construction the projection matrix, here with a cone-beam setup. A voxel is projected from the solution volume onto a projection plane from the line going from the ray source through the voxel center and onto the projection plane. After the weights for one voxel has been calculated, the weights for the next voxel covered by the beam will be calculated according to the voxels projection on the projection plane and so on.

This is done by first having a given voxel projected from a line going from the ray source through the center of the voxel and onto the projection plane. The voxel will have a 2D projection on the projection plane covering parts of a number of pixels. How much of a pixel is covered by the voxel will determine the weight that is contributed from that pixel to the weighting of the voxel [16].

Like was done for the *Joseph* method in subsection 2.1.4, the weighting is calculated by determining the center position of the projected voxel and then by bilinear interpolation calculating the distance from this center to the center of all covered projection pixels, as seen in figure 2.7.

The weighting of the projection pixels can even be implemented in the same way as the weighting of neighboring voxels in the *Joseph* method, which will be covered more in chapter 4.

For the forward projection, the method will directly add the contribution of a voxel to the affected projection pixels after the weighting has been calculated from the voxel projection on the projection plane. As for the backprojection, as

the name of the projection method implies, the sum of the weights calculated from the position of the projected voxel on the projection plane will be back-projected into the solution volume and added as weighting of the corresponding voxel.

Another major difference compared to the *Joseph* method is that the *backprojection method* is a voxel-driven method. The meaning of this is that for a given projection angle, each voxel covered by the beam is in turn projected onto the projection plane. After the weights for the current voxel has been determined, the next voxel of the current row of the solution volume that is also covered by the beam will be projected onto the projection plane and have its weight calculated. This way, instead of processing one ray at a time until all rays for the current angle have been covered, each voxel that is within the beam from the current projection angle will be handled by the projection method one by one.

This way of orienting the projection method is usually computationally much slower when computing the forward projection compared to ray-driven methods, but the computation of the backprojection is in turn much faster.

2.2 Comparing the Projection Methods

As already mentioned, different projection methods will have different advantages and disadvantages over one another when used in certain circumstances. Choice of projection method can lead to greatly differing results and whether these variations are good or bad depend entirely on the implementation and the specific problem at hand. The following section will use numerical examples to illustrate, analyse and discuss how the choise of projection method can impact the reconstructed solution and the performance of your implementation.

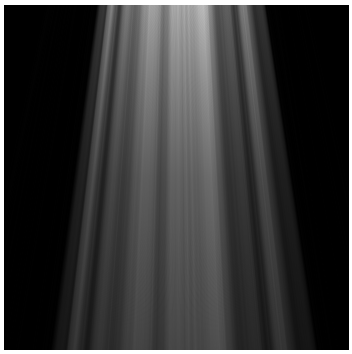
2.2.1 Reconstructions

When comparing the *line length* method with the *Joseph* method, the difference between the two are very noticeable for low iteration counts and even more so for a low number of projection angles.

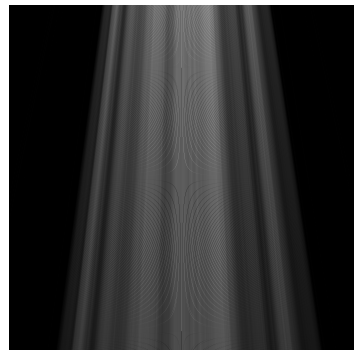
Figure 2.8 on the facing page shows the reconstructed images for the center slice of a scanned walnut, which is the measured data we will use as test problem throughout the report.

In the first two images of figure 2.8 we see the reconstruction of a 2048^3 problem with 1 iteration and a single projection angle.

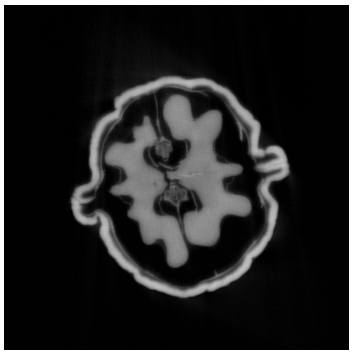
The *Joseph* method is represented in figure 2.8a and the *line length* method in figure 2.8b.



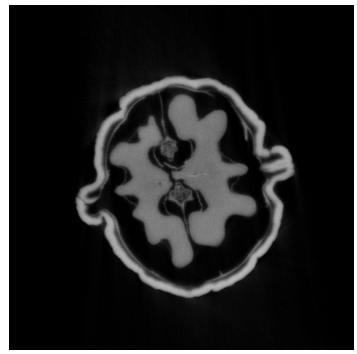
(a) Joseph kernel solution with 1 projection angle at center slice.



(b) Line kernel solution with 1 projection angle at center slice.



(c) Joseph kernel solution with 400 projection angles at center slice.



(d) Line kernel solution with 400 projection angles at center slice.

Figure 2.8: Solutions with Joseph and Line kernels at resolution 2048^3 and 1 iteration, from respectively 1 and 400 projection angles at the center slice along the vertical axis. At low projection angles it is very obvious to notice circular line artefacts in the Line kernel reconstruction whereas these are much less noticeable for the Joseph kernel. As the number of projection angles increase these artefacts become less apparent, and are mostly gone with 400 angles for the center slice along the vertical axis.

It is easy to notice the circular line artefacts present in the *line length* reconstruction, where none are present for the *Joseph* method. When looking at the reconstruction with 400 projection angles in figure 2.8d on the preceding page these line artefacts are no longer noticeable for the *line length* method.

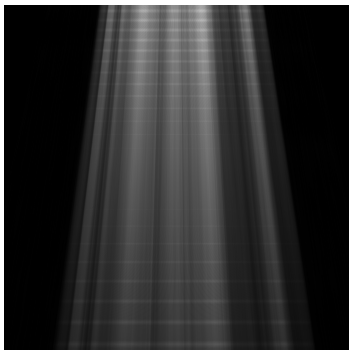
It should be noted that the 400 angle reconstruction for the *Joseph* method in figure 2.8c on the previous page is sligtly more blurred around the edges compared to figure 2.8d on the preceding page, which is also to be expected. However, when one increases the iteration count the blurriness is removed for both methods.

Not only does the iteration count and number of projection angles have a large impact on the reconstructed images, so does the angle, or level, we view the reconstruction at. In figure 2.8 on the previous page we viewed reconstructions along the center slice, which is at the same level of the ray source and center of the detector plane. When either moving down or up along the vertical axis away from the center slice, we encounter that the distance between rays of the same cone beam increases the further from the source the rays travel.

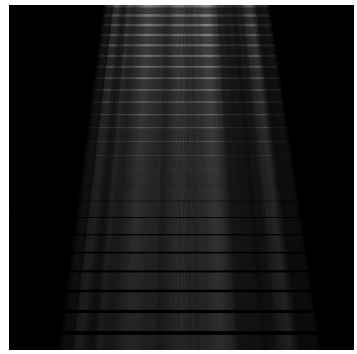
Because of this we see that some voxels farther from the source will have no rays intersecting them and that some voxels close to the source will have many rays intersecting them. With the *line length* method the voxels with no intersecting rays will not be weighted and as such be black and invisible in our reconstruction and voxels with many rays intersecting will be over exposed and weighted more heavily than other voxels. Of course this will have less of an impact as the number of projection angles increase, but can be an issue for problems with less data available.

This behaviour is shown in figure 2.9 on page 36 where we look at the reconstruction of the 50th slice away from the center along the vertical axis. Figure 2.9b on page 36 very clearly shows the behaviour of the *line length* method when the data is sparse, where with only one projection angle, whereas this is much less expressed for the *Joseph* method in figure 2.9a on page 36. As discussed in subsection 2.1.4 we observe this behaviour in a much lesser degree for the *Joseph* method because a voxel with no rays intersecting it will still be weighted by the neighboring voxels which have rays intersecting them. And when moving closer to the ray source, the voxels with many rays intersecting them will still bleed out to the neighboring voxels and thus have less of an impact.

Even when the measurement data is increased we still observe the same behaviour for the *line length* method to some degree. In figure 2.9d on page 36 the reconstruction of the *line length* method with 400 projections at slice 50 still has obvious line artefacts, whereas none are present for the *Joseph* reconstruction in figure 2.9c on the following page.



(a) Joseph kernel solution with 1 projection angle at slice 50 from center.



(b) Line kernel solution with 1 projection angle at slice 50 from center.

(c) Joseph kernel solution with 400 projection angles at slice 50 from center.

(d) Line kernel solution with 400 projection angles at slice 50 from center.

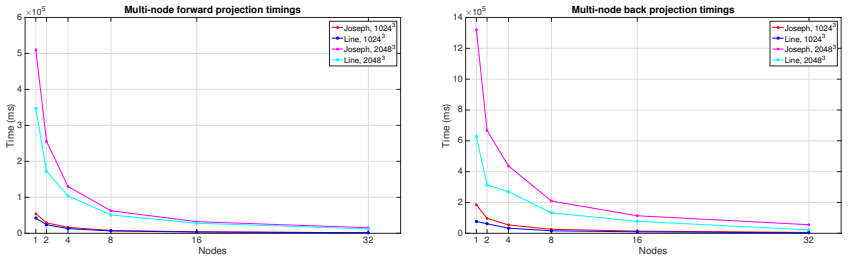
Figure 2.9: Solutions with Joseph and Line kernels at resolution $2048x2048x2048$ and 1 iteration, from respectively 1 and 400 projection angles at slice 50 along the vertical axis with 0 as the center slice. When the reconstruction is moved away from the center slice along the vertical axis, the line artefacts become much more pronounced compared to what was observed in figure 2.8 on page 33. This is especially true for the Line kernel reconstruction where there has now been introduced obvious circular lines for the 400 angles reconstruction as can be seen in figure 2.9d.

2.2.2 Performance of the forward- and backprojections

The disadvantage of the *Joseph* method when compared to the *line length* method is the longer computation time. Where the forward projection for the *Joseph* method need to update four elements (the intersected voxel and three neighboring voxels along the two non-dominant directions, to the sides and diagonally to the intersected voxel), the *line length* method only need to update one.

In figure 2.10a and 2.10b the computation timings for the forward- and backprojections respectively for 1 through 32 nodes are plotted.

The time spend on the computations drops as expected as the work is divided on more and more nodes, but we see that the *Joseph* kernel spends about 1.5 times the amount of time on the forward projections compared to the *line length* method, while for the backprojections its about twice the time.



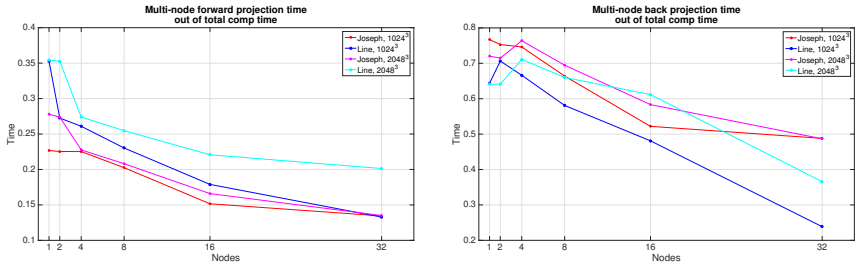
(a) Computation time spent during forward projections for Joseph and Line kernel calls.

(b) Computation time spent during backprojections for Joseph and Line kernel calls.

Figure 2.10: Computation time for problem sizes 1024^3 and 2048^3 which is spent computing forward projections **(a)** and backprojections **(b)**. The time shown is the greatest time clocked for that operation among all used nodes.

Figure 2.11 on the next page shows how much of the total computation time is spent doing forward and backprojections for the two methods for different problem sizes and node numbers.

From figure 2.11b on the following page we can see that between 75% to almost 95% of time is spent doing the backprojections when only working on a couple of nodes, while the time spent on both forward- and backprojections steadily decreases as the node number increases. This is also to be expected as communication time easily becomes a bottleneck as more nodes are added.



(a) Computation time spent during forward projections out of the total computation time for Joseph and Line kernels.

(b) Computation time spent during backprojections out of the total computation time for Joseph and Line kernels.

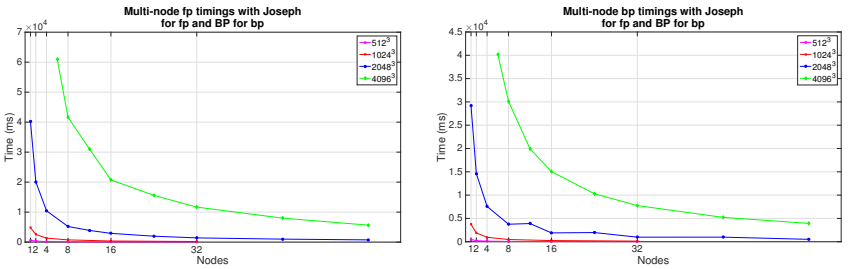
Figure 2.11: Computation time for problem sizes 1024^3 and 2048^3 which is spent computing forward projections **(a)** and backprojections **(b)** out of the total computation time.

A cause for the backprojections taking up so much computation time compared to the forward projections is due to both the *Joseph* and *line length* methods being ray-driven projection methods. While this is advantageous while doing the forward projections, it becomes a significant timesink when computing the backprojections as the method will need to write to the voxels along each specific ray in the solution volume.

This leads to having to jump around in the solution volume and continuously load in different parts to memory. A voxel-driven projection method can lead to big performance improvements in this aspect by being able to take full advantage of how a row of voxels will be loaded to the cache and then finish updating all loaded voxels before moving new voxels to memory. This is obviously more advantageous when doing the backprojection than only updating one or two of the loaded voxels before loading in the next part to the cache.

