CSE 591: GPU Programming

Case Study: GPU-Accelerated Cone-Beam CT

Klaus Mueller

Computer Science Department

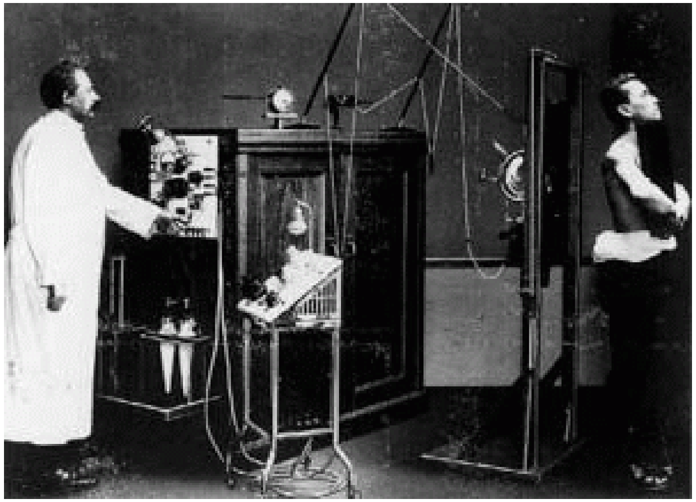
Stony Brook University

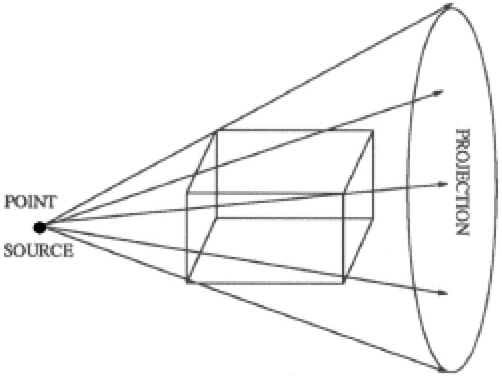
History: X-Rays

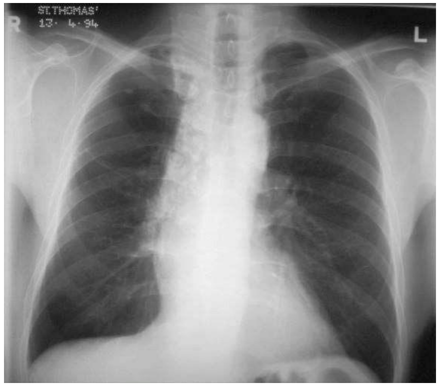
Wilhelm Conrad Röntgen

- 8 November 1895: discovers X-rays.
- 22 November 1895: X-rays Mrs. Röntgen’s hand.
- 1901: receives first Nobel Prize in physics

An early X-ray imaging system:





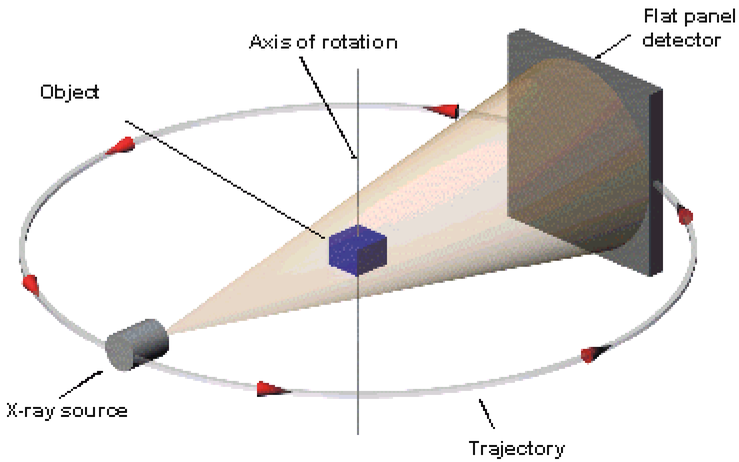


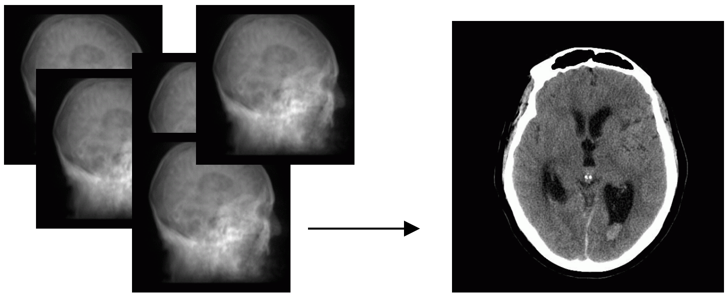
Note: so far all we can see is a projection across the patient:

History: Computed Tomography

The breakthrough:

- acquiring many projections around the object enables the reconstruction of the 3D object (or a cross-sectional 2D slice)



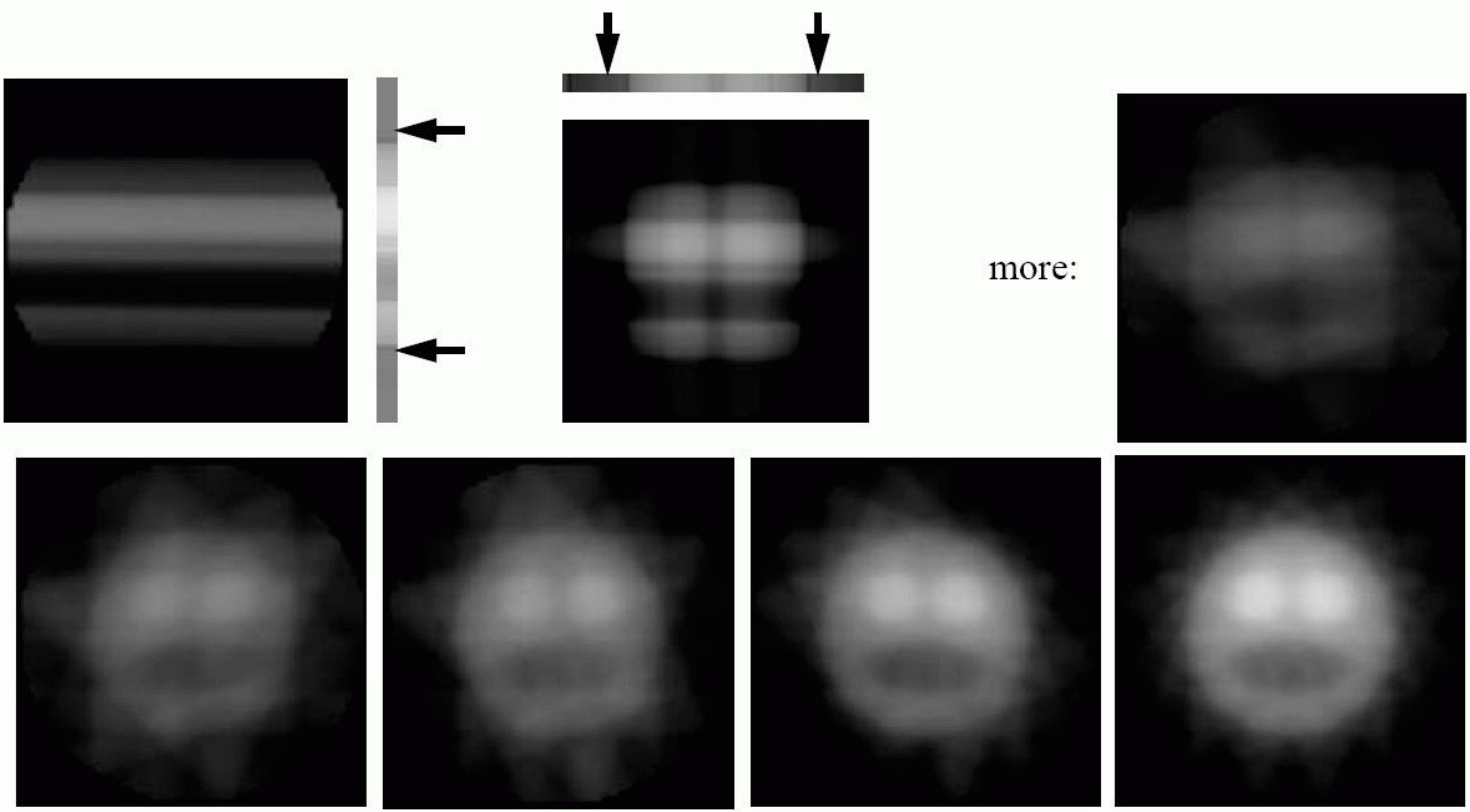


CT reconstruction pioneers:

- 1917: Johann Radon establishes the mathematical framework for tomography, now called the Radon transform.
- 1963: Allan Cormack publishes mathematical analysis of tomographic image reconstruction, unaware of Radon's work.
- 1972: Godfrey Hounsfield develops first CT system, unaware of either Radon or Cormack's work, develops his own reconstruction method.
- 1979 Hounsfield and Cormack receive the Nobel Prize in Physiology or Medicine.

