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

SPIE Medical Imaging

CT Reconstruction Examples

Klaus Mueller
Stony Brook University
Computer Science
Stony Brook, NY

Fang Xu
Siemens USA Research
Princeton, NJ

Overview

SPIE Medical Imaging

What to expect:

- details on the parallelization of various fundamental CT reconstruction algorithms
- insights on CUDA implementations of these
- details of memory configuration on CUDA
- CUDA optimization approach using share memory

Decomposition

SPIE Medical Imaging

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

FBP

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

S: scanner projections
I: identity projection/volume

Algebraic

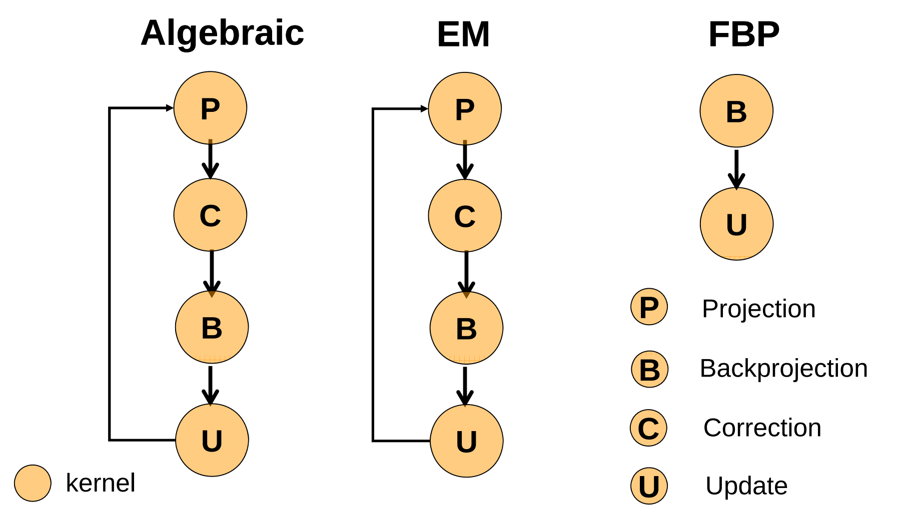
$$v_j = v_j + \frac{\sum_{p_i \in P_\varphi}\left(\frac{\lambda\left(p_i - \sum_{l=0}^{N^3-1} v_l \cdot w_{il}\right)}{\sum_{l=0}^{N^3-1} w_{il}}\right) w_{ij}}{\sum_{p_i \in P_\varphi} w_{ij}} = \boxed{v_j + \frac{B(\lambda \frac{S-P(V)}{P(I)})}{B(I)}}$$

OS-EM

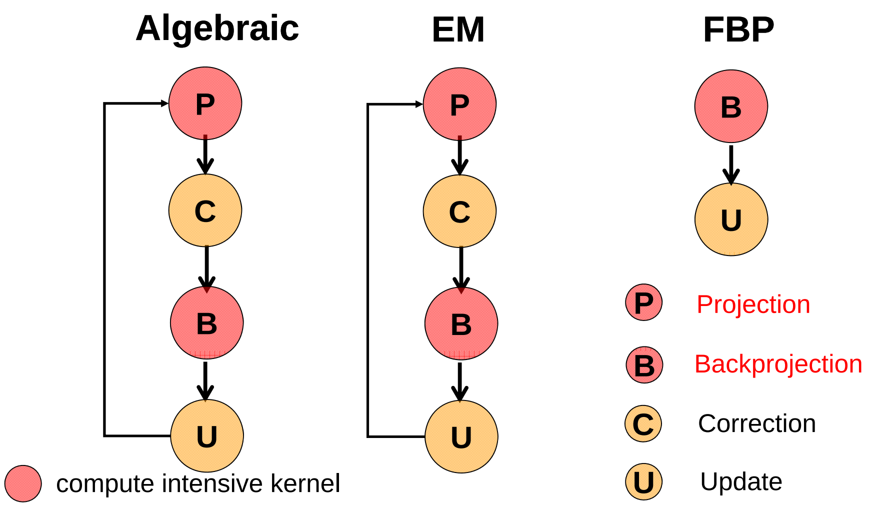
$$v_j = \frac{v_j}{\sum_{p_i \in P_{set}} w_{ij}}\left(\sum_{p_i \in P_{set}}\left(\frac{p_i}{\sum_{l=0}^{N^3-1} v_l \cdot w_{il}}\right) w_{ij}\right) = \boxed{\frac{v_j}{\sum_{p_i \in P_{set}} B(I)}\left(\sum_{p_i \in P_{set}} B \frac{S}{P(V)}\right)}$$

Kernel-Centric Reconstruction

SPIE Medical Imaging

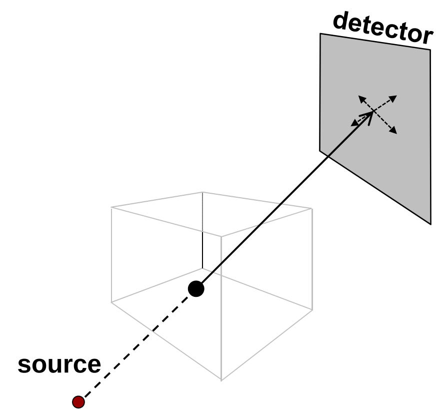


Kernel-Centric Reconstruction



Backprojection: Options

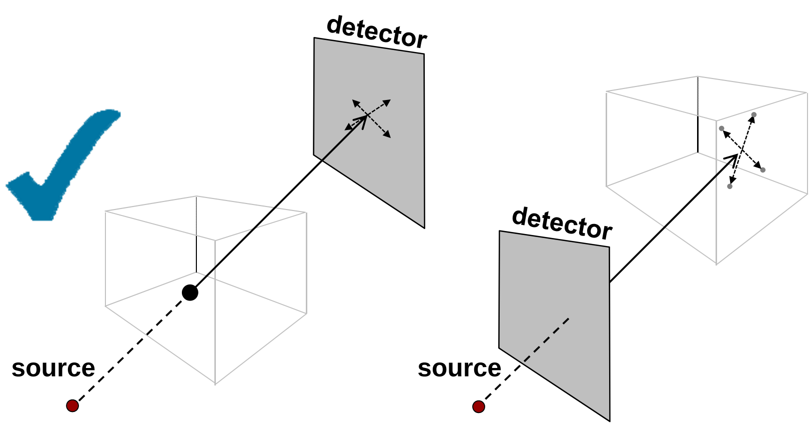
- voxel-driven: sample in projection space
- one write per thread



