MIC-GPU: High-Performance Computing for Medical Imaging on Programmable Graphics Hardware (GPUs)

SPIE Medical Imaging

GPU-Accelerated CT Reconstruction

Klaus Mueller, Wei Xu, Ziyi Zheng **Fang Xu**

Computer Science
Center for Visual Computing
Stony Brook University, NY

Siemens USA Research
Princeton, NJ

Overview

What to expect:

- details on the parallelization of various fundamental CT reconstruction algorithms
- details and fragment code for graphics-style GPU implementations of these
- insights on GPGPU-style implementations of these
- comparisons of results obtained with both

Decomposition

$$P:\ p_i = \sum_{j=0}^{N^3-1} \left(v_j \cdot w_{ij} \right) \qquad B:\ v_j = \sum_{i=0}^{M_\varphi - 1} \left(p_i \cdot w_{ij} \right)$$

FBP

$$v_j = \sum_{p_i \in P_{set}} p_i w_{ij\_fdk} = \sum_{p_i \in P_{set}} B \cdot S$$

S: scanner projections

I: identity projection/volume

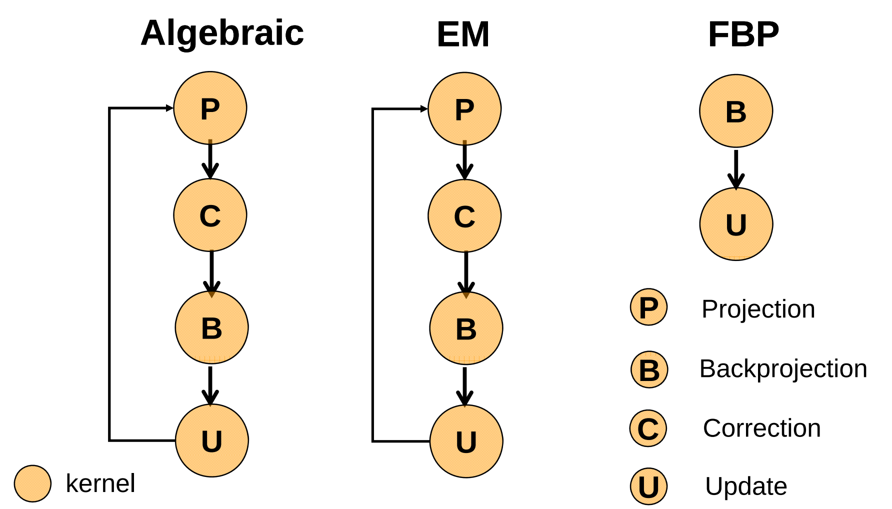
Algebraic

$$v_j = v_j + \frac{\displaystyle\sum_{p_i \in P\varphi} \left(\frac{\lambda \left(p_i - \sum_{l=0}^{N^3-1} v_l \cdot w_{il} \right)}{\sum_{l=0}^{N^3-1} w_{il}} \right) w_{ij}}{\displaystyle\sum_{p_i \in P_\varphi} w_{ij}} = v_j + \frac{B(\lambda \frac{S - P(V)}{P(I)})}{B(I)}$$

OS-EM

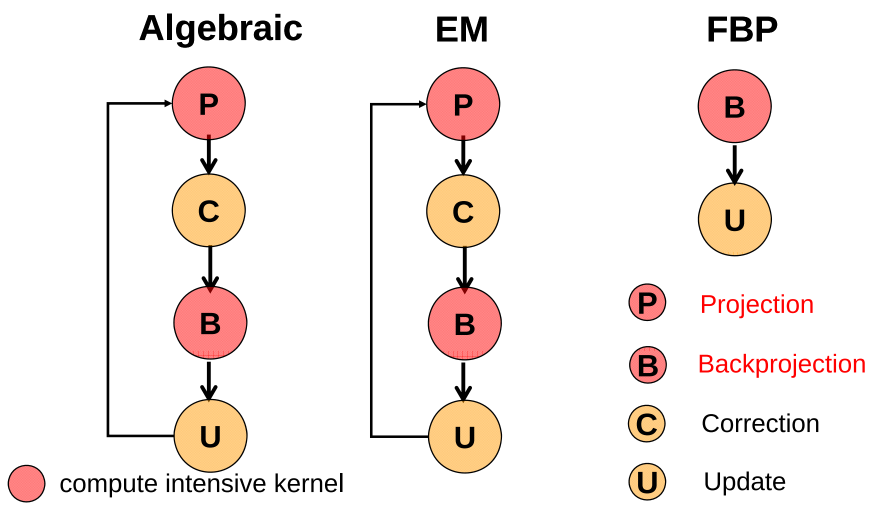
$$v_j = \frac{v_j}{\sum_{p_i \in P_{set}} w_{ij}} \left(\sum_{p_i \in P_{set}} \left(\frac{p_i}{\sum_{l=0}^{N^3-1} v_l \cdot w_{il}} \right) w_{ij} \right) = \frac{v_j}{\sum_{p_i \in P_{set}} B(I)} \left(\sum_{p_i \in P_{set}} B \frac{S}{P(V)} \right)$$

Kernel-Centric Reconstruction



Kernel-Centric Reconstruction

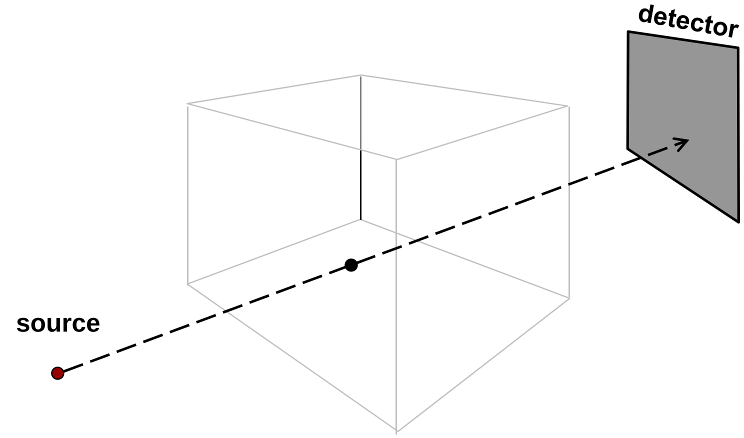
SPIE Medical Imaging



Backprojection

SPIE Medical Imaging

Sample in projection space, voxel-driven



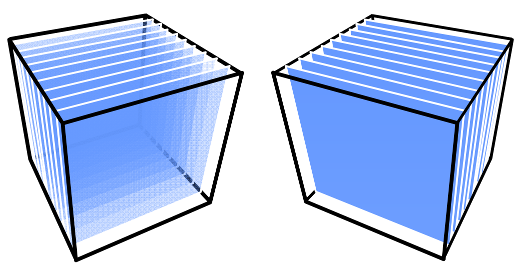
Volume Representation

SPIE Medical Imaging

3D texture is only supported on latest hardware

2D texture stacks are used

Axis aligned (x, y, z), easy to compute and store



Transformation Matrix

SPIE Medical Imaging

A 4x4 matrix *M* transforms 3D voxel coordinates to 2D pixel coordinates on the detector

Perform perspective divide if necessary (cone-beam)

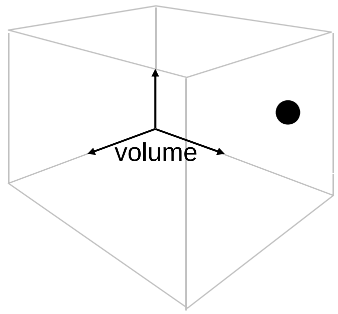
Composition of the matrix from graphics point of view

- model-view matrix
- projection matrix
- translation / scaling matrix

$$\begin{bmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \\ a_{30} & a_{31} & a_{32} & a_{33} \end{bmatrix} \begin{bmatrix} x_v \\ y_v \\ z_v \\ 1 \end{bmatrix} = \begin{bmatrix} x_h \\ y_h \\ z_h \\ w_h \end{bmatrix} \qquad P_\varphi(U,V) = \left(\frac{x_h}{w_h}, \frac{y_h}{w_h}\right)$$

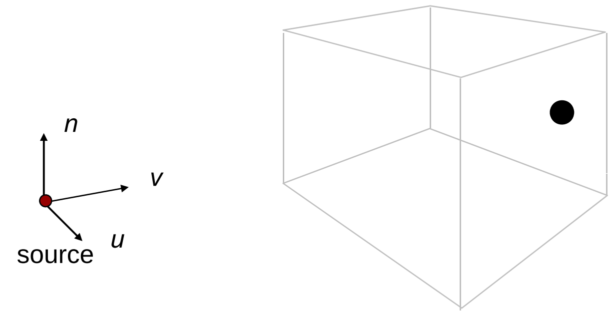
Decomposition

$$\begin{bmatrix} x_v \\ y_v \\ z_v \\ 1 \end{bmatrix}$$



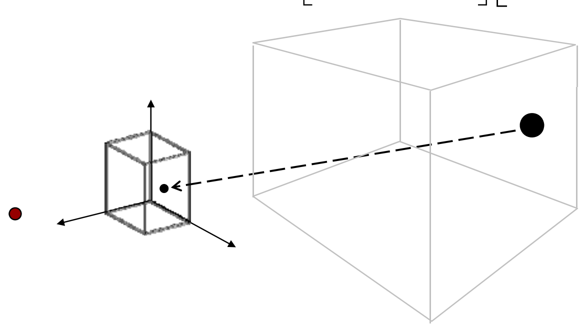
Decomposition: Model/View

$$\begin{bmatrix} u_x & u_y & u_z & -\vec{u}\cdot\vec{s} \\ v_x & v_y & v_z & -\vec{v}\cdot\vec{s} \\ n_x & n_y & n_z & -\vec{n}\cdot\vec{s} \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_v \\ y_v \\ z_v \\ 1 \end{bmatrix}$$



