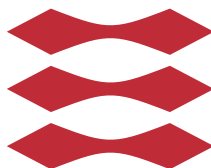
Block Algebraic Methods for 3D Image Reconstructions on GPUs

Kenneth Kjær Nielsen



Kongens Lyngby 2014

Technical University of Denmark
Department of Applied Mathematics and Computer Science
Matematiktorvet, building 303B,
2800 Kongens Lyngby, Denmark
Phone +45 4525 3351
compute@compute.dtu.dk
www.compute.dtu.dk

Resumé

Der bliver i denne rapport udforsket muligheden for at lave effektiv tredimensionel tomografisk rekonstruktion på *General Purpose Graphical Processing Units* (GPGPU). Tomografi er en teknik til at rekonstruere det indre af et objekt ud fra målinger af absorberet energi fra projektioner taget i forskellige vinkler. Når dette udregnes på en computer betegnes det som *Computed Tomography* (CT).

Dette vil ofte lede til et underbestemt ligningssystem, der derfor er yderst følsomt over for støj. Arbejdet i denne rapport fokusere på implementeringen af en familie af metoder som indeholder naturlig regularisering i form af antallet af iterationer.

Der blev taget udgangspunkt i en *open source* pakke kaldet *ASTRA*, som indeholder flere implementeringer til at lave tomografisk rekonstruktion på GPGPU'er. Én af metoderne derfra (SIRT) kan betragtes som et specielt eksempel på de metoder som rapporten fokusere på. Ved at bruge de metoder som er beskrevet i denne rapport er det muligt at lave tomografisk rekonstruktion væsentligt hurtigere en med den mest sammenlignelige metoder fra ASTRA.

Implementeringen blev lavet i CUDA C og C++, og indeholder wrappers som eksponere koden til matlab.

Summary

This report investigates the possibility of making effective three dimensional tomographic reconstructions on *General Purpose Graphical Processing Units* (GPGPU). Tomography is a technique to recreate the interior of an object, from measurements of absorbed energy from projections taken at different angles. When these calculations are done on a computer it is called *Computed Tomography* (CT).

This will often lead to a underdetermined system of equations which therefore is highly influenced by noise. The work described in this report will focus on the implementation of a family of methods which contains a natural regularisation in terms of the number of used iterations.

As a starting point there was used an *open source* package called *ASTRA* which contains multiple methods for doing tomographic reconstructions on GPGPU's. One of these methods (SIRT) can be considered as a special case of the family of methods this report is focusing on. Using the methods described in this report it is possible to make faster tomographic reconstructions than by using the similar method from ASTRA.

The implementation was made in CUDA C and C++ and contains wrappers which exposes the code to Matlab.

Contents

Resumé i

Summary ii

1 Introduction 1
1.1 Structure of the Thesis . . . 2

2 Tomography 4
2.1 Semiconvergence . . . 5
2.2 SIRT . . . 6
2.3 ART . . . 7
2.4 Block iterative methods . . . 7
2.5 Approximations of the projection matrix . . . 8
2.5.1 Line length method . . . 8
2.5.2 Joseph's method . . . 9
2.5.3 Transposed method . . . 10

3 General test settings and hardware specifications 13
3.1 Test setup 1 . . . 13
3.2 Test setup 2 . . . 14
3.3 Hardware . . . 15

4 CUDA 18
4.1 Memory types . . . 20
4.1.1 Device memory . . . 20
4.1.2 Shared memory . . . 21
4.1.3 Constant memory . . . 22
4.1.4 Texture memory . . . 22
4.1.5 Surface memory . . . 23

5 Benchmarking CUDA memory reads and copy operations 24
5.1 Memory reads . . . 24
5.1.1 One dimension . . . 25
5.1.2 Two dimensions . . . 26
5.1.3 Three dimensions . . . 27
5.2 Copy operations . . . 29

6 ASTRA and analysis of the 3D SIRT implementation 33
6.1 General setup . . . 33
6.2 SIRT . . . 34
6.2.1 Implementation of the forward projection . . . 35
6.2.2 Backward projection . . . 37
6.3 Complexity analysis of SIRT implementation . . . 37
6.4 Angle dependency of the forward projection . . . 39

7 New implementations based on ASTRA 42
7.1 New SIRT implementation based on ASTRA . . . 42
7.2 SART implementation based on ASTRA . . . 44

8 New implementations based on new ordering, new boundary conditions, and shared memory 48
8.1 New forward projection using textures . . . 49
8.2 New back projection using textures . . . 52
8.3 New SART implementation . . . 53
8.3.1 Forward projection without textures . . . 57
8.3.2 Forward projection utilizing shared memory . . . 59
8.4 Implementation of a General Block-Iterative method . . . 60

9 Final results and comparison 64

10 Conclusion and future work 69

A Implementation of a general Block-Iterative method 70

Nomenclature 73

Bibliography 75

Chapter 1

Introduction

3D image reconstruction is a technique for reconstructing the interior of an object from measurements of its projections at different angles. The mathematical theory behind this technique is called tomography, and when performed on a computer it is referred to as computed tomography (CT)[4].

A projection is a detector image of the interaction between the object and radiation (X-ray, electronic, or optical) that is made to propagate through it.

CT is used in a wide range of applications such as biomedical imaging [6], geophysical prospecting [13], materials science [16], and probably the most important and well-known example; the medical CT-scanner [17].

CT image reconstruction techniques can be classified roughly into two categories; analytical reconstruction methods and algebraic iterative reconstruction methods. The analytical methods, such as filtered back projection (FBP) in 2D and the Feldkamp-Davis-Kress (FDK) method in 3D [8] are based on analytical formulas and can be either exact or approximated. The algebraic methods are based on work from the Polish mathematician Stefan Kaczmarz [11] and the Italian mathematician Gianfranco Cimmino [5], who independently developed iterative algorithms for solving linear systems. In 1970 Gordon, Bender and Herman rediscovered Kaczmarz's method applied in medical imaging [18] and called the method the algebraic reconstruction technique (ART). Another class

of algebraic iterative methods, of which Cimmino's method is an example, are called simultaneous iterative reconstructive techniques (SIRT) [1].

CT scanners of today use analytical reconstruction techniques. These techniques are computationally relatively cheap so that they can deliver real-time updated images, but they also require a large number of projections for a good reconstruction. This means, that the achievable image quality is directly related to the x-ray dose given to a patient, and in order to obtain an image of sufficiently high quality, a relatively high x-ray dose must be used.

Algebraic iterative methods have the ability to achieve improved imaging quality over analytical methods in terms of noise and artefact reduction and contrast enhancement, particularly for low-dose CT [17]. The improved images from algebraic techniques, however, come at the price of a much higher computational cost. Motivated by an increasing focus on the potentially harmful effects of CT-scans, and also the excess of data produced with many projections, e.g., in particle accelerators, a recent trend in CT research has been to develop algebraic techniques for modern computer architectures [19]. In particular, the approach of blocking the SIRT algorithm [7], e.g., such that each projection is handled separately (the so-called SART algorithm [2]) has proven to be advantageous.

This project is devoted to the development and implementation of fast block-algebraic iterative reconstruction methods to perform 3D image reconstruction using general purpose graphical processing units (GPGPUs).

We will adopt and expand an existing state-of-the-art package ASTRA for 3D image reconstruction in CUDA. This package will be our starting point and the key reconstruction routines of will be analyzed and benchmarked of different platforms and phantom test problems. We combine the two iterative methods ART and SIRT by following the approach for an efficient multi-threaded CPU implementation described in [19]. Finally, we will describe the design choices we have made in order to achieve our own best implementation and compare the performance of the different methods.

1.1 Structure of the Thesis

The aim of the current thesis is to present the step-wise development of our CUDA C block-iterative algebraic reconstruction implementation and compare its performance to the existing package ASTRA for different problem setups.

The chapters of the thesis are organized in the following way:

- Chapter 2: In this chapter we describe two main algebraic algorithms SIRT and ART for dealing with tomography, and their advantages and disadvantages, concerning convergence and parallelism
- Chapter 3: We describe the details of the test problem setups and the hardware specifications of the platforms used in this thesis.
- Chapter 4: In the next chapter we briefly describe key issues of the CUDA programming model and memory types and texture usage. For rigorous details on CUDA, however, we refer to the CUDA C best practices guide [14] and CUDA C programming guide [15].
- Chapter 5: In this chapter we present some basic benchmark test of the memory reads from textures in order to evaluate them regarding the random access pattern and compare to the device memory.
- Chapter 6: We introduce in this chapter ASTRA and analyze the existing fastest GPU implementation for 3D reconstruction.
- Chapter 7: This chapter discusses the first steps to make a new block-algebraic implementation based on ASTRA's 3D SIRT implementation.
- Chapter 8: This chapter describes the best implementation we could achieve and presents the performance results for the different test cases.
- Chapter 9: Here we will compare the methods in terms of total computing time.
- Chapter 10: Conslusion and future work.

To produce the test results, examples and figures a large number of scripts and code files have been created in MATLAB, C++, and CUDA C. These have been run using the following software: MATLAB R2013b (8.2.0.701), gcc version 4.8.1, and CUDA toolkit 5.5.

Chapter 2

Tomography

Tomographic reconstruction deals with the problem of reconstructing a density map of a given domain Ω from a finite set of projections where a known amount of energy is absorbed through the domain. In the continuous case the absorbed energy is expressed in terms of a transformation called the Radon transform [8], which maps a line $L \in \Omega$ into a real number, i.e. $L \to \mathbb{R}$.
Let $d(x)$ denote the density at the point $x \in \Omega$ then we can define The Radon Transform for any line $L \in \Omega$ as:

$$Rd(L) = \int_{x \in L} d(x) ds$$

The rays can be travelling in different configurations. We will focus on a configuration called *parallel beam* where all the rays are travelling in parallel with each other. Another method worth mentioning is called *cone beam* where all the rays a emitting from a single point and is projected onto a two dimensional plane.

In practice we will be dealing with a discrete domain Ω, which we will assume is uniformly spaced. We will refer to the elements in the projection as *pixels*, and the elements in the domain as *voxels*. The individual voxels will be arranged in a sequential matter. This means that for a test case of size: $sizeX \times sizeY \times sizeZ$

the element at location $i, j, k \in \Omega$ is found as:

$$x(i + j \cdot sizeX + k \cdot sizeX \cdot sizeY) = x(i, j, k)$$

The p projections of size $sizeU \times sizeV$ will be stored in a vector b is a similar fashion.

Instead of working directly with the Radon transform, we will be dealing with a linearised approximation A, which we will call the projection matrix. Different approximation schemes will be described in section 2.5. Using such an approximation, we can model the system as :

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

In all real life applications the projections are measured by some physical instrument, and is subjected to small errors ϵ.

$$Ax = b + \epsilon \tag{2.1}$$

There are many different methods for solving this type of problem, such as *Filtered Back Projection* (FBP) and *Feldkamp-Davis-Kress* (FDK) [8], least squares, bfgs. But if x is larger than b, these methods become highly influenced by the noise. We will instead seek to find a regularized solution.

2.1 Semiconvergence

In this section we will describe semiconvergence as a concept.

Two classic methods for solving (2.1) will be described in section 2.2 and 2.3. They are both iterative methods, and the regularization is controlled by the number of iterations. If the noise ϵ is not zero, they both have a special type of convergence called semiconvergence.

Semiconvergence can be explained with a test problem with a known solution $\hat{x}$. If there is semiconvergence the approximation x_k converges towards a fixed point $\tilde{x}$, but at some iteration we achieve a better approximation to the true solution $\hat{x}$ than the limit point $\tilde{x}$. So for some, $i \in \mathbb{N}$ we will see that $\|\hat{x} - x_i\| < \|\hat{x} - \tilde{x}\|$.

The semiconvergence property is illustrated in figure 2.1

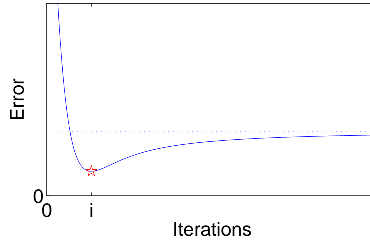


Figure 2.1: Illustration of semiconvergence. The red dot illustrates the best approximation, which is found at the i'th iteration.

2.2 SIRT

Simultaneous Iterative Reconstruction Techniques (SIRT) are a family of methods which are parallel in the individual equations, that is that in each iteration all the equations in A can be used at the same time. This mean these algorithms are parallel in the sense of a matrix vector product. These algorithms make use of two extra matrices $T \in \mathbb{R}^{n\times n}$ and $M \in \mathbb{R}^{m\times m}$, and a relaxation parameter λ. The algorithms is shown in Algorithm 2.1.

Algorithm 2.1: SIRT

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

These methods contains a fair level of parallelism, but compared to other methods they require many iterations before a good approximation is achieved. This means that even though the parallelism is well suited for the architecture of a GPGPU, the sheer number of iteration might cause the overall reconstruction time to be too high.

Different choices of T and M will lead to different methods, but as it is shown in [19], they all behave very similar as long as λ is set properly. We will therefore restrict ourselves to use, *Cimmino* weights, where $T = I$ and $M = \operatorname{diag}(1/\|a_i\|_2^2)$, and a_i denotes the i'th row in A.

We note that SIRT can refer to slightly different version in the literature, and that we refer to the method used in [19].

2.3 ART

Algebraic Reconstruction Technique (ART) is a reconstruction method which only uses a single row a_i of A in each update of the volume x. So in each iteration we will make m updates. This means that the parallelism in this method is restricted to the parallelism of a vector product. The algorithm is shown in Algorithm 2.2.

Algorithm 2.2: ART

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

The ART algorithm has an asymptotic convergence if $0 < \lambda < 2$, and is known for having a fast convergence rate [19].

2.4 Block iterative methods

We wish to combine the two methods, ART and SIRT, in a way that keeps the fast convergence from ART and still contains a high level of parallelism. To do this we will divide A into p sub matrices $A_i,\ i \in \{1, \cdots, p\}$ such that each sub matrix contains some of the rows from A. We will divide b into sub vectors in the same manner, this means that A and b can be written as:

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

We then combine the methods such that we make sequential updates, from each block, using SIRT updates. This algorithm can be seen in Algorithm 2.3.

A special case of this method is call SART [2]. In this method the number of blocks equals the number of projection angles. We note that if the number of blocks is set to one, the method is equivalent to the SIRT method.

Algorithm 2.3: BLOCKIT

$x^{k,0} = x^{k-1}$
for i in 1 to m
  $x^{k,i} = P_{\mathcal{C}}\left(x^{k,i-1} + \lambda T A^T M \left(b - A x^{k,i-1}\right)\right)$
end
$x^k = x^{k,m}$

2.5 Approximations of the projection matrix

We will now describe three different ways to make a linear approximations of the Radon Transformation. The first method is described because it is a widely used method, and because it is the one used in [19]. The two following methods is used by ASTRA and those we will make use of in our implementations. Further approximations can bee seen in [9].

General for all the methods is that each ray will define a single row a_i in A, and we will use vectors u and v to span the projection plane.

2.5.1 Line length method

The first approximation scheme we will describe is called *line length*. Here the elements $a_{i,j}$ in A is set to be the distance each ray is travelling inside each voxel. This is illustrated in figure 2.2.

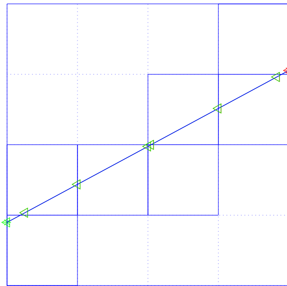


Figure 2.2: Illustration of the *line length* approximation scheme. The elements $a_{i,j}$ in a row a_i is set to the length the ray travels through each voxel.

