Fast Cone-beam CT reconstruction using GPU

Giovanni Di Domenico

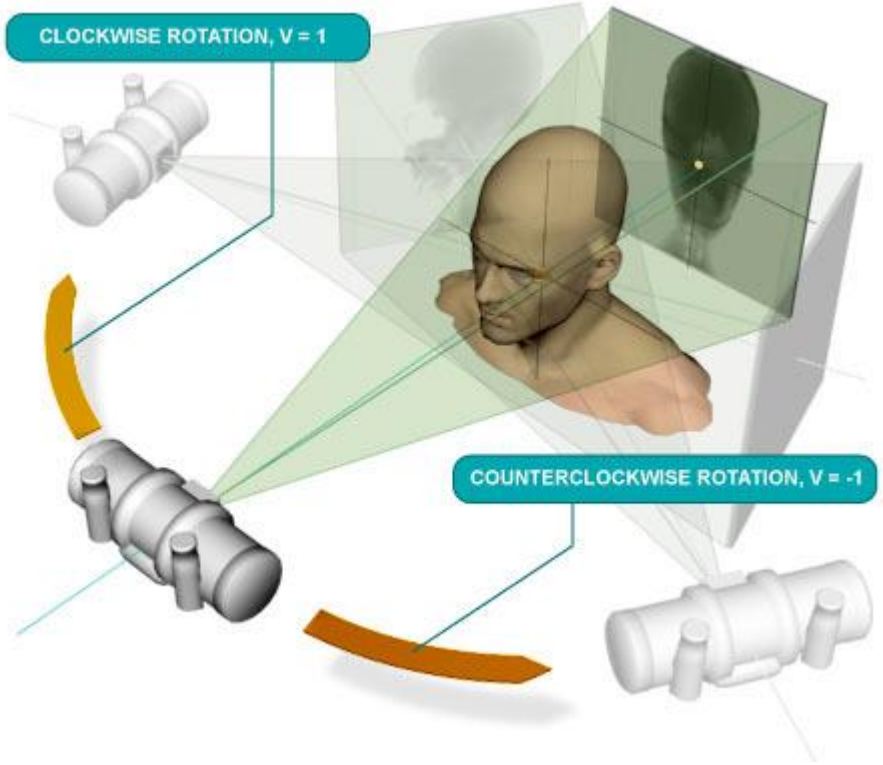
Università degli Studi di Ferrara & INFN Sezione di Ferrara

GPU computing in High Energy Physycs
10 – 12 September 2014 – Pisa

Outline

- Cone-beam CT systems
- Feldkamp Davis Kress (FDK) algorithm
- GPU implementations
- Optimization steps
- Future developments

