Bauhaus-Universität Weimar
Faculty of Media
Degree Programme Computer Science and Media

Implementation and Parallelisation of Cone-Beam Computed Tomography Backprojection on CPU and GPU

Master's Thesis

Bastian Weber Matriculation Number 90013
born [redacted]

1. First Referee: Prof. Dr.-Ing. Volker Rodehorst
2. Second Referee: PD Dr. Andreas Jakoby

Submission date: August 29th, 2016

Abstract

Cone-beam computed tomography (CT) is gaining more and more popularity for many applications in engineering and science. The most common cone-beam CT reconstruction algorithm is the Feldkamp-Davis-Kress (FDK) algorithm. However, most CT reconstruction software is only available closed-source. The few known open-source implementations are too slow and not suited well for practical use. In this thesis we present a parallelised, fast and extensible open-source implementation of the FDK algorithm using C++ and CUDA, able to run on the CPU as well as on one or multiple GPUs. We will describe the algorithm, discuss questions that arose and challenges we were facing and evaluate the results with regard to performance and quality.

Contents

1 Introduction

Computed tomography is a technique widely used today in medical and industrial applications. In [1] a very good explanation of its fundamentals can be found, which we will use as a basis here. Computed tomography is based on x-radiation (short: x-ray), which was discovered by Wilhelm Konrad Röntgen in 1895 as "a new kind of rays" [2]. This is also where the German term for x-radiation, Röntgenstrahlung (Röntgen-radiation), originates from. Thanks to his discovery Röntgen received the first Nobel prize in Physics in 1901. X-rays are able to penetrate solid materials, being attenuated in the process. This attenuation can then be captured by a sensor, giving an image of the attenuation properties of an object, which correlate to the density of its material. On the resulting images high-density objects appear darker and low-density objects brighter [3]. This way it is possible to "look inside" of objects using x-rays. In 1917, Johann Radon was the first to describe the reconstruction of a function from its projections, laying the basis for CT-reconstruction [4].

The origins of the term tomography are the Greek words $\tau o \mu o \varsigma$ (slice) and $\gamma \rho \alpha \phi \epsilon \iota \nu$ (to write). The first CT scanner was developed in 1972 by Godfrey N. Hounsfield, and the first whole-body CT scanner in 1974 by Robert S. Ledley. During the CT-scanning process an x-ray tube and an x-ray detector capture images of an object from many different angles. These images cover the entire field of view and consist of small pixels, each representing one thin x-ray beam. Figure 1.1 [1] shows a fan-beam geometry and a parallel-beam geometry. As a result, the line attenuation measurements for all possible angles and all possible distances from the centre are obtained. From this the x-ray attenuation properties of every point can then be reconstructed. Consequently, using computed tomography, it is possible to produce cross-sections of the scanned object. Using more advanced techniques the object can also be rendered as a three-dimensional model. Recent CT scanners use scintillator crystals in combination with photodiodes for producing images. The scintilla-

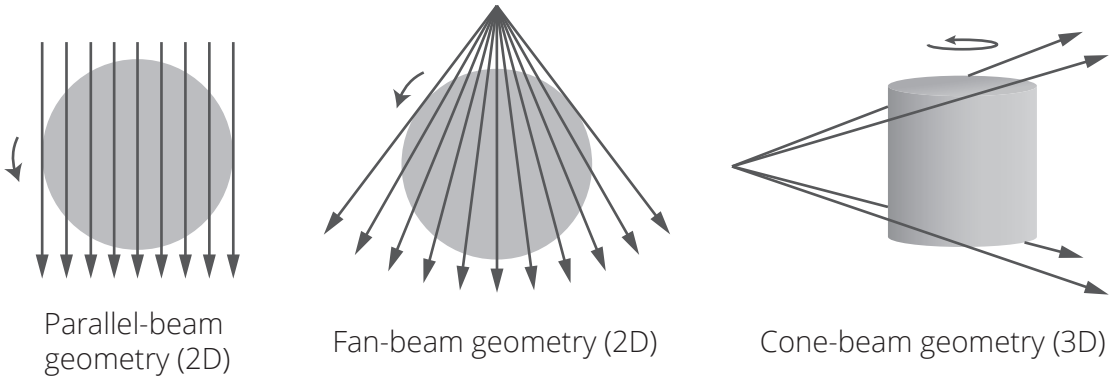


Figure 1.1: *The basic geometry of a parallel-beam, a fan-beam, and a cone-beam CT*

tor crystals convert x-radiation to visible light, enabling the photodiodes to sense it and produce a response in the form of an electric current.

Cone-beam CT, which uses an x-ray point source and a two-dimensional detector, is gaining popularity for several reasons. In comparison to parallel-beam or fan-beam CT it provides higher image resolution, better radiation utilisation and easier hardware implementation. Furthermore, the time necessary for obtaining the projections is shorter than with the conventional techniques. Apart from biomedical science, material engineering and industrial non-destructive evaluation it is used for non-destructive inspection systems and explosive detection systems in airports. [5; 3]

The most popular approximate cone-beam reconstruction algorithm is the Feldkamp algorithm. It is limited by circular scanning, spherical specimen reconstruction and longitudinal image blurring [5]. Mathematically, the reconstruction can be seen as an inverse Radon transform, while the projection is a forward Radon transform. The inverse Radon transform is approximated by backprojection, which means that the projections are basically "smeared" back through the volume from the angle at which they were taken. For obtaining a sharp reconstruction a prefiltering of the images is necessary. [3]

1.1 Motivation

In the year 2014 the Bauhaus-Universität Weimar acquired a *General Electrics Phoenix Nanotom M* cone-beam CT scanning system, driven by the efforts of the chairs of *Polymere Werkstoffe* (polymeric materials), *Baustatik und Baufestigkeit* (architectural statistics and structural strength), *Werktstoffe des Bauens* (materials of construction), *Simulation und Experiment* (simulation and experiment) and *Computer Vision in Engineering*. The new system was intended to be used to analyse materials without having to destroy samples, as is necessary for microscopic examination. This allows researchers to capture and analyse the decay of building structures and its temporal progress with high accuracy. [6]

For this purpose, a commercial closed-source CT reconstruction solution is currently used. This makes it impossible to alter or extend the used algorithms and techniques, and causes scientists to be dependent on an external party. The Computer Vision chair of the study programme Computer Science and Media has, however, a scientific interest in researching CT reconstruction techniques and related algorithms. The results of this research could also prove beneficial for the other chairs using the CT scanner and reconstruction for their purposes.

Therefore, the aim of this work is to create a basis for an open-source CT cone-beam reconstruction application that can be used not only by the Bauhaus Universität Weimar but also by the whole scientific community to conduct practical research in the field of CT reconstruction. Consequently, the resulting solution could one day render the need for third-party products unnecessary.

1.2 Goals

Our vision was to create a program that implements the Feldkamp-David-Kress cone-beam backprojection algorithm. It was our aim to maximise performance as much as we could and be able to keep up with the competition in this regard, while offering correct reconstruction results. The focus was put on speed rather than on quality optimisation for now. Memory optimisation, however, was not one of the primary goals. The application should be able to run parallelised on CPUs as well as on one or multiple GPUs, thus being usable

with a wide range of different hardware configurations. At the same time the multi-GPU support should make it possible to accelerate execution to virtually any level desired by adding and utilising as many graphics processing units as available hardware allows. Furthermore, the resulting application should be suited for actual practical use and be able to process the scan results of the Bauhaus-Universität Weimar's CT-scanner. Thus, there should be provided a method to view the output, and, if possible, compatibility to other third-party rendering applications.

1.3 Related Work

The FDK-algorithm was originally introduced in [7]. The books [8] and [9] give a detailed explanation of this algorithm, as well as many other CT techniques and related mathematical concepts. In [1] there is an overview on the computed tomography topic in general. The authors of [5] describe their attempts to achieve a fast implementation of the Feldkamp algorithm using a curved-voxel-approach and precomputed mapping tables. In [3] the implementation of the Feldkamp algorithm on a CELL Broadband Engine microprocessor architecture is described.

There are several commercial closed-source CT reconstruction applications available that implement the FDK algorithm. However, there is only one open-source application known to us that does so, which is *OSCaR* (Open Source Cone-beam Reconstructor), an open-source Matlab implementation of the FDK algorithm, that was developed by the University of Toronto, supported by the *American Association for Physicists in Medicine* (AAPM), the *Natural Sciences and Engineering Research Council* (NSERC) of Canada and by the *Mathematics of Information Technology and Complex Systems* (MITACS) Canadian research network. [10; 11]

1.4 Structure

The explanation of the underlying algorithm will be performed in chapter 3. Then, chapter 4 discusses the basic realisation of the implementation of the algorithm on the CPU, the tools which were used, and some practical concerns about the usage of the program and

its public availability as an open source application. In chapter 5, the actual parallelisation of the algorithm is explained. This comprises the parallelisation on the CPU as well as the GPU implementation, which is inevitably parallelised. Chapter 6 then discusses the results and the success with regard to our goals.

2 Fundamentals

In this chapter, a few field specific terms used in this work are explained. Readers with prior knowledge of computer science may want to skip ahead.

Bi-linear interpolation Bi-linear interpolation is an interpolation technique used with two-dimensional data structures such as images. Interpolation is necessary in order to obtain values for points that lie between the discrete data elements of a two-dimensional data structure forming a regular grid, such as an image. Bi-linear interpolation is the combination of linear interpolations on two dimensions. When performing bi-linear interpolation, two linear interpolations are performed on the first dimension and then one linear interpolation between the two resulting values is performed on the second dimension. Figure 2.1 illustrates bi-linear interpolation between the colours of four pixels. Alternatives to bi-linear interpolation include nearest-neighbour interpolation (less complex) or bi-cubic interpolation (more complex).

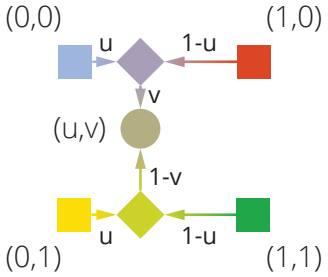


Figure 2.1: Visualisation of bi-linear interpolation. First, the two pixels in each row are interpolated linearly. Then the two resulting values are again interpolated linearly to obtain the final result.

CUDA Blocks and threads When processing data on the GPU using CUDA, there is usually one thread for each data element processed. These threads are not all active at the

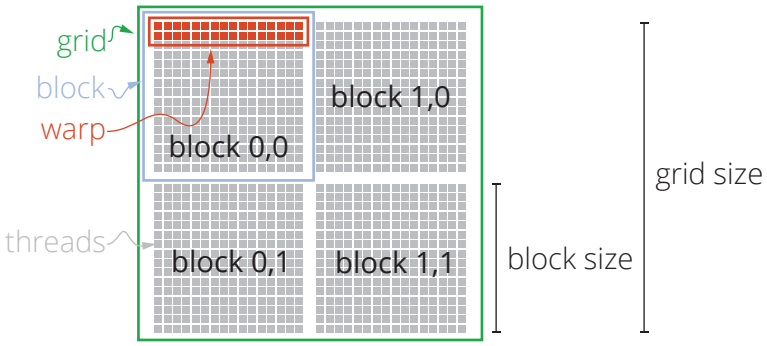


Figure 2.2: Visualisation of CUDA blocks and threads. The grid can have up to three dimensions and is subdivided into blocks. A warp then is formed out of 32 sequential threads from one block. Inside each of the blocks, threads are ordered in an x-fastest manner.

same time. Instead, a unit of threads is organised into what is known as a *block*. The blocks together from what is called the *grid*. The amount of resulting blocks depends on how many threads or data elements there are in total and on the size of the blocks. This block size can be chosen by the programmer within certain bounds. Different block sizes result in different levels of performance. Each thread has a thread-ID within its block and each block has a block-ID within the grid. Inside of a kernel, both of these IDs as well as the block size are available and can be used to calculate the index of the data element currently being processed (see Equation 2.1). Block sizes can be one-dimensional, two-dimensional, or three-dimensional. Consequently, the thread-IDs can be one-dimensional, two-dimensional, or three-dimensional vectors. The same goes for the grid. The threads from each block are ordered in an x-fastest manner [12, section 2.2].

$$globalIndex = blockID \times blockSize + threadID \tag{2.1}$$

The threads belonging to one block are always executed on the same streaming multiprocessor. For this reason, each of the threads from a block accesses the same L1 cache, which is important regarding memory coalescing. Furthermore, the threads from one block have access to a shared memory and their execution can be synchronised. Shared memory was not utilised in our application.

Even though the threads from within one block are executed on the same streaming multiprocessor, they are not all active at the same time. Only 32 threads can execute instructions

on one streaming multiprocessor at a time. Such a group threads is called a *warp*. Because of this warp size, the block size should always be divisible by 32. This is important, because only then it can be evenly divided into warps. Otherwise warps with less than 32 threads would occur, in which case the capabilities of the streaming multiprocessors could not be fully utilised. Figure 2.2 shows the relation between the grid, blocks, warps and threads. [13]

CUDA streams CUDA streams are used to asynchronously compute concurrent tasks executed on the same device. Kernel launches on the GPU generally are asynchronous, i.e. the CPU thread they are launched from does not block until the kernel execution is finished. However, kernels are executed in the order in which they are launched. Sometimes it is desirable to let kernels run on the GPU asynchronously. This is the purpose that streams serve. All the kernels launched on one stream, Stream A, will be executed in the order they have been added to that stream. However, kernels launched on a second stream, Stream B, are not in any way synchronised to Stream A and may be executed at an arbitrary point in time with regard to the kernels on Stream A. Basically, streams serve the same function as threads do on the CPU: They allow the independent, detached execution of multiple functions. [12, section 3.2.5.5; 14]

We, for example, use streams to upload images to the GPU and to simultaneously preprocess them to a running reconstruction kernel.

Coordinate system orientation A three-dimensional Cartesian coordinate system can either have right-handed (positive) orientation or left-handed (negative) orientation [15]. A coordinate system is called right-handed if rotating the x-axis towards the y-axis results in a counter-clockwise (positive) rotation when looking from positive z-direction to the origin; otherwise it is called left-handed.

Fourier Transform The Discrete Fourier Transform (DFT) is a fundamental transform in digital signal processing. It enables the decomposition of a discrete signal into its frequency components, that can then be evaluated or manipulated. One common application of the DFT is creating frequency filters such as high-pass or low-pass filters. The Fast Fourier Transform (FFT) is a fast algorithm for computing a DFT. Assuming a symmetric signal, an even-length DFT is fully described by its first $\frac{N}{2} + 1$ values; an odd-length DFT is fully de-

scribed by its first $\frac{N+1}{2}$ values. For both, even- and odd-length DFTs, the output length can be calculated as $\lfloor \frac{N}{2} + 1 \rfloor$. [16]

FFT Plan The data structures required for the intermediate results of an FFT are called an FFT plan. Once this plan has been created, multiple FFTs can be executed using it. The advantage is that these data structures do not have to be allocated and deallocated multiple times, saving execution time.

Overhead We refer to computational overhead as computational effort that has system-based causes and is necessary only for the management of the program itself rather than for the actual task that is to be completed. For example, creating a new thread on the CPU takes a certain computational effort that has to be taken into account, but does not contribute anything to the completion of the actual task. The same is the case for memory allocations, deallocations, or copy operations to the video memory. This additional workload is called overhead.

We also use the term *thread scheduling overhead*. By this we mean the overhead that occurs when assigning tasks to threads. For example, let's assume we want to perform an operation on 100 elements in four parallel threads. One possibility would be to launch one thread for each element, resulting in 100 threads that are launched at once. Another option would be to statically divide the elements into four subsets, launch just four threads and let each thread process the subset it was assigned. The second option has a much smaller computational overhead than the first one.

Page-locked memory Modern operating systems use virtual memory (also called page file), i.e. memory that can be used like RAM but is actually stored somewhere else, e.g. on the hard disk drive. This technique is used to handle cases where the amount of physical RAM is insufficient. Moving content from the physical to the virtual memory is called *paging*. *Page-locked memory*, or *pinned host memory*, is memory allocated in the RAM that cannot be moved to virtual memory.

The GPU driver cannot access data from pageable host memory directly. When copying such data to the VRAM it has to copy it to a temporary page-locked portion of main memory first, and can then transfer it from there to the GPU memory. The copy operation can bypass the CPU if the data in the RAM is already page-locked. First, this makes the copy operation faster under certain conditions. Second—and that is most important for us—

the copy operation can happen asynchronously. This means that the CPU thread from which the copy operation is executed does not block until the operation terminates, but continues execution. [17; 12, section 3.2.4]

Region of interest A region of interest is a boundary that limits a data range to a certain extent. It is essentially a crop. In our case the term is used to describe a cuboid-shaped area that can be defined by the user. The reconstruction is then confined to this area.

RAM and VRAM We refer to *RAM* or *main memory* as the *random access memory* of the computer. This is the memory that the CPU can access directly. We refer to *VRAM* as the *video-RAM*, i.e. the memory that the GPU can access directly. It is also called *global device memory* or just *global memory* in the context of GPUs.

Rounding There are different rounding notations that have been used in the equations in this work. The notation used in equation 2.2 means that a is rounded to the next smallest integer, while the notation from equation 2.3 means that a is rounded to the next larger integer. Rounding to the nearest integer was written as can be seen in equation 2.4.

$$\lfloor a \rfloor \tag{2.2}$$

$$\lceil a \rceil \tag{2.3}$$

$$\lfloor a \rceil \tag{2.4}$$

Single and double precision Computer systems use floating point data types to represent real numbers, i.e. decimal fractions. Like all data types these can have different sizes. Most common are the types *float*, or *single precision*, and *double*, or *double precision*. The length of these data types depends on the processor architecture and the compiler. We, however, refer in this thesis to *32-bit-float* as *single precision* and to *64-bit-double* as *double precision*. While *double* requires twice as much memory as *float*, it also provides higher accuracy. On the CPU the differences in computational effort between *double* and *float* are rather small; however, on the GPU these are usually bigger. This also depends on the GPU model used.

Volume and voxels A *volume* is basically a three-dimensional grid structure. It consists of discrete individual elements that are ordered along three orthogonal axes. Each of these

elements has one particular value and is called a *voxel*. This value can, for example, be an intensity or a colour.

3 Algorithm

During the scanning procedure, the detector of the x-ray scanner captures x-ray images of the object in small angular steps [1]. For this purpose, either the object is rotated, or the x-ray source and the x-ray detector revolve around the object. Either method results in the same relative movement. Figure 3.1 schematically illustrates this procedure. The goal of the algorithm is then to create a three dimensional density volume from these images via backprojection.

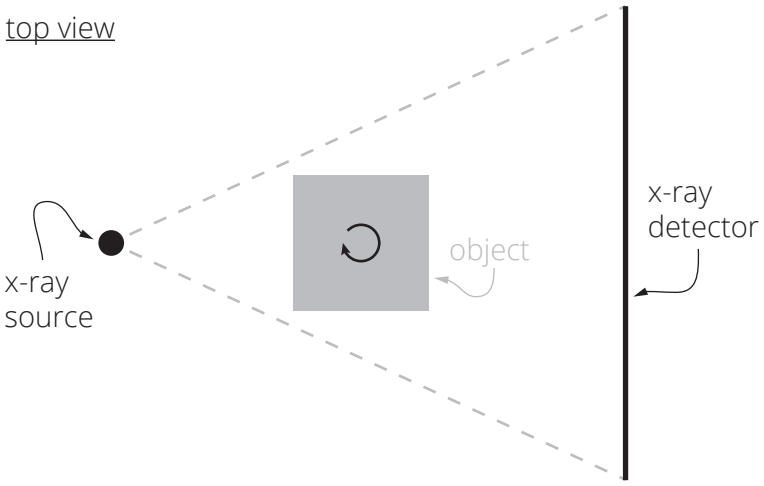


Figure 3.1: *The scanning procedure. The object rotates, allowing the x-ray source and the x-ray detector to capture images in small angular steps.*

3.1 Prerequisites

3.1.1 Coordinate Systems and Geometry

There are two coordinate system representations that have to be understood in order to understand the reconstruction process. The transformation between these coordinate systems is one of the essential aspects of the algorithm.

Figure 3.2: *The coordinate systems of the image and the volume*

The first one is the image coordinate system, which is of two-dimensional Cartesian type. It is used with the input images. Its horizontal axis, the columns of the image, is denoted u, the vertical axis, the image rows, v. The second coordinate system is the coordinate system of the volume. It is a right-handed three-dimensional Cartesian coordinate system. Its axes are denoted x, y and z. Both coordinate systems are illustrated in figure 3.2.

Regarding the capturing geometry, the x-ray source and the x-ray detector basically can be seen as a normal photographic camera. Like visible light, the rays intersect in a projection centre and produce an image on the image plane. Figure 3.3 shows this geometric model [8, p. 373]. The rays originate from the projection centre and are projected onto the physical detector behind the object. The distance the x-ray source has from this detector is denoted SD, the source-detector-distance. This distance together with the size of the detector defines the opening angle of the fan or, in other words, the field of view. The virtual detector is an image of the physical detector at the distance FCD, the focus-centre-distance, from the projection centre. The focus-centre-distance can be chosen arbitrarily as long as the pixel size is calculated accordingly (compare section 3.1.3).

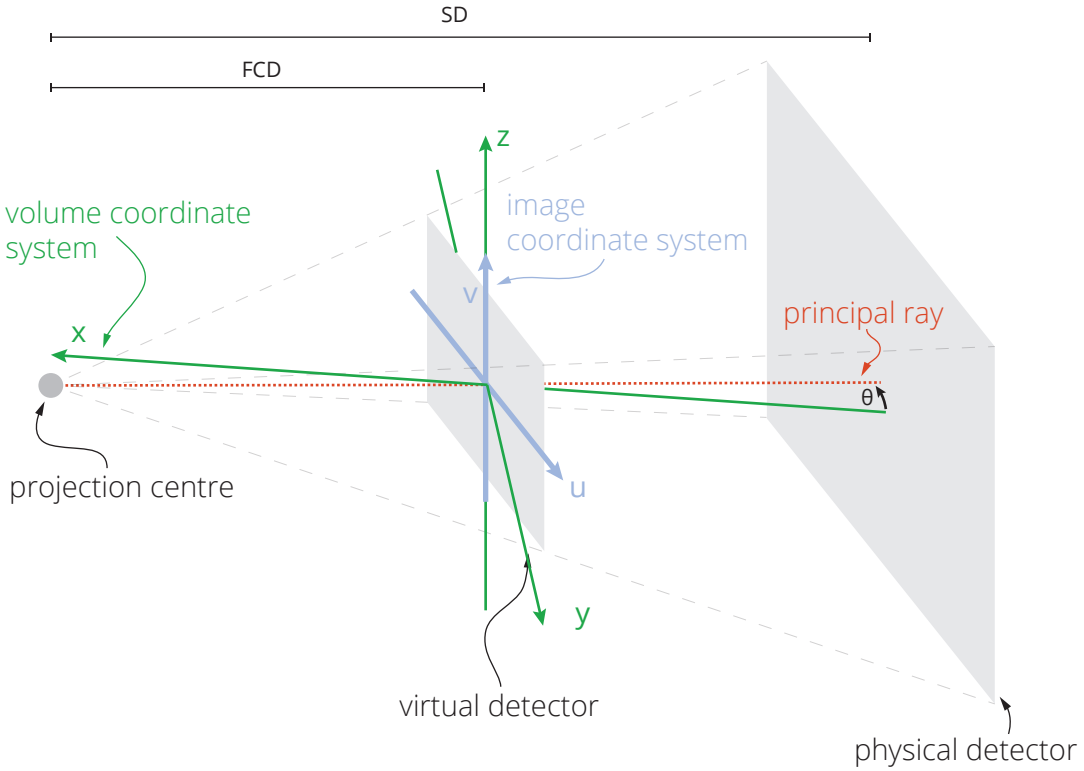


Figure 3.3: The geometry of the reconstruction

The volume-coordinate-system is shown in green. Its origin is at the distance FCD from the projection centre, being the pivot point of the rotation. The virtual detector is our image plane. It is perpendicular to the principal ray and intersects the origin of the volume coordinate system. The origin of the volume-coordinate-system and the image-coordinate system are at an identical point in 3D-space.

The angle θ is the angle at which an image was captured. It is defined as the angle between the principal ray and the x-axis of the volume-coordinate-system. The orientation of the image coordinate system relative to the image plane and the projection centre stays the same for all values of θ. However, the orientation towards the volume-coordinate-system changes as the angle θ changes. Projecting points from the volume-coordinate-system into the image-coordinate-system is one necessary step of the backprojection algorithm.

3.1.2 Input Data

First and foremost, the x-ray images form the input data of our algorithm. Along with these, a couple of capturing parameters have to be provided:

Angles The angle θ at which the individual image has been taken in degrees. This parameter is required for each image.

Z-offset The offset of the individual image in the vertical direction in units of length. This parameter is required for each image.

FCD The focus centre distance, which is the distance of the x-ray source to the virtual detector in units of length. This distance can theoretically be chosen arbitrarily as long as the pixel size is calculated correspondingly as explained in section 3.1.1.

Pixel size The size of one pixel on the virtual detector in units of length. Square pixels are assumed.

U-offset The offset of the rotation axis from the image centre in the horizontal direction in units of length.

Base intensity The intensity that is measured when no object is between the x-ray source and the detector. This value must be provided relative to the maximal possible pixel value (which would be 255 for an 8 bit or 65535 for a 16 bit image). Thus, it must be in the interval $[0 - 1]$. This has the advantage that images with different bit depth can be used without requiring different values.

The *z-offset* or *height offset* can be used as a technique to prevent artifacts in the reconstruction. If there is a defect that is consistent amongst all images, this will result in artifacts in the reconstruction. For example, a dead pixel at a certain position in the image could cause ring-like artifacts in the reconstruction. To counter this, a z-jitter can be used when performing the scan. This means that the vertical position of the object that is being scanned will be varied slightly for each image. As a result, defects on the image plane will, seen relatively to the volume, always be in different locations and will, in the end, average out in the reconstruction and thus be invisible. If this technique is not used, the z-offset values can just be zero or any arbitrary constant for each images.