This is evident when looking at figure 2.12 on the next page showing the time for computing the forward projections with the *Joseph* method in figure 2.12a and backprojections with the *Backprojection* method from subsection 2.1.5 in figure 2.12b where the backprojections now are computed faster than the forward projections using the voxel-driven method for the backprojections.

The peaks seen for 12 and 24 nodes in figure 2.12b for the backprojections, which are not seen for the forward projections in figure 2.12a, are likely caused by communications.



(a) Computation time spent during forward projections using the *Joseph* method for the forward projections.

(b) Computation time spent during backprojections using the *Backprojection* method from subsection 2.1.5.

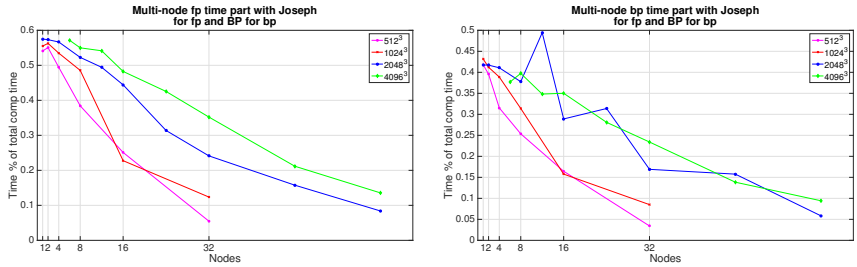
Figure 2.12: Computation time for problem sizes 512^3, 1024^3, 2048^3 and 4096^3 which is spent computing forward projections **(a)** with the *Joseph* projection method and backprojections **(b)** with the *Backprojection* projection method. The time shown is the greatest time clocked for that operation among all used nodes.

Figure 2.13 on the following page shows the percentage of the total computation time that is taken up by respectively doing forward projections and backprojections. In figure 2.13a the time spent on computing the forward projections out of the total computation time is above 50% for all the shown problem sizes using less than 4 nodes with the time spent being higher the larger the problem size.

As seen in figure 2.13b, the time spent on backprojections is close to 40% for all the problem sizes using two nodes or less. Compared to using the ray-based *Joseph* method for both forward and backprojections it gives a big performance improvement to instead use the voxel-driven *Backprojection* method to do the backprojections. Simultaneously, how the *Backprojection* method approximates the physical model is a lot like how the *Joseph* method does leading to the methods complimenting each other well as a forward-/backprojection pair.

From both figure 2.12 and figure 2.13 on the following page can be seen how the computation times for the forward projections and backprojections decrease as more nodes are included in the execution. This is both because the amount of work is distributed on more computation nodes and because communications between nodes and devices play a larger role as more nodes are added.

This can be seen by how the percentage of time spent doing both forward- and backprojections out of the total computation time decreases with the inclusing of more computation nodes in figure 2.13 on the next page.



(a) Computation time spent during forward projections using the *Joseph* method for the forward projections out of the total computation time.

(b) Computation time spent during backprojections using the *Backprojection* method from subsection 2.1.5 for the backprojections out of the total computation time.

Figure 2.13: Computation time for problem sizes 512^3, 1024^3, 2048^3 and 4096^3 which is spent computing forward projections **(a)** with the *Joseph* projection method and backprojections **(b)** with the *Backprojection* projection method out of the total computation time.

We will look at the impact and cause of barrier timings which is where communications are included in subsection 5.2.2.

Chapter 3

Block-methods

In image reconstruction there are a lot of different approaches and methods available, and the choice and implementation of these will have a big impact on performance. Our focus is primarily on large-scale 3D problems involving, at times, huge amounts of data, both in terms of measured detector images and the final reconstruction volume. In this chapter, we will discuss so-called Block-methods, which are well suited for large-scale problems.

What we refer to by block-methods, are methods which split the input data into blocks and use the blocks to sequentially update the solution for each subsequent block. This is not to be confused with methods which just partition the input data to fit on the computation device but without updating the solution as these blocks are applied.

After discretization and construction of our projection model from the linear system in equation 2.1 on page 24 many reconstruction algorithms will revolve around the following two operation:

- The *forward projection* of x consisting of the matrix-vector multiplication $f_p = Ax$.

- The *back projection* of f_p consisting of the matrix-vector multiplication $b_p = A^T b$, where A^T denotes the transpose of A.

By the forward projection operation the projections f_p for the image x is simu-

lated, while the back projection will distribute each projection along the given projection line or angle and sum up the contributions from all of these projections in the given voxel that they travel through.

In some implementations the forward projections f_p are computed for the domain followed by a correction $f_{p_{corr}} = (b - f_p)\lambda \circ f_w$, where f_w is an array of weights from the forward projection corresponding to the elements of f_p, λ is a relaxation parameter and $\circ$ is element-wise multiplication. For example for the Cimmino method, as mentioned in subsection 1.4.1, algorithm 1 on page 11, the weights f_w correspond to $diag(AA^T)^{-1}$.

We determined in section 1.4 that we would like a reconstruction method that is suitable for parallel computation on multicore and especially manycore architecture as well as has a fast convergence rate. We then looked briefly at the two methods SIRT and ART, where the problem we face is that while SIRT is suitable for a parallel implementation, it has a slow convergence rate, and the other way around for ART.

3.1 Block-Iteration method

We now wish to combine the two methods SIRT and ART, introduced in section 1.4, so that we get the convergence of ART and the parallelism of SIRT in a block method that is suitable for HPC application. To combine them in a block method, we split A and right-hand side b of our linear system from equation 2.1 on page 24 into p blocks so we get:

$$A = \begin{pmatrix} A_1 \\ A_2 \\ \vdots \\ A_p \end{pmatrix}, \quad \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_p \end{pmatrix}, \quad A_\ell \in \mathbb{R}^{m_\ell \times n}, \quad \ell = 1, \ldots, p. \tag{3.1}$$

Here m_ℓ is the number of rows of the ℓth block, A_ℓ. The methods can then be combined by making sequential updates from each of the blocks of A, similar to how ART treats rows in A, using SIRT updates. This is called a block-sequential method and was initially described by Elfving [7].

We then have a generic algorithm of the form

Algorithm 3 Generic block-sequential

$x^{k,0} = x^{k-1}$
for i **in** $1 \to m$ **do**
$x^{k,i} = P_C\left(x^{k,i-1} + \lambda T A^T M\left(b_i - Ax^{k,i-1}\right)\right)$
end for
$x^k = x^{k,m}$,

where matrices T (computed from A) and $M_\ell \in \mathbb{R}^{m_\ell \times n}, \ell = 1, \ldots, p$, as mentioned in subsection 1.4.1, define which SIRT method is used for the blocks.

Alternately, we can simultaneously process all the blocks of A like how SIRT treats the rows of A, using ART updates on each block instead, and we would have a block-parallel method. Such a method has been described by Gordon and Gordon [9].

The study done by Sørensen and Hansen [19] shows that by choosing a block-sequential structure we can utilize that our "building blocks" are SIRT iterations to have a method that is suited for multi-core, while the error reduction per iteration is still close to that of ART.

By using a block-parallel approach we get ART iterations on each block which are performed independently and in parallel, and then conbined linearly in the end to produce the result of an iteration.

3.1.1 BLOCK-IT

In this thesis we use a particular case of the generic block-sequential method in 3, where T and M are defined as

$$T = I \quad \text{and} \quad M = diag\left(\frac{1}{\|a_i\|_2^2}\right),$$

which is the same as for the Cimmino method as mentioned in subsection 1.4.1. Furthemore, the number of blocks can be freely chosen.

This choice of block method has the advantage that it does not require the computation of how many rays will intersect a given voxel in the backprojection. This is typically needed for other choices of T, see e.g. SART in subsection 3.1.2.

3.1.2 SART

The Simultaneous Algebraic Reconstruction Technique (SART) is another special case of BLOCK-SEQUENTIAL in algorithm 3 on the previous page where T and M are defined by the 1-norm of the elements of A. In the original article by Andersen and Kak [1] one block is equal one projection, but generally the number of projections per block is completely optional.

Algorithm 4 SART

$x^{k,0} = x^{k-1}$
for i **in** $1 \rightarrow m$ **do**
    $x^{k,i} = P_C\left(x^{k,i-1} + \lambda T A^T M\left(b_i - A x^{k,i-1}\right)\right)$
end for
$x^k = x^{k,m}$,

where T and M are defined by the 1-norm $T = diag\left(\|\text{column}_j\|_1\right)$ and $M = diag\left(\|\text{row}_i\|_1\right)$.

Simultaneous Algebraic Reconstruction Technique, or SART, is the name given the method by the linear algebra community. In the tomography world and for ASTRA the same method as algorithm 4, with the same T and M but for only a single block, is referred to as SIRT, which can be a cause for much confusion.

3.2 Advantages of block-sequential methods

As previously mentioned, the goal of combining the ART and SIRT methods into a block algebraic iterative reconstruction method based on a partitioning of the linear system is to retain the better multicore properties of the SIRT method while combined with the faster semiconvergence of the ART method into a single method.

Block methods such as these are separated into two distinct classes. The first class consists of the methods that access the blocks sequentially in each iteration and use the updated solution from the previously computed blocks at each new block. The second class of methods refers to those that compute a separate result for each block in parallel and at the end combine the results before going to the next iteration.

The first class of methods will be referred to as block-sequential methods and the second class as block-parallel methods. Computational results by Sørensen and Hansen [19] have shown that the block-sequential methods give preferrable results when executed on multicore architecture compared to the block-parallel methods.

By using the updated solution when computing each new block, the block-sequential methods obtains a much faster reduction of the error per iteration compared to the block-parallel methods where each block is computed independently.

While many will only use a single projection per block which, as mentioned, is also introduced in the original article by Andersen and Kak [1], experience tells that it can be advantageous to use between one to four or more projections per block depending on the total number of projections. As one can tell, the projections when using four projections per block will be located in a close vicinity and without much deviation in the projected angle if many hundreds of equidistant projections are available.

As mentioned, all of the following convergence tests have been made on the tomographic walnut data, meaning that they are conducted on experimental data and as such can not show the true convergence of the methods. They can however conclude on how the methods fare and compare in a more real-to-life scenario.

The relative error of the image reconstructions are defined from the Frobenius norm of the solution at each iteration step compared to the zero-solution. Be-

cause of the semi-convergence of the methods, these norms will theoretically move closer to the true solution of the problem and stabilize before moving toward the naivé solution. Because we are dealing with experimental data however, we can not verify semi-convergence from these graphs. Nevertheless, we can assume to be close to the true solution once the Frobenius norm of our reconstructions stabilize and get closer to a horizontal curve. In the convergence plots below, the solution norms have been plotted against the smallest among the error norms of the solutions.

When using block algebraic iterative reconstruction methods to solve problems in tomography, the ill-posed nature of the problems makes regularization techniques a very impacting factor in the quality of the reconstructed solutions. The choice of the relaxation parameter for the methods is a study in and of itself, and while it is not a focus of this work it is nonetheless important to be aware of the impact this choice will have on the obtained results.

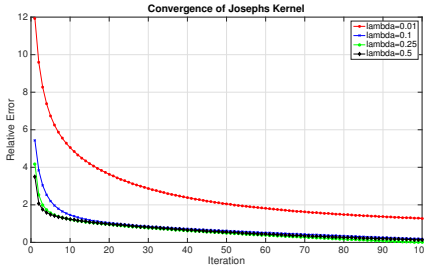
The following section will illustrate how the relaxation parameter can impact the reconstructed solution and how this choice will need to be adaptive when problem parameters change, for example between underdetermined and overdetermined systems.

3.2.1 Underdetermined system

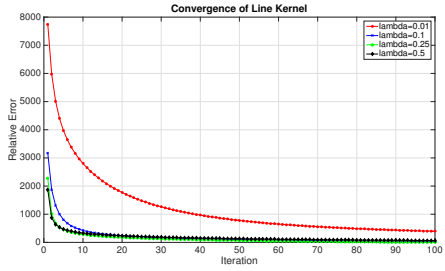
Figure 3.1 on the next page show the convergence of the *Joseph* and *line length* implementations as well as the image reconstructions at 3, 6, 9 and 12 iterations of the Walnut data from section 1.6. These results are from a problem of size 256^3 with 200 projection angles.

The convergence shown in figure 3.1a and 3.1b is not the true convergence of the solution as that is unattainable for a real-case problem with noise introduced to the data. Instead it is the relative error based on the Frobenius norm of the solution obtained after 100 iterations, which also explains why it converges at 0 on the figures.

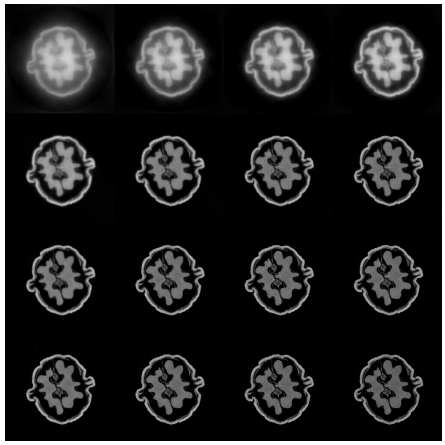
While this is not the true convergence of the methods, as it would also be expected to observe semi-convergence of the iterative method, it can as previously mentioned be used to see where the most work is done during the 100 iterations, or where the most results are obtained with highest efficiency throughout the iterations.



