PROTON COMPUTED TOMOGRAPHY: MATRIX DATA GENERATION THROUGH GENERAL PURPOSE GRAPHICS PROCESSING UNIT RECONSTRUCTION

A Thesis

Presented to the

Faculty of

California State University,

San Bernardino

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

in

Computer Science

---

by

Micah Harris Witt

March 2014

PROTON COMPUTED TOMOGRAPHY: MATRIX DATA GENERATION THROUGH GENERAL PURPOSE GRAPHICS PROCESSING UNIT RECONSTRUCTION

A Thesis

Presented to the

Faculty of

California State University,

San Bernardino

---

by

Micah Harris Witt

March 2014

Approved by:

Ernesto Gomez, Advisor, School of Computer Science and Engineering	Date
Reinhard Schulte	
Keith Schubert	

ABSTRACT

Proton computed tomography (pCT) is an image modality that will improve treatment planning for patients receiving proton radiation therapy compared with the current techniques, which are based on X-ray CT. Images are reconstructed in pCT by solving a large and sparse system of linear equations. The size of the system necessitates matrix-partitioning and parallel reconstruction algorithms to be implemented across some sort of cluster computing architecture. The prototypical algorithm to solve the pCT system is the algebraic reconstruction technique (ART) that has been modified into parallel versions called block-iterative-projection (BIP) methods and string-averaging-projection (SAP) methods. General purpose graphics processing units (GPGPUs) have hundreds of stream processors for massively parallel calculations. A GPGPU cluster is a set of nodes, with each node containing a set of GPGPUs. This thesis describes a proton simulator that was developed to generate realistic pCT data sets. Simulated data sets were used to compare the performance of a BIP implementation against a SAP implementation on a single GPGPU with the data stored in a sparse matrix structure called the compressed sparse row (CSR) format. Both BIP and SAP algorithms allow for parallel computation by creating row partitions of the pCT linear system. The difference between these two general classes of algorithms is that BIP permits parallel computations within the row partitions yet sequential computations between the row partitions, whereas SAP permits parallel computations between the row partitions yet sequential computations within the row partitions. This thesis also introduces a general partitioning scheme to be applied to a GPGPU cluster to achieve a pure parallel ART algorithm while providing a framework for column partitioning to the pCT system, as well as show sparse visualization patterns that can be found via specified ordering of the equations within the matrix.

ACKNOWLEDGEMENTS

I would like to thank the entire faculty and staff of the School of Computer Science and Engineering at California State University of San Bernardino, particularly my advisor Dr. Ernesto Gomez and my committe member Dr. Keith Schubert. I would like to thank my committee member Dr. Reinhard Schulte of Loma Linda University and Medical Center for his assistance in this research. I would also like to thank my family for their support.

DEDICATION

To my brother who showed me that hacking is as much to do with reading your opponent as it is system administration knowledge, I thank you. I would like to make a dedication to loved ones not with us.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1. INTRODUCTION

1.1 Proton Radiation Therapy and the Motivation for Proton Computed Tomography

Proton radiation therapy has emerged as a way to treat deep-seated tumors that has advantages over X-ray radiation therapy. Proton therapy was first proposed by Robert Wilson in 1946 [25] while working on the design of the Harvard Cyclotron Laboratory and the first patients were treated in 1954 at the Lawrence Berkeley Laboratory (LBL), California[23]. In 1961 scientists at the Harvard Cyclotron Laboratory in collaboration with the Massachusetts General Hospital realized the advantage of stopping the proton beam at the edge of the tumor by introducing Bragg peak treatments as opposed to LBL's "shoot-through" style treatment. The first hospital-based proton treatment facility was installed at Loma Linda University Medical Center (LLUMC) in California [22], dozens of others have now been constructed throughout the world, and by 2008 over 60,000 patients had received proton therapy [16]. Heavy charged particles, including protons, have finite ranges with a Bragg peak just before the end so that they can accurately apply a higher dose at depth.

In order to target the Bragg peak of the proton beam at the precise location of the tumor, the distribution of proton relative stopping powers (RSP) within the patient must be well known prior to treatment. X-ray computed tomography (CT) scanning of the patient is the current method used to obtain the RSP. X-ray CT computes the RSP by reconstructing the linear X-ray attenuation coefficients, given in Hounsfield units using an empirically derived calibration curve. Due to this conversion, there is a 3% to 20% range error, which may result in an under-dose to the tumor volume or an over-dose to the surrounding healthy tissue.

Proton computed tomography (pCT) is an imaging modality that tracks protons as they traverse objects to be imaged (Figure 1.1). The blue arrow represents the path of a proton. The cube object represents the calorimeter, which measures the energy loss of a proton. The four square-grids represent the detector planes of the scanner, which measure the location and direction in which the proton is moving. From this information, one proton history forms one linear equation. Running the whole scan of the patient establishes a linear system in which the pCT method of directly reconstructing the RSP produces more accurate proton range prediction than X-ray CT. Accurate range prediction is important, for example, for sparing critical normal structures.

1.2 General Purpose Graphics Processing Units

General purpose graphics processing units (GPGPUs) were originally intended to process the massively parallel calculations that are inherent in computer graphics. GPGPUs are now becoming more commonly used to perform a wide range of calculations, particularly in medical imaging. The GPGPU assigns a grid of tens of thousands of threads to operate on a problem, such as applying textures to individual pixels, and utilizes those threads in parallel across hundreds of stream processors

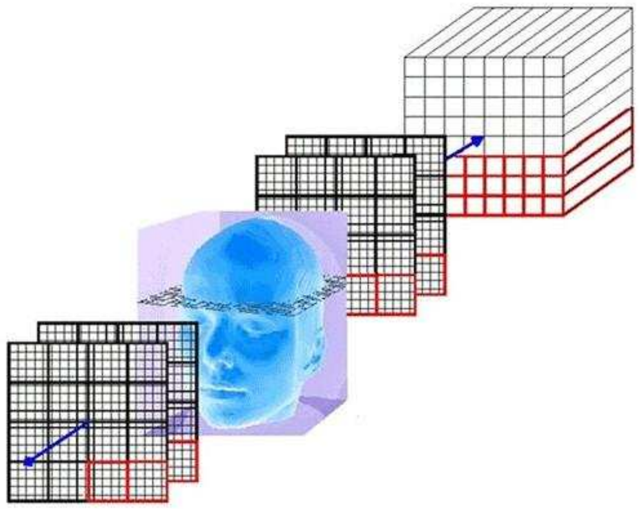


Fig. 1.1: Proton CT concept. Image from [17].

to enhance performance. Each stream processor is a single-instruction in-order execution unit. Figure 1.2 shows a picture of the architecture of a single GPGPU. There are 30 symmetric-multiprocessors, each with 8 stream processors so that the GPGPU in Figure 1.2 has 512 total stream processors. A GPGPU program requires copying data from CPU host memory to the GPGPU device memory, performing algorithms on the data using kernel functions, producing output, and copying the results back to the CPU memory. Synchronizing the symmetric multi-processors is one of the major challenges to GPGPU computing that was addressed in this work.

The GPGPU exploits the parallelism of a computation by launching kernel functions. Each time a kernel function is launched, a grid is formed. Structured within the grid are thread-blocks, and within the thread-blocks are threads. Every thread executes the same code, which is specified in the kernel functions. According to user-defined configuration parameters, the GPGPU grid can either be one or two

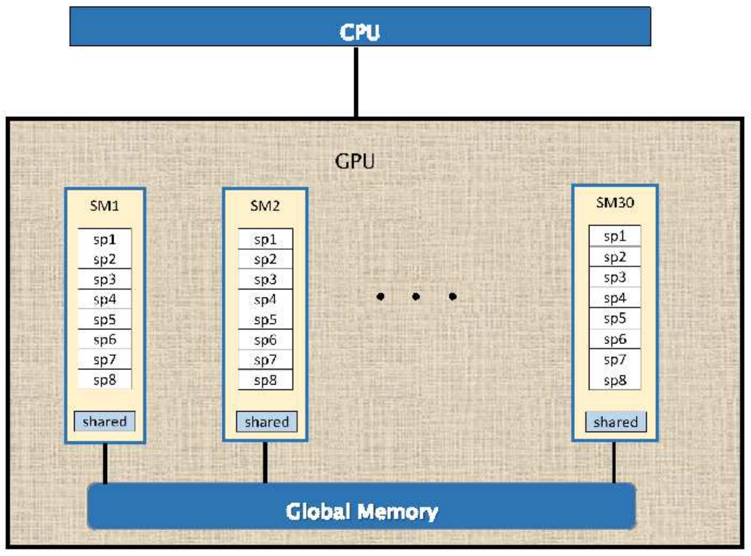


Fig. 1.2: GPGPU architecture consisting of a group of 30 symmetric multiprocessors (SM), each with 8 stream processors (SP). Each SM has its own local shared memory that its SPs can access, and the GPGPU device has global memory that all SPs can access.

dimensional, and the thread-blocks can be either one, two, or three dimensional. Choosing the dimensions of the grid and thread-blocks should be based on the nature of the computation that the kernel is to solve. With a one dimensional grid the thread-blocks are indexed in one dimension, and with a two dimensional grid the thread-blocks are indexed in two dimensions. With one dimensional thread-blocks the threads are indexed in one dimension, with two dimensional thread-blocks the threads are indexed in two dimensions, and with three dimensional thread-blocks the threads are indexed in three dimensions.

Kirk's and Hwu's text on GPGPU processing discusses the history of GPGPU computing, GPGPU architecture, and the CUDA library which allows programming on Nvidia Corporation GPGPUs [11]. The book also discusses how to enhance the

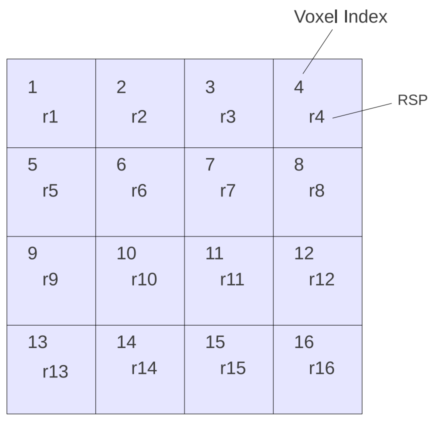


Fig. 1.3: A grid consisting of 16 voxels, each with its index and RSP (r). This two-dimensional image represents the cross-section of a three-dimensional object.

performance of GPGPU processing based on some real-world problems.

1.3 Linear System for Proton Computed Tomography

A pCT image is reconstructed by solving a very large system of approximately 100 million linear equations with 10 million variables for a head-size object. The solution provides the RSP of every partition (voxel) of the object. In order to solve such a system in a reasonable time frame so that pCT can be practical for clinical use, a parallel projection algorithm must be implemented across a multi-processor computing system.

The first step in developing the pCT system of equations is to partition the object into a three dimensional grid where each unit of the grid is called a voxel to mean a volume pixel, and each voxel is numbered with its own index value.

For simplicity, consider a cross-section of the object that is made up of 16 voxels

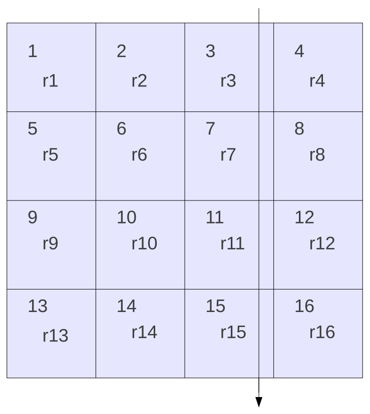


Fig. 1.4: The path of a proton through the grid and formation of a single linear equation of the pCT system. The row of the A-Matrix is: 0 0 1 0 0 0 1 0 0 0 1 0 0 0 1 0, the entry in the b-vector is: r3+r7+r11+r15 (assuming constant chord length).

(see Figure 1.3). The RSP of each voxel is a material property that provides information of how much energy the proton loses in that voxel. For example, bone will have a higher RSP value than soft tissue.

The unknown RSP values inside the object make up the vector $x \in \mathbb{R}^n$ in the system

$$Ax = b,$$

where n is the number of voxels in the object. The A matrix and b vector are obtained as follows: During a pCT scan, incoming protons of a preset initial energy (typically 200 MeV) traverse the object from many direction between 0 and 360 degrees. The most likely paths (MLP) of the protons inside the object are calculated using a Bayesian approach based on the entry and exit locations of the protons at the object boundaries [19]. The protons stop within an energy detector, which is calibrated to determine the traversed water-equivalent path lengths (WEPL). The i^{th}

row of matrix A has the dimension n (i.e. the number of voxels). If the j^{th} voxel of the object was intersected by the MLP of the i^{th} proton, the matrix element a_{ij} will be given the value of the estimated intersection length; otherwise $a_{ij} = 0$. Suppose, for a complete scan, m protons have traversed the object of n voxels; then $A \in \mathbb{R}^{m \times n}$. The m-dimensional vector b contains the estimated WEPL values of the m protons.

Figure 1.4 shows how a single row of matrix A is set up using the MLP information of that the proton corresponding to that row.

Formally, for $i = 1, 2, \ldots, m$ and $j = 1, 2, \ldots, n$

The elements of A, $a_{ij} := \{$ estimated intersection length if the i^{th} proton intersected the j^{th} voxel, 0 otherwise $\}$

$x :=$ the image vector of RSP values for all voxels to in the reconstructed object

$b :=$ the WEPL vector of the proton MLPs.

Note, that $a_{ij} = 1$ was assumed throughout this thesis for simplicity, which does not affect the validity of the results obtained.

1.4 Outline of Thesis

With the overall goal of providing research to advance the pCT project towards clinical use, the following objectives were accomplished: a simulator was developed to generate realistic pCT data, which is described in Chapter 2. Chapter 3 analyzes the performance of two classes of parallel algorithms on a single GPGPU using data generated by the proton simulator. Chapter 4 explores a method to perform a parallel ART algorithm on the pCT system. Chapter 5, the final chapter, outlines an approach to partition the sparse linear system of pCT reconstruction across an entire GPGPU cluster using a sorting scheme for the proton histories. It also discusses future directions. The Appendix contains additional detail of the random number generator used by the proton simulator, contains code segments for the simulator and the GPGPU recontruction code, and shows the specifications of the GPGPU device used.

2. PROTON SIMULATOR FOR PROTON COMPUTED TOMOGRAPHY DATA

2.1 Chapter Introduction

The pCT project is a large effort of several institutions (pCT Collaboration) and data is needed for testing reconstruction algorithms and their implementation on GPGPU hardware before pCT can be used clinically. The proton simulator produced by this research and presented at the IEEE Nuclear Science Symposium And Medical Imaging Conference in Anaheim 2012 can generate realistic data sets for this purpose [26]. The simulator can generate data sets that are made available for the entire pCT Collaboration in order to test different reconstruction algorithms and their implementation.

2.2 Phantoms

The simulator developed in this work uses a general class of phantoms called called a non-homogeneous ellipse object (NEO) that was inspired by Herman's digital head phantom [5]. The NEO can be modified by adding different RSP regions representing anatomical features of a human head. In previous work, the high energy physics simulation tool GEANT4 [1] was used to generate data sets for pCT testing [17]. Generation of these data sets with GEANT4 has been complex and time-consuming therefore, a simulator that can quickly generate high-quality data sets will be bene-

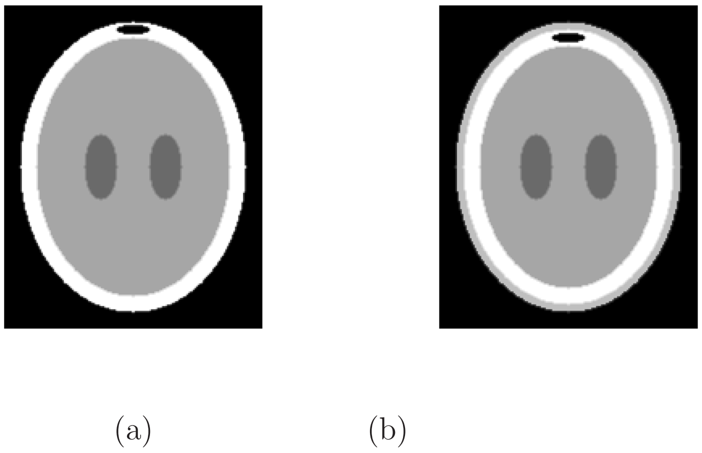


Fig. 2.1: Non-homogeneous Ellipse Objects (NEOs): (a) NEO 1, (b) NEO 2. The difference between these two phantoms is that NEO 2 includes an outer skin layer.

ficial in developing a clinically practical reconstruction methodology.

