Electronic Letters on Computer Vision and Image Analysis 12(1):1-16, 2013

Comprehensive Analysis of High-Performance Computing Methods for Filtered Back-Projection

Christian B. Mendl[1], Steven Eliuk[2], Michelle Noga[2] and Pierre Boulanger[2]

Christian B. Mendl and Steven Eliuk are equal contributors

[1] *Mathematics Department, Technische Universität München, Boltzmannstraße 3, 85748 Garching, Germany*

[2] *Servier Virtual Cardiac Centre, Department of Radiology and Diagnostic Imaging, University of Alberta, Canada*

Email: mendl@ma.tum.de, sneliuk@ualberta.ca, mnoga@ualberta.ca, pierreb@cs.ualberta.ca

Received 18th August 2012; accepted 20th Feb 2013

Abstract

This paper provides an extensive runtime, accuracy, and noise analysis of Computed Tomography (CT) reconstruction algorithms using various High-Performance Computing (HPC) frameworks such as: "conventional" multi-core, multi threaded CPUs, Compute Unified Device Architecture (CUDA), and DirectX or OpenGL graphics pipeline programming. The proposed algorithms exploit various built-in hardwired features of GPUs such as rasterization and texture filtering. We compare implementations of the Filtered Back-Projection (FBP) algorithm with fan-beam geometry for all frameworks. The accuracy of the reconstruction is validated using an ACR-accredited phantom, with the raw attenuation data acquired by a clinical CT scanner. Our analysis shows that a single GPU can run a FBP reconstruction 23 time faster than a 64-core multi-threaded CPU machine for an image of 1024×1024. Moreover, directly programming the graphics pipeline using DirectX or OpenGL can further increases the performance compared to a CUDA implementation.

Key Words: X-ray imaging and computed tomography, image reconstruction – analytical methods, parallel computing

1 Introduction

Interactive Computed Tomography (CT) reconstruction techniques are critical for assistance during surgery [11], patient monitoring, and closed-loop surgical manipulations. Reliable and fast image

Correspondence to: <mendl@ma.tum.de>

Recommended for acceptance by <X. Otazu>
ELCVIA ISSN:1577-5097
Published by Computer Vision Center / Universitat Autònoma de Barcelona, Barcelona, Spain

reconstruction of clinical CT scans is essential to reduce both X-ray dose and accelerate reconstruction runtime, allowing for more sophisticated iterative update filters to enhance reconstruction quality. These new reconstruction algorithms profit considerably from the broad availability and exponential performance growth of consumer GPUs [16, 13, 14]. For example, Okitsu et.al. [13] optimize the Feldkamp-Davis-Kress (FDK) reconstruction algorithm using the CUDA framework, and achieve higher performance than the standard graphics pipeline. On the other hand, Xu and Mueller [16] claim that the graphics pipeline can be faster with the same algorithm when balancing the load among the pipeline components. Analogous to GPU implementations, several studies focus on multi-core CPU frameworks. For example, Zeng et.al. [17] integrate several techniques (like optimizations of data structures and multi-threading) to optimize the FBP algorithm on 2 cores. Hofmann et.al. [7] find that multi-core CPUs are on equal levels as GPUs for 3D reconstruction.

In this paper, we directly compare these High Performance Computing (HPC) frameworks for Computed Axial Tomography (CAT) with *fan-beam* geometry. We provide and compare implementations of the FBP algorithm, using:

- "conventional" multi-core CPUs, employing a highly parallelizable and multi-threaded reconstruction algorithm;
- CUDA, treating the GPU as multiprocessor;
- DirectX HLSL (High Level Shading Language) to exploit the built-in hardwired graphics components of the GPU.

To test our programs, we reconstruct an American College of Radiology (ACR) accredited phantom [10] from the raw attenuation data acquired by a Siemens Definition Flash 128-slice CT-scanner. We analyze the implementations in terms of running time as well as accuracy by comparing the results with a double-precision reference implementation. Finally, we investigate the Noise Power Spectrum (NPS) of the overall scanning and reconstruction procedure using homogeneous areas located on the phantom.

In our investigation, we omit a comparison with OpenCL due to its close similarity to CUDA. One expects similar results if the graphics drivers for CUDA and OpenCL are equally well optimized.

Current methods for cone-beam CT focus on the use of FDK-type algorithms that use a perspective transform to accelerate the reconstruction process. These methods require a flat detector array and frequently use re-binning techniques to accelerate the reconstruction [1]. However, re-binning algorithms tend to introduces additional non-uniformities in the projection space and should be avoided in the interest of accuracy [6, 1]. For this reason, we investigate reconstruction algorithms which directly work with the raw fan-beam attenuation values produced by commercial CT scanners. Moreover, if projections are acquired cyclically, the direct algorithms can start the reconstruction process immediately, as opposed to the re-binning approach.

Outline: Section 2 provides a brief review of the FBP algorithm, which relies on the integral in Eq. (1) below. It is very accurate since the scanner geometry is taken into account during reconstruction and various filters are applicable in the convolution stage. In Section 3, we provide technical details of the implementations. We present a thorough performance evaluation on various High Performance Computing (HPC) architectures and quantify noise and error in Section 4. Specifically, our analysis includes a runtime study of all methods based on various hardware platforms, with single- as well as double-precision floating-point formats.

2 Theoretical Framework of the FBP Algorithm

In this section, we briefly review the theoretical framework of the FBP algorithm [8, 5, 15, 4] which can be traced back to the original work of Radon.

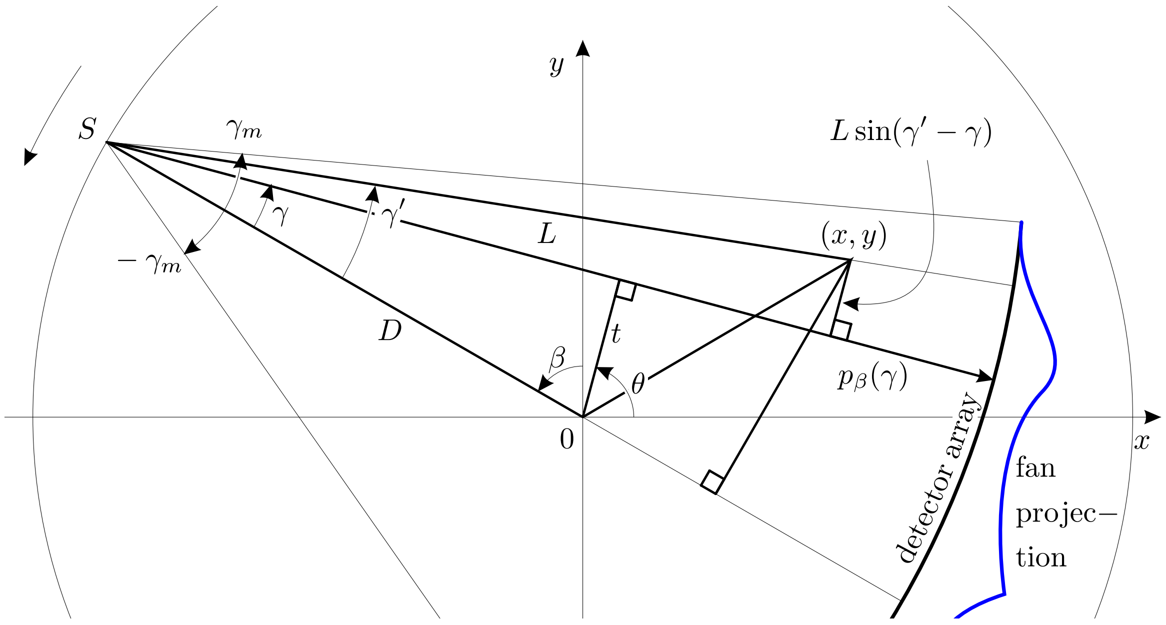


Figure 1: Fan-beam geometry (adapted from Ref. [8]): the source S emits nondiffracting (X-ray) radiation, which is attenuated when passing through the subject (patient at the origin of the coordinate system, not shown). The attenuated radiation reaches the detectors on the opposite side, which are arranged on a circular fan. Thus, each individual detector can uniquely be identified via its fan-angle γ. During the scan, the whole geometry rotates around the stationary patient, such that the rotation angle β traverses 360°.

The fan-beam geometry is illustrated in Fig. 1. A punctual radiation source S and a corresponding array of detectors rotate around the subject at the origin (rotation angle β), with the detectors arranged on a circular arc (fan-angle γ). Let D denote the fixed distance between S and the rotation center. Each measured fan projection $p_\beta(\gamma)$ (or "ray") is uniquely identified by β and γ. According to the derivation in Ref. [8], one can write the reconstruction equation for the 2-D case as:

$$f(x,y) = \frac{1}{2}\int_0^{2\pi} W_\beta\big(\mathrm{Rot}(\beta)\,(x,y)^T\big)\ \mathrm{d}\beta \tag{1}$$

with the rotation matrix $\mathrm{Rot}(\beta) := \left(\begin{smallmatrix}\cos\beta & \sin\beta\\ -\sin\beta & \cos\beta\end{smallmatrix}\right)$ and

$$W_\beta(x,y) := \frac{1}{L(x,y)^2}\,\tilde{Q}_\beta(\gamma'(x,y)),$$

where

$$L(x,y) := \left\|(x, D-y)^T\right\|_2 = \sqrt{x^2+(D-y)^2}, \tag{2}$$

$$\gamma'(x,y) := \arctan\left(\frac{x}{D-y}\right), \tag{3}$$

and $\tilde{Q}_\beta(\gamma')$ defined below. The equation for L and γ' can be directly obtained from Fig. 1 when setting $\beta = 0$. The function $\tilde{Q}_\beta$ is defined as a convolution:

$$\tilde{Q}_\beta(\gamma') := (\tilde{p}_\beta \star c_f)(\gamma') = \int_{-\gamma_m}^{\gamma_m} \tilde{p}_\beta(\gamma)\, c_f(\gamma' - \gamma)\, \mathrm{d}\gamma \tag{4}$$

with

$$\tilde{p}_\beta(\gamma) := p_\beta(\gamma) \cdot D \cos\gamma, \quad c_f(\gamma) := \left(\frac{\gamma}{\sin\gamma}\right)^2 c(\gamma)$$

as well as

$$c(\gamma) := \int_{-\infty}^{\infty} |\rho|\, \mathrm{e}^{2\pi \mathrm{i}\, \rho\gamma}\, \mathrm{d}\rho = \left(\mathcal{F}^{-1} |\cdot|\right)(\gamma).$$

In the last equation, $\mathcal{F}^{-1}$ denotes the inverse Fourier transform. If one ignores the factor $\gamma/\sin\gamma$ in the filter c_f as a first approximation, and expresses the convolution (4) using Fourier transformation, one gets

$$\tilde{Q}_\beta(\gamma') \approx \left(\mathcal{F}_1^{-1}\, \tilde{G}(\cdot, \beta)\, |\cdot|\right)(\gamma') \equiv \int_{-\infty}^{\infty} \tilde{G}(\rho, \beta)\, |\rho|\, \mathrm{e}^{2\pi \mathrm{i}\, \rho\gamma'}\, \mathrm{d}\rho \tag{5}$$

with

$$\tilde{G}(\rho, \beta) := (\mathcal{F}_1\, \tilde{p}_\beta)(\rho) = \int_{-\gamma_m}^{\gamma_m} \tilde{p}_\beta(\gamma)\, \mathrm{e}^{-2\pi \mathrm{i}\, \rho\gamma}\, \mathrm{d}\gamma. \tag{6}$$

To summarize, the FBP algorithm consists of two steps: First, the filtering operation described by Eq. (4), which can be rewritten as Fourier transformations in Eqs. (5) and (6). Second, the actual back-projection described by Eq. (1) as an integral over β.

3 Implementation

This section contains the technical description of our implementations. In each subsection, we will explain how to map the FPB algorithm to the standard graphics pipeline or CDUA framework of modern GPUs, and to multi-core CPUs.

framework	DirectX	CUDA	CPU
algorithm	Alg. 1, 2	Alg. 3	
FFT/convolution	Cooley-Tukey	cuFFT	FFTW3
memory/cache	texture cache (2D local)	shared memory	shared memory
interpolation	linear	linear	linear

Table 1: Implemented FBP reconstruction techniques

Since single-precision (32-bit) floating-point format is sufficient for rendering purposes, consumer GPUs typically have limited support for double precision format (64-bit). Accordingly, we will limit our graphics pipeline implementation to single-precision as. On the other hand, double-precision arithmetic is abundant on CPUs with minimal speed-loss. Recent higher-end GPU cards such as Nvidia's Quadro, Tesla, and now Kepler architectures are designed with double precision arithmetic units for scientific computing.

Table 1 provides an overview of our implementations.

3.1 Graphics Pipeline (DirectX HLSL)

We will compare two slightly different versions of the implementation. First, Fig. 2 illustrates a direct approach [16] of how to map the back-projection operation in Eq. (1) to the graphics pipeline. The central idea is the following. We implement the β-rotation by rendering a rotated full-screen square rectangle for each discretized angle β. The filtered input texture in Fig. 2 has the following structure: each row corresponds to an angle β, and each column to a detector (fan-angle γ). Back-projection can be visualized as "smearing" the highlighted blue bar in the input texture over the reconstruction plane, which is indicated by the grayscale pixel lines. As an advantage, we have to setup the 4 rotated vertices of each rectangle only once in the initialization stage, and the complete back-projection operation can be performed by a single "draw" function call.

Algorithm 1: DirectX (HLSL) Back-Projection Pixel Shader, rotated squares (see Fig. 2)

```
Input: $x, y$: coordinates, $z$: projection index (from texture coordinate register); $\mathrm{tex}$: input
       texture. Constants: $D$: distance of the radiation source $S$ from the rotation center at the
       origin; $C$: texture coordinate factor; $A$: fan arc length; $N$: number of projections
1  $r \leftarrow 0$
2  $(x', y') \leftarrow (x, D - y)$
3  $\gamma' \leftarrow \arctan(x'/y')$                          // Eq. (3)
4  $t \leftarrow C \cdot (\gamma'/A + 0.5)$
5  $\mathrm{Lsq} \leftarrow x'^2 + y'^2$                        // Eq. (2)
6  $r \leftarrow r + \mathrm{tex.sampleAt}(t, z)/\mathrm{Lsq}$
7  $z \leftarrow z + 0.25$
8  $(x', y') \leftarrow (-y, D - x)$                          // rotate by $\pi/2$
9  $\langle$repeat lines 3 – 6$\rangle$
10 $z \leftarrow z + 0.25$
11 $(x', y') \leftarrow (-x, D + y)$                          // flip signs (rotate by $\pi$)
12 $\langle$repeat lines 3 – 6$\rangle$
13 $z \leftarrow z + 0.25$
14 $(x', y') \leftarrow (y, D + x)$                           // rotate by $3\pi/2$
15 $\langle$repeat lines 3 – 6$\rangle$
16 $r \leftarrow r/N$                                         // normalize
17 $r \leftarrow r + \mathrm{tex}_{\mathrm{swap}}.\mathrm{sampleAt}(\text{pixel loc})$   // accumulate
Output: $r$
```

Algorithm 1 shows the pixel shader pseudo-code of the back-projection rendering pass, implementing the algorithm shown in Fig. 2. In particular, the rotation by β is already performed. There are 3

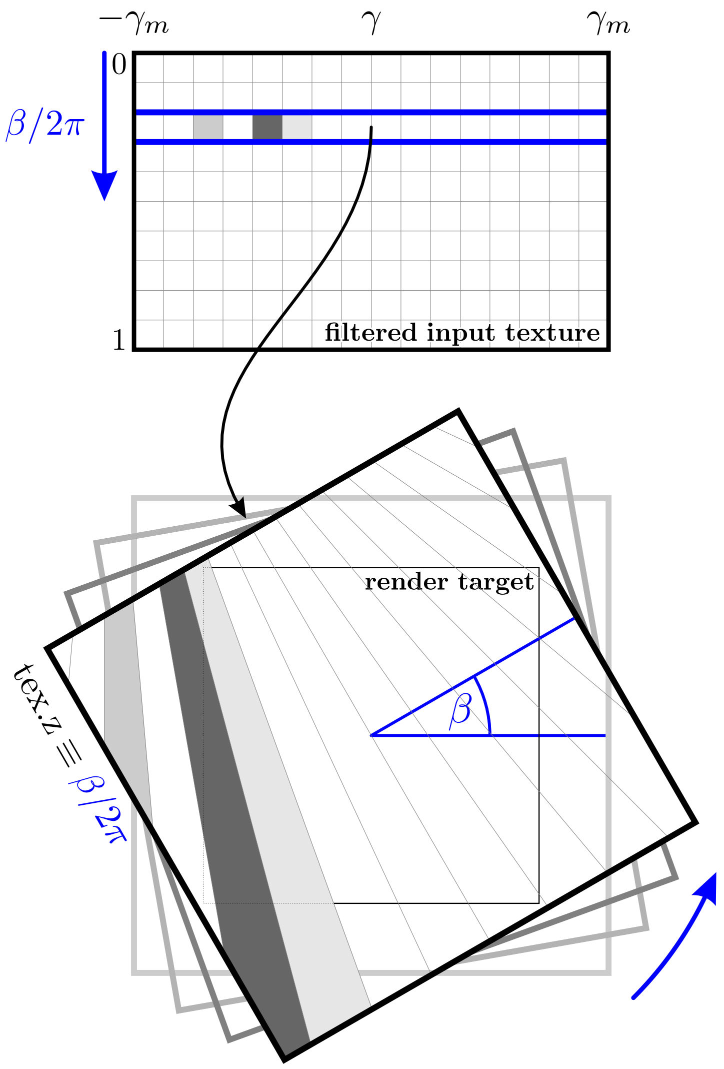


Figure 2: Illustration of the back-projection step by rotated squares. Each column of the filtered input texture corresponds to one detector on the fan (with γ between $-\gamma_m$ and γ_m), and each row to a rotation angle β (see Fig. 1). In a straightforward implementation, we can rotate the 4 vertices of each square rectangle by β in the setup stage, and sample the rows in the input texture via the $\mathrm{tex.z}$ texture coordinate (which has the same value for all four vertices within a square). Alternatively, the rotation can also be performed in the pixel shader.

texture coordinates x, y, z, with x and y the conventional ones, while $z \equiv \beta/2\pi$ encodes the rotation angle of the current rectangle. In order to accumulate all projections, one is tempted to sum the values at each pixel via the ADD blend-state. However, we have observed that this leads to a large performance decrease, presumably due to the interleaved read-write operations on the render target texture. Instead, in our implementation, we use two textures to accumulate all projections (see line 17), swapping them as output render target and additional input for each back-projection. Additionally, we exploit the right-angle symmetry noted e.g. in Ref. [17], leading to 4 similar blocks in algorithm 1, Thus, we require only $1/4$ of the rotated rectangles, and on average only $1/4$ texture reads of the swap texture per texture read of the filtered input texture.

Unfortunately, the DirectX HLSL language offers no explicit fine-tuned control of "coalescent" memory accesses, thus we have to rely on the intrinsic (2D local) texture caching by the graphics driver. Different from 3D reconstructions, the memory *bandwidth* is of no concern, but the memory access patterns matter due to cache latency. Note that in our case, the accesses to memory are rather challenging due to the jumps of the z variable in algorithm 1.

Nevertheless, profiling our implementation shows that performance is limited by the *computations* in the pixel shader with the expensive $\arctan$ evaluations. For fan-beam geometry, we see no possibility to avoid these calculations; lookup tables, e.g., did not increase the performance.

Algorithm 2: DirectX (HLSL) Back-Projection Pixel Shader, optimized version

```
Input: $x, y$: coordinates; tex: input texture. Constants: $D$: distance of the radiation source from
       the rotation center at the origin; $C$: texture coordinate factor; $A$: fan arc length; $N$:
       number of projections, $s_\beta := \sin(\Delta\beta)$, $c_\beta := \cos(\Delta\beta)$
1 $r \leftarrow 0$
2 for $i \leftarrow 0;\ i < N/4;\ i \leftarrow i + 1$ do
3     $z \leftarrow (i + 0.5)/N$                                  // add bias (half a texel)
4     $\langle$same code as lines 2 – 15 in algorithm 1$\rangle$
5     $\begin{pmatrix} x \\ y \end{pmatrix} \leftarrow \begin{pmatrix} c_\beta & s_\beta \\ -s_\beta & c_\beta \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}$      // rotate $(x, y)$ by $\Delta\beta$
6 end
Output: $r/N$
```

Algorithm 2 shows the pixel-shader code of our second implementation. We have replaced the rotated squares by a for-loop in the pixel shader. In this case, no swap texture is required for accumulating the projections since the summation is performed in the pixel shader. To save the calculation of $\sin(\beta)$ and $\cos(\beta)$ for each loop iteration, we pre-compute $\sin(\Delta\beta)$ and $\cos(\Delta\beta)$ for $\Delta\beta := 2\pi/N$ (where N is the number of projections), and then iteratively rotate (x, y) by $\Delta\beta$ in line 5.

To implement the filtering operation, the graphics pipeline has to carry out the forward and inverse Fourier transforms (5) and (6) for each row of the input texture (i.e., projection at angle β). We have manually implemented the well-known Cooley-Tukey FFT algorithm [2] for that purpose. The entries of the input vector correspond to discretized values of γ. In order to prevent spatial domain aliasing and speed up the FFT, we zero-pad the input vector to the 2^{nd}-next power of 2. For 736 detectors, this gives 2048 in our case.

3.2 GPU as Multi-Processor (CUDA)

The CUDA implementation of the FBP algorithm is more straightforward. Due to the additive nature of the integral (1), no atomic operations are required as each thread is responsible for one pixel location. The multi-GPU CUDA based algorithm requires a distribution of workload to each GPU. In our case, a simple division of workload is adequate and results in efficient processing. Other methods that rely on work-queues were found to be ineffective in speeding up running time, and for that reason we employ the simplest form instead. For efficient computation on the GPU one should ensure that the distribution of blocks and threads make use of all streaming multi-processors (SMs). A simple hardware query is usually adequate, and the distribution based upon these values sufficient to ensure the GPU is almost fully utilized.

Algorithm 3: CUDA Back-Projection Kernel

```
Input: Constants: $D$: distance between rotation center and radiation source; $i_{\text{iso}}$: defines
       isocenter channel location; $\beta$: projection start angle; $N$: number of projections; $\Delta\beta$:
       angle increment; $A$: angle between radial fan channels
$i_x \leftarrow \text{blockDim}.x \times \text{blockIdx}.x + \text{threadIdx}.x$
$i_y \leftarrow \text{blockDim}.y \times \text{blockIdx}.y + \text{threadIdx}.y$
$x \leftarrow \text{minXDimension} + i_x \times \text{pixelSize}$
$y \leftarrow \text{minYDimension} + i_y \times \text{pixelSize}$
$\theta \leftarrow \arctan(x/y)$
$r \leftarrow \sqrt{x^2 + y^2}$
$s \leftarrow 0$                                                // integral summation
while $i < N$ do
    $y' \leftarrow r \times \cos(\beta - \theta)$
    $x' \leftarrow r \times \sin(\theta - \beta)$
    $\gamma \leftarrow \arctan\left(\frac{x'}{D - y'}\right)$
    if $\gamma < \gamma_{\max}$ then
        $\text{Lsq} \leftarrow x'^2 + (D - y')^2$
        $i_{\text{chan}} \leftarrow i_{\text{iso}} + \gamma \times A$
        $s \leftarrow s + \text{tex.Lookup}(i_{\text{chan}})/\text{Lsq}$
    end
    $\beta \leftarrow \beta + \Delta\beta$
end
$\text{rec}[\text{imageSizeW} \times i_y + i_x] \leftarrow s$
```

We employ texture-based storage in CUDA to make use of the hardware-accelerated and cached linear interpolation. However, this feature is only available at single precision computations, and thus double-precision implementation requires (slower) standard global memory storage on the GPU.

For the filtering operations (5) and (6), we make use of the dedicated and highly optimized CuFFT libraries [12]. As for the graphics pipeline implementation, we zero-pad the frequency space to the 2^{nd}-next power of 2 (resulting in 2048 for 736 detector channels). In order to avoid any high frequency drop-off at the transition from detector channel 736 to (virtual) 737, we use a linear decrease of the values up to channel 800. Last, template-methods were used for easy comparison of numerical accuracy at a given level of precision.

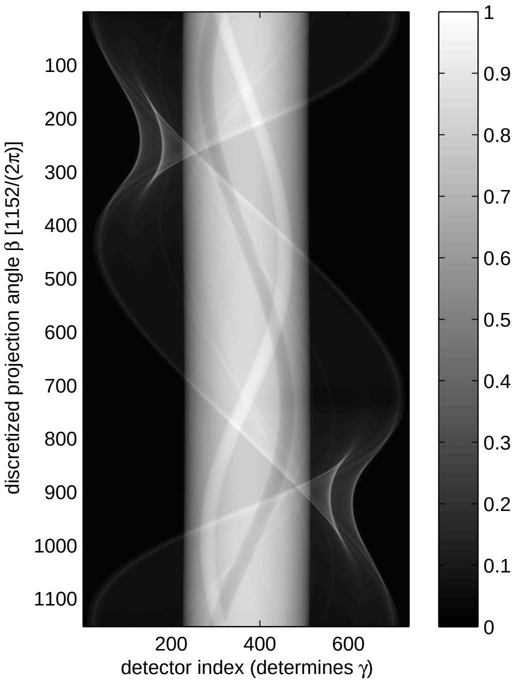
3.3 Multi-Core, Multi-Threaded CPUs

The data layout is an optimized STL (standard template library)-valarray structure. This data structure is specifically optimized for storing values instead of pointers or objects, enabling vector-based manipulations. Likewise, we use cache-friendly coalesced data reads for incremental updates.

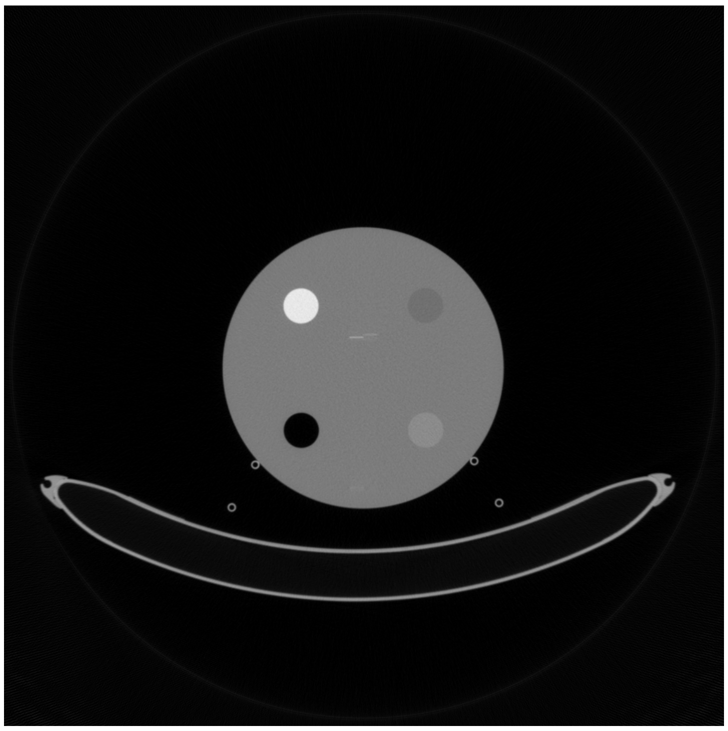
The multi-core CPU implementation is similar to the CUDA implementation, except that we apply the highly optimized FFTW3 library [3] for the forward and inverse Fourier transforms in the convolution step. As in the other implementations, we use a standard linear ramp filter in the frequency domain.

4 Results

We have tested our implementations of the FBP algorithm on various HPC environments. In all cases, the reconstructed image has dimension 1024×1024, which is appropriate for dense sensor arrays. Fig. 3 shows the measured scan data (input to the algorithm) on the left size and the reconstructed image on the right side.



(a) scan data



(b) reconstructed image, showing the ACR-accredited phantom

Figure 3: Reconstruction of a 1024×1024 image (b) from the scan data (a) via the FBP algorithm (DirectX implementation). The scanner has 736 individual detectors (x-axis in (a)) and scans the object from 1152 projection angles β (y-axis in (a)) which uniformly cover $360°$. For the filtering operation via FFT, we zero-pad each row vector of length 736 to 2048 entries.

4.1 Hardware Test Platform

Testing was conducted on two systems: a MacPro 5.1 with dual 6-core Intel Xeon X5660 Westmere 2.66 GHz 12MB L3 cache processors, 64 GB-ECC 1333Mhz memory, and dual Nvidia Quadro 4000 GPUs, and an SGI Altix 4700 32×dual-core Intel Itanium 9150M 1.66 GHz 24MB L3 cache processors, 256GB-ECC 1333Mhz shared-memory machine. Raw data was compiled from the scan of a ACR-accredited phantom [10] and scanning was conducted using a Siemens Definition Flash (128 slice) CT-scanner. We have installed Windows 7 (parallel to Mac OSX Lion) on the MacPro computer to run the DirectX implementation.

system	CPU 1-core SGI	CPU 64-core SGI	CPU 1-core MacPro	CPU 12-core MacPro	CUDA 1-GPU MacPro	CUDA 2-GPU MacPro	CUDA 1-GPU MacPro *fastmath*	CUDA 2-GPU MacPro *fastmath*	DirectX 1-GPU MacPro
single-precision (32-bit) floating-point									
time [s]	290.96	4.62	89.94	6.16	0.98	0.52	0.38	0.28	0.19
speedup	$1\times$	$63\times$	$3.24\times$	$47\times$	$297\times$	$560\times$	$766\times$	$1039\times$	$1531\times$
double-precision (64-bit) floating-point									
time [s]	291.32	4.67	90.15	6.18	1.78	1.01	–	–	–
speedup	$1\times$	$62\times$	$3.23\times$	$47\times$	$164\times$	$288\times$	–	–	–

Table 2: Runtime comparisons of our FBP reconstruction implementations (736 detectors, 1152 projection angles, reconstructed image 1024×1024)

4.2 Runtime Analysis

Table 4.2 summarizes the running times of our FBP implementations (optimized algorithm 2 for DirectX), both for single- and double-precision when available. To measure the actual computation time and exclude initial setup steps like loading a texture from disk, we have sequentially executed the algorithm 100 times and then taken the average. The runtime details for the CPU and GPU implementations are discussed in the following two subsections.

4.2.1 CPU Implementations

The typical modality for high-performance computation is a multi-threaded application running on a shared memory environment. We tested our FBP implementations starting from a single-threaded serial execution to 128 threads on two shared-memory systems, and obtained substantial speedups, as shown in Fig. 4. Specifically, on the 64-core SGI system we achieved a speedup of $63\times$ (see Table 4.2), which is almost linear. On the 12-core MacPro5.1 system, we observe a speedup of $14.6\times$ as compared to the serial, single-threaded execution of the algorithm on this machine. This resulted in 4.62 s and 6.16 s reconstruction times, respectively (see Table 4.2). The greater-than linear speedup in the number of cores on the MacPro system is likely due to the OSX-Lion thread scheduling policy and the Hyper-Threading ability of the Intel Westmere processors.

For double-precision floating-point reconstruction on the CPUs, performance hardly decreased at all when compared to single-precision (see Table 4.2).

In summary, our implementation indeed scales well beyond 2 cores. As expected, the running times on the CPUs are indifferent to floating-point precision, which indicates that all arithmetic units inherently support double precision without performance loss.

4.2.2 GPU Implementations

Our three GPU implementations all outperform the conventional multi-core CPU environments in terms of speed (see Table 4.2). Specifically, the single-GPU DirectX implementation (HLSL) using algorithm 2 was able to consistently achieve an average speedup of $1531\times$ as compared to single-threaded serial execution. The CUDA implementation (fastmath) achieved a speedup of $766\times$ for one GPU and $1039\times$ for two GPUs using single-precision cached textures, which store the projection data. Remarkably, the GPU graphics pipeline (DirectX) outperforms the multi-processor CUDA

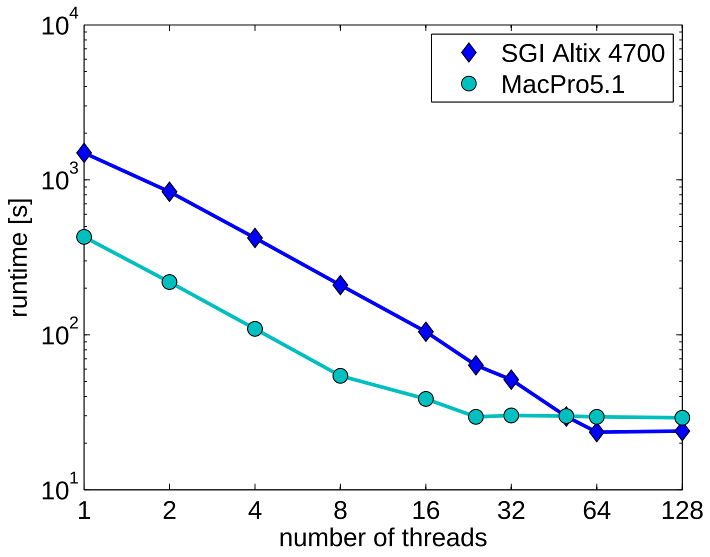


Figure 4: Runtime comparison of the multi-core CPU implementations, dependent on the number of threads. Note the logarithmic scale of both axes and the linear performance increase, as well as the saturation once the system reaches full utilization.

implementation by a factor of 2. This could be explained by the computational overhead of a more general architecture like CUDA, the slight differences between Alg. 2 and Alg. 3, or differing qualities of the graphics drivers.

As mentioned in Section 3.2, CUDA double-precision requires conventional global-memory on the GPU. We observe that our double-precision implementations are consistently slower (see Table 4.2), only obtaining a $164\times$ speedup for one-GPU and $288\times$ speedup for two-GPUs. This effect can be explained by the smaller number of internal double-precision cores of the Quadro 4000 GPU. Namely, the specification sheet lists twice the theoretical peak performance ($486.4\,\mathrm{GFlops}$) for single-precision as compared to double precision ($243.2\,\mathrm{GFlops}$).

4.3 Comparison with Other Studies

The paper by Zeng et.al. [17] closely resembles our study in terms of the algorithm. The authors provide CPU implementations for 1160 projections with 672 detector channels and reconstruct 512×512 images. They report runtimes of $52\,\mathrm{s}$ (conventional, non-optimized) and $1.35\,\mathrm{s}$ (optimized, 2 threads) on two Intel Xeon $3.2\,\mathrm{GHz}$ processors. Extrapolating to 1024×1024, this gives $5.4\,\mathrm{s}$ for the optimized version, and linearly scaling to 12 cores with $2.66\,\mathrm{GHz}$ and our geometry results in approximately $1.2\,\mathrm{s}$. For comparison, our runtime on the Intel Xeon X5650 $2.66\,\mathrm{GHz}$ processors is $6.18\,\mathrm{s}$ (see Table 4.2). This discrepancy likely stems from SIMD techniques in their implementation, the use of highly optimized Intel compilers and cache-friendly reordering of input data.

We have implemented the geometric symmetry enhancements in Zeng et.al. on CPUs and indeed noticed a substantial performance increase. Importantly, data re-ordering is essential to ensure good cache performance. (Note that this reordering is only possible after the scan of a slice is complete.) Thus our single-threaded CPU implementations should be regarded as *baseline* (similar to the "conventional" method in Zeng et.al.). We remark that the runtime of their conventional method is $206\,\mathrm{s}$ when scaled to our problem size and processor speeds.

Finally, our CUDA and DirectX implementations are still faster by a factor of 3 and 6, respectively, compared to $1.2\,\mathrm{s}$ runtime of the optimized version in [17] (after re-scaling to our problem size and 12 CPU cores).

4.4 Error Analysis

Since we are using data from a real CT scanner, we have to distinguish between errors/noise introduced by the scanner itself and computational errors in the FBP implementation. We will first estimate the computational errors by comparing the results with our double-precision CPU implementation as reference, and then evaluate the overall noise induced by the complete reconstruction procedure (scanner and computation).

4.4.1 Computational Reconstruction Errors

Since all our implementations are based on the same FBP algorithm, any deviation should be due to

- the use of single versus double floating-point arithmetics;
- the use of GPU "fast-math" modes which do not adhere to the IEEE-754 standard;
- deviations in the implementations of basic mathematical functions (e.g., the sine function) in the C standard library (CPU) versus pixel shader libraries (GPU);
- deviations of texture addressing on a single texel level (e.g., the graphics card driver might implicitly add a small bias to the texture coordinates, which changes the precise pixel-by-pixel alignment of the reconstructed images).

4.4.2 Overall Noise Estimation

Estimating noise resulting from the scanning and reconstruction procedure is an essential aspect of validating and interpreting the scan data. In our case, we make use of the fact that the scanned phantom has several homogeneous portions, which should ideally result in corresponding homogeneous areas in the reconstructed image. Assuming a perfectly homogeneous phantom, one can directly quantify the noise of the scanning procedure by the small inhomogeneities in the reconstructed image. Fig. 5 illustrates our approach in detail. We define nine "regions of interest" (ROIs) of the reconstructed image. The first four correspond to cylindrical rods inside the ACR-accredited phantom, which have known uniform materials (such as water, acrylic, or polyethylene). The attenuation measured by the CT scanner reflects the physical properties of the materials and can be used for calibration.

In our case, the error is the deviation from the mean value, that is, the variance

$$\frac{1}{N_x N_y}\,\|\Delta f\|_2^2, \quad \Delta f := f - \langle f \rangle$$

inside the ROIs, where $N_x \times N_y$ is the dimension (in pixels) of the corresponding rectangle. Table 3 lists these variances for all implementations. Since the values hardly differ, we speculate that the noise mostly stems from the scanner itself.

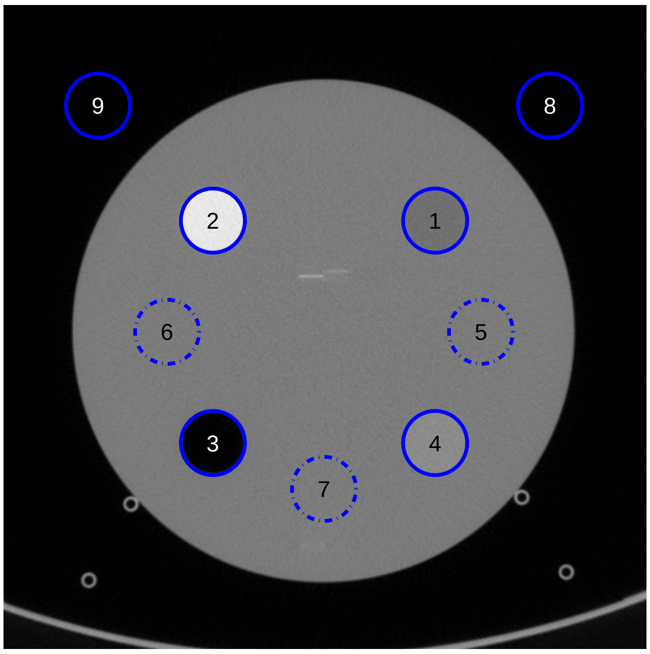


Figure 5: Noise estimation via nine homogeneous “regions of interest” (ROIs) of the reconstructed image (blue circles). The regions $(1, 2, 3, 4)$ enclose the four cylindrical rods inside the phantom. (Their main purpose is assessing the CT number of different materials.) The three dashed regions $(5, 6, 7)$ represent the internal uniform portion. Finally, the two outer homogeneous regions $(8, 9)$ represent air. We analyze a $32 \times 32\,\text{pixel}^2$ sample from the center of each ROI (and thus avoid the boundary of the circle).

In order to investigate the fluctuations in more detail, we employ the *Noise Power Spectrum* NPS (spectral density of the noise), as follows (also compare with Ref. [9]). Recall the definition of the discrete Fourier transform in 2 dimensions:

$$(\mathcal{F}_2\,\Delta f)\,(k,\ell) := \sum_{x,y} \Delta f(x,y)\,\mathrm{e}^{-2\pi \mathrm{i}(k\,x/N_x + \ell\,y/N_y)},$$

where x runs from 0 to $N_x - 1$ and y from 0 to $N_y - 1$. We define the noise power spectrum of f as

$$\mathrm{NPS}_f(k,\ell) := \frac{1}{N_x N_y}\left|(\mathcal{F}_2\,\Delta f)\,(k,\ell)\right|^2, \tag{7}$$

which depends on the frequency values k and ℓ. Note that $\sum_{k,\ell} \mathrm{NPS}_f(k,\ell) = \|\Delta f\|_2^2$ due to Plancherel’s theorem.

Fig. 6 illustrates the NPS of ROI 1 from Fig. 5. Interestingly, the noise is contained in characteristic frequencies up to $\nu_{\max} \approx 5/32\,\text{pixel}^{-1}$. Defining the coordinate system such that Fig. 3b extends from -0.5 to 0.5 in both x- and y-direction, the smallest period of the noise in these coordinates equals

$$\frac{1}{\nu_{\max}} \approx \frac{0.5}{1024\,\text{pixel}}\,\frac{32}{5}\,\text{pixel} = \frac{1}{320} = 0.0031\ldots$$

For comparison, half of this period is close to the distance 0.0014 between two neighboring fan rays $p_\beta(\gamma)$ at the origin (see Fig. 1). Thus we speculate that the NPS can be explained by the detector discretization error, but we would need to test on another system with different sensor geometry to verify our hypothesis.

precision	CPU single	CPU double	CUDA single	CUDA double	CUDA single *fastmath*	DirectX single
ROI 1	8.8	8.8	8.7	8.8	8.7	10.7
2	3.9	4.0	4.1	3.9	3.9	5.0
3	5.9	5.9	5.9	5.9	5.9	7.5
4	8.6	8.6	8.6	8.6	8.6	9.8
5	6.9	6.9	6.9	6.9	6.9	9.3
6	7.8	7.8	7.8	7.8	7.8	8.0
7	7.3	7.3	7.1	7.3	7.3	7.3
8	4.2	4.2	4.1	4.2	4.2	3.7
9	4.2	4.2	4.0	4.2	4.2	5.6

Table 3: Estimated noise (variance of f restricted to a $32 \times 32\,\mathrm{pixel}^2$ rectangle) of all ROIs, in units $[10^{-5}]$.

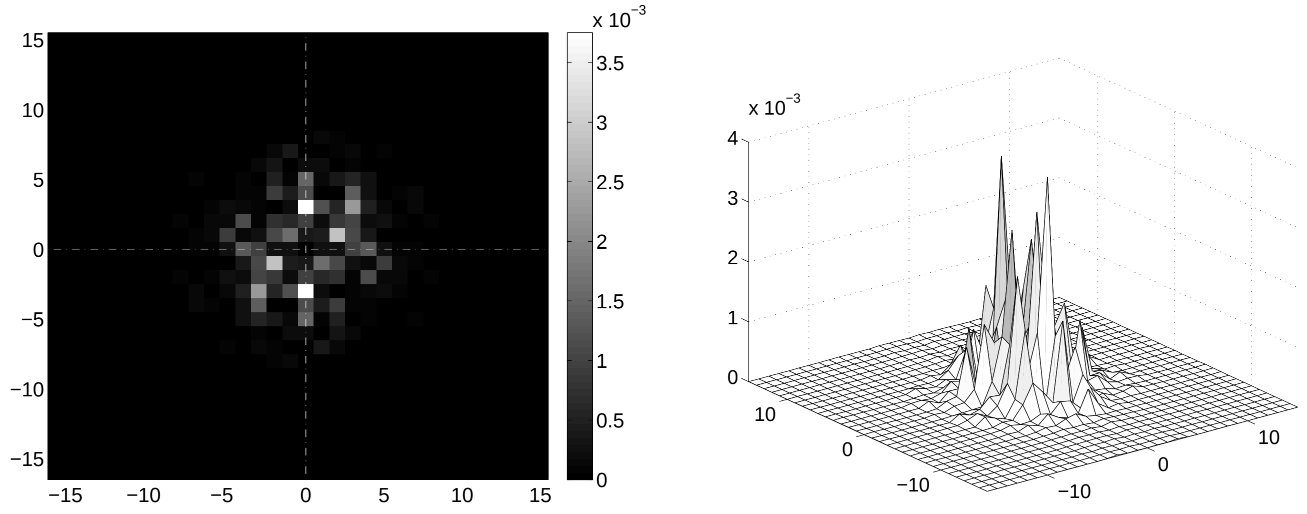


Figure 6: Noise power spectrum of the deviations from the mean value (Eq. (7)) for ROI 1, seen in Fig. 5, both as 2D and 3D plot. We observe that characteristic frequencies up to $\nu_{\max} \approx 5/32\,\mathrm{pixel}^{-1}$ dominate the noise.

5 Conclusion

Our analysis shows that single-precision floating point is sufficient for the reconstruction task. This is of particular interest as consumer-level GPUs are mainly single-precision floating point architectures. While the error analysis reveals small differences in accuracy between our implementations (as expected from using different floating-point precisions), these differences are hardly or not at all noticeable in the reconstructed image. Namely, they are smaller than the typical $8-10$-bit grayscale available on most clinical medical imaging displays.

Thus, the choice of software and hardware framework mostly depends on reconstruction runtime and hardware cost, which clearly favors the GPU implementations (see Table 4.2). Interestingly, we have achieved the best performance by making use of the GPU graphics pipeline (DirectX/OpenGL). Nevertheless, employing the GPU as general-purpose multiprocessor (CUDA) is more straightforward from a programming perspective and practicable for a larger set of algorithms.

Acknowledgements

CM is supported by the Technische Universität München, Germany, SE by the Natural Sciences and Engineering Research Council of Canada, Servier Laboratories Canada, and the Department of Radiology and Diagnostic Imaging University of Alberta. MN holds a research grant from Servier Laboratories. PB holds research grants from the Natural Sciences and Engineering Research Council of Canada and Servier Laboratories Canada.

References

[1] D. Bharkhada, H. Yu, K. Zeng, E. Bai, and G. Wang. A Comparative Study on Interpolation Methods for Controlled Cardiac CT. *International Journal of Imaging Systems and Technology*, 17(2):91–98, 2007.

[2] James W. Cooley and John W. Tukey. An Algorithm for the Machine Calculation of Complex Fourier Series. *Math. Comput.*, 19:297–301, 1965.

[3] Matteo Frigo and Steven G. Johnson. The Design and Implementation of FFTW3. *Proc. IEEE*, 93:216–231, 2005.

[4] R. C. Gonzalez and R. E. Woords. *Digital Image Processing 3rd Edition*. Pearson Prentice Hal, 2010.

[5] C.L. Luengo Hendriks, M. Van Ginkel, P.W. Verbeek, and L.J. Van Vliet. The generalized Radon transform: sampling, accuracy and memory considerations. *Pattern Recogn.*, 38:2494–2505, 2005.

[6] G. T. Herman and H. Lung. Reconstruction from divergent beams: A comparison of algorithms with and without rebinning. *Computers in Biology and Medicine*, 10(3):131–139, 1980.

[7] Hannes G. Hofmann, Benjamin Keck, Christopher Rohkohl, and Joachim Hornegger. Comparing performance of many-core CPUs and GPUs for static and motion compensated reconstruction of C-arm CT data. *Med. Phys.*, 38:468–473, 2011.

[8] Avinash C. Kak and Malcolm Slaney. *Principles of Computerized Tomographic Imaging*. IEEE Press, 1988.

[9] Marie Foley Kijewski and Philip F. Judy. The noise power spectrum of CT images. *Phys. Med. Biol.*, 32:565–575, 1987.

[10] Cynthia H. McCollough, Michael R. Bruesewitz, Michael F. McNitt-Gray, Krista Bush, Thomas Ruckdeschel, J. Thomas Payne, James A. Brink, and Robert K. Zeman. The phantom portion of the American College of Radiology (ACR) Computed Tomography (CT) accreditation program: Practical tips, artifact examples, and pitfalls to avoid. *Med. Phys.*, 31:2423–2442, 2004.

[11] Paul M. Novotny, Jeff A. Stoll, Nikolay V. Vasilyev, Pedro J. del Nido, Pierre E. Dupont, Todd E. Zickler, and Robert D. Howe. GPU based real-time instrument tracking with three-dimensional ultrasound. *Med. Image Anal.*, 11(5):458–464, 2007.

[12] NVIDIA. CuFFT Library, May 2011.

[13] Y. Okitsu, F. Ino, and K. Hagihara. High-performance cone beam reconstruction using CUDA compatible GPUs. *Parallel Comput.*, 36:129–141, 2010.

[14] Eric Papenhausen, Ziyi Zheng, and Klaus Mueller. GPU-Accelerated Back-Projection Revisited: Squeezing Performance by Careful Tuning. *Workshop on High Performance Image Reconstruction*, 1:1–4, 2011.

[15] J. L. Prince and J. M. Link. *Medical Imaging Signals and Systems*. Pearson Prentice Hal, 2007.

[16] Fang Xu and Klaus Mueller. Real-Time 3D Computed Tomographic Reconstruction Using Commodity Graphics Hardware. *Phys. Med. Biol.*, 52:3405–3419, 2007.

[17] Kai Zeng, Erwei Bai, and Ge Wang. A fast CT reconstruction scheme for a general multi-core PC. *International Journal of Biomedical Imaging*, 2007:29160, 2007.