2013

X-ray CT on the GPU

Ravikiran Tadepalli
Iowa State University

X-ray CT on the GPU

by

Ravikiran Tadepalli

A thesis submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Major: Computer Science

Program of Study Committee:
Joseph Nahum Gray, Co-Major Professor
Oliver Eulenstein, Co-Major Professor
Leslie Miller

Iowa State University

Ames, Iowa

2013

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Nondestructive testing (NDT) is a collection of analysis techniques used by scientists and technologists as a way of analyzing the interior of an object without damaging the object. Since the analysis is done without damaging the object, NDT is an extremely valuable technique used in various industries for troubleshooting and research. CNDE has a long history of working with a variety of industrial sectors which include ***Aerospace*** (commercial and military aviation) ***and Defense Systems*** (ground vehicles and personnel protection); ***Energy*** (nuclear, wind, fossil); ***Infrastructure and Transportation*** (bridges, roadways, dams, levees); and ***Petro-Chemical*** (offshore, processing, fuel transport piping) to provide cost-effective tools and solutions. X-ray tomography is the procedure of using X-rays for generating tomographic slices of the required object. The object is bombarded with X-rays and the scanned image intensity values are collected on a detector. A significant drawback in X-ray tomography is the amount of data collected. It is generally huge in the order of gigabytes and hence the processing of data presents a big challenge. One way to speed up the processing of data is to run the programs on a cluster. CNDE uses a 64 node Beowulf cluster to do the reconstruction of an image. However with the advent of the GPU (Graphic Processing Unit) we have a far more cost efficient and time efficient hardware to run the reconstruction algorithm. The GPU can be fitted into a single PC, costs 10 times less than the cluster and also has a longer life time. This thesis has two major components to it. One of it is the desvelopment of new preprocessing and post processing techniques (includes filters, hot pixel removal etc.) to improve the quality of the input data and the other is the implementation of these techniques as well as the reconstruction program on the GPU using CUDA. Speedup on the GPU is not just a matter of porting the developed algorithms in parallel onto the hardware like in a cluster. GPU architecture is extremely complex and involves the usage of many different types of memory each with its own advantages and disadvantages and also many other optimization techniques for accessing and processing the data. These new techniques as well as the introduction of GPU are a significant addition to X-ray program here at CNDE.

CHAPTER 1. INTRODUCTION

1.1 Introduction

X-ray computed tomography (CT) is a NDE technique of analyzing the interior of an object by bombarding it with X-rays from different directions and collecting the transmitted data on a detector. Since it was introduced in the 1970's, CT has become a very important tool in medical imaging as it allows doctors to view slices of internal organs of a patient. CT has been used in the imaging of many important organs like head (detection of tumors, hemorrhage etc.), lungs (detecting changes in the lung parenchyma), heart, abdomen etc. Apart from medical imaging it also finds extensive use in the area of material science (inspection of ceramics, castings etc.) and also aerospace industry (inspection of turbine blades, rocket motors etc.). Let us now see a brief history of X-ray CT as to when the major developmental milestones were made.

The mathematical foundation of tomography is an integral transform called the radon transform established by J.Radon [1] in the year 1917. Radon transform allows us to estimate the value of a function in a region of space if the line integrals are known for all the ray paths in the region. In X-ray CT the function to be determined is the x-ray attenuation coefficient over the object whereas the intensities sensed by the detector give the value of the line integrals. Hounsfield and Cormack [2, 3] received a Nobel Prize in 1979 in medicine for their X-ray brain scanner with a reconstruction time of 2 days. The next major advance in CT tomography is the development of Fourier weighted back projection developed by Ramachandran and Laxminarayan [5]. This allows us to get a very accurate cross sectional view of the object with a very few assumptions. Apart from the main advances mentioned above many other image preprocessing techniques and post processing techniques have been developed for improving the quality of the collected CT data.

A major hurdle in CT is the amount of data to be reconstructed, visualized and analyzed. A single multicore PC is not a good choice for handling data collected in the order of gigabytes. One major step forward was the introduction of a cluster of processors in the CT reconstruction process. This allowed all the steps in CT to be effectively parallelized onto the nodes of a cluster when computations on datasets are independent of one another. The CT

reconstruction which took days to complete on a single PC now just took minutes on a cluster. This thesis introduces the next big milestone in CT tomography which is the introduction of Graphics Processing Unit (GPU). A major part of this thesis discusses the implementation of algorithms on the GPU. The algorithms involved in all the stages of CT tomography like preprocessing, reconstruction and postprocessing are discussed in this thesis.

1.2 Motivation

This thesis has two main motivations to it. The first is the development of new algorithms in the preprocessing and post processing stages. These new algorithms give better results than some previously developed algorithms at CNDE for the same purpose. This includes ring artifact removal using a bilateral filter and hot pixel removal using a shifted sliding window. The second motivation is to port the newly developed algorithms as well as the entire reconstruction algorithm on to the GPU. This research involves finding methods to effectively use the GPU architecture so as to get the maximum speed up over the cluster.

1.3 Chapters

Chapter 2 outlines the previous work done in the fields of X-ray tomography as well as in the development of algorithms on the GPU which have guided the various methods used as part of this thesis.

Chapter 3 describes the real time radiographic inspection setup and also the various major stages involved in the CT process. A brief overview of the equipment used as well as the process done is given.

Chapter 4 gives a brief overview of the most important concepts involved in CUDA program development which include the memory model, hardware model, execution model and the best programming practices.

Chapter 5 discusses the implementation of some general purpose image processing techniques on the GPU. This includes the development of many widely used filters (like the median filter and the bilateral filter) on the GPU and also fine tuning their performance.

Chapter 6 discusses the development of new preprocessing and post processing techniques like ring artifact removal using bilateral filtering and hot pixel removal using a shifted sliding window and also describes their implementation on the CUDA enabled GPU.

Chapter 7 discusses the implementation of the entire reconstruction program on the GPU.

Chapter 8 summarizes the work, presents the conclusion and lists the future work to be done.

CHAPTER 2. LITERATURE REVIEW

This chapter will outline the previous work done in CT tomography by the x-ray group at CNDE and will also discuss the other research done regarding the topics discussed in this thesis. The discussion will focus on the previous research done in both the development of image processing algorithms and their implementation on the GPU.

Let us first review the available literature on the basics of CT tomography. The mathematical foundations of the Radon transform which forms the basis of CT tomography was laid down by Radon in his paper "On the Determination of Functions from Their Integral Values along Certain Manifolds" [1]. It describes how to estimate the value of a function in a region of space if the line integrals are known for all the ray paths in the region. The application of this transform to determine the variable x-ray absorption coefficient in two dimensions is described by Cormack in his revolutionary paper "Representation of a Function by Its Line Integrals, with Some Radiological Applications" [2]. Based on the results published by Cormack, the first CT scanner was built by Hounsfield and his team (described in detail in the paper "Computed transverse axial tomography, Description of the system" [3]). "Radiological Imaging: The Theory of Image Formation, Detection, and Processing" by Barrett and Swindell [4] describes the entire reconstruction process in detail. The Fourier weighted backprojection which is a major component of the reconstruction algorithm in this thesis is described in detail by Laxminarayan and Ramachandran in "Three-dimensional Reconstruction from Radiographs and Electron Micrographs: Application of Convolutions instead of Fourier Transforms" [5]. These were the major milestones in the field of X-ray tomography and the reconstruction methods resulting from all these developments were standardized needing little extension since then. The major focus from then on in CT tomography has been to increase the performance since datasets were becoming huge (in the order of terabytes) and considerable processing was required for these datasets.

Now we will discuss all the research done by the x-ray group here at CNDE leading to this thesis. The initial development of a fast and relatively cheap system for quick tomographic inspection by using an available real time radiography setup was done by

Vivekanand Kini [6]. The algorithm was implemented on a single PC using the filtered backprojection technique for cone beam projections. It involves collecting the projections of all the slices in the object on the detector by bombarding it with x-rays along all possible fan beams. The projection data is then obtained from the collected slice data and the required filters (smoothing and kernel) are applied in the frequency domain before filtered backprojection is done to get the reconstructed slices. The entire reconstruction algorithm takes 3 hours on a single PC for an image size of 900. This algorithm continues to be used at CNDE with little or no modifications except for the addition of new preprocessing and post processing techniques developed at a later stage. This algorithm is discussed in detail in chapter 7 so as to understand how the various stages of reconstruction operate and also determine how to effectively port each of the stages on to the GPU. In addition, beam drift correction as a preprocessing step based on the air position entered by the end user is also discussed by Kini.

The next major stage in the research done by the x-ray group at CNDE was the development of an integrated computer control system for X-ray material characterization by Lu [7]. In order to develop these control programs, device drivers and control classes were developed in C++ for several PC boards, including a motion control indexer (PC23 and AT6000), multi-channel analyzer (MCA) and frame grabber (DT3155). Each class encapsulates the necessary functions to control the underlying hardware, thus providing a tool for other programmers to rapidly develop new custom data acquisition programs. These classes have been continually updated to adapt to the most recent drivers and also to add more functionality to the motion control systems.

We now have a well-defined system for collecting and reconstructing the data. What is needed is a way to visualize and analyze the reconstructed data. Towards this end, a 3D data visualization system for the reconstructed image was developed by Fan [8]. This system allows us to effectively view all the slices in the reconstructed image and also analyze the image to collect the necessary features. In addition, Fan was also responsible for adding more functionality to the computer control system. The system developed until now is cost effective but not time effective.

As we have seen above the reconstruction algorithm developed by Kini on a single PC takes 3 hours for an image of size 900. In order to speed up the reconstruction algorithm, it was parallelized on a cluster. This implementation of the algorithm on the Linux cluster of 64 nodes was done by Zhang [9]. As a result the time for filtered backprojection was reduced from 3 hours to 15 minutes for an image size of 900. This is a very crucial step in the development of time efficient algorithms for reconstruction and was in usage at CNDE before the GPU was introduced as part of this thesis. The cluster has a very simple architecture as compared to the GPU and porting the reconstruction algorithm onto it is an example of an embarrassingly parallel problem. MPI was used to parallelize the algorithm on to a collection of 45 nodes by dividing the workload of reconstruction such that each of the nodes does filtered backprojection on one of the sinograms. Apart from this, Zhang also added some additional features to the 3D visualization program such as the histogram lookup table, region of interest clipping etc. which help the end user to analyze the reconstructed slices to collect some useful information.

In order to improve the quality of the input sinogram data and also the reconstructed slice data many preprocessing and post processing algorithms have been integrated since then into the reconstruction algorithm. Nadia [10] was responsible for adding preprocessing algorithms like ring artifact removal and hot pixel removal to the reconstruction algorithm. Both algorithms help improve the quality of the reconstructed image for image analysis by removing artifacts caused by defective detector pixels. The hot pixel identification and removal algorithm implemented here is based on using the error tolerance entered by the end user and the calculation of standard deviation of the individual pixels. Apart from this Nadia also added some additional features to the existing 3D visualization program like blob analysis and percentage porosity analysis. Blob analysis involves identifying blobs having the same intensity values, separating overlapping blobs using the erosion algorithm and regaining the lost information during erosion by a dilation algorithm. Percentage porosity analysis involves analyzing a sample by acquiring useful information such as concentration of material per cubic volume and also the internal structure of the sample. Some of the above algorithms (like ring artifact removal and hot pixel removal) have been ported onto the GPU

as part of this thesis. Other algorithms (like blob analysis) are also expected to be ported onto the GPU in the future.

This was the existing body of work at CNDE before the introduction of the GPU. This thesis introduces the development of basic image processing filters on the GPU (like median filter, bilateral filter etc. discussed in chapter 5), new image processing algorithms on the GPU (like ring artifact removal, hot pixel removal etc. discussed in chapter 6) and the implementation of the entire reconstruction algorithm on the GPU (discussed in chapter 7). As a result of the techniques introduced in this thesis a lot more image enhancement can be done on the reconstructed slices thereby enabling effective image analysis. Also the time for the reconstruction algorithm has been reduced from 15 minutes on the Linux cluster to a minute and half on the GPU thereby allowing a real time image analysis. The architecture of a GPU is far more complicated as compared to a cluster and hence is discussed in detail in chapter 4. The various stages involved in the development of the reconstruction algorithm on the GPU is discussed in detail in chapter 7.

We shall now review the main sources of information used during program development on the CUDA enabled GPU. “Programing massively parallel processors” [11] by David Kirk gives a good introduction to CUDA programming for beginners explaining all the programming constructs involved in detail. It explains the process and techniques involved in porting an algorithm onto the GPU using both basic algorithms like matrix multiplication to real time examples like molecular visualization. Two case studies are discussed which explain in fine detail all the performance enhancing techniques used for program development on the GPU. The NVIDIA programming guide [13] gives a good summary of all the core programming concepts involved including the programming model, the memory model and the hardware model. It also enumerates the various hardware and software limitations of GPU’s with different compute capabilities. NVIDIA cards with different compute capabilities have different resources (like amount of register memory, shared memory, constant cache, number of threads per block and so on) which need to be taken into consideration during programming a GPU. NVIDIA best practices [14] summarizes all the main performance enhancement techniques involved in program development on a GPU and separates them into high priority, medium priority and low

priority techniques. This is one of the most useful documents as far as program development on the GPU is concerned. These techniques guide the development of all the algorithms in this thesis and have been adhered to in a consistent manner.

Many standard API's have been used to ease the program development on the GPU. NVIDIA CUDA Runtime API [15] describes the runtime library of CUDA in detail by giving the signatures of all the functions involved at a single place. NVIDIA Math API [16] discusses the math library of CUDA in detail and has been used to optimize the performance of instructions during algorithm development. NVIDIA CUFFT library [17] discusses the various functions and their usage for the computation of the various Fourier transforms on the GPU platform. This library is used in various algorithms like COR detection and also the reconstruction algorithm.

Debugging, memory checking and performance analysis are important components of program development on the GPU. Debugging allows us to examine the intermediate values of a program during the step by step execution. The CUDA emulator provided by visual studio was used for debugging in the beginning stages of program development. NVIDIA CUDA compiler (NVCC) [19] was also used in the beginning stages of program development for performance tuning to study the resource allocation on the GPU. Setting some flags allows us to study the shared memory usage, register usage and so on. This information can then be used along with the CUDA GPU occupancy calculator to optimize the performance of a program. The Occupancy calculator [23] takes the compute capability and resource usage from the end user and returns the number of active blocks and active warps per multiprocessor. It also gives a graphical display of the resource consumption allowing the end user to study the effects of changing the resource allocation. NVDIA Profiler [22] was also used briefly before switching to the best performance analysis tool for the GPU as discussed below.

Since its release, NVIDIA Parallel Nsight 3.0 [21] has been a major component of program development here at CNDE. It allows for efficient debugging of heterogeneous applications using a well-designed GUI. It allows step by step debugging of an application allowing the end user to see all the intermediate state values of warps, threads, variable in all memories and so on. It also does memory checking and informs the end user during

invalid memory access. A visual profiler is also included to help in optimizing the GPU code performance and its usage has been demonstrated throughout the thesis. In the beginning stages of program development on the GPU, many problems were faced during integration of CUDA with visual studio. Parallel Nsight also simplifies the integration by automatically setting up the required paths and environment variables, installing the proper toolkit and also providing built in CUDA templates for program development in the visual studio environment making it extremely easy for a beginner to start programming in CUDA. This tool has been used for all the algorithms in this thesis for performance tuning of the kernels involved.

After developing some basic algorithms on the GPU using the above mentioned resources, the next main stage in this thesis involves the development of image processing algorithms on the GPU. "Digital Image Processing" by Gonzalez [24] provides a good introduction to all the basic concepts and methodologies used for digital image processing. It discusses both image enhancement in the spatial domain (used for the preprocessing and postprocessing algorithms) and image enhancement in frequency domain (used for the reconstruction algorithm). NVIDIA image convolution with CUDA [25] discusses convolution filtering in image processing and shows the implementation of a separable convolution filter on a CUDA enabled GPU. This gives some very efficient performance enhancing techniques for filtering on the GPU. Tomasi et al. [26] describes the bilateral filtering of gray and color images as a method to smooth images while preserving edges by means of a nonlinear combination of nearby image values. It forms the basis for the development of the bilateral filter used in the ring artifact removal algorithm discussed in this thesis. It is an extremely efficient edge preserving filter and can be ported very effectively onto the GPU because of the data independent nature of the computations involved. Much performance tuning has been done as part of the research conducted in this thesis. Similar performance tuning methods for CUDA accelerated neighborhood de noising filter involving pre computation of some parameters and also prefetching of some data values are discussed in Zheng et al. [27]. Further improvements can include the usage of texture memory to store some pre-calculated values as discussed by Stall [28]. Also a 3D bilateral filter can be

implemented and optimized as discussed in Bethel [29] with varying parameters such as different kernel configurations; different memory usage and also different thread block sizes.

Median filter is also a good candidate for ring artifact removal but it cannot be ported efficiently on to the GPU because of the sorting complexity involved. A median filter has been developed on the GPU as part of research but has not been found to be very time efficient algorithm. Further improvements to the median filtering algorithm involve filtering by using histogram bins mentioned in the GPU Computing Gems [30] . Perrrot [31] discusses one more novel method for median filtering on the GPU which can be integrated into the existing algorithm. Some other approaches for median filtering include high performance median filtering using commodity graphics hardware which provides a speedup of around 4 times over a 4-core CPU by Chen et al. [32] and also a GPU accelerated vector median filter by Aras et al. [33] which gives a speedup of 100X over a CPU implementation of the same algorithm. Apart from all these image enhancement techniques in the spatial domain mentioned above techniques for image enhancement in the frequency domain have also been developed like Nielsen [34] which discusses real time wavelet filtering on the GPU.

The next stage of research conducted in this thesis involves the development of new preprocessing and postprocessing algorithms and porting them onto the GPU. This includes ring artifact removal, hot pixel removal, beam drift correction and also the beam hardening algorithm. Many different approaches have been developed over time for ring artifact removal. They can be mainly classified into three categories namely methods based on adjusting the experimental setup, methods running artifact removal algorithms in the preprocessing stage on the sinogram and methods running artifact removal algorithms in the post processing stage on the reconstructed slice. Both the preprocessing and post processing techniques can be done in different geometric planes (polar or Cartesian). The preprocessing techniques developed in this thesis (both ring artifact removal and hot pixel removal) run on the input sinogram files in the Cartesian plane. The post processing techniques (like the beam hardening algorithm) run on the reconstructed slices in the Cartesian plane. The most commonly used methods in the first category are flat field correction method and movable detector array method. Yousuf et al. [35] discusses an efficient ring artifact reduction method based on projection data for micro CT images where the defective lines are corrected in the

image before the filtered back projection. Here a horizontal median filter is applied on the sinogram for the purpose of ring artifact removal. It is an efficient ring artifact removal method but ring artifact removal using bilateral filter discussed in this thesis gives better results and can be ported very efficiently onto the GPU. Munch et al. [36] describes a fast, stable and powerful method for stripe and ring artifact removal with combined wavelet and Fourier filtering. It describes ring artifact removal in the frequency domain rather than the spatial domain used for the algorithm in this thesis. A slightly modified approach has been developed before at CNDE and has not been found to be too effective and hence the new technique in the spatial domain. Kalender et al. [37] discusses a dedicated image based ring artifact correction for high resolution micro CT based on median filtering of the reconstructed image and working on a transformed version of the image in polar coordinates. Emraan et al [38] discusses methods to classify ring artifacts into different categories and use adaptive correction schemes so that different or slightly modified algorithms can be used for ring artifact removal. Chen et al. [39] discusses the acceleration of ring artifact correction for flat detector computed tomography using the CUDA framework. An implementation of the newly developed ring artifact removal algorithm on the GPU is also discussed in this thesis. Many different approaches have been implemented including using an efficient matrix transpose technique developed by NVIDIA.

The other major preprocessing algorithm implemented as part of this thesis is the hot pixel removal algorithm. Some early approaches to hot pixel removal and their potential problems are discussed by Hill et al. [40]. Nadia [10] is responsible for the development of the original hot pixel identification and removal algorithm at CNDE based on the calculation of standard deviation of the individual pixels and the error tolerance entered by the end user. This thesis identifies the defects of the algorithm proposed by Nadia and proposes an algorithm based on several new measures. These include measures for hot pixel removal in case of contiguous pixels, using a running mean instead of static mean for a more accurate measure and so on. More recent methods for detection and correction of hot pixels using statistical methods of data analysis were proposed by Robert [41].

Kini [6] describes beam drift correction as a preprocessing step in CT reconstruction based on input of estimated air position from the end user. This preprocessing step has also

been ported onto the GPU and gives a significant speedup when compared to the algorithm on the cluster.

The final stage in the reconstruction algorithm is the post processing stage composed of beam hardening. Herman et al. [42] discusses the beam hardening in computed tomography by the computation of the attenuation of a mono energetic X-ray beam from the attenuation of a poly energetic X-ray beam. Beam hardening is an important post processing step to improve the quality of the output image. This has also been implemented on a GPU at CNDE using the bilateral filtering and matrix transpose implementations on the GPU discusses in this thesis. Ruetsch et al. [43] discusses optimizing the matrix transpose on a GPU. Matrix transpose is an extremely important component of many of the algorithms developed here at CNDE like ring artifact removal, beam hardening etc. Doing it on the GPU by the technique called diagonal reordering gives a very high speed up equivalent to the matrix copy on the GPU and will be discussed further in this thesis.

The next stage of research conducted in this thesis involves implementing the cone beam reconstruction algorithm using filtered backprojection on the CPU. The dissertation by Kini [6] was used as a major guide to understanding the various components of the algorithm since the same algorithm continues to be used with little or no change at CNDE. Understanding all the major components of the algorithm and how they relate to each other is a very important step to efficiently port the algorithm onto the GPU. Many other papers also discuss improvements to cone beam reconstruction algorithm using filtered backprojection. Turbell [44] presents a novel method that gives images of better quality for the FDK method. He also presents an improvement over the existing cone beam backprojection using helical backprojection which gives faster results and better quality images. Wang [45] discusses some improvements to the existing cone beam reconstruction algorithm and also validates them using simulation. Mueller [46] discusses a different fast implementation of the algebraic methods used for 3D reconstruction. Handy [47] discusses 3D CT reconstruction via fan to parallel rebinning. The existing algorithm at CNDE has been rewritten as part of research. The code has been cleaned up and validation is done for all the inputs of the end user. Also the code has been divided into separate modules (like modules for preprocessing, postprocessing, reconstruction, COR Detection, sinogram operations, array operations etc.)

so as to facilitate easy integration of new components in the future. Lots of error checking and memory checking has been added to the existing code. Many of the newly developed algorithms have also been integrated into it.

Now that we have a robust reconstruction algorithm on the CPU we need to find ways to parallelize the algorithm in an efficient fashion. The algorithm developed by Kini [6] takes 3 hours for an image of size 900 on a uniprocessor PC and needs to be implemented in a faster manner for real time 3D image analysis. Zhang [9] ported the algorithm onto the Beowulf cluster at CNDE reducing the time needed to 15 minutes. Many other papers also discuss the parallelization of the reconstruction algorithm in detail. Li et al. [48] discusses parallel iterative CT cone beam reconstruction on a Linux PC cluster and reports the speedups and efficiency factors. Mukherjee [49] discusses a primitive implementation with only the backprojection steps speeded up leaving out the weighing and filtering steps. The code is implemented in both C and MATLAB on the CPU and in both CUDA and OPENCL on the GPU. These performances are analyzed and compared. Scherl [50] implements backprojection using the CUFFT library and also making each threads compute multiple voxels so as to reduce register usage and also computation time. Maier et al. [51] discusses a GPU accelerated CT reconstruction system which provides a speedup of 56 over a uniprocessor system. Okitsu et al. [52] discusses the acceleration of cone beam reconstruction using the many performance tuning techniques like memory coalescing, reducing global memory access by reusing the data, eliminating local memory usage and launching multiple kernels simultaneously. Schwarz et al. [53] discusses the texture unit as a performance booster in CT reconstruction on a Fermi GPU. The porting of the reconstruction algorithm in this thesis is completely unique and has been developed independently of other approaches as can be seen in the performance tuning stages done for the algorithm. Also it implements many novel methods for increasing performance and gives very good results.

Apart from the reconstruction algorithm, the COR Detection algorithm has also been ported onto the GPU. COR is a very important parameter involved in 3D CT reconstruction. Inaccurate estimates of COR can significantly hinder the quality of the final image. Many different methods for determining the center of rotation (COR) for the same type of tomographic scans are discussed in Donath et al. [54]. A new COR detection algorithm has

been developed by the X-ray group here at CNDE as part of the research. The porting of this algorithm is also discussed in chapter 7 along with reconstruction.

CHAPTER 3. STAGES IN X-RAY CT

We shall now briefly discuss each of the stages in X-ray CT. This gives a general overview as to how the entire CT process works and what is the functionality of each of the stages as well as the challenges faced in each of the stages. The major stages are as follows:

3.1 CT Data Acquisition

The CT system in use at CNDE is a high resolution, low power system which uses an amorphous silicon detector for image acquisition as well as a frame grabber to transfer the images to a cluster RAM for storage and display. The high resolution allows us to view fine details of small images and the low power enables us to view them at a short distance. Larger images do not fit in the field of view of this system and for those objects a medium resolution high power system is used. The process of collecting data is as shown below in Figure 3.1.

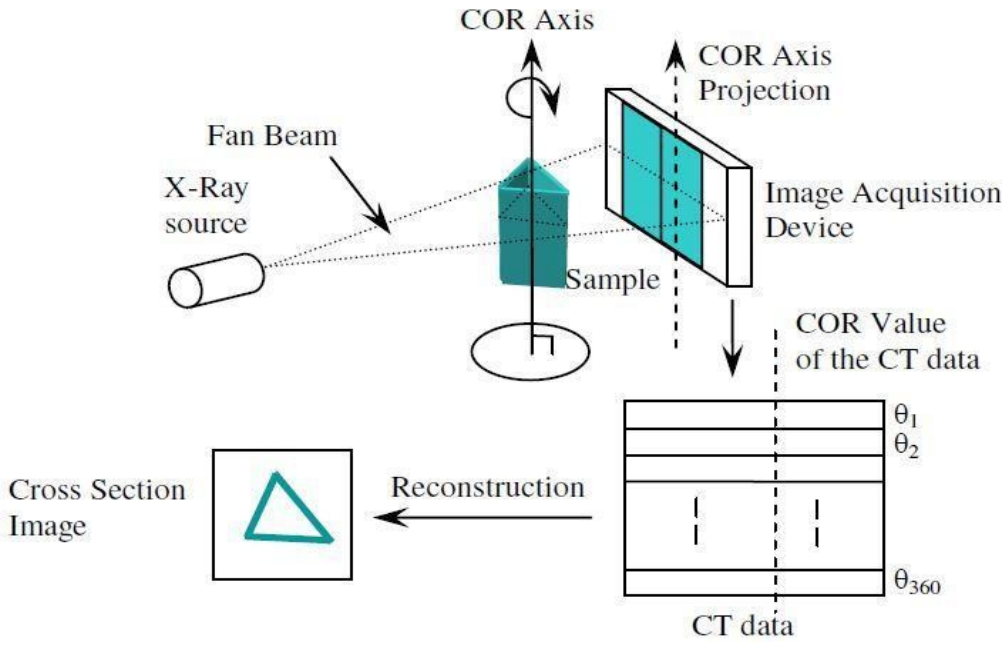


Figure 3.1 CT data acquisition

The object to be analyzed is bombarded with X-rays of high intensity from an X-ray source along all the possible fan beams and the object intensity values are collected on an amorphous silicon detector. The image acquisition device converts these intensity values into

data and stores them. As the X-rays pass through the sample it is absorbed or attenuated to different degrees depending on the geometry and the material of the sample. Each row on the detector collects the intensity values of a slice in the object. The object is then rotated by a small degree and the intensity values are again collected. Thus we have a row of intensity values for each angle of rotation for each of the slices in the object. For example let us say we rotate the object by a degree every time we collect the intensity values. So we have 360 different image intensity row values for a slice. These values are collectively called a sinogram. We have a sinogram for each of the slices in the object and these sinograms are then used to reconstruct the interior of the object using the filtered back projection algorithm. This back projection algorithm along with some preprocessing and postprocessing techniques is called the reconstruction program.

3.2 CT Data Handling

Once the data is acquired by the image acquisition device it is sent to the RAM of the master node on the CNDE Linux cluster by an FTP network connection. The CNDE Linux cluster is composed of the root node (HAL) and 64 other processors. A program “reassem” is launched on the cluster to reassemble all the binary files into standard ASCII sinogram files. These ASCII files are then stored on the hard drive at HAL.

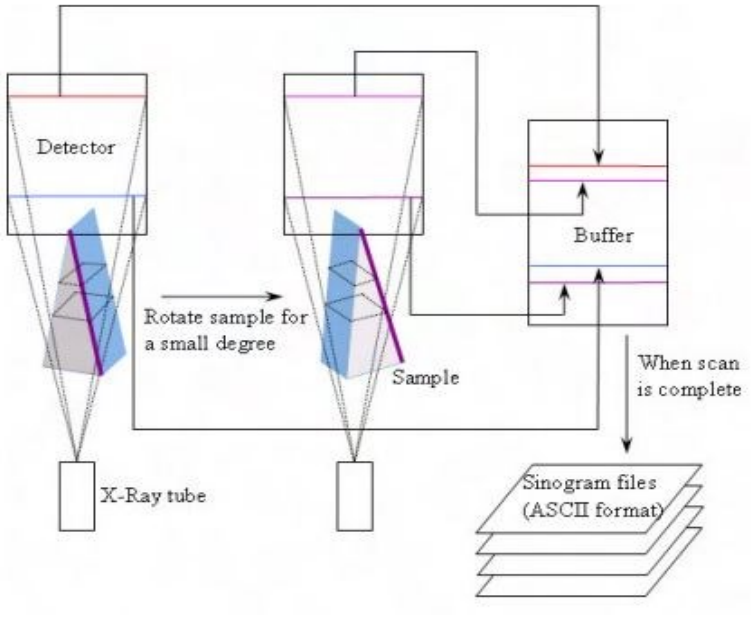


Figure 3.2 CT data handling

3.3 CT Data reconstruction

The sinogram files thus obtained need to be reconstructed to generate the 3D image. Each of the sinogram files when reconstructed using filtered backprojection generates a slice in the image. These 2D slices on reassembling give the 3D image as shown in the following figure.

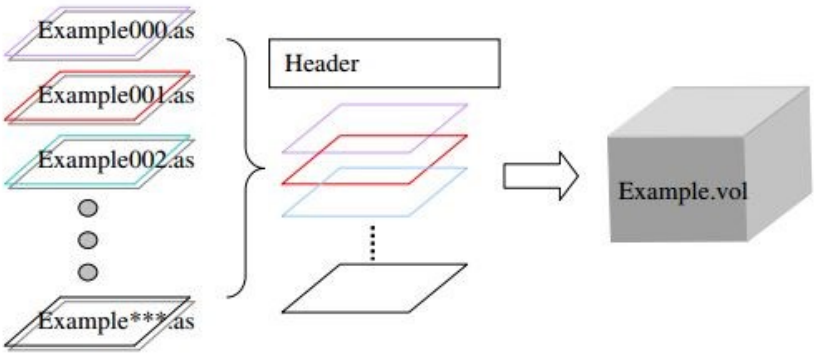


Figure 3.3 Forming the 3D image from 2D slices

The datasets collected for the purpose of CT scan are generally huge in the order of gigabytes. Therefore reconstructing them and also analyzing the reconstructed image is not feasible on a single PC. Presently at CNDE a cluster of 64 nodes is used to speed up the reconstruction by running the process in parallel. The below figure illustrates the concept of parallelization and the idea of sending job to cluster for parallel processing through Ethernet Connection.

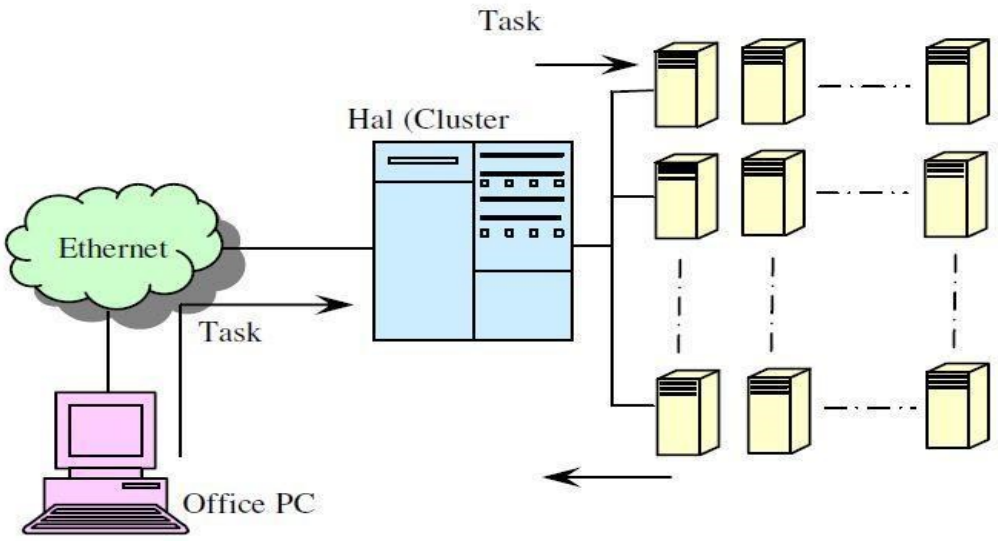


Figure 3.4 Parallel image reconstruction

3.4 3D Image Visualization

Visualization of data is an extremely important component of X-ray CT which allows us to examine the interiors of a sample in a convenient manner. Volume rendering allows us to project the 3D sample onto a 2D image. Towards this goal, visualization software was developed by Fan [8] in Visual C++ to visualize the CT data at CNDE. It allows us to rotate, zoom and effectively view the 3D data. A dolomite sample viewed using the 3D visualization program is shown in Figure 3.4

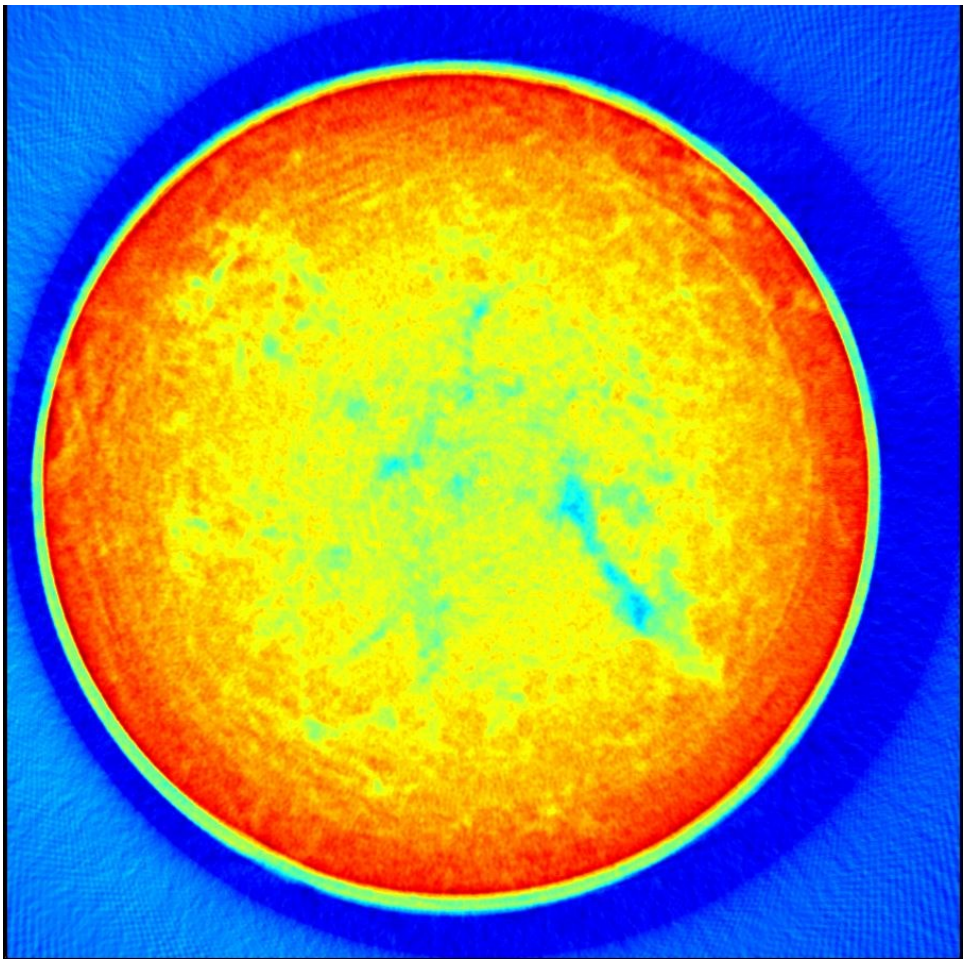


Figure 3.5 Dolomite sample using 3D visualization

Apart from 3D visualization many tools for 3D image analysis have also been developed here at CNDE and will be discussed in the next section.

3.5 3D Image Analysis

Many features have been added to the 3D visualization program which allows us to analyze the reconstructed data to collect useful information for the end user. The main such tools added by Zhang [9] include histogram lookup table, mapping, region of interest clipping, normalization, and transparency. The histogram lookup table of the dolomite sample is shown in Figure 3.5.

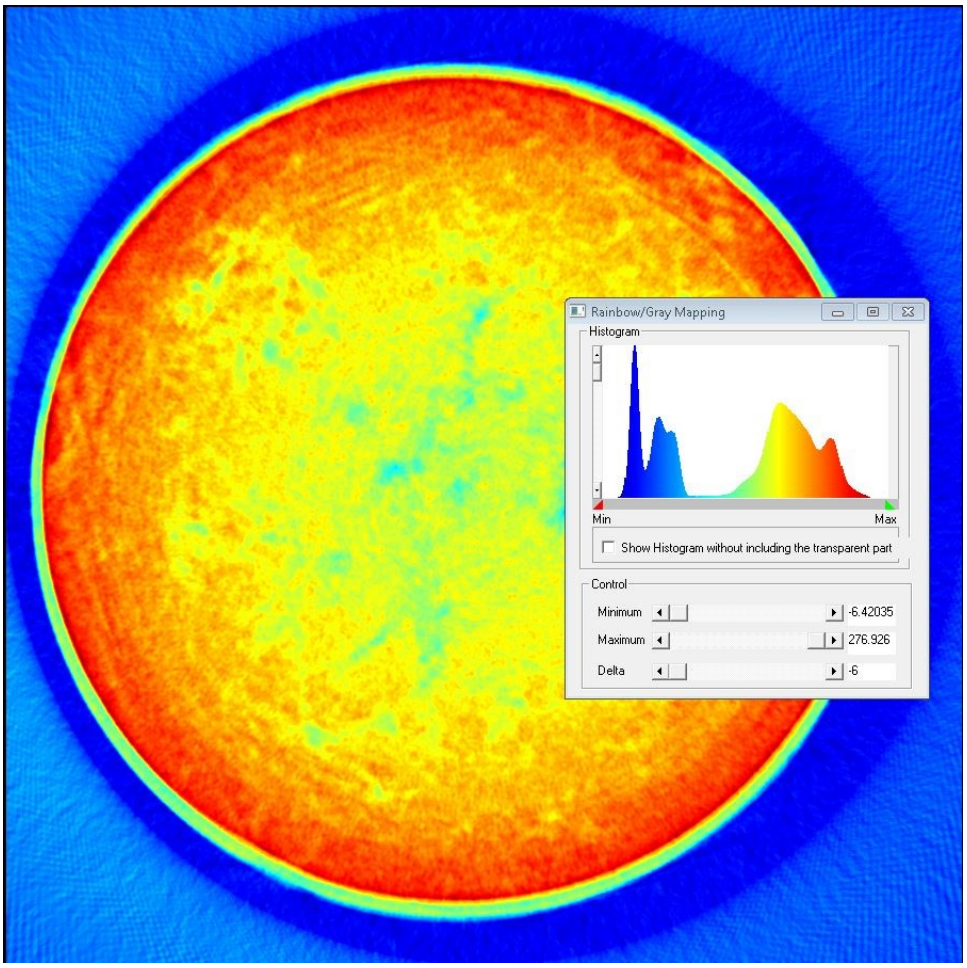


Figure 3.6 Histogram lookup table of the dolomite sample

A region of interest of the dolomite sample is shown in Figure 3.6.

