**Universit degli studi di Cassino e del Lazio Meridionale**

Pattern Recognition

# Parallel Processing Project Report

## CT Reconstruction: Filtered Back Projection

**Parallel Processing Systems**

***Authors:***

Pierpaolo Venditelli
Tewodros W. Arega
Abdullah Thabit Zohaib
Salahuddin

***Supervisor:***
Saverio DI VITO

# Contents

**1 Introduction** **2**
1.0.1 Hardware . . . . . 2

**2 Radon Transform** **3**
2.1 Parallelization Strategy . . . . . 4

**3 Filteration and Inverse FFT** **8**

**4 BackProjection** **12**

**5 Results and Conclusions** **15**
5.1 Results . . . . . 15
5.2 Conclusions . . . . . 18

# Chapter 1

# Introduction

Image reconstruction in Computed Tomography is a mathematical process that generates tomographic images from X-ray projection data acquired at many different angles around the patient. Image reconstruction has fundamental impacts on image quality and therefore on radiation dose. For a given radiation dose it is desirable to reconstruct images with the lowest possible noise without sacrificing image accuracy and spatial resolution. Reconstructions that improve image quality can be translated into a reduction of radiation dose because images of the same quality can be reconstructed at lower dose. [1] This is an important operation and it both need to be effective and fast. As a group, we thought that parallelize the operation of image reconstruction via Filtered Back Projection would be a good idea for practicing with the skills acquired during the Parallel Processing Systems course.
The approach we had was divided in three:

- Study the mathematical theory behind the Filtered Back Projection
- Divide into small tasks the architecture
- Apply Parallel Programming for each tasks

### 1.0.1 Hardware

For the developing of this project we used one single laptop, having the following specifications:

* CPU i7 8750H, 2.2GHz to 4.1GHz, 6 cores + 6 with hypertreading
* 16 GB Ram DDR4
* NVidia GeForce RTX 2060, 6GB VRAM DDR6, 1920 Cuda Cores
* Ubuntu 18.04 LTS
* Cuda Version 10.0

The rest of the paper is divided as following: *Section II* introduces the concept behind the **Radon Transform**. *Section III* discuss the **Filtering and Convolution** phase while *Section IV* gives an overview on the **Back Projection** to obtain the final image. In the end *Section V* shows the **Results** and gives **Conclusion**. Each of the sections will have a mathematical overview of the task and the discussion on how to parallelize it with eventually the code.

# Chapter 2

# Radon Transform

The acquisition of the subject in Computed Tomography is a multidimensional inverse problem where the goal is to give an estimation of a system (usually the patient) through a finite number of projections (represented by the parallel beam passing through the patient in different angles).[link wikipedia (tomographic reconstruction)]

Applying the Radon transform on an image f(x,y) for a given set of angles can be thought of as computing the projection of the image along the given angles. The resulting projection is the sum of the intensities of the pixels in each direction, i.e. a line integral. There are two distinct Radon transforms. The source can either be a single point (fan beam) or it can be an array of sources (parallel beam). In this project, we are implementing CT reconstruction based on parallel beam x-rays.

The Radon transform is a mapping from the Cartesian rectangular coordinates $(x, y)$ to a distance and an angel $(r, \theta)$, also known as polar coordinates. The result of radon transform is called Sinogram.

$$R(r,\theta) = \int_{-\infty}^{+\infty} f(rcos(\theta) - zsin(\theta), rsin(\theta) - zcos(\theta))dz$$

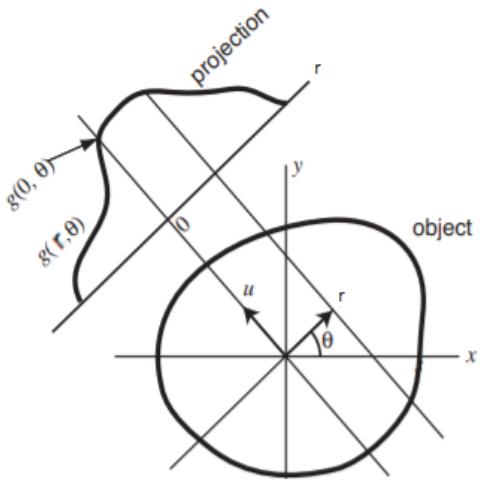


Figure 2.1: Radon Transform

## 2.1 Parallelization Strategy

Radon transform involves the sum of intensity values of the pixels in a projection line at a given raduis $r$ and angle $\theta$ as can be seen from the figure 2.1. In case, a point in the line resides outside image pixels or between the pixels, we used bilinear interpolation to interpolate the point to its four nearest pixels. If the size of original image is $n$ by $m$, then the size of sinogram will be $t$ by $nr$ (where $t$ is the range of angles and $nr$ is the number of sensors). In total, we will have a line integral of $t * nr$ projections.