This method contains a lot of branching, in order to determine which voxels are traversed by a ray. This is not a problem for a CPU implementation, but it is not efficient on a GPGPU.

2.5.2 Joseph's method

The second method was introduce by Joseph[10], and it is the one used in ASTRA when they want to use A as Ax (forward projection). In this method we find the dominant direction (the axis in which the ray are travelling fastest). We then traverse the volume in this dominant direction. In each step we set the elements $a_{i,j}$ in a_i to be the weights used in linear interpolation between the neighbouring voxels.

This is illustrated in figure 2.3

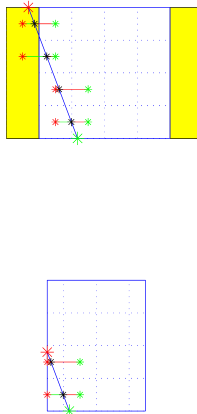


Figure 2.3: Illustration of the Joseph'method in 2d, imposed with two different boundary conditions. The method trace a ray through the volume with unit steps and then sets the elements in a row a_i of A, to the bilinear interpolation weights of the neighbouring voxels. Left image has zero padded element surrounding the volume, which effectively increases the volume with a half voxel to each side,*expanding boundary*. Right image sets all values to zeros which have passed the center of the bordering voxels, *clamping boundary*.

It is possible to impose different boundary statement on this method. In ASTRA

they zero-pad all around the volume, which means that given a point just out side the volume, they will interpolate between zero and the nearest points in the volume, see figure 2.3, we call this method *expanding boundary* condition. A different approach could be to set the points that have passed the center of the boundary voxel, to zero, because it is not contained in the volume. We will refer to this as *clamping bounday* condition. This second approach is cheaper in terms of memory footprint, but it decreases the effective size of the voxels on the boundary see figure 2.3. We will make use of both boundary conditions later on in the report.

There is a third simple condition which could have been utilized, where points near the boundary is interpolated to a zero element if the interpolating point is inside the volume. But we will not utilize this method.

2.5.3 Transposed method

The final approximation is used in ASTRA when they use it translated as $A^T b$. They do not state explicit that they use a different version for back projection, but in [9] they augment for the use of a voxel driven approach in the back projection. The method calculates the projection of each voxel center in the projection plane, and then make a linear interpolation between the nearest pixels.

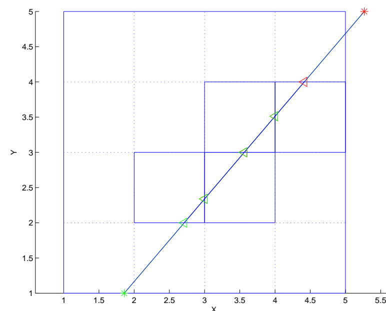


Figure 2.4: Illustration of the *transposed method*, which is an approximation scheme for the transpose of the projection matrix, i.e. A^T.

Again it is possible to impose different boundary conditions on this method. Again ASTRA has chosen to zero pad along the boundary, so if the projection is defining a point just outside the projection plane ASTRA will interpolate with zero. We will again refer to this kind of boundary condition as an *expanding boundary* condition. Another way would simply discard the projection because

it is not in the projection plane, like in the *Joseph's* method. We will use both methods later in the report.

2.5.3.1 Projection of a voxel into the plane

We will now describe how a voxel is projected into the projection plane. We will do this by finding the projection of the voxel along the v-vector direction, and note that the projection along the u-vector is done in a similar fashion.

Assume that the projection direction (typically the direction of a ray) is given by $f = (rayX, rayY, rayZ)$.

For a given plane with normal direction r which intersects a point x_0 we can compute the distance from a point x to the plane plane multiplied with the length of the normal vector as:

$$r(x - x0) = \|r\| \, \|x - x0\| \cos(r, x - x0) \tag{2.2}$$

We will first span a plane between f and u, and define a unit normal vector to this plane as:

$$n = \frac{u \times f}{\|u \times f\|}$$

And then project v into this normal direction n, which is given by

$$v_n = \frac{vn}{\|n\|^2} n$$

We want to find a number c such that x is projected into cv and note that this number can be computed from equation 2.2 by introducing a vector r with same direction as v_n and the length $\frac{1}{\|v_n\|}$. This vector can be computed as:

$$\begin{aligned} r = & \frac{v_n}{\|v_n\|^2} = \frac{\frac{vn}{\|n\|^2} n}{\left\| \frac{vn}{\|n\|^2} n \right\|^2} = \frac{\frac{vn}{\|n\|^2} n}{\frac{\|vn\|^2}{\|n\|^4} \|n\|^2} \\ = & \frac{vn}{\|vn\|^2} n = \frac{n}{vn} = \frac{\frac{f \times u}{\|f \times u\|}}{\frac{v(f \times u)}{\|f \times u\|}} = \frac{f \times u}{v(f \times u)} \end{aligned} \tag{2.3}$$

So we finally have that:

$$c = r(x - x0) = r \cdot x - r \cdot x0 = r \cdot x - d, \qquad d = -r \cdot x0$$

This means that the projection of a voxel, into the v-vector can be done in terms of a dot product and a subtraction.

Chapter 3

General test settings and hardware specifications

In this chapter we will describe the general test setups and the hardware platforms used to test the implementations.

There will be used two general test setups, and these will be described first.

3.1 Test setup 1

The first test setup consists of a volume of $N^3 = 256^3$ voxels, $p = 133$ projections with $m^2 = 256^2$ pixels. This test setup calculates 133 ray directions, using *Lebedev quadrature* [12]. This ensure that the directions are uniformly distributed over the unit sphere.

We then span a projection space perpendicular to the ray direction. We choose a basis for the projection space, as a u and a v vector which we calculate using Rodrigues' Rotation Formula [3].

This test setup ensures that there is used a high variety of projections, in terms of projection angles and orientation of the projection space. This is illustrated

in figure 3.1 where the u and v vectors are shown emitting from the ray vector. In figure 3.2 we see we have segmented the ray directions, by their dominant direction, and shown the u and v vectors projection into the space perpendicular to the dominant direction.

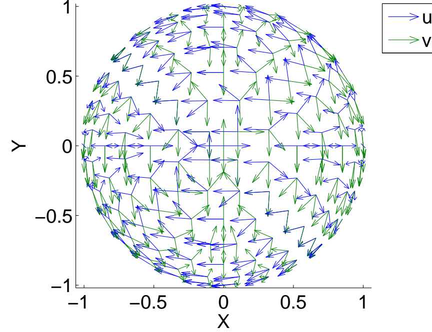


Figure 3.1: Illustration of the projection spanned with Lebedev quadrature, and the projection space which is calculated by Rodrigues' Rotation Formula.

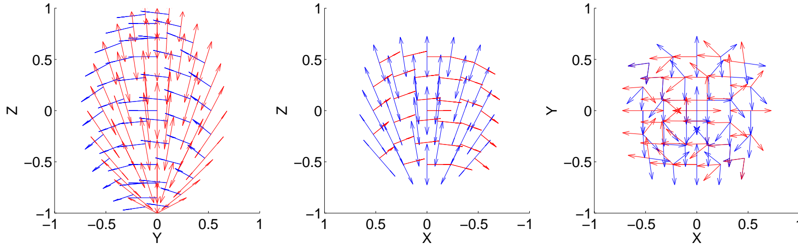


Figure 3.2: Illustration of the projections used in test setup 1, segmented into their dominant directions. The u and v vector is then projected into the space perpendicular to the dominant direction.

3.2 Test setup 2

The second test is done in three different sizes, namely:

- Small test consists of a volume of size 128^3 and 66 projections each of size 128^2

- Medium test consists of a volume of size 256^3 and 133 projections each of size 256^2.
- Large test consists of a volume of size 512^3 and 300 projections each of size 512^2.

The projection angles are set from # projections, uniformly distributed numbers t in the interval $[0, \pi]$, where :

xy-plane

$$f = \begin{pmatrix} \sin(t) \\ -\cos(t) \\ 0 \end{pmatrix}, \qquad s = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}, \qquad u = \begin{pmatrix} \cos(t) \\ \sin(t) \\ 0 \end{pmatrix}, \qquad v = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$$

xz-plane

$$f = \begin{pmatrix} \sin(t) \\ 0 \\ -\cos(t) \end{pmatrix}, \qquad s = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}, \qquad u = \begin{pmatrix} \cos(t) \\ 0 \\ \sin(t) \end{pmatrix}, \qquad v = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}$$

yz-plane

$$f = \begin{pmatrix} 0 \\ \sin(t) \\ -\cos(t) \end{pmatrix}, \qquad s = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}, \qquad u = \begin{pmatrix} 0 \\ \cos(t) \\ \sin(t) \end{pmatrix}, \qquad v = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}$$

This test setup gives less variation in the $u - v$ plane for neighbor directions compared to test setup 1.

3.3 Hardware

We use two computer platforms for testing, these can be seen in table 3.1.

We have utilized NVIDIA's test script to measure the effective memory bandwidth for different copy operations. This result is seen in table 3.2.

Table 3.1: Hardware specifications

		gpulab03	gpulab06
	OS	Ubuntu 10.04.4 LTS	Ubuntu 10.04.4 LTS
CPU:	Name:	Intel Core i7 CPU 930	Intel Core i7-3820
	Architecture:	x86_64	x86_64
	CPU op-mode(s):	32-bit, 64-bit	32-bit, 64-bit
	CPU(s):	8	8
	Thread(s) per core:	2	2
	Core(s) per socket:	4	4
	CPU socket(s):	1	1
	NUMA node(s):	1	1
	Vendor ID:	GenuineIntel	GenuineIntel
	CPU family:	6	6
	CPU family:	6	6
	Model:	26	45
	Stepping:	5	7
	CPU MHz:	2801.000	3600.000
	Virtualization:	VT-x	VT-x
	L1d cache:	32K	32K
	L1i cache:	32K	32K
	L2 cache:	256K	256K
	L3 cache:	8192K	10240K
GPU 0	Name	Tesla C2050	Tesla K20c
	Memory	2687 MB	4800 MB
	Multiprocessors MP	14	13
	CUDA Cores pr MP	32	192
	CUDA cores	448	2496
	GPU Clock rate:	1147 MHz (1.15 GHz)	706 MHz (0.71 GHz)
	Memory Clock rate:	1500 Mhz	2600 Mhz
	Memory Bus Width:	384-bit	320-bit
	L2 Cache Size:	786432 bytes	1310720 bytes
GPU 1	Name	GeForce GT 240	Tesla K20c
	Memory	1023 MB	4800 MB
	Multiprocessors	12	13
	CUDA Cores pr MP	8	192
	CUDA cores	96	2496
	GPU Clock rate:	1340 MHz (1.34 GHz)	706 MHz (0.71 GHz)
	Memory Clock rate:	1700 Mhz	2600 Mhz
	Memory Bus Width:	128-bit	320-bit
	L2 Cache Size:		1310720 bytes

Table 3.2: Results from *NVIDIA*s test function "bandwithTest.cu".

	gpulab03	gpulab06
host to device	5224.6MB/s	5527.0MB/s
device to host	4357.5MB/s	5018.2MB/s
device to device	103774.8MB/s	144043.5MB/s

Chapter 4

CUDA

Graphical Processing Unit (GPU) evolved from a large customer demand to make high resolution of rendering in two and three dimensions. These tasks are very computational demanding, but highly parallel. This made the GPU's to evolve to especially be efficient to handle embarrassingly parallel or at least highly parallel workload efficiently. As the GPU's grew in processing capability it became clear that this computational power could be utilized to also do computations as a *General Purpose Graphical Processing Unit* (GPGPU). This came from a idea that all the computational powers of the computer should be utilized, and made *NVIDIA* develop a vendor specific package called *Compute Unified Device Architecture* (CUDA), which was first introduced in 2007. CUDA is an extension to the C programming language, and makes it possible for programmers to utilize the processing power of a NIVDIA graphic card.

All CUDA programs consists of two parts. A part that runs on the *Central Processing Unit* (CPU) which is called the *host* and a parts that runs on the GPU which is called the *device*. The host code is controlling the entire program by invoking different parts of the device code at the appropriate places.

The device code is managed through a grid system, where the threads are grouped into blocks which is scheduled to run on the *Multiprocessors* (MP). The threads within a block can run in parallel on one multiprocessor. The MPs are designed to execute hundreds of threads concurrently, and it divides each

block into warps of 32 threads. The individual threads within a warp are called lanes. The MPs contain three types of caches, namely a constant cache, a texture cache and a cache shared between the *shared memory* and *L1* cache [20]. These memory types will be described in the following sections.

All the data that is needed for computations, must be transferred through the CPU and then through the PCI-e port to the *Dynamic Random Access Memory* (DRAM) on the graphic card, which is the main working memory on the GPU. This memory transfer can be a bottle neck because it has a low bandwidth of 8GB/s which makes it the slowest bandwidth in the system. The transfer rates and typical latencies within the GPU is illustrated in figure 4.1. Because of the low bandwidth on the PCI-e port, it is beneficial to minimize the data transfer between the host and the device.

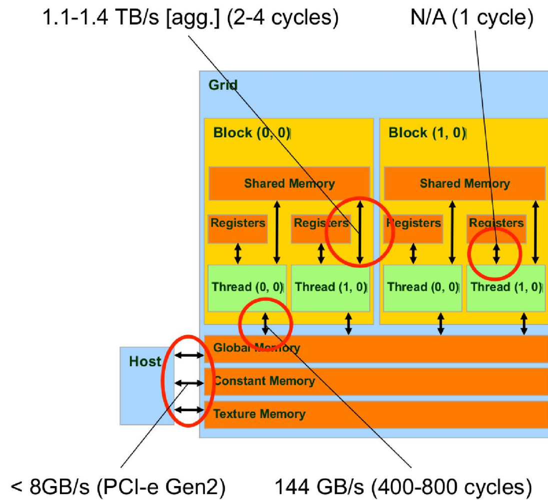


Figure 4.1: Illustration of the memory layout on the MPs, including transfer rates and typical latencies between the different types of memory. These numbers correspond to *gpulab03* in table 3.1.

A function which can be executed to run on the GPU is called a *kernel*. It is possible to make different kernels run concurrently on the GPU using *streams*. If there is created multiple streams, a kernel call or a copy operation can be set to run on different streams. These streams will then be delegated to run on the MP. It is possible to make synchronizations between the streams if there are some dependences that require so.

4.1 Memory types

On the GPU there is a set of different types of memory blocks, besides the DRAM. We will in this section present 5 types of memory and their properties.

4.1.1 Device memory

All the data in the RAM memory, is visible to all the blocks, and data requests are cached in a L1 cache on the MPs.

It is an advantage to use a one dimensional data layout. Special care should be taken if the data is needed in grid-wise fashion, such as with an volume or a projection. This follows from the way the device memory is accessed within a thread block. The device memory is accessed in memory transactions of a fixed size determined by the specific GPU architecture. Only segments whose first address is a multiple of the transaction size can be read in a single transaction. This is explained more detailed in [15, Section 5.3.2].

This transaction size has therefore an implication on how 2D and 3D data should be stored in a one dimensional device memory. To ensure the best alignment it is recommended to use pitched memory, which is memory with padded elements, in the first dimension, to meet this alignment issue. The new size of the first dimension is called the pitch. In figure 4.2 is shown an illustration of the memory use of 2D memory which is allocated in this fashion.