Radon Cormack Hounsfield

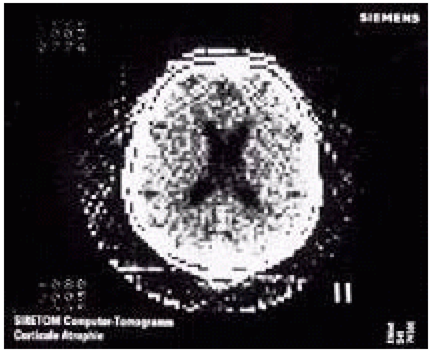
Computed Tomography: Concept



Computed Tomography: Past and Present

Image from the Siemens Siretom CT scanner, ca. 1975

- 128x128 matrix.



Modern CT image acquired with a Siemens scanner

- 512x512 matrix

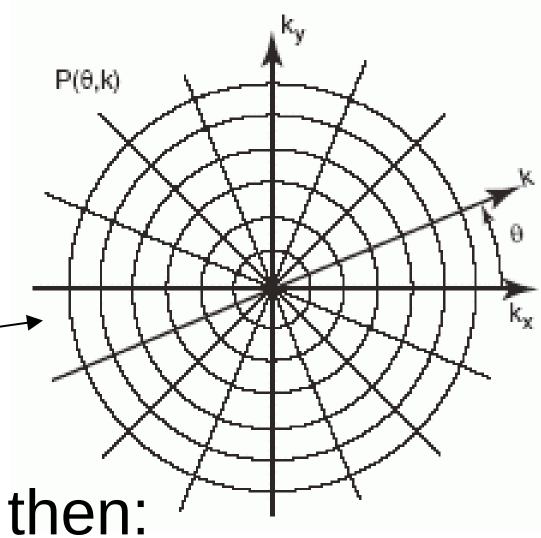
Slice Viewer

The Fourier Slice Theorem

To understand the blurring we need more theory → the *Fourier Slice Theorem* or *Central Slice Theorem*

- it states that the Fourier transform $P(\theta,k)$ of a projection $p(r,\theta)$ is a line across the origin of the Fourier transform $F(k_x,k_y)$ of function $f(x,y)$

polar grid

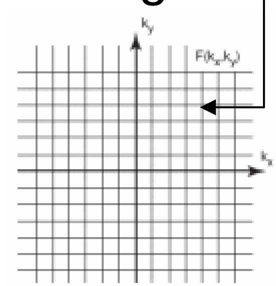


A possible reconstruction procedure would then:

- calculate the 1D FT of all projections $p(r_m,\theta_m)$, which gives rise to $F(k_x,k_y)$ sampled on a polar grid (see figure)
- resample the polar grid into a cartesian grid (using interpolation)
- perform inverse 2D FT to obtain the desired $f(x,y)$ on a cartesian grid

However, there are two important observations:

- interpolation in the frequency domain leads to artifacts
- at the FT periphery the spectrum is only sparsely sampled



Filtered Backprojection: Concept

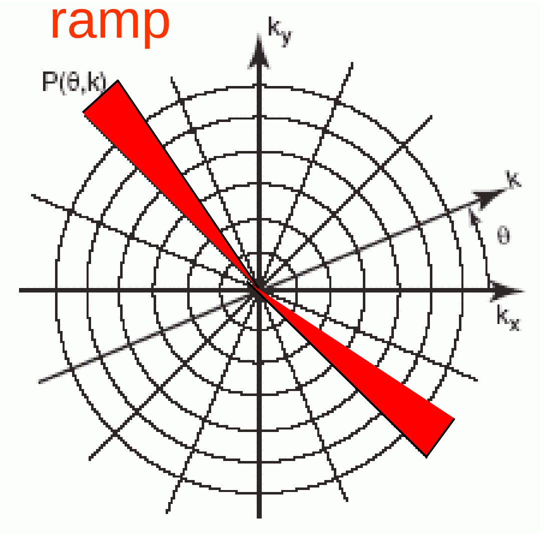
To account for the implications of these two observations, we modify the reconstruction procedure as follows:

- filter the projections to compensate for the blurring
- perform the interpolation in the spatial domain via backprojection

  → hence the name *Filtered Backprojection*

Filtering -- what follows is a more practical explanation (for formal proof see the book):

- we need a way to equalize the contributions of all frequencies in the FT's polar grid
- this can be done by multiplying each $P(\theta,k)$ by a ramp function → this way the magnitudes of the existing higher-frequency samples in each projection are scaled up to compensate for their lower amount
- the ramp is the appropriate scaling function since the sample density decreases linearly towards the FT's periphery



Filtered Backprojection: Equation and Result

1D Fourier transform of $p(r,\theta)$ → $P(k,\theta)$

ramp-filtering

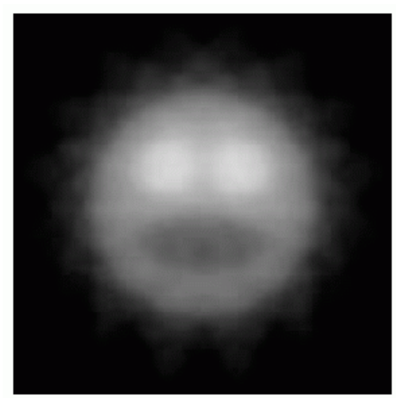
$$f(x,y) = \int_0^{\pi} \left(\int_{-\infty}^{\infty} P(k,\theta) \cdot |k| \cdot e^{i2\pi kr} dk \right) d\theta$$

inverse 1D Fourier transform → $p(r,\theta)$

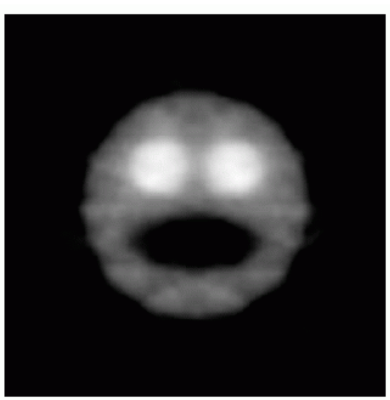
backprojection for all angles

Recall the previous (blurred) backprojection illustration

- now using the filtered projections:



not filtered



filtered

Backprojection: Practical Considerations

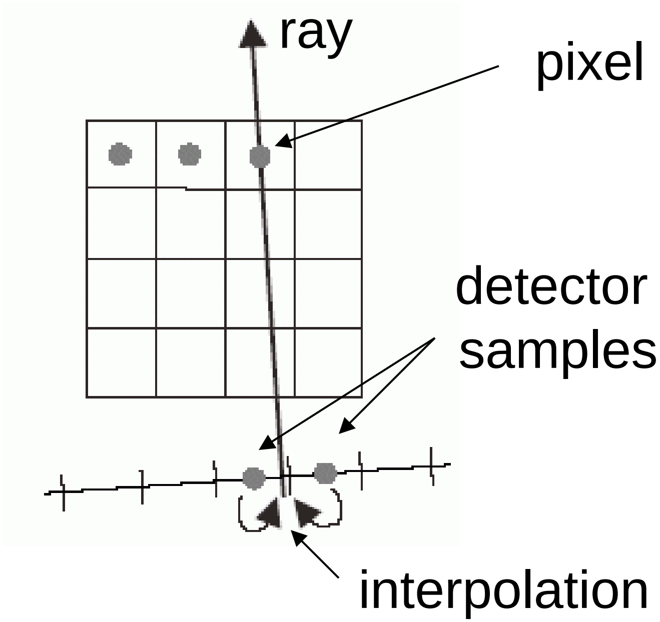
A few issues remain for practical use of this theory:

- we only have a finite set of *M* projections and a discrete array of *N* pixels (x_i, y_j)

$$b(x_i, y_j) = B\{p(r_n, \theta_m)\} = \sum_{m=1}^{M} p(x_i \cdot \cos\theta_m + y_j \cdot \sin\theta_m, \theta_m)$$

- to reconstruct a pixel (x_i, y_j) there may not be a ray $p(r_n, \theta_n)$ (detector sample) in the projection set

  → this requires interpolation (usually linear interpolation is used)



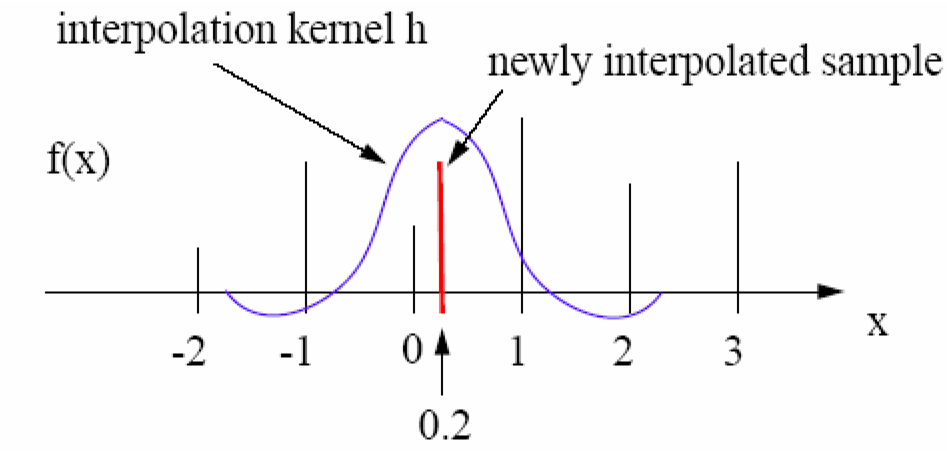
- the reconstructions obtained with the simple backprojection appear blurred (see previous slides)