Backprojection: Options

- voxel-driven: sample in projection space
- one write per thread

- pixel-driven, sample in volume space
- multiple write per thread (scatter)

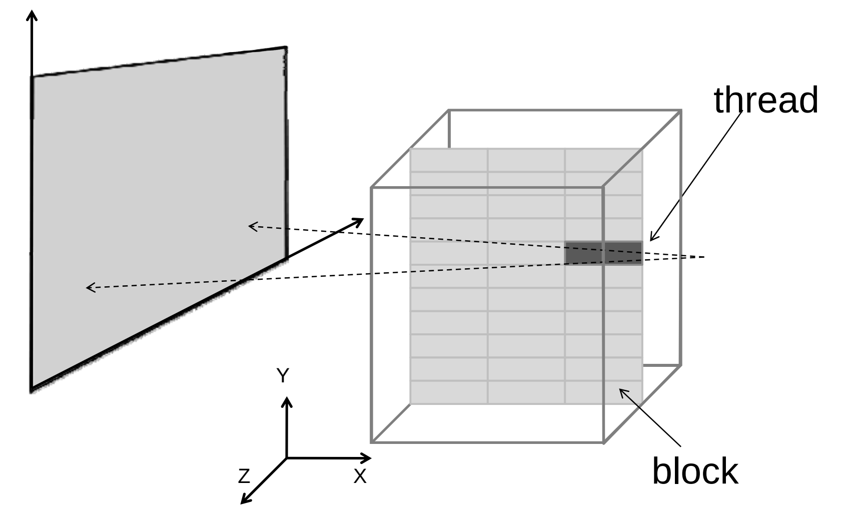


CUDA Memory Revisit

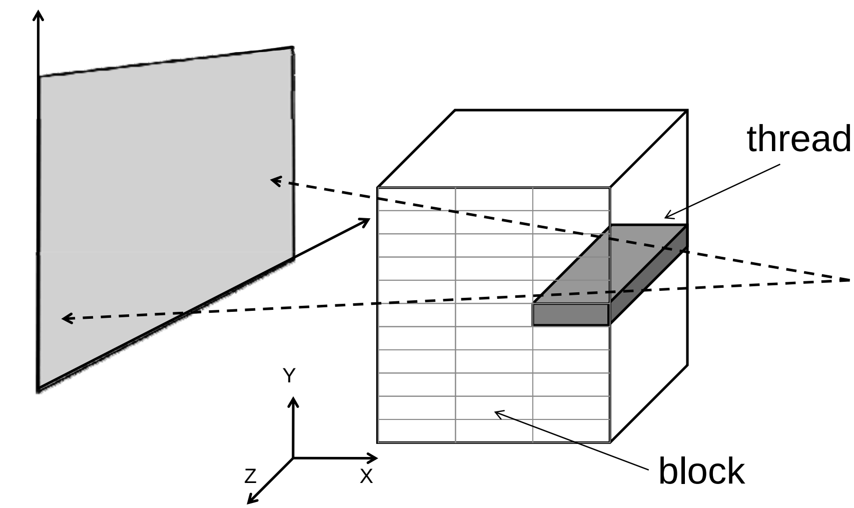
	Global Memory	Texture Memory
Access	Read/Write	Read only
Cached	No	Yes
Subject to coalescing	Yes	No
Interpolation	No support	Hardwired
Dimension	arbitrary	1D, 2D, 3D (supported after CUDA 2.0)

volume **projections**

CUDA Configuration: 2D



CUDA Configuration: 3D



Transformation Matrix

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

Perform perspective divide if necessary (cone-beam)

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

CUDA Implementation

[Host]:

for all projections P_i, trigger kernel on device

[Device]: per thread

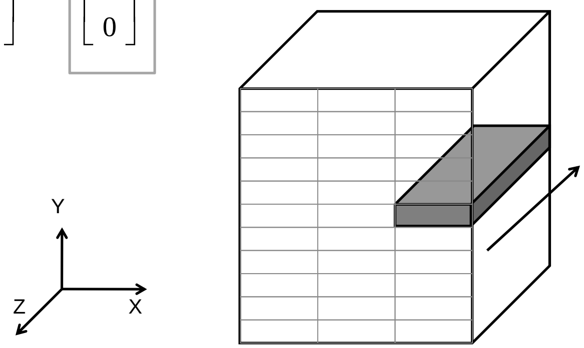
$$M^{3\times 4} \cdot \begin{bmatrix} x_v \\ y_v \\ z_v \\ 1 \end{bmatrix} = \begin{bmatrix} x_h \\ y_h \\ w_h \end{bmatrix}$$

loop through each voxel in the thread

- obtain voxel coordinates in volume space
- compute projected coordinates on the detector using a 3x4 transformation matrix *M*
- perform perspective-divide if needed $P_\varphi(u,v) = \left(\frac{x_h}{w_h}, \frac{y_h}{w_h}\right)$
- depth weighting if needed
- interpolate pixel values on the detector image (bilinear)
- accumulate sampled values on voxel

Incremental Computation

$$M \cdot \begin{bmatrix} x_v \\ y_v \\ z_v + \Delta z \\ 1 \end{bmatrix} = M \cdot \begin{bmatrix} x_v \\ y_v \\ z_v \\ 1 \end{bmatrix} + M \cdot \boxed{\begin{bmatrix} 0 \\ 0 \\ \Delta z \\ 0 \end{bmatrix}}$$



CUDA sample code (2D)

```
__global__ void
backproject( float* g_odata, int width, int height, float* mat, float z_coord)
{
    // calculate normalized texture coordinates
    unsigned int x = blockIdx.x*blockDim.x + threadIdx.x;
    unsigned int y = blockIdx.y*blockDim.y + threadIdx.y;

   float4 vc = make_float4((float)x, (float)y, z, 1.0);           // voxel coordinates
   float4 ec;         // eye coordinates

   ec.x = mat[0]*vc.x  + mat[1]*vc.y  + mat[2]*vc.z  + mat[3]*vc.w;
   ec.y = mat[4]*vc.x  + mat[5]*vc.y  + mat[6]*vc.z  + mat[7]*vc.w;
   ec.z = mat[8]*vc.x  + mat[9]*vc.y  + mat[10]*vc.z + mat[11]*vc.w;

   float2 pc = make_float2(ec.x/ec.z, ec.y/ec.z);                 // pixel coordinates

    // read from texture and write to global memory
    g_odata[y*width + x] += tex2D(tex, pc.x, pc.y);
}
```