In this work, two NEO phantom versions, NEO 1 and NEO 2 shown in Figure 2.1 (a) and (b), respectively, were used to generate simulation examples. Both phantoms are bounded by an outer ellipse with major and minor axis length of 180 mm and 140 mm, respectively, and have inner regions representing the cerebral ventricles filled with cerebro-spinal fluid (RSP of 0.9), brain tissue (RSP of 1.04), an air-filled frontal sinus (RSP of 0.0), and compact skull bone (RSP of 1.6). NEO 2 also includes an outer skin layer.

The results presented in this thesis were produced with a 2D version of the NEO phantoms. The 2D simulation examples used 1 mm x 1 mm pixels and a reconstruction area of 200 mm x 160 mm, consistent with the cross-sectional size of an adult human head, resulting in 32,000 pixels. For assigning RSP values for each pixel, one can choose between three methods. One should note that each point in the elliptical regions of the NEO has a defined RSP value. The *center-point method* selects the

RSP at the center of each pixel. The *corner-point averaging method* uses the average of the RSP values at the four corners of the pixel, providing a smoother transition between different RSP regions than the *center-point method*. The *weighted-area averaging method* provides the smoothest transition between two different RSP regions by setting the boundary pixel values equal to the sum of each region's RSP times the fraction of its area with respect to the pixel area. The simulation examples in this thesis used the *corner-point averaging method*. Figures 2.2 and 2.3 provide an illustration of the RSP selection methods. Figure 2.2 shows an image of the NEO phantom in color with an enlarged pixel on the boundary of brain matter and one of the cerebral ventricle regions. Figure 2.3 shows the three different methods for selecting the RSP for each voxel. Figure 2.3(a) is for the *center-point method* and it shows that the RSP in the center, the blue region, will be assigned to that pixel. Figure 2.3(b) is for the *corner-point-averaging method* and it shows that the RSP at the four corners will be averaged together, two from the blue region, and two from the green region, to assign the value to that pixel. Figure 2.3(c) is for the *weighted-area-averaging method*. The RSP for that pixel is calulated as follows: RSP = (A1*RSP1 + A2*RSP2)/2, where A1 is the area of the green region or cerebral ventricle, RSP1 is the RSP for the cerebral ventricle, A2 is the area of the blue region, or brain matter, and RSP2 is the RSP for the brain matter.

The simulator can also create 3D phantoms by stacking a user-defined number of slices with user-defined height of 2D phantoms, which results in an elliptical cylinder. Stacking 200 1mm-slices of the phantom (for example) results in an elliptical cylinder within a reconstruction volume of 200x200x160 = 6.4 million voxels, where each voxel

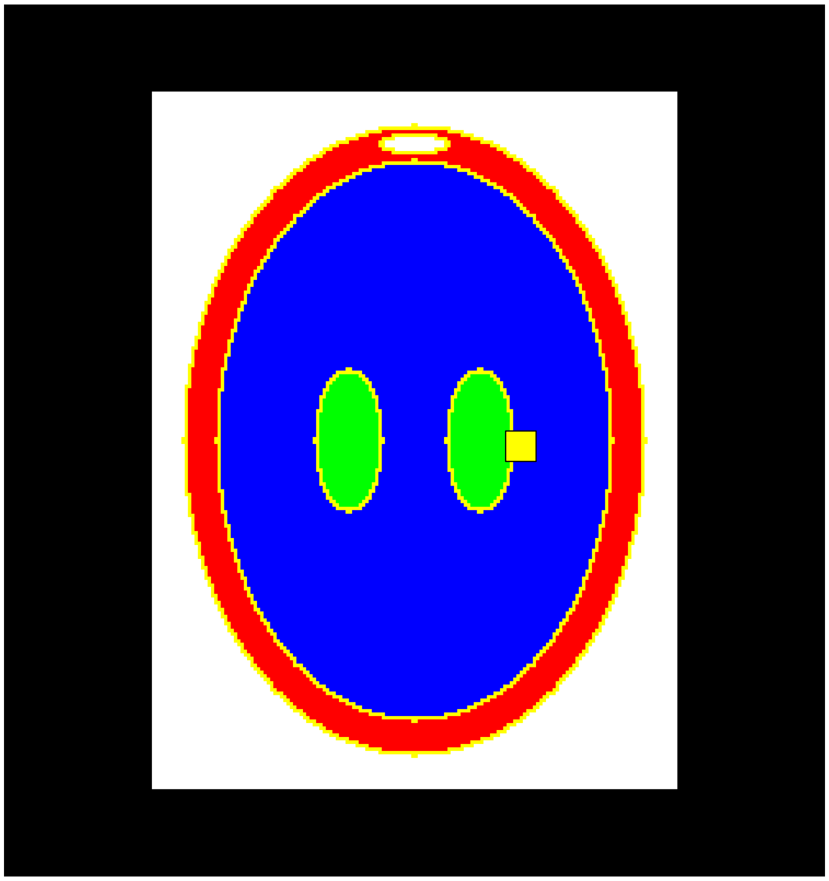


Fig. 2.2: NEO phantom in color. The black region is the region outside the reconstruction space, the white color repseresents air, which surrounds the outer ellipse and also lies within the outer skull region. The blue color represents brain matter, the green color represents the cerebral ventricles, and the yellow color is for pixels that lie on intersecting regions of different RSP. The image shows an enlarged yellow pixel.

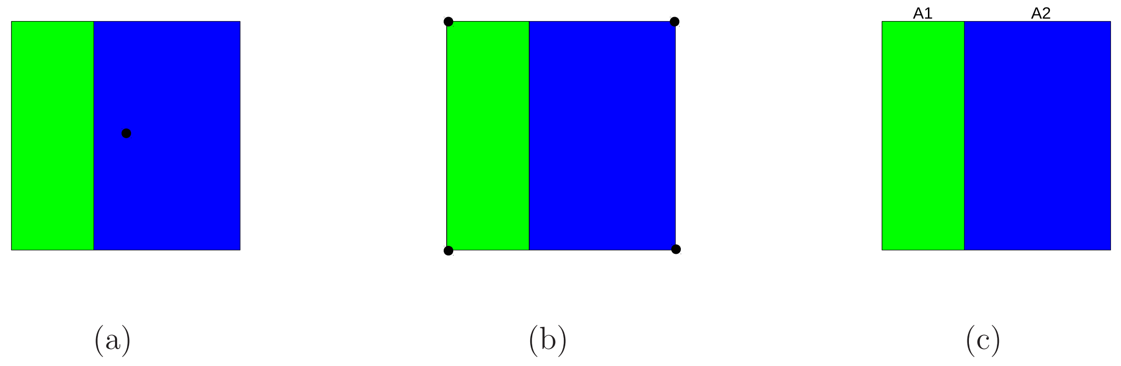


Fig. 2.3: Selecting the RSP for the enlarged pixel that lies on the boundary of brain matter (blue) and one of the cerebral ventricles (green): (a)center-point method, (b)corner-point-averaging method, (c) weighted-area-averaging method.

is 1 mm^3, creating data volumes consistent with the clinical setting of a human head.

2.3 Simulated Proton Paths

The path of a proton traversing the object to be scanned can be characterized at a specified depth by its lateral and vertical coordinates and lateral and vertical directions relative to the direction of the proton beam. The scattering in the lateral and vertical directions can be considered as two independent statistical processes so that the scattering in vertical direction can be excluded from simulations with a 2D phantoms. With the general direction of the proton beam as the u-axis, and the t-axis to represent the lateral displacement, the location and direction of a proton at any depth u_1 is given by the 2D vector,

$$y_1 = \begin{pmatrix} t_1 \\ \theta_1 \end{pmatrix}$$

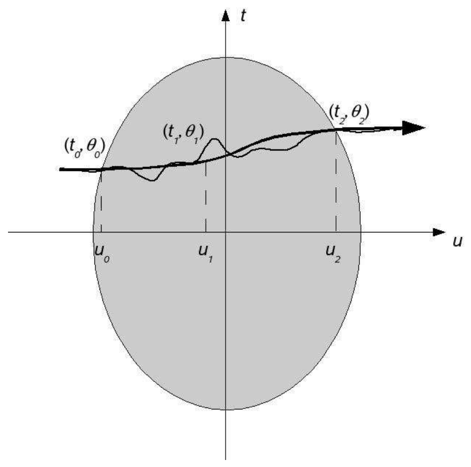


Fig. 2.4: Scattering geometry in the u-t plane. Image from [17].

where t_1 is the lateral displacement and θ_1 is the angular deviation relative to the u-axis. Figure 2.4 shows a proton path traversing an object located in the $u - t$ plane. The entry and exit vectors are displayed at their appropriate depth, respectively. The elements of the exit vector are distributed according to a joint-normal probability density function with mean zero.

The simulator generates parallel proton paths that are transported from a virtual point source at infinite distance into the reconstruction space and then into the phantom. The entry points into the reconstruction volume are uniformly distributed over a user-selected interval along the horizontal t-axis of the beam-specific coordinate system (Figure 2.5) for the 2D simulation and along the horizontal axis and vertical v-axis for the 3D simulation. In the simulation examples, the initial displacements were uniformly distributed over the interval [-125 mm, 125 mm] along the t-axis. A cone beam option is planned a future version of the simulator.

Once the entry point into the phantom is determined, the exit point is calculated by projecting a straight-line path with the addition of a lateral and angular displacement to model multiple Coulomb scattering (MCS) [12]. The actual proton path connects the entry and exit points and then, outside the phantom, the path continues as a straight line in exit direction. The simulator allows for either straight-line or cubic-spline paths between entry and exit points. The simulation examples of this thesis used straight-line paths.

The cubic spline path is calculated as follows:

A 3^{rd} order polynomial $q(x)$ for which

$$q(x_1) = y_1 q(x_2) = y_2 q'(x_1) = k_1 q'(x_2) = k_2$$

is

$$q(x) = (1-t)y_1 + ty_2 + t(1-t)(a(1-t) + bt)$$

where

$$t = \frac{x - x_1}{x_2 - x_1}$$

and

$$a = k_1(x_2 - x_1) - (y_2 - y_1)$$

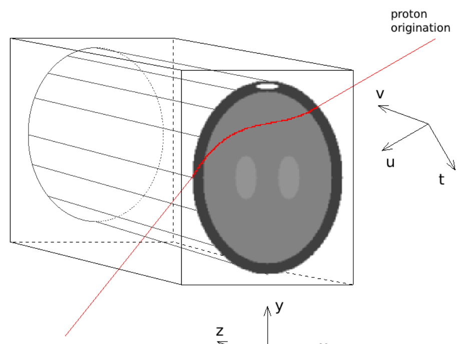


Fig. 2.5: Proton simulator geometry. A randomly generated proton path passing through the digital phantom in the reconstruction space is shown. The t, v, u axes are the beam-specific coordinate system and the x, y, z axes are the coordinates of the global reconstruction space.

$$b = -k_2(x_2 - x_1) + (y_2 - y_1).$$

A Markov process to model an even more realistic path is planned for a future version of the simulator.

Proton paths are directed from a user-specified number of projection angles with a user-defined angular spacing interval. The simulation examples used 180 projection angles with 2-degree spacing intervals, covering a full circle with a series of parallel 2D beams. For each projection, the proton paths are generated in the *ut*-plane of the beam-specific coordinate system.

The random variables describing lateral and angular displacement at the proton exit are approximately distributed according to a bivariate random normal distribution described by Highland's formalism of MCS [8] with parameters recommended by the Particle Data Group [14]. For the 2D mode of this simulator, a bivariate normal distribution is used to generate random number pairs to represent the exiting lateral displacement and angular deviation due to MCS of the proton within the phantom.

For the 3D mode, the random exit parameters are independently calculated for the horiozontal $u-t$ plane and the vertical $u-v$ plane, respectively.

For the bivariate normal distribution, let the two random variables be X and Y, then the probability density function is

$$\begin{aligned} f(x,y) &= \frac{1}{2\pi\sigma_x\sigma_y\sqrt{(1-\rho^2)}} exp\left(-\frac{1}{2(1-\rho^2)}\right. \\ &\qquad \left[\frac{(x-\mu_x)^2}{\sigma_x^2} + \frac{(y-\mu_y)^2}{\sigma_y^2}\right. \\ &\qquad \left.\left. -\frac{2\rho(x-\mu_x)(y-\mu_y)}{\sigma_x\sigma_y}\right]\right), \end{aligned}$$

where

$$\begin{aligned} \rho &= corr(X,Y) = \frac{v_{XY}}{\sigma_{xy}}, \\ v_{XY} &= cov(X,Y), \\ \mu &= \begin{bmatrix} \mu_x \mu_y \end{bmatrix} \\ \Sigma &= \begin{bmatrix} \sigma_x^2 & \sigma_{xy} \\ \sigma_{xy} & \sigma_y^2 \end{bmatrix} \end{aligned}$$

The simulator has its own bivariate normal random number generator (BNRNG) described in the Appendix A. The BNRNG begins with two uniform random variables generated using the $rand()$ function in the C standard library. It then converts those two uniform random variables to two standard independent random normals using the Marsaglia Polar-Method (modification of Box-Muller Method) [13],[3]. The two independent normals are then converted into a pair of bivariate normals using the covariance matrix.

The elements of a given row of the A-matrix are the chord lengths of a given path through the pixels/voxels it intersected. In the simulator, the element $a_{i,j}$ is the chord length of the i^{th} path through the j^{th} pixel/voxel. For simplicity, the simulation examples used a constant chord length of the pixel size (1 mm). Using an estimate for actual chord length or mean chord length could result in a more accurate reconstruction, though doing so would require more computation and thus increase the time complexity [18].

The path of the proton is generated in the $t - v - u$ coordinate system and then mapped to the stationary $x - y - z$ coordinate system using a Givens rotational matrix. The simulator then takes the x, y, z location of the path and intersects the path with the voxels of the reconstruction space. Each proton path will intersect very few of the voxels in the entire reconstruction space, so the vast majority of the entries in each row will be zero. To identify every voxel intersected by a proton path, the simulator uses a method similar to the Digital Difference Analyzer method [5].

The noiseless WEPL value of the i^{th} proton is calculated forming the inner product of the i^{th} row vector with the actual-solution-vector of the phantom. The simulator also has a feature that enables added WEPL noise. To do this, the noiseless WEPL value is converted into an equivalent energy value of the exiting proton using the NIST PSTAR database [15]. Next, a random energy value is drawn from a normal distribution with mean equal to the noiseless energy and a standard deviation calculated using Tschalar's theory of energy straggling [24]. This noisy energy value is then converted back into a noisy WEPL value.

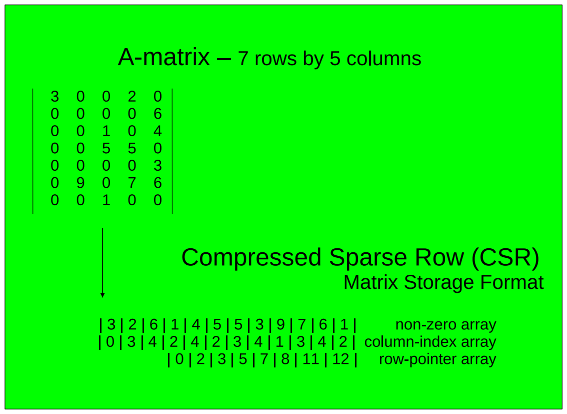

A-matrix – 7 rows by 5 columns
3 0 0 2 0
0 0 0 0 6
0 0 1 0 4
0 0 5 5 0
0 0 0 0 3
0 9 0 7 6
0 0 1 0 0
Compressed Sparse Row (CSR)
Matrix Storage Format
| 3 | 2 | 6 | 1 | 4 | 5 | 5 | 3 | 9 | 7 | 6 | 1 | non-zero array
| 0 | 3 | 4 | 2 | 4 | 2 | 3 | 4 | 1 | 3 | 4 | 2 | column-index array
| 0 | 2 | 3 | 5 | 7 | 8 | 11 | 12 | row-pointer array

Fig. 2.6: An example of converting a dense matrix to compressed sparse row format.

