A Fast And High-Quality Cone Beam Reconstruction Pipeline Using The GPU

Thomas Schiwietz[a,c], Supratik Bose[b], Jonathan Maltz[b], Rüdiger Westermann[c]

[a]Siemens Corporate Research, 755 College Road East, Princeton, NJ 08540, USA
[b]Siemens Oncology Care Systems, 4040 Nelson Avenue, Concord, CA 95420, USA
[c]Technische Universität München, Computer Graphics & Visualization Group, Boltzmannstrasse 3, 85748 Garching, Germany

ABSTRACT

Cone beam scanners have evolved rapidly in the past years. Increasing sampling resolution of the projection images and the desire to reconstruct high resolution output volumes increases both the memory consumption and the processing time considerably. In order to keep the processing time down new strategies for memory management are required as well as new algorithmic implementations of the reconstruction pipeline. In this paper, we present a fast and high-quality cone beam reconstruction pipeline using the Graphics Processing Unit (GPU). This pipeline includes the backprojection process and also pre-filtering and post-filtering stages. In particular, we focus on a subset of five stages, but more stages can be integrated easily. In the pre-filtering stage, we first reduce the amount of noise in the acquired projection images by a non-linear curvature-based smoothing algorithm. Then, we apply a high-pass filter as required by the inverse Radon transform. Next, the backprojection pass reconstructs a raw 3D volume. In post-processing, we first filter the volume by a ring artifact removal. Then, we remove cupping artifacts by our novel uniformity correction algorithm. We present the algorithm in detail. In order to execute the pipeline as quickly as possible we take advantage of GPUs that have proven to be very fast parallel processors for numerical problems. Unfortunately, both the projection images and the reconstruction volume are too large to fit into 512 MB of GPU memory. Therefore, we present an efficient memory management strategy that minimizes the bus transfer between main memory and GPU memory. Our results show a 4 times performance gain over a highly optimized CPU implementation using SSE2/3 commands. At the same time, the image quality is comparable to the CPU results with an average per pixel difference of 10^{-5}.

Keywords: Cone beam reconstruction, Filtered Backprojection, Curvature Smoothing, FFT, High-Pass Filtering, Ring Artifact Removal, Cupping Artifact Removal, GPU

1. INTRODUCTION

In the current generation of cone beam scanners, a typical projection image is obtained with 1024×1024 sample points at 12-bit dynamic range. Usually, between 360 and 720 images are acquired during one rotation around the patient. Consequently, high quality data volumes can be reconstructed from those images. On the other hand, the data set size can grow up to several giga bytes and the processing time increases tremendously, too. Furthermore, for accurate and high-quality output volumes all calculations have to performed in 32-bit floating point precision. This improves the image quality over fix-point formats at the price of even more memory consumption and longer processing time. To handle the large amount of data, an efficient memory management strategy is required to achieve fast turn-around times.

In our work, we look at the entire cone beam reconstruction pipeline beginning from the acquired raw data to the reconstructed output volume. Obviously, we cannot address all stages in detail, but we focus on the most

Further author information: (Send correspondence to Thomas Schiwietz)
Thomas Schiwietz: schiwiet@in.tum.de
Supratik Bose: supratik.bose.ext@siemens.com
Jonathan Maltz: jonathan.maltz@siemens.com
Rüdiger Westermann: westermann@in.tum.de

important and interesting stages. More stages can be implemented and integrated easily. In addition to the filtered backprojection (Feldkamp, Davis, Kress[1] (FDK) algorithm), we present the following GPU-accelerated pre- and post-filters in the pipeline:

1. Pre-Filtering: Since raw data is usually noisy we apply a non-linear curvature-based filter to all projection images. The filter smoothes the image in uniform regions but preserves the edges by taking the local curvature and gradient magnitude into account. The algorithm is based on work by Neskovic[2] et al.

2. Pre-Filtering: The inverse Radon transform theory requires the images to be high-pass filtered. We use the FFT to convolve the projection images with a high-pass filter. The GPU-accelerated FFT implementation is based on our previous paper.[3, 4]

3. Backprojection: The projection images are backprojected into the output volume. Our backprojection implementation is voxel-driven. That is, for each voxel in the output volume we ask for the corresponding pixel in the projection images. This is implemented using projective texture mapping described by Segal[5] et al.. Geometry-wise, a voxel is only considered as reconstructed correctly, if a certain percentage of all backprojection images have contributed to it. Therefore, we count the number of rays passing through each voxel and discard voxels below a user-defined minimum ray limit. In contrast to publications by Mueller and Xu,[6] our implementation does not require two output volumes oriented along the axes perpendicular to the patient axis. This makes the combination pass obsolete and saves half of the memory.