Decomposition: Projection

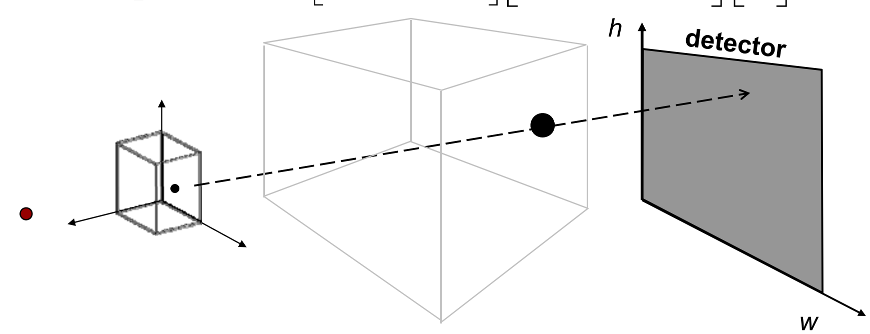
$$\begin{bmatrix} \frac{2n}{w} & 0 & 0 & 0 \\ 0 & \frac{2n}{h} & 0 & 0 \\ 0 & 0 & \frac{f+n}{n-f} & \frac{2fn}{n-f} \\ 0 & 0 & -1 & 0 \end{bmatrix} \begin{bmatrix} u_x & u_y & u_z & -\vec{u}\cdot\vec{s} \\ v_x & v_y & v_z & -\vec{v}\cdot\vec{s} \\ n_x & n_y & n_z & -\vec{n}\cdot\vec{s} \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_v \\ y_v \\ z_v \\ 1 \end{bmatrix}$$



n,*f*: near/far viewing frustrum extent

Decomposition: Window

$$\begin{bmatrix} \frac{w}{2} & 0 & 0 & 0 \\ 0 & \frac{h}{2} & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 1.0 \\ 0 & 1 & 0 & 1.0 \\ 0 & 0 & 1 & 1.0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \frac{2n}{w} & 0 & 0 & 0 \\ 0 & \frac{2n}{h} & 0 & 0 \\ 0 & 0 & \frac{f+n}{n-f} & \frac{2fn}{n-f} \\ 0 & 0 & -1 & 0 \end{bmatrix} \begin{bmatrix} u_x & u_y & u_z & -\vec{u}\cdot\vec{s} \\ v_x & v_y & v_z & -\vec{v}\cdot\vec{s} \\ n_x & n_y & n_z & -\vec{n}\cdot\vec{s} \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_v \\ y_v \\ z_v \\ 1 \end{bmatrix}$$



Full Transformation Matrix

Embedded transformation on graphics hardware (OpenGL)

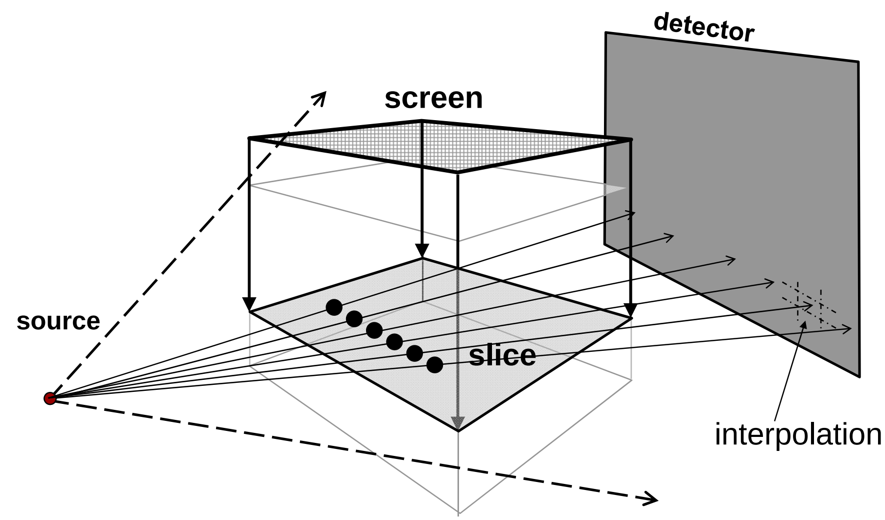
4th coordinate (w) contains voxel depth value

$$\begin{bmatrix} \frac{w}{2} & 0 & 0 & 0 \\ 0 & \frac{h}{2} & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 1.0 \\ 0 & 1 & 0 & 1.0 \\ 0 & 0 & 1 & 1.0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \frac{2n}{w} & 0 & 0 & 0 \\ 0 & \frac{2n}{h} & 0 & 0 \\ 0 & 0 & \frac{f+n}{n-f} & \frac{2fn}{n-f} \\ 0 & 0 & -1 & 0 \end{bmatrix} \begin{bmatrix} u_x & u_y & u_z & -\vec{u}\cdot\vec{s} \\ v_x & v_y & v_z & -\vec{v}\cdot\vec{s} \\ n_x & n_y & n_z & -\vec{n}\cdot\vec{s} \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_v \\ y_v \\ z_v \\ 1 \end{bmatrix} = \begin{bmatrix} x_h \\ y_h \\ z_h \\ w_h \end{bmatrix}$$

$$w_h = n_x \cdot x_v + n_y \cdot y_v + n_z \cdot z_v - \vec{n}\cdot\vec{s} = \vec{n}\cdot\vec{v} - \vec{n}\cdot\vec{s}$$

$$|w_h| \Rightarrow \text{voxel depth}$$

Implementation



Implementation: GPGPU

Acquire transformation matrix *M*

Pass into fragment shader

Orthographically render the volume slice to be reconstructed

In the fragment shader, compute detector-plane coordinates:

- generate homogenous location coordinates for each fragment
- multiply by *M*

Then, also in the fragment shader:

- perform perspective-divide (*x*, *y*)
- extract *w* to compute depth weights
- interpolate densities (bilinear interpolation)

Fragment Code Sample

```
float  fVoxel = 0;
float4 vWorldPos = float4(wpos.x, y, wpos.y, 1); // voxel coordinates
for (int s = 0; s < iProjNo; s++){ // loop over projections
   // transform to detector positions, using 1st and 2nd matrix rows
   float2 vDetPos = float2(dot(vMatR0, vWorldPos), dot(vMatR1,vWorldPos));
   // compute depth values, using 3rd matrix row
   float  fDepth = dot(vMatR3, vWorldPos);
   vDetPos /= fDepth;                  // perspective divide
   fWeight = pow(fSO/fDepth, 2);    // compute depth weights
   // sample+accumulate
   fVoxel += fWeight* tex3D(texProj3D, float3(vDetPos.x, vDetPos.y, idx+0.5);
}
```

Implementation: Graphics

Vertex and fragment shader

Transformation in the vertex shader, much faster!

- acquire transformation matrix *M*
- pass into the vertex shader
- orthographic rendering of the slice to be reconstructed
- pass 3D coordinates of the vertices into the vertex shader
- in the vertex shader
  - multiply proxy polygon vertices coordinates with *M*

Fragments get rasterized

Only perform texture sampling in the fragment shader

- perspective-divide on each fragments, compute depth-weight
- interpolate densities (bilinear interpolation)

Code Sample

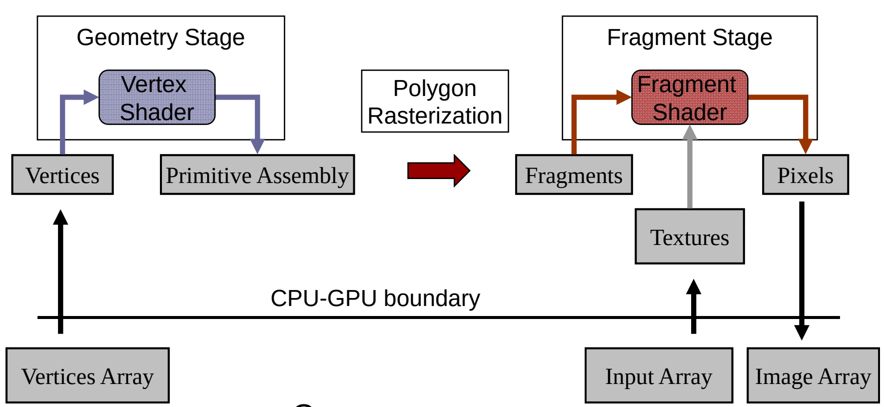
Fragment code:

```
float  fDepthRCP = 1/vTransPos.w;              // reciprocal of voxel depth
float2 vDetPos = vTransPos.xy*fDepthRCP;       // perspective division
float  fWeight = pow(fSO*fDepthRCP, 2);        // compute voxel weights
float  fOldVal = texRECT(texSlice, wpos);      // sample original value
float  fNewVal = texRECT(texProj, vDetPos);    // sample incoming value
float  fOutVal += fOldVal + fNewVal*fWeight;   // accumulate
return fOutVal;
```

GL code in host program:

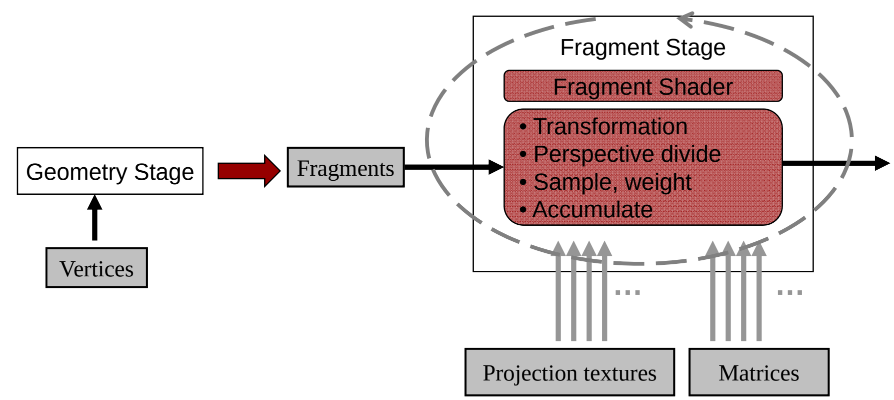
```
glBegin(GL_POLYGON);
    glTexCoord3f(v00.x, v00.y, v00.z); glVertex3f(v00.x, v00.y, v00.z);
    glTexCoord3f(v10.x, v10.y, v10.z); glVertex3f(v10.x, v10.y, v10.z);
    glTexCoord3f(v11.x, v11.y, v11.z); glVertex3f(v11.x, v11.y, v11.z);
    glTexCoord3f(v01.x, v01.y, v01.z); glVertex3f(v01.x, v01.y, v01.z);
glEnd();
```