The number of voxels of the reconstruction volume defines the number of columns in the A-matrix. A reconstruction volume of 200 mm x 160 mm x 200 mm bounds the typical adult human head. A 3D simulation with a reconstruction volume of 6.4 million voxels and 64 million proton histories creates the system $Ax = b$ with $A \in \mathbb{R}^{64M \times 6.4M}$, $b \in \mathbb{R}^{64M}$. The simulator writes the A-matrix to a file in a format that can be easily read into compressed sparse row (CSR) matrix format [2] taking approximately 100 GB of disk space (Figure 2.6). Storing the A-matrix in dense form would require over 1 peta-byte of disk space (1 peta $= 10^{15}$). CSR format is the most appropriate sparse structure to store the pCT system because as each history is recorded, a new row of the system is added, one-by-one. The data generated from a 2D simulation follows the same format.

2.5 Chapter Summary

The motivation for simulated data is clear and at this point there exists a proton simulator to generate high-quality pCT data sets quickly and at low cost, saving the expense of performing test pCT runs in the laboratory. Data distribution is an important aspect of the collaborative effort, and this simulation code will be available for free use for all groups involved in the pCT project. The customizeable parameters of the simulations allow other experts to generate data and perform further analysis, [26].

3. ALGORITHM COMPARISON AND GPGPU IMPLEMENTATION

3.1 Chapter Introduction

The system $Ax = b$ is too large to be solved by a direct method such as Gaussian elimination. The number of columns of A equals the number of voxels to be reconstructed, which can be of the order of 10 million columns for a 3D head-sized object. To get a precise image, the voxels have to be relatively small. Using a voxel size of 1 mm x 1 mm x 1 mm to partition the reconstruction space of 256 mm x 256 mm x 150 mm will result in a system with just under 10 million columns. The number of rows should be about 10 times larger than the number of columns so that every voxel will get intersected by on average 10 protons, which will be necessary to get a relativley noise-free image. Therefore, the matrix A will have around 100 million rows by 10 million columns, which is in fact much too large for direct or explicit solution methods.

3.2 Algebraic Reconstruction Technique (ART)

The algebraic reconstruction technique (ART) is an iterative method to solve sparse systems in which there are many zeros in the A matrix, which is the case in the pCT problem as will be discussed later. First published by Kaczmarz in 1937, [10], ART applies successive orthogonal projections onto convex sets - SOPOCS, the sets being

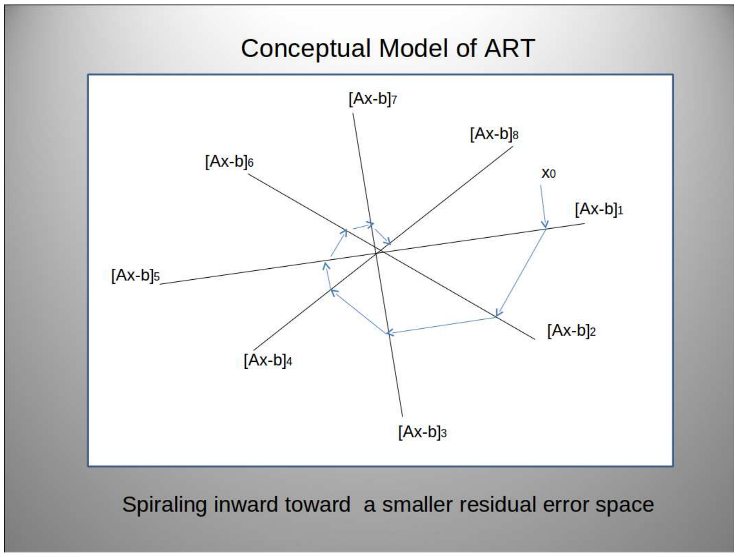


Fig. 3.1: ART: successive orthogonal projections onto the hyperplanes where each hyperplane represents a linear constraint. In this figure, the hyperplanes denoted by black lines are indexed from 1 to 8. The x0 indicates the start point and the blue arrows denote the projections.

the hyperplanes

$$H_i = \{\langle a^i, x\rangle = b_i : x \in \mathbb{R}^n, i = 1, 2, \ldots, m\}.$$

SOPOCS "spirals" the solution into a subspace of smaller residual error as can be seen in Figure 3.1.

Given the control sequence $\{i(k)\}_{k=0}^{\infty}$, where $i(k) = k \bmod m + 1$, the ART algorithm is as follows:

Initialize $x^0 \in \mathbb{R}^n$ arbitrarily.

For $k = 0, 1, \ldots$

$$x^{k+1} = x^k + \lambda_k \frac{b_{i(k)} - \left\langle a^{i(k)}, x^k \right\rangle}{||a^{i(k)}||^2} a^{i(k)}$$

where $\{\lambda\}_{k=0}^{\infty}$ is a sequence of user-determined relaxation parameters.

Given the vast size of the pCT linear system of equations, using ART would require too much time and memory on a single computer making it impractical for clinical use. Block-iterative-projection (BIP) and string-averaging-projection (SAP) algorithms allow for the system to be partitioned into sets of hyperplanes (rows) called blocks or strings, respectively, so that much of the calculations can be done in parallel and less memory is required. BIP algorithms perform parallel computation within a block and require sequential processing between blocks. On the "flip" side, SAP algorithms perform parallel computation between strings and require sequential computation within strings.

3.3 String Averaging Projection Algorithms

Figure 3.2 provides a graphical representation of how SAP algorithms project the iterative solutions into regions of lower residual error. The figure shows how the current iterate x^k is projected sequentially onto three different strings of hyperplanes (H1, H2, H6; H4, H6; H6). The resulting three points, forming the vertices of the blue triangle, are then combined to form the new iterate x^{k+1}.

String-avergaing-projection (SAP) Algorithm:

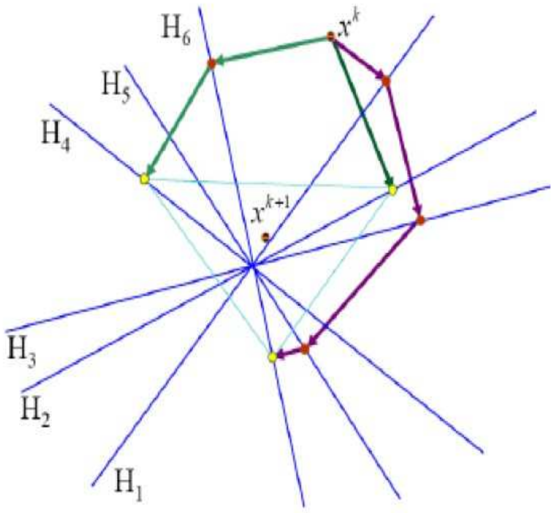


Fig. 3.2: Illustration of SAP [17].

x^0 arbitrary, given x^k, for each $t = 1, 2, ..., M$ set $y^0 = x^k$ and calculate, for $i = 0, 1, ..., m(t) - 1$,

$$y^{i+1} = y^i + \lambda_k \frac{b_i - \left\langle a^i, y^i \right\rangle}{||a^i||^2} a^i$$

where $\{\lambda_k\}_{k=0}^{\infty}$ is a sequence of user-defined relaxation parameters and let $y^t = y^{m(t)}$ for each $t = 1, 2, ..., M$. Then calculate the next iterate by

$$x^{k+1} = \sum_{t=1}^{M} w_t y^t,$$

where the w_ts are weights such that $\sum_{t=1}^{M} w_t = 1$.

Figure 3.3 further illustrates the functionality of the SAP algorithms with a flow chart. The current iterate x^k is sent to all strings where it is projected sequentially onto all hyperplanes of the string. The endpoints of each string projection are averaged to get the new iterate x^{k+1}.

On the GPGPU, each time a kernel function is launched (or called), it opens a new grid of blocks of thread processes, as specified by the configuration parameters. The code in the kernel function then specifies how the threads operate on the data. The implementation of SAP on the GPGPUs in this research used three kernel functions

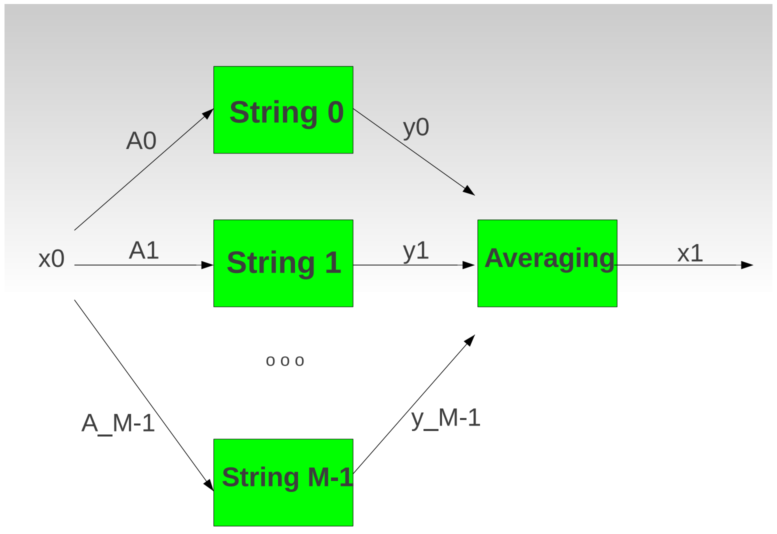


Fig. 3.3: An illustration of the flow of SAP algorithms.

(refer to Appendix C, Listing 3.5, for the SAP CUDA code). The first kernel assigns the current x-iterate to each string. The second kernel performs the projections. In this kernel, a thread process of the GPGPU grid is assigned to a set of hyperplanes (string), and each thread process performs row projections and then writes the result of the last projection into temporary device memory. Thus, each string requires a temporary memory equal to n - the number of columns of the system. As the number of string partitions, M, increases, the number of rows within each string decreases and so the time complexity of the calculation is reduced in this projection step of the kernel. This demonstrates the inverse relationship between time performance and memory complexity. The third kernel averages the updates of each string into the new x-iterate.

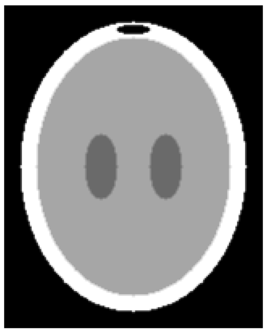
3.3.1 Reconstruction Results

Figures 3.4 through 3.6 show examples of reconstructions obtained from simulated data without added WEPL noise. All reconstructions in this subsection were performed with 20 cycles of iterations through all proton histories using the SAP algorithm with 100 string partitions. Figure 3.4 illustrates the 2D reconstruction of two slightly different head phantoms, NEO 1 and NEO 2. The reconstructions clearly show the different anatomical regions, characterized by different RSP values, for both phantom reconstructions.

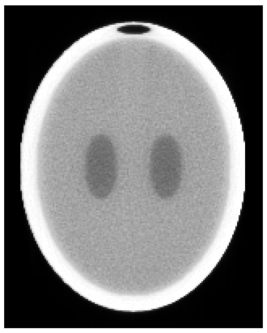
Figure 3.5 analyzes the dependence of the image quality on the number of proton histories entering the reconstruction area, expressed as multiples of the number of pixels (32,000 pixels). The resulting images show decreasing image noise with increasing number of histories. The reason for the image noise, which is present despite the noiseless WEPL values, is the use of constant chord lengths in the reconstruction. Obviously, the noise introduced by inaccurate chord lengths can be compensated by a larger number of proton histories.

Figure 3.6 demonstrates the effects of different relaxation parameters on image quality. There is a decrease of image noise with an increase in λ, however, the effect of increasing λ on noise reduction appears to saturate between $\lambda = 0.1$ and $\lambda = 0.5$.

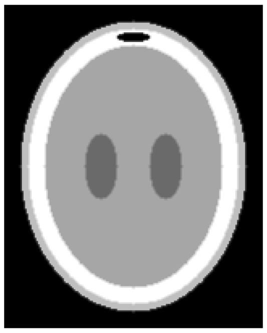
Figures 3.7 and 3.8 demonstrate the effects of different numbers of histories and relaxation parameters on the quantitative accuracy of RSP values using line profiles through the two ventricles of the NEO 1 phantom. Given a sufficient number of histories and choice of an adequate relaxation parameter, accurate reconstruction of the RSP values was obtained.



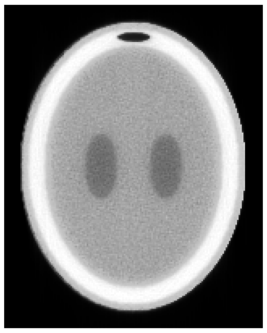
(a)



(b)



(c)



(d)

Fig. 3.4: Reconstructions ($n = 200 \times 160$ pixels) of the NEO 1 and NEO 2 phantoms ($\lambda = 0.1$). (a) NEO 1 phantom, (b) reconstruction of NEO 1 phantom, (c) NEO 2 phantom, (d) reconstruction of NEO 2 phantom.

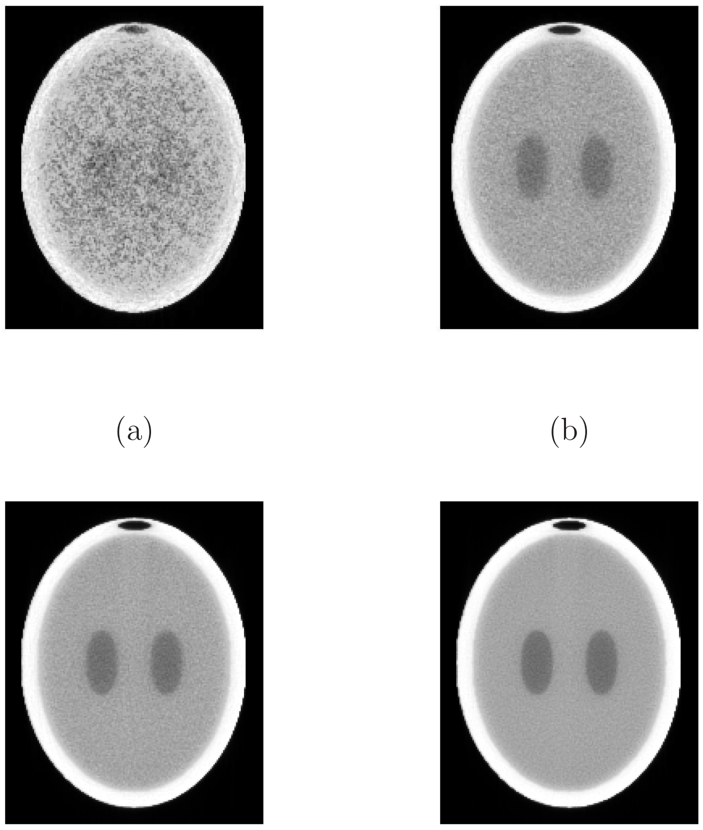

(a) (b) (c) (d)

Fig. 3.5: Reconstruction of the NEO 1 phantom with different number of histories m, expressed as multiples of the number of pixels n ($\lambda = 0.1$). (a) $m = n$, (b) $m = 5n$, (c) $m = 10n$, (d) $m = 20n$.

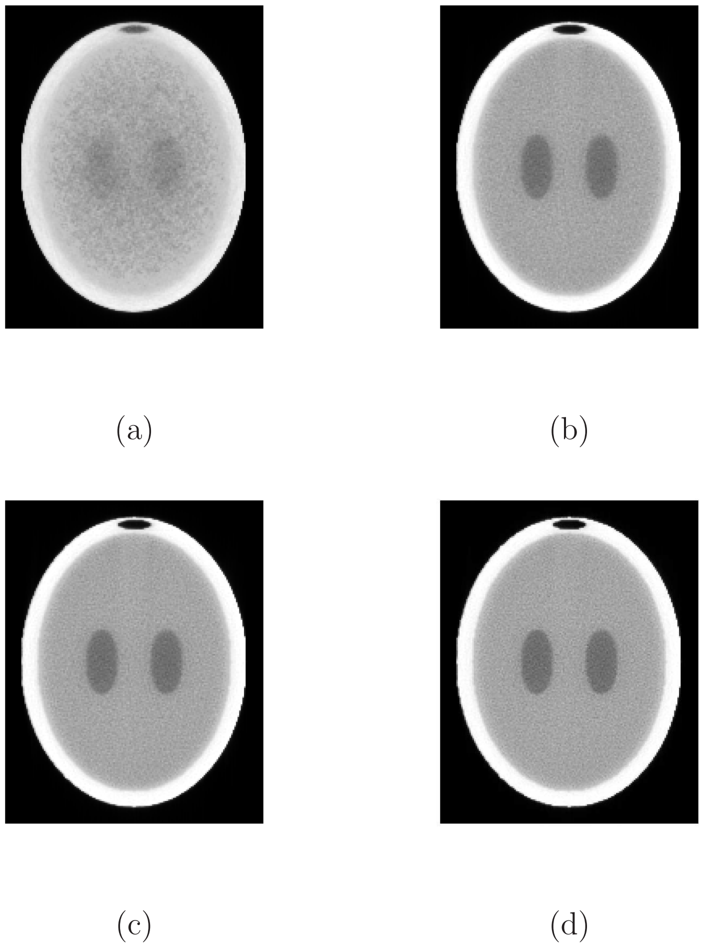


Fig. 3.6: Reconstructions of the NEO 1 phantom using different relaxation parameters λ. (a) $\lambda = 0.01$, (b) $\lambda = 0.1$, (c) $\lambda = 0.2$, (d) $\lambda = 0.5$.

