Motion correction in X-ray tomography

submitted by

Ander Biguri

for the degree of Doctor of Philosophy

of the

University of Bath

Department of Electrical and Electronic Engineering

September 2017

Abstract

bla

Contents

1 Introduction **4**

2 X-ray imaging in medicine **5**

3 The image reconstruction problem **6**
3.1 Geometry of CBCT 6
3.2 FDK 6
3.3 Iterative reconstruction algorithms 6
3.3.1 Algebraic Reconstruction Techniques 8
3.3.2 Conjugate Gradient Least Squares 10
3.3.3 Statistical inversion 10
3.3.4 Total variation minimization with POCS 10
3.3.5 Total variation regularization via Rudin-Osher-Fatemi model . . 20
3.4 Discussion 23

4 GPU methods in tomography **24**
4.1 Hardware used in this research 26
4.2 GPGPU architecture 26
4.2.1 Exploiting GPGPUs for CT 29
4.3 The projection operator 30
4.3.1 Ray-voxel intersection method 31
4.3.2 Grid-interpolated methods 35
4.3.3 Comments on optimization 37
4.3.4 Differences between operators 39
4.4 The backprojection operator 44
4.4.1 Voxel-driven backprojection 44
4.4.2 Backprojection weights 46
4.4.3 Comments on optimization 50
4.5 Benchmark 50

4.6 The TIGRE toolbox . . . 53
4.6.1 Geometry in TIGRE . . . 53
4.6.2 Structure . . . 54
4.7 Summary . . . 58

5 Experiments and Applications **62**
5.1 Algorithm Experiments . . . 63
5.1.1 Convergence rates . . . 63
5.1.2 Total variation minimization . . . 67
5.2 Iterative Algorithms in Different CT Applications . . . 71
5.2.1 Medical Head CBCT from The Christie Hospital . . . 71
5.2.2 Cryo Soft X-Ray Tomography at Diamond Light Source . . . 71

6 Quality analysis of motion correction **73**

7 Conclusions and Future work **74**

Chapter 1

Introduction

Chapter 2

X-ray imaging in medicine

Chapter 3

The image reconstruction problem

This chapter tries to explain the mathematics behind CT reconstruction, the FDK algorithm and iterative reconstruction algorithms. After the formal proposition of the mathematical problem of integrating over straight lines,the FDK algorithm is introduced. Then, the alternative proposal of the iterative algebraic methods is shown, followed by a wide variety of different algorithms that can be used to solve the algebraic problem. These include gradient descend techniques, Krylov subspace methods, statistical approaches and compressed sensing techniques. Finally, a discussion of the challenges that arise from the use iterative algorithms are described.

3.1 Geometry of CBCT

3.2 FDK

3.3 Iterative reconstruction algorithms

Nowadays the FDK algorithm is the most widely used algorithm, and only until very recently [CITE] it has been the only algorithm available in any commercial medical or industrial CT device. While using FDK is advantageous in some cases, often the algorithm behaves poorly, specially when errors arise in the data, or the amount of data is limited. This is because FDK is based on an analytical approximation of straight path integrals in continuous spaces. Reality is far from straight path integrals, as due to X-ray physics photons do not behave linearly. Photons from CT machines are polychromatic, and human tissue is behaves non-linearly in respect to X-ray energy.

Additionally, Compton scattering is a common effect, where the photons get reflected in different angles dependent on their energy. Apart from photon physics related errors, electronic noise is always present in detector technology being the only feasible way of avoiding it longer exposition times, harmful to live tissue. Limited data can additionally harm the image reconstruction, as CT has generally less detector data than the amount of voxels are desired to reconstruct. All these effects make CT image reconstruction a challenging problem and have strong effect in the behaviour of FDK. As an alternative to FDK, iterative algebraic reconstruction algorithms try to minimize a functional by updating the image continuously and comparing it to the measured data. These algorithms have been shown to improve reconstruction quality when the data is noisy and/or limited. Additionally, as they are an algebraic tool, they allow careful tuning of the mathematics, enabling them to change their behaviour.

Iterative algorithms in CT generally reffer to those algorithms that, as the name says, iterate, but solve the linearized model

$$Ax = b + \tilde{e} \tag{3.1}$$

on where $x \in \mathbb{R}^{N_{voxels}}$ is a vector representing the lexicographically ordered voxels of the 3D image, $b \in \mathbb{R}^{N_{pixels}}$ a vector of the detector measured pixels. A is the linearized model matrix, a matrix that describes the behaviour of the CT system. Each row of the matrix A is describes the behaviour of the X-rays that affect each single pixel in the detector. However, this matrix is so big that in practize its explicit form is impossible to store, and the matrix product operations Ax (or projection) and $A^T b$ are implemented instead. The next chapter goes into a bit more detail on how to operate with matrix A and its limitations. Errors from measurement are inevitable in any application, and there are linearization errors, as no model is perfect. In equation 3.1, $\tilde{e}$ represents all those errors.

As an exact solution for x can not be found, the problem in equation 3.1 is minimized as

$$\hat{x} = \arg\min_x \|Ax - b\|^2 + R(x), \tag{3.2}$$

on where $R(x)$ is an optional regularization function. This minimization function has been widely studied in mathematics and there are multiple algorithms that can solve it. However not all algorithms that solve the equation can be used in CT reconstruction, due to the nature of the A matrix, as it is very big (approximately $10^8 \times 10^8$ in a standard medical image) and very sparse (approximately 0.0017% of sparsity in a standard medical image). This makes the matrix severely ill-conditioned and impossible to store in memory. Additionally, often the CT problem can be underdetermined, making the

problem ill-posed and further constraining the possible algorithms that can be used. That said, a wide variety of algorithm have been proposed to solve the CT algebraic problem and new ones are still being published. This section discusses a few of the available and most common algorithms that have been studied in this work.

3.3.1 Algebraic Reconstruction Techniques

Arguably the most well known iterative algorithm is the method known as the algebraic reconstruction technique (ART)[8], known as the Kaczmarz method outside the CT imaging field. The ART algorithm, for matrix elements $a_{ij} \in R$ is defined as

$$x^{n+1} = x^n + \frac{b_i - \langle a_i, x^n \rangle}{\|a_i\|^2} a_i^T, \tag{3.3}$$

where a_i is the i-th row of matrix A and $\langle \, , \rangle$ denotes the inner product. The ART method projects the image into the hyperplane described by the equation in row i. Generally the method includes a relaxation parameter λ_n that controls the update size, and that decreases with the iteration number. This generally avoids the cyclical convergence that the method describes when the solution is not unique (the intersection of the hyperplanes is not a single point). By relaxing the update step, the algorithm converges to a single point. Generally the algorithm is also run with some inequality constrains, the most common one being a positivity constrain for x, as negative values are not physically possible.

Studies on the convergence of the ART algorithm show[6] that randomly choosing the order of the rows in each iteration (n) increase the convergence rate, even more if the probabilities of picking rows are different than one (different methods propose different probabilities)[18][11].

However, the ART method has a major disadvantage: the image x^{n+1} needs to be updated i times each iteration. In current CT applications, and specifically in CBCT, the amount of rows in the matrix i.e. the total amount of independent pixel measurements in the detector is a massive number. Following the same definition of standard medical image size from the thesis, a 512×512 detector, with 360 projection means that the amount of rows is in the order of 10^8. In order to update the image less, the Simultaneous Iterative Reconstruction Technique (SIRT)[9] can be used, a method that is very similar to Cimmino's method, that updates the image using simultaneously (instead of sequentially) all data in the measurement b, thus each iteration is a single update. While SIRT generally solves the problem of the high amount of updates in ART, it suffers from a very slow convergence in comparison, and will generally plateau in a solution that is not as good as what ART provides. The SIRT algorithm can be

described in matrix form as

$$x^{n+1} = x^n + \lambda_n V A^T W^{-1} (b - Ax) \tag{3.4}$$

where $V = 1/\text{diag}\left(\sum_j a_{ij}\right)$ and $W = \text{diag}\left(\sum_i a_{ij}\right)$.

However, middle ground has also been proposed. Kak and Andersen proposed[1] the Simultaneous Algebraic Reconstruction Technique (SART) on where the image is updated using simultaneously all data from each X-ray projection, but still updating the image multiple times per iteration. Finally, the update can also be done using block-based methods, or Oriented Subsets (OS) with a variety of methods generally described as OS-SART[2] methods.

Similarly as with ART, the order of the subsets in both OS-SART and SART have influence in the convergence, with a lower impact than in ART. In this work an three methods have been implemented, a completely ordered method, a randomized ordered method with full sampling (i.e. all projections are ensured to be used once and only once per iteration) and an angular distance based one. This last one orders the subsets by selecting the next one as the subset with largest angular distance from the ones already used. The heuristic rationale is that the projections at larger angular distance update the image by a bigger step than projections angularly near. In all further result in this thesis, the default ordering is random, unless otherwise explicitly stated.

Relaxation parameter λ

As previously mentioned, changing the relaxation parameter per iteration can be of advantage, by avoiding cyclical convergence and often by increasing the general convergence rate. On of the commonly used methods for the reduction of lambda is simply reducing it by a reduction factor each iteration as

$$\lambda_{n+1} = \lambda_n * r_\lambda \tag{3.5}$$

where r_λ is some value close to one, such as $r_\lambda = 0.99$ or $r_\lambda = 0.999$. However this method, while useful to avoid cyclical convergence in ART methods, is of less use in simultaneous methods, as it generally slows the convergence rate.

It is worth noticing that this family of algorithms is very closely related to the well known gradient descend methods, as the gradient of equation 3.2 is proportional to $A^T(Ax - b)$, or in other words $V = I$ and $W = I$ in equation 3.4. The gradient descend methods have been widely studied in the past years[19][14], as the Neural Networks community tries to find faster converging methods to train the nets they research. Among other methods proposed, Nesterov proposed an accelerated version of the gra-

dient descend[13], that obtains a rate of convergence of $1/n^2$. The proposed update updates the result image in each iteration by pushing it in the current update and previous update direction combined. Nesterovs Accelerated Gradient (NAG) defines

$$\lambda_{n+1} = \frac{1 + \sqrt{1 + 4\lambda_n^2}}{2} \tag{3.6}$$

$$\gamma_n = \frac{1 - \lambda_n}{\lambda_{n+1}} \tag{3.7}$$

$$y^{n+1} = x^n - \frac{1}{\beta}\nabla f(x^n) \tag{3.8}$$

$$x^{n+1} = (1 - \gamma_n)y^{n+1} + \gamma y^n \tag{3.9}$$

with $\lambda_0 = 1$ and β being the Lipschitz smoothness of the function f. The line in equation 3.8 can be replaced by the SART/OS-SART/SIRT update on equation 3.4 to obtain an accelerated convergence rate. Some experimental results on the convergence of the algorithms can be found in Chapter 5.

3.3.2 Conjugate Gradient Least Squares

The conjugate gradient for the least squares problems (CGLS)

3.3.3 Statistical inversion

3.3.4 Total variation minimization with POCS

Sometimes solving a regularized problem may result in a better final image than just trying to solve the data constrain with the model. This is especially useful in more ill-conditioned problems, such as when the data is very noisy (thus the model does not fit the data with accuracy) or when few projections are available (the system is more under-determined). In these cases, regularisation can add a user constrain in the image domain that pushes the algorithm towards an specific solution among all the multiple possibilities. While a variety of regularization techniques and norms exist, the most suitable for CT imaging is the total variation (TV) norm.

The total variation norm is defined as the sum of the 2-norms of the directional gradients of the variable,

$$\|x\|_{\mathrm{TV}} = \sum_n \left\| \sum_\alpha \partial_\alpha x_n \right\|_2 . \tag{3.10}$$

Applied to CT imaging, the total variation norm is the sum of the total change occurred in the image. An image with less total variation would be an image that

would have less change, or more flat, same valued regions. Regularizing with the TV norm as a minimization term will yield an image that is piecewise smooth and it happens that most of the objects imaged in CT scanners are piecewise smooth in linear attenuation, even more in medical CT imaging.

However, solving the minimization problem in equation XX is not trivial with the TV regularization. One of the first robust algorithm is the so called Adaptive Steepest Descend, Projection Onto Convex Subsets, or ASD-POCS algorithm[17]. This algorithm not only minimizes the data constrain with TV regularization but also adaptively controls the TV minimization update, in order to adapt its strength according to the data constrain update. Several adaptations and improvements of this algorithm have been proposed in the literature[CITES], all based in the same mathematical basis.

ASD-POCS

The previous algorithms discussed in this chapter where unconstrained minimization methods. While the TV minimization problem can be solved similarly (see section 3.3.5) formalizing the algorithm as a non-linear constrained minimization adds an advantage in the case where there system is under-determined. In an unconstrained problem such as in equation XX, the balance between the data constrain and the regularization constrain can be tuned via a hyperparameter, but in the case of an under-determined system, multiple solutions for the data fidelity term may exist. By reformulating it as it is shown in the rest of this section, the image with the same data fidelity 2-norm but the lowest TV norm can be chosen.

The minimization will yield an image $\vec{x}^*$ that minimizes

$$\vec{x}^* = \arg\min_x \|\vec{x}\|_{\mathrm{TV}} \tag{3.11}$$

subject to

$$\|A \cdot \vec{x} - \vec{b}\| \leq \epsilon, \tag{3.12}$$

$$\vec{x} \geq 0. \tag{3.13}$$

As previously described in this chapter, the data fidelity on equation 3.12 while desired to be zero, it will never reach to zero, due to inconsistencies in the data, model, noise, etcetera. Thus, in this algorithm it is introduced as a inequality constrain, instead of as the minimization problem itself. This introduces the parameter ϵ in the algorithm, the maximum 2-norm allowed for the data inconsistency. The problem in hand is now non-linear, due to the constrains, but convex.

The conditions for a constrained minimization to find the optimal solution can be obtained by satisfying the Karush Kuhn-Tucker conditions (a generalization of the Lagrange multiplies for inequality constrains). First, the Lagrangian for the current problems needs to be defined, as

$$\mathbf{L} = \|\vec{x}\|_{\mathrm{TV}} + \lambda_0 \cdot (\|A \cdot \vec{x} - \vec{b}\|^2 - \epsilon^2) - \vec{\lambda} \cdot \vec{x}, \tag{3.14}$$

where $\vec{\lambda}$ is a vector of the same size as the image, but λ_0 is a single value. Two inequality constrains are imposed to the Lagrange multipliers, namely non-negativity

$$\lambda_i \geq 0, \tag{3.15}$$

and complementarity

$$h_i(\vec{x})\lambda_i = 0, \tag{3.16}$$

where $i = 0, 1, ..., N_{pixels}$, and h_i is an alternative form of the inequality constrains as

$$h_0(\vec{x}) = \|A \cdot \vec{x} - \vec{b}\|^2 - \epsilon^2 \leq 0 \tag{3.17}$$

$$h_i(\vec{x}) = -x_i \leq 0 \qquad\qquad i \in [1, N_{pixels}] \tag{3.18}$$

Thus, only when the inequalities are violated does h_i turns non-zero, and with the complementarity condition, does the corresponding λ_i turns zero. A solution can be found for $\vec{x}$ when the gradient of the Lagrangian is zero, and if the differential operator is defined as

$$\nabla_{\vec{x}} Q(\vec{x}) = \sum_i \partial_{x_i} Q(\vec{x}) \vec{\delta_i} \tag{3.19}$$

where $\vec{\delta_i}$ is the Kronecker delta. The gradient of the Lagrangian can be then written as

$$\begin{aligned} \nabla_{\vec{x}} \mathbf{L} &= \nabla_{\vec{x}} \|\vec{x}\|_{\mathrm{TV}} + \lambda_0 \nabla_{\vec{x}} h_0(\vec{x}) + \sum_{i=1}^{N_{pixels}} \lambda_i \nabla_{\vec{x}} h_i(\vec{x}) = 0 \\ &= \nabla_{\vec{x}} \|\vec{x}\|_{\mathrm{TV}} + 2\lambda_0 A^T \cdot (A \cdot \vec{x} - \vec{b}) - \vec{\lambda} = 0 \end{aligned} \tag{3.20}$$

Further simplification can be applied to equation 3.20. As the non-negativity constrains are only active in zero valued voxels, the Lagrange multipliers are zero for

strictly positive voxels. Thus, by adding an indicator function

$$\vec{x}_{indic} = \begin{cases} 1 & \vec{x} \neq 0 \\ 0 & \vec{x} = 0 \end{cases} \tag{3.21}$$

the Lagrangian gradient can be shortened to

$$\nabla_{\vec{x}} \mathbf{L} = \text{diag}(\vec{x}_{indic}) \left(\nabla_{\vec{x}} \|\vec{x}\|_{\text{TV}} + \lambda_0 \nabla_{\vec{x}} h_0(\vec{x}) \right) = 0. \tag{3.22}$$

Separating this new equation into two vectors,

$$\begin{aligned} \vec{d}_{\text{TV}} &= \text{diag}(\vec{x}_{indic}) \left(\nabla_{\vec{x}} \|\vec{x}\|_{\text{TV}} \right) \\ \vec{d}_{\text{data}} &= \text{diag}(\vec{x}_{indic}) \left(\nabla_{\vec{x}} h_0(\vec{x}) \right) \end{aligned} \tag{3.23}$$

brings to the Karush Kuhn-Tucker conditions: $\vec{x}$ will be an optimal condition if $\vec{d}_{\text{TV}}$ and $\vec{d}_{\text{data}}$ are pointing in exactly the opposite direction. In practice the algorithm will only check if the vectors are pulling in opposite direction (by computing the dot product) and that the inequality constrains are satisfied. By checking the direction of the vectors the algorithm ensures that even if the data constrain is satisfied, only the optimal solution regarding both TV norm and data fidelity is chosen.

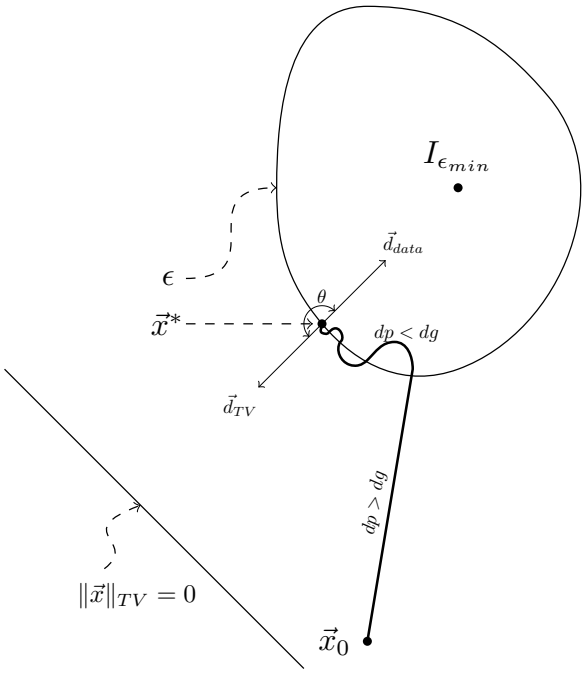


Figure 3-1: Conceptual diagram of the ASD-POCS algorithm path to the solution.

Figure 3-1 shows a conceptual diagram of the ASD-POCS algorithm. There is an area around the image with minimum data constrain, $I_{\epsilon_{min}}$. The solution $\vec{x}^*$ generally lies on the boundary of the area with the user specified ϵ. From an initial image $\vec{x}_0$, the algorithm walks towards the area of acceptable ϵ more strongly than towards the area of minimum TV, as the step sizes of the vectors $\vec{d}_{\text{TV}}$ and $\vec{d}_{\text{data}}$, dp and dg respectively, are adaptively controlled to be $dp > dg$. Once the image is within acceptable 2 norm, then the step size is changed in order to have stronger $\vec{d}_{\text{TV}}$ ($dp < dg$). The optimal solution can be found when both vectors point in opposing direction, or in other words, when the angle between them is 180 degrees, or the cosine of it is -1,

$$\cos\theta = \frac{\vec{d}_{\text{TV}} \cdot \vec{d}_{\text{data}}}{\|\vec{d}_{\text{TV}}\|\|\vec{d}_{\text{data}}\|}. \tag{3.24}$$

The pseudocode for the algorithm can be seen in algorithm 1. The algorithm is essentially solving the two vector in equation 3.23, the data vector in lines [5-8] and the TV vector in lines [18-22]. Line [9] enforces the positivity constrain. In the algorithm, dtv is initialized according to α, an user specified TV hyperparameter for TV, together with dp, the step size performed by the data constrain. After the TV minimization is performed, the step size of the TV vector is rechecked. If the TV minimization step is too big (bigger than the data step size), and the desired ϵ is still not achieved, the step size is reduced further. This method of adaptively setting the step size of the TV iteration relating to the data step size is what ensures the optimal condition is achieved. Finally, the stopping criteria relies in either achieving the desired ϵ with with a desired $\cos\theta$, or stopping due to reaching a maximum amount of iterations (β decreases with iteration number). In the original proposition of the ASD-POCS algorithm (and shown here), the data constrain is solved using SART, however any other algorithm solving the same minimization problem can be used here (e.g. CGLS or OS-SART).