Graphics Pipeline Revisited



One pass

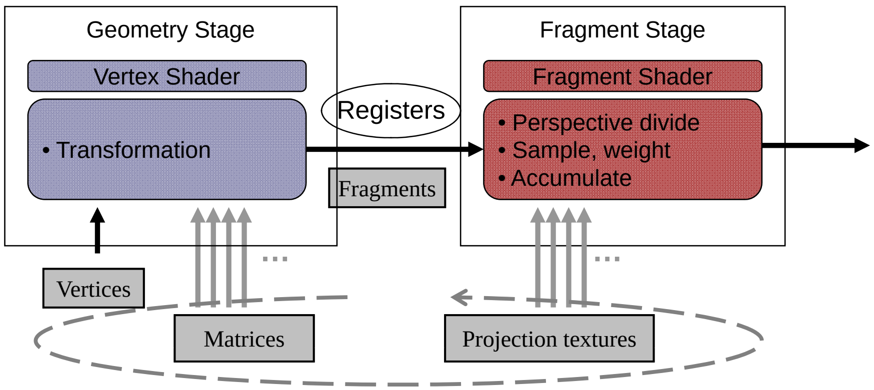
Pipeline 1: GPU as a Programmable Multi-Processor (MP-GPU)



Fragments contain the (x,y,z) voxel coordinates

Pipeline 2: GPU as a Programmable Graphics Processor (AG-GPU)

SPIE Medical Imaging



Fragments contain the (u,v) detector space coordinates

Graphics Pipeline Benefits

SPIE Medical Imaging

Graphics-aware pipeline (AG-GPU) is considerably faster (~3×) than MP-GPU

- graphics facilities are hardwired!

There are further features that have their origins in graphics and come with GPUs:

- early fragment kill → eliminate fragments based on some condition before they even enter the fragment processor (AG-GPU+)
- hardwired 32-bit floating-point precision linear interpolations, matrix and vector arithmetic (+, -, *), frame-buffer blending and compositing
- RGBA parallelism

see Xu/Mueller, *Physics Medicine & Biology*, vol. 52, pp. 3405–3419, 2007

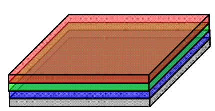
RGBA Parallelism

SPIE Medical Imaging

Exploit geometric mapping parallelism

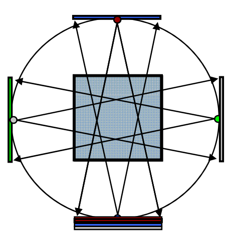
Volume packing

- adjacent 4 volume slices → RGBA



Projection packing

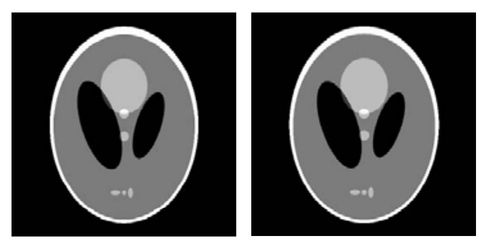
- symmetry in projection layout
- requires all projections beforehand



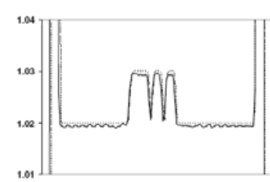
Example: Feldkamp Cone-Beam Reconstruction

SPIE Medical Imaging

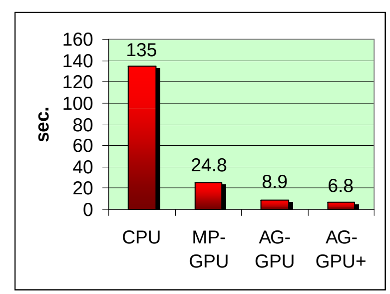
360 projections (1024^2, general position), 512^3 volume



CPU GPU



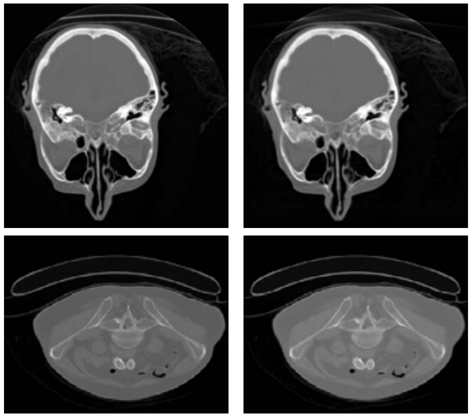
tumor profiles



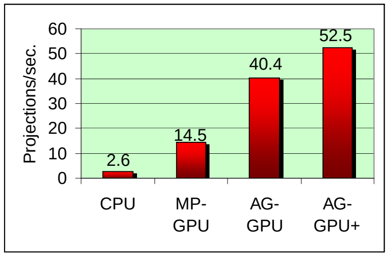
performance

Expressed in Projections/Sec.

360 projections, 512^3 volume



Original GPU-recon



performance in projections/s

GPU Enables Visual CT

High reconstruction frame rate enables injection of occasional volume rendering step

Also enables D^2VR: real-time volume visualization directly from projection data

see Xu/Mueller, *Proc. Volume Graphics Workshop*, pp. 23-30, 2006

Next: Iterative Algorithms

SART, EM

All require a projection simulation step

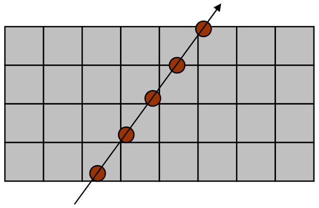
- should be as accurate as possible

Projection

Sampling in volume space, ray-driven

Raycasting methods [Krueger'03]

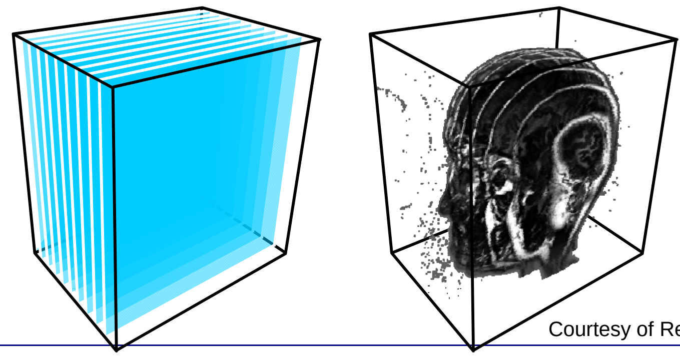
- represent volume as a single 3D texture
- rasterize the detector image, generate fragments
- cast rays, sample and accumulate
- no support for easy write



Projection

Slice-based method [Rezk-Salama'00]

- represent volume as a stack of 2D slices
- project each slice onto detector plane using texture mapping
- composite buffer to accumulate



Courtesy of Rezk-Salama

Projection: Ping-Pong Buffer

Buffer = Buffer + Incoming

Avoid simultaneous read and write

Bounce back and forth between two buffers in each pass

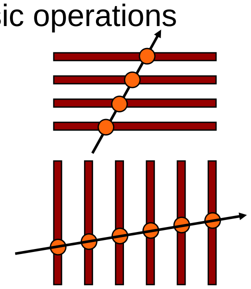
```
const GLenum gltex[2] = {GL_COLOR_ATTACHMENT0_EXT, GL_COLOR_ATTACHMENT1_EXT};
int SOURCE_BUFFER = 0;
#define DESTINATION_BUFFER !SOURCE_BUFFER
#define SWAP() SOURCE_BUFFER = DESTINATION_BUFFER
glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT, gltex[SOURCE_BUFFER],
   GL_TEXTURE_RECTANGLE_NV, tex[SOURCE_BUFFER], 0);
glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT, gltex[DESTINATION_BUFFER],
   GL_TEXTURE_RECTANGLE_NV, tex[DESTINATION_BUFFER], 0);
for (…){
   glDrawBuffer(gltex[DESTINATION_BUFFER);
   glBindTexture(GL_TEXTURE_RECTANGLE_NV, tex[SOURCE_BUFFER]);
   …
   SWAP();
}
```


Projection

3D transformation & texture mapping are intrinsic operations

Volume slice stacks

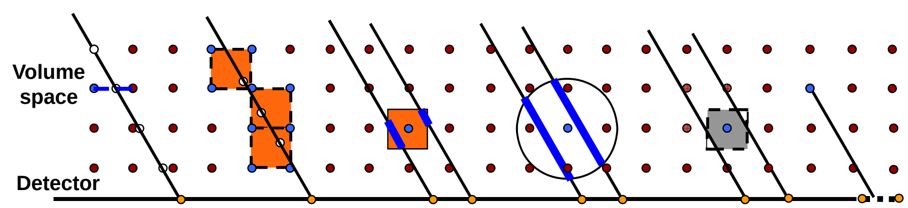
- select the slice most parallel to the detector
- non-uniform sampling, on-slice NN/BI
- perform as well as conventional sampling [Xu'06]
  - grid interpolated
  - box line integrated
  - etc.



splat

Projection

Investigated various schemes in terms of accuracy:



It was shown that the convenient slice-interpolated scheme is qualitatively competitive to the more involved ones listed here.