4. Post-Filtering: Once a raw volume is reconstructed, reconstruction artifacts are removed by post-processing filters. First, we apply a ring artifact removal. Typically, ring artifacts can be observed along the patient axis due to detector calibration errors. The ring artifacts are extracted using median filtering and circular smoothing. Finally, the isolated ring artifacts are removed from the original image. This filter is based on an algorithm by Zellerhoff[7] et al..

5. Post-Filtering: The second post-processing filter is a cupping artifact removal. Low-frequency artifacts are removed from the output slices. In order to eliminate the artifacts, we convolve every slice with a Butterworth kernel and add the filter response to the slice weighted by a scaling factor.

To improve the reconstruction pipeline speed, we take advantage of the computational power of modern graphics processing units (GPUs). Although GPUs are not Turing-complete, they provide a powerful subset for parallel single instruction multiple data (SIMD)-type operations which outperform current CPUs even with streaming SIMD extension (SSE) 2 or 3 optimization. Furthermore, there is a second type of parallelism available on modern graphics cards: 32 to 48 pixel units exist on board executing SIMD operations in parallel. To better understand the architecture and advantages of using the GPU, we will present a brief overview of GPU programming in Section 2. A good resource for GPGPU related articles, discussion forums and news can be found online.[8] We also use our GPU accelerated implementation of the Fast Fourier transform (FFT) presented in.[3, 4]

In the past years, there have been a couple of cone beam reconstruction papers exploiting the power of GPUs like for example by Mueller and Xu.[6] While they focus mainly on the reconstruction process itself using various algorithms like backprojection and algebraic reconstruction, the focus of our work is on the entire reconstruction pipeline with pre- and post-processing filters in high accuracy using floating-point precision.

In order to get the maximum performance of a GPU all computations have to be executed by the GPU and the bus transfer between GPU memory and main memory must be minimized. Therefore, we implement the entire reconstruction pipeline using GPU programs (shaders). Our goal was to upload the projection images to GPU memory right after acquisition and download only the final output volume. Unfortunately, the memory consumption is very high. Therefore, we split the volume into several parts (chunks) and reconstruct them separately. We will discuss our memory management strategy in Section 4.

The results show that our implementation is both fast and accurate. Including all bus transfer time, our implementation is about 4 times faster than our SSE3 optimized CPU implementation at full 32-bit floating-point precision. The average per-pixel error is in the order of magnitude 10^{-5}. Section 5 presents the results in detail.

The remainder of this paper is organized as follows. We present an overview of GPU programming in Section 2. In Section 3 we discuss the stages of our GPU accelerated reconstruction pipeline. In Section 4 we present our memory management algorithm and the corresponding pipeline algorithm. Timing and accuracy results are shown in Section 5. Finally, we conclude our work in Section 6.

2. GPU PROGRAMMING

This section is taken from our previous publication.[4] We have included the section to give the reader a short introduction to GPU programming.

We briefly review the use of the GPU as a co-processor for arbitrary algorithms, known as *general-purpose* GPU (*GPGPU*) programming. For a more comprehensive introduction, we refer to the books[9–12] and to the website.[8]

First, we describe how graphics cards handle data storage. Graphics cards provide readable and writable data structures in GPU memory. Basically, there are three types of data structures available: 1D, 2D, and 3D arrays, all referred to as *textures*. Among them, 2D textures provide the fastest update performance. The array elements of a texture are called *texels*. Each texel can have up to four float-typed components. The four components are often called the *RGBA* channels, as they are originally used to represent the red, green, blue, and alpha intensities of a color for rendering.

To set values in a texture, the GPU processing *pipeline* consisting of several stages is utilized. This procedure is also referred to as "rendering" which is carried over from the original convention when screen pixels are drawn by the pipeline. Figure 1 illustrates three important stages in the graphics card pipeline, which we elaborate below.

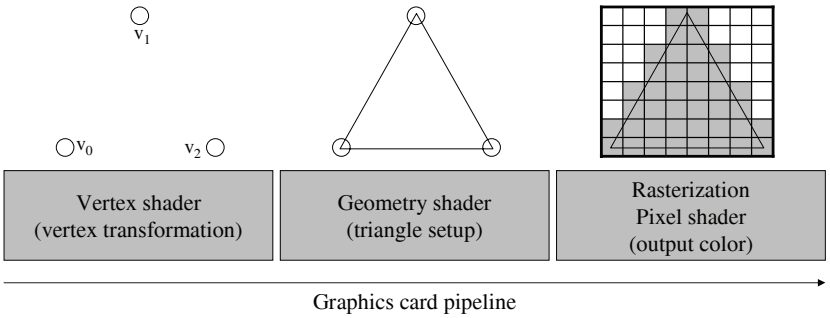


Figure 1. The graphics card pipeline with the three most important stages: the vertex shader, the geometry shader and the rasterizer/pixel shader.