The algorithm has 7 parameters that need to be set up: β and β_{red} are the initial value and reduction ratio of the SART hyperparameter, similarly α and α_{red} serve as hyperparameter and reduction ratio for the TV minimization. r_{max} controls the maximum allowed ratio of change between the data minimization and the TV minimization, in order to adapt the step sizes. The number of iterations the TV minimization performs per iteration of the data minimization is defined as n_{TV}. Finally, the allowed data error is ϵ, as described before. The initial values of the variables in the algorithm are a key factor on its convergence. Empirical tests show that wrong parametrization of the algorithm can lead to severely noisy reconstructions. An study of the sensitivity of these parameters to changes has been performed by Lohvithee *et al*[12]. The study shows that some parameters can be safely set up to a static value regardless of the

data, such as the data hyperparameters, but that ϵ, n_{TV} and α are critical parameters to tune in order to get an usable reconstruction, and they are heavily data dependant. While some algorithms have successfully replaced the initial set of α by some data based heuristics [CITE PCSD][1] to the best of the authors knowledge there is no mathematical proposal for setting these parameters. The biggest drawback of this method is that several reconstructions may be needed to find the best parameters for an specific application.

Note that this minimization approach, while used for TV minimization in the original article, can be used for a variety of different minimization functions. For example, the TV minimization step could be replaced by a prior image minimization[CITE PICS], or any other convex minimization function. Similarly, the data minimization step can be replaced by any other minimization algorithm, as long as it minimizes the problem in equation XX

[1]These algorithms, namely PCSD and Aw-PCSD, are also available in TIGRE, by Manasavee Lohvithee.

Algorithm 1 ASD-POCS

Set: $\beta, \beta_{red}, n_{TViter}, \alpha, \alpha_{red}, r_{max}$
$\vec{x} = 0$;
while Stopping criteria not met **do**
    $\vec{x}_{prev} = \vec{x}$
    for n_{angles} **do**
        $\vec{x} = \vec{x} + \beta V A^T W^{-1}(\vec{b} - A\vec{x})$ ▷ SART update
    end for
    $\beta = \beta * \beta_{red}$
    $\vec{x} = \max(0, \vec{x})$ ▷ Enforce positivity
    $\vec{x}_{out} = \vec{x}$
    $\epsilon_{now} = \|A\vec{x} - \vec{b}\|$ ▷ Current ϵ
    $dp = \|\vec{x} - \vec{x}_{prev}\|$ ▷ Change in $\vec{d}_{\text{data}}$
    if first iteration **then**
        $dtv = \alpha * dp$ ▷ Initialize TV hyperparameter
    end if
    $\vec{x}_{prev} = \vec{x}$

    for n_{TViter} **do** ▷ TV update
        $\vec{dx} = \nabla_{\vec{x}} \|\vec{x}\|_{TV}$
        $\hat{dx} = \vec{dx} / \|\vec{dx}\|$
        $\vec{x} = \vec{x} - dtv \cdot \hat{dx}$
    end for
    $dg = \|\vec{x} - \vec{x}_{prev}\|$ ▷ Change in $\vec{d}_{\text{TV}}$
    if $dg > r_{max} * dp$ **and** $\epsilon_{now} > \epsilon$ **then**
        $dtv = dtv * \alpha_{red}$
    end if
    ▷ Check stopping criteria
    $\cos\theta = \vec{dp} \cdot \vec{dg} / \|\vec{dp}\| \cdot \|\vec{dg}\|$
    if $(\cos\theta < -0.9$ **and** $\epsilon_{now} < \epsilon)$ **or** $\beta < 0.005$ **then**
        Stop
    end if
end while

B-ASD-POCS-β

Xue *et al*[CITE] proposed a faster converging modification of the ASD-POCS algorithm by adding a relaxed Bregman iteration to the overall algorithm. The Bregman iteration updates the projection data with part of the remaining error, thus reducing the residual faster and reaching convergence faster. The top level pseudo-code of the B-ASD-POCS-β can be seen in algorithm 2

Algorithm 2 B-ASD-POCS-β

```
Set: β, β_red, n_TViter, α, α_red, r_max
β update ratio:c < 1
β update period: T
x⃗ = 0;
while Stopping criteria not met do
    for n_Bregman do
        ASD-POCS algorithm
    end for
    b⃗ = b⃗ + β · (b⃗^0 − A · x⃗)
    Update β = c · β if current iteration is update period T multiplier
end while
```

The gradient of the TV norm

In order to minimize the TV norm via gradient descend, the gradient of the TV norm needs to be computed, $\nabla_{\vec{x}}||\vec{x}||_{TV}$, being $\vec{x}$ the vectorized form of a N-dimensional image.

The main challenge wit the $\nabla_{\vec{x}}||\vec{x}||_{TV}$ term is that $||\vec{x}||_{TV}$ is not differentiable in the general case. However, in the CT case, $\vec{x}$ can be described as x_{ijk}, a regularly discretized mesh of directional indices i, j, k of maximum value $i_{max}, j_{max}, k_{max}$. The gradient of x has an additional Cartesian index α:

$$g^{\alpha} = (\nabla x)^{\alpha} = \partial_{\alpha} x \tag{3.25}$$

$$g^{\alpha}_{ijk} = \partial_{\alpha} x_{ijk}. \tag{3.26}$$

The TV norm can me then defined as sum of the 2-norms of the gradient of x, g, over the Cartesian coordinate, resulting in a scalar.

$$\|x\|_{\mathrm{TV}} = \sum_{ijk} \sqrt{\sum_{\alpha} \left(g_{ijk}^{\alpha}\right)^2} = \sum_{ijk} \sqrt{\sum_{\alpha} \left(\partial_{\alpha} x_{ijk}\right)^2}, \tag{3.27}$$

This is the term that the total variation regularization algorithm minimizes with a gradient descend. In order to perform this, the gradient of this term with respect to x is needed, now defined in a scalar field

$$(\nabla_{\vec{x}} ||x||_{TV})_{ijk}. \tag{3.28}$$

This derivative can be expanded to a 3 component value for each x_{ijk} as:

$$\begin{aligned}
(\nabla_x \|x\|_{\mathrm{TV}})_{ijk} &= \frac{\partial}{\partial x_{ijk}} \|x\|_{\mathrm{TV}} = \partial_{x_{ijk}} \sum_{i'j'k'} \sqrt{\sum_{\alpha} \left(\partial_{\alpha} x_{i'j'k'}\right)^2} \\
&= \sum_{i'j'k'} \partial_{x_{ijk}} \sqrt{\sum_{\alpha} \left(\partial_{\alpha} x_{i'j'k'}\right)^2} \\
&= \sum_{i'j'k'} \frac{\sum_{\alpha} \left(\partial_{\alpha} x_{i'j'k'}\right) \partial_{x_{ijk}} \left(\partial_{\alpha} x_{i'j'k'}\right)}{\sqrt{\sum_{\alpha} \left(\partial_{\alpha} x_{i'j'k'}\right)^2}}.
\end{aligned} \tag{3.29}$$

This term now contains ∂_{α} derivatives, i.e. derivatives in the Cartesian coordinate system $[x, y, z]$. These are defined as

$$\begin{aligned}
\partial_x x_{i'j'k'} &= \lim_{h \to 0} \frac{x_{i'+h,j',k'} - x_{i',j',k'}}{h} \\
\partial_y x_{i'j'k'} &= \lim_{h \to 0} \frac{x_{i',j'+h,k'} - x_{i',j',k'}}{h} \\
\partial_z x_{i'j'k'} &= \lim_{h \to 0} \frac{x_{i',j',k'+h} - x_{i',j',k'}}{h}.
\end{aligned} \tag{3.30}$$

However, x is discrete, thus the limit definition of the derivative can not be used to numerically compute it, but an approximation of it can. By setting $h = 1$, equation 3.30 becomes the backward finite differences of the first order approximation of a derivative, a very computationally cheap operation. The derivative w.r.t. the Cartesian coordinate can be rewritten as

$$\partial_\alpha x_{i'j'k'} = \delta_{\alpha x}\left(x_{i',j',k'} - x_{i'-1,j',k'}\right) + \delta_{\alpha y}\left(x_{i',j',k'} - x_{i',j'-1,k'}\right) + \delta_{\alpha z}\left(x_{i',j',k'} - x_{i',j',k'-1}\right) \tag{3.31}$$

where δ_α is a Kronecker delta for the Cartesian axis. The other partial derivative term that appears in equation 3.29 is $\partial_{x_{ijk}}\left(\partial_\alpha x_{i'j'k'}\right)$. As the derivative is w.r.t. x_{ijk}, each component of x is an independent variable, thus $\partial_{x_{ijk}}\left(\partial_\alpha x_{i'j'k'}\right)$ is zero everywhere but in indices $i = i' \wedge j = j' \wedge k = k'$, where the derivative is 1. The term then becomes

$$\begin{aligned}
\partial_{x_{ijk}}\partial_x x_{i'j'k'} &= \partial_{x_{ijk}}\left(x_{i',j',k'} - x_{i'-1,j',k'}\right) = \delta_{i',i}\delta_{j',j}\delta_{k',k} - \delta_{i'-1,i}\delta_{j',j}\delta_{k',k} \\
&= \delta_{i',i}\delta_{j',j}\delta_{k',k} - \delta_{i',i+1}\delta_{j',j}\delta_{k',k} \\
\partial_{x_{ijk}}\partial_y x_{i'j'k'} &= \partial_{x_{ijk}}\left(x_{i',j',k'} - x_{i',j'-1,k'}\right) = \delta_{i',i}\delta_{j',j}\delta_{k',k} - \delta_{i',i}\delta_{j'-1,j}\delta_{k',k} \\
&= \delta_{i',i}\delta_{j',j}\delta_{k',k} - \delta_{i',i}\delta_{j',j+1}\delta_{k',k} \\
\partial_{x_{ijk}}\partial_z x_{i'j'k'} &= \partial_{x_{ijk}}\left(x_{i',j',k'} - f_{i',j',k'-1}\right) = \delta_{i',i}\delta_{j',j}\delta_{k',k} - \delta_{i',i}\delta_{j',j}\delta_{k'-1,k} \\
&= \delta_{i',i}\delta_{j',j}\delta_{k',k} - \delta_{i',i}\delta_{j',j}\delta_{k',k+1}.
\end{aligned} \tag{3.32}$$

These terms are practically a selecting function for i', j', k' ,matching only in the indices $i, i+1, j, j+1, k, k+1$ in the sum of the right hand side of equation 3.29. However the indices are limited to $i' \in [1, i_{max}]$, $j' \in [1, j_{max}]$ and $k' \in [1, k_{max}]$. As boundary conditions, Neumann boundary conditions are set to zero. To enforce that, a Kronecker deltas can be introduced for each index, $(1 - \delta_{i,i_{max}})$, with the same approach with the other indices.

Finally, substituting in equation 3.29, the gradient of the TV norm can be described as

$$
\begin{aligned}
(\nabla_x \|x\|_{\mathrm{TV}})_{ijk} &= \sum_{i'j'k'} \frac{\sum_\alpha \left(\partial_\alpha x_{i'j'k'}\right) \partial_{x_{ijk}} \left(\partial_\alpha x_{i'j'k'}\right)}{\sqrt{\sum_\alpha \left(\partial_\alpha x_{i'j'k'}\right)^2}} \\
&= \sum_{i'j'k'} \frac{\partial_x x_{i'j'k'} \partial_{x_{ijk}} \left(\partial_x x_{i'j'k'}\right) + \partial_y x_{i'j'k'} \partial_{x_{ijk}} \left(\partial_y x_{i'j'k'}\right) + \partial_z x_{i'j'k'} \partial_{x_{ijk}} \left(\partial_z x_{i'j'k'}\right)}{\sqrt{\sum_\alpha \left(\partial_\alpha x_{i'j'k'}\right)^2}} \\
&= \frac{\partial_x x_{i,j,k} + \partial_y x_{i,j,k} + \partial_z x_{i,j,k}}{\sqrt{\sum_\alpha \left(\partial_\alpha x_{i,j,k}\right)^2}} \\
&- \frac{\left(1-\delta_{i,i_{max}}\right) \partial_z x_{i+1,j,k}}{\sqrt{\sum_\alpha \left(\partial_\alpha x_{i+1,j,k}\right)^2}} - \frac{\left(1-\delta_{j,j_{max}}\right) \partial_y x_{i,j+1,k}}{\sqrt{\sum_\alpha \left(\partial_\alpha x_{i,j+1,k}\right)^2}} - \frac{\left(1-\delta_{k,k_{max}}\right) \partial_z x_{i,j,k+1}}{\sqrt{\sum_\alpha \left(\partial_\alpha x_{i,j,k+1}\right)^2}}.
\end{aligned} \tag{3.33}
$$

Equation 3.33 is the numerical approximation of the gradient of the total variation norm, and describes scalar field of the same size of the image. The same approach can be used with central and forward differences to obtain a similar equation, however central differences may not correctly minimize the TV norm of the image. As central differences do not take into account the value of the current voxel ijk, a chequerboard pattern would have zero TV norm, and this is the opposite of the purpose of the algorithm, therefore only numerical approximations of derivatives that take immediately adjacent pixel values into account can be used (such as forward or backward finite differences).

3.3.5 Total variation regularization via Rudin-Osher-Fatemi model

A different minimization approach to POCS is the approach proposed by Jia *et al*[7], that uses the Rudin-Osher-Fatemi (ROF) model for total variation minimization, widely used in the denoising literature[15][5][20]. By starting from the same minimization problem, namely

$$
\hat{x} = \arg\min_x \|Ax - b\|^2 + \lambda \|x\|_{TV}, \tag{3.34}
$$

and a forward-backward splitting algorithm[4] is used to split the minimization into two alternating steps, the TV and the data steps. If the optimality condition is considered to be

$$
\frac{\partial}{\partial x_\alpha} \|Ax - b\|^2 + \lambda \frac{\partial}{\partial x_\alpha} \|x\|_{TV} = 0, \tag{3.35}
$$

being α the set of Cartesian coordinates, then the problem can be split into the following equations, where g is a auxiliary function and $\mu > 0$:

$$\lambda \frac{\partial}{\partial x_\alpha} \|x\|_{TV} = \mu(x - g) \tag{3.36}$$

$$\frac{\partial}{\partial x_\alpha} \|Ax - b\|^2 = -\mu(x - g). \tag{3.37}$$

By solving for g, the simplified version of the algorithm can be seen in 3.

Algorithm 3 TV minimization with ROF model

1: Solve: $g = x - \frac{\lambda}{\mu} \frac{\partial}{\partial x_\alpha} \|Ax - b\|^2$ ▷ SART
2: Minimize: $x = \arg\min_x \|x\|_{TV} + 0.5 * \mu \|x - g\|^2$
3: Enforce positivity: $x = \max(0, x)$

The first line of the algorithm its essentially a gradient descend iteration, which is essentially a SART iteration. Note that the SART iteration can be replaced by other data-minimization algorithms such as CGLS. The second line is the ROF model, widely researched in image denoising. The ROF model tries to find the image x with minimum total variation subject to having the minimal deviation from its original value g. By changing the value of he hyperparameter μ, the strength of this regularization is controlled. A high μ will ensure that the image is very similar to its original value, while a small μ will be more lax. The advantage of this approach compared to the ASD-POCS algorithm is that it requires no extra projection or backprojection operations. Additionally, minimizing the ROF model is a very well studied problem in the image processing field, and it has lead to finding computationally efficient methods.

In the article by Jia *et al*, they solve the ROF model via gradient descend and controlling its step size with Armijo's rule. In this work a different approach is taken, based on the image processing literature.

Primal Dual formulation of the ROF model

As previously shown in line 2 of algorithm 3, the ROF model can be formulated as

$$\hat{x}_{ROF} = \arg\min_x \|x\|_{TV} + \frac{\mu}{2} \|x - g\|^2. \tag{3.38}$$

A solution of this problem using a primal-dual (PDU) approach has been proposed in literature[21], by changing the minimization equation to a saddle point optimization problem. While a wide variety of methods have been proposed to minimize the ROF model[15][20][3], the PDU method has the advantage of being very parallelizable, thus

a perfect fit for GPU computing. The dual variable can be proposed by using the TV definition of $\|x\|_{TV} = \|\nabla x\|$ and observing the following consequence of the Cauchy-Schwartz inequality

$$\|\nabla x\| = \arg\max_{\|\mathbf{p}\|\leq 1} \|\mathbf{p}\nabla x\|, \tag{3.39}$$

where $\mathbf{p} = (p^1, p^2, p^3)^T$ (for the 3D case) is the said dual variable. Note that each p^i is the size of the image x. Equation 3.38 can be then rewritten as

$$\hat{x}_{ROF} = \arg\min_{x} \arg\max_{\|\mathbf{p}\|\leq 1} \|\mathbf{p}\nabla x\| + \frac{\mu}{2}\|x - g\|^2. \tag{3.40}$$

The primal and dual updates can be both obtained from this equation. For the primal update, differentiating the equation according to x results in

$$-\nabla \cdot \mathbf{p} + \mu(x - g) = 0, \tag{3.41}$$

and one can solve it for x by performing a gradient descend update as

$$x^{n+1} = x^n(1 + \tau_P^n) + \tau_P^n \left(g + \frac{1}{\mu}\nabla \cdot \mathbf{p} \right), \tag{3.42}$$

where τ_P is the primal step size. The dual update can be computed similarly, by differentiating equation 3.40 according to $\mathbf{p}$, the following equation si obtained:

$$\nabla x + \mathbf{p}\alpha = 0, \tag{3.43}$$

where α is a Lagrange multiplier for the inequality constrain $\|\mathbf{p}\| \leq 1$. This equation can be maximized with a gradient ascend method as

$$\mathbf{p}^{n+1} = \Pi_{B_0} \left(\mathbf{p}^n + \tau_D^n \nabla x \right), \tag{3.44}$$

where $\Pi_{B_0}(\mathbf{p}) = \frac{\mathbf{p}}{\max\{1, \|\mathbf{p}\|\}}$ is a projection onto the unit ball centred in the origin.

The PDU algorithm consists in updating $\mathbf{p}$ and x iteratively, by alternating the updates. In [21][10] a step size update is proposed for the primal and dual step sizes:

$$\begin{aligned} \tau_D^n &= 0.3 + 0.02n \\ \tau_P^n &= \frac{1}{\tau_D^n}\left(\frac{1}{6} - \frac{5}{15 + n} \right). \end{aligned} \tag{3.45}$$

The same update is used in this work, as the images in their work are structurally

similar to CT images and empirical test showed satisfactory results. The algorithm can be shown to converge as it is shown in [22] that the primal-dual gap decreases with each update of x^n, and the gap is suggested as a control variable for the stopping criteria. In this work the algorithm has been implemented without the stopping criteria check, and an user specified parameter for the number of iterations is passed as an input, with a default value of 50, as it empirically showed good results.

The discretization of the divergence and gradient operators are a key factor when numerically computing the PDU algorithm, as they need to be consequent with each other. Thus, the gradient can be approximated using forward differences, but as the divergence is the adjoint of the gradient, it must be approximated with backwards differences.

3.4 Discussion

Chapter 4

GPU methods in tomography

Tomographic reconstruction in 3D is not only challenging due to the mathematics of the reconstruction, but also comes with a significant computational burden. As an example, the detector may have 512^2 pixel per projection and the size of the reconstructed image is generally of the order of 512^3 voxels in medical applications. However, in micro-tomography the detectors can get to sizes such as 2000^2 pixels per angle with images of 2000^3 voxels. Such sizes are considerably big for standard computers and applying reconstruction techniques. Both "one pass" as FDK or iterative methods are massively computationally expensive, needing up to weeks to reconstruct the image if run on CPUs. This is in most cases an unreasonable waiting time, as images are required for immediate diagnosis or treatment adjustment when taken, so a faster solution is needed.

In iterative reconstruction techniques, the computational problem relies on the matrix A, generally used twice per iteration in most algorithms, by doing a projection $(A \cdot x)$ and a backprojection $(A^T \cdot b)$. The construction of this matrix is not possible in 3D tomography, due to its size. If we consider the medical case sizes presented before, with projections over 360 different angles, building explicitly the A matrix would require thousands of gigabytes of RAM memory just to store the 0.0017% of the matrix that has non-zero values. In order to avoid that, the common technique is to compute $A \cdot x$ and $A^T \cdot b$ as a single operation instead of computing A explicitly. This is possible because the matrix-vector operation $A \cdot x$ describes the result of the integral of the x-rays over the image and $A^T \cdot b$ describes a "smear" of the projection data onto the image in the corresponding voxels. Interestingly, both of this operands necessitate a massive amount of very independent and simple calculations.

Over the past years computational technology has evolved significantly. But Moore's law, which expresses the halving of transistor size every two years, is coming to an end

due to physical laws. Transistors are currently on the 14nm scale and, while they are expected to reach 5nm by 2020, this clearly already breaks Moore's law. As transistors reach this scale, quantum mechanics starts to have an important effect, especially quantum tunnelling where the electrons could just "jump"[1] to the other side of the transistor regardless of its state. Unless a breakthrough in the understanding and manufacturing of new materials that can overcome such effects is discovered, 5nm transistors is approximately the hard limit to how far processors can evolve. To overcome this limitation, research in different computer architectures has been a hot topic in the past decades. This lead to two separate, but similar advances: High Performance Computers (HPCs) and Graphic Processing Units (GPUs). HPCs developed in order to be able to accommodate the growing computational demands of researchers and industry, where big data and heavy parallel computations became more common. These are massive installations with an enormous power consumption and running costs. GPUs instead, advanced their technology in order to accommodate the more demanding graphic (or visual) specifications of mainly the video-game industry in personal computers. GPUs are intrinsically designed to run in personal devices, such as laptops or mobile phones, so they are designed to be not only fast but small, low consumption and cheap overall.

The computations on high-end graphics in video-games require similar algorithms and processes than some of the big computational problems in both industry and research. It turns out that the development of high throughput GPUs for video-games has brought a tool that can significantly help research methods nowadays, to the point that a new term has been coined: GPGPUs or General Purpose GPUs. GPGPUs are widely used in research, for example in molecular dynamic simulations[**?**], astrophysical hydrodynamics[**?**], artificial intelligence[**?**], and many more. The massively parallel architecture and large number of independent processors make GPGPUs the perfect tool to deal with computed tomography applications.

This chapter describes the core computational code used in this work. First, a further description of the GPU architecture is given. Then, the special features of these processors that make tomography, and especially iterative reconstruction algorithms, a good fit for GPGPUs is shown. Next, a detailed description is given of how the projection operator has been implemented and optimized for two different projection approaches. And finally, a similar description for both of the backprojection modes implemented is given.

This chapter focuses on CBCT geometry, but parallel geometries are also implemented in the code. Everything (both computational and algorithmic) discussed for CBCT is essentially the same for any other geometry, with some minor tweaks

[1] Explanations about quantum mechanics are far beyond the goals of this thesis.

applied. All the methods discussed in this section and in the TIGRE toolbox are available open source and free to use/access in the paper[**?**] and GitHub repository github.com/CERN/TIGRE.

4.1 Hardware used in this research

Prior to the description of hardware architectures and programming tricks, it is important to describe the hardware that this research was developed on. While the work presented here applies to almost all different hardware types, it may not be 100% applicable to any GPU. GPUs are constantly changing so some features and computational tricks used for the acceleration of CT code are recent additions to GPGPUs and similarly new improvements to GPGPUs will come in the future that may render obsolete the information presented in this chapter. Therefore, this work presents techniques and benchmarks based on the hardware that was available and, while most of the techniques are applicable to most of GPUs, the mileage may vary. Additionally, most of the terminologies used in this chapter are related with NVIDIA/CUDA.

The research on this (and further) chapters of this thesis has been performed on a PC with a Windows 7 x64 operating system, with 32Gb of on board RAM, and a SSD hard drive. The GPU used was a NVIDIA Tesla 40k, that has 2880 stream processors, a clock frequency of 745-875 MHz and an on board RAM of 12GB. The processing power is generally described in terms of floating-point operations per second or FLOPS, and this particular GPU has a theoretical throughput of 4.29-5.04 TFLOPS on single-precision numbers.

4.2 GPGPU architecture

In order to better describe how GPU acceleration boosts tomographic reconstruction, a description of the hardware architecture of GPUs is required. As previously mentioned, GPUs are a technology that evolved from the increasing requirements of the entertainment industry in general but particularly from video games. In order to be able to have more realistic real-time graphics, where the environment reacts to the user interface by changing light reflections, textures, and simulating the physics of objects, a high throughput hardware is required. All these effects need a large amount of simple arithmetic to be computed and a fast access memory, faster than any modern CPU can handle. For that reason computers started having GPUs, special dedicated hardware that included hundreds of small, low-power, low-speed processors. These processors are significantly worse than any CPU, but the high number of them

allows very high throughput for any arithmetically heavy algorithm. However, this high computation-intensive output design also means that GPUs perform weakly on programs that need control flow and caching. This information is key for the correct design of GPU algorithms as will become more evident further in this chapter.

The general diagram of the GPU architecture is presented in figure 4-1. It has 3 main parts, the computational core, the device memory (DRAM) and the communication with the CPU via PCI express (PCIe). The principal part of the GPU is the computational part. This consists of several stream multiprocessors (SMs), 15 in the Tesla k40, that are responsible for the distribution of the instructions to the stream processors (SPs) or CUDA cores. Each of the SPs (2880 in total) can run a single *block* of instructions at any time, each of them consisting of up to 1024 parallel[2] *threads*. These blocks must be running the same algorithm, or *kernel*. Theoretically, each of the SPs can have 64 *concurrent warps*, or execution instructions running at the exact same clock cycle, each of these warps having 32 threads. This means that the Tesla k40 can have a theoretical maximum of 30720 arithmetic operations simultaneously executing. This is never reached as some of this time may be spent in flow control or memory reads, thus slowing the execution.

The second part of the GPU are the memory types. There are 3 types of memory in the GPU: registers, shared memory and device random access memory (DRAM). The main difference between them is in accessibility (what subset of the SPs can access it) and speed. The lowest level memory is register memory. This is a thread level memory, used to store all local variables during execution. It is fast access and low in size. No other thread or block can access this memory. The second memory type is shared memory. This memory is used when blocks need to share information, such as the result of a computation in the middle of the kernel. Each SM has its own shared memory and it is local to them. All blocks can access to it, but careful usage of it is needed as the parallel nature of the computation could mean multiple writing on the same memory by different blocks. This memory is bigger (up to 48KB) and it is fast access, but slower than registers. Finally, there is DRAM memory. DRAM memory is a global large (12GB) memory on the board. This memory is widely used as it is the place where memory before kernel calls is allocated and where the results are stored after kernel calls. However, GPU code will often need to work with big data, thus the DRAM read/write is also commonly used within kernels. DRAM accessing can lead to significant bottlenecks as a single memory access needs 200 to 300 clock cycles to be carried.

[2]In GPU computing, one should not confuse *parallel* with *concurrent*. Parallel means that the same instruction is divided into pieces, but they are not necessarily executed at the same time.

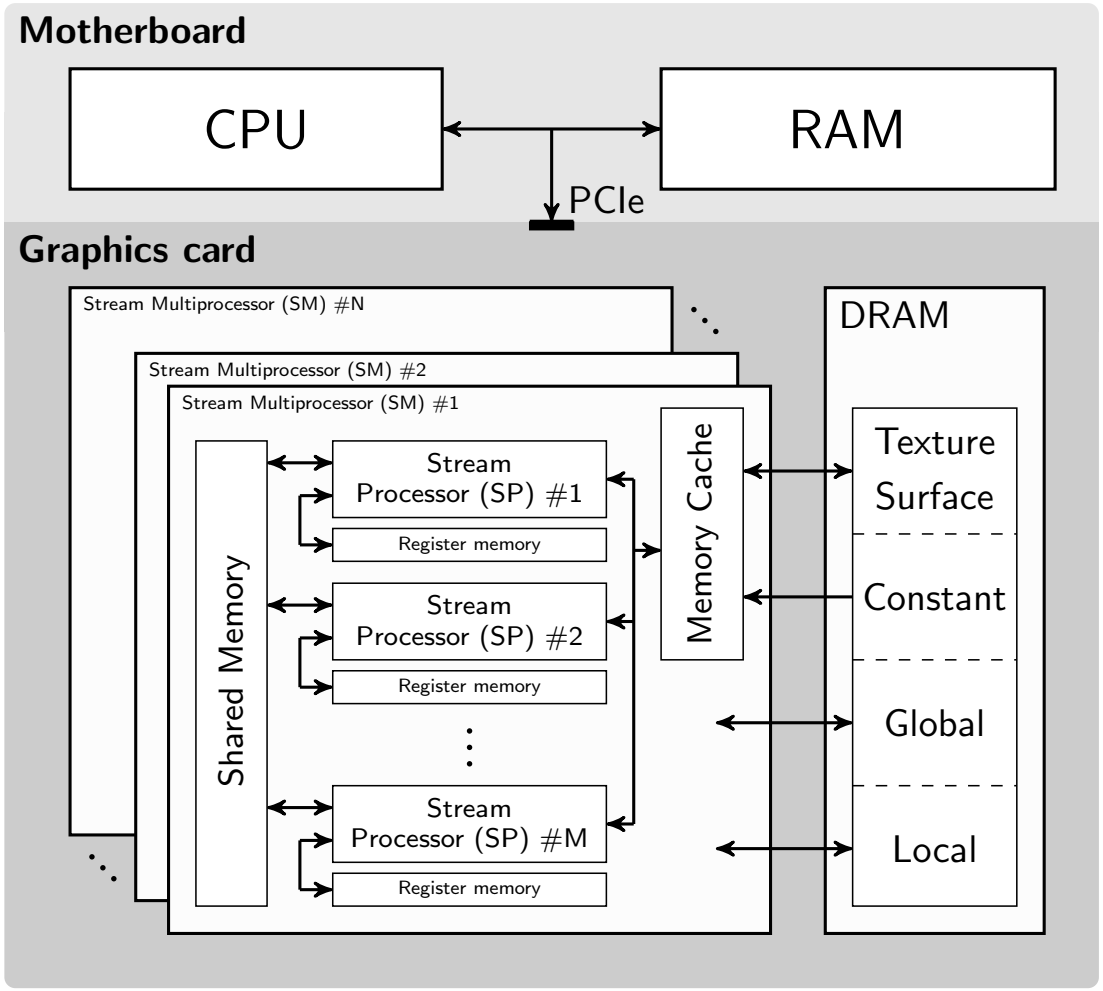


Figure 4-1: Diagram of a the basic architecture of a GPU. It shows how processing units and memory is structured inside a graphics card.

To improve memory throughput, the GPU contains so-called L1/L2 cache memory on each of the SMs. This cache is designed to improve execution of *memory coalescing* warps. The cache, having a faster communication bus with the threads (needing about 80 cycles for a single memory read) stores a large amount of memory each time a single read to the DRAM is performed. The memory is loaded with locality, i.e., a memory chunk around the first sampled value is loaded, as the cache expects subsequent memory reads to be adjacent. Thus, when designing kernels, the order of the memory access can have a big influence on the final speed of the program.

Two memory types can be used when data locality needs to be exploited in cache access: texture and surface memory. The difference between them is that the former is read only while the later can also be written. These memories can be defined as 3D shaped, thus data locality can be exploited in all spatial dimensions. Additionally, both can be accessed with floating point index values, and the memory value is returned with user-chosen interpolated methods. If cached memory is desired, but not in a spatially correlated form, then the read only constant memory can be used. This memory is still cached and can be used to speed up values that may be needed often. Then there is the global memory, which is read and write but not cached. Finally, local memory is an extension of shared memory when it gets filled.

The third important section is the communication between CPU-GPU. This is done via PCI-express ports and it is a slow communication process. Passing data from CPU to GPUs is relatively fast (500Mb/s), but can be a bottleneck in memory-heavy applications such as CT imaging.

4.2.1 Exploiting GPGPUs for CT

In CT, two main operations need to be accelerated: projection and backprojection. While iterative algorithms are defined as algebraic methods using a big system matrix A, the matrix itself is rarely used alone in the equations, it is generally used as $A * x$ or $A^T * b$. Fortunately, these two operations have a physical meaning, as the projection is the integral of the image over the straight X-ray paths, and the backprojection is the "smearing" of the detector data over the image back in the direction of the source, also following straight paths. While computing the values of the rows and columns of the system matrix seems hardly possible in real time, these two operations can be performed abstracting from their algebraic equivalent. The projection operation is an integration over independent, yet closely related, straight paths. By being able to sample the image domain in parallel, the operation can be performed at high speed. Similarly, the backprojection relies on building straight lines from the source to the voxels or detectors (dependent on the backprojection type), and sampling the projections. These

operations are completely independent, hence the idea for GPU parallelism as threads will reach their maximum performance when they do not need to communicate with each other. Additionally, both operations need accurate sampling over large volumes, thus texture memory with interpolation is ideal as it is hardware optimized, giving faster interpolated values than in a CPU or any possible interpolation kernel. Thus the massive parallelism and texture memory are they key features of GPU computing that make it the ideal tool for accelerating tomography. The following sections provide more details.

4.3 The projection operator

The projection operator is the numerical equivalent of the X-ray integral that defines the model of X-ray tomography (see equation XX). This operator models the idealized physics, where all the X-rays have an infinitely small width and travel in a straight path to the center of each detector pixel, and all with the same energy. While this may not represent the physics accurately enough to be a reliable X-ray simulation tool, it is exactly what iterative methods need, the $A * x$ operation.

There are several method to simulate forward projection, all of them easily parallelizable. One of these is the distance-driven projection [**?**][**?**], where the ray-voxel and detector intersections are all projected in a mid-plane and values are accumulated there. Alternatively, the voxel-driven projector[**?**], where the whole voxel (square or other shape[**?**]) is projected onto the detector and its values spread among all the corresponding pixels. With a similar approach, the separable footprints technique[**?**][**?**] approximates the footprint of the voxel in the detector for speed-up. According to the authors it is more accurate than the distance-driven projection and faster than both the voxel-driven and distance-driven ones. Finally, in ray-driven projection[**?**][**?**][**?**] methods the line path is integrated. Among these, the most important variations worth mentioning are the infinitesimally small exact path, area path, and the grid-interpolated path. The exact path computes either the length of an infinitesimally narrow path or the area of a finite width path over each voxel and uses that as a weight for the integral. The grid-interpolated projector instead sets a fixed sample length and interpolates voxel values. According to a study by Fang and Mueller[**?**], the most accurate methods are the ray-driven ones.

This work has focused on the infinitesimally small ray-voxel intersection and grid-interpolated methods only because in both the desired accuracy and speed have been reached. The grid-interpolated method also is a key method for the following chapters (see Chapter 6 for more information). As the projection is basically an integral of

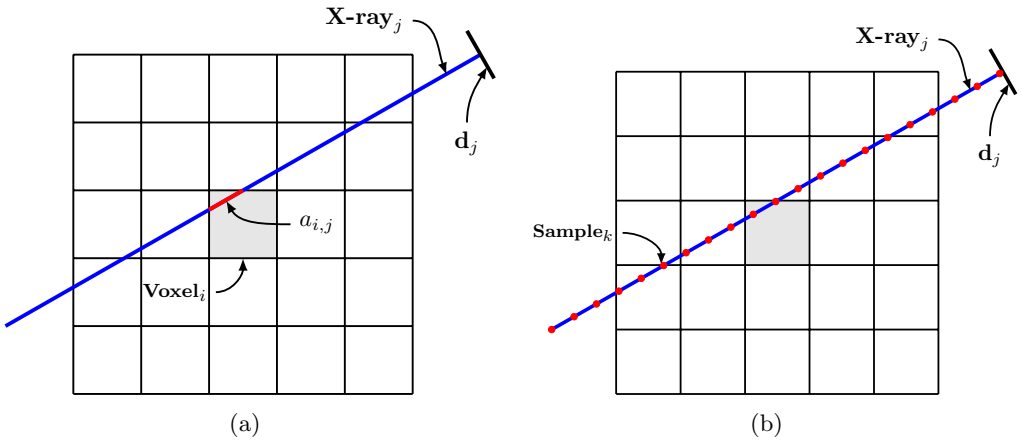


Figure 4-2: (a) Diagram of the projection operation using the line-voxel intersection methods and (b) diagram of the interpolated sampling method.

a volume over thousands of independent paths, it is straightforward to parallelize by independently computing each ray. The ray-voxel intersection method is equivalent to the algebraic representation, where the A matrix contains the length of the intersection between voxels and the path. Those are multiplied and added to the voxel values themselves to obtain the detector values. However, while not feasible in CPUs, the grid-interpolated method can use the practically free (i.e., very little time overhead) texture memory interpolation for speed-up in GPUs. The difference between the methods can be seen in figure 4-2.

4.3.1 Ray-voxel intersection method

As previously mentioned, this method relies on accumulating the length of the intersection between a straight path and voxels multiplied by the voxel value. The X-ray integral from equation XX can be discretized as in equation 4.1, where d_{uv} is the detector value of X-ray uv, $\mathbb{I}(ijk)$ are the voxel values of the image and $l_{uv,ijk}$ the length of the path within each voxel. Note that here $l_{uv,ijk}$ are the same values as the elements of matrix A. Computing $l_{(uv,ijk)}$ requires some geometrical computations, not only the reliable detection of the intersections between the lines and voxel boundaries, but also avoiding computing it on voxels where the line does not intersect.

$$d_{(uv)} = \sum_{ijk=0}^{N_{\mathbb{I}}} l_{(uv,ijk)} * \mathbb{I}(ijk) \tag{4.1}$$

The algorithm to compute d_{uvj} has been taken from Jacobs[**?**] improvement on

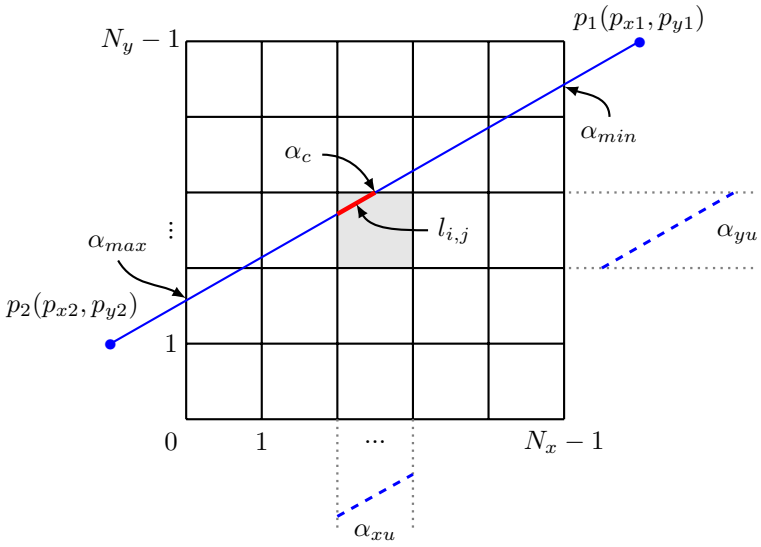


Figure 4-3: Diagram of the relevant variables for Jacobs' ray-tracing algorithm.