Cone-beam CT system

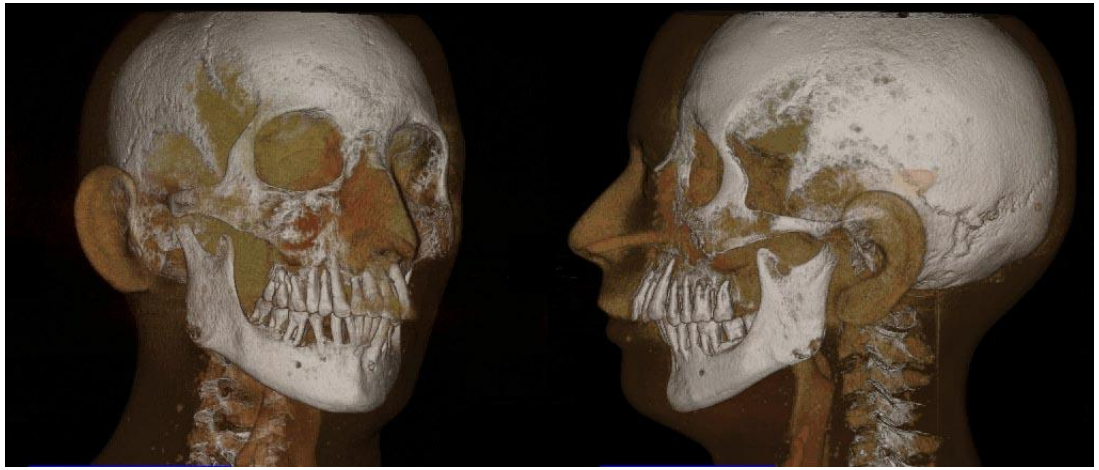


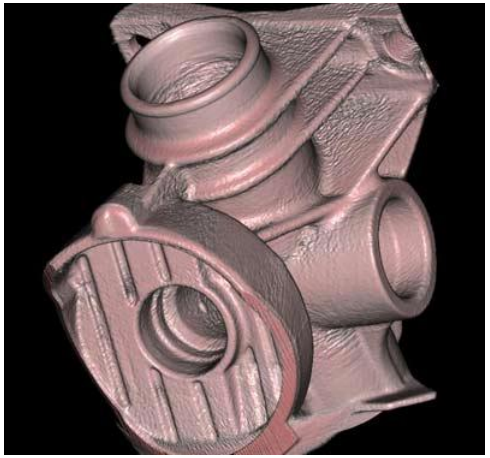
Tipically, a CBCT system tipically has:

- X-ray tube (30kVp – 120 kVp),
- flat panel detector (2048 x 2048),
- rotating gantry (200-1000 angular step),
- reconstructed volume size ~ 512^3

Cone-beam CT applications

- Non-destructive testing
- Dental imaging
- Microtomography
- Image-Guided Interventions





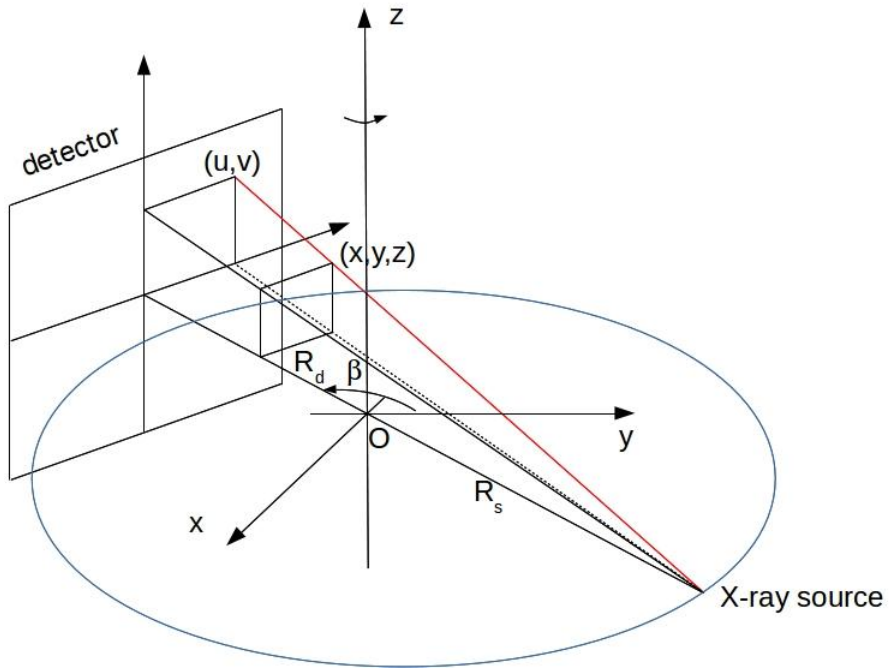
Circular Cone-beam Projection Geometry

In cone-beam CT, the detector acquires the line integral of attenuation coefficient $\mu(x,y,z)$ as function of rotation angle β:

$$g(u,v,\beta) = \int \mu(\vec{r}_o(\beta) + \alpha\hat{\theta})d\alpha$$

where:

- $\vec{r}_o(\beta)$ is the source position as function of rotation angle,
- $\hat{\theta}$ is an x-ray line beam



FDK algorithm - 1

- The Feldkamp-Davis-Kress algorithm is used to obtain an approximate solution of 3D tomographic problem.

$$\hat{\mu}(x,y,z) = \frac{1}{2}\int_0^{2\pi} d\beta \frac{1}{U^2} \int_{-u_m}^{u_m} du \frac{D}{\sqrt{D^2+u^2+v^2}} \cdot g(u,v,\beta) \cdot h(u-u')$$

where

- $\frac{D}{\sqrt{D^2+u^2+v^2}}$ is the cosine weighting function,
- $U = \frac{R_S - x\sin\beta + y\cos\beta}{R_S}$ is the backprojection weighting factor

FDK algorithm - 2

The FDK has three steps:

$$\hat{\mu}(x,y,z) = \frac{1}{2}\int_0^{2\pi} d\beta \frac{1}{U^2} \int_{-u_m}^{u_m} du \frac{D}{\sqrt{D^2+u^2+v^2}} \cdot g(u,v,\beta) \cdot h(u'-u)$$

1 - normalization step

2 - convolution step with ramp filter h(u)

3 - backprojection step

The reconstructed volume size is tipically 128^3, 256^3, 512^3, 1024^3.