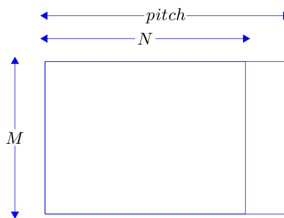


Figure 4.2: Illustration of typical the memory consumption for a domain of size $N \times M$, when using pitched memory. Here the pitch is illustrated in elements, it is normally given in bytes.

Pitched memory can be allocated by the CUDA function *cudaMallocPitch()* which takes a pointer to the data and a *size_t* element in which the pitch size is returned. It can also be stored in a *cudaPitchedPtr* type, which contains:

- *size_ t* pitch, given in bytes
- *void ** ptr, contains a pointer to the data. To use this pointer it is necessary to recast it as a pointer to the data type.
- *size_ t* xsize, Logical width of allocation in elements
- *size_ t* ysize, Logical height of allocation in elements

Given a 3D data set of size X, Y, Z which is contained in a array with a pointer *ptr* to pitched memory of floating point precision. Then the retrieve the (i, j, k)'th element from the pointer, one would have to do:

$$ptr\left[i + (j + k * Y)\frac{\texttt{pitch}}{\texttt{sizeof(float)}}\right]. \tag{4.1}$$

A *cudaPitchedPtr* can be allocated with the CUDA function *cudaMalloc3D()*. This takes a *cudaPitchedPtr* pointer and an *extend* element which contains the size of the desired element.

To achieve the best performance it is important that the threads within a warp is accessing neighbouring elements, this allows the transactions to be collected in a minimal number of *cache lines*. A cache line is 128 bytes and is the minimal size of cache memory that will be affected during a transaction. Such access pattern where neighbouring threads is accessing neighbouring elements is called *coalesced*.

4.1.2 Shared memory

Shared memory (SM) is residing in an on-chip 64KB memory block which is divided between SM and the L1 cache. The memory is divided such that there is 16KB for either memory type and 48KB to the other type.

The shared memory is shared between a block, and can be used to communicate between threads within a block. Because it is fast, and visible for the threads within the same block it makes sense to think of it as a manually managed cache memory.

There is a small setback regarding synchronization, because all threads has to be "ready" for it to be used. This require that there is set a synchronization point using *_ syncthreads()* which only affects threads within the same block.

4.1.3 Constant memory

The constant memory is cached memory which is visible to all the blocks. It is read only, which means that the memory has to be set from the host. Because it is cached and shared over all blocks it is ideal to store common parameters that is used by all blocks.

4.1.4 Texture memory

Textures was build into GPUs to make more realistic looking objects. These objects could as an example be rendered using re-sampling where the colours is calculated using a simple interpolation. In general the texture hardware consist of a computational pipeline which can be modified to use a set of the following elements:

- Scale normalized texture coordinates
- Perform boundary computations
- Convert texture coordinates to addresses with 2D or 3D locality
- Fetch 2, 4 or 8 texture elements for 1D, 2d or 3D textures and linearly interpolate between them

This pipeline is taken from [20].

Textures were introduced to CUDA because they allows the programmer to utilize an extra cache memory, and to use the elements in the pipeline. So textures can be set up in several ways, which utilizes different parts of the pipeline but they are only able to do *read only* operations.

There is designed a data structure called *cudaArray* which is optimized for 3D spatial locality, through some unspecified space filling curve [20]. This structure is designed to be bound to textures or surface memory.

It is possible to bind one, two or three dimensional data to a texture, and for one and two dimensional data it is also possible to bind sequential device memory such as pitched memory. But three dimensional data has to be transferred into a *cudaArray*, before it is bound to a texture.

4.1.5 Surface memory

Surface memory uses the same cache as the texture memory. It can be used for both read and writhe operations, but it dos not allow for the use of the interpolation as the texture memory do. Like the texture it can be bound to a *cudaArray*, where is has to be specified that is for surfaces. This means that if the memory reads do not follow the correct access patterns for device memory, but has some locality in the fetches, there can be achieved a higher bandwidth [15, Section 5.3.2].

CHAPTER 5

Benchmarking CUDA memory reads and copy operations

In this chapter, we present some benchmark tests of CUDA memory reads and writes. These operations are crucial for the algorithms considered later in the thesis in terms of performance. The memory access patterns required when tracing a ray through a 3D object of voxels corresponds to a strided accesses in sequential memory and depends very much on the ray direction. Here we study general strided memory accessing when using textures and compare to similar accesses in regular device memory. We will also make benchmark tests of the required *cudaMemcpy3d()* CUDA function which will be used for copying data between host and device, and between device and device.

5.1 Memory reads

We start by making tests of the texture reference. Since we will be using the linear interpolation later on, this will be activated in all the following test. We will test the one and two dimensional textures on gpulab03, and the three

dimensional texture on gpulab06, described in table 3.1.

5.1.1 One dimension

First we will examine how the one dimensional textures compare to device memory and shared memory regarding the random access pattern. To test this we will invoke a single block where each lane makes a number of memory reads, and the reads are separated by a stride. A stride is a jump which skip a number of elements. The memory reads will be arranged such that the reads are as coalesced as possible. This is done by making the memory reads by each lane jump with a factor $blockSize * (stride + 1)$. The memory read pattern can be seen in figure 5.1. The test is designed such that the memory is only read through a single time. This means that the total size of memory will increase as a function of the stride. This will not effect the runtime because the allocation and copy operation is separated from the timing.

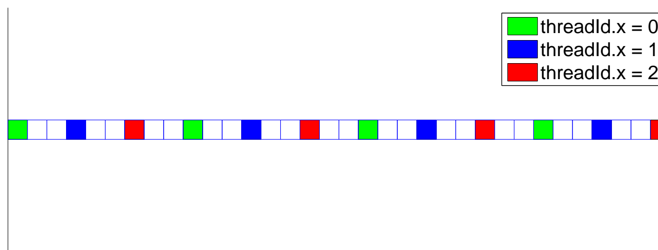


Figure 5.1: Illustration of the memory reads, with a stride 2 and a block size 3.

By increasing the size of the stride we will get an increasing number of cache misses, which will illustrate the influence random access has to the overall run time. We have run this test with different number of threads per block, and the result for 1 and 32 threads can be seen in figure 5.2. We note that in this test the elements are put into the shared memory and used immediately thereafter without any synchronization within the warp.

We see that for a low stride, the linear memory and the shared memory are faster than textures, but if the number of threads is high enough, the textures perform better when the stride is high since there are more cache misses. Because the shared memory is performing better for low strides, there will be some examples where there are some randomness to the access pattern, but where the shared memory outperforms the texture. The crucial point here is if the elements can

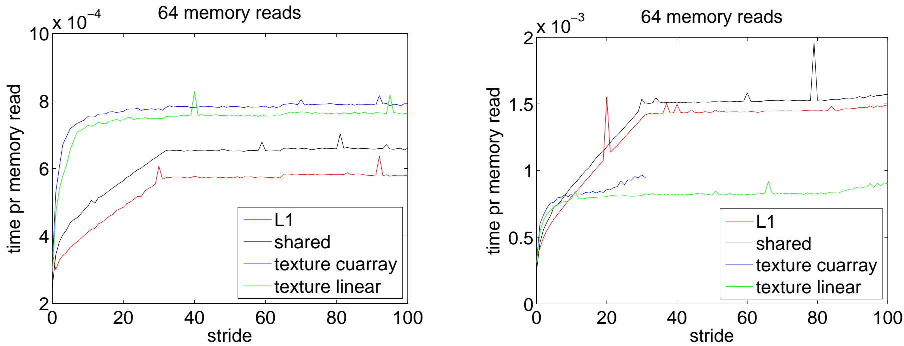


Figure 5.2: Benchmark test of one dimensional memory reads using L1 cache, shared memory and textures with and without the use of cudaArray. In the left figure we used a single thread and in the right figure we used 32 threads. Both tests are measured in milliseconds. This is tested on gpulab03.

be put coalesced into the shared memory, then we will expect that a shared memory approach will outperform the texture.

5.1.2 Two dimensions

When testing the two dimensional textures we will be focusing on the directions in which the data is traversed. And we will also test the performance compared to the number of threads within a block.

In order to test if the performance of the texture is influenced by which direction the data is traversed, we have set up a texture map and make data reads in the four directions, that is row wise from top and bottom, and column wise from left and right. The result is shown in figure 5.3, where we see that the spatial locality preformed by the space-filling curve is performing equally well in all four directions. We also note that the performance performance achieved by the texture which is bound to device memory is very comparable the the one bound to the cudaArray. This is very surprising because the spatial locality should be achieved by the index system within the cudaArray, and there should therefore be a difference in the number of cache misses on the device memory for the different angles. We have made these test with strides op to a size of 5 elements which gave the same result.

Since it is common to see an influence of the number of threads within a block

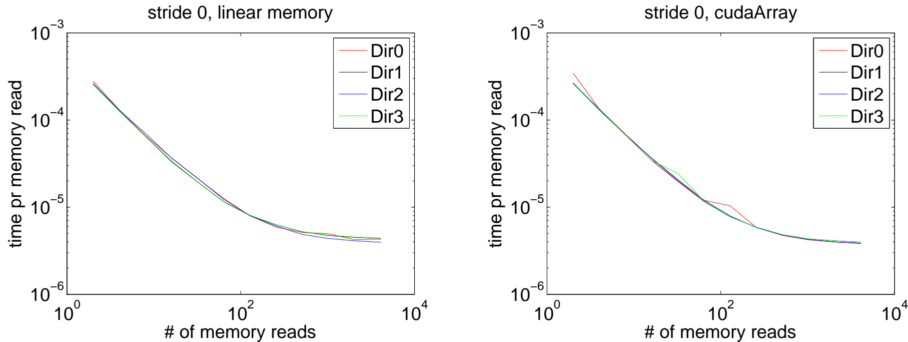


Figure 5.3: Benchmark test of the number of memory reads through a 2D texture in the four different directions. Dir0 is increasing in the second entry, Dir1 is increasing in the first entry, Dir2 is decreasing in the second direction and Dir3 is decreasing in the first direction. The left image is bound to linear memory , and in the right it is bound to a cudaArray. This is tested on gpulab03. All the tests is measured in milliseconds.

on the performance, as in the one dimensional case, we also test this for the two dimensional case. The results are shown in figure 5.4. From this we see that there is a significant influence by the number of threads within a block and that this behaviour is the same for all four directions when bound to a *cudaArray*. Again we note that the difference in performance by the texture bound to device memory and the *cudaArray* is surprisingly small, but we do note a small variation when the number of threads is around 176. For both bounding types we note that there seems to be an influence within the number of warps that the threats are divided into. This indicates that the number of threads within a block should be a multiple of 16 or 32, which coincide with a warp or a half warp. This is a very common dependence in CUDA programming, and explains some of jumping behaviour seen in figure 5.3.

5.1.3 Three dimensions

To test the three dimensional textures we will use a kernel, which traverse the whole data structure in either the X-axis, Y-axis or the Z-axis. This is done by taking a direction vector f and finding the dominant direction of f, i.e. the axis containing the largest absolute value of f. Then we normalize f with respect to the dominant direction. We then traverse the data in a two dimensional grid perpendicular to f in the dominant direction, where each element in the grid

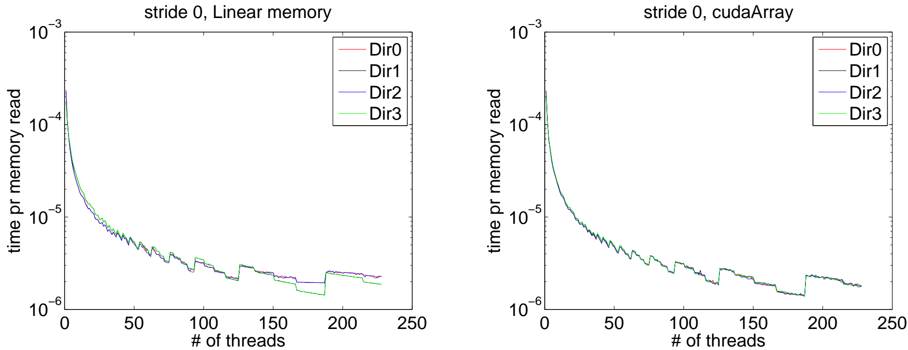


Figure 5.4: Benchmark test of the influence the number of threads within a block has on memory reads through a 2D texture. The test is done in the four different directions. Dir0 is increasing in the second entry, Dir1 is increasing in the first entry, Dir2 is decreasing in the second direction and Dir3 is decreasing in the first direction. The left image is bound to linear memory , and in the right it is bound to a cudaArray. All the tests is measured in milliseconds.

represent a thread. This ensures that all threads makes the same number of memory reads. All the elements that are read outside the data is set to zero by the texture reference, the result is shown in figure 5.5. This test is made from test setup 1.

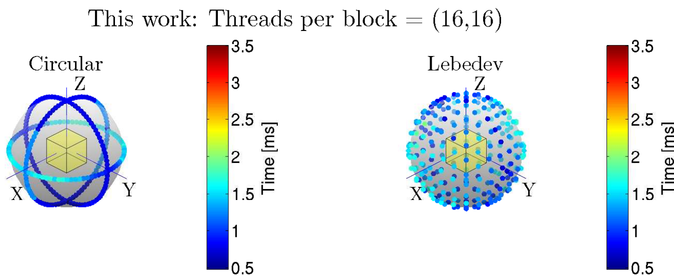


Figure 5.5: Bandwidth test texture reads through a 3D domain of 256^2 elements, as influenced by the direction. There is used a total of 256^2 threads divided into blocks of size of (16×16). This is tested on gpulab06

From figure 5.5 it is clear that there is a difference between the different directions. The best directions has a speed up of around a factor 2.4 compared

to the worst directions. More surprisingly we note that for rays travelling in a direction close to the $a - axis$ we see that a small change in the direction leads to a noticeable change in the performance. This change is related to the orientation of the two dimensional grid in which the rays are placed.

5.2 Copy operations

We will be using the *cudaMemcpy3D* function later on to copy a volume between the host and the device, and between different data structures on the device. We are therefore interested in measuring the performance of this function.

We will first test the performance in copying data from linear memory on the CPU into pitched memory on the GPU. There is a max bandwidth on the PCI-e port on 8 GB, for copy operations between the host and device, so we would expect a transfer rate below this. We will only be testing this from host to device and the result can be seen in figure 5.6, along with an interpolated line which estimates the bandwidth.

From figure 5.6 we see that the estimated bandwidth is approximately 5.16 GB/s on gpulab03 and 5.47 GB/s on gpulab06. So for both computers it is below 8GB as expected. Furthermore, we note using *NVIDIA*'s sample function "bandwith-Test.cu" that the bandwidth for *pageable* memory is slightly higher than what we achieved with *cudaMemcpy3D*. The results from "bandwithTest.cu" can be seen in table 3.2. Seeing this result we would also expect a slight decrease in the performance if we tested it from *device* to *host*.

Likewise it is of interest to see the performance on copyoperations from *cudaPitchedPtr* to a *cudaArray* and between to *cudaArray*s. The latter test is regarding *cudaArray* which can be bound to a surface to one that can be bound to a texture. The results are shown in figure 5.7 and figure 5.8. We note that both of these test is coping elements from the device to the device.

In both cases it is copy operations internally on GPU, so we would expect a comparable result to the result for *device to device* seen in table 3.2. But the achieved result show that the performance is off by a factor 12 on gpulab03 and 16 on gpulab06 which is much less than expected. Some of this difference could properly be due to difference in the ordering of the elements since the elements in the *cudaArray* is arranged in a different order than those in the *cudaPitchedPtr*. This means that the data copy can not be done coalesced.