- *Vertex shader*: A stream of *vertices* enters the vertex shader stage. A vertex is a data structure on the GPU with attributes such as position vectors for certain coordinate systems. In this stage, vertex attributes are manipulated according to the objectives of the applications. Traditionally, user-defined vertex shader programs are used to transform position vectors from one space to another.
- *Geometry shader (triangle setup)*: In this stage, sets of three vertices are selected to setup triangles according to a geometry shader program.
- *Rasterization / Pixel shader*: Using the three vertices of each triangle as knot points, a scanline algorithm generates texels that are circumscribed by the triangle. All vertex attributes are interpolated bilinearly. User-defined pixel shader programs are executed for each rasterized texel, which can access and work with the interpolated vertex attributes. The following types of instructions are available in this stage.
  - *Arithmetical instructions* perform addition, multiplication, exponent, etc.

– *Data access instructions* allow reading values from GPU memory.
– *Control flow instructions* allow branching and loops.

A large number of temporary registers is available for intermediate results in the program. Each texel is rasterized and processed independently and potentially in parallel. However, no processing order can be assumed by the programmer. The return value of a pixel shader program is primarily a four-component vector, which is explained in detail next.

Particularly, the texture to be updated is set as the *render target*. This implies that the pixel shader now writes values to the texture instead of the screen. The area of the texture that the programmer wants to write to is covered by a quadrilateral defined by four vertices and two triangles. Using the graphics card pipeline, a pixel shader program is executed for each texel in the target texture. With this procedure, arbitrary calculations can be performed. In the context of the reconstruction pipeline, the projection images and the output volume are represented as textures and updated using pixel-shader programs.

Currently, the two major graphics programming APIs are *DirectX* and *OpenGL*. Both APIs provide similar functionality for graphics cards programming. Additionally, there are so called *shading languages* to program the vertex and pixel shader units on the graphics card. The DirectX shading language is called high-level shading language (*HLSL*); and the one for OpenGL is the OpenGL shading language (*GLSL*). Both languages provide similar functionality and their syntax is closely related to the C language. Our implementations have been written in DirectX with HLSL.

3. PIPELINE STAGES

In this section we describe the pipeline stages that we have accelerated using the GPU. Our pipeline consists of the following five stages

1. Curvature Smoothing (projection image pre-processing)
2. High-pass filter (projection image pre-processing)
3. Backprojection (volume reconstruction)
4. Ring artifact removal (volume post-processing)
5. Cupping artifact removal (volume post-processing)

In the following we describe the details of theses stages.

3.1. Curvature Smoothing

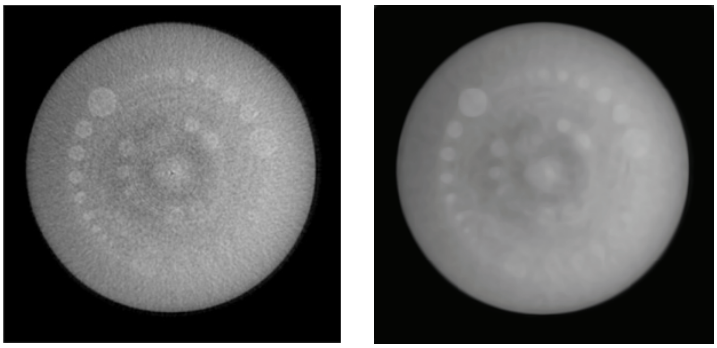
Usually, the projection image raw data is noisy. The quality of the reconstructed output volume can be improved dramatically, if the noise in the projection images is reduced. An isotropic low-pass filter is not suitable because it not only reduces the noise but smoothes out the edges, too. Instead, we use a non-linear curvature based filter. The algorithm smoothes the image iteratively according to the local curvature and gradient magnitude. It reduces noise while it preserves edges with larger gradient magnitude. The algorithm has been published by Neskovic et al..[2] In the following, we describe the details of the algorithm:

A continuous scalar field ϕ is smoothed iteratively by the function $\phi^{i+1} = \phi^i + \beta C|\nabla\phi|$ where C is the mean curvature

$$C = \frac{1}{2}(\kappa_1 + \kappa_2) = \frac{|\nabla\phi|^2 trace(H(\phi)) - \nabla\phi^T H(\phi)\nabla\phi}{2|\nabla\phi|^3}, \tag{1}$$

$H(\phi)$ denotes the Hessian matrix of ϕ and β is a free parameter in range $[0;1]$. Equation 1 is derived from the fundamental forms as described by Farin[13] and Eberly.[14] In two dimensions, Equation 1 evaluates to

$$C = \frac{\frac{\partial^2\phi}{\partial x^2}(\frac{\partial\phi}{\partial y})^2 + \frac{\partial^2\phi}{\partial y^2}(\frac{\partial\phi}{\partial x})^2 - 2\frac{\partial\phi}{\partial x}\frac{\partial\phi}{\partial y}\frac{\partial^2\phi}{\partial x\partial y}}{2|\nabla\phi|^3} \tag{2}$$



(a) (b)

Figure 2. (a) A noisy input image. (b) The same image after eight iterations of curvature smoothing. Obviously, the noise is gone and the edges of the bright circles are still sharp.

