CT RECONSTRUCTION USING GPU - CUDA PROGRAMMING

PARALLEL PROCESSING SYSTEMS PROJECT REPORT

PROF SAVERIO DI VITO

BY
UMAMAHESWARAN RAMAN KUMAR
CHUNXIA LI

JUNE 25, 2017

UNIVERSITY OF CASSINO AND SOUTHERN LAZIO, ITALY

1. INTRODUCTION

X-ray computed tomography is a widely adopted medical imaging method that uses projections to recover the internal image of a subject. Since the invention of X-ray computed tomography in the 1970s, several generations of CT scanners have been developed. As 3D-image reconstruction increases in popularity, the long processing time associated with these machines has to be significantly reduced before they can be practically employed in everyday applications. Parallel computing is a technique that utilizes multiple resources to process a computational task simultaneously. Each resource computes only a part of the whole task thereby greatly reducing computation time. In this project, we use parallel computing technology to implement the GPU-accelerated back-projection algorithm for X-ray CT reconstruction with Texture Memory.

The aim of this work is to implement the back-projection algorithm on the GPU. In the following sections, we will give a brief background of CT, what a GPU is, and how it is organized. In section 2, we describe how an object is reconstructed from its projections using the Back-projection algorithm. The implementations are described in section 3. Finally, in section 4, we show the results, the performance and considerations.

2. BACKGROUND

X-RAY COMPUTED TOMOGRAPHY

X-ray computed tomography (CT) is an imaging modality or procedure whereby an X-ray tube is used to generate an X-ray beam that passes through an object while a detector placed on the other side of the object records the attenuated X-ray signals, which are called projections. During the X-ray exposure, the projection data from different directions are gathered and processed with the aid of a computer to produce 2D or 3D cross-sectional or volumetric images of the object. This technique of imaging by section is sometimes referred to as *tomography*. Because of its ability to reveal the internal anatomical structures of an object, it has surpassed traditional X-ray technology as a preferred/superior diagnostic method.

In X-ray CT, an X-ray of intensity I_0 is emitted and passes through an object; the intensity is attenuated and measured at intensity I_1 on the detector. An empirical Lambert-Beer's law reveals the relationship between the intensities and the characteristics of the object radiated by the X-ray:

$$-\frac{I_1}{I_0} = \int_L \mu(x)dx.$$

where $\mu(x)$ is the intrinsic attenuation coefficient of the material at location *x*, and *L* is the path of the X-ray through the object.

The attenuation coefficient characterizes the rate at which X-rays are weakened by scattering or absorption as they propagate through the object. Given the input and output X-ray intensities $I_0$and I_1, respectively, the line integral of the object attenuation coefficient along the ray path can be determined. The task of computed tomography is to calculate the attenuation coefficient function $\mu(x)$ at various locations based on the available information, I_0 and I_1, gathered by the detector at different directions (different $\theta's$). Parallel beam of X-ray Computed Tomography (CT) is shown below:

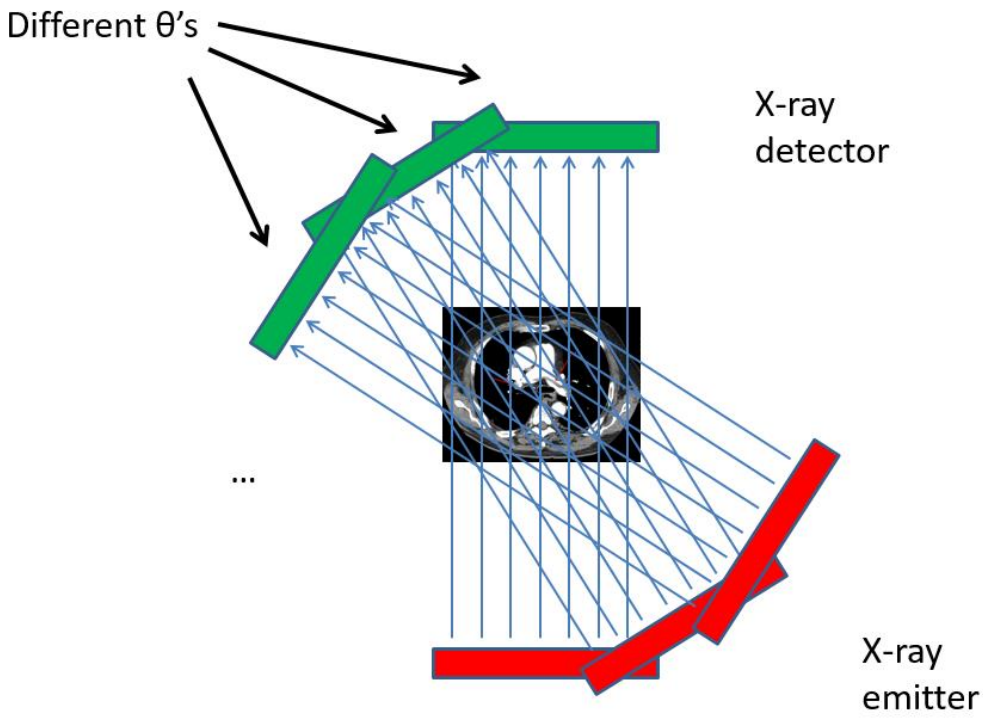

Fig 2.1: Parallel beam CT

INTRODUCTION TO GPU DEVICES

The aim of this part is to give a simple introduction to how a GPU is organized. The graphic card used on this work is the GTX-950. This GPU is organized into 16 highly threaded multiprocessors called Streaming Multiprocessors (SMs). In the GTX-950 each SM is composed by 32 streaming processors or SPs, for a total of 512 cuda cores. From a CUDA programmer point of view the GPU is a device that executes a given number of threads at a time. Threads can be identified (indexed) using 1, 2 and 3 dimensions and are organized into blocks. A block can also be indexed by 1, 2 or 3 dimensions and executes on a SM. Each block is executed in batches of 32 threads called warps. The maximum number of warps that can reside on a SM is 48. To allow scalability blocks are queued into the GPU.

In a memory respect, the GPU has access to four types of memory which are: i) global memory, ii) texture memory, iii) shared memory and iv) constant memory. Global memory is the biggest region of memory, but it is also the slowest, because it is outside of the GPU. Texture memory is a special way of accessing the global memory and only presents advantages when there is data locality. Texture memory is accessed through the texture hardware. Shared memory resides inside of the SM and therefore it is very fast but it is also a very small region of memory. Finally, constant memory is the fastest memory on the GPU but it has two downsides. First it is read-only (at least from the GPU side), and second it is a very small region. A representation of a CUDA memory model is presented as below.

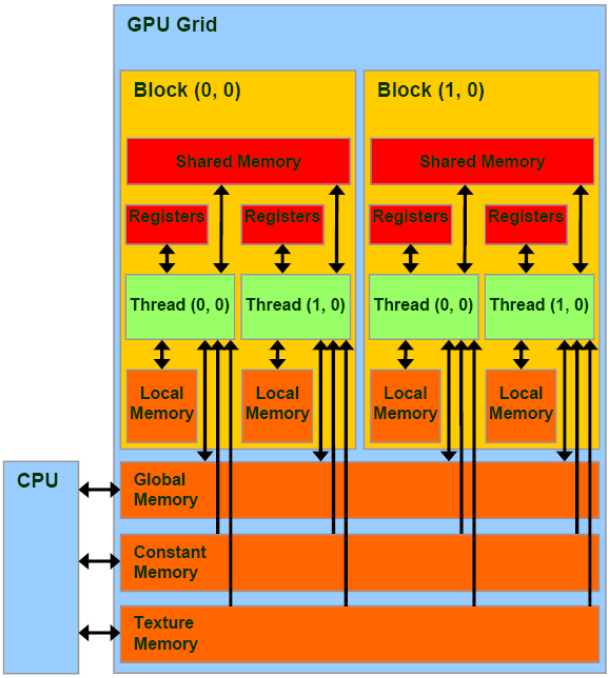

Fig 2.2: CUDA memory model

In this project, we implement back-projection algorithm for X-ray CT reconstruction with Texture Memory. The back-projection data will be stored in texture memory. The use of texture memory presents two advantages. First it is a fast access memory when there is a great number of accesses to the same neighbouring address, and second the interpolation is done in hardware.