Because the copy operation between the *cudaPitchedPtr* and the *cudaArray*

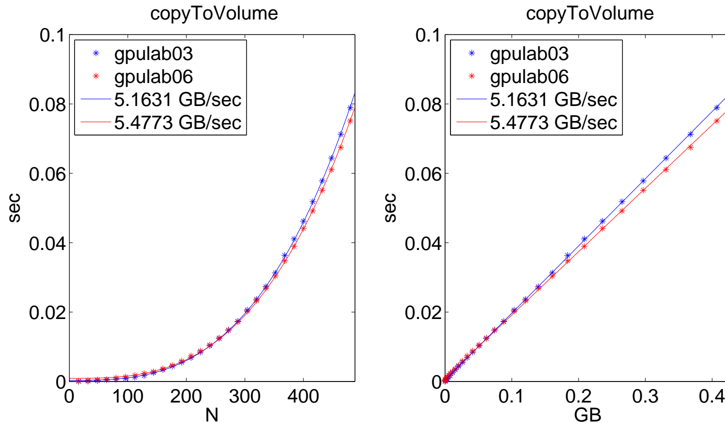


Figure 5.6: Bandwidth test of *cudaMemcpy3D*, transferring form linear memory on the host to a *cudaPitchedPtr* on the device. We are transferring a volume with sizes N^3. In the image on the left we see time as a function of N, and in the right figure we see the time as a function of the requested size in GB (not the pitched size). This copy operation is copying from host to device.

is so poor, it is of interest see if there is a better performance when copying between *cudaArray*'s where one can be bound to a surface and the other to a texture. This is because it is possible to write to surface memory. If the internally arrangement in the two *cudaArray*'s is similar, it should be possible to achieve a bandwidth similar to the one seen in table 3.2 for device to device. But the performance we have estimated in figure 5.8 is similar to the one in 5.7 and therefore off with the same factors. We conclude that if it is the copy bandwidth which is limiting the code it would not give a speed-up by storing the elements in a surface instead of a *cudaPitchedPtr*. We finally note that if the problem with coalesced copy was the explanation for the low bandwidth between a *cudaPitchedPtr* it indicates that the arrangement within the two *cudaArray* types also differs.

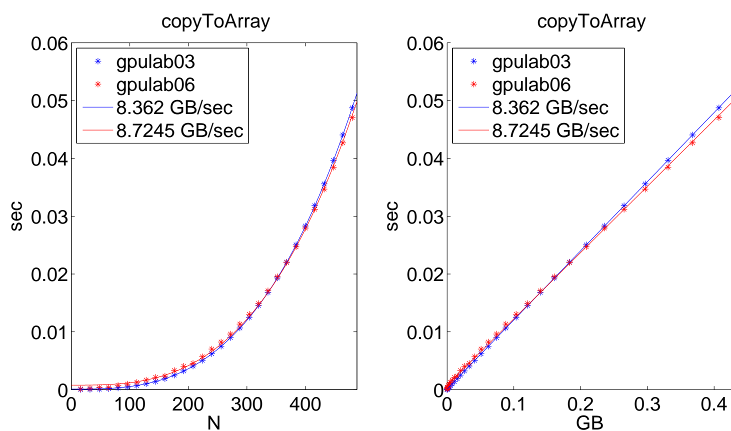


Figure 5.7: Bandwidth test of *cudaMemcpy3D*, transferring form linear memory on the device to a cudaArray on the device. We are transferring a volume with sizes N^3. In the image on the left we see time as a function of N, and in the right figure we see the time as a function of the requested size in GB (not the pitched size). This copy operation is copying from device to device.

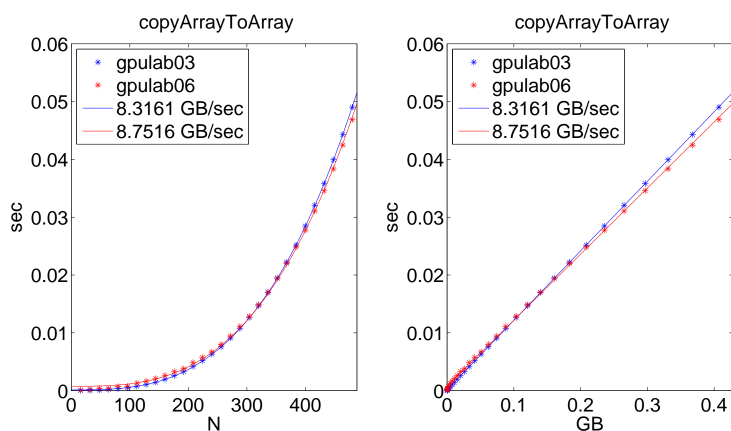


Figure 5.8: Bandwidth test of *cudaMemcpy3D*, transferring between a cudaArray which can be bound to a texture to a cudaArray which can be bound to a surface. We are transferring a volume with sizes N^3. In the image on the left we see time as a function of N, and in the right figure we see the time as a function of the requested size in GB (not the pitched size). This copy operation is copying from device to device.

CHAPTER 6

ASTRA and analysis of the 3D SIRT implementation

All Scale Tomographic Reconstruction Antwerp (ASTRA) is an existing software package for tomographic reconstruction which includes functions written in CUDA C. The package is exposed to Matlab through a set of wrapper functions written in *mex / C++*, and contains implementations of different algorithms for doing tomography in two and three dimensions. We will be focusing on the SIRT implementation, which will be used as a starting point for our implementation of the algorithms described in section 2.4. The package is implemented at *Vision Lab* which is a is a research lab of the Department of Physics of the University of Antwerp. We will be looking at ASTRA v 1.3 and not the 1.4 version because the release of this version was after the start of the MSc project. All tests in this chapter is done on the *gpulab06* computer described in table 3.1.

6.1 General setup

For all the implementations there is a common setup. First the volume and the projections are set from Matlab. Once it is set, Matlab uses a handle that

represents the geometries and the data stored on the GPU. This means that the data is not visible in Matlab, before it is specifically called back.

Once the volume and the projection geometry is set, they can be used by one of the implemented algorithms. This separates the algorithm, volume and projections, and makes everything very versatile.

It is possible to define the projections in different ways, and we will here be focusing on the parallel projections. If the projection angles are to be chosen freely, they need to be set as a matrix, where each row defines a single projection angle. Such a row must contain

$$(rayX, rayY, rayZ, dX, dY, dZ, uX, uY, uZ, vX, vY, vZ) \tag{6.1}$$

where ray is the direction, d is the center of the detector, and u and v are vectors, which define the projection plane. It is also necessary to define the number of pixels the projections contain in the u'th and v'th directions.

ASTRA stores the projections in a *cudaPitchedPtr* where the elements is stored sequentially in memory and ordered as $(u, angle, v)$. The volume is also stored as a cudaPitchedPtr where the elements are ordered as (x, y, z). In both cases we refer to the storage format found in equation 4.1.

6.2 SIRT

One of the algorithms implemented for three dimensional tomographic reconstruction is SIRT, however, by examining their code it is clear that they are referring to a slightly a different algorithm than the one we presented in Algorithm 2.1.

Instead what they are doing is shown in Algorithm 6.4, where $P_{\mathcal{C}_{up}}$ is setting every value below a tolerance and $P_{\mathcal{C}_{down}}$ is setting every value above a tolerance to the tolerance value. These operators are both optional. They use fixed weight matrices, which is set to $T = \text{diag}\left(\frac{1}{\|a_j\|_2}\right)$ and $M = \text{diag}\left(\frac{1}{\|a_i\|_2}\right)$, where a_j is the j'th row in A and a_i is the i'th column in A.

A big difference between this approach and the one described in Algorithm 2.1 is that the step size λ is omitted, corresponding to $\lambda = 1$. This could have a

Algorithm 6.4: ASTRA SIRT

$$x^k = P_{\mathcal{C}_{up}}\left(P_{\mathcal{C}_{down}}\left(x^{k-1} + TA^T M\left(b - Ax^{k-1}\right)\right)\right)$$

large impact on the performance, and the number of iterations needed to make a reliable reconstruction as shown in [19].

It is possible to set a flag that will alter the algorithms to use *super sampling*, but in order to make tests, which are as compatible as possible with the algorithm shown in Algorithm 2.1 we have chosen not to use these. Furthermore, we have disabled $P_{\mathcal{C}_{down}}$, and set the tolerance in $P_{\mathcal{C}_{up}}$ to zero.

In ASTRA they do not pre compute the projection matrix A and store it, instead they calculate the elements when they are needed. Besides saving memory, this is also opening up for a design choice. The first possibility is to track a single ray through the volume and make the updates as it traverse the volume, which is called a *ray driven* approach. The second approach is to go through each voxels in the volume, and then make the update according to the rays which enters the voxel. This is called a *voxel driven* approach.

The people behind ASTRA has in [9] advocated for the use of different approximations for the operation $FP(x) = Ax$ called the *forward projection* and $BP(b) = A^T b$ called the *back projection*. This is done to make it possible to utilize a ray driven approach for the forward projections and a voxel driven approach for the backward projection, which they have concluded is beneficial. Their implementions will be described in section 6.2.1 and 6.2.2.

6.2.1 Implementation of the forward projection

In order to to analyse the complexity of their implementation and the used memory footprint wee will a system with a volume of size N^3 and p projections of size m^2. The forward projection is in ASTRA implemented as a ray driven approach. They have implemented the Joseph's method described in section 2.5.2, with the *expanding boundary condition*. We note that the calculation done within the forward projection function call is actually equivalent to $b - Ax$. The volume is copied into a *cudaArray* of size $(N+2)^3$ with zeroes padded all around the volume. This has the effect that points just outside the volume is interpolated between the boundary point and a point with the value zero. The *cudaArray* is then bound to a 3D texture with linear interpolation. The

allocated memory footprint in the forward projection is:

$$\text{Allocated floating points in FP } = (N+2)^3 \tag{6.2}$$

The information about the projections angles, shown in equation 6.1, is put into constant memory of $1024 \cdot 12$ elements. This is beneficial since it saves space in the L1 cache and utilizes the constant memory cache and can be done since it will be used by all threads without modifications. But it has the drawback that there is a maximal number of projection angles of 1024 angles. This limit could have been removed by a loop, but this is not included in their implementation.

For each projection angle, we find a *dominant direction* which is the axis containing the largest numerical value of *ray* in equation 6.1. If consecutive angles have the same dominant direction they are grouped into blocks of up to four angles. The rays are then divided into a two dimensional grid of thread blocks, each of size 32×4. The first dimension describes the location in a given projection, and the second dimension describe the angle in the block of angles. This has the disadvantage that if the consequtive rays do not have the same dominant direction then three quarters of the threads will be treminated without doing any work. But these threads will be contained within the same warp.

For each block of up to four angles a CUDA stream is created and the dominant direction is found. The volume is then divided into slices in the dominant direction and each slice is appended to the stream. This allows the different angles to be computed concurrently, and ensures that there is no race conditions, since all writes to the same ray is done sequentially.

Inside the CUDA kernel, each thread handles one ray. Since the dominant direction is known, it is possible to re-parametrize the ray in the form:

$$\tilde{f} = \begin{pmatrix} \tilde{f}_{max} \\ \tilde{f}_{dir1} \\ \tilde{f}_{dir2} \end{pmatrix} = \begin{pmatrix} 1 \\ ay \\ az \end{pmatrix} t + \begin{pmatrix} 0 \\ by \\ bz \end{pmatrix}. \tag{6.3}$$

The entire volume is then traversed one step at the time in the dominant direction. This means that there are N texture lookups for each ray, giving pm^2N texture lookups in each iteration. The coordinate updates means that there are used $\mathcal{O}(pm^2N)$ flops (floating point operations) in each iteration.

We note that the implementation requires a copy operation from a *cudaPitchedPtr* into a *cudaArray* of the form described in figure 5.7. which is a slow operation

(that can only copy 8.36 GB/s on gpulab03 and 8.72 GB/s on gpulab06). However this will not have any noticeable impact on the overall performance because it is only needed once per iteration.

6.2.2 Backward projection

The back projection is in ASTRA implemented as a voxel driven approach. They have implemented the *transposed method* described in section 2.5.3, with *expanding boundary* condition. All the projections are copied into a *cudaArray* of size $p(m+2)^2$ with zeroes padded around each projection. So in the back projection there is allocated:

$$\text{Allocated floating points in BP } = p(m+2)^2 \tag{6.4}$$

The *cudaArray* is then bound to a 3D texture with linear interpolation, and the clamping border condition. ASTRA's approach to back projection is to project the location of each voxel onto the plane spanned by u and v, and then interpolate between neighbouring points. They also make this lookup if the projected points are outside the limits of the projection. This has the effect that a voxel which is projected onto a point in the boundary of the projection is interpolated between the boundary pixel and a pixel with the value zero. Points further out will be set to zero by the textures *clamp* border condition. This means that there are pN^3 texture lookups and interpolations.

In the ASTRA implementation, the reconstruction volume is separated into x-y slices in the z direction. Each thread block is defined as two-dimensional. The first dimension sets how many threads work on a slice at the same time, and the second dimension sets how many slices are taken into account by a given thread block.

Finally they make several consecutive kernel launches, where each updates the whole volume, but only with 64 projection angles at a time.

6.3 Complexity analysis of SIRT implementation

The main loop of ASTRA's implementation of SIRT follows the steps of Algorithm 6.4. It is shown in Listing 6.1.

The complexity of the different steps in terms of Flops and memory operations *Mobs* can be summarized as follow:

- Copy all projections [line 269] : Flops 0 - Mops pm^2.
- Forward projection [line 277] : Flops $\mathcal{O}(pNm^2)$ - Mops $\mathcal{O}(pm^2)$ - Texture Flops $\mathcal{O}(pNm^2)$ - Texture Mops $\mathcal{O}(pNm^2)$ - Memcpy3D $(N+2)^3$
- Apply weights in M [line 280] : Flops pm^2 - Mops pm^2
- Clear tempData volume [line 282] : Flops 0 - Mops N^3
- Backward projection [line 298] : Flops $\mathcal{O}(pN^3)$ - Mops $\mathcal{O}(N^3)$ - Texture Flops $\mathcal{O}(pN^3)$ - Texture Mops $\mathcal{O}(pN^3)$ - Memcpy3D $p(n+2)^3$
- Apply weights in T and add x [line 313] : Flops $\mathcal{O}(N^3)$ - Mops $\mathcal{O}(N^3)$
- Apply weights in T and add x [line 316] : Flops 0 - Mops $\mathcal{O}(N^3)$

In total we can see that there is used in the order of $\mathcal{O}(pm^2N + pN^3)$ flops, $\mathcal{O}(pm^2N + pN^3)$ texture flops, $\mathcal{O}(pm^2N + pN^3)$ texture memory operations, $\mathcal{O}(pm^2N + pN^3)$ device memory operations, and $p(m+2)^2 + pm^2n$ memory operations from device memory to a *cudaArray*, of the type described in figure 5.7.