Implementing radon transform serially will take a lot of time because the number of line integrals are very high particularly if the original image size is large. To make the radon transform computation faster, we parallelized radon transform by assigning one thread for each computation of a pixel value in the sinogram or for each line integral of a projection. In each parallel thread, a for loop was used to iterate along the integration axis and perform summation of their pixel values. Bilinear interpolation was used to interpolate image values on the grid defined by the detector and integration axes.

The *sinogram kernel()* function we created is structured as follows:

**Algorithm 1** Sinogram Function

- Allocate GPUś global memory
- Copy image data from host to device
- Launch *radontransform* kernel
- Allocate host memory for the output
- Copy data from device to host
- Free memory

The kernel we used for this function is a simple rotation kernel. It applies bilinear interpolation in order to get the coordinates of the image in which to calculate the line integral, which is the sum of all the intensities along that line. Here the code for the kernel.

Listing 2.1: Sinogram Kernel

```
/*
 * Kernel Name : sinogram_kernel
 ********************************************************
 * This kernel takes in the filtered sinogram and backprojects it.
    BackProjection
 * is calculated using the formula of inverse radon transform.
 ********************************************************
 * angles: int : total number of angles in the sinogram
 * sensors : int : number of sensors in the sinogram.
 * theta: float *: the values of theta at which the image
    projections are required
                  in the sinogram
 * x_min: float : this is minimum pixel distance in the x direction
```

```
 * y_min: float : this is the minimum pixel distance in the y
    direction
 * r_min: float : this is the minimum pixel distance along the
    diagonal direction
 * dx: float : the x pixel width
 * dy: float: the y pixel width
 * dr: float : the pixel diagonal length
 * Width: int : the width of the original image to be reconstructed
 * Height: int: the height of the original image to be
    reconstructed
 * img: float *: the image whose sinogram needs to be calculated
 * sinogram_output: float *: the sinogram that has been calculated
 ************************************************************
 * Returns void
 */

__global__ void sinogram_kernel(float * img,float* sinogram_output,
    float dx, float x_min, float dy, float y_min, int sensors,
   float dr, float r_min, int angles, float* theta,int Width,int
   Height)
{
    unsigned int sensor_no = blockIdx.x * blockDim.x + threadIdx.x;
    unsigned int angle_no = blockIdx.y * blockDim.y + threadIdx.y;

    if (sensor_no < sensors && angle_no < angles)
    {
        float sum = 0;
        float r = sensor_no * dr + r_min;
        int ind_x,ind_y;
        float d00,d11,d10,d01;
        float a,b;
        float result_temp1,result_temp2;
        for (int z_idx = 0; z_idx < sensors; z_idx++)
        {
            float z = z_idx * dr + r_min;


            // Transform coordinates------from r, theta to x, t
               ----------------------------------------------
            float r_real = (r * cosf(theta[angle_no]) + z * sinf(
               theta[angle_no]) - x_min)/dx + 0.5f;
            float z_real = (z * cosf(theta[angle_no]) - r * sinf(
               theta[angle_no]) - y_min)/dy + 0.5f;

            //BILINEAR INTERPOLATION START
               if ((r_real<Width)&&(z_real<Height))
               {



                   ind_x = floor(r_real);
                   a     = r_real-ind_x;

                   ind_y = floor(z_real);
                   b     = z_real-ind_y;

                   if (((ind_x)   < Width)&&((ind_y)   < Height))
                         d00 = img[ind_y*Height+ind_x];   else d00
                      = 0;
                   if (((ind_x+1) < Width)&&((ind_y)   < Height))
```

```
                        d10 = img[ind_y*Height+ind_x+1]; else d10
                    = 0;
                if (((ind_x)   < Width)&&((ind_y+1) < Height))
                        d01 = img[(ind_y+1)*Height+ind_x];   else
                    d01 = 0;
                if (((ind_x+1) < Width)&&((ind_y+1) < Height))
                        d11 = img[(ind_y+1)*Height+ind_x+1]; else
                    d11 = 0;

                 result_temp1 = a * d10+ (-d00 * a + d00);


                 result_temp2 = a * d11 + (-d01 * a + d01);
                 sum += b * result_temp2 + (-result_temp1 * b +
                    result_temp1);

            }

        //  BILINEAR INTERPOLATION END

    }

    sinogram_output[angle_no*sensors + sensor_no] = sum;
  }
}
```