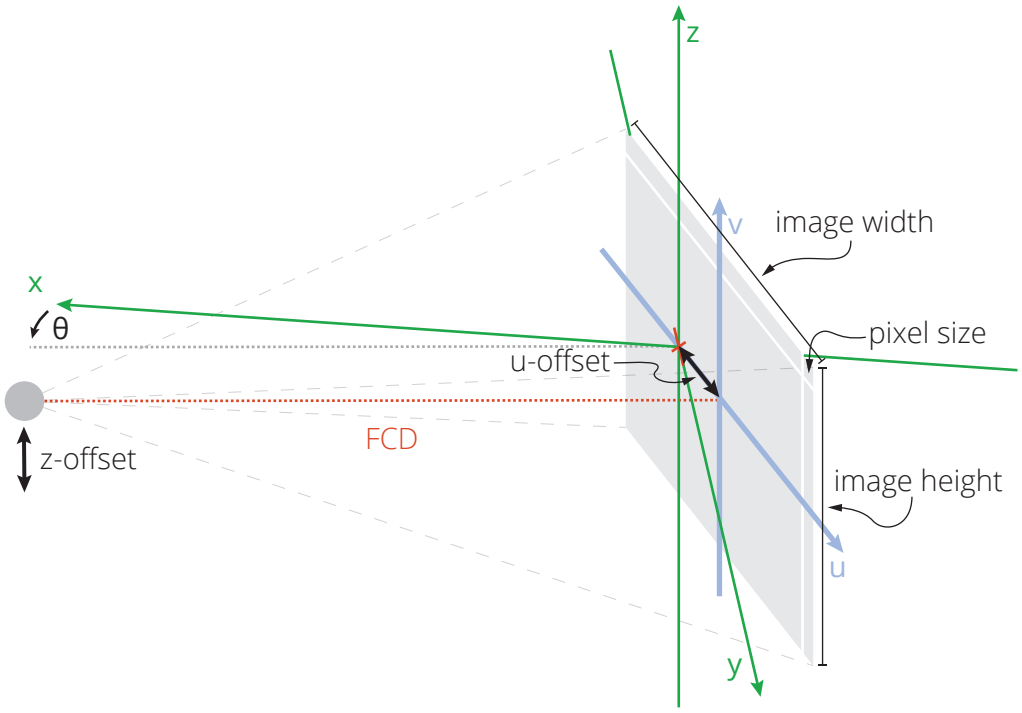


Figure 3.4: Visualisation of the input parameters

These parameters are provided through a configuration file, a detailed description of which can be found in 4.4. There are several further parameters which do not need to be supplied manually, but are automatically deduced from the images:

Image width The width of the images in pixels

Image height The height of the images in pixels

Volume size X The resolution of the volume in X-direction. This is proportional to the image width.

Volume size Y The resolution of the volume in Y-direction. This is proportional to the image width.

Volume size Z The resolution of the volume in Z-direction. This is proportional to the image height.

This implies that the size of the volume is always relative to the size of the images. Although this is currently the case in our implementation, it would also be possible to choose the resolution of the volume independently from the image resolution. However, what is already possible is to constrain the reconstruction to a region of interest. The exact volume dimensions can be calculated according to equation 3.36 in section 3.2.4. Figure 3.4 visualises these parameters.

The unit of length can be any arbitrary measure, e.g. millimeters. For the algorithm it does not make a difference which unit is used in particular, as long as all parameters use the same one. Our implementation internally uses pixels as a unit of measurement. Therefore, after reading the input parameters, all their values are converted to pixels using the pixel size parameter provided. This makes it particularly easy to transform coordinates between image and volume because all components use the same unit of measurement.

Along with these essential parameters there are some additional parameters which can be specified to control the behaviour of the reconstruction process:

Filter type For the preprocessing of the images there are three different types of frequency filters available:

- Ram-Lak
- Shepp-Logan
- Hann

These filters are explained in more detail in section 3.2.1.

Volume bounds These are six parameters, two for each dimension (minimum and maximum), that define a region of interest. Only this region of interest will be reconstructed.

3.1.3 Units

As a unit we use $pixel$ in both coordinate systems, which is defined as the distance between one pixel and its direct neighbour along one of the image coordinate system's axes.

Per definition, the two coordinate systems are isotropic, which means that this distance is the same as the distance of one voxel to its direct neighbour along one of the volume coordinate system's axes. This makes the transformation of a coordinate between the two coordinate systems as simple as possible. Furthermore, all integer coordinates can be used without scaling as indices to access the corresponding voxel or pixel in the actual physical representation of the volume or image in memory, where they are stored in the form of an array-like data type.

All parameters which might be provided in "real-world measures" such as *millimetres* will be converted to the unit *pixels*. This conversion has only to be done once when the parameters are read, which makes the computational effort minimal. Otherwise such a conversion would be necessary each time a voxel is accessed, requiring additional computational operations to be performed during the reconstruction and likely impairing reconstruction performance. Please note that there are still some coordinate transformations necessary when converting between coordinates and indices, due to the fact that the origin of the coordinate systems is in the centre of the volume or in the centre of the image respectively, while the indexing of array-like data structures always starts at zero and does not know negative values. Thus, the coordinates have to be shifted when transforming them to indices. Taking into account the possibility that only a certain region of interest will be reconstructed, instead of the whole volume, the transformation involves some additional calculations, which will be explained in section 3.2.4.

The conversion itself is quite simple. It should be noted that we work with square pixels. Assuming the pixel size p is known in an arbitrary unit, a constant c measured in said unit can be transformed to the measure $pixel$ by

$$c_{\text{converted}} = \frac{c}{p}, \tag{3.1}$$

resulting in the converted constant $c_{\text{converted}}$.

The pixel size p is the size of one pixel on the virtual detector, i.e. the projection of the real detector to the distance FCD. The pixel size of one pixel on the real detector, p_{real}, can be calculated from the physical detector's size (s_u, s_v) and its resolution (r_u, r_v) as shown in equations 3.2 and 3.3.

$$p_{\text{real}} = \frac{s_u}{r_u} \tag{3.2}$$

$$p_{\text{real}} = \frac{s_v}{r_v} \tag{3.3}$$

Knowing the distance of the x-ray detector to the x-ray source, SD, and given a focus centre distance FCD, it is possible to calculate the pixel size p on the virtual detector by multiplying p_{real} by the ratio of FCD to SD as in equation 3.4.

$$p = p_{\text{real}} \frac{FCD}{SD} \tag{3.4}$$

Taking equations 3.2, 3.3 and 3.4 into account, equation 3.1 can also be written as one of the following:

$$c_{\text{converted}} = c \, \frac{SD \, r_u}{FCD \, s_u} \tag{3.5}$$

$$c_{\text{converted}} = c \, \frac{SD \, r_v}{FCD \, s_v} \tag{3.6}$$

The pixel size p and the focus centre distance FCD have to be known for the reconstruction. They are usually obtained in the calibration stage, using a calibration object consisting of two ruby spheres.

3.2 Reconstruction

In [8, pp. 384–368] the Feldkamp, Davis and Kress reconstruction is described as in equations 3.7 to 3.11. Note that the formulas have been adapted to match our coordinate system. The operand in the term $(x\cos(\theta) + y\sin(\theta))$ seems to be given wrong in [8], or at least does not match the given axis orientation. It has been changed according to [5] and [9, p. 105].

$$f(x,y,z) = \underbrace{\int_0^{2\pi}}_{\substack{\text{integral over} \\ \text{all angles}}} \underbrace{\frac{FCD^2}{U(x,y,\theta)^2}}_{\text{weight}} \underbrace{h_\theta\big(\overbrace{u(x,y,\theta), v(x,y,z,\theta)}^{\substack{\text{projection from volume coordinates} \\ \text{to image coordinates}}}\big)}_{\substack{\text{sample from filtered image at} \\ \text{projection point of voxel}}} \mathrm{d}\theta \tag{3.7}$$

$$U(x,y,\theta) = FCD - (x\cos(\theta) + y\sin(\theta)) \tag{3.8}$$

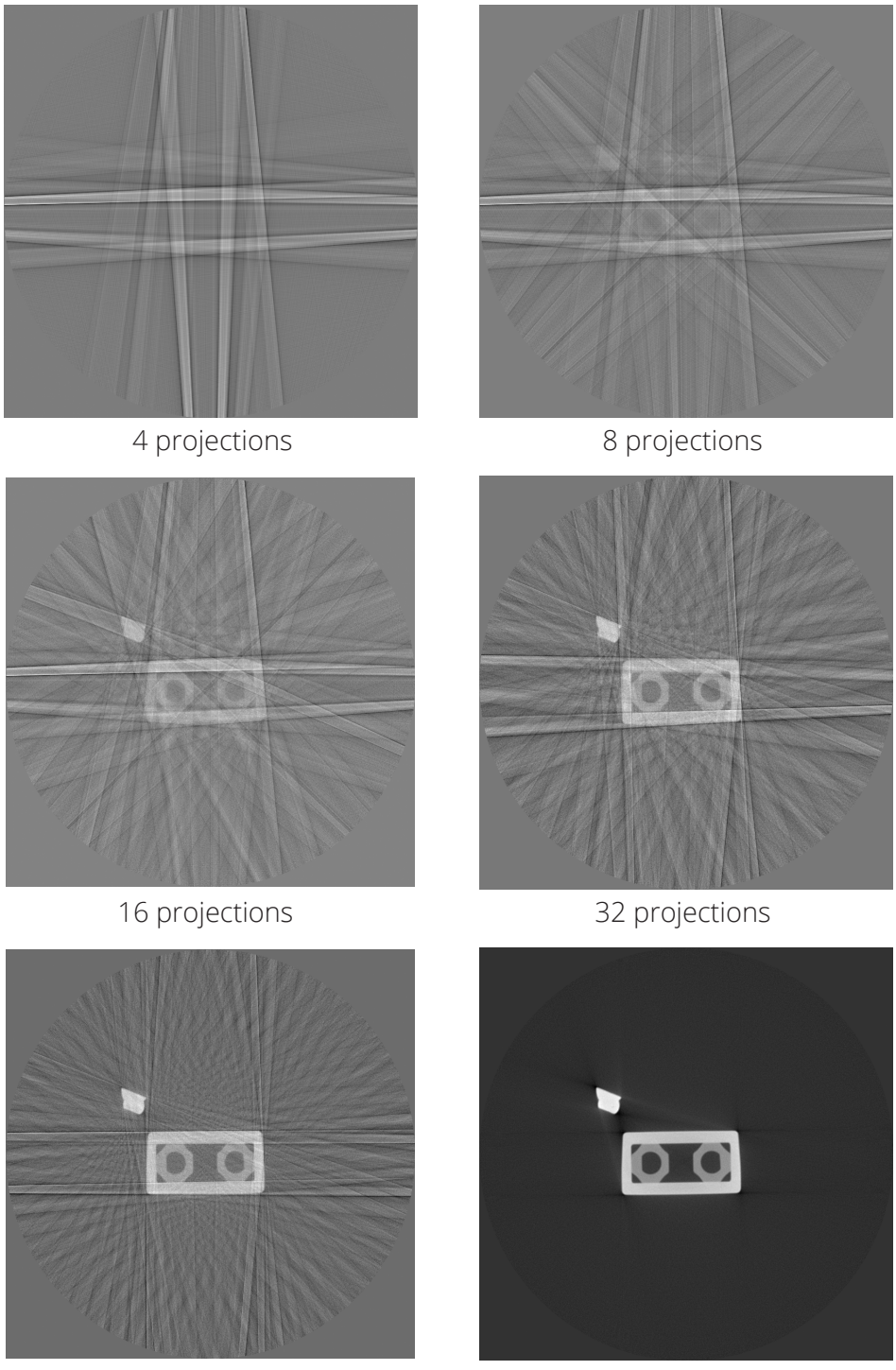


Figure 3.5: *Cross sections of backprojection results with varying amounts of projections*

$$u(x, y, \theta) = \frac{(-x\sin(\theta) + y\cos(\theta))\, FCD}{FCD - (x\cos(\theta) + y\sin(\theta))} \tag{3.9}$$

$$v(x, y, z, \theta) = \frac{zFCD}{FCD - (x\cos(\theta) + y\sin(\theta))} \tag{3.10}$$

$$h_\theta(u, v) = \frac{1}{2}\Bigg(\underbrace{\phi_\theta(u, v)}_{\substack{\text{pixel at} \\ \text{coordinate } (u,v)}} \underbrace{\frac{FCD}{\sqrt{FCD^2 + u^2 + v^2}}}_{\text{cosine weight}}\Bigg) \star \underbrace{g(a)}_{\text{frequency filtering}} \tag{3.11}$$

Equation 3.11 shows the preprocessing of the image, consisting of the Feldkamp or cosine weight and the highpass filtering in the frequency domain, which is the first step in the algorithm. Equations 3.9 and 3.10 show the projection of a voxel from the volumes x-y-z-coordinate system to the u-v-coordinate system of the image plane. Equation 3.7 combines these parts to one algorithm that integrates the filtered pixel values over all angles, weighted by a per-voxel weight, to reconstruct the value of one voxel.

To give a general idea of the discretised version of the algorithm, it can be broken down to the essential steps shown in Algorithm 1. Figure 3.5 shows how the backprojection result gradually becomes clearer as the number of projections used increases.

Data: The images and the capturing parameters
Result: Reconstructed CT volume

```
1 allocate volume and initialise it with 0;
2 foreach image do
3     preprocess image;
4     foreach voxel do
5         project voxel coordinate to image coordinate system;
6         sample density value from image at projected voxel coordinate;
7         calculate weight;
8         add weighted density value to current value of voxel;
```

Algorithm 1: *Essential steps of the discretised FDK-algorithm*

3.2.1 Image Preprocessing

Before backprojection can be performed on an image, the image has to be preprocessed. We assume our images to have one channel. If this is not the case, the channels will be reduced upon reading of the image. Preprocessing then comprises the following steps:

1. Conversion to 32 bit float representation and scaling to identical data range
2. Normalisation with regard to the x-ray source's base intensity and logarithmic scaling of the image values
3. Application of the Feldkamp cosine weights
4. Application of a high-pass filter in the frequency domain

Figure 3.6 shows the intermediate results of the the preprocessing pipeline on an exemplary image.

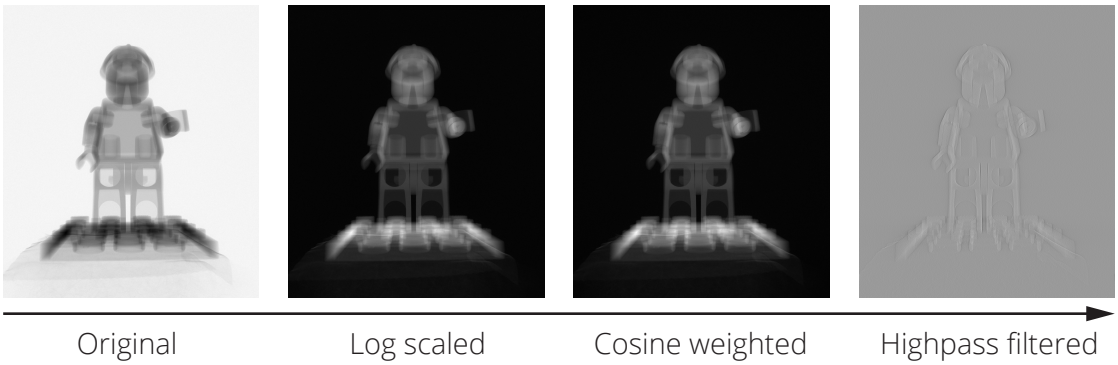


Figure 3.6: *The progression of one projection through the preprocessing pipeline (images normalised for better visibility)*

Conversion The volume that is to be reconstructed contains density values. These are represented by floating point numbers with 32 bit precision in our case. Thus the images are converted to a 32 bit float representation, as well. The data range of the pixels' values depends on the image's data type. If the image is in 8 bit unsigned integer representation, the values will be in the interval $[0, 255]$, if it is in 16 bit unsigned integer representation, they will be in-between $[0, 65535]$ and given 32 bit float representation their range will be

$[0, 1]$. As it is possible that images of different bit depths are used, the image data of all images will be scaled to the interval $[0, 1]$ after the conversion to 32 bit float representation. Equation 3.12 shows the scaling for images of arbitrary bit depth.

$$I_0^{u,v} = \frac{I^{u,v}}{2^b - 1}, \tag{3.12}$$

Here I is the image represented as a pixel matrix and $I^{u,v}$ denotes the pixel at the coordinate (u, v) of I. The value b is the bit depth of the image, which would be 8 in the 8-bit case and 16 in the 16-bit case.

Often, the bit depth of the image file is higher than the bit depth of the actual contained data. For example, acquisition may happen with only 12 bit accuracy, but the data is still stored in a 16 bit format. In any case, this operation is only dependent on the bit depth of the image.

Logarithmic Scaling In the next step the projections are normalised with regard to the x-ray source's base intensity and a logarithmic scaling is applied to the pixel values.

$$I_1^{u,v} = -\ln\left(\frac{I_0^{u,v}}{\iota}\right), \tag{3.13}$$

Here $\ln$ denotes the natural logarithm and ι denotes the base intensity of the x-ray source. This amplifies differences in the shadows and compresses the highlights. Furthermore, the negation causes the image to become inverted, which means the high-density areas of the x-ray scan become white and the low-density areas become black. [1; 18; 19; 20]

Cosine Weighting Afterwards the Feldkamp or cosine weighting is applied to the image as shown in equation 3.14.

$$I_3^{u,v} = I_2^{u,v} \frac{FCD}{\sqrt{FCD^2 + u^2 + v^2}} \tag{3.14}$$

The denominator $\sqrt{FCD^2 + u^2 + v^2}$ is the distance of the pixel (u, v) on the image plane from the projection centre [8, p. 373]. Consequently, the weight is the ratio of this distance by the focus centre distance. As a result, the values of pixels which are further away from

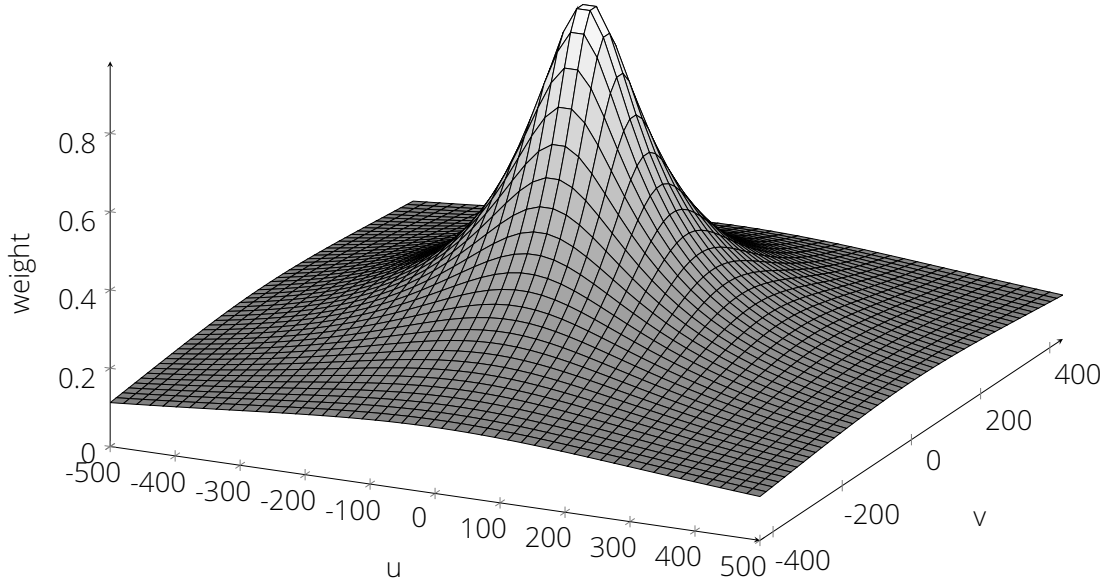
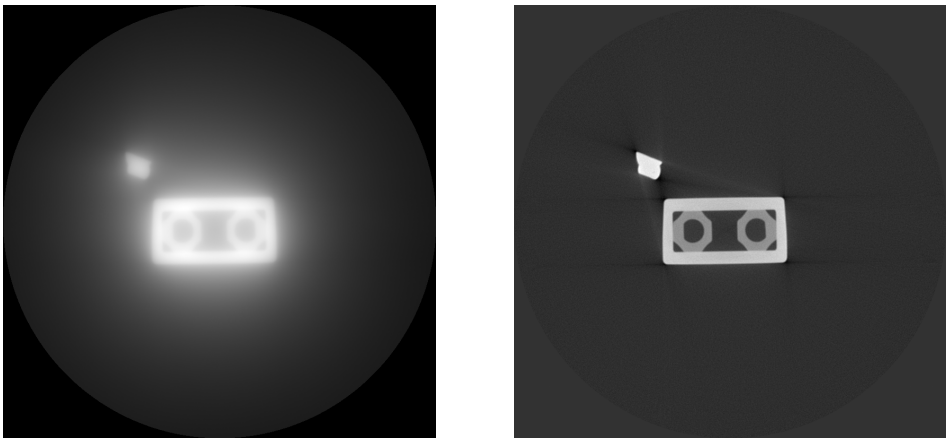


Figure 3.7: Three-dimensional plot of the cosine weights for an image of size 1000 × 875 pixels at a focal center distance of 75 pixels (scaled for better visibility)

the image centre are attenuated. Figure 3.7 shows a 3-dimensional plot of the cosine weights for an exemplary image. [8, pp. 384, 386; 5; 3; 18]

Frequency Filtering Lastly, a highpass filtering of the image will be performed in the horizontal direction. Since this happens in the frequency domain, first a one-dimensional Fourier transform of each individual row of the image is undertaken. This results in a complex-valued frequency spectrum of size $\left(\frac{w}{2}+1, h\right)$, where w is the width and h the height of the original image. The change in width happens because all values above the Nyquist-Frequency threshold, which is $\frac{N}{2}+1$, are merely the complex conjugates of the values below this threshold, according to the Nyquist-Shannon sampling theorem [16]. Therefore a packed format called CCS (complex-conjugate-symmetrical) is used [21]. After the transformation to the frequency domain a weighting filter is applied, which attenuates certain frequencies, and the image is then transformed back into spatial domain. Equation 3.15 shows the transformation process.

$$I_2 = \mathcal{F}^{-1}\left(\mathcal{F}\left(I_1\right) \circ W\right) \tag{3.15}$$



Without highpass filtering | With highpass filtering

Figure 3.8: *Comparison of the reconstruction result with and without the highpass filtering step. Without this step, no correct reconstruction is obtained.*

Here $\mathcal{F}$ is the one-dimensional row-wise Fourier transform and $\mathcal{F}^{-1}$ the inverse one-dimensional row-wise Fourier transform. W denotes the weight matrix, which is multiplied with the frequency spectrum in an element-wise fashion.

There are three options for the weight filter in our implementation. The default one is the Ram-Lak filter, which is a simple ramp filter:

$$W^{u,v}_{RamLak} = \frac{u}{n}, \tag{3.16}$$

where n is the amount of columns in the spectrum or in other words the Nyquist frequency. The Ram-Lak filter yields the sharpest result of all the prefilters. However, sometimes this is not desirable because of the resulting noise. For reducing the noise in the volume, several frequency filters exist that attenuate the high frequencies. The first one is the Shepp-Logan filter, which is a sinc (sinus cardinalis) window filter multiplied with the Ram-Lak filter:

$$W^{u,v}_{SheppLogan} = W^{u,v}_{RamLak} \frac{\sin\left(\frac{\pi u}{2n}\right)}{\frac{\pi u}{2n}}, \tag{3.17}$$

The second one is the so called Hann filter, being a Hann window filter multiplied with the Ram-Lak filter:

$$W^{u,v}_{Hann} = W^{u,v}_{RamLak} \frac{1 + \cos\left(\frac{\pi u}{n}\right)}{2}, \tag{3.18}$$

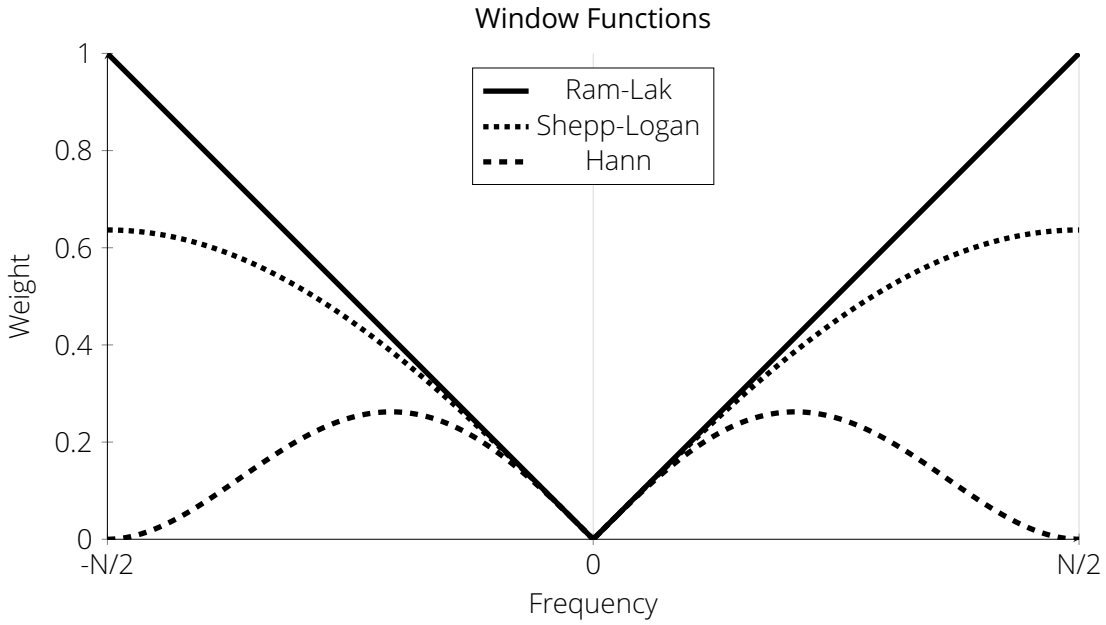


Figure 3.9: Comparison of the different window functions used for frequency filtering

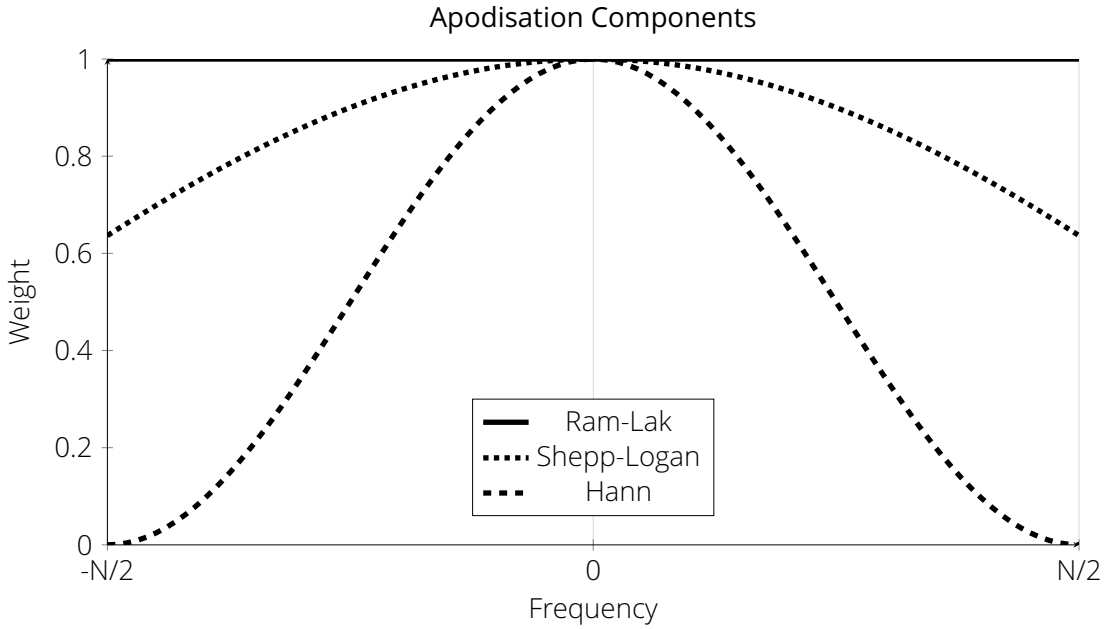


Figure 3.10: Comparison of the apodisation components of the window functions used

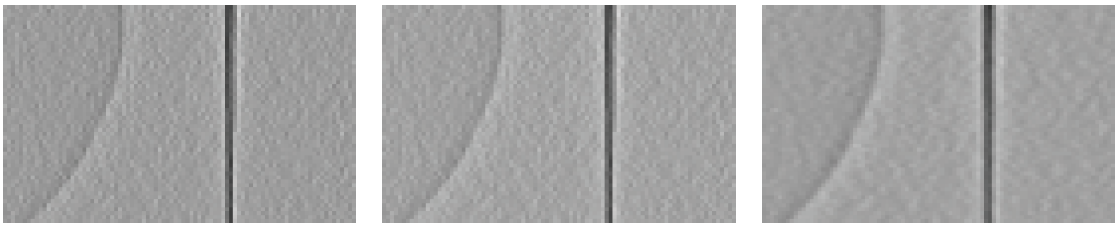


Figure 3.11: *Comparison of the different frequency filters*

Figure 3.9 shows a visual comparison of the three window filters described. As can be seen, the Shepp-Logan filter attenuates the high frequencies stronger than the Ram-Lak filter, and the Hann filter does so even more than the Shepp-Logan filter. The apodisation components of the different filters alone are plotted in Figure 3.10. Figure 3.11 shows a comparison of the results of the different frequency filters. The more of the high frequencies are removed, the softer the result becomes, yielding less noise but also less sharply defined edges in the resulting reconstruction. Please note that in our case only the part of the window function for positive values of x is used, accordingly, taking the absolute value of x has been omitted in equations 3.16, 3.17, and 3.18. Figure 3.8 shows the effect the highpass filter has on the reconstruction result.