We note that the weight T has the same size as the volume, which is copied to a temporary dataset giving a memory footprint of $3N^3$. The weights M likewise has the same size as all the projections, and is also copied into a temporary projection, which gives the footprint $3pm^2$. This means that the total memory footprint is determined as the largest of the footprint used in the forward and the backward projections, so from equation 6.2 and 6.4 we can see that the memory footprint is:

$$\text{Total memory footprint in FP} = (N+2)^3 + 3N^3 + 3pm^2 \qquad (6.5)$$

and

$$\text{Total memory footprint in BP} = p(m+2)^2 + 3N^3 + 3pm^2. \qquad (6.6)$$

Listing 6.1: ASTRAs SIRT implimentaion. There are lines which is removed on compile time with a "#if 0" command. These lines are omitted here. The code i found in *astra-1.3/cuda/3d/sirt3d.cu*

```
266   // iteration
267   for (unsigned int iter = 0; iter < iterations && !shouldAbort;
        ↪ ++iter) {
268     // copy sinogram to projection data
269     duplicateProjectionData(D_projData, D_sinoData, dims);
270
271     // do FP, subtracting projection from sinogram
272     if (useVolumeMask) {
273         duplicateVolumeData(D_tmpData, D_volumeData, dims);
274         processVol3D<opMul>(D_tmpData, D_maskData, dims);
275         callFP(D_tmpData, D_projData, -1.0f);
276     } else {
277         callFP(D_volumeData, D_projData, -1.0f);
278     }
279
280     processSino3D<opMul>(D_projData, D_lineWeight, dims);
281
282     zeroVolumeData(D_tmpData, dims);
```

```
298     callBP(D_tmpData, D_projData);
```

```
313     processVol3D<opAddMul>(D_volumeData, D_tmpData, D_pixelWeight,
          ↪ dims);
314
315     if (useMinConstraint)
316       processVol3D<opClampMin>(D_volumeData, fMinConstraint, dims);
317     if (useMaxConstraint)
318       processVol3D<opClampMax>(D_volumeData, fMaxConstraint, dims);
319   }
```

6.4 Angle dependency of the forward projection

The forward projections are traversing a three dimensional texture where it makes linear interpolation between neighbouring voxels. It is therefore interesting to evaluate the impact the ray direction has to the performance.

We wish to evaluate the performance for each direction independently. But if we used ASTRA unmodified, we would have a overhead of launching three quarters of the rays which would not be doing any work. We have therefore altered the limit of the angles per block such that each block only contains a single direction.

The setup is comparable with the one in figure 5.5 in the sense that the number of texture lookups is the same. The volume these are made on are sligtly larger, that is $(256+2)^3$ compared to 256^3. The result is shown in figure 6.1 and we see a general decrease in the performance compared to what is seen in figure 5.5.

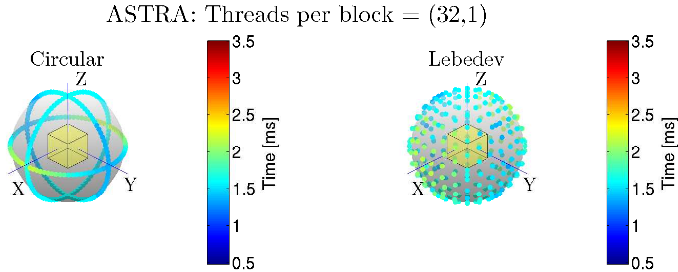


Figure 6.1: Illustration of the time consumption for the forward projection, with thread blocks of size 32×1. Measured for different angles, and orientations of the projection plane, and are based on test setup 1. Note that these timings are without the copy operation from a *cudaPitchedPtr* to a *cudaArray*. The placement of a dot indicates the direction angle, and the the color indicates the time spent on this projection angle. To the left: the projection angles are aranged on circles. To the right: they are placed as Lebedev directions.

Some of the difference we have detected could be contributed the size of the thread blocks. Which in this case is lower than that used in figure 6.1, where there was used a blocks of size of 16×16. In practice there will often be used more than a single angle at a time, which means that the used block size would be larger. To make a more comparable test we have therefore also tested their forward projection where there are used the same number of threads, as we did in figure 5.5, where the rays are divided into blocks of size 256×1. The result is seen in figure 6.2.

In this test we see a clear speed up in the best angles, where the result is comparable to figure 5.5. But we also see a decrese in the angles which is not preforming as well compared to figure 6.1.

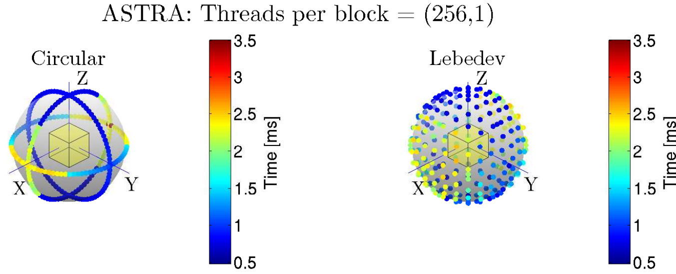


Figure 6.2: Illustration of the time consumption for the forward projection, with thread blocks of size 256 × 1. Measured for different angles, and orientations of the projection plane, and are based on test setup 1. Note that these timings are without the copy operation from a *cudaPitchedPtr* to a *cudaArray*. The placement of a dot indicates the direction angle, and the the color indicates the time spent on this projection angle. To the left: the projection angles are aranged on circles. To the right: they are placed as Lebedev directions.

Table 6.1: Total runtime for ASTRA's SIRT implementation in a the test setup described in section 3.2. The measurements is made in Matlab. To hide the latency of the invoketion, each invoketion is 20 times. There is tested for 20 of such invoketions and the mean and standart deviation is shown here.

	Small test		Large test	
	mean	std	mean	std
X,Y-plane	0.2734 sec	$8.3009 \cdot 10^{-5}$ sec	4.3161 sec	$4.8833 \cdot 10^{-4}$ sec
X,Z-plane	0.4041 sec	0.0089 sec	6.8034 sec	0.0327 sec
Y,Z-plane	0.8819 sec	0.0061 sec	18.1012 sec	0.0326 sec

Chapter 7

New implementations based on ASTRA

In this chapter we will start by making a SIRT implementation as described in section 2.2, with *Cimmino* weights. All tests in this chapter are done on the *gpulab06* computer described in table 3.1.

We will then use this as a starting point for a simple SART implementation. This implementation will later be used to make general design choices, for the rest of our implementations.

The idea behind the implementations described in this chapter is to use the setup provided by ASTRA. But we want it to be completely separated from ASTRA, such that the ASTRA library is left unchanged. This is achieved by linking against ASTRA. For simplicity these implementations will derive from classes within ASTRA.

7.1 New SIRT implementation based on ASTRA

The first step was to alter the SIRT implementation such that it is on the form described in section 2.2. It is therefore necessary to alter the weights such that

it uses the *Cimmino* weights.

This means that since $T = I$ we could eliminate *D_pixelWeight*, and thereby save N^3 numbers of floating point precision in the memory footprint. But since this specific implementation is derived from a class in ASTRA where this pointer is allocated, it will still be allocated in memory. This approach means that the function $processVol3D\langle opAddMul\rangle$ seen in listing 6.1 line 280 must be substituted with with a function *processVol3D_opAddMul* that does not multiply with the weights, but instead makes multiplications with the step length λ. This function use the same number of Flops, but the new one makes pm^2 fewer memory operations.

It is sufficient to alter the implementation seen in 6.1, to use the new weights and the new function *processVol3D_opAddMul* in order to make a SIRT implementation of the form seen in 2.2. But we note that since we don't use a volume mask, the branching in line 272 can be eliminated. We can also reduce the clamping operations since we only use P_{up}, which is clamping up to zero. The finished version is seen in listing 7.1.

Listing 7.1: New SIRT implimentaion, as described in 2.1.

```
1333  for (unsigned int iter = 0; iter < iterations && !shouldAbort;
          ↪ ++iter)
1334  {
1335    duplicateProjectionData(D_projData, D_sinoData, dims);
1336    callFP(D_volumeData, D_projData, -1.0f);
1337    processSino3D<opMul>(D_projData, D_lineWeight, dims);
1338    zeroVolumeData(D_tmpData, dims);
1339    callBP(D_tmpData, D_projData);
1340    processVol3D_opAddMul(D_volumeData, D_tmpData, lambda, dims);
1341    processVol3D<opClampMin>(D_volumeData, 0.0f, dims);
1342  }
```

The SIRT implementation was tested on the second test setup, described in section 3.2. In table 7.1 is shown the average runtime for 20 iterations. These time measurements are made within the C++ implementation, so there is not the same overhead in terms of invoking the function, as there was for the test of ASTRA's implementation seen in table 6.1.

From this result we note a small decrease in the runtime compared to result from ASTRA. This comes from the saving in memory operations in the new function *processVol3D_opAddMul*.

In this implementation we have also measured the time consumption used for

Table 7.1: Total runtime for the new SIRT implementation which is based on ASTRA. The test setup described in section 3.2 as test setup 2. It is tested 20 times and the mean and standard deviation is shown here.

	Medium test		Large test	
	mean	std	mean	std
X,Y-plane	0.2704 sec	$7.10 \cdot 10^{-5}$ sec	4.1603 sec	$4.850 \cdot 10^{-4}$ sec
X,Z-plane	0.3588 sec	0.003496 sec	6.7418 sec	0.05357 sec
Y,Z-plane	0.8742 sec	0.006308 sec	17.410 sec	0.08713 sec

each function, and it was clear that the main consumption was used in the forward and backward projections. The time consumption for the projections in a single iteration is shown in table 7.2.

Table 7.2: Measurement of the time consumption, for the forward and back projection, in the new SIRT implementation. These functions both reside in the ASTRA framework.

		Medium test	Large test
X,Y-place	Forward projection	0.08581 sec	1.327 sec
	Back projection	0.1624 sec	2.768 sec
X,Z-place	Forward projection sec	0.1793 sec	3.558 sec
	Back projection	0.1750 sec	3.122 sec
Y,Z-place	Forward projection	0.2469 sec	5.677 sec
	Back projection	0.6307 sec	11.457 sec

From table 7.2 it is clear that both projections are very influenced by the projection angles. Specifically the results for the back projections are surprising because the volume is traversed in the same manner independently of the dominant direction. Therefore the results for the back projection should be the same for all angles.

7.2 SART implementation based on ASTRA

Before we proceed with a general block iterative method, we note that there will be an issue with the data structure used for the projections. When the number

of blocks is high, compared to the number of projections, the ordering (u, a, v) used by ASTRA will not be beneficial because a single projection angel is not stored continuously in the memory. It will therefore be more beneficial to use a structure where the data is stored as (u, v, a). To illustrate the impact of this ordering we have implemented a block iterative method where the number of blocks is equal to the number of projection angles, which means that we in this section will make a SART implementation as described in section 2.4.

We have made the implementations such that it takes a copy of a single slice from the projection and a slice from the weights. We then work on such slices, for each angle, as it is done in the SIRT implementation. The implementation is seen in Listing 7.2, and we note that two function calls for the projections have changed names. But they are actually referring to the same projections as is used in SIRT. The difference is that it is now only possible to use ASTRA's parallel projections, where in the SIRT method it was also possible to set it up such that it uses the cone projection. Since we no longer make a copy of all projections, but only copy a single angle out of the projection and the weights, we see that the memory footprint is a little smaller that those in equation 6.5 and 6.6. In this implementation the memory footprint is:

$$\text{Total memory footprint in FP} = (N+2)^3 + 3N^3 + 2pm^2 + 2m^2$$

and

$$\text{Total memory footprint in BP} = (m+2)^2 + 3N^3 + 2pm^2 + 2m^2$$

because the *D_pixelWeight* is still allocated.

Listing 7.2: SART implementation based on ASTRA functions.

```
1177  for (unsigned int iter = 0; iter < iterations && !shouldAbort;
        ↪ ++iter)
1178  {
1179    for (unsigned int angles = 0 ; angles < dims.iProjAngles ;
          ↪ ++angles)
1180    {
1181      // copy sinogram to projection data
1182      duplicateProjectionDataSlice(D_pd, D_sinoData, dims, angles);
1183      duplicateProjectionDataSlice(D_lw, D_lineWeight, dims,
            ↪ angles);
1184      Par3DFP(D_volumeData, D_pd, DIMS, &par3DProjs[angles], -1);
1185      processSino3D<opMul>(D_pd, D_lw, DIMS);
1186      zeroVolumeData(D_tmpData, DIMS);
1187      Par3DBP(D_tmpData, D_pd, DIMS, &par3DProjs[angles]);
1188      processVol3D_opAddMul(D_volumeData, D_tmpData, lambda, dims);
1189      processVol3D<opClampMin>(D_volumeData, 0, dims);
1190    }
1191  }
```

The method is again tested on the tests described in section 3.2, and the result is shown in table 7.3.

Table 7.3: Total runtime for the new SART implementation which is based on ASTRA. The test setup described in section 3.2 as test setup 2. It is tested 20 times and the mean and standard deviation is shown here.

	Medium test		Large test	
	mean	std	mean	std
X,Y-plane	5.7798 sec	0.0221 sec	60.50 sec	0.0264 sec
X,Z-plane	5.9772 sec	0.0216 sec	63.54 sec	0.0213 sec
Y,Z-plane	6.0424 sec	0.0016 sec	66.36 sec	0.0384 sec

We see that the total runtime has increased by a factor of 20. This is contributed by a number of things. Beside the issue with the data structure, the individual functions is only used once in the SIRT but they are now called once for each angle in each iteration. The tuning of these are therefore much more important in a SART method, and in any block iterative method with a high number of blocks. These functions are therefore now a real contributor to the runtime. In table 7.4 we have shown the total time consumption for all the functions in the test used in the X,Y-plane. Because the rest of the functions are stable with respect to the projection angle, we only show the forward and backward projection in the other to test cases.

From these results we see wast majority of the time is spent in the forward projection, and that this is also the function with most angle dependence. This function is not designed to only be working on a single angle at a time, and it now makes a copy operation, from a *cudaPitchedPtr* to a *cudaArray*, of size N^3 for each angle. This copy operation was found in figure 5.7 to be slow (8.72 GB/s). This bandwidth implies that there for our small test case with 133 angles and a volume of size 256^3 is used approximately $p \cdot N^3/(1024^3 \cdot 8.72) = 0.95sec$ per iteration in copy operations.

Table 7.4: Measurement of the time consumption, for the forward and back projection, in the new SART implementation. These functions both reside in the ASTRA framework.

		Small test	Large test
X,Y-place	duplicate data	0.0486 sec	0.24 sec
	Forward projection	2.9492 sec	41.82 sec
	opMulSin	0.1679 sec	0.76 sec
	zeroVolume	0.1954 sec	1.55 sec
	backProjection	0.6740 sec	5.74 sec
	volumeOpAddMul	0.9045 sec	5.96 sec
	opClapMin	0.8918 sec	4.59 sec
X,Z-place	Forward projection	3.1133 sec	44.85 sec
	Back projection	0.6876 sec	5.73 sec
Y,Z-place	Forward projection	3.2108 sec	47.44 sec
	Back projection	0.6370 sec	6.03 sec

Chapter 8

New implementations based on new ordering, new boundary conditions, and shared memory

In this chapter we will present a number of implementations where the data structure in the projections is changed from a (u, a, v) ordering to a (u, v, a) ordering. We will also make fundamental changes in the boundary condition of Joseph's method, in both the forward and backward projection. We will now use the *clamping boundary* condition described in 2.5.2 and 2.5.3. All tests in this chapter is done on the *gpulab06* computer described in table 3.1.

The reordering of the elements in the projections is done because we wish to focus on a single projection angle at the time. The ordering (u, a, v) used by ASTRA braces the projection angles together such that a single angle is not stored sequentially in memory. We will therefore instead use a (u, v, a) ordering, because this will keep a complete projection angle stored sequentially in memory.