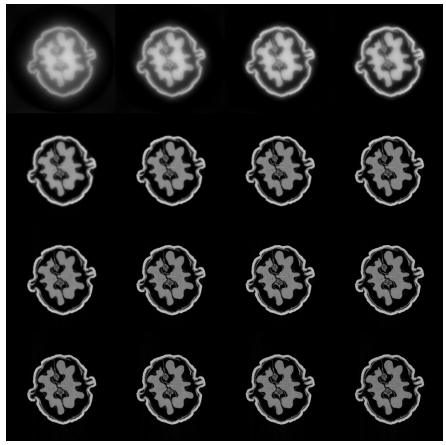
(a) Convergence Joseph



(b) Convergence Line



(c) *Joseph* image reconstructions



(d) *Line* image reconstructions

Figure 3.1: Convergence of the *Joseph* **(a)** and *line length* **(b)** kernel for a problem of size 256^3 with 200 projections. The relative error is defined as the Frobenius norm of the solution compared to the zero-solution. These norms have been plotted against the highest Frobenius norm obtained through the 100 iterations, which is when the convergence has stabilized and moves no closer to the true solution. As these results are conducted on a real world data case, it is not possible to calculate the convergence against the true solution and from this show the semi-convergence of the method. *Joseph*, **(c)** and *line length* **(d)** methods image reconstructions of the center slice along the vertical axis for a 256^3 problem with 200 projection angles. From first to fourth row is: $\lambda = 0.01$, $\lambda = 0.1$, $\lambda = 0.25$, $\lambda = 0.5$, while from first column through fourth is: 3rd, 6th, 9th and 12th iterate.

It can be seen that the convergence happens very fast for both *Joseph* and *Line* methods. That the two methods behave very similarly is to be expected as the same reconstruction technique is used. The most noteworthy is the difference in behaviour by using different relaxation parameter and how this is very dependant on the system size and how determined it is - in this case how many projection angles are included.

From the first to fourth row of figure 3.1c and 3.1d on the preceding page with the relaxation parameter $\lambda = 0.01,\ 0.1,\ 0.25$ and 0.5 from first to fourth column. We can see from figure 3.1a and 3.1b that a larger relaxation parameter is suitable for a system at close to full rank and that our solution is close to convergence after only a couple of iterations. Here a λ between 0.25 and 0.5 seems to be ideal for both methods.

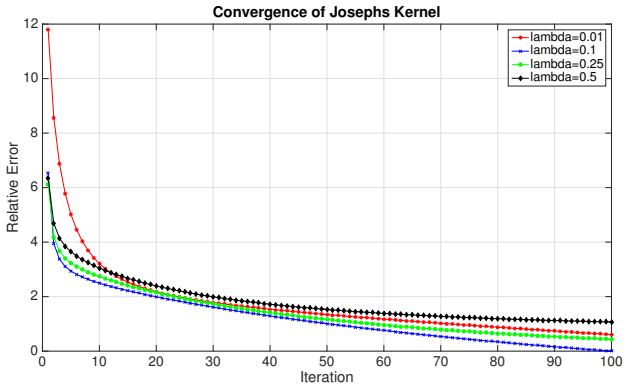
Figure 3.1c and 3.1d also show that the solution is not desirable for a low λ and that the methods converge and does not improve noticably after a couple of iterations for a well-chosen relaxation parameter.

3.2.2 Overdetermined system

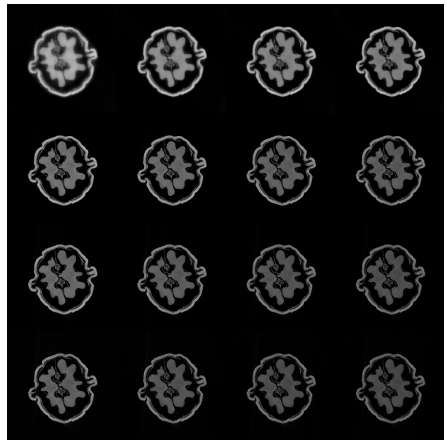
It can be seen from figure 3.1 on the previous page how an underdetermined system will benefit from a larger relaxation parameter, and while there is not much difference between going from 0.5 to 0.1, when the value drops to 0.01 it will have a large negative impact on the solution.

Figure 3.2 on the facing page shows the reduction in relative error and the reconstruction results for a larger and more overdetermined system of size 1024^3 with 1600 projection angles.

From figure 3.2a on the next page we see that a lower relaxation parameter works better for an overdetermined system, where we here have an ideal λ at 0.1. The reconstructions are obviously not good for $\lambda = 0.01$, but for the other λ values it is harder to see noticeable differences. And compared to the big gap for an underdetermined system in figure 3.1 on the preceding page when going from 0.01 to 0.1, this gap has been largely reduced for the overdetermined system as can be seen in figure 3.2 on the next page



(a) Convergence Joseph



(b) Convergence Joseph

Figure 3.2: Convergence of the Joseph kernel for a problem of size 1024^3 with 1600 projections. As for figure 3.1 on page 47 the relative error in figure **(a)** is defined as the Frobenius norm of the solution compared to the zero-solution. These norms have been plotted against the highest Frobenius norm obtained through the 100 iterations, which is when the convergence has stabilized and moves no closer to the true solution. As these results are conducted on a real world data case, it is not possible to calculate the convergence against the true solution and from this show the semi-convergence of the method.

3.3 Block reconstruction software

For this thesis we have developed a block reconstruction software based on Algebraic Iterative Reconstruction (AIR) for 3D tomography. The development consisted of modifications to an existing software that implemented the special case SART algorithm. The software works for both CPU and GPU computation and is intended for 3D tomographic reconstruction using High-Performance-Computing facilities.

3.3.1 General structure

Figure 3.3 on the next page shows the general structure of the implemented block-sequential software framework followed by a more detailed look at the block loop. The following main steps are executed when using the software.

- First we decompose the domain, which is to split the solution domain X into subdomains $X_1, X_2, \dots X_k$, denoted processor-domains, where a processor-domain will typically represent a computation node or a device that will be tasked with handling computations for that specific domain. Data are then loaded onto the device as indicated by the domain decomposition.

- Then communications are initialized, such as allocating memory on host and devices for x, f and b.

- The main algebraic reconstruction loop over iterations are then started in parallel for each processor-domain.

- The total number of projection angles are split into blocks of a partition of projection angles. E.g with 400 projection angles these can be split into 200 blocks with two projection angles per block or 100 blocks with 4 angles per block. Each device now loop through all of these blocks until having covered all projection angles for each solution iteration.

- For the current block of projection angles, the projection angles that actually intersects the current processor-domain are determined and the kernel for the chosen projection method is executed for each angle with the task

of calculating the forward projections.

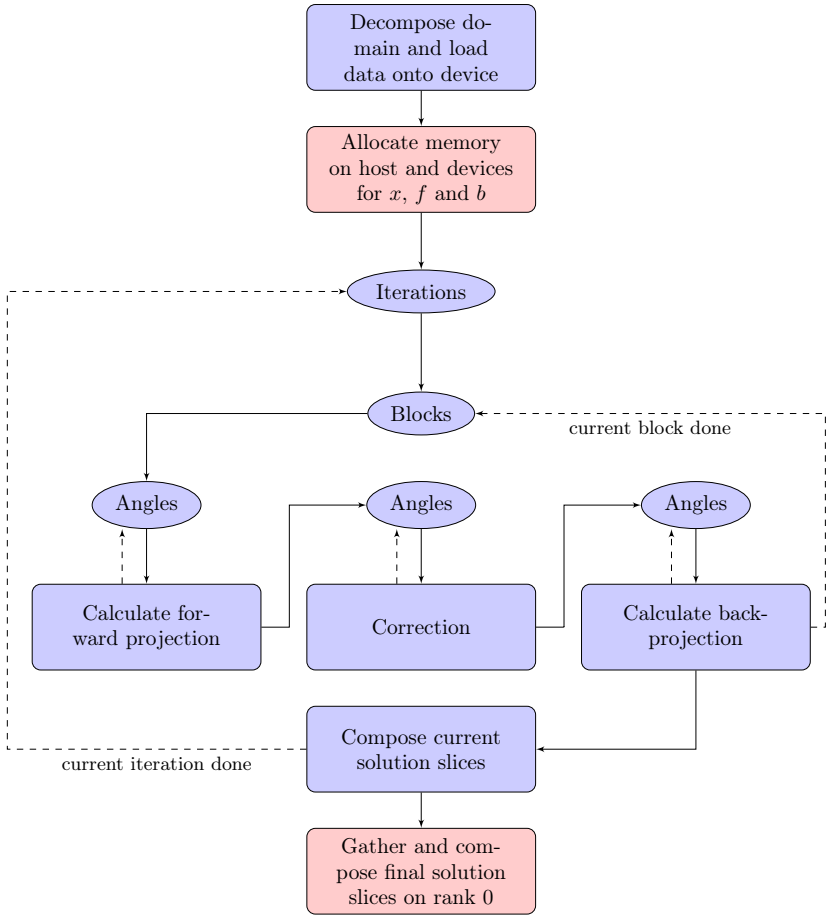


Figure 3.3: Block reconstruction flowchart; blue boxes show required steps of the algebraic iterative reconstruction and red boxes communication and overhead steps. Round boxes and dashed lines represent loops

- After the forward projections for all angles intersecting the processor-domain have been computed, correction is calculated for the current forward projections of the processor-domain.

- When all forward projections have been corrected, the projection kernel is again executed for all rays intersection the processor-domain with the task of calculating the backprojections.

- After all backprojections have been calculated for the current processor-domain, the loop continues to the next block of projection angles until all blocks are done.

- Once this is accomplished, the current solution slices are composed and the next iteration will start.

- Once the last iteration finishes, the final solution slices for the entire solution volume will be gathered from all devices and composed on the main device (MPI rank 0).

While figure 3.3 on the previous page illustrates the overall structure of the reconstruction, the inside of the block loop is expanded in figure 3.4 on the next page.

3.3.2 The block loop

The main steps of the detailed block loop of the block reconstruction software are given by:

- For each block, consisting of a number of projection angles, a forward projection, correction, and backprojection will be calculated.

- After decomposing the domain into processor-domains, each processor-domain will consist of a number of equally sized cachedomains, covering the entirety of the processor-domain for the given device.

- The forward projection for the current block will loop through these cachedomains and calculate the forward projection for all projected rays hitting this cachedomain with a call to the projection kernel, for instance the Joseph kernel.

- After the computations, the forward projections are communicated between all nodes and the calculation of the forward projections for the next cachedomain will start.

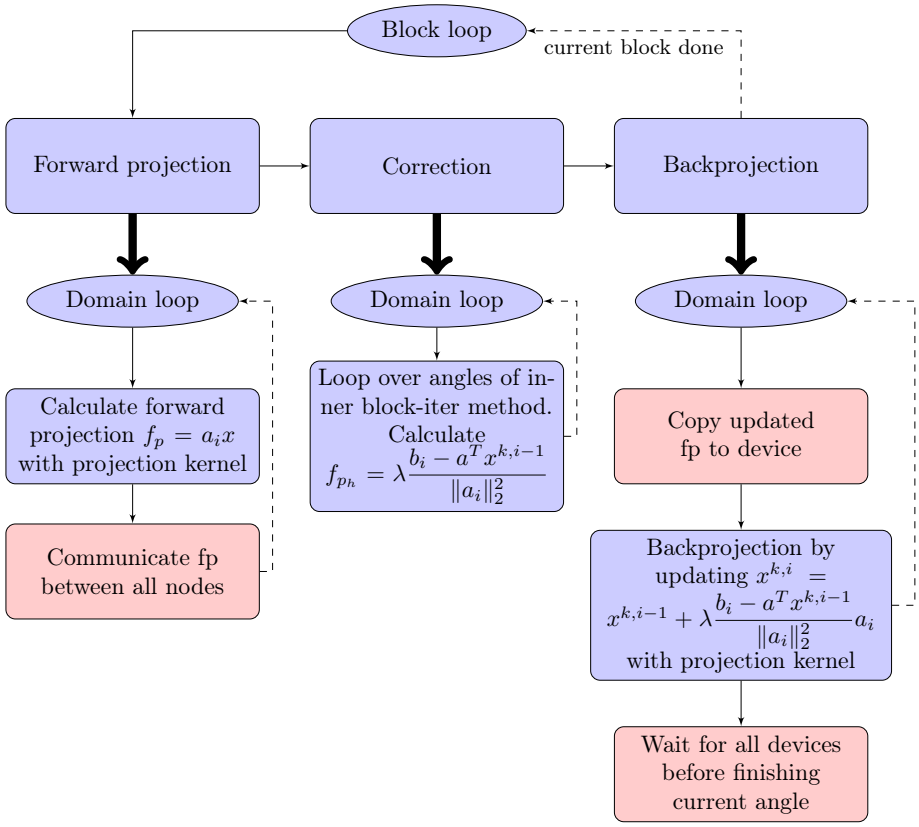


Figure 3.4: Block reconstruction flowchart for block loop; blue boxes show required steps of the algebraic iterative reconstruction and red boxes communication and overhead steps. Round boxes and dashed lines represent loops

- Once the forward projections are done, the correction $f_{p_{corr}}$ will be made for all angles of the current block and cachedomains in the processor-domain for the device before moving on to the backprojection.

- As the correction step finishes, the backprojection will be executed by again looping over all projection angles of the current block. For each cachedomain in the processor-domain for the device, the updated forward projection is copied to device and the backprojection is calculated by updating the image X with a call to the projection kernel.