CUDA sample code (3D)

```
__global__ void
backproject3D( float* g_odata, int nx, int ny, int nz, float* mat)
{
   unsigned int x = blockIdx.x*blockDim.x + threadIdx.x;
   unsigned int y = blockIdx.y*blockDim.y + threadIdx.y;

   // first voxel on the line (z = 0)
   float4 ec;
   ec.x = mat[0]*x  + mat[1]*y  + mat[3];
   ec.y = mat[4]*x  + mat[5]*y  + mat[7];
   ec.z = mat[8]*x  + mat[9]*y  + mat[11];

   float2 pc = make_float2(ec.x/ec.z, ec.y/ec.z);  // pixel coordinates
   int index = y*nx + x; int num = nx*ny;
   g_odata[index] += tex2D(tex, pc.x, pc.y);

   // rest of voxels on the line
   for (int z = 1; z < nz; z++)
   {
          ec.x += mat[2]; ec.y += mat[6]; ec.z += mat[10];   // update incrementally

          pc = make_float2(ec.x/ec.z, ec.y/ec.z);
          g_odata[z*num + index] += tex2D(tex, pc.x, pc.y);
   }
}
```


Next: Iterative Algorithms

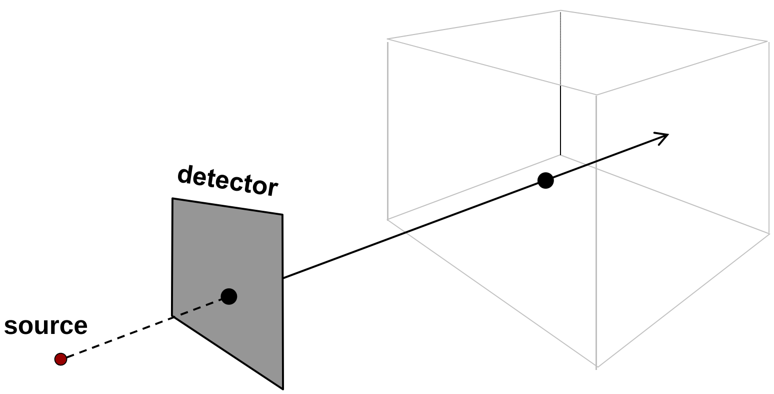
SART, EM

All require a projection simulation step

- should be as accurate as possible

Projection

Sample in volume space (pixel-driven / ray-driven)



CUDA Memory Revisit

	Global Memory	Texture Memory
Access	Read/Write	Read only
Cached	No	Yes
Subject to coalescing	Yes	No
Interpolation	No support	Hardwired
Dimension	arbitrary	1D, 2D, 3D (supported after CUDA 2.0)

projections **volume**

Projection: Memory

Ray-driven: sampling in volume space (trilinear interpolation)

Volume can be represented as either

- a single 3D texture (supported after CUDA 2.0)
- stacks of 2D textures
  - A 3rd interpolation between adjacent 2D slices

Inverse Transformation Matrix

voxel coordinates *pixel coordinates*

$$\underbrace{\begin{bmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \end{bmatrix}}_{M^{3x4}} \begin{bmatrix} x_v \\ y_v \\ z_v \\ 1 \end{bmatrix} = \underbrace{\boxed{\begin{bmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{bmatrix}}}_{M^{3x3}} \begin{bmatrix} x_v \\ y_v \\ w_v \end{bmatrix} + \underbrace{\boxed{\begin{bmatrix} a_{03} \\ a_{13} \\ a_{23} \end{bmatrix}}}_{C} = \begin{bmatrix} x_h \\ y_h \\ w_h \end{bmatrix}$$

$$P_{\varphi}(u,v) = \left(\frac{x_h}{w_h}, \frac{y_h}{w_h}\right)$$

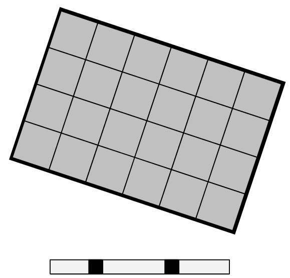
$$\begin{bmatrix} x_v \\ y_v \\ w_v \end{bmatrix} = \left(M^{3\times3}\right)^{-1} \left(\begin{bmatrix} x_h \\ y_h \\ w_h \end{bmatrix} - C \right) = \begin{pmatrix} row_x \\ row_y \\ row_z \end{pmatrix} \left(\begin{bmatrix} P_u \\ P_v \\ 1 \end{bmatrix} w_h - C \right)$$

voxel coordinates *pixel coordinates*

Projection Algorithm

Raycasting methods [Krueger'03]

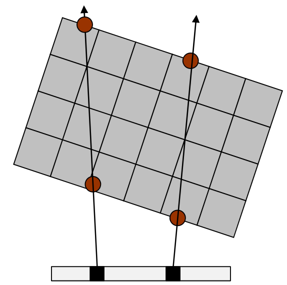
- [Host]:
  - generate volume bounding box (aligned with axis X/Y/Z)
  - generate threads for each pixel (ray), trigger kernel on device



Projection Algorithm

Raycasting methods [Krueger'03]

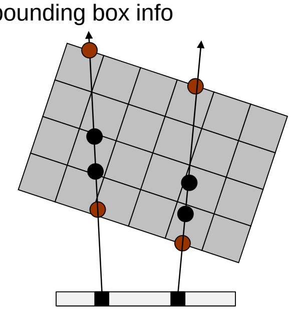
- [Host]:
  - generate volume bounding box (aligned with axis X/Y/Z)
  - generate threads for each pixel (ray), trigger kernel on device
- [Device]: in each thread
  - obtain ray entry & exit points using volume bounding box info
  - get ray directions using entry & exit points



Projection Algorithm

Raycasting methods [Krueger'03]