3. ALGORITHM

Mathematically, the back-projection operation is defined as:

$$f_{BP}(x,y) = \int_0^{\pi} p(x\cos\phi + y\sin\phi, \phi)d\phi$$

Geometrically, the back-projection operation simply propagates the measured sinogram back into the image space along the projection paths. The geometrical interpretation of back projection is shown below:

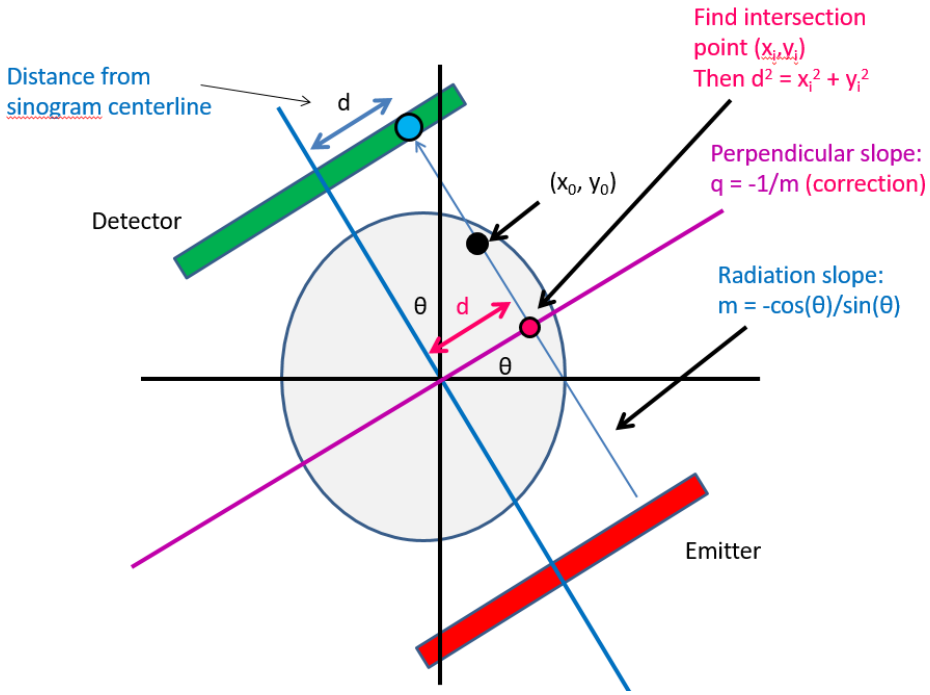

Fig 3.1: Geometrical interpretation of back-projection

As we know, each location on detector corresponds to multiple x_0's. For x_0, we need to find that the radiation measurement corresponds to the line passing through x_0 at each θ as line 1: (point-slope):

$$(y_i - y_0) = m(x_i - x_0)$$

We can get radiation slope: m = -cos(θ)/sin(θ), and then know the perpendicular slope

q = -1/m (correction) and corresponds to line 2:

$$y_i = qx_i$$

After that, we combine and solve the equations above and get the position of intersection point:

$$x_i = \frac{y_0 - mx_0}{q - m}, \quad y_i = qx_i$$

Then, we have the distance from measurement (sinogram) center line:

$$d = \sqrt{{x_i}^2 + {y_i}^2}$$

To reconstruct point, sum measurement along every line passing through that point and finally we can get the back projection of each slice at each angle.

4. IMPLEMENTATION

GLOBAL DECLARATIONS

The sinogram image is not passed as an input argument because it is accessed through the texture reference global variable.

```
texture<float, 2, cudaReadModeElementType> texReference;
```

The template arguments to the '*texture*' are explained below:

- 1st argument – *ReturnType*
  The type of the value returned when accessing texture memory. It is set to 'float' because the texture memory can interpolate the point even if the dimension referred is not an integer value.
- 2nd argument – *Dimension*
  The number of dimensions of the texture memory. It can be 1, 2 or 3 for 1D, 2D and 3D. In this case it is '2' as the sinogram is a two dimensional image.
- 3rd argument – *ReadMode*
  Optional parameter type to represent the read mode of the texture memory. The parameter defaults to '*cudaReadModeElementType*'.