- Once the backprojection is done for all cachedomains, we wait for all devices to finish before exiting the current block and continuing to the next with calculating the forward projections.

Chapter 4

Implementing the *Joseph* method

When developing High-Performance Computing applications one has to constantly bear in mind the hardware that is going to execute the application to achieve the highest level of performance possible. For modern applications, this hardware is multicore and manycore architectures.

During development, the areas to focus on is the parts that should be executed in parallel and how to avoid data races. When and where is data needed in memory and how to access it as quickly as possible when it is needed for computation. If the implementation is meant to be executed on multiple devices a further level of complexity is added to the structure of the implementation.

The *Joseph* projection kernel has been developed with these aspects in mind and with the option of being executed with domain decomposition across multiple devices and computation nodes to be relevant in High-Performance Computing.

4.1 Building the kernel

The method is implemented to act as a kernel that can be called for any size subdomain (e.g. cacheblocks) on device or processing unit and is therefore meant to be as performance oriented as possible. When the kernel is called, the leading direction (fastest direction ray travels) is passed to the kernel as well as

the beam configuration (parallel or cone) and if the current operation is forward or back projection.

The *Joseph* kernel implementation consists of an outer loop going through all rays that might hit the current domain and an inner main loop body that goes through the domain along the voxels hit by a ray. As covered in subsection 2.1.4, the *Joseph* method is a ray-driven projection method and as such follows the voxels hit by a single ray at a time compared to tracking all the rays going through a single voxel at a time.

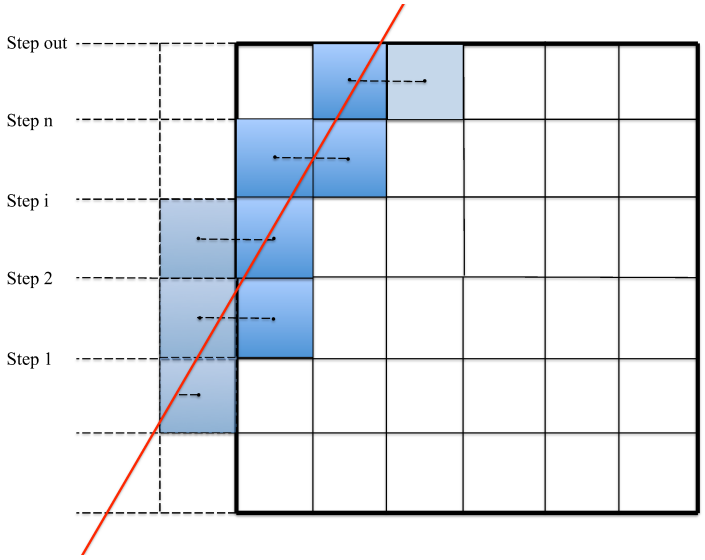
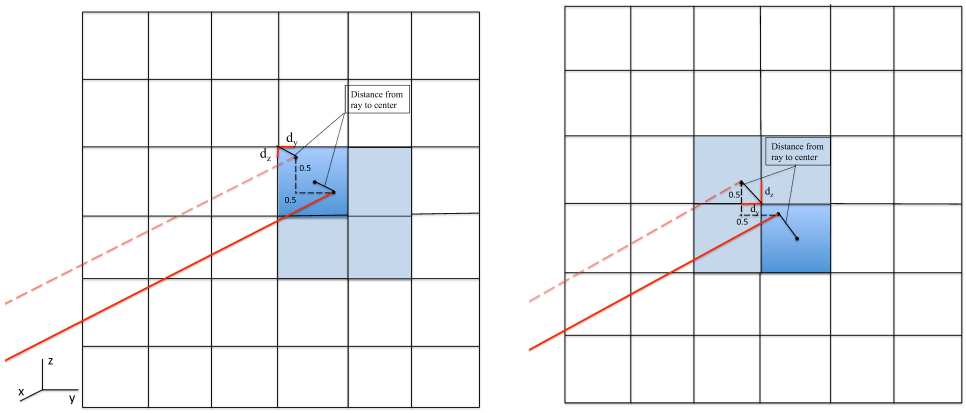


Figure 4.1: Illustration of the implemented *Joseph* steps. Position is moved to first intersection of the zero-boundary, and if the ray intersects the zero-voxel between the voxel center and the domain boundary then *Joseph* steps start here.

Before entering the outer loop we add 1/2 voxel on each side of the domain in the non-leading directions and subtracts 1/2 voxel from the current position to account for the zero-boundary. This position correction due to the zero-padding layer that is added to the domain insures that the accessed indices does not go out of bounds during the kernel call.

As the outer ray loop is entered, the position of the ray at its origin as well as the ray direction is calculated. Next we compute the ray's nearest and farthest intersection with the domain and check whether the distance that the ray traverses the domain is above a set threshold to make sure the ray intersects the domain.

Before entering the main loop body, the position is now moved to the first intersection of the domain and we determine how many voxels are in the leading direction from the "bottom" of the domain until exiting it. We need this because we instead of calculating which voxels the ray hits, we instead step from the "bottom" of the domain and up in the leading direction and only make computations if the ray has entered the domain. If it is determined that the ray has yet to enter the domain or the zero-padding boundary, we take a step in the leading direction until the ray intersects the domain before doing any computations. This is illustrated in figure 4.1 on the facing page.



(a) The Joseph method when the ray hits a voxel "below" the center of the voxel

(b) The Joseph method when the ray hits a voxel "above" the center of the voxel

Figure 4.2: Illustration of how the Joseph method's weighting of voxels is implemented in 3D where the ray is travelling in a dominant direction x and intersects a plane spanned from the other two directions y and z. For the Joseph method, the distances from an intersecting ray to the center of the voxel and neighboring voxels are needed. As for the implementation of this, the ray intersection point is shifted half a voxel left and up (half a step back along row and column). This makes it easier to calculate the distance to the center of the voxels as this distance is now just the distance to the upper left corner of the voxels from the new ray intersection point. As for what voxels are to be weighted, this is determined by what voxel the new ray intersection point is located in as shown in **(a)** and **(b)**.

After making sure the ray is in the domain, the weights are calculated for the current and neighboring voxels where applicable and summed up in the forward projection. This weighting scheme from the *Joseph* method is illustrated in figure 4.2. As a ray intersects a voxel, the distance from the ray to the center

of the intersected voxel, let's call this $d_{current}$, as well as the neighboring voxel, $d_{neighbor}$, is calculated. The weight, w, is now determined by

$$\begin{aligned} w_{current} &= (1 - d_{current}) \\ w_{neighbor} &= (1 - d_{neighbor}), \end{aligned}$$

and added to the corresponding voxels. As $d_{current} + d_{neighbor} = 1$ it is only necessary to determine $d_{current}$, as $w_{neighbor}$ can then be found by $w_{neighbor} = d_{current}$.

As can be seen in figure 4.2a on the preceding page a hit voxel will have 3 neighbors in the 3-dimensional case. For ease of implementation, the ray intersection point is shifted 0.5 units back along the indominant directions. It is then easy to see that the distance from the original intersection point to the center of each implicated voxel will be the same distance as from the shifted intersection point to the upper right corner of each implicated voxel - which conveniently also corresponds to the indices of the voxels.

Shifting the ray intersection point by 0.5 units will still affect the same voxels as originally even if the shiftet point is located in a neighboring voxel from the original as can be seen from figure 4.2b on the previous page.

For the forward projection the summed weights will be summed up for the current ray as follows:

$$\begin{aligned} f_p+ =& (1 - d_y) \cdot (1 - d_z) \cdot X(x, y, z), \\ f_p+ =& \quad d_y \cdot (1 - d_z) \cdot X(x, y + 1, z), \\ f_p+ =& (1 - d_y) \cdot \quad d_z \cdot X(x, y, z + 1), \\ f_p+ =& \quad d_y \cdot \quad d_z \cdot X(x, y + 1, z + 1), \end{aligned}$$

where f_p is the summed weights for the current projection angle, d_y and d_z are the distances from the ray to voxel center in y and z directions (as can be seen from figure 4.2a on the preceding page), and $X(x, y, z)$ is the value of voxel (x, y, z) of the solution volume. For example in figure 4.2a on the previous page $(1 - d_y) \cdot (1 - d_z)$ will be the distance from the ray to the center of the top right voxel that is actually hit by the ray and so on.

If the kernel is called for the back projection, we instead multiply the weights to the forward projection and add to the corresponding voxel in the solution:

$$
\begin{aligned}
X(x,y,z) &= (1-d_y)\cdot(1-d_z)\cdot f_p(pixel),\\
X(x,y+1,z) &= d_y\cdot(1-d_z)\cdot f_p(pixel),\\
X(x,y,z+1) &= (1-d_y)\cdot d_z\cdot f_p(pixel),\\
X(x,y+1,z+1) &= d_y\cdot d_z\cdot f_p(pixel),
\end{aligned}
$$

where $pixel$ is the specific detector pixel hit by the current ray for the current projection angle and $f_p(pixel)$ is the summed weights added to the forward projection for that detector pixel and ray.

After these steps has been taken we step to the next layer in the leading direction and continues until the current ray exits the domain. This process is illustrated in 2D in figure 4.1 on page 56.

The GPU implementation of the kernel can be seen in appendix A on page 77.

4.2 *Joseph* method using domain decomposition

When implementing the *Joseph* method it is important to decide on how to handle the boundary region of the domain as the method will want to write to neighboring elements. This is especially important when implementing the method to work with domain decomposition as multiple boundary regions will appear inside of the solution domain as the domain is decomposed into smaller subdomains and distributed to separate computation units.

As previously mentioned, a problem that can be difficult to solve when implementing the *Joseph* method is how to handle when a computation unit responsible for a certain subdomain wants to write to an element of a neighboring subdomain. To deal with this problem one could choose to look inside of the domain and basically use the outer layer of each subdomain as a boundary layer for the method, a so-called "inverted" boundary. This would however be a poor solution and especially so in the case of domain decomposition as it would result in a loss of data for each single subdomain within the solution domain.

Instead one can add an extra outer boundary layer to each subdomain consisting of 0-elements as we don't actually want to write anything to these elements - a so-called zero-padding.

4.2.1 Explicit zero-padding

When a ray in the *Joseph* method reaches a voxel on the boundary of the domain, we are faced with a problem of how to evaluate the weighting of the neighboring voxel on the outside of the domain. Likewise, if a ray intersects a voxel just outside of the domain, we will need a method to evaluate if the neighboring voxel on the boundary of the domain should be weighted or not. To help us with this, we will add an extra layer of one voxel deep to the boundary of the domain in every direction and call this the *zero-padding*.

This extra layer will consist of zero voxels because the weighting from the zero-padding is not to be included in our projections. For voxels that exist on the boundary of domains neighboring the current domain, this weighting is already done in the corresponding neighbor domains. And as for voxels that are outside of the entire solution volume there are no need for them to be weighted.

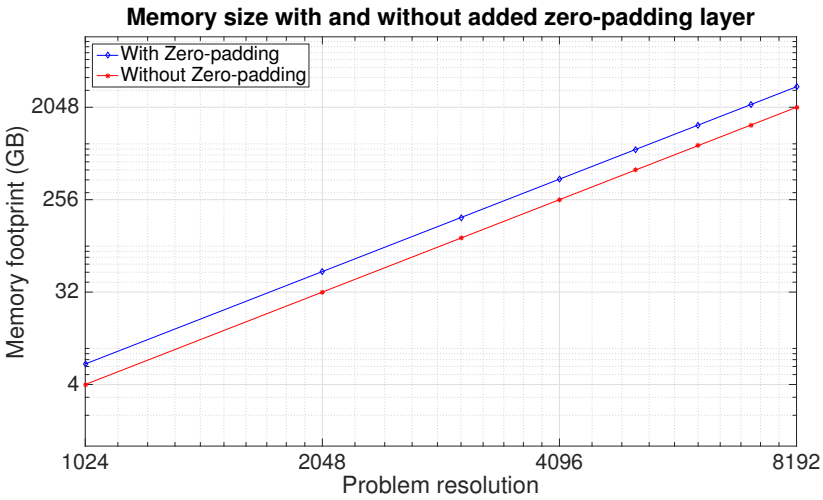


Figure 4.3: Memory footprint in GB for different problem sizes with and without zero-padding. The zero-padding layer is added to the cacheblocks which for these results are of size $8 \times 8 \times 128$ before added zero-padding.

Adding an extra layer of one voxel deep to the domain doesn't sound like it will increase the memory size of our problem by much. However, when we take the lowest level domain decompostion over cacheblocks into account, this potentially introduces a big memory barrier for large problems when every cacheblock in the entire domain will receive an extra layer, making the problem unnecessarily larger.

When running on the GPU where the on-board memory is still limited (<12GB) this quickly becomes an issue in terms of the size of the problem we can solve on a single node. As can be seen from figure 4.3 on the facing page showing the memory footprint as a function of problem resolution with and without zero-padding for cacheblocks of size $8 \times 8 \times 128$, the needed system memory increases exponentially as the problem size increases. This is even more evident when zero-padding is added as the number of cacheblocks increases with the problem resolution.