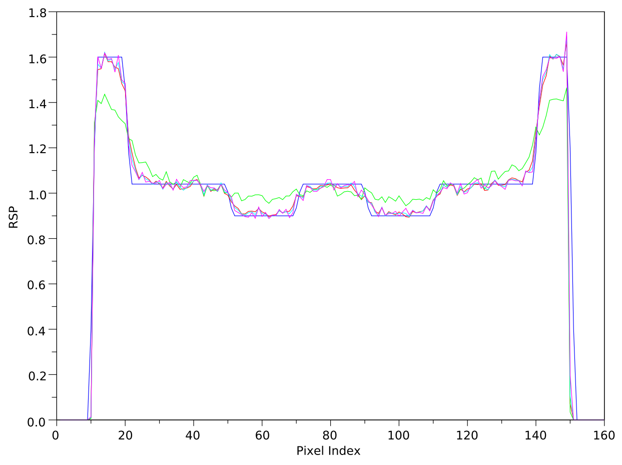


Fig. 3.7: Line profiles through the ventricles of the NEO 1 phantom for different relaxation parameters λ: NEO 1 (blue), $\lambda = 0.01$ (green), $\lambda = 0.1$ (red), $\lambda = 0.2$ (cyan), $\lambda = 0.5$ (magenta).

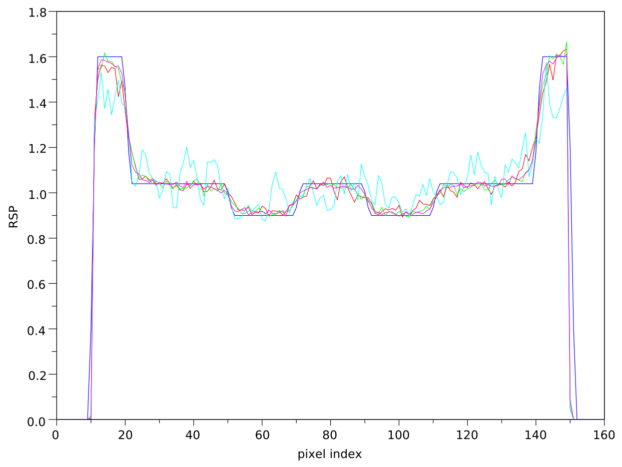


Fig. 3.8: Line profiles through the ventricles of the NEO 1 phantom using different numbers of histories: NEO 1 (blue), $m = n$ (cyan), $m = 5n$ (red), $m = 10n$ (green), $m = 20n$ (magenta).

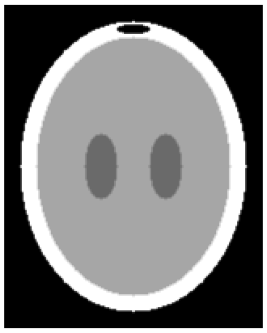
3.3.2 Different String Sizes

Figures 3.9 and 3.10 analyze the impact of the string partition sizes on the image quality. Figure 3.9 displays the reconstructed image after the first 10 cycles using a constant relaxation parameter of λ=0.1. Figure 3.9 (a) is the NEO 1 phantom; Figures 3.9 (b) - (f) show the reconstructed images using 100, 200, 400, 800, and 1000 string partitions, respectively.

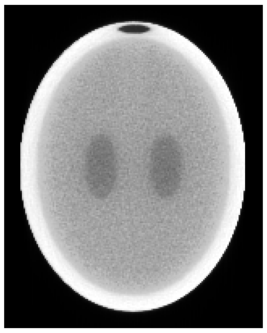
Figure 3.10 displays the reconstructed image after the first 10 cycles adjusting the relaxation parameter λ to be proportional to the number of string partitions, starting with $\lambda = 0.1$ for 100 string partitions. Figure 3.10 (a) is the NEO 1 phantom; Figures 3.10 (b) - (f) show the reconstructed images using 100, 200, 400, 800, and 1000 string partitions, respectively. A comparison between the Figures 3.9 and 3.10 reveals the significance of keeping the relaxation parameter proportional to the string partition size in a given range.

Figure 3.11 shows a plot of the relative error, $\sum_{j=1}^{n} |x_j^* - x_j^n| / \sum_{j=1}^{n} |x_j^*|$, between the actual solution x^*-vector for the NEO 1 phantom and the solutions for the SAP implementations varying the number of string partitions using a constant relaxation parameter of λ=0.1. The Figure shows that increasing the number of string partitions without changing the relaxation parameter leads to slower convergence.

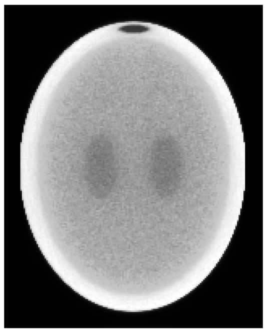
Figure 3.12 shows a plot of the relative error for the SAP implementations varying the number of string partitions with proportional relaxation parameters. The Figure shows that keeping the relaxation parameters proportional to the number of string sizes leads to minimal differences in the convergence.



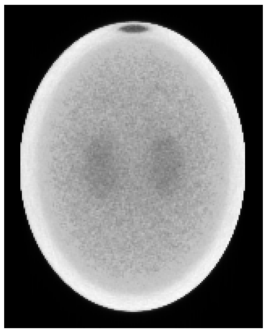
(a)



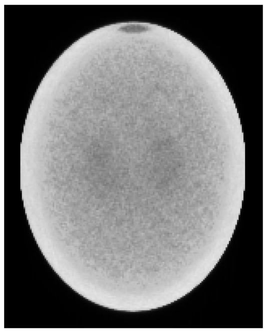
(b)



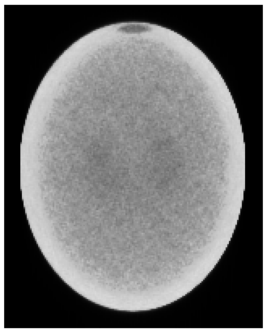
(c)



(d)

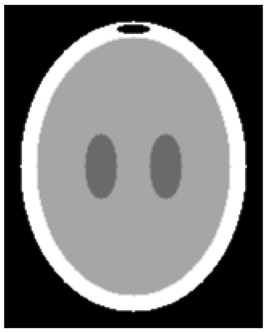


(e)

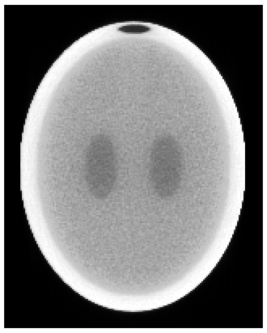


(f)

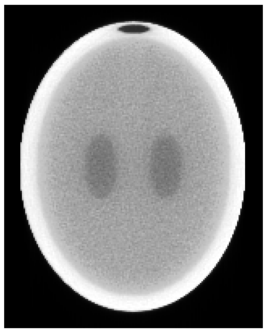
Fig. 3.9: Reconstruction images after 10 cycles of SAP using a constant relaxation parameter of λ=0.1: (a) original NEO 1 phantom (b) 100 string partitions (c) 200 string partitions (d) 400 string partitions (e) 800 string partitions (f) 1000 string partitions.



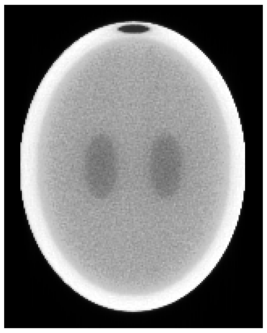
(a)



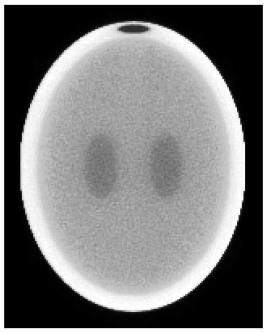
(b)



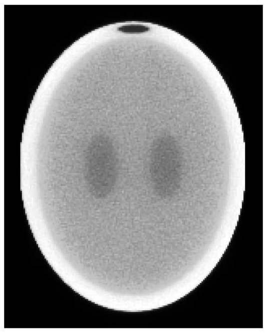
(c)



(d)



(e)



(f)

Fig. 3.10: Reconstruction images after 10 cycles of SAP: (a) original NEO 1 phantom (b) 100 string partitions and λ=0.1 (c) 200 string partitions and λ=0.2 (d) 400 string partitions and λ=0.4 (e) 800 string partitions and λ=0.8 (f) 1000 string partitions and λ=1.0.

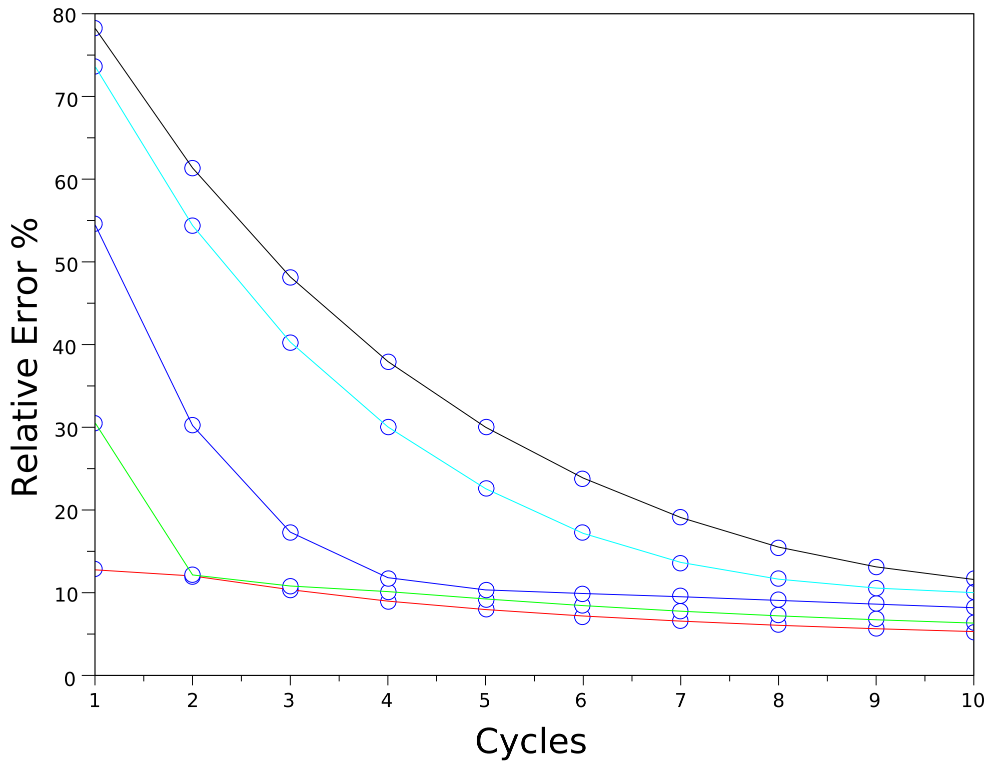


Fig. 3.11: Plot of the relative error percent for SAP using varying the number of string partitions with constant λ=0.1 for cycles 1 to 10. The red line is for the 100 string partitions, the green line is for the 200 string partitions, the blue line is for the 400 string partitions, the cyan line is for the 800 string partitions, and the black line is for the 1000 string partitions.

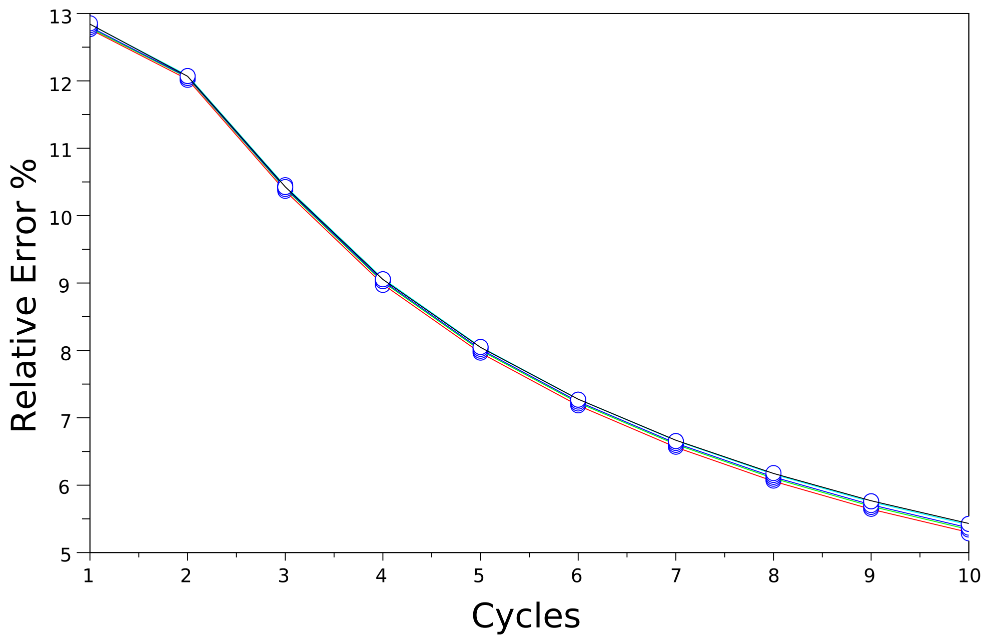


Fig. 3.12: Plot of the relative error percent for SAP using varying the number of string partitions with proportional relaxation parameters for cycles 1 to 10. The red line is for the 100 string partitions with λ=0.1, the green line is for the 200 string partitions with λ=0.2, the blue line is for the 400 string partitions with λ=0.4, the cyan line is for the 800 string partitions with λ=0.8, and the black line is for the 1000 string partitions with λ=1.0.

3.3.3 Execution Times for Different Numbers of String Partitions

Tables 3.1 - 3.5 show the execution times using the Tesla 2090 GPGPU for SAP. For all kernel functions, the grid used was one-dimensional. The execution was performed with either 1, 10, 50, 100, 1,000, or 32,768 threads. When using between 1 and 100 threads there was only one thread-block of one dimension. For the 1,000 thread executions, there were 10 one-dimensional thread-blocks, each with 100 threads. For the implementation which utilized 32,768 threads, there was a one dimensional grid of 64 thread-blocks, each containing 512 threads.

The columns of Tables 3.1 - 3.5 indicate the execution times for different number of threads used. When using less than 400 string partitions, the speedup achieved by increasing the number of GPGPU thread processes was negligible as reflected in Tables 3.1 and 3.2. For 400 string partitions (Table 3.3) increasing the number of thread processes to 1,000 does achieve speedup. However, beyond utilization of 1,000 threads no further speedup was seen. This can be explained by the fact that the GPGPU card has 512 core processors and its resources reaches full capacity between 100 threads and 1,000 threads. Table 3.4 shows that using 800 string partitions leads to greater speedup than 400 string partitions. Table 3.5 shows that 1,000 string partitions results in the maximum speedup.