Siddon's method[**?**]. For the sake of clarity, the algorithm is described for the 2D case, but the extension to 3D is trivial. The algorithm is based on computing the intersection between the path and the x and y planes and iterating over the ray until the next intersection is found. The diagram of the variables used in the algorithm can be seen in figure 4-3. The following derivation assumes that the pixels are of size 1 and the image domain starts at $(0, 0)$, as this assumption reduces the number of operations needed. Assuming non-trivial rays from point (p_{x1}, p_{y1}) to point (p_{x2}, p_{y2}), the parametric representation of the ray is

$$p_{12} = \begin{cases} p_x(\alpha) & = p_{x1} + \alpha(p_{x2} - p_{x1}) \\ p_y(\alpha) & = p_{y1} + \alpha(p_{y2} - p_{y1}), \end{cases} \tag{4.2}$$

where $\alpha \in [0, 1]$. In order to know the number of intersections, the initial and final intersections of the ray in the image are needed. The intersection points in terms of alpha can be defined as

$$\alpha_x(i) = \frac{(i - p_{x1})}{p_{x2} - p_{x1}} \tag{4.3}$$

$$\alpha_y(j) = \frac{(j - p_{y1})}{p_{y2} - p_{y1}}, \tag{4.4}$$

where i and j are indices of the pixels. If the number of planes is defined as N_x and N_y, one can compute the minimum and maximum α values (i.e., values of the line at

the boundary of the image) by comparing the values of α in each direction as

$$\alpha_{min} = \max\Big(\min\big(\alpha_x(0), \alpha_x(N_x - 1)\big), \min\big(\alpha_y(0), \alpha_y(N_y - 1)\big)\Big) \tag{4.5}$$

$$\alpha_{max} = \min\Big(\max\big(\alpha_x(0), \alpha_x(N_x - 1)\big), \max\big(\alpha_y(0), \alpha_y(N_y - 1)\big)\Big). \tag{4.6}$$

Note that this is equivalent to writing

$$\alpha_{min} = \max\Big(\alpha_{xmin}, \alpha_{ymin}\Big) \tag{4.7}$$

$$\alpha_{max} = \min\Big(\alpha_{xmax}, \alpha_{ymax}\Big). \tag{4.8}$$

Next, the planes where the rays first cross in each direction need to be computed. This can be achieved by looking at the different α values. For the x dimension, equations 4.9-4.12 show how to compute the plane index i_{min} and i_{max} if $p_{x1} < p_{x2}$ and equations 4.13-4.16 when $p_{x1} > p_{x2}$. The same logic applies to j_{min} and j_{max}.

$$\alpha_{min} = \alpha_{xmin} \quad \rightarrow \quad i_{min} = 1 \tag{4.9}$$

$$\alpha_{min} \neq \alpha_{xmin} \quad \rightarrow \quad i_{min} = \lceil p_x(\alpha_{xmin}) \rceil \tag{4.10}$$

$$\alpha_{max} = \alpha_{xmax} \quad \rightarrow \quad i_{max} = N_x - 1 \tag{4.11}$$

$$\alpha_{max} \neq \alpha_{xmax} \quad \rightarrow \quad i_{max} = \lfloor p_x(\alpha_{xmax}) \rfloor \tag{4.12}$$

$$\alpha_{min} = \alpha_{xmin} \quad \rightarrow \quad i_{max} = N_x - 2 \tag{4.13}$$

$$\alpha_{min} \neq \alpha_{xmin} \quad \rightarrow \quad i_{max} = \lfloor p_x(\alpha_{xmin}) \rfloor \tag{4.14}$$

$$\alpha_{max} = \alpha_{xmax} \quad \rightarrow \quad i_{min} = 0 \tag{4.15}$$

$$\alpha_{max} \neq \alpha_{xmax} \quad \rightarrow \quad i_{max} = \lceil p_x(\alpha_{xmax}) \rceil. \tag{4.16}$$

At this point, the α_x and α_y values of the first intersection point can be obtained by substituting either (i_{min}, j_{min}) or (i_{max}, j_{max}) (depending on the relationship of p_1 and p_2) into equations 4.3 and 4.4. Additionally, one can compute the number of planes that the ray crosses (N_p) with the following equation:

$$N_p = (i_{max} - i_{min} + 1) + (j_{max} - j_{min} + 1). \tag{4.17}$$

In order to be able to start iterating over a given line there are just two pieces of

information missing, namely the initial pixel coordinates and the α_u, the maximum step in each direction for a unit of change. The initial pixel coordinates can be computed as

$$i = \left\lfloor p_x\Big(\frac{min(\alpha_x, \alpha_y) + \alpha_{min}}{2}\Big)\right\rfloor \tag{4.18}$$

$$j = \left\lfloor p_y\Big(\frac{min(\alpha_x, \alpha_y) + \alpha_{min}}{2}\Big)\right\rfloor. \tag{4.19}$$

The maximum α that can happen in a unit of change in each direction, or α_{xu} and α_{yu} are defined as in equation 4.20. Additionally it is useful to have a variable that controls the unit direction of the ray, i_u and j_u, as in equation 4.21.

$$\alpha_{xu} = \frac{1}{|p_{x2} - p_{x1}|} \tag{4.20}$$

$$i_u = \begin{cases} 1 & \text{if} \quad p_{x1} < p_{x2} \\ -1 & \text{otherwise} \end{cases} \tag{4.21}$$

Defining l_{tot} as the Euclidean distance between p_1 and p_2 and initializing the current α, $\alpha_c = \alpha_{min}$, the iterative method to follow the X-ray path can be described as follows. Check if the next intersection is in x or y by comparing the α values of each direction, then choose to update the direction that has a smallest α. When updating, compute the length of the next distance and update the α, pixel index and integral values. The update when $\alpha_x < \alpha_y$ can be seen in equations 4.22-4.25, and the opposite case in equations 4.26-4.29.

$$d_{ray} = d_{ray} + (\alpha_x - \alpha_c) \cdot l_{tot} \cdot \mathbb{I}(i, j) \tag{4.22}$$

$$i = i + i_u \tag{4.23}$$

$$\alpha_c = \alpha_x \tag{4.24}$$

$$\alpha_x = \alpha_x + \alpha_{xu} \tag{4.25}$$

$$d_{ray} = d_{ray} + (\alpha_y - \alpha_c) \cdot l_{tot} \cdot \mathbb{I}(i, j) \tag{4.26}$$

$$j = j + j_u \tag{4.27}$$

$$\alpha_c = \alpha_y \tag{4.28}$$

$$\alpha_y = \alpha_y + \alpha_{yu} \tag{4.29}$$

This process is repeated N_p times and, while there may be degenerate cases where a

cross-section between an x and y plane is repeated, the algorithm will compute a length of zero the second time, thus resolving the situation without need of a check.

This algorithm is highly parallelizable. It needs no memory but for a few scalars and, once the values of the required variables are computed, the iterative process that takes most of the time is defined by 4 simple equations. A few straightforward optimizations are also possible, such as multiplying by l_{tot} outside the for loop, in the end of the iterative process and precomputing the few scalar operands that are reused during the process to minimize the number of algebraic operations.

From the iterative section, the memory reads, $\mathbb{I}(i, j)$, are the most computationally expensive part. As previously commented, a single memory read takes 80 cycles as a best case, compared to just one for an algebraic operator involving two scalars. This is true for proper memory access. If the memory is accessed in a random manner, the memory latency increases massively. Thus making sure that single warps (32 simultaneous threads) read from memory in a similar matter is key. Additionally, *thread divergence* can slow down the overall execution. Thread divergence refers to the case where, due to control flow such as a different path in an **if** condition, threads compute different things and finish at different times. If this happens, they will stay idle until the slowest threads are finished, effectively wasting time. In order to decrease memory latency, texture memory is used. One of the features of texture memory is that the cache will assume data locality, thus loading a chunk of memory around the sampled value. If threads read around it, then the memory reads are faster. In order to implement that, the X-rays are divided into divU$\times$ divV pieces, and launched in each block together, as seen in figure 4-4. This ensures that the all rays are as close to each other as possible and hence that samples are taken next to each other. Interestingly, this approach also minimizes thread divergence, as X-rays very close to each other will cross each voxel boundary in a similar manner, and will most likely have the same N_p number of intersections. It is important to note though that the figure oversimplifies the real case scenario. The concurrent memory reads are not happening in a parallel plane to the detector as, due to the cone angle, some paths are longer than others and some paths can intersect more voxels than others. However, if $divU \times divV$ is small enough, then this divergence is small enough that it has no effect on the computation time because the difference in the source-to-detector direction is still within the cached memory size.

4.3.2 Grid-interpolated methods

The grid-interpolated method is significantly less complex than the ray-voxel intersection method. As shown in figure 4-2(b), it merely relies on sampling the X-ray

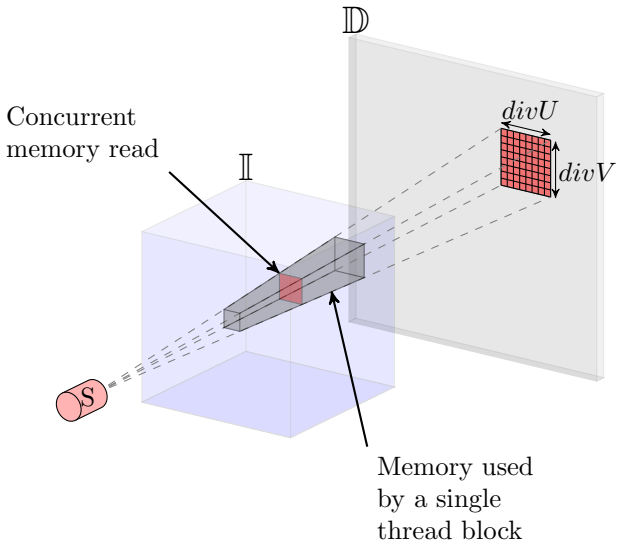


Figure 4-4: Diagram of the block level execution and memory access to increase data locality. Each block is composed of $divU \times divV$ rays, which are executed in parallel. This ensures that the concurrent memory reads are spatially adjacent, thus decreasing memory latency.

paths with a uniform Euclidean sampling rate. It can be described using equation 4.30 and its pseudocode is given in 4.

$$d_{(uv)} = \sum_{\alpha} \Delta l * \mathbb{I}(p_x(\alpha), p_y(\alpha), p_z(\alpha)), \tag{4.30}$$

where α is the parameter of the line equation and takes decimal values between 0-1 if defined in terms of the source and detector locations (see equation 4.2). Note that now the image $\mathbb{I}$ is sampled using not integer values, but decimal values, thus requiring interpolation. The texture memory cache includes hardware accelerated linear interpolation, so this is interpolation method used. Whereas in equation 4.1 the length in this projection method, Δl, is a constant the same for all samples and thus can be taken out of the summation.

The memory latency is significantly decreased by choosing the same structure for block and thread organizing as used in the other method and shown in figure 4-4. The sampling rate of α is the relevant parameter to set and, as discussed by Jia *et al*[**?**], it defaults to half the voxel size to ensure all data are used. The implementation in TIGRE additionally blocks the user from choosing a sampling rate larger than a voxel by limiting it to that size and changing the interpolation to nearest neighbour.

This method of projection is arguably more realistic than the ray-voxel projection

Algorithm 4 Grid interpolated projection

Precompute geometric constants
Require: N_{ray} threads organized in divU×divV blocks
for X-ray path **do**
    Compute $[x_p, y_p, z_p]$ sample position
    Sum+= Image(x_p, y_p, z_p)
end for
Detector(u, v)=$\Delta l \cdot Sum$
Ensure:

in the case where the voxel size is very big (the resolution is very low) as it treats image information as a continuous domain instead of square boxes.

4.3.3 Comments on optimization

In order to minimize the computation times several optimization tricks have been used in both projection types. As one of the challenges of GPU programming is that often the only way of knowing how to accelerate code or what approach to use is intuition and testing, describing the optimization tricks and tests used for acceleration can be important. Three tricks that are worth mentioning are implemented, namely choosing an optimal coordinate system, precomputing geometric unit changes, and the way out of bounds memory is handled. Aside from these, other small tweaks can be found in the code, such as never computing trigonometric functions inside the kernels, but they are all relatively trivial and no further comment on them will be made.

Optimal coordinate system

When designing a kernel one of the main considerations is to minimize the amount of arithmetic operations happening inside. In tomography specifically, the geometry is the most variable of the parameters. Images can be arbitrarily fine or coarse and voxels can be anisotropic with a wide range of sizes in each direction and so can the detector. Thus, when writing the kernel, a coordinate system is needed that can accommodate the flexibility of the geometry while still being minimal in arithmetic operations. In TIGRE, the following has been chosen for the projector operator.

As the image memory is static, and data needs to be sampled from it in the projection coordinate system, the image will never move, while the detector and the source will rotate around the axis of rotation. Thus the new system (x_p, y_p, z_p) is aligned with the image edges and it is centred at the first lexicographically indexed voxel, at the bottom corner of the image as seen in figure 4-5. Additionally, the units of the new system are set as voxels, regardless of the size of the voxels in each direction. This

new system reduces the amount of arithmetic operations as each sample point is now an index in the image memory, while in each kernel the vector from the source to the detector position must be computed arithmetically. Thus, in the main loop where the new sample location is needed, once the next point in the line is computed no more operations are needed to convert to memory index. This alone saves a significant time.

Precomputation of geometry

In order to generate the new coordinate system, some transformations need to be applied to the input geometry, as the units, source location and detector locations need to be set up. All these operations are performed outside the kernel and output 2 points and 2 vectors. The points are the location of the source and first detector pixel with respect to the new coordinate system origin, rotation angle and other geometric definitions. The vectors describe the unit change in projection coordinate systems of the detector pixel location, thus by knowing the location of the first detector pixel and their unit changes, finding any detector pixel location from its index is trivial.

Additionally, the advantage of structuring the geometry computations as described is that new geometric transformations can be implemented with zero increase in computational cost. For example, the addition of the rotation with 3 degrees of freedom of the detector, offset of the image with respect to the axis of rotation or offset of the detector implies no change in computational cost, the only change needed is the change in the 2 points and vectors described.

Sampling outside the image

In the ray-voxel method, sampling outside the image is not a worry as the amount of cross-sections and the initial intersection are computed in the algorithm. However, in the grid-interpolated method, a start and end point for sampling need to be chosen.

As the texture memory is cached and handled differently than a direct read into memory, it has multiple tunable features. One of these is the possibility of selecting the behaviour when a memory read is out of bounds, which in the case of this work is set to zero. It is important to note that this means the code will not generate an error for out of bounds memory reads. Two versions of the grid-interpolated method can be tested, one where there is a conditional check to see if the current point is inside the image and if true then sample, and another where the line is sampled from the source to the detector. Empirical tests show that avoiding the conditional statement and sampling over the whole path is 33% faster than checking if the sample is within bounds, even with geometry definitions where more than 50% of the samples lie outside the image.

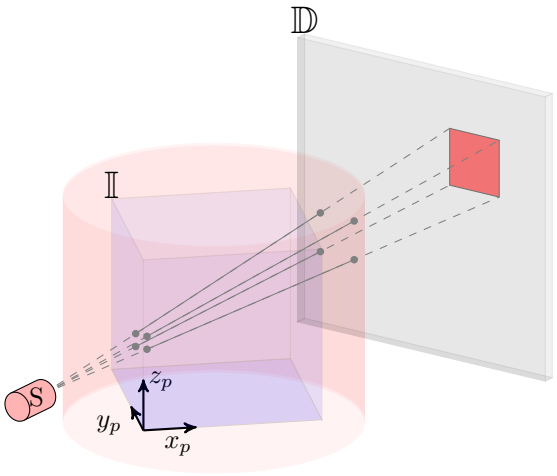


Figure 4-5: Diagram of the projection coordinate system and sampling region. In both projection operations the new coordinate system ((x_p, y_p, z_p)) has its origin on the lexicographically first voxel center. The red cylinder shows the sampling region for the grid-interpolated method, where the kernels sample from memory.

This can be explained by the fact that CUDA cores are quite slow with code control flow and possibly[3] by the cache taking virtually no time returning a zero value when accessed out of bounds.

Sampling the x-ray path from the start to the end is not optimal either, even avoiding conditional statements. To minimize the memory reads, the diameter of a cylinder that encloses the image is precomputed and the rays are sampled from the beginning of the cylinder to either the detector or to the end of the cylinder, whichever comes first. This approach speeds the projection kernel by another 20%. This is shown in figure 4-5.

4.3.4 Differences between operators

The projection operators, while effectively simulating the same physics, have slightly different results due to the methods used. The difference between the two projection operators is enhanced when the voxel size of the images is very big, i.e., when the image has low resolution. Figure 4-6 shows this effect. A projection of the 3D Shepp-Logan phantom is shown at different resolutions for both projection types. In the figure, four image resolutions can be seen for the same size, 64^3, 128^3, 256^3 and 512^3 from top to bottom. From left to right the first two columns show the ray-voxel intersection and the grid-interpolated method, while the last two columns show a zoomed in version of

[3]This is undocumented so, while it is very likely, it is hard to claim with full certainty.

the same projections. Figure 4-7 shows the differences between the projections.

The ray-voxel intersection method does introduce higher aliasing-like artefacts to the projection, as opposed to the interpolating method that smooths everything. Note however that when the image resolution gets higher, the differences are almost indistinguishable. None of the projection modes is better or worse. One could argue that the ray-voxel method aligns better with the discretization of the domain, or that the interpolated method is better because it generates images that are closer to what is measured in a real detector.

When used in reconstruction, the differences between the images reconstructed with algorithms using one or the other projection types are insignificant, with generally a maximum value of about 0.1% of the highest value in the image. This can be seen in figure 4-8, where a reconstruction of the XCAT[16] phantom of size 256^3 using OS-SART with 200 iterations and 100 projections is shown. It shows the result using both projection operators, and figure 4-9 the contrast enhanced differences (the colourmap is enhanced to 10% of the maximum data value) of both reconstructions against the original image. Both reconstructed images are visually very similar and the enhanced difference images show structural differences in the error, but they are still of the same level. The sum of square errors shows a slight higher value for the ray-voxel method, but not big enough to be significant. This difference is even smaller when working with higher resolution images. TIGRE defaults to the interpolated projection in the algorithms.

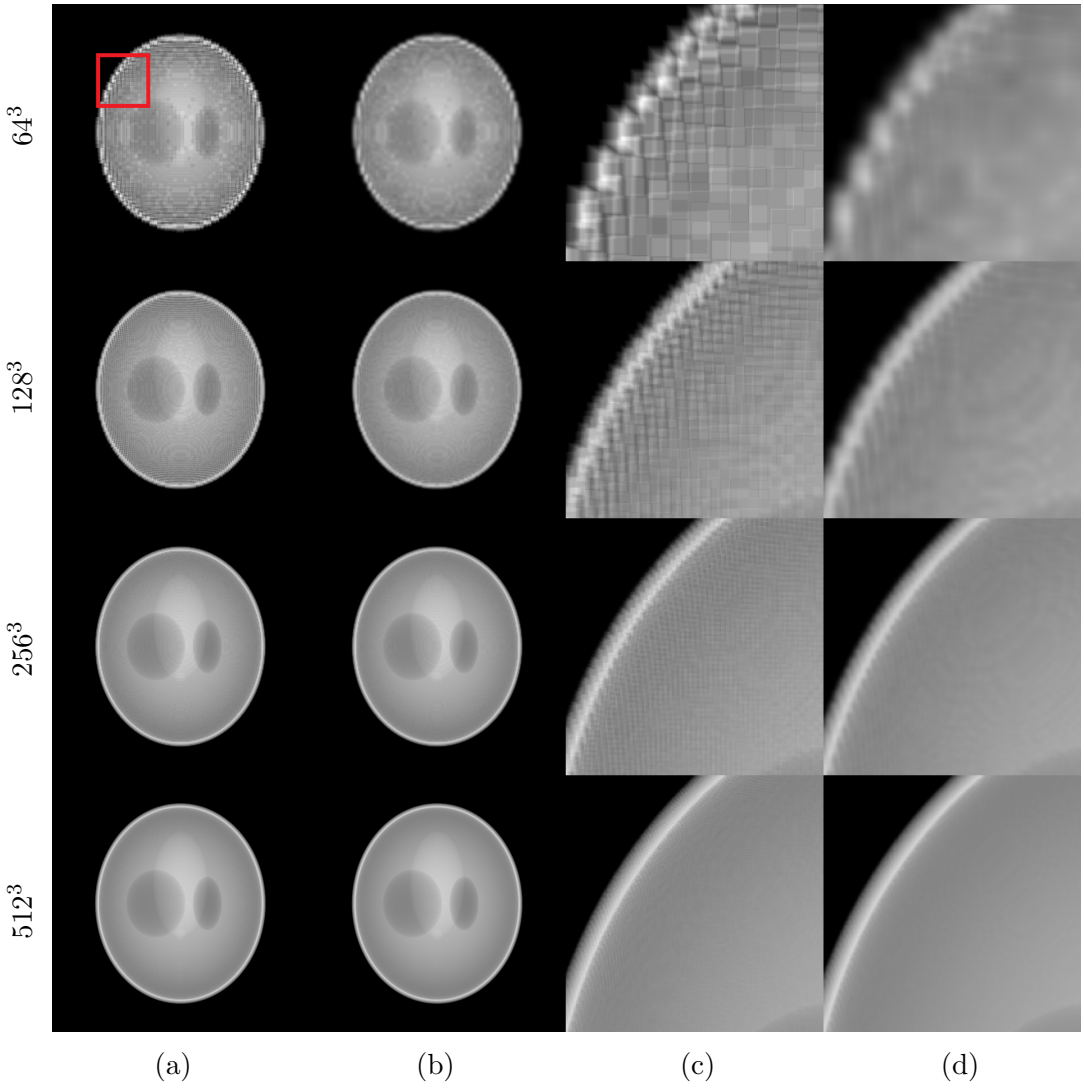


Figure 4-6: Different projection modes for different image resolutions. From top to bottom, the image resolution is 64^3, 128^3, 256^3 and 512^3 respectively. From left to right, (a) the ray-voxel intersection projection; (b) the grid-interpolated projection; (c) a zoomed in version of (a); and (d) a zoomed in version of (b).

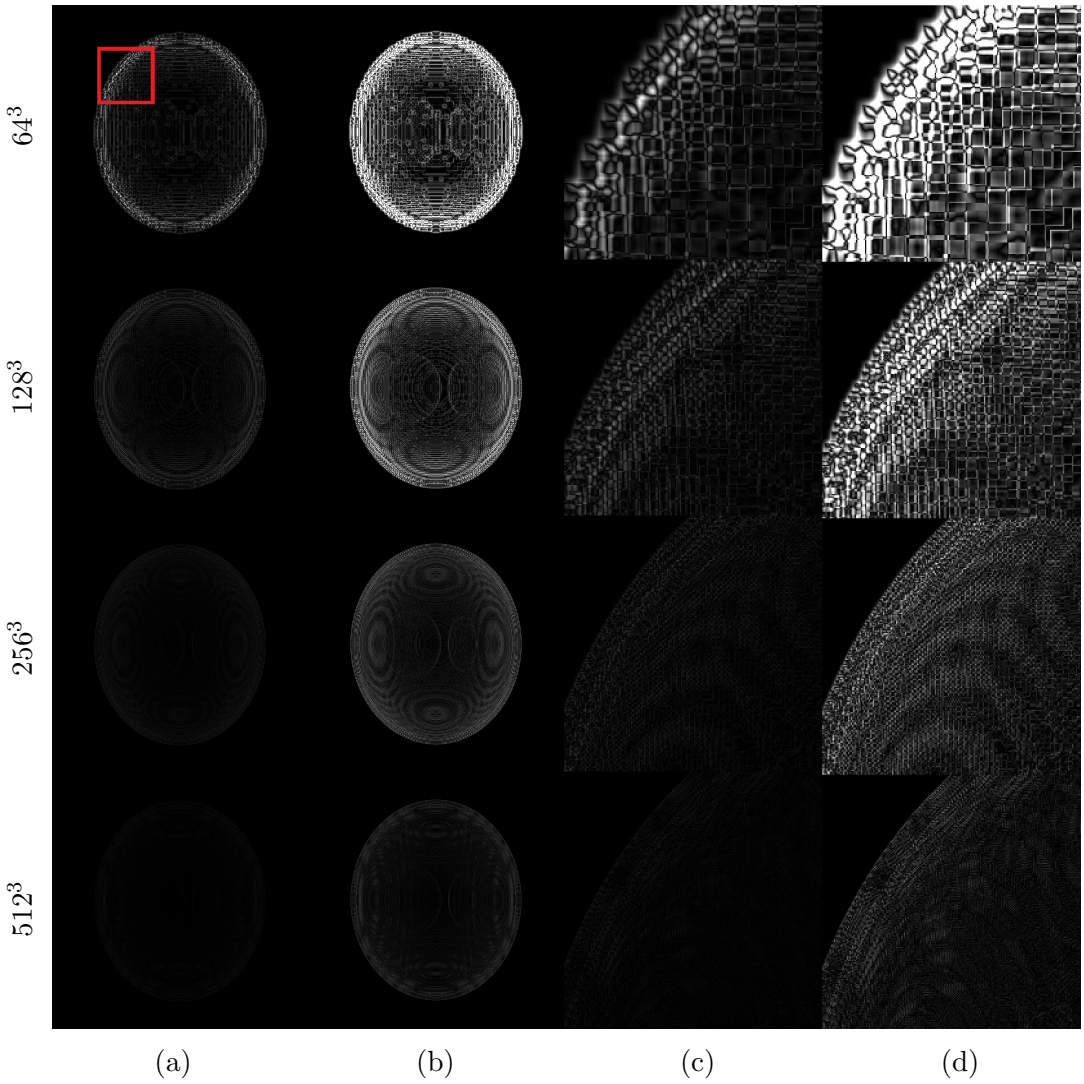
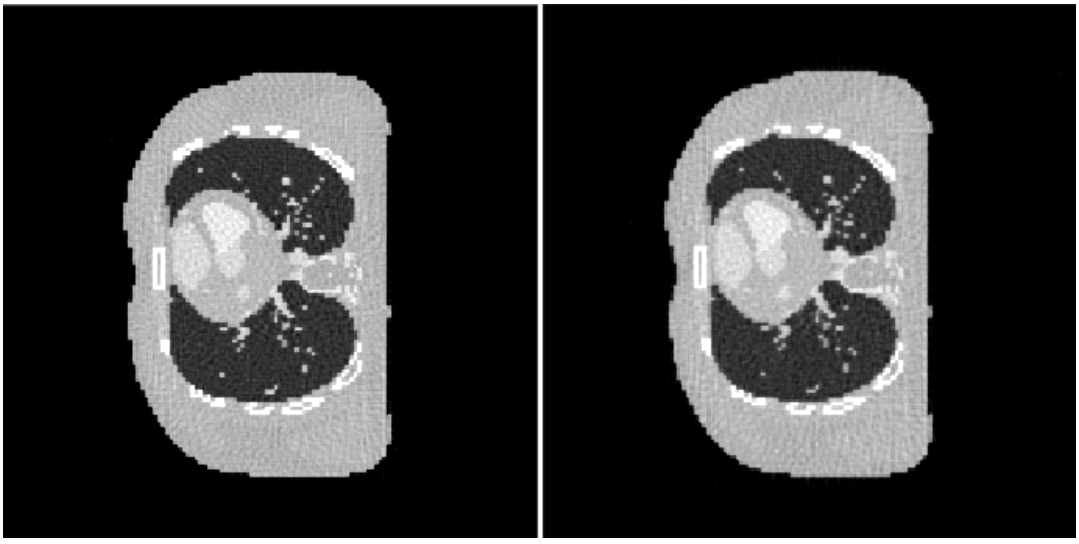
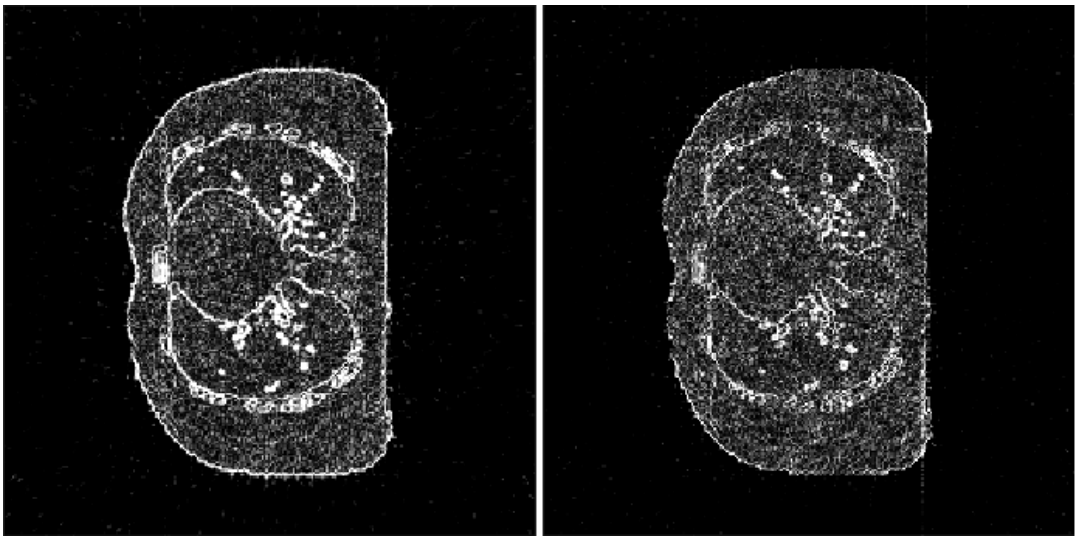


Figure 4-7: Difference between projection modes for different image resolutions. From top to bottom, the image resolution is 64^3, 128^3, 256^3 and 512^3 respectively. From left to right, (a) the absolute difference between the projections; (b) contrast enhanced version of (a) by cropping the colourmap to 25% of the maximum; (c) a zoomed in version of (a); and (d) a zoomed in version of (b).



(a) (b)

Figure 4-8: XCAT phantom reconstruction of size 256^3 using OS-SART with 200 iterations and 100 angularly uniformly sampled projections. (a) Reconstruction using ray-voxel intersection projection and (b) using the interpolated-projection method.



(a) (b)

Figure 4-9: Difference between the original XCAT phantom and the reconstructions of figure 4-8. The colour limits have been set to 10% of the maximum intensity of the original data. (a) Difference using ray-voxel intersection projection and (b) difference with the interpolated projection method.

4.4 The backprojection operator

The backprojection operator (or $A^T * b$ in algebraic notation) is the operator that updates the image using the information in the projection images. This update is performed by an operation often described in the literature as a smearing of the projection data into the image, as if it were butter on toast, but following the path from detector to source.

As with projection, multiple methods to perform this operation have been proposed in the literature. The most commonly used method is voxel-driven backprojection[**?**][**?**], where the path from the source to each voxel center is generated and extended until the detector is reached. Then the value in the detector is sampled (using interpolation, as it is likely that it does not fall in the center of a pixel) and the voxel value updated.

Other methods include the separable footprints method[**?**], where the footprint of the voxel in the detector is precomputed and approximated and the voxel values are updated according to the detector values overlapping the voxel footprint. A conceptually similar backprojection relies on having spherical shaped image value representation, instead of square voxels[**?**]. The backprojection using basis-functions also updates the image values according to the footprint of these spherical voxels.

The distance-driven method[**?**] is also applicable to backprojection by performing the same operation as for projection: the computation of all voxel-pixel intersections in an imaginary mid-plane. Finally, ray-voxel intersection driven methods also exist[**?**], in both single ray or multiple ray per voxel modes. This method requires multiple voxel updates per backprojection, but can give a matched result, i.e., the backprojection method is the same as the projection method. This has been shown to give better results[**?**] and allows the use of any iterative method Krylov subspace methods require matched backprojection.

In this work, voxel-driven backprojection has been implemented. The rationale being that it is a method which is fast and easy to implement yet accurate. Additionally, a quasi-matched backprojection can be implemented to allow Krylov subspace algorithms (see section 4.4.2 for more information). Finally, this method is most appropriate for the method proposed in Chapter 6.

4.4.1 Voxel-driven backprojection

The underlying idea of voxel-driven backprojection is simple, the path between the source and the center of each voxel is computed and the intersection of that path with the detector is computed. Then, the detector is sampled at the intersection point (using interpolation) and the voxel is updated with that value. Due to the cone angle,

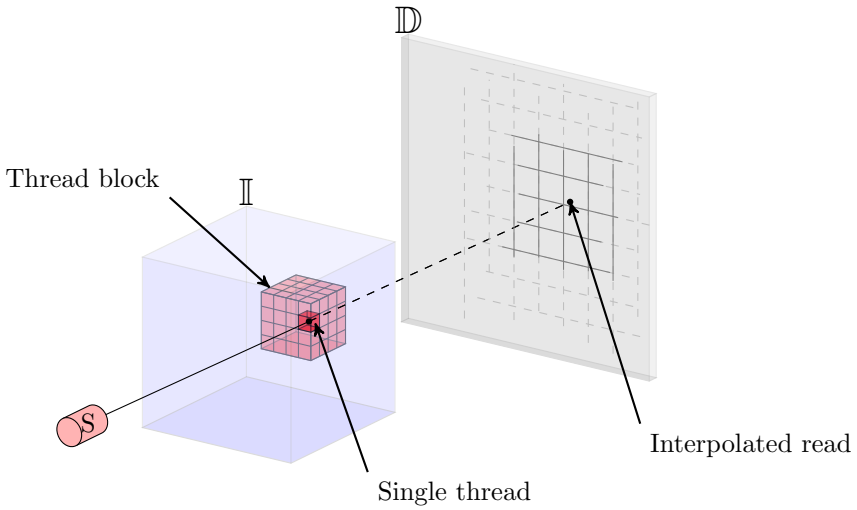


Figure 4-10: Simple voxel-driven backprojection. Each kernel is subdivided into square blocks and each thread updates a single voxel using interpolated memory reads in the detector.

a weighting factor is also applied to each voxel (more details are given in the next section).

To accelerate this operator in a GPU, the naive approach is to assign a thread per voxel and to assign a square amount of threads to a single block (by dividing it into divX×divY×divZ threads) to maximize cache hits on texture memory (used for the interpolation of the detector values). The block size is empirically set to 8x8x8 for fastest execution. This approach, shown as a diagran in figure 4-10 and as pseudocode in algorithm 5, does indeed result in a fast kernel. However, the backprojection requires a significantly higher total number of threads than the projection operation, while repeating the same thing for each voxel (in multiple bakcprojection updates) and reading in the same memory very often. In order to improve the cache memory hits and minimize redundant arithmetic computations, a series of improvements have been studied by Papenhausen *et al*[**?**] and further optimized by Zinsser *et al* [**?**].

The main idea behind the optimization is the minimization of memory latency. In order to do that a multiple voxel, multiple projection per thread kernel is designed. If in each of the threads when a single voxel is updated multiple projections (32 in this case) are used, these will be spatially nearby, thus the L2 cache will speed up the memory reading process provided the angular distance between projections is small. Additionally, if each of the threads computes a small subset of voxels ($N_{voxelThread}$) in the z direction (8 in this case), not only are the memory cache hits are likely to be

Algorithm 5 Naive voxel-driven backprojection

1: **for** Projection **do**
2: Precompute geometric constants
Require: N_{voxel} threads organized in divX×divY×divZ blocks
3: Compute $[u, v]$ sample position
4: Compute w weight
5: Image(x, y, z)+=$w \cdot$ Detector(u, v)
Ensure:
6: **end for**

increased, but also the computational operations reduced as the computation of the location of each voxel requires fewer operations. In general these tweaks increase the occupancy of the SMs, decreasing the amount of time the threads stand idle waiting for memory. The diagram of the new optimized kernel is shown in figure 4-11 and its pseudocode is given in algorithm 6. The code is divided in pieces to allow each block to have divX×divY threads each (16×32 in this case). To minimize global memory reads, the image voxel values that are updated in each kernel are pre-loaded. Then, for each projection, the geometric constants that describe the location of the detector and image pixels are loaded from constant memory. Next, the backprojection is performed for each voxel being updated. This approach further increases occupancy as in execution the thread does not wait for the memory read to finish before computing the next loop, thus hiding memory latency even more. Finally, the image is updated with the auxiliary variable. This step also decreases the memory latency, as fewer global memory write operations are needed.

4.4.2 Backprojection weights

Due to the cone shape the backprojection needs a weight for each voxel, as different paths have different ray lengths and hence have a different effect in the detector. In the projection operator, the length of the path is used in each detector pixel as a weight for the update, but in the backprojection operation the weight is not as straightforward to compute. In the algebraic definition of iterative algorithms, the weight of each voxel is the length of a specific ray within that voxel. If the matrices were fully known, it would be straightforward to compute (as it is the sum of the columns of A), however this doesn't apply to voxel-driven backprojection. In the GPU version, a projection to compute such lengths per backprojection would be needed, ultimately slowing the code considerably.

Algorithm 6 Optimized voxel-driven backprojection

for $N_{kernels}$ **do**
    Precompute geometric constants *per projection*
Require: divX×divY threads organized in $\frac{N_x}{\text{divX}} \times \frac{N_y}{\text{divY}} \times \frac{Nz}{N_{voxelThread}}$ blocks
    for $N_{voxelThread}$ **do**
        auxImage(voxThread)=Image(x, y, z)
    end for
    for Projections (in this kernel) **do**
        Load geometric constants for this projection
        for $N_{voxThread}$ **do**
            Compute $[u, v]$ sample position
            Compute w weight
            auxImage(voxThread)+=$w \cdot$ Detector(u, v)
        end for
    end for
    for $N_{voxThread}$ **do**
        Image(x, y, z)=auxImage(voxThread)
    end for
Ensure:
end for

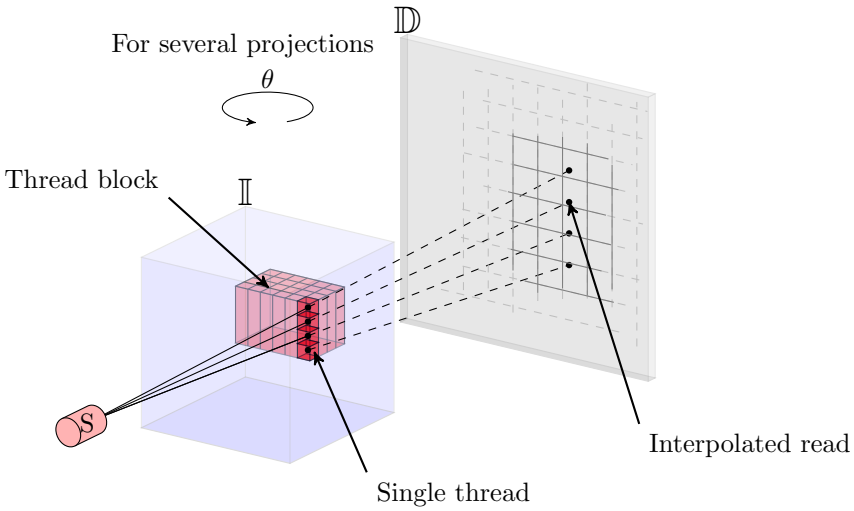


Figure 4-11: Optimized voxel-driven backprojection. Each kernel is subdivided into blocks and each thread updates a series of vertical voxels using interpolated memory. Each kernel additionally operates in a series of projections, not in a single one.

FDK weights

A simple approach is to use the weights of the FDK algorithm as backprojection weights. As described in section XX, the FDK backprojection weight is computed as

$$w_{x,y} = \frac{R^2}{(R + y \cdot \sin\theta - x \cdot \cos\theta)^2}, \tag{4.31}$$

where R is the distance from the source to the axis of rotation, θ the projection angle and x and y the location of the voxel. Note how this equation is independent of z, thus the weights can be precomputed after line 7 (instead of line 10) in algorithm 6, resulting in a minor speed-up.

Pseudo-matched weights

The FDK weights are good in the iterative algorithms that normalize the result of the backprojection afterwards (such as SART), as the overall scale and effect of the backprojections is removed in the algorithm by an opposing weighting factor. However, some algorithms require matched backprokection, i.e., a backprojection operator that is mathematically equivalent to the adjoint of the projection operator. This is not the case with FDK weights. Algorithms such as CGLS cannot work unless a matched backprojection is implemented. As previously explained, a fully matched backprojection, while possible, would slow down the kernels considerably due to the need of also computing projection operations in the backprojection. However Jia *et al*[7] propose a weight to match the backprojection.

In order to derive the weight, a functional analysis approach needs to be taken. For the sake of simplicity and cohesion with Jia *et al*, the notation in this section differs from the rest of the thesis.

If an image is represented as a function $f(\mathbf{x})$, where $\mathbf{x} = (x, y, z) \in \mathbb{R}^3$, a projection operator A^θ maps $f(\mathbf{x})$ onto a different function on the projection plane with angle θ as:

$$A^\theta[f](\mathbf{u}) = \int_0^{L(\mathbf{u})} \mathrm{d}l \; f(\mathbf{x}_s + \mathbf{n}l), \tag{4.32}$$

where $\mathbf{x}_s$ is the coordinate of the source, $\mathbf{n}$ a unit vector in the projection direction and $\mathbf{u} \in \mathbb{R}^2$ the coordinates in the detector. $L(\mathbf{u})$ is the length of the X-ray path.

Let $f(.) : \mathbb{R}^3 \to \mathbb{R}$ and $g(.) : \mathbb{R}^2 \to \mathbb{R}$ be smooth enough functions in image and projection domain respectively. In order to to have an operator A^{θ^T} that is the adjoint of A^θ, it should satisfy the condition

$$\langle f, A^{\theta^T} g \rangle = \langle A^\theta f, g \rangle, \tag{4.33}$$

where $\langle \cdot, \cdot \rangle$ is the inner product. In integral form, this inner product equality can be expressed as

$$\int \mathrm{d}\mathbf{x} f(\mathbf{x}) A^{\theta^T}[g](\mathbf{x}) = \int \mathrm{d}\mathbf{u} A^{\theta}[f](\mathbf{u}) g(\mathbf{u}), \tag{4.34}$$

or if the functional derivative of both sides with respect to $f(\mathbf{x})$ is taken, then as

$$A^{\theta^T}[g](\mathbf{x}) = \frac{\partial}{\partial f(\mathbf{x})} \int \mathrm{d}\mathbf{u} A^{\theta}[f](\mathbf{u}) g(\mathbf{u}). \tag{4.35}$$

Equation 4.35 can be rewritten as

$$A^{\theta^T}[g](\mathbf{x}) = \int \mathrm{d}\mathbf{u} g(\mathbf{u}) \frac{\partial}{\partial f(\mathbf{x})} A^{\theta}[f](\mathbf{u}), \tag{4.36}$$

and equation 4.32 can be rewritten as equation 4.37 using a delta function.

$$A^{\theta}[f](\mathbf{u}) = \int \mathrm{d}l \mathrm{d}\mathbf{x}\ f(\mathbf{x}) \delta(\mathbf{x} - \mathbf{x}_s - \mathbf{n}l). \tag{4.37}$$

Finally, by substituting equation 4.37 in 4.36, the adjoint of the projection operator can be expressed as

$$A^{\theta^T}[g](\mathbf{x}) = \int \mathrm{d}l \mathrm{d}\mathbf{u}\ g(\mathbf{u}) \delta(\mathbf{x} - \mathbf{x}_s - \mathbf{n}l) = \frac{L^3(\mathbf{u}^*)}{L_0 l^2(\mathbf{x})} g(\mathbf{u}^*), \tag{4.38}$$

where $\mathbf{u}^*$ is the intersection point between the x-ray path and the detector plane, $l(\mathbf{x})$ the distance between the source and a voxel, and L_0 the source to detector distance. This equation, however, applies to the integral form of the description of the system, while ultimately in the computer the matrix form is used. By changing the inner product to a vector form, the final adjoint over all projections becomes

$$A^{\theta^T}[g](\mathbf{x}) = \frac{\Delta x \Delta y \Delta z}{\Delta u \Delta v} \sum_{\theta} \frac{L^3(\mathbf{u}^*)}{L_0 l^2(\mathbf{x})} g^{\theta}(\mathbf{u}^*), \tag{4.39}$$

where $(\Delta x, \Delta y, \Delta z)$ are the sizes of each voxel in each direction and $(\Delta u, \Delta v)$ the sizes of the detector pixels.

This backprojector weight is very close to a matched backprojection operator. According to Jia *et al* the numerical errors are less than 1%. This work did not replicate their results. This mismatch can lead to inaccuracies in Hounsfield units in the final reconstruction and to divergent behaviour in the Krylov subspace algorithms, but only in late iterations.

4.4.3 Comments on optimization

To have a fast execution of the code, as for the projection, a geometry that minimizes the amount of arithmetic operations inside the kernels is proposed. The backprojection coordinate system (x_b, y_b, z_b) is defined to have unit sizes of the detector pixel size in u, v for y_b and z_b respectively, and 1mm in x_b. The origin of the system is located in the center of the first lexicographically ordered pixel in the detector and is always aligned to the detector (i.e., the image rotates while the detector-source system stays in the same location). All precomputing operations performed in the projection geometry are also performed in here.

In the CUDA sense, two extra optimizations have been performed. By defining divX, divY and $N_{voxelThread}$ (from algorithm 6) as compilers, an instruction to the CUDA compiler can be passed to unroll all the loops. Loop unrolling refers to replacing a for loop by a repetition of each line of code per iteration, one after the other. By doing this, the kernel does not need to have flow control (loop iteration, condition, variable) that needs increasing and checking, thus increasing the total kernel performance by 20% in our case. A small speed-up is also obtained by defining the texture memory as layered memory, thus disabling interpolation in the third dimension.

4.5 Benchmark

This section shows the computational times for the projection and backprojection kernels. This section does test the performance of the kernels themselves, but the actual calls to the kernels do have some overhead of memory input and output, as it is a significant amount of memory that needs to be moved in every call. All computational time results show time per projection, for different image and projection sizes. Figure 4-12 shows the projection times in milliseconds for both ray-voxel intersection and grid interpolated projection modes. The projection operation is dependent in both detector an image size, and it takes about the same time for both types of projection, with maximum of 100ms with 1024^2 detector and 1024^3 image.

Figure 4-13 shows the backprojection times for FDK weights and pseudo-matched weights. As the kernels are optimized for adjacent projection calls the computational times do not scale with more projections, a test using the maximum projections per kernel is also performed. Figure 4-14 shows kernel times per projection when multiple projections are updated. The maximum computational time on these tests (1024^2 detector with 1024^3 image) show 75ms and 180ms for the FDK and pseudo-matched weights when a single projection is used, but 45ms and 99ms per projection when 32 projections (maximum simultaneously used projections in a single kernel) are used.

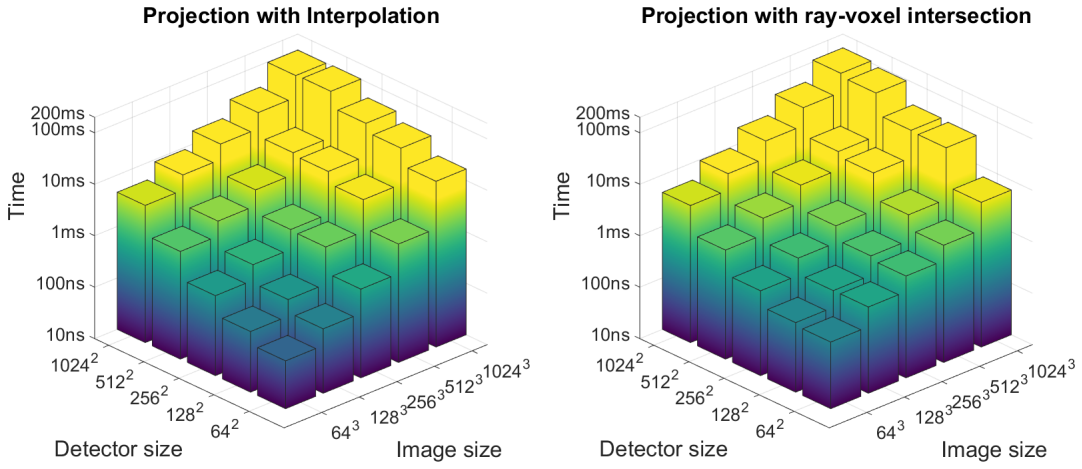
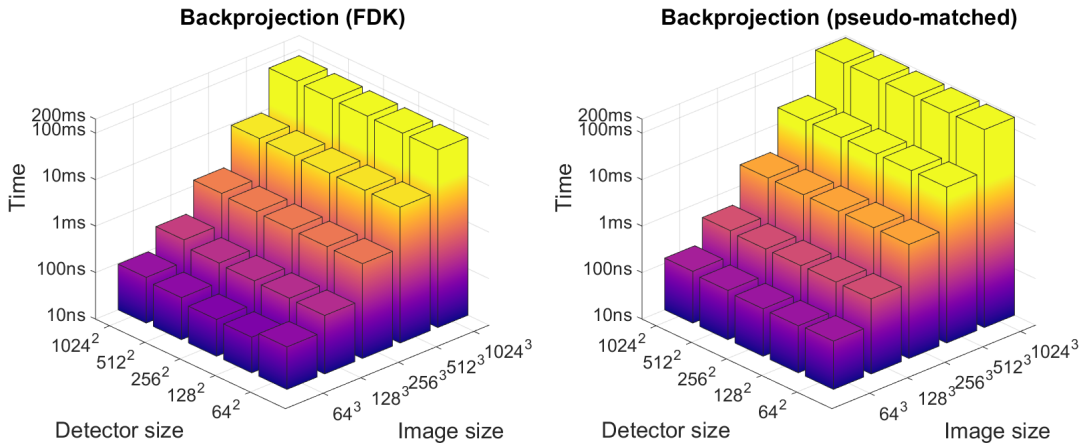


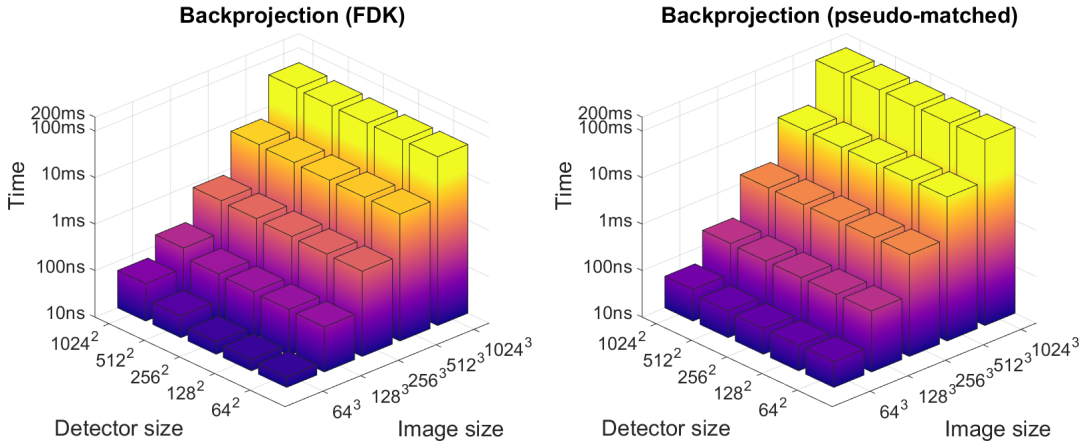
Figure 4-12: Computational times per projection for the projection operation for (a) grid interpolated (2 samples per voxel) and (b) ray-voxel intersection modes.

Note that generally the pseudo-matched weights are slower. This is explained by the fact that the chosen kernel structure completely masks the memory latency with FDK weights, but with matched weights the arithmetic operations for the weight need both more computations and registers, thus slowing down the kernel significantly. Additionally an extra multiplication kernel is needed for the final normalization from equation 4.39. Note also how the backprojection times are not dependant on the projection size, only in the image size. This is not a big surprise, considering how the kernels are designed.



(a) (b)

Figure 4-13: Computational times per projection for the backprojection operation when launched with a single projections for (a) FDK weights and (b) pseudo-matched weights.



(a) (b)

Figure 4-14: Computational times per projection for the backprojection operation when launched with 32 projections for (a) FDK weights and (b) pseudo-matched weights.

4.6 The TIGRE toolbox

In order to have an easy tool to implement algorithms but still have the GPU acceleration on hand, a MATLAB-CUDA toolbox has been created, the Tomographic Iterative GPU-based Reconstruction toolbox, or TIGRE toolbox. TIGRE is a modular, easy to use fast toolbox for cone and parallel beam computed tomography, focusing in the implementation of a variety of iterative reconstruction algorithms. All four families of algorithms described in chapter 3 are implemented in TIGRE, namely, FDK, statistical inversion (MLEM), the gradient descend family (SART,OS-SART,SIRT), Krylov subspace family (CGLS) and TV regularized family (ASD-POCS, OS-ASD-POCS, B-ASD-POCS-β, SART-TV). This section describes the features of TIGRE, the geometry supported, the general structure of the toolbox and how to implement an algorithm on it. This section is partially based in article [**?**].

4.6.1 Geometry in TIGRE

The geometry of CBCT in TIGRE can be represented as in figure 4-15. An X-ray source, S, is located at distance DSO from a centre of rotation O, where the origin of a cartesian coordinate system is located. The X-ray source irradiates a cone-shaped region containing the image volume $\mathbb{I}$ and a detector $\mathbb{D}$ measures the intensity of the photons attenuated following the Beer-Lamber law. The image is centred at position O$'$, which is displaced by $\overrightarrow{V_{orig}}$ from the coordinate system origin. The detector, located at distance DSD from the source and centred at D$'$, has an offset of $\overrightarrow{V_{det}}$ from D, which is a point lying in the xy-plane at distance DSD $-$ DSO from the origin. A projection coordinate system uv is defined centred at the lower left corner of the detector. During the measurement acquisition, the source and the detector rotate around the z-axis at an angle of θ from their initial position. Additionally, the detector can rotate on its own center by 3 axis of rotation, useful to account for mechanical errors[CITE]. Finally the center of rotation (COR) offset has also been implemented, a common offset in CT machines where the sample rotates, instead of the detector-source system.

While the diagram shows the geometry for CBCT, TIGRE also supports 3D parallel beam geometry, and by setting the offsets of the image accordingly, helical beam geometries. Additionally, if a correct size of the detector is chosen the geometry can be modified to allow 2D reconstruction, however TIGRE is not designed for 3D geometries, as GPU accelerations in 2D are not essential. Code snippet 4.1 shows the code to define the geometry in TIGRE.

The geometric variables described above are used in the TIGRE Toolbox to perform the necessary operations for image reconstruction, as shown in code snippet 4.1. It is

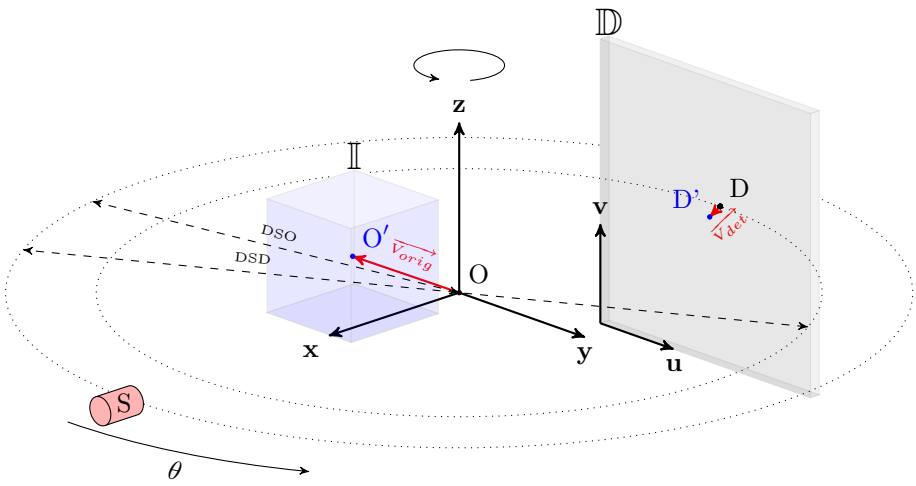


Figure 4-15: Diagram of the geometric definition of a TIGRE reconstruction. Image, and detector offsets are supported, as well as any arbitrary size for the image and the detector, both total and pixel-wise.

worth mentioning that both $\overrightarrow{V_{det}}$, $\overrightarrow{V_{orig}}$, COR and the rotation of the detector are vectors that can be defined per projection angle θ.

Code Snippet 4.1: Geometry definition in TIGRE

```
%% Geometry structure definition.
% Distances
geo.DSD = 1536;                                 % Distance Source Detector
geo.DSO = 1000;                                 % Distance Source Origin
% Detector parameters
geo.nDetector=[512; 512];                       % number of pixels
geo.dDetector=[0.8; 0.8];                       % size in mm of each pixel
geo.sDetector=geo.nDetector.*geo.dDetector;     % total size of the detector in mm
geo.rotDetector=[0;0;0];                        % euler angles of the rotation
% Image parameters
geo.nVoxel=[512;512;512];                       % number of voxels in the image
geo.sVoxel=[256;256;256];                       % total size of the image in mm
geo.dVoxel=geo.sVoxel./geo.nVoxel;              % size in mm of each voxel
% Offsets
geo.offOrigin   =[0; 0; 0];                     % V_orig
geo.offDetector=[0; 0];                         % V_det
geo.COR=0;                                      % Centre of Rotation offset
```

4.6.2 Structure

TIGRE has been designed to be modular, in order to facilitate prototyping with instant acceleration and to allow easy of the toolbox. The main building blocks are the projection ($A(x)$) and back projection ($A^T(b)$) operators. In the TIGRE Toolbox,

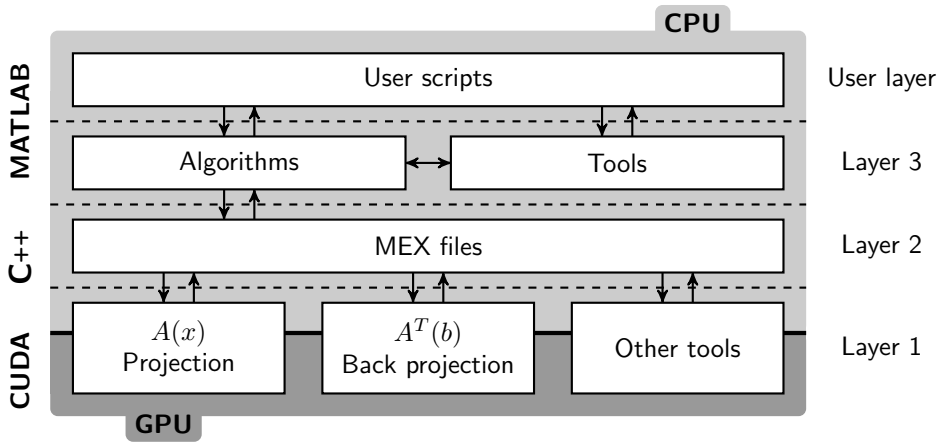


Figure 4-16: Diagram of the structure on TIGRE toolbox.

these two blocks have been optimized for GPU computing using CUDA, as described in the beginning of this chapter. They lie in the lowest layer of the toolbox design and are constantly used by the other layers. The algorithms themselves lie in the topmost layer and are all coded in MATLAB, which provides the power and flexibility of a high-level language. To be able to communicate between the low-level, hardware-oriented CUDA and the high-level, design-oriented MATLAB, a set of the so-called *MEX functions* are needed. The toolbox has been designed not to have any specific data types or classes. Instead, it comprises only the basic MATLAB types, such as matrices and structures.

The high level algorithms are designed to have multiple parameters and full customization. The only required parameters to all algorithms are the data, geometry, angles and number of iterations. Each algorithm has a series of tunable parameters. Generally every parameter affecting the algorithm can be set up to have a different value, allowing users that want to study algorithm behaviour full customization, but having default parameters in case the users want an easy-to use algorithm. Multiple algorithm initialization modes, angle ordering schemes and other features are also available.

Using TIGRE

This code demonstrates the reconstruction of the RANDO head phantom (data obtained in the Christie Hospital, Manchester, UK) using three different algorithms with the geometry defined in code snippet 4.1. The data set contains 360 equidistant projections. Once the data have been loaded using the code of snippet 4.2, the results of figure 4-17 can be obtained without the need for any more code. Information about

total computation time and computation time per iteration are shown. Only some of the possible optional parameters to the algorithms are shown in the snippet. The reader is referred to the published documentation for advanced options and for insight into their numerical ranges.

Code Snippet 4.2: RANDO head data reconstruction

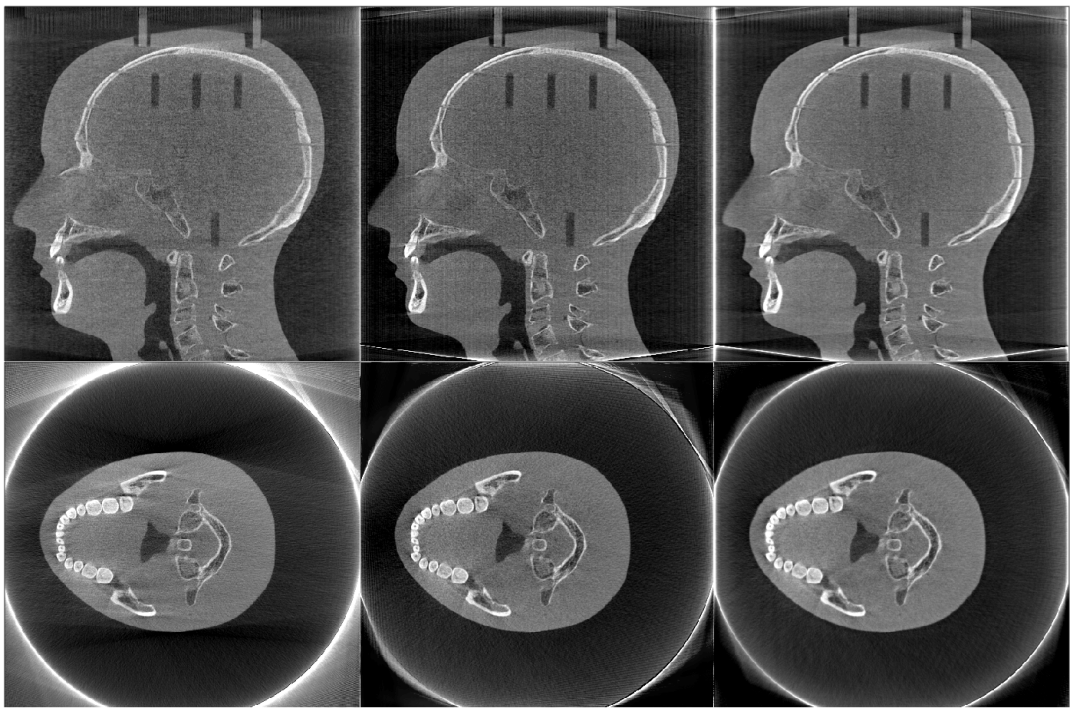
```
% Define Geometry & load data

% From the data, the projection angles (in radians) must have been read
angles= 0:359*pi/180; % as an example

%% Reconstruct image with different algorithms
% FDK
imgFDK=FDK(data,geo,angles);
% CGLS
iterCGLS=15;
imgCGLS=CGLS(data,geo,angles,iterCGLS);
% OS-SART with multi-grid initialization
iterOSSART=70;
imgOSSART=OS_SART(data,geo,angles,iterOSSART,'BlockSize',20,'Init','multigrid');
```

Implementation of an algorithm in TIGRE

To demonstrate the facility with which anyone can develop new algorithms using the TIGRE toolbox, this section presents a side by side comparison of an algorithm definition and its TIGRE equivalent code, using the GPU accelerated features. For the sake of brevity, the CGLS algorithm has been chosen. In table 4.1 the definition of the CGLS iterations and the implementation in TIGRE are shown. From the code snippet, it is worth highlighting the limited use of library related functions, as one of the strengths of TIGRE for the developer point of view is the easy to use Application Programming Interface (API). The only difference in the code from a completely standard MATLAB script is the use of the function Ax() and Atb(), the main building blocks of the toolbox. This allows anyone with MATLAB code for solving image reconstruction to easily modify their code by just changing the matrix-vector operations by TIGRE GPU functions. Note that the functions inside TIGRE do generally have more code than the one shown here, as several options and performance enhancing MATLAB tools are used.



(a)	(b)	(c)
FDK	OS-SART	CGLS
14s total	20m59s total	6m8s total
	31s per iteration	18s per iteration

Figure 4-17: RANDO head phantom reconstructed with (a) FDK, (b) OS-SART 40 iterations and (c) CGLS 20 iterations, for 360 equidistant projections. The computational times are also shown.

Table 4.1: CGLS algorithm as definition, and implemented in TIGRE

$x_0 = 0;\ d_0 = b;\ r_0 = A^T b; p_0 = r_0;$

$t_0 = Ar_0;\ \gamma_{k-1} = \|r_0\|^2;$

for $k = 1$ **to** $k =$ maxiter

$\alpha_k = \gamma_{k-1} / \|t_{k-1}\|^2$

$x_k = x_{k-1} + \alpha_k t_{k-1}$

$d_k = d_{k-1} - \alpha_k t_{k-1}$

$r_k = A^T d_k$

$\gamma_k = \|r_k\|^2$

$\beta_k = \gamma_k / \gamma_{k-1}$

$p_k = r_k + \beta_k p_{k-1}$

$t_k = Ap_k$

end

```
% Initialize variables
x=zeros(geo.nVoxel');
d=b;
r=Atb(b,geo,angles,'matched'); %TIGRE
p=r;
t=Ax(r,geo,angles); %TIGRE
gamma_1=norm(r(:));

% Loop until user defined maxiter
for k=1:maxiter
    alpha=gamma_1/norm(t(:));
    x=x+alpha*t;
    d=d-alpha*t;
    r=Atb(d,geo,angles,'matched');%TIGRE
    gamma=norm(r(:));
    beta=gamma/gamma_1;
    gamma_1=gamma;
    p=r+beta*p;
    t=Ax(p,geo,angles); %TIGRE
end
% x is the solution.
```

4.7 Summary

In this chapter we have presented a MATLAB/CUDA toolbox for fast 3D X-ray image reconstruction. While the toolbox has reasonably good performance – reducing to minutes an image reconstruction with complex iterative algorithms – and a wide variety of tools, improvements are possible.

The projection and back projection operators have been fully implemented in the GPU, but the algorithms are fully in CPU so a memory management overhead exists because the data need to be introduced and extracted from the GPU twice per iteration. This design has been proposed in order to have the algorithms in a high-level language, as an algorithm implementation cycle in a low-level language like C++ is significantly longer than in MATLAB. As an estimate, if the algorithms were written in C++/CUDA directly, an improvement in computation time of up to 50% could be achieved in some cases. However, this would increase the difficulty of adding new algorithms to the toolbox. The final decision was that the advantages of a high-level programming language for new algorithms are better than the possible benefits of dou-

bling the speed, which is already reasonably good.

Further improvements in the core GPU kernels of the toolbox would also be possible. While the speeds reached by this method are arguably state of the art, some improvements to the kernel structure to decrease even more memory latency and computational times have been proposed in the literature. It is impossible to know with certainty that any of the methods published will increase the speed of the code, partly because GPU architecture has changed over the past year massively, thus code may be faster in specific GPUs but slower in others, and partly because most of the published papers do not contain code to be benchmarked against, and use different geometric parameters each. Thompson and Lionheart[**?**] propose a method that exploits structure similarities and works if the cone angle is smaller than 45 degrees, that is the case of most commercial CBCT machines. The work in this thesis tries not to impose limitations on the possible geometries, but modifying the code to trigger the accelerated version proposed by Thompson and Lionheart is an interesting possibility. Chou *et al*[**?**] claim speed-up by re-structuring the kernels to lauch multiple treads for multiple rays in parallel. While initial trials for replicating the multithreads in the early stages of the work in this thesis failed (as times where not changes) the projection operator has intensively been modified since then. This work has potential to accelerate up to 600% the projection operation according to the article. Finally, the method proposed by Gao[**?**] claims significant speed-up over the "naive" Siddon's method, the code provided with their article shows slower execution for both the "naive" and accelerated versions than the code used in this work. Nevertheless, it would be worth implementing their approach.

The backprojection operator has been more optimized than the projection operation, using better techniques, thus speed-wise it is performing well. However, there are certainly faster methods, as TIGRE ranks between 5 and 7 in the RabbitCT[4] benchmark. The RabbitCT benchmark is also recorded in different machines, thus some of the faster methods are due to multi-GPU parallelism exploits or just simply due to faster GPUs. That said, there is certainly room for improvement. Possibly the biggest improvement to the backprojection would be the implementation of completely matched projection adjoint code, as it leads to better recosntruction[**?**]. Thompson and Lionheart[**?**] propose a technique, so does Gao[**?**].

Comparing the forward and back projection speeds to the ASTRA toolbox[**?**], TIGRE is 2 times slower at its worst. This can be easily explained by two factors. Firstly, the geometric options for CBCT are more flexible in TIGRE than in ASTRA, thus requiring more floating-point operations. Secondly, ASTRA implements an advanced ray

[4]https://www5.cs.fau.de/research/projects/rabbitct

splitting that increases memory latency in the GPU and that makes use of overlaps between X-ray paths at different angles[**?**]. Adding all the discussed effects that would decrease the time performance, all algorithms run about 3 times more slowly in TIGRE than in ASTRA, which constitutes the state of the art. Numerically, the differences between ASTRA and TIGRE are in absolute value of the order of 10^{-3}, which is about 0.01% in relative terms. This difference can be attributed to accumulated floating point errors due to different numerical approaches in the GPU code.

To speed up further the toolbox, a multi-GPU approach could also be taken. Currently, TIGRE does not support multi-GPU architectures. A further weakness of the toolbox is the small number of functions for data loading and post-processing. However, work will be continued, hopefully filling this gap in the near future. The single GPU limitation of TIGRE also limits the image size. Currently, 12GB is the maximum amount of memory on a GPU board, thus limiting the possible size of the images that can be reconstructed. Nevertheless, there is no problem to reconstruct a 1024^3 image with most algorithms so the maximum image size is still big.

The TIGRE Toolbox has been designed with the objective of reducing the gap between image reconstruction research and the end users of tomographic images. While research in reconstruction creates new algorithms every year, end users only have access to FDK implementations. With these two groups in mind, the toolbox:

- has easy-to-use "black box" algorithms, making it extremely straightforward for researchers who are only interested in the quality of the images to test different algorithms without them requiring any knowledge of how the algorithms work;
- has easy-to-use building blocks (projection and back projection operators) that allow algorithm developers to test new methods using a high-level programming language but with the performance of the lowest level, GPU languages.

The code is released as open source under a BSD 3-clause license, allowing anyone to download, test, modify and improve it. While the toolbox was originally designed for CBCT image reconstruction, an option for 3D parallel-beam CT reconstruction has also been included allowing for more geometries, e.g., synchrotron data. Further tweaking the geometry structure of the toolbox also permits 2D fan- and parallel-beam reconstructions.

The minimum requirements to run the toolbox are strongly dependent on the image size desired, as memory is the strongest limiting factor both on the CPU and GPU side. Generally speaking, any NVIDIA GPU with a compute capability higher than 3.5 would be sufficient to reconstruct arbitrarily large images. We recommend having at least 3 times the desired image size in GPU memory and 8 times in RAM in the computer.

As an example, for a 512^3 image, 2GB of GPU memory and 6GB of computer RAM is the suggested minimum. The computing power (number of processors in the GPU and processor performance of the CPU) will have a strong effect on the speed of image reconstruction.

Chapter 5

Experiments and Applications

In the previous two chapters the mathematical and computational challenges of image reconstruction for CT have been discussed. In chapter 3, a detailed description of a variety of different algorithms has been presented, including the ART family of algorithms, CGLS, MLEM and few TV approaches for smooth reconstruction, as well as the classic FDK reconstruction. Additionally in chapter 4 the computational aspect of CT is discussed, on where the problems computing the exact adjoint of the projection operation and mainly the computational burden of some of the operations have been mentioned. Considering the variety of available methods and the specifics of the implementation of the software developed, the TIGRE toolbox, experiments on how these algorithms compare and behave are due. Further than that, the performance of these algorithm in different experimental datasets is also an important analysis.

This chapter shows experimental analysis on both of the topics. First a variety of convergence analysis with different algorithms using synthetic data are performed, showing the differences not only between algorithms, but also between option on parameter selections. The section tries to illustrate and perhaps help build intuition on all the different parameters and options that each of these algorithms has, both within the algorithms themselves and among the different ones. Additionally some highlights on the practical challenges that the use of the algorithms entail in real applications are given.

In the second section of this chapter, a few examples of some of the algorithms are shown in different CT applications, both cone and parallel beam. Data from various different applications, from medicine to science has been tested using the TIGRE toolbox. While quantitative analysis is not possible with these datasets because the truth is not known, some insight in how the algorithms behave in each case is discussed.

5.1 Algorithm Experiments

This section experiments with a variety of algorithms and parameter within then, and shows how they behave with different synthetic data in simulation studies.

5.1.1 Convergence rates

On chapter 3 the convergence rates of the algorithms has been mentioned, as well as computational times. Different algorithms will reach to different residuals at the same iteration, and thus understanding which ones can converge faster, thus theoretically give a better result earlier, is important. However, at the scale of the CBCT problem, faster no only means reaching a residual that is smaller in the same iterations, as the computational burden of each of the iterations also needs to be considered. And, as the backprojection operator is not exactly the adjoint of the projection operator, an effect that the classic formulation of these algorithms do not take into account can happen: divergence. All the algorithms (at least in this work) are mathematically designed to always reduce the residual each iteration, but that formulation relies on a correct adjoint operator. Thus, sometimes, when the algorithms in TIGRE have a solution very close to minimum residual solution, they may diverge. The code in the toolbox does generally check for divergence and stop, but one of the effect that can be observed is that some algorithm will always diverge in a residual that is larger than others. This means that some algorithm can, regardless of their computational times, reach to a better solution than others.

All test in this section are performed in the XCAT phantom[16], in a 128^3 voxel size and 256^2 detector. Different number of angles are used, always uniformly distributed along the full circle. Figure 5-1 shows cross sections of the phantom on its middle plane and figure 5-2 shows 3 projections of the phantom as simulated for the following tests.

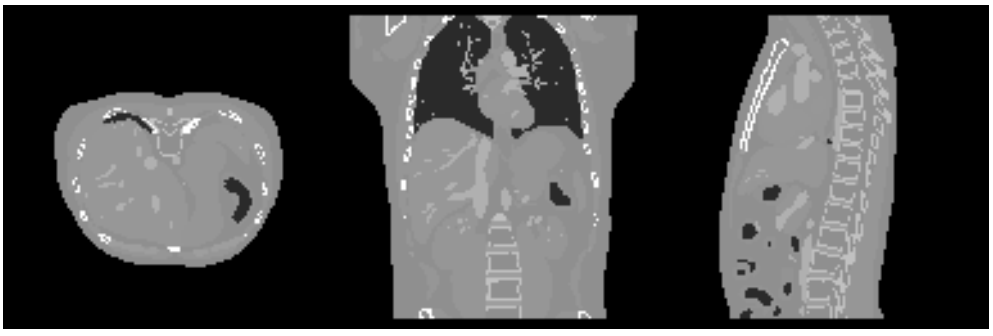


Figure 5-1: Cross section of the XCAT phantom in its middle plane in the three axis, for 128^3 voxels.

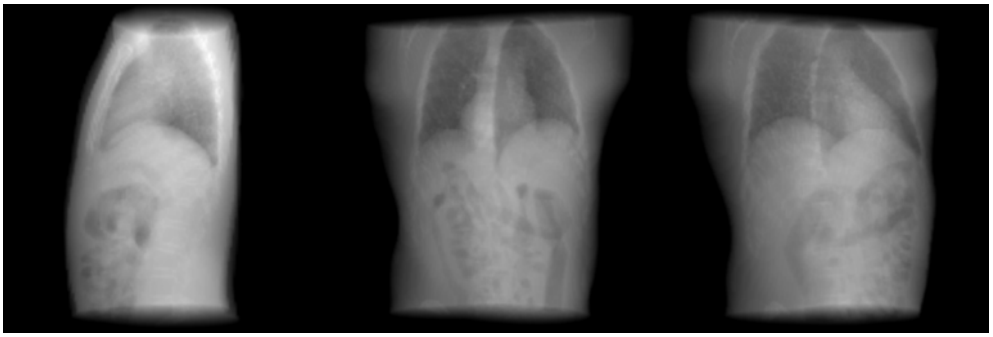


Figure 5-2: Simulate projections of the XCAT phantom in three different projection angles for a 256^2 detector.

Update ordering in SART. An analysis in the different ART-type algorithms is presented in this section first. One of the discussed parameters that has an effect in the convergence rate of the ART-type algorithms is the ordering of the used projections. Research has shown that in ART, the decrease of the residual with respect to how the rows are chosen for the updates is significantly changed[CITE], however in the algorithms feasible for big scale tomography, this effect is seen in a smaller scale. Figure 5-3 shows the convergence of SART during 150 iterations using 100 projections as data. The same configuration of SART is run using ordered, randomly ordered and angular distance maximizing ordering schemes for the update order. While minor, the figure shows how random ordering does generally increases the convergence rate of the algorithm, at no computational cost. This is the default value in the software. Note that in this test there is no reduction of the relaxation parameter λ.

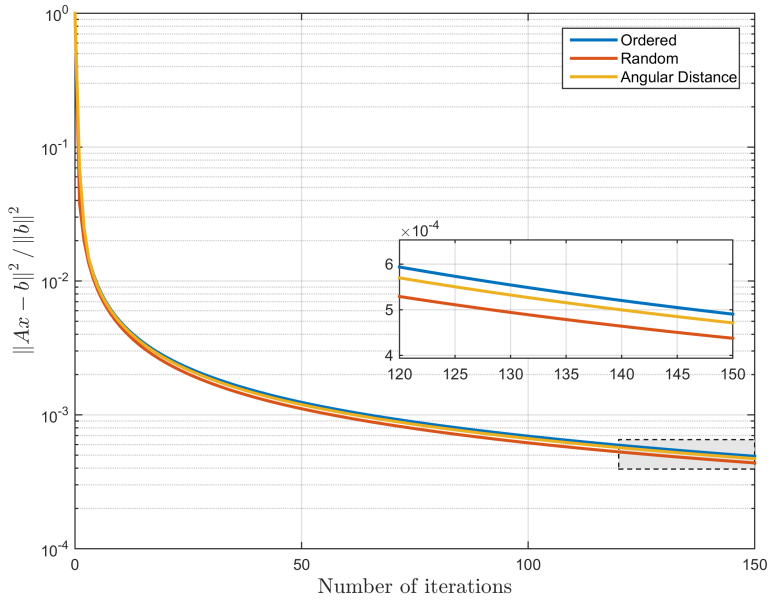


Figure 5-3: Normalized residual versus iteration of SART compared to different angle ordering schemes, using 100 projections and no relaxation parameter reduction

Comparison between SART, OS-SART and SIRT. These algorithms have very different convergence, as updating the image per-equation has the effect of converging faster. However, the computational times are greatly reduced by updating more rows at the same time. This effect can be seen in figure 5-4, on where the convergence versus iteration of these three algorithms is plotted. Note the convergence difference between SART and SIRT, where SIRT doesn't reach SART's residual even after 1000 iterations, however, each iteration of SIRT is two orders of magnitude faster than SART. OS-SART provides a middle ground alternative. Due to the specifics of the acceleration procedures for backprojection, OS-SART speeds are closer to SIRT than to SART (i.e. the speed does not change linearly with the image updates per iteration), however it is more prone to divergent behaviour in TIGRE. In the figure, OS-SART stops converging after 48 iterations. Of course, this behaviour is very data-specific, and there are multiple cases where it does not diverge. Figure XX shows the result images of these three algorithms after 150 iterations (48 for OS-SART). Note that the example being show is low data, so even in the best case, the images are slightly noisy.

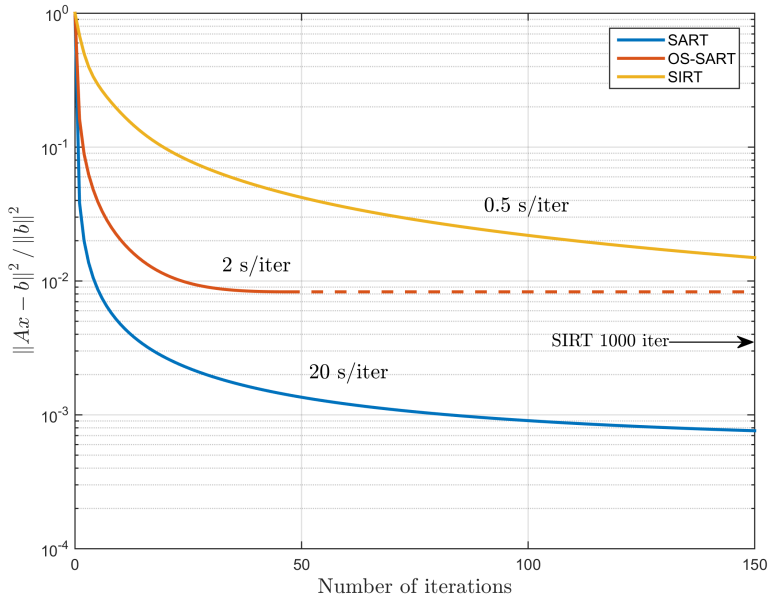


Figure 5-4: Normalized residual vs iteration number for SART, OS-SART and SIRT using 100 projections and no relaxation parameter reduction.