KERNEL FUNCTION

```
__global__ void backprojection(float* doutput, int numberOfAngles,
                               int stepAngle, int projectionWidth)
```

The '__global__' keyword is used to specify that function is a kernel function and it is callable from the host. The function takes four input parameters:

- *float* doutput*
  Pointer to the output reconstructed image
- *int numberOfAngles*
  Number of projection angles of the sinogram
- *int stepAngle*
  Angle between two successive projections of the sinogram
- *int projectionWidth*
  Width of each projection of the sinogram

```
if(yIndex <= projectionWidth && xIndex < projectionWidth){

  for (int n=0, angle=0 ; n< numberOfAngles; n++, angle=angle+stepAngle){

    if(angle != 0 && angle != 90 && angle != 180){
      // Calculate the index of projection value
    }

    else if(angle == 0 || angle == 180){
      // Calculate the index of projection value when dy is 0
    }

    else{
      // Calculate the index of projection value when dx is 0
    }

    // Increment pixel value with the value of the projection element

  }

  // Assign the pixel value to the output image pointer

}
```

There are some additional threads running in one grid because the projection width is not exactly divisible by the grid size. The first '*if*' loop is to make sure only the relevant threads are being executed. Each valid thread will execute a loop for all the angles in the sinogram to find the value of projection affecting that particular pixel of the output image.

```
tex2D(texReference, center+distance, n)
```

The values of the projections are fetched from the texture memory with '*tex2D*' function. The arguments of the function are:

- 1st argument – *Reference to texture memory*
  The reference variable to the texture memory declared in the global scope.
- 2nd argument – *x*
  The x index of the value that is referred. This index value is calculated following the algorithm seen earlier.
- 3rd argument – *y*
  The y index of the value that is referred. This index is the current value of the loop.

MAIN FUNCTION

- Set device to be used
- Read and display input sinogram image

- Allocate memory for host and device pointers of output image in host and device memory respectively
- Allocate memory for cuda array in the device for the input sinogram image
- Bind texture memory to cuda array of sinogram
- Calculate grid size based on block size
- Call kernel function by passing the required parameters
- Synchronize device to host
- Copy output reconstructed image from device to host
- Convert image from float to integer to display
- Free all memory

KERNEL CONFIGURATION

```
dim3 blockSize;
blockSize.x = BLOCK_SIZE;
blockSize.y = BLOCK_SIZE;

// Assign grid size
dim3 gridSize;
gridSize.x = sinogramMat.cols/blockSize.x;
gridSize.y = sinogramMat.cols/blockSize.y;

// Adjust grid size based on size of sinogram image
if(sinogramMat.cols % blockSize.x > 0){
      gridSize.x = gridSize.x + 1;
}
if(sinogramMat.cols % blockSize.y > 0){
      gridSize.y = gridSize.y + 1;
}

backprojection<<<gridSize, blockSize>>>(…);
```

- Block size

  The size of the block is usually fixed by the user or calculated dynamically by querying the GPU configuration. Considering the size of the input sinogram image and assuming that the output image size will not exceed the total number of threads running, the block size is fixed to be 16x16.

- Grid size

  The size of the grid is calculated from the size of the block and the input sinogram image because ideally it is required that the number of threads running should be equal to the output image size. This is not always possible since the dimension of the output image is not always divisible by the block size and so an extra block is allocated to run few required threads and some additional threads. It is important to make sure in the kernel function that these additional threads are kept idle.