- [Host]:
  - generate volume bounding box (aligned with axis X/Y/Z)
  - generate threads for each pixel (ray), trigger kernel on device
- [Device]: in each thread
  - obtain ray entry & exit points using volume bounding box info
  - get ray directions using entry & exit points
  - cast rays, inside the loop:
    - sample in volume space
    - accumulate values
    - step forward equidistantly

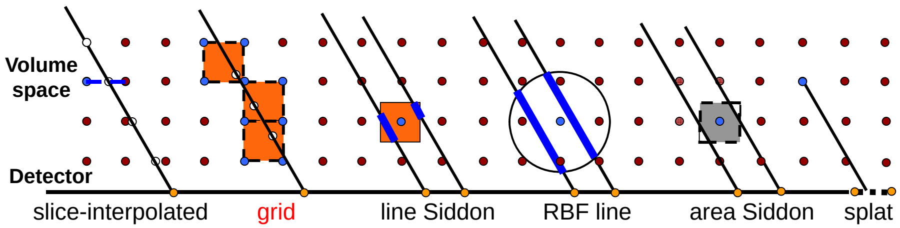


CUDA sample code

```
__global__ void
project(float *image_output, int width, int height, float3 volDim)
{
    // volume bounding box
    float3 volBox0 = make_float3(0, 0, 0);
    float3 volBox1 = volDim;

    // obtain ray direction and entry point by intersecting with the bounding box
    float3 rayDir, rayEntry;
    int stepNumber = getRayInfo(volBox0, volBox1, &rayDir, &rayEntry);

    // march along ray from back to front, accumulating color
    float sum = 0.0f;
    for(int s=0; s<stepNumber; s++)
    {
            float3 pos = rayEntry + rayDir*s;
            pos = pos/volDim;                                   // map position to [0, 1]
            float sample = tex3D(tex, pos.x, pos.y, pos.z);     // read from 3D texture
            sum += sample;                                      // accumulate result
    }

    // pixel coordinates
    int x = __umul24(blockIdx.x, blockDim.x) + threadIdx.x;
    int y = __umul24(blockIdx.y, blockDim.y) + threadIdx.y;

    // write output value
    if ((x < width) && (y < height))
    {
            int i = __umul24(y, width) + x;
            image_output[i] = sum;
    }
}
```

Projection Accuracy

Investigated various schemes in terms of accuracy:



It was shown that the convenient grid-interpolated (trilinear) 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 '06)*

Example: Iterative Algorithms

Kernel selection depends on algorithms

Projection/Backprojection

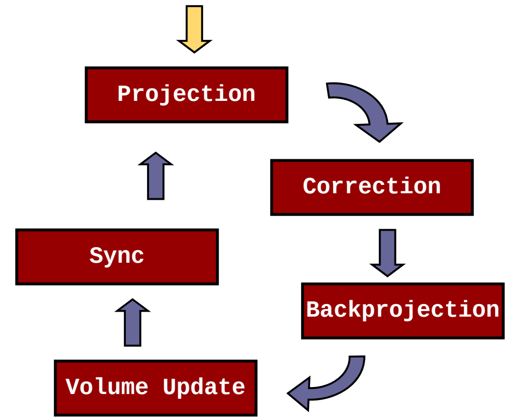
Correction

- pixel-wise operation
- subtraction (algebraic)
- division (EM)

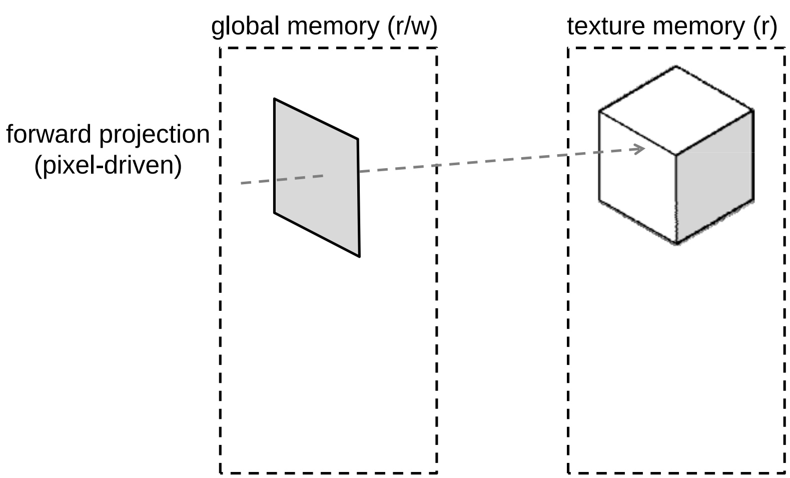
Update

- addition (algebraic)
- multiply (EM)