This is only a selection of window functions. There are a variety of other possible filters that could be tried, for example the Blackman, Butterworth, or Hamming filters. After this final step the image is ready to be used for the actual backprojection stage. [8, pp. 244–247; 1; 22; 19; 5; 18]

3.2.2 The Reconstructable Cylinder

Before proceeding to the actual backprojection stage we should think about which part of the volume actually is reconstructable. The filtered backprojection algorithm is an integral over all angles, which in our discretised version means a sum over all angles. Only voxels that are visible on every single image can be correctly reconstructed. The volume itself has the shape of a cuboid, but not every point inside the cuboid is also visible on all of the images. The portion of the volume that is actually reconstructable can be seen in figure 3.12, marked in blue. We, however, approximate it as a cylinder, which means we include

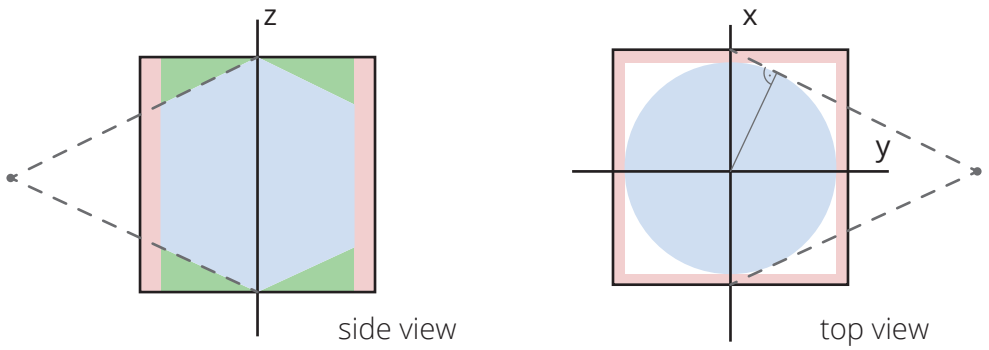


Figure 3.12: The shape of the reconstructable cylinder viewed from the side and from the top. The blue areas show the exact shape of the reconstructable cylinder. The blue areas together with the green areas represent our approximation of the reconstructable cylinder. The red areas show the part of the volume that is not allocated.

the parts marked in green. This way it is easier to compute if a voxel is inside or outside this portion. Using the exact shape would require too much computational effort compared to the effort saved by not having to include the outside voxels in the backprojection.

Figure 3.13 shows the geometry of the reconstructable cylinder. The lengths of the line segments marked in green are known to us. The variable b is half of the virtual detector's width.

$$b = \frac{r_u}{2} \tag{3.19}$$

To calculate the radius r of the reconstructable cylinder, marked in red, we first calculate the length of c.

$$c = \sqrt{FCD^2 + b^2} \tag{3.20}$$

Then, using the cathetus theorem, the lengths of p and q can be obtained.

$$\begin{aligned} p &= \frac{b^2}{c} \\ q &= \frac{FCD^2}{c} \end{aligned} \tag{3.21}$$

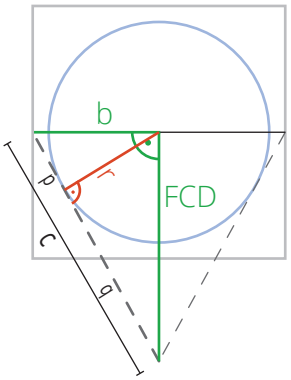


Figure 3.13: Geometry of the reconstructable cylinder. The lengths of the green line segments are known; the black and red parts are unknown. The goal is to find the radius r.

Using c, p and q and the altitude theorem we can now calculate the radius r of the reconstructable cylinder.

$$r = \sqrt{pq} = \sqrt{\frac{b^2 FCD^2}{c^2}} \tag{3.22}$$

After replacing b and c we obtain the final formula for calculating the radius r of the reconstructable cylinder, which is only dependent on FCD and r_u.

$$r = \sqrt{\frac{FCD^2 \left(\frac{r_u}{2}\right)^2}{FCD^2 + \left(\frac{r_u}{2}\right)^2}} \tag{3.23}$$

Only voxels within this radius from the z-axis will be reconstructed. Furthermore, the volume's size in x- and y-direction will only be as much as this radius and not as much as the full width of the images. This means that the parts which are marked in red in figure 3.12 will not be allocated at all, saving some memory.

3.2.3 Backprojection

At the beginning of the backprojection, the volume V is initialised to zero. Here r denotes the radius of the reconstructable cylinder, as explained in section 3.2.2.

$$V^{x,y,z} = 0 \qquad \forall\,(x, y, z) \in \{0, \ldots, r-1\} \times \{0, \ldots, r-1\} \times \{0, \ldots, r_v - 1\} \tag{3.24}$$

Once an image has been preprocessed, the actual backprojection can be performed with this image. For this the algorithm loops over each voxel and lets each voxel undergo the following procedure. First, the voxel's x-, y- and z-indices have to be transformed to x-, y- and z-coordinates in our volume's coordinate system representation. These transformations will be explained in detail in section 3.2.4.

Secondly, we check if the voxel is inside the reconstructable cylinder. Section 3.2.2 gives a detailed explanation how this formula was obtained.

$$x^2 + y^2 < \frac{FCD^2 \left(\frac{r_u}{2}\right)^2}{FCD^2 + \left(\frac{r_u}{2}\right)^2} \tag{3.25}$$

Squaring the right side instead of taking the square root of the left side of the equation spares the expensive execution of the square-root operation for each voxel. The right side of the equation has only to be computed once and will stay constant during the execution of the backprojection.

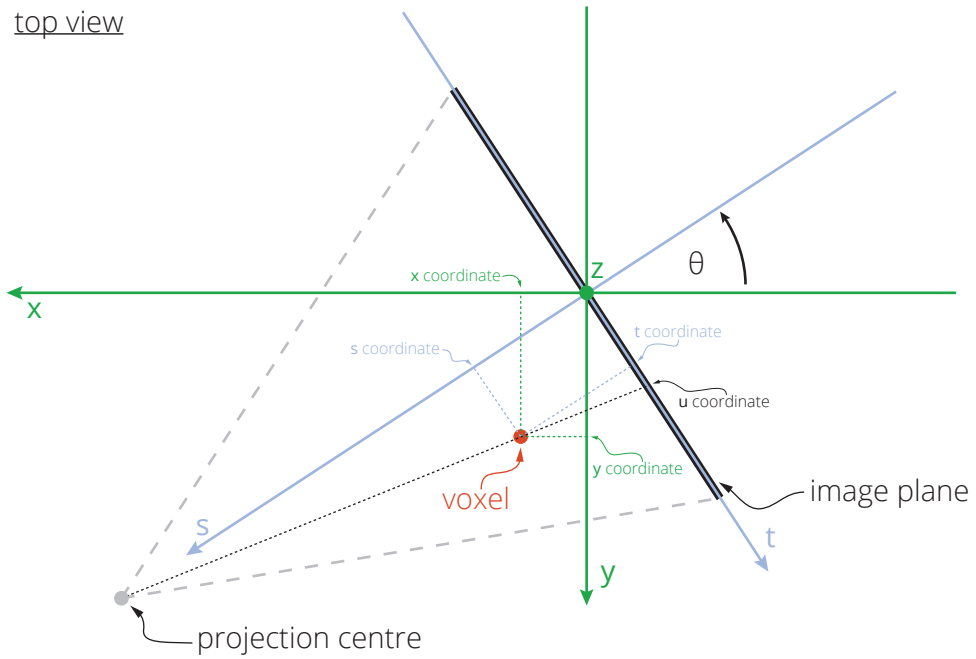


Figure 3.14: *The s-t-z-coordinate system relative to the x-y-z coordinate system. The s-t-z-coordinate system is relative to the "camera". The s-axis points towards the projection centre; the t-axis lies inside the image plane and is perpendicular to the s-axis and the z-axis.*

Next, the voxel's coordinates are transformed into another, temporary, coordinate system. This coordinate system has the same axis layout as the volume's coordinate system, but is rotated around the z-axis so that what formerly was the x-axis is now pointing towards the camera. This axis is then called *s*. The former y-axis rotates equivalently and stays perpendicular to the other axes. After the rotation it is denoted the t-axis. The z-axis stays in its original position. After the rotation, the coordinate system is still Cartesian and right-handed. Figure 3.14 shows the relation between the two coordinate systems. The advantage here is that the s-axis is parallel to the principal ray of the camera. It is pointing directly towards the image centre. Thus the s-axis is perpendicular to the image plane and the t- and z-axes are parallel to the image plane. Furthermore, the t-axis is parallel to the u-axis and the z-axis parallel to the v-axis of the image plane. This is the first step of our

transformation from volume coordinates to image coordinates. Given a voxel's coordinates x and y and the angle θ at which the image was taken, they can be transformed to s and t in the following way [3; 8, p. 386; 5]:

$$\begin{aligned} s &= x\cos(\theta) + y\sin(\theta) \\ t &= -x\sin(\theta) + y\cos(\theta) \end{aligned} \tag{3.26}$$

Afterwards the u-offset, which is the offset of the rotation axis from the image centre, is added to the t-coordinate, and the z-offset, which is unique to each projection, is added to the z-coordinate.

$$\begin{aligned} t &= t + u_{\text{offset}} \\ z &= z + z_{\text{offset}} \end{aligned} \tag{3.27}$$

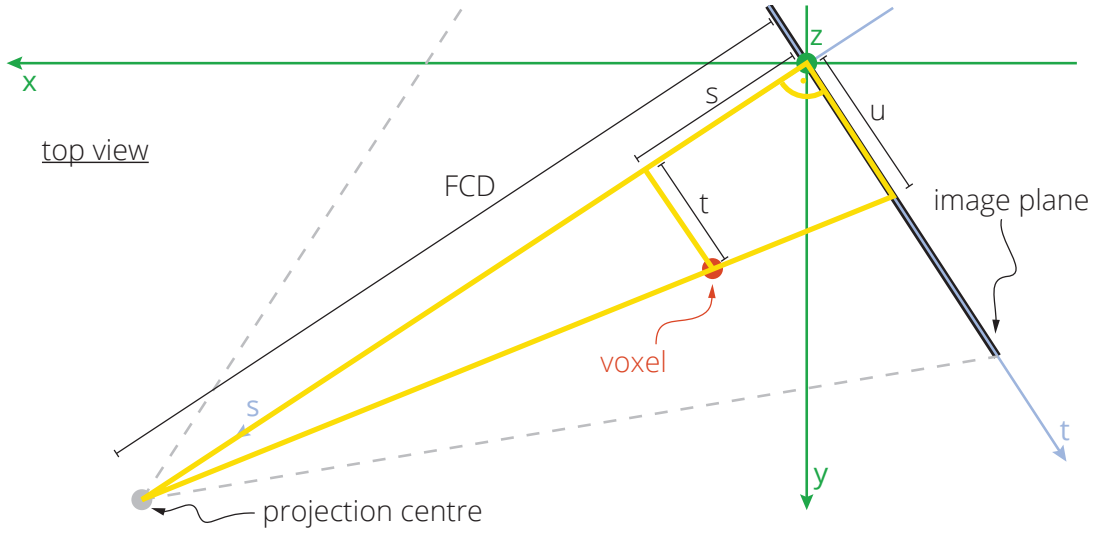


Figure 3.15: Calculation of the u-coordinate from the s-coordinate, the t-coordinate and the focus-centre-distance using the intercept theorem

The next step is to transform s, t and z to the image plane's coordinates u and v. If the s-coordinate were zero, meaning the voxel lay directly inside the t-z-plane, or if the projection type were not perspective but parallel instead, the u-coordinate would be equal to the t-coordinate and the v-coordinate equal to the negated z-coordinate. However, since we are dealing with a perspective projection, voxels that lie not on the s-t-plane but in front of it or behind it are projected onto the image plane along the rays originating from the projection

centre. This can be calculated by using the s-coordinate and the focus centre distance FCD along with the intercept theorem, as shown in equations 3.28 and 3.29 [3; 5; 8, p. 386]. The orange triangle in figure 3.15 illustrates this for the calculation of the u-coordinate.

$$u = \frac{tFCD}{FCD - s} \tag{3.28}$$

$$v = \frac{zFCD}{FCD - s} \tag{3.29}$$

This gives us the exact u and v coordinates at which we have to take a sample from the image. This is a floating point coordinate, so it probably is situated somewhere in-between the discrete pixels of the image. One option here is to just take a nearest-neighbour sample. We choose, however, to perform a bi-linear interpolation. This means the surrounding pixels' values are linearly averaged on both axes.

Before this is done, we check if the calculated coordinate is actually inside the image. Not all portions of the volume are actually captured on every image. If the coordinate is outside the image, then the algorithm needs to stop for this voxel and can proceed with the next one. In programming logic, not stopping at this point and trying to access non-existent values in memory may even cause unrecoverable errors.

After confirming that the calculated coordinate is located inside the image, we can proceed with the bi-linear interpolation. Therefore we calculate the coordinates of the four surrounding pixels by rounding the u and v coordinates up or down.

$$\begin{aligned} u_0 &= \lfloor u \rfloor \\ u_1 &= \lceil u \rceil \\ v_0 &= \lfloor v \rfloor \\ v_1 &= \lceil v \rceil \end{aligned} \tag{3.30}$$

Here u_0 is the u coordinate of the closest pixel to the left and u_1 the u coordinate of the closest pixel to the top of the calculated point in the image. The values v_0 and v_1 have the same meaning with respect to the v axis. The ratios between the pixels on the u and the v, denoted f_u and f_v, dimensions are then obtained by:

$$\begin{aligned} f_u &= u - u_0 \\ f_v &= v - v_0 \end{aligned} \tag{3.31}$$

Now, before accessing the pixels in the image, these coordinates have to be transformed to indices. These transformations are described in section 3.2.4. Next, an intensity value a can be interpolated, and a weight w can be calculated [9, p. 105; 8, pp. 384, 386; 5].

$$\begin{aligned} w &= \frac{FCD^2}{(FCD - s)^2} \\ a &= (1 - f_v)\left((1 - f_u)\, I_3^{u_0,v_0} + f_u I_3^{u_1,v_0}\right) + f_v\left((1 - f_u)\, I_3^{u_0,v_1} + f_u I_3^{u_1,v_1}\right) \end{aligned} \tag{3.32}$$

The term $FCD - s$ is the distance from the projection centre to the point that is the projection of the current voxel onto the principal ray [8, p. 384]. Thus, the weight w is the squared ratio of the focus centre distance by the distance between the projection centre and the plane perpendicular to the detector, in which the voxel is located. As a result, the weight w is relative to the reciprocal distance of the voxel to the source [8, p. 386]. Therefore, this weight is *one* for all voxels that lie inside the virtual detector plane, smaller than *one* for all voxels that lie in front of the virtual detector plane, i.e. between the projection centre and the detector plane, and greater than *one* for all voxels that lie behind the virtual detector plane.

In the end, the value a is added to the corresponding voxel, weighted by the weight w. Here V denotes the volume.

$$V^{x,y,z} = V^{x,y,z} + wa \tag{3.33}$$

3.2.4 Coordinate Transformations

In the algorithm, several transformations of voxel coordinates to voxel indices and of pixel coordinates to pixel indices are necessary. These transformations will be explained in this section.

Volume First, let's assume there is no region of interest and the whole volume will be reconstructed. In this case, the volume bounds are defined as given in equation 3.34, where r is the radius of the reconstructable cylinder (compare section 3.2.2) and r_v is the

height of the images.

$$\begin{aligned} x &\in \{0, \dots, r-1\} \\ y &\in \{0, \dots, r-1\} \\ z &\in \{0, \dots, r_v-1\} \end{aligned} \tag{3.34}$$

However, it is possible to define a region of interest. This region of interest is defined in a relative manner by the six floating point values $x_{\text{from}_f}, x_{\text{to}_f}, y_{\text{from}_f}, y_{\text{to}_f}, z_{\text{from}_f}, z_{\text{to}_f} \in \mathbb{R}$ which are all in the interval $[0, 1]$. Furthermore $x_{\text{from}_f} < x_{\text{to}_f}, y_{\text{from}_f} < y_{\text{to}_f}$ and $z_{\text{from}_f} < z_{\text{to}_f}$ is required. Here 0 denotes the minimum value and 1 the maximum value of the corresponding dimension. Since these bounds are relative, they are interchangeable among instances of the same data set with different resolutions. These relative bounds are then converted to absolute, integer bounds:

$$\begin{aligned} x_{\text{from}} &= \lfloor x_{\text{from}_f} r \rceil \\ x_{\text{to}} &= \lfloor x_{\text{to}_f} r \rceil \\ y_{\text{from}} &= \lfloor y_{\text{from}_f} r \rceil \\ y_{\text{to}} &= \lfloor y_{\text{to}_f} r \rceil \\ z_{\text{from}} &= \lfloor z_{\text{from}_f} r_v \rceil \\ z_{\text{to}} &= \lfloor z_{\text{to}_f} r_v \rceil \end{aligned} \tag{3.35}$$

Taking the region of interest into account, the actual dimensions of the volume are:

$$\begin{aligned} x_{\text{max}} &= x_{\text{to}} - x_{\text{from}} \\ y_{\text{max}} &= y_{\text{to}} - y_{\text{from}} \\ z_{\text{max}} &= z_{\text{to}} - z_{\text{from}} \end{aligned} \tag{3.36}$$

Given these bounds a volume index (x_i, y_i, z_i) can be converted to a volume coordinate (x_c, y_c, z_c) in the following way:

$$\begin{aligned} x_c &= x_i - \frac{x_{\text{max}}}{2} + x_{\text{from}} \\ y_c &= y_i - \frac{y_{\text{max}}}{2} + y_{\text{from}} \\ z_c &= z_i - \frac{z_{\text{max}}}{2} + z_{\text{from}} \end{aligned} \tag{3.37}$$

The same transformation can be reversed:

$$\begin{aligned} x_i &= x_c + \frac{x_{\text{max}}}{2} - x_{\text{from}} \\ y_i &= y_c + \frac{y_{\text{max}}}{2} - y_{\text{from}} \\ z_i &= z_c + \frac{z_{\text{max}}}{2} - z_{\text{from}} \end{aligned} \tag{3.38}$$

Image These transformation are similar for image coordinates and image indices. An image index (u_i, v_i) is transformed into an image coordinate (u_c, v_c) by:

$$\begin{aligned} u_c &= u_i + \frac{r_u}{2} \\ v_c &= -v_i + \frac{r_v}{2} \end{aligned} \tag{3.39}$$

Note that the v-coordinate is inverted. This is necessary because the v axis has opposite direction in the two coordinate systems. Also this transformation can be inverted easily:

$$\begin{aligned} u_i &= u_c - \frac{r_u}{2} \\ v_i &= -v_c - \frac{r_v}{2} \end{aligned} \tag{3.40}$$

Precomputation Some of the calculations in these equations can be precomputed. This is beneficial because they have to be performed for every voxel and thus have a considerable impact on execution times. The following precomputations are made:

$$\begin{aligned} x_{\text{pre}} &= \frac{x_{\text{max}}}{2} - x_{\text{from}} \\ y_{\text{pre}} &= \frac{y_{\text{max}}}{2} - y_{\text{from}} \\ z_{\text{pre}} &= \frac{z_{\text{max}}}{2} - z_{\text{from}} \\ u_{\text{pre}} &= \frac{r_u}{2} \\ v_{\text{pre}} &= \frac{r_v}{2} \end{aligned} \tag{3.41}$$

Then the formulas of equation 3.37 can be simplified as:

$$\begin{aligned} x_c &= x_i - x_{\text{pre}} \\ y_c &= y_i - y_{\text{pre}} \\ z_c &= z_i - z_{\text{pre}} \end{aligned} \tag{3.42}$$

The formulas of equation 3.38 turn into:

$$\begin{aligned} x_i &= x_c + x_{\text{pre}} \\ y_i &= y_c + y_{\text{pre}} \\ z_i &= z_c + z_{\text{pre}} \end{aligned} \tag{3.43}$$

Equivalently, equation 3.39 can be simplified as:

$$\begin{aligned} u_c &= u_i + u_{\text{pre}} \\ v_c &= -v_i + v_{\text{pre}} \end{aligned} \tag{3.44}$$

As can equation 3.40:

$$\begin{aligned} u_i &= u_c - u_{\text{pre}} \\ v_i &= -v_c - v_{\text{pre}} \end{aligned} \tag{3.45}$$

These simplifications spare a fair amount of execution cycles, reducing each transformation to only additions and subtractions.

4 Implementation

We started with a basic implementation of the algorithm on the CPU. At first, some decisions had to be made about the structure of the implementation, dependent on which goals should be met. These included:

- Should all images be loaded at once and kept in the main memory?
- How should the volume be stored in memory?
- Should the entire volume be kept in the main memory the whole time?
- Should the algorithm loop *for each image over all voxels* or for *each voxel over all images*?

Furthermore, some questions concerning practical use had to be answered:

- How should the user interact with the program, i.e. which form of interface should be provided?
- How should the input of the data work, i.e. how should the data necessary for the reconstruction be supplied to the program by the user?

4.1 Algorithm Core

When handling large amounts of data it is always necessary to decide which constraints are acceptable and which are not. The assumption that we made is that main memory is somewhat limited but extensible quite easily. This means that no memory should be wasted, but that there is also no reason to compromise on performance in order to reduce

memory demand. Furthermore, the memory demand for the reconstruction volume is, in most typical scenarios, within the realm of what can be handled with a reasonable financial expenditure.

Loop Order Before we have a closer look at memory utilisation, we should think about the structure of the reconstruction loop, i.e. the loop order. We do know that there are basically two loops necessary: the one which loops over all voxels (being actually three encapsulated loops, see lines 12, 13 and 22 in Algorithm 3) and the loop over all images (see line 3 in Algorithm 3). This results in two options, both with different constraints:

1. **Iterate over all images and then per image over all voxels**

    - Subsequent memory requests always happen within the same image, i.e. the memory addresses are in close proximity. This takes advantage of the processing unit's caching capabilities. Especially when memory access does not happen byte-wise, as is the case with most modern architectures, but in batches of one or multiple words that are fetched in one access and then remain in one of the much faster caches, this is very important for performance.

    - Also the voxels are accessed sequentially, offering good performance in this regard as well, due to the aforementioned reasons.

    - It is sufficient to store only one image in the memory at a time.

    - The algorithm could theoretically act as an online-algorithm, meaning while the scanning process is still in action, the reconstruction can already begin. As soon as the next image arrives, it is used for reconstruction. This way the reconstruction could more or less run in parallel to the scanning itself.

2. **Iterate over all voxels and then per voxel over all images**

    - Because the algorithm constantly accesses data from different images, memory access happens in an erratic, non-sequential manner. This completely counters the established caching mechanisms previously described.

- Only one voxel is accessed at a time; in total each voxel is finished before the next voxel is dealt with. From a memory access point of view this is even faster than the sequential but repeated voxel access.
- All images have to be kept in memory at all times.
- The algorithm cannot work in an online fashion. All images have to be present from the start.

Judging from the aspects listed above, it is quite apparent that the first option is the better choice since its advantages clearly outweigh its disadvantages. The only thing that could theoretically be slightly faster with option two is the voxel access. However, the difference is probably neglectable, especially considering the other drawbacks.

Image Loading As a result of choosing option one, every image is only required once. Therefore, a new question arises: Should the images be loaded into the main memory prior to the reconstruction procedure, meaning that all of them are present in the memory at the same time, or should they be loaded one by one as they are needed for the reconstruction, meaning that only one image is kept in the main memory at a time? In this case, the advantages and disadvantages are less numerous, yet that does not mean the decision is a simpler one:

1. **Load all images into memory**
   - This consumes more main memory.
   - No delays due to file operations can happen during the reconstruction.
2. **Load only one image at a time**
   - Memory for a maximum of two images at one time is required, assuming that one image is used for reconstruction while another one is being prefetched in the background.
   - Delays due to file operations and preprocessing during the reconstruction are possible.