5. RESULTS

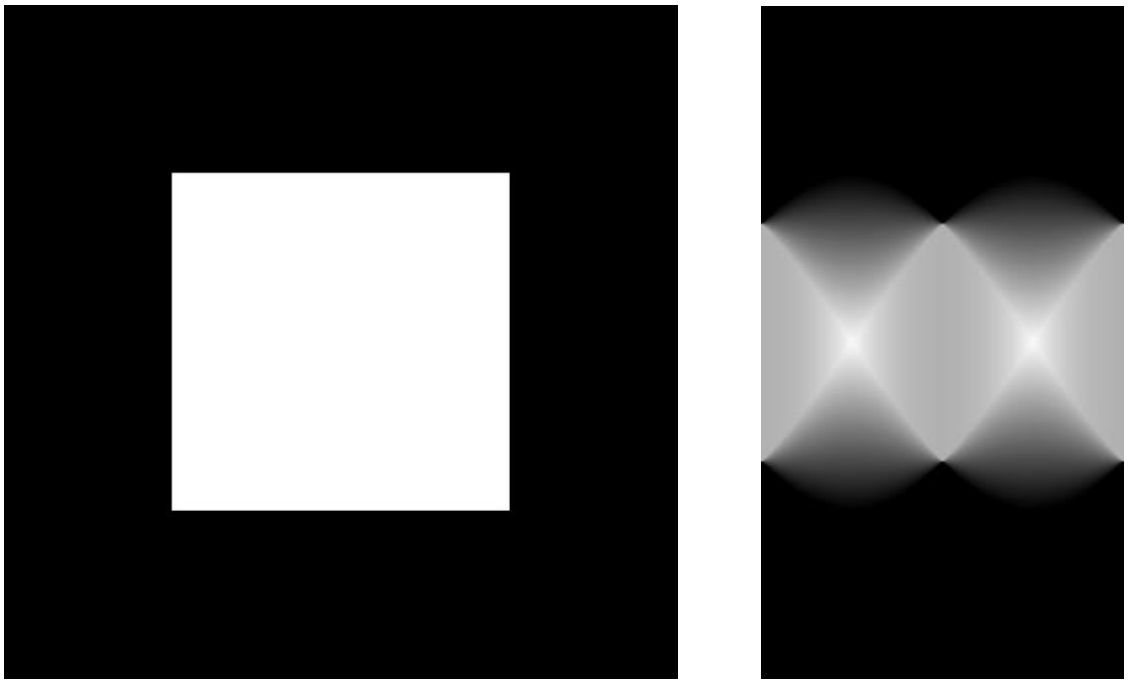

Fig 4.1: Original image (left) & Sinogram (right)