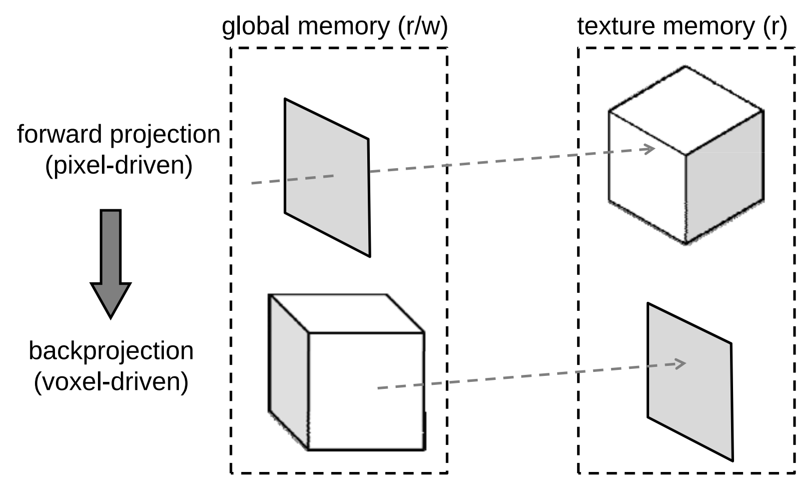
Sync



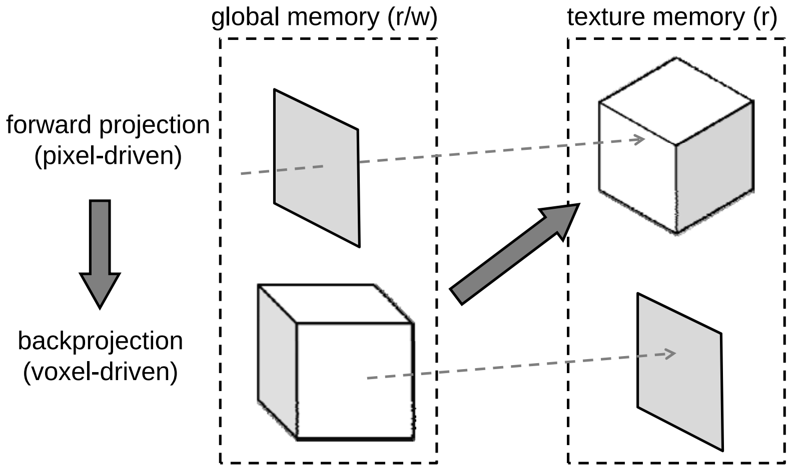
Sync



Sync



Sync



Emission CT: Attenuation Modeling

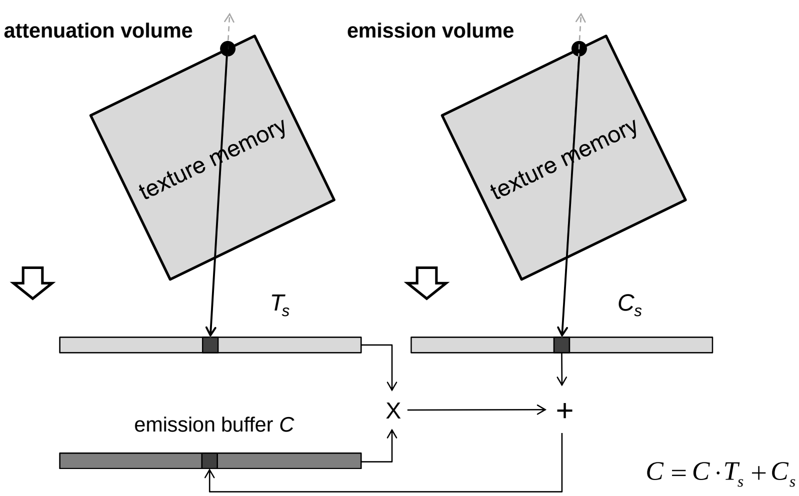
Two volumes

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

Composition

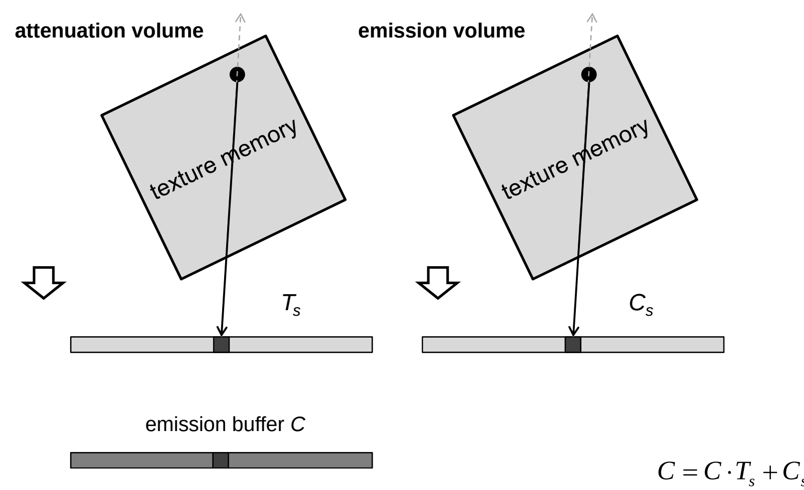
- forward projection: back-to-front compositing
- backward projection: front-to-back

Attenuation Modeling : Projection



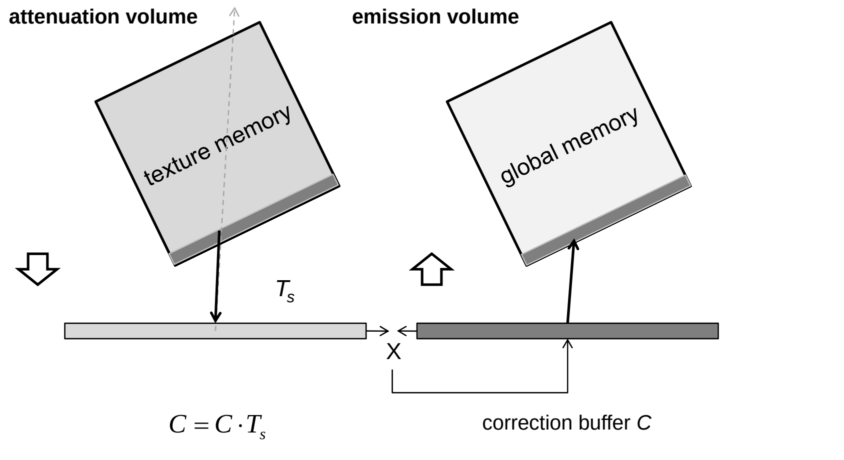
$$C = C \cdot T_s + C_s$$

Attenuation Modeling : Projection



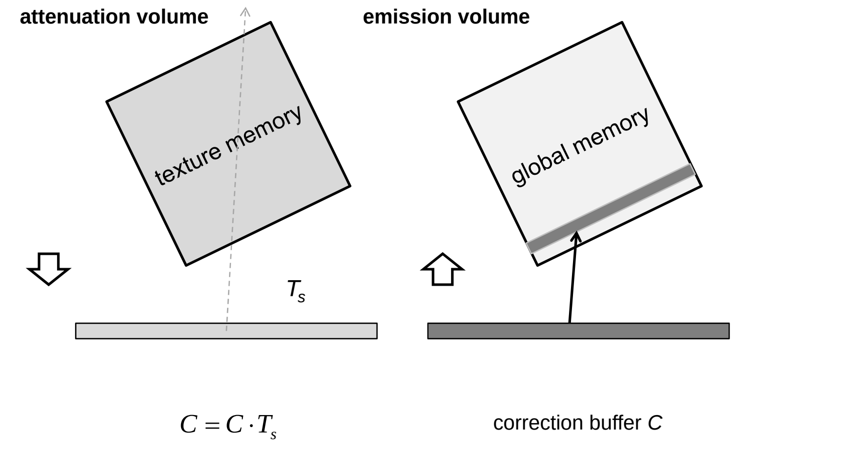
$$C = C \cdot T_s + C_s$$

Attenuation Modeling : Backprojection



slice by slice update

Attenuation Modeling : Backprojection



slice by slice update

Scattering Effects

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

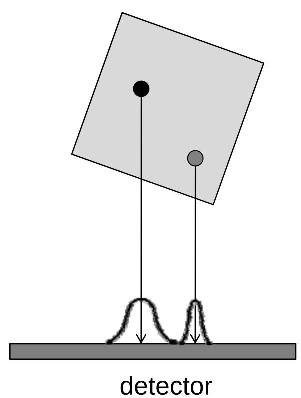
For projection

- attenuation adjusted kernels
- distance adjusted kernels

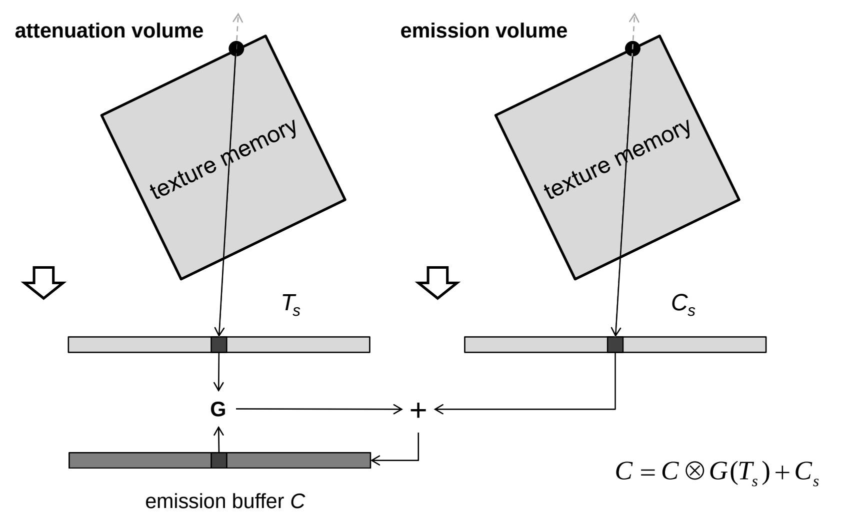
Projection: back-to-front order

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

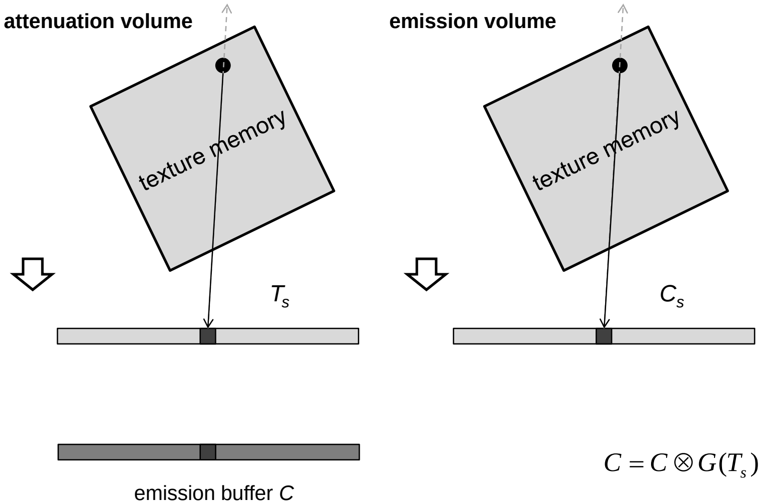
Backprojection: front-to-back



Scattering: Projection

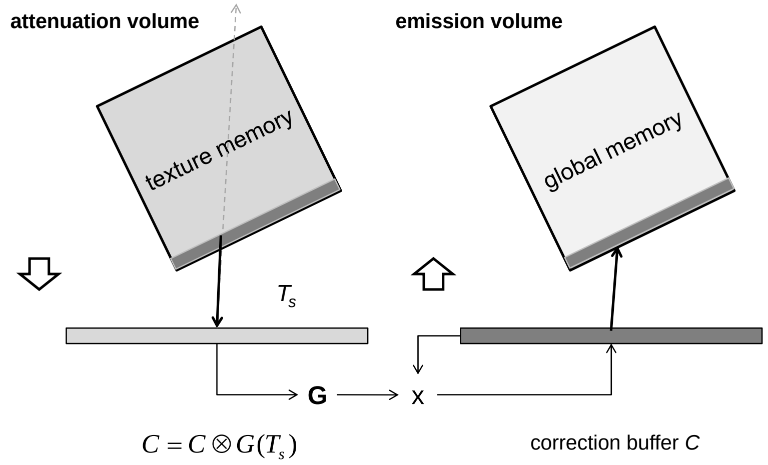


Scattering: Projection



$$C = C \otimes G(T_s) + C_s$$

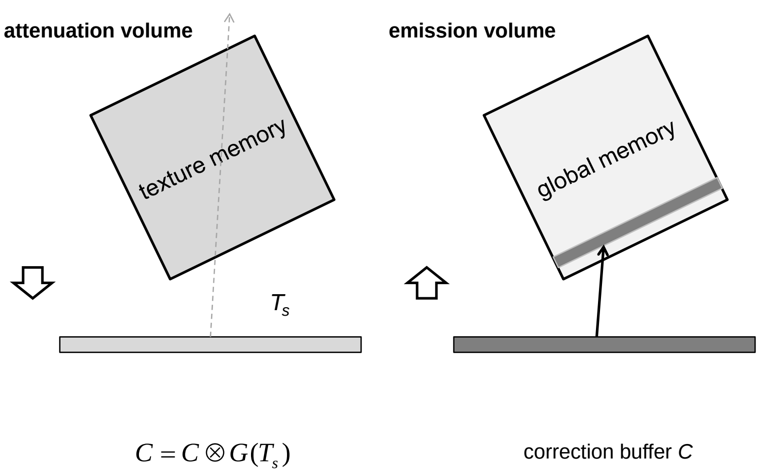
Scattering: Backprojection



$$C = C \otimes G(T_s)$$

slice by slice update

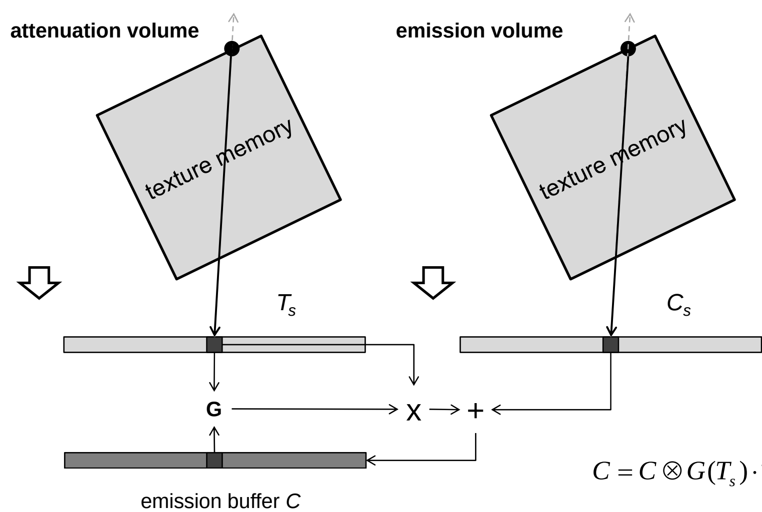
Scattering: Backprojection



$$C = C \otimes G(T_s)$$

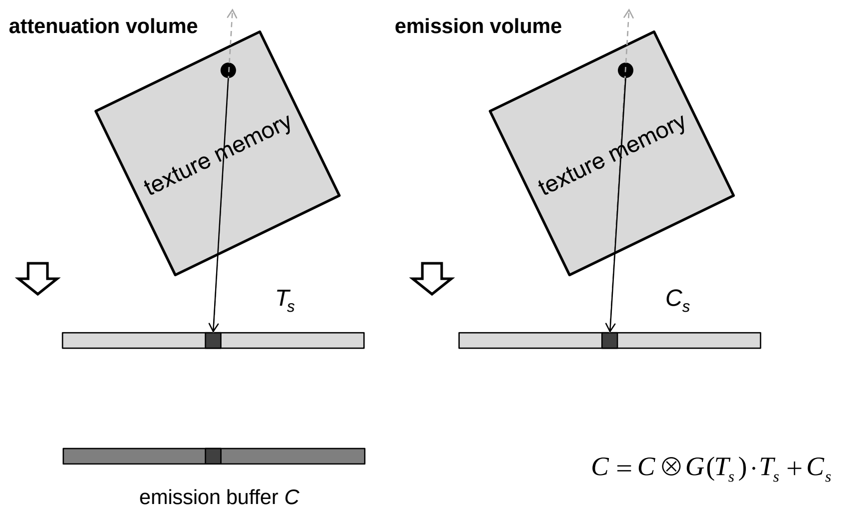
slice by slice update

Attenuation + Scattering: Projection

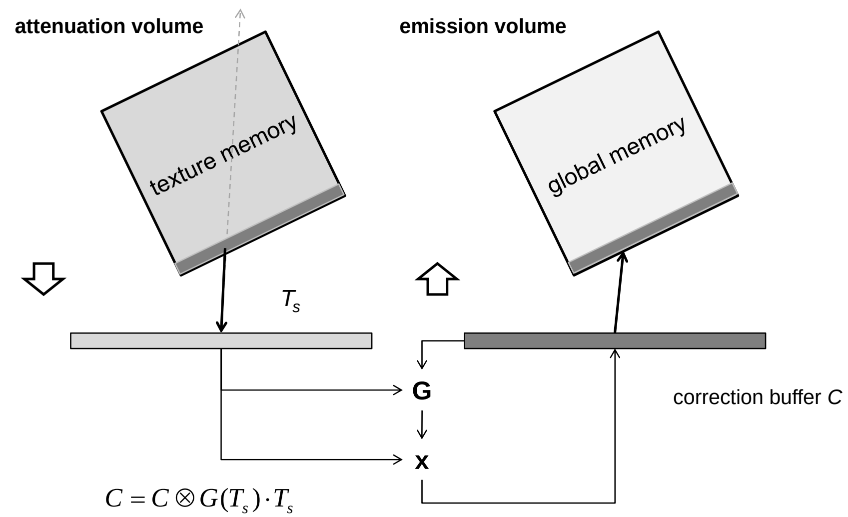


$$C = C \otimes G(T_s) \cdot T_s + C_s$$

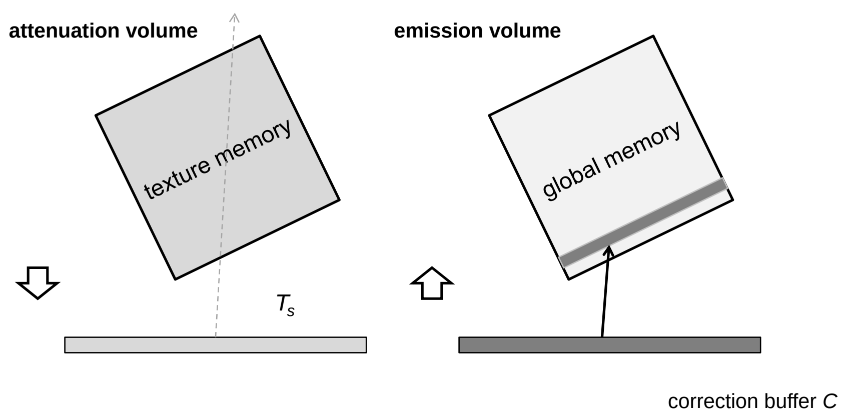
Attenuation + Scattering: Projection



Attenuation + Scattering: Backproj.



Attenuation + Scattering: Backproj.



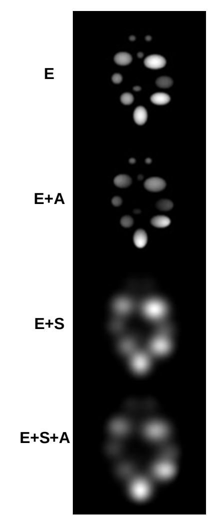
$$C = C \otimes G(T_s) \cdot T_s$$

Simulation Results

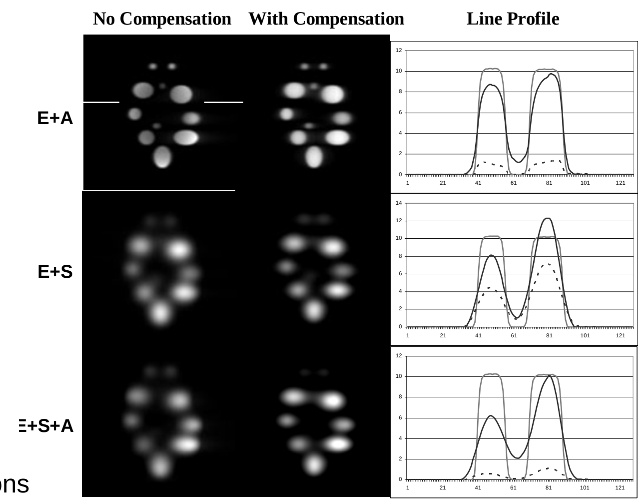
Scattering creates substantially more blur

Attenuation weakens the projections of emissions traversing highly attenuating material

- E: emission only
- A: attenuation correction
- S: scattering



Reconstruction Results



10 iterations

Generalization to Iterative Pipeline

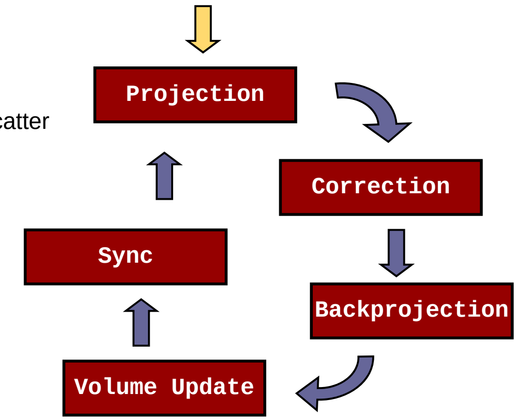
Kernel selection depends on algorithms

Projection/Backprojection
- attenuation only
- attenuation + emission
- attenuation + emission + scatter

Correction
- subtraction (algebraic)
- division (EM)

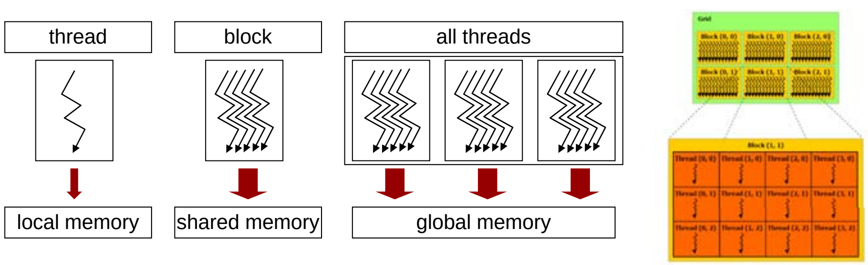
Update
- addition (algebraic)
- multiply (EM)

Sync

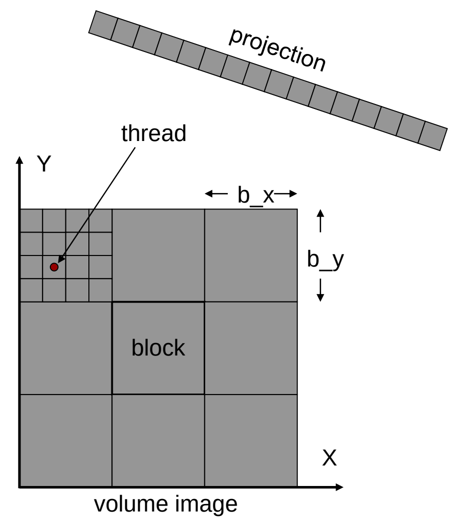


CUDA Optimization: Memory

	Global Memory	Shared Memory
Access	Read/Write	Read/Write
Location	Off-chip	On-chip
Scope	All threads	Threads in a block
Capacity	Large (hundreds of MBs)	Small (16KB)
Subject to coalescing	Yes	No
Latency	400-600 cycles	Single cycle



Example: Backprojection



volume: global memory

projection: global or texture memory

Host Code:

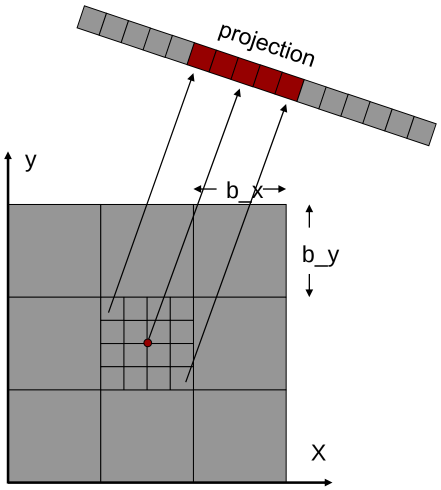
```
dim3 blockDim(b_x, b_y);
dim3 gridDim(v_x/blockDim.x, v_y/blockDim.y)
Recon<<<gridDim, blockDim>>>(…)
```

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];
```

Optimize via Shared Memory, I

SPIE Medical Imaging



volume image

Load projection data to be sampled into shared memory before backprojection to reduce global memory read

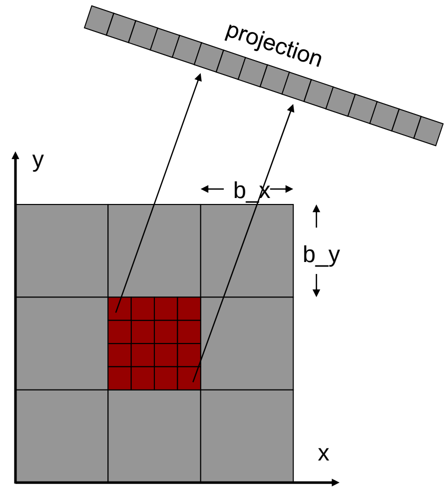
volume: global memory

projection: global memory → shared memory

```
__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

SPIE Medical Imaging



volume image

Load sub-volume data into shared memory before backprojection to reduce global memory read/write.

volume: global memory → shared memory

projection: global memory or texture memory

```
__shared__ float sharedMem[];

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

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