We discretize Equation 2 using central differences

$$\begin{aligned}
\frac{\partial \phi}{\partial x} &= \frac{p_{i+1,j} - p_{i-1,j}}{2}, \\
\frac{\partial \phi}{\partial y} &= \frac{p_{i,j+1} - p_{i,j-1}}{2}, \\
\frac{\partial^2 \phi}{\partial x^2} &= p_{i+1,j} - 2p_{i,j} + p_{i-1,j}, \\
\frac{\partial^2 \phi}{\partial y^2} &= p_{i,j+1} - 2p_{i,j} + p_{i,j-1}, \\
\frac{\partial^2 \phi}{\partial x \partial y} &= \frac{(p_{i-1,j-1} + p_{i+1,j+1}) - (p_{i-1,j+1} + p_{i+1,j-1})}{4}
\end{aligned} \tag{3}$$

where $p_{i,j}$ denotes the pixel intensity at the raster location (i, j). The curvature C is weighted by the gradient magnitude $|\nabla\phi|$ for normalization purposes. In order to prevent a division by zero in Equation 2, the algorithm returns 0 if the gradient magnitude $|\nabla\phi|$ gets small. Since no derivatives can be computed in the boundary region accurately, we use an out-flow boundary condition: the curvature of the direct neighbor perpendicular to the border tangent is replicated. After that, the image pixels $p_{i,j}$ are updated by the weighted curvature $\beta C|\nabla\phi|$. Usually, four to six iterations are sufficient for satisfying results.

We implement the algorithm using the GPU as follows: Two textures are allocated: one for the input image ϕ that is updated in each iteration, and the other one for the curvature C. A pixel shader program computes the curvature and writes the result to the curvature texture. With the discretization in Equation 3 only direct neighbors have to be accessed. The boundary condition is realized by rendering eight line primitives: four side borders and four corners. Texture coordinates are specified to address the source location where to copy the curvature from. Afterwards, another shader program updates the image texture $\phi$with respect to β.

3.2. High-Pass Filtering

According to the inverse Radon transformation, a high-pass filter has to be applied to the projection images before the backprojection takes place. Commonly used filters are the *Ram-Lak*, *Shepp-Logan*, or the *Hamming* filter. In our work, we convolve the image with the Ram-Lak filter. We transform the image to frequency domain, multiply the result pixel-wise by $|f|$ where f is the frequency, and transform the result back to spatial domain. Implementation-wise we use our FFT implementation[3, 4] on the GPU for the transformations.

3.3. Backprojection

Principally, our backprojection implementation works voxel-driven. That is, we ask for each voxel in the volume what the corresponding pixels in the projection image are. We use 2D textures to represent both the projection

images and the slices of the output volume in GPU memory. The slices of the volume are updated using the techniques described in Section 2 by rendering a quadrilateral covering the entire texture. 3D texture coordinates in projection space are specified at each vertex of the slices. Using the rasterization units the coordinates at the vertices are interpolated across the slice. A pixel shader program fetches the interpolate coordinate inside the slice and performs the perspective division to project it to two dimensions. Next, the resulting 2D coordinate is checked against the margin of a valid area in the projection image. The valid area of projection image is user-defined. Only if the coordinate is inside the valid area the projection image is sampled and accumulated to the current slice of the volume weighted by the inverse squared depth. If the sample location is outside the valid area, the current voxel does not receive a contribution from the current projection image. We count the number of rays passing from the valid area of all projection images through all voxels. Only voxels with a user-defined percentage of valid rays hits are considered as reconstructed correctly. Invalid voxels are set to zero in post-processing.

3.4. Ring Artifact Removal

Ring artifacts appear in the reconstructed volume because of detector calibration errors. Since the detector calibration can never be completely accurate, a post-processing filter helps to reduce the ring artifacts that arise typically. The algorithm is a 2D filter that is applied to all slices of the volume separately as the ring artifacts appear equally in each slice and not volumetrically. Basically, the algorithm extracts the ring artifacts and subtracts them from the original image. In detail, the algorithm works in four steps:

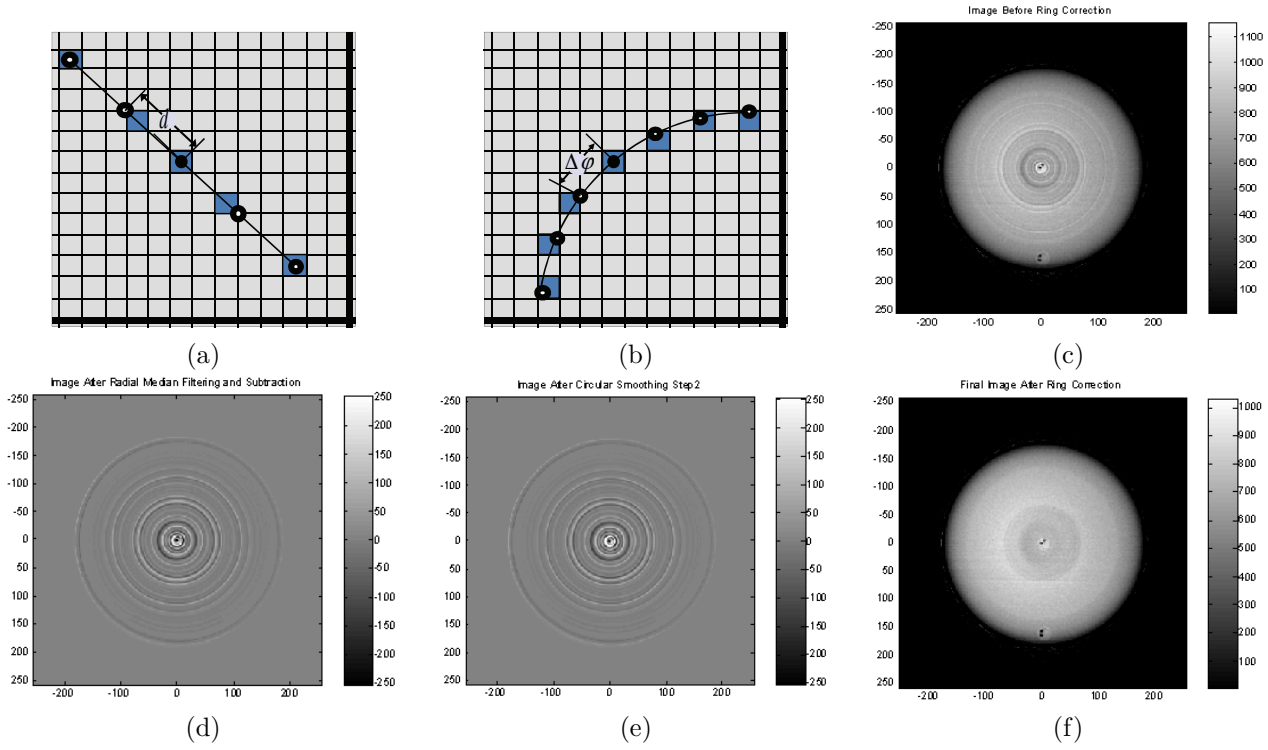


Figure 3. Figure (a) shows the sampling pattern for the median filter. The median filter samples four additional values in the distances $[-2d; -d; d; 2d]$ along the radial line connecting the current pixel to the center of the image in the lower right corner. Image (d) shows a median filtered image that was subtracted from the original input image. Figure (b) shows the sampling pattern for the circular smoothing: it smoothes along the circular neighborhood. The circle mid-point is always located in the center of the image while the radius is the distance to the pixel to be smoothed. The samples are taken in equidistant angular steps $\Delta\varphi$. Image (e) shows the effect of the filter applied on Image (d). Image (c) shows a slice with significant ring artifacts. After applying the ring artifact removal algorithm the Image (f) is free of artifacts.

1. *Dynamic range restriction*: The dynamic range of the image is clamped to 11-bit $[0; 2048]$ to avoid introducing artifacts from objects with high contrast.

2. *Median Filtering*: Each pixel is filtered by the median of five samples along the radial line to the image center. Figure 3 (a) depicts the radial line intersecting the image center at the lower right corner. The distance d between the sample points is the ring width depending on the gantry geometry. With a median of five samples the sample position are located at the distances $[-2d; -d; 0; d; 2d]$ along the radial line. Finally, the median is subtracted from the original image in order to extract the ring artifacts.

3. *Circular Smoothing*: The extracted ring artifact image is refined by a circular smoothing filter in order to reduce the noise. The center of the circles is always located in the image center while the radius is the distance to the image center. The filter averages values along the circular neighborhood. We use a constant angular distance $\Delta\varphi$ between the sample points. In our implementation we use six samples in both directions on the circle resulting in the average of 13 samples. Figure 3 (b) depicts the geometry of the circular sampling locations.

4. *Correction*: The extract ring artifacts are subtracted from the original image.

The GPU implementation precomputes the sample locations for both the median and the circular filter pixel-wise. That is, we use Cartesian coordinates instead of polar coordinates in the precomputed tables. Furthermore, we store the linear addresses of the sample locations in the texture components to save memory. The linear address l at location (x, y) is computed with $l(x, y) = (x \cdot w) + y$ where w is the image width. The inverse functions $l^{-1}(a)$ are defined as $l_x^{-1}(a) = (int)a\%w$ and $l_y^{-1}(a) = (int)a/w$. The median filter requires four additional sample along the radial line for each pixel. A four component texture is sufficient to store the locations. A pixel shader program delinearizes the four positions, samples the values and computes the median by bubble sorting the values and returning the middle element subtracted from the pixel value in the original image. Similarly, the circular smoothing filter precomputes the sample locations in textures. Using the same addressing scheme, three textures with four components are allocated to store the 12 sample locations. A pixel shader program samples the 13 values and returns the average. In a final pass, the extracted ring artifacts are subtracted from the original image.

3.5. Cupping Artifact Removal

Cupping artifacts occur where a significant proportion of the lower energy photons in the beam spectrum have been scattered and absorbed. Typically, the artifacts are visible as low frequency error resulting in a non-uniform intensity distribution inside the image. In order to correct the cupping artifacts the low frequency errors must be identified and corrected. It can be observed that the cupping artifacts appear in range 0.4 to 4 cycles per image dimension. This filter band needs to be amplified and added to the original image. Formally, it can be expressed by the equation

$$f_{corr}(x, y) = F^{-1}[F[f(x, y)](1 + kW(u, v))] \tag{4}$$

where $f(x, y)$ is the two dimensional input image, $f_{corr}(x, y)$ is the corrected image, $F(x, y)$ is the Fourier transform of the image $f(x, y)$, $W(u, v)$ is a low-pass filter to extract the cupping artifacts and k is a filter gain scalar. We use the the Butterworth filter for $W(u, v)$

$$W_n(\omega) = \frac{1}{\sqrt{1 + (\frac{\omega}{\omega_h})^{2n}}} - \frac{1}{\sqrt{1 + (\frac{\omega}{\omega_l})^{2n}}}, \tag{5}$$

where $\omega = \sqrt{u^2 + v^2}$ the angular frequency in radians per second, ω_l the low cut off frequency, ω_h the high cut off frequency, and n the order of the filter. In particular, the following parameter values are used:

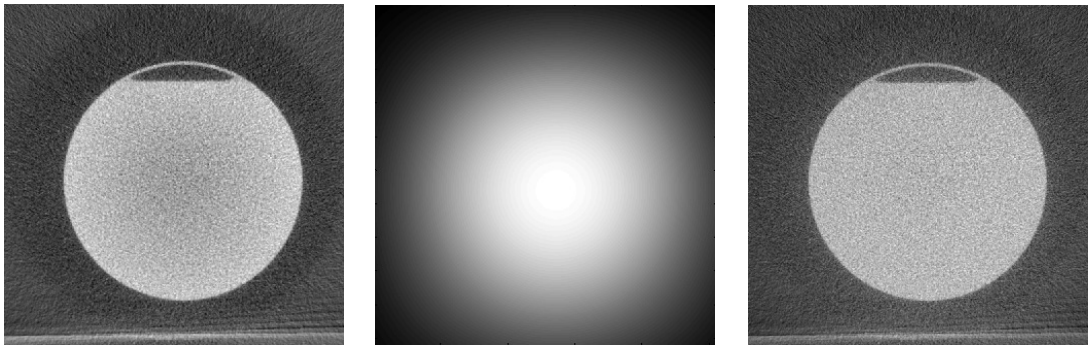
- Low cut off frequency ω_l: 2 cycles per meter
- High cut off frequency ω_h: 3 cycles per meter

- Filter order n: 2
- Filter gain k: 2.25

The filter response is the low frequency cupping artifacts residing in the image. It is added to the original image weighted by the filter gain k and shifted by $(avg + max)/2$ with the average and maximum value of the in the filter response. Figure 4 shows an input image (a) convolved with a Butterworth kernel (b). Image (c) is the (weighted) addition of image (a) and (b).

This algorithm has the following properties:

- Translation invariant: low sensitivity to location of the object center
- Low sensitivity to the object size
- Little effect when cupping artifacts are not present
- Very fast as only one forward/backward 2D FFT has to be computed



(a) (b) (c)

Figure 4. (a) An input image. Typical cupping artifacts can be seen in the center of the image. The contrast falls off towards the center of the image. (b) The filter response after convolving the input image with the Butterworth filter. (c) The image after cupping artifact correction. Now, the slice has a uniform intensity distribution.

In our GPU implementation, we first calculate the Butterworth filter kernel and store the filter coefficients in a 2D texture. Then, we convolve the filter kernel with the input image using our GPU-based implementation of the FFT.[3,4] In order to compute the correct weight and shift for the correction, the maximum and average value in the filter response have to be determined. Therefore, we use two reduce operations on the filter response texture: a reduce maximum and a reduce average.[15] Now, the DC of the filter response is corrected by subtracting $(avg + max)/2$ from each pixel weighted by the user-defined filter gain k. Finally, the original image is added resulting in an image without cupping artifacts.

With all pipeline stages, our reconstructed output volume is now of very high quality.

4. MEMORY MANAGEMENT

Since GPU memory is limited and usually not big enough to store all projection images and the output volume, a memory management strategy is necessary. In this section, we discuss two memory management strategies and derive reconstruction pipeline algorithms from them.

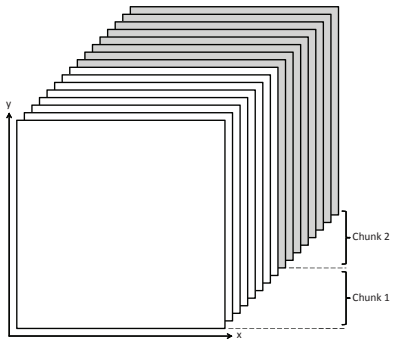


Figure 5. The output volume is divided into a number of chunks. A chunk is a stack of slices representing a part of the entire volume.

4.1. Data Swapping

Nowadays, graphics cards have typically 512 MB of RAM. One can easily see that this is not enough memory to reconstruct a volume with $512 \times 512 \times 512$ voxels in floating-point precision. A volume like this takes 512 MB. This does not fit into GPU memory since the graphics card driver uses some megabytes for internal buffers and, moreover, the projection images need to be stored in GPU memory as well. Therefore, a memory management strategy is required that swaps data between main memory and GPU memory. In general, both projection images and slices of the output volume can be swapped. Since the output volume does not fit into memory in one piece, we divide the output volume in *chunks* as depicted in Figure 5. A chunks is a stack of slices representing a part of the entire volume. The slices of all chunks resemble the entire output volume. We describe two swapping strategies for projection images and chunks in the following, but hybrid forms are possible also. In particular, we are interested in the amount of bus traffic between main memory and GPU memory. This is a bottleneck and must be minimized for optimal performance. To quantify the bus traffic we denote the number of projection images as p_n and the size of one image in bytes as p_s. Similarly, we denote the number of chunks as c_n and the size of one chunk in bytes as c_s. In this context, we call copying data from main memory to GPU memory *upload* and vice versa *download*.

Two swapping strategies:

1. *Swapping chunks*: The projection image is the central pipeline element that stays in GPU memory until it is not needed anymore ($p_n p_s$ bytes upload, 0 bytes download). Each projection image is first pre-processed and then backprojected to the volume. Since the volume does not fit into GPU memory in one piece, all chunks have to be swapped in and out for each projection image ($p_n \cdot c_n c_s$ bytes upload and download).

2. *Swapping projection images*: First, all projection images are pre-processed ($p_n p_s$ bytes upload) and then stored in main memory ($p_n p_s$ bytes download). Next, the chunks are processed sequentially. For each chunk, all projection image are backprojected. That means that all projection images have to be uploaded for each chunk ($c_n \cdot p_n p_s$ bytes upload). Once a chunk is reconstructed completely, post-processing filters can be applied directly before it is downloaded ($c_n c_s$ downloads).

What strategy produces the smaller bus traffic depends on the size of a projection image and the size of the output volume. A typical scenario for us looks like this: We have $p_n = 360$ projection images with 512×512 float-valued pixels $p_s = 1$ MB and the desired output volume with $512 \times 512 \times 512$ floating-point voxels is divided into $c_n = 3$ chunks. Each chunk takes about $c_s = 170$ MB. Swapping chunks causes about 360 GB traffic, while swapping projection images causes only 2,25 GB of traffic. Therefore, our implementation is based on the second approach.

Swapping chunks	bytes upload	bytes download
Projection images	$p_n p_s$	0
Chunks	$c_n c_s \cdot p_n$	$c_n c_s \cdot p_n$
Swapping projection images	**bytes upload**	**bytes download**
Projection images	$p_n p_s + c_n \cdot p_n p_s$	$p_n p_s$
Chunks	0	$c_n c_s$

Table 1. Bus transfer in memory swapping strategies where p_n is the number of projection images and c_m is the number of chunks. The size in bytes represented by p_s and c_s respectively.

4.2. Swapping Projection Images Algorithm

Swapping projection images leads to the following reconstruction algorithm (in pseudo code):

```
for all projection Images
{
    Upload projection image to GPU memory
    Pre-Processing: Curvature Smoothing
    Pre-Processing: High-pass Filtering
    Download projection image to main memory
}

for all chunks
{
    for all projection images
    {
        Upload projection image to GPU memory
        Backproject image onto all slices of the chunk
    }

    Post-Processing: Ring Correction on all slices of the chunk
    Post-Processing: Cupping Artifact Removal on all slices of the chunk
    Download chunk to main memory
}
```

First, all projection images are preprocessed using the GPU. Therefore, the images are uploaded to GPU memory, the curvature smoothing filter and the high-pass are applied and the results are downloaded and saved in main memory. Then, the output volume is reconstructed chunk-by-chunk. All projection images are uploaded sequentially to GPU memory and backprojected to all slices of the chunk. Afterwards, the ring artifact removal and the cupping artifact removal are applied to the chunk in post-processing. Finally, the chunk is downloaded to main memory. This procedure is repeated for all chunks.

5. RESULTS

All our measurements were done using Windows XP on a Xeon Dual Core Processor with 3.2 GHz processor and 2 GB RAM. The graphics card is an NVIDIA Quadro FX 4500 with 512 MB RAM. Table 2 shows timings on the CPU and on the GPU of three different data sets. We have reconstructed the data sets with varying output volume resolutions. Conclusively, our GPU implementation is approximately four times faster than our SSE2 optimized CPU implementation.

	CPU time (seconds)	GPU time (seconds)
input: 360 images (1024×1024) output: $512 \times 512 \times 256$ voxel	178	53
input: 200 images (1024×1024) output: $256 \times 256 \times 256$ voxel	35	18
input: 200 images (1024×1024) output: $512 \times 256 \times 256$ voxel	123	40

Table 2. We have measured the time of an entire reconstruction pipeline run including all pre-processing filters, backprojection and post-processing. This table shows timings for three different data sets using our CPU and GPU implementations. The data sets vary in the size and number of projection images as well as the output volume size.

6. CONCLUSION

We have presented a GPU implementation of a fast and high-quality cone beam reconstruction pipeline with two pre-processing image filters, backprojection and two post-processing image filters. Additional pipeline stages can be integrated easily. With the help of this implementation and the architecture of the GPU, we are able to improve the speed performance of a reconstruction run by a factor of 4. In addition, the resulting image quality is virtually the same for both CPU-based and GPU-based implementations with an average pixel error of 10^{-5}. From our experiments, we have shown that the GPU is very suitable not only for backprojection, but for a entire reconstruction pipeline with very high accuracy. With increasingly growing amount of data, it is possible that the GPU-based cone beam reconstruction pipeline will be indispensable in the future.

REFERENCES

1. L. A. Feldkamp, L. C. Davis, and J. W. Kress, "Practical cone-beam algorithm," *Optical Society of America Journal A* **1**, pp. 612–619, June 1984.
2. P. Neskovic and B. B. Kimia, "Geometric smoothing of 3d surfaces and non-linear diffusion of 3d images."
3. T. Schiwietz and R. Westermann, "GPU-PIV," in *VMV*, pp. 151–158, 2004.
4. T. Schiwietz, T. Chang, P. Speier, and R. Westermann, "MR image reconstruction using the GPU," in *Medical Imaging 2006: Visualization, Image-Guided Procedures, and Display. Proceedings of the SPIE (2006).*, pp. 646–655, Apr. 2006.
5. M. Segal, C. Korobkin, R. van Widenfelt, J. Foran, and P. Haeberli, "Fast shadows and lighting effects using texture mapping," *SIGGRAPH Comput. Graph.* **26**(2), pp. 249–252, 1992.
6. F. Xu and K. Mueller, "Accelerating popular tomographic reconstruction algorithms on commodity PC graphics hardware," *IEEE Transaction of Nuclear Science* , 2005.
7. M. Zellerhoff, B. Scholz, E.-P. Ruehrnschopf, and T. Brunner, "Low contrast 3D reconstruction from C-arm data," in *Medical Imaging 2005: Visualization, Image-Guided Procedures, and Display. Edited by Galloway, Robert L., Jr.; Cleary, Kevin R. Proceedings of the SPIE, Volume 5745, pp. 646-655 (2005).*, M. J. Flynn, ed., pp. 646–655, Apr. 2005.
8. Information about GPGPU programming. `http://www.gpgpu.org`.
9. J. D. Foley, A. van Dam, S. K. Feiner, and J. F. Hughes, *Computer graphics (2nd ed. in C): principles and practice*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1996.
10. S. Thilaka and L. Donald, *GPU Gems 2*, ch. Medical Image Reconstruction with the FFT, pp. 765–784. Addison-Wesley, 2005.
11. M. Woo, Davis, and M. B. Sheridan, *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 1.2*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.
12. K. Gray, *Microsoft DirectX 9 Programmable Graphics Pipeline*, Microsoft Press, Redmond, WA, USA, 2003.
13. G. Farin, *Curves and surfaces for CAGD: a practical guide*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.
14. D. H. Eberly, *3D Game Engine Design*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001.

15. J. Krüger and R. Westermann, "Linear algebra operators for GPU implementation of numerical algorithms," *ACM Transactions on Graphics (TOG)* **22**(3), pp. 908–916, 2003.