- see Xu / Mueller, "A comparative study of popular interpolation and integration methods for use in computed tomography," *IEEE 2006 International Symposium on Biomedical Imaging (ISBI '060*

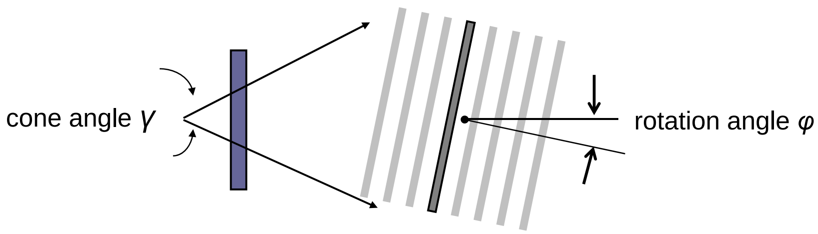
Implementation

Create polygon proxies for each slice

Associate texture (slice content) with the polygon

Render polygons

- 4 vertices carry voxel coordinates
- 3D transformation is done on GPU, for ALL points in the polygon
- volume slice gets interpolated in the fragment shader



Pixel-Wise Operation

Correction

- algebraic: scanned projection – simulated projection
- EM: scanned projection / simulated projection

Weighting, slice updating…

```
glMatrixMode(GL_MODELVIEW); glLoadIdentity();
glMatrixMode(GL_PROJECTION); glLoadIdentity();
gluOrtho2D(0, w, 0, h); glViewport(0, 0, w, h);
glBegin(GL_POLYGON);
   glMultiTexCoord2iARB(GL_TEXTURE0_ARB, 0, 0);
   glMultiTexCoord2iARB(GL_TEXTURE0_ARB, w, 0);
   glMultiTexCoord2iARB(GL_TEXTURE0_ARB, w, h);
   glMultiTexCoord2iARB(GL_TEXTURE0_ARB, 0, h);
   … …
glEnd();

float diff   = texRECT(tex0,texcoords0) - texRECT(tex1, texcoords1);
float weight = texRECT(tex2, texcoords2);
float corr   = (weight == 0) ? 0: diff / weight;
```


Attenuation Modeling

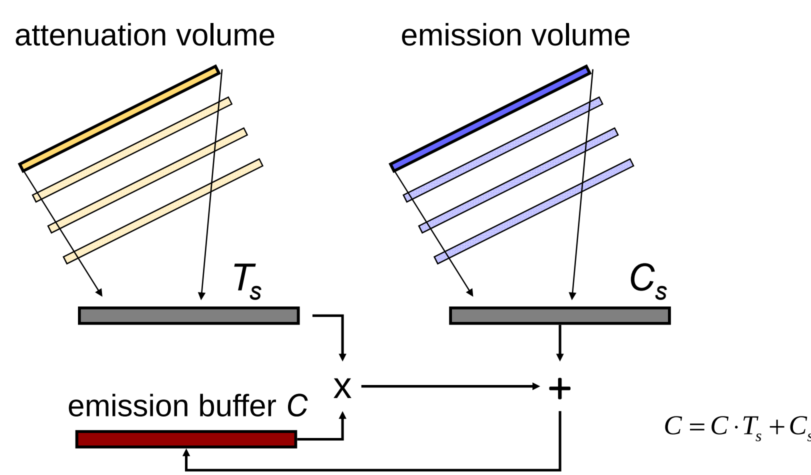
Two sliced volumes

- attenuation A + emission C (under reconstruction)
- first normalize A to [0…1]
- then compute T = 1 - A

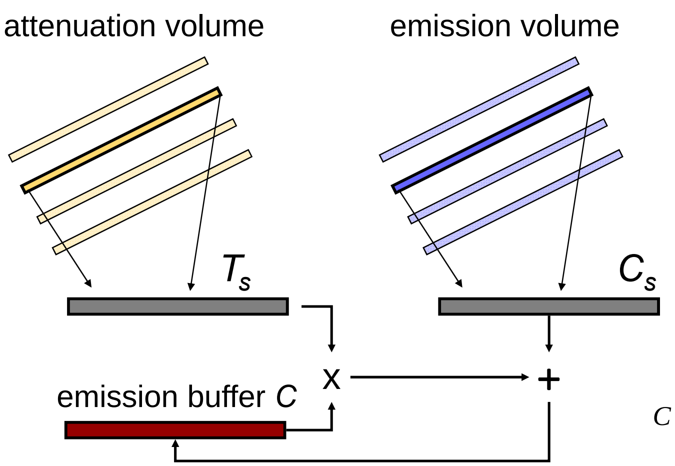
Composition

- front-to-back: emission + transparency buffer
- back-to-front: emission buffer (simpler)

Attenuation Modeling: Projection

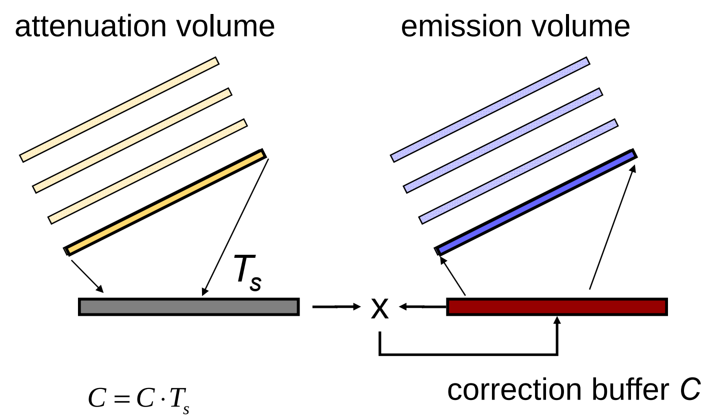


Attenuation Modeling: Projection



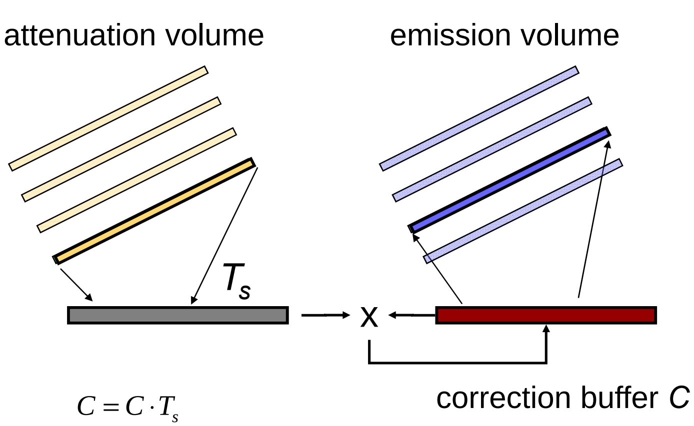
$$C = C \cdot T_s + C_s$$

Attenuation Modeling: Backprojection



$$C = C \cdot T_s$$

Attenuation Modeling: Backprojection



$$C = C \cdot T_s$$

Scattering Effects

Recursive convolution using a Gaussian filter [Bai'00][Zeng'00]

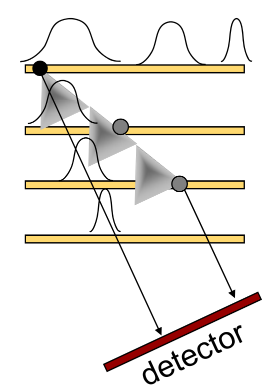
For slice-based projection

- attenuation adjusted kernels
- distance adjusted kernels
- direction adjusted kernels

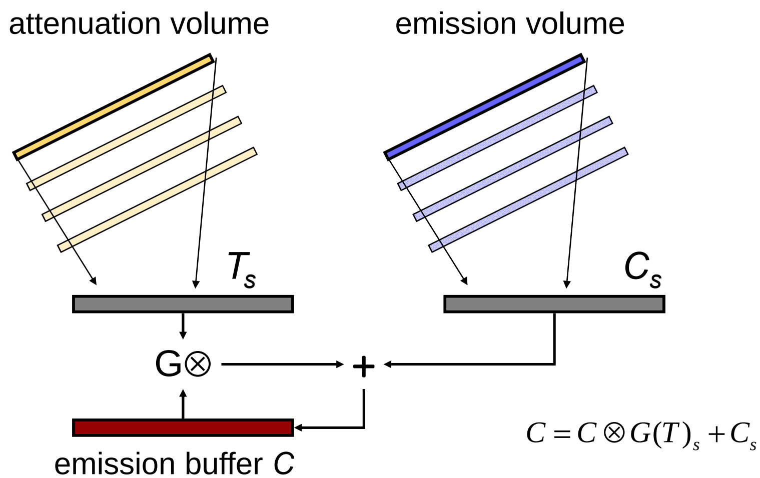
Projection: Back-to-front order

- adjusted Gaussian blurring
  - ➔ scatter energy
- multiply with (1 - attenuation volume)
  - ➔ attenuate energy
- add to the slice in the front
  - ➔ accumulate energy