3.4 Block Iterative Projection Algorithms

As described earlier, BIP algorithms allow for parallelism within the row partitions referred to as blocks, and then require sequential updates after each block iteration. In more precise mathematical notation:

Tab. 3.1: Execution times for the SAP reconstructions in milli-seconds with 100 string partitions for cycles 1 through 10 using a different number of GPGPU threads, t_g, in the kernel functions.

cycle	t_g=32768	t_g=1000	t_g=100	t_g=50	t_g=10	t_g=1
1	8760	10070	9250	9060	7630	6480
2	13070	11940	12700	11210	8450	7800
3	16260	15490	16350	14190	9430	8270
4	18480	18880	18780	17800	9810	9870
5	21810	21060	21140	19750	11050	10010
6	24900	24920	24270	22610	12810	11520
7	27820	28250	27950	25380	13720	11730
8	31350	30240	30470	29020	14080	12980
9	33360	34110	33620	31010	15100	14390
10	36460	37620	37360	33860	17020	15510

Tab. 3.2: Execution times for the SAP reconstructions in milli-seconds with 200 string partititions for cycles 1 through 10 using a different number of GPGPU threads, t_g, in the kernel functions.

cycle	t_g=32768	t_g=1000	t_g=100	t_g=50	t_g=10	t_g=1
1	8670	7690	7790	7120	7090	8570
2	9230	9500	8770	8540	7580	10280
3	11810	10740	10360	10020	7720	12560
4	12540	12220	12800	12080	9550	15040
5	14310	13360	15100	13640	9090	17400
6	16540	15040	15970	15010	9810	18950
7	17760	17370	16810	15730	10410	21760
8	20400	18820	19300	17170	11870	24020
9	22280	20940	19850	19470	13420	26850
10	24390	21170	22250	19910	12410	27950

Tab. 3.3: Execution times for the SAP reconstructions in milli-seconds with 400 string partitions for cycles 1 through 10 using a different number of GPGPU threads, t_g, in the kernel functions.

cycle	t_g=32768	t_g=1000	t_g=100	t_g=50	t_g=10	t_g=1
1	7360	7020	7290	6920	7740	11690
2	9230	7270	7410	7910	8430	16530
3	10480	9330	9360	7790	8660	20760
4	11960	9540	9130	9700	8580	24680
5	12190	9760	10040	10640	9360	30360
6	13420	10810	11580	10010	10030	34660
7	15350	12580	11710	11070	12200	39170
8	17070	13370	13620	11490	12310	43880
9	18040	13660	14110	13040	12290	49430
10	18630	13980	14240	13550	13010	54690

Tab. 3.4: Execution times for the SAP reconstructions in milli-seconds with 800 string partitions for cycels 1 through 10 using a different number of GPGPU threads, t_g, in the kernel functions.

cycle	t_g=32768	t_g=1000	t_g=100	t_g=50	t_g=10	t_g=1
1	6780	6450	6820	7100	6860	15340
2	8090	7830	6680	7570	9470	25380
3	9550	7170	7620	8100	10010	34870
4	9880	7390	7730	8490	11100	44510
5	9740	8390	9010	8750	11960	54550
6	10630	8970	8770	9490	12800	64050
7	11990	8630	9430	10150	13950	72960
8	13240	9170	9850	10920	15880	81670
9	13240	10870	11290	11410	17700	91700
10	14630	9990	11620	11160	18980	102000

Tab. 3.5: Execution times for the SAP reconstructions in milli-seconds with 1000 string partitions for cycles 1 through 10 using a different number of GPGPU threads, t_g, in the kernel functions.

cycle	t_g=32768	t_g=1000	t_g=100	t_g=50	t_g=10	t_g=1
1	6230	6820	7120	6230	7390	17480
2	7790	7630	6590	7580	9240	29510
3	7680	7010	8200	7370	10770	42550
4	8990	7060	8400	8770	12480	54240
5	9730	8310	8690	9130	13640	65040
6	9620	8960	9140	9510	15560	76770
7	10320	8110	8860	9930	16560	89640
8	11550	9040	9430	11650	17430	101410
9	12350	8840	11230	12270	18940	112630
10	13000	10330	10290	12780	21490	124510

Block-iterative-projection (BIP) Algorithm:

$$x^{k+1} = x^k + \lambda_k \sum_{i \in I_{t(k)}} w^k(i) \frac{b_i - \left\langle a^i, x^k \right\rangle}{||a^i||^2} a^i$$

where $I = \{1, 2, \ldots, m\}$; I is partitioned into M blocks such that $I = I_1 \cup I_2 \cup \cdots \cup I_M$; $\{t(k)\}_{k=0}^{\infty}$ is a control sequence over the set $\{1, 2, \ldots, M\}$ of block indices; w^k are weight vectors of the form $w^k = \sum_{i \in I_{t(k)}} w^k(i) e^i$ where e^i is the i^{th} standard basis vector; $\{\lambda_k\}_{k=0}^{\infty}$ is a sequence of user-defined relaxation parameters.

One type of BIP algorithm is called Ordered subsets simultaneous algebraic reconstruction technique (OS-SART). OS-SART was used for the BIP recontructions in this thesis.

OS-SART Algorithm:

For $j = 1, .., n$

$$x_j^{k+1} = x_j^k + \lambda_k \frac{1}{\sum_{i \in I(k)} a_j^i} \sum_{i \in I_{t(k)}} \frac{b_i - < a^i, x^k >}{\sum_{l=1}^{n} a_l^i} a_j^i$$

where $\{\lambda_k\}_{k=0}^{\infty}$ is a sequence of user-defined relaxation parameters.

Figure 3.13 illustrates the flow of BIP algorithms. Only the first block is assigned the iterate x_k, and then each row in the first block performs the projection using that iterate, and then sums up each row's projection. The sum of each row's projection is sent to the next block, in which that block will repeat the process and so on until the last block finishes its projections and outputs the next iterate.

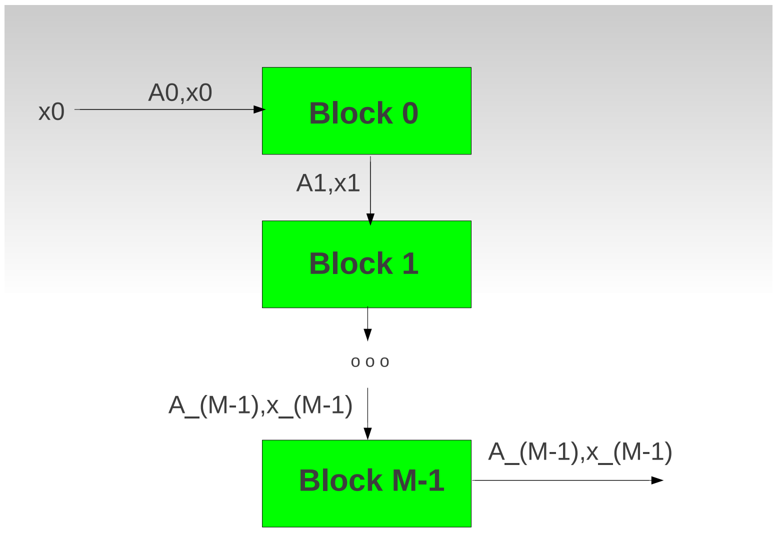


Fig. 3.13: An illustration of the flow of BIP algorithms.

3.4.1 Pre-Sorting the Matrix Partitions

As each proton history is recorded and a path is calculated, then a row of the system is formed. Therefore these rows can be "stacked" one after the other so that the matrix data naturally flows into CSR matrix format. This data structure fits efficiently with the parallel row projections that are performed within a block. Each thread can be assigned to project on the nonzeros within its row based on the starting and ending indexes of the nonzero array which are provided in the row-pointer. However, the next step within the block is to sum up all the scaled rows by taking the sum of each column. Since the column indexes are unsorted, each thread assigned to perform the sum of a column must search the entire column-index array for a matching index before adding, which is grossly inefficient. An efficient implementation of the BIP methods requires a more sophisticated sparse matrix format that combines CSR and compressed sparse column (CSC) so that the column summing operations can run efficiently. This requires pre-sorting the sections of the nonzero array defined by their block sizes with respect to the column indexes of each block's section in the array, then recording the column-sorted-order of the nonzeros, and generating a column-pointer.

Suppose the matrix is

$$\begin{bmatrix} 0 & 0 & a_0 & 0 & 0 & 0 & 0 & a_1 & a_2 \\ 0 & a_3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & a_4 & 0 & 0 & a_5 & 0 \\ 0 & 0 & 0 & a_6 & a_7 & a_8 & 0 & 0 & 0 \\ a_9 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & a_{10} & 0 & a_{11} & 0 & 0 & 0 & 0 & a_{12} \end{bmatrix}$$

and suppose block length of 3, then this matrix with 6 rows will form two blocks. The CSR format for block 0 is

$$nonZero = \begin{bmatrix} a_0 & a_1 & a_2 & a_3 & a_4 & a_5 \end{bmatrix}$$

$$rowIndex = \begin{bmatrix} 0 & 0 & 0 & 1 & 2 & 2 \end{bmatrix}$$

$$columnIndex = \begin{bmatrix} 2 & 7 & 8 & 1 & 4 & 7 \end{bmatrix}$$

$$rowPointer = \begin{bmatrix} 0 & 3 & 4 & 6 \end{bmatrix}$$

and after the column sorting, the data is arranged as follows

$$colInd = \begin{bmatrix} 1 & 2 & 4 & 7 & 7 & 8 \end{bmatrix}$$

$$rowIndex = \begin{bmatrix} 1 & 0 & 2 & 0 & 2 & 0 \end{bmatrix}$$

$$nonZero = \begin{bmatrix} a_3 & a_0 & a_4 & a_1 & a_5 & a_2 \end{bmatrix}$$

$$colInd = \begin{bmatrix} 0 & 0 & 1 & 2 & 2 & 3 & 3 & 3 & 5 & 6 \end{bmatrix}$$

$$columnSortedOrder = \begin{bmatrix} 3 & 0 & 4 & 1 & 5 & 2 \end{bmatrix}.$$

For block 1, the data in CSR form is

$$nonZero = \begin{bmatrix} a_6 & a_7 & a_8 & a_9 & a_{10} & a_{11} & a_{12} \end{bmatrix}$$

$$rowIndex = \begin{bmatrix} 3 & 3 & 3 & 4 & 5 & 5 & 5 \end{bmatrix}$$

$$columnIndex = \begin{bmatrix} 3 & 4 & 5 & 0 & 1 & 3 & 8 \end{bmatrix}$$

$$rowPointer = \begin{bmatrix} 0 & 3 & 4 & 7 \end{bmatrix}$$

and after the column sorting, the data is arranged as follows

$$colInd = \begin{bmatrix} 0 & 1 & 3 & 3 & 4 & 5 & 8 \end{bmatrix}$$

$$rowIndex = \begin{bmatrix} 4 & 5 & 3 & 5 & 3 & 3 & 5 \end{bmatrix}$$

$$nonZero = \begin{bmatrix} a_9 & a_{10} & a_6 & a_{11} & a_7 & a_8 a_{12} \end{bmatrix}$$

$$colInd = \begin{bmatrix} 0 & 1 & 1 & 3 & 5 & 6 & 7 & 7 & 7 & 8 \end{bmatrix}$$

$$columnSortedOrder = \begin{bmatrix} 9 & 10 & 6 & 11 & 7 & 8 & 12 \end{bmatrix}.$$

The implememtation of the BIP method, specifically OS-SART, on the GPGPU in this thesis uses three kernel functions. The first kernel performs the projections and writes each update to temporary memory. The next kernel uses the threads of the grid to rearrange the data into column sorted format. Then the third kernel performs the column sum update. Every block in sequence executes the three kernels until the last block finishes and writes the next x-iterate to host memory (refer to Appendix C, Listing 3.4).

3.4.2 Ordered Subsets Simultaneous Algebraic Reconstruction Technique

Figures 3.14 shows reconstructed images after the first four cycles of the NEO 1 phantom using the OS-SART algorithm with $\lambda = 0.1$, a zero vector as the intitial iterate. Figure 3.14 (a) shows the original NEO 1 phantom. Figure 3.14 (b) shows the reconstruction images with block size of 5,000. Figure 3.14 (c) shows the reconstruction images with block size of 10,000. Figure 3.14 (d) shows the reconstruction images with block size of 20,000. Figure 3.14 (e) shows the reconstruction images with block size of 40,000. Figure 3.14 (f) shows the reconstruction images with block size of 80,000. As the block size is increased beyond 10,000 as in Figures 3.14 (c) - (f), information gets leaked beyond the boundary of the outer ellipse. Therefore, given the size of this data set, block sizes beyond 10,000 will not be optimal in regards to image quality.

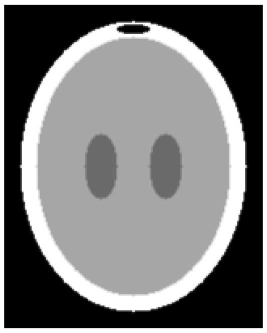
Figure 3.15 shows a relative error plot of the OS-SART algorithm, varying the block sizes while keeping the relaxation parameter constant at λ=0.1. Similar to the analysis conducted on the SAP algorithm, increasing the block size while keeping the relaxation parameter constant leads to slower convergence. For faster convergence, the relaxation parameter should be proportional to block size.

3.4.3 Execution Times Using Different Block Sizes

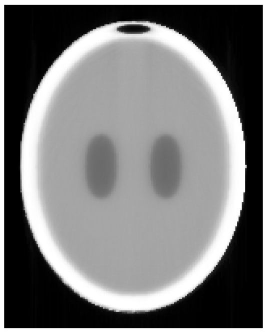
Tables 3.6 through 3.10 display the execution times for the reconstuctions of different block sizes using the Tesla M2090 GPGPU (the specifications of the Tesla M2090 GPGPU are listed in Appendix D, Listing 4.6). The Tables list the times in milli-seconds of the reconstructions after each iteration and for a different number of GPGPU thread processes assigned to the kernel functions. The Tables indicate that regardless of the block sizes used – 5,000, 10,000, 20,000, 40,000, 80,000 – the timing performance remained relatively the same. Since the GPGPU used has around 500 core processors, increasing block size to numbers of the stated amounts will not speed up performance. At block size of 5,000 the resources of the GPGPU are at full capacity.

On the furthest right column, only one thread is used. Using one thread, the GPGPU acts only as a single processor machine, and the timing results indicate the resulting poor performance. Using more threads than 500 does not improve speed up significantly because the GPGPGU has only 500 core processors.

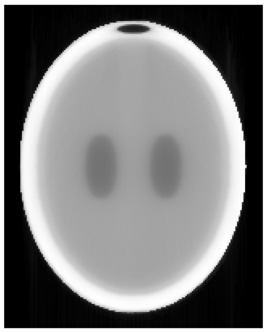
As the number of threads increases from the 1 to 1,000, the problem scales well.



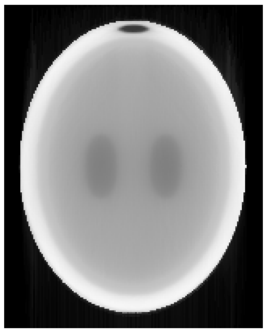
(a)



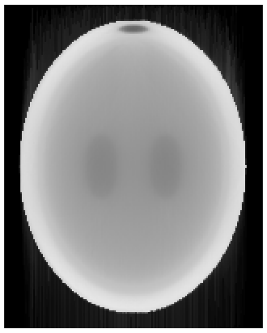
(b)



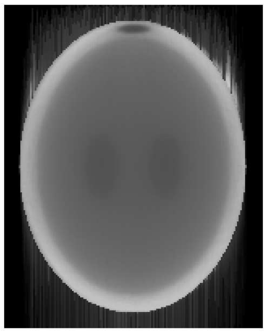
(c)



(d)



(e)



(f)

Fig. 3.14: Reconstruction images after 4 cycles of OS-SART with a constant relaxation parameter of λ=0.1: (a) original NEO 1 phantom (b) 5,000 block size (c) 10,000 block size (d) 20,000 block size (e) 40,000 block size (f) 80,000 block size.

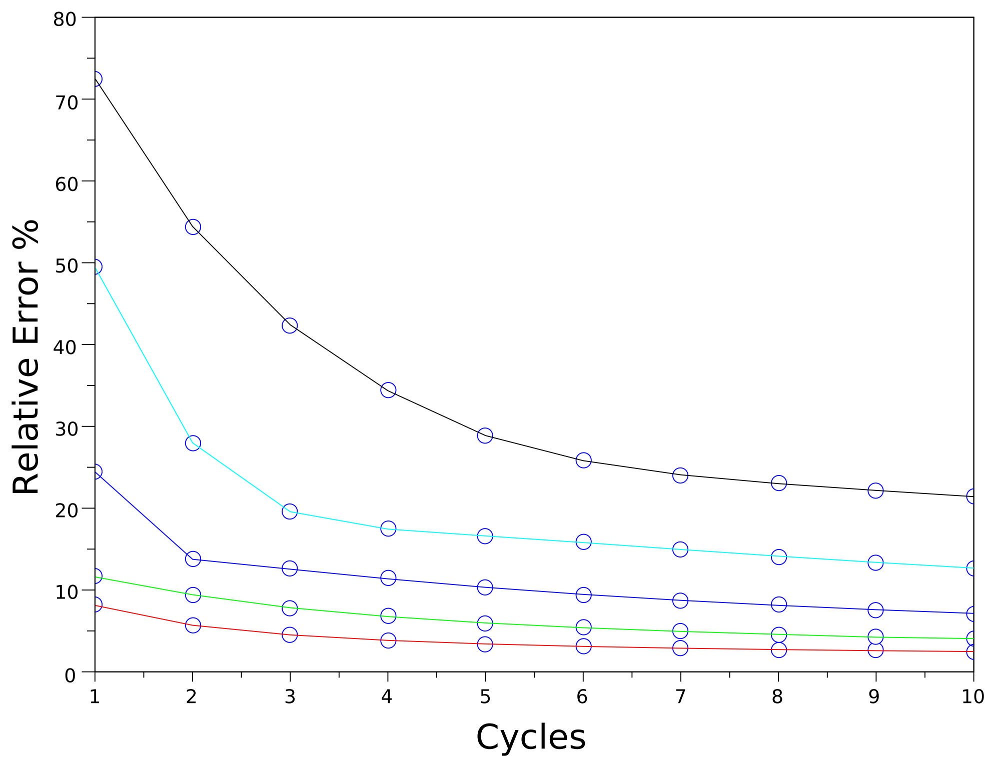


Fig. 3.15: Plot of the relative error percent for OS-SART varying the block sizes with constant λ=0.1 for cycles 1 to 10. The red line is for the 5,000 block size, the green line is for the 10,000 block size, the blue line is for the 20,000 block size, the cyan line is for the 40,000 block size, and the black line is for the 80,000 block size.