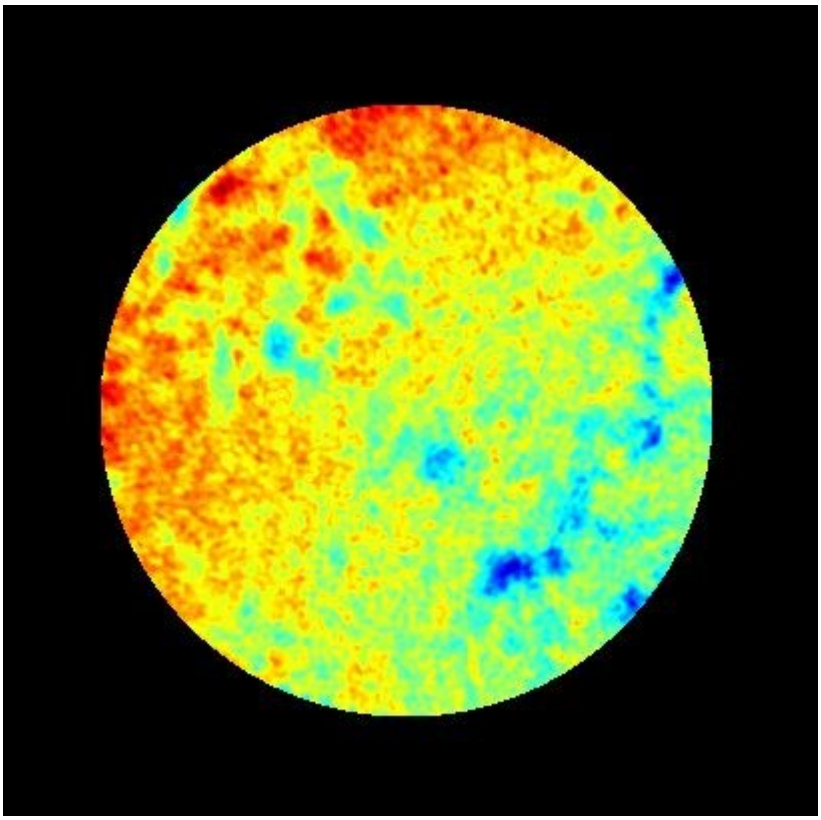


Figure 3.7 ROI of the dolomite sample

In addition, features like blob identification and percentage porosity analysis have been added by Nadia [10].

3.6 Image Enhancement

Many tools have also been developed to enhance the quality of the reconstructed image like all the preprocessing (filtering, ring artifact removal, hot pixel removal) and postpostprocessing (beam hardening) algorithms developed in this thesis. These algorithms need to be integrated into the 3D visualization program for effective usage.

CHAPTER 4. INTRODUCTION TO CUDA

4.1 Why GPU?

The main idea behind the speedup we get while running programs on a GPU as compared to a CPU is that the GPU is specifically designed for compute intensive, highly parallel computation. As compared to a normal CPU more transistors are devoted to data processing rather than data caching and flow control as shown in Figure 4.1. This type of architecture is especially suited for data parallel algorithms where the same arithmetic instruction is executed on many elements in parallel. The parallel data is handled by multiple threads with each thread doing the same arithmetic computation. GPU's are being increasingly used in fields as diverse as bioinformatics, computational chemistry, computational finance, databases, medical imaging, weather prediction, molecular dynamics etc.

Figure 4.1 CPU versus GPU

4.2 Programming model

As we know from the discussion above different threads execute the same function on different data elements. The common function executed by all the threads in CUDA is called a kernel. It is defined using __global__ declaration specifier and the configuration of threads associated with the kernel is specified in angular brackets <<<>>> during the kernel launch. During the kernel launch the threads are launched as a grid of one dimensional or two

dimensional thread blocks to facilitate the distribution of blocks across the GPU processor cores in different ways and also across any number of cores. The dimensions of the grid are specified as an integer vector type dim3 and passed as the first parameter of <<<>>> syntax. Inside the kernel code the different blocks in a grid can be accessed using the one-dimensional or two dimensional index of the variable blockIdx. The dimensions of the thread block are also specified as an integer vector type dim3 and passed as the second parameter of <<<>>> syntax. Inside the kernel code the different threads in a block can be accessed using the one-dimensional, two-dimensional or three-dimensional index of the variable threadIdx. The index of a thread and its thread ID are related in the following way. For a one-dimensional block, they are the same. For a two-dimensional block of size (D_x, D_y), the thread ID of a thread of index (x, y) is ($x + y * D_x$). For a three-dimensional block of size (D_x, D_y, D_z), the thread ID of a thread of index (x, y, z) is ($x + y * D_x + z * D_x * D_y$). An instance of a grid of thread blocks is shown in the Figure 4.2 with the grid composed of 6 blocks and each block composed of 12 threads. When programmed through CUDA, the GPU is viewed as a compute DEVICE capable of executing a very high number of threads in parallel. It operates as a coprocessor to the main CPU or HOST. Both the device as well as the host maintains their own DRAM referred to as host memory and device memory respectively. One can copy data from one DRAM to the other using the cudaMemcpy() function.

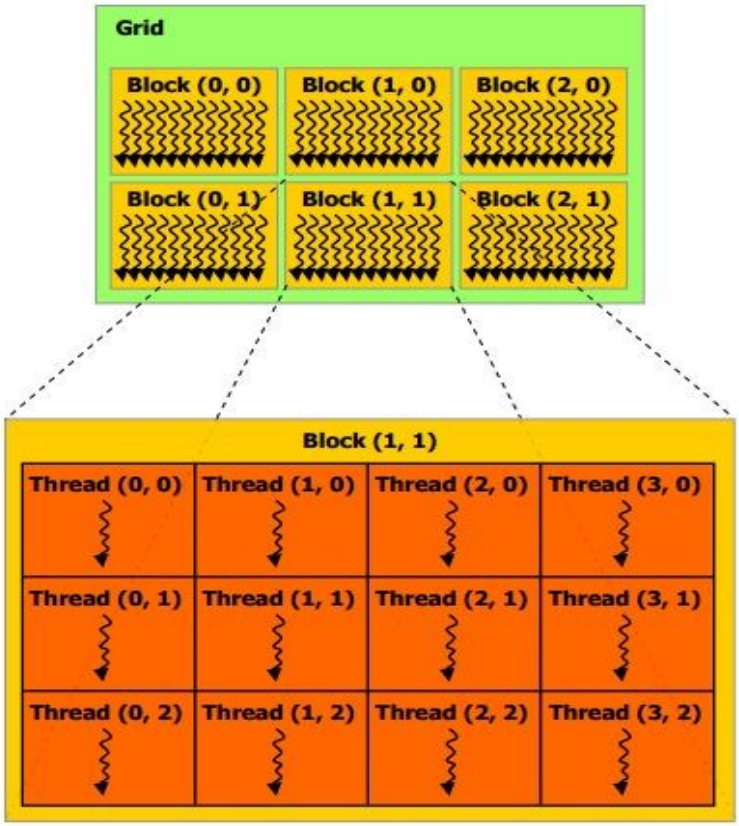


Figure 4.2 Grid of thread blocks

4.3 Memory model

The GPU has many different kinds of memories with different latencies and each having their own unique properties. The memory model of the GPU is as shown in the Figure 4.3. Efficient access and management of the different memory types is a very important aspect of CUDA program development. We shall now discuss each of the memory types in detail.

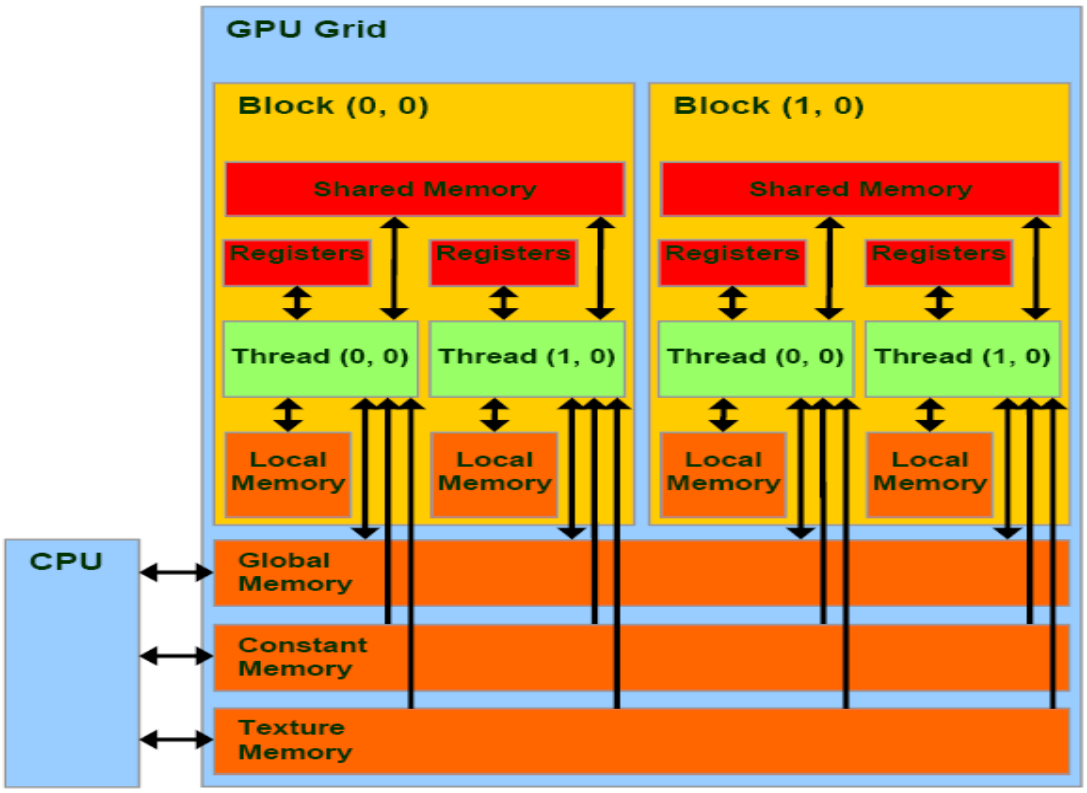


Figure 4.3 Memory model

Registers: Registers are the fastest accessible memory type on a GPU. When the kernel is launched each of the threads creates its own register memory. If the amount of register memory needed by all the threads in a block exceeds the available register memory per multiprocessor the kernel launch fails. The registers can be accessed in zero clock cycles per instruction except for delay in two particular cases when there are register bank conflicts or register read after write dependencies. The performance in both these cases is dependent on the number of threads per block. In case of bank conflicts we get maximum performance when the number of threads per block is a multiple of 64. The delay caused by register read after write dependencies can be ignored as long as there are at least 192 active threads per multiprocessor.

Shared memory: Shared memory has much lower access latency when compared to global memory. As long as there are no bank conflicts, accessing shared memory is as fast as accessing a register. If the amount of shared memory needed by a block exceeds the available shared memory per multiprocessor the kernel launch fails. We shall now briefly discuss

about the organization and the optimization of shared memory access. Shared memory is organized in the form of 16 banks all of which can be accessed simultaneously at the rate of 32 bits per two clock cycles. During allocation of shared memory for a block each of the consecutive 32 bit words are assigned to a consecutive bank. In a warp of 32 threads, access to shared memory is first done by threads in the first half warp and then in the other half warp. Thus there is no conflict for access of shared memory between threads in the first half warp and threads in the second half warp. However there maybe conflicts for access of shared memory between threads of the same half warp. Any memory read or write request made to different banks can be serviced simultaneously in a single transaction. However if more than one thread in the half warp requests access to the same memory bank we have bank conflicts and the accesses must be either broadcast or serialized. The broadcast mechanism works as follows. If multiple threads request addresses within the same 32 bit word it can be broadcast to all the threads simultaneously. Overall the memory access mechanism of shared memory can be described by the following algorithm. A memory read request made of several addresses is serviced in several steps over time, one step occurring every two clock cycles. At each step a subset of all the remaining addresses needed are serviced as follows:

➔ Select one of the words pointed to by the remaining addresses as the broadcast word.

➔ Include in the subset all the addresses within the broadcast word and also one address from each bank from the remaining addresses.

The broadcast word as well as the address picked for each bank in each cycle is not specified. Figures 4.4-4.5 show the shared memory access for 4 different cases.

Texture memory: Texture memory space is cached. If we have a cache miss, the texture fetch costs one memory read from the device memory. If we have a cache hit, the texture fetch costs one memory read from the texture cache. Also the texture cache is optimized for 2D spatial locality so that threads accessing texture address close to each other achieve the best performance. Texture memory will not further discussed here since it is not used in any of the CUDA implementations in this thesis.

Constant memory: The constant memory is cached. If we have a cache miss, the read will cost one memory read from the device memory. If we have a cache hit, the read will

cost one memory read from the constant cache. If all the threads in a half warp read the same address in the cache the cost of reading is the same as reading from a register. The cost scales linearly as different addresses are read by different threads. Constant memory will not further discussed here since it is not used in any of the CUDA implementations in this thesis.

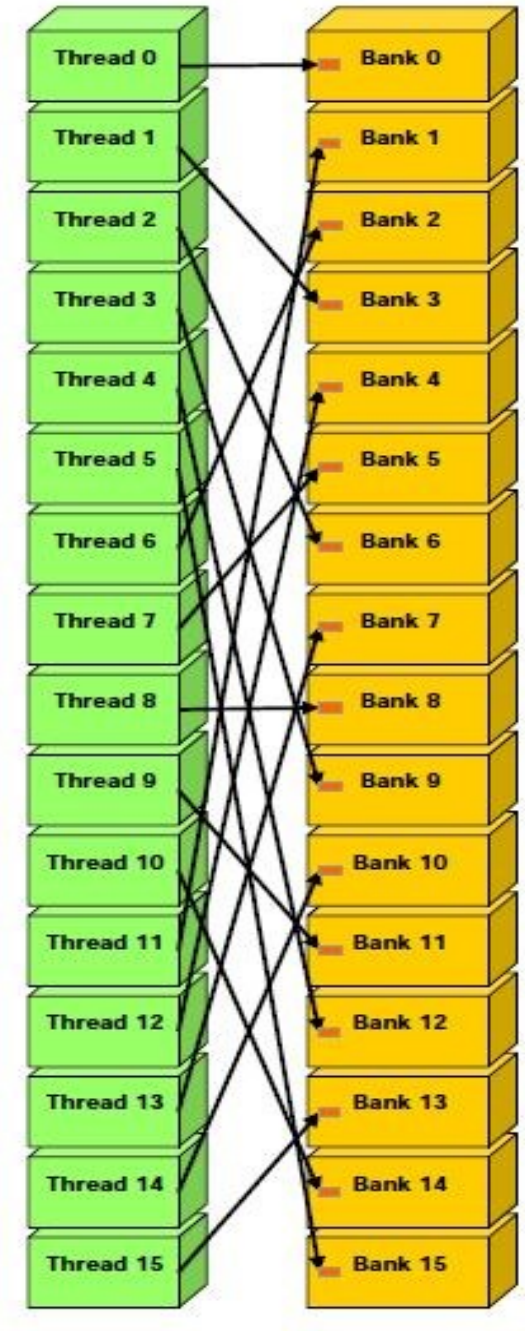


Figure 4.4 Shared memory access without bank conflicts

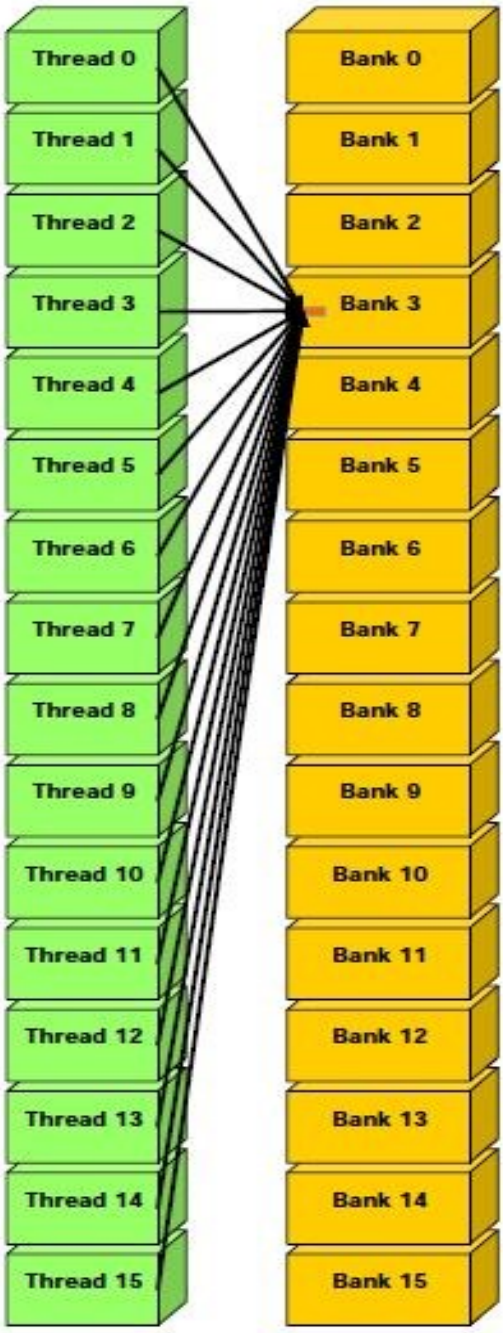


Figure 4.5 Shared memory without bank conflicts(broadcast)

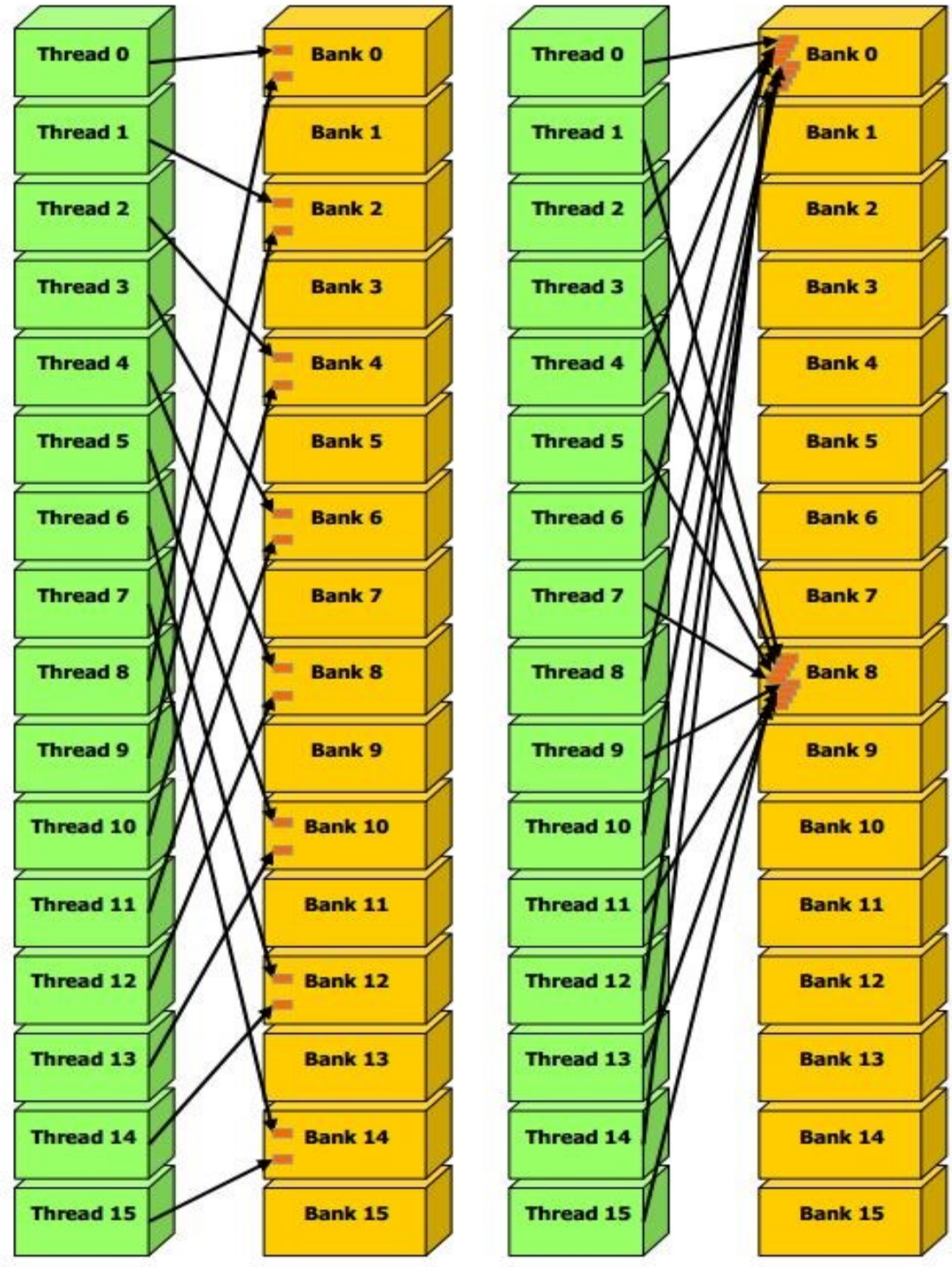


Figure 4.6 Shared memory with 2-way and 8-way bank conflict

Local memory: Local memory is not cached. Access to local memory is as expensive as access to global memory.

Global memory: Global memory is not cached. Access to global memory is very costly and since it is the place where most of the data is stored, optimizing data access speed from it is an extremely important component of CUDA programming. Again if the total amount of global memory needed exceeds the total available global memory, kernel launch will fail. The warps associated with a block of the kernel are divided into half warps and global memory access is done per half warp. Thus the access pattern of a half warp does not affect the memory access performance of a different half warp. We need to follow certain rules to coalesce the global memory access. These rules however are dependent on the compute capability of the GPU. The compute capability of the GPU is defined by a major revision number and a minor revision number. Devices having the same major revision number have the same core architecture. The minor revision number represents small incremental changes in the core architecture. We shall now discuss conditions for global memory coalescence in devices with different compute capabilities

Devices with compute capability 1.0 or 1.1: Global memory accesses by threads in a half warp are coalesced into one or two memory transactions if it satisfies all the following conditions

➔ Threads can access 4 byte words resulting in a 64 byte transaction, 8 byte words resulting in a 128 byte transaction, 16 byte words resulting in two 128 byte transactions.

➔ The 16 words must lie in the segment size equal to memory transaction size.

➔ The k-th thread in the half warp must access the k-th word.

If the above requirements are not met, separate transactions are issued for each of the threads and the memory throughput is significantly reduced.

Devices with compute capability 1.2 or 1.3: Global memory accesses by threads in a half warp are coalesced into a single memory transaction as soon as words accessed by all the threads lie in the same segment of size equal to

➔ 32 bytes if the threads access 1 byte words

➔ 64 bytes if the threads access 2 byte words

➔ 128 bytes if the threads access 4 byte or 8 byte words

The difference from the previous case discussed above is that the threads need not access the words in sequence. Whatever may be the access pattern unused words are still read. This leads to wastage of bandwidth. To minimize the amount of bandwidth wasted the hardware always tries to reduce the transaction to the smallest possible size. The entire process of memory access works as follows:

➔ Start with the lowest numbered active thread. Identify the segment in which it is present.

➔ Identify all the other active threads present in the same segment.

➔ Minimize the transaction size. If it is a 128 byte segment and only the lower or upper half is used reduce the transaction size to 64 bytes. If it is a 64 byte segment and only the lower or upper half is used reduce the transaction size to 32 bytes.

➔ Transfer the segment data and mark the threads which have been served inactive.

➔ Repeat the entire process until all the threads in the half warp have been serviced.

Devices with compute capability 2.0 or higher: We shall not discuss the details in this thesis since the devices have not been used in the implementations done.

The Figures 4.8 and 4.9 show some of the scenarios which arise during memory coalescing among devices of different compute capabilities.

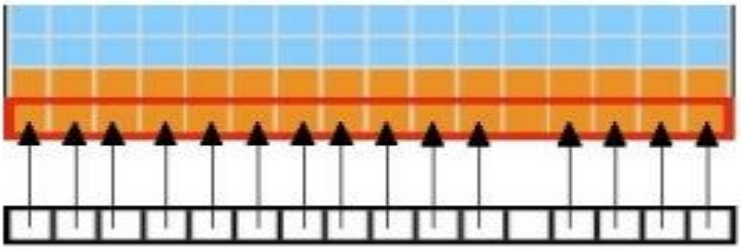


Figure 4.7 Coalesced access in compute capabilities 1.1-1.3

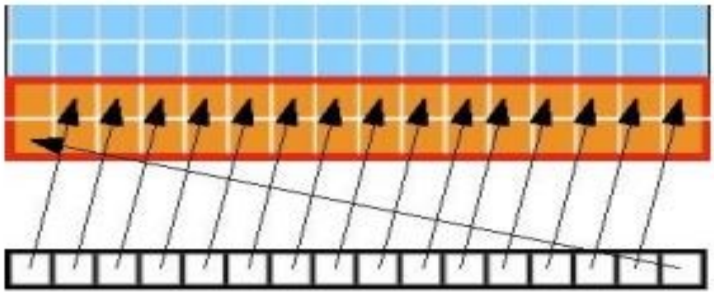


Figure 4.8 Seperate transaction in 1.0,1.2 but coalesced transaction in 1.2,1.3

As we can have seen above we have different kinds of memories with different bandwidths. It is always better to avoid accesses to memories having a high latency. A general programming pattern for memory access used in CUDA program development goes as follows :

➔ Copy the data from the CPU to the GPU global memory.

➔ Launch the kernel to do the required computation

➔ Load the data from the global memory into the shared memory.

➔ Using __syncthreads() ensure that all the threads in the block have loaded data into their respective shared memory locations before the threads start using each other's data.

➔ Process the data in the shared memory using registers and local memory for temporary values.

➔ Again using __syncthreads() ensure that shared memory has been updated with the results.

➔ Load the processed data from the shared memory into the device global memory.

➔ Copy back the processed data from the GPU global memory to the CPU.

4.4 Hardware model

We shall now discuss the hardware model of the GPU on which the CUDA program development is done. CUDA architecture is based on a scalable array of multithreaded streaming multiprocessors (SMs). As soon as the kernel is launched from the CPU the various blocks of the kernel are assigned to the available multiprocessors of the GPU as shown in the Figure 4.10. The number of blocks assigned to a multiprocessor is dependent on

the various hardware limitations like the amount of shared memory, amount of register memory and other architecture specific limitations(like the total number of threads, warps, blocks and so on). Based on these limitations one or more thread blocks can be assigned to a multiprocessor. The number of available multiprocessors also varies from GPU to GPU. For example, Tesla C1060 on which most of the program development in this thesis has been done has 30 multiprocessors. All the threads of a thread block execute simultaneously on the multiprocessor. As soon as one thread block finished execution new thread blocks are assigned to the now vacated multiprocessor. As we can see from the figure greater the number of multiprocessors, greater will be the number of thread blocks executed simultaneously and hence faster will be the execution of the kernel. We can determine the number of threads blocks assigned as well as study the hardware resources allocated using Parallel Nsight 3.0.

Let us now study the hardware model of a GPU as shown in Figure 4.11. A multiprocessor has the following components

- ➔ Eight scalar processor(SP) cores
- ➔ Two special function units for transcendentals
- ➔ A multithreaded instruction unit

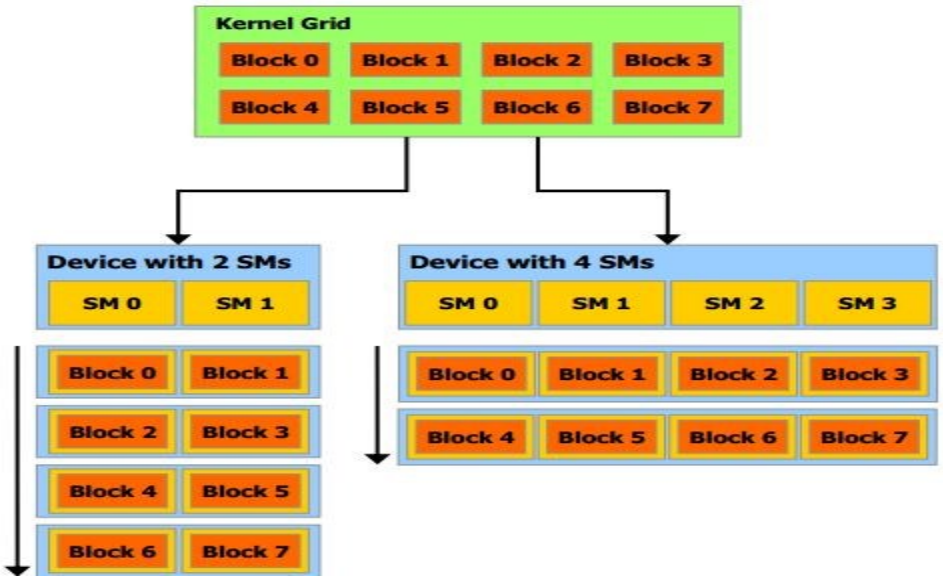


Figure 4.9 Assignment of blocks to multiprocessors

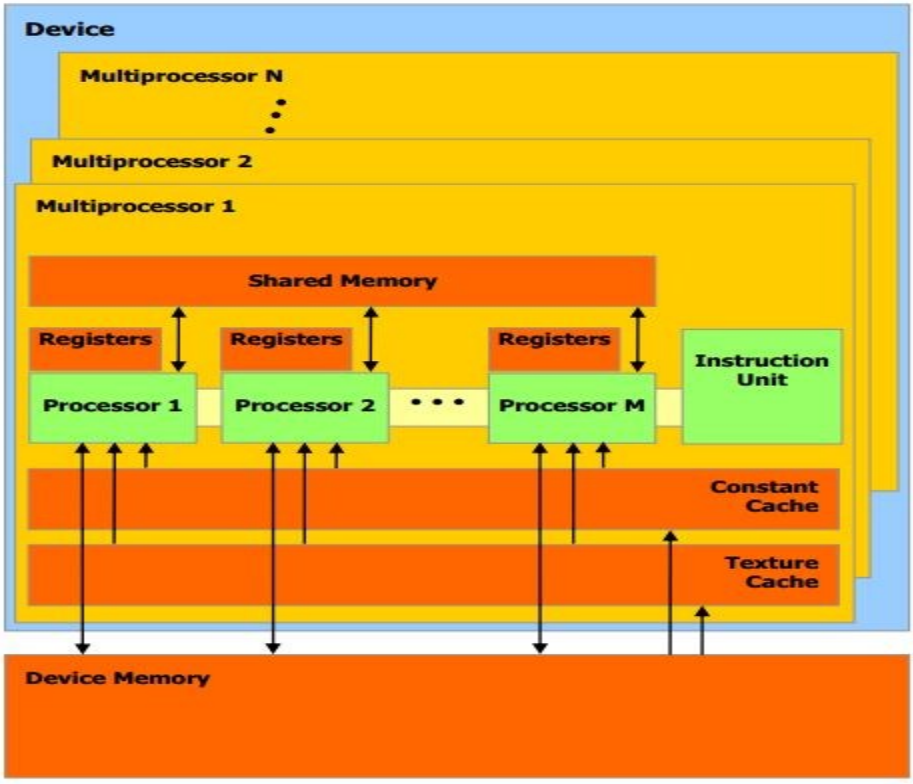


Figure 4.10 Hardware model of a GPU

➔ A set of local 32 bit registers per processor

➔ A parallel data cache or shared memory that is shared by all the processor cores

➔ A read only data cache that is shared by all the processor cores which speeds up reads from constant memory space which is a read only region in the device memory.

➔ A read only data cache that is shared by all the processor cores which speeds up reads from texture memory space which is a read only region in the device memory. The multiprocessor accesses the texture cache through the texture unit.

The multiprocessor is responsible for the creation, management and execution of concurrent threads with zero scheduling overhead. It implements the __syncthreads() instruction which provides barrier synchronization among the threads. The multiprocessor executes threads using the SIMT model (single instruction multiple threads). Each of the threads is mapped to a scalar processor core and each scalar thread executes independently with its own instruction address and register state. As soon as the kernel is launched the blocks are assigned to the all the multiprocessors on the GPU according to some conditions

which shall be discussed later in detail (involving various factors like shared memory, register configuration etc.). The multiprocessor then divides the block of threads into warps based on the thread ID. In every instruction cycle the multiprocessor selects a warp and issues the next instruction to the threads of the warp.

4.5 Best Practices

There are some general guidelines which should be followed during program development on a GPU and they will be summarized in this section. These practices are graded in order of priority such that higher priority practices give much better optimization when followed as compared to lower priority practices. Following the high priority practices can give as much as ten times better performance whereas the medium priority practices and the lower priority practices affect performance to a much lower extent. We shall see some of the best practices below.

4.5.1 High Priority practices

Maximum performance benefit: The amount of performance benefit that we get is entirely dependent on the amount of code which can be parallelized on the GPU. If the code which cannot be parallelized is more when compared to code which can be, parallelizing the program isn't going to achieve much benefit when it is run on the GPU. To get the maximum performance benefit we should have a program where a large portion of the code can be parallelized on the GPU (like in most of the image processing applications).

Data transfer between host and device: The peak bandwidth between the device memory and the GPU is much higher than the peak bandwidth between the host memory and the device memory. Thus it is important to minimize the data transfer between the CPU and the GPU. This means that some intermediate data structures should be created on the GPU without being mapped onto the host memory. This also means running some kernels doing intermediate calculations on the GPU even if they do not give any significant speedup when compared to the CPU.

Memory Coalescing: As we have seen above while discussing the memory model of the GPU memory coalescing of the global memory is extremely important if you want to achieve to achieve high bandwidth for global memory.

Memory instructions: Memory instructions include any reads or writes to global, local or shared memory. The instructions involving access to global or local memory have additional 400-600 cycles of memory latency. Therefore it is preferable to minimize the access of global or local memory and shared memory should be used instead.

Warp divergence: Different execution paths should be avoided inside the same warp. Any flow control instruction (if, switch, do, for, while) can significantly affect the instruction throughput by causing threads of the same warp to diverge; that is, to follow different execution paths. If this happens, the different execution paths must be serialized, increasing the total number of instructions executed for this warp. When all the different execution paths have completed, the threads converge back to the same execution path.

4.5.2 Medium Priority Practices

Shared memory and memory banks: As we have seen in section 4.3 shared memory is allocated in the form of banks and bank conflicts should be avoided to facilitate maximum bandwidth from shared memory.

Use shared memory often: Shared memory access is much faster when compared to global memory. Use shared memory to prevent redundant access of global memory. Transfer the data from the global memory into the shared memory, manipulate the data in the shared memory and then transfer the data back into global memory.

Number of threads per block: The number of threads per block should be a multiple of 32 to facilitate optimal computing efficiency and also memory coalescing.

Fast math library: Use fast math library whenever speed trumps precision.

4.5.3 Low priority practices

Kernel arguments: If the kernel has a long list of arguments place some arguments into constant memory to save shared memory.

Shift operations: Use shift operations to avoid expensive modulo and division calculations.

Automatic conversion: Avoid automatic conversion of doubles to floats.

4.6 General Specifications

The algorithm development has been done on a Tesla C1060 GPU Computing processor. It has 30 multiprocessors with each processor having 8 processors and has a compute capability of 1.3. The specifications for cards of various compute capabilities which determine all the limitations for kernel execution are as follows:

Specifications for Compute Capability 1.0:

➔ The maximum number of threads per block is 512;

➔ The maximum sizes of the x-, y-, and z-dimension of a thread block are 512, 512, and 64, respectively;

➔ The maximum size of each dimension of a grid of thread blocks is 65535;

➔ The warp size is 32 threads;

➔ The number of registers per multiprocessor is 8192;

➔ The amount of shared memory available per multiprocessor is 16 KB organized into 16 banks The total amount of constant memory is 64 KB;

➔ The total amount of local memory per thread is 16 KB;

➔ The cache working set for constant memory is 8 KB per multiprocessor; Appendix A. Technical Specifications

➔ The cache working set for texture memory varies between 6 and 8 KB per

➔ multiprocessor;

➔ The maximum number of active blocks per multiprocessor is 8;

➔ The maximum number of active warps per multiprocessor is 24;

➔ The maximum number of active threads per multiprocessor is 768;

➔ For a one-dimensional texture reference bound to a CUDA array, the maximum

➔ width is 213;

➔ For a one-dimensional texture reference bound to linear memory, the maximum

➔ width is 227;

- For a two-dimensional texture reference bound to linear memory or a CUDA
- array, the maximum width is 216 and the maximum height is 215;
- For a three-dimensional texture reference bound to a CUDA array, the
- maximum width is 211, the maximum height is 211, and the maximum depth is
- 211;
- The limit on kernel size is 2 million PTX instructions;

Specifications for Compute Capability 1.1:

- Support for atomic functions operating on 32-bit words in global memory

Specifications for Compute Capability 1.2:

- Support for atomic functions operating in shared memory and atomic functions operating on 64-bit words in global memory;
- Support for warp vote functions (see Section B.11);
- The number of registers per multiprocessor is 16384;
- The maximum number of active warps per multiprocessor is 32;
- The maximum number of active threads per multiprocessor is 1024.

Specifications for Compute Capability 1.3:

- Support for double-precision floating-point numbers;

CHAPTER 5. IMAGE PROCESSING ON THE GPU

5.1 Introduction

In this chapter we will discuss the implementation of some of the commonly used image enhancement algorithms like spatial filters on the GPU. These techniques form the basis of the preprocessing (ring artifact removal) and postprocessing (beam hardening) algorithms discussed in the next chapter. Many common filters have been developed on the GPU for noise reduction as part of the research including simple blurring filters like the box filter and Gaussian filter as well as edge preserving filters like the median filter and bilateral filter. These techniques will be implemented on both the CPU and the GPU and their performances will be compared. Also multiple versions of the filters have been developed to improve performance using performance enhancing techniques like shared memory access, maximizing multiprocessor occupancy and utilizing faster hardware based instructions. Apart from the filters we shall also discuss an efficient matrix transpose technique developed by NVIDIA which is an important component of many of the image processing algorithms developed as part of research.

5.2 Filtering

Filtering is an image enhancement technique done to process an image so that the resultant filtered image is more suitable to the end user than the original image. Filtering is generally done in either the spatial domain based on the direct manipulation of the pixels in the image or in the frequency domain based on manipulation of the Fourier transform of an image. All the filtering techniques used in the preprocessing and postprocessing algorithms involve spatial filtering while the reconstruction algorithm involves filtering in the frequency domain. Spatial filtering generally involves replacing all the pixel values in the image with some function (also called response for that pixel) of the neighboring pixels (also called the mask of the pixel). In linear spatial filters like the box filter and the Gaussian filter the response of a pixel is given by sum of products of the filter coefficients or weights and the neighboring pixels in the mask. In general linear filtering of an image of size M x N with a filter mask of size m x n is given by the equation (5.1). On the other hand in nonlinear spatial

filters like the median filter or the bilateral filter the pixel is not explicitly replaced by any sum of products as in equation (5.1) but is replaced by some other value based conditionally on the values of pixels in the neighborhood like the median of the mask in case of median filtering or photometric similarity in case of bilateral filtering.

$$g(x,y) = \sum_{s=-a}^{a}\sum_{t=-b}^{b} w(s,t) f(x+s, y+t)$$

$$Where$$

$$a = (m-1)/2$$

$$b = (n-1)/2$$

$$x = 0,1,2,3 \ldots M-1$$

$$y = 0,1,2,3 \ldots N-1$$

(5.1)

One important concern in all the spatial filters implemented in this thesis is regarding the borders of an image. The pixels at the image borders do not have a mask of the requisite size as compared to other interior pixels. This issue can be handled in various ways. One of the ways is to pad the image with zeroes or the border image pixel values and use these padded values as part of the mask. Instead of padding the image we can use a partial mask for pixels when a full mask cannot be obtained. A third way is to wrap around the image and use values at the beginning of the opposite border. A final way (used for all the filters in this thesis) is to avoid filtering the border pixels which do not have a mask of the requisite size. Thus in the resultant response image we will have stripes of pixels left untouched by the filter. We shall now discuss the implementation of all the basic spatial filters on the GPU implemented as part of research.

5.3 Box Filter and implementation on the GPU

An averaging or mean filter is one of the simplest smoothing spatial filters used in many image enhancement algorithms. The filter coefficients in the mask may all be equal(used in this thesis) or may vary according to the geometric distance from the pixel

under consideration(the weight generally falls off as distance increases). In general averaging filtering of an image of size M x N with a filter mask of size m x n is given by the equation (5.2). Box filter is a special case of a spatial averaging filter when all the coefficients in the mask are equal to 1 and is used mainly for noise reduction and blurring. It is an example of a low pass filter in that higher frequencies are attenuated while lower frequencies are left unchanged. On applying a box filter to an image any pixel having a sharp change of intensity value as compared to the neighboring pixels (as is the case with noisy pixels) is replaced by a pixel with an averaged normal intensity value thereby reducing noise. Thus we get a smoothed image retaining the important features but leaving out the noise in the image. An unwanted side effect of this phenomenon is the blurring of the edges as even the edges of an image have sharp transitions in intensity values. This can be prevented by using edge preserving filters discussed in the following sections. However blurring is not a totally unwanted phenomenon and has many real time applications. One of its uses is to bridge small gaps in lines and curves so that incomplete or discontinuous edges can be filled out. A different use is to get a gross representation of the objects of interest by making smaller objects blend with the background and making larger objects more noticeable. The size of objects to be blended with the background depends on the size of the filter.

$$g(x,y) = \left(\sum_{s=-a}^{a}\sum_{t=-b}^{b} w(s,t) f(x+s, y+t)\right) / \left(\sum_{s=-a}^{a}\sum_{t=-b}^{b} w(s,t)\right)$$

$$Where$$
$$a = (m-1)/2$$
$$b = (n-1)/2$$
$$x = 0,1,2,3 \ldots M-1$$
$$y = 0,1,2,3 \ldots N-1$$

(5.2)

The computation of the mean of an individual pixel in a row of the sinogram data is independent of the computation for other pixels. Hence the algorithm is highly parallelizable and is very suitable for porting onto the GPU. Also the application of the algorithm on a

single sinogram file is independent of the application on other sinogram files. Thus all the sinogram files are handled in a sequential manner. A sinogram file is first copied from the CPU onto the GPU. Then the kernel responsible for the algorithm is launched on the GPU. The filtered sinogram data is then copied back from the GPU onto the CPU. This process is then repeated for all the sinogram files to be filtered.

We shall now discuss the implementation of the kernel in detail. The inputs of the algorithm are as follows:

- ➔ Location of the first sinogram in the dataset
- ➔ The first file in the dataset to be filtered
- ➔ Number of files to be filtered
- ➔ Radius of the box filter
- ➔ Number of iterations
- ➔ Location of the filtered sinogram files if they are to be written

The output of the algorithm is as follows:

- ➔ Filtered sinogram dataset at the destination specified by the end user
- ➔ avf <<<>>>:

Functionality: Responsible for applying the box filter algorithm on the required sinogram dataset

Execution configuration parameters:

$$\left\{\begin{array}{l}(blockDimx = 512),\\ (blockDimy = 1),\\ (gridDimx = step),\\ (gridDimy = ((width - 2 * avgFilRad)/blockDimx)(+1\ if\ not\ an\ exact\ multiple),\\ (\text{sharedmemorysize} = (\text{blockDim.x} * \text{sizeof(float)}))\end{array}\right\}$$

Description: We launch the kernel as a 2D grid of blocks with blocks having the same $(blockIdx.\text{x})$ value responsible for elements in one of the rows of the sinogram. Each block is in turn composed of 512 threads with each thread responsible for applying the box filter algorithm on one of the pixels. The elements to be loaded by a block (512 + (2 * avgFilRad) elements) are loaded from the global memory into the shared memory in two stages. In the first stage 512 of these elements are loaded by all the threads in the block. Then

the additional (2 * avgFilRad) elements needed are loaded in multiple iterations with only the lower numbered threads participating in the last iteration so as to facilitate memory coalescing. The filtered values are calculated using the required pixels in the mask with equal weights assigned to all of the pixels. Finally the filtered values are loaded back into the global memory. Each of these stages is synchronized using __syncthreads() to ensure instruction synchronization and memory visibility.

Performance tuning: Separate threads could have been used in the load stage for loading the extra (2 * avgFilRad) elements and left idle during the calculation of the mean values. But as the radius of the box filter increases this technique leads to inefficient thread and shared memory usage. Global memory is always accessed in a coalesced fashion across all compute capabilities. Shared memory is used to reduce access to the high latency global memory as each of the pixel values are used multiple times and also because the pixel values are shared by all the threads in a block. Warp divergence is mostly absent. Shared memory is also accessed without any bank conflicts present. The execution configuration parameters have been chosen so as maximize the warp occupancy on Tesla C1060 as measured using Nvidia Parallel Nsight 3.0. Also having a large enough number of threads (512) reduces the global memory loads needed.

Future improvements: Use of texture memory to take advantage of spatial locality of the algorithm. Port the algorithm onto the next generation NVDIA architectures (Kepler and Fermi).

The effects of applying the algorithm on one of the sinograms in the dataset as well as on the reconstructed slice are shown in Figure 5.1. The process is then repeated for filters of varying sizes and also varying number of iterations as shown in Figure 5.2. In general, it can be seen that noise in all the cases has been reduced to a great extent but also blurring of the image has occurred across the edges.

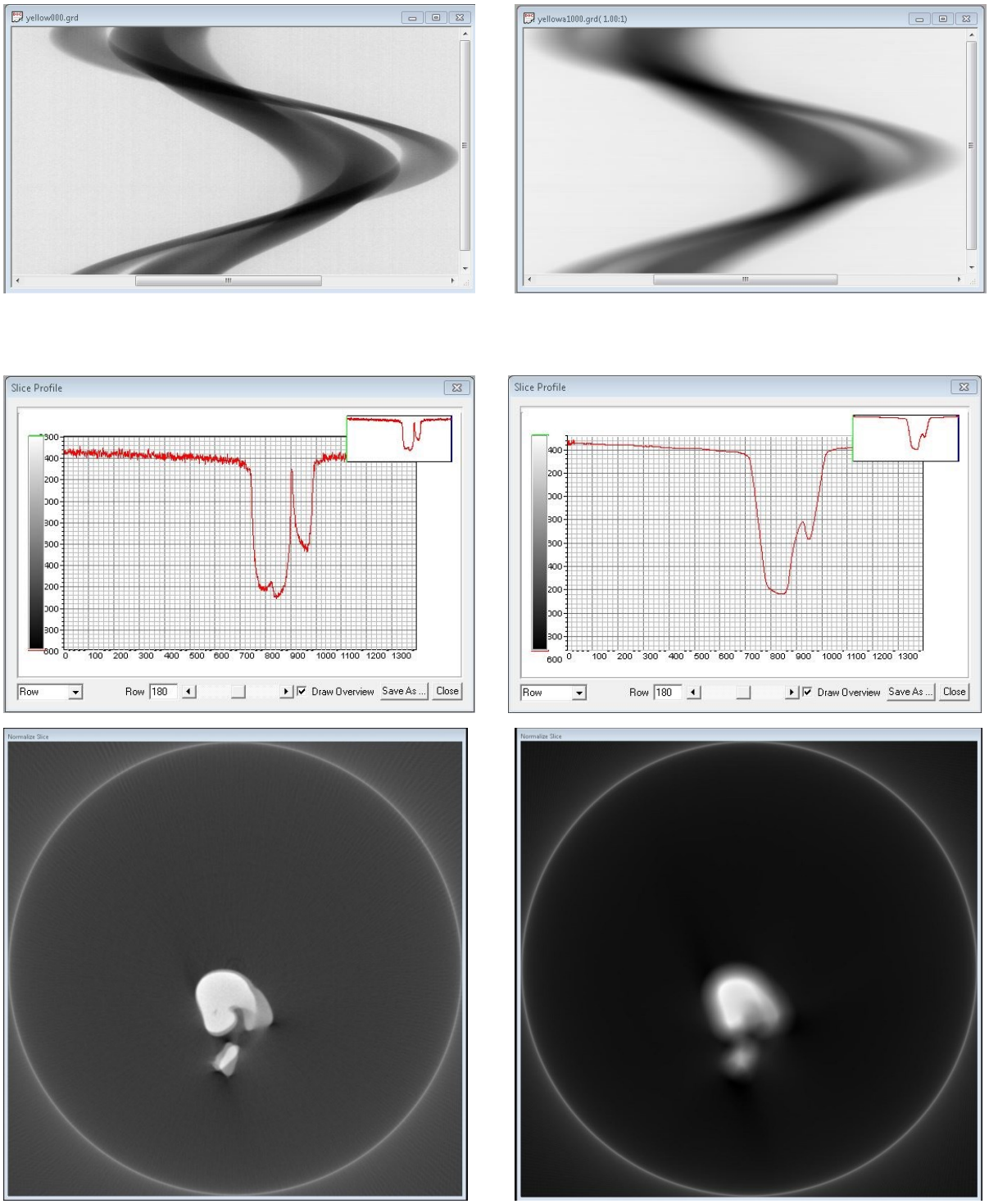


Figure 5.1 Sinogram data, slice profile and reconstructed slice before and after box filtering

Also greater the size of the mask or greater the number of iterations used greater is the blurring.

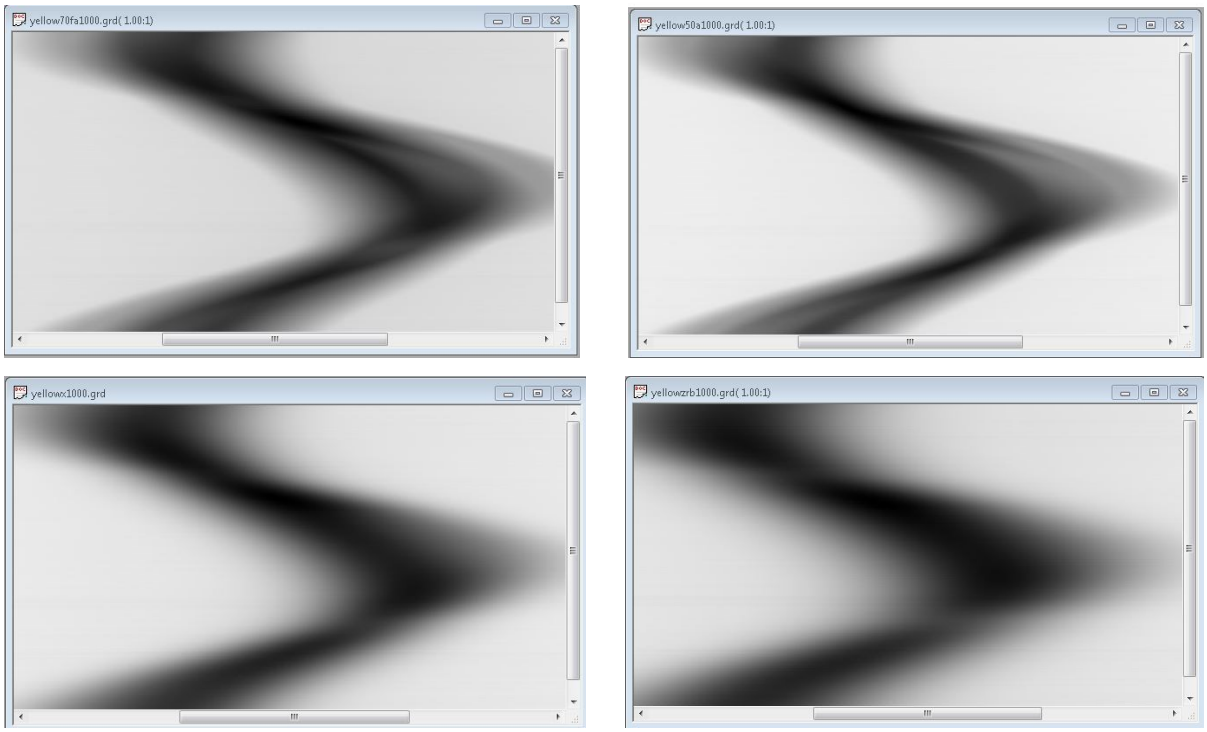


Figure 5.2 (Top) The sinogram data for filter sizes 50 and 70. (Bottom) The sinogram data for 10 and 20 iterations.

The time taken by various instances of the filtering algorithm varying the radius of the filter and the number of iterations is shown in Table 5.1 and the corresponding graph is depicted in Figure 5.3.

Radius of filter	Time(milliseconds)		
	1 iteration	10 iterations	20 iterations
30	4	47	93
50	6	63	123
70	7	78	140

Table 5.5.1 Time taken by the box filter

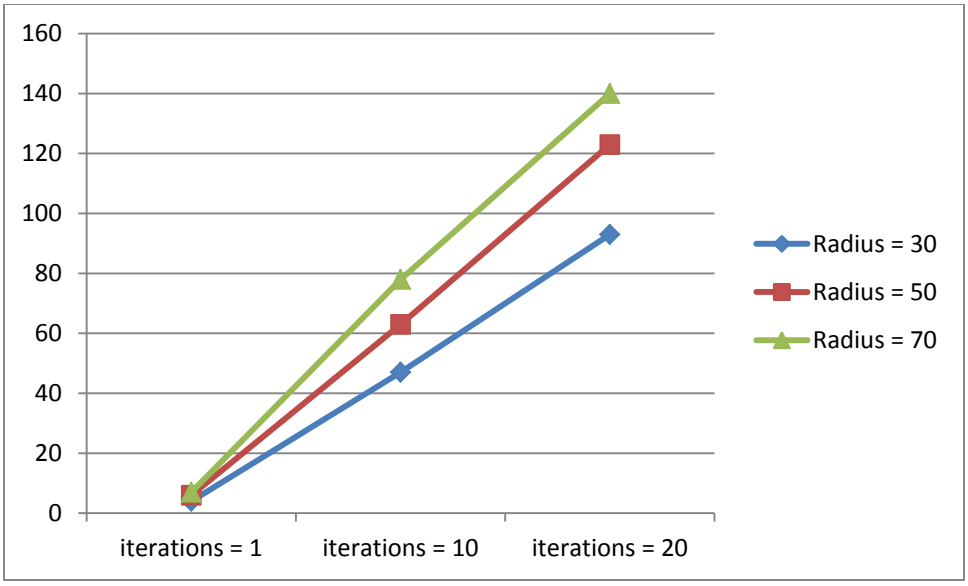


Figure 5.3 Time taken by the box filter (milliseconds)

A CPU version of the median filter has also been implemented to measure the performance gain on the GPU and verify the results of the filtering. The results are shown in Table 5.2. This performance can further be improved by following all the improvements possible for the kernel as mentioned above.

FILTER SIZE	CPU TIME (milliseconds)	GPU TIME (milliseconds)	SPEEDUP
30	132	4	33
50	220	6	37
70	280	7	40

Table 5.2 CPU versus GPU(Box Filter)

The occupancy graph (showing the various bottlenecks for kernel launch) and the timeline of execution for one of the kernels (showing the various stages of execution of kernel and their timings) as measured by Nvidia Parallel Nsight 3.0 are shown in Figure 5.4

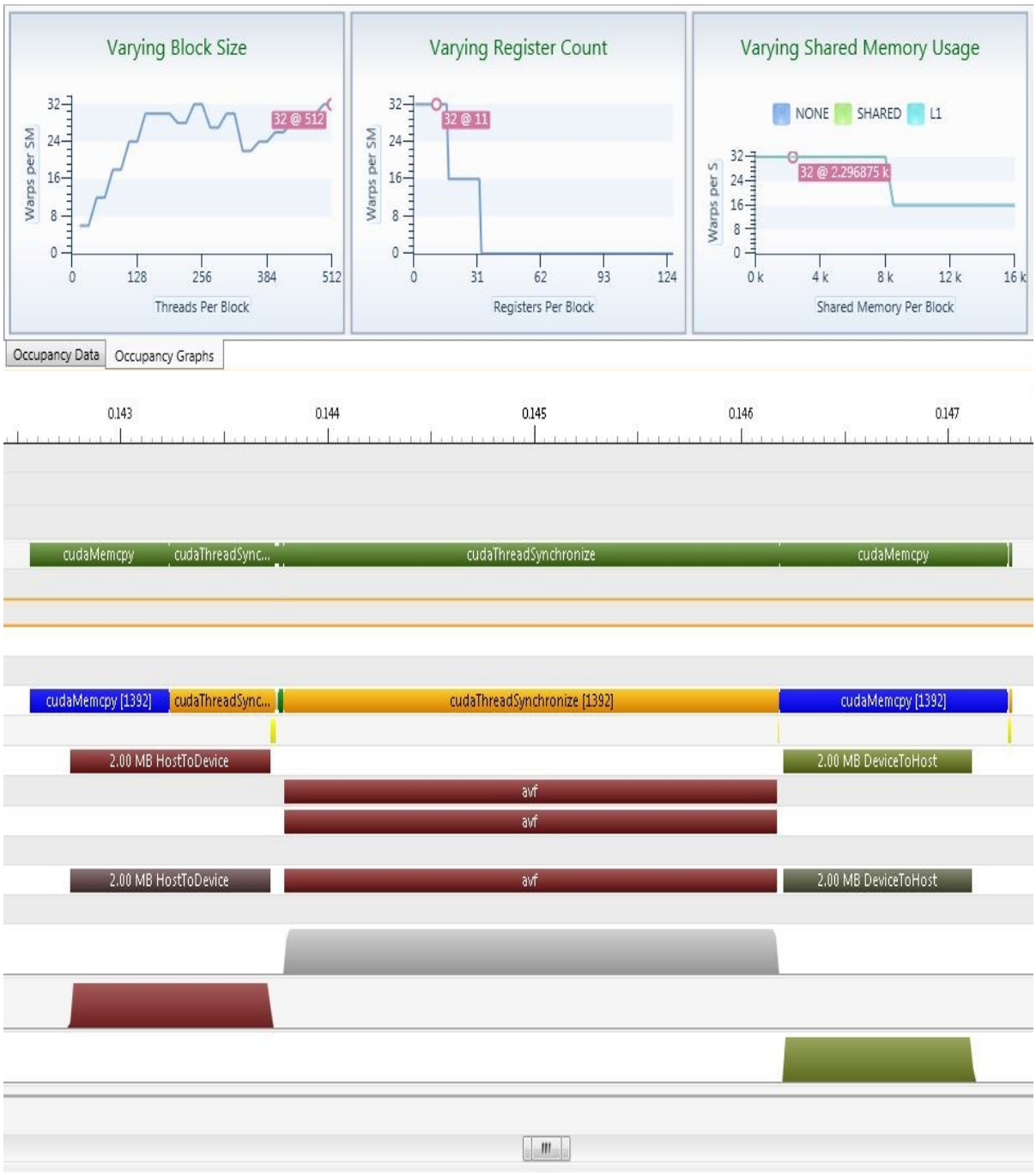


Figure 5.4 Occupancy graph and timeline of execution

5.4 Gaussian Filter and implementation on the GPU

The box filter we have discussed in the previous section is not a good low pass filter for many applications since it has an uneven response to higher frequencies. Though it is easy to implement, the edges are blurred to a very great extent (since equal weights are assigned to all the pixels in a mask) leading to loss of vital information. A better spatial low pass filter used in many image processing applications is the Gaussian filter. It also attenuates higher frequencies in an image but does so in a far more even manner when compared to the box filter. Like the box filter it is also used in noise reduction and image blurring. The Gaussian filter outputs a weighted average of each pixel's neighborhood based on the Gaussian distribution, with the average weighted more towards the value of the central pixels. This is in contrast to the box filter's uniformly weighted average. Because of this, a Gaussian filter provides gentler smoothing and preserves edges better than a similarly sized mean filter. Thus it is very frequently used in conjugation with edge detection algorithms like the Sobel edge detection or the Canny edge detection. In particular Gaussian filter is used to remove Gaussian noise but is not effective at removing a different type of noise called salt and pepper noise (removal of this types of noise is generally done using a median filter discussed in the next section). The Gaussian distribution in one and two dimensions is given by equations (5.3) and equation (5.4) respectively. We can infer from these equations that greater the value of σ slower will be the fall off of weights in the Gaussian distribution (shown in Figure 5.5 for 1D Gaussian filtering). In theory, the Gaussian distribution is non-zero everywhere, which would require an infinitely large convolution kernel, but in practice it is effectively zero more than about three standard deviations from the mean, and so we can truncate the kernel at this point. One more special property of Gaussian filtering is that it is convolution separable. It means that the application of a 2D filter on the image can be divided into the application of two consecutive 1D filters on the image. This property has been used to do Gaussian filtering on the image during the post processing stage in two dimensions as part of research. A row filter is applied on the image followed by a matrix transpose followed by a column filter. Separable filters can be adapted onto the GPU in a more efficient fashion as they allow efficient shared memory usage (due to smaller kernel size) and also efficient bandwidth usage (due to reduced pixel data access).

$$g(x) = \exp^{-((x * x) /(2 * \sigma * \sigma))} \quad (5.3)$$

Where g(x) = Gaussian weight at a distance x
σ = standard deviation

$$g(x, y) = \exp^{-((x*x+y*y)/(2*\sigma*\sigma))} \quad (5.4)$$

Where g(x,y) = Gaussian weight at a distance (x,y)
σ = standard deviation

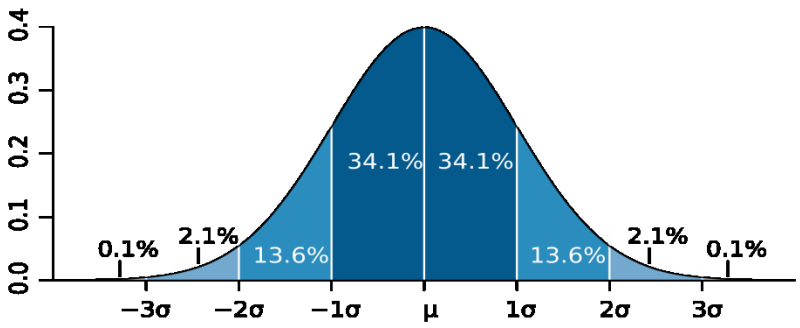


Figure 5.5 1D Gaussian Distribution

The computation of the filtered value of an individual pixel in a row of the sinogram data is independent of the computation for other pixels. Hence the algorithm is highly parallelizable and is very suitable for porting onto the GPU. Also the application of the algorithm on a single sinogram file is independent of the application on other sinogram files. Thus all the sinogram files are handled in a sequential manner. A sinogram file is first copied from the CPU onto the GPU. Then the kernel responsible for the algorithm is launched on the GPU. The filtered sinogram data is then copied back from the GPU onto the CPU. This process is then repeated for all the sinogram files to be filtered.We shall now discuss the implementation of the kernel in detail. The inputs of the algorithm are as follows:

- ➔ Location of the first sinogram in the dataset
- ➔ The first file in the dataset to be filtered
- ➔ Number of files to be filtered
- ➔ Radius of the Gaussian filter
- ➔ Sigma domain value

➔ Number of iterations

The output of the algorithm is as follows:

➔ Filtered sinogram dataset at the destination specified by the end user

➔ gaf <<<>>>:

Functionality: Responsible for applying the Gaussian filter algorithm on the required sinogram dataset

Execution configuration parameters:

$$\begin{Bmatrix} (blockDim.x = 512), \\ (blockDim.y = 1), \\ (gridDim.x = step), \\ (gridDim.y = ((width - 2 * gauFilRad)/blockDim.x)(+1\ if\ not\ an\ exact\ multiple), \\ (\text{shared memory size } = (\text{blockDim.x} * \text{sizeof(float)})) \end{Bmatrix}$$

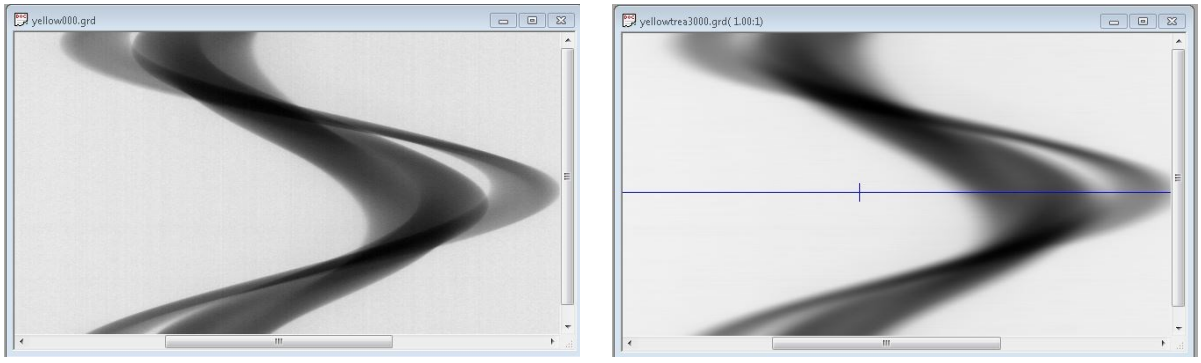
Description: We launch the kernel as a 2D grid of blocks with blocks having the same $(blockIdx.\text{x})$ values responsible for elements in one of the rows of the sinogram. Each block is in turn composed of 512 threads with each thread responsible for applying the Gaussian filter algorithm on one of the pixels. The elements to be loaded by a block (512 + (2 * gauFilRad) elements) are loaded from the global memory into the shared memory in two stages. In the first stage 512 of these elements are loaded by all the threads in the kernel. Then the additional (2 * gauFilRad) elements needed are loaded in multiple iterations with only the lower numbered threads participating in the last iteration so as to facilitate memory coalescing. Then filtered values are calculated using the required pixels in the mask with weights assigned based on Gaussian distribution. Finally the filtered values are then loaded back into the global memory. Each of these stages is synchronized using __syncthreads() to ensure instruction synchronization and memory visibility.

Performance tuning: Separate threads could have been used in the load stage for loading the extra (2 * gauFilRad) elements and left idle during the calculation of the filtered values. But as the radius of the Gaussian filter increases this technique leads to inefficient thread and shared memory usage. Global memory is always accessed in a coalesced fashion across all compute capabilities. Shared memory is used to reduce access to the high latency

global memory as each of the pixel values are used multiple times and also because the pixel values are shared by all the threads in a block. Warp divergence is mostly absent. Shared memory is also accessed without any bank conflicts present. The execution configuration parameters have been chosen so as maximize the warp occupancy on Tesla C1060 as measured using Nvidia Parallel Nsight 3.0. Also having a large enough number of threads (512) reduces the global memory loads needed.

Future improvements: Use of faster but less accurate math instructions. Precomputation of the spatial map used for Gaussian distribution in the global memory. Use shared memory to store the precomputed map instead of the global memory. Use constant memory to store the precomputed map instead of the shared memory. Use texture memory to take advantage of spatial locality of the algorithm. Port the algorithm onto the next generation NVDIA architectures (Kepler and Fermi).

The effects of applying the algorithm on one of the sinograms in the dataset as well as on the reconstructed slice are shown in Figure 5.5. The process is then repeated for filters of varying radii, varying sigma domain values and also varying number of iterations and the results are shown in Figure 5.6. It can be seen that like the box filter, Gaussian filter also does noise reduction but it blurs edges to a lesser extent. Increasing any one of the three parameters increases both noise reduction and blurring.



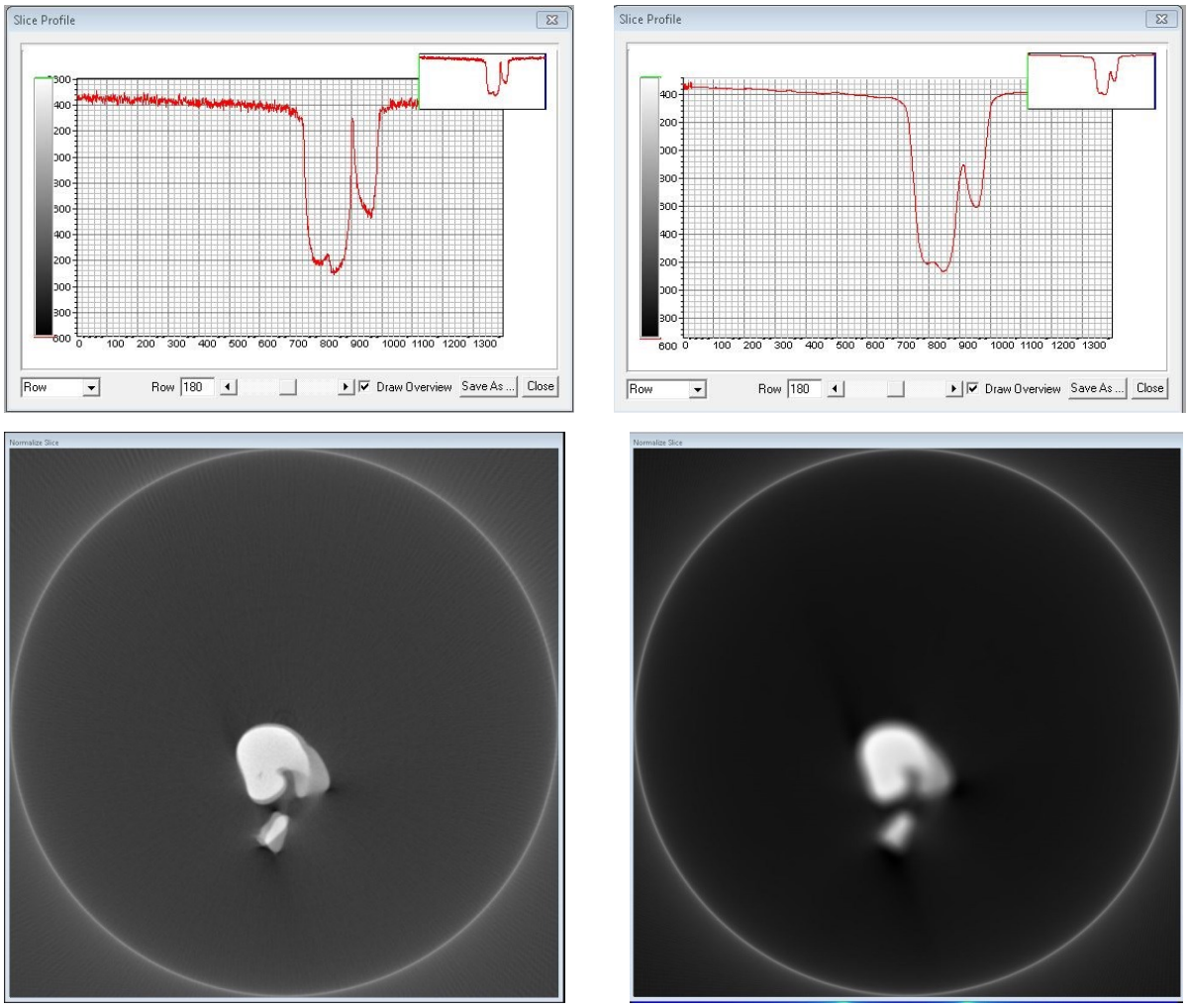
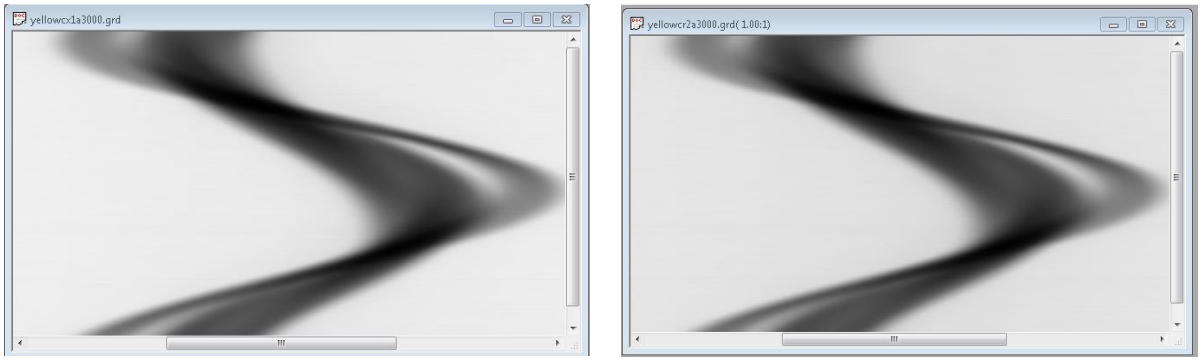


Figure 5.6 Sinogram data, slice profile and reconstructed slice before and after box filtering



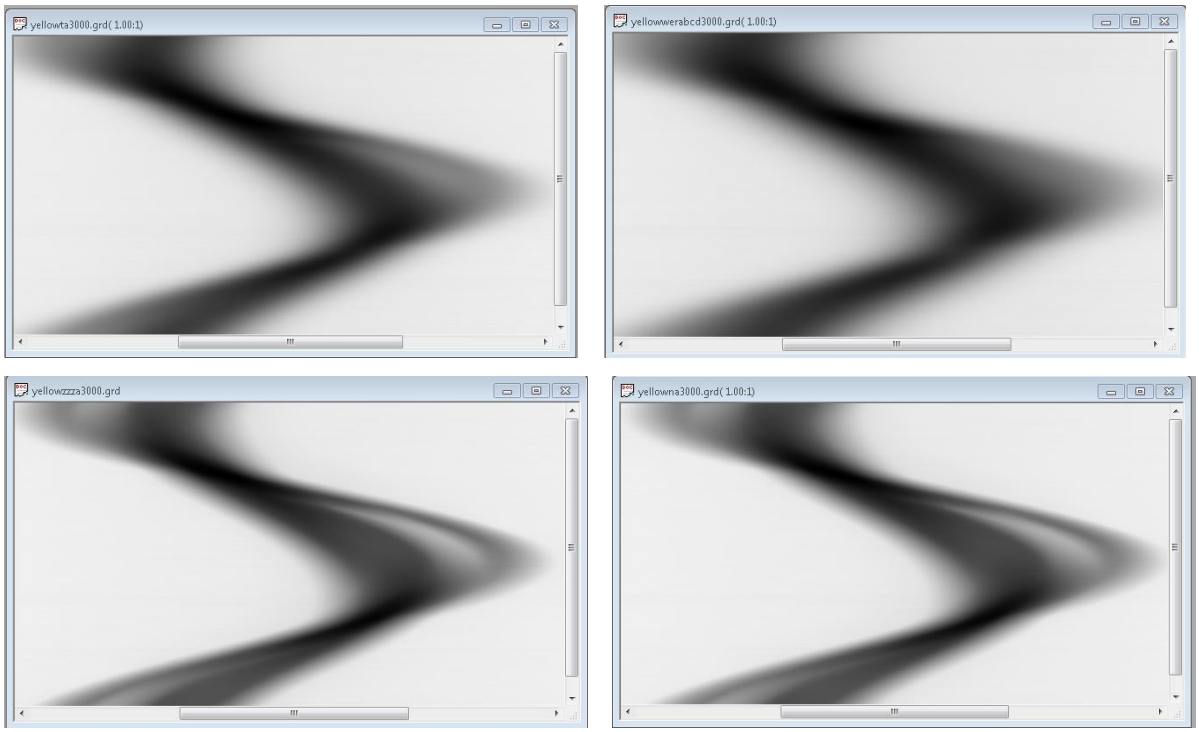


Figure 5.7 (Top) The sinogram data for filter sizes 50 and 70. (Middle) The sinogram data for 10 and 20 iterations. (Bottom) The sinogram data for sigma = 20 and 30.

The time taken by various instances of the filtering algorithm varying the radius of the filter and the number of iterations is shown in Table 5.2 and the corresponding graph is depicted in Figure 5.7.

Radius of filter	Time(milliseconds)		
	1 iteration	10 iterations	20 iterations
30	8.8	93	203
50	12	140	288
70	20	188	375

Table 5.3 Time taken by the Gaussian filter for various parameters.

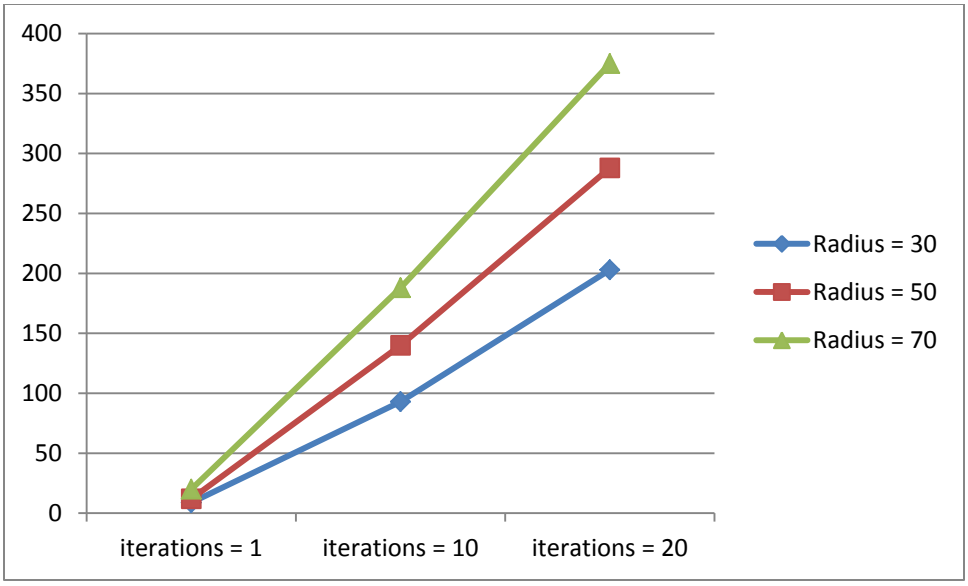


Figure 5.8 Time taken by the Guassian filter for various parameters(ms)

A CPU version of the median filter has also been implemented to measure the performance gain on the GPU and verify the results of the filtering. The results are shown in Table 5.4. This performance can further be improved by following all the improvements possible for the kernel as mentioned above.

FILTER SIZE	CPU TIME (milliseconds)	GPU TIME (milliseconds)	SPEEDUP
30	1233	8.8	140.11
50	1977	12	164.75
70	2699	20	134.95

Table 5.4 CPU versus GPU(Gaussian filter)

The occupancy graph (showing the various bottlenecks for kernel launch) and the timeline of execution for one of the kernels (showing the various stages of execution of kernel and their timings) as measured by Nvidia Parallel Nsight 3.0 are shown in Figure 5.9

Figure 5.9 Occupancy graph and timeline of execution

5.5 Median Filter and implementation on the GPU

Median filtering is an example of a more general class of spatial nonlinear filters called order statistic filters. Order statistic filters differ from normal spatial filters in the sense

that the value of the filtered pixel is not determined by weighing the pixels in the mask but by ordering the pixels in the mask according to some property. This property can be the median of the intensity values in the mask (in the case of median filter), the minimum of values in the mask (in case of a min filter) or the maximum of values in the mask (in case of a max filter). Also unlike the box filter it is an example of nonlinear filtering since the filtered value is not based on some weights assigned to the mask (and hence does not follow the properties required by a linear function like additivity and homogeneity). Median filters are also used for noise reduction and blurring. However unlike the previous filters they are better at preserving edges in an image while also doing good noise reduction. Median filters are in particular very suitable for removing a kind of noise called as salt and pepper noise just as a Gaussian filter is suitable for Gaussian noise. Salt and pepper noise in an image can be identified by a low number of pixels having either a maximum intensity value or a minimum intensity value. Development of both the box and the Gaussian filter on the GPU are examples of embarrassingly parallel problems as little or no effort is required to separate the problem into a number of parallel tasks. This is often the case where there exists no dependency (or communication) between those parallel tasks. However the development of a median filter on the GPU is far more complicated because of the sorting complexity involved (sorting of intensity values in the mask is needed to determine the median value in the mask). We shall now see why sorting of data does not follow the data parallel computation requirement of a GPU and hence is not a good choice for parallelization on the GPU. Let us start with loading the elements to be filtered into the shared memory. Sorting for one of the elements involves rearranging the neighboring pixels to get the median value. However for a data parallel sorting computation to be performed on a different pixel the order of the neighboring pixels should be left unchanged. We can avoid this problem by allocating some additional shared memory to each of the threads and do the sort of data there. While we can do this for small filters we quickly run out of shared memory for larger filters. A brute force sorting algorithm using only global memory has also been implemented on the GPU not giving very good results. Instead of sorting the elements in the mask we propose a new method in this thesis based on estimating the position of the elements involved. As we shall see finding an order statistic does not always involve sorting of the input data. Median

filtering can also be done on the post processed slices but it hasn't been implemented yet. A separable median filter can be used to approximate the 2D median filter. The separable filter again has potential for faster implementation as compared to the normal 2D filter.