Backprojection: Front-to-back

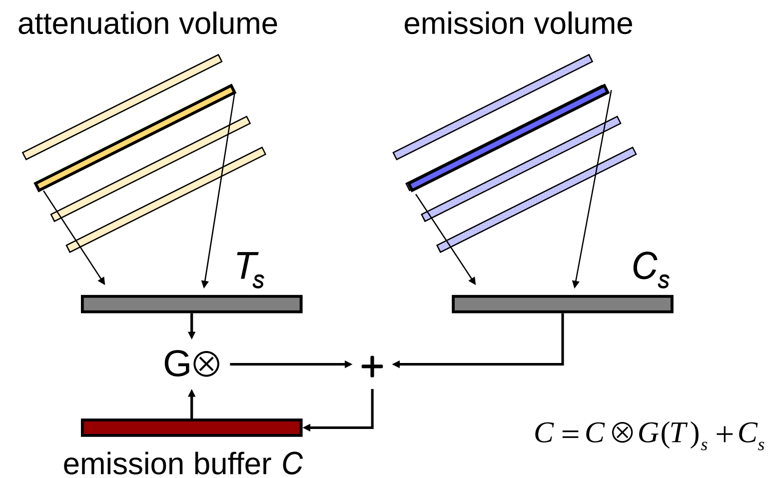


Scattering: Projection



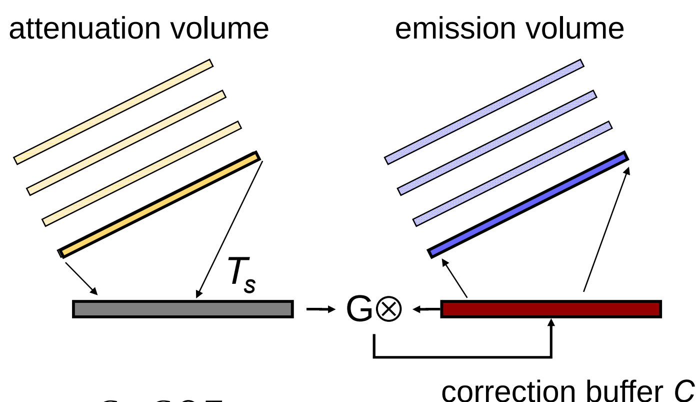
$$C = C \otimes G(T)_s + C_s$$

Scattering: Projection



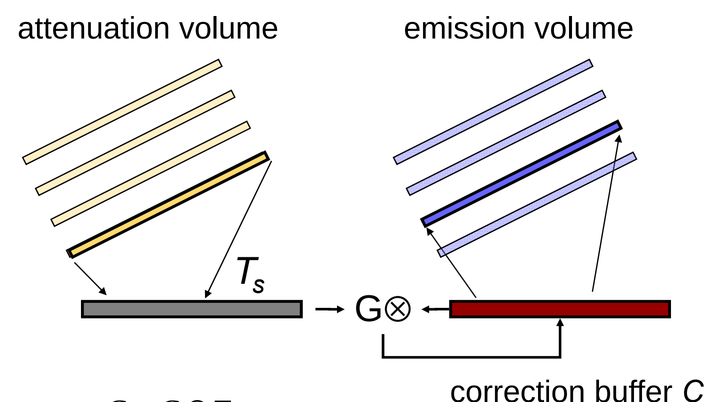
$$C = C \otimes G(T)_s + C_s$$

Scattering: Backprojection



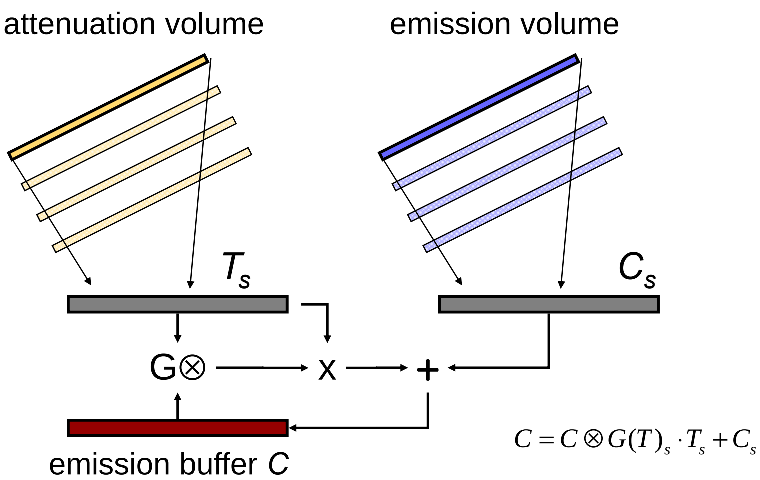
$$C = C \otimes T_s$$

Scattering: Backprojection



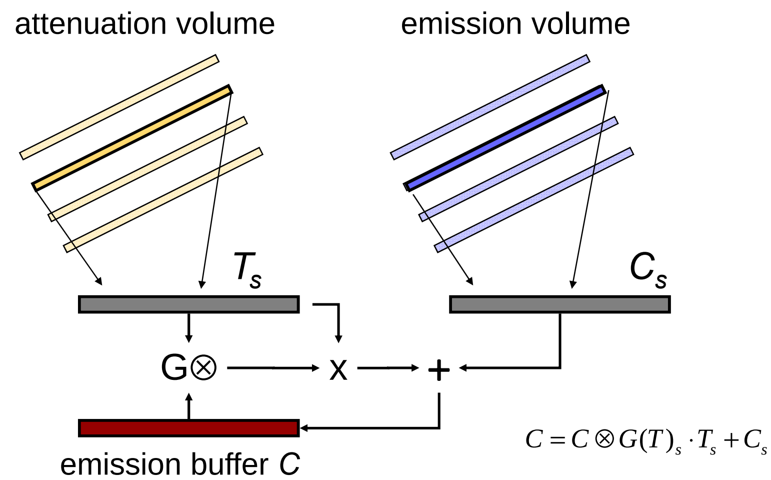
$$C = C \otimes T_s$$

Attenuation + Scattering: Projection



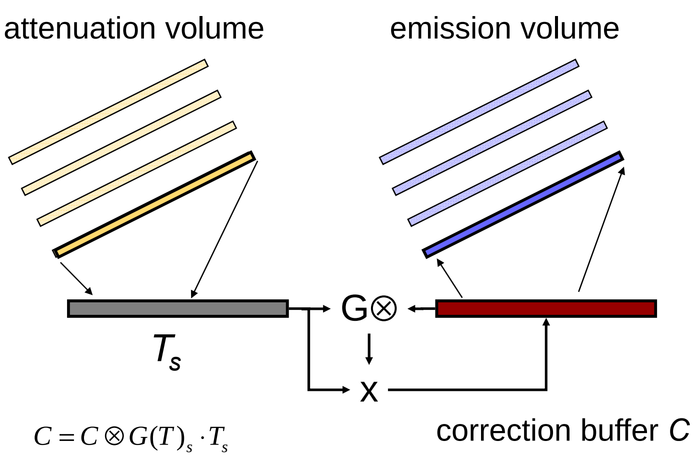
$$C = C \otimes G(T)_s \cdot T_s + C_s$$

Attenuation + Scattering: Projection



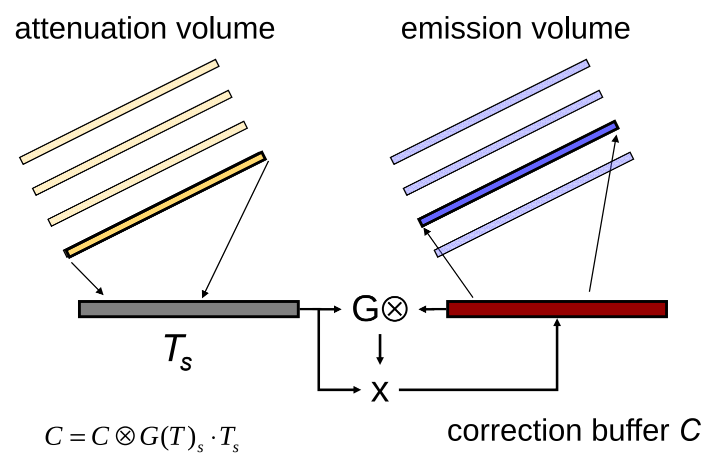
$$C = C \otimes G(T)_s \cdot T_s + C_s$$

Attenuation + Scattering: Backproj.



$$C = C \otimes G(T)_s \cdot T_s$$

Attenuation + Scattering: Backproj.



$$C = C \otimes G(T)_s \cdot T_s$$

Texture Operations

We may store transparency and emission volumes in 3D textures

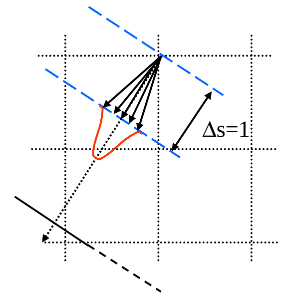
However, 3D textures do not provide efficient write-back capabilities

- solution: perform forward projection with 3D textures and back-projection with 2D textures
- however, 3D/2D texture switching is expensive

Therefore, we use a 2D texture scheme for all operations

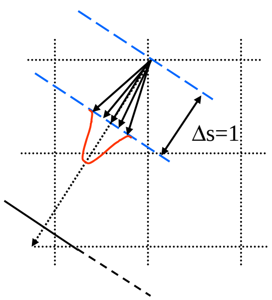
- using an axis-aligned scheme
- but, can this be as accurate than a true 3D texture scheme?