Relaxation parameter. The choice of a proper relaxation parameter does significantly change the speed on which a solution is found, and can avoid infinitely iterating through the same hyperplanes in case of an under determined or noisy solution. In TIGRE, two methods are present, as described in chapter 2: multiplying the relaxation parameter by a reduction factor after each iteration, and the Nesterov accelerated update, that does not technically update the relaxation parameter, but updates the image in each iteration using a iteration specific combination ration of the gradients of the current and previous iteration. It requires more memory, as it needs one extra image-sized variable to store the previous update, but the the algorithm finds a solution considerably faster, as it can be seen in figure 5-5. In the figure each of the SART, OS-SART and SIRT algorithms residual is plotted, and in each of them three versions are displayed, no relaxation parameter update, reduction with $r_{red} = 0.99$ and Nesterov update. In the plot it can also be seen that reducing the relaxation parameter by a ratio, while a good approach in SART-based hybrid algorithms, such as the TV minimizing ones in TIGRE, leads with slower residual reduction and ultimately with a worse image.

In figure 5-6, the solution found by the three algorithms using reduction of the relaxation parameter, using a Nesterov update and using static relaxation parameter of

$\lambda = 1$ can be seen side to side . The superior solution found by Nesterov is visually clear, and both SART and OS-SART reach a minimum in very few iterations. While SART does reach a better image (both in residual and error) without using Nesterov's update, the difference is minimal. It is important to note that using Nesterov's update, likely due to its fast convergence, leads to a faster divergent behaviour by the algorithms, thus the residual needs to be checked in each iteration leading to some computational overheat.

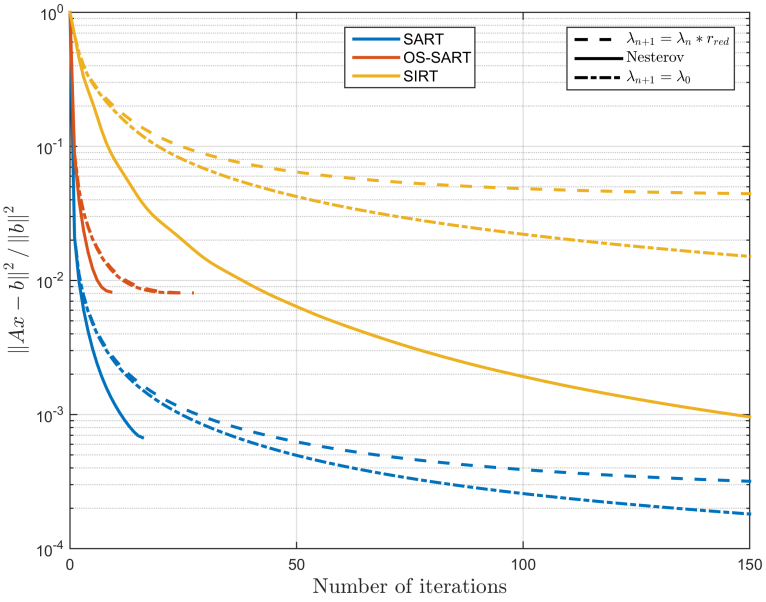


Figure 5-5: Normalized residual vs iteration number for SART, OS-SART and SIRT using 100 projections and different relaxation parameter reduction modes. If the relaxation parameter is reduced by a constant ratio the residual reduction worsens, and if reduced using Nesterovs update, it converges very fast.

5.1.2 Total variation minimization

There are 4 total variation minimizing algorithms in TIGRE, with 2 different minimization functional in total. As previously described, ASD-POCS, OS-ASD-POCS and b-ASD-POCS-β minimize the TV using a POCS minimization technique by minimizing the data constrain and TV norm independently using gradient descend. SART-TV however uses the ROF model for the TV-minimization step. The total variation algorithms are designed for applications on where the image is piecewise-smooth, as they will try to minimize the gradient on the image, by creating single valued patches in

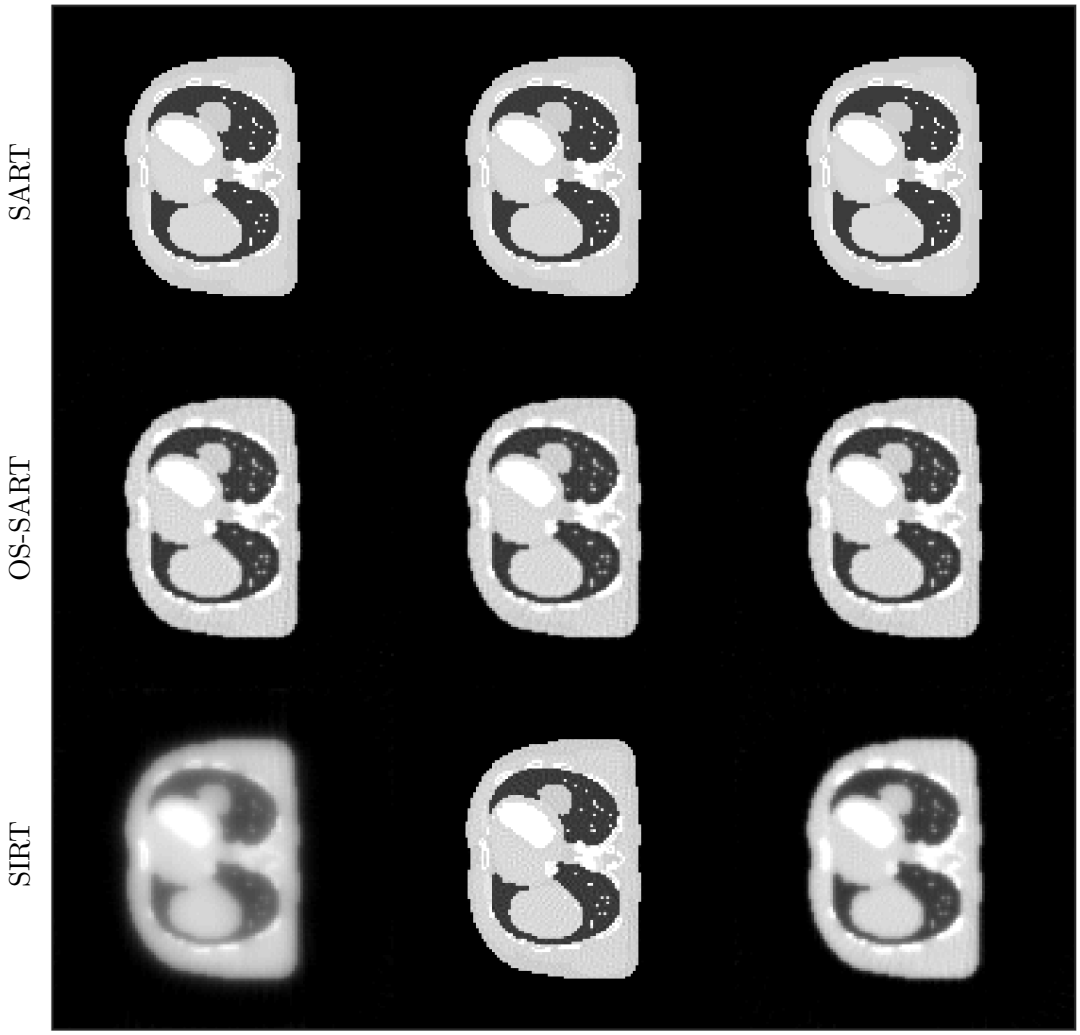


Figure 5-6: Reconstructed images with different relaxation parameter updates. OS-SART stops at iteration 24 in all but in the Nesterov case, where it stops in the iteration number 9. SART stops at iteration number 16 for Nesterov.

Table 5.1: NRMSE for the reconstructed images in figure 5-7

	FDK	OS-SART	b-ASD-POCS-β	SART-TV	ASD-POCS	OS-ASD-POCS
NRMSE	0.1373	0.0678	0.0338	0.0267	0.0304	0.0442

the image. In CT, the most noisy images are reconstructed when either the data is very noisy (generally due to small acquisition times and/or low energy X-rays) or when the data is limited, either from limited angle or more importantly limited amount of projections.

An example of the behaviour of the TV algorithms with the same dataset as in figures 5-1 and 5-2 is shown in figure 5-7. In this case, 30 uniformly sampled projections are used, perturbed with Poisson and Gaussian noise, to simulate photon scattering and electronic noise respectively. The figure shows FDK and OS-SART reconstructions, and the 4 mentioned TV algorithms. It is visually clear that the TV algorithms do provide a smoother reconstruction, and with less Normalized root mean squared error (NRMSE), as seen in table 5.1. The reconstruction by FDk is plagued with noise, and while the main structural features can be seen, most of the detail is lost. Even the bones themselves are practically indistinguishable from noise. OS-SART does reconstruct a smoother image, something predictable as it minimizes de 2-norm, and while one can see the more details in the image, it is still low. The rest 4 TV algorithms can be seen to flatten out the attenuation levels to similar values, thus reducing most of the noise. Additionally most of the features get clearly separated from the attenuation levels of the surrounding tissues and some of the algorithms (such as ASD-POCS) are able to reconstruct even single wide pixel structures correctly. It is important to note that while the parameters used to tune this specific TV reconstruction (available in the demo number 9 in the TIGRE toolbox), they are far from optimal and very sensitive[12]. When choosing the exact optimal parameters for the TV reconstruction algorithms, the result images tend to be significantly better than the ones shown here, but the parameter space is very data dependant and massive, thus to the author's knowledge, no parameter selection method has been proposed in the literature.

To illustrate the sensitivity to parameter selection, the algorithm SART-TV is run with 3 different values for number of TV-iterations per SART iteration for the same data set used in the previous test. The results can be seen in figure 5-8, where one can clearly see how a small changes can have a devastating effect in the output image. If a few more TV iterations are added to (b), the image gets a bit smoother and some detail is lost, as expected. However, if few iterations are removed from (b), the image can get completely destroyed. Note that these values are only applicable to this image

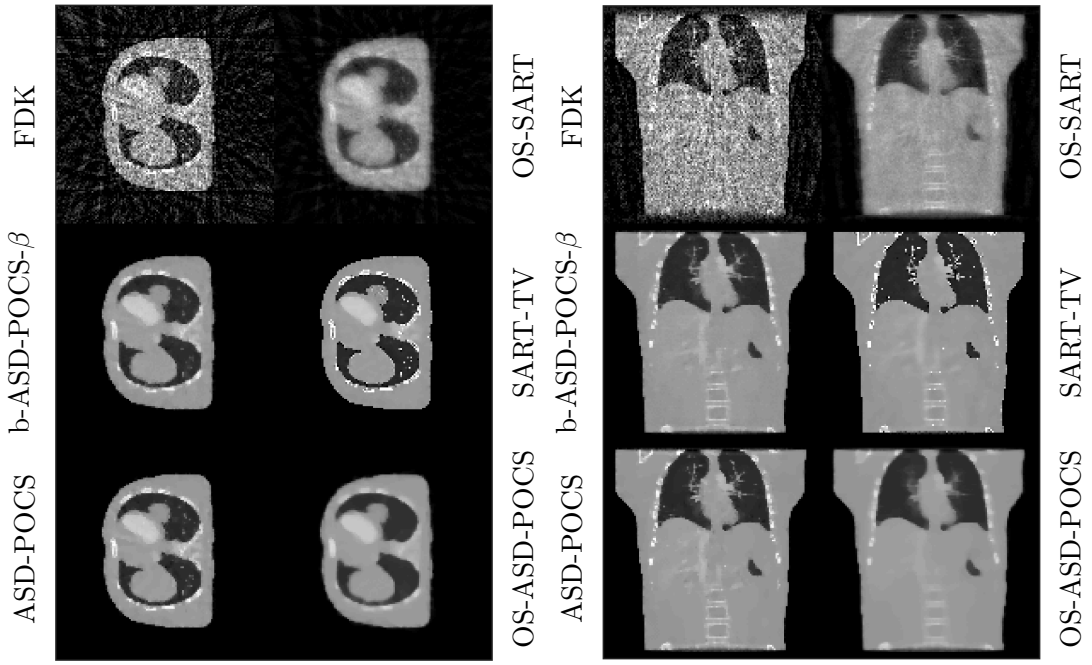
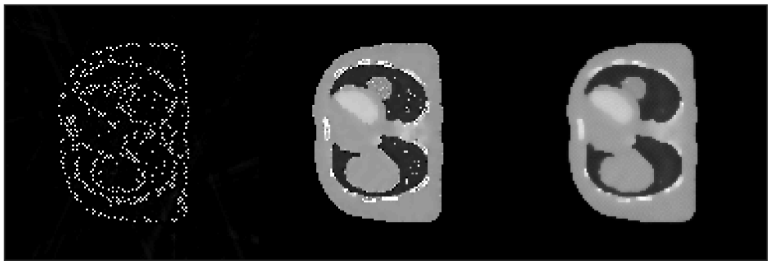


Figure 5-7: Reconstructed images using FDK, OS-SART and the TV algorithms b-ASD-POCS-β, SART-TV, ASD-POCS and OS-ASD-POCS with a limited amount and noisy data. Both figures show the same data and algorithms, but with a different cross-section of the image.



(a) (b) (c)

Figure 5-8: SART-TV algorithms with different amount of TV iterations per SART iteration, (a) 32 iterations, (b) 40 iterations, (c) 48 iterations.

with the exact amount of noise and projections. Different experiments may not show this behaviour or may be more intolerant to parameter change. This is arguably the biggest limitation for the common use of TV algorithms in real application. As an advantageous point, once the good parameters are found, generally the algorithm will perform similarly in similar images, thus application specific parameters may be an option.

5.2 Iterative Algorithms in Different CT Applications

This section tries to illustrate the effect of different algorithms within the TIGRE toolbox in a series of datasets.

5.2.1 Medical Head CBCT from The Christie Hospital

5.2.2 Cryo Soft X-Ray Tomography at Diamond Light Source

Cryo soft X-ray tomography (Cryo-SXT) is a relatively new[CITE] technology to image micrometer size biological samples in full 3D. Generally, cell-imaging is performed with electron microscopy (EM) machines and all its variants (transmission electron microscopy, scanning electron microscopy, cryo-electron microscopy, electron tomography, etcetera), however these techniques have very limited penetration (less than 1μm) and thus often require slicing of the samples for volumetric imaging. Cryo-SXT uses the so called water window for X-ray energies around the 500eV energy range. Unlike in higher energies, where everything is invisible, water becomes transparent but carbon-based tissues are clearly visible on that range. Thus, while with lower resolution than most EM, cryooSXT allows for full volumetric visualization of the cells without damaging the samples. In order to be able to image in the extremely accurate setup in

both sample handling and X-ray parameters, these cryoSXT images are captured in synchrotron facilities. The data used in this work is from the B24 beam-line at the Diamond Light Source.

However, cryoSXT data has several sources of errors that make its reconstructed images significantly noisy. The typical penetration depth of soft X-rays is around 10μm, while the samples are generally an order of magnitude bigger than that in height and width. Thus cryoSXT is a limited angle problem, where most of the datasets are sampled over a 120 degrees arc. In the extrema of this range, the images in the detector tend to have little or none information in some parts of the image due to photons not reaching the detector. Additionally, the low intensity and small sample size do mean that there the detector data is very noisy, as photons spread out more (in pixel dimension) and less photons reach the detector. The size of the sample also comes with errors in the mechanical systems of the imaging set up. When working in a scale of microns, any small vibration is visible and considerably offsets/perturbs the measured data. Generally this type of errors are removed by pre-processing using alignment techniques, but the algorithms involved are often not fault proof, and the data used in reconstruction ends up having some misalignment errors. Few of these errors can be seen in the sinogram of the dataset labelled as "2017_0207_Trypanosoma_33", on where misalignment and missing data problems are visually perceptible.

Chapter 6

Quality analysis of motion correction

Chapter 7

Conclusions and Future work

Bibliography

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

[2] Y. Censor and T. Elfving. Block-iterative algorithms with diagonally scaled oblique projections for the linear feasibility problem. *SIAM Journal on Matrix Analysis and Applications*, 24(1):40–58, 2002.

[3] Antonin Chambolle. An algorithm for total variation minimization and applications. *Journal of Mathematical imaging and vision*, 20(1):89–97, 2004.

[4] Patrick L Combettes and Valérie R Wajs. Signal recovery by proximal forward-backward splitting. *Multiscale Modeling & Simulation*, 4(4):1168–1200, 2005.

[5] Joan Duran, Bartomeu Coll, and Catalina Sbert. Chambolle's projection algorithm for total variation denoising. *Image processing on Line*, 2013:311–331, 2013.

[6] G. T. Herman and L. B. Meyer. Algebraic reconstruction techniques can be made computationally efficient [positron emission tomography application]. *IEEE Transactions on Medical Imaging*, 12(3):600–609, Sep 1993.

[7] Xun Jia, Yifei Lou, John Lewis, Ruijiang Li, Xuejun Gu, Chunhua Men, William Y Song, and Steve B Jiang. Gpu-based fast low-dose cone beam ct reconstruction via total variation. *Journal of X-ray science and technology*, 19(2):139–154, 2011.

[8] S. Kaczmarz. Angenäherte Auflösung von Systemen linearer Gleichungen. *Bulletin International de l'Académie Polonaise des Sciences et des Lettres*, 35:355–357, 1937.

[9] Avinash C. Kak, Malcolm Slaney, IEEE Engineering in Medicine, and Biology Society. *Principles of computerized tomographic imaging*. IEEE Press, New York, 1988. Published under the sponsorship of the IEEE Engineering in Medicine and Biology Society.

[10] Florian Knoll, Markus Unger, Clemens Diwoky, Christian Clason, Thomas Pock, and Rudolf Stollberger. Fast reduction of undersampling artifacts in radial MR angiography with 3D total variation on graphics hardware. *Magnetic resonance materials in physics, biology and medicine*, 23(2):103–114, 2010.

[11] Ji Liu and Stephen Wright. An accelerated randomized kaczmarz algorithm. *Mathematics of Computation*, 85(297):153–178, 2016.

[12] Manasavee Lohvithee, Ander Biguri, and Manuchehr Soleimani. Limited cone beam reconstruction using edge preserving total variation regularisation. 2017.

[13] Yurii Nesterov. A method of solving a convex programming problem with convergence rate o (1/k2). In *Soviet Mathematics Doklady*, volume 27, pages 372–376, 1983.

[14] Sebastian Ruder. An overview of gradient descent optimization algorithms. *CoRR*, abs/1609.04747, 2016.

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

[16] WP Segars, G Sturgeon, S Mendonca, Jason Grimes, and Benjamin MW Tsui. 4D XCAT phantom for multimodality imaging research. *Medical physics*, 37(9):4902–4915, 2010.

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

[18] Thomas Strohmer and Roman Vershynin. A randomized kaczmarz algorithm with exponential convergence. *Journal of Fourier Analysis and Applications*, 15(2):262–278, 2009.

[19] Ilya Sutskever, James Martens, George Dahl, and Geoffrey Hinton. On the importance of initialization and momentum in deep learning. In *International conference on machine learning*, pages 1139–1147, 2013.

[20] Curtis R Vogel and Mary E Oman. Iterative methods for total variation denoising. *SIAM Journal on Scientific Computing*, 17(1):227–238, 1996.

[21] Mingqiang Zhu and Tony Chan. An efficient primal-dual hybrid gradient algorithm for total variation image restoration.

[22] Mingqiang Zhu, Stephen J Wright, and Tony F Chan. Duality-based algorithms for total-variation-regularized image restoration. *Computational Optimization and Applications*, 47(3):377–400, 2010.