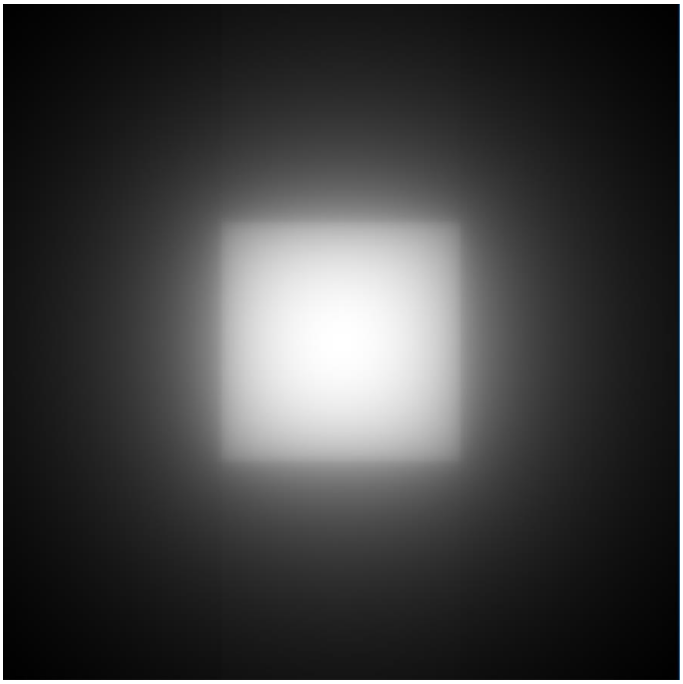

Fig 4.2: Reconstructed image

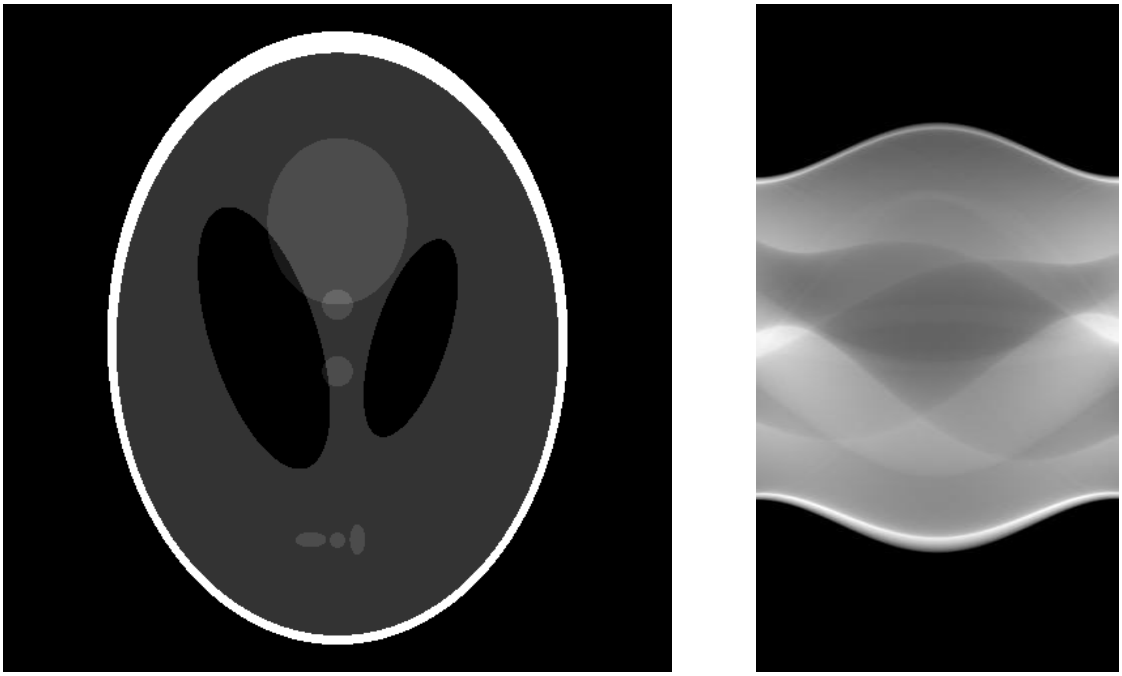

Fig 4.3: Original image (left) & Sinogram (right)

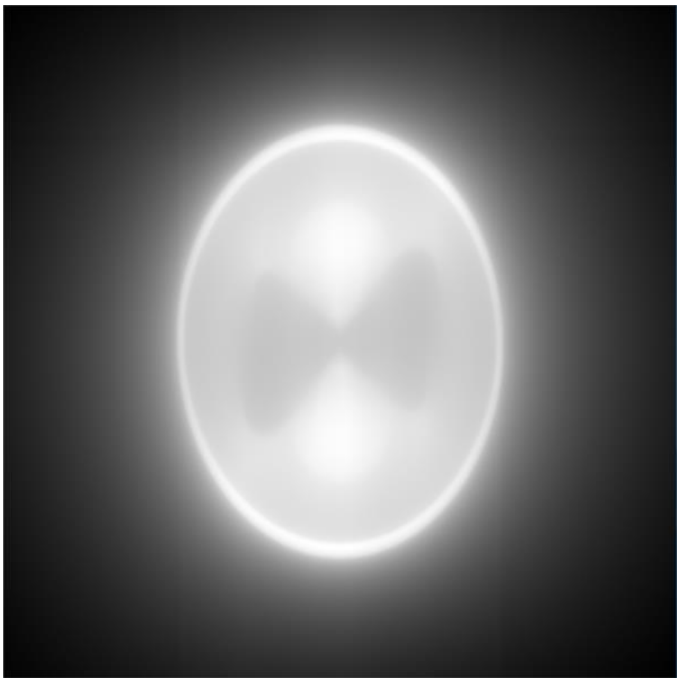

Fig 4.4: Reconstructed image

6. COMPARISON

The below table shows the time taken in seconds for the reconstruction of the two images shown both for oval and square. It is cleared observed that GPU has given very high performance compared to CPU because the reconstruction problem is highly parallelizable.