Tab. 3.6: Execution times for the OS-SART reconstructions in milli-seconds with block size of 5,000 rows for iterations 1 through 10 using a different number of GPGPU threads, t_g, in the kernel functions.

cycle	t_g=32768	t_g=1000	t_g=100	t_g=50	t_g=10	t_g=1
1	4260	4640	5420	6900	16250	47940
2	6080	6280	8480	10990	29560	90970
3	5870	7160	10190	13170	41420	132890
4	6810	7380	12080	17160	54800	175070
5	8930	8500	14140	19580	67430	217090
6	8550	10290	15990	22780	79190	259640
7	11050	10810	18140	27270	91630	301630
8	11120	11960	20260	29890	104370	343940
9	12250	12110	21850	33630	116960	385970
10	11910	13690	23970	35780	129720	429050

The problem does not scale well between the threads being increased beyond 1,000 threads. The Tables presented in this research show that the speedup is independent of the block size. The data suggests that a block size between 1 and 5,000 will be more optimal to image quality. Finding the optimal block size for timing performance requires further analysis.

3.5 Chapter Summary

This chapter described the ART algorithm in mathematical notation, also providing a visual image of how the solution x-vector projects orthogonally into a subspace of lower residual error. Next, the chapter discussed how to implement SAP on a

Tab. 3.7: Execution times for the OS-SART reconstructions in milli-seconds with block size of 10,000 rows for iterations 1 through 10 using a different number of GPGPU threads, t_g, in the kernel functions.

cycle	t_g=32768	t_g=1000	t_g=100	t_g=50	t_g=10	t_g=1
1	4280	5340	6430	6660	16690	51610
2	5400	6190	8860	10590	30530	96370
3	7130	6720	10520	13660	42900	141740
4	7020	7670	12210	17770	56570	186960
5	7840	8920	14300	21110	69370	231820
6	9860	9180	16190	23090	82350	276550
7	10680	11060	18220	26970	95740	322010
8	10310	11330	20510	29870	108810	367160
9	11890	13290	22040	34240	120320	412400
10	13420	14140	23950	37560	134680	458160

Tab. 3.8: Execution times for the OS-SART reconstructions in milli-seconds with block size of 20,000 rows for iterations 1 through 10 using a different number of GPGPU threads, t_g, in the kernel functions.

cycle	t_g=32768	t_g=1000	t_g=100	t_g=50	t_g=10	t_g=1
1	5160	5470	5870	6760	16580	53660
2	6210	6200	8110	10290	29770	101450
3	6440	7350	10050	13440	44140	149950
4	8200	7530	11930	16690	57240	197470
5	8960	9580	13760	19860	69180	245230
6	9140	10310	15560	24440	83460	293390
7	10830	11440	17520	27920	96720	341200
8	10900	11260	20110	31170	110000	388740
9	11820	13120	21830	33600	123250	437230
10	12850	14380	23950	37360	135640	484480

Tab. 3.9: Execution times for the OS-SART reconstructions in milli-seconds with block size of 40,000 rows for iterations 1 through 10 using a different number of GPGPU threads, t_g, in the kernel functions.

cycle	t_g=32768	t_g=1000	t_g=100	t_g=50	t_g=10	t_g=1
1	4590	4580	5890	6720	17420	55390
2	6280	6480	8100	10260	31150	105370
3	7210	7450	10500	14530	43660	154380
4	7190	8670	12490	16750	57710	204460
5	8200	9340	14480	20800	70750	253080
6	9190	10460	16710	23510	83730	302590
7	10830	12720	18530	26900	97160	352200
8	10750	12200	19800	30240	110460	401440
9	11460	13410	21680	34070	123750	451270
10	12450	14460	26030	36970	136770	501010

Tab. 3.10: Execution times for the OS-SART reconstructions in milli-seconds with block size of 80,000 rows for iterations 1 through 10 using a different number of GPGPU threads, t_g, in the kernel functions.

cycle	t_g=32768	t_g=1000	t_g=100	t_g=50	t_g=10	t_g=1
1	4610	4990	6670	7900	18880	55460
2	5200	6550	7700	11300	32770	105140
3	7550	7310	10770	13670	46400	155020
4	7320	8700	11990	18030	59160	204560
5	8190	8900	14760	20250	72740	254330
6	9340	10300	15750	23850	86260	303780
7	10250	10780	17850	28250	99230	352920
8	12180	11490	20920	31570	112430	402490
9	12260	13500	21970	34040	125280	452130
10	13860	14260	24920	38310	138780	501460

single GPGPU. The next section discussed how to implement OS-SART on a single GPGPU which requires an intermediary sorting step to get the data into a form that can be processed by all BIP methods. This chapter revealed that the finely tuned parameters within each algorithm, SAP or OS-SART, were more important to the reconstruction results than the actual choice between the two.

4. PARALLELIZING ART FOR GPGPU IMPLEMENTATION

4.1 Chapter Introduction

Reconstruction of pCT images on a GPGPU cluster is an ongoing study. As described earlier in this thesis, each node of the cluster has a set of GPGPUs. The cluster used in this thesis, for example, has seven nodes, with each node containing either two or three GPGPUs. To find an optimal reconstruction for clinical pCT, reconstruction on a single node must be analyzed before testing implementations on an entire cluster. However, given the different algorithms to choose from, the different types of GPGPUs, and the experimental data, the optimal choice becomes very complex. This leads to a pure parallel implementation of ART and the possibility of a future study on hybrid BIP-SAP algorithms (hybrid in this context means an intelligent combination of the two classes of algorithms). The goal of this chapter is to provide an introduction into a parallel version of ART to be implemented on a GPGPU cluster.

4.2 Parallel ART

Given x^k, the sequential ART algorithm calculates x^{k+1} as follows:

$$x^{k+1} = x^k + w_k(b_i(k) - a^{i(k)}x^k)a^{i(k)}$$

with $w_k = \lambda_k/||a^{i(k)}||^2$. Subsequently, x^{k+2} can be expressed as a function of x^{k+1}:

$$x^{k+2} = x^{k+1} + w_{k+1}(b_{i(k+1)} - a^{i(k+1)}x^{k+1})a^{i(k+1)}$$

Substituting for x^{k+1} and manipulating algebraically yields:

$$= x^{k+1} + w_{k+1}(b_{i(k+1)} - a^{i(k+1)}(x^k + w_k(b_{i(k)} - a^{i(k)}x^k)a^{i(k)}))a^{i(k+1)}$$

$$= x^{k+1} + w_{k+1}(b_{i(k+1)} - a^{i(k+1)}x^k - a^{i(k+1)}w_k(b_{i(k)} - a^{i(k)}x^k)a^{i(k)})a^{i(k+1)}$$

$$= x^{k+1} + w_{k+1}(b_{i(k+1)} - a^{i(k+1)}x^k - w_k(b_{i(k)} - a^{i(k)}x^k)a^{i(k+1)}a^{i(k)})a^{i(k+1)}$$

Suppose the rows $a^{i(k+1)}, a^{i(k)}$ are orthogonal (independent) so that $a^{i(k+1)}a^{i(k)} = 0$, then

$$x^{k+2} = x^{k+1} + w_{k+1}(b_{i(k+1)} - a^{i(k+1)}x^k)a^{i(k+1)}$$

$$= x^k + w_k(b_{i(k)} - a^{i(k)}x^k)a^{i(k)} + w_{k+1}(b_{i(k+1)} - a^{i(k+1)}x^k)a^{i(k+1)}$$

In ART, calculation of the current row projection depends on the previous row projection. However, as shown above, given the independence condition, $a^{i(k+1)}a^{i(k)} = 0$, x^{k+2} does not depend on the previous row projection, $x^{(k+1)}$, allowing for parallelism. Grouping independent rows leads to the question regarding the difference between inter-independence versus intra-independence.

4.2.1 Inter-Independent Parallel ART Partitions

Suppose there are two disjoint partitons of the A-matrix, where a partition is a set of row vectors of the pCT system. These partitions are not inter-independent from one another if both partitions have a matching column index inside each's respective set of rows for which there is a non-zero matrix element at that column index. If there is no matching column index holding a non-zero between the two partitions, then these two partitions are inter-independent from one another when performing the ART algorithm. Note that two rows are independent or orthogonal to one another if they have no matching column index holding a nonzero because their dot product is zero. Therefore, performing ART in parallel on both partitions and then adding the two update solution vectors yields the same result as the fully-sequential ART algorithm. If there are M partitions in which each partition is inter-independent from every other partition, then ART can be performed on all the partitions in parallel as well. Inter-independent parallel ART would follow the same GPGPU implementation as string-averaging algorithms without the string weights.

4.2.2 Intra-Independent Parallel ART Partitions

Suppose there is a single partition of matrix A, and for all of the rows in this partition, there is no matching column index for which a nonzero occurs, or in other words every row in the partition is orthogonal to every other row in that partition. Then, using the same starting x-vector, the ART projection can be performed on each row in parallel and then adding each row's scaled update together gives the same result as fully sequential ART. Intra-independent parallel ART would follow the same GPGPU implementation as block iterative algorithms without the block weighting. In OS-SART for instance, the dependency violations are weighted down by dividing by the column sums (A dependency violation is the term used to describe when there exists a matching column index between two rows for which there is a nonzero).

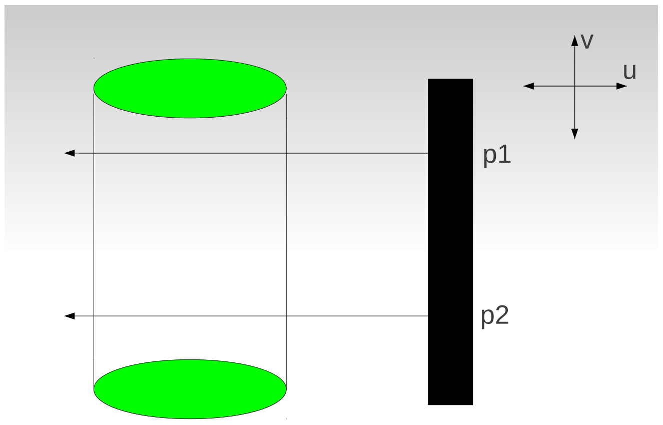


Fig. 4.1: Two proton histories with an idealized parallel path. The proton with high vertical displacement, p1, traverses the upper part of the cylinder object, the proton with low vertical displacement, p2, traverses the lower part. Note that proton histories do not intersect each other.

4.3 Assigning Proton Histories to Inter-Independent Partitions

Figure 4.1 shows two proton histories traversing a cylinder object. Cutting out proton histories with high angular deviations from the pCT system will ensure that proton histories travel nearly parallel to the u-axis, regardless of the angle along the gantry from which the proton is projected. This is because the gantry rotates around the center of the reconstruction space along the ut-plane, so that every history is near perpendicular to the v-axis. Therefore, proton histories with high intitial vertical displacement will traverse the upper part of the object, and histories with low vertical displacement will traverse the lower part. This idealized view is more complicated in reality because, due to MCS, protons will deviate from their initial path due to scattering inside the object.

4.3.1 Column Partitioning

Parallel ART and the BIP and SAP algorithms provide a framework for row partitioning to a general matrix system. However, by observing the geometry in which the specific pCT matrix system is created, column partitioning schemes may be added to the existing row partitioning algorithms. Column partitioning involves parallel processors solving for different sections of the object, thereby, each one solving for only a subset of $x \in \mathbb{R}^n$. Therefore each processor responsible for a smaller column-space. This will improve the memory complexity of the existing ART-based algorithms. Suppose there are two partitions, the first with histories with high vertical displacement, and the other with low vertical displacement. Then these partions will be inter-independent from one another. The histories from the first partition will all traverse the upper part of the object so that none of them will intersect any voxel of the lower part of the object. The histories from the other partition only intersect the lower part. Therefore, by doing the simple voxel assignment in which the lower part of the reconstruction space is assigned the lower-valued indexes, the next higher part of the reconstruction space gets the next higher sequence of voxel indexes, and so on, until the highest part of the reconstruction space receives the highest sequence of voxel indexes as in Figure 4.2. Note that the vertical v-axis describing the vertical displacement of the proton paths will always be alligned with the vertical z-axis of the stationary reconstruction space.

Using l to index the x-direction, m for the y-direction, and n for the z-direction of the reconstruction space, then the one dimensional voxel index, v, of the reconstruction space which becomes the column index within the pCT linear system is

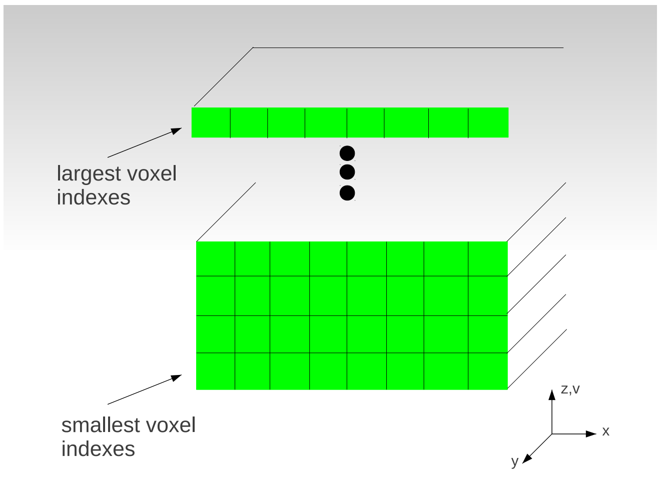


Fig. 4.2: Voxel indexing illustration

$$v(l, m, n) = np_x p_y + mp_x + l$$

where p_x is the number of partitions in the x-direction of the reconstruction space and p_y is the number of partitions in the y-direction. Given a $v(l, m, n)$, inverting to return (l, m, n) requires integer division and modular arithmetic:

$$n = v/(p_x p_y)$$

$$m = [vmod(p_x p_y)]/p_x$$

$$l = [vmod(p_x p_y)]modp_x.$$

Figure 4.3 shows proton histories assigned to partitions based on their vertical displacement. Assuming that the angular displacement in the vertical direction is zero for argument sake, then the histories can be grouped into bins according to their vertical displacement. Then by assigning the histories in the bins to the matrix

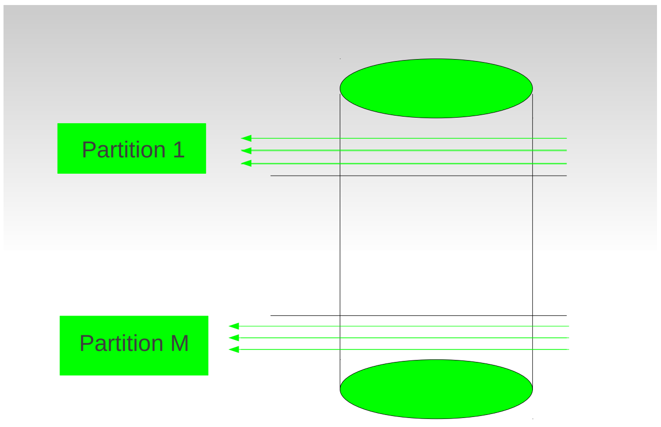


Fig. 4.3: When scattering is neglected, partitions can be assigned to sections of the object.

partitions, the partitions will be inter-independent from one another, so that they can be processed in parallel, and also, each partition of rows (proton histories) will only have nonzero column entries in the section of the object for which the bin is assigned. This methodology combines row and column partioning so that the partitions will be of a more manageable size for the nodes within the GPGPU cluster architecture to process. Each GPGPU of each node can solve for sections (vertical slices) of the object independently without communicating their individual updates until the end of the reconstruction.