The kernel is called through this line:

```
dim3 dimBlock(16, 16, 1); //256 threads per block
dim3 dimGrid((numSensors + dimBlock.x - 1) / dimBlock.x, (numAngles
    + dimBlock.y - 1) / dimBlock.y, 1);

rotationKernel<<<dimGrid, dimBlock>>>(d_img,d_sg, dx, minX, dy,
   minY, numSensors, dr, minR, numAngles, d_angles,numX,numY);
```

After applying this function to our original image we get the following results:

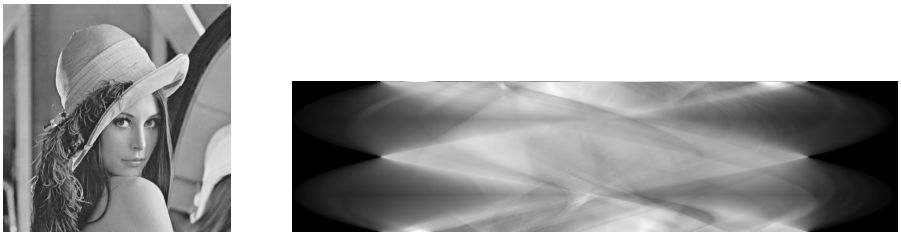

Figure 2.2: Original Image(512*512) and its Radon Transform

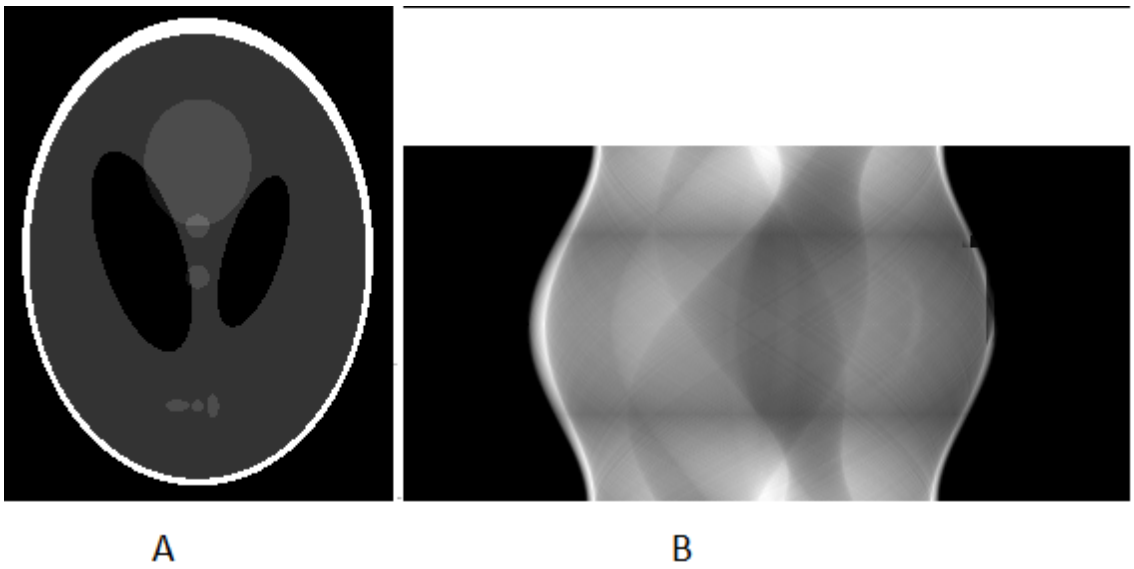


Figure 2.3: Original Image(256*256)(A) and its Radon Transform(B)

Here is a table comparing the elapsed time of radon transform using serial and parallel implementation for two different sized images.