These alterations means that it is necessary to rewrite most of the functions described up until now. The functions that could have been reused, are so

simple that we chose to rewrite them as well. The functions described in the rest of the project will not be linked against ASTRA.

We will now describe the new implementation of the forward and the and back projection, the rest of the utility functions will not be described in detail.

8.1 New forward projection using textures

In this section we will describe a new implementation of a forward projection, which uses the data ordering (u, v, a). This version will be inspired by the ASTRA implementation and will like the version in ASTRA also make use of a three dimensional texture reference. But it will use the *clamping boundary* condition described in 2.5.2 instead of the *extending boundary* condition. This has the benefit that the memory is better aligned in the copy operation, and that the forward projection uses less memory, since the memory allocation is only:

$$\texttt{Allocated mem in FP} \ = N^3 \tag{8.1}$$

This boundary condition is implemented by transferring a volume of size N^3 into a *cudaArray* using *cudaMemcpy3D*, and then utilizing the *cudaAddressModeBorder* boundary condition. This setting sets all texture calls that are outside the boundary to zero.

Like in ASTRA we will start by finding a *dominant direction* and then traverse the entire volume in this direction. This means that each ray will make N texture interpolations, also if the ray is outside the volume. We will not divide the volume into large slices, in the dominant direction. This has the drawback that we lose a little precision compared to ASTRA, because they calculate a new starting point for each slice. We on the other hand only calculate the starting point for a ray once, and then make N updates to this value, all with 32 bit floating point precision.

Because we only look at a single projection angle at the time, it will be suitable to use a grid system for the threads which span the projection plane. We have chosen to let the u direction span the first dimension in the thread blocks, and let the v direction span the second direction. We have tested three different grid configurations with the test setup described in section 3.1. This is the same test as we used for testing the forward projection in ASTRA, i.e. in figure 6.1 and 6.2.

We first test the forward projection implementation with a thread block of size

16×16 threads. In this configuration, each warp will fill two rows of a thread block. The runtime as a function of the projection angle is seen in figure 8.1. From this test we note that we achieve a generally faster result, than what was

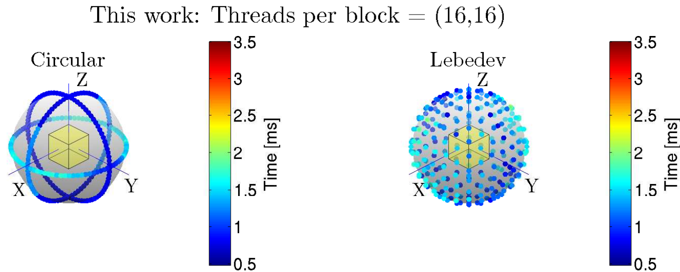


Figure 8.1: Illustration of the time consumption for the new forward projection, with thread blocks of size 16×16. Measured for different angles, and orientations of the projection plane, and are based on test setup 1. Note that these timings are without the copy operation from a *cudaPitchedPtr* to a *cudaArray*. The placement of a dot indicates the direction angle, and the the color indicates the time spent on this projection angle. To the left: the projection angles are arranged on circles. To the right: they are placed as Lebedev directions. This test is done on gpulab06.

achieved in ASTRAS 32×1 thread configuration, which is close to their standard configuration. This is also a more uniform result than what was achieved with ASTRA's 256×1 thread configuration, which contains the same number of threads in a block. But to make a more fair comparison, we should also test the new implementation with the same thread blocks as we used to test ASTRA. These test is shown in figure 8.2 and 8.3.

From figure 8.2 we see the most uniform result, with respect to the projection angles and the direction of u and v, and achieve in general good performance compared to all earlier tests.

From figure 8.3 we see the most irregular result with respect to the chosen projection angles. There are still angles which preforms well, but they are not completely uniform with respect to a dominant direction.

The results shown in figure 8.1 and 8.2 both seems uniform with respect to the dominant direction. We will now combine the results from the circular tests, and show their matched performance in figure 8.4. This will help us determine the block size, and chose whether the block size should be the same for all dominant

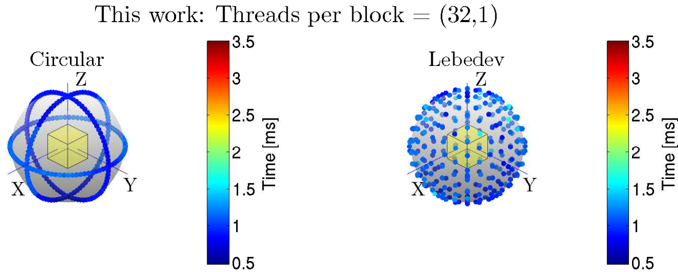


Figure 8.2: Illustration of the time consumption for the new forward projection, with thread blocks of size 32×1. Measured for different angles, and orientations of the projection plane, and are based on test setup 1. Note that these timings are without the copy operation from a *cudaPitchedPtr* to a *cudaArray*. The placement of a dot indicates the direction angle, and the the color indicates the time spent on this projection angle. To the left: the projection angles are arranged on circles. To the right: they are placed as Lebedev directions. This test is done on gpulab06.

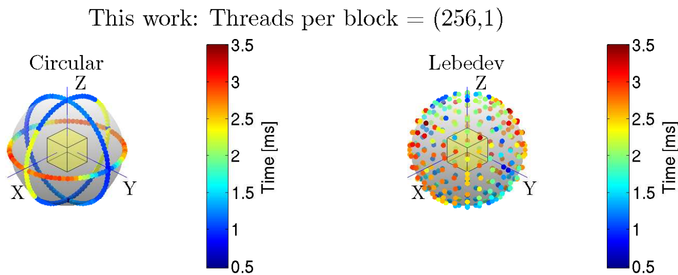


Figure 8.3: Illustration of the time consumption for the new forward projection, with thread blocks of size 256×1. Measured for different angles, and orientations of the projection plane, and are based on test setup 1. Note that these timings are without the copy operation from a *cudaPitchedPtr* to a *cudaArray*. The placement of a dot indicates the direction angle, and the the color indicates the time spent on this projection angle. To the left: the projection angles are arranged on circles. To the right: they are placed as Lebedev directions. This test is done on gpulab06.

directions.

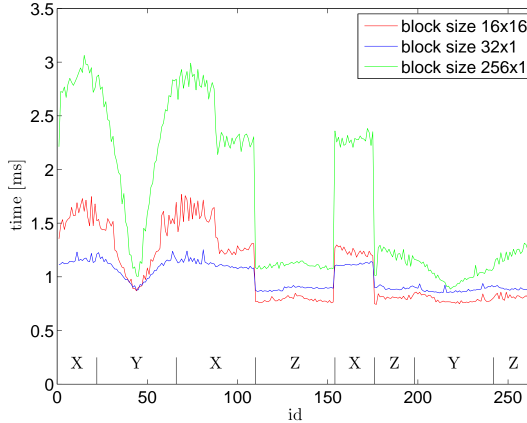


Figure 8.4: The matched timings from the circular test in test 2 for different sizes of thread blocks. This indicates which size is most efficient for different dominant directions. This test is done on gpulab06.

From figure 8.4 we conclude that we should chose a block size of 32×1 when the dominant direction is the x-axis, and 16×16 when the z-axis is the dominant direction. The choice for the y-axis is not conclusive, and we therefore chose the block size 32×1 because it is generally slightly more regular.

8.2 New back projection using textures

This section will give a short description of the changes we have made in order to make a new back projection. This back projection will work on a single projection, and assumes that the data in the projection is ordered as (u, v, a).

This function is also texture based, but because we only work on a single projection angle at the time we have used a two dimensional texture, instead of the three dimensional used in ASTRA. We have also changed the boundary condition in the back projection. This is changed to the *clamping boundary* condition, described in section 2.5.3. Like in the new forward projection, we have made this change by altering the texture boundary condition, which now is set to *cudaAddressModeBorder*. This sets all texture calls outside the boundary to zero. Since we only need a single projection angle, and uses the clamping boundary condition, we will in the back projection now only allocate :

$$\texttt{Allocated mem in BP} = m^2 \tag{8.2}$$

We use the same thread blocks and traverse the volume in the same way that ASTRA do, except that we only look at a single projection angle. We have not utilized the constant memory for the common parameters, but instead put these 8 numbers in device memory. We then access the parameters through the L1 cache. This should not have any impact on the performance, because all the texture lookups are done in the texture cache, which leaves the L1 cache almost unused.

8.3 New SART implementation

In this section, we will combine the new forward projection and backward projection implementations into a fully functional SART algorithm.

The weights *D_pixelWeight* that ASTRA allocated, is not present in this implementation. So with the memory savings in the forward and back projection the memory we now have a memory footprint at :

$$\begin{aligned}&\texttt{memory fotprint in second independent SART implementaion:}\\ &\texttt{in forward projection} = 4N^3 + 2pm^2 + m^2\\ &\texttt{in back projection} = 3N^3 + 2pm^2 + 2m^2\end{aligned} \tag{8.3}$$

The implementation is seen in listing 8.1, and is very similar to the previous implementation which was based on ASTRA.

In line 56 and 59 are shown the two projections, and we note that this implementation of the forward projection also subtract the measured projection, i.e. $b - Ax$ instead of just Ax. In line 57 we apply the weights from M and in line 60 we apply the step length λ and add the result to the existing volume. Line 61 is rounding all values bellow zero to zero, i.e. P_{up}.

Line 60 and 61 both uses $\mathcal{O}(pN^3)$ memory operations and Flops per iteration, which is increased by a factor p compared to SIRT.

The number of Flops in the forward projection is unchanged because we traverse the volume in the same manner as in SIRT, but we note that we now make pN^3 copy operations, from a *cudaPitchedPtr* to a *cudaArray*, per iteration. So this has also increased by a factor p compared to SIRT, of the slow type of memory transfer shown in figure 5.7.

Listing 8.1: First SART implementation, which is independent from ASTRA

```
48  for (unsigned int iter = 0; iter < iterations ; ++iter)
49  {
50    for (unsigned int angles = 0 ; angles < dims.iProjAngles ;
        ↪ ++angles)
51    {
52      processVol3DsetValue(D_tmpData, 0.0f, dims);
53      size_t tmp
          ↪ =pitchProjections/sizeof(float)*angles*dims.iProjV;
54      ASTRA_CUDA_ASSERT(cudaMemcpy2D( d_sino , pitchs ,
          ↪ &d_projections[tmp] , pitchProjections ,
          ↪ dims.iProjU*sizeof(float) , dims.iProjV ,
          ↪ cudaMemcpyDeviceToDevice ));
55      lw = &d_weights[ pitchWeights / sizeof(float) *
          ↪ angles*dims.iProjV];
56      Par3DFP_singleAngle(D_volumeData,d_sino, pitchs, dims,
          ↪ par3DProjs[angles],-1);
57      sinogram2dOpMul<<< (dims.iProjU+31)/32,32 >>>(d_sino, lw,
          ↪ pitchs, dims.iProjU, dims.iProjV); // ny
58      cudaDeviceSynchronize();
59      Par3DBP_singleAngle(D_tmpData,d_sino, pitchs, dims,
          ↪ par3DProjs[angles]);
60      processVol3D_opAddMulV2(D_volumeData, D_tmpData, lambda,
          ↪ dims); // ny
61      processVol3DopClampZero(D_volumeData, dims);
62    }
63  }
```

The implementation is tested on test setup 2 described in section 3.2, and the average runtime is shown in table 8.1. We see that we have saved about 2/3 of the runtime for the medium problem and about 1/2 of the runtime for the large problem compared to the version which uses ASTRA's functions. We also note that this version seems more uniform with respect to the projection angles.

Table 8.1: Total runtime for the new SART implementation which is not based on ASTRA. The test setup described in section 3.2 as test setup 2. It is tested 20 times and the mean and standart deviation is shown here.

	Small		Medium		Large	
	mean	std	mean	std	mean	std
X,Y-plane	0.1979 sec	0.0011 sec	1.91 sec	0.004 sec	30.12 sec	0.05 sec
X,Z-plane	0.1970 sec	0.0004 sec	1.92 sec	0.003 sec	31.00 sec	0.05 sec
Y,Z-plane	0.1998 sec	0.0028 sec	1.96 sec	0.005 sec	33.92 sec	0.08 sec

We have measured the time consumption used for each function, and it was

clear that we now use the main time consumption in the forward projections. The time consumption for the projections in a single iteration is shown in table 8.2.

Table 8.2: Measurment of the time consumption in a single iteration of the first independent SART implementation.

		Small	Medium	Large
X,Y-place	Forward projection	0.1246 sec	1.206 sec	18.526 sec
	sinogram2dOpMul	0.0060 sec	0.0223 sec	0.096 sec
	Par3DBP_singleAngle	0.0329 sec	0.1863 sec	2.478 sec
	Vol3D_opAddMulV2	0.0164 sec	0.2838 sec	5.243 sec
	Vol3DopClampZero	0.0091 sec	0.1316 sec	2.308 sec
	Vol3DsetValue	0.0052 sec	0.0886 sec	1.538 sec
	copy time	0.0010 sec	0.0026 sec	0.009 sec
X,Z-place	Forward projection	0.1250 sec	1.2195 sec	19.515 sec
	Back projection	0.0330 sec	0.1887 sec	2.497 sec
Y,Z-place	Forward projection	0.1261 sec	1.2396 sec	22.130 sec
	Back projection	0.0330 sec	0.1878 sec	2.502 sec

The back projection is no longer a big time consumer and the other functions, apart from the forward projection, take relatively more time. We therefore focus on optimizing these utility functions by merging them into the back projection function, which already traverses the entire volume and can easily do the updates.

As a positive side effect we can eliminate the temporary volume $D_tmpData$ and thereby reduce the memory footprints in Equation 8.3 by N^3.

The new SART implementation, where the step length and the rounding operation have been merged into the back projection is shown in Listing 8.2.

Listing 8.2: First SART implementation, which is independent from ASTRA

```
396   for (unsigned int iter = 0; iter < iterations ; ++iter)
397   {
398     for (unsigned int angles = 0 ; angles < dims.iProjAngles ;
          ↪ ++angles)
399     {
400       size_t tmp
            ↪ =pitchProjections/sizeof(float)*angles*dims.iProjV;
401       ASTRA_CUDA_ASSERT(cudaMemcpy2D( d_sino , pitchs ,
            ↪ &d_projections[tmp] , pitchProjections ,
            ↪ dims.iProjU*sizeof(float) , dims.iProjV ,
            ↪ cudaMemcpyDeviceToDevice ));
402       lw =
            ↪ &d_weights[pitchWeights/sizeof(float)*angles*dims.iProjV];
403       Par3DFP_singleAngle(          D_volumeData,d_sino, pitchs,
            ↪ dims, par3DProjs[angles],-1);
404       sinogram2dOpMul<<< (dims.iProjU+31)/32,32 >>>(d_sino, lw,
            ↪ pitchs, dims.iProjU, dims.iProjV); // ny
405       cudaDeviceSynchronize();
406       Par3DBP_singleAngleV2opClampZero(D_volumeData,d_sino,
            ↪ pitchs, dims, par3DProjs[angles],lambda);
407     }
408   }
```

This second independent SART implementation is also tested for test case 2. And the total runtime per iteration is shown in Table 8.3.

Table 8.3: Total runtime for the second SART implementation which is not based on ASTRA. The test setup described in section 3.2 as test setup 2. It is tested 20 times and the mean and standart deviation is shown here.