Interpolation

Often we want to estimate the formerly continuous function from the discretized function represented by the matrix of sample points

This is done via *interpolation*

Concept:

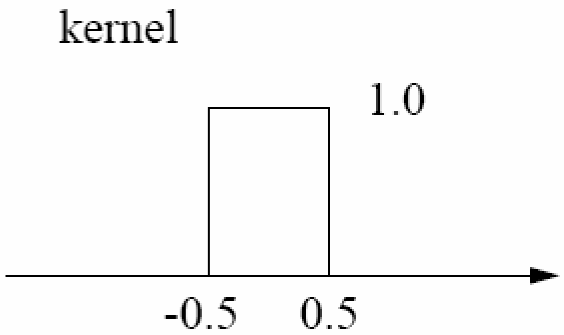


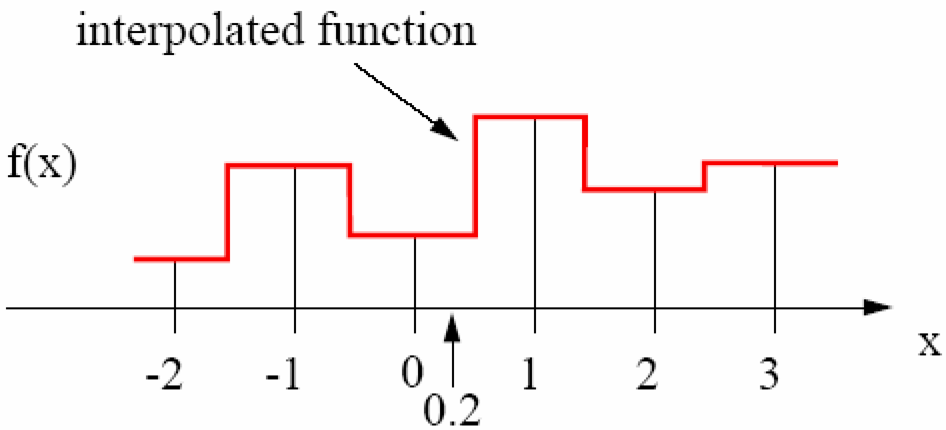
- center the interpolation kernel (filter) *h* at the sample position and superimpose it onto the grid
- multiply the values of the grid samples with the kernel value at the superimposed position
- add all the products → this gives the value of the newly interpolated sample
- in the shown case:

$$f(0.2) = h(-0.2)\, f(0) + h(-1.2)\, f(-1) + h(0.8)\, f(1) + h(1.8)\, f(2)$$

Interpolation Kernels (1)

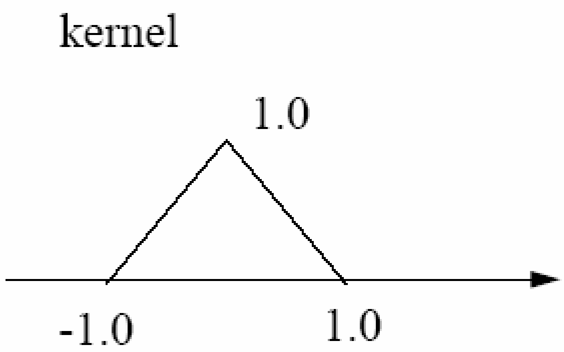
- Nearest Neighbor:

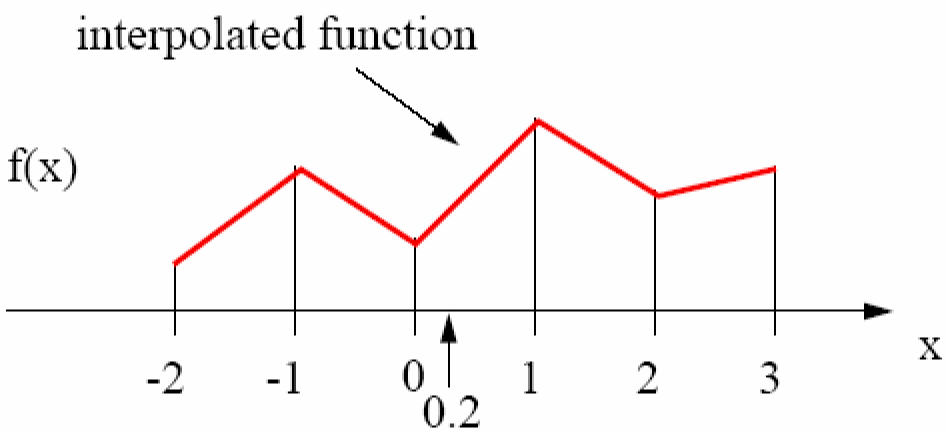




- simply pick the value of the nearest grid point: $f(0.2) = f(trunc(0.2+0.5) = f(round(0.2))$

- Linear filter:

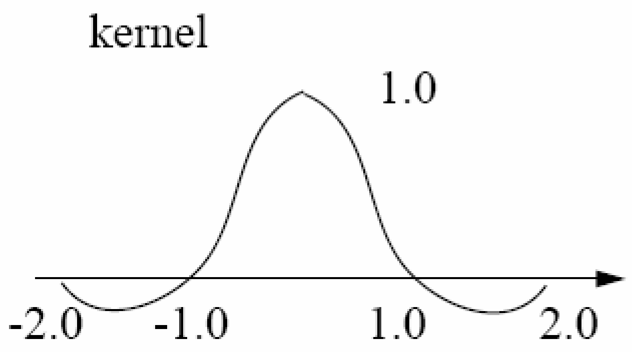


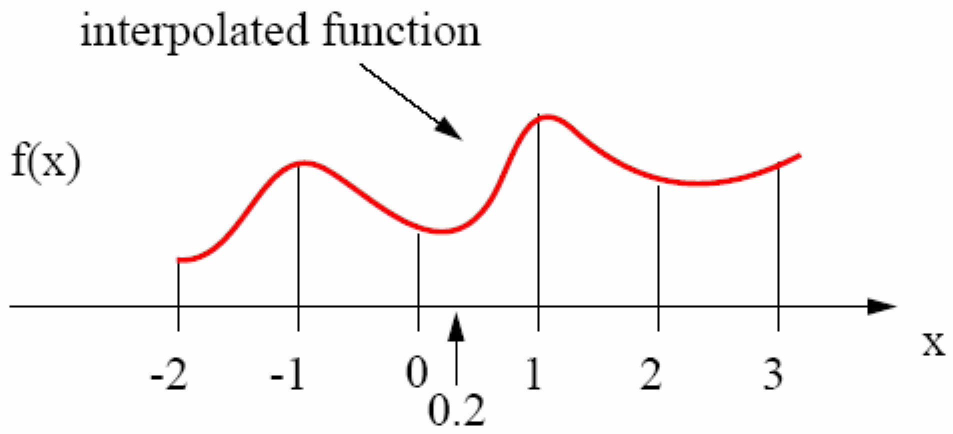


- use a linear combination of the two neighboring grid values: $f(0.2) = 0.2 \cdot f(1) + 0.8 \cdot f(0)$

Interpolation Kernels (2)

- Cubic filter:





An additional popular filter is the Gaussian function

Discussion:

- nearest neighbor is fastest to compute (just one add), gives sharp edges, but sometimes jagged lines
- linear interpolation takes 2 mults and 1 add and gives a piecewise smooth function
- cubic filter takes 4 mults and 3 adds, but gives an overall smooth interpolated function
- linear interpolation is most popular in many application