Table 2.1: Elapsed time of Radon Transform

| **Image Size** | **Serial** | **Parallel** |
|---|---|---|
| 512 * 512: | 20473.5ms | 0.01406ms |
| 256 * 256: | 4977.76ms | 0.01502ms |

# Chapter 3

# Filteration and Inverse FFT

Once the sinogram has been generated from the Radon Transform, the next step to do in order to compute reconstruction is to filter it with a high pass filter. This step is crucial because otherwise the reconstructed image will be corrupted by the noise. The used filter is the ram lack filter.

The Ram- lack filter is a filter having the following behaviour:

$$A_1(\omega) = |\omega| \cdot \sqcap_L(\omega) = \begin{cases} |\omega| & \text{if } |\omega| \leq L, \\ \\ 0 & \text{if } |\omega| > L. \end{cases}$$

The sinogram in input is indeed an image in the cartesian space with the difference that each column of the matrix represent an angle in range [0,179] and each row represent an integral line.

To complete the reconstruction Back Projection and the Interpolation are the two last step after filtering. Back Projection simply takes in input the filtered sinogram and rotates it by 180 and for each degree it calculate the sum rowise of the image. This sum is then interpolated and the image is reconstructed. The interpolation methods used are two: bilinear interpolation and Nearest neighbour interpolation. It is defined for some discrete function $g$ as:

$$\mathscr{I}_W(g)(x) = \sum_m g(m) \cdot W\left(\frac{x}{d} - m\right) \qquad \text{for } -\infty < x < \infty.$$

If $W$ weighting function is square wave function,
i.e.

$$\sqcap_{1/2}(x) \;\; = \;\; \begin{cases} 1 & \text{if } |x| < 1/2, \\ \\ 0 & \text{if } |x| > 1/2. \end{cases}$$

then it is called *nearest neighbor* interpolation.
If $W$ is tent function,
i.e.

$$\bigwedge(x) \;\; = \;\; \begin{cases} 1 & \text{if } |x| \leq 1, \\ \\ 0 & \text{if } |x| > 1. \end{cases}$$

then it is called *linear* interpolation.

As a group we came up with a function that does all this job receiving the sinogram calculated in the previous part. The following pseudo code summarizes the

**Algorithm 2** backprojection()

Allocate space

Copy data from host to device

Calculate FFT (cufftc2c, forward)

Invoke kernel for **ramp filter**

Calculate inverse FFT (cufft c2c, inverse)

Invoke kernel to get the **real part**

Copy data from device to host

Free memory

main parts of the function to complete the back projection.

This part is more complex than the previous part, It is composed from **3** kernels: The **rampFiltKernel** filters the FFT calculated on the sinogram obtained from *sinogram()*. The Kernel for ramp filtering uses a ram lack filter. This filter is an high pass filter since we want to enhance high frequencies coming from the FFT and reduce the noise contained in the lower frequencies. The code is the following:

Listing 3.1: filterationkernell

```
/*
 * Kernel Name : filterationkernel
 ************************************************
 * This kernel takes in the fft of the sinogram and multiplies with
     the
 * ramlak filter. This is intended to suppress the low frequencies
    and intensify
 * the high frequency content.
 ************************************************
 * filter_subject: cufftComplex* : This the fft of the sinogram
    that needs to be
                                        filtered.
 * sensors : int : number of sensors in the sinogram.
 * angles : int : number of angles in the sinogram.
 ************************************************
 * Returns void
 */

__global__ void filterationkernel(cufftComplex* filter_subject, int
    sensors, int angles)
{
    unsigned int sensor_no = blockIdx.x * blockDim.x + threadIdx.x;
    unsigned int angle_no = blockIdx.y * blockDim.y + threadIdx.y;

    if (sensor_no < sensors && angle_no < angles)
    {
        filter_subject[sensor_no + sensors*angle_no].x *= ((
           sensor_no< sensors - sensor_no) ? sensor_no :(sensors -
           sensor_no)) / (float)sensors;
        filter_subject[sensor_no + sensors*angle_no].y *= ((
           sensor_no< sensors - sensor_no) ? sensor_no :(sensors -
           sensor_no)) / (float)sensors;
    }
}
```

In this kernel we get 2 indexes, *ii* and *jj*, the first is representing the number of sensors while the second the number of angles.
In the if statement we construct the ramlack filter by taking the minimum between the two opposite points.