The computation of an order statistic for an individual pixel in a row of the sinogram data is independent of the computation for other pixels. Hence the algorithm is parallelizable and is suitable for porting onto the GPU. Also the application of the algorithm on a single sinogram file is independent of the application on other sinogram files. Thus all the sinogram files are handled in a sequential manner. A sinogram file is first copied from the CPU onto the GPU. Then the kernel responsible for the algorithm is launched on the GPU. The filtered sinogram data is then copied back from the GPU onto the CPU. This process is then repeated for all the sinogram files to be filtered. We shall now discuss the implementation of the kernel in detail. The inputs of the algorithm are as follows:

- ➔ Location of the first sinogram in the dataset
- ➔ The first file in the dataset to be filtered
- ➔ Number of files to be filtered
- ➔ Radius of the median filter
- ➔ Number of iterations
- ➔ Location of the filtered sinogram files if they are to be written

The output of the algorithm is as follows:

- ➔ Filtered sinogram dataset at the destination specified by the end user
- ➔ mef <<<>>>:

Functionality: Responsible for applying the median filter algorithm on the required sinogram dataset

Execution configuration parameters:

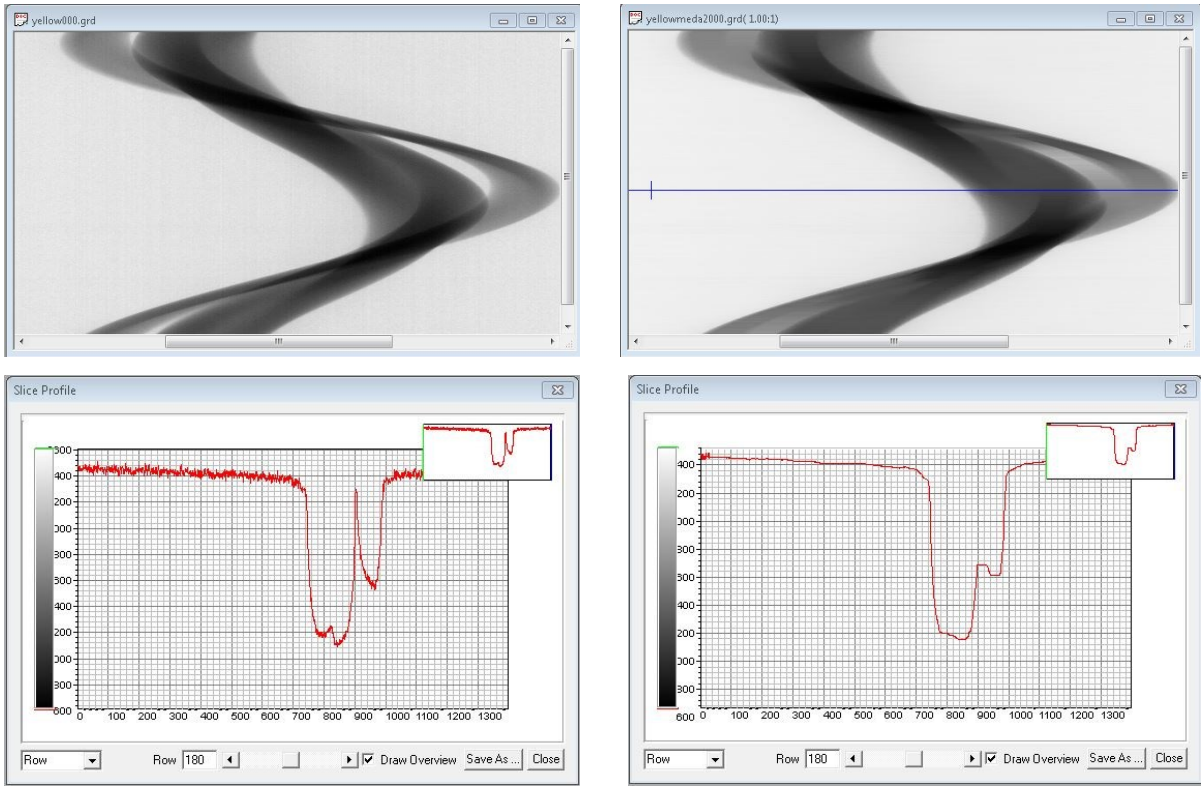
$$\begin{Bmatrix} (blockDim.x = 512), \\ (blockDim.y = 1), \\ (gridDim.x = step), \\ (gridDim.y = ((width - 2 * medFilRad)/blockDim.x)(+1 \; if \; not \; an \; exact \; multiple), \\ (\text{shared memory size } = (\text{blockDim.x} * \text{sizeof(float)})) \end{Bmatrix}$$

Description: All the elements in one of the rows of the sinogram are handled by blocks having the same (*blockIdx*.x) value. All the blocks having the same (*blockIdx*.x) value but differing (*blockIdx.y*) values calculate the medians of a group of 512 pixels with each thread handling a unique pixel. The elements to be loaded by a block (512 + (2 * medFilRad) elements) are loaded from the global memory into the shared memory in two stages. In the first stage 512 of these elements are loaded by all the threads in the kernel. Then the additional (2 * medFilRad) elements needed are loaded in multiple iterations with only the lower numbered threads participating in the last iteration so as to facilitate memory coalescing. The thread responsible for identifying the median of a pixel in its mask then does the following operations. For each of the elements in the mask it computes a count of the number of elements in the mask it is greater than or equal to. If the count is greater than or equal to the radius of the median filter the pixel under consideration is considered to be a candidate for the median of the mask. This process is repeated for all the pixels in the mask. Also in each of the steps the minimum value of all the possible median candidates is maintained and updated whenever necessary. This minimum value is considered to be the median of the mask of pixels and is used to replace the pixel value associated with the thread. Finally the filtered values are then loaded back into the global memory. Each of these stages is synchronized using __syncthreads() to ensure instruction synchronization and memory visibility.

Performance tuning: Separate threads could have been used in the load stage for loading the extra (2 * medFilRad) elements and left idle during the calculation of the filtered values. But as the radius of the median filter increases this technique leads to inefficient thread and shared memory usage. Global memory is always accessed in a coalesced fashion across all compute capabilities. Shared memory is used to reduce access to the high latency global memory as each of the pixel values are used multiple times and also because the pixel values are shared by all the threads in a block. Warp divergence is mostly absent. Shared memory is also accessed without any bank conflicts present. The execution configuration parameters have been chosen so as maximize the warp occupancy on Tesla C1060 as measured using Nvidia Parallel Nsight 3.0. Also having a large enough number of threads (512) reduces the global memory loads needed.

Future improvements: Research more efficient GPU methods to compute the median of a mask. Use texture memory to take advantage of spatial locality of the algorithm. Port the algorithm onto the next generation NVDIA architectures (Kepler and Fermi).

The effects of applying the algorithm on one of the sinograms in the dataset as well as on the reconstructed slice are shown below. The process is then repeated for filters of varying radii and also varying number of iterations. It can be seen that in both the cases noise has been reduced to a great extent but the edges of the image have also been preserved unlike in the case of box or Gaussian filtering. Also greater the size of the mask or greater the number of iterations used greater is the noise reduction.



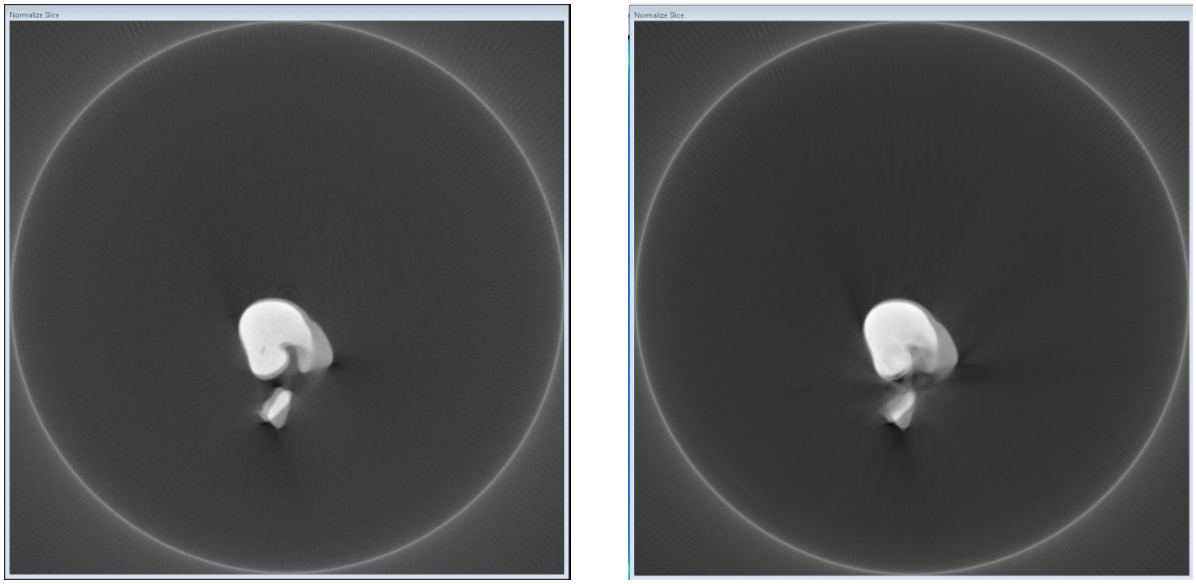


Figure 5.10 The sinogram data, slice profile and the reconstructed slice before and after median filtering

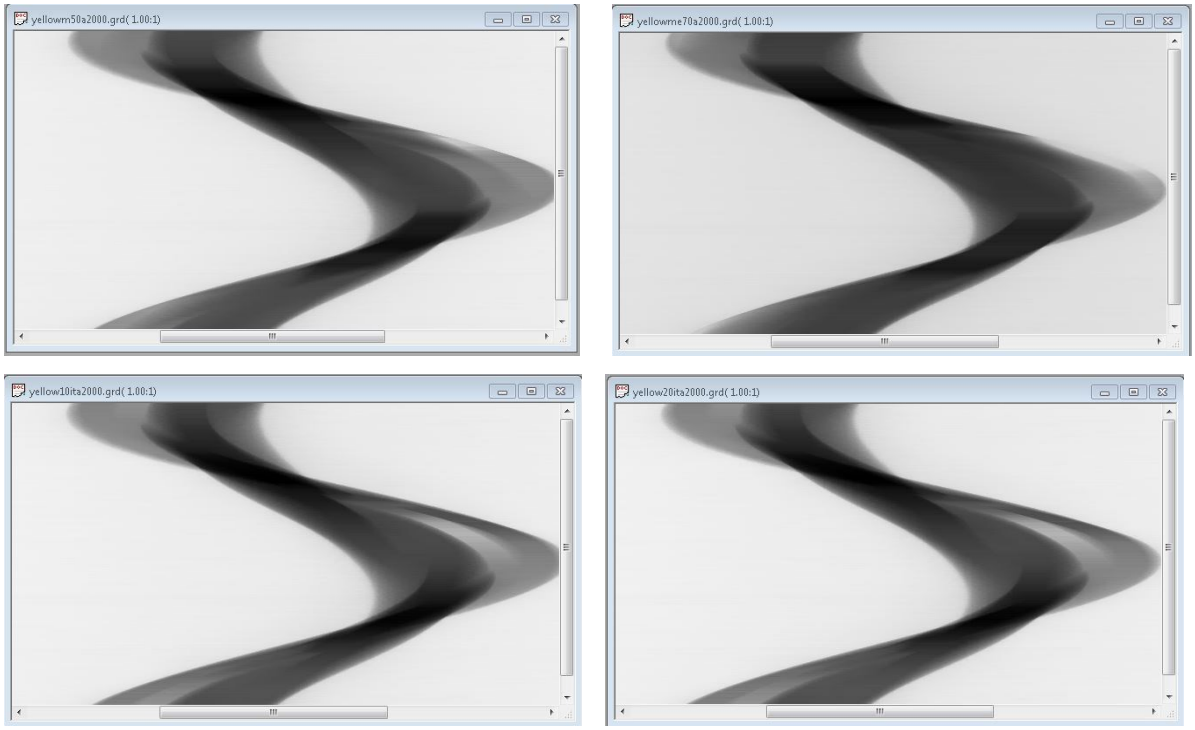


Figure 5.11 The sinogram data for filter sizes 50 and 70. (Bottom) The sinogram data for 10 and 20 iterations.

The time taken by various instances of the filtering algorithm varying the radius of the filter and the number of iterations is shown in Table 5.5 and the corresponding graph is depicted in Figure 5.10.

Radius of filter	Time(milliseconds)		
	1 iteration	10 iterations	20 iterations
30	203	2060	4118
50	562	5390	10764
70	1045	10140	20233

Table 5.5 Time taken by the median filter for various parameters.

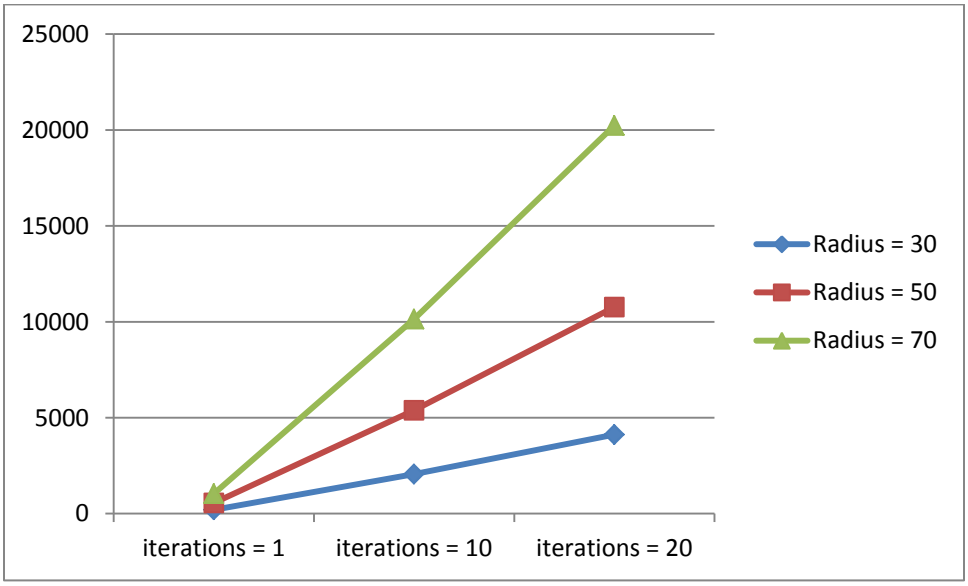


Figure 5.12 Time taken by the median filter for various parameters.

A CPU version of the median filter has also been implemented to measure the performance gain on the GPU and verify the results of the filtering. The results are shown in Table 5.6. This performance can further be improved by following all the improvements possible for the kernel as mentioned above.

FILTER SIZE	CPU TIME (milliseconds)	GPU TIME (milliseconds)	SPEEDUP
30	3853	203	18.9
50	6911	562	12.29
70	11016	1045	10.54

Table 5.6 CPU versus GPU (median filter)

The occupancy graph (showing the various bottlenecks for kernel launch) and the timeline of execution for one of the kernels (showing the various stages of execution of kernel and their timings) as measured by Nvidia Parallel Nsight 3.0 are shown in Figure 5.9

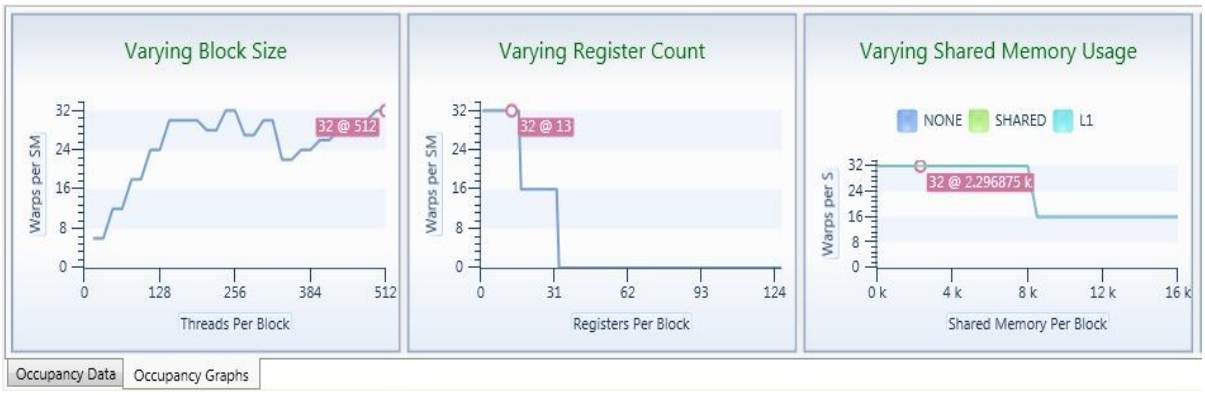


Figure 5.13 Occupancy graph and Timeline of execution

5.6 Bilateral Filter and implementation on the GPU

All the filters thus implemented on the GPU have some defects in them making them unsuitable candidates for preprocessing algorithms. The box filter though easy to implement and effective at reducing the noise blurs the edges to a great extent. Same is the effect of the Gaussian filter which blurs the edges to a lower extent but blurs them nonetheless. The median filter implemented on the GPU is a very good choice for an edge preserving smoothing filter but is cumbersome to implement and not computationally efficient (even on the GPU). In this section we shall discuss the implementation of a new nonlinear, non-iterative, local filter called bilateral filter on the GPU which eliminates both the defects of the median filter. The efficiency of this filter allows it to be an integral part of many of the preprocessing and postprocessing algorithms like ring artifact removal and beam hardening.

In general bilateral filter is used in many computational photography applications such as tone mapping, style transfer, relighting, denoising etc. Bilateral filtering also involves the computation of the weighted average of nearby pixels and is an improvement over an already existing filter namely the Gaussian filter. As we have seen before Gaussian convolution computes a weighted average of pixels in the neighborhood with weights falling off with distance from the pixel being filtered. The intuition behind this is that the pixel values vary slowly across the image slice and hence it is better to average together pixels nearby to each other. This assumption fails at pixels across edges which are geometrically very close but may have extremely divergent values and hence leads to the blurring of edges. The most popular techniques developed in image enhancement to overcome this defect are anisotropic diffusion and bilateral filtering. In anisotropic diffusion the local variation is calculated using partial differential equations and the weights are assigned based on these local variations. Solving partial differential equations is iterative in nature and cannot be ported properly onto the GPU due to issues of efficiency. Let us now study the bilateral filtering in detail. Two pixels in an image can be close to one another (occupy nearby spatial locations in domain) or they can be similar to one another (have nearby photometric values in range). In domain filtering (like Gaussian filtering) the weights fall off by distance whereas in range filtering the weights fall off with dissimilarity between pixels. Combined domain and range filtering is called bilateral filtering. The spatial and photometric weights assigned to neighboring pixels are given the following equations

$$b_d(x) = \exp^{-((x * x)/(2 * \sigma_d * \sigma_d))} \quad (5.5)$$

Where $b_d(x)$ = Spatial Filter weight at a distance x

σ_d = standard deviation in domain

$$b_r(x) = \exp^{-((p * p)/(2 * \sigma_r * \sigma_r))} \quad (5.6)$$

Where $b_r(x)$ = Photometric Filter weight at a distance x

$I(P)$ = *Photometric value of the pixel under consideration P*

$I(x)$ = *Photometric value of the pixel at a distance x*

p = I(P) - I(x)

σ_r = standard deviation in range

The above equation represents an often used special case of bilateral filtering when both the domain and range filtering are based on Gaussian distribution. If a collection of pixels are in a region without edges the filter acts as a normal Gaussian filter and averages away the small, weakly correlated differences between pixel values caused by noise. If they are in a region with edges the bilateral filter acts in a more sophisticated fashion demonstrated by the Figure 5.14. Figure 5.14(a) shows a sharp edge between a dark and bright region with a 100 gray level step perturbed with a Gaussian noise of 10 gray levels. Let us say we do bilateral filtering on a pixel on the brighter boundary with a mask radius of 23 pixels taking 5 pixels into consideration in domain (σ_d) and 100 gray levels into consideration in range (σ_r). The combined spatial photometric weights applied to neighboring pixels in the mask by the bilateral filter are shown in figure 5.14(b). The pixels on the brighter boundary contribute a lot in the filtering process and the pixels on the darker boundary do not contribute at all. This is because the spatial weights on the darker boundary are suppressed by their photometric weights. The overall effect of bilateral filtering is seen in figure 5.14(c). As we can see the image has been smoothed while still preserving the edges. Only bright pixels are taken into consideration when a bright pixel is being filtered just as only dark pixels are taken into consideration when a dark pixel is being filtered. The values of (σ_d) and (σ_r) can be input from the end user and can be used to manipulate the image in whatever form necessary. (σ_d) controls the number of pixels in the spatial neighborhood which affect the filtering process. Greater the value of (σ_d), greater will be the smoothing since pixels from a wider neighborhood affect the filtering process. (σ_r) determines the gray levels of neighboring pixels to be considered during the filtering process. Greater the value of (σ_r), greater will be the blurring of edges since more pixels from across the edge will affect the filtering process.

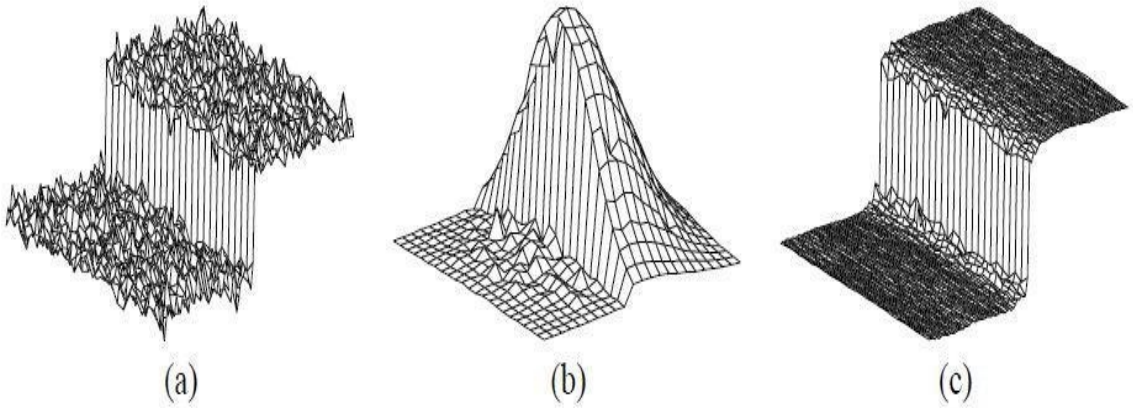


Figure 5.14 Bilateral filtering across an edge

The computation of a filtered value for an individual pixel in a row of the sinogram data is independent of the computation for other pixels. Hence the algorithm is parallelizable and is very suitable for porting onto the GPU. Also the application of the algorithm on a single sinogram file is independent of the application on other sinogram files. Thus all the sinogram files are handled in a sequential manner. A sinogram file is first copied from the CPU onto the GPU. Then the kernel responsible for the algorithm is launched on the GPU. The filtered sinogram data is then copied back from the GPU onto the CPU. This process is then repeated for all the sinogram files to be filtered. We shall now discuss the implementation of the kernel in detail. The inputs of the algorithm are as follows:

- Location of the first sinogram in the dataset
- The first file in the dataset to be filtered
- Number of files to be filtered
- Radius of the bilateral filter
- Sigma domain value(σ_d)
- Sigma range value (σ_r)
- Number of iterations
- Location of the filtered sinogram files if they are to be written

The output of the algorithm is as follows:

- Filtered sinogram dataset at the destination specified by the end user

➔ bif <<<>>>:

Functionality: Responsible for applying the bilateral filter algorithm on the required sinogram dataset

Execution configuration parameters:

$$\left\{\begin{array}{l}\left(blockDim.x = 512\right),\\ \left(blockDim.y = 1\right),\\ \left(gridDim.x = step\right),\\ \left(gridDim.y = \left(\left(width - 2*bilFilRad\right)/blockDim.x\right)\left(+1\ if\ not\ an\ exact\ multiple\right),\\ \left(\text{shared memory size } = \left(\text{blockDim.x * sizeof(float)}\right)\right)\end{array}\right\}$$

Description: All the elements in one of the rows of the sinogram are handled by blocks having the same $(blockIdx.\text{x})$ value. All the blocks having the same $(blockIdx.\text{x})$ value but differing $(blockIdx.\text{y})$ values calculate the filtered value of a group of 512 pixels with each thread handling a unique pixel. The elements to be loaded by a block (512 + (2 * bilFilRad) elements) are loaded from the global memory into the shared memory in two stages. In the first stage 512 of these elements are loaded by all the threads in the kernel. Then the additional (2 * bilFilRad) elements needed are loaded in multiple iterations with only the lower numbered threads participating in the last iteration so as to facilitate memory coalescing. Then each of the threads computes the filtered value of one of the pixels assigning the appropriate spatial and photometric values to pixels in the mask. Finally the filtered values are then loaded back into the global memory. Each of these stages is synchronized using __syncthreads() to ensure instruction synchronization and memory visibility.

Performance tuning: Many performance improvements have been done as part of research since bilateral filter is a very important component of many algorithms. Faster math library has been used instead of a slower more accurate math library for exponential functions (__expf() is used instead of expf()). The loss is precision is extremely low and permissible for the filtering process. Also the Gaussian domain weights can be precomputed and loaded into the shared memory instead of calculating them multiple times. Separate threads could have been used in the load stage for loading the extra (2 * bilFilRad) elements

and left idle during the calculation of the filtered values. But as the radius of the bilateral filter increases this technique leads to inefficient thread and shared memory usage. Global memory is always accessed in a coalesced fashion across all compute capabilities. Shared memory is used to reduce access to the high latency global memory as each of the pixel values are used multiple times and also because the pixel values are shared by all the threads in a block. Warp divergence is mostly absent. Shared memory is also accessed without any bank conflicts present. The execution configuration parameters have been chosen so as maximize the warp occupancy on Tesla C1060 as measured using Nvidia Parallel Nsight 3.0. Also having a large enough number of threads (512) reduces the global memory loads needed. Both the faster math library as well as the pre-computation of the Gaussian spatial map has been found to give a performance gain and have been integrated into the algorithm.

Future improvements: Take advantage of redundant calculations involved in the computation of weights (both spatial and photometric) to improve performance. Use constant memory to store the precomputed map instead of the shared memory. Use texture memory to take advantage of spatial locality of the algorithm. Port the algorithm onto the next generation NVDIA architectures (Kepler and Fermi).

The effects of applying the algorithm on one of the sinograms in the dataset as well as on the reconstructed slice are shown below. The process is then repeated for filters of varying radii and also varying number of iterations. It can be seen that in both the cases noise has been reduced to a great extent but the edges of the image have also been preserved unlike in the case of box or Gaussian filtering. Also greater the size of the mask or greater the number of iterations used greater is the blurring. It is also repeated for changing (σ_d) and (σ_r) values. Greater the (σ_d) value more is the smoothing because of a greater number of neighboring pixels involved in the filtering process. Greater the (σ_r) value more is the blurring of edges since greater number of pixels across the edge get involved in the blurring process.

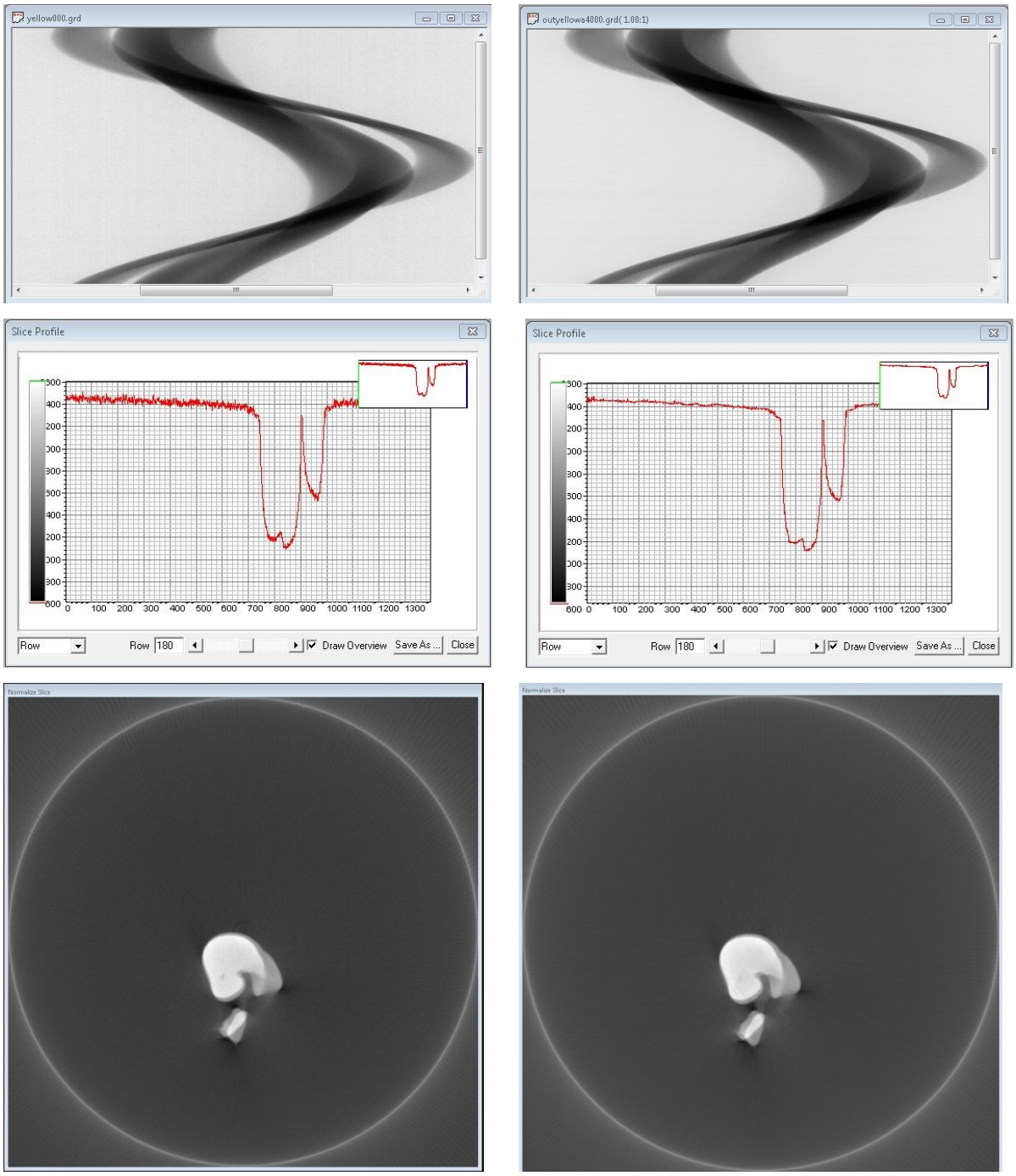


Figure 5.15 The sinogram data, slice profile and the reconstructed slice before and after bilateral filtering

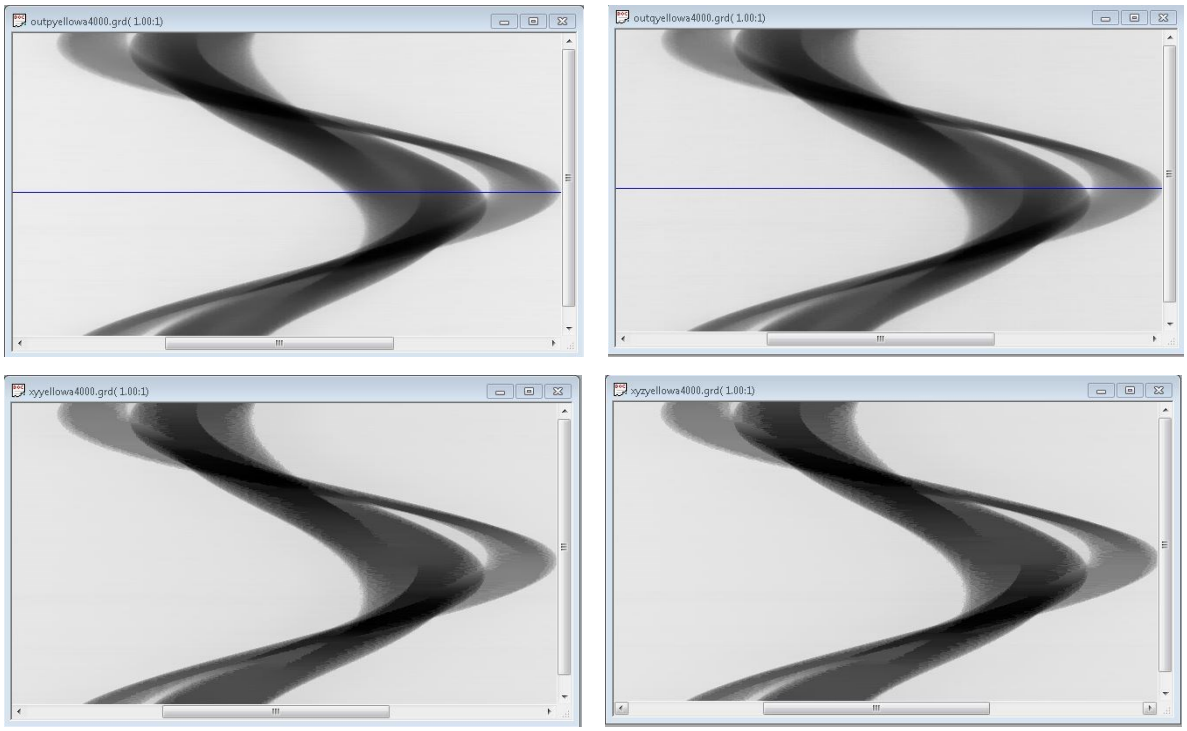


Figure 5.16 (Top) The sinogram data for filter sizes 50 and 70. (Bottom) The sinogram data for 10 and 20 iterations.

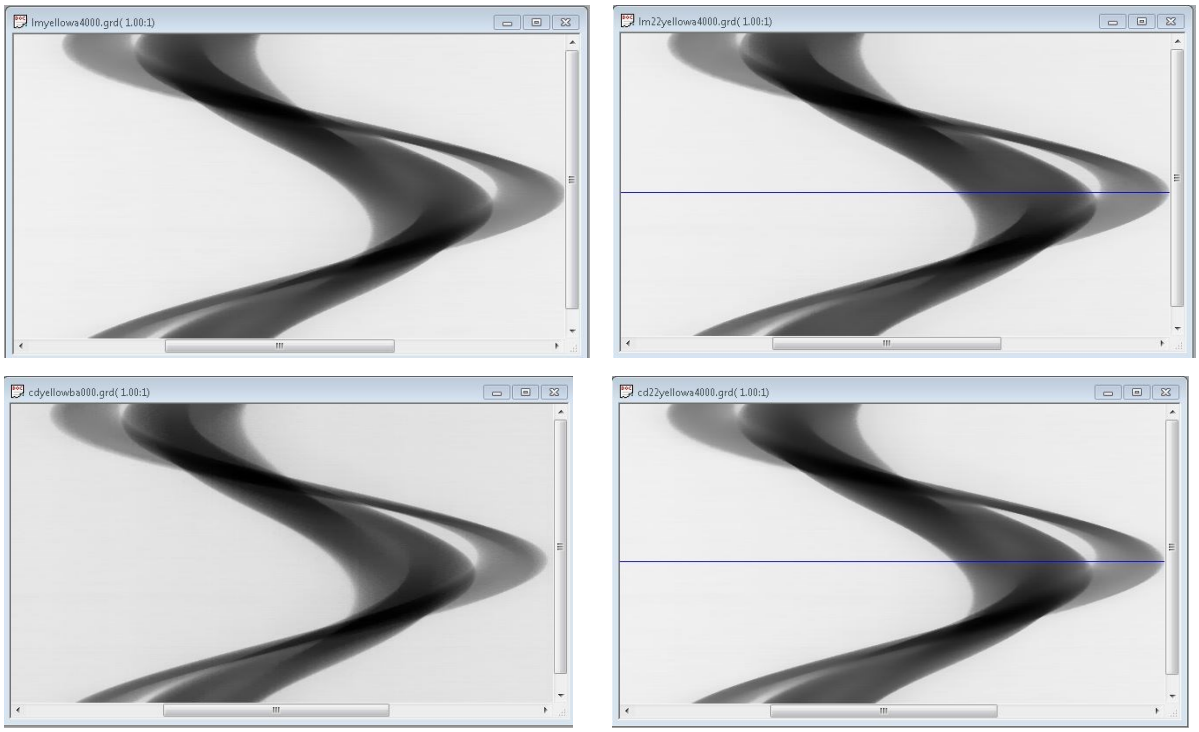


Figure 5.17 Top) The sinogram data for sigma domain values 10 and 100. (Bottom) The sinogram data for sigma range values 30 and 100

The time taken by various instances of the filtering algorithm varying the radius of the filter and the number of iterations is shown in Table 5.5 and the corresponding graph is depicted in Figure 5.10.

Radius of filter	Time(milliseconds)		
	1 iteration	10 iterations	20 iterations
30	16	171	328
50	18	250	499
70	31	328	655

Table 5.7 Time taken by the bilateral filter for various parameters.

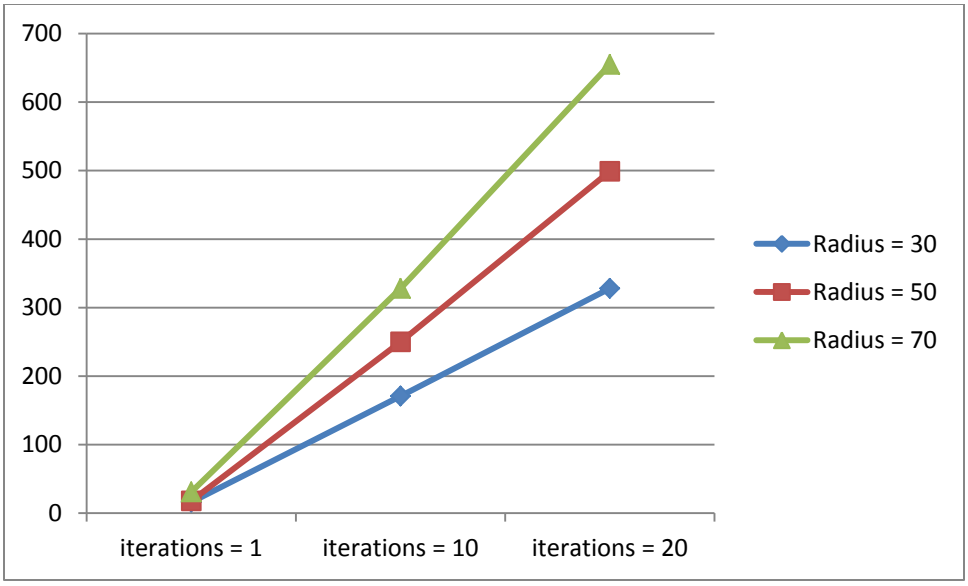


Figure 5.18 Time taken by the bilateral filter for various parameters.

A CPU version of the median filter has also been implemented to measure the performance gain on the GPU and verify the results of the filtering. The results are shown in Table 5.6. This performance can further be improved by following all the improvements possible for the kernel as mentioned above.

FILTER SIZE	CPU TIME (milliseconds)	GPU TIME (milliseconds)	SPEEDUP
30	2464	16	154
50	4071	18	226
70	5538	31	178

Table 5.8 CPU versus GPU (bilateral filter)

The occupancy graph (showing the various bottlenecks for kernel launch) and the timeline of execution for one of the kernels (showing the various stages of execution of kernel and their timings) as measured by Nvidia Parallel Nsight 3.0 are shown in Figure 5.9

Figure 5.19 Occupancy graph and Timeline of execution

5.7 Matrix Transpose using the CUDA enabled GPU

Matrix transpose is an extremely important component of many image processing algorithms. Post processing of slices in two dimensions (like the application of separable filters like Gaussian or median filter) involves matrix transpose of data. Beam hardening algorithm discussed in the next chapter uses matrix transpose for bilateral filtering in two dimensions. The ring artifact removal algorithm developed as part of this thesis involves bilateral filtering in one dimension along the pixel elements and also mean filtering in a second dimension along the various projections taken for the sinogram. We have already

discussed one dimensional bilateral filtering and a one dimensional mean filtering in the previous sections. They can be combined together in an efficient fashion using matrix transpose. Since it is a very important component of many algorithms, instead of using an inefficient matrix transpose developed as part of the research, a very efficient transpose using a technique called diagonal block reordering developed by NVIDIA will be used. Let us now discuss about the technique in detail.

During this entire discussion the execution time of a matrix transpose will be compared to that of a matrix copy. Matrix copy can be implemented on the GPU very effectively at timings that are negligible compared to that of a bilateral filter or a mean filter. Thus if we can implement matrix transpose as efficiently as a matrix copy then we can use matrix transpose as an integral part of all the algorithms without any deterioration in performance. The timings shall be measured for operations done on a 2048 x 2048 matrix. The input and output matrices on the GPU are represented by the single dimensional arrays idata and odata respectively. A variable 'nreps' is used to represent the number of times data movement is performed between idata and odata. Also this variable is used in two different ways outside the kernel (Loop over kernel) and also inside the kernel outside the for loop doing the main functionality (Loop in kernel). In all versions of matrix transpose and matrix copy a thread block is responsible for doing the required functionality on a 32 x 32 tile in the matrix. The thread block is launched with 32 x 8 threads with each thread responsible for the operations on 4 elements. The bandwidth of matrix copy and matrix transpose at all stages of optimization as measured by NVIDIA will be shown in table 5.9.

Let us first discuss the naive versions of matrix transpose and matrix copy. In both the cases a thread reads data from idata and writes to odata, using the same location in case of matrix copy and the transpose location in case of matrix transpose. The effective bandwidths for both the cases are shown in table 5.9. As we can see the bandwidth of the naïve matrix transpose is very less compared to the bandwidth of naïve matrix copy.

One of the defects in the naive transpose algorithm is related to memory coalescing. In matrix copy both the arrays are accessed in a coalesced fashion whereas in matrix transpose only idata is accessed in a coalesced fashion. In coalesced transactions the threads in a half warp access 16 contiguous 32-bit words, or one half of a row of a tile. In the naive

copy all the threads in the half warp write to odata in a similar manner as a read from idata in coalesced fashion. However in the case of naïve transpose different threads in a half warp write data to different 16-byte segments resulting in 16 different transactions, irrespective of the compute capability. A method for bypassing this issue is to use shared memory in the kernel (__shared float tile[32][32]) . We first load the tile into the shared memory. Then we make the threads in a half warp access noncontiguous locations in the shared memory to put the data into contiguous locations in the global memory. A similar shared memory copy has also been implemented for comparison. The effective bandwidths for both the cases are shown in table 5.9.

Again as we can see from the table above even though the memory is being accessed in a coalesced manner there has not been a significant improvement in the performance of the algorithm. We need to identify other factors affecting the bandwidth of the algorithm. One possible factor might be shared memory conflicts. While writing the data to odata all the threads in a half warp access data from the same memory bank leading to 16 way bank conflict. Thus the next optimization technique we adopt is removal of shared memory bank conflicts. We can just do this by increasing the size of the number of columns in shared memory by 1 (__shared float tile[32][33]) . By doing this we make the shared memory access completely conflict free. Again the effective bandwidths for both the cases are shown in table 5.9.

As we can see from the table shared memory bank conflicts is not the main cause behind the bandwidth loss in matrix transpose. The paper ascribes the real cause behind the loss in performance to a new phenomenon called partition camping. As we have seen in chapter 4 shared memory is divided into 16 banks of 32 bit width each. To access shared memory effectively all the threads in a half warp should access memory in different banks otherwise bank conflicts will result. Similarly global memory is divided into either 6 partitions (on 8- and 9- series GPUs) or 8 partitions (on 200- and 10-series GPUs) of 256-byte width each. To access global memory effectively all the active half warps should be evenly divided amongst partitions. If they are not done so, many half warps ‘camp’ at relatively few partitions decreasing the effective bandwidth considerably. When thread

blocks are launched they are assigned to the multiprocessors based on an one-dimensional block ID given by the following equation

bid = blockIdx.x + gridDim.x * blockIdx.y (5.7)

Let us discuss how the 2048 x 2048 matrix is assigned to the various partitions when we have 8 partitions. Of the matrix we can assign 64 floats to each of the partitions. The first 512 floats of row 1 are assigned to the 8 partitions with 64 floats to each partition. Again 512 floats in the same row 1 will be assigned to the 8 partitions and so on until all the 2048 elements in a row will be assigned to the 8 partitions in 4 cycles. The next row in the matrix will be assigned in a similar manner. Thus all the columns in the matrix will be assigned to the same partition. We know that each block handles a tile of 32 x 32 elements. Using the above equation the blocks are assigned to global memory partitions as shown in the following figure.

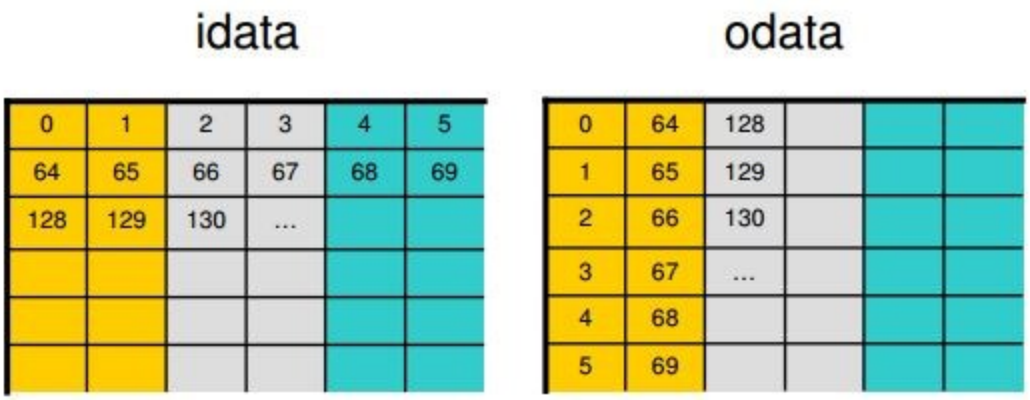


Figure 5.20 Partition Camping

The solution proposed to this phenomenon in the paper is a technique called diagonal reordering. The programmer cannot change the way blocks are assigned to the multiprocessors but he can change the way the launched blocks operate on the data. The launched blocks can be ordered according to Cartesian coordinates or diagonal coordinates as shown in the following figure

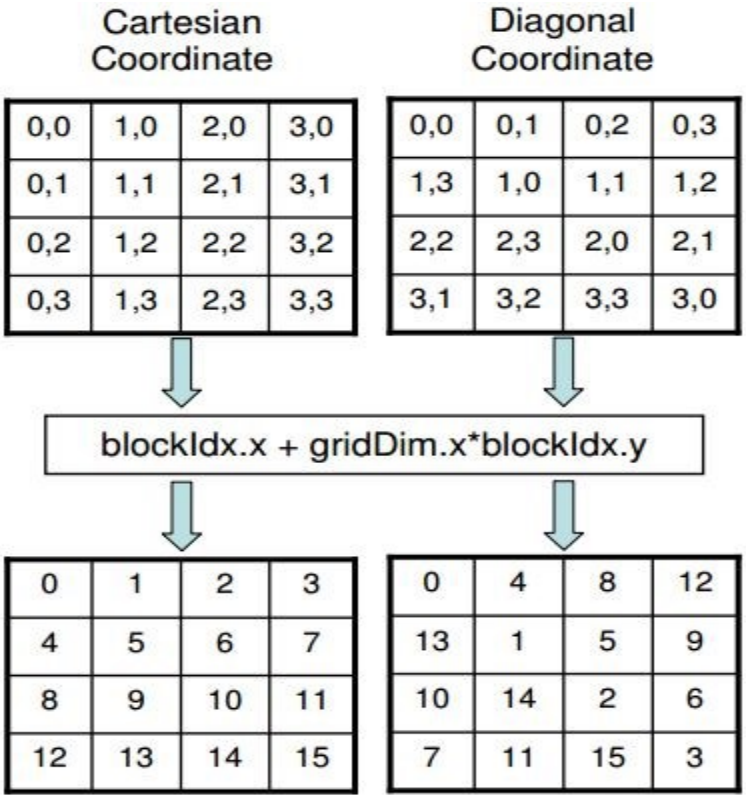


Figure 5.21 Diagonal Reordering

To change the way in which the launched blocks operate on the data we use the following equations to get the modified block numbers

$$blockIdx\,\text{modified}.y = blockIdx.x; \tag{5.8}$$

$$blockIdx\,\text{mod}\,ified.x = (blockIdx.x + blockIdx.y)\,\%\,gridDim.x; \tag{5.9}$$

The assignment of blocks to the tiles in the matrix using diagonal coordinates is different from the assignment of blocks using Cartesian coordinate as shown in the following figure

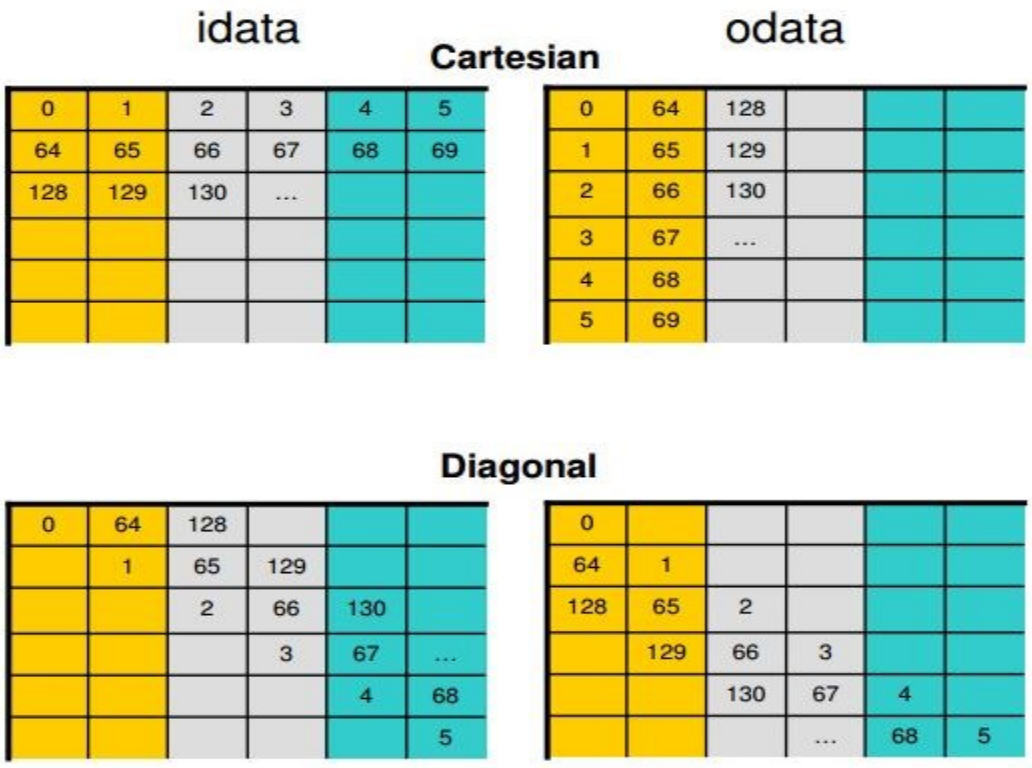


Figure 5.22 Partition Camping Avoided

As we can see in the above figure the pairs of blocks (0,1), (2,3), (4,5) and so on will read data from different partitions in case of both reading data from idata and writing data to odata. Thus the problem of partition camping doesn't arise. The bandwidth of diagonal reordering as compared to all the previous bandwidth optimization techniques as well as that compared to matrix copy is shown in table 5.9. We can see that matrix transpose using diagonal reordering increases bandwidth to be almost equal to that of matrix copy. Hence it can be used as an integral part of all the preprocessing and postprocessing algorithms without any deterioration in performance.

	Effective Bandwidth (GB/s) 2048x2048, GTX 280	
	Loop over kernel	Loop in kernel
Simple Copy	96.9	81.6
Shared Memory Copy	80.9	81.1
Naïve Transpose	2.2	2.2
Coalesced Transpose	16.5	17.1
Bank Conflict Free Transpose	16.6	17.2
Fine-grained Transpose	*80.4*	*81.5*
Coarse-grained Transpose	*16.7*	*17.1*
Diagonal	69.5	78.3

Table 5.9 Matrix Transpose Timings as given by NVIDIA

CHAPTER 6. PREPROCESSING ON THE GPU

6.1 Introduction

This chapter discusses the new preprocessing and postprocessing algorithms that have been developed as part of the research and also their implementation on the GPU using CUDA. The ring artifact removal algorithm improves the quality of the input sinogram data before reconstruction by using bilateral or median filtering discussed in the previous chapter. The hot pixel algorithm employs a new shifted sliding window method to remove contiguous hot pixels in the preprocessing stage. The beam hardening algorithm improves the quality of the reconstructed image by removing artifacts caused by differential absorption of photons using both the bilateral filtering and the GPU matrix transpose discussed before. We shall now see each of these algorithms in detail.

6.2 Ring artifact removal and implementation on the GPU

Ring artifacts are an extremely common feature in x-ray tomography. They may be caused due to many factors. Defective pixels with nonlinear responses to incoming intensity also called dead pixels are one of the major causes of ring artifacts. They cause sharp rings in the reconstructed image and vertical stripes in the input sinogram. If there are contiguous dead pixels on the detector we get bands of rings and stripes in the image and the sinogram respectively. Apart from the dead pixels miscalibrated pixels also cause ring artifacts less marked than those caused by dead pixels. Ring artifacts can also be caused by scintillator screens affected by dust, dirt or scratches. Variation in the intensity of the incoming fan beam is one more cause of ring artifacts. In the final reconstructed image, the intensity of these artifacts is also affected by tube voltage. The gray values in the reconstructed slices are modified by these ring artifacts. As a result analysis of the reconstructed image for noise reduction or image segmentation becomes difficult. Therefore removal of ring artifacts is an integral part of the reconstruction process. The following figures show ring artifacts in a sinogram and also in a reconstructed slice.

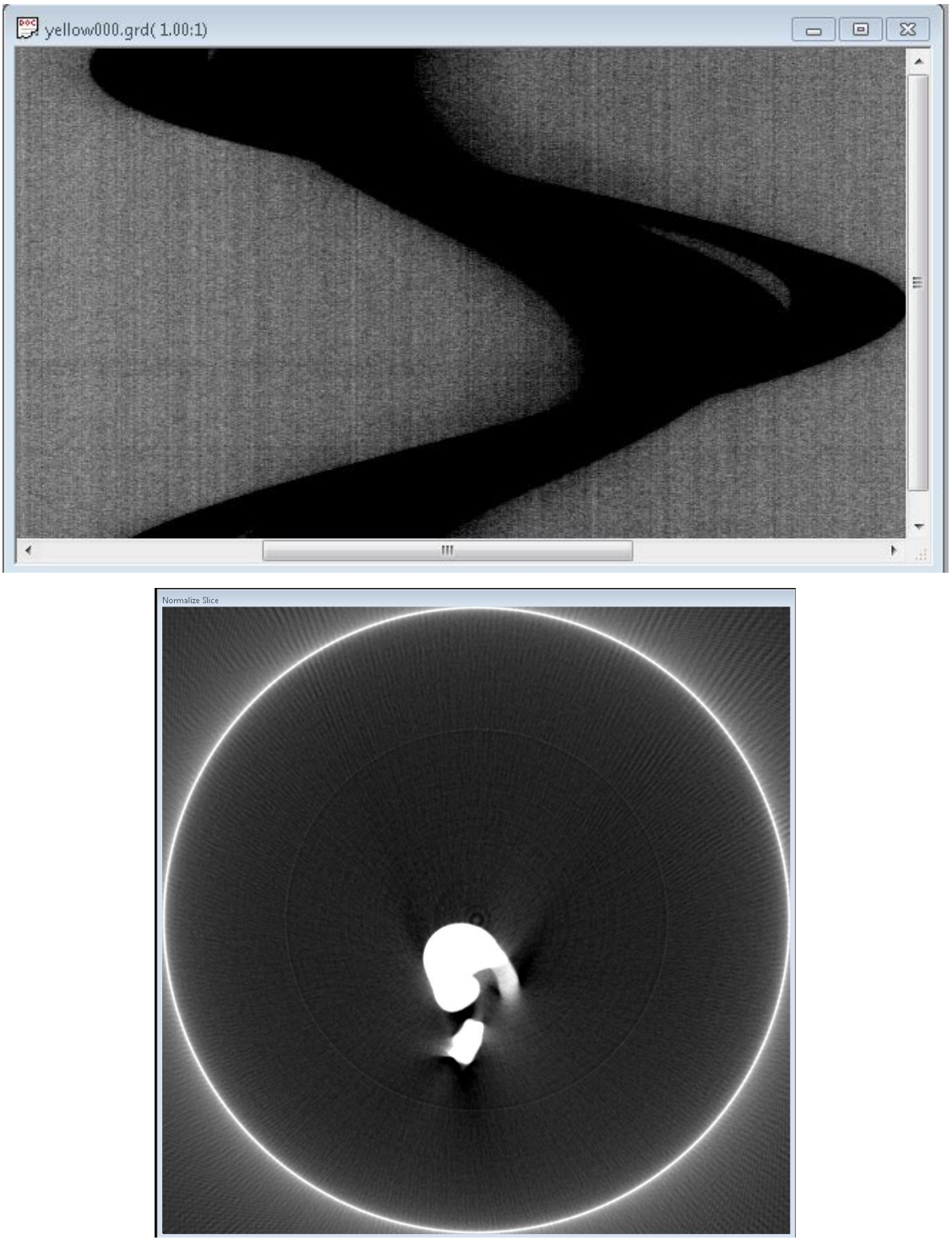


Figure 6.1 Ring artifacts in a sinogram and a slice

Many different approaches have been developed over time for ring artifact removal. They can be mainly classified into three categories namely methods based on adjusting the experimental setup, methods running artifact removal algorithms in the preprocessing stage on the sinogram and methods running artifact removal algorithms in the post processing stage on the reconstructed slice. Both the preprocessing and post processing techniques can be done in different geometric planes (polar or Cartesian). In general, ring artifact removal done in the polar plane is more effective than that done in the Cartesian plane. The most commonly used methods in the first category are flat field correction method and movable detector array method. In movable detector array approach to reduce the non-uniform sensitivity of different detector elements the detector array is moved during data acquisition. The suppression of ring artifacts is done by averaging the values of detector pixels. This is not a cost effective method since a special arrangement of detector is needed. Flat field correction is a standard calibration procedure used for everything from cameras to telescopes. In flat field correction the object is bombarded with the x-ray beam twice. The first time image acquisition is done without placing the object in the x-ray beam. The second time image acquisition is done placing the object in the x-ray beam. The frames thus obtained are subtracted and adjusted to get the calibrated image frame. Since they are not cost effective many methods have been developed which are not based on hardware correction. Emraan et al [38] discusses methods to classify ring artifacts into different categories and use adaptive correction schemes so that different or slightly modified algorithms can be used for ring artifact removal. As we have seen before ring artifacts caused by defective detector pixels and by defects in the scintillator screen are brighter compared to ring artifacts caused by miscalibrated detector pixels. The first category rings are classified as type I rings and a different ring artifact removal method is used for them as them as compared to the second category rings which are classified as type II rings. Raven [58] implements ring artifact removal algorithm using Butterworth low pass filter. Using this method type I ring artifacts are completely removed whereas type II ring artifacts are suppressed to a great extent without significant loss of information. Hasan et al. [56] discusses ring artifact removal using different iterative morphological filters (IMF) for different ring patterns. Lee et al. [57] discusses ring artifact removal using 2D sliding variable size average and weighted average

filters. One more method using combined wavelet Fourier filtering was proposed in Munch et al. [36]. This thesis introduces a novel method for ring artifact removal using bilateral filtering or median filtering and also implements this algorithm on a CUDA enabled GPU with very good speedup. We have already studied the main components involved in the previous chapter namely bilateral filtering on the GPU, median filtering on the GPU, matrix transpose using diagonal reordering on the GPU and mean filtering on the GPU. In this chapter we shall see how they combine effectively to remove ring artifacts from a sinogram.

The ring artifact removal algorithm has the following steps:

➔ An edge preserving smoothing spatial filter (median or bilateral) is applied on the input sinogram to get the filtered sinogram.

➔ The filtered sinogram data is then subtracted from the original data to get the trend composed of the ring artifact pattern, noise pattern and some small details lost during filtering.

➔ This trend is averaged across the rows of the sinogram to get the ring artifact pattern.

➔ The ring artifact pattern is then subtracted from the original sinogram to get the sinogram data without the ring artifacts.

Let us now describe each of the steps and their effect on the sinogram shown in the below figure in detail. As we can see in the following figure the sinogram has many ring artifacts which should be eliminated using the ring artifact removal algorithm.

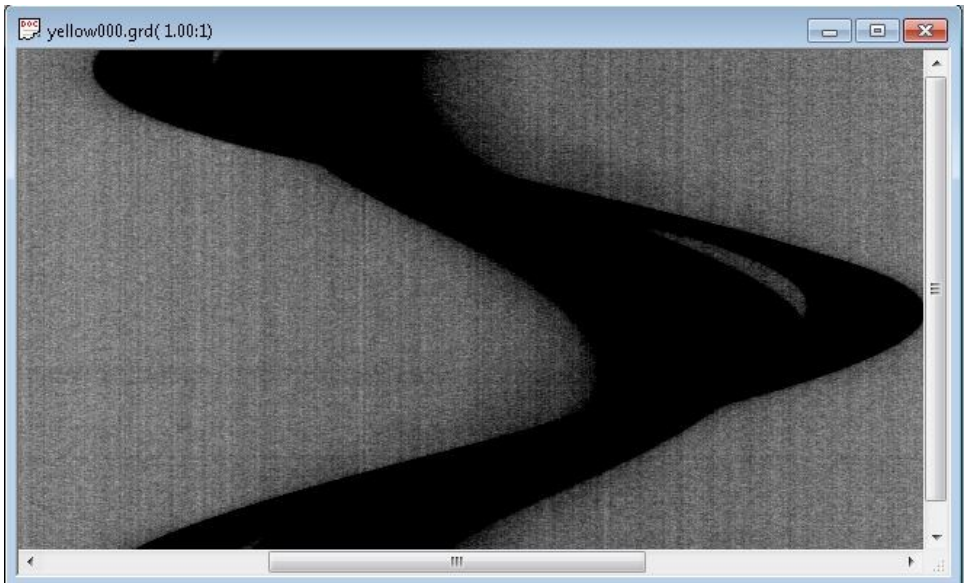


Figure 6.2 Ring artifacts to be removed by the algorithm

We have already discussed and implemented a bilateral filter in the previous chapter. The effect of applying a bilateral filter with radius = 30, sigma domain value = 10, sigma range value = 50 and number of iterations = 1 on the sinogram is as shown below.

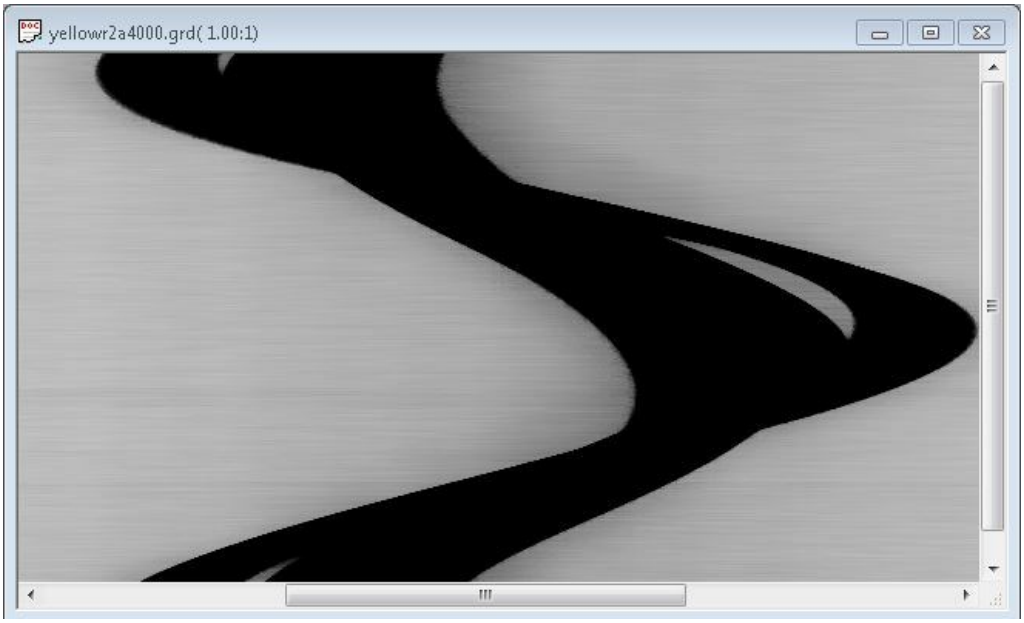


Figure 6.3 Applying bilateral filter on the sinogram

We can see from the above figures that not only are the rings present in the sinogram removed completely but some small features present in the image as well as the noise are also eliminated. The small scale details removed (whose size is dependent on radius of the filter) should be preserved in the filtered sinogram. If we subtract the filtered sinogram from the original sinogram we are left with trend as shown in the following figure

Figure 6.4 Trend obtained by the algorithm

The trend shown above includes the following three components

- ➔ Ring artifacts
- ➔ Noise in the original sinogram
- ➔ Small features lost due to filtering

The rings which are vertical constants are thus obscured by the other components in the trend. In order to extract the ring artifact pattern from the above data we average out the rows in the vertical direction. This averaging not only suppresses the random noise, it will also average out the details that were accidently picked up by the filter and which ideally should have been preserved. Applying the averaging along the rows transforms the trend data shown above to the data shown below which shows the ring artifacts present in the sinogram.

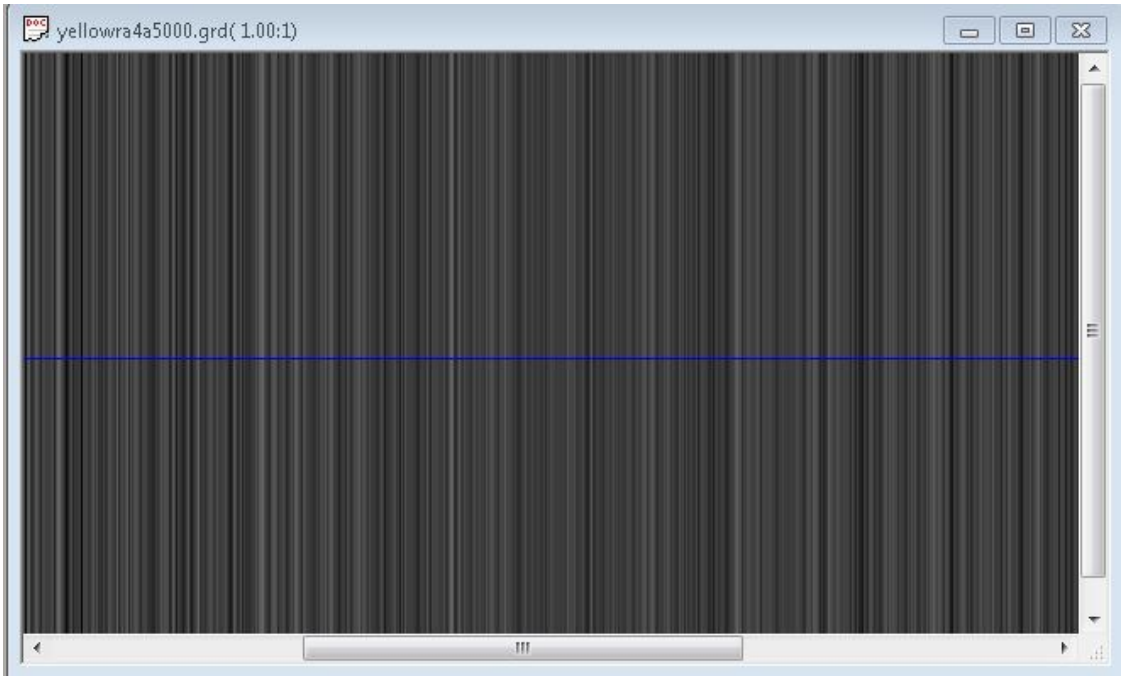
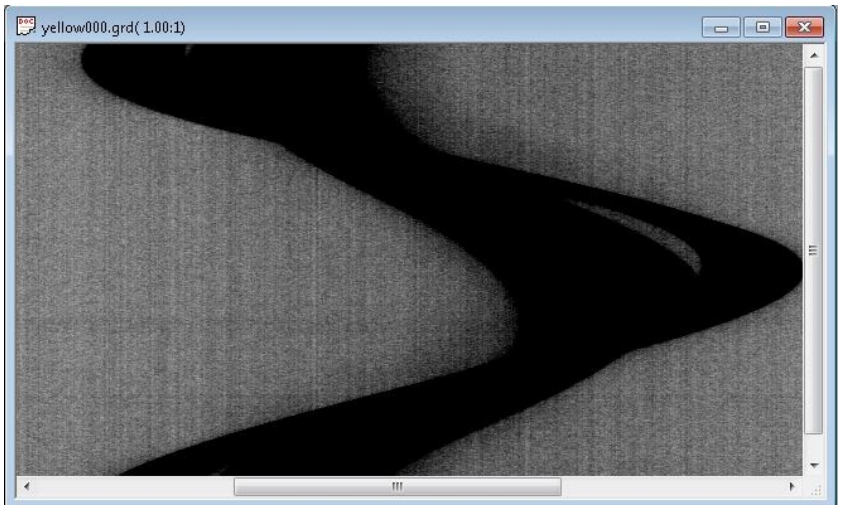


Figure 6.5 Ring artifacts in the sinogram

Now that we have isolated the ring artifact pattern we need to subtract them from the original sinogram to get the ring artifact corrected sinogram. The below figure shows the data in all the three stages of the algorithm namely the sinogram data, the ring artifact pattern and the ring artifact corrected sinogram.



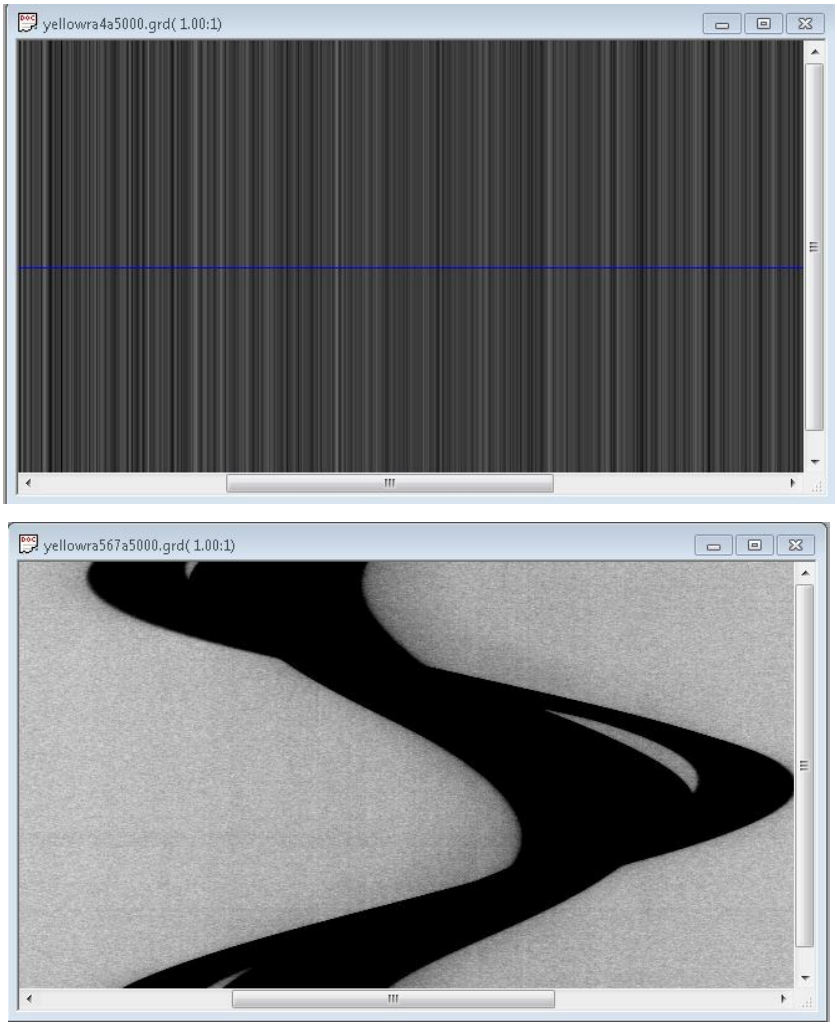


Figure 6.6 Original Sinogram, ring artifacts in the sinogram and sinogram without ring artifacts

The effect of applying this algorithm on the reconstructed slice is as shown below.

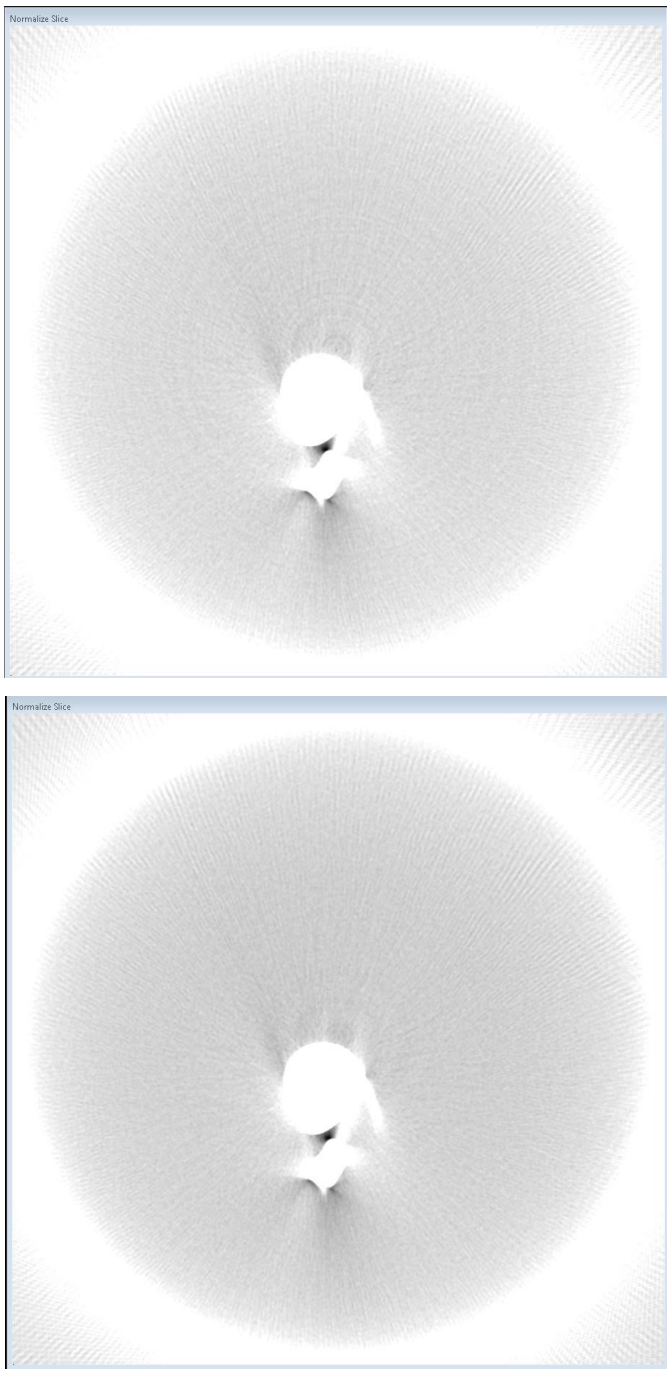


Figure 6.7 Original slice, Slice with ring artifacts removed

The size of the filter may affect the small details removed but the averaging filter corrects out the details removed. Hence the size of the filter almost has not effect on the image as shown below using filters of different sizes.

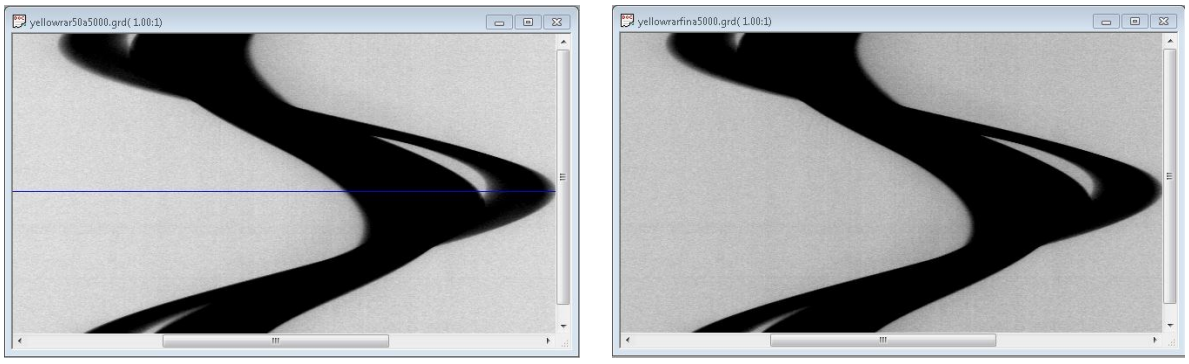


Figure 6.8 Ring artifact removal using bilateral filters of size 50 and 70

Instead of a bilateral filter we can also use a median filter in the algorithm. The effect of ring artifact removal with median filtering is shown in the following figure.

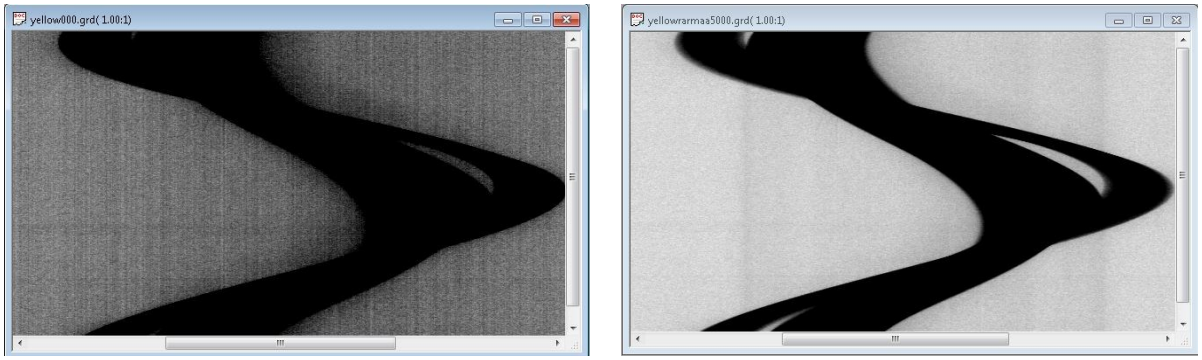


Figure 6.9 Ring artifact removal using median filtering

We shall now discuss the implementation of the ring artifact removal algorithm on the GPU. The inputs of the algorithm are as follows:

- ➔ Location of the first sinogram in the dataset
- ➔ The first file in the dataset to be corrected
- ➔ Number of files to be corrected
- ➔ Number of iterations of the algorithm to be implemented
- ➔ Filter to be used for ring artifact removal(median or bilateral)
- ➔ Radius of the median filter (in case of median filtering)
- ➔ Radius of the bilateral filter (in case of bilateral filtering)
- ➔ Sigma domain value (in case of bilateral filtering)
- ➔ Sigma range value (in case of bilateral filtering)

➔ Location of the corrected sinogram files if they are to be written

The output of the algorithm is as follows:

➔ Filtered sinogram dataset at the destination specified by the end user.

The application of the ring artifact removal algorithm on an individual pixel in a row of the sinogram data is independent of the application on other pixels. Hence the algorithm is highly parallelizable and is very suitable for porting onto the GPU. Also the application of the algorithm on a single sinogram file is independent of the application on other sinogram files. Thus all the sinogram files are handled in a sequential manner. A sinogram file is first copied from the CPU onto the GPU. Then the kernels responsible for the algorithm are launched on the GPU. The corrected sinogram data is then copied back from the GPU onto the CPU. This process is then repeated for all the sinogram files to be corrected.

We shall now discuss each of the kernels in detail in the order they are launched:

➔ mef<<<>>> :

Functionality: Responsible for computing the trend by doing median filtering on the sinogram dataset (if median filtering is chosen by the end user for ring artifact removal).

Description: Already described in the previous chapter.

➔ bifgv1<<<>>> :

Functionality: Responsible for computing the trend by doing bilateral filtering on the sinogram dataset (if bilateral filtering is chosen by the end user for ring artifact removal).

Description: Already described in the previous chapter

➔ avgfv1<<<>>> :

Functionality: Responsible for averaging the rows of the trend to get the ring artifact pattern.

Execution configuration parameters:

$$\begin{Bmatrix} (blockDim.x = 16), \\ (blockDim.y = 1), \\ (gridDim.x = 1), \\ (gridDim.y = ((width - 2 * avgFilRad)/blockDim.x)(+1 \; if \; not \; an \; exact \; multiple), \\ (\text{shared memory size } = 0 \text{ bytes}) \end{Bmatrix}$$

Description: We launch the kernel as a 2D grid of blocks with blocks having the same ($blockIdx.x$) value responsible for averaging elements in 16 of the columns in the trend. Each block is in turn composed of 16 threads with each thread responsible for averaging one of the columns. The elements in all the rows of the trend are loaded successively and the average of the columns is calculated and stored in a register. Finally the averaged values are loaded back into each of the rows in the global memory. Each of these stages is synchronized using __syncthreads() to ensure instruction synchronization and memory visibility.

Performance tuning: Global memory is always accessed in a coalesced fashion across all compute capabilities. Registers are used instead of shared memory since each of the pixel value is used just once. Warp divergence is totally absent. The execution configuration parameters have been chosen so as maximize the warp occupancy on Tesla C1060 as measured using Nvidia Parallel Nsight 3.0. Since Tesla C1060 has 30 multiprocessors, having a low number of threads (16) ensures that each of the multiprocessor is assigned at least one block.

Future improvements: Use of texture memory to take advantage of spatial locality of the algorithm. Port the algorithm onto the next generation NVDIA architectures (Kepler and Fermi).

➔ arrayOperations<<<>>> :

Functionality: Responsible for adding, subtracting, dividing or multiplying the input arrays.