	Small		Medium		Large	
	mean	std	mean	std	mean	std
X,Y-plane	0.1659 sec	0.0002 sec	1.411 sec	0.002 sec	21.10 sec	0.002 sec
X,Z-plane	0.1657 sec	0.0002 sec	1.425 sec	0.001 sec	22.13 sec	0.004 sec
Y,Z-plane	0.1658 sec	0.0002 sec	1.439 sec	0.001 sec	24.75 sec	0.012 sec

From this result we see a general improvement of a factor 1/3 compared to the previous version with separate utility functions. This is empathized in table 8.4 where we have shown the time spend for each function in a single iteration.

We note that all the time spend in the back projection is almost unchanged.

The forward projection is the main time consumer. This high consumption can be contributed to the slow copy operation from a *cudaPitchedPtr* to a *cudaArray*

Table 8.4: Measurment of the time consumption in a single iteration of the second independent SART implementation.

		Small	Medium	Large
X,Y-place	Forward projection	0.1260 sec	1.2032 sec	18.527 sec
	sinogram2dOpMul	0.0060 sec	0.0223 sec	0.097 sec
	Par3DBP_singleAngle	0.0332 sec	0.1860 sec	2.482 sec
	copy time	0.0012 sec	0.0030 sec	0.010 sec
X,Z-place	Forward projection	0.1260 sec	1.2167 sec	19.536 sec
	Back projection	0.0331 sec	0.1875 sec	2.500 sec
Y,Z-place	Forward projection	0.1257 sec	1.2298 sec	22.146 sec
	Back projection	0.0331 sec	0.1872 sec	2.506 sec

seen in figure 5.7, (which has a bandwidth of 8.72 GB/s).

$$small = 66 * (128^3) * 4/(1024^3 * 8.72) = 0.0591 sec$$
$$medium = 133 * (256^3) * 4/(1024^3 * 8.72) = 0.9533 sec$$
$$large = 300 * (512^3) * 4/(1024^3 * 8.72) = 17.2018 sec$$

Since we can not bind a three dimensional texture to sequential device memory, and these timing stands for the vast majority of the total time consumption we conclude that a texture approach is inefficient for a block iterative method, with a high number of blocks.

8.3.1 Forward projection without textures

We will now present an implementation of a forward projection which dos not use the textures. This function will make the same kind of interpolation as the previous one, but it will make it without the use of textures. Because the volume is not copied into a *cudaArray*, and that we are still not using the *D_tmpData*, this version saves $2N^3$ floating points in the forward projections memory footprint, compared to equation 8.3.

For each ray, we find the intersection with the volume, and then traverse the inner part of the volume in the same way as it was done in the previous methods.

We will still let the thread blocks span the projection plane, and use thread blocks of the size (16×16). The first dimension of the thread block represents the u-vector in the projection plane. We note that this method induces a high

rate of cache misses. In fact the only ray directions which can induce coalesced data reads, is when the rays dominant direction is in the Y - or Z axis. In order to have coalesced data reads, u-vectors projection into a slice in the dominant direction, has to be close to parallel to the x-axis. All other combinations will induce cache misses for almost all memory reads.

We have inserted this method into the SART implementation, and shown the time per iteration in table 8.5, and the time used by the forward projection in table 8.6.

Table 8.5: Total runtime for the SART implementation which uses the forward projection that is not texture based. The test setup described in section 3.2 as test setup 2. It is tested 20 times and the mean and standart deviation is shown here.

	Small		Medium		Large	
	mean	std	mean	std	mean	std
X,Y-plane	0.1056 sec	0.0012 sec	1.4146 sec	0.0042 sec	25.5038 sec	0.0364 sec
X,Z-plane	0.1056 sec	0.0011 sec	1.4813 sec	0.0040 sec	39.5126 sec	0.0606 sec
Y,Z-plane	0.0957 sec	0.0012 sec	1.2758 sec	0.0033 sec	27.7527 sec	0.0177 sec

When we compare the results seen in table 8.5 and 8.6 with the earlier result in table 8.3 and 8.4, we see a better performance in the small test case. This is because we no longer make the memory transfer needed to bind the memory to a texture, and because the volume is so small that there will be much more memory reads that do not result in cache misses. But as the volume increases, there are more and more cache misses, which give rise to the bad scaling properties seen in table 8.6. The positive results for the small test problem indicates that this basic idea is the right approach.

Table 8.6: Measurment of the forward projection, in a single iteration, of the SART implementation.

		Small	Medium	Large
X,Y-place	Forward projection	0.0759 sec	1.2191 sec	23.014 sec
X,Z-place	Forward projection	0.0758 sec	1.2846 sec	36.912 sec
Y,Z-place	Forward projection	0.0660 sec	1.0776 sec	25.207 sec

8.3.2 Forward projection utilizing shared memory

We will now present a forward projection which utilizes the shared memory. The idea is to split the volume, into cubes that can fit in the shared memory, and then find the rays that intersects the cube.

This is done by inserting the inner part of the previous method, into an existing frame work. That is, we have taken the part of the code where we find the intersection for a ray in a cube, and make the interpolation within the cube, into a framework that partition the volume into smaller cubes of size 16^3. It then make use of a grid of threads, each of 16×16 threads, and let these blocks span the space perpendicular to the dominant direction. It then traverse the volume, by moving the blocks along the dominant direction.

We have tested this forward projection on test setup 1, and the result is shown in figure 8.5. From this result we see that the time spend in each projection angle has increased compared to the result seen in figure 8.1 and 8.2. The new result is also more irregular with respect to projection angle and orientation of the projection space. But we note that the results in figure 8.1 and 8.2 was without the slow copy operation from a *cudaPitchedPtr* to a *cudaArray*. This means that this new result is better when it comes to a SART implementation.

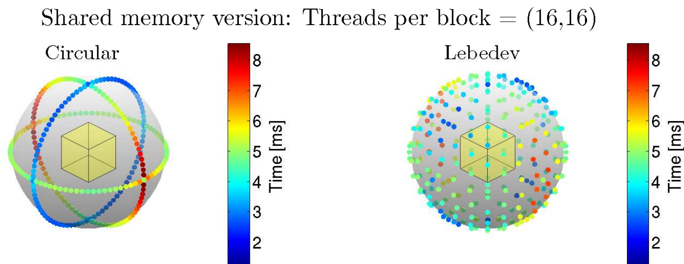


Figure 8.5: Illustration of the time consumption for the forward projection that utilizes the shared memory, with thread blocks of size 16×16. Measured for different angles, and orientations of the projection plane, and are based on test setup 1. Note that these timings are without the copy operation from a *cudaPitchedPtr* to a *cudaArray*. The placement of a dot indicates the direction angle, and the the color indicates the time spent on this projection angle. To the left: the projection angles are aranged on circles. To the right: they are placed as Lebedev directions. This test is done on gpulab06.

We have used this forward projection together with a back projection, from

the same framework, and combined them into a SART implementation. This implementation is also tested on test setup 1, and the result is shown in table 8.7. Where we see an improvement compared to all previous SART implementations.

Table 8.7: Total runtime for the SART implementation which uses the forward projection that is not texture based and utilizes the shared memory. The test setup described in section 3.2 as test setup 2. It is tested 20 times and the mean and standart deviation is shown here.

	Small		Medium		Large	
	mean	std	mean	std	mean	std
X,Y-plane	0.0604 sec	0.0000 sec	0.8629 sec	0.0000 sec	14.219 sec	0.0009 sec
X,Z-plane	0.0658 sec	0.0000 sec	0.9493 sec	0.0001 sec	15.600 sec	0.0016 sec
Y,Z-plane	0.0698 sec	0.0000 sec	1.0174 sec	0.0001 sec	16.793 sec	0.0011 sec

We have also measured the time consumption of the forward projection, in a single iteration. This is shown in table 8.8. This also shows an improvement in the forward projection compared to all the earlier versions.

Table 8.8: Test of the forward projection utelizing shared memory. These timings is taken from a single SART iteration.

	Small	Medium	Large
X,Y-projection	0.0472 sec	0.7144 sec	11.782 sec
X,Z-projection	0.0526 sec	0.7985 sec	13.128 sec
Y,Z-projection	0.0567 sec	0.8651 sec	14.275 sec

This makes us conclude that for a block iterative method, with a high number of blocks it is more efficient to use a method which utilize the shared memory, than one that utilises the texture cache.

8.4 Implementation of a General Block-Iterative method

We will in this section describe the necessary steps in order to alter the SART implementation into a general *Block Iterative Method.*

Because the optimal forward projection was build into another framework than the rest of the functions, the forward projection used in this section will be the

texture based version. But we note that it would be interesting to see the result using the forward projection based on shared memory.

In order to make a method which can divide the angles into an arbitrary number of blocks $nrBlocks$, in the interval $[1, p]$, we first note that the size of the blocks has a large impact on the step size λ as shown in [19]. We therefore has to divide the projection angles into blocks of similar size. This is not possible in general, but we can find blocks that at most differs in size with a single projection angle. This is done by calculating the size of the largest block c as $c = \lceil p/nrBlocks \rceil$. We then calculate the number blocks which should have this size, as $anglesPrBlockUntilID = p + nrBlocks - nrBlocks \cdot c$. This guaranties that the rest of the blocks has the size $c - 1$.

We will need to modify the forward projection such that it can work on an arbitrary number of angles, while only make the copy into the *cudaArray* once. And we also alter the back projection such that it takes the step length λ.

We split the loop over the angles into two blocks, one which works on block sizes c and a part that works on $c - 1$.

The implementation is shown in Appendix A.

We have tested how the time consumption per iteration scales with the number of blocks, and the result is shown in figure 8.6. From this we see a linear dependence, which we manly contribute to the linear scaling of the number of copy operations there is needed, from a *cudaPitchedPtr* into a *cudaArray*.

We have also tested the implementation, as a SIRT implementation, i.e. where the number of blocks is equal to one. The result is shown in table 8.9, where we note that we achieve a more uniform result with respect to the projection angles than ASTRA, and the version using ASTRA's functions. We now also achieve average time consumptions in all cases that are as low as the best angles using ASTRA's functions.

We have also tested the general *Block Iterative Method* as a SART method, on test case 2, i.e. where the number of blocks equals the number of projections. The result are shown in table 8.10 and the consumption for the individual functions is shown in table 8.11.

These results indicates a slight increase in time consumption compared the the SART method which used the same forward projection, seen in table 8.3. Comparing the individual functions in table 8.11 with table 8.4 we see that this difference comes from the use clamping operation in
processVol3DopClampZero.

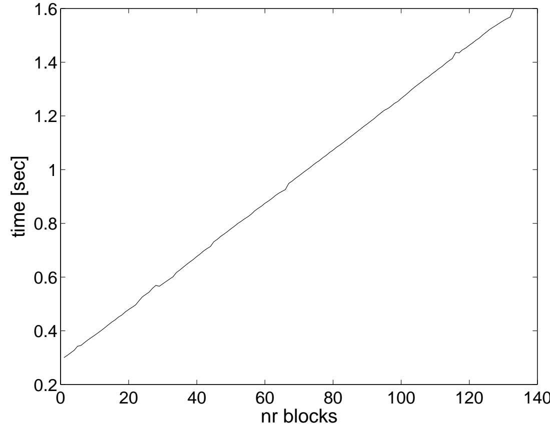


Figure 8.6: Illustration of the dependency of the block size in time consumption.

Table 8.9: Total runtime for the general Block Iterative method, used as a SIRT method. The test setup described in section 3.2 as test setup 2. It is tested 20 times and the mean and standart deviation is shown here.

	Small		Medium		Large	
	mean	std	mean	std	mean	std
X,Y-plane	0.0523 sec	0.0001 sec	0.3001 sec	0.0017 sec	4.1105 sec	0.0006 sec
X,Z-plane	0.0471 sec	0.0001 sec	0.3136 sec	0.0001 sec	5.1388 sec	0.0061 sec
Y,Z-plane	0.0472 sec	0.0001 sec	0.3283 sec	0.0014 sec	7.8214 sec	0.0090 sec

Table 8.10: Total runtime for the general Block Iterative method, used as a SART method. The test setup described in section 3.2 as test setup 2. It is tested 20 times and the mean and standart deviation is shown here.

	Small		Medium		Large	
	mean	std	mean	std	mean	std
X,Y-plane	0.1854 sec	0.0064 sec	1.5941 sec	0.0051 sec	24.1411 sec	0.0911 sec
X,Z-plane	0.1781 sec	0.0002 sec	1.5956 sec	0.0051 sec	24.8753 sec	0.0725 sec
Y,Z-plane	0.1789 sec	0.0003 sec	1.6259 sec	0.0048 sec	27.7972 sec	0.1031 sec

Table 8.11: Measurment of the time consumption of the individual functions in the general Block Iterative method used as a SART method.

		Small	Medium	Large
X,Y-place	Forward projection	0.1308 sec	1.2563 sec	19.380 sec
	sinogram2dOpMul	0.0060 sec	0.0223 sec	0.096 sec
	Par3DBP_singleAngle	0.0335 sec	0.1866 sec	2.477 sec
	processVol3DopClampZero	0.0095 sec	0.1324 sec	2.273 sec
	copy time	0.0011 sec	0.0026 sec	0.009 sec
X,Z-place	Forward projection	0.1299 sec	1.2610 sec	20.218 sec
	Back projection	0.0331 sec	0.1877 sec	2.496 sec
Y,Z-place	Forward projection	0.1300 sec	1.2753 sec	22.819 sec
	Back projection	0.0333 sec	0.1874 sec	2.503 sec

Chapter 9

Final results and comparison

We will in this chapter make some evaluations on fastest reconstructions we could achieve. The fastest method was the SART implementation based on the forward projection that utilizes the shared memory. All tests in this chapter is done on the *gpulab06* computer described in table 3.1.

We will first compare the total reconstruction time, for the best possible result, on a system corupted with Gaussian noise e, which is scaled such that the relative noise level is fixed at $\frac{\|e\|_2}{\|b\|_2} = 0.05$.

This is done for the SART method where the optimal step length λ is found as described in [19]. For a comparison we have also tested ASTRA, and both the results are seen in figure 9.1. Because we want the total reconstruction time, we will include the time spent on invoking the GPU, and the transfer both ways, between the host and device. From this result we see that we can reach the best reconstruction in 3.8 sec, with the SART method. While it takes ASTRA's SIRT method 192.93 sec.

The SIRT method is known to have a slow convergence rate. It would therefore be more fair to test how fast it reaches a given tolerance. In figure 9.2 we have shown the convergence of ASTRAs SIRT implementation and of the SART

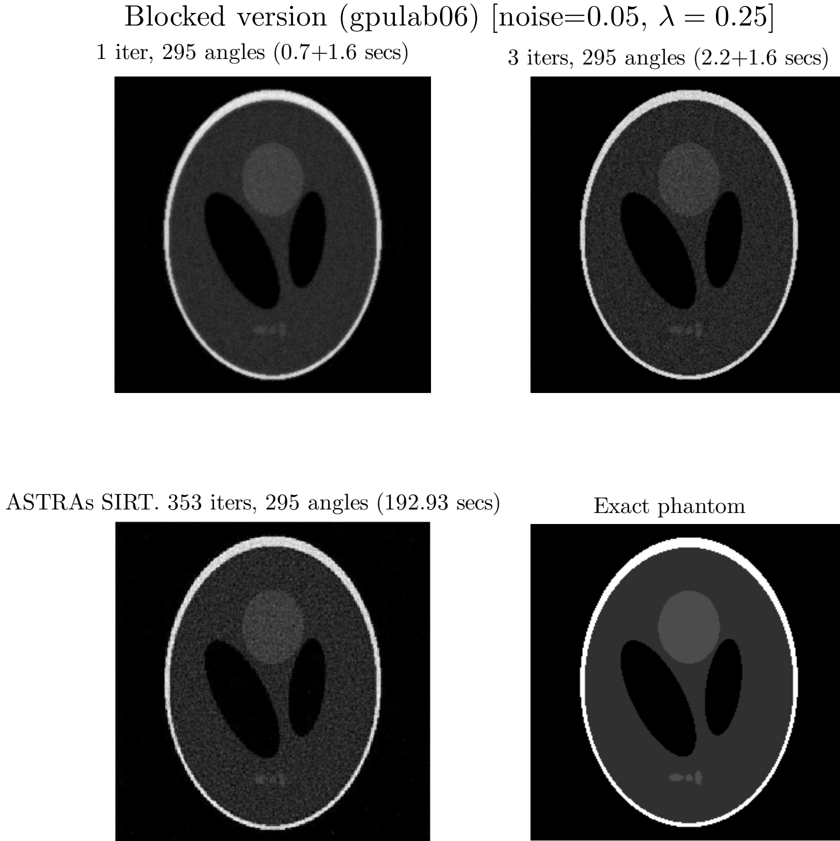


Figure 9.1: Result from a test system with $N = 256$, $p = 295$ and $m = 256$. The projection is corupted with scaled gausian noise, s.t. relative error is 0.05.

method for comparison. And it is clear that the SART method converges much faster, and that it is close to the least squares solution, when the SIRT method reaches the optimal reconstruction. We have also shown the first reconstruction of ASTRAs SIRT method which has an relative error bellow 0.2. We have first made the convergence test, to find the necessary number of iterations. Then the SIRT implementation is tested, and a total reconstruction takes 87.78 sec. Which is still far from the result obtained by the SART implementation.

We have made an similar test of general Block-Iterative implementation, where it is tuned as a SIRT method, i.e. the number of blocks is set to 1. The step length is in this test not fine tuned, but for $\lambda = 0.009$ we have achieved the same

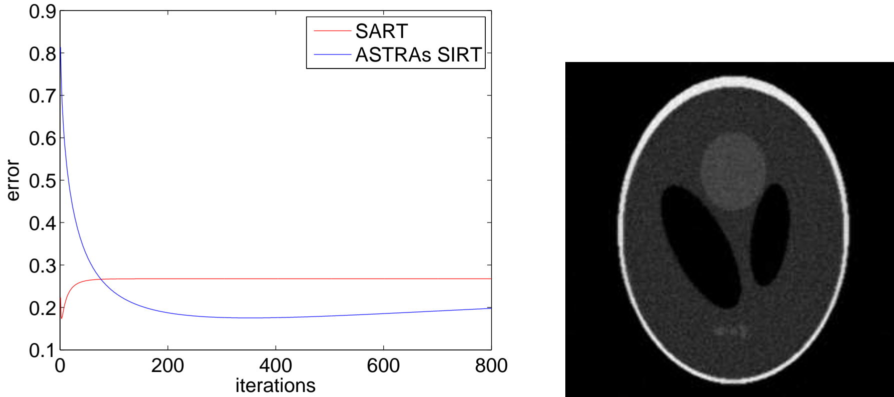


Figure 9.2: ASTRAs SIRT. Result from a test system with $N = 256$, $p = 295$ and $m = 256$. The projection is corrupted with scaled Gaussian noise, s.t. relative error is 0.05. Left: relative error $\frac{\|e\|_2}{\|b\|_2}$ as a function of the number of iterations. Note that, although it is hidden by the y-axis, the best SART reconstruction is not achieved in the first iteration but rather converges within the first 3 iterations. Right: The first reconstruction with a relative error bellow 0.2.

relative error at 80 iterations, which took 57.83 second. The obtained result and the convergence is shown if figure 9.3.

To ensure the correctness the SART method it is also tested on a system, which is not corrupted with noise. This system is similar in size to the one in figure 9.1. In this case it should converge to the exact solution, and from the result seen in figure 9.4, we visually validate the result.

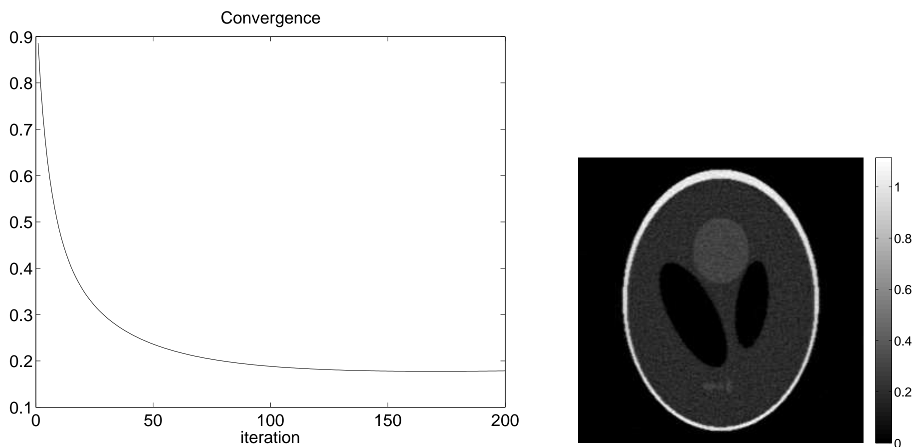
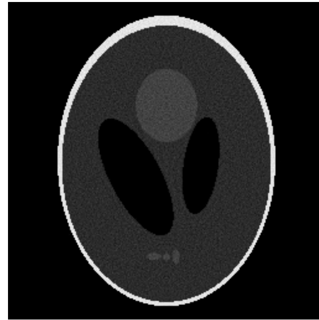


Figure 9.3: Block-Iterative. Result from a test system with $N = 256$, $p = 295$ and $m = 256$. The projection is corrupted with scaled gausian noise, s.t. relative error is 0.05. Left: relative error $\frac{\|e\|_2}{\|b\|_2}$ as a function of the number of iterations. Right: The first reconstruction with a relative error bellow 0.2.

Blocked version (gpulab06) [noise=0.0, $\lambda = 0.8(133)|1.4(295)$]

1 iter, 133 angles (0.8+0.7 secs)

20 iters, 133 angles (15.5+0.7 secs)



1 iter, 295 angles (1.7+1.6 secs)

20 iters, 295 angles (33.5+1.6 secs)

Figure 9.4: Result from a test system with $N = 256$, $p = 295$ and $m = 256$. This system is not corrupted with error.

Chapter 10

Conclusion and future work

During this Thesis we have shown that it is possible to implement efficient *Block-Iterative* methods on General Purpose Graphical Processing Units, using CUDA C, for tomographic reconstruction.

We have started with an existing state of the art package ASTRA, for 3D image reconstruction using CUDA C. From this starting point we have developed methods which is faster and has a more stable runtime for each iteration. These methods has also reduced the total memory footprint considerably.

The ASTRA package utilizes textures in their methods. We have shown that for Block-Iterative methods with a high number of blocks it is more efficient to use methods that utilizes the shared memory.

The methods we have implemented are all based on a *parallel beams* projections. It would be natural to extend the work with a *cone beam* setup.

Further more it would be of interest to implement methods that utilizes multiple GPU's.

Appendix A

Implementation of a general Block-Iterative method

Listing A.1: First SART implementation, which is independent from ASTRA

```
453   for (unsigned int iter = 0; iter < iterations ; ++iter) {
454     ASTRA_CUDA_ASSERT( cudaMallocPitch(  &d_sino, &pitchs,
          ↪ dims.iProjU*sizeof(float), dims.iProjV*anglesPrBlock ));
455     int numBlocksDone = 0;
456     unsigned int angles = 0;
457     for (unsigned int i = 0 ; i < anglesPrBlockUntilID ; ++i ,
          ↪ angles += anglesPrBlock)
458     {
459 numBlocksDone++;
460       size_t tmp
            ↪ =pitchProjections/sizeof(float)*angles*dims.iProjV;
461       ASTRA_CUDA_ASSERT(cudaMemcpy2D( d_sino , pitchs ,
            ↪ &d_projections[tmp] , pitchProjections ,
            ↪ dims.iProjU*sizeof(float) , dims.iProjV*anglesPrBlock
            ↪ , cudaMemcpyDeviceToDevice ));
462       lw = &d_weights[ pitchWeights/sizeof(float) *
            ↪ angles*dims.iProjV ];
463       Par3DFP_multiAngle(D_volumeData,d_sino, pitchs, dims,
            ↪ &par3DProjs[angles],-1,anglesPrBlock);
464       sinogram2dOpMul<<< (dims.iProjU+31)/32,32 >>>(d_sino, lw,
            ↪ pitchs, dims.iProjU, dims.iProjV*anglesPrBlock);
465       cudaDeviceSynchronize();
466       for (int ang =0; ang < anglesPrBlock ; ++ang)
467         Par3DBP_singleAngleV2( D_volumeData,
              ↪ &d_sino[pitchs/sizeof(float)*dims.iProjV*ang],
```

```
        ↪ pitchs, dims, par3DProjs[angles+ang],lambda);
468
469      processVol3DopClampZero(D_volumeData, dims);
470    }
471    cudaFree(d_sino);
472    if(!(numBlocks == numBlocksDone) )
473    {
474    ASTRA_CUDA_ASSERT(cudaMallocPitch(  &d_sino, &pitchs,
        ↪ dims.iProjU*sizeof(float),
        ↪ dims.iProjV*(anglesPrBlock-1) ));
475    for (unsigned int i = 0 ; i < ( numBlocks - numBlocksDone ) ;
        ↪ ++i , angles += anglesPrBlock-1)
476    {
477      size_t tmp
          ↪ =pitchProjections/sizeof(float)*angles*dims.iProjV;
478      ASTRA_CUDA_ASSERT(cudaMemcpy2D( d_sino , pitchs ,
          ↪ &d_projections[tmp] , pitchProjections ,
          ↪ dims.iProjU*sizeof(float) ,
          ↪ dims.iProjV*(anglesPrBlock-1) ,
          ↪ cudaMemcpyDeviceToDevice ));
479      lw = &d_weights[ pitchWeights/sizeof(float) *
          ↪ angles*dims.iProjV];
480      cudaDeviceSynchronize();
481      Par3DFP_multiAngle(D_volumeData,d_sino, pitchs, dims,
          ↪ &par3DProjs[angles],-1,anglesPrBlock-1);
482      sinogram2dOpMul<<< (dims.iProjU+31)/32,32 >>>(d_sino, lw,
          ↪ pitchs, dims.iProjU, dims.iProjV*(anglesPrBlock-1));
483      for (int ang =0; ang < anglesPrBlock-1 ; ++ang)
484        Par3DBP_singleAngleV2( D_volumeData,&d_sino[
            ↪ pitchs/sizeof(float) * dims.iProjV*ang], pitchs,
            ↪ dims, par3DProjs[angles+ang],lambda);
485
486      processVol3DopClampZero(D_volumeData, dims);
487    }
488    cudaFree(d_sino);
489    }
490  }
```

Nomenclature

λ	A relaxation parameter that controls the impact of each update
A	Projection matrix, described in Chapter 2
A_i	Sub matrix consisting of a set of rows from the projection matrix A
a_i	Is the i'th row in the projection matrix A
b	Vectorized version of the projections.
b_i	Sub vector consisting of a set of the elements in b, corresponding to the sub matrix A_i
x	Vectorized version of the discrete domain, described in Chapter 2
ART	Algebraic Reconstruction Technique
ASTRA	All Scale Tomographic Reconstruction Antwerp, is an existing software package for Tomographic Reconstruction, using CUDA C.
back projection	The operation $BP(b) = A^T b$
cache lines	A cache line is 128 bytes and is the minimal size of cache memory that will be affected during a transaction.
coalesced	A data read patteren neighbouring threads is accessing neighbouring elements

CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
device	CUDA C code running on the GPU
DRAM	Dynamic Random Aces Memory is the working memory block, and there is a separate block for the CPU and GPU
forward projection	The operation $FP(x) = Ax$.
GPGPU	General Purpos Graphical Processing Unit
GPU	Graphical Processing Unit
host	C/C++ code that is running on the CPU
kernel	A function that can run on the GPU
L1	L1 cache memory
MPs	Multiprocessors
PCIe	The connection between the CPU and the GPU, which has a low bandwidth of 8GB/s
ray driven	A projection which focus on a fixed ray and make updates accordingly to the voxel it comes in contact with.
Shared memory	Manuel managed memory, which resides on the MPs and the memory block is divided with the L1 cache.
SIRT	Simultaneous Iterative Reconstruction techniques
voxel driven	A projection which focus on a fixed voxel and make updates accordingly to the rays it comes in contact with.
wrapper	A piece of software that allows a program to be used by other programming languages.

Bibliography

[1] H. A. van der Vorst A. van der Sluis. SIRT- and CG-type methods for the iterative solution of sparse linear least-squares problems. *Lin. Alg. Appl*, 130:257–302, 1990.

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

[3] Serge. Belongie. Rodrigues' rotation formula. from mathworld–a wolfram web resource, created by eric w. weisstein. `http://mathworld.wolfram.com/RodriguesRotationFormula.html`.

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

[5] G Cimmino. Calcolo approssimato per le soluzioni dei sistemi di equazioni lineari. *La Ricerca Scientifica*, XVI, 1938.

[6] A. Schoenegger Sabine Pruggnaller A. S. Frangakis D. Castaño-Díez, D. Moser. Performance evaluation of image processing algorithms on the gpu. Volume 164(Issue 1).

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

[8] L. A. Feldkamp, L. C. Davis, and J. W. Kress. Practical cone-beam algorithm. *J Opt Soc Am*, pages 612–619, 1984.

[9] Willem Jan Palenstijn Joost Batenburg and Jan Sijbers. Projection and backprojection in tomography: design choices and considerations.

[10] PETER M. JOSEPH. An improved algorithm for reprojecting rays through pixel images. *IEEE TRANSACTIONS ON MEDICAL IMAGING*, VOL. MI-1:192–196, 1982.

[11] S. Kaczmarz. Angena ̈herte auflo ̈sung von systemen linearer gleichungen. *Bulletin de l'Acad émie Polonaise des Sciences et Lettres*, A35:355–357, 1937.

[12] V.I. Lebedev and D.N. Laikov. "a quadrature formula for the sphere of the 131st algebraic order of accuracy". *Doklady Mathematics*, 59(3):477–481, 1999.

[13] Klaus Mosegaard and Albert Tarantola. Monte carlo sampling of solutions to inverse problems. 100(B7):12431–12447, 1995.

[14] NVIDIA Corporation. *CUDA C Best Practice Guide*, October 2012.

[15] NVIDIA Corporation. *CUDA C Programming Guide*, October 2012.

[16] Z. Sükösd P. C. Hansen, H. O. Sørensen and H. F. Poulsen. Reconstruction of single-grain orientation distribution functions for crystalline materials. 2(2):593–613, 2009.

[17] Vannier M Pan X, Sidky EY. Why do commercial CT scanners still employ traditional, filtered back-projection for image reconstruction? *Inverse Problems*, 25:1230009, 2009.

[18] R. BENDER R. GORDON and G. T. HERMAN. Algebraic reconstruction techniques (art) for three-dimensional electron microscopy and x-ray photography. *J. theor. Biol.*, 29:471–481, 1970.

[19] Hans Henrik B. Sørensen and Per Christian Hansen. Implementation of block algebraic iterative reconstruction methods, 2014.

[20] Nicholas Wilt. *The Cuda Handbook.* Addison-Wesley Pearson Education, edition 1 edition, 2013.