	CPU(s)	GPU(s)
Oval(512 x 512)	10.09	0.54
Square(1000 x 1000)	35.62	1.95

7. FUTURE ENHANCEMENTS

Listed below are few enhancements that can be done to improve the result and to make it more efficient:

- Filtered back projection
- Handle many sinogram images

8. CONCLUSION

We can use the GPU to accelerate the reconstruction of the X-ray CT scan from the sinogram. The results and comparisons also show that the back-projection algorithm is easily parallelizable and this gives a very good advantage over the traditional CPU reconstruction.

APPENDIX

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"
#include <stdio.h>
#include <iostream>
#include <opencv2/core/core.hpp>
#include <opencv2/core/types_c.h>
#include <opencv2/imgproc/imgproc.hpp>
#include <cmath>
#include <opencv2/highgui/highgui.hpp>


#define PI 3.14159265
#define BLOCK_SIZE 16
#define STEP_ANGLE 1


texture<float, 2, cudaReadModeElementType> texReference;


/*****************************************************************************
***************
* Name                  :       backprojection
* Input parameters      :       float* doutput        -  Device pointer to
output image
*                               int numberOfAngles    -  Total number of
projection angles
*                                                       int stepAngle            -
Angle between each projection
*                                                       int projectionWidth   -  Width
of projection
* Return type           :       void
* Description           :       Kernel function to reconstruct CT scan image
from sinogram
*
*****************************************************************************
****************/
__global__ void backprojection(float* doutput, int numberOfAngles, int
stepAngle, int projectionWidth)
{
      // x and y index of output image calculated from block id and
thread id
      int yIndex = blockIdx.y*blockDim.y + threadIdx.y;
      int xIndex = blockIdx.x*blockDim.x + threadIdx.x;

      // If the thread executed is within the output image size
      if(yIndex <= projectionWidth && xIndex < projectionWidth){

            //Intermediate variables
            float center = projectionWidth / 2;
            float angleRadians;
            float slopeProjection, slopeNormal;
            float x, y;
            float xIntersection, yIntersection;
            float distance;
            float pixelValue = 0.0;

            //Loop for all the angles in the sinogram
```

```
            for (int n=0, angle=0 ; n< numberOfAngles; n++,
angle=angle+stepAngle){

                  //If angle is other than 0, 90, 180
                  if(angle != 0 && angle != 90 && angle != 180){

                        //Slope of projection line
                        angleRadians = angle * PI /180.0;
                        slopeProjection = - cos(angleRadians) /
sin(angleRadians);

                        //Slope of line normal to projection line
                        slopeNormal = - 1 / slopeProjection;

                        //Intersection of projection and normal line
                        x = xIndex - center;
                        y = yIndex - center;
                        xIntersection = (y - slopeProjection * x) /
(slopeNormal - slopeProjection);
                        yIntersection = slopeNormal * xIntersection;

                        //Distance of the required projection value from
center of projection
                        distance = sqrt(xIntersection*xIntersection +
yIntersection*yIntersection);
                  }

                  //If angle is 0 or 180
                  else if(angle == 0 || angle == 180){
                        //Calculate distance if slope of projection line
is 0
                        x = xIndex - center;
                        distance = x;
                  }

                  //If angle is 90
                  else{
                        //Calculate distance if slope of projection line
is infinity
                        y = yIndex - center;
                        distance = y;
                  }

                  //Increment pixel value with the value of the
projection
                  pixelValue = pixelValue + tex2D(texReference,
center+distance, angle);

            }

            //Assign pixel value to final output image
            doutput[yIndex*projectionWidth + xIndex] = pixelValue;

      }



}
```

```
/*****************************************************************************
************
* Name                  :       main
* Input parameters      :       void
* Return type           :       int  -  Successful execution of program
* Description           :       Main loop where program begin
*
*****************************************************************************
************/
int main()
{

	//Set the device to be used
	cudaSetDevice(0);

	// Read input sinogram image and display
	cv::Mat sinogramMat =
cv::imread("C:\\Users\\admin\\Desktop\\CTReconstruction\\InputData\\Matla
bSinogram\\Square\\Sinogram.jpg");
	cv::imshow("Sinogram original", sinogramMat);
	cvWaitKey(0);

	// Rotate the sinogram image to have projections along the rows
	sinogramMat = sinogramMat.t();
	cv::imshow("Sinogram rotated", sinogramMat);
	cvWaitKey(0);