First of all, what we can say is that the total amount of work stays the same, given either one of the two options. In both cases each image has to be read from disk once and has to be preprocessed once. The difference mainly exists in *when* this happens, and as a result of this, in the amount of memory required. Having tried both possibilities, we came to the conclusion that reading only the image currently needed is the better option. We implemented this in a way that while one image is used for the reconstruction, the next one is simultaneously read from disk and preprocessed in another thread, as can be seen in Figure 4.1. This successfully hides latencies occurring due to disk operations, and

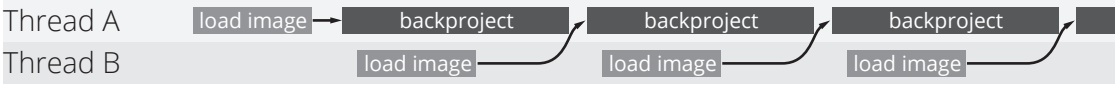


Figure 4.1: *Visualisation of the multi-thread approach that hides the latencies for loading the projections on the CPU*

prevents them from having an impact on the reconstruction performance. This way, the reconstruction takes essentially the same amount of time as with the preloading. If we take the time that option one needs for loading the images prior to the reconstruction process into consideration, option two is, in total, even faster. This is illustrated in Figure 4.2. Consequently, we decided to not preload the images. This also has the advantage

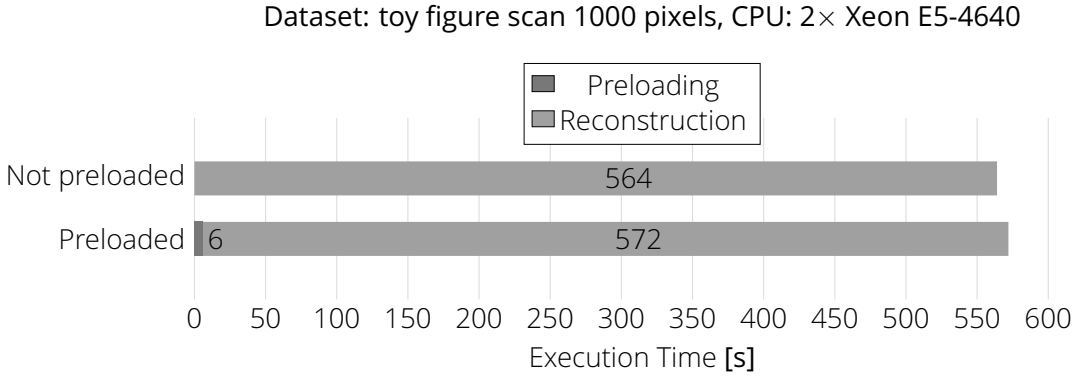


Figure 4.2: *Comparison of the execution times with and without preloading of the images to the main memory prior to the reconstruction*

of sparing a lot of main memory. Considering that they are converted to a 32 bit floating point representation, the size of all images together can easily surpass the size of the 3D-volume. As an example, in the case of a dataset with 1199 images and a resolution of 875

$\times$ 1000 pixels each, the amount of memory required for storage would be 3.9 GB. The volume itself only requires 2.9 GB. Particularly on systems with less memory, this enables the reconstruction of far bigger volumes than would be possible when keeping all images inside the memory.

Volume The way we designed our implementation, the volume itself is kept inside the main memory during the whole reconstruction process. This is the most straightforward way of implementation and ensures optimal reconstruction performance. However, it would be quite simple to modify the algorithm later on, in a way that the volume is reconstructed part-wise, without requiring substantial changes to the code. We are already providing the functionality for reconstructing only a region of interest of the volume. Using the same approach, it could automatically be split into multiple parts that then are reconstructed on their own and saved to non-volatile memory. This way the reconstruction of volumes of nearly arbitrary size would be possible independent of main memory capacity. We use a similar approach for the reconstruction on the GPU, which will be explained in chapter 5. However, this would come with some limitations. It would, for example, no longer be possible to view the volume. In this case, there are other applications that also allow out-of-core rendering of volumes which do not entirely fit into main memory.

Index Order What also has to be considered is the order in which we iterate through the different dimensions, i.e. the individual voxels. By this we mean the order of the loops in the lines 12, 13 and 22 of Algorithm 3. Theoretically, since there are three loops, there were $3!$ or 6 possible ways to order these loops. However, either the orderings x-y-z (called z-fastest) or z-y-x (called x-fastest) are commonly used. For example, assuming z-fastest ordering the voxels would be accessed in the order $(0,0,0), (0,0,1), (0,0,2) \ldots$, and assuming x-fastest ordering the order would be $(0,0,0), (1,0,0), (2,0,0) \ldots$.

Now the question is why that would make a difference. There is one small detail that does matter, which is the condition in line 14. All voxels with the same z-coordinate share the result of this condition. In other words: Solely the x- and y-coordinates of a voxel determine if it is inside the reconstructable cylinder or not. So all voxels inside one row along the z-dimension have either to be computed or not to be computed. Now, if the z-loop is the innermost loop, then, given that the condition is false, the whole inner part, including the z-

loop, can be skipped. If the z-loop was one of the outer loops, the condition would have to be moved to the very inside of the three loops. As a result, it would have to be evaluated for each individual voxel, causing much more computational overhead. Therefore, the z-fastest index ordering is optimal in this case.

Another thing that must be considered is that the index order of the voxels in the memory should be identical to the order they are accessed in, due to the caching mechanisms that have been previously described. For the CPU implementation a z-fastest index ordering was used, while x-fastest ordering is optimal for the GPU implementation. Therefore, we created our own volume class that can switch between either x-fastest or z-fastest ordering. It internally holds a contiguous C-style array and allows per-element access similar to the *vector* class coming with the C++ standard libraries. The projection from a three-dimensional to a one-dimensional index is handled by this class. Switching from x-fastest to z-fastest mode will change how the contained array is interpreted. All member functions of the class will treat it according to the index ordering that is currently set. Such member functions, for example, provide the functionality for obtaining a cross section of the volume or for writing the volume to disk. However, if the index order is not known at compile time, the code cannot be optimised to the same degree as otherwise. To counteract this, we also provide the functionality for obtaining a pointer to a single element, element row, or element plane of the volume. These pointers can then be used to iterate through the volume in a very efficient way. However, the user has to take responsibility for preventing memory access violations and other possible problems themselves. The volume class also is templated, allowing the loading of volumes of different data types.

4.2 Pseudocode

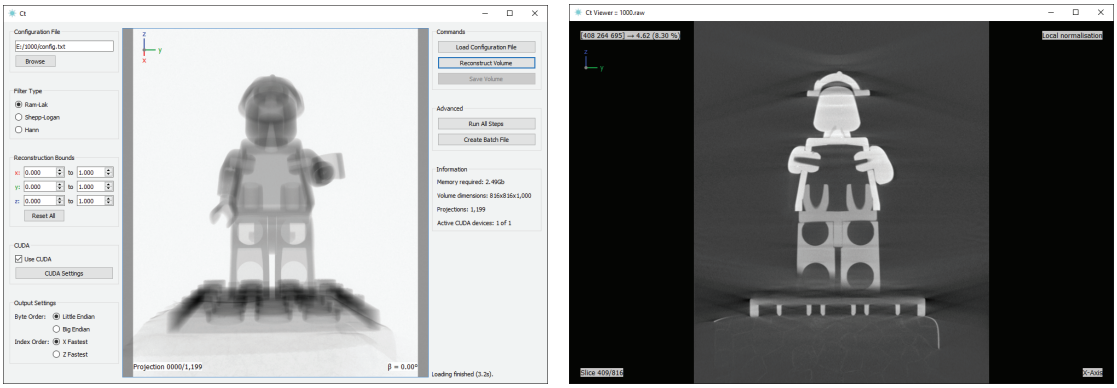
Algorithm 2 shows the preprocessing and Algorithm 3 the backprojection in a simplified form as pseudocode. Although these are capable of giving a general idea of how the implementation of the algorithm works, they lack the in-depth descriptions of all the subroutines and do not have the complexity of the actual implementation. Describing everything in pseudocode in detail is not feasible and shall not be done at this point. However, the actual C++ code comes attached to this work and will allow interested readers to examine it as extensively as desired.

Data: image
Result: Preprocessed image

convert **image** to 32bit float;
scale **image**'s data range to **[0, 1]**;
normalise **image** with regard to x-ray source's base intensity;
apply logarithmic scaling to **image**;
apply the Feldkamp weights to **image**;
convert **image** to frequency domain as **spectrum**;
apply window filter to **spectrum**;
convert spectrum back to spatial domain as **image**;

Algorithm 2: *Preprocessing*

4.3 Interface



Ct Ct Viewer

Figure 4.3: *Screenshots of the graphical user interface of the CT reconstruction software (left) and the viewer (right)*

We decided to add a graphical user interface because it simplifies the setup process of the reconstruction significantly for the user. Moreover, errors that the user might have undergone can be diagnosed more easily and feedback is provided during the reconstruction process. There are, however, some situations when a command line interface is more useful. It enables potential users to still use the program, even if there is only a command line available, or if they want to execute reconstructions from a batch script. It thus was decided to provide a hybrid solution: If launched without any command line parameters, the program would bring up a graphical interface. If, however, command line parameters are

Data: The images and the capturing parameters
Result: Reconstructed CT volume

```
1  allocate volume and initialise it with 0;
2  radiusSquared = (FCD^2 * (imageWidth/2)^2)/(FCD^2 + (imageWidth/2)^2);
3  foreach projection do
4      if is first projection then
5          load and preprocess projection;
6      else
7          get preprocessed projection from prefetching thread;
8      if is not last projection then
9          launch thread that loads and preprocesses next projection (asynchronously);
10     calculate sine and cosine of projection's angle;
11     obtain pointer to volume as volumePtr;
12     foreach x in volume coordinates do
13         foreach y in volume coordinates do
14             if x^2 + y^2 > radiusSquared then
15                 advance volumePtr by zSize and continue with next iteration;
16             t = (−x * sine + y * cosine) + uOffset;
17             s = x * cosine + y * sine;
18             u = (t * FCD)/(FCD − s);
19             if u is outside image bounds then
20                 advance volumePtr by zSize and continue with next iteration;
21             w = (FCD/(FCD − s))^2;
22             foreach z in volume coordinates do
23                 v = ((z + heightOffset) * FCD)/(FCD − s);
24                 if v is inside image bounds then
25                     convert u and v to indices;
26                     get intensity values of four nearest neighbours of (u, v);
27                     interpolate the four intensity values bi-linearly;
28                     weight resulting value with w;
29                     add weighted value to voxel at volumePtr;
30                     advance volumePtr by 1;
```

Algorithm 3: Backprojection

passed when launching the program, it solely runs as a console application. Also transitioning between the two interface types was made possible. A user can setup a reconstruction via the graphical user interface and then have the program automatically generate a batch script containing a call that executes that exact reconstruction with the settings they specified. Figure 4.3 shows a screenshot of the interfaces of the CT reconstruction software and the viewer application that is also provided.

The following settings can be made via both the graphical and the command line interface:

- Path to the configuration file
- Filter type for the reconstruction
  - Ram-Lak
  - Shepp-Logan
  - Hann
- Region of interest for the reconstruction
  - One upper and one lower bound per dimension, in between $[0, 1]$
- Choice between CPU and GPU reconstruction
- Choice between CPU and GPU preprocessing when reconstructing on the GPU
- GPUs to be utilised for the reconstruction
- GPU weights α and β as explained in section 5.3.1
- Amount of VRAM to be kept free for other applications
- Byte order and index order of saved files
  - Little endian or big endian
  - Z fastest or x fastest

Listing 4.1: Layout of the Configuration Files

```
1  .
2  0.06554696
3  ccw
4  0
5  74.9996
6  0.02749
7
8  Test00001.tif   0               0
9  Test00002.tif   0.300250209     0
10 Test00003.tif   0.600500417     0
11 ( ... )
```

4.4 Data Input: Configuration Files

Each reconstruction may require up to about 1000 image files or more. These must be provided to the program in some capacity. When providing a graphical interface, it might be possible to include a file selection dialog for that purpose. However, this is not possible on the command line. Furthermore, although this is sufficient for providing the files, it is not possible to include the parameters (angle, height offset) for them. For these reasons, we decided to make use of configuration files. These contain the parameters and the paths of the image files. This comprises only the fixed parameters of the scan geometry and not the settings that influence the result of the reconstruction and might be subjected to change. This way it is possible to provide a scan as a folder of files that can be distributed easily. Anyone can then do a reconstruction of this scan by simply loading the included configuration file without having to worry about its parameters. Such a configuration file could, for example, also be automatically generated by the scanner software.

Listing 4.1 shows the layout of such a configuration file. Line 1 contains the *path* to the folder where the images are located. This path can be absolute or relative to the location of the configuration file. In the example, the point character means that the images are located in the same directory as the configuration file. Line 2 contains the *pixel size*, being the width or height of one pixel, with the assumption of square pixels. Line 3 then contains the *rotation direction* which must be either of the strings *cw* or *ccw* for clockwise or counter-clockwise respectively. This describes the rotation of the camera relatively to the scanned object. In line 4 the *u-offset* is specified, meaning how far the rotation axis is off of the horizontal centre of the image. Line 5 contains the *focus-centre-distance*, and line 6 the x-

ray source's *base intensity*. Then there follows an empty line. Afterwards a list begins that contains all of the images and the per-image parameters. This is the *name of the file*, the *angle* at which it was captured in degrees and its *z-offset* separated by tab characters.

4.5 About the Program

Our implementation was realised in C++ and CUDA. This implementation enables the user to display the sinogram and to scroll through the images one by one, as well as displaying cross-sections of the final reconstruction. The volume can be saved as a raw binary file with different encodings. Along with the binary file, two sidecar files with identical name are saved, one *TXT* file and one *VGI* file. The *TXT* file contains information about the reconstruction and the volume in a human readable format. The *VGI* file makes it possible to load the volume in other applications as well, such as *VGStudio*.

Furthermore, a viewer application was also created that can load raw volume files of many different data types and encodings, display them in the form of cross sections and print the absolute and relative data values of the voxels. This viewer uses the aforementioned *TXT* file to automatically obtain the volume parameters, as to require no further user input. However, it is possible to load virtually any raw volume with said viewer. If no meta-information about the volume is found, a dialog is displayed that makes it possible for the user to enter it manually. Here a wide variety of data types and encodings is supported.

The application can also be used in command line mode. The graphic interface also allows the generation of a batch file which will execute the reconstruction in command line mode using the settings currently selected in the graphic user interface.

For the image handling the *OpenCV* library was utilised. *Qt* was used for creating the graphic user interface. For the parallelisation, as described in chapter 5, *OpenMP* (CPU) and *CUDA* (GPU), including *CUFFT*, were used.

The software is open source, and the code repository as well as precompiled binaries for Windows and Linux are publicly available at `https://bitbucket.org/bastian_weber/ct`. It is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International Public License. For licensing details see `http://creativecommons.org/licenses/`

by-nc-sa/4.0/legalcode. A current revision of the code and the program is provided together with this work. However, the online repository should always be preferred for the most recent state. Releases, i.e. revisions that are supposed to be stable, are marked in the revision tree.

5 Parallelisation

One of our goals was to make the algorithm as fast as we possibly could. Consequently, we tried to use all the parallelisation capabilities of modern hardware. On the one hand this was traditional multi-threading on multi-core CPUs. On the other hand, it was parallelisation on the GPU, referred to as general purpose computation on graphics processing units (GPGPU), or just GPU processing.

GPUs are specialised for compute-intensive, highly parallel computation. They are especially good at problems which can be expressed as data-parallel computations, meaning problems where the same procedure is executed for many data elements in parallel, as is the case with CT reconstruction. Graphics processors devote more transistors to data processing than to data caching and flow control as is the case with traditional CPUs, putting them at an advantage for said applications. [12]

5.1 CPU

For the parallelisation on the CPU we used the OpenMP library. It allows parallelisation of loop-like program structures in a very straightforward manner, which makes it perfect for our application, and provides different options for scheduling iterations over the different cores/threads. Furthermore, it is already included in some of the most popular C++ compilers, for example the Microsoft Visual C++ or the GNU C++ compiler, which makes integration relatively simple.

When trying to parallelise the algorithm, we were again confronted with some choices or decisions that had to be made. The first one was how to distribute the workload across multiple threads. There were two options:

1. **Distribute the images across multiple threads and let each thread simultaneously do the reconstruction of one image**

    - Data races could become a problem if multiple threads try to modify the value of the same voxel. To produce a correct result, it needs to be ensured that only one thread at a time tries to modify one voxel, or potential imprecisions had to be accepted. It is a difficult question to answer whether these were neglectable or not.

    - The order of the additions of the values from different images to one voxel might vary. Floating point addition is not associative [23], thus the results could be different each time the reconstruction is performed.

    - Multiple images are read and kept in memory at the same time. This consumes more memory.

    - Scheduling overhead is very low because the amount of processing done by each thread is rather large before a new task has to be assigned to it.

    - Work balance between threads is even.

2. **For each image, split the volume into parts and let each thread reconstruct one of these parts**

    - No concurrent access of the data of one voxel happens. Thread safety requires no further measures.

    - The additions to one voxel are always performed in the same order, hence no associativity is required.

    - Only one image has to be kept in memory at a time.

    - Scheduling overhead is larger than with option one. For each individual image multiple task assignments happen to each thread.

    - Work balance is not necessarily even. It depends on how the volume is split, and how the parts are spread across threads.

Although option one appears to have a few more advantages, especially when it comes to work balance and thread scheduling, we decided to go with option two mostly because of the fact that thread safety and data races are not a problem here. Trying to synchronise thread's data accesses would result in a greatly degenerated performance. Since there is no reasonable way of doing this, the only option would be to accept the potential inaccuracies. However, it is very hard to assess how much of an impact these might have in practical use, considering that they are greatly dependent on chance. Furthermore, this would mean that our result would no longer be deterministic.

Apart from that, option two is generally not much worse than option one. To ensure equal workloads, we essentially split the volume into slices of height one along the x-dimension, as shown in Figure 5.1, which are then dynamically scheduled across threads. As soon

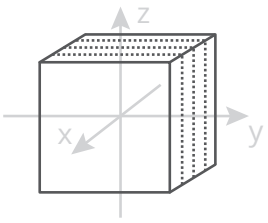


Figure 5.1: *Illustration of how the volume is split into slices, which are then reconstructed by individual CPU threads. Each slice has a depth of exactly one voxel along the x-axis. These slices are then assigned to the CPU threads alternatingly.*

as one thread finishes working on one slice, it is assigned a new one. The different slices require different computational effort, depending on the amount of voxels contained that actually have to be computed (i.e. that are inside the reconstructable cylinder). The dynamic scheduling, however, ensures that the workload is still balanced across threads.

Now the question may arise of why we do not split the volume along the z-dimension. Using this dimension, the workload would be equal for each slice because the amount of voxels that have to be computed is the same for each x-y-plane, regardless of the z-coordinate. Still, this would not be a good choice, because the z-loop is the innermost loop. Parallelising this loop would yield the maximum scheduling overhead. Only one voxel would be assigned to each thread at a time. However, it should always be the goal to assign tasks, as comprehensive as possible, to each thread at a time. Therefore, we parallelised

the outermost loop, i.e. the x-loop, thus minimising the scheduling overhead as much as possible.

This way the scheduling overhead is slightly bigger than with option one, because for each image multiple tasks have to be assigned to the different threads. However, we believe that the practical impact of this is neglectable. We also ran some prototypic tests, the result of

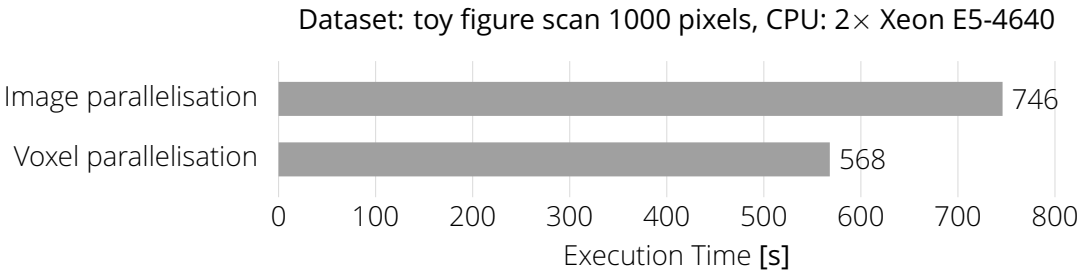


Figure 5.2: Comparison between the execution times resulting from parallelising the image versus the voxel loop

which can be seen in Figure 5.2. As it turned out, the parallelisation across multiple images is significantly slower than parallelisation on the voxel side. Trying to find an explanation for this behaviour turns out to be mere guesswork. One reason could be that when processing multiple images in parallel, 32 in this case, the storage unit must handle many simultaneous requests. Caching issues could also be a reason that causes this decrease in performance. However, since they all have their individual caches, the cores of a CPU are supposedly quite independent when it comes to memory access. So having them access different locations in memory (the different images) should not be too much of a problem. Whatever the explanation might be, this confirms that our choice to parallelise the processing per image was the right one. Another positive side-effect is that we save the memory that would otherwise be required for the additional images.

Preprocessing was not parallelised because it caused the performance to be worse in some situations and did not yield any improvement. See section 6.2.2 for test results.

This concludes the CPU parallelisation. Due to the generally simple structure of main memory and main processor there were no principal changes to the structure of the algorithm necessary. Evaluation results regarding the achieved speedups can be found in section 6.

5.2 GPU

In contrast to the main memory and the CPU, the GPU is much more limited when it comes to memory. Current mainstream GPUs usually offer about four gigabytes of VRAM. Older models, mobile or low-budget ones might even come with merely two or one gigabyte, and the maximum that can be expected on a current GPU is about eight or twelve gigabytes. Furthermore, the VRAM on a GPU cannot be expanded, as is possible with the main memory of a computer system. This shows the difficulties when trying to implement a largely memory-dependent algorithm on the GPU.

The GPU can only access data that is stored inside the VRAM. It essentially acts like a second memory layer. Everything that should be put in the VRAM has to be in the RAM first. From there it can be uploaded to the VRAM and used for processing by the GPU. The result, i.e. the processed data, must then be downloaded to the RAM again to be used further. This up- and downloading can cause significant overhead, and it should be minimised as much as possible.

It becomes quite obvious that an approach is necessary that can adapt to the amount of video memory available and can perform the reconstruction part-by-part. This raises the following questions:

- How should the volume be stored on the GPU?
- How should the images be stored on the GPU?
- How should the whole algorithm be structured?

If we think about the volume, it must be quite clear that it can only be present in GPU memory one part at a time. These parts then must be reconstructed individually. Concerning the images, it would be desirable to have them all permanently stored in the video memory. However, this is not possible, due to their size. Taking a dataset comprising 1199 images of a resolution of 845×1000 pixels as an example, the memory required for storing these images in 32 bit floating point representation would be 3.9 GB. For most GPUs this is too much data, and it will be even more for larger datasets. Thus, storing all images in the VRAM at once is not feasible. Consequently, the volume parts must be reconstructed one at a time and the images must be uploaded and used for reconstruction one at a time.

Therefore, we split the volume into parts along the z-axis, as shown in Figure 5.3. Each

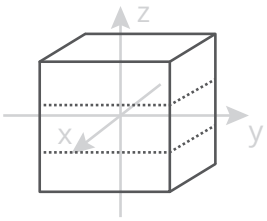


Figure 5.3: *Illustration of how the volume is split into smaller chunks, which can then be reconstructed on the GPU individually*

of these parts is reconstructed on the GPU individually. Afterwards, it is downloaded and added to the whole volume that lies inside main memory. Algorithm 4 illustrates this procedure.

This defines the general structure of the algorithm on the GPU. The reconstruction itself can still be performed the same way as on the CPU. The calculation for multiple voxels will be executed in parallel GPU threads, while the different images will be used one by one. For each image there is one reconstruction pass. As a result, after each reconstruction pass a new image must be uploaded to the GPU and preprocessed. This causes a latency, and in the meantime the reconstruction pauses. However, we wanted to utilise the GPU as best as possible. Therefore, we came up with a two-stream approach that runs reconstruction kernels alternatingly on two different GPU streams. While one stream is running the reconstruction for one image, the other stream asynchronously uploads the next image and preprocesses it. Then the roles of the two streams are switched (compare lines 8, 9 and 10 of Algorithm 4). This way we are able to hide the latencies between the reconstruction kernels to a certain degree. Figure 5.4 illustrates this approach.

Figure 5.4: *Visualisation of the multi-stream approach that hides the latencies for loading the projections on the GPU*

To be able to perform a host-to-device memcopy operation asynchronously, the host memory must be page locked [24; 12, section 3.2.4] (compare line 13 of Algorithm 4). Then the

copying can happen directly without involving the CPU. For this we used the *HostMem* class that is included in the CUDA module of OpenCV.

Image Preprocessing As mentioned, we perform the image preprocessing on the GPU as well. Another option would have been to keep it on the CPU side and perform it before the image is uploaded to the GPU. Practical testing showed that the performance is more or less identical for single-GPU execution and small images. However, performing the preprocessing on the GPU takes some of the workload off of the CPU in case of multiple GPUs involved. In this case the CPU would have to preprocess the images for each GPU; consequently, the CPU load would increase proportionally to the amount of GPUs involved. Figure 5.5 illustrates the difference in multi-GPU execution time between CPU and GPU preprocessing for a larger dataset. Multi-GPU parallelisation is explained in more detail in

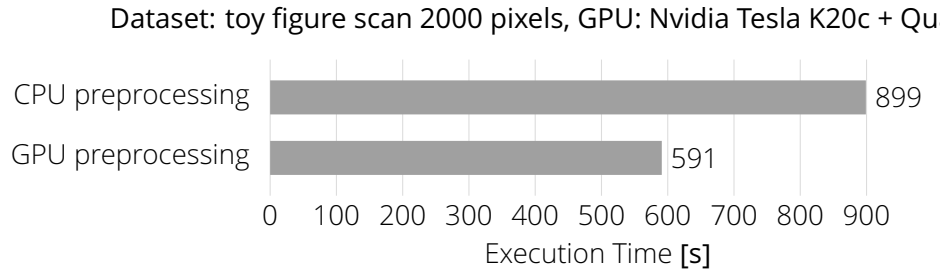


Figure 5.5: Comparison between the execution times resulting from CPU preprocessing versus GPU preprocessing

section 5.3. These results may vary for datasets of different sizes and other hardware configurations. For instance, on a system with a slow GPU but a fast CPU, CPU preprocessing may be advantageous. Therefore we provide the functionality to choose between GPU and CPU preprocessing in our application.