4.3.2 How to Deal with Overlap

The scattering of the protons inside the object will lead to some displacement. Therefore proton histories assigned to neighboring bins may cross the bin boundary as illustrated in Figure 4.4. With proper bin height corresponding to the removal of proton histories beyond a vertical angular displacement threshold, then it can be as-

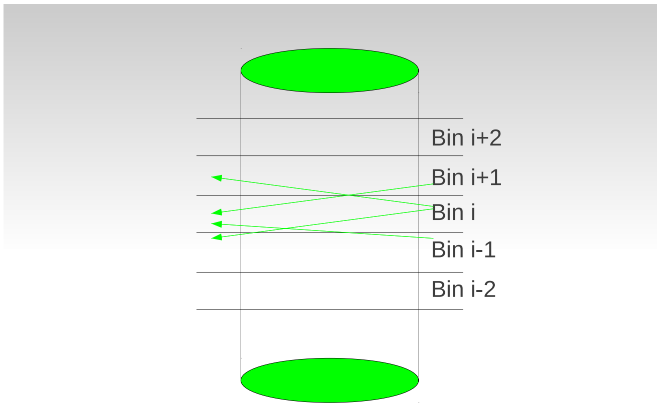


Fig. 4.4: Proton overlap between bins: Protons in bin i will overlap into bin $i-1$ and bin $i+1$ but will not overlap into bin $i-2$ or bin $i+2$.

sured that the histories in bin i overlap only into the adjacent neighboring bins, bin $i-1$ and bin $i+1$. Also, the only other histories that will overlap into bin i will be histories assigned to either bin $i-1$ or bin $i+1$. Therefore, a partition composed of histories from bin i will be inter-independent from a partition composed of histories from bin $i+2$ as well as bin $i-2$. Therefore, a shuffling technique can be used to implement inter-independent parallel ART between GPGPUs of a cluster. The shuffling technique is a two-step process with both steps illustrated in Figures 4.5 and 4.6, respectively. Given a set of GPGPUs, the first GPGPU can be assigned to the first bin partition, the second GPGPU to the third bin partition, the third GPGPU to the fifth, etc., until all GPGPUs have been assigned to the odd bin partitions. In the first step of the shuffling process, the GPGPUs all process the odd numbered bins in parallel using the initial iterate, as in Figure 4.5. Then, in the second step, shown in Figure 4.6, the GPGPUs shift their assignment to the adjacent bins so that each

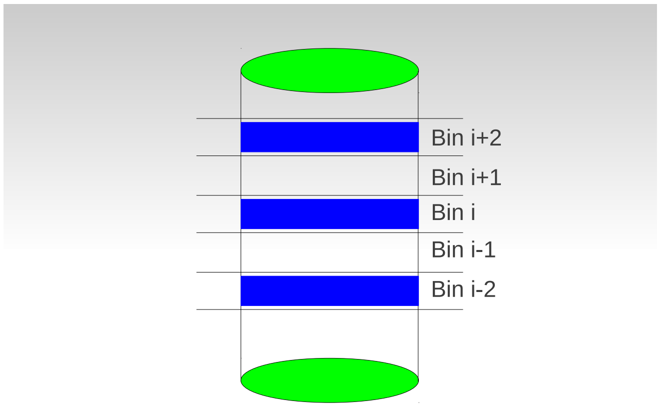


Fig. 4.5: In doing step 1 of the inter-independent parallel ART, only partitions of odd numbered bins are processed.

GPGPU processes even-numbered bin partitions in parallel, using the intermediary iterate that each GPGPU calculated during step one. The set of GPGPUs performing both steps completes one cycle of inter-independent parallel ART.

4.4 Chapter Summary

This chapter presented several of the major issues regarding the implementation of the parallel ART algorithm. The difference between inter-independent and intra-independent ART partitions was discussed and a method of assigning proton histories to partitions to allow for inter-independent parallel ART was illustrated. Column partitioning to the pCT system was introduced as well as a binning technique to deal with overlap between proton histories.

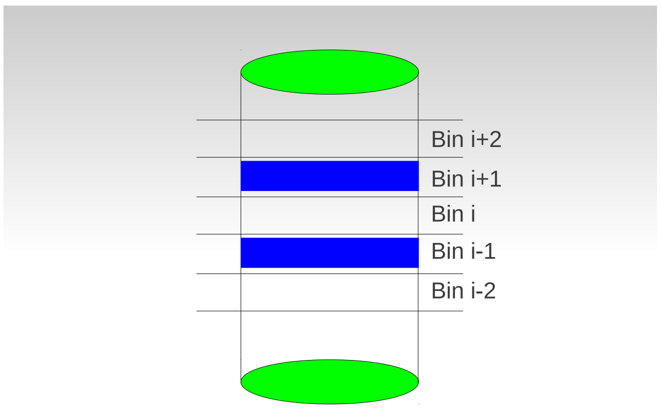


Fig. 4.6: In doing step 2 of the inter-independent parallel ART, only partitions of even numbered bins are processed.

5. SUMMARY, CONCLUSION, AND FUTURE DIRECTIONS

This thesis discussed several essential topics in pCT reconstruction. Chapter 2 described a proton simulator that was developed in this research to generate realistic pCT data sets to be used in the analysis of parallel reconstruction algorithms on GPGPU clusters. Chapter 3 provided a thorough analysis of SAP and BIP reconstructions on data generated by the proton simulator. Chapter 4 explored methods to implement parallel ART on a GPGPU cluster. What follows in this chapter are several future directions for pCT research.

5.1 The Simulator as a Tool for Future pCT Research

Fine-tuning and optimizing parameters of pCT reconstruction from pCT data that requires a rapid turn-around of pCT reconstructions with realistic data sets. The simulator presented in this work allows for pCT data sets to be created through a variety of options using digital head phantoms and transport parameters for protons. It models a clinical setting in which virtual proton beams are directed from multiple angles simulating a 360-degree virtual proton gantry. Other features of the simulator, such as the ability to add noise or to use different path options with different accuracy, will be beneficial in systematically analyzing error sources of pCT reconstruction. In the future, it is forseen that this simulator will use more complex head phantoms that provide more realistic anatomical features varying in dimension. Recently, such a realistic head phantom was created by a LLUMC PhD student, Valentina Giacometti [7].

5.2 Comparison Between Pure Parallel ART Against BIP and SAP Algorithms

The question of whether or not pure parallel ART outperforms the BIP and SAP algorithms with respect to time complexity and image quality remains uncertain. Intuitively, since pure parallel ART performs all row projections with no dependency violations between rows, parallel ART should produce higher quality images with fewer cycles than the BIP and SAP images because there is no weighting. The better performance of pure parallel ART, however, must offset the extra preprocessing necessary to create intra and/or inter independent partitions. Using the simulator to generate thousands of different data sets on a variety of combinations of phantoms and simulation parameters, and then running parallel ART algorithms against BIP and SAP algorithms will demonstrate the advantages or lack of advantages of pure parallel ART.

5.3 Sparse Matrix Visualization Patterns

In the future, recognizing patterns within the linear system may allow the improvement of clinical efficiency of pCT. Figures 5.1 and 5.2, as example, show a 2-D visualization of two different orderings of the same subset of proton histories. All histories were projected from angle 1 along the virtual gantry with linear paths through the NEO phantom. This is a subset of 1000 histories. The vertical axis represents the

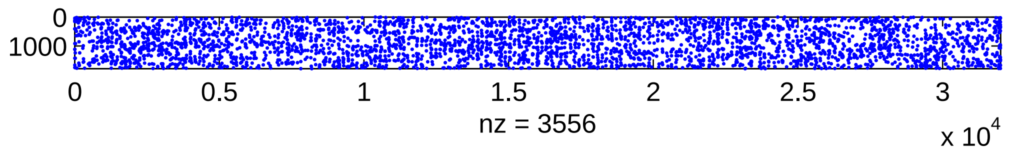


Fig. 5.1: Sparse matrix visualization image of a 2-dimensional data set generated by the proton simulator ordered by the temporal sequence of which they were created.

row index. The horizontal axis represents the location of each intersected voxel in the non-zero sparse matrix array (the data was stored in CSR format). In the entire data set, 35,560,000 voxels were intersected. For each proton history, represented by a horizontal row of the recangular area or system, there are only two blue dots plotted, one for the minimum and one for the maximum column index, respectively. This creates a picture that the human eye can interpret without misrepresenting the data while displaying the intended concept.

In Figure 5.1, the histories are ordered by the temporal sequence of which they were created in the simulator, whereas in Figure 5.2, the histories have been sorted by their lateral displacement along the t-axis in the ut-plane. The image that uses sequential ordering demonstrates a completely chaotic system in which the blue dots (representing the non-zero elements of the matrix) are randomly scattered throughout the matrix. The image resulting from sorting by lateral displacement shows a linear and diagonal structure to the matrix that can be exploited for faster computation

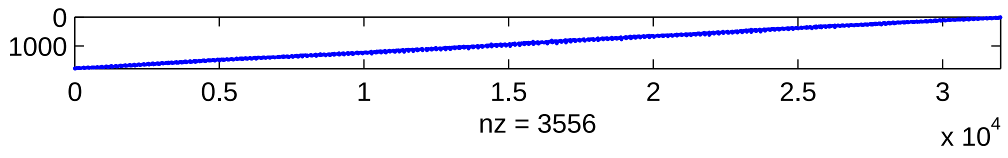


Fig. 5.2: Sparse matrix visualization image of a 2-dimensional data set generated by the proton simulator ordered by lateral displacement along the t-axis of the ut-plane.

and improved storage, though the cost of sorting must be taken into consideration. Exploiting the sparse linear patterns of the system could be beneficial the ideas of parallel ART.

5.4 Matrix Partioning across a GPGPU Cluster

The data sets produced by the simulator are written to disk memory in a format that can be read into data structures that will later be used in the implementation of new and existing parallel projection algorithms across GPGPU clusters. Figure 5.3 shows the conceptual design of a typical GPGPU cluster as a collection of nodes, with each node comprised of multiple GPGPUs. In the Figure, there are M total nodes connected by a bus within the single cluster, and each node connects to a set of GPGPUs.

A further step is to develop a general *A*-matrix partitioning scheme that will assign these partitions to nodes within a cluster and to GPGPUs within a node matching

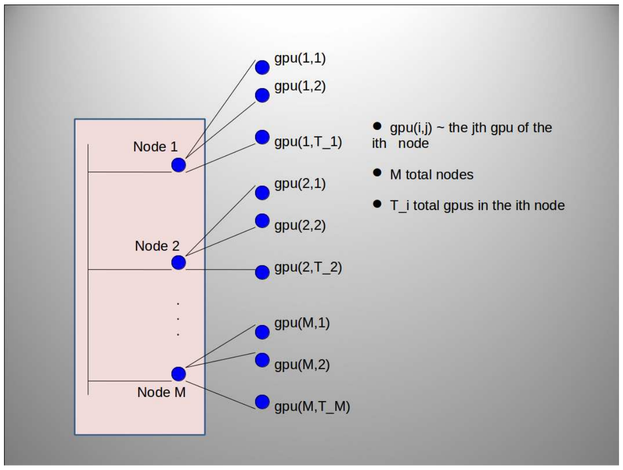


Fig. 5.3: Conceptual design of a GPGPU cluster with M nodes. Each node has Ti GPUs.

the structure inherent in the spatial and temporal acquisition scheme of the pCT system and the physical nature of the data to the internal architecture of the GPGPU cluster efficiently. This will allow rendering as many GPGPUs active as possible and minimizing the need of data transfer between different nodes. Figure 5.4 illustrates a general assignment scheme of matrix partitions to GPGPUs in a GPGPU cluster. The matrix A is partitioned into M sub-matrices, and then each of the M sub-matrices is further partitioned into sub-partitions. The expected outcome of future research is a GPGPU-based reconstruction scheme that optimizes pCT reconstruction with respect to image quality, reconstruction time, and hardware expenses.

5.5 Final Conclusion

To further advance science, the pCT project is an ongoing collaborative effort involving several universities and a major proton treatment center (LLUMC). The pCT simulator, including source code and documentation, will be made readily available

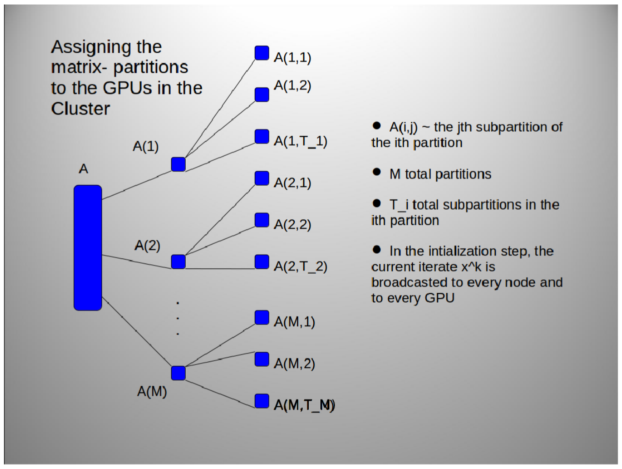

Assigning the matrix- partitions to the GPUs in the Cluster
A
A(1)
A(2)
A(M)
A(1,1)
A(1,2)
A(1,T_1)
A(2,1)
A(2,2)
A(2,T_2)
A(M,1)
A(M,2)
A(M,T_M)
A(i,j) ~ the jth subpartition of the ith partition
M total partitions
T_i total subpartitions in the ith partition
In the intialization step, the current iterate x^k is broadcasted to every node and to every GPU

Fig. 5.4: General concept of matrix parititioning and assignment to GPGPUs in a GPGPU cluster.

to other researchers (physicists and engineers) interested in the development of pCT. The majority of the simulator and reconstruction codes can be found in Appendices B and C of this thesis. At this point, the GPGPU reconstruction code has been written in CUDA, all simulation code has been written in C++, the graphics package utilizes the OpenGL libraries, and all data analysis code has been written in C++, MatLab, and SciLab. A reconstruction using the ART algorithm has yet to be implemented on the data generated by the simulator developed in this research. When this task were accomplished, then experts will be able to make better judgement on whether or not a pure parallel version of ART should be pursued. Scientific and mathematical intuition leads to the conclusion that the ART algorithm will outperform the BIP and SAP algorithms with respect to image quality though no empirical evidence actually supports this premise. It is important to take into accoutn the added sorting costs to make the data suitable for pure parallel ART. Also, the implementation of either a hybrid SAP-BIP algorithm or a pure parallel ART across a GPGPU cluster is an important future goal. It is likely that the visualization of sparse pCT matrix patterns will lead to better utilization of sorting the pCT system matrix.