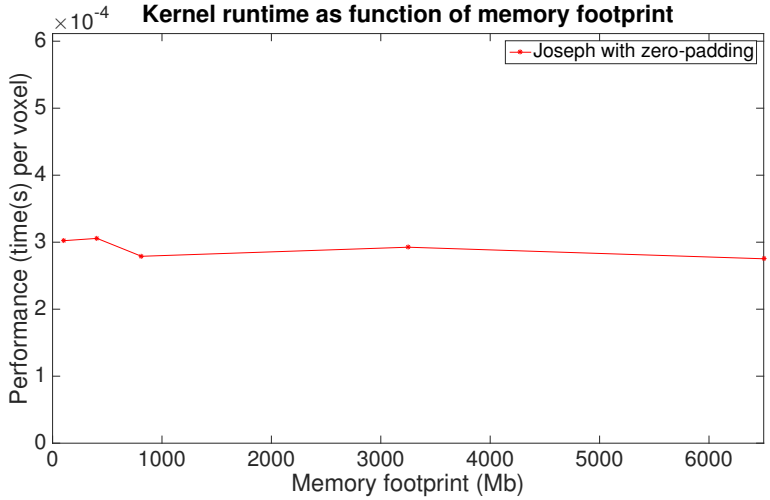


Figure 4.4: Performance in time per voxel for differing memory footprint. The performance per voxel is consistent as the problem size and memory footprint increases. The slight increase in performance at the 2nd and 3rd points are due to the problem size along the z-axis being reduced by half, from 512 to 256 for and 1024 to 512. This causes the number of cache-rows to be reduced by half as well and causes a slight performance increase.

However even if there is an increase in memory footprint with the addition of the explicit zero-padding, this does not have an impact on the performance per voxel, as can be seen in figure 4.4. With increasing problem sizes and memory footprint, the performance per voxel is consistent. This means that using explicit zero-padding will have a comparable performance to using virtual zero-padding, which will be introduced in subsection 4.2.2. As such, the main detriment in using explicit zero-padding is in the large increase in memory footprint, limiting the size on problems that can be run on the available hardware.

4.2.2 Virtual zero-padding

Ideally, we would like for this layer to be implemented in such a way that there will be no reading or writing to this extra boundary layer so that we avoid having to allocate and transfer extra memory for the solution.

This can be accomplished by using a virtual zero-padding instead of the explicit zero-padding previously described.

Code 4.1: Virtual zero-padding for the Joseph kernel

```
const bool inside1 = iy > 0  && iz > 0;
const bool inside2 = iy < ny && iz > 0;
const bool inside3 = iy > 0  && iz < nz;
const bool inside4 = iy < ny && iz < nz;
const float voxelval1 = inside1 ? X[voxel1] : 0.0f;
const float voxelval2 = inside2 ? X[voxel2] : 0.0f;
const float voxelval3 = inside3 ? X[voxel3] : 0.0f;
const float voxelval4 = inside4 ? X[voxel4] : 0.0f;
```

This so-called virtual zero-padding can be accomplished by adding a simple check during the inner while-loop of our implementation, for both forward and backward projections, that checks if a voxel is in fact inside of the true domain before reading or writing to the value of the voxel and simply reads a zero if it is not.

This simple check is shown in code 4.1, where the boolean variables *inside*1 to *inside*4 determine if the current voxel index is inside or outside of the domain and *voxelval*1 to *voxelval*4 update the value of the voxel intersected by the ray and relevant neighboring voxels in the solution volume.

While the virtual zero-padding requires extra conditional checks inside of the main loop body, this doesn't actually impact performance in any noticable way and instead saves a potentially huge addition to the memory in use.

4.2.3 Domain decomposition on multiple devices

Due to the nature of the *Joseph* method and how voxels neighboring the intersecting ray are affected when calculating the weights, implementing the method to work with domain decomposition on multiple devices requires a certain finesse.

As a ray going through the inside the boundary of a neighboring domain can

potentially influence the weighting of voxels in the current domain, the implementation will have to keep track of rays not only intersecting the current domain but also passing through the near vicinity of the boundary of the current domain.

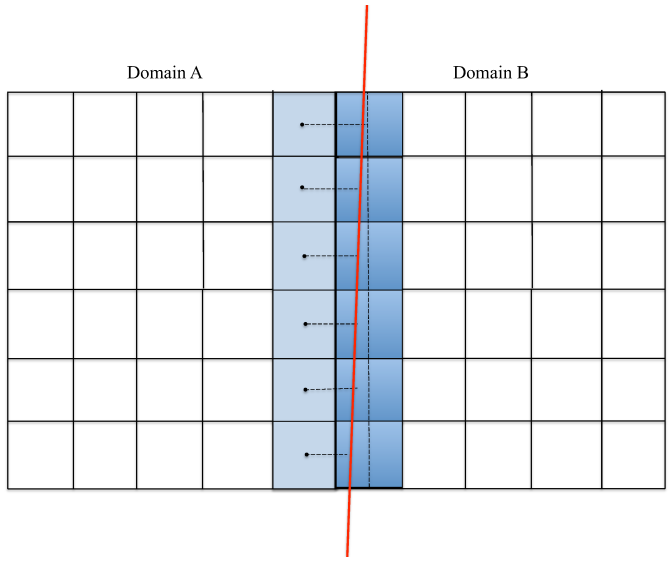


Figure 4.5: Weighting of the *Joseph* method on the boundary between bordering domains. Domain A will need to be aware of rays operating at the boundary of the domain as those rays till potentially add weight to the outer layer of Domain A, while when operating in Domain B, the implemented weighting scheme will want to add weights to the outer boundary of the domain which is in fact inside of Domain A.

When a ray intersects a voxel on the boundary between two domains it will influence the weighting of voxels in both domains. If a ray intersects a boundary voxel in the current domain, the implementation will want to write a weight to a voxel that is outside the current domain. Therefore we must check and specify that said voxel is outside of the current domain to avoid this.

If the ray instead intersects the voxel on the circumference of the current domain, that is a voxel in a neighboring domain, it will still add a weight to voxels in the current domain. Therefore, even if a ray doesn't explicitly intersects the current domain it is still necessary to track and include the ray, as long as it intersects a neighboring voxel at a point 0.5 units or closer to the current domain. An illustration of this dependency is shown in figure 4.5.

If the intersection point is outside 0.5 units from the boundary, the ray will be determined as being outside of the one voxel circumference of the current domain when the new ray intersection point has been calculated by shifting 0.5 units to the "left" and "up" as shown in figure 4.2 on page 57.

Because of these two circumstances, an extra complexity is added when implementing the *Joseph* method together with domain decomposition. When working in a specific domain we neither want to read from nor write to elements in any other domains as that would require loading memory from another device and also possibly introduce a data race where the value of the read element changes after being read.

The implementation of the zero-padding circumvenes this issue, as the weighting for elements in the neighboring domain will now lie in the zero-padding and be negated.

Chapter 5

Large-scale performance results

The focus of this chapter is to document and analyse the results of the large-scale numerical experiments conducted throughout the project, with the aim of reviewing the performance of the implemented methods when executed on state-of-the-art multicore and manycore architecture.

The aim is to find the optimal High-Performance Computing parameters to get the most out of the available hardware resources and achieve the best possible scaling of the implementation.

5.1 Blocking and domain decomposition

To get the best performance for our implementation we will look into two main areas of tweaking for the problem being run on the implementation.

First is to be aware of the limitations of the hardware the implementation is executed on, such as the size of the device memory cache as introduced in section 1.5. As mentioned in subsection 1.5.4, blocking of the problem that is being executed is used to take the most advantage of the limited high-speed memory that is available on the device by limiting the amount of data that is being transfered at a time to areas of the memory that is actually needed for the current computations.

Figure 5.1 shows how the performance of the computations are impacted by different choices of cachedomain sizes when blocking the problem.

The choice of 8 voxels is made because due to the implementation being in single-precision, 8 voxels can fit into a cache line, as a cache line can hold 32 bytes and a floating point number in single-precision consists of 4 bytes.

As can be seen from figure 5.1, it is advantageous to have the cachedomains be of size 8×8 along the x- and y-directions as the program is read and executed by row by the processing unit and the rows of the cachedomain then fit into the cache lines.

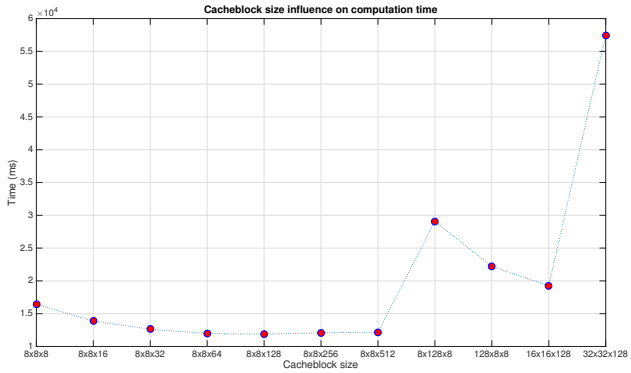


Figure 5.1: Performance with different cachedomain sizes for the *Joseph* kernel with explicit zero-padding. The timings are for the Walnut problem of size 1024^3 with 40 projections and 1 iteration run on a single node with two GPUs. The domain has been decomposed with two domains split along the vertical z-axis and distributed with one subdomain per device. The blocks are oriented in x-, y- and z-directions such that a cachedomain of for example size $8 \times 16 \times 32$ would have 8 voxels along the x-direction, 16 along the y-direction and 32 along the z-direction.

From the numerical experiments of which the results are shown in figure 5.1, it was determined that the most ideal size for the cachedomains was $8 \times 8 \times 128$. If the size along the z-direction is too small, too many cachedomains will continually have to be loaded to the cache without taking full advantage of the size of the cache. Likewise, if the size of the cachedomain along the z-direction is too large, the cachedomain will no longer fit into the cache as neatly, which will in turn reduce the performance.

As can be seen from figure 5.1 on the facing page the choice of cachedomain size can have a large impact on the performance, and just increasing the size of the blocks slightly to $16 \times 16 \times 128$ can have a reduction in computation time of close to 30%.

Another consideration that can increase or potentially harm the performance of the implementation is how the solution domain is decomposed prior to computation.

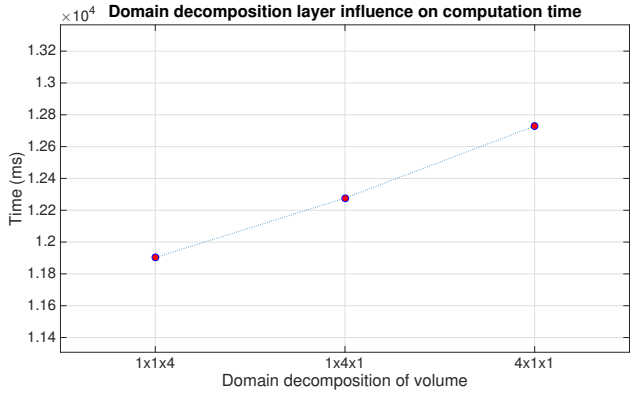


Figure 5.2: Performance of the *Joseph* kernel illustrating the impact of how the domain is decomposed along rows or columns on the computation times. Like for the results in figure 5.1 on the facing page, the timings are for the Walnut problem of size 1024^3 with 40 projections and 1 iterations with cachedomains of size $8 \times 8 \times 128$ run on a single node with two GPUs. The domains have been distributed with two domains per GPU, and the domain decomposition $1 \times 1 \times 4$ means that the domain has been decomposed with 1 along the x-axis, 1 along the y-axis and 4 along the z-axis for a total of 4 processor domains.

Decomposing the domain horizontally rather than vertically when possible, that is to say along the $x-$ and $y-$ directions before the $z-$ direction, is the most advantageous for the performance of the implementation. This can also be seen from figure 5.2 where the Walnut problem has been executed with a resolution of 1024^3 with 40 projections and cachedomains of size $8 \times 8 \times 128$ with three different domain decomposition distributions with four subdomains for each distribution. By splitting the domain along the z-axis as $1 \times 1 \times 4$ the best performance is achieved while the worst is by splitting the domain along the x-axis as $4 \times 1 \times 1$.

From this, it can be deemed that the domain should first be split along the z-axis after which, when the number of domains become larger, the next split should be along the y-axis, and only when the number of domains become increasingly large should the solution domain be split along the x-axis.

The reason for not continually splitting the domain along the z-axis as the number of subdomains increase is both because the subdomains has to be of greater size than the cachedomains, and because a too large split of the domain along a single axis will decrease the likelihood of the density of the data distributed among all the subdomains being homogeneous. This will cause an uneven load on the devices in use and a waste of computational resources. If for example a device has been allocated a subdomain lying on the very edge of the solution domain, and this subdomain has a very low density of data when compared to the interior of the domain, this device will finish computations on the subdomain faster than the computations will be done for the remaining subdomains. This will cause the finished device to wait for the remaining subdomains to be finished and waste the computational resources which could be used more efficiently otherwise.

5.2 GPU cluster performance

The following performance tests have all been conducted on the DeiC Abacus GPU Cluster covered in section 1.7, where each computation node consists of two GPUs.

All of the test problems on all node configurations have been conducted with cachedomains of size $8 \times 8 \times 128$ with two iterations.

As the GPU cluster is based on a queuing system with multiple users, the performance can deviate from what would otherwise be obtainable when the number of computation nodes increases as it can be difficult to ensure that all nodes are available from the same or few closely connected switches.

5.2.1 Scaling of the implementation

In High-Performance Computing the scaling of an implementation is defined by how well the performance of the implementation increases when using more hardware resources for executing the same problem.

How well an implementation scales can be measured by the speedup calculated from Amdahl's law, stating that the theoretical speedup of the implementation is given by

$$S(s) = \frac{T}{T(s)} = \frac{1}{1 - p + \frac{p}{s}}, \tag{5.1}$$

where T is the initial execution time of the whole task before any improvement to the resources of the system and the efficiency p is the percentage of the execution time of the task that has benefit of the increase in resources to the system. s is the factor that the execution time $T(s)$ is sped up after benefitting from the improvement in system resources and $S(s)$ is the actual speedup of the execution time $T(s)$ compared to the execution time T for the unimproved resources.

If an implementation scales well it will have a big impact on the execution time of the implementation if the number of hardware resources available increase or decrease.

The scalability of an implementation can be further divided into *strong scaling* and *weak scaling*. Strong scaling is when the execution time of the implementation varies with the number of computation units for a fixed total problem size. A weak scaling implementation will have an execution time that varies with the number of computation units for a fixed problem size per computation unit.

Both strong and weak scaling is relevant for implementations for problems in computed tomography as the problems are usually on a large scale. As such, for an implementation to have a weak scaling is still relevant as adding more computation units not only is desired to reduce the overall execution time but also to increase the resolution of the reconstruction.

The total computation times for the walnut data for problem sizes from 512^3 to 4096^3 with the *Joseph* method used for forward projections and the *Back-projection* method for backprojections are shown in figure 5.3 on the following page in a double-logarithmic plot. The dotted lines are regression graphs of Amdahl's law and represent the theoretical performance of the implementation for the different problem sizes.

The actual performance and the theoretical performance is the same until the measured performance starts deviating at 16 nodes for the smaller problem sizes and a bit later when nearing 32 nodes for the larger problem sizes.

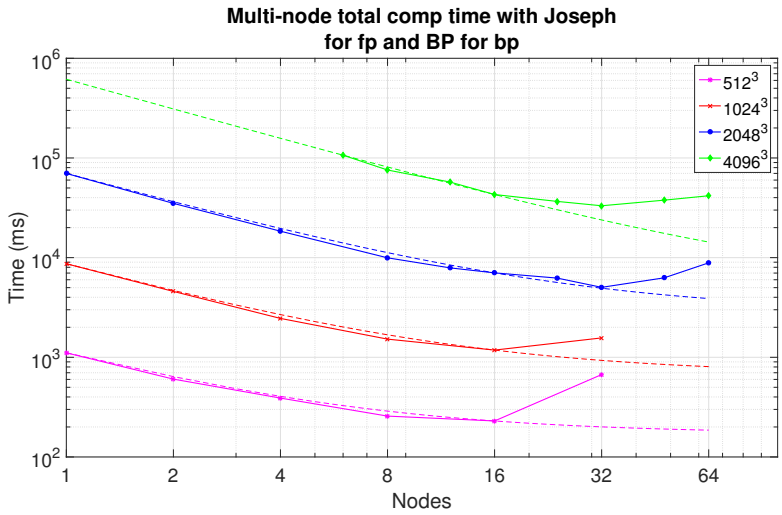


Figure 5.3: Total computation time with the *Joseph* method for forward projections and *Backprojection* method for backprojections for the walnut data of different problem sizes. The dotted lines are regression graphs of Amdahl's law for each problem size with efficiency p numbers for the problem sizes $512^3 = 0.845$, $1024^3 = 0.921$, $2048^3 = 0.959$ and $4096^3 = 0.992$, describing the percentage of parallellization of the code for the different problem sizes.

The computation nodes are distributed on switches with 18 nodes per switch. Due to this, once the number of computation nodes exceeds the amount of nodes available on a single switch, the performance will receive a hit as the nodes become distributed on two different switches. For nodes to communicate in between switches is much more time consuming than communication across the same switch.

Figure 5.4 on the next page shows the speedup of the implementation for problem sizes 512^3, 1024^3 and 2048^3. The dotted lines are again regression graphs of Amdahl's law, this time representing the theoretical speedup of the implementation. As can be seen, the implementation follows Amdahl's law almost perfectly until reaching 16 nodes where the smaller problem sizes of 512^3 and 1024^3 again deviates and the speedup start dropping.

The speedup of the 2048^3 problem still follow the theoretical speedup very closely through 32 nodes, as was also seen in figure 5.3 of the total computation time, and is expected to start dropping when going above 32 nodes.

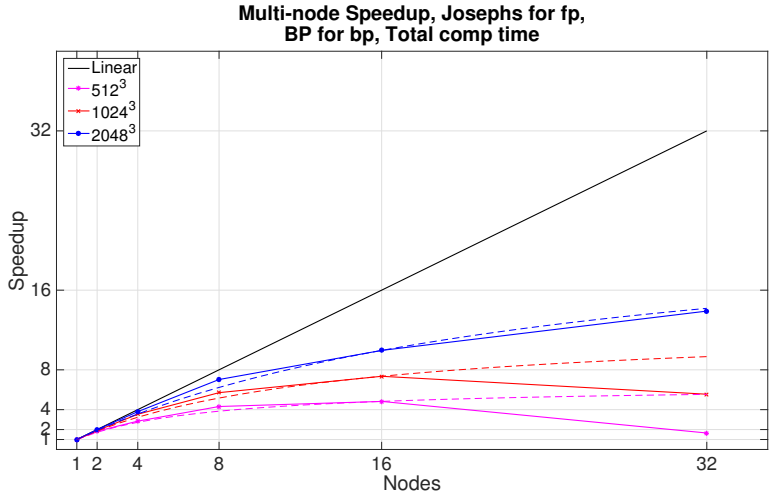


Figure 5.4: Speedup of the total computation time with the *Joseph* method for forward projections and *Backprojection* method for backprojections. The dotted lines are regression graphs of Amdahl's law for each problem size with the same efficiency numbers as stated in figure 5.3 on the facing page.

From one to four nodes the 1024^3 and 2048^3 problems have a linear speedup and the 2048^3 problem even has a speedup close to 8 with 8 computation nodes. From these results it can be seen that the implementation scales very well, especially when looking at the performance below 16 nodes.

The reason for the speedup at 8 nodes being higher than the theoretical speedup is because of how the Amdahl regression graphs are calculated from the 16 nodes points. The reason for choosing this point for the calculation is to account for the larger time consumed by communications when more nodes are added. Due to the way the compute center is set up with 18 nodes per switch, the computation time at 32 nodes is much longer than the theoretical computation time because of the increase from communications across swithces. This is why calculating the regression graphs from the 16 nodes points gives a more accurate approximation than it would with the 32 nodes points, and also why the theoretical speedup is lower than the computed speedup for 8 nodes.

5.2.2 Barriers and communication

When working with implementations for large-scale problems on large manycore systems, and especially on GPU clusters, barriers and communication between devices will have an increasingly large impact on the total runtime of the implementation as the number of devices or computation nodes increase.

Communication represents the time spent doing communication between ranks during runtime, such as sending or receiving data, while barriers function to halt and synchronize ranks. Barriers are necessary to avoid deadlocks that is when multiple ranks or processes try to access the same resource at the same time and the waiting process is still holding another resource that the process it is waiting for needs before it can finish. The result is that neither process can progress and a deadlock is in place.

In our implementation the functions MPI_Isend and MPI_Irecv is used to send the updated forward projection of one block from one rank to all other ranks who will receive the updated forward projection to be used in their own computations.

Barriers are necessary here to keep all ranks in synch such that computations on the next block is not started before the current is finished.

Figure 5.5 on the next page show the sum of all timings of the necessary MPI barriers during computations across all nodes for differing problem sizes. A large value will as such mean a large waiting time across all nodes during computations resulting in a waste of resources.

As expected, the total MPI barrier timings increase as more nodes are added as the communication and synchronization between computation nodes will increase. The MPI barrier timings especially increase dramatically when the problem size is increased, and for the 2048^3 problem we almost see an exponential increase in MPI barrier timings with the increase in number of nodes.

This trend was also seen back in subsection 2.2.2 where the percentage of time out of the total computation time spent doing forward- and backprojections drastically decreased with the increase in number of computation nodes. For problem sizes 512^3 and 1024^3 the timings got close to 20% for forward projections and as low as 15% for backprojections with 16 computation nodes. This means that a majority of the remaining time out of the total computation time will have been spent during communication and waiting at an MPI barrier.

From this it can be seen that finding ways to reduce communication and time

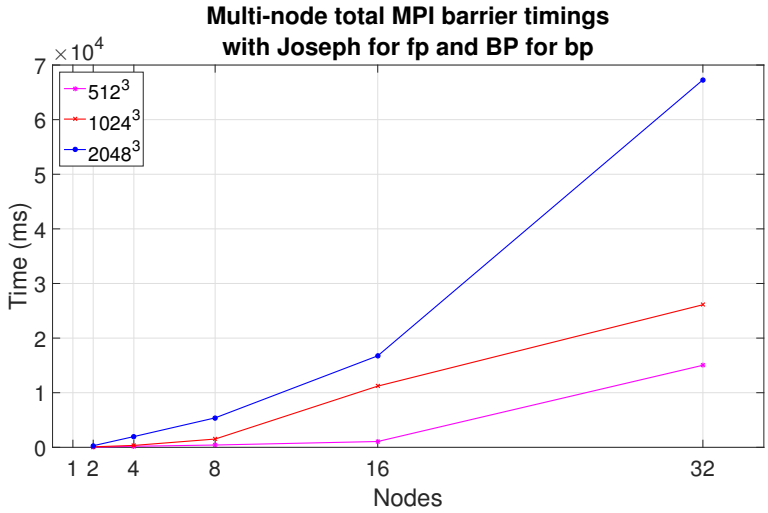


Figure 5.5: Total MPI barrier timings for given number of nodes and problem sizes with 400 projection angles. The values shown are the sum of all MPI barrier timings during computation across all nodes. A higher value will mean that a greater amount of time will have been spent on communications and waiting at barriers as an average across all nodes.

waiting at barriers or to hide these behind computations can potentially have a very big impact on the performance of implementations for High-Performance Computing.

Chapter 6

Conclusion and Future Work

The goal of the thesis was to develop CPU and GPU kernels in C++ implementing the projection method introduced by Joseph for use with large-scale tomographic reconstruction problems in an existing High-Performance Computing framework and compared to other projection methods. Furthermore background regarding the theoretical foundation of the involved algorithms should be presented and analysed.

This has been accomplished and the implementation tested on large-scale experimental data on state-of-the-art multi-node hardware.

The performance and convergence of the method has been compared to the already implemented line length projection method and has been found to have a slightly better scaling and a marginaly longer run-time compared to the line length method.

The Joseph projection method was found to work very well for use with the forward projections but quite slow as a backprojection due to the ray-driven nature of the method. Instead it proved more effective to combine the method with the voxel-driven Backprojection method to have the Joseph method handle the forward projections and Backprojection method the backprojections. This led to a big reduction in time needed for the backprojections and a comparable reconstruction quality due to the similarity of the projection methods.

The main problem with the implementation of the Joseph method was with the larger memory footprint when using explicit zero-padding, but this was circumvened with the addition of a virtual zero-padding to the implementation.

The implemented method is well-suited for High-Performance Computing in tomographic image reconstruction and performs especially well when combined with the *Backprojection* projection method for a faster computation of the back-projections.

The convergence and the image reconstruction results of the method were tested, and it was found that the Joseph implementation delivered significantly better reconstruction results for the used data compared to the *line length* method. This was especially so when facing an underdetermined problem with a lack of available data (projection angles).

The block algebraic iterative reconstruction methods based on block-sequential and block-parallel versions of the row-oriented Kaczmarz algorithm (ART) was investigated and analysed in the literature, and the implementation of the block-sequential method, proven to be superior for multicore computing, was tested and analysed for the best performance with domain decomposition.

6.1 Future Work

Finally we will discuss how the work in this thesis can be extended, and how the performance of the methods might be improved.

A more thorough research and analysis could be done in the field of block methods, where a column-oriented version of the block-sequential method could be implemented and compared. Another area that could be tested is the use of both CPUs and GPUs simultaneously for the computations and what the optimal distribution of the work-load would be.

In the very large-scale reconstructions we have seen that communication between nodes will become the major bottleneck on current HPC systems. It would therefore be of much interest to develop techniques to hide this communication better, e.g. by using several projections per block and sending and receiving data for one projection while computing another.

Further, a research into the optimal domain decomposition configuration for increased performance would be of interest. This could be done by performing a single round of forward projection and backprojection to use as a basis for automating an updated domain decomposition based on the distribution of signal data in the domain to tax the computation units responsible for each domain equally to increase resource efficiency.

Appendix A

Code

A.1 Joseph GPU Kernel, main loop body

Code A.1: Joseph GPU Kernel main loop body

```
while(voxels--)
            {

                // Truncate to integer and make sure that we are
    inside domain.
                int iy = y;
                int iz = z;


                if (ix >= 0 && ix <= nx &&
                    iy >= 0 && iy <= ny &&
                    iz >= 0 && iz <= nz) {


                    // Read current voxel value before update.
                    const int voxel1 = ix*ldx + iy     *ldy + iz
      *ldz;
                    const int voxel2 = ix*ldx + (iy+1) *ldy + iz
      *ldz;
                    const int voxel3 = ix*ldx + iy     *ldy + (iz
    +1) *ldz;
                    const int voxel4 = ix*ldx + (iy+1) *ldy + (iz
    +1) *ldz;
                    const float voxelval1 = X[voxel1];
                    const float voxelval2 = X[voxel2];
                    const float voxelval3 = X[voxel3];
                    const float voxelval4 = X[voxel4];
```

```
                    // Calculate distance from exit point to center of voxel in y,z directions
                    float dsy = fabs(y - iy);
                    float dsz = fabs(z - iz);

                    // Summing up weights for four surrounding voxels after moving point 1/2 and rounding to beginning of current voxel
                    // the shortest distance to center of voxel is given the largest weight
                    if (operation == 'f') {

                        fpsum += (1 - dsy)*(1 - dsz) * voxelval1;

                        fpsum += (     dsy)*(1 - dsz) * voxelval2;

                        fpsum += (1 - dsy)*(     dsz) * voxelval3;

                        fpsum += (     dsy)*(     dsz) * voxelval4;

                    }

                    if (iy > 0 && iz > 0) {
                        fwsum += (ssy * ssy + ssz * ssz + 1.0f);
                    }

                    // For back projections
                    if (operation == 'b') {
                        float xv1 = (1 - dsy)*(1 - dsz) * fp[pixel];
                        float xv2 = (     dsy)*(1 - dsz) * fp[pixel];
                        float xv3 = (1 - dsy)*(     dsz) * fp[pixel];
                        float xv4 = (     dsy)*(     dsz) * fp[pixel];
                        if (iy == 0) { xv1 = 0; xv3 = 0; };
                        if (iy == ny) { xv2 = 0; xv4 = 0; };
                        if (iz == 0) { xv1 = 0; xv2 = 0; };
                        if (iz == nz) { xv3 = 0; xv4 = 0; };


                        xv1 = voxelval1 + xv1 > 0.0f ? xv1 : -voxelval1;
                        xv2 = voxelval2 + xv2 > 0.0f ? xv2 : -voxelval2;
                        xv3 = voxelval3 + xv3 > 0.0f ? xv3 : -voxelval3;
                        xv4 = voxelval4 + xv4 > 0.0f ? xv4 : -voxelval4;
                        atomicAdd(&X[voxel1], xv1);
                        atomicAdd(&X[voxel2], xv2);
                        atomicAdd(&X[voxel3], xv3);
                        atomicAdd(&X[voxel4], xv4);
```

```
66                      }
                    }
68
                    // Step one voxel in dominant direction
70                  x += ssx;
                    y += ssy;
72                  z += ssz;
                    ix += stepx;
74
                } // end while loop over hit voxels
```

A.2 Full Joseph GPU Kernel

Code A.2: Full Joseph GPU Kernel

```
#include "kernel_josephs_gpu.h"
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <cuda_runtime.h>
#include <helper_functions.h>    // includes cuda.h and cuda_runtime_api.h
#include <helper_cuda.h>         // helper functions for CUDA error check
#include "parameters.h"
#include "boundingbox.h"
#include "gpu/helper_gpu.h"


#define _LOOP_VER1
//#define _LOOP_VER2


#define DEBUG 0 //operation=='f' && pixel==4*2+3 //pixel==39 || pixel==40
#define DEBUGCALL 0

//
////////////////////////////////////////////////////////////////////////////////

//KERNEL_JOSEPHS   Raytracing using Joseph's algorithm on a level 3 domain
//
//  GPU version of kernel_josephs_body
//
//  kernel_josephs<operation, beam>(...)
//
//    Traces rays of a projection through the current solution domain doing
//    a line integration and a given operation on every voxel that is hit.
//
//  The operation can be:
//       'f' forward  :  Forward projection computing ax = Ax.
//       'b' backward :  Backward projection computing x := x+A^T*(b-ax)./aa.
//
// Input:
//   N3          Number of voxels along each dimension of third sub-domain.
//   P           Number of pixels along each dimension of a projection.
//   x0, y0, z0  Top-left corner of the current domain cube.
//   xk          Nx*Ny*Nz times 1 array that holds the current solution at level 1.
```

```
39 //    ax          Px*Py times 1 array that holds the current
       projetion at level 1.
   //    rx, ry, rz  Unit vector specifying the direction of the rays.
41 //    ox, oy, oz  Vector specifying the origin of the rays.
   //    ux-vz       Rotation matrix (first two columns).
43 //    dsource     Distance from origin to source.
   //    ddetect     Distance from origin to detector.
45 //    dpixel      Distance between two adjacent detector pixels.
   //    droplen     Float specifying the drop lenght for which all ray
       /cube
47 //                intersections shorter in length are "dropped".
   //    ldx,ldy,    Indexing of x-, y-, and z-dimensions, where ldx
       always
49 //    ldz         represents the leading dimension. ldx=1, ldy=nx,
       ldz=nx*ny.
   //
51 // Output:
   //    xk          Possibly updated solution array for current level
       3 domain.
53 //    ax          Possibly updated projection array for current
       level 3 domain.
   //
55 //
       ////////////////////////////////////////////////////////////////////////


57 template <char operation, char beam, char dir>
   __global__ void kernel_josephs_gpu_body(float x0, float y0, float
       z0,
59                                         float * __restrict__ X,
                                           float * __restrict__ fp,
61                                         const int nx, int ny, int
       nz,
                                           const float Px, const float
        Py,
63                                         const int mini1, const int
       minj1,
                                           const int dj1, const int
       elms,
65                                         const float dsource,
                                           const float ddetect,
67                                         const float dpixel,
                                           const float tol,
69                                         const float lambda,
                                           float rx, float ry, float
       rz,
71                                         const float ox, const float
        oy, const float oz,
                                           const float ux, const float
        uy, const float uz,
73                                         const float vx, const float
        vy, const float vz,
                                           int ldx, int ldy, int ldz,
75                                         const int size)
   {
```

```
77      // Get thread and block parameters.
        const int tid = threadIdx.x;
79      int blk;
        if        (dir == 'x') {
81          blk = blockIdx.x+blockIdx.y*gridDim.x+blockIdx.z*gridDim.x*
        gridDim.y;
        } else if (dir == 'y') {
83          blk = blockIdx.y+blockIdx.x*gridDim.y+blockIdx.z*gridDim.x*
        gridDim.y;
        } else if (dir == 'z') {
85          blk = blockIdx.z+blockIdx.x*gridDim.z+blockIdx.y*gridDim.z*
        gridDim.x;
        }
87
        // Move (x0,y0,z0) to top-left voxel of this cache blocking
        domain.
89      x0 += blockIdx.x*nx; y0 += blockIdx.y*ny; z0 += blockIdx.z*nz;

91      // Move X pointer accordingly.
        X += (long)blk*size;
93
        // Main direction used for cone beam.
95      float rx0 = rx, ry0 = ry, rz0 = rz;

97      int entcnt_total = 0, stepcnt_total = 0, misscnt = 0;

99      // Move back into zero-padding layer.
        X -= ldy+ldz;
101
        // Convert int to float and extend volume by 1/2 voxel on y+z
        sides.
103     float Nx = nx;
        float Ny = ny+1;
105     float Nz = nz+1;
        y0 -= 0.5f; z0 -= 0.5f;
107
        // Find ray index bounds.
109     #include "boundingbox.inc"

111 #ifdef _LOOP_VER1
        const int di = maxi-mini, dj = maxj-minj;
113     for (int ray = tid; di > 0 && dj > 0 && ray < di*dj; ray+=
        blockDim.x){
            {
115             // Pixel coordinate in (x,y)-plane
                int pi = ray/dj;
117             int pj = ray - pi*dj;
                pi += mini;
119             pj += minj;
    #else
121      for (int pi = mini+threadIdx.y; pi < maxi; pi += blockDim.y) {
            for (int pj = minj+threadIdx.x; pj < maxj; pj += blockDim.x
        ) {
123 #endif
```

```
125         // Current pixel number is determined from (pj,pi) (j
    in x-dir and i in y-dir)
            const int pixel = (pi-mini1)*dj1+(pj-minj1);
127
            float x, y, z, absrx;
129         if (beam == 'p') {

131             // Parallel beam.

133             // Ray origin in level 3 domain coordinates (move
    detector pixel positions to source).
                x = ux*pj+vx*pi+ox-(dsource+ddetect)*rx-x0;
135             y = uy*pj+vy*pi+oy-(dsource+ddetect)*ry-y0;
                z = uz*pj+vz*pi+oz-(dsource+ddetect)*rz-z0;
137
                absrx = rx >= 0 ? rx : -rx;
139         }
      else if (beam == 'c') {
141
      // Cone beam
143
            // Ray direction (from source position to detector
    pixel position).
145         rx = ux*pj+vx*pi+ox+dsource*rx0;
            ry = uy*pj+vy*pi+oy+dsource*ry0;
147         rz = uz*pj+vz*pi+oz+dsource*rz0;

149         // Ray origin in level 3 domain coordinates.
            x = -dsource*rx0-x0;
151         y = -dsource*ry0-y0;
            z = -dsource*rz0-z0;
153

155         // Making sure that rx is positive as the leading
    dimension
            absrx = rx >= 0 ? rx : -rx;
157
      }
159

161         // Compute nearest and farthest intersection of ray and
     domain
            // and whether ray hits the domain
163
            //Find inverse slopes (direction ray travels in x,y and
     z)
165         float isx = rx > 0 ? absrx/rx : -absrx/rx; isx = rx ==
    0 ? 0 : isx;
            float isy = ry > 0 ? absrx/ry : -absrx/ry; isy = ry ==
    0 ? 0 : isy;
167         float isz = rz > 0 ? absrx/rz : -absrx/rz; isz = rz ==
    0 ? 0 : isz;
            const float ssx = rx/absrx;
169         const float ssy = ry/absrx;
            const float ssz = rz/absrx;
```

```
171
            // Nearest intersection:
173         const float tx1 = rx >= 0 ? -x : -(-x+Nx);
            const float ty1 = ry >= 0 ? -y : -(-y+Ny);
175         const float tz1 = rz >= 0 ? -z : -(-z+Nz);
            float tmin = MAX(ty1*isy, tz1*isz);
177         tmin = MAX(tmin, tx1*isx);

179         // Farthest intersection:
            const float tx2 = rx >= 0 ? (-x+Nx) : x;
181         const float ty2 = ry >= 0 ? (-y+Ny) : y;
            const float tz2 = rz >= 0 ? (-z+Nz) : z;
183         float tmax = MIN(ty2*isy, tz2*isz);
            tmax = MIN(tmax, tx2*isx);
185
            // Check whether ray hits domain
187         const float tEnd = tmax - tmin - tol; // Length of
    domain that ray traverses in dominant direction
            if (tEnd < 0.0f){  // If tEnd < 0, ray is outside of
    domain and loop body is skipped
189             misscnt++;
                continue;
191         }

193         // Initialize sum.
            float fpsum = 0.0f;
195         float fwsum = 0.0f;

197         // Note: Moving to first intersection of domain, not of
     zy plane.
            x += tmin * ssx;
199         y += tmin * ssy;
            z += tmin * ssz;
201
            // Determine number of voxel steps in the dominant
    direction from "bottom"
203         int voxels = tEnd+1;

205         // Truncate x to integer and make sure that we are
    inside domain.
            int ix = x;
207         if (ix == nx) ix = ix-1;
            const int stepx = rx >= 0 ? 1 : -1;
209
            // DEBUG counters initialization
211         int stepcnt = 0, skipcnt = 0, entcnt = 0;

213         // Loop until ray exits domain (ray must be within 0.5f
     of true domain)
            while(voxels--)
215         {

217             // Truncate to integer and make sure that we are
    inside domain.
                int iy = y;
```

```
219                 int iz = z;
                    //if (iy == ny) iy = iy-1;
221                 //if (iz == nz) iz = iz-1;

223                 if (ix >= 0 && ix <= nx &&
                        iy >= 0 && iy <= ny &&
225                     iz >= 0 && iz <= nz) {

227
                        // Read current voxel value before update.
229                     const int voxel1 = ix*ldx + iy     *ldy + iz
      *ldz;
                        const int voxel2 = ix*ldx + (iy+1) *ldy + iz
      *ldz;
231                     const int voxel3 = ix*ldx + iy     *ldy + (iz
    +1) *ldz;
                        const int voxel4 = ix*ldx + (iy+1) *ldy + (iz
    +1) *ldz;
233                     const float voxelval1 = X[voxel1];
                        const float voxelval2 = X[voxel2];
235                     const float voxelval3 = X[voxel3];
                        const float voxelval4 = X[voxel4];
237
                        // Calculate distance from exit point to center
     of voxel in y,z directions
239                     float dsy = fabs(y - iy);
                        float dsz = fabs(z - iz);
241
                        // Summing up weights for four surrounding
    voxels after moving point 1/2 and rounding to beginning of
    current voxel
243                     // the shortest distance to center of voxel is
    given the largest weight
                        if (operation == 'f') {
245
                            fpsum += (1 - dsy)*(1 - dsz) * voxelval1;
247
                            fpsum += (     dsy)*(1 - dsz) * voxelval2;
249
                            fpsum += (1 - dsy)*(     dsz) * voxelval3;
251
                            fpsum += (     dsy)*(     dsz) * voxelval4;
253

255                         if (iy > 0 && iz > 0) {
                                fwsum += (ssy * ssy + ssz * ssz + 1.0f)
    ;
257                         }

259                     }

261                     // For back projections
                        if (operation == 'b') {
263                         float xv1 = (1 - dsy)*(1 - dsz) * fp[pixel
    ];
```

```
                        float xv2 = (     dsy)*(1 - dsz) * fp[pixel
    ];
265                     float xv3 = (1 - dsy)*(     dsz) * fp[pixel
    ];
                        float xv4 = (     dsy)*(     dsz) * fp[pixel
    ];
267                     if (iy == 0) { xv1 = 0; xv3 = 0; };
                        if (iy == ny) { xv2 = 0; xv4 = 0; };
269                     if (iz == 0) { xv1 = 0; xv2 = 0; };
                        if (iz == nz) { xv3 = 0; xv4 = 0; };
271
                        // ### modified >> Slightly faster version.
273                     xv1 = voxelval1 + xv1 > 0.0f ? xv1 : -
    voxelval1;
                        xv2 = voxelval2 + xv2 > 0.0f ? xv2 : -
    voxelval2;
275                     xv3 = voxelval3 + xv3 > 0.0f ? xv3 : -
    voxelval3;
                        xv4 = voxelval4 + xv4 > 0.0f ? xv4 : -
    voxelval4;
277                     atomicAdd(&X[voxel1], xv1);
                        atomicAdd(&X[voxel2], xv2);
279                     atomicAdd(&X[voxel3], xv3);
                        atomicAdd(&X[voxel4], xv4);
281
                    }
283             }

285             // Step one voxel in dominant direction
                x += ssx;
287             y += ssy;
                z += ssz;
289             ix += stepx;

291         } // end while loop over hit voxels

293         if(operation == 'f'){
                atomicAdd(&fp[pixel], fpsum);
295             atomicAdd(&fp[pixel+elms], fwsum);
            }
297
        } // end ray for loop / pj loop
299 }// end pi loop
} // end kernel_josephs_gpu
301

303 // Template function wrappers.
// - For some reason, using template functions decrease performance
    significantly (maybe branch predication bad?).
305 // - Therefore we use C functions wrappers in order to reuse body
    code but still have good performance.
// - We alse use the wrappers to set parameters to the current
    dominant direction.
307
```

```
309 #ifdef _LOOP_VER1
    #define BLOCK dim3(128)
311 #else
    #define BLOCK dim3(24, 4)
313 #endif

315
    #define KERNEL_WRAPPER(name, op, beam, opchar, beamchar, dirchar, x
        , y, z) \
317         void kernel_##name##_##op##_##beam##_##x(const param_dom *
        dom, \
                                                    const param_bbox *
        bbox, \
319                                                 const param_ang *
        ang, \
                                                    const parameters *
        param) \
321         {
             \
                const int x0 = dom->xyz[0];
             \
323                const int y0 = dom->xyz[1];
             \
                const int z0 = dom->xyz[2];
             \
325                const int Nx = dom->N[0];
             \
                const int Ny = dom->N[1];
             \
327                const int Nz = dom->N[2];
             \
                const int dx = MIN(Nx, dom->cacheblock[0]);
             \
329                const int dy = MIN(Ny, dom->cacheblock[1]);
             \
                const int dz = MIN(Nz, dom->cacheblock[2]);
             \
331                const int nx = dx+2*param->zeropadding;
             \
                const int ny = dy+2*param->zeropadding;
             \
333                const int nz = dz+2*param->zeropadding;
             \
                const int ldx = 1;
             \
335                const int ldy = nx;
             \
                const int ldz = nx*ny;
             \
337                float * __restrict__ X = dom->X_d;
             \
               X += param->zeropadding*(ldx+ldy+ldz);
             \
339                const dim3 grid = dim3((N##x/d##x),(N##y/d##y),(N##z/d
        ##z)); \
```

```
            kernel_##name##_body<opchar, beamchar, dirchar>
        \
341             <<<grid, BLOCK, 0, 0>>>
        \
                (x##0, y##0, z##0,
        \
343              X, dom->fp_d,
        \
                 d##x, d##y, d##z,
        \
345              param->P[0],
        \
                 param->P[1],
        \
347              bbox->mini, bbox->minj, bbox->dj, bbox->elms,
        \
                 param->dsource,
        \
349              param->ddetect,
        \
                 param->dpixel,
        \
351              param->droplen,
        \
                 param->lambda,
        \
353              ang->r##x, ang->r##y, ang->r##z,
        \
                 ang->o##x, ang->o##y, ang->o##z,
        \
355              ang->u##x, ang->u##y, ang->u##z,
        \
                 ang->v##x, ang->v##y, ang->v##z,
        \
357              ld##x, ld##y, ld##z,
        \
                 nx*ny*nz);
        \
359         if (DEBUGCALL) printf("\nGPU KERNEL CALL: x0=%i y0=%i
    z0=%i Nx=%i Ny=%i Nz=%i nx=%i ny=%i nz=%i ldx=%i ldy=%i ldz=%i\
    n", x##0, y##0, z##0, N##x, N##y, N##z, n##x, n##y, n##z, ld##x
    , ld##y, ld##z); \
            if (DEBUGCALL) printf("GPU KERNEL CALL: X_d=%p fp_d=%p
    mini=%i maxi=%i minj=%i maxj=%i\n", dom->X_d, dom->fp_d, bbox->
    mini, bbox->maxi, bbox->minj, bbox->maxj); \
361     }

363

#define KERNEL_WRAPPERS(name)                   \
365   KERNEL_WRAPPER(name, f, par, 'f', 'p', 'x', x, y, z);    \
      KERNEL_WRAPPER(name, f, par, 'f', 'p', 'y', y, x, z);    \
367   KERNEL_WRAPPER(name, f, par, 'f', 'p', 'z', z, x, y);    \
      KERNEL_WRAPPER(name, b, par, 'b', 'p', 'x', x, y, z);    \
369   KERNEL_WRAPPER(name, b, par, 'b', 'p', 'y', y, x, z);    \
      KERNEL_WRAPPER(name, b, par, 'b', 'p', 'z', z, x, y);    \
```

```
371   KERNEL_WRAPPER(name, f, cone, 'f', 'c', 'x', x, y, z);     \
      KERNEL_WRAPPER(name, f, cone, 'f', 'c', 'y', y, x, z);     \
373   KERNEL_WRAPPER(name, f, cone, 'f', 'c', 'z', z, x, y);     \
      KERNEL_WRAPPER(name, b, cone, 'b', 'c', 'x', x, y, z);     \
375   KERNEL_WRAPPER(name, b, cone, 'b', 'c', 'y', y, x, z);     \
      KERNEL_WRAPPER(name, b, cone, 'b', 'c', 'z', z, x, y);
377
  // Determine dominant direction and call appropriate kernel.
379 #define CALL_KERNEL_DOMINANT_DIRECTION(name, operation)
                          \
    const int beam = param->beam;
          \
381   const float absx = ABS(ang->rx);
          \
      const float absy = ABS(ang->ry);
          \
383   const float absz = ABS(ang->rz);
          \
      if (absx >= absy && absx >= absz) {
          \
385     if (beam == 'p') {
          \
        kernel_##name##_##operation##_par_x(dom, bbox, ang, param); \
387     } else if (beam == 'c') {
          \
        kernel_##name##_##operation##_cone_x(dom, bbox, ang, param);\
389     }
          \
    } else if (absy >= absx && absy >= absz) {
          \
391     if (beam == 'p') {
          \
        kernel_##name##_##operation##_par_y(dom, bbox, ang, param); \
393     } else if (beam == 'c') {
          \
        kernel_##name##_##operation##_cone_y(dom, bbox, ang, param);\
395     }
          \
    } else {
          \
397     if (beam == 'p') {
          \
        kernel_##name##_##operation##_par_z(dom, bbox, ang, param); \
399     } else if (beam == 'c') {
          \
        kernel_##name##_##operation##_cone_z(dom, bbox, ang, param);\
401     }
          \
    }
403
  KERNEL_WRAPPERS(josephs_gpu);
405
  void gpu::kernel_josephs_f(const param_dom *dom,
407                            const param_bbox *bbox,
                               const param_ang *ang,
```

```
409                             const parameters *param)
{
411     cudaSetDevice(dom->device);
        CALL_KERNEL_DOMINANT_DIRECTION(josephs_gpu, f);
413     if (cudaGetLastError() != cudaSuccess) { printf("
        kernel_josephs_gpu : Kernel launch error in %s: line %d.\n",
        __FILE__, __LINE__); exit(0); }
}
415
void gpu::kernel_josephs_b(const param_dom *dom,
417                             const param_bbox *bbox,
                                const param_ang *ang,
419                             const parameters *param)
{
421     cudaSetDevice(dom->device);
        CALL_KERNEL_DOMINANT_DIRECTION(josephs_gpu, b);
423     if (cudaGetLastError() != cudaSuccess) { printf("
        kernel_josephs_gpu : Kernel launch error in %s: line %d.\n",
        __FILE__, __LINE__); exit(0); }
}
```

Bibliography

[1] Anders H Andersen and Avinash C Kak. Simultaneous algebraic reconstruction technique (sart): a superior implementation of the art algorithm. *Ultrasonic imaging*, 6(1):81–94, 1984.

[2] Joost Batenburg, Willem Jan Palenstijn, and Jan Sijbers. Projection and backprojection in tomography: design choices and considerations. *WADGMM 2010*, page 106, 2010.

[3] Douglas P Boyd, Sholom M Ackelsberg, John L Couch, Kristian R Peschmann, and Roy E Rand. Historical perspectives and recent progress in the development of x-ray computed tomography. In *IEEE nuclear science symposium conference record emdash 1990*, 1990.

[4] Weixing Cai and Ruola Ning. Preliminary study of a phase-contrast cone-beam computed tomography system: the edge-enhancement effect in the tomographic reconstruction of in-line holographic images. *Optical Engineering*, 47(3):037004–037004–12, 2008. doi: 10.1117/1.2897284. URL `http://dx.doi.org/10.1117/1.2897284`.

[5] DTU Computing Center. Mpi communication benchmarks. URL `http://www.hpc.dtu.dk/?page_id=1546`. visited on: 2016-03-07.

[6] Gianfranco Cimmino and Consiglio Nazionale delle Ricerche. *Calcolo approssimato per le soluzioni dei sistemi di equazioni lineari*. Istituto per le applicazioni del calcolo, 1938.

[7] Tommy Elfving. Block-iterative methods for consistent and inconsistent linear equations. *Numerische Mathematik*, 35(1):1–12, 1980.

[8] LA Feldkamp, LC Davis, and JW Kress. Practical cone-beam algorithm. *JOSA A*, 1(6):612–619, 1984.

[9] Dan Gordon and Rachel Gordon. Component-averaged row projections: A robust, block-parallel scheme for sparse linear systems. *SIAM Journal on Scientific Computing*, 27(3):1092–1117, 2005.

[10] Peter M Joseph. An improved algorithm for reprojecting rays through pixel images. *Medical Imaging, IEEE Transactions on*, 1(3):192–196, 1982.

[11] Benjamin Keck. *High Performance Iterative X-Ray CT with Application in 3-D Mammography and Interventional C-arm Imaging Systems*. PhD thesis, The Technical Faculty of the Friedrich-Alexander-University of Erlangen-Nurnberg, 2014.

[12] Louis Landweber. An iteration formula for fredholm integral equations of the first kind. *American journal of mathematics*, 73(3):615–624, 1951.

[13] Shih-Chung B Lo. Strip and line path integrals with a square pixel matrix: A unified theory for computational ct projections. *Medical Imaging, IEEE Transactions on*, 7(4):355–363, 1988.

[14] Van-Giang Nguyen and Soo-Jin Lee. Graphics processing unit-accelerated iterative tomographic reconstruction with strip-integral system model. *Optical Engineering*, 51(9):093203–1–093203–11, 2012. doi: 10.1117/1.OE.51.9.093203. URL `http://dx.doi.org/10.1117/1.OE.51.9.093203`.

[15] Kenneth K. Nielsen. Block algebraic methods for 3d image reconstructions on gpus. Master's thesis, Technical University of Denmark, DTU, Kgs. Lyngby, DK2800 Denmark, 2014.

[16] Christopher Rohkohl, Benjamin Keck, HG Hofmann, and Joachim Hornegger. Technical note: Rabbitct—an open platform for benchmarking 3d cone-beam reconstruction algorithmsa). *Medical Physics*, 36(9):3940–3944, 2009.

[17] DeiC SDU. Sdu deic abacus cluster hardware and network configuration. URL `https://deic.sdu.dk/setup/hardware`; `https://deic.sdu.dk/setup/network`. visited on: 2016-03-07.

[18] Siemens. Medical scanners for x-ray ct and mri. URL `http://www.healthcare.siemens.com/computed-tomography`; `http://www.healthcare.siemens.com/magnetic-resonance-imaging`. visited on: 2016-06-01.

[19] Hans Henrik B Sørensen and Per Christian Hansen. Multicore performance of block algebraic iterative reconstruction methods. *SIAM Journal on Scientific Computing*, 36(5):C524–C546, 2014.

[20] Wikipedia. Cpu architecture, . URL `https://en.wikipedia.org/wiki/Central_processing_unit`; `https://en.wikipedia.org/wiki/Microprocessor`. visited on: 2016-06-8.

[21] Wikipedia. Manycore and gpu architecture, . URL https://en.wikipedia.org/wiki/Graphics_processing_unit; https://en.wikipedia.org/wiki/General-purpose_computing_on_graphics_processing_units. visited on: 2016-06-8.

[22] Wikipedia. Medical tomography images, . URL https://en.wikipedia.org/wiki/Positron_emission_tomography; https://en.wikipedia.org/wiki/Electrical_resistivity_tomography. visited on: 2016-06-01.

[23] Wikipedia. Tomography, . URL https://en.wikipedia.org/wiki/Tomography. visited on: 2016-05-30.