FDK algorithm - 3

- Perspective projection for backprojection step

1) $$\vec{r}_p = A \cdot \vec{r} \qquad \begin{pmatrix} u \\ v \\ w \end{pmatrix} = \begin{pmatrix} a_0 & a_3 & a_6 & a_9 \\ a_1 & a_4 & a_7 & a_{10} \\ a_2 & a_5 & a_8 & a_{11} \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

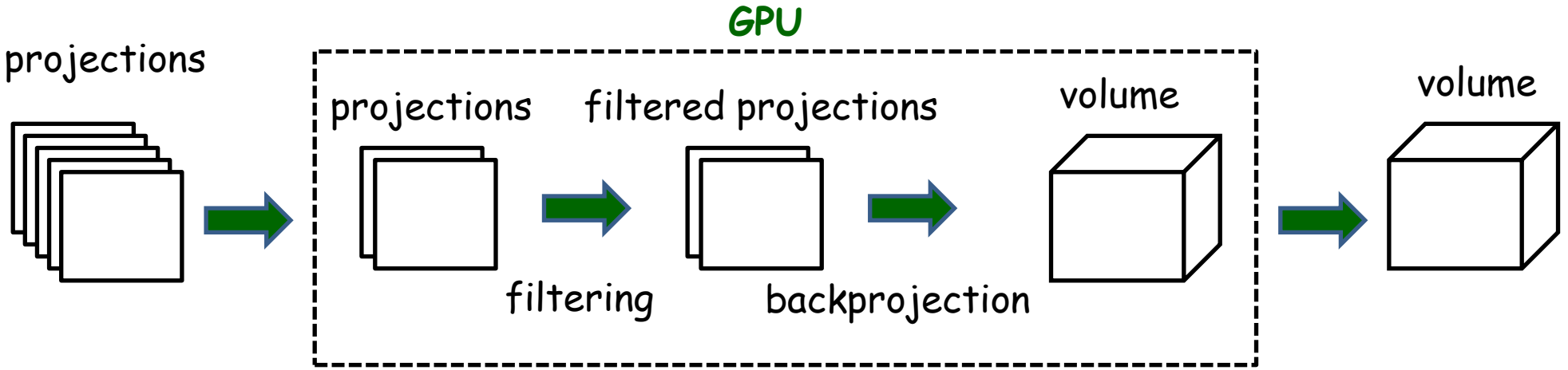
$$u = \frac{R_s + R_d}{R_s - x\sin\beta + y\cos\beta}(x\cos\beta + y\sin\beta)$$

2) $$v = \frac{R_s + R_d}{R_s - x\sin\beta + y\cos\beta} z$$

$$w = \frac{R_s - x\sin\beta + y\cos\beta}{R_s}$$

Data distribution and parallelization -1

- A 512^3 voxel volume requires at least 512 MB of memory space, it is no easy to store the entire volume and the projections on GPU device memory.
- We have decided to store the entire volume (or a 512^3 portion if the size > 512^3) in device memory and to load the projections into device memory when needed and remove it after its backprojection.



Data distribution and parallelization -2

In the first implementation (naive code) we transfer the first projection P_1 to the device global memory and perform:

1. weighting step by using 1 Cuda thread per pixel,
2. 1-D filtering step by using CUDA - FFT library applied to each projection with stride Nv,
3. backprojection step by using 1 Cuda thread for K voxels along z-direction.

The 3 steps are repeated for the remaining projections.

Setup for tests

Two datasets are used for testing the naive inplementation in CUDA:

- **Dataset 1** consists of 200 projections acquired on 360° circular scan trajectory. The size of each projection is 1024x512.
- **Dataset 2** consists of 642 projections acquired on 360° circular scan trajectory. The size of each projection is 256x192.

The GPU devices used are a GTX-680 and a GTX-Titan .

GPU	GTX-680	GTX-Titan
Architecture	Kepler GK104	Kepler GK110
Performance [GFlops]	3090.4	4500
Texture fillrate [GT/s]	128.8	187.5
Bandwidth [GB/s]	192.2	288.4

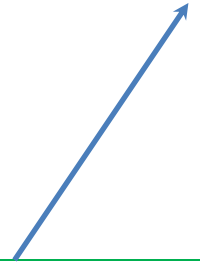
Results: naive code vs OMP code

GTX680	Step0 [s]	Step1_2 [s]	Step3 [s]	Total [s]
dataset1	0.4	6.99	12.41	19.8
dataset2	0.18	2.21	4.37	6.76
Titan	**Step0[s]**	**Step1_2 [s]**	**Step3 [s]**	**Total [s]**
dataset1	0.57	5.69	8.05	14.31
dataset2	0.45	1.81	3.1	5.36
OpenMP	**Step0 [s]**	**Step1_2 [s]**	**Step3 [s]**	**Total [s]**
dataset1	0.67	5.09	206.95	212.71
dataset2	0.21	1.26	80.96	82.43

- step 0: normalization
- step 1: cosine weighting
- step 2: filtering
- step 3: backprojection

Optimizing backprojection step - 1

```
Input: P_n, A_n, L, O_L, R_L, f_L
Output: update value of f_L
for i=0 to L-1
 for j=0 to L-1
  for k=0 to L-1
   x= O_L+iR_L; y=O_L+jR_L; z=O_L+kR_L;
   f_L (i,j,k)=f_L (i,j,k)+P_n(u_n(x,y,z),v_n(x,y,z))/w_n^2(x,y,z);
  end for
 end for
end for
```



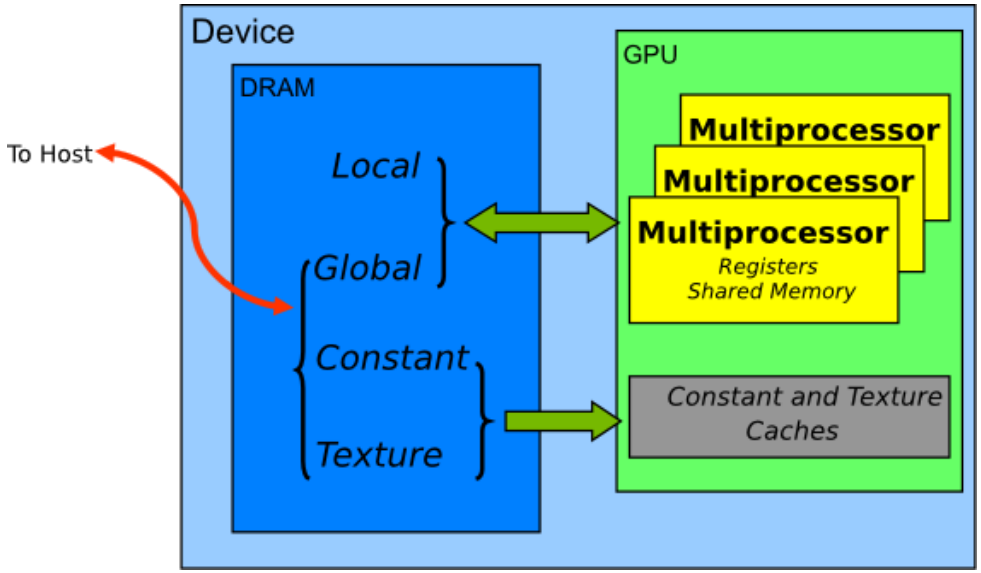
The update value calculation requires: a) perspective transformation b) bilinear interpolation on projection

Optimizing backprojection step - 2

- The acceleration technique we have used to optimize backprojection is to maximize the effective memory bandwidth because the backprojection step is a memory intensive operation.

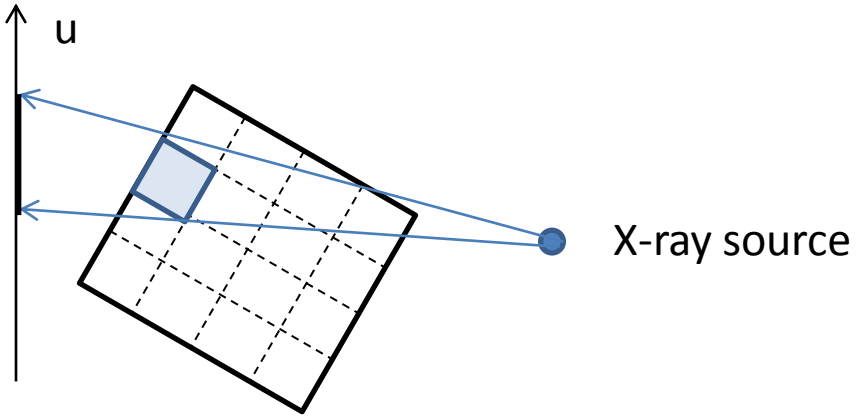
Constant memory:
read-only memory, size 64 kB, it is cached, optimized when warp of threads read same location

Texture memory:
read-only memory, it is cached, the texture cache is optimized for 2D spatial locality, so threads of the same warp that read texture addresses that are close together will achieve best performance.



Optimizing backprojection step - 3

1. Memory coalescing
   - Organize the threads inside a block to access contiguous memory location: the volume is LxLxL is processed by a grid (L/Bx)x(L/By)x(L/K),
2. Global memory access reduction
   - Use a kind of memory that has a cache mechanism (texture and constant memory)
   - Increase the number of projections processed by a single kernel to reduce the number of access to volume global memory,



RabbitCT platform

- RabbitCT[1] is an open and uniform benchmark enviroment for backprojection performance.
- It has a accessibile dataset with the matrix A with the geometry information: the dataset size is 496 projections acquired on a 200° circular short scan trajectory. The size of each projection image is 1248x960.
- The user reconstruction algorithm must be inserted in a dynamic library module called by RabbiCTRunner program. At the end of user code execution the program gives some benchmark information .

[1] RabbitCT an open platform for benchmarking 3D cone-beam reconstrution algorithms. C. Rohkohl et al., Med. Phys. 36 (9), 3940-3944, 2009.

Backprojection code version 0, 1 and 2

- The backprojection version 0 is the naive implementation,
- In the code v1, the matrix *A* is moved to constant memory on GPU device by using the call cudaMemcpyToSymbol(…);
- One projection is loaded in texture memory by using cudaArray and accessed by tex2D(…)
  - 2D textures have a cache mechanism and device bilinear interpolation.
- In the code v2 we have loaded more projections in texture memory to reduce the number of access to the volume global memory.

Results for v_0, v_1 and v_2 on GTX-680

version	Size	Time [s]	Occupancy [%]	Error [HU]	GUPs	Kernel [%]
V0	128	1.18	83	0.03	0.82	31.6
V0	256	2.51	84	0.03	3.09	68.1
V0	512	10.32	84	0.03	6.01	90.6
V1	128	0.98	89	0.16	0.99	22.4
V1	256	1.39	89	0.16	5.58	61.8
V1	512	4.77	89	0.16	12.90	81.8
V2 (4)	128	1.04	89	0.16	0.93	21.75
V2 (4)	256	1.29	92	0.16	6.16	60.29
V2 (4)	512	3.00	95	0.16	20.67	70.30

Cuda FDK results

Backproiection step comparison

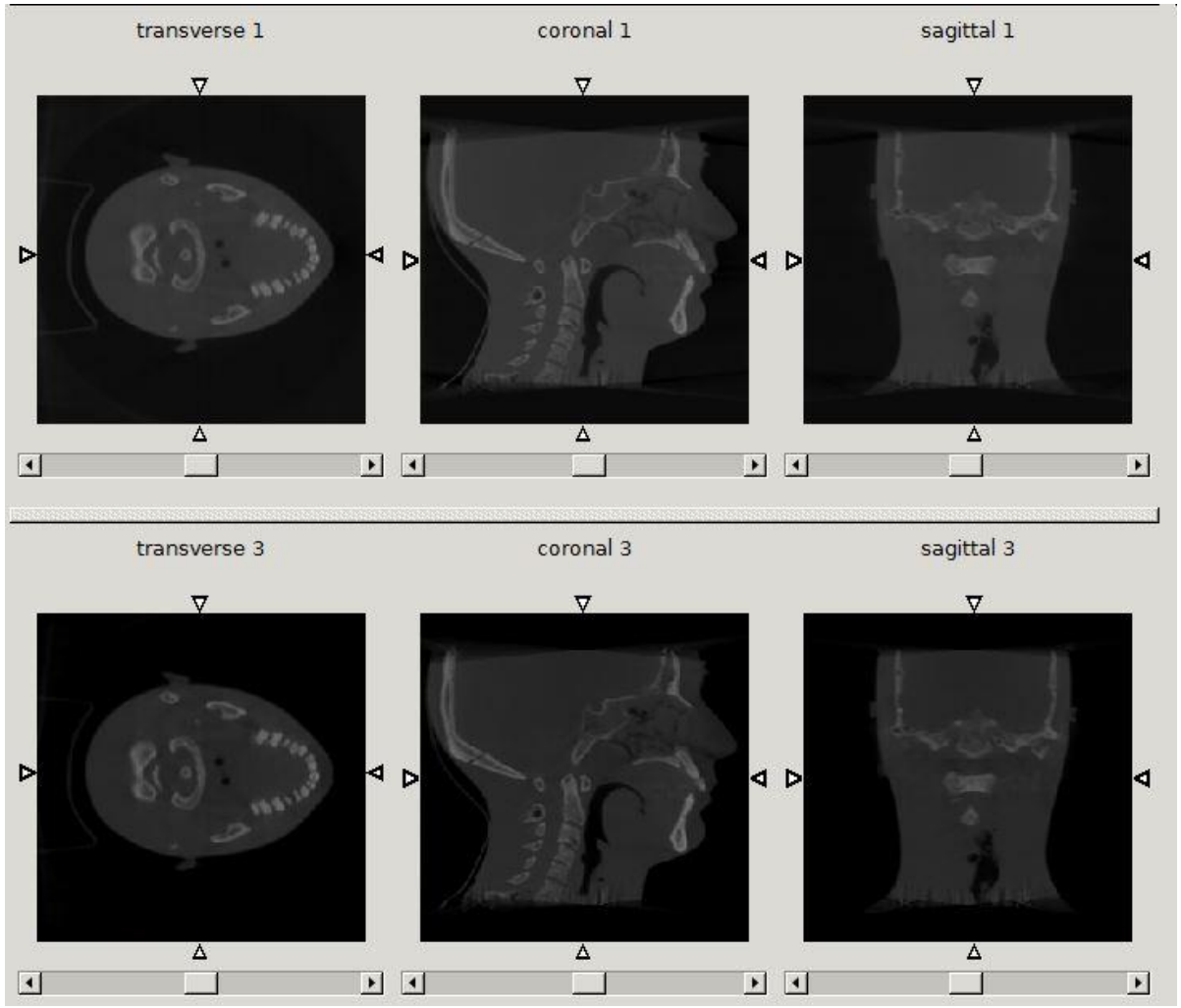
GTX680	V0 [s]	V1 [s]	V2 [s]	OMP[s]
dataset1	12.41	3.75	2.27	296.6
Titan	**V0 [s]**	**V1 [s]**	**V2 [s]**	**OMP [s]**
dataset1	8.05	2.24	1.42	206.95

All steps comparison

GTX680	V0 [s]	V1 [s]	V2 [s]	OMP[s]
dataset1	19.8	11.15	9.66	302.3
Titan	**V0 [s]**	**V1 [s]**	**V2 [s]**	**OMP [s]**
dataset1	14.31	8.49	7.68	212.7

Dataset_1 reconstructed images

CUDA



$\langle \Delta I \rangle / \langle I \rangle = 4.0 \times 10^{-3}$

OMP

$\Delta I_{max} / Ipix = 4.0 \times 10^{-2}$

Summary

- Using GPU + CUDA in Cone-Beam CT reconstruction we are able to accelerate (> x25) the FDK reconstruction process expecially the backprojection step.
- The reconstruction time is less than 10 s for our datasets for reconstruction volume of 512^3 (real-time reconstruction).
- To increase the performance:
  - Reduce the backprojection time by using Cuda streams to overlap data transfer operations with kernel executions
  - Work on optimization of filtering step.

Texture usage

```
/* texture 2D */
texture<float,cudaTextureType2D,cudaReadModeElementType> tex_img;

/* cudaArray for texture – projection binding*/
cudaArray *array_img;

static cudaChannelFormatDesc channelDesc;
….
 // set-up texture parameters
  tex_img.addressMode[0] = cudaAddressModeBorder;
  tex_img.addressMode[1] = cudaAddressModeBorder;
  tex_img.filterMode         = cudaFilterModeLinear;
  tex_img.normalized         = false; // don't access with normalized texture coords

  // copy image data to array, bind the array to the texture
  channelDesc = cudaCreateChannelDesc(32,0,0,0,cudaChannelFormatKindFloat);
  cudaMallocArray( &array_img, &channelDesc, rcgd->S_x, rcgd->S_y );
…
 // copy projection on device in a texture object
  cudaMemcpyToArray( array_img, 0, 0, &I_n[0], Nu * Nv * sizeof(float), cudaMemcpyHostToDevice);
  cudaBindTextureToArray( tex_img, (cudaArray*)array_img, channelDesc);
```