Algorithm 4 shows the structure of the GPU reconstruction algorithm in a much simplified form as pseudocode.

Index Order and Memory Coalescing Index ordering also plays a role on the GPU, probably even more than on the CPU. Memory coalescing means that larger portions of memory (e.g. multiple words) can be fetched from the global device memory in one access and are then present in a much faster cache. One warp, consisting of 32 threads on every current

Data: The images and the capturing parameters
Result: Reconstructed CT volume

```
1  allocate the volume in RAM;
2  allocate required data structures on the GPU side (e.g. images, FFT plan, etc.);
3  estimate maximal volume chunk size according to free video RAM;
4  iteratively try to allocate as large a chunk as possible;
5  foreach volume part do
6      set chunk to zero;
7      foreach projection do
8          swap stream with backgroundStream;
9          swap gpuImage with backgroundGpuImage;
10         swap plMemory with backgroundPlMemory;
11         precalculate sine and cosine of projection angle;
12         load projection from disk;
13         copy projection to page-locked memory plMemory;
14         upload plMemory to gpuImage in VRAM; preprocess gpuImage on the GPU;
15         synchronise backgroundStream;
16         run the reconstruction with gpuImage, sine, cosine and other capturing
             parameters on the GPU;
17     download chunk and add it to volume;
18 delete chunk from VRAM;
19 delete all allocated data structures on the GPU side;
```

Algorithm 4: *Reconstruction on the GPU*

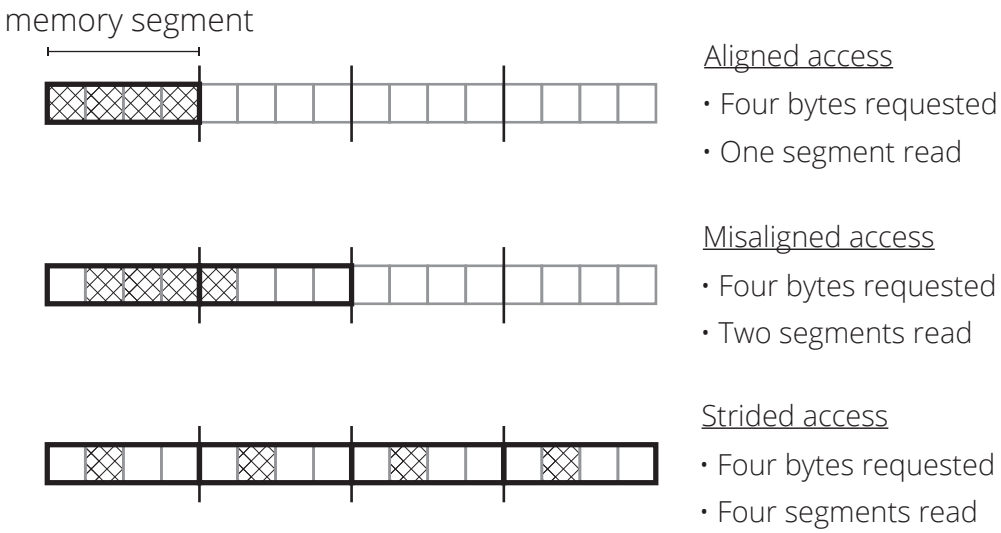


Figure 5.6: *Visualisation of memory alignment and its effects (from best to worst)*

Nvidia GPU, can then access the data from this cache without requiring additional accesses to global memory. Even more than that: Any block that runs on the same streaming multiprocessor has access to the very same L1 cache and can access this data from there as long as it is present. Data should therefore be stored in memory contiguously and aligned to assure that the amount of accesses to global memory is kept at a minimum. Memory alignment and memory coalescing are explained schematically in Figure 5.6 and Figure 5.7 [14], respectively. To ensure sequential access and to prevent strided access the index ordering of the volume's voxels in VRAM should match the index ordering of the threads, which is x-fastest [12, section 2.2]. We thus use x-fastest index ordering. [25, section 9.2.1; 26; 14]

Since accesses to the images happen more or less unpredictably during the reconstruction, these cannot be optimised with respect to memory coalescing. However, during the preprocessing, when all pixels of an image are accessed sequentially, memory coalescing is of importance as well.

In CUDA, both L1 cache and shared memory use the same hardware memory. Thus, it is possible to prioritise either L1 cache or shared memory by software. We use the command *cudaDeviceSetCacheConfig(cudaFuncCachePreferL1)* to prefer L1 cache, which is more important for us, as we do not use shared memory. As a result, there will be 48 Kb of L1 cache

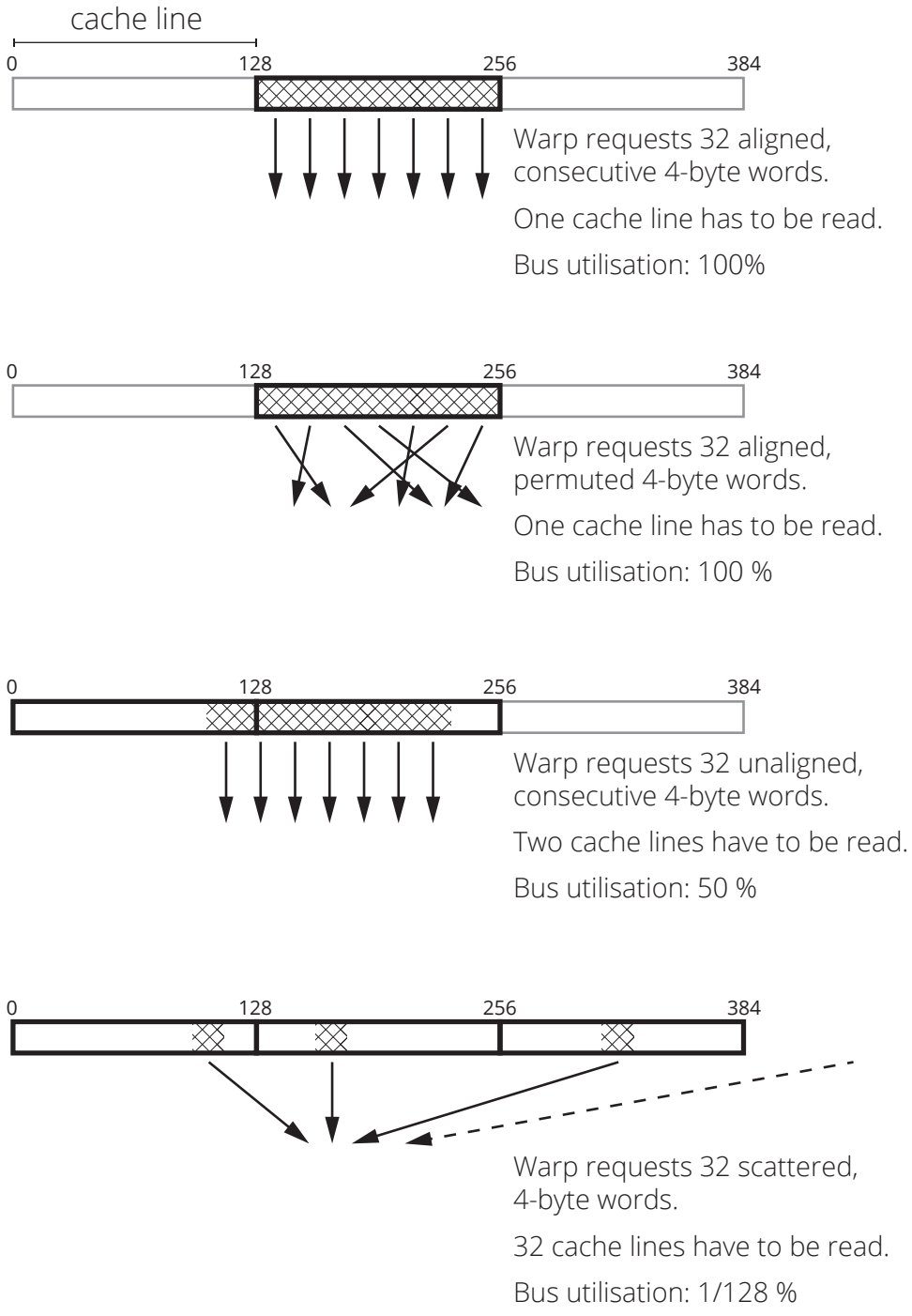


Figure 5.7: Visualisation of memory coalescing and its effects, from best to worst

and 16 Kb of shared memory, instead of 16 Kb L1 cache and 48 Kb shared memory, which is the default. [12, section G.3.1; 14]

Even though an x-fastest index ordering was used for the volume on the GPU side, contrary to the z-fastest index ordering used for the CPU implementation, this is not a problem due to the flexible volume class explained in section 4.1, that can switch between both ordering types. Without this, the volume would first have to be downloaded to a temporary location in the RAM, and afterwards the elements would need to be reordered to match the index ordering on the CPU side. This would require extra RAM and extra computation time. For allocation of the array structure holding the volume on the GPU the CUDA API function *cudaMalloc3D* was used, which takes care of memory alignment automatically [12, section 3.2.2].

GPU Memory Allocation For the volume, we require one contiguous piece of VRAM. The CUDA API provides functions for querying the amount of free memory. However, this doesn't necessarily mean that this memory also is contiguous. Therefore, just trying to allocate as much memory as is free will fail a majority of the time. Thus, we came up with an approach that tries to allocate memory for the volume iteratively. If the allocation fails, the amount of memory is reduced and allocation is tried again. This is repeated until it succeeds (see Algorithm 5).

Once memory for the volume has been allocated, it is kept until the whole reconstruction finishes. For the next volume part, the contents of the allocated VRAM chunk are downloaded to the main memory, it is reset to zero and then reused. The downloading can be performed as a one to one copy from VRAM to RAM that can be realised in one single *cudaMemcpy3D* invocation, since the memory layout of our volume is identical in main memory and video memory. Only the last volume part may be smaller than than the allocated chunk. However, this is not a problem. Since our memory alignment in main memory and video memory match, we can simply download a portion of the allocated chunk. This way, memory on the GPU has only to be allocated once in the beginning and to be freed once at the very end, providing maximal efficiency.

Data: sliceCnt, decreaseSliceStep
Result: Pointer to allocated volume

```
1  repeat
2      try to allocate memory for sliceCnt amount of volume slices;
3      if not successful then
4          if decreaseSliceStep < sliceCnt AND decreaseSliceStep > 0 then
5              sliceCnt = sliceCnt − decreaseSliceStep;
6              cudaGetLastError();          // recover from non-sticky cuda errors
7          else
8              break;
9  until allocation succeeded or sticky cuda errors occurred;
10 if allocation did not succeed then
11     terminate algorithm;
```

Algorithm 5: *Allocation of the memory on the GPU*

Blocks and Threads The block size that was chosen for the execution of the reconstruction kernel is $16 \times 16 \times 1$. Through systematic testing we could identify this as the configuration that delivered the best performance on the available hardware. However, this might differ for other devices. The optimal block size is hard to predict, and finding it is not a simple task. It is also dependent on the hardware used [25, section 10.5]. Nevertheless there are some comprehensible reasons why one block size might be faster than another, especially concerning the "shape" of the blocks, meaning their extensions in x-, y-, and z-dimension.

One of these reasons is memory coalescing. The x-, y-, and z-coordinates of a voxel are calculated from its position in the grid. The position in the grid is calculated from the block coordinate and the thread coordinate as shown in equation 5.1.

$$\begin{aligned} x &= x_{\text{thread}} + x_{\text{block}} x_{\text{dim}} \\ y &= y_{\text{thread}} + y_{\text{block}} y_{\text{dim}} \\ z &= z_{\text{thread}} + z_{\text{block}} z_{\text{dim}} \end{aligned} \tag{5.1}$$

Here x, y and z are the resulting coordinates, while x_{thread}, y_{thread} and z_{thread} are the coordinates of the thread in the block and x_{block}, y_{block} and z_{block} are the coordinates of the block

in the grid. The variables x_{dim}, y_{dim} and z_{dim} are the dimensions of one block. Inside memory, the volume is stored with an x-fastest ordering. Consequently, the threads grouped in one block ought to access voxels that lie next to each other on the x-dimension, since all threads of one block are executed on the same streaming multiprocessor, having access to the same L1 cache. Usually one cache line is 128 bits long, or 16 bytes [25, section 9.2.1; 14]. One voxel has the size of four bytes (32 bits). Thus, the amount of contiguous voxels on the x-dimension being worked on by the threads of one block should be at least four and should be divisible by four. In our case this is 16, which is four times four and will require four memory accesses.

Having two-dimensional blocks with equal size on the x- and y-dimensions resulted in better performance than having just one-dimensional blocks. It is difficult to say why this is the case. It is probably unrelated to the volume accesses, but it could be related to the image accesses. Two voxels which lie next to each other on the x-axis are likely to be projected next to each other in an image where the camera is looking along the y-axis, while two voxels lying next to each other on the y-axis are likely to be projected next to each other in an image where the camera is looking along the x-axis. This means these voxels are likely to be in the same image row and quite close to each other. From a caching point of view, this is advantageous. The images are saved in a row-fastest manner. This means the image rows are contiguous in memory. Reading multiple pixels that lie next to each other in one row has a certain chance of being performed in one memory transaction, whereas accessing multiple pixels that lie next to each other in one column always requires strided memory access, which is slow [25, section 9.2.1]. This is the reason why it is advisable to choose the z-size of the blocks to be *one*.

Now an objection may arise, saying that having one-dimensional blocks where all threads lie along the x-dimension would also fulfill this criteria. If viewed along the x-axis, these voxels would probably be projected onto the same point on the image. However, what has to be taken into account is that we are not working with parallel but with perspective projection. This means voxels that lie apart further in depth will be projected onto different image rows, as shown in Figure 5.8. The further they lie apart, the more rows they will span in the image. From a memory access point of view this is not optimal and will result in more strided accesses.

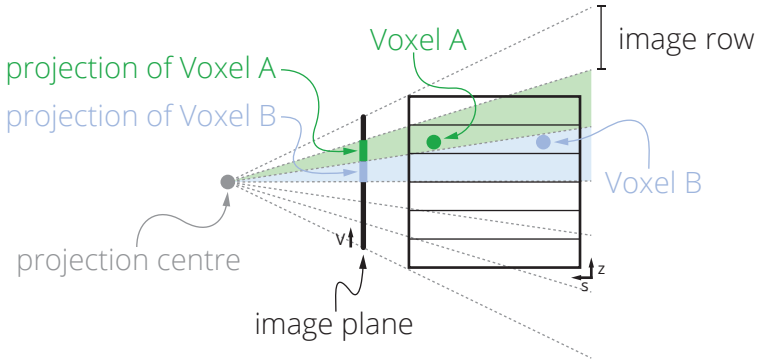


Figure 5.8: The illustration demonstrates how voxels lying in the same x-y-plane may be projected onto different image rows.

Another guideline for the choice of block size is that the total number of threads per block should always be a multiple of 32 because one warp consists of 32 blocks [25, section 9.2.1.2]. Otherwise, at the end of each block, there would be warps which are comprised of less than 32 threads. As a result, a part of the computational unit would be idle and device occupancy would not be optimal. With the block size we chose there are 256 threads in total, which is a multiple of 32. [27]

5.3 Multi-GPU

Parallelising the GPU implementation across multiple GPUs adds another layer of complexity. The total work will now have to be split into parts, which are then distributed across the GPUs. These GPUs might be different models, they might have different computing power and different amounts of memory, and thus they might require different amounts of time for the same amount of work.

Assigning Work The question we have to ask ourselves is: How do the GPUs obtain their piece of work? One option would be to let them "grasp" as much as they can handle, which would be bound by the size of their memory. It turns out that this is not a very good idea. In case of a small volume, one GPU would take everything for itself while there would nothing be left for the other ones. This is not suitable for achieving a balanced distribution, since the

amount of memory a GPU has available does not signify anything about its computational power.

Another idea would be to divide the volume into smaller pieces (like a cake) and let each GPU grab one at a time and then process it. This way work would be distributed more equally, and in the end it would be more likely that the GPUs finish more or less simultaneously. However, processing small pieces of the volume creates much more overhead than processing larger chunks, because for each reconstructed piece all images must be loaded and preprocessed.

In the end, it turns out that some a priori knowledge is necessary. Each GPU must have a weight that specifies the percentage of the workload it should execute and must be proportional to its performance. How this weight is obtained or calculated is not of importance for the algorithm itself. More about this is explained in section 5.3.1. Now the volume can be divided into parts that are then assigned to the different GPUs. This division happens along the same dimension along which the volume is split in the single-GPU case due to the limitations of VRAM, which is the z-axis.

Addressing Multiple GPUs The next question we have to answer is how we address the different GPUs programmatically. It is possible to execute commands on different GPUs from one CPU thread by switching the active device using the CUDA API function *cudaSetDevice*. In this case all actions normally performed on one GPU must now be performed on every GPU, requiring a lot of switching for each call. For example, the reconstruction kernel would first be launched on GPU one, then on GPU two and so on. Since kernel launches are asynchronous, this is possible. When the kernels have been launched, a *cudaDeviceSynchronize* call would be necessary to wait for all GPUs to finish their kernels. Then it can be continued with the next iteration. Although theoretically possible, this approach would introduce a lot of overhead. First of all, some actions cannot be performed asynchronously, others only under certain conditions. This would mean, as soon as such a function is called on one GPU, the CPU thread blocks until it terminates, afterwards it is called on the next GPU and so on. Furthermore, switching the device and synchronising the devices has a certain overhead, which would occur every time *cudaSetDevice* or *cudaDeviceSynchronize* are called. On top of that, this would basically tie the different GPUs together. They would have

to wait for each other to finish at the end of every iteration, making this a very rigid and inflexible system. Consequently, it becomes apparent that this is not a very good approach.

Instead, it is better to run the reconstruction for each GPU in a separate CPU thread. Each GPU can then process its assigned part in the same way it would do in the single-GPU case. For the GPU itself, only that portion exists. It does not know that it takes part in achieving a higher goal, and it cannot determine whether it is performing the task alone or together with other GPUs. The only thing that has to be ensured is that the workloads for the different GPUs are adequate. Algorithm 6 illustrates how the parent thread launches the different parts of the reconstruction in individual CPU threads for the different GPUs.

Data: List of devices to be used
Result: Reconstructed volume

```
1 calculate or obtain weights for GPUs;
2 foreach GPU do
3     calculate start and end slice index based on GPU weight;
4     launch reconstruction on that GPU in a new thread;
5 foreach thread do
6     wait for thread to finish;
```

Algorithm 6: *Launching of the reconstruction on multiple GPUs in individual threads*

5.3.1 GPU Load Distribution

When using two identical GPUs, load distribution is pretty simple. An optimal workload distribution will be achieved if both GPUs do 50 % of the work. In this case they both require the same amount of time to process their parts, and the time consumed will be exactly 50 % of the time one GPU would need on its own.

When two different GPUs work together, this is no longer the case. The work will have to be distributed in a way that each GPU does the portion of the work that matches its power. Therefore, weights are introduced that specify how much of the total workload a GPU will have to perform. Then the following question arises: Assuming we know how long

each GPU needs for performing the whole task on its own, what is the theoretically optimal execution time that could be achieved when both GPUs work together?

First of all, it is necessary to acknowledge that the amount of time needed for execution will be smallest when both GPUs finish exactly at the same point in time. The factors that distribute the work between the GPUs shall be called f_a for the first and f_b for the second GPU. Their sum must be one. Furthermore, let d_a be the duration that the first GPU needs to process the whole task on its own, and d_b the duration that the second GPU needs to do so. From these assumptions follow equations 5.2 and 5.3.

$$f_a d_a = f_b d_b \tag{5.2}$$

$$f_a + f_b = 1 \tag{5.3}$$

Thus, the total duration required to complete the task using both GPUs will then be d as shown in equation 5.4.

$$d = f_a d_a = f_b d_b \tag{5.4}$$

Combining equation 5.2 and 5.3 yields the equations 5.5 and 5.6 for calculating the weights f_a and f_b.

$$f_a = \frac{d_b}{d_a + d_b} \tag{5.5}$$

$$f_b = 1 - f_a = 1 - \frac{d_b}{d_a + d_b} = \frac{d_a}{d_a + d_b} \tag{5.6}$$

Thus, we can conclude that the best case duration for parallel execution can be obtained as shown in equation 5.7.

$$d = \frac{d_a d_b}{d_a + d_b} \tag{5.7}$$

However, this is only a theoretical measure. In reality, the performance of the parallelised execution could still be slightly better, since the formulas don't take the steps of device memory allocation, deallocation and device to host memory copying into consideration. In the single-GPU case these steps have to be performed by each GPU alone. In the case of two GPUs, each GPU will only have to allocate a fraction of the device memory it would otherwise need to allocate when performing the whole work on its own and will just have to download that portion, as well. Since the algorithm runs in an independent thread for each GPU, these operations will be performed in parallel, resulting in a performance gain.

What has been shown here for the case of two GPUs can be applied to cases with more than two GPUs equivalently.

Practical realisation of GPU weight distribution In the implementation we decided to make the GPU weights dependent on GPU features that are related to processing performance, in particular the amount of streaming multiprocessors and the theoretical memory bandwidth. We hoped that this would produce a sufficient load distribution without the need for substantial manual interference. First we calculate a weight $w_{i_{\text{SM}}}$ for each GPU i based on its amount of streaming multiprocessors s_i.

$$w_{i_{\text{SM}}} = \frac{s_i}{\sum_{i=0}^{N-1} s_i} \tag{5.8}$$

Next, we calculate a second weight $w_{i_{\text{BW}}}$ based on each GPU's theoretical peak memory bandwidth b_i.

$$w_{i_{\text{BW}}} = \frac{b_i}{\sum_{i=0}^{N-1} b_i} \tag{5.9}$$

These two weights are then combined and weighted by the constants α and β, yielding a final weight w_i for each GPU.

$$w_i = \frac{w_{i_{\text{SM}}}^{\alpha} w_{i_{\text{BW}}}^{\beta}}{\sum_{i=0}^{N-1} w_{i_{\text{SM}}}^{\alpha} w_{i_{\text{BW}}}^{\beta}} \tag{5.10}$$

The constants α and β can be chosen by the user to influence the weights.

This weight calculation, which is based on physical properties of the GPUs, was thought to make it possible for the user to find two coefficients α and β that would produce an optimal weight distribution for an arbitrary number of GPUs independent of the dataset. However, the practical application showed that this doesn't work as well as desired. A majority of the time, it is necessary to readjust the coefficients for each dataset of different size. Furthermore, it is difficult for the user to anticipate what result their changes to the coefficients will have. It would most likely be of more practical utility to allow the direct designation of the weights for each individual GPU. However, this would make the setup more tedious, because as many coefficients as there are GPUs would need to be defined while still having to ensure that their sum yields *one*.

Another conceivable approach would be to have a test run with the processing and reconstruction for just *one* image on every GPU. From the measured execution times the relative weights of the GPUs could then be calculated. However, we do not know how precise and robust this method would be. Nevertheless, this is an approach that could be put to the test in the future.

For the measurements included in section 6 that were taken of multi-GPU execution the coefficients α and β are always mentioned in the text.

6 Evaluation

After the Algorithm was implemented and parallelised, it was tested with regard to performance and quality of the reconstruction. The results shall be discussed in this chapter.

6.1 Experiment Setup

6.1.1 Hardware Configurations

There were three different hardware configurations that we used for evaluating the performance of our implementation. The first one was a consumer configuration with one CPU and one GPU, and had the following specifications:

- Intel Core i7 4770K CPU, 4 physical/8 logical cores at 3.5 GHz
- Nvidia Geforce GTX 970 GPU at 1253 MHz with 4 GB VRAM at 224 GB/s, providing 13 streaming multiprocessors

The second one was a server configuration with two CPUs and two GPUs. The specifications of this configuration were:

- 2× Intel Xeon E5-4640 CPU, 16 physical/16 logical cores each at 2.4 GHz (32 cores total)
- Nvidia Tesla K20c GPU at 706 MHz with 4.8 GB VRAM at 208 GB/s, providing 13 streaming multiprocessors
- Nvidia Quadro K5200 GPU at 771 MHz with 8 GB VRAM at 192 GB/s, providing 12 streaming multiprocessors

The third system was a laptop configuration with the following specifications:

- Intel Core i7 3630QM, 4 physical/8 logical cores at 2.4 GHz
- 2x Nvidia Geforce GT 650M at 835 MHz with 4 GB VRAM (each) at 80 GB/s, providing 2 streaming multiprocessors each

The first hardware configuration mentioned represents a typical consumer PC system. The CPU has a quite good per-core performance, but can only offer 4 physical cores. Additionally, hyperthreading is supported, which maybe will provide some extra speedup. The graphics card is a higher-range consumer card of a very recent generation. Here we expect a significant speedup from using GPU computing versus CPU computing.

Hardware configuration two is a server configuration, which already offers massive parallelisation on the CPU, featuring two Xeon processors with 16 cores each. This results in 32 cores total. However, the individual cores are rather slow, and hyperthreading was not available on our system. Furthermore, it has two graphics cards to offer. One of them, the Tesla K20c, is dedicated to GPU computing. However, we don't know if this means that the card will yield better performance. From what the manufacturer suggests, it should be especially advantageous when performing double precision computations. The Quadro card is also a card dedicated to professional users, but cannot offer any special GPU computing features. It is expected to work more or less like a consumer-grade card. Both cards are of an earlier design cycle, thus their performance is not expected to be absolutely top-notch.

The third system was a portable PC, otherwise known as "laptop", and all of the contained hardware was particularly designed for such. The CPU is still a quite recent release. When it comes to CPUs, the performance of the mobile versions is usually not much worse than that of their stationary counterparts. However, regarding the GPU we expect a higher performance hit compared to the desktop variants. What is the most interesting about this configuration, is that it features a dual-SLI setup with two identical graphics cards. As this is the only available configuration providing this setup, it was very useful for us for the evaluation of the multi-GPU parallelisation.

The amount of RAM is not mentioned in the configurations because it is not of importance for the tests we conducted. We always assume that there is enough RAM available to hold the whole volume. If this is not the case, the program cannot run. However, the

operating system might handle this case by providing virtual memory, but this will degrade performance. In our case there was always enough RAM available.

6.1.2 Data Sets

For the evaluation of the implementation we mainly used one dataset, which is a CT scan of a toy figure. This dataset comprises 1199 images. We created versions of different size in order to evaluate the performance of the algorithm at different cardinalities and simulate different scenarios. The sizes referred to in this chapter are:

256 pixels set Images of size 224×256

- This dataset is very small, thus the time consumed per reconstruction iteration (i.e. per image) is very short. Overhead and latencies originating from file handling, etc., will have a bigger impact in this scenario.
- The resulting volume will have a size of 46 Mb.
- Approx. 15 billion voxel operations are necessary for the reconstruction.