Execution configuration parameters:

$$\begin{cases} (blockDim.x = 512), \\ (blockDim.y = 1), \\ (gridDim.x = step), \\ (gridDim.y = ((width - 2 * \text{bi}lFilRad)/blockDim.x)(+1 \ if \ not \ an \ exact \ multiple), \\ (\text{shared memory size } = 0 \text{ bytes}) \end{cases}$$

Description: We launch the kernel as a 2D grid of blocks with blocks having the same ($blockIdx.x$) value responsible for elements in one of the rows of the sinogram. Each block is in turn composed of 512 threads with each thread responsible for doing the arithmetic operations on one of the pixels. The array values are loaded, operated upon and then loaded back into the global memory. Each of these stages is synchronized using __syncthreads() to ensure instruction synchronization and memory visibility.

Performance tuning: Global memory is always accessed in a coalesced fashion across all compute capabilities. Registers are used instead of shared memory since each of the pixel value is used just once. Warp divergence is totally absent. The execution configuration parameters have been chosen so as maximize the warp occupancy on Tesla C1060 as measured using Nvidia Parallel Nsight 3.0.

Future improvements: Use of texture memory to take advantage of spatial locality of the algorithm. Port the algorithm onto the next generation NVDIA architectures (Kepler and Fermi).

The time needed for the implementation of ring artifact removal algorithm for varying parameters using bilateral filter and median filter are shown in the tables below and the corresponding graphs are also shown.

Radius of filter	Time(milliseconds)		
	1 iteration	10 iterations	20 iterations
30	18	180	343
50	30	270	515
70	45	350	687

Table 6.1 Timings for ring artifact removal algorithm using a bilateral filter

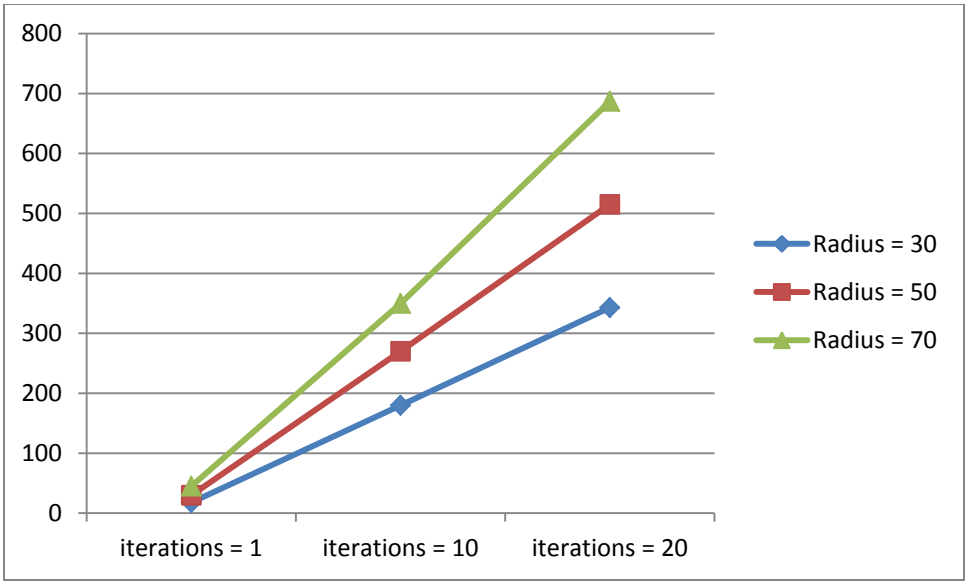


Figure 6.10 Timings for ring artifact removal algorithm using a median filter

6.3 Hot pixel removal algorithm and implementation on the GPU

Hot pixels are also caused by defective detector element's responses. However, hot pixels have a much higher deviation with respect to the neighboring pixels as opposed to ring artifacts. The ring artifact removal algorithm even if it is run for multiple iterations is not effective in removing the hot pixels. Hence a separate algorithm has been developed to remove the hot pixels based on the error tolerance entered by the end user.

An algorithm has been developed for hot pixel correction as part of research conducted by Nadia at CNDE. We will first discuss the existing algorithm and its defects and later discuss the modified algorithm. We will be using the following terminology during the discussion

x = Pixel number in a sinogram having (nx) columns,

y = Projection number in a sinogram having $(ntheta)$ rows,

$w[x][y]$ = One of the elements in the sinogram data

$numDev$ = Number of pixels used in the calculation of deviation entered by the end user (for the modified hot pixel algorithm),

$numPix$ = Number of possible contiguous hot pixels entered by the end user (for the modified hot pixel algorithm),

noisem[*x*][*y*] = The noise of one of the elements in the sinogram data calculated using both the left window and the right window,

noisel[*x*][*y*] = The noise of one of the elements in the sinogram data calculated using only the left window twice (in the modified hot pixel algorithm),

noiser[*x*][*y*] = The noise of one of the elements in the sinogram data calculated using only the right window twice (in the modified hot pixel algorithm),

devm[*x*] = The root mean square deviation of one of the elements in the sinogram data calculated using *noisem*[*x*][*y*],

devl[*x*] = The root mean square deviation of one of the elements in the sinogram data calculated using *noisel*[*x*][*y*] (in the modified hot pixel algorithm),

devr[*x*] = The root mean square deviation of one of the elements in the sinogram data calculated using *noiser*[*x*][*y*] (in the modified hot pixel algorithm),

runningMean = Varying mean computed using deviations in the left and right window (in the modified hot pixel algorithm),

runningStddev = Varying standard deviation computed using deviations in the left and right window (in the modified hot pixel algorithm),

errTol = error tolerance entered by the end user for hot pixel identification,

p = pixel under consideration

The hot pixel algorithm written by Nadia works as follows:

1) Allocate memory for the sinogram data, the deviation data and the noise data namely *w*[*x*][*y*], *devm*[*x*] and *noisem*[*x*][*y*] respectively.

2) Calculate the noise of each of the pixels for all the projections in the CT scan. The noise calculation in this algorithm involves four of the neighboring pixels in the sinogram (two to the left and two to the right).

$$noisem[x,y] = 6 \times w[x,y] - 5 \times (w[x-1][y] + w[x+1][y]) + 2 \times (w[x-2][y] + w[x+2][y]) \quad (6.1)$$

3) Calculate the deviation strip as root mean square of noise along the projections for each pixel in the sinogram.

$$devm[\,x\,] = \sum_{y=1}^{ntheta} (\; noisem[\,x\,][\,y\,] * noisem[\,x\,][\,y\,]\;) \tag{6.2}$$

$$devm[\,x\,] = \sqrt{devm[\,x\,] / (\,ntheta - 1)} \tag{6.3}$$

4) Calculate the mean of the deviation strip.

$$mean = \sum_{x=1}^{nx} devm[\,x\,] \tag{6.4}$$

$$mean = mean / nx \tag{6.5}$$

5) Calculate the standard deviation of the deviation strip.

$$stddev = \sum_{x=1}^{nx} (\,devm[\,x\,] - mean\,)\,(\,devm[\,x\,] - mean\,) \tag{6.6}$$

$$stddev = sqrt(stddev / (nx - 1)) \tag{6.7}$$

6) Check whether each of the pixels p meets the deviation criteria for being a hot pixel or not. A pixel may be classified as a hot pixel if it follows the following two deviation criteria

a) The absolute difference between the deviation of a pixel and the mean of the deviation strip is greater than the product of error tolerance and standard deviation of the deviation strip.

$$\left| devm[\,p\,] - \text{mean} \right| > (\,\text{errTol} \times \text{stddev}\,) \tag{6.8}$$

b) The deviation of the pixel is greater than or lesser than the deviations of two of the neighboring pixels to the left and two the neighboring pixels to the right.

$$\forall i \in \{1,2\}\,((\,devm[p] < devm[p-i]\,) \wedge (\,devm[p] < devm[p+i]\,)) \tag{6.9}$$

OR

$$\forall i \in \{1,2\}\,((\,devm[p] > devm[p-i]\,) \wedge (\,devm[p] > devm[p+i]\,)) \tag{6.10}$$

7) If a pixel is identified as a hot pixel, replace its gray scale value with the average values of the two neighboring pixels (one to the left and one to the right).

$$w[\,x,y\,] = (\,w[\,x\text{-}1,y\,] + w[\,x+1,y\,]\,) / (2.0) \tag{6.11}$$

The application of this algorithm to the slice data is shown below.

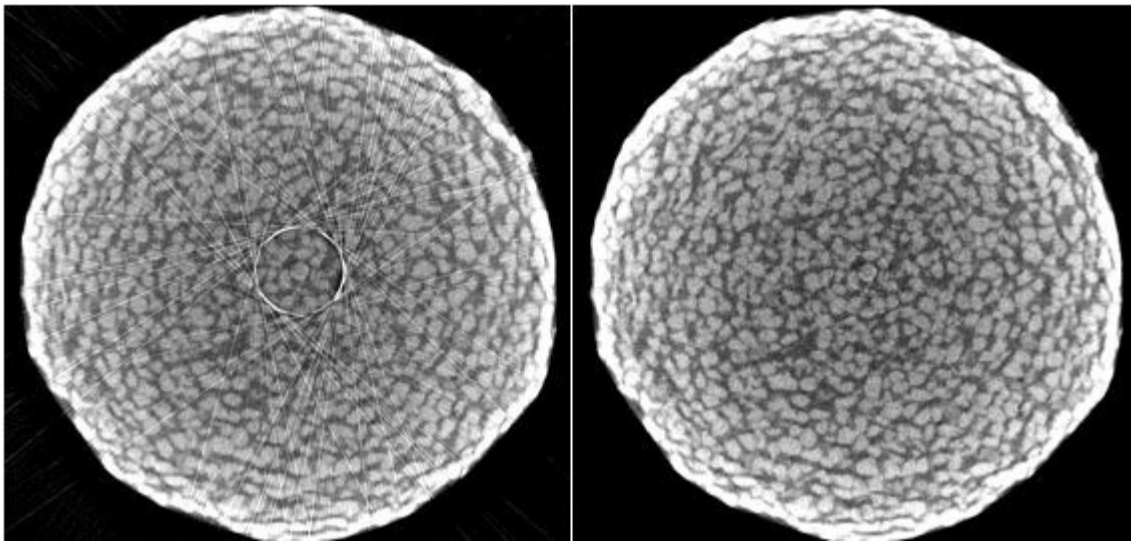


Figure 6.11 Effect of Hot pixel removal algorithm (Nadia) on a slice

The change in gray scale values before and after applying hot pixel removal algorithm are also shown below.

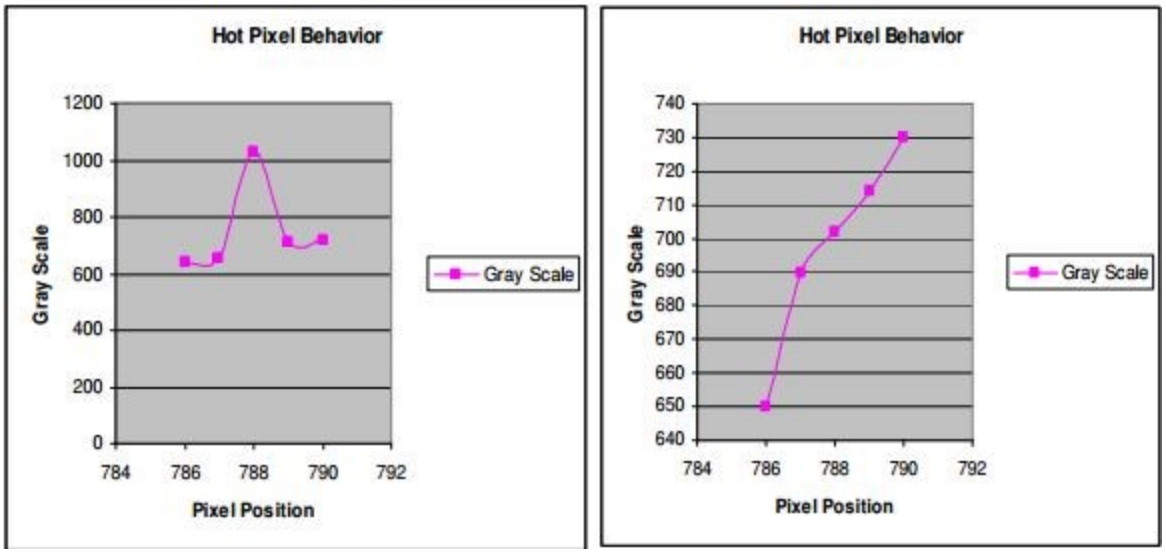


Figure 6.12 Effect of Hot pixel removal algorithm (Nadia) on gray scale values

The existing algorithm has some defects in it and does not function in an entirely accurate fashion.

The defects are as follows:

The deviation criterion (6a) is not accurate as the deviation of a pixel is compared to the mean and standard deviation of the entire deviation strip. The mean and standard deviation of the entire strip is not an accurate measure as the pixel values can vary a lot across edges of the object being analyzed and also otherwise.

The deviation criterion (6b) is also not accurate as the neighboring pixels themselves might be hot pixels thereby affecting the comparison of the deviations.

If a normal pixel has a neighboring hot pixel the deviation of the normal pixel is affected a lot. This incorrect deviation is then used for comparison during hot pixel identification

Due to the defect mentioned in point 3 the hot pixel's correct deviation will be compared with a neighboring normal pixel's incorrect deviation.

The averaging done in (7) is also not accurate since four of the neighboring pixels are being used for calculation of noise in (2) and for comparison in (6b) while only two of the neighboring pixels are being used to put in the average value.

Because of the defects described above a modified algorithm is proposed in this thesis. We shall first discuss how the existing defects are overcome and then discuss the new modified hot pixel removal algorithm. In this entire discussion each pixel will be associated with a left window and a right window of pixels. The left window is composed of $(numDev)$ pixels taken by leaving out $(numPix-1)$ pixels on the left whereas the right window is composed of $(numDev)$ pixels taken by leaving out the $(numPix-1)$ pixels on the right.

The modifications made to the existing algorithm (presented in order of the above defects) are as follows.

Instead of comparing the deviation of a pixel with the mean and standard deviation of the entire deviation strip, we just compare it with the mean and standard deviation of the deviations of the left and right window of pixels. Since in most of the cases the pixels in the immediate neighborhood have similar values this is a more accurate measure compared to taking the entire deviation strip.

We modify the deviation criterion in (6b) to compare the deviations of the main pixel to the pixels in its left and right window. Thus comparison with contiguous hot pixels (if any present) is avoided.

The deviations of pixels in the neighborhood of the hot pixels are not affected as the window involved in their calculation will exclude the hot pixels. The deviations of pixels in the left and right window will still be affected by the hot pixels and will have abnormal mean and standard deviation values. However they will not be identified as hot pixels as the abnormal standard deviation will more than compensate for the abnormal mean even for a small error tolerance entered by the end user.

For each of the pixels three deviation values are calculated. The first deviation $devm[x]$ is calculated in the normal fashion taking pixels in the left window and pixels in the right window. The second deviation $devl[x]$ is calculated taking only the pixels in the left window twice. The third deviation $devr[x]$ is calculated taking only the pixels in the right window twice. Then the first deviation of a pixel $devm[x]$ is compared with the second deviation of pixels in the left widow $devl[x]$ and third deviation of pixels in the right window $devr[x]$. Thus if a pixel is a hot pixel then its abnormal deviation value will be compared with the normal deviations values in both the left and the right window.

The mean of gray scale values of pixels in the left and right window is used instead of just averaging two neighboring pixels.

The modified hot pixel algorithm used in this thesis works as follows.

1) Allocate memory for the sinogram data, the deviation data and the noise data ($w[x][y]$, $devm[x], devl[x], devr[x], noisem[x][y], noisel[x][y]$ and $noiser[x][y]$).

2) Calculate the noise of each of the pixels for all the projections in the CT scan. For the calculation of the three proposed deviation values three different noise values are calculated for each pixel using the following equations.

For example if $(numDev = 2)$

$$noisem[\,x,y\,] = 6 \times w[\,x,y\,] - 5 \times (\,w[\,x\text{-}numPix\,][\,y\,] + w[\,x + numPix\,][\,y\,]\,) + 2 \times (\,w[\,x\text{-}numPix\text{-}1\,][\,y\,] + w[\,x + numPix + 1\,][\,y\,]\,) \quad (6.11)$$

$$noisel[\,x,y\,] = 6 \times w[\,x,y\,] - 5 \times (w[\,x\text{-}numPix\,][\,y\,] + w[\,x\text{-}numPix\,][\,y\,]) + 2 \times (w[\,x\text{-}numPix\text{-}1\,][\,y\,] + w[\,x\text{-}numPix-1\,][\,y\,]) \quad (6.12)$$

$$noiser[\,x,y\,] = 6 \times w[\,x,y\,] - 5 \times (w[\,x+numPix\,][\,y\,] + w[\,x+numPix\,][\,y\,]) + 2 \times (w[\,x+numPix+1\,][\,y\,] + w[\,x+numPix+1\,][\,y\,]) \quad (6.13)$$

3. Calculate the three deviation values as root mean square noise for each pixel in the slice.

$$devm[\,x\,] = \sum_{y=1}^{ntheta} (\; noisem[\,x\,][\,y\,] * noisem[\,x\,][\,y\,]) \quad (6.14)$$

$$devm[\,x\,] = \sqrt{devm[\,x\,] / (ntheta - 1)} \quad (6.15)$$

$$devl[\,x\,] = \sum_{y=1}^{ntheta} (\; noisel[\,x\,][\,y\,] * noisel[\,x\,][\,y\,]) \quad (6.16)$$

$$devl[\,x\,] = \sqrt{devl[\,x\,] / (ntheta - 1)} \quad (6.17)$$

$$devr[\,x\,] = \sum_{y=1}^{ntheta} (\; noiser[\,x\,][\,y\,] * noiser[\,x\,][\,y\,]) \quad (6.18)$$

$$devr[\ x\ ]\ =\ \sqrt{\text{devr}[\ x\ ]\ /\ (\ \text{ntheta - 1})}$$

(6.19)

3) For each of the pixels calculate the running mean of deviations using $(numDev)$ pixels in the left window and $(numDev)$ pixels in the right window.

For hot pixel removal using one deviation value,

$$runningMean = \sum_{i=1}^{(numDev)} devm[p-(numPix-1)-\mathrm{i}] + \sum_{i=1}^{(numDev)} devm[p+(numPix-1)+\mathrm{i}]$$

(6.20)

For hot pixel removal using three deviation values,

$$runningMean = \sum_{i=1}^{(numDev)} devl[p-(numPix-1)-\mathrm{i}] + \sum_{i=1}^{(numDev)} devr[p+(numPix-1)+\mathrm{i}]$$

(6.21)

4) For each of the pixels calculate the running standard deviation of deviations using $(numDev)$ pixels in the left window and $(numDev)$ pixels in the right window

For hot pixel removal using one deviation value,

$$runningStddev = \sum_{i=1}^{numDev} (devm[p-(\text{num}Pix-1)-i]-\text{runningMean})^2 + \sum_{i=1}^{numDev} (devm[p+(\text{num}Pix-1)+i]-\text{runningMean})^2 \quad (6.22)$$

For hot pixel removal using 3 deviation values,

`

$$runningStddev = \sum_{i=1}^{numDev} (devl[p-(\text{num}Pix-1)-i]-\text{runningMean})^2 + \sum_{i=1}^{numDev} (devr[p+(\text{num}Pix-1)+i]-\text{runningMean})^2 \quad (6.23)$$

5) Check whether each of the pixels meets the deviation criteria for hot pixel or not. A pixel may be classified as a hot pixel if it follows the following two deviation criteria

a. The absolute difference between the deviation of a pixel and the running mean is greater than the product of error tolerance and running standard deviation.

$$| devm[\ p\ ] - \text{runningMean} | > (\ \text{errTol} \times \text{runningStddev}\ ) \quad (6.24)$$

b. The deviation of the pixel either greater than or smaller than four of the neighboring pixels in the left window and the right window.

For hot pixel removal using one deviation value,

$$\forall i \in \{1,(numDev)\}((devm[p] < devm[p-(numPix-1)-i]) \wedge (devm[p] < devm[p+(numPix-1)+i]))$$

(6.25)

OR

$$\forall i \in \{1,(numDev)\}((devm[p] > devm[p-(numPix-1)-i]) \wedge (devm[p] > devm[p+(numPix-1)+i]))$$

(6.26)

For hot pixel using three deviation values,

$$\forall i \in \{1,(numDev)\}((devm[p] < devl[p-(numPix-1)-i]) \wedge (devm[p] < devr[p+(numPix-1)+i]))$$

(6.27)

OR

$$\forall i \in \{1,(numDev)\}((devm[p] > devl[p-(numPix-1)-i]) \wedge (devm[p] > devr[p+(numPix-1)+i]))$$

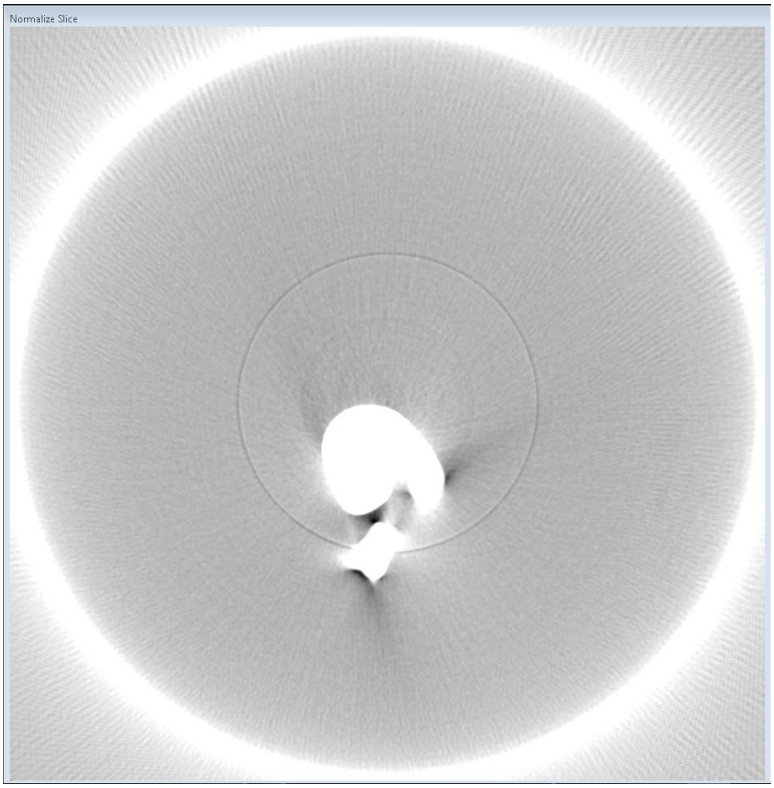
(6.28)

6) If a pixel is identified as a hot pixel, replace the gray scale value with the average values of the pixels in the sliding window.

$$w[x,y] = \sum_{i=1}^{numDev} w[x-(numPix-1)-i,y] + \sum_{i=1}^{numDev} w[x+(numPix-1)+i,y]$$

(6.29)

Application of the hot pixel removal algorithm to the slice data is as shown below.



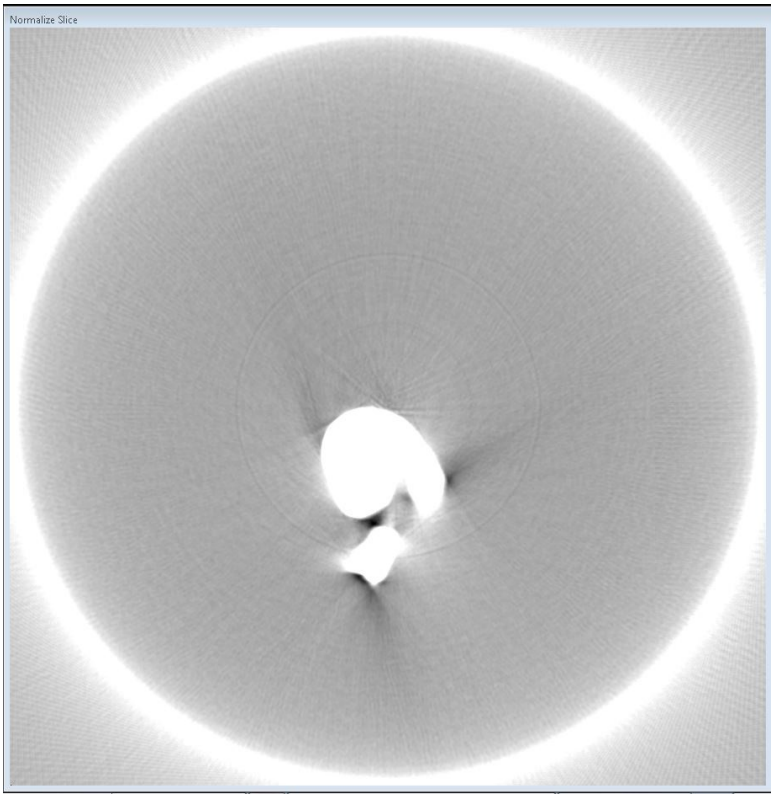


Figure 6.13 Applying hot pixel removal algorithm to a slice containing hot pixels

We shall now discuss the implementation of the modified hot pixel removal algorithm on the GPU. The inputs of the algorithm are as follows:

- ➔ Location of the first sinogram in the dataset
- ➔ The first sinogram in the dataset to be processed
- ➔ Number of files to be processed
- ➔ Error tolerance(also called deviation factor)
- ➔ Number of contiguous hot pixels to be corrected
- ➔ Number of pixels to be used in the calculation of deviation
- ➔ Number of different deviations to be used(one or three)
- ➔ Number of iterations of the algorithm to be done
- ➔ Location of the processed sinogram files if they are to be written

The output of the algorithm is as follows:

- ➔ Processed sinogram dataset with hot pixels removed at the destination specified by the end user

The application of the hot pixel removal algorithm on an individual pixel in a row of the sinogram data is independent of the application on other pixels. Hence the algorithm is highly parallelizable and is very suitable for porting onto the GPU. Also the application of the algorithm on a single sinogram file is independent of the application on other sinogram files. Thus all the sinogram files are handled in a sequential manner. A sinogram file is first copied from the CPU onto the GPU. Then the kernels responsible for the algorithm are launched on the GPU. The corrected sinogram data is then copied back from the GPU onto the CPU. This process is then repeated for all the sinogram files to be corrected. We shall now discuss each of the kernels in detail in the order they are launched:

➔ calcDev <<<>>> :

Functionality: Responsible for calculating the deviations required for hot pixel identification.

Execution configuration parameters:

$$\begin{cases} \left(blockDim.x = (32 + 2 * (\text{ numDev} + \text{numPix} - 1))\right), \\ (blockDim.y = 1), \\ (gridDim.x = 1), \\ (gridDim.y = ((\text{width} - 2 * (\text{numDev} + \text{numPix} - 1)) / 32)(+1 \text{ if not an exact multiple}) \\ \text{shared memory size=(blockDim.x * sizeof(float))} \end{cases}$$

Description: Each of the individual blocks having different $(blockIdx.y)$ values but the same $(blockIdx.\text{x})$ values is responsible for handling a group of 32 pixels. All the threads in the block first load one of the pixels each in a row from the global memory into the shared memory so that the 32 pixels as well as the left and right windows needed are loaded. Then 32 of these threads calculate the noise and deviations (one or three deviations) of one pixel each. The process is repeated for each of the rows in the sinogram and the so computed deviations are then loaded back into the global memory. Each of these stages is synchronized using __syncthreads() to ensure instruction synchronization and memory visibility.

Performance tuning: Global memory is always accessed in a coalesced fashion across all compute capabilities. Shared memory is used to reduce access to the high latency global memory as each of the pixel values are used multiple times and also because the

threads need to share the loaded pixel values with each other. Warp divergence is mostly absent. Shared memory is also accessed without any bank conflicts present. The execution configuration parameters have been chosen so as maximize the warp occupancy on Tesla C1060 as measured using NVIDIA Parallel Nsight 3.0. The kernel has been developed on a Tesla C1060 processor which has 30 multiprocessors. Having a small number of threads per block ensures that all the multiprocessors are occupied.

Future improvements: Use of texture memory to take advantage of spatial locality of the algorithm. Port the algorithm onto the next generation NVDIA architectures (Kepler and Fermi).

➔ findHP <<<>>> :

Functionality: Responsible for identifying the hot pixels based on the deviations calculated above.

Execution configuration parameters:

$$\begin{cases} (blockDim.x = (32 + 2 * (\text{ numDev} + \text{numPix} - 1))), \\ (blockDim.y = 1), \\ (gridDim.x = 1), \\ (gridDim.y = ((\text{width} - 2 * (\text{numDev} + \text{numPix} - 1)) / 32)(+1 \text{ if not an exact multiple}) \\ \text{shared memory size=(3 * blockDim.x * sizeof(float))} \end{cases}$$

Description: Each of the individual blocks having different $(blockIdx.y)$ values but the same $(blockIdx.\text{x})$ values is responsible for handling a group of 32 deviation values. All the threads in the block first load one or three deviation values calculated for each of the pixels from the global memory into the shared memory. Then 32 of these threads calculate the running mean and running standard deviation for one pixel each using one or three deviation values. The two deviation criteria for hot pixel removal are checked and the hot pixel identification information is loaded into the global memory. Each of these stages is synchronized using __syncthreads() to ensure instruction synchronization and memory visibility

Performance tuning: Intermediate data structures are created on the GPU itself without copying back the data to the CPU. Global memory is always accessed in a coalesced fashion across all compute capabilities. Shared memory is used to reduce access to the high latency global memory as each of the deviation values are used multiple times and also because the threads need to share the loaded deviation values with each other. Warp divergence is present but is not a major factor due to the small number of hot pixels present. Shared memory is also accessed without any bank conflicts. The kernel has been developed on a Tesla C1060 processor which has 30 multiprocessors. Having a small number of threads per block ensures that all the multiprocessors are occupied at least once.

Future improvements: Use of texture memory to take advantage of spatial locality of the algorithm. Port the algorithm onto the next generation NVDIA architectures (Kepler and Fermi).

➔ putAvg <<<>>> :

Functionality: Responsible for correcting the gray scale values of hot pixels by replacing them with the average of nearby values.

Execution configuration parameters:

$$\begin{cases} (blockDim.x = (32 + 2 * (\text{ numDev} + \text{numPix} - 1))), \\ (blockDim.y = 1), \\ (gridDim.x = 1), \\ (gridDim.y = ((\text{width} - 2 * (\text{numDev} + \text{numPix} - 1)) / 32)(+1 \text{ if not an exact multiple}) \\ \text{shared memory size=(blockDim.x * sizeof(float))} \end{cases}$$

Description: Each of the individual blocks having different ($blockIdx.y$) values but the same ($blockIdx.x$) values is responsible for handling a group of 32 columns in the sinogram. The hot pixel information is first loaded from the global memory to the registers. Then one of the rows of the sinogram is loaded into the shared memory. Then if the column corresponding to the thread is identified to be a hot pixel, the pixel element in the column is replaced by the average of the gray scale values of pixels in the left and right window thereby correcting the hot pixel data. These corrected pixel values are loaded back into global

memory. This process is repeated for all the rows in the sinogram. Each of these stages is synchronized using __syncthreads() to ensure instruction synchronization and memory visibility

Performance tuning: Global memory is always accessed in a coalesced fashion across all compute capabilities. Registers are used instead of shared memory since the hot pixel identification value is used just once and also need not be shared between threads. Warp divergence is present but is not a major factor due to the small number of hot pixels present. The execution configuration parameters have been chosen so as maximize the warp occupancy on Tesla C1060 as measured using Nvidia Parallel Nsight 3.0. Since Tesla C1060 has 30 multiprocessors, having a low number of threads (32) ensures that each of the multiprocessor is assigned at least one block.

Future improvements: Use of texture memory to take advantage of spatial locality of the algorithm. Port the algorithm onto the next generation NVDIA architectures (Kepler and Fermi).

The time needed for the implementation of hot pixel removal algorithm for varying parameters is shown in the table below and the corresponding graph is also shown.

Time(milliseconds)		
1 iteration	10 iterations	20 iterations
8	81	172

Table 6.2 Hot pixel removal algorithm timings for various parameters

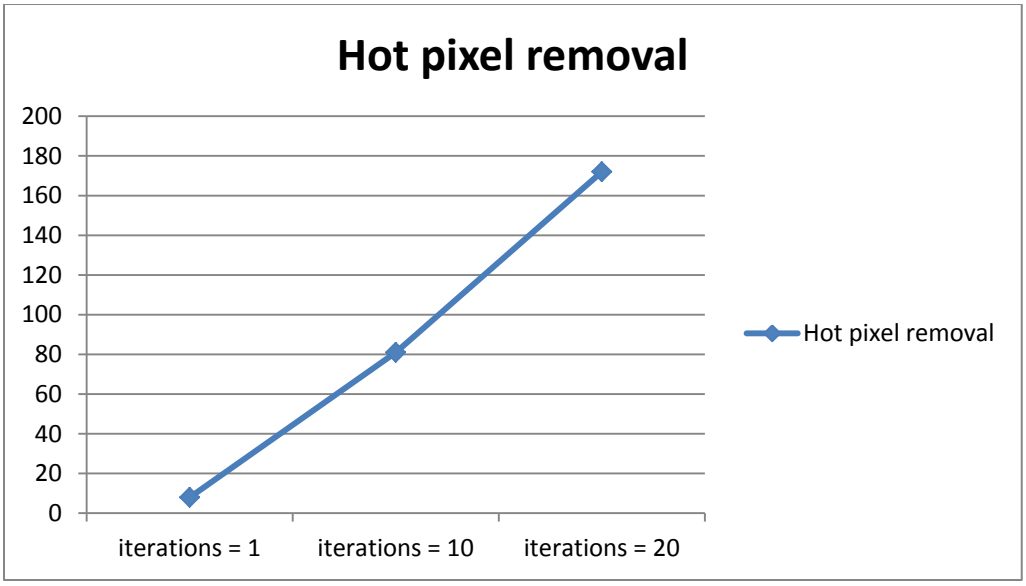


Figure 6.14 Hot pixel removal algorithm timings for various parameters

6.4 Beam hardening algorithm and its implementation on the GPU

Beam hardening algorithm is used in the post processing stage to improve the quality of the reconstructed slice and the algorithm has been integrated into the main reconstruction program. Since most of the functionality is straightforward we shall not discuss it in detail. The functions used in various stages of the beam hardening algorithm are as follows:

CT_Ring(): Responsible for setting the values of the slice outside the ring radius to a constant value

remAir(): Responsible for setting slice intensity values below a threshold (representing air) to a constant value

bifgv1<<<>>>: Responsible for doing bilateral filtering on the slice in one dimension.

transposeDiagonal<<<>>>: Responsible for doing matrix transpose using diagonal reordering on the slice in order to do bilateral filtering in the second dimension.

bifgv1<<<>>>: Responsible for doing bilateral filtering on the slice in the second dimension.

transposeDiagonal<<<>>>: Responsible for doing matrix transpose using diagonal reordering on the slice to restore data to its original form before copying it back onto the GPU.

CHAPTER 7. RECONSTRUCTION ON THE GPU

7.1 Introduction

The CT reconstruction algorithm is based on Radon transform established by J .Radon [1] in the year 1917. The logic behind the transform is that, for a region if the line integrals are known for all ray paths through the region it is possible to determine the value of the function in the region. The x-ray intensities detected by the detector represent the line integrals. The x-ray attenuation coefficient over the object represents the function over the region.

The first real time reconstruction algorithm was developed by Hounsfield and Cormack [2, 3] who received a Nobel Prize in 1979 for a reconstruction time of 2 days. Another major advancement was the Fourier Weighted Back projection developed by Ramachandran and Laxminarayan [5] which allowed us to get the cross sectional view of the object with unprecedented accuracy with a very few assumptions.

One major concern in CT reconstruction is the amount of computation involved on huge datasets and the corresponding huge amount of time taken. To speed up the entire process the algorithm has been ported from the PC onto a cluster by Zhang [9]. The next big step in speeding up the algorithm is the use of high performance graphics cards which provide much greater computational power than the cluster at a much lesser cost. A CT reconstruction algorithm running in 3 hours for an image size of 900 was first developed for a single PC at CNDE by Vivekanand Kini [6] in 1997. It was then ported onto a Linux cluster by Zhang [9] in 2003 reducing the total time taken to just 15 minutes for the same size. This thesis involves porting the algorithm onto a NVDIA graphics card reducing the time further to a minute or half a minute depending on the architecture of the card used.

7.2 Theoretical Foundation

We shall now discuss the mathematical basis of CT reconstruction in detail. Doing this enables us to identify the major stages in the reconstruction algorithm and also aids in efficiently porting the algorithm onto the GPU.

7.2.1 Radon Transform

As described before, Radon transform forms the basis of the reconstruction algorithm. Consider the figure 7.1. The object to be reconstructed is represented by the two dimensional function $f(x,y)$ in the XY plane.

Let (t) be the length of the normal from the origin to the line AA' and (θ) be the angle subtended by the normal to the positive X-axis.

Then the equation of the line AA' in normal form is given by the following equation

$$x\cos\theta + y\sin\theta = t \tag{7.1}$$

The projection $P_\theta(t)$ of the line AA' over the object $f(x,y)$ can be represented as the following line integral

$$P_\theta(t) = \int_{(\theta,t)\,line} f(x,y)ds$$

$$= \int_{-\infty}^{+\infty}\int_{-\infty}^{+\infty} f(x,y)\partial(x\cos\theta + y\sin\theta - t)dxdy \tag{7.2}$$

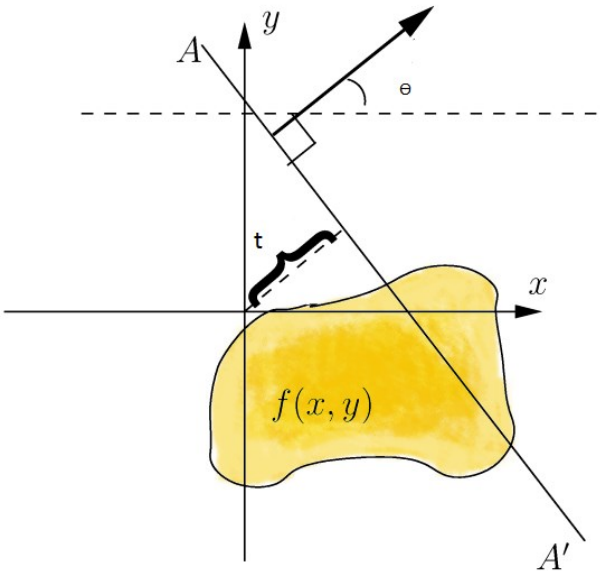


Figure 7.1 Radon Transform

$P_\theta(t)$ is called the Radon Transform of the function $f(x,y)$ over the line AA'.

7.2.2 Fourier Slice Theorem

Figure 7.2 illustrates the Fourier Slice Theorem. The Fourier Slice Theorem can be stated as follows: Take a two dimensional function $f(r)$ and project it onto a one dimensional line to get the projection value $p(x)$.Then do Fourier transform of that projection $p(x)$. We get the same result if we do the 2D Fourier transform of the function first and then slice it through the origin parallel to the projection line as shown below.

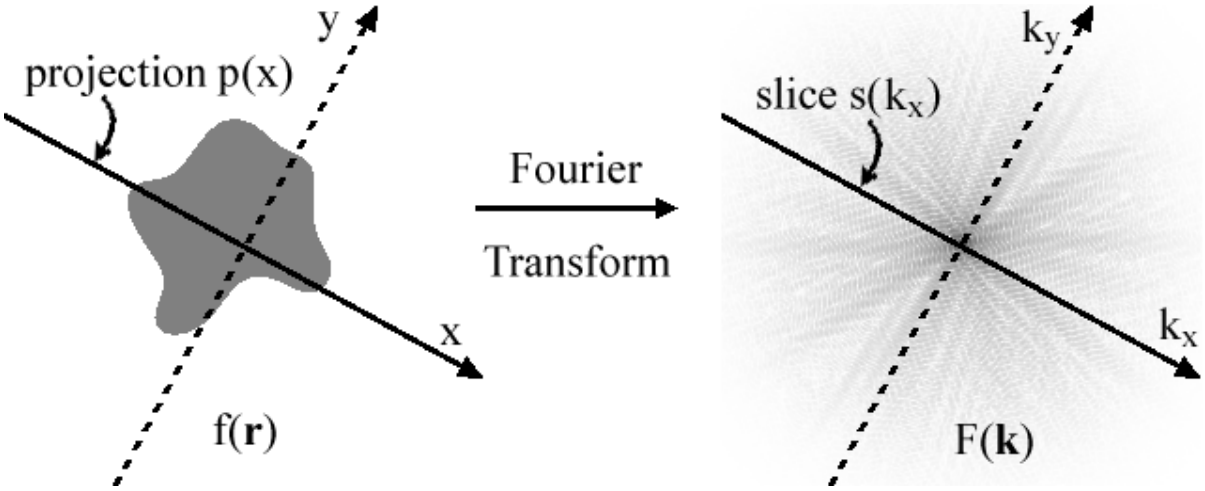


Figure 7.2 Fourier slice theorem

7.2.3 Interpolation

According to the Fourier slice theorem stated above, for an object represented by the function $f(x,y)$, the Fourier transform of each of the projections taken at an angle (θ) gives the value of Fourier transform of the object $F(u,v)$ at the same angle (θ). If infinite projections are taken for the object to be reconstructed, we will be able to determine the Fourier transforms of the object $F(u,v)$ at all points in the (u,v) plane. Then the object $f(x,y)$ can be reconstructed using the inverse Fourier transform as shown in the following equation.

$$f(x,y)=\int_{-\infty}^{\infty}\int_{-\infty}^{\infty}F(u,v)\,e^{j2\pi(ux+uy)}\,dudv \tag{7.3}$$

Instead of infinite number of projections being taken, in reality only a finite number of projections can be taken which gives estimates of $F(u,v)$ along the radial lines in the u-v plane (One of the radial lines is shown in Figure 7.3). In order to use Equation (7.3) we must interpolate from these finite number of projections to get the values of all the projections. Moreover as one gets away from the center of the u-v plane the density of points on the radial lines becomes sparser. Thus during interpolation we get a greater error in the calculation of the high frequency components of an image as compared to the low frequency components.

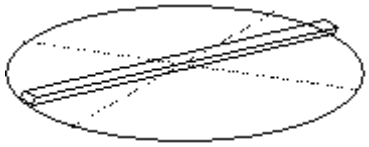


Figure 7.3 Radial estimates

7.2.4 Filtered Back projection

Filtered backprojection is a method for interpolation in the frequency domain. It is shown in the Figure 7.4. As we have seen in the previous section we only have a finite number of projections and hence a finite number of Fourier transforms. Figure 7.4 (b) shows a Fourier transform along a radial line.

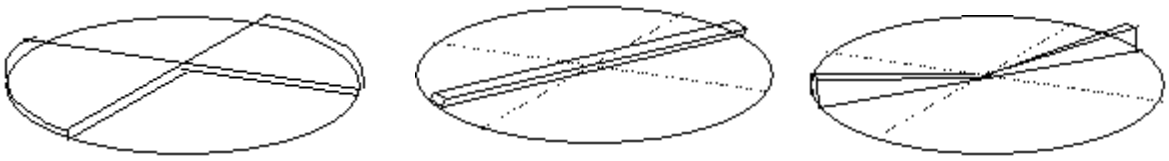


Figure 7.4 (a), (b), (c) Filtered Back projection

Using this radial line data we have to compute the values of Fourier transform in a pie slice shown in Figure 7.4 (a). For this purpose we apply the weighing shown in Figure 7.4 (c) in the frequency domain so that data shown Figure 7.4 (b) is an approximation of Figure 7.4 (a). As seen in the figure this weighing function is a high pass filter and hence the name filtered back projection.

Making further approximations from Equation (7.3), the final reconstructed picture may be obtained using the following equation

$$Let\ Q_\theta(t) = \int_{-\infty}^{\infty} P_\theta(t')h(t-t')dt'$$

Where

$$h(t) = \int_{-\infty}^{\infty} |w| e^{j2\pi wt}$$

$$f(x,y) = \pi / K * (\sum_{i=1}^{K} Q_\theta(x\cos\theta_i + y\sin\theta_i)$$

where the K angles are *those for which the projections are known*

(7.4)

7.2.5 Backprojection Kernel

The back projection kernel can be given by the following equation

$$Let\ W = the\ highest\ frequency\ component\ in\ each\ projection$$

$$a = \text{interval at which projections are sampled } (=1/2\,\text{W})$$

$$\text{h(na)} = \begin{Bmatrix} 1/4a^2, n=0 \\ 0, n\ even \\ -1/n^2\pi^2a^2, n\ odd \end{Bmatrix}$$

(7.5)

7.2.6 Lambert's law

Figure (7.5) depicts the law in a one dimensional case. The interaction of x-rays with homogenous isotropic material is given by the following equation

$I = I_0 e^{-\mu x}$

Where

I_0 *is the incident beam* intensity

I is the transmitted beam intensity

μ is the linear absorption coefficient of *the material*

x is the dis tan *ce traversed by the x – rays in the material*

(7.6)

\

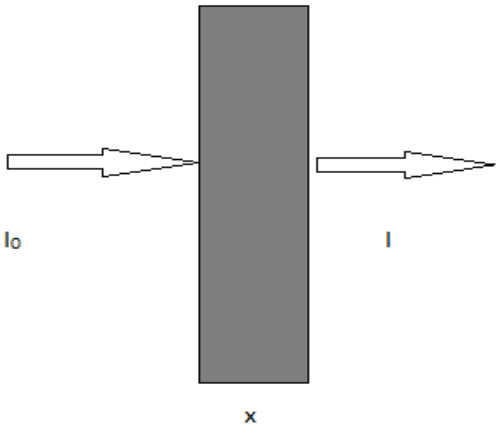


Figure 7.5 Lamberts law in 1D case

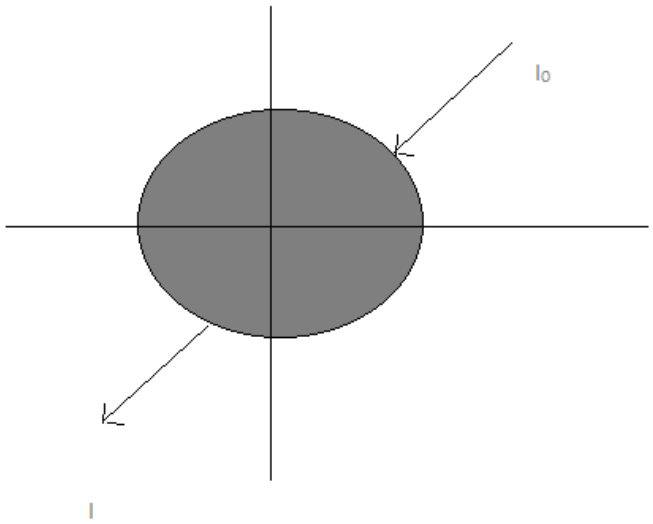


Figure 7.6 6 Lamberts law in 2D case

Figure (7.6) depicts the law in a two dimensional case. For a two dimensional case the Equation (7.6) can be rewritten as Equation (7.7) or Equation (7.8).

$$I = I_0 e^{-\int_{ray} \mu(x,y) ds} \tag{7.7}$$

$$\int_{ray} \mu(x,y) ds = \ln(I_0 / I) \tag{7.8}$$

In equation (7.8) the left hand side represents the ray integral for the projection. Therefore to calculate the ray integral we need to measure the incident intensity I_0 and transmitted intensity I.

7.2.7 Acquiring projection

As we have seen in Equation (7.8) we need to determine the incident and the transmitted intensity values to compute the ray integral for the projection. There are many different kinds of detectors which can be used to determine the intensity values of the X-ray beam like scintillation type detectors and image intensifier detectors. Most of the traditional

CT systems use scintillation type detectors which measure the transmitted intensity values by counting photons. Commonly used scintillation materials are Germanium or Sodium Iodide. The advantage of using such detectors is very high dynamic range (proportional to the time taken to acquire the sample) and sensitivity to x-ray energy which allows the object function $f(x,y)$ to be reconstructed in a narrow band of energy. The disadvantage is that the cost of the detector array and the electronics involved becomes exorbitant. Most of the medical imaging applications use this type of detectors. In image intensifier detectors, the transmitted intensity values are measured by converting the incident X-ray flux into visible light. This light is then captured by a camera and digitized. The advantage of using this system is the low cost and affordability to many industrial applications. The disadvantage is loss of energy sensitivity which results in $f(x,y)$ integrated over the x-ray spectrum. This is the type of intensifier used here at CNDE.

The image intensifier output contains a fluorescent image representative of the intensity of the image. This intensity I is converted by the camera into a video signal. We have a frame grabber that digitizes this signal with an 8-bit analog to digital convertor in real time. Using equation (7.8), the projection data to be used for reconstruction is given by the following equation

$$R_\beta(na) = \ln(G_0 / G_\beta(na))$$

Where

G_0 *is the gray scale level* intensity without the sample

$G_\beta(na)$ *is the slice of the digitized image at angle* β

a is the sampling interval

(7.9)

The slice of the digitized image at angle β namely $G_\beta(na)$ consists of gray scale values ranging from 0 to G_{max} where

$$G_{max} = 2^8 - 1$$

(7.10)

Once the X-ray flux is high enough to saturate the image intensifier the value of G_0 cannot be determined. Therefore we can assume that

$$G_0 = G_{\max} \tag{7.11}$$

The minimum possible value of $G_\beta(na)$ and the corresponding maximum value of $R_\beta(na)$ is zero and infinity respectively. This represents the possibility that the material is either infinitely thick $(x = \infty)$ or has infinite attenuation coefficient $(\mu = \infty)$. None of these cases really occur in reality. The practical reason for an infinite $R_\beta(na)$ value is that the gray scale intensity resulting from the line integral is less than minimum non zero value which is 1. To prevent this we need to clip the slice to allow for a minimum of $G_\beta(na) = 1$.

Using the minimum of $G_\beta(na) = 1$, the maximum value of projection is given by

$$R_{\beta(\max)} = \ln(G_{\max} / 1) \tag{7.12}$$

The normalized value of the projection is then given by

$$R_{\beta_i}(na) = \ln(G_{\max} / G_{\beta_i}(na)) / \ln(G_{\max}) \tag{7.13}$$

7.2.8 Determining the sampling interval

The distance between the individual sampling elements on the detector is called the sampling interval. It is a very important parameter which needs to be measure accurately for the reconstruction algorithm. We can determine the sampling interval using the following procedure. We first obtain the projection of a regular shape such as a rectangular plate by placing it on the face plate of the image intensifier. We can manually measure the plate dimension in inches and also obtain number of pixels occupied by the projection of the plate. Then the sampling interval (A) is given by,

$$A = (\text{plate dimension in inches} / \text{number of pixels occupied}) \tag{7.14}$$

To remove any random errors we repeat the process for several dimensions and average out the values.

In the case of fan beam geometry, there is a magnification factor (m) involved. The magnification factor (m) is given by

$$m = (DD / D)$$

Where

$$DD = source\ to\ sample\ dis\tan ce$$

$$D = source\ to\ \det ector\ dis\tan ce$$

(7.15)

Then the effective sampling interval (a) is given by,

$$a = (A / m)$$

(7.16)

7.2.9 Fan Beam Filtered Backprojection

All the above sections deal with reconstructing images using parallel filtered backprojection technique. In parallel projection the method for collecting data is as follows. The object to be analyzed is linearly scanned over the entire length of projection, rotated through a certain angle and then scanned again over the entire length of projection and so on. This is very costly in terms of time and only suited in the case where there is only one detector element. Thus instead of using parallel beam projection we use a different types of projection called fan beam projection. In fan beam projection we use a point source of radiation to give a fan shaped beam used for collecting projections of the object. In such a case instead of just a single detector element we can use multiple detector elements simultaneously to get projection values. At every step we rotate the object by a small angle and repeat the entire process. The multiple detector elements themselves can be arranged in a straight line or an arc to collect the projection values.

As we have seen above in parallel beam projection we use the following equations for reconstruction

The normalized value of the projection is given by the following equation

$$R_{\beta_i}(na) = \ln(G_{\max} / G_{\beta_i}(na)) / \ln(G_{\max})$$

(7.17)

The final reconstructed picture is obtained using the following equation

$$Let\ Q_\theta(t) = \int_{-\infty}^{\infty} P_\theta(t')h(t-t')dt'$$

Where

$$h(t) = \int_{-\infty}^{\infty} |w| e^{j2\pi wt}$$

$$f(x,y) = \pi / K * (\sum_{i=1}^{K} Q_\theta(x\cos\theta_i + y\sin\theta_i)$$

where the K angles are *those for which the projections are known*

(7.18)

We can obtain the corresponding equations for fan beam projection just by making some minor adjustments to the above equations.

We modify the normalized projection values as follows

$$R'_{\beta_i}(na) = R_{\beta_i}(na) \times (\mathrm{D}^2 - \tau s) / (D \times \sqrt{D^2 + s^2})$$

(7.19)

The final reconstructed picture can be obtained using the following modified equation

$$f(x,y) = 2\pi / K \times \sum_{i=1}^{K} (Q_{\beta_i}(s') / \mathrm{U}^2(x,y,\beta_i)$$

(7.20)

The equation and its terminology will be explained below.

We shall now discus all the various steps involved in Fan Beam Filtered Backprojection.

We shall use the following terminology in the below discussion

$DD = Source\ to\ \det ector\ dis\tan ce$

$D = Source\ to\ sample\ dis\tan ce$

$A = Size\ of\ individual\ \det ector\ elements$

$N = Number\ of\ elements\ in\ one\ projection$

$COR = Center\ of\ rotation$

$K = Number\ of\ projections$

$n = (0,1400)$

$m\ represents\ the\ rows\ of\ the\ slice\ data$

$\beta_i = angles\ for\ which\ the\ fan\ projections\ are\ known$

$\tau = displacement\ of\ the\ center\ of\ rotation\ from\ the\ midline$

The various steps involved are as follows

Step 1:

The first step is to obtain normalized projection data $R_{\beta_i}(na)$ from the slice data $G_{\beta_i}(na)$ using the following equation

$$R_{\beta_i}(na) = \ln(G_{\max} / G_{\beta_i}(na)) / \ln(G_{\max}) \tag{7.21}$$

Step 2:

The next step is to generate for each fan projection $R_{\beta_i}(na)$ the corresponding modified projection $R'_{\beta_i}(na)$ (correction for fan beam projection) by equation (7.22)

$$R'_{\beta_i}(na) = R_{\beta_i}(na) \times (\mathrm{D}^2 - \tau s) / (D \times \sqrt{D^2 + s^2}) \tag{7.22}$$

Step 3:

The noise present in the modified projections will be greatly amplified after convolution with the backprojection filter. To suppress the high frequency noise we need to smooth the modified projections before convolution. We use the hamming window as the smoothing filter. The projection data is convolved with the hamming window given by the following equation

$$k[n] = \begin{Bmatrix} \varsigma - \xi \cos(2\pi n / M), & 0 \le n \le M \\ 0, & otherwise \end{Bmatrix}$$

$where\ \varsigma = 0.54\ and\ \xi = 0.46$

$M = size\ of\ the\ window\ (10\ in\ the\ program)$

n is a power $of\ 2\ for\ FFT\ implementation$

(7.23)

Now that we have smoothed the modified projections we can apply the backprojection kernel without any extreme amplification of noise as shown in step 4.

Step 4:

After smoothing done in step 3, use the modified smoothed projection and convolve it with the backprojection kernel $g(na)$ to generate the corresponding filtered projection using the following equation

$$Q'_{\beta_i}(na) = R_{\beta_i}'(na) * \mathrm{g(na)}$$

Where

$$\mathrm{g(na)} = \mathrm{h(na)} / 2,$$

$$\mathrm{h(na)} = \begin{Bmatrix} 1/4a^2, & n = 0 \\ 0, & n\ is\ even \\ -1/n^2\pi^2a^2, & n\ is\ odd \end{Bmatrix},$$

and hence

$$\mathrm{g(na)} = \begin{Bmatrix} 1/8a^2, & n = 0 \\ 0, & n\ is\ even \\ -1/2n^2\pi^2a^2, & n\ is\ odd \end{Bmatrix},$$

(7.24)

Step 5:

Using the smoothed filtered projections obtained from the previous step; perform weighted back projection to get the reconstructed image using the following equation again

$$f(x,y) = 2\pi / K \times \sum_{i=1}^{K} (Q_{\beta_i}(s') / \mathrm{U}^2(x,y,\beta_i)$$

(7.25)

As we can see from the above equation we need to calculate two parameters (s') and (U) to perform weighted back projection. The calculation of those parameters is done as follows. After rotation through an angle (β) the coordinates of the detector in the rotated system (p,q) corresponding to the original coordinates (x,y) are given as follows

$$p = x\cos\beta + y\sin\beta$$
$$q = -x\sin\beta + y\cos\beta$$

(7.26)

Using the rotated coordinates, the values of the two parameters needed for back projection s' and U are computed as follows

$$s' = (\mathrm{Dp}/(\mathrm{D}-\mathrm{q})) - \tau$$
$$\mathrm{U}(\mathrm{x},\mathrm{y},\beta) = (\mathrm{D}-\mathrm{q})/\mathrm{D}$$

(7.27)

Using the parameter(s'), we can find out the detector element (n') which contributes to the reconstructed slice during filtered back projection. We calculate (n') as follows

$$n' = s' \times 1400/L$$

(7.28)

The value of (n') may not be an integer in which case we choose the nearest integer. If the value of (n') is less than 0 or greater than 1399 It makes no contribution to the weighted back projection in equation (7.25). Having known all the parameters, the contribution made by the detector elements to the reconstructed slice in one of the rotations is given by

$$(Q_{\beta_i}(\mathrm{n}')/\mathrm{U}^2(x,y,\beta_i))$$

(7.29)

7.2.10 Cone beam Filtered Back projection

Just as fan beam projection is an improvement over parallel beam projection, cone beam projection is an improvement over fan beam projection. In the case of fan beam projection, we just get the projections of one of the slices of the object at a time. We need to reassemble all the reconstructed slices one at a time to get the reconstructed image. This again is very time consuming. A further improvement is cone beam projection where an X-ray beam is used to get information about all the slices in the image along all the possible fans simultaneously.

The steps involved in cone beam reconstruction are as follows:

Step 1:

The first step is to log and scale the data as in the case of fan beam projection to obtain the normalized projection data $R_{\beta_i}(na, ma)$ from the slice data $G_{\beta_i}(na, ma)$ using the following equation

$$R_{\beta_i}(na, ma) = \ln(G_{\max} / G_{\beta_i}(na, ma)) / \ln(G_{\max}) \tag{7.30}$$

Step 2:

The next step is to generate for each fan projection $R_{\beta_i}(na, ma)$ the corresponding modified projection $R'_{\beta_i}(na, ma)$ (correction for cone beam projection) using the following equation

$$R'_{\beta}(s, z') = R'_{\beta}(s, z') \times (D^2 - \tau s) / (D\sqrt{(D^2 + s^2 + z'^2)})$$

Where

$$z' = Dz / D - q \tag{7.31}$$

Step 3:

Smooth the projection values using the Hamming window $k(na, ma)$. The hamming window used is similar to the one shown in equation (7.23).

Step 4:

Convolve the modified projection values $R_{\beta_i}^{'}(na,ma)$ with $g(na,ma)$ to generate the corresponding filtered projection.

$$Q_{\beta_i}^{'}(na,ma) = R_{\beta_i}^{'}(na,ma) * \mathrm{g(na,ma)} \tag{7.32}$$

Step 5:

Perform weighted back projection of each row of filtered projection along the corresponding tilted fan using the following equation

$$f(x,y,z) = 2\pi / K \sum_{i=1}^{K} (1/U^2(x,y,z'\beta_i)) \times Q_{\beta_i}^{'}(s',z') \tag{7.33}$$

As we can see from the above equation we need to calculate three parameters $(s^{'})$, $(z^{'})$ and (U) to perform weighted back projection. The calculation of those parameters is done as follows. Using the rotated coordinate system shown in equation (7.26)

$$\begin{aligned} U &= (D/(D-q)) \\ s' &= (Dp/(D-q)-\tau) \\ z' &= Dz/(D-q) \end{aligned} \tag{7.34}$$

Using the values $(s^{'},z^{'})$ we find out the detector element $(n^{'},\mathrm{m}^{'})$ which participates in weighted back projection. We calculate $(n^{'},\mathrm{m}^{'})$ as follows

$$\begin{aligned} n' &= s' \times 1400/L \\ \mathrm{m}' &= z' \times 450/L \end{aligned} \tag{7.35}$$

Again either $(n^{'})$ or $(m^{'})$ may not be integers in which case their value is rounded off to the nearest integer value. If the value of $(n^{'})$ or $(m^{'})$ is less than 0 or greater than 1399 It makes no contribution to the weighted back projection in equation (7.33). Having known all the parameters, the contribution made by the detector elements to the reconstructed slice in one of the rotations is given by

$$(Q_{\beta_i}(\mathrm{n}',\mathrm{m}') / \mathrm{U}^2(x, y, \mathrm{z}'\beta_i))$$

(7.36)

7.3 Implementation of the reconstruction algorithm on the GPU

We shall now discuss the implementation of the reconstruction algorithm on the GPU. The inputs of the algorithm are as follows:

- ➔ Location of the first sinogram in the dataset
- ➔ The first file in the dataset to be reconstructed
- ➔ Number of files to be reconstructed
- ➔ Location of the output reconstructed slices
- ➔ Size of the reconstructed image

The output of the algorithm is as follows:

- ➔ Reconstructed slices at the destination specified by the end user

The reconstruction algorithm implemented here at CNDE is divided into two stages for the purpose of effectively porting it onto the GPU. The first stage involves the first four steps of the cone beam filtered back projection discussed in the previous section. This includes getting the normalized projection data from the slice data (by logging and scaling the data), computing the modified projection values (adjusting the values for cone beam projection), convolution with the smoothing filter (to suppress the high frequency noise) and convolution with the back projection filter (to get the filtered projections). The second stage involves the fifth step of the cone beam filtered back projection discussed in the previous section. This includes doing the weighted back projection of each of the rows of filtered projection by computing the required parameters n', m' and U. This division into stages is done based on the amount of computation involved and also based on the nature of the computations.

The first stage involves far less computation as compared to second stage. This includes both the cumulative arithmetic computations done and also the memory loads done from the global memory. Moreover the time taken in the first stage is always constant since the size of the sinogram remains fixed. Also the nature of computations in the first stage is similar to all the preprocessing algorithms discussed in the previous sections. Scaling the

data, logging the data and modifying the projection values can be done on each of the individual pixels in an independent fashion making them embarrassingly parallel problems. The Fourier transforms to be applied as part of convolution (both smoothing and filtering) can be done on each of the individual rows of the sinogram independently. Finally the first stage itself can be done separately on each of the individual sinogram files.

The second stage has far greater computation as compared to the first stage. Far more arithmetic operations need to be performed (involving the computation of the parameters n', m' and U) and also many more memory loads from the global memory need to be done (including loading the sinogram and slice multiple times for weighted back projection). Also unlike the first stage, the time taken in the second stage increases in a nonlinear manner with the size of the image to be reconstructed. The nature of computations in the second stage is different as compared to those in the first stage. Although the computation of parameters is independent for each of the pixels in the slice, the values once calculated can be reused for all the slices on the GPU. Thus, the number of sinograms and their corresponding slices which can be simultaneously fitted on the GPU are calculated. The values computed are then reused for all these slices. Apart from this, the sinogram once loaded can be used for multiple rows in the same slice. Further much performance tuning is necessary in this stage as compared to the first stage and the necessary techniques will be discussed later in this section.

The control flow of the reconstruction algorithm along with the preprocessing algorithms is as follows. A sinogram file is initially copied from the CPU onto the GPU. It is then passed through all the algorithms responsible for preprocessing (discussed in the previous chapters) and then passed into the first stage of the reconstruction algorithm. The so processed sinogram is then stored on the GPU. This process is repeated for all the sinograms which can be simultaneously stored on the GPU. The processed batch of sinograms is then passed into the second stage of the reconstruction algorithm. Then the next batch of sinograms which can be stored simultaneously are processed in a similar manner until all the sinograms in the dataset have been processed.

We shall now discuss each of these stages in detail.

First stage:

The first stage of the reconstruction algorithm is invoked using the following method

.

```
void prepareSino(cufftHandle& plan1,cufftHandle& plan2,float *dataMatrix,cufftReal *rna,cufftComplex * rnaf,cufftReal *smtg,cufftComplex *smtgf,cufftReal *srnag,cufftComplex *srnagf,cufftReal *gnag,cufftComplex * gnagf,float *sinog,cufftComplex *qnagf,int& width,int& step,int& eWidthSino,float& sourceToDetectorD,float& sourceToSampleD,float& COR,float& CORinterval,float& xmax,float& zmax,float& a,float& toualpha,int& fileCounter,int& firstFile,int& slicePerFile,int& sinoCounter)
```

We shall now study in order the inputs, performance tuning done and the various steps of the first stage of filtered backprojection.

Inputs of the first stage:

The Fourier transform plans needed for convolution on the GPU are passed as $(\ plan1\ )$ and $(\ plan2\ )$ respectively. The input sinogram data is passed in $(\ dataMatrix[]\ )$. Also all the other pointers needed to hold the real and complex data during the reconstruction process and other parameters needed for reconstruction are also passed in the same method.

The 1D CUFFT plans $(\ plan1\ )$ and $(\ plan2\ )$ (passed as parameters to an encapsulated method $convlv()$) are created at the beginning of the algorithm with the following signatures

$$cufftPlan1d\left(\&plan1, eWidthSino, CUFFT\_R2C, step\right);$$
$$cufftPlan1d\left(\&plan2, eWidthSino, CUFFT\_C2R, step\right);$$

They are described below in detail:

$(\ plan1\ )$:

Functionality: This plan is used to do a forward Fast Fourier Transform from real to complex domain for single precision data on the GPU.

Parameters:

$cufftHandle:\ plan1$
$transform\ size:\ eWidthSino$
$transform\ type:\ CUFFT\_R2C$
$batch\ size:\ step$

Description: It takes data of size $(eWidthSino)\,(cufftReal)$ elements and does a Forward Fast Fourier transform to $(eWidthSino)\,(cufftComplex)$ complex elements by batching together $(step)$ number of transforms so as to speed up the process. It is used four times per sinogram data in the reconstruction algorithm to do an FFT twice on the sinogram data $(rna[]\,to\,rnaf[])$ and also to do FFT once each on the smoothing filter $(smtg[]\,to\,smtgf[])$ and the kernel filter $(gnag[]\,to\,gnagf[])$.

$(plan2)$:

Functionality: This plan is used to do an inverse Fast Fourier Transform from complex to real domain for single precision data on the GPU.

Parameters:

$cufftHandle$: $plan2$
$transform\ size$: $eWidthSino$
$transform\ type$: $CUFFT\_C2R$
$batch\ size$: $step$

Description: It takes data of size $(eWidthSino)\,(cufftComplex)$ elements and does an inverse Fast Fourier transform to $(eWidthSino)\,(cufftReal)$ complex elements by batching together $(step)$ number of transforms so as to speed up the process. It is used two times per sinogram data in the reconstruction algorithm to do an inverse transform from the convolved data in the complex domain to the convolved real data $(srnagf[]\,to\,srnag[])$ and $(qnagf[]\,to\,sinog[])$.

Performance tuning done for the first stage:

➔ Using the 1D CUFFT plans in a manner shown above gives us a lot of performance gain. This idea of batched transforms has been taken from the implementation of the FFTW library. Creation of plans necessary for doing Fourier transforms on the CPU or the GPU is extremely costly. The plan once created can be reused for Fourier transforms of the same size. Thus the plan is reused for all the rows in the sinogram. Apart from reusing the plan, the CUFFT library also gives an additional performance gain over FFTW by effectively using the GPU resources.

➔ The performance tuning done for each of the individual kernels is explained along with the description of kernels below.

Steps in the first stage of filtered backprojection:

➔ Copy the sinogram data from the CPU (*dataMatrix*[]) onto the GPU (*rna*[]) using cudaMemcpy(). Both the arrays are padded so as to facilitate both memory coalescing on the GPU and also to meet the size requirements for a Fast Fourier Transform.

➔ Call dumpprojection() which handles scaling, logging and modifying the projection data by calling the kernel $dum\Pr oj <<<>>>$. This kernel works as follows:

➔ $dum\Pr oj <<<>>>$:

Functionality: Responsible for scaling, logging and modifying the projections values.

Execution configuration parameters:

$$\left\{\begin{array}{l}(blockDim.x = 512),\\ (blockDim.y = 1),\\ (gridDim.x = step),\\ (gridDim.y = (eWidthSino / blockDim.x),\\ (\text{shared memory size } = 0 \text{ bytes})\end{array}\right\}$$

Description: We launch the kernel as a 2D grid of blocks with blocks having the same $(blockIdx.\mathrm{x})$ value responsible for elements in one of the rows of the sinogram. Each block is in turn composed of 512 threads with each thread responsible for doing the required functionality on one of the pixels. The elements are loaded from the global memory into the registers and scaling, logging and modifying the values is done in the registers. The so modified values are loaded back into the global memory.

Performance tuning: Global memory is always accessed in a coalesced fashion across all compute capabilities. Shared memory is not used since the different threads of the block need not cooperate with each other. Warp divergence is totally absent. The execution configuration parameters have been chosen so as maximize the warp occupancy on Tesla C1060 as measured using Nvidia Parallel Nsight 3.0.

Future improvements: Use of texture memory to take advantage of spatial locality of the algorithm. Port the algorithm onto the next generation NVDIA architectures (Kepler and Fermi).

➔ The modified projection values are then sent for convolution with the smoothing filter by calling the method convlv().

➔ Inside convlv(), cufftExecR2C() is executed on the modified projection values (*rna*[] *to rnaf*[]) using (*plan*1) .

➔ Inside convlv(), cufftExecR2C() is also executed on the smoothing filter values (*smtg*[] *to smtgf*[]) using (*plan*1) .

➔ Inside convlv(), the kernel *cv*<<<>>> is launched to multiply these transforms component by component. It is described in detail below.

➔ *cv*<<<>>>:

Functionality: Responsible for multiplying components of the complex numbers of the transforms.

Execution configuration parameters:

$$\left\{\begin{array}{l}(blockDim.x = 32),\\ (blockDim.y = 1),\\ (gridDim.x = step),\\ (gridDim.y = (eWidthSino / blockDim.x),\\ (\text{shared memory size } = 0 \text{ bytes})\end{array}\right\}$$

Description: We launch the kernel as a 2D grid of blocks with blocks having the same (*blockIdx*.x) value responsible for multiplying the components of complex numbers corresponding to one of the rows of the sinogram. Each block is in turn composed of 32 threads with each thread responsible for doing the required functionality on one of the data elements. The elements are loaded from the global memory into the registers and the functionality implemented on them (multiplication of the corresponding components of complex numbers) is shown below. The below code also shows how to handle structure memory allocated on the GPU in the form of (*cufftComplex*) complex numbers The

structure memory is accessed in a coalesced fashion. The so modified values are loaded back into the global memory.

```
float x1,y1,x2,y2;
gindex = blockIdx.x * ((eWidthSino + 2) / 2) + blockIdx.y * blockDim.x + threadIdx.x;
x1 = rnaf[gindex].x;
x2 = smtgf[gindex].x;
y1 = rnaf[gindex].y;
y2 = smtgf[gindex].y;
srnaf[gindex].x = ((x1 * x2) - (y1 * y2))/(eWidthSino);
srnaf[gindex].y = ((x1 * y2) + (x2 * y1))/(eWidthSino);
```

Performance tuning: Global memory is always accessed in a coalesced fashion across all compute capabilities. Shared memory is not used since the different threads of the block need not cooperate with each other. Warp divergence is totally absent. The execution configuration parameters have been chosen so as maximize the warp occupancy on Tesla C1060 as measured using Nvidia Parallel Nsight 3.0. The small number of threads ensures that all the multiprocessors on Tesla C1060 are occupied at least once.

Future improvements: Use of texture memory to take advantage of spatial locality of the algorithm. Port the algorithm onto the next generation NVDIA architectures (Kepler and Fermi).

➔ Inside convlv(), cufftExecC2R() is executed on the smoothed complex values (*srnagf*[] *to srnag*[]) using (*plan*2) to get the smoothed projection values.

➔ After convolution with the smoothing filter, the smoothed projection values are sent for convolution with the kernel filter by again calling the method convlv().

➔ Inside convlv(), cufftExecR2C() is executed on the smoothed projection values (*srnag*[] *to srnagf*[]) using (*plan*1).

➔ Inside convlv(), cufftExecR2C() is also executed on the kernel filter values (*gnag*[] *to gnagf*[]) using (*plan*1).

➔ Inside convlv(), the kernel $cv<<<>>>$ is launched again to multiply these transforms component by component.

➔ Inside convlv(), cufftExecC2R() is executed on the filtered complex values $(qnagf[]\,to\,\sin og[])$ using $(plan2)$ to get the filtered projection values.

➔ The entire process is repeated for a batch of sinograms which can be simultaneously stored on the GPU.

➔ These processed batch of sinograms are then passed onto the second stage of the reconstruction algorithm.

➔ The remaining batches of sinograms are processed in the same manner in the succeeding iterations.

Second stage:

The second stage of the reconstruction algorithm is invoked using the following method.

```
void computeAndInterpolate(float *crc,float *cphic,float *crg,float *cphig,float *bug,float
*bng,float& angle_step,float& sourceToSampleD,float& tou,float& toualpha,float& a,int&
width,int& step,int& eWidthSino,int& size,float *sinog,float *ig,int& numberOfSlices)
```

We shall now study in order the inputs, performance tuning done and the various steps of the second stage of filtered backprojection.

Inputs of the second stage:

The batch of sinograms to be processed all at once in the second stage is passed in the array $(\sin og[])$. Also pointers for some pre-computed data needed in the second stage of the algorithm are also passed along with some other parameters required for filtered backprojection. $(numberOfSlices)$ represents the number of slices or sinograms in a batch which can be stored simultaneously on the GPU.

Performance tuning done for the second stage:

➔ In the initial stages of algorithm development, a naïve brute force algorithm was developed without good results (The time taken was approximately equal to the time taken by the Beowulf cluster).

➔ Every time a sinogram row is back projected through the image slice, two parameters (u,n) need to be computed for each of the elements in the image slice. The contribution of the sinogram to the element under consideration in the image slice is given by $\text{sinog[n]}/\text{u}$. The computation of these parameters (u,n) involves sine, cosine and arctan arithmetic calculations. These calculations are extremely costly and they need to be calculated for the slice every time a new row is loaded. Thus we need to find a way to reduce the time involved in these calculations.

➔ Fast trigonometric functions are used without any major loss in accuracy in the calculation of these parameters.

➔ Also these parameters once computed for a slice for a particular row can be reused for all the slices in the batch placed simultaneously placed on the GPU.

➔ The computation of these parameters (u,n) involves two major components. The first component remains unchanged for all the rows of all the sinograms whereas the second component depend upon the row being back projected.

➔ The first component values are precomputed on the CPU and copied onto the GPU. They are reused for all the rows of all the sinograms and also for all the slices in a batch.

➔ The second component values are computed each time a new row is loaded from the sinogram and then reused for all the slices.

➔ Apart from the calculation of these parameters, the next most time consuming component of weighted back projection is loading the sinogram and slice data from the global memory on the GPU. We need to find a way to reuse the data once loaded.

➔ Reusing the sinogram data is far more advantageous in terms of time saved as compared to reusing the slice data. So sinogram data reuse is done for the implementation on the GPU.

➔ To reuse the sinogram data, we need to hold the parameters (u,n) in registers. Thus registers can be used to hold the (u,n) parameters for say r rows and the sinogram once loaded is reused for all the r rows.

➔ The register usage is measured using parallel Nsight 3.0.

➔ Also a thread in responsible for two of the elements separated by ($blockDim.x$) elements instead of just a single element to evenly distribute the global memory access.

➔ Moreover the number of threads in a block is reduced so that greater number of registers can be used to hold the parameters resulting in greater sinogram reuse.

➔ Also the datatype ($short$) is used instead of ($float$) to hold the parameter (n) since (n) is always a short integer.

➔ If the value of (n) is less than 0 or greater than 1399, the sinogram does not contribute to the slice data. Thus for every row we have some of the pixels in the sinogram participating in the weighted backprojection and a stretch of pixels not participating at all. This results in lots of warp divergence in the algorithm. This in turn is avoided by adding an extra element in the shared memory, putting the value (0.0) in it and using it for filtered back projection. Thus irrespective of the value of n there is always an access from the shared memory preventing warp divergence.

➔ The performance tuning done for each of the individual kernels is explained along with the description of kernels below.

Steps in the second stage of filtered backprojection:

➔ Precompute the first component of parameters needed for weighted backprojection using the function compute_crc_cphic(). These values are stored in arrays ($\mathrm{crc}[], cphic[]$). These values are unique for each of the (x, y) coordinates on the slice and need to be computed only once for all the slices.

➔ Copy the precomputed values onto the GPU arrays ($\mathrm{crg}[], cphig[]$).

➔ The parameters (u, n) to be calculated for filtered backprojection are the same for all the coordinates (x, y) of all the slices corresponding to a particular degree. So we compute these parameters (u, n) for a particular degree using the values ($\mathrm{crg}[], cphig[]$) and store them in ($bu\,\mathrm{g}[], bng[]$). This computation is done using the following function compute_u_n(). It has the following kernel which does the necessary calculations on the GPU.

➔ bun<<<>>>:

Functionality: Responsible for computing the parameters (u,n) needed for filtered backprojection for a particular degree.

Execution configuration parameters:

$$\left\{\begin{array}{l}(blockDim.x = 512),\\ (blockDim.y = 1),\\ (gridDim.x = size),\\ (gridDim.y = (size / blockDim.x)(+1\ if\ not\ an\ exact\ multiple),\\ (\text{shared memory size } = 0 \text{ bytes})\end{array}\right\}$$

Description: We launch the kernel as a 2D grid of blocks with blocks having the same $(blockIdx.\text{x})$ value responsible for elements in one of the rows of the slice. Each block is in turn composed of 512 threads with each thread responsible for doing the required functionality on one of the slice elements. Initially the values needed for the computation $(\text{crg}[], cphig[])$ are loaded from the global memory into the registers. Compute the values (u,n) corresponding to the slice element. If value of n is not an integer, round it off to the nearest integer value. If the value is greater than or equal to zero or less than width we increment it by one (to prevent warp divergence as explained previously) and put the values in the global memory array $(bng[])$. If this is not the case we just put in a value of 0 for n in the global memory. Also we square the value of U and put it in the global memory array $(bug[])$.

Performance tuning: Global memory is always accessed in a coalesced fashion across all compute capabilities. Shared memory is not used since the different threads of the block need not cooperate with each other. Warp divergence is totally absent. The execution configuration parameters have been chosen so as maximize the warp occupancy on Tesla C1060 as measured using Nvidia Parallel Nsight 3.0.

Future improvements: Port the algorithm onto the next generation NVDIA architectures (Kepler and Fermi).

➔ Now that we have computed the required parameters (u,n) for a particular degree we can calculate the contributions of one of the rows of all the sinograms for all the slices. We do this by calling the function interpolation().

➔ In interpolation we launch a kernel ip<<<>>> which does the filtered backprojection. It is described below.

➔ ip<<<>>>:

Functionality: Responsible for doing filtered backprojection using the parameters (u,n) calculated by bun<<<>>> using one of the rows representing the projection for one of the angles of all the sinogram for all the slices which can be stored on the GPU.

Execution configuration parameters: In the next page.

$$
\left\{
\begin{array}{l}
\text{if((size >= 64) \&\& (size < 128))} \\
\text{\{} \\
\text{blockDim.x = 32;} \\
\text{\}} \\
\text{else if((size >= 128) \&\& (size < 256))} \\
\text{\{} \\
\text{blockDim.x = 64;} \\
\text{\}} \\
\text{else if((size >= 256) \&\& (size < 512))} \\
\text{\{} \\
\text{blockDim.x = 128;} \\
\text{\}} \\
\text{else if((size >= 512) \&\& (size < 1024))} \\
\text{\{} \\
\text{blockDim.x = 256;} \\
\text{\}} \\
\text{else if((size >= 1024) \&\& (size < 2048))} \\
\text{\{} \\
\text{blockDim.x = 512;} \\
\text{\},} \\
\left(blockDim.y = 1\right), \\
\left(gridDim.x = size / N\right), \\
\left(gridDim.y = (size / (2 * blockDim.x))\right)\left(+1 \; if \; not \; an \; exact \; multiple\right), \\
\left(\text{shared memory size } = \text{ (eWidthBP+1)*sizeof(float) bytes}\right)
\end{array}
\right.
$$

Where

$N = number\ of\ rows\ of\ a\ slice\ handled\ by\ one\ of\ the\ blocks$

Description: All the steps in the kernel execution are as described below.

➔ We launch the kernel as a 2D grid of blocks with blocks having the same (*blockIdx.x*) value responsible for doing filtered backprojection on elements in r rows of all the slices which can be stored on the GPU.

➔ Blocks with the same $(blockIdx.x)$ value but differing $(blockIdx.y)$ values (0 or 1) are responsible for doing the filtered backprojection on either the first half columns of the r slice rows in all slices or the second half respectively.

➔ Each block is in turn composed of $(blockDim.x)$ threads.

➔ All the threads in the first block with $(blockIdx.y) = 0$ do filtered backprojection on two elements separated by $(blockDim.x)$ elements in each of the r rows of all the slices.

➔ In case of the second block with $(blockIdx.y) = 1$ two distinct cases arise. In the first case only a subset of the threads in the second block do the filtered backprojection on one of the elements in each of the N rows of all the slices. In the second case all the threads do filtered backprojection on one of the elements in each of the N rows of all the slices and only a subset of threads do filtered backprojection on the second element in each of the N rows of all the slices separated from the first element handled by the block by $(blockDim.x)$ elements.

➔ Initially all the (u,n) parameters necessary for filtered backprojection are loaded into the registers by the threads in the blocks. The parameters to be loaded are determined as mentioned above. These parameters remain constant for all the slices stored on the GPU.

➔ The thread with $(threadIx.x = 0.0)$ puts zero in the first element of the shared memory. The reasoning behind is explained in performance tuning.

➔ Now the sinogram corresponding to one of the slices is loaded by the $(blockDim.x)$ threads in multiple $(eWidthBP/(blockDim.x))$ iterations from the global memory into the shared memory.

➔ Then the filtered backprojection values are calculated using the (u,n) parameters stored in the registers and the sinogram data stored in the shared memory. Again the elements of a slice handled by a thread are determined as mentioned above.

Performance tuning: Global memory is always accessed in a coalesced fashion across all compute capabilities. Shared memory is used since the different threads of the block to cooperate with each other. Warp divergence is totally absent. The execution configuration parameters have been chosen so as maximize the warp occupancy on Tesla C1060 as measured using Nvidia Parallel Nsight 3.0.

Future improvements: Port the algorithm onto the next generation NVDIA architectures (Kepler and Fermi). Take advantage of the better resources offered by the new architectures.

➔ Now this entire process is repeated for each of the rows of all the sinograms in the batch stored on the GPU for all the slices in the batch stored on the GPU.

7.4 Results of the reconstruction algorithm

Figures 7.7 – 7.10 show the results of reconstructing all the slices of an image of size 900.

Figure 7.11-7.14 show the intermediate results during weighted back projection.

Table 7.1 shows the timings for reconstruction on a GPU as compared to a CPU and cluster.

Table 7.2 shows the timings needed for reconstructing images of different sizes.

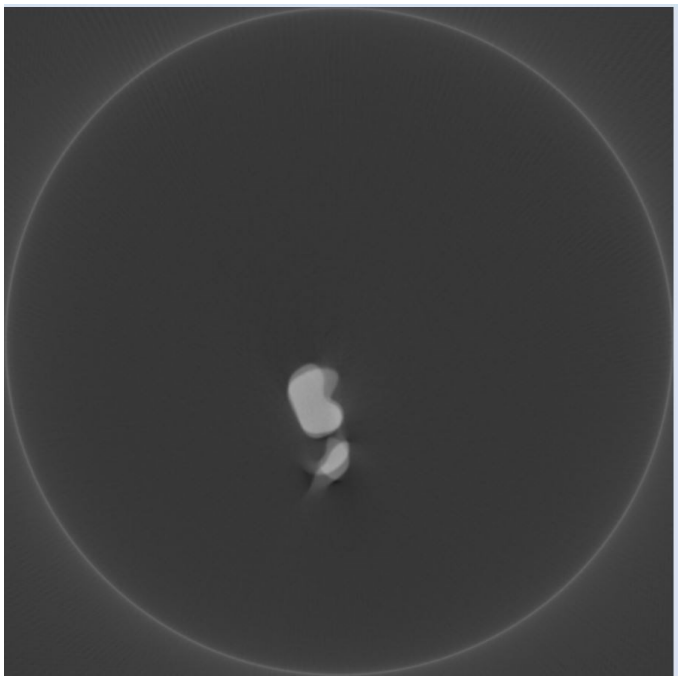


Figure 7.7 Slice 0

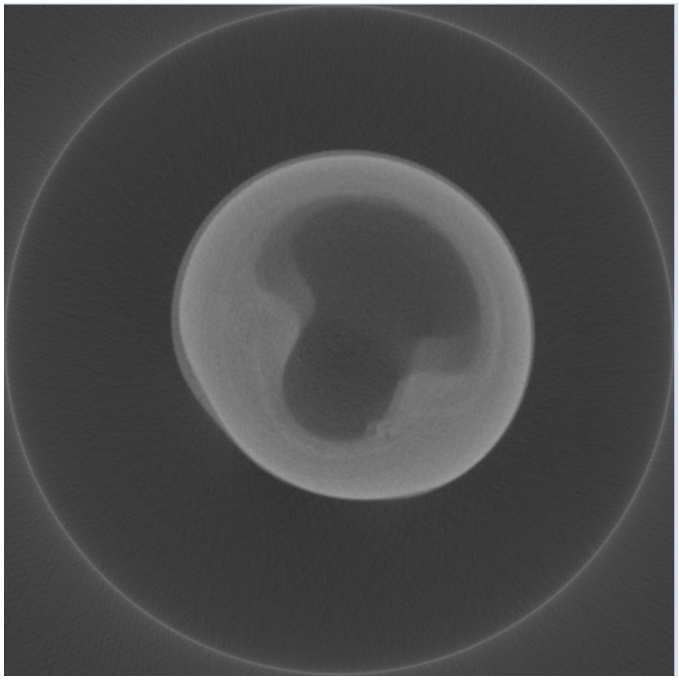


Figure 7.8 Slice 50

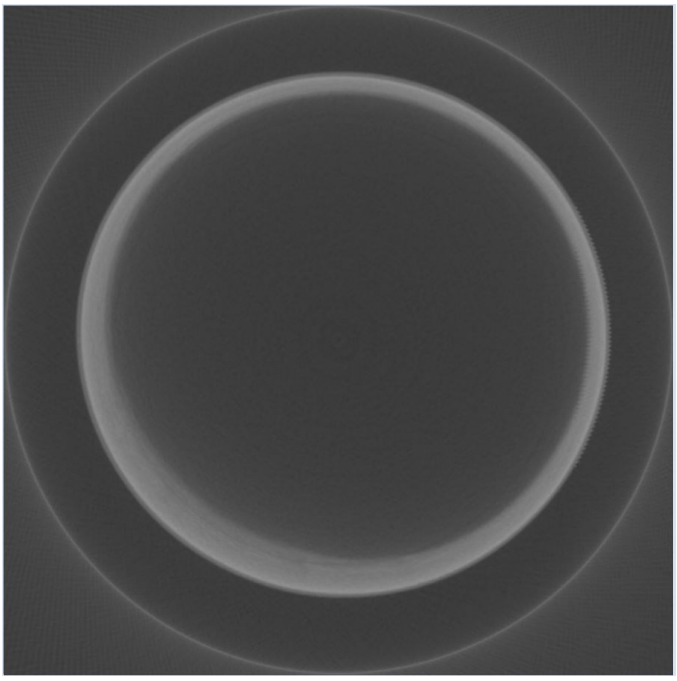


Figure 7.9 Slice 100

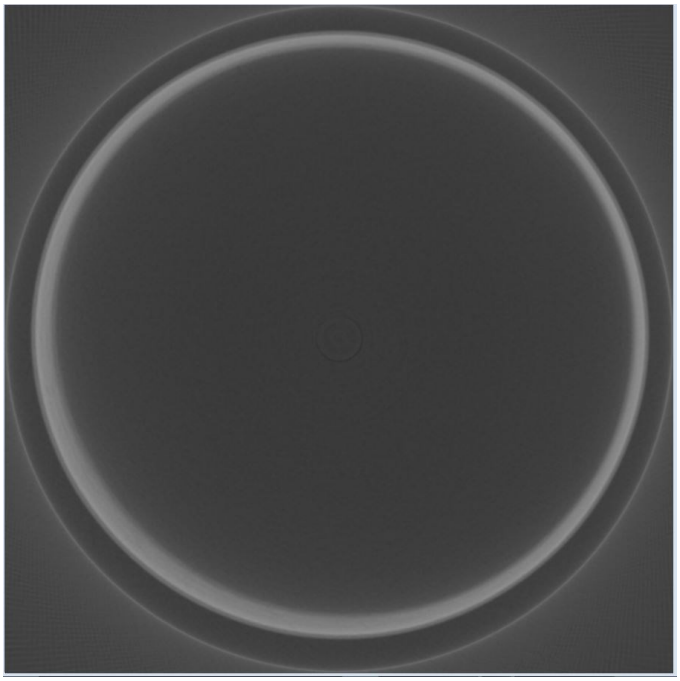


Figure 7.10 Slice 150

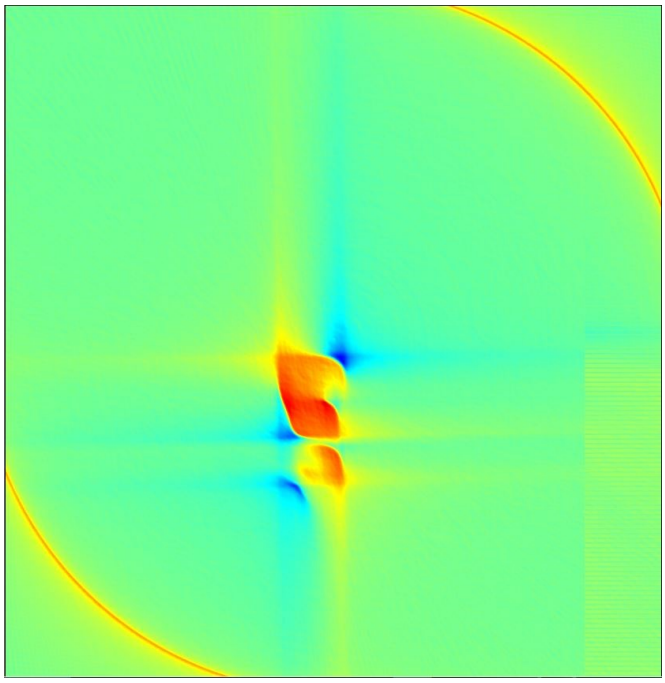


Figure 7.11 Weighted backprojection implemented for 90 rows in the sinogram

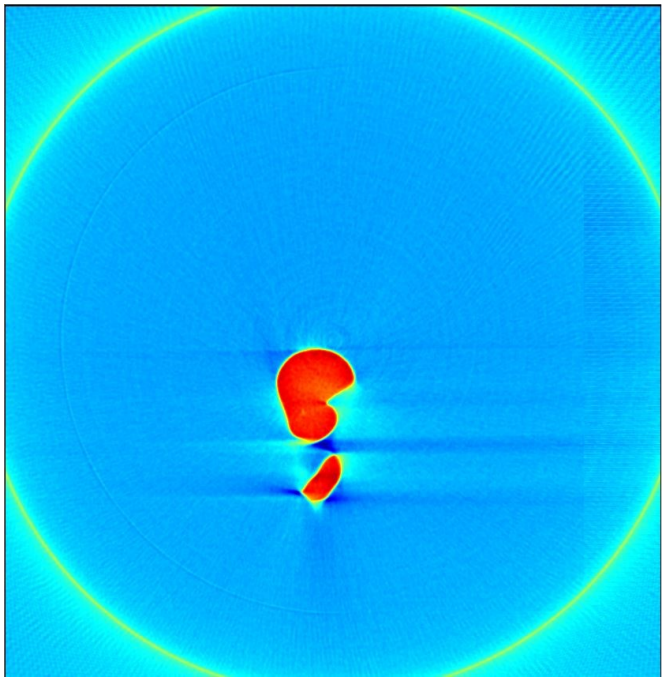


Figure 7.12 Weighted backprojection implemented for 180 rows in the sinogram

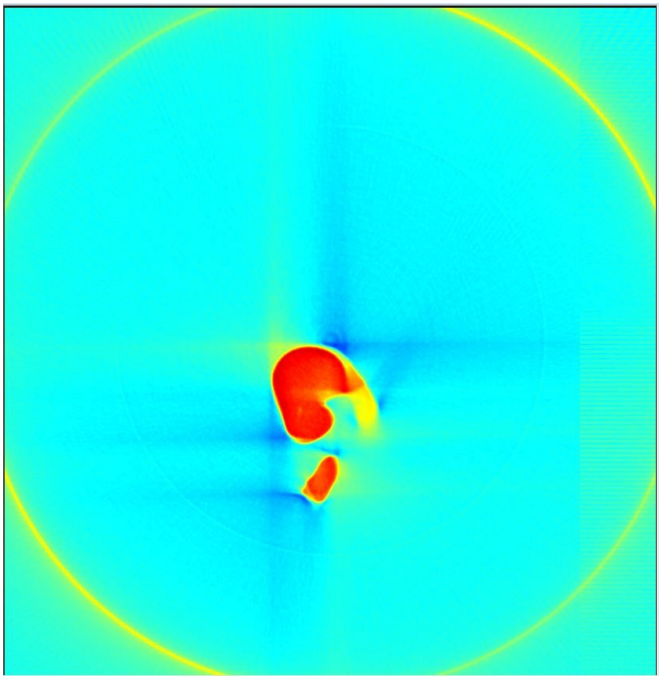


Figure 7.13 Weighted backprojection implemented for 270 rows in the sinogram

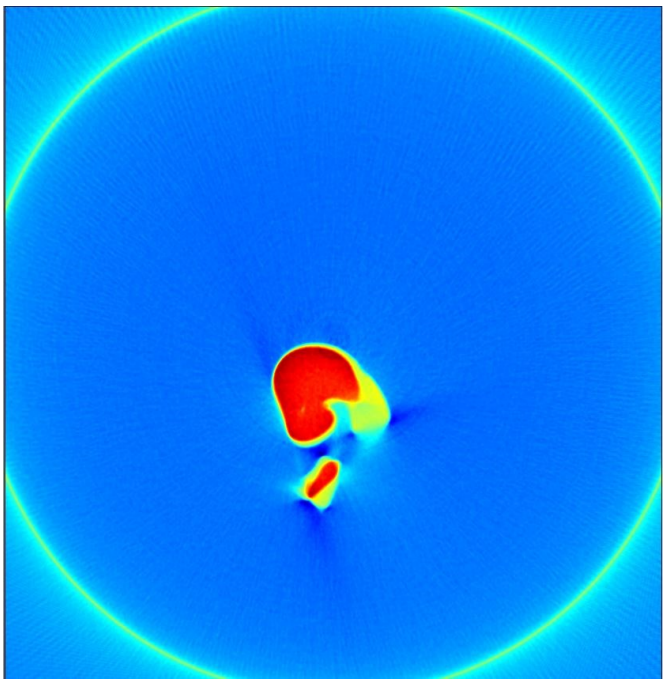


Figure 7.14 Weighted backprojection implemented for 360 rows in the sinogram

Device	Reconstruction time (on a dataset of size 1400 x 360 x 450) (Image size = 900)
CPU-Serial Execution	3 hours
CNDE Cluster	15 minutes
Tesla C1060	90 seconds
Tesla C2075	30-40 seconds (expected)

Table 7.1 Reconstruction timings as compared to CPU as cluster

Size of image	Time(seconds)
300	35
600	57
900	90

Table 7.2 Reconstruction timings for images of different sizes.

7.5 COR Detection and implementation on the GPU

A COR Detection algorithm was developed as part of research conducted by the x-ray group at CNDE. Since it is an integral part of the reconstruction algorithm running on the GPU we shall briefly discuss the algorithm and then discuss it's porting onto the GPU. The COR of a row on the detector is determined by taking the axis of rotation of the object and projecting it along the fan beam corresponding to the row onto the detector(as shown in Figure 7.7). If the axis of rotation of the object is perfect then the COR of all the rows of the detector remain the same. If the object is slightly tilted along its axis of rotation then the COR of all the rows will be different from each other. For proper reconstruction of the object the COR must be determined in an accurate fashion. Greater the resolution of the object greater is the accuracy to which the COR must be determined. Not having a proper COR significantly affects the quality of the image. Small errors in COR detection create a blurred image. Large errors in COR detection create halos around the image which can affect the quality of the image significantly and can obscure the features of the image being analyzed.

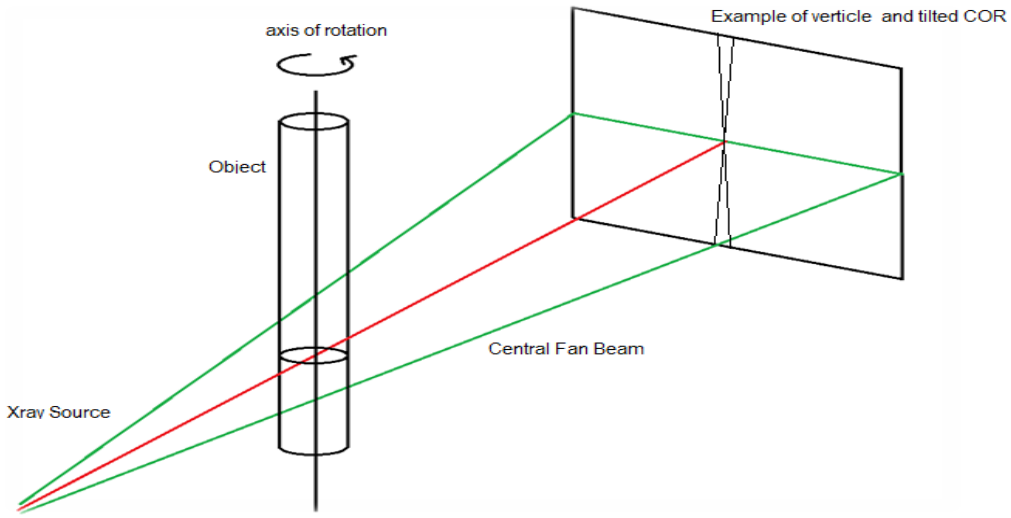


Figure 7.15 COR of an image

We shall now discuss the implementation of the COR Detection algorithm on the GPU. The inputs of the algorithm are as follows:

➔ Location of the first sinogram in the dataset

➔ Option to determine the COR(1 = manual change of COR), (2 = Detecting the COR using CORDetect algorithm), (3 = Using the firstCOR in the input sinogram)

➔ Manual COR entered by the end user (option = 1)

➔ Size of image to be reconstructed by the CORDetect algorithm(option = 2)

➔ First COR to be tested for by the CORDetect algorithm (option = 2)

➔ Last COR to be tested for by the CORDetect algorithm (option = 2)

➔ Do trend removal(Y/N)(option = 2)

➔ Filter size for trend removal(trend removal = 'Y',option = 2)

The output of the algorithm is as follows:

➔ COR of the object

The algorithm is launched by calling the function CORDetect() which does the following functionality

➔ Set the required parameters for COR Detection using setCorDetectParameters()

➔ Do the reconstruction of the first sinogram to give an image of size entered by the end user and by using the first COR entered by the end user.

➔ Copy the image from the GPU onto the CPU.

➔ If the first COR is being used compute the vector needed to determine the rows to be averaged using the function computeVectorCORDetect()

➔ Copy a small number of rows of the image and store then in an array data[].

➔ Repeat the above steps for all the COR's in the range entered by the end user and store all the required rows or columns in the array data[].

➔ Copy the data[] back on to the GPU

➔ Compute the mean frequency for all the trial images using computeFFTCD().

➔ Filter the mean frequencies using the filter size entered by the end user and subtract them from the original frequencies to get trend using filterFFTCD().

➔ The maximum of the corresponding vector gives the COR and is determined using findMaxCD().

The COR Detection algorithm is called after taking the reconstruction info from the end user and before the actual reconstruction is done. The algorithm prompts the end user as to whether he wants to manually change the COR, use the COR Detect algorithm to change the COR or to use the first COR in the sinogram. If the COR Detection is run then the end user is again prompted as to what COR he wants to use based on the accuracy of COR predicted by the COR Detection algorithm.

The results of using a proper COR and an improper COR are shown below.

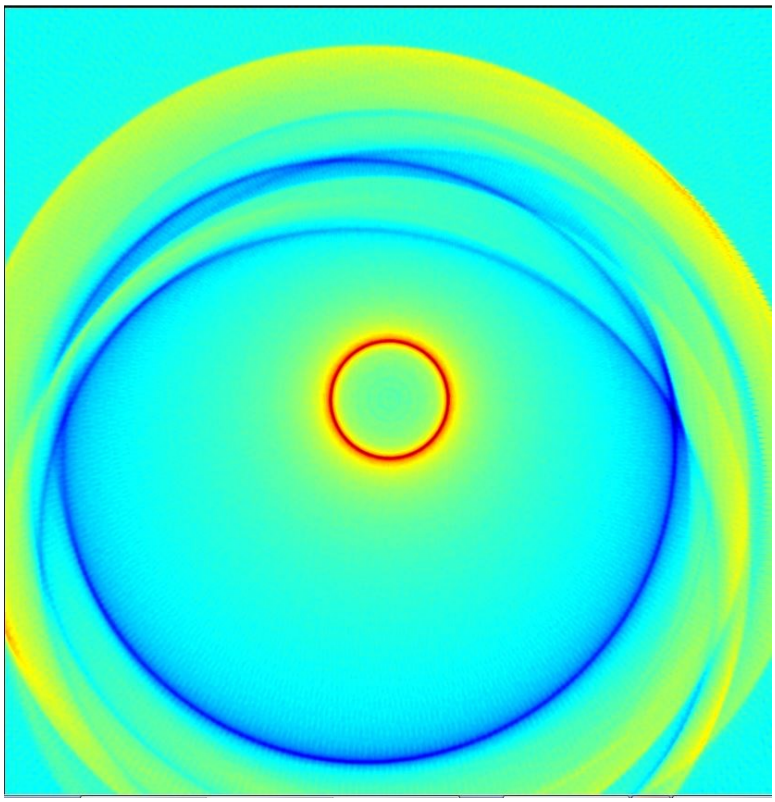


Figure 7.16 Incorrect COR used

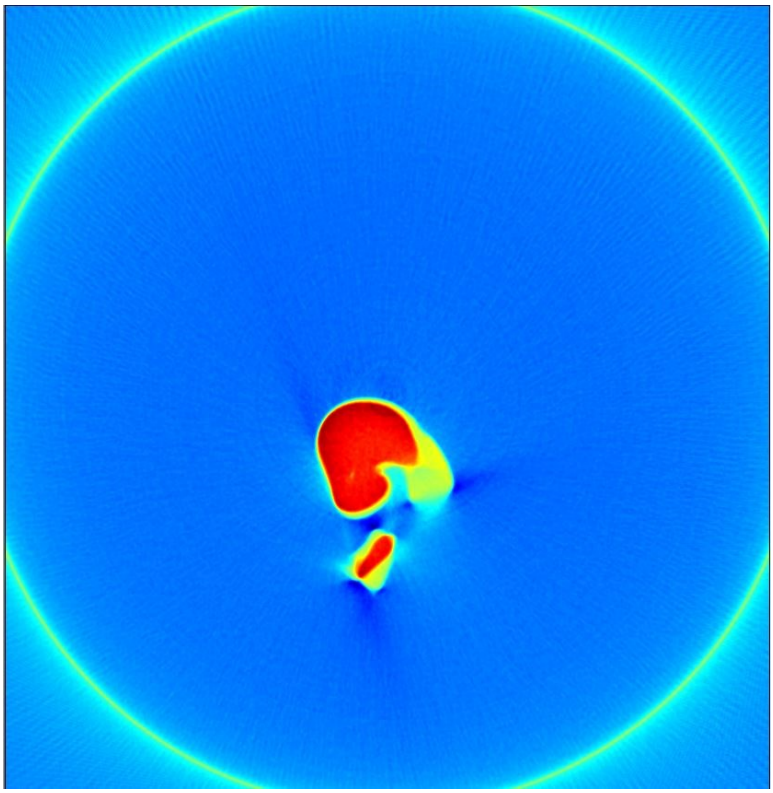


Figure 7.17 Correct COR used.

7.6 The implementation flow of the entire reconstruction algorithm

We shall now see how the entire algorithm (including the basic filters, preprocessing algorithms, COR Detection algorithm, reconstruction algorithm and post processing algorithm) is integrated into a single program implemented on the GPU. The major steps involved in reconstruction algorithm are as follows

➔ First the entire reconstruction information is obtained from the end user in the function getSinoReconstructionInfo(). It includes the preprocessing information for basic image processing algorithms(like radii of the filters, number of iterations needed etc), information for advanced image processing algorithms(ring artifact removal, beam drift correction and hot pixel removal), information for the reconstruction algorithm(like the size of the image to be reconstructed) and also information for post processing algorithms(like beam hardening). The other information needed is also taken from the end user including the location of the first sinogram in the dataset, first sinogram in the dataset to be processed,

number of sinograms to be processed, location of the processed sinograms and location of the output vol files. Validation is done for all the entries made by the end user using validateStringEntry(), validateFloatEntry() and validateIntEntry() for string entries, floating point entries and integer entries respectively.

➔ Open the first sinogram file in the dataset specified by the end user to read in the parameters of the header and tail of the sinogram file.

➔ Check the format of input sinogram file using checkBinaryAscii(). The fomat affects the manner and time in which the file is processed. Reading in a binary file is far faster as compared to reading in an ASCII file. If needed the end user can convert the format of all the sinograms in the dataset using the file format conversion program.

➔ Read in the sinogram header using readAndCheckSinoHeader(). The function also checks the format of the header of the input sinogram file as to whether all the required fields are present in the proper format or not.

➔ Display the header using displaySinoHeader() so that the end user can get an idea of the parameters in the header.

➔ Calculate the paddings needed by the sinogram file using calculateSinoPadding() to facilitate memory coalescing across all compute capabilities(1.0 – 1.3). This includes padding in four dimensions (left padding, right padding, upper padding, lower padding).

➔ Display the paddings used to the end user using displaySinoPadding().

➔ Create an array (dataMatrix[])on the CPU to hold a single sinogram file.

➔ Read in the sinogram tail using readAndCheckSinoTail (). The function also checks the format of the tail of the input sinogram file as to whether all the required fields are present in the proper format or not.

➔ Display the tail using displaySinoTail () so that the end user can get an idea of the parameters in the tail.

➔ Close the first sinogram file in the dataset opened.

➔ If preprocessing needs to be done by the end user create the required data structures on the GPU and the CPU to hold the sinogram file and also all the intermediate data and information about the data. The data structures created depend on the preprocessing algorithm being used.

➔ Set the elements of all the data structures created on the GPU for preprocessing to zero using cudaMemset() so as to avoid any garbage values from being considered during calculations.

➔ Set the elements of all the data structures created on the CPU for preprocessing to zero using arraySet () so as to avoid any garbage values from being considered during calculations.

➔ Synchronize all the operations done until now using cudaThreadSynchronize().

➔ If reconstruction needs to be done set all the parameters needed for reconstruction in setReconParameters() including the COR for reconstruction in case of manual COR change, the parameters for COR Detection in case of CORDetect algorithm, the values of parameters like tou, toualpha etc.

➔ If COR Detect algorithm needs to be run create the necessary data structures on the CPU and the GPU.

➔ Create all the necessary data structures for reconstruction on the CPU and the GPU (including memory allocation for kernel filter and smoothing filter, memory allocation for the precomputed values and so on)

➔ Create the necessary data structures for storing as many sinogram files as can be stored on the GPU.

➔ Create the necessary data structures for storing as many reconstructed slices as can be stored on the GPU.

➔ In the algorithm implemented the entire sinogram data and slice data is stored at once on the GPU since Tesla C1060 has 6GB of global memory which is enough to store 450 sinogram files and also their reconstructed slices.

➔ At each of the above memory allocation steps do error checking for enough memory allocation and also display the memory usage on the GPU to the end user using cudamemoryinfo().

➔ Set the elements of all the data structures created on the GPU for reconstruction to zero using cudaMemset() so as to avoid any garbage values from being considered during calculations.

➔ Set the elements of all the data structures created on the CPU for reconstruction to zero using arraySet () so as to avoid any garbage values from being considered during calculations.

➔ Create the 1D plan for the real to complex Fourier transforms to be done on the GPU.

➔ Create the 1D plan for the complex to real Fourier transforms to be done on the GPU.

➔ Form the kernel filter on the CPU using formKernelFilter ().

➔ Form the smoothing filter on the CPU using formHammingFilter().

➔ Copy the kernel filter from the CPU on to the GPU using cudaMemcpy().

➔ Copy the smoothing filter from the CPU on to the GPU using cudaMemcpy().

➔ If COR Detection is desired by the end user run the COR Detection algorithm using CORDetect().

➔ Change the COR for reconstruction to the manual COR or the COR detected by the COR Detect algorithm using changeCOR().

➔ Adjust the reconstruction parameters set using setReconParameters() to reflect the changes made to the COR using adjustCORReconParameters().

➔ Display the reconstruction parameters to be used for reconstruction to the end user using displayReconParameters().

➔ Start reading in each of the sinogram files to be processed by reading in the individual sinograms.

➔ Check the sinogram file header using readAndCheckSinoHeader().

➔ Read in the sinogram file and scale it as well as pad it for memory coalescing using getSinoMatScaledPadded().

➔ Apply the box filter algorithm on the sinogram for the number of iterations specified by the end user using meanFilter().

➔ If the end user wants to write out the sinogram data at this stage write out the data by removing the padding using putSinoMatWPad(). This is done for every stage of preprocessing as needed.

➔ Apply the median filter algorithm on the sinogram for the number of iterations specified by the end user using medianFilter ().

➔ Apply the gaussian filter algorithm on the sinogram for the number of iterations specified by the end user using gaussianFilter ().

➔ Apply the bilateral filter algorithm on the sinogram for the number of iterations specified by the end user using bilateralFilter ().

➔ Apply the ring artifact removal algorithm on the sinogram for the number of iterations specified by the end user using ringArtifactRem ().

➔ Apply the beam drift correction algorithm on the sinogram for the number of iterations specified by the end user using beamDriftCorr ().

➔ Apply the hot pixel correction algorithm on the sinogram for the number of iterations specified by the end user using hotPixelCorr ().

➔ If preprocessing is not done, read in the sinogram file and scale it as well as pad it for memory coalescing using getSinoMatScaledPadded().

➔ If preprocessing is done, scale the sinogram using arrayScalePad().

➔ Perform the first stage of reconstruction using prepareSino().

➔ Store the sinogram as a batch on the GPU.

➔ Read in the sinogram header using readAndCheckSinoTail(). The function also checks the format of the tail of the input sinogram file as to whether all the required fields are present in the proper format or not.

➔ Display the tail using displaySinoHeader() so that the end user can get an idea of the parameters in the header.

➔ Write out the tail using writeSinoTail().

➔ Do the second stage of reconstruction using computeAndInterpolate().

➔ Write out the header of the vol file using writeVolHeader().

➔ Write out the vol file using writeVolFile().

➔ Write out the tail of the vol file using writeVolTail().

➔ If necessary, do beam hardening using beamHardening().

➔ Destroy all the created data structures on the CPU and GPU using delete[] and cudaFree().

CHAPTER 8. Conclusions and Future work

8.1 Conclusions

➔ The bilateral filtering implemented as part of this thesis is a very valuable addition to the image processing algorithms at CNDE.

➔ The new preprocessing algorithms implemented as part of research (including ring artifact removal and hot pixel removal) function very efficiently as compared to the previous algorithms.

➔ The implementation of the reconstruction algorithm on the GPU is a very valuable addition to the research done by the X-ray group at CNDE since it gives almost 10 times speedup over the existing cluster at 10 times lesser cost.

➔ The entire reconstruction algorithm has been rewritten with proper commenting as part of research.

➔ The existing code is very robust with validation done for all the inputs entered by the end user.

➔ Moreover, the reconstruction algorithm has been divided into modules depending on the functionality so as to facilitate easy addition of new algorithms to the existing modules.

➔ Many other additional modules like COR Detection module and post processing module have also been integrated with the reconstruction algorithm.

➔ Much flexibility has been added to the existing algorithm like doing multiple iterations of a particular algorithm, doing the algorithms in any order wanted and also writing out the required processed sinogram files at any stage.

8.2 Future work

➔ Research more efficient usage of the existing architecture (Usage of constant and texture memory).

➔ Port the algorithms onto the next generation NVDIA architectures (Kepler and Fermi).

➔ Research more efficient methods for porting the existing algorithms (like median filtering) onto the GPU

- ➔ Implement more efficient preprocessing and reconstruction algorithms
- ➔ Port some more existing algorithms at CNDE (erosion dilation, XRSIM, CTSIM etc.) on to the GPU.
- ➔ Research the use of GPU clusters.

BIBLIOGRAPHY

[1] J.Radon, "*On the Determination of Functions from Their Integral Values along Certain Manifolds*", IEEE transactions on medical imaging Vol. MI-5 NO. 4, December 1986.

[2] A.M.Cormack, "*Representation of a Function by Its Line Integrals, with Some Radiological Applications*", Journal of applied physics Vol. 34 Number 9, September 1963.

[3] G.N.Hounsfield, "*Computed transverse axial tomography, Description of the system*", British Journal of radiology Vol. 46, July 1973.

[4] H.Barett, W.Swindell, "*Radiological Imaging: The Theory of Image Formation, Detection, and Processing*", Academic Press First edition, October 1996.

[5] G.N.Ramachandran, A.V.Laxminarayan, "*Three-dimensional Reconstruction from Radiographs and Electron Micrographs: Application of Convolutions instead of Fourier Transforms*", Proceedings of the National Academy of Sciences of the United States of America Vol. 68, September 1971.

[6] V.Kini*, "Tomographic Inspection System Using X-rays",* M.S Thesis Iowa State University, 1994.

[7] H.Lu, "*Implementation of integrated Computer Control for X-Ray material Characterization Systems*", M.S Thesis Iowa State University, 2000.

[8] P.Fan, "*High Resolution CT Data Acquisition software and 3D Data Visualization tool*", M.S Thesis Iowa State University, 2001.

[9] J.Zhang, "*Development of a High Resolution 3D Computed Tomography System: Data Acquisition, Reconstruction and Visualization*", M.S Thesis Iowa State University, 2003.

[10] N.Sheikh, "*Medium resolution Computed Tomography through phosphor screen detector and 3D image analysis*", M.S Thesis Iowa State University, 2006.

[11] B.Kirk, W.Hwu, "*Programming Massively Parallel Processors: A Hands-on Approach*", Morgan Kaufmann First edition, February 2010.

[12] NVIDIA Corporation, "*NVIDIA CUDA Getting Started Guide For Microsoft Windows*", Version 5.5, July 2013.

[13] NVIDIA Corporation, "*CUDA C Programming guide*", Version 5.5, July 2013.

[14] NVIDIA Corporation, "*CUDA C Best Practises guide*", Version 5.5, July 2013.

[15] NVIDIA Corporation, "*CUDA Runtime API*", Version 5.5, July 2013.

[16] NVIDIA Corporation, "*CUDA Math API*", Version 5.5, July 2013.

[17] NVIDIA Corporation, "*CUDA CUFFT API*", Version 5.5, July 2013.

[18] NVIDIA Corporation, "*CUDA Samples*", Version 5.5, July 2013.

[19] NVIDIA Corporation, "*CUDA Compiler Driver NVCC*", Version 5.5, July 2013.

[20] NVIDIA Corporation, "*CUDA-MEMCHECK*", Version 5.5, July 2013.

[21] NVIDIA Corporation, "*Nsight Visual Studio Edition",* Version 3.2, July 2013.

[22] NVIDIA Corporation, "*Profiler User's Guide*", Version 5.5, July 2013.

[23] NVIDIA Corporation, "*CUDA GPU Occupancy Calculator*", NVIDIA Corporation.

[24] C.Gonzalez, E.Woods, "*Digital Image Processing*", Prentice Hall Third edition, August 2007.

[25] NVIDIA Corporation , V.Podlozhnyuk, "*Image Convolution with CUDA*", NVIDIA CORPORATION, June 2007.

[26] C.Tomasi, R.Manduchi, "*Bilateral Filtering for Gray and Color Images*", Proceedings of the IEEE International Conference on Computer Vision, January 1998.

[27] Z. Zheng, W.Xu and K.Mueller, "*Performance Tuning for CUDA-Accelerated Neighborhood Denoising Filters*", Workshop on High Performance Image Reconstruction, July 2011

[28] L.Staal, "*Bilateral Filtering with CUDA*", Course Project University of Aarhus, December 2011.

[29] E.Bethel, "*Exploration of Optimization Options for Increasing Performance of a GPU Implementation of a Three-Dimensional Bilateral Filter*", January 2012.

[30] W.Hwu, "*GPU Computing Gems Emerald Edition*", February 2011.

[31] G.Perrot, S.Domas, R.Couturier, "*Fine-tuned high-speed implementation of a GPU-based median filter*", Journal of Signal Processing Systems, July 2013.

[32] W.Chen, M.Beister, Y.Kyriakou, M.Kachelrie, "*High Performance Median Filtering using Commodity Graphics Hardware*", Nuclear Science Symposium Conference Record, November 2009.

[33] R.Aras, Y.Shen, "*GPU Accelerated Vector Median Filter*", Modsim World Conference, November 2010.

[34] E.Nielson, "*Real Time Wavelet Filtering on the GPU*", M.S Thesis Norwegian University of Science and Technology, May 2007.

[35] M.A.Yousuf, M.Asaduzzaman, "*An Efficient Ring Artifact Reduction Method Based on Projection Data for Micro-CT Images*", Journal of Scientific Research, November 2009.

[36] B.Munch, P.Trtik, F.Marone, M.Stampanoni, "*Stripe and ring artifact removal with combined wavelet — Fourier filtering*", Optics Express Vol. 17, December 2009.

[37] Y.Kyriakou, D.Prell, W.Kalendar, "*Ring artifact correction for high-resolution micro CT*.", Physics in Medicine and Biology, September 2009.

[38] E.Anas, Y.Lee, K.Hasan, "*Classification of ring artifacts for their effective removal using type adaptive correction schemes",* Computers in Biology and Medicine Vol. 41, March 2011.

[39] W.Chen, D.Prell, Y.Kyriakou, W.Kalendar, "*Accelerating ring artifact correction for flat detector CT using the CUDA framework*", Physics of Medical Imaging, March 2010.

[40] S.Hill, B.Landsman, "*Cosmic Ray and Hot Pixel Removal from STIS CCD Images*", HST Calibration Workshop, January 1997.

[41] G.Robert, "*Software for detection and correction of defective pixels from a digital sensor array using statistical methods of data analysis*", Project Petru Maior University, February 2011.

[42] T.Hermann, "*Correction for Beam Hardening in Computed Tomography*", Physics in Medicine and Biology, December 1977.

[43] NVIDIA CORPORATION, G.Ruetsch, P.Micikevicus, "*Optimizing matrix transpose in CUDA*", NVIDIA CORPORATION, Januray 2009.

[44] H.Turbell, "*Cone Beam Reconstruction Using Filtered Backprojection*", Ph.d Thesis Linkoping University, February 2001.

[45] G.Wang, H.Lin, C.Cheng, M.Shinokazi, "*A General Cone-beam Reconstruction Algorithm*", IEEE Transactions on Medical Imaging Vol. 12, September 1993.

[46] K.Mueller, R.Yagel, J.Wheeler, "*Fast Implementations of Algebraic Methods for 3D Reconstruction from Cone-Beam Data*", IEEE Transactions on Medical Imaging, September 1998.

[47] G.Handy, "*3D CT Cone Beam Reconstruction Via Fan to Parallel Rebinning*", Project University Of Michigan, December 2012.

[48] X.Li, J.Ni, G.Wang, "*Parallel iterative cone beam CT image reconstruction on a PC cluster*", Journal of X-ray science and technology, August 2004.

[49] S.Mukherjee, N.Moore, J.Brock, M.Leeser, "*CUDA and OpenCL Implementations of 3D CT Reconstruction for Biomedical Imaging*", Proceedings of IEEE High Performance Extreme Computing, September 2012.

[50] H.Scherl, B.Keck, M.Kowarschik, J.Hornegger, "*Fast GPU-Based CT Reconstruction using the Common Unified Device Architecture (CUDA)*", Nuclear Science Symposium Conference Record Vol. 6, October 2007.

[51] D.Maier, "*GPUCT: A GPU Accelerated CT Reconstruction System*", B.S Thesis University of Virginia, March 2007.

[52] Y.Okitsu, F.Ino, K.Hagihara, "*High-performance cone beam reconstruction using CUDA compatible GPUs",* Parallel Computing Vol. 36, March 2010.

[53] K.Schwartz, "*The Texture Unit as Performance Booster in CT-Reconstruction*", Siemens Healthcare, September 2011.

[54] T.Donath, F.Beckmann, A.Schreyer, "*Automated determination of the center of rotation in tomography data*", Journal of Optical Society, May 2006.

[55] C.Raven, "*Numerical removal of ring artifacts in microtomography*", Review of Scientific Instruments Vol. 69, August 1998.

[56] K.Hasan, F.Sadi, Y.Lee, "*Removal of ring artifacts in micro-CT imaging using iterative morphological filters*", Signal, Image and Video Processing Vol. 6, March 2012.

[57] E.Anas, Y.Lee, K.Hasan, "*Removal of ring artifacts in CT imaging through detection and correction of stripes in the sinogram*", Physics Medicine Biology Vol.55, March 2010.