Interpolation in Higher Dimensions

- All interpolation kernels shown here are separable

  $h(x, y) = h(x) \cdot h(y)$ and $h(x, y, z) = h(x) \cdot h(y) \cdot h(z)$

- Linear interpolation

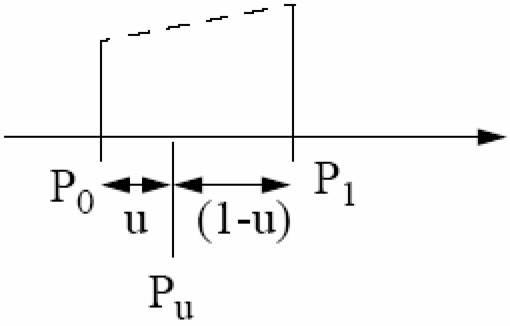
  assume: grid distance = 1.0

  P_u is the location of the sample value

  P_0 and P_1 are neighboring grid points

  then: $u = P_u - P_0$

  $f(x) = f(P_u) = (1 - u) \cdot f(P_0) + u \cdot f(P_1)$

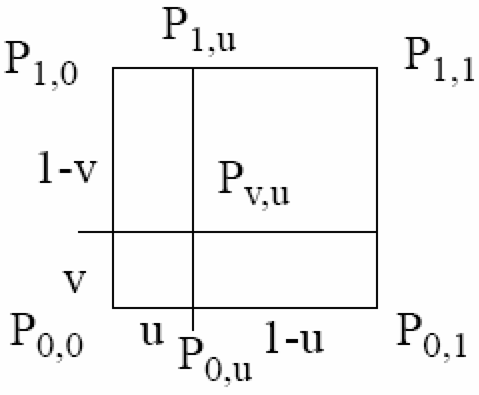


- Bilinear interpolation

  $f(P_{0,u}) = (1 - u) \cdot f(P_{0,0}) + u \cdot f(P_{0,1})$

  $f(P_{1,u}) = (1 - u) \cdot f(P_{1,0}) + u \cdot f(P_{1,1})$

  $f(P_{v,u}) = (1 - v) \cdot f(P_{0,u}) + v \cdot f(P_{1,u})$

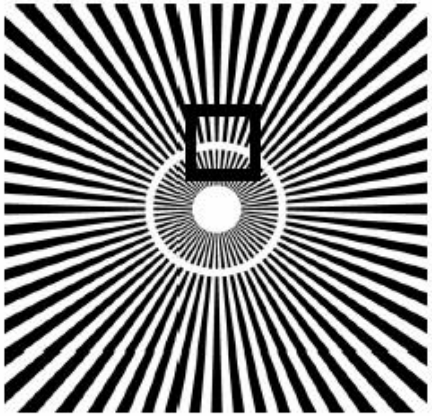


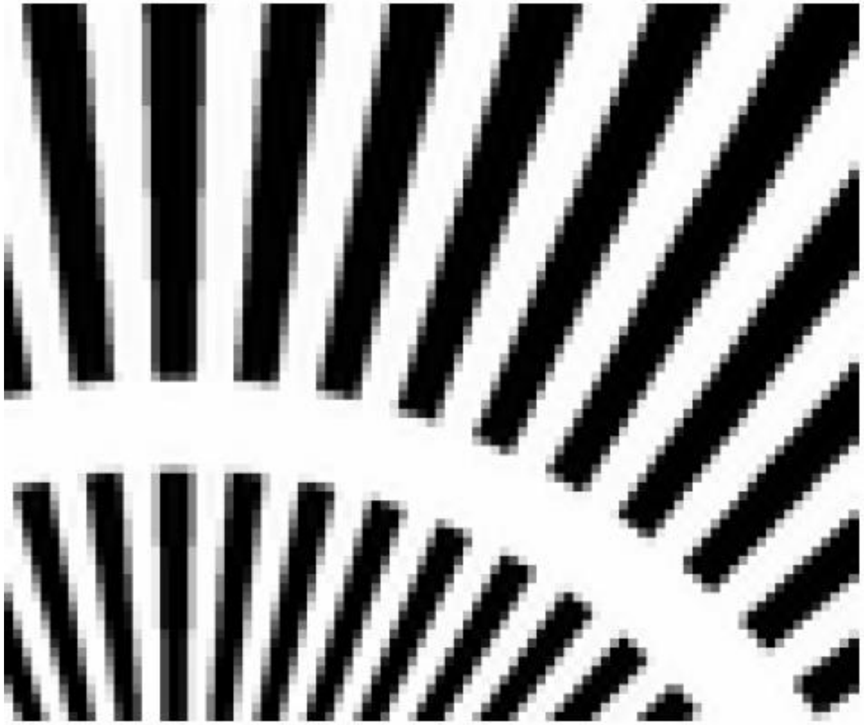
$\rightarrow f(x, y) = f(P_{v,u}) = (1-v)(1-u) f(P_{0,0}) + (1-v) u f(P_{0,1}) + v (1-u) f(P_{1,0}) + v u f(P_{1,1})$

Interpolation Quality

Example:

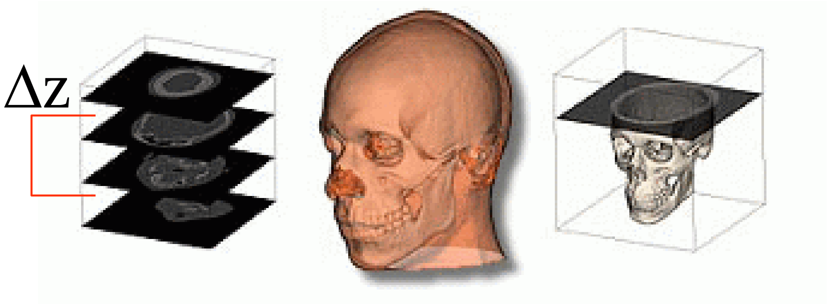
- resampling of a portion of the star image onto a high resolution grid
- magnification factor ~20





Imaging in Three Dimensions: Spiral CT

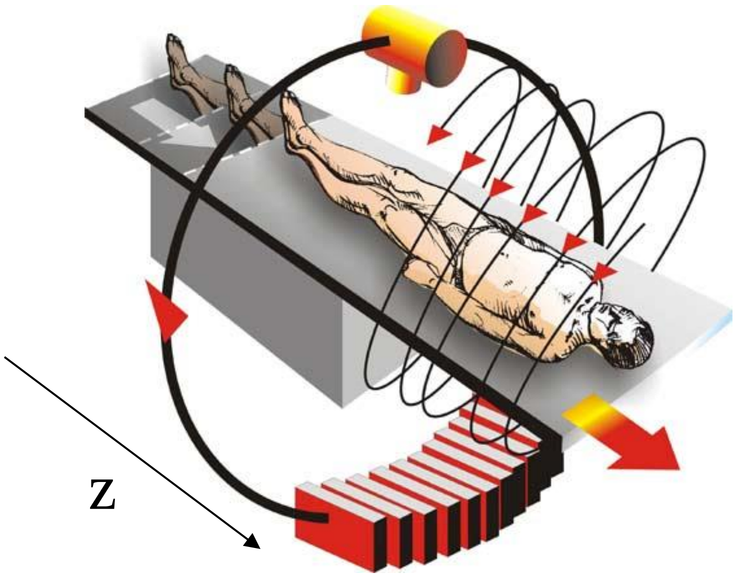
Sequential CT



- advance table with patient after each slice acquisition has been completed
- stop-motion is time consuming and also shakes the patient
- the *effective thickness* of a slice, Δz, is equivalent to the beam width Δs in 2D
- similarly: we must acquire 2 slices per Δz to combat aliasing

Spiral (helical) CT

- table translates as tube rotates around the patient
- very popular technique
- fast and continuous
- *table feed* (*TF*) = axial translation per tube rotation
- *pitch* = TF / Δz



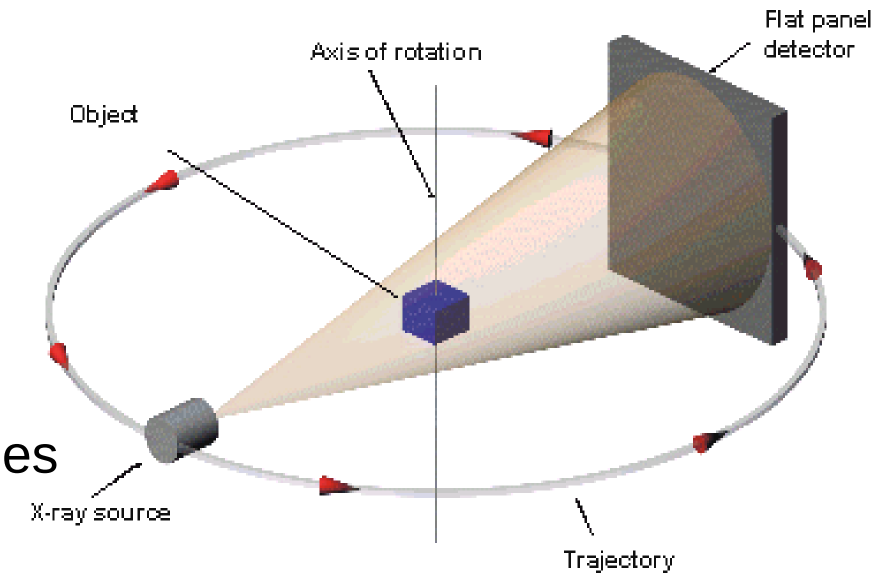
3D Reconstruction From Cone-Beam Data

Most direct 3D scanning modality

- uses a 2D detector
- requires only one rotation around the patient to obtain all data (within the limits of the cone angle)
- reconstruction formula can be derived in similar ways than the fan beam equation (uses various types of weightings as well)
- a popular equation is that by Feldkamp-Davis-Kress (FDK)
- backprojection proceeds along cone-beam rays

Advantages

- potentially very fast (since only one rotation)
- often used for 3D angiography



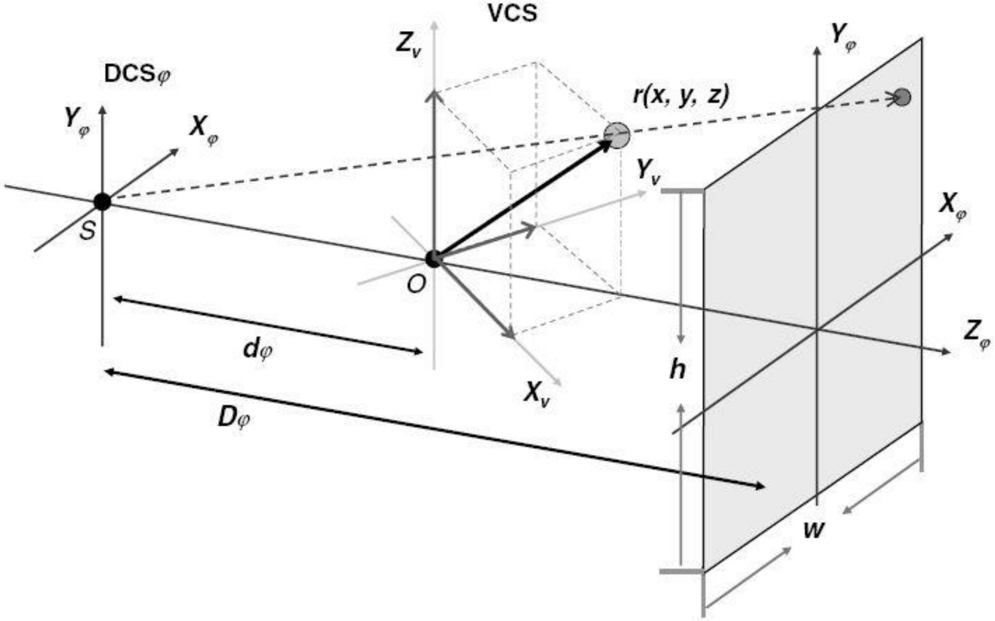
Downsides

- sampling problems at the extremities
- reconstruction sampling rate varies along z

Cone-Beam Reconstruction Geometry

Per voxel, for each angle

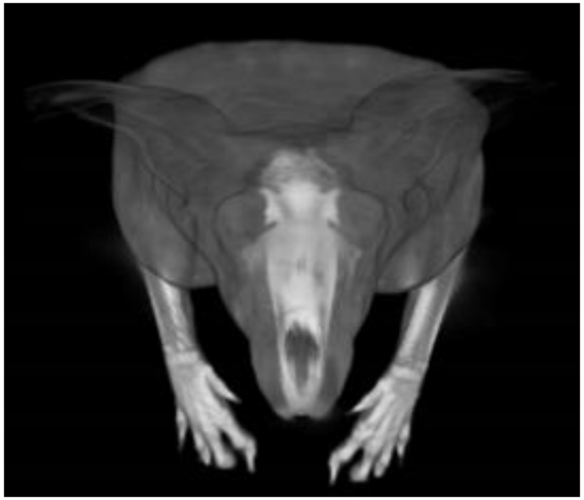
- determine ray from voxel to source
- intersect with detector plane
- determine detector pixels
- interpolate these
- do depth weighting
- add contribution to voxel



$$v_\phi(\boldsymbol{r}) = \frac{d_\phi^2}{(d_\phi + \boldsymbol{r} \cdot z_\phi)^2} \cdot \mathrm{Int}(P_\phi(X_\phi(\boldsymbol{r}), Y_\phi(\boldsymbol{r}))),$$

$$X(\boldsymbol{r}) = \frac{\boldsymbol{r} \cdot x_\phi}{d_\phi + \boldsymbol{r} \cdot z_\phi} D_\phi, \qquad Y(\boldsymbol{r}) = \frac{\boldsymbol{r} \cdot y_\phi}{d_\phi + \boldsymbol{r} \cdot z_\phi} D_\phi.$$

Rabbit CT



Benchmarking framework:

- developed By Rohkohl et al.
- FDK backprojection algorithm
- 496 projections of a rabbit
- 1248 X 960 pixels each

Advantages:

- enables true comparisons
- embeds the system matrix already
- ‘just’ accelerate the backprojection
- measures timings
- measures reconstruction errors

Leaderboard

- benchmark new code
- 256^3, 512^3, 1024^3 volume reconstructions

Rabbit CT Leaderboard (May 14, 2013)

Ranking

Problem size: 256 | **512** | 1024

Rank	Algorithm Description	q_{rmse} * [HU]	Error Hist.†	PSNR‡ [dB]	Time+ [s]	Performance*
1	**Thumper** Submitter: Timo Zinßer Institution: Siemens AG RabbitCT dataset version: 2 Thumper is a CUDA-based back-projection implementation.	0.16		88.30	0.8	100.00%
2	**CERA on GTX 680** Submitter: Matthias Elter Institution: Siemens AG RabbitCT dataset version: 2 The CUDA 5.0 based CERA back-projection implementation (extended with CUDA streams) running on a NVIDIA GTX 680.	0.18		87.38	0.9	87.94%
3	**RapidRabbit** Submitter: Eric Papenhausen, Ziyi Zheng Institution: Stony Brook University RabbitCT dataset version: 1 A CUDA 3.0 based back projection implementation using a variety of optimization techniques.	2.84		63.17	3.0	26.68%
4	**CERA on GTX 670** Submitter: Matthias Elter Institution: Siemens AG RabbitCT dataset version: 1 A CUDA 4.2 based back-projection implementation running on a NVIDIA GTX 670 GPU.	2.84		63.17	3.4	23.20%
5	**CERA on GTX 570** Submitter: Matthias Elter Institution: Siemens AG RabbitCT dataset version: 1					

Rabbit CT Leaderboard (May 14, 2013)

Ranking

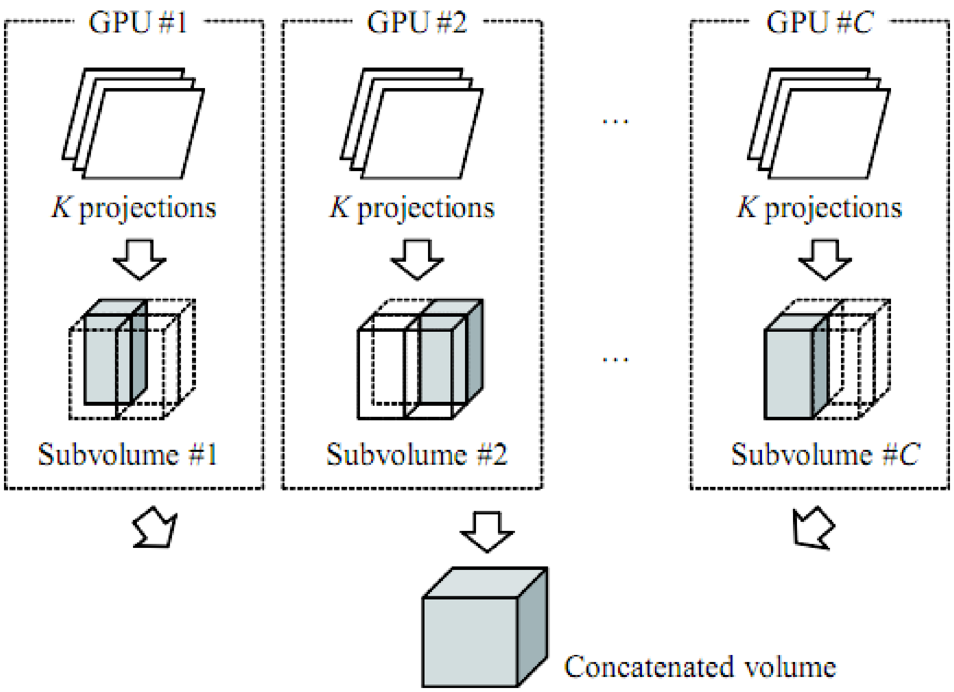
Problem size: 256 | **512** | 1024

Rank	Algorithm Description	q_{rmse}* [HU]	Error Hist.†	PSNR‡ [dB]	Time+ [s]	Performance*
1	**Thumper** Submitter: Timo Zinßer Institution: Siemens AG RabbitCT dataset version: 2 Thumper is a CUDA-based back-projection implementation.	0.16		88.30	0.8	100.00%
2	**CERA on GTX 680** Submitter: Matthias Elter Institution: Siemens AG RabbitCT dataset version: 2 The CUDA 5.0 based CERA back-projection implementation (extended with CUDA streams) running on a NVIDIA GTX 680.	0.18		87.38	0.9	87.94%
3	**RapidRabbit** Submitter: Eric Papenhausen, Ziyi Zheng Institution: Stony Brook University RabbitCT dataset version: 1 A CUDA 3.0 based back projection implementation using a variety of optimization techniques.	2.84		63.17	3.0	26.68%
4	**CERA on GTX 670** Submitter: Matthias Elter Institution: Siemens AG RabbitCT dataset version: 1 A CUDA 4.2 based back-projection implementation running on a NVIDIA GTX 670 GPU.	2.84		63.17	3.4	23.20%
5	**CERA on GTX 570** Submitter: Matthias Elter Institution: Siemens AG					