700 pixels set Images of size 613×700

- This dataset was solely used for the tests run on the GTX 650M GPU because the amount of VRAM required for the reconstruction matches the 2 GB of VRAM this GPU has to offer, thus utilising a majority of it while still allowing the graphics card to reconstruct the whole volume in one pass.
- The resulting volume will have a size of 0.98 GB.
- Approx. 315 billion voxel operations are necessary for the reconstruction.

1000 pixels set Images of size 875×1000

- This is the biggest dataset that can still be reconstructed in one pass by a GPU with 4 GB of VRAM. Thus, execution times should scale best with this dataset when comparing CPU vs. GPU and GPU vs. multi-GPU executions. Overhead

will presumably not have a major impact. This dataset represents a typical scenario of practical use.

- The resulting volume will have a size of 2.85 GB.
- Approx. 917 billion voxel operations are necessary for the reconstruction.

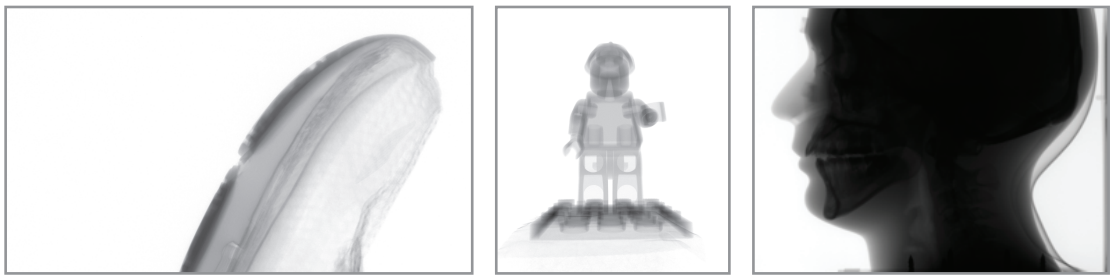
2000 pixels set Images of size 1750 × 2000

- This dataset is the largest one used. Even on the GPUs with the most VRAM (8 GB) multiple passes will be required for reconstruction. Therefore some overhead will be introduced by the division of the reconstruction process into multiple passes. This also represents a typical scenario of practical use, being the reconstruction of large datasets.
- The resulting volume will have a size of 22.82 GB.
- Approx. 7 trillion voxel operations are necessary for the reconstruction.

One voxel operation is referred to as the operation of taking one sample out of one image and adding it to the value of one voxel. The size of the reconstructable cylinder was not taken into account for the calculation of this measure.

There was also a second dataset that was used for evaluation, being a scan of a sneaker. However, this dataset is only referred to twice in the sections 6.2.3 and 6.2.5. It comprises a set of 501 images, each at a resolution of 1000 × 571 pixels. Being 2.13 GB in size, it is nearly as large as the 1000 pixels toy figure set. However, with only less than half as many images, only approx. 286 billion voxel operations are required for the reconstruction. Figure 6.1 shows one x-ray image from each of the three datasets.

For the comparison of our implementation to OSCaR the skull phantom data set that comes with OSCaR was used. It comprises 641 images with a resolution of 256 × 192 pixels.



Sneaker dataset | Toy figure dataset | Skull phantom dataset

Figure 6.1: *One x-ray image from each of the datasets used*

6.2 Performance

6.2.1 CPU Parallelisation

Singlethreaded CPU Compared to Multithreaded CPU

Figure 6.2 shows a comparison between singlethreaded and multithreaded CPU performance. The two Xeon processors with at total of 32 cores achieve a speedup of factor 27.5

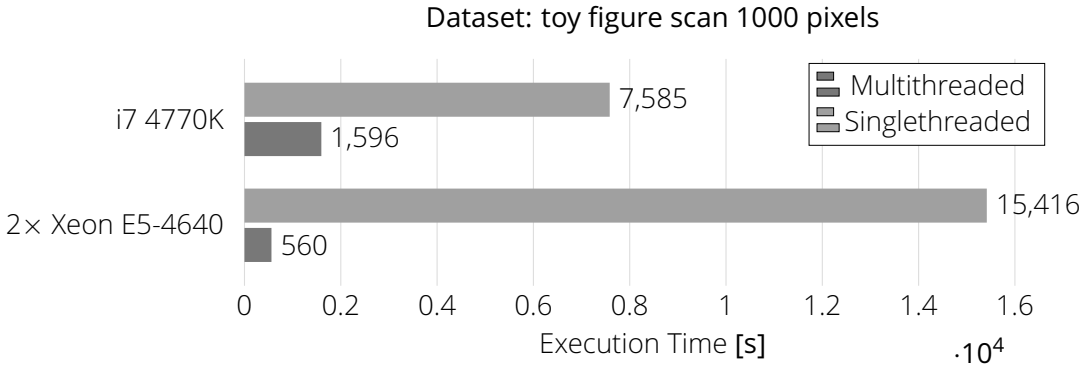


Figure 6.2: *Comparison of the execution times of singlethreaded and multithreaded CPU execution*

for the 1000 pixel dataset, remaining slightly behind what would be the theoretical optimum. The amount of concurrent threads probably introduces a fair amount of scheduling overhead here. The four-core i7 CPU with hyperthreading achieves a 4.7-times speedup, which is quite remarkable. A more elaborated discussion about the benefit of hyperthreading can be found in section 6.2.2.

6.2.2 Hyperthreading

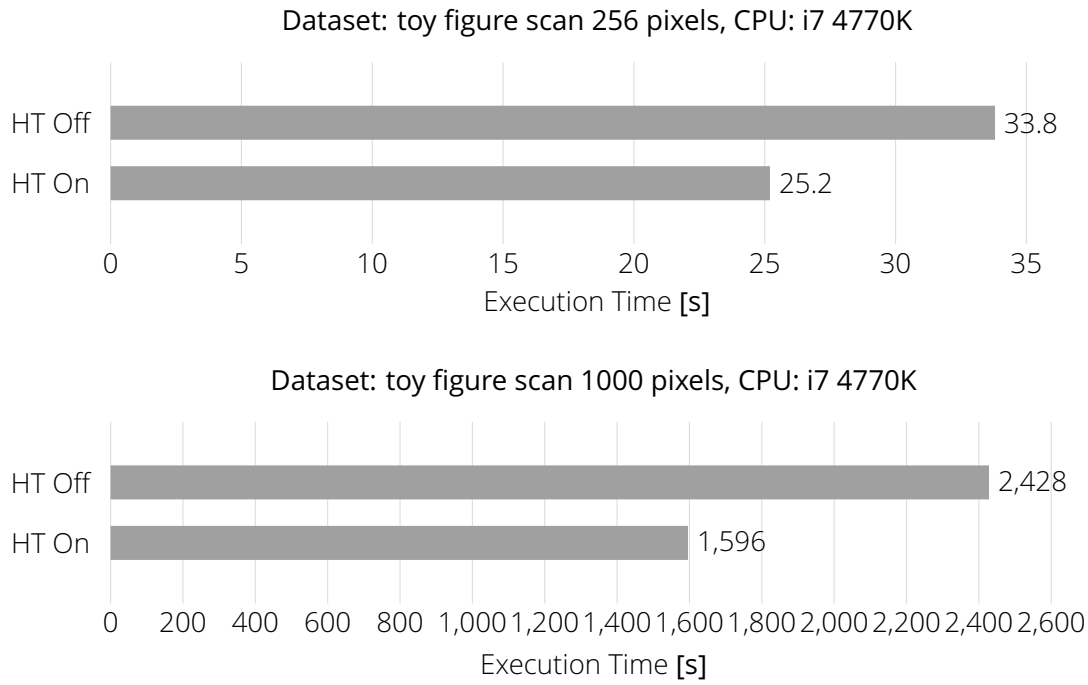


Figure 6.3: *Comparison of the execution times of hyperthreaded and non-hyperthreaded CPU execution*

Figure 6.3 shows the benefit gained through the use of hyperthreading on a four-core CPU. With the small 256 pixels data set, a reduction in execution time of 26 % is obtained, while in case of the 1000 pixel data set, the execution time is even reduced by 34 %. This is remarkable for such a feature.

Parallelisation Overhead

As we came to realise when designing our program, parallelisation is not always beneficial. The results in Figure 6.4 show that parallelising the image preprocessing does actually result in a drastically longer execution time on the Xeon CPUs and a slightly worse execution time on the i7 CPU in the case of the smaller volume. In case of the bigger one the difference is not so pronounced, but the performance of the parallelised version is still worse than that of the non-parallelised one. The reason for this can only be the scheduling overhead that is being introduced by the new threads. The preprocessing runs simultaneous to the back-

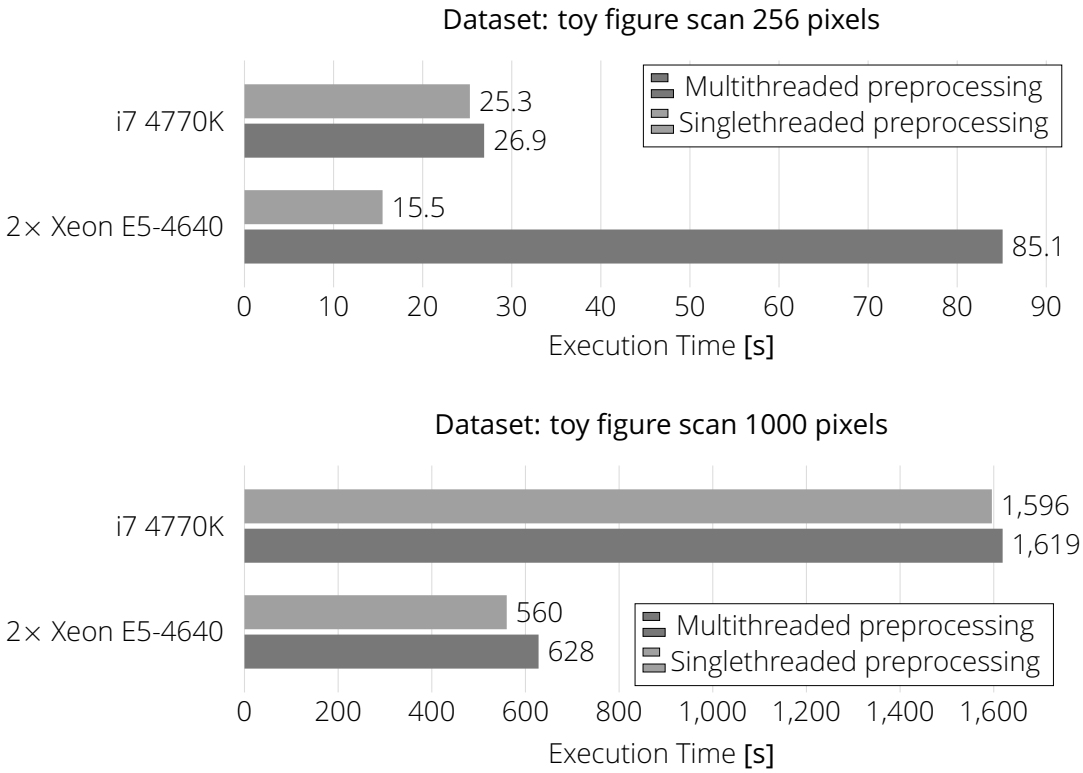


Figure 6.4: *Comparison of the execution times using parallelised and non-parallelised preprocessing on the CPU*

projection. This means that on the Xeon CPU there already are 32 active threads working on the reconstruction. Introducing another 32 threads for the preprocessing apparently rather hurts the multithreading performance. Additionally, the data processed in the preprocessing is very small; thus, the real time benefit in parallelisation would still probably be neglectable. In combination, these two factors lead to a generally worse overall performance if the preprocessing is multithreaded. On the i7 processor the effect is not as big, certainly due to the significantly lower total amount of threads that need to be scheduled.

If the data set becomes larger, the effects are also much less pronounced, most likely because the execution time of each reconstruction kernel is higher in this case, and the introduced overhead is relatively small.

This reminds us that parallelisation is not automatically beneficial in every case.

6.2.3 GPU Parallelisation

GPU Compared to CPU

GPU parallelisation is one of the core aspects of our implementation. Figure 6.5 shows a comparison of the speedup gained by using the GPU over the CPU on single-GPU configurations. First, let us have a look at the 1000 pixel data set. On the consumer grade system, a speedup of factor 21 can be reached in comparison with multithreaded CPU execution. If we take singlethreaded CPU execution as a reference, the speedup is as much as 101 times. Theoretically, a similar CPU with 129 cores would be necessary to achieve the GPU result, not taking the thread scheduling overhead into account.

If we examine the results of the server-grade system, CPU and GPU are noticeably closer together, since with 32 individual cores the parallelism is already quite high on the CPU, and the installed GPUs are rather slow. The Tesla card can reach a speedup of factor five and the Quadro card a factor of four. However, in practical application this is still a significant advantage. Furthermore, the two GPUs can be used for reconstruction together. The multi-GPU performance will be discussed in section 6.2.5.

If we have a look at the very small 256 pixel dataset the differences are not as big as in the medium size case. The speedups are 17 times on the consumer-grade and seven/four

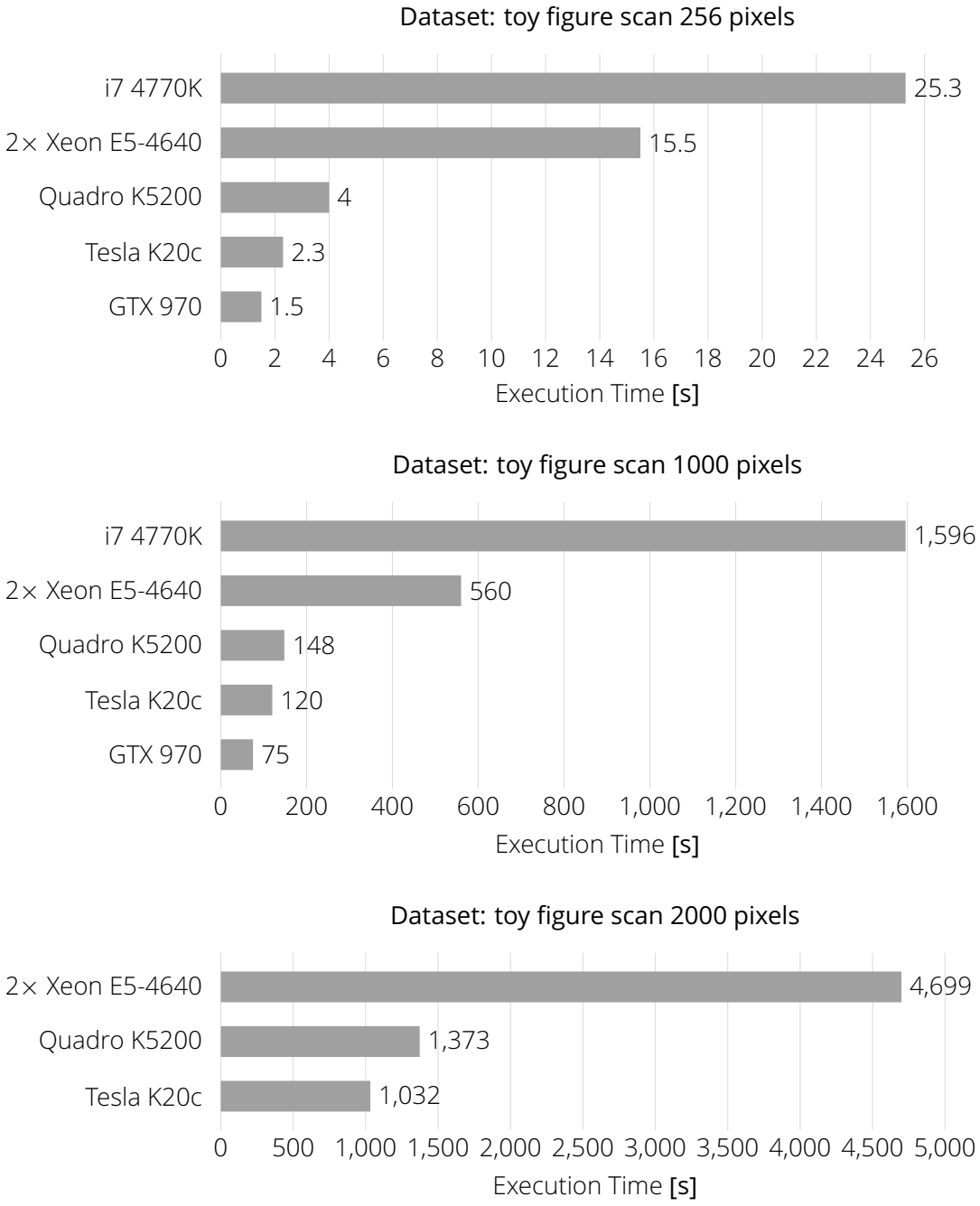


Figure 6.5: *Comparison of the execution times of different GPUs and (multithreaded) CPUs*

times on the server-grade configuration. Naturally, the overhead is big in this scenario compared to the short time spent on the actual reconstruction. The Tesla card is the only one that delivers a better result in this scenario. However, such a small volume is a quite unrealistic use case.

For the 2000 pixels dataset the GPUs have to run multiple passes because the whole volume does not fit into the video memory. For this data set we do not have measurements of the consumer system, as it does not have enough RAM to run this reconstruction. On the server configuration the speedup is factor six on the Tesla and factor three on the Quadro card. What is interesting is that the performance of the Tesla card improves compared to its performance with the 1000 pixel data set, while that of the Quadro card becomes worse. Yet, seen in total, the speedup is comparable to that of the medium sized data set.

In conclusion we can say that the GPU implementation provides a big improvement in performance, which has a tremendous impact on practical use. As an example: A reconstruction taking 10 minutes on the GPU would take three hours and 33 minutes on the CPU, assuming the i7/GTX970 configuration.

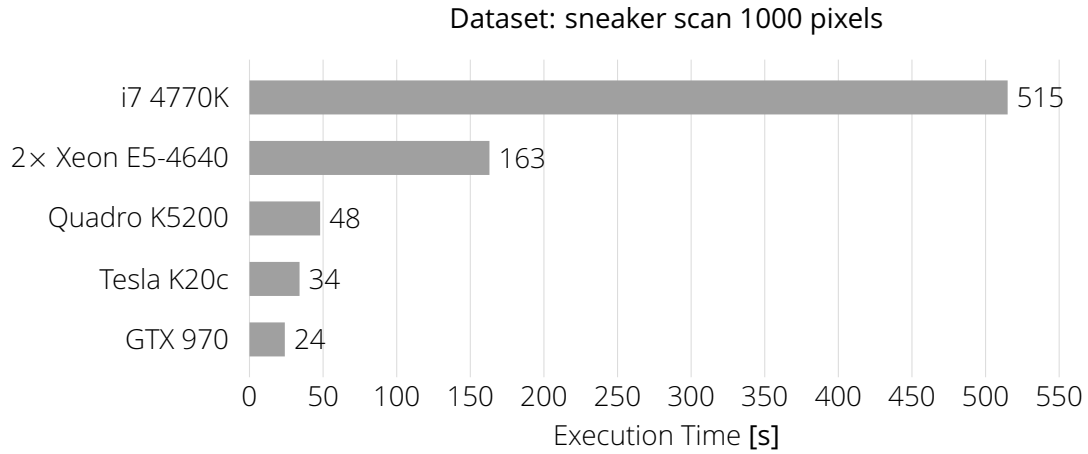


Figure 6.6: *Comparison of the execution times of different GPUs and (multithreaded) CPUs*

To confirm our results we ran the test again with the sneaker data set. This data set is similar in size but comprises much fewer images. The outcome is shown in Figure 6.6. As can be seen, the numbers essentially confirm the measurements taken with the toy figure dataset.

6.2.4 Impact of VRAM Size

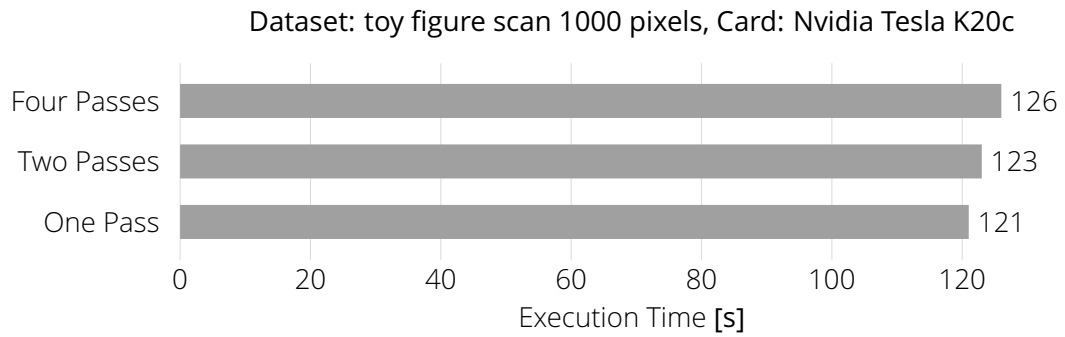


Figure 6.7: *Comparison of the execution times with different VRAM sizes.*

Another interesting aspect is how big of an impact the video memory size has on reconstruction time. To evaluate this we ran the reconstruction multiple times on the GPU but artificially restricted the amount of VRAM in a way that the volume had to be split into two or four parts respectively. Figure 6.7 illustrates the results of this experiment. There is a decrease in performance, but it is much less than was expected, considering that for every pass all images have to be loaded and preprocessed again. This means that GPUs with small memory can unleash their potential, as well.

6.2.5 Multi-GPU Parallelisation

Multi-GPU Compared to Single-GPU

Figure 6.8 shows a comparison of the execution times of single-GPU and multi-GPU execution for three datasets of different sizes. Table 6.1 lists the results along with the theoretically possible optimums. These were calculated from the individual execution times of the GPUs as explained in section 5.3.1. Column three shows the percentage of the obtained execution times relative to the theoretical optimums, and columns four and five list the speedups of the multi-GPU execution relative to each individual GPU.

For the 1000 pixel volume, letting the Tesla and Quadro cards work together results in a duration of 68 s instead of 148 s for only the Quadro or 120 s for only the Tesla card,

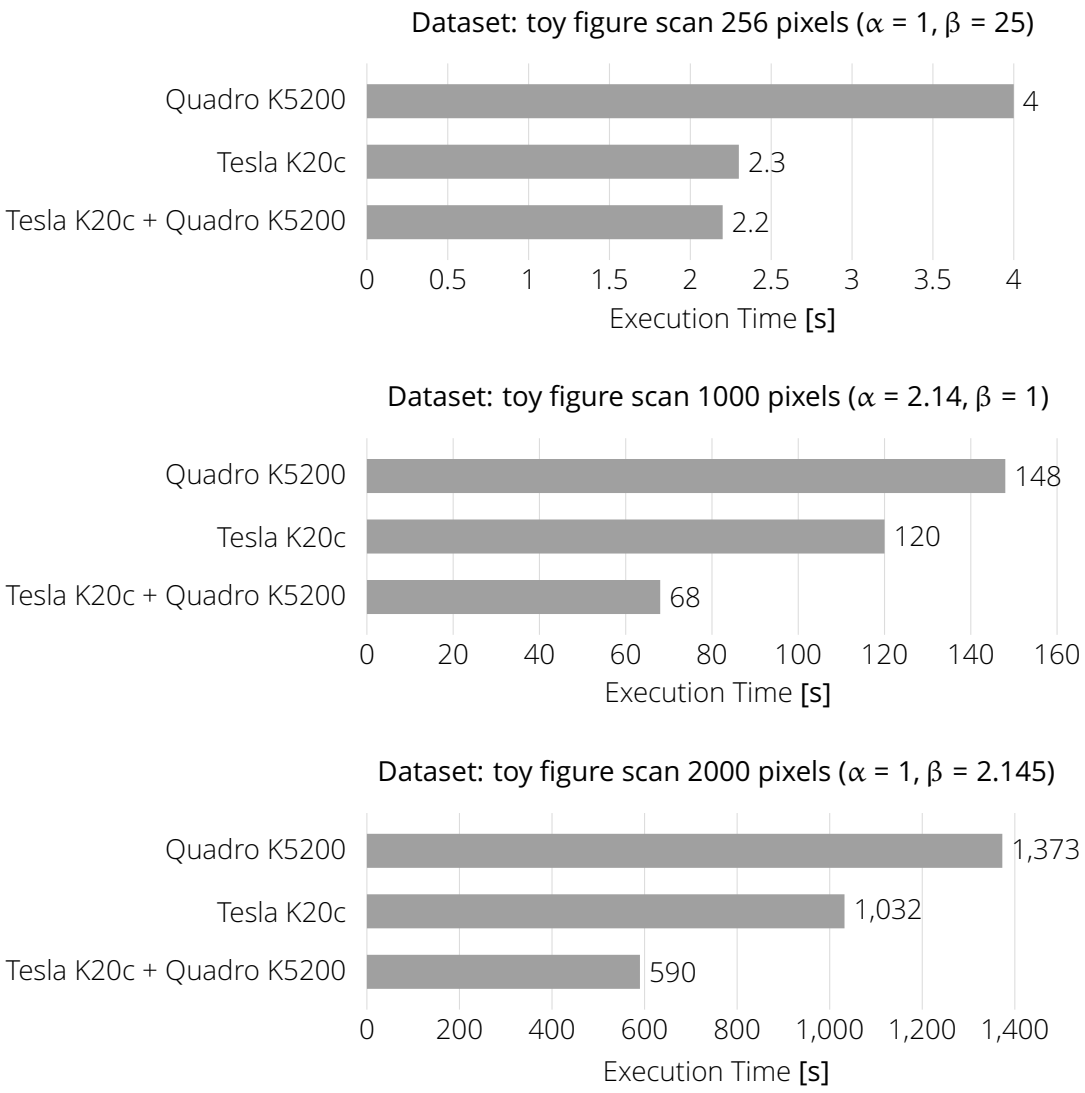


Figure 6.8: *Comparison of the execution times of single-GPU and multi-GPU execution*

Dataset	Theoretical Optimum	Actually Achieved	Relative to Optimum	Speedup over Quadro	Speedup over Tesla
256 pixels	1.46 s	2.2 s	151 %	×1.82	×1.05
1000 pixels	66.3 s	68 s	103 %	×2.18	×1.76
2000 pixels	589 s	590 s	100 %	×2.33	×1.75

Table 6.1: Comparison of the achieved speedups using multi-GPU execution

respectively. Being 103 % of the theoretical optimum of 66.3 s, this result surpasses our expectations and shows that the parallelism across multiple GPUs works very well.