Using 2D Textures

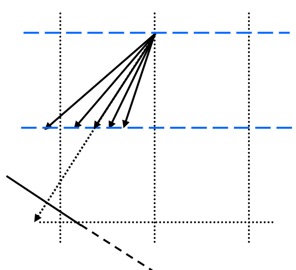


3D texture

Using 2D Textures

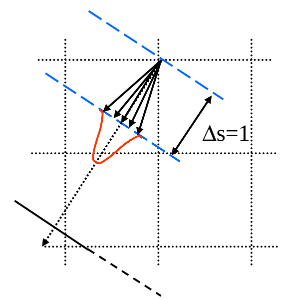


3D texture

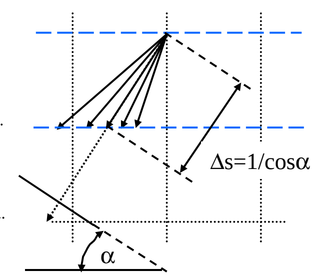


2D texture

Using 2D Textures

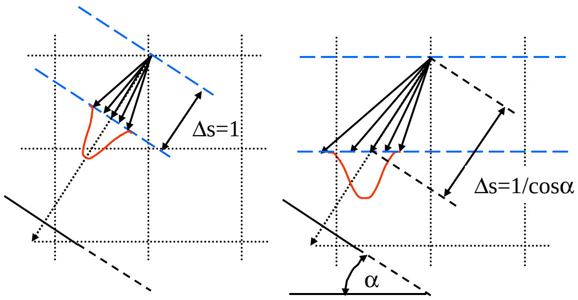


3D texture



2D texture

Using 2D Textures

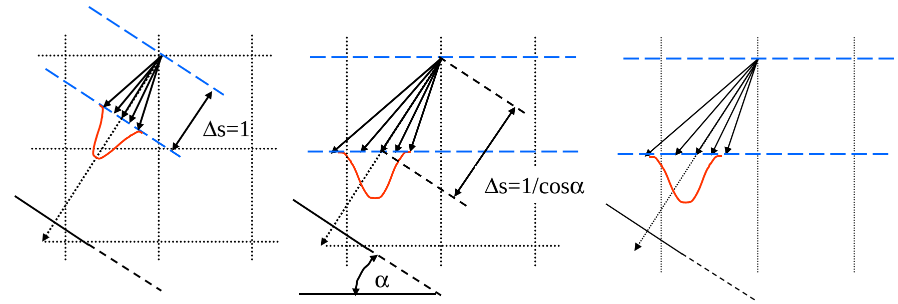


3D texture

2D texture

- scale kernel width by 1/cosα

Using 2D Textures



3D texture

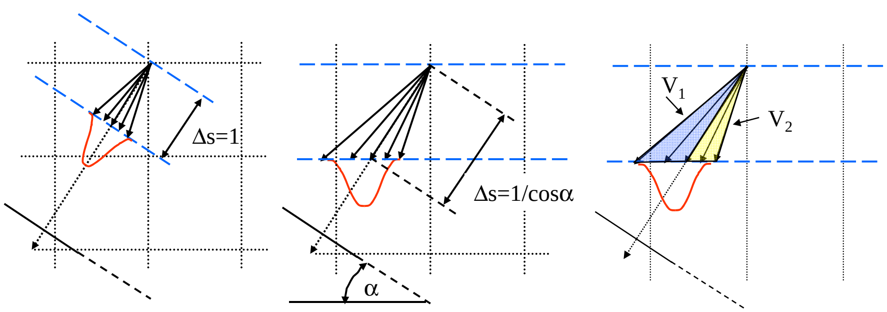
2D texture

- scale kernel width by 1/cosα

2D texture

- scale kernel width by 1/cosα

Using 2D Textures



3D texture

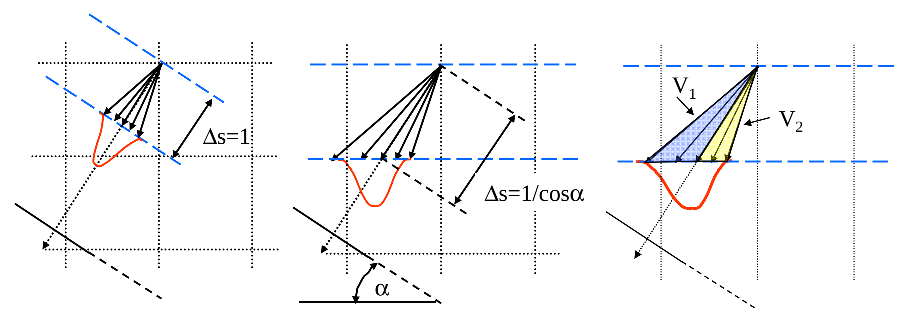
2D texture

- scale kernel width by 1/cosα

2D texture

- scale kernel width by 1/cosα

Using 2D Textures



3D texture

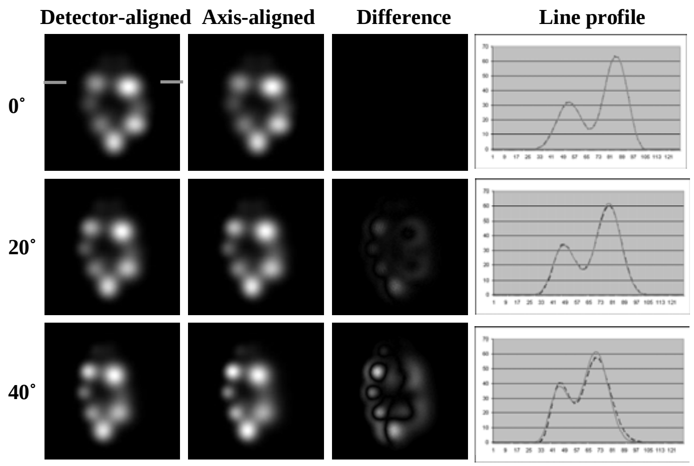
2D texture

- scale kernel width by 1/cosα

2D texture

- scale kernel width by 1/cosα
- adjust kernel width for perspective distortion

Results: Texture Schemes



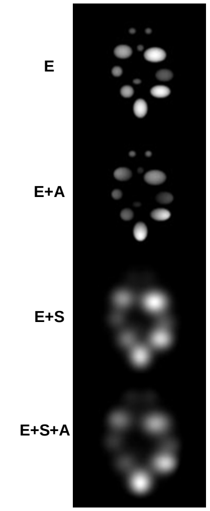
RMS error within 1-2%

Results: Simulations

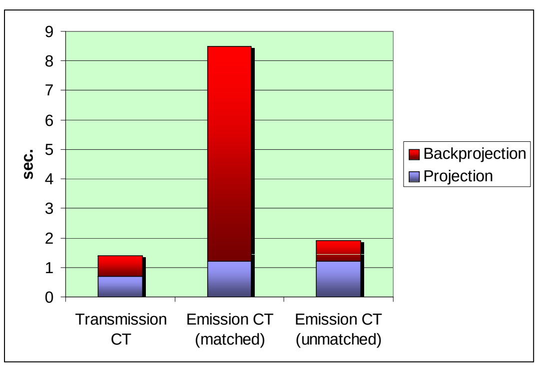
Scattering creates substantially more blur

Attenuation weakens the projections of emissions traversing highly attenuating material

- both with and without scattering.



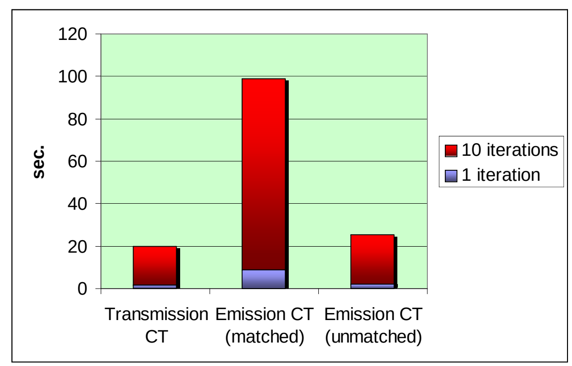
Results: Performance



Adding attenuation and scattering to projector adds little overhead

- compensated back-projector 5 more expensive
- thus, unmatched projector-backprojector significantly more efficient
- will be used for reconstruction

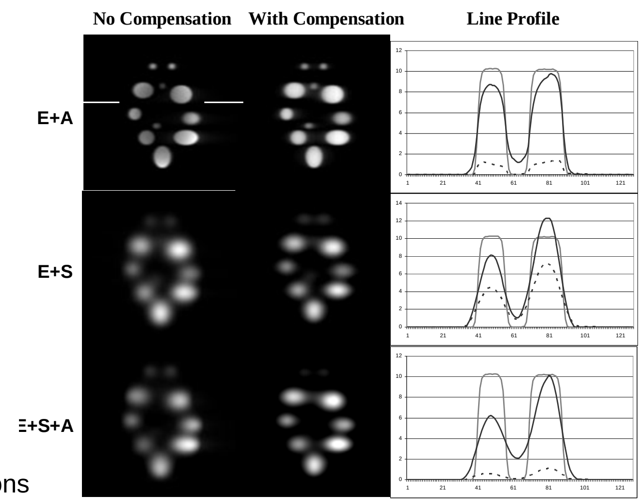
Results: Performance



Adding attenuation and scattering to projector adds little overhead

- compensated back-projector 5 more expensive
- thus, unmatched projector-backprojector significantly more efficient
- will be used for reconstruction

Results: Reconstructions



10 iterations

Next:

CT pipeline implementation with:

- graphics-style programming
- GPGPU-style programming with CUDA

Advanced issues:

- load balancing
- large datasets

Iterative Pipeline

Two stack representation

Kernel selection depends on algorithms

Projection/Backprojection

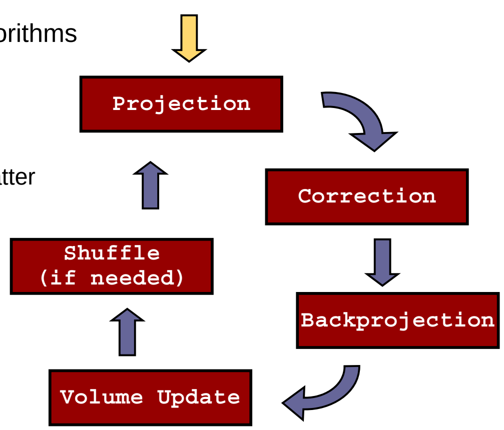
- attenuation only
- attenuation + emission
- attenuation + emission + scatter

Correction

- subtraction (algebraic)
- division (EM)

Update

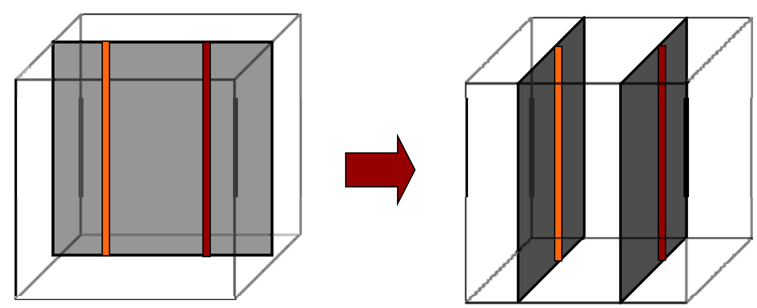
- addition (algebraic)
- multiply (EM)



Shuffle

Maintain data consistency

Data transfer/copy



Feldkamp (FDK) Filtered-Backprojection

SPIE Medical Imaging

Three consecutive steps

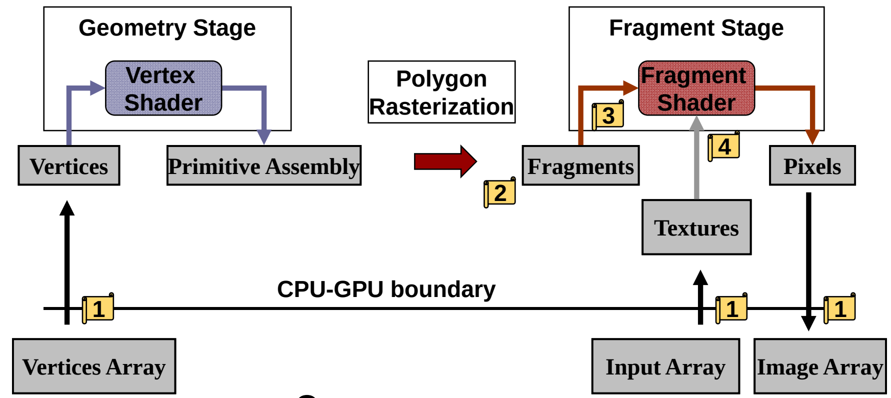
- projection space filtering
- backprojection
- volume space weighting