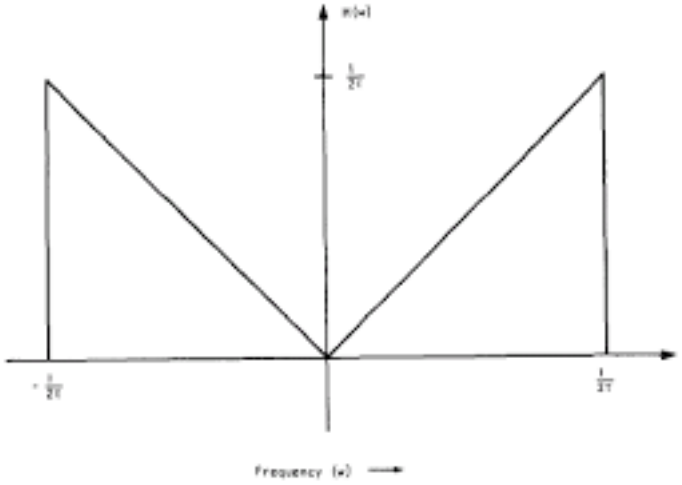


Figure 3.1: RamLack Filter

Listing 3.2: inversefft$_real$

```
/*
 * Kernel Name : inversefft_real
 ********************************************
 * This kernel takes in the inverse fft of the filtered sinogram
    and returns
 * only the real part of the inverse fft.
 ********************************************
 * filter_subject: cufftComplex* : This the fft of the sinogram
    that needs to be
                                    filtered.
 * sensors : int : number of sensors in the sinogram.
 * angles : int : number of angles in the sinogram.
 ********************************************
 * Returns void
 */

 __global__ void inversefft_real(float* real_ifft, cufftComplex*
    ifft, int len_ifft)
{
    unsigned int index = blockIdx.x * blockDim.x + threadIdx.x;
    if (index < len_ifft)
    {
        real_ifft[index] = ifft[index].x;
        }
}
```

The second kernel takes only the real part of the inverse FFT. This is possible as **cufftComplex** is a data type that consists of interleaved real and imaginary components. In the end we BackProject and interpolate in order to reconstruct the image.

# Chapter 4

# BackProjection

After Sinogram Filtration, the image can be reconstructed by redistributing the signal measured in the projection space to the image space. This can be simply done by applying Inverse Radon Transform to the filtered Sinogram, which can be defined according to the following formula:

$$f(x,y) = \int_0^{\pi} \tilde{g}(xcos(\theta) + ysin(\theta), \theta) d\theta$$

Therefore, each pixel in $f(x,y)$ can be retrieved by integrating the projections at each angle $\theta$. A simple for loop over the angles can be used to perform this task. Giving the fact that pixel values are only dependent on the Sinogram values and independent from each other, this task can be parallelized.

A simple kernel was made such that each pixel in the reconstructed image was treated as a separate thread. In each thread, the integration (summation) over projection angle was performed by a for loop with 1D nearest neighbor interpolation for in between values as shown in Listing 4.1.Alogrithm 3 below shows how the kernel was invoked.

**Algorithm 3** backprojection()

Allocate memory for backprojection

Invoke kernel for **backprojection**

Copy data from device to host

Free memory

Listing 4.1: backprojection$_k$*ernel*

```
/*
 * Kernel Name : backprojection_kernel
 ***********************************************
 * This kernel takes in the filtered sinogram and backprojects it.
    BackProjection
 * is calculated using the formula of inverse radon transform.
 ***********************************************
 * angles: int : total number of angles in the sinogram
 * sensors : int : number of sensors in the sinogram.
 * theta: float *: the total number of thetas used.
 * x_min: float : this is minimum pixel distance in the x direction
```

```
 * y_min: float : this is the minimum pixel distance in the y
     direction
 * r_min: float : this is the minimum pixel distance along the
     diagonal direction
 * dx: float : the x pixel width
 * dy: float: the y pixel width
 * dr: float : the pixel diagonal length
 * Width: int : the width of the original image to be reconstructed
 * Height: int: the height of the original image to be
     reconstructed
 * filtered_sinogram: float *: filtered sinogram of the image
 * output_recon: float *: this is the reconstructed image that is
     to be used
 ************************************************
 * Returns void
 */

 __global__ void backprojection_kernel(int angles,int sensors,
    float *theta,float x_min, float dx, int Width,float y_min,
    float dy, int Height,float r_min, float dr, float *output_recon
    ,float *filtered_sinogram)
{
    unsigned int x_index = blockIdx.x * blockDim.x + threadIdx.x;
    unsigned int y_index = blockIdx.y * blockDim.y + threadIdx.y;

    if (x_index < Width && y_index < Height)
    {
        float x, y, unscaled_r;
        float sum = 0;
        int sensor_index;

        x = x_min + x_index * dx;
        y = y_min + y_index * dy;

        for (int theta_idx = 0; theta_idx < angles; theta_idx++)
        {
            unscaled_r = x*cosf(theta[theta_idx] * M_PI / 180.0f) +
                y*sinf(theta[theta_idx] * M_PI / 180.0f);
            sensor_index = (unscaled_r - r_min) / dr ;
            sum +=  filtered_sinogram[theta_idx*sensors +
               sensor_index];
        }
        output_recon[x_index + Width * y_index] = sum;
    }
}
```

The excution of this kernel results in a reconstructed image of the original one, Figure 4.1 shows a sample image along with its reconstruction.

Table 4.1 below shows a performance comparison between a serial MATLAB implementation and a CUDA parallel implementation of back projection. Two image sizes were tested: 512x512 and 256x256, where the gain in performance out of parallelization was only noticed with the image 512x512. It is clear that data transfer from host to device and then back to host is the bottleneck here; therfore, for small images parallelization would be inefficent, such was the case for the image of size 256x256 .

Figure 4.1: Lena before (left) and after Reconstruction (right)

Table 4.1: Elapsed time of Back Projection

| **Image Size** | **Serial** | **Parallel** |
|---|---|---|
| 512 * 512: | 0.545s | 0.267 s |
| 256 * 256: | 0.175s | 0.74 s |

# Chapter 5

# Results and Conclusions

## 5.1 Results

The Program we developed was tested in comparison with a serial implementation.

| Implementation Type | Sinogram | Filteration | BackProjection |
|---|---|---|---|
| **Serial (256x256)** | 0.49s | 0.1 ms | 0.18s |
| **Parallel (256x256)** | 0.094s | 0.008 ms | 0.207s |
| **Serial (512 x 512)** | 2.04s | 0.25 ms | 0.55s |
| **Parallel (512 x 512)** | 0.119s | 0.01 ms | 0.309s |

Figure 5.1: Comparison of Serial and Parallel Implementations for two Images

As we can see our implementation gets significant better results compared with the serial implementation, specially during the radon transform in which gets around 206 times faster. We Used the Nividia Occupancy Calculator to calculate the occupancy graphs for our GPU. The following figures illustrate the occupancy graphs of our GPU.

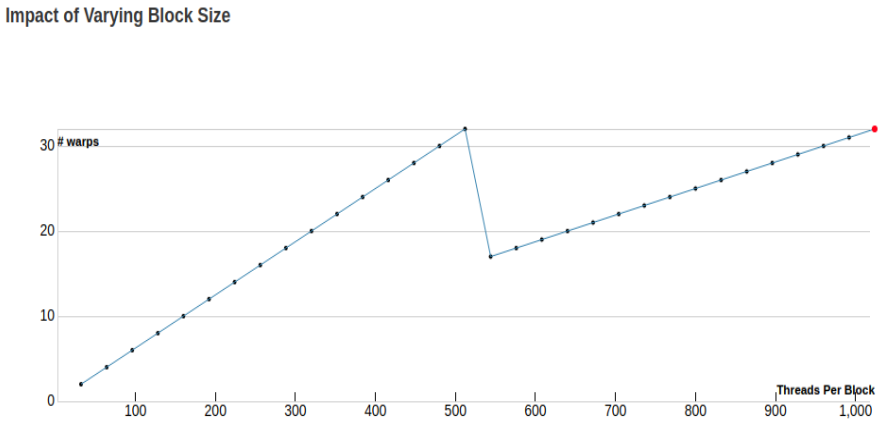


Figure 5.2: Occupancy Plot of Block Size for RTX 2060.The red point represents the upper limit.