In case of the very small 256 pixel data set the speedups are, as expected, slightly worse with 151 % of the theoretical optimum. Considering how short the execution times are in this case, it is easy to see that the overhead must have a very big impact, proportionally to them.

The best results were actually obtained in the case of the 2000 pixel volume, which needs to be reconstructed in multiple passes, due to its size. Given the execution times of the individual GPUs, which are 1373 s and 1032 s, the achieved 590 s almost perfectly match the theoretical optimal combined execution time of 589.16 s. However, the measurements were quite erratic in this scenario. Therefore, we averaged the results of multiple runs.

For confirmation we wanted to test the multi-GPU execution on a system with two identical GPUs. The only available configuration meeting this criterion was a laptop that was set up with two Nvidia GT 650M graphics cards. The results we obtained from these devices are shown in Figure 6.9. As a result of the utilisation of both GPUs the execution time is

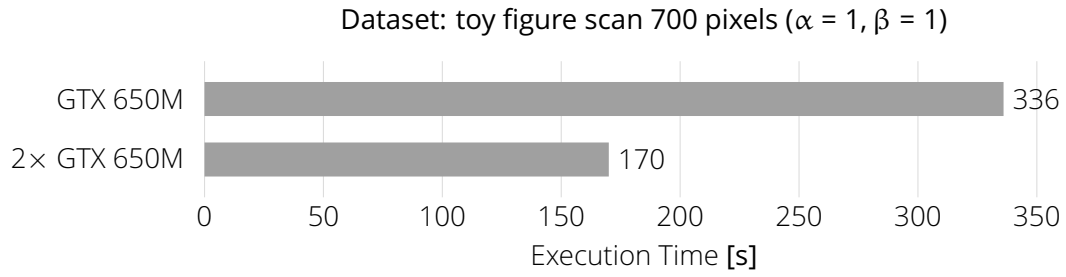


Figure 6.9: Comparison of the execution times of single-GPU and multi-GPU execution with two identical GPUs

reduced by 49.4 %, which confirms the results of our previous experiments.

If we now revisit the CPU results, we can say that the Tesla and Quadro GPUs together achieve a speedup of approximately factor eight over the Xeon CPUs. Considering that CPU parallelisation with two processors and a total of 32 cores is already quite extensive, this shows that there is still useful potential in GPU computing, even in this case.

All in all, we can say that the multi-GPU parallelisation is very successful, as far as we can see, reduces execution times linearly proportional to the amount of graphics units used. Unfortunately, lacking the required hardware, we were not yet able to test the implementation with more than two GPUs.

6.2.6 Scaling

The next aspect that shall be examined is how the execution times scale with the size of the volume. In Figure 6.10 the light grey bars represent the size of six volumes with different resolutions relative to the size of the 1000 pixel volume. The medium and dark grey bars represent the CPU and GPU execution times relative to those required for reconstructing the 1000 pixel dataset. Figure 6.11 is based on the data from Figure 6.10 and shows the relative deviation of the CPU and GPU execution times from the values that were expected based on the proportions in size. The GPU execution times expose a deviation of up to +15 % from what was expected. The 1000 pixel volume apparently needs the least time per voxel. For the 256 pixel volume the deviation is the biggest, which is not much of a surprise since overhead that occurs apart from the actual reconstruction has the biggest impact here. The relative values also make the difference appear larger than it actually is, which is absolutely only about 0.3 s. The biggest volume comes with a deviation of +8 %. The overhead introduced by it needing to be reconstructed in multiple passes is most likely one of the reasons for this.

If we have a look at the CPU numbers, we can say that the variance here is generally about +12 % (from −7 % to +5 %) except for the 256 pixel volume where it is +65 % or about 6 s. There is apparently a larger overhead in the CPU- than in the GPU-reconstruction. Generally, the CPU implementation is most effective with the 800 pixel data set.

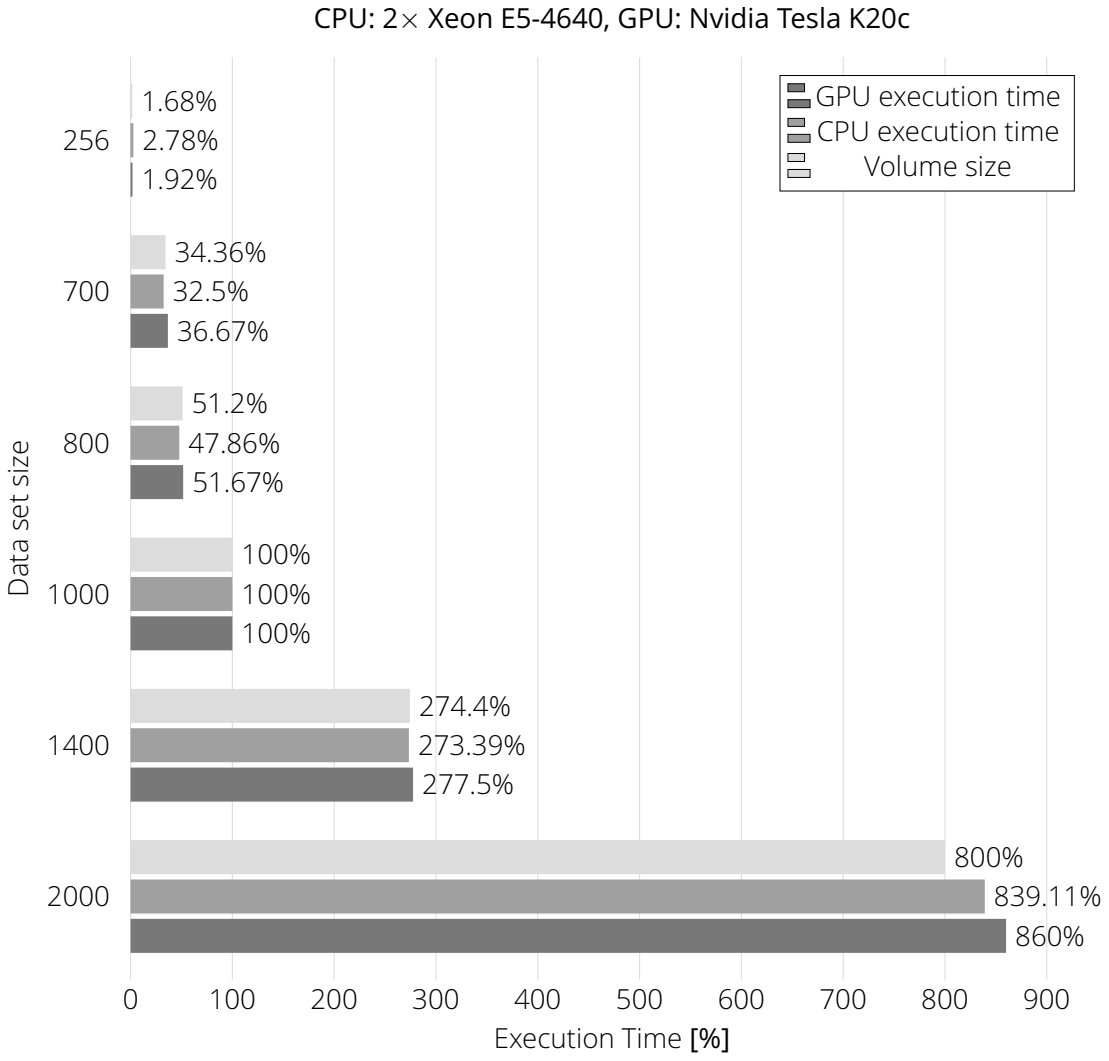


Figure 6.10: Scaling of the execution time in correlation to data set size on CPU and GPU relative to the 1000 pixel data set

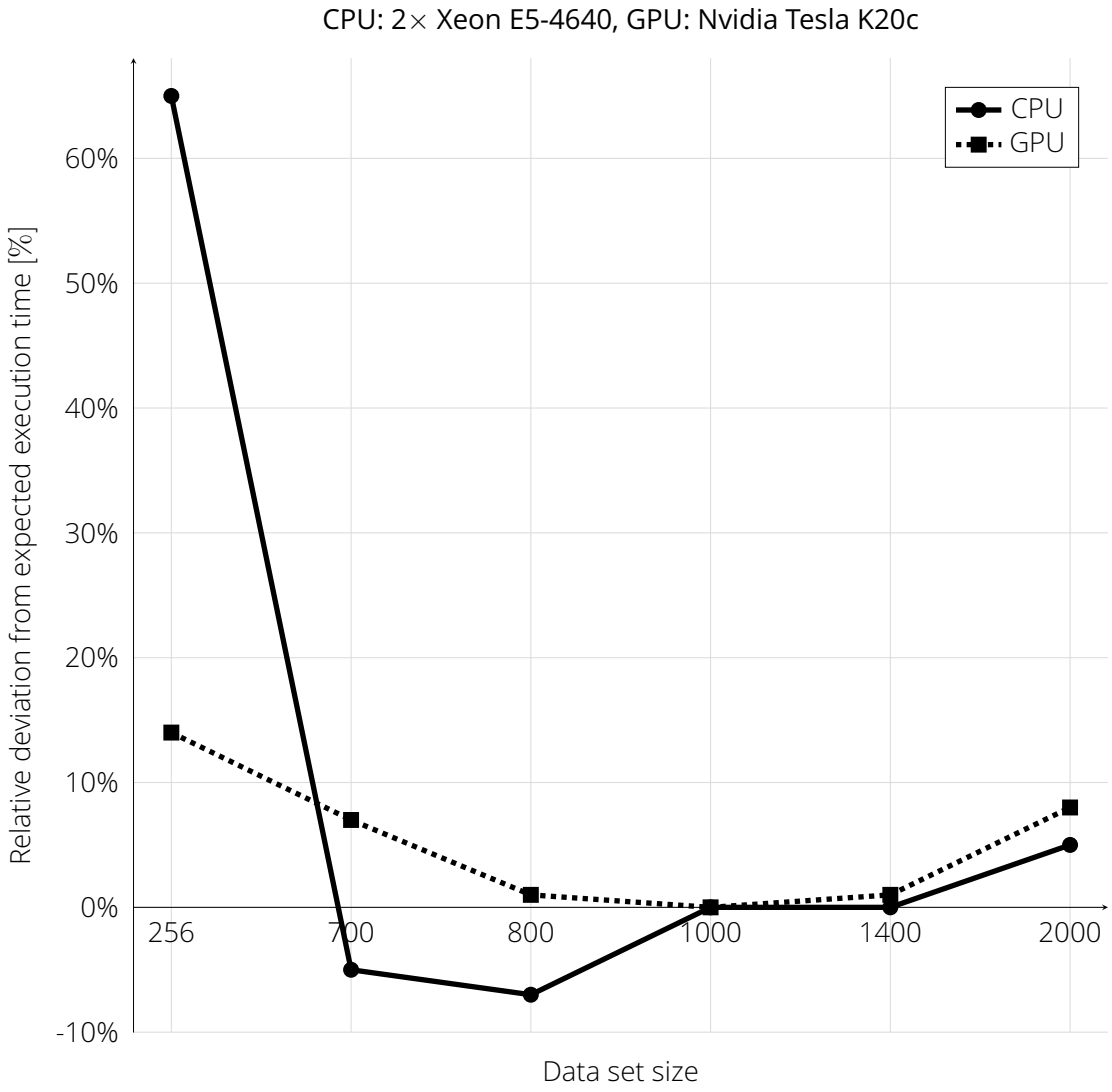


Figure 6.11: Relative deviation of the execution times from the expected ones relative to the 1000 pixel data set on CPU and GPU

6.2.7 Single Precision Compared to Double Precision

In our implementation we used 32 bit floating point precision for in- and output as well as all intermediate results, as this provides absolutely sufficient accuracy. However, it would also be conceivable to use double precision for either the intermediate results alone, to minimise rounding errors, or also for input and output data. The latter would come at the cost of doubling all memory requirements, whereas the former only would affect computation time. Using double over single precision on the CPU does not usually lead to a significant decrease in performance. However, on the GPU this is slightly different, as Figure 6.12 displays. On the Tesla card the increase in computation time is still acceptable, which

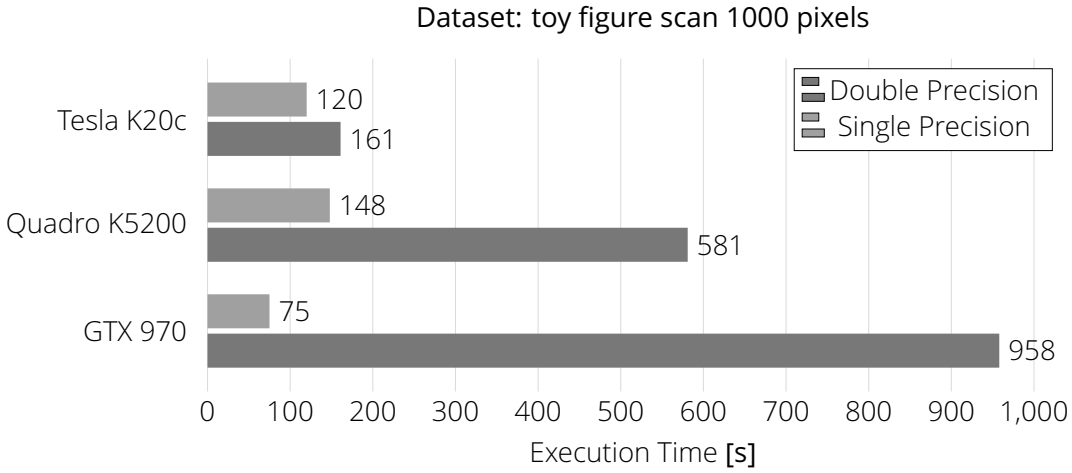


Figure 6.12: Comparison of the execution times for single and double precision GPU computing

is due to the fact that these cards are designed for GPU computing and double precision computing in particular. On the Quadro card, the execution time increases by factor 3.9 over the single precision case. However, on the consumer-grade GTX 970 card the penalty is even higher, with an increase of 12.7 times.

This shows the reasons for the price difference between these cards. While when using floating point precision the GTX 970 surpasses the two much more expensive, professional cards and thus offers a much better price to value ratio. The Tesla and Quadro cards gain ground when it comes to using double precision, making the GTX 970 card last.

Fortunately, double precision is unnecessary for our application thus far, but this highlights the importance of this aspect is. Using only floating point precision it is possible to gain a

decent speedup by using GPU computing over the CPU on *any* GPU. Otherwise, expensive professional-grade Tesla cards were necessary to achieve this goal. To give a slight impression of the price difference: At the time when this thesis was composed, the Tesla K20c cost about 3211 €, the Quadro K5200 about 1721 € and the GTX 970 about 309 €.

6.2.8 Impact of Storage Data Throughput

Since we are reading data directly from non-volatile memory during the reconstruction, we were interested in whether the data throughput of the storage media used had an influence on the performance. Therefore, we conducted a test where the images were stored once on a conventional hard disk and once on a solid state drive. Solid state drives usually have much smaller access times and higher data rates. As Figure 6.13 shows, no substantial difference could be measured, which leads us to the conclusion that our implementation successfully hides potential delays introduced by disk operation.

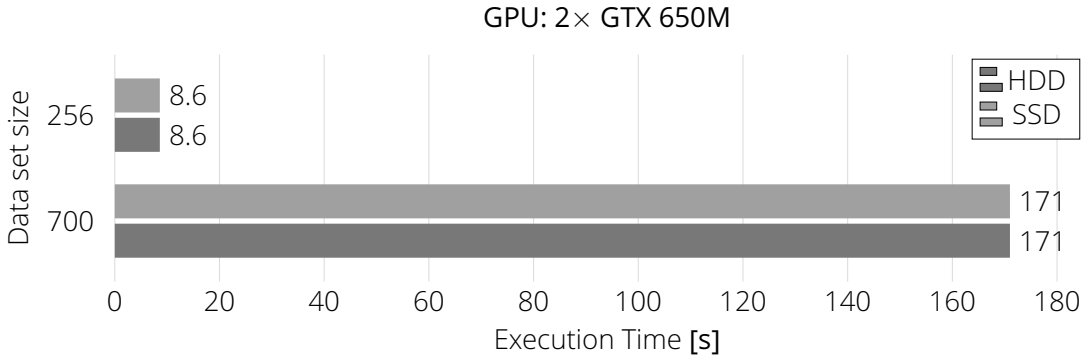


Figure 6.13: *Comparison between the execution times of using an SSD versus an HDD for data storage*

6.2.9 Comparison to OSCaR

OSCaR is one of the few open-source reconstruction tools in existence. Therefore we want to compare its results to that of our implementation. However, it has to be pointed out that OSCaR was written in *MATLAB*, which does not facilitate good performance. According to [11] it was also not the goal of the authors to produce the fastest implementation. However,

we conducted a few tests with the skull phantom data set that comes along with the OSCaR application. The results are shown in Figure 6.14. Unsurprisingly, our implementation is much faster, using CPU processing as well as using GPU processing.

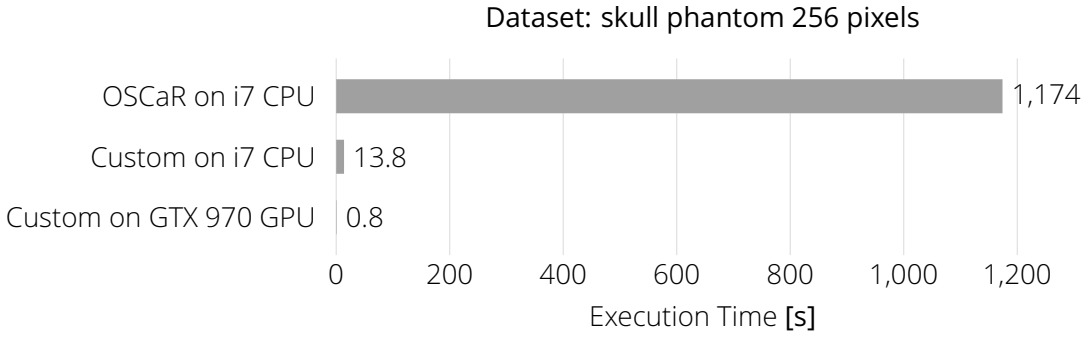


Figure 6.14: Execution times of OSCaR in comparison to our custom implementation

6.3 Memory

6.3.1 CPU Processing

Assuming an image resolution of $w \times h$ pixels, an upper bound for the RAM required in bytes can be calculated as in equation 6.1.

$$\overbrace{4}^{\text{bytes per element}} \left(\underbrace{2wh}_{\text{images}} + \underbrace{w^2 h}_{\text{volume}} + \overbrace{2h\left(\frac{w}{2}+1\right)}^{\text{FFT result (frequency spectrum)}} \right) + \underbrace{D}_{\text{FFT plan}} \tag{6.1}$$

The memory for the FFT result and the FFT plan are necessary for the preprocessing. The FFT result of the row-wise FFT only has a width of $\frac{w}{2}+1$ due to its symmetry and has two channels, one for the real and one for the imaginary component. The variable D is an upper bound of memory required for the FFT plan. This depends on the library used for the FFT; therefore, we can not give a more exact approximation. The formula does not take into account the memory that the program itself occupies. However, this ought to be neglectable.

What is noticeable here is that the memory required is independent of the amount of images, as only two at a time are stored in memory. The biggest change to the memory demand happens if the image resolution, and thus the volume size, is changed.

Equation 6.1 assumes a complete reconstruction of the volume and does not take a potential region of interest into account. Assuming a region of interest of size $x \times y \times z$ gives a slightly different result for the second summand:

$$\overbrace{4}^{\text{bytes per element}} \Bigg(\underbrace{2wh}_{\text{images}} + \underbrace{xyz}_{\text{volume}} + \overbrace{2h\Big(\frac{w}{2}+1\Big)}^{\text{FFT result (frequency spectrum)}} \Bigg) + \underbrace{D}_{\text{FFT plan}} \tag{6.2}$$

6.3.2 GPU Processing

Equation 6.3 shows an upper bound for the RAM required for the GPU reconstruction in bytes, and equation 6.4 does the same while taking a region of interest into account.

$$\overbrace{4}^{\text{bytes per element}} (\underbrace{wh}_{\text{image}} + \underbrace{w^2h}_{\text{volume}} + \overbrace{2wh}^{\text{page-locked memory}}) \tag{6.3}$$

$$\overbrace{4}^{\text{bytes per element}} (\underbrace{wh}_{\text{image}} + \underbrace{xyz}_{\text{volume}} + \overbrace{2wh}^{\text{page-locked memory}}) \tag{6.4}$$

The amount of RAM required for the GPU reconstruction is only slightly less than that for the CPU reconstruction. Only the memory for the FFT is no longer necessary. Since the index ordering on the CPU and the GPU is identical, no memory overhead occurs and the copying can happen directly to the target volume in RAM. Otherwise the GPU volume would first have to be copied over and then to be projected to the target index ordering, requiring extra memory for the temporary storage.

An upper bound for the VRAM required is given in equation 6.5. Once more, the equation changes slightly when a region of interest is taken into account (6.6).

$$\max\left(\overbrace{d}^{\text{image bit depth}}\ \underbrace{2wh+}_{\text{input images}}\ \overbrace{4}^{\text{bytes per element}}\ \underbrace{\left(2wh+\overbrace{w^2h}^{\text{volume}}+\overbrace{4h\left(\frac{w}{2}+1\right)}^{\text{FFT results}}\right)}_{\text{temporary images}}+\underbrace{D}_{\text{FFT plan}}\ ,\ \overbrace{V}^{\text{VRAM size}}\right) \tag{6.5}$$

$$\max\left(\overbrace{d}^{\text{image bit depth}}\ \underbrace{2wh+}_{\text{input images}}\ \overbrace{4}^{\text{bytes per element}}\ \underbrace{\left(2wh+\overbrace{xyz}^{\text{volume}}+\overbrace{4h\left(\frac{w}{2}+1\right)}^{\text{FFT results}}\right)}_{\text{temporary images}}+\underbrace{D}_{\text{FFT plan}}\ ,\ \overbrace{V}^{\text{VRAM size}}\right) \tag{6.6}$$

There are four images that are stored in memory, two for each stream. One of these is the initial image of depth d, which can be one, two or four bytes. The other one is the result of the preprocessing, which is also used for intermediate results and has a depth of four bytes. The memory for the FFT result is now required in the GPU memory, but the amount stays the same as in the CPU case. Again, the FFT also plan requires a certain, unknown amount of memory. It is possible to query this amount via the CUFFT API, once a plan has been set up.

The main difference to the CPU implementation is that the algorithm is adaptive, i.e. it can also work with less VRAM. In this case, the upper bound for the memory is the size of the VRAM itself, denoted V. However, there is a minimum that is necessary to enable the algorithm to execute. This minimum is described in equations 6.7 and 6.8 with and without region of interest, respectively.

$$\overbrace{d}^{\text{image bit depth}}\ \underbrace{2wh+}_{\text{input images}}\ \overbrace{4}^{\text{bytes per element}}\ \underbrace{\left(2wh+\overbrace{w^2}^{\text{volume slice}}+\overbrace{4h\left(\frac{w}{2}+1\right)}^{\text{FFT results}}\right)}_{\text{temporary images}}+\underbrace{D}_{\text{FFT plan}} \tag{6.7}$$

$$\overbrace{d}^{\text{image bit depth}}\ \underbrace{2wh+}_{\text{input images}}\ \overbrace{4}^{\text{bytes per element}}\left(\underbrace{2wh+}_{\text{temporary images}}\ \overbrace{xy}^{\text{volume slice}}\ \overbrace{+4h\left(\frac{w}{2}+1\right)}^{\text{FFT results}}\right)+\underbrace{D}_{\text{FFT plan}} \tag{6.8}$$

The only change here is the term w^2 instead of w^2h, which is just one x-y-slice instead of the whole volume. This is a theoretical minimum. In the current implementation the iterative memory allocation is not designed to work as precisely, instead the adjustment steps are coarser. The size of these steps is defined by a constant in the code and cannot be modified from outside the program. However, if desired, this could be adjusted according to need.

6.4 Quality

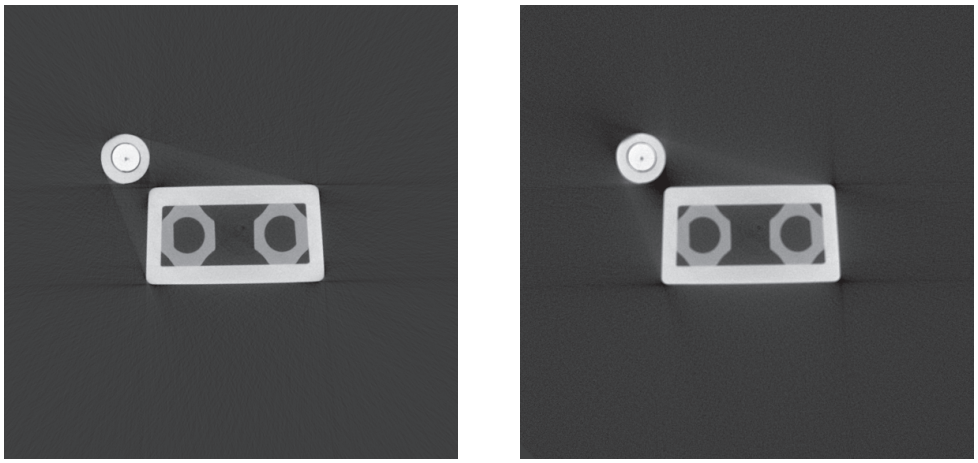
Now that the performance has been thoroughly evaluated, the quality of the reconstruction result shall be examined, as well. What makes this difficult, is that there is no exact reference to which we could objectively compare our results. We will, however, evaluate the quality subjectively and compare it to that of the commercial solution *Phoenix datos|x 2* as well as to that of the open-source implementation *OSCaR*.

6.4.1 General Evaluation

Subjectively, the reconstruction result looks like an accurate representation of the scanned object, which can be seen in the photo shown in Figure 6.15. Enclosed cavities are clearly outlined and the different materials can be easily distinguished. The scan exhibits, however, a number of severely visible beam hardening artifacts. This is due to the omission of physical filters that absorb the soft x-rays during the scanning procedure. Our application does time not provide any mechanisms for reducing such artifacts at the current.