	// Calculate total bytes to be allocated for storing a output
reconstructed image
	size_t bytes = sizeof(float) * sinogramMat.cols * sinogramMat.cols;

	// Allocate 1D float pointer equal to size of sinogram
	float *sinogram = new float[sinogramMat.rows*sinogramMat.cols];

	// Host and device pointer of the output reconstructed image
	float *houtput;
	float *doutput;



	//Allocate memory for reconstructed image in host and device
	houtput = (float*)malloc(bytes);
	cudaMalloc(&doutput, bytes);

	//Copy sinogram Mat to 1D float pointer
	for (int r = 0 , k=0; r < sinogramMat.rows; r++)
	{
		for (int c = 0; c < sinogramMat.cols; c++)
		{
			sinogram[k] = (uchar)sinogramMat.at<uchar>(r,c*3);
			k++;
		}
	}

	// Cuda array to bind with texture memory
	cudaArray_t cArray;

	// Format of the texture element
	cudaChannelFormatDesc channel;

	// Create channel description
```

```
	channel = cudaCreateChannelDesc<float>();

	// Allocate memory to cuda array in device for sinogram
	cudaMallocArray(&cArray, &channel, sinogramMat.cols,
sinogramMat.rows);

	// Copy sinogram image from host to device
	cudaMemcpyToArray(cArray, 0, 0, sinogram, sizeof(float) *
sinogramMat.cols * sinogramMat.rows, cudaMemcpyHostToDevice);

	// Set texture memory filter mode and address mode
	texReference.filterMode = cudaFilterModePoint;
	texReference.addressMode[0] = cudaAddressModeWrap;
	texReference.addressMode[1] = cudaAddressModeClamp;

	// Bind texture to cuda array specifying the channel
	cudaBindTextureToArray(texReference, cArray, channel);

	// Assign block size
	dim3 blockSize;
	blockSize.x = BLOCK_SIZE;
	blockSize.y = BLOCK_SIZE;

	// Assign grid size
	dim3 gridSize;
	gridSize.x = sinogramMat.cols/blockSize.x;
	gridSize.y = sinogramMat.cols/blockSize.y;

	// Adjust grid size based on size of sinogram image
	if(sinogramMat.cols % blockSize.x > 0){
		gridSize.x = gridSize.x + 1;
	}
	if(sinogramMat.cols % blockSize.y > 0){
		gridSize.y = gridSize.y + 1;
	}

	// Measure the computational time of kernel function
	cudaEvent_t start, stop;
	float elapsedTime;

	cudaEventCreate(&start);
	cudaEventRecord(start,0);

	// Kernel call
	backprojection<<<gridSize, blockSize>>>(doutput, sinogramMat.rows,
STEP_ANGLE, sinogramMat.cols);

	cudaEventCreate(&stop);
	cudaEventRecord(stop,0);
	cudaEventSynchronize(stop);

	cudaEventElapsedTime(&elapsedTime, start,stop);
	printf("Elapsed time : %f ms\n" ,elapsedTime);

	// Synchronize device to get output
	cudaDeviceSynchronize();

	// Unbind texture from cuda array
	cudaUnbindTexture(&texReference);

	// Copy reconstructed image from device to host
```

```
	cudaMemcpy(houtput, doutput, bytes, cudaMemcpyDeviceToHost);

	// Copy 1D reconstructed output image to Mat format
	cv::Mat reconstructed_float = cv::Mat(sinogramMat.cols,
sinogramMat.cols, CV_32F, houtput);

	// Convert float image to int to display
	cv::Mat reconstructed_int;
	double min, max;
	cv::minMaxLoc(reconstructed_float, &min, &max);
	if (min!=max)
		reconstructed_float.convertTo(reconstructed_int, CV_8U,
255.0/(max-min), -255.0*min/(max-min));

	// Resize the image to half to display
	resize(reconstructed_int, reconstructed_int,
cv::Size(reconstructed_int.cols/2, reconstructed_int.rows/2));
	cv::imshow("Reconstructed image", reconstructed_int);
	cvWaitKey(0);

	// Free memory
	free(sinogram);
	free(houtput);
	cudaFree(doutput);
	cudaFreeArray(cArray);

	return 0;
}
```