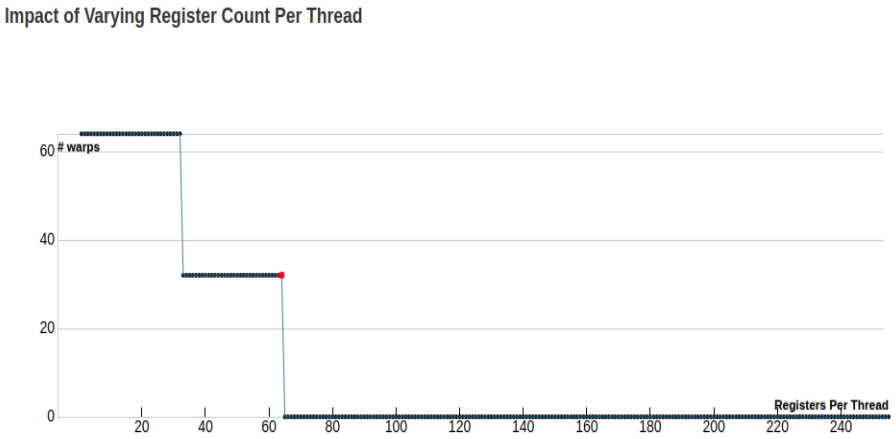


Figure 5.3: Occupancy Plot of Register Count for RTX 2060.The red point represents the upper limit.

We used Nvidia Visual Profiler in order to calculate the run time device meterics for our kernels. The Nvidia Visual Profiler gave us the following results for the sinogram and backprojection kernels.

| backprojKernel(int, int, float*, float, float, int, float, float, int, float, float, float*, float*) | |
|---|---|
| Queued | n/a |
| Submitted | n/a |
| Start | 566.65084 ms (566,650,843 ns) |
| End | 568.82326 ms (568,823,264 ns) |
| Duration | 2.17242 ms (2,172,421 ns) |
| Stream | Default |
| Grid Size | [32,32,1] |
| Block Size | [16,16,1] |
| Registers/Thread | 21 |
| Shared Memory/Block | 0 B |
| Launch Type | Normal |
| ▼ Occupancy | |
| Theoretical | 100% |
| ▼ Shared Memory Configurati | |
| Shared Memory Executed | 32 KiB |
| Shared Memory Bank Size | 4 B |
| ▼ Dependency Analysis | |
| Time on Critical Path | 2.17242 ms (2,172,421 ns) |
| Waiting Time | 0 ns |

Figure 5.4: Nvidia Visual Profiler Results for backprojection Kernel.

| sinogramKernel(float*, float*, float, float, float, float, int, float, float, int, float*, int, int) | |
|---|---|
| Queued | n/a |
| Submitted | n/a |
| Start | 178.19502 ms (178,195,021 ns) |
| End | 207.97198 ms (207,971,981 ns) |
| Duration | 29.77696 ms (29,776,960 ns) |
| Stream | Default |
| Grid Size | [91,12,1] |
| Block Size | [16,16,1] |
| Registers/Thread | 26 |
| Shared Memory/Block | 0 B |
| Launch Type | Normal |
| ▼ Occupancy | |
| Theoretical | 100% |
| ▼ Shared Memory Configurati | |
| Shared Memory Executed | 32 KiB |
| Shared Memory Bank Size | 4 B |

Figure 5.5: Nvidia Visual Profiler Results for sinogram Kernel

We investigated the variation of the block size on the time taken by each kernel

to complete. This behaviour was investigated both for the backprojection and the sinogram kernels. The following graph illustrates the variation.

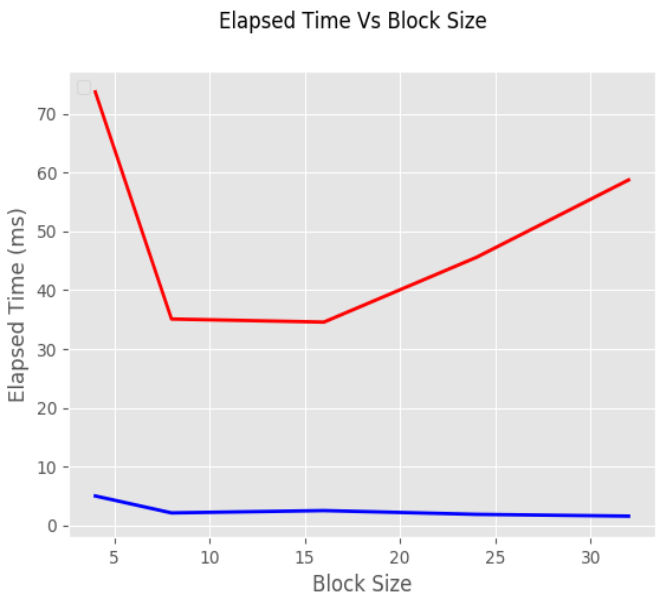


Figure 5.6: Block Size Variation Effect on Sinogram(blue) and BackProjection(red) kernels.

In the end, in order to compensate for the time delay because of the use of global memory in the kernels, we investigated the use of texture memory for the sinogram. The read-only texture memory space is cached. Therefore, a texture fetch costs one device memory read only on a cache miss; otherwise, it just costs one read from the texture cache. 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. [1] The following improvements in the execution time were seen after the use of texture memory for the sinogram kernel.

**sinogram_kernel(float*, float*, float, float, float, float, int, float, float, int, float*, int, int)**

| | |
|---|---|
| Queued | n/a |
| Submitted | n/a |
| Start | 144.88335 ms (144,883,350 ns) |
| End | 175.9536 ms (175,953,598 ns) |
| Duration | 31.07025 ms (31,070,248 ns) |
| Stream | Default |
| Grid Size | [91,12,1] |
| Block Size | [16,16,1] |
| Registers/Thread | 26 |
| Shared Memory/Block | 0 B |
| Launch Type | Normal |
| ▼ Occupancy | |
| Theoretical | 100% |
| ▼ Shared Memory Configurati | |
| Shared Memory Executed | 32 KiB |
| Shared Memory Bank Size | 4 B |

Figure 5.7: Nvidia Visual Profiler Results for Sinogram Kernel with Global Memory.

sinogram_kernel(float*, float*, float, f...

| | |
|---|---|
| Queued | n/a |
| Submitted | n/a |
| Start | 158.41117 ms (158,411,167 ns) |
| End | 183.64164 ms (183,641,638 ns) |
| Duration | 25.23047 ms (25,230,471 ns) |
| Stream | Default |
| Grid Size | [91,12,1] |
| Block Size | [16,16,1] |
| Registers/Thread | 23 |
| Shared Memory/Block | 0 B |
| Launch Type | Normal |
| ▼ Occupancy | |
| Theoretical | 100% |
| ▼ Shared Memory Configurati | |
| Shared Memory Executed | 32 KiB |
| Shared Memory Bank Size | 4 B |

Figure 5.8: Nvidia Visual Profiler Results for Sinogram Kernel with Texture Memory

## 5.2 Conclusions

During the development of this project, we learnt how to tackle the parallelization of a real world problem like CT Reconstruction. We investigated how the variation of block size, number of threads effects the execution performance. We further explored how to overcome the loss of time during the memory access of the global memory by using texture memory. Shared memory could not be used in this regard because of no sharing of data in the respective calculation of the thread. Hence, for this reason, the texture memory was the obvious better choice. We built on this understanding by learning what limits the performance of our algorithm using the GPU. We learnt how to profile our kernels using the Nvidia Visual Profiler.

# Bibliography

[1] Texture Memory Basics
http://cuda-programming.blogspot.com/2013/02/texture-memory-in-cuda-what-is-texture.html

[2] Matlab Code for Comparison
https://github.com/rehmanali1994/easy_computed_tomography.github.io/tree/master/

[3] CT Reconstruction Theory
http://www.guillemet.org/irene/articles