Figure 6.15: *Photo of the scanned toy figure*

Figure 6.16 shows our result in comparison to that of the commercial CT reconstruction software. As it turns out, the result of the



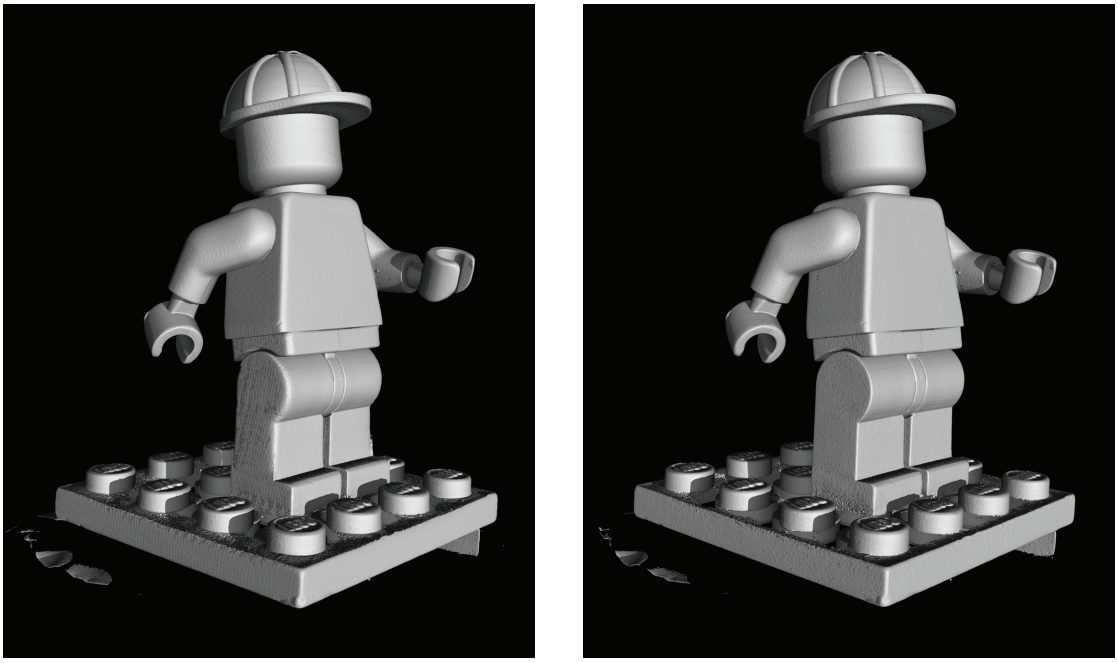
Commercial software Custom implementation

Figure 6.16: *The result of our custom implementation in comparison to that of a commercial reconstruction software (cross section)*

commercial software is slightly sharper than that of our implementation. It could be that the third-party software, which works hand in hand with the CT scanner, had access to the exact individual angles at which each of the projections was captured, which was unfortunately not the case for us. If we had had access to this information, our reconstruction would possibly be equally sharp. Instead, an equidistant division of the interval $[0 - 360]$ was used. The beam hardening artifacts appear to be slightly reduced in the result of the commercial application. It might be that some postprocessing technique has been used here to reduce such artifacts. The noise levels of both versions appear to be quite similar.

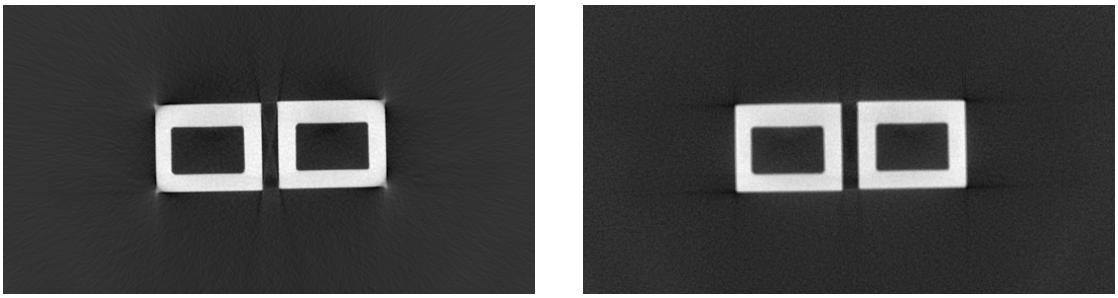
If rendered as an isosurface model, as can be seen in Figure 6.17, the differences are hardly noticeable. However, the edges of the toy figure's legs appear much softer in the model of the third-party application. The cross sections reveal some artifacts at the edges of the legs (Figure 6.18) that are not present in our reconstruction. In both reconstructions at the bottom of the feet some parts which should be opaque are partially transparent, the reason for this most likely being the beam hardening artifacts. Those are also presumably responsible for the holes that show in the bottom plate.

Comparing the histograms of the reconstructions, as seen in Figure 6.20, it can be observed that the peaks are more salient in the third-party result. This coincides with the higher sharpness being found there. However, seen as a whole, the histograms look quite



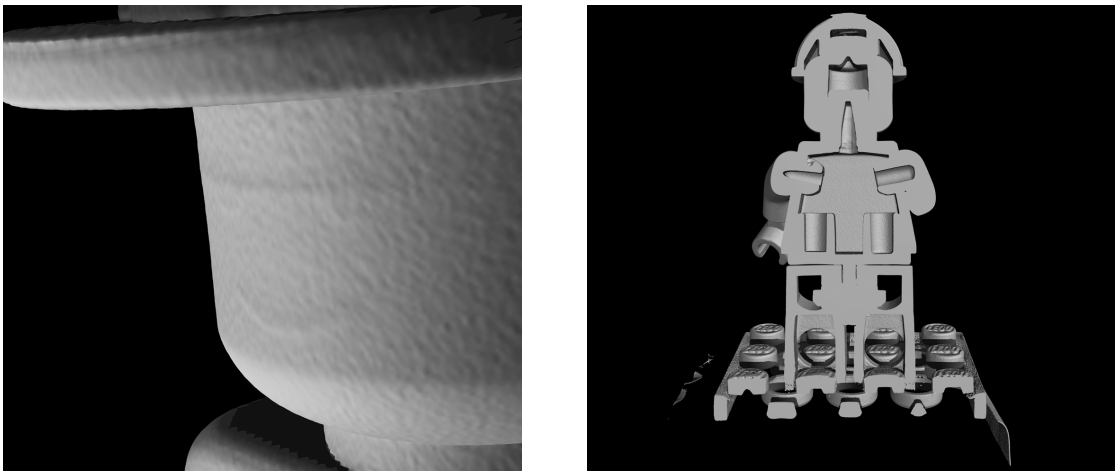
Commercial software Custom implementation

Figure 6.17: *The result of our custom implementation in comparison to that of a commercial reconstruction software (isosurface rendering rendered with VGStudio)*



Commercial software Custom implementation

Figure 6.18: *The reconstruction produced by the commercial application shows artifacts at the corners of the legs, which are not visible in our custom implementation. The effects of these artifacts can also be seen in Figure 6.17*



(a) (b)

Figure 6.19: In a closer view the printed face of the toy figure can be recognised (a). The intersection through the reconstructed model shows enclosed cavities (b).

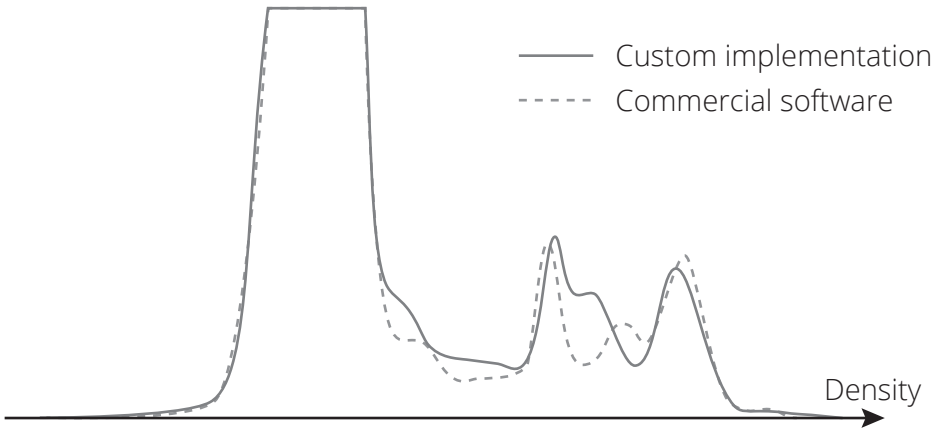
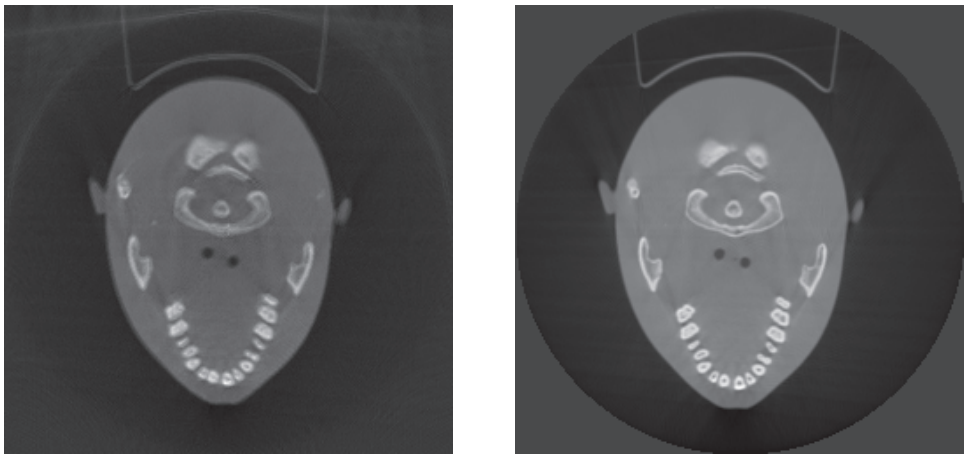


Figure 6.20: Comparison between the histograms of the results of our custom implementation and a commercial reconstruction software (histogram cropped at the top)

similar. What must also be taken into account is that the regions of interest in the two reconstructions were not perfectly identical.

6.4.2 Comparison to OSCaR

Figure 6.21 shows a comparison of our results to those of the open-source FDK reconstruction software OSCaR, using the skull phantom dataset that is included in the OSCaR package. Please note that there might be slight differences in the cross section because the

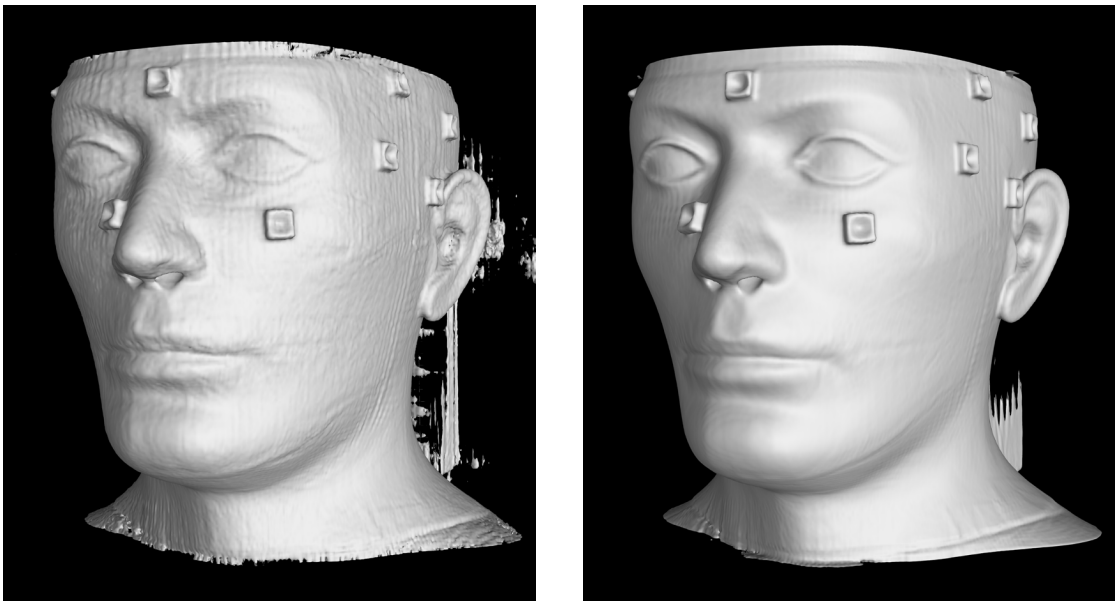


OSCaR Custom implementation

Figure 6.21: Comparison of a cross section of the results of our custom implementation and the OSCaR open-source reconstruction software

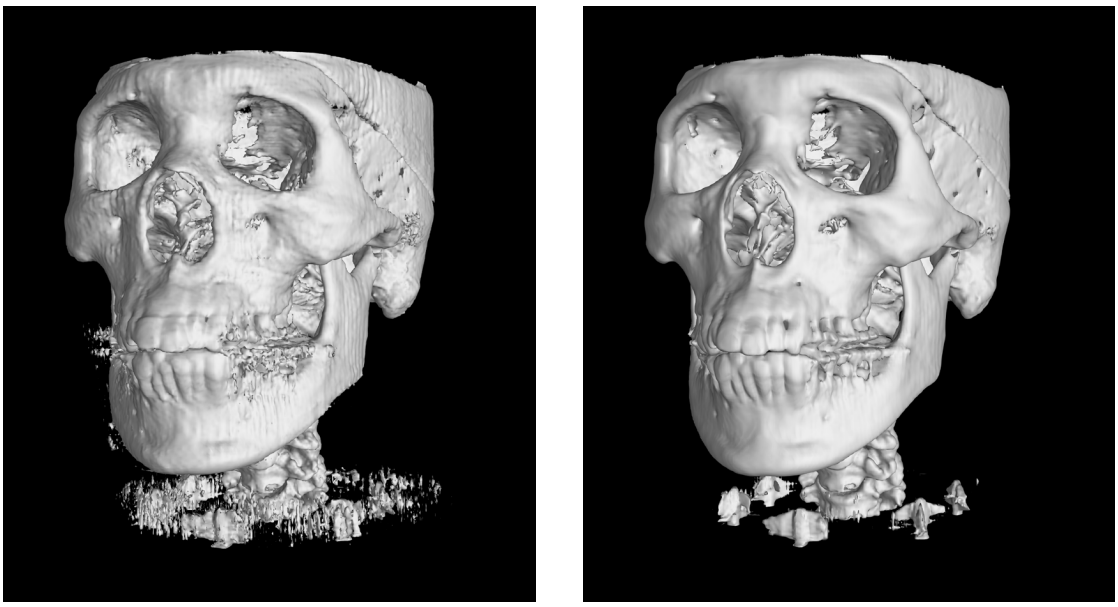
way the reconstruction bounds are designated in OSCaR makes it difficult to produce an identical output volume. Thus, the cross section shown in the figure might be at a slightly different coordinate. The edge of the reconstructable cylinder is sharper in our reconstruction because we do not reconstruct parts that lie outside of this cylinder.

Our reconstruction appears to be notably sharper and less noisy. This is probably due to the fact that OSCaR only uses nearest neighbour interpolation [11], which introduces aliasing and chessboard-like noise. From the general appearance, however, the two reconstructions look very similar. Figures 6.22 and 6.23 show the two results rendered as isosurface models using two different thresholds, and Figure 6.24 shows a rendering of our reconstruction using volume ray casting.



OSCaR Custom implementation

Figure 6.22: The result of our custom implementation in comparison to that of OSCaR (isosurface rendering of the skin rendered with VGStudio)



OSCaR Custom implementation

Figure 6.23: The result of our custom implementation in comparison to that of OSCaR (isosurface rendering of the bones rendered with VGStudio)

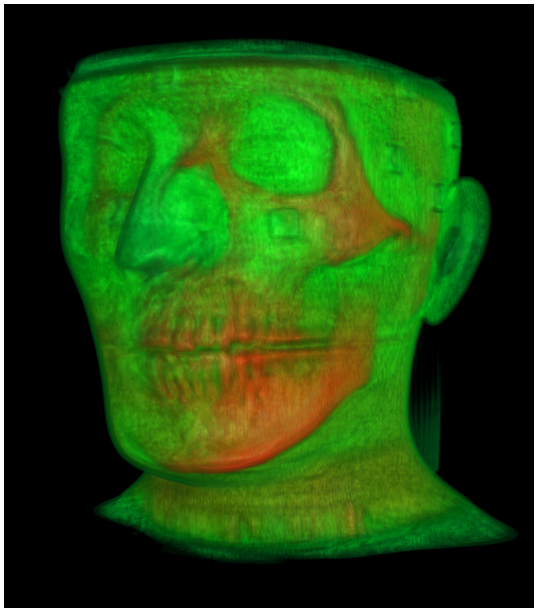


Figure 6.24: Volume rendering of our reconstruction of the skull phantom. Low density materials are shown in green, high density materials in red (rendered using VGStudio).

6.5 Measurement Inaccuracies

As usual, the measurements we took showed a certain variance. Most of the time these were only minor differences. At one point the results of multiple tests were averaged; in this case it is noted in the text. Generally, the values represent best case results.

We could make out some factors that had a deterministic influence on performance. For example, on the consumer-grade system the GPU execution was fastest when it was freshly booted. After some uptime, when multiple other applications were running, the measured execution times became longer. This might be due to the fact that today general programs, such as web browsers, utilise the GPU as well. Consequently, our general advice is to avoid running unnecessary applications when the aim is achieving top-level performance.

The server-grade machine was a multi-user system, which is another circumstance that could have had an influence on measurement fidelity. Unfortunately, it was not possible to have the system at hand exclusively for our purposes. Thus, measurements taken at different points in time might expose slight differences.

At a few points, the results of the same tests on the same hardware show slightly different values. This inconsistency is due to the fact that we always ran test instances which were meant to be compared directly to each other in one batch, for better comparability and to ensure equal preconditions. The same test might therefore have been run multiple times, yielding slightly different outcomes each time.

After the composition of this written work further optimisations have been made to the program. Therefore, the performance might be different in the current revision. Generally, we can say that it has improved. The performance measurements included in this chapter are based on revision 539, except for the comparison to OSCaR, which is based on revision 585.

7 Conclusion and Future Work

Considering the results, we are quite content with how well we succeeded in reaching our goals. The parallelisation options we implemented have proven to be quite effective. The CPU parallelisation provides a significant speedup compared to singlethreaded CPU execution, GPU processing gives an additional increase in performance over parallelised CPU execution, and multi-GPU execution can further reduce these execution times proportionally to the amount of utilised GPUs. Ultimately, we are able to reach execution times of approximately 10 minutes for large volumes on potent systems.

While the performance is high, the quality level of the reconstructions produced by commercial solutions is still slightly ahead of our results. However, rendered as an isosurface model or through volume raycasting, the difference in the results is scarcely noticeable.

What is most important to us is that we created an openly available solution that is capable of reconstructing virtually any cone-beam CT-scan, given that the parameters are known. Basic viewing capabilities are also provided, not only for the volumes saved by our application but also for any raw volume, as long as information about the volume's data type and dimensions are known. The support for *VGI* sidecar files also enables potential users to view and analyse our reconstructions in third-party applications like *VGStudio*.

Furthermore, the results of this thesis create a foundation for further research; a foundation that can be extended to improve on old or meet further new requirements. This could, for example, be quality enhancements to the reconstruction algorithm or postprocessing techniques like artifact reduction. Completely different reconstruction approaches could be put to the test, as well. The GPU implementation could be extended to support not only the CUDA but also the OpenCL API, resulting in support for AMD graphics cards as well. The builds and build configuration scripts that are currently available for Microsoft Windows and Linux could be extended to add out-of-the-box support for Apple Macintosh.

It might also be desirable to convert the resulting volumes to other data types for the sake of saving memory or to provide compatibility with other applications. An important point that should be investigated in the future is the intelligent weight distribution between multiple different GPUs when running a multi-GPU reconstruction. The performance could also be optimised further. Approaches using curved voxels, as proposed in [5], could possibly be investigated. Precomputed mapping tables for the s- and t-coordinates might also help to improve performance, although it is questionable whether this will provide any advantage on the GPU, where accesses to global memory are very expensive and should be minimised. Apart from these internal optimisations, options that favour speed over quality, such as nearest-neighbour interpolation instead of bi-linear interpolation, could be provided as well. Hybrid parallelisation is not a good option for this algorithm, as the majority of the computational effort is required for the backprojection. The only conceivable possibility is letting the CPU do the image preprocessing, which has not proven beneficial thus far. Lastly, the evaluation of the quality of the obtained results should be extended to data sets where all information, including the exact angles of the individual projections, are available.

Bibliography

[1] P. Suetens. “X-ray computed tomography”. In: *Fundamentals of Medical Imaging*. Cambridge University Press, Aug. 2009. Chap. 3.

[2] W. C. Röntgen. *Über eine neue Art von Strahlen*. 1896. URL: http://www.xtal.iqfr.csic.es/Cristalografia/archivos_10/Uber_eine_neue_art_von_strahlen_ocr.pdf (visited on 08/21/2016).

[3] M. Sakamoto et al. *An Implementation of the Feldkamp Algorithm for Medical Imaging on Cell*. Tech. rep. IBM Corporation, 2005. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.90.9210&rep=rep1&type=pdf (visited on 08/23/2016).

[4] J. Radon. “Über die Bestimmung von Funktionen durch ihre Integralwerte längs gewisser Mannigfaltigkeiten”. In: *Akad. Wiss.* Vol. 69. 1917, pp. 262–277. URL: http://people.csail.mit.edu/bkph/courses/papers/Exact_Conebeam/Radon_Deutsch_1917.pdf (visited on 08/21/2016).

[5] A. Shih, G. Wang, and P.-C. Cheng. “Fast Algorithm for X-ray Cone-beam Microtomography”. In: *Microscopy and Microanalysis* 7 (01 Jan. 2001), pp. 13–23. ISSN: 1435-8115. URL: http://journals.cambridge.org/article_S1431927601010005 (visited on 08/23/2016).

[6] Bauhaus-Universität Weimar. *Fakultät Bauingenieurwesen weiht Nano-CT-System ein*. Oct. 2014. URL: https://www.uni-weimar.de/en/civil-engineering/news/news/titel/fakultaet-bauingenieurwesen-weiht-nano-ct-system-ein/ (visited on 07/18/2016).

[7] L. A. Feldkamp, L. C. Davis, and J. W. Kress. *Practical cone-beam algorithm*. June 1984. DOI: 10.1364/JOSAA.1.000612. URL: http://josaa.osa.org/abstract.cfm?URI=josaa-1-6-612 (visited on 08/23/2016).

[8] T. M. Buzug. *Computed Tomography - From Photon Statistics to Modern Cone-Beam CT*. Springer, 2008.

[9] A. Kak and M. Slaney. *Principles of Computerized Tomographic Imaging*. Society for Industrial and Applied Mathematics, 2001. DOI: 10.1137/1.9780898719277.

[10] University of Toronto. *OSCaR - An Open-Source Cone-Beam CT Reconstruction Tool for Imaging Research*. URL: http://www.cs.toronto.edu/~nrezvani/OSCaR.html (visited on 07/18/2016).

[11] N. Rezvani et al. *OSCaR: Open Source Cone-beam Reconstructor*. URL: http://www.cs.toronto.edu/~nrezvani/OSCaR-02.zip (visited on 07/27/2016).

[12] Nvidia. *CUDA C Programming Guide*. Sept. 2015. URL: http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html (visited on 07/13/2016).

[13] M. Wolfe. *Understanding the CUDA Data Parallel Threading Model*. Feb. 2010. URL: https://www.pgroup.com/lit/articles/insider/v2n1a5.htm (visited on 07/07/2016).

[14] P. Micikevicius. *Fundamental Optimizations*. Nov. 2010. URL: http://www.nvidia.com/content/PDF/sc_2010/CUDA_Tutorial/SC10_Fundamental_Optimizations.pdf (visited on 07/07/2016).

[15] M. Kitagawa. *Coordinate Systems*. URL: www.utdallas.edu/atec/midori/Handouts/coordinate_systems.pptx (visited on 07/13/2016).

[16] K. R. Rao and P. C. Yip. "The Discrete Fourier Transform". In: *The Transform and Data Compression Handbook*. CRC Press, 2001. Chap. 2.

[17] M. Harris. *How to Optimize Data Transfers in CUDA/C++*. Dec. 2012. URL: https://devblogs.nvidia.com/parallelforall/how-optimize-data-transfers-cuda-cc/ (visited on 07/13/2016).

[18] J. Siewerdsen. *3D Filtered Backprojection - Fundamentals, Practicalities, and Applications*. URL: http://www.aapm.org/meetings/amos2/pdf/59-17243-37526-878.pdf (visited on 07/07/2016).

[19] O. Képalkotás. *Medical Imaging - Reconstruction*. URL: http://oftankonyv.reak.bme.hu/tiki-index.php?page=Reconstruction (visited on 07/07/2016).

[20] A. Malecki and J. Herzen. *X-Ray Computed Tomography*. Oct. 2015. URL: https://www.ph.tum.de/academics/org/labs/fopra/docs/userguide-79.en.pdf (visited on 07/07/2016).

[21] Itseez. *OpenCV 3.1.0*. URL: http://docs.opencv.org/3.1.0/d2/de8/group__core_array.html#gadd6cf9baf2b8b704a11b5f04aaf4f39d (visited on 08/11/2016).

[22] O. Képalkotás. *Medical Imaging - Realization of the filtered backprojection*. URL: http://oftankonyv.reak.bme.hu/tiki-index.php?page=Relaization%20of%20the%20filtered%20backprojection (visited on 07/07/2016).

[23] N. Whitehead and A. Fit-Florea. *Precision & Performance: Floating Point and IEEE 754 Compliance for NVIDIA GPUs*. Tech. rep. Nvidia Corporation. URL: http://developer.download.nvidia.com/assets/cuda/files/NVIDIA-CUDA-Floating-Point.pdf (visited on 08/23/2016).

[24] Nvidia. *NVIDIA CUDA Library Documentation*. URL: http://horacio9573.no-ip.org/cuda/group__CUDART__MEMORY_g6ee90dad01ae562e08405ce8131bbdc5.html#g6ee90dad01ae562e08405ce8131bbdc5 (visited on 07/13/2016).

[25] Nvidia. *CUDA C Best Practices Guide*. Sept. 2015. URL: http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html (visited on 08/23/2016).

[26] P. Nee. *Introduction to GPGPU and CUDA Programming*. July 2013. URL: https://cvw.cac.cornell.edu/(X(1)S(3js5tgj2xjetmelgkggkpt01))/gpu/coalesced?AspxAutoDetectCookieSupport=1 (visited on 07/07/2016).

[27] Rennich S. Luitjens J. *CUDA Warps and Occupancy*. Dec. 2011. URL: http://on-demand.gputechconf.com/gtc-express/2011/presentations/cuda_webinars_WarpsAndOccupancy.pdf (visited on 07/07/2016).

List of Figures

Statement of Authorship

I hereby certify that this master's thesis has been composed by myself and describes my own work, unless otherwise acknowledged in the text.

Place, date

Signature