Rapid Rabbit (June, 2011)

Approach:

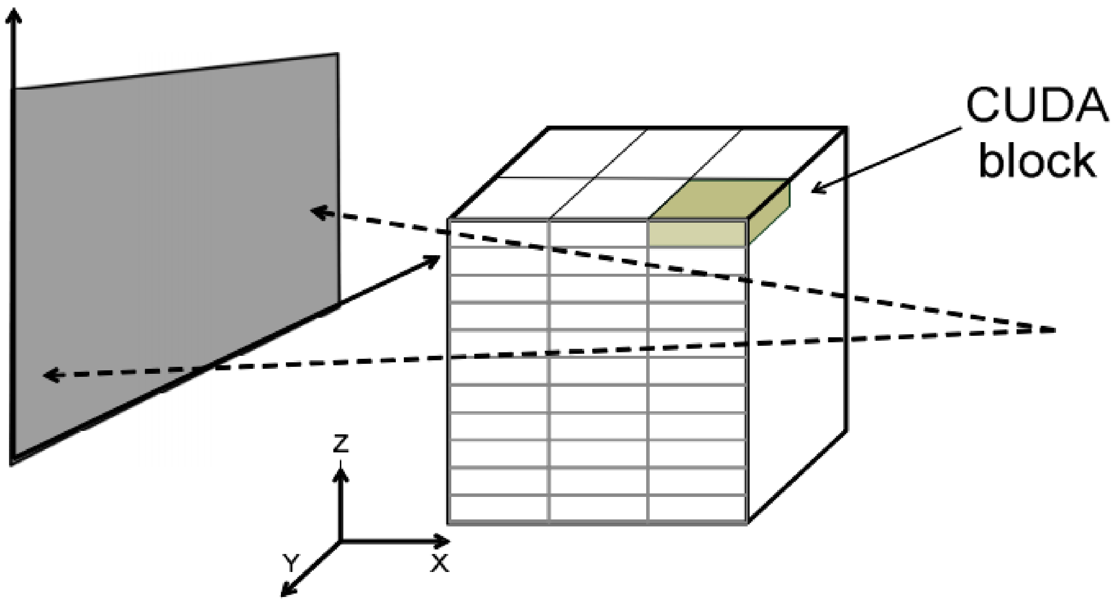
- voxel parallelism
- each thread block computes a subset of the volume



Setup

Approach:

- each thread computes an array of voxels



Thread Block Dimension: 16 x 16 x 4

Naïve Implementation

Approach:

- volume, projection image, and projection matrix stored in global memory
- explicit bi-linear interpolation

Naïve Implementation

```
row = blockIdx.y * blockDim.y + threadIdx.y
col = blockIdx.x * blockDim.x + threadIdx.x

FOR k = 0 to L

  x = O_L +col * R_L
  y = O_L + row * R_L
  z = O_L + k * R_L

  w = A[2] * x + A[5] * y + A[8] * z + A[11]
  u = (A[0] * x + A[3] * y + A[6] * z +A[9]) / w
  v = (A[1] * x + A[4] * y + A[7] * z + A[10]) / w

  result = interpolate (u, v)
  result = result / w^2

  f_L[k * L^2 + row * L + col] += result
END
```

Naïve Results

256^3:

- Total: 7.77 s
- Mean: 15.66 ms
- Error: 8.04 HU^2
- GUPS: 0.99

512^3:

- Total: 42.6 s
- Mean: 86.06 ms
- Error: 8.04 HU^2
- GUPS: 1.45

Floating Point to Memory Access Ratio is 4:1

Explicit Bi-linear interpolation = Low Occupancy

ASIC

Approach:

- Projection Image stored in Texture Memory
- Projection Matrix stored in Constant Memory
- ASIC = Fast 2D texture interpolation

ASIC

```
texture<float, 2> texRef
__constant__ float A[12]

row = blockIdx.y * blockDim.y + threadIdx.y
col = blockIdx.x * blockDim.x + threadIdx.x
FOR k = 0 to L
  result = f_L[k * L^2 + row * L + col]

  x = O_L +col * R_L
  y = O_L + row * R_L
  z = O_L + k * R_L

  w = A[2] * x + A[5] * y + A[8] * z + A[11]
  u = (A[0] * x + A[3] * y + A[6] * z +A[9]) / w
  v = (A[1] * x + A[4] * y + A[7] * z + A[10]) / w
  result += tex2D ( texRef, (u + 0.5), (v + 0.5)) / w^2

  f_L[k * L^2 + row * L + col] = result
END
```

ASIC Results

256^3:

- Total: 3.53 s
- Mean: 7.13 ms
- Error: 8.07 HU^2
- GUPS: 2.19

512^3:

- Total: 10.8 s
- Mean: 21.82 ms
- Error: 8.07 HU^2
- GUPS: 5.73

Lower bound of 2 global memory accesses

All threads access same constant memory location

ASIC Interpolation = Fewer Registers = Higher Occupancy

Fully Optimized

Approach:

- Similar to ASIC, but increased thread granularity
- Each thread operates on same voxels
- 4 projections per kernel invocation

Fully Optimized

```
texture<float, 2> tRef, tRef2, tRef3, tRef4
__constant__ float A[48]

row = blockIdx.y * blockDim.y + threadIdx.y
col = blockIdx.x * blockDim.x + threadIdx.x
FOR k = 0 to L
  result = f_L[k * L^2 + row * L + col]
                          . . .
  // mapping voxel (x,y,z) to projection 1 and backproject
  w = A[2] * x + A[5] * y + A[8] * z + A[11]
  u = (A[0] * x + A[3] * y + A[6] * z +A[9]) / w
  v = (A[1] * x + A[4] * y + A[7] * z + A[10]) / w
  result += tex2D ( tRef, (u + 0.5), (v + 0.5)) / w^2

  //repeat for projection 2 with A[12-23] and tRef2
  //repeat for projection 3 with A[24-35] and tRef3
  // repeat for projection 4 with A[36-47] and tRef4

  f_L[k * L^2 + row * L + col] = result
END
```

Fully Optimized Results

256^3:

- Total: 2.71 s
- Mean: 5.47 ms
- Error: 8.07 HU^2
- GUPS: 2.86

512^3:

- Total: 6.07 s
- Mean: 12.25 ms
- Error: 8.07 HU^2
- GUPS: 10.2

Still Lower bound of 2 global memory accesses per iteration

Overall memory accesses decreased by a factor of 4

Greater than 4 projections led to degraded performance

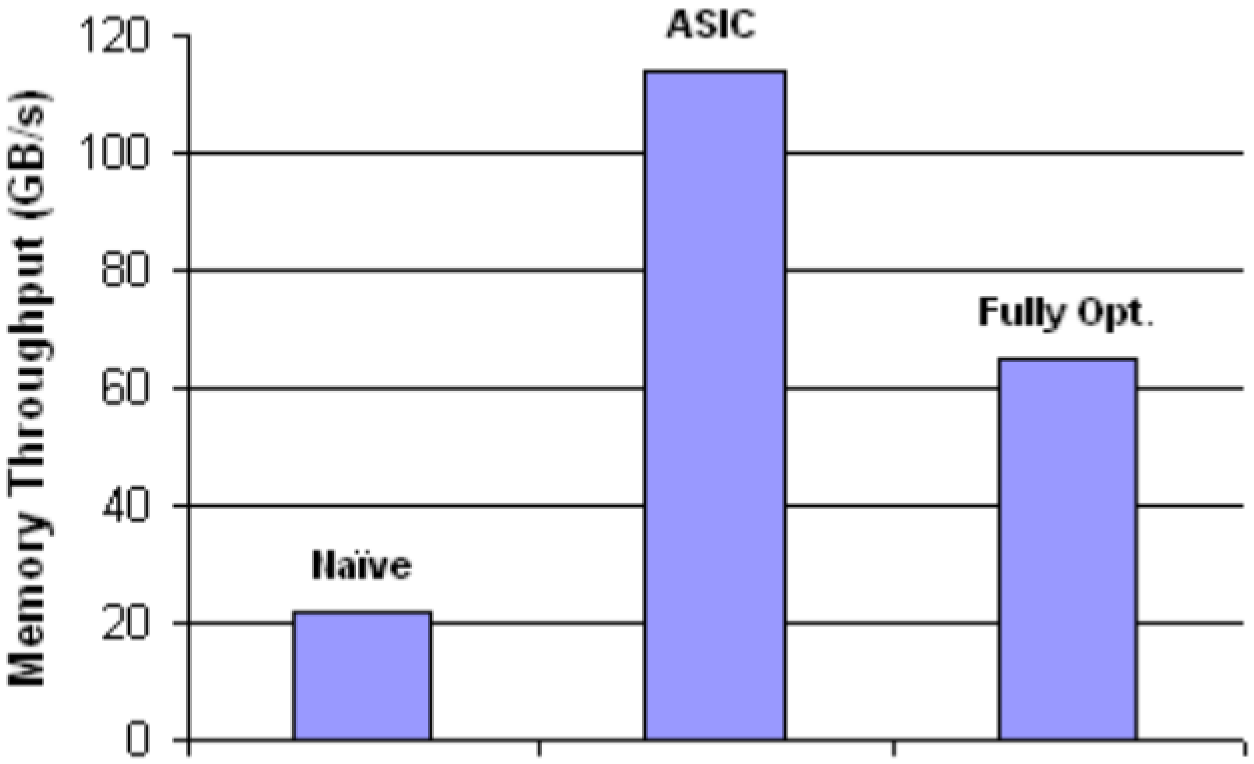
Results

Configuration	Volume	Total	Mean	Error	Speed-up	GUPS
Naïve	256^3	7.77s	15.66ms	$8.04HU^2$	N/A	0.99
ASIC	256^3	3.53s	7.13ms	$8.07HU^2$	2.19	2.19
Fully Opt.	256^3	2.71s	5.47ms	$8.07HU^2$	1.3	2.86
Naïve	512^3	42.6s	86.08ms	$8.04HU^2$	N/A	1.45
ASIC	512^3	10.8s	21.82ms	$8.07HU^2$	3.9	5.73
Fully Opt.	512^3	6.07s	12.25ms	$8.07HU^2$	1.78	10.2

Results

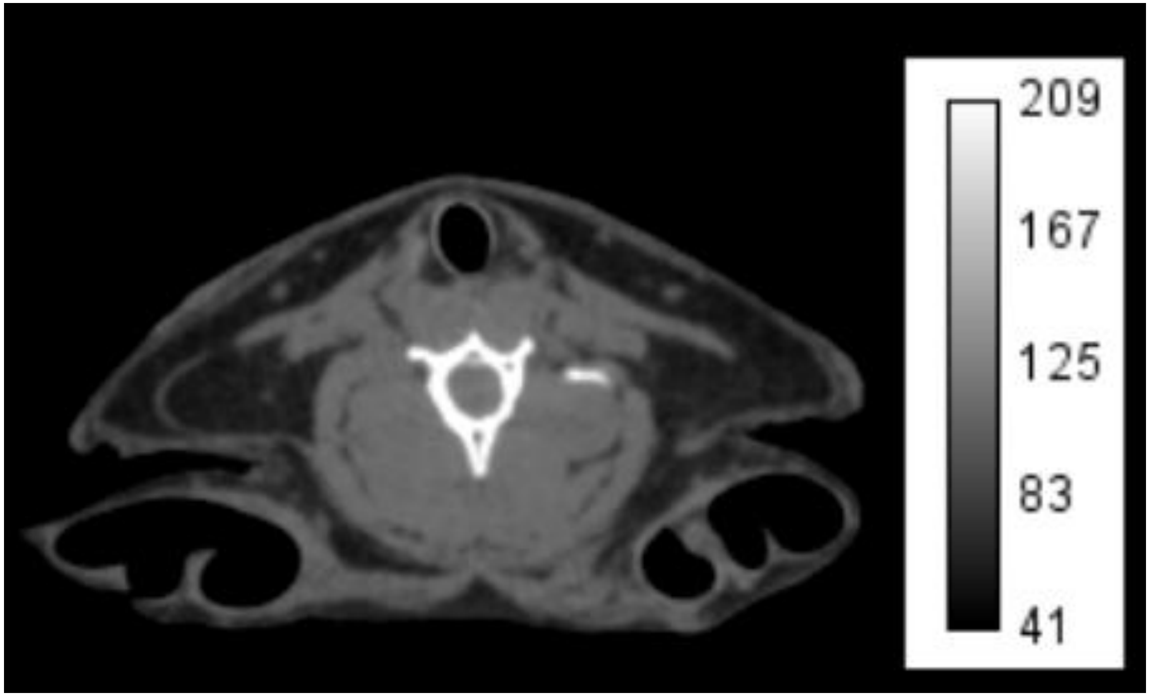
Configuration	Volume	Total	Mean	Error	Speed-up
Best Known	256^3	3.843s	7.75ms	$8.07HU^2$	N/A
Fully Opt.	256^3	2.713s	5.47ms	$8.07HU^2$	1.4
Best Known	512^3	13.94s	28.11ms	$8.07HU^2$	N/A
Fully Opt.	512^3	6.076s	12.25ms	$8.07HU^2$	2.29

Results



ASIC
Fully Opt.
Naïve
Memory Throughput (GB/s)
120
100
80
60
40
20
0

Results



Click for more paper

Eric Papenhausen, Ziyi Zheng, and Klaus Mueller. "GPU-accelerated back-projection revisited: squeezing performance by careful tuning." Workshop on High Performance Image Reconstruction (HPIR). 2011.

Optimizations

Successful:

- Pre-fetching
- Page-locked memory

Unsuccessful:

- Loop Unrolling
- Fast Math

Common Sense Optimizations

New Rabbit on the Block: Thumper (March 2013)

Improves upon Rapid Rabbit

Initial code (kernel A)

```
compute position of first voxel

for I input projections do
    compute homogeneous detector coordinates q[i] of first voxel
end

for K consecutive voxels along the z-axis do
    zero-initialize sum s of weighted back-projected values
    for I input projections do
        dehomogenize detector coordinates q[i]
        compute back-projected value by texture fetching
        update sum s of weighted back-projected values
        update homogeneous detector coordinates q[i]
    end
    update volume at current voxel with computed sum s
    (optionally) synchronize threads in thread block
end
```

Click for more info and paper

Timo Zinsser, and Benjamin Keck. "Systematic performance optimization of cone-beam back-projection on the Kepler architecture." Proceedings of the 12th Fully Three-Dimensional Image Reconstruction in Radiology and Nuclear Medicine (2013): 225-228.

Analyze Bottlenecks

Step 1:

- reduce the voxel size from 0.5 mm to 10^{-6} mm
- as a result, all computed detector coordinates are virtually identical
- the hit rate of the texture cache rises to almost one hundred percent

Step 2:

- disable the texture fetching completely

Step 3:

- turn off the volume update
- this removes all memory accesses
- leaves only the arithmetic and control flow instructions

Caution

- do not allow the compiler to eliminate more code than intended
- these modifications also tend to reduce the register count
- so allocate a suitable amount of shared memory to retain the occupancy of the original kernel

Analyze Bottlenecks

In table:

- I(nstruction), M(emory), T(exture)

Test	Kernel	Sync	I	K	B_x	B_y	Occupancy	I | – | –	I |M| –	I |M| T	Time	GUPS
1	A	no	1	512	32	8	1.000	1091 ms	4710 ms	4707 ms	9141 ms	7.3
2	A	no	4	512	32	8	0.750	575 ms	1199 ms	1196 ms	7802 ms	8.5
3	A	yes	4	512	32	8	1.000	554 ms	1185 ms	1169 ms	2710 ms	24.6
4	A	no	4	8	32	8	0.750	685 ms	980 ms	1085 ms	1990 ms	33.4

Test 1:

- kernel A processes one projection at a time
- specified tile width B_x = 32 ensures that the volume updates are performed by fully coalesced memory transactions
- we see that the memory transfer takes much longer than the computation of the arithmetic instructions (I– vs. IM-)
- time is almost doubled by the cache misses of the texture fetching (IMT vs. Time)

Analyze Bottlenecks

In table:

- I(nstruction), M(emory), T(exture)

Test	Kernel	Sync	I	K	B_x	B_y	Occupancy	I | – | –	I |M| –	I |M| T	Time	GUPS
1	A	no	1	512	32	8	1.000	1091 ms	4710 ms	4707 ms	9141 ms	7.3
2	A	no	4	512	32	8	0.750	575 ms	1199 ms	1196 ms	7802 ms	8.5
3	A	yes	4	512	32	8	1.000	554 ms	1185 ms	1169 ms	2710 ms	24.6
4	A	no	4	8	32	8	0.750	685 ms	980 ms	1085 ms	1990 ms	33.4

Test 2:

- when we process four projections in one kernel, the memory transfer size is reduced considerably (IM-)
- the compute-only kernel also runs much faster, because the number of integer-based index computations is minimized as well (I--)
- however, the time penalty induced by the cache misses of the texture fetching remains very high(IMT vs. Time)

Analyze Bottlenecks

In table:

- I(nstruction), M(emory), T(exture)

Test	Kernel	Sync	I	K	B_x	B_y	Occupancy	I | – | –	I |M| –	I |M| T	Time	GUPS
1	A	no	1	512	32	8	1.000	1091 ms	4710 ms	4707 ms	9141 ms	7.3
2	A	no	4	512	32	8	0.750	575 ms	1199 ms	1196 ms	7802 ms	8.5
3	A	yes	4	512	32	8	1.000	554 ms	1185 ms	1169 ms	2710 ms	24.6
4	A	no	4	8	32	8	0.750	685 ms	980 ms	1085 ms	1990 ms	33.4

Test 3:

- we activate the optional synchronization
- this prevents the divergence of the threads in one thread block with respect to the loop over the voxels along the z-axis
- as a result, the texture fetching is accelerated considerably and the computation time is reduced by about 65% (IMT vs. Time)
- the configuration results in a total of 16 waves of thread blocks, which iterate through the volume along the z-axis one after another.

Analyze Bottlenecks

In table:

- I(nstruction), M(emory), T(exture)

Test	Kernel	Sync	I	K	B_x	B_y	Occupancy	I | – | –	I |M| –	I |M| T	Time	GUPS
1	A	no	1	512	32	8	1.000	1091 ms	4710 ms	4707 ms	9141 ms	7.3
2	A	no	4	512	32	8	0.750	575 ms	1199 ms	1196 ms	7802 ms	8.5
3	A	yes	4	512	32	8	1.000	554 ms	1185 ms	1169 ms	2710 ms	24.6
4	A	no	4	8	32	8	0.750	685 ms	980 ms	1085 ms	1990 ms	33.4

Test 4:

- a kernel processes only 8 voxels
- this relocates the large scale movement along the z-axis from the loop inside the kernel to the third dimension of the grid of thread blocks
- this improves the hit rate of the texture cache even more
- yields the lowest time

Reordering The Loop

Observations

- the cache misses of the texture fetching constitute the major performance bottleneck
- the corresponding textures continuously contend for the limited amount of cache memory
- the memory transfers for the volume update take longer than the computations
- could alleviate the latter by having more projections but this would be bad for the former

Solution

- reverse the nested loop order

Reordering The Loop

```
compute position of first voxel

for K consecutive voxels along the z-axis do
    zero-initialize sum s[k] of weighted back-projected values
end

for I input projections do
    compute homogeneous detector coordinates q of first voxel
    for K consecutive voxels along the z-axis do
        dehomogenize detector coordinates q
        compute back-projected value by texture fetching
        update sum s[k] of weighted back-projected values
        update homogeneous detector coordinates q
    end
end

for K consecutive voxels along the z-axis do
    update volume at current voxel with computed sum s[k]
end
```

Results

Kernel B

Test	Kernel	Sync	I	K	B_x	B_y	Occupancy	I | – | –	I |M| –	I |M| T	Time	GUPS
1	A	no	1	512	32	8	1.000	1091 ms	4710 ms	4707 ms	9141 ms	7.3
2	A	no	4	512	32	8	0.750	575 ms	1199 ms	1196 ms	7802 ms	8.5
3	A	yes	4	512	32	8	1.000	554 ms	1185 ms	1169 ms	2710 ms	24.6
4	A	no	4	8	32	8	0.750	685 ms	980 ms	1085 ms	1990 ms	33.4
5	B	—	4	8	32	8	0.875	542 ms	989 ms	1179 ms	1527 ms	43.6
6	B	—	8	4	16	16	0.750	528 ms	725 ms	966 ms	1296 ms	51.4
7	B	—	16	4	16	32	0.750	506 ms	550 ms	826 ms	1051 ms	63.4
8	B	—	32	4	16	32	0.750	489 ms	494 ms	756 ms	969 ms	68.7

Test 5:

- replace kernel A with kernel B, but keep all other parameters identical. We clearly observe an improved hit rate of the texture cache.

Following three tests:

- we increase the number of projections I and tune the other parameters to obtain minimal computation times

Data Transfer Optimizations

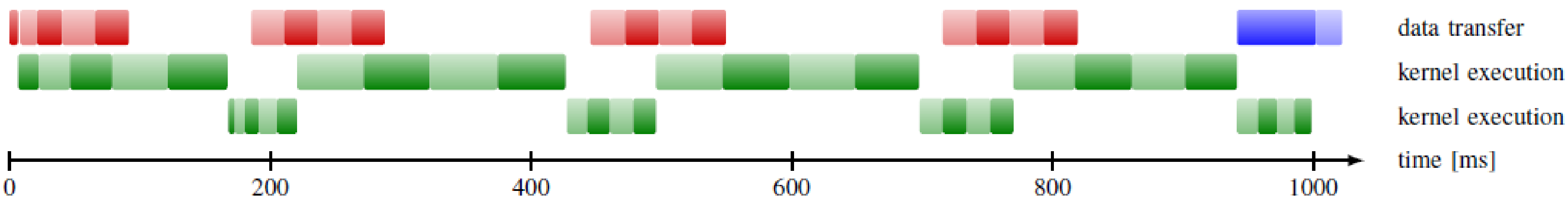
Transfer requirements

- the 512^3 volume results into 2,779 MB of data
- takes about half a second on the system.
- use the ability of our GPU to overlap kernel execution and data transfer to hide this latency
- use CUDA asynchronous kernel launches and asynchronous memcopy functions

Data Transfer Optimizations

Strategy

- add 8 projections in each transfer until reaching optimum of 32
- divide volume into two parts (384 and 128 xy slices, resp.)
- gives rise to two kernel executions
- this makes it possible to overlap the download of the first part of the volume with the reconstruction of the second part of the volume



Results

Volume	Implementation	Type	RMSE	Time	GUPS
512^3	fastrabbitEX [4]	CPU	—	7.45 s	8.94
	RapidRabbit [7]	GPU	—	2.98 s	22.3
	Thumper [this work]	GPU	0.021 HU	0.99 s	67.7
1024^3	fastrabbitEX [4]	CPU	—	43.8 s	12.2
	CERA [-]	GPU	—	36.1 s	14.7
	Thumper [this work]	GPU	0.021 HU	6.04 s	88.2

Rapid Rabbit Strikes Back (June 2013)

Inherits insight from Thumper plus additional tricks

Faster perspective divide

- original code

  $w = a_2x + a_5y + a_8z + a_{11}$

  $u = (a_0x + a_3y + a_6z + a_9) / w$

  $v = (a_1x + a_4y + a_7z + a_{10}) / w$

- using fast inverse square root

  $w = a_2x + a_5y + a_8z + a_{11}$

  $w' = rsqrt(w * w)$

  $u = (a_0x + a_3y + a_6z + a_9) * w'$

  $v = (a_1x + a_4y + a_7z + a_{10}) * w'$

  $result \mathrel{+}= tex2D(tRef, (u+0.5), (v+0.5) * w' * w')$

Click for more info and paper

Eric Papenhausen, Klaus Mueller. "Rapid rabbit: Highly optimized GPU accelerated cone-beam CT reconstruction." IEEE Nuclear Science Symposium and Medical Imaging Conference (NSS/MIC), 2013.

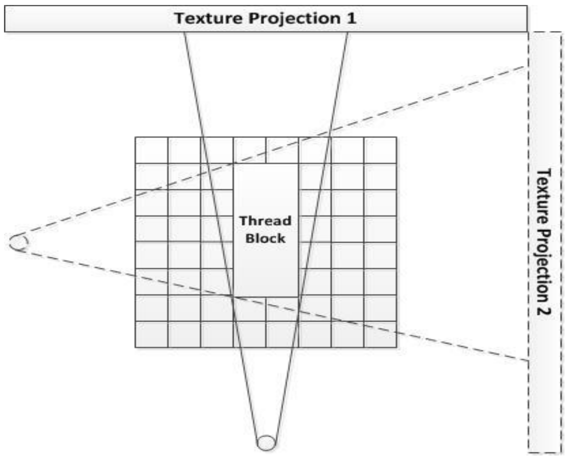
Rapid Rabbit Strikes Back

Observation:

- noticed a performance dip between the third and eighth kernel execution
- each kernel execution would last approximately 50 milliseconds at the beginning
- then the kernel executions would gradually get slower until it reached around 65 milliseconds, and then get faster toward the end.
- this dip in performance was because the cache locality was worse in the middle than at the beginning and end of the execution

Fix:

- transpose the volume at 45°
- simply swap x and y indexes



Rapid Rabbit Strikes Back

Observation:

- we thought atomic operations were slow
- but with the Kepler architecture, atomics are implemented in an ASIC
- furthermore, atomic operations are executed asynchronously with the calling thread

How to take advantage:

- accumulate the results into the volume using atomics
- we know that this will be a fast operation since there are no read/write collisions between threads
- thus the asynchronous nature of atomics guarantees that each thread will not have to stall after a write to global memory

Rapid Rabbit Strikes Back

Observations:

- in order for us to take advantage of CUDA streams, we have to page-lock the projection memory
- page-locked memory, however, is a scarce resource
- we cannot simply page-lock all the projections at the beginning

What to do:

- backproject 64 projections before loading another 64 projections into the page-locked memory
- performing the memory copy in another thread to hide some latency
- use ping pong scheme
- one buffer is used for back projection
- the other is copied to switch the buffers

Rapid Rabbit: Results

Implementation	Thumper	Baseline	+RSQRT	+Transpose	+Multi-Threaded	+Atomics
Timing	0.993 s	2.2 s	1.01s	0.967 s	0.951 s	0.921 s
GUPS	67.7	30.3	65.9	68.8	70.2	72.3

The Saga Continues…

Come, join the rabbit race

Be the fastest rabbit on the block!