Projection space filtering using FFT

- FFT is sequential
- GPU-based FFT
  - [Govindaraju'06]
  - [Sumanaweera'05]
- good for >10k elements
- FFTW

Graphics Pipeline Revisited

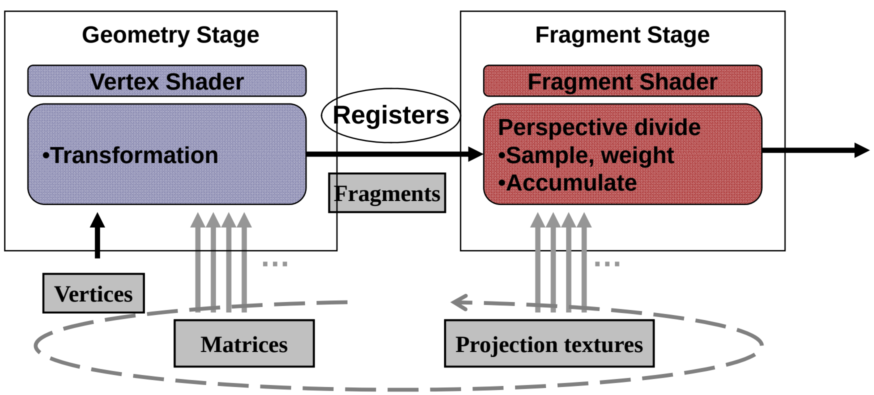
SPIE Medical Imaging



One pass

Pipeline: Graphics

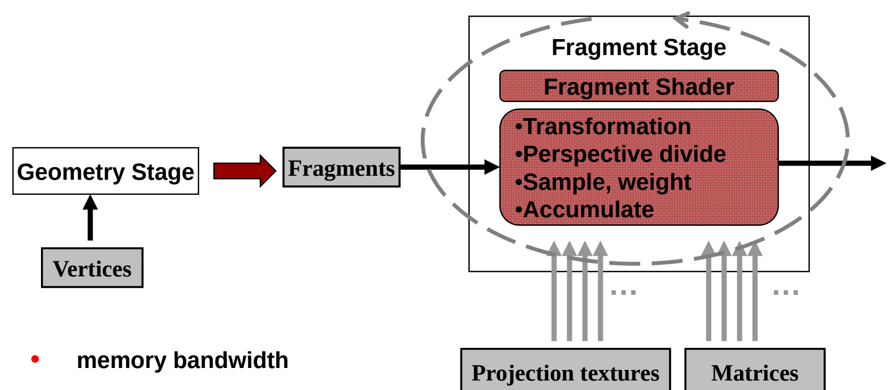
SPIE Medical Imaging



- **number of projections in one pass limited by the registers (maximum = 8)**
- **more balanced pipeline**

Pipeline: GPGPU

SPIE Medical Imaging



- **memory bandwidth**
- **instruction fill rate**
- **more projections, less passes & writes**
- **limitation of # of texture units ➔ 3D texture, high latency**

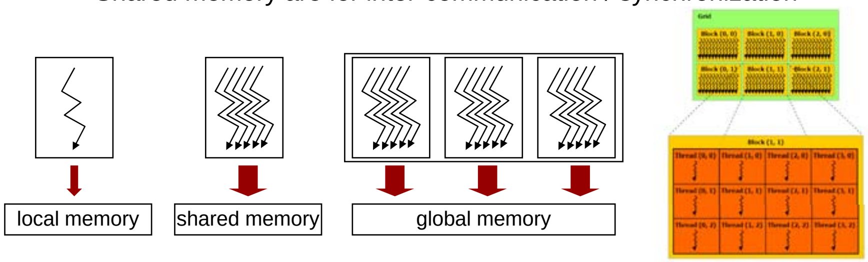
Specific Implementation: CUDA

SPIE Medical Imaging

C code for serial programs with parallel kernels

Two level hierarchy: blocks, grids

- Threads divided/grouped into blocks (scalable design)
- All threads run the same sequential kernel program
- Unique IDs are assigned to threads to compute address
- Shared memory are for inter-communication / synchronization



CUDA Features

SPIE Medical Imaging

Standard functions: `sinf(), atanf(), ceil()…`

Data types: `float4, int3...`

Texture type: `texture<float, 4>`

Run-time support

- Memory allocation: `cudaMalloc(), cudaFree()…`
- Memory copy: host ←→ device, device ←→device
- Texture management, OpenGL interoperability, etc.

Variable/Function qualifiers

```
__global__void kernel(){},  __constant__ float array[32];
```


Setup and Execute Basics

SPIE Medical Imaging

Design memory management

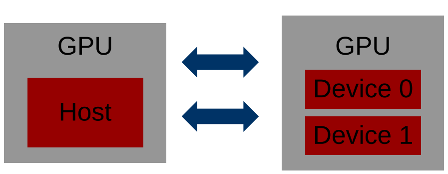
```
dim3 gridSize(50, 10); // 500 thread blocks
dim3 blockSize(4, 4, 8); // 128 threads per block
```

Implement and launch GPU kernels

```
kernel <<<gridSize, blockSize>>> (...); // kernel launch
```

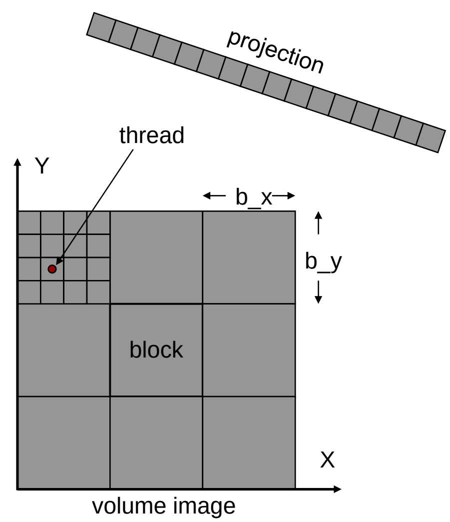
Data transfer

```
cudaMemCopy() (host ↔ device, device ↔ device)
```



Example: Backprojection

SPIE Medical Imaging



Maximum number of threads per block is 512

Maximum sizes of the X, Y, and Z dimension of a thread block are 512, 512,and 64

Number of threads per block is multiple of the warp size (32, 64, 128…)

Host Code:

```
dim3 blockDim(b_x, b_y);
dim3 gridDim(v_x/blockDim.x, v_y/blockDim.y)
Recon<<<gridDim, blockDim>>>(…)
```

Device Code:

```
x = blockIdx.x * b_x + threadIdx.x;
y = blockIdx.y * b_y + threadIdx.y;
p = RadonTransform(x, y);
volume[x, y] += projection[p];
```

Shared Memory Revisit

On-chip memory, small but extremely fast (16KB)

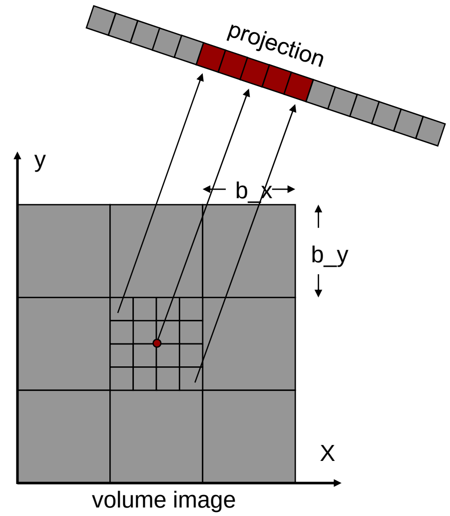
Data that are visible only to all threads in a thread block

Synchronization using barriers

- Prevent Read/Write conflicts

…step 1…
_syncthreads()
…step2

Optimize via Shared Memory, I



Load projection data to be sampled into shared memory before backprojection.

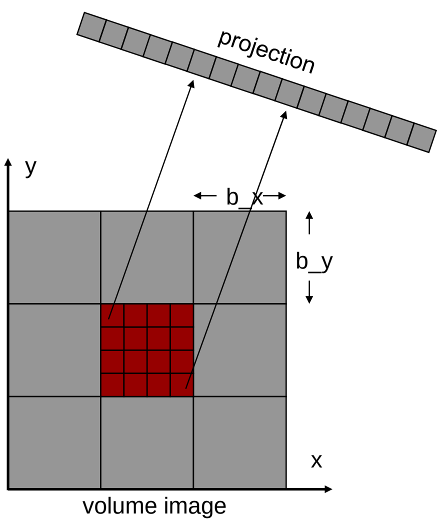
Reduce global memory read.

```
__shared__ float sharedMem[];

center.x = blockIdx.x * b_x + b_x/2;
center.y = blockIdx.y * b_y + b_y/2;
p_center = RadonTransform(center);

loadProjectionDataIntoSharedMem();
synchronizeThreads();
x = blockIdx.x * b_x + threadIdx.x;
y = blockIdx.y * b_y + threadIdx.y;
p = RadonTransformAndShift(x, y);
volume[x, y] = sharedMem[p];
```


Optimize via Shared Memory, II



Load sub-volume data into shared memory before backprojection.