APPENDIX A

RANDOM NUMBER GENERATOR

The lateral displacement of the entry of a proton into an object along the t-axis is drawn from a uniform distribution on the closed interval [-125,125] and the entry angle is equal to the angle of the proton beam. The lateral and angular displacement for the exit of a proton follow a joint normal distribution. The program will use its own joint normal random number generator to obtain values for the lateral and angular displacement for the exit of a proton. The random numbers are generated by the following process:

Let $U = \begin{bmatrix} u \\ v \end{bmatrix}$, u, v follow N(0,1), let $A \in \mathbb{R}^{2x2}$ be symmetric, then to obtain a joint normal random variable pair $X = \begin{bmatrix} x \\ y \end{bmatrix}$

$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} u \\ v \end{bmatrix} \text{ with } x, y \text{ following } N\Big(0, \begin{bmatrix} \sigma_1^2 & \sigma_{12} \\ \sigma_{12} & \sigma_2^2 \end{bmatrix}\Big)$$

Therefore set $E\Big[\begin{bmatrix} x \\ y \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}^T\Big] = \begin{bmatrix} \sigma_1^2 & \sigma_{12} \\ \sigma_{12} & \sigma_2^2 \end{bmatrix}$

$\Leftrightarrow$

$$\begin{bmatrix} E[xx] & E[xy] \\ E[xy] & E[yy] \end{bmatrix} = \begin{bmatrix} \sigma_1^2 & \sigma_{12} \\ \sigma_{12} & \sigma_2^2 \end{bmatrix}$$

$$E[xx] = E\Big[\begin{bmatrix} u & v \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} \begin{bmatrix} a & b \end{bmatrix} \begin{bmatrix} u \\ v \end{bmatrix}\Big]$$

$= E\big[(au+bv)^2\big]$

$= E\big[a^2u^2 + 2abuv + b^2v^2\big]$

$= a^2E[u^2] + 2abE[uv] + b^2E[v^2]$, $E[u^2] = 1 = E[v^2], E[uv] = 0$ because u, v are independent. And so

$E[xx] = a^2 + b^2 = \sigma_1^2$, (1)

Similarly $E[yy] = b^2 + c^2 = \sigma_2^2$, (2)

$E[xy] = E\big[\begin{bmatrix} u & v \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} \begin{bmatrix} b & c \end{bmatrix} \begin{bmatrix} u \\ v \end{bmatrix} \big]$

$= E\big[(au+bv)(bu+cv)\big] = E\big[abu^2 + b^2uv + acuv + bcv^2\big]$

$abE[u^2] + b^2E[uv] + acE[uv] + bcE[v^2]$

Therefore

$E[xy] = ab + bc = \sigma_{12}$ (3)

Equations (1),(2),(3) form a system of 3 nonlinear equations of three variables each, let $x = (a, b, c)$ so that

$$F(x) = \begin{bmatrix} F_1 \\ F_2 \\ F_3 \end{bmatrix} = \begin{bmatrix} a^2 + b^2 - \sigma_1^2 = 0 \\ b^2 + c^2 - \sigma_2^2 = 0 \\ ab + bc - \sigma_{12} = 0 \end{bmatrix}$$

Newton's nonlinear iterative method will be applied to solve for $x = (a, b, c)$

```
$x_0 =$ initial guess
for $k = 0, 1, 2, ...$
solve $J_f(x_k)s_k = -F(x_k)$ for $s_k$
$x_{k+1} = x_k + s_k$
end
```

The Jacobian of F,

$$J_F(x) = \begin{bmatrix} \frac{\partial F_1}{\partial a} & \frac{\partial F_1}{\partial b} & \frac{\partial F_1}{\partial c} \\ \frac{\partial F_2}{\partial a} & \frac{\partial F_2}{\partial b} & \frac{\partial F_2}{\partial c} \\ \frac{\partial F_3}{\partial a} & \frac{\partial F_3}{\partial b} & \frac{\partial F_3}{\partial c} \end{bmatrix} = \begin{bmatrix} 2a & 2b & 0 \\ 0 & 2b & 2c \\ b & a+c & b \end{bmatrix}.$$

Setting $Js = F$ results in a 3x3 linear system easily solvable using Cramer's method.

To generate the pair of standard normal random variables, this program generates 2 uniform random variables U, V in the interval [-1,1] and then uses the Marsaglia Polar Method (modification of Box-Muller Method) to convert the uniform random variables U, V into the standard normal random variables Z_1, Z_2:

$Z_1 = U\sqrt{\frac{-2\ln s}{s}}$

$Z_2 = V\sqrt{\frac{-2\ln s}{s}}$

$s = U^2 + V^2$

if $s = 0$ or $s \geq 1$ then start over.

The variances of the lateral and angular displacement, σ_1, σ_2, and the covariance, σ_{12}, are functions of the exit depth of the proton, and these values will be drawn from a look-up table, precalculated for water using Highland's scattering formalism [1] with parameters recommended by the Particle Data Group [2].

APPENDIX B

PROTON SIMULATOR CODE

Listing 2.1: neoCode

```
/**********************PHANTOM****************************/


class Ellipse {
 public:
  Ellipse(float xxc,float yyc,float aa,float bb) :
    xc(xxc),yc(yyc),a(aa),b(bb) {};
  float xc;
  float yc;
  float a;
  float b;
};
// NonhomogeniousEllipseObject phantom:
// a vector of ellipses for a convex object;
// the last element of v must be the boundary ellipse;
// if an ellipse is inside another ellipse than the
// smaller one (on the inside) must come before the
// bigger one in the E vector
class neo {
public:
  neo(vector<Ellipse> e,vector<float> r) : E(e),rsp(r) {};
  vector<Ellipse> E;
  vector<float> rsp;
```

```
};
neo CreateDefaultNeo(float s) {
  vector<Ellipse> E;
  vector<float> rsp;
  Ellipse e1(s*0,s*0,s*70,s*90);
  Ellipse e2(s*0,s*85,s*10,s*2.5);
  Ellipse e3(s*0,s*0,s*60,s*80);
  Ellipse e4(s*20,s*0,s*10,s*20);
  Ellipse e5(s*-20,s*0,s*10,s*20);
  E.push_back(e5);
  E.push_back(e4);
  E.push_back(e3);
  E.push_back(e2);
  E.push_back(e1);
  rsp.push_back(0.9); // ventricles
  rsp.push_back(0.9); // ventricles
  rsp.push_back(1.04); // brain tissue
  rsp.push_back(0.0); // air pocket
  rsp.push_back(1.6); // skull
  return neo(E,rsp);
}
class LPoint {
 public:
  LPoint() {
    x=0;
    y=0;
    l=0;
  }
  LPoint(float xx,float yy,int ll) :
    x(xx),y(yy),l(ll) {};
  float x;
  float y;
  int l;
};
float calculateTriangleArea(LPoint A,LPoint B,
 LPoint C) {
  float output=fabs((A.x*(B.y-C.y)+B.x*(C.y-A.y)
   +C.x*(A.y-B.y))/2.0);
  return output;
}
bool equalf(float a,float b) {
  float tol=0.00001;
  if (a<b+tol && a>b-tol)
    return true;
  else
    return false;
```

```
}
bool IsPointInEllipse(float xc,float yc,float a,float b,
 float k,float h) {
  float yup=h+b*sqrt(1.0-pow(xc-k,2)/pow(a,2));
  float ylo=h-b*sqrt(1.0-pow(xc-k,2)/pow(a,2));
  if (yc<=yup && yc>=ylo)
    return true;
  else
    return false;
}
float FindRSPatPoint(float x,float y,neo phant) {
  float RSP=0;
  float xcPhan,ycPhan,aPhan,bPhan;
  for (int i=0; i<phant.E.size(); i++) {
    xcPhan=phant.E[i].xc;
    ycPhan=phant.E[i].yc;
    aPhan=phant.E[i].a;
    bPhan=phant.E[i].b;
    if (IsPointInEllipse(x,y,aPhan,bPhan,xcPhan,ycPhan)) {
      RSP=phant.rsp[i];
      return RSP;
    }
  }
  return RSP;
}
float FindRSPatPoint_DefaultNeo(float xc,float yc) {
  float RSP;
  if (IsPointInEllipse(xc,yc,10,20,-20,0) ||
   IsPointInEllipse(xc,yc,10,20,20,0))
    RSP=0.9;
  else if (IsPointInEllipse(xc,yc,60,80,0,0)) {
    RSP=1.04;
  }
  else if (IsPointInEllipse(xc,yc,10,2.5,0,85))
    RSP=0;
  else if (IsPointInEllipse(xc,yc,70,90,0,0))
    RSP=1.6;
  else
    RSP=0;
  return RSP;
}
// method==0 --> centerPointMethod      vs
// method==1 --> Corner Point Averaging
float* CreateNeoSlice(neo phantom,int ypartitions,
 int xpartitions,float boxLength,float boxWidth,
 int method) {
```

```
int sizeXtrue=xpartitions*ypartitions*sizeof(float);
float *Xtrue=(float*)malloc(sizeXtrue);
float yshift=boxLength/2.0;
float xshift=boxWidth/2.0;
float ystep=boxLength/ypartitions;
float xstep=boxWidth/xpartitions;
float x0,x1,x2,x3,x4,y0,y1,y2,y3,y4;
float xc,yc,RSP,temp0,temp1,temp2,temp3,temp4;
float xdot1l0,xdot2l0,xdot1l2,xdot2l2,
      ydot1l1,ydot2l1,ydot1l3,ydot2l3;
float base,rbase,height,rheight,area1,area2;
float a,b;
float voxelArea=xstep*ystep;
int counter1=0;
if (method==0) { //centerPointMethod
  for (int i=0; i<ypartitions; i++) {
    for (int j=0; j<xpartitions; j++) {
      x1=j*xstep-xshift;
      x2=x1+xstep;
      y1=yshift-i*ystep;
      y2=y1-ystep;
      xc=(x1+x2)/2.0;
      yc=(y1+y2)/2.0;
```

```
        RSP=FindRSPatPoint(xc,yc,phantom);
        Xtrue[i*xpartitions+j]=RSP;
      }
    }
  }
  else if (method==1) { //corner point averaging method
    for (int i=0; i<ypartitions; i++) {
      for (int j=0; j<xpartitions; j++) {
        x1=j*xstep-xshift;
        y1=yshift-i*ystep;
        x2=x1+xstep;
        y2=y1;
        x3=x2;
        y3=y1-ystep;
        x4=x1;
        y4=y3;
        temp1=FindRSPatPoint(x1,y1,phantom);
        temp2=FindRSPatPoint(x2,y2,phantom);
        temp3=FindRSPatPoint(x3,y3,phantom);
        temp4=FindRSPatPoint(x4,y4,phantom);
        RSP=(temp1+temp2+temp3+temp4)/4.0;
        Xtrue[i*xpartitions+j]=RSP;
      }
    }
  }
  else {
    for (int i=0; i<ypartitions; i++) {
      for (int j=0; j<xpartitions; j++) {
        cout<< i << "␣" << j << endl;
        x0=j*xstep-xshift;
        y0=yshift-i*ystep;
        x1=x0+xstep;
        y1=y0;
        x2=x1;
        y2=y0-ystep;
        x3=x0;
        y3=y2;
        temp0=FindRSPatPoint(x0,y0,phantom);
        temp1=FindRSPatPoint(x1,y1,phantom);
        temp2=FindRSPatPoint(x2,y2,phantom);
        temp3=FindRSPatPoint(x3,y3,phantom);
        if (equalf(temp0,temp1) && equalf(temp0,temp2)
         && equalf(temp0,temp3)) {
          RSP=temp0;
          Xtrue[i*xpartitions+j]=RSP;
        }
        else {
          vector<LPoint> iP; //P is a temporary vector
          // to hold the intersection points of the
          // ellipses and the voxels.  It should be
          // of size 2, unless more than 1 ellipse
          // intersect a single voxel, however we'll
          // keep the voxel size small enough to avoid
          // this.  So if the size is not 2, use center
          // point method
          for (int k=0; k<phantom.E.size(); k++) {
            xc=phantom.E[k].xc;
            yc=phantom.E[k].yc;
            a=phantom.E[k].a;
            b=phantom.E[k].b;
            xdot1l0=xc+a*sqrt(1-pow(y0-yc,2)/pow(b,2));
            xdot2l0=xc-a*sqrt(1-pow(y0-yc,2)/pow(b,2));
            xdot1l2=xc+a*sqrt(1-pow(y2-yc,2)/pow(b,2));
            xdot2l2=xc-a*sqrt(1-pow(y2-yc,2)/pow(b,2));
            ydot1l1=yc+b*sqrt(1-pow(x1-xc,2)/pow(a,2));
            ydot2l1=xc-b*sqrt(1-pow(x1-xc,2)/pow(a,2));
            ydot1l3=xc+b*sqrt(1-pow(x3-xc,2)/pow(a,2));
            ydot2l3=xc-b*sqrt(1-pow(x3-xc,2)/pow(a,2));
            if (xdot1l0<x1 && xdot1l0>x0 &&
             1-pow(y0-yc,2)/pow(b,2)>0)
              iP.push_back(LPoint(xdot1l0,y0,0));
            if (xdot2l0<x1 && xdot2l0>x0 &&
             1-pow(y0-yc,2)/pow(b,2)>0)
              iP.push_back(LPoint(xdot2l0,y0,0));
            if (xdot1l2<x1 && xdot1l2>x0 &&
             1-pow(y2-yc,2)/pow(b,2)>0)
              iP.push_back(LPoint(xdot1l2,y2,2));
            if (xdot2l2<x1 && xdot2l2>x0 &&
             1-pow(y2-yc,2)/pow(b,2)>0)
              iP.push_back(LPoint(xdot2l2,y2,2));
            if (ydot1l1<y0 && ydot1l1>y3 &&
             1-pow(x1-xc,2)/pow(a,2)>0)
              iP.push_back(LPoint(x1,ydot1l1,1));
            if (ydot2l1<y0 && ydot2l1>y3 &&
             1-pow(x1-xc,2)/pow(a,2)>0)
              iP.push_back(LPoint(x1,ydot2l1,1));
            if (ydot1l3<y0 && ydot1l3>y3 &&
             1-pow(x3-xc,2)/pow(a,2)>0)
              iP.push_back(LPoint(x0,ydot1l3,3));
            if (ydot2l3<y0 && ydot2l3>y3 &&
             1-pow(x3-xc,2)/pow(a,2)>0)
              iP.push_back(LPoint(x0,ydot2l3,3));
```

```
    }
    if (iP.size()!=2) { //center-point-method
      xc=(x0+x1)/(float)2;
      yc=(y0+y3)/(float)2;
      RSP=FindRSPatPoint(xc,yc,phantom);
      Xtrue[i*xpartitions+j]=RSP;
    }
    else {
      vector<LPoint> cP; //corner points
      cP.push_back(LPoint(x0,y0,-1));
      cP.push_back(LPoint(x1,y1,-1));
      cP.push_back(LPoint(x2,y2,-1));
      cP.push_back(LPoint(x3,y3,-1));
      vector<LPoint> sP; //shape points
      if (iP[0].l>iP[1].l) {
        LPoint tempLPoint=iP[0];
        iP[0]=iP[1];
        iP[1]=tempLPoint;
      }
      int la=iP[0].l;
      int lb=iP[1].l;
      sP.push_back(iP[0]);
      for (int ind=la+1; ind<=lb; ind++)
    sP.push_back(cP[ind]);
  sP.push_back(iP[1]);
  if (sP.size()<3) { //center-point-method
    xc=(x0+x1)/(float)2;
    yc=(y0+y3)/(float)2;
    RSP=FindRSPatPoint(xc,yc,phantom);
    Xtrue[i*xpartitions+j]=RSP;
    counter1++;
  }
  else {
    float area1=0;
    int tempIndex=1;
    for (int s=0; s<sP.size()-2; s++) {
      LPoint tempA=sP[0];
      LPoint tempB=sP[tempIndex];
      LPoint tempC=sP[tempIndex+1];
      area1+=calculateTriangleArea(tempA,tempB,
       tempC);
      tempIndex++;
    }
    area2=voxelArea-area1;
    RSP=(1.0/voxelArea)*(area1*temp2+area2*temp4);
    Xtrue[i*xpartitions+j]=RSP;
```

```
              }
            }
          }
        }
      }
  }
  return Xtrue;
}
float* CreateXtrueNeo3d(neo MyNeo,int zparts,int yparts,
 int xparts,float boxLength,float boxWidth,int method) {
  float* Neo2d=CreateNeoSlice(MyNeo,yparts,xparts,
   boxLength,boxWidth,method);
  float* XtrueNeo3d=
   (float*)malloc(zparts*yparts*xparts*sizeof(float));
  for (int k=0; k<zparts; k++) {
    for (int ii=0; ii<yparts*xparts; ii++) {
      XtrueNeo3d[k*yparts*xparts+ii]=Neo2d[ii];
    }
  }
  free(Neo2d);
  return XtrueNeo3d;
}
```

Listing 2.2: pathCode

```
// proton history object, constant chord length
class History2d {
public:
  History2d(vector<int> c,float w)
   : colInd(c),wepl(w) {};
  vector<int> colInd;
  float wepl;
};

// KnownHull indicates that we intersected the proton path
// with the ellipse boundary of the neo phantom
History2d* generateProtonPath2d_KnownHull(float theta,
 float* Xtrue,int yparts,int xparts,float boxLength,
 float boxWidth,neo MyNeo,int pathOption,int sliceIndex) {
  int startCol=sliceIndex*yparts*xparts;
  vector<int> colInd,nullvec;
  float wepl=0;
  int col,localCol;
  PointVec Pin,Pt,Pout;
  float thetaOut;
  float a=MyNeo.E[MyNeo.E.size()-1].a; // semi major xaxis
  //length of outer ellipse
```

```
float b=MyNeo.E[MyNeo.E.size()-1].b; // semi minor yaxis...
float ystep=boxLength/(float)yparts;
float xstep=boxWidth/(float)xparts;
// the shifts to center the recon space
float yshift=boxLength/2;
float xshift=boxWidth/2;
float gantryRad=3000;
vector<PointVec3> scatParam=scatteringParameters();
PointVