Fast Radon Transform via Fast Nonuniform FFTs on GPUs

Stefano Marchesini, Filipe Maia, J. Pien (cuda consultant), Chao Yang, Andre. Schirotzek, Alastair McDowell, Howard Padmore
Lawrence Berkeley National Laboratory, Berkeley, CA

Introduction

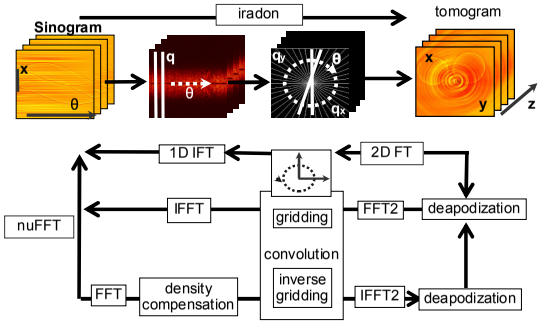
Fast Radon and inverse Radon transforms form the computational kernels of many image reconstruction algorithm. We have recently developed a reconstruction algorithm for X-ray phase contrast tomography using a compressive sensing technique. In our approach, we formulate the reconstruction problem as an L1-minimization problem, i.e.

$$\min \|x\|_1$$
$$\text{subject to} \quad \|\mathit{filter}\,(\,\text{Radon}\{x\} - \text{data})\|_2 \le \varepsilon$$

where x is the phase contrast image to be reconstructed and *filter* is a polynomial interpolation operator designed to reduce the ringing artifact in the reconstruction, and ε is a regularization parameter we choose in advance. We solve the L1-minimization problem by using the SPGL1 software developed by E. van den Berg and M. P. Friedlander [1]. The software requires us to provide a function to perform y = Radon{x}. Due to the large volume of data produced by fast detectors at beam line (BL832) at the advanced light source at Lawrence Berkeley National Lab, we would like perform phase contrast tomographic reconstruction in real time.

Fast Radon Transform

The 2D Radon tranform of an image x can be implemented in a number of ways. One of the most efficient ways to perform Radon{x} is to first perform a 2D FFT of x on a regular grid, and then interpolate the transform onto a polar grid before 1D inverse Fourier transforms are applied to interpolated points along the same radial lines.



The interpolation between the Cartesian and polar grid is known as can be carried out using a "gridding" algorithm that maintains desired accuracy and low computational complexity. The gridding algorithm essentially allows us to perform a non-uniform FFT.

The Gridding Algorithm

Inverse gridding: Convolve Kaiser Bessel (KB) kernel from regular samples (x_r, y_r) to regular grid points (x_i, y_i)

$$G_r(x_r,y_r)=\sum_{x_i,y_i}\frac{1}{D(x_i,y_i)}G_i(x_i,y_i)K(x_i-x_r,y_i-y_r)$$

Gridding: Convolve KB kernel from irregular samples (xi,yi) to regular grid points(x_r, y_r).

$$G_i(x_i,y_i)=\sum_{x_r,y_r}G_r(x_r,y_r)K(x_r-x_i,y_r-y_i)$$

Density compensation:

$$D_i(x_i,y_i)=\sum_{k,j}K(x_i-x_j,y_i-y_j)$$

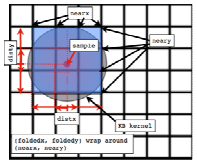
Deapodization:

$$d=\frac{1}{FFTK(x_r,y_r)}$$

constant

[1] E. van den Berg and M. P. Friedlander, SIAM J. on Sci. Comp. 31, 890-912, 2008.

GPU Implementation (prototype code in MATLAB and Jacket)



```
% Kaiser-Bessel filter
beta=2.3934*2;
% Kaiser-Bessel window, it could use look up tables (LUT):
KB=@(x,y) KB1(nj*2+1,x,beta)'*KB1(nj*2+1,y,beta);

function w=KB1(M,x,beta)
bes = abs(besseli(0,beta));
w = besseli(0,beta*sqrt(1-(2*x/M).^2))/bes;
```

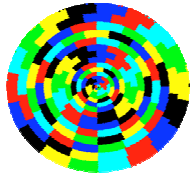
Gridding:
- Loop through samples
- Sum shifted kernels

```
function GI=reg2ir(xi,yi,G,KB,nj)
% input: xi,yi, irregular sample coordinates
%       G:  regular sample values
%       KB: convolution kernel (nj: width)
% output: irregular sampled G
GI=complex(zeros(size(xi))); %sample values
[Ns,Ms]=size(G);  % image size
ni=numel(xi);     % number of samples
jj=-nj:nj;        %kernel index/ grid location

for ii=1:ni; %loop over sampling points
   nearx=round(xi(ii)); %nearest grid point x
   neary=round(yi(ii)); %nearest grid point y
 % calculate distance between xi and grid points jj+nearx
   distx=xi(ii)-(nearx+jj);   %nj long vector
   disty=yi(ii)-(neary+jj);   %nj long vector
   foldedx=mod(nearx+jj,Ns)+1;%wrap around boundary
   foldedy=mod(neary+jj,Ms)+1;%wrap around boundary
   %add up G*KB
   GI(ii)=sum(sum(G(foldedy,foldedx).*KB(distx,disty)));
end
```

Density compensation:
Loop through irregular samples for comparison

divide in tiles



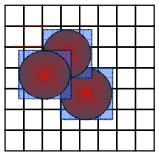
```
function D=irreg2irreg(xi,yi,KB,siz)
% xi,yi, irregular coordinate
%          KB, kernel, siz, image boundary
% output: D density
ni=numel(xi);

D=zeros(ni,1);
for ii=1:ni %  loop over all sampling points
    %  distance between ii_th sampling point
    %  and all sampling points
    distx=(xi(ii)-xi); %ni long vector
    disty=(yi(ii)-yi); %ni long vector
    %wrap around boundary
    distx=distx-round(distx/(siz(1)))*(siz(1));
    disty=disty-round(disty/(siz(2)))*(siz(2));
    % Calculate density
    D(ii)=D(ii)+sum(sum(KB(distx,disty)));
end
```

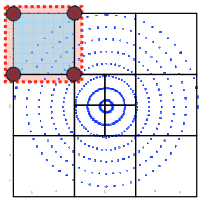
Inverse gridding:
- Loop through grid points
- Add sampled kernel
- Slow!

```
function G=irreg2reg(xi,yi,GI,KB,nj,siz)
% xi,yi, GI irregular coordinate and values
% KB is the kernel, nj kernel width, siz is the size of the
resulting grid
Ns=siz(1); % boundary of the image
Ms=siz(2); % boundary of the image
G=zeros(siz); %initial blank image
jj=-nj:nj;    %kernel index/ grid location

for ii=1:numel(GI);     %loop over sampling points
   nearx=round(xi(ii));   %nearest grid point x
   neary=round(yi(ii));   %nearest grid point y
   % calculate distance between xi and grid points jj+nearx
     distx=xi(ii)-(nearx+jj); %nj long vector
     disty=yi(ii)-(neary+jj); %nj long vector
     foldedx=mod(nearx+jj,Ns)+1; %wrap around boundary
     foldedy=mod(neary+jj,Ms)+1; %wrap around boundary
     G(foldedy,foldedx)=G(foldedy,foldedx)+... %add
                        GI(ii)*KB(distx,disty);
end
```



Improvement:
- Pre-bin (see [2])
- Divide among tiles
- Load balance



Performance:
- FFT (3 msecs for 1k by 1k pixels)
- Gridding (5 msecs for 180 by 1k pixels)
- Inverse gridding (1 msec)

We achieved 50x speedup on a Tesla GPU Compared to a fast CPU!

[2] www.nvidia.com/docs/IO/47905/ECE757_Project_Report_Gregerson.pdf