Reduce global memory read/write.

```
__shared__ float sharedMem[];

loadSubVolumeDataIntoSharedMem();
loadSubVolumeCoordinatesIntoSharedMem();
loadTransformationMatrixIntoSharedMem();
synchronizeThreads();

p = RadonTransform(x, y);
sharedMem[x, y] = Image[p];
Volume[x, y] = sharedMem[x, y]
```


Precision of Computation

Variation from IEEE 754 but comparable to other processors

Getting better every generation (double precision supported in GTX 20)

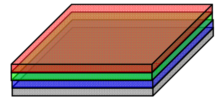
	NVIDIA Tesla T10	SSE2	Cell SPE
Precision	IEEE 754	IEEE 754	IEEE 754
Rounding modes for FADD and FMUL	All 4 IEEE, round to nearest, zero, inf, -inf	All 4 IEEE, round to nearest, zero, inf, -inf	All 4 IEEE, round to nearest, zero, inf, -inf
Denormal handling	Full speed	Supported, costs 1000's of cycles	Supported only for results, not input operands (input denormals flushed-to-zero)
NaN support	Yes	Yes	Yes
Overflow and Infinity support	Yes	Yes	Yes
Flags	No	Yes	Yes
FMA	Yes	No	Yes
Square root	Software with low-latency FMA-based convergence	Hardware	Software only
Division	Software with low-latency FMA-based convergence	Hardware	Software only
Reciprocal estimate accuracy	24 bit	12 bit	12 bit + step
Reciprocal sqrt estimate accuracy	23 bit	12 bit	12 bit + step
log2(x) and 2^x estimates accuracy	23 bit	No	No

Acceleration via RGBA

SPIE Medical Imaging

Volume packing

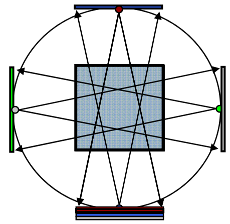
- adjacent 4 volume slices ➔ RGBA
- constant *Y* offset
- 1 matrix + 1 offset vector
- extract channel values
- reduce # of write-to-textures



$$\begin{bmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \\ a_{30} & a_{31} & a_{32} & a_{33} \end{bmatrix} \begin{bmatrix} x_v \\ y_v + k \\ z_v \\ 1 \end{bmatrix} = \begin{bmatrix} x_h \\ y_h \\ z_h \\ w_h \end{bmatrix} + k \cdot \begin{bmatrix} a_{01} \\ a_{11} \\ a_{21} \\ a_{31} \end{bmatrix}$$

Projection packing

- symmetry in projection layout
  - four orthogonal views
  - [φ, φ+90°, φ+180°, φ+270°]
  - no re-interpolation ➔ 4-fold speedup
- need all projections



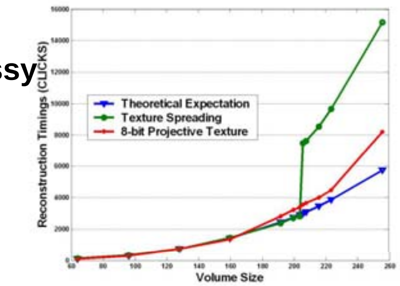
Memory Management

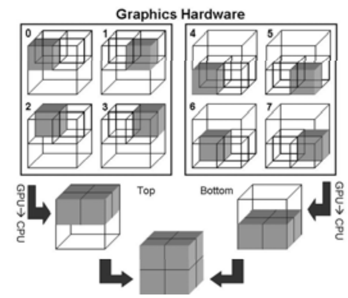
SPIE Medical Imaging

Scalability

Texture compression is lossy

Volume partition and stitch





Vol.	Projs.	Partition	Time
128^3	160	Yes/No	1.7s/1.7s
256^3	160	Yes/No	12s/45s

NVIDIA Geforce FX5900

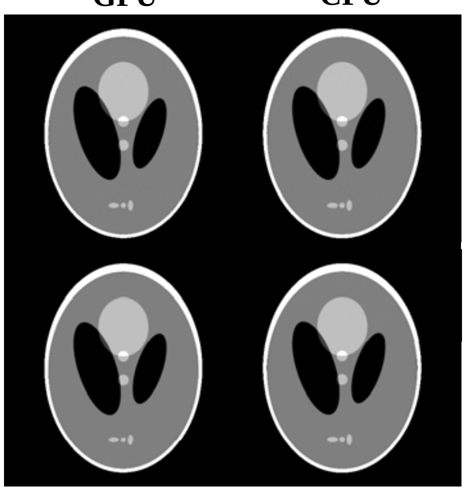
FDK: Reconstruction Quality Parallel-Beam

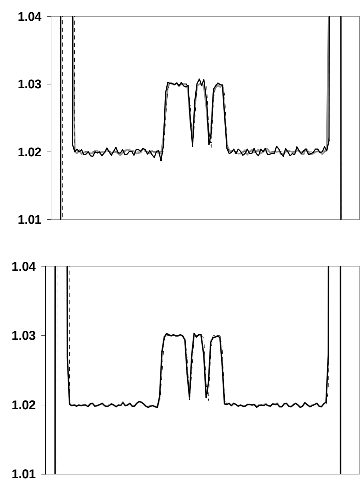
SPIE Medical Imaging

GPU CPU

Box

Linear





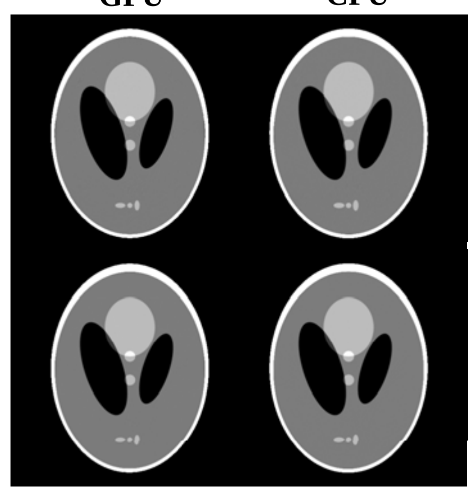
FDK: Reconstruction Quality Cone-Beam

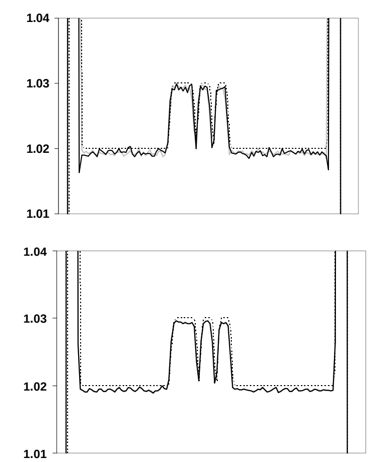
SPIE Medical Imaging

GPU CPU

Box

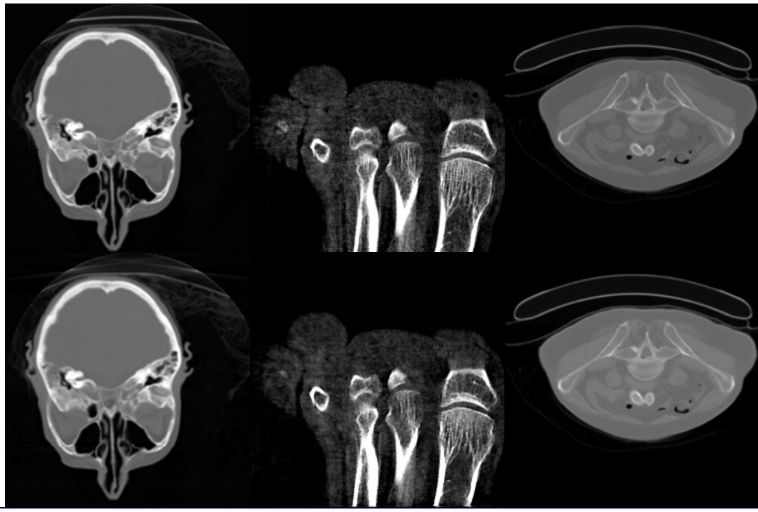
Linear





FDK: Medical Datasets

Head Toes Abdominal Aorta



Original

Reconstructed

Iterative: Performance

NVIDIA 8800 GTX, 768MB memory

Platform	Algorithm	Volume	Projections	1 iter.	3 iter.
SGI	SART	128^3	80	1.1min	3.1min
GPU	SART	128^3	80	2s	6.1s
GPU	OS-EM	128^3	80	5s	15s
GPU	SART	256^3	160	7.5s	22s
GPU	SIRT	256^3	160	5.0s	15s

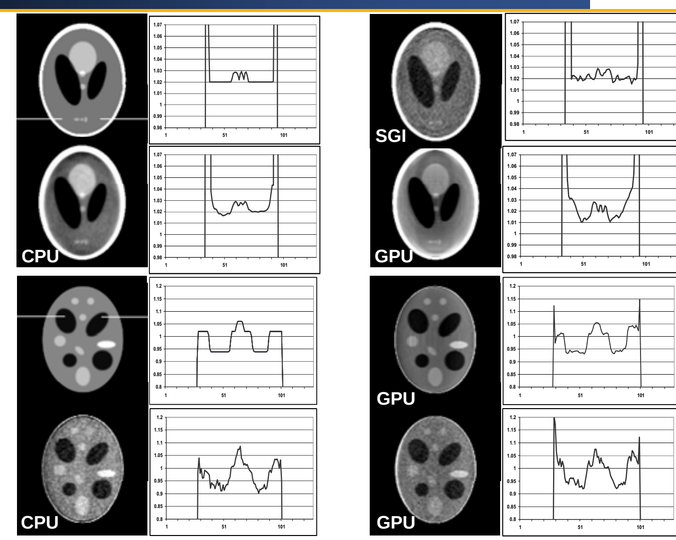
Iterative: Performance

Algorithm	Volume	Projections	1 iter.	3 iter.
Emission SART	128^3	80	8.5s	25s
Emission SART	256^3	160	12s	35s

Iterative: Reconstruction Quality

SART

EM



Course Schedule

1:30 – 2:00: Introduction (Klaus Mueller)

2:00 – 2:45: Graphics-style GPU programming with CG (Wei Xu)

2:45 – 3:00: GPGPU-style GPU programming with CUDA (Ziyi Zeng)

Coffee Break

3:30 – 4:00: GPGPU-style GPU programming with CUDA (Ziyi Zeng)

4:00 – 4:20: CT reconstruction pipeline components (Klaus Mueller)

4:20 – 5:20: GPU-accelerated CT reconstruction (Fang Xu)

5:20 – 5:30: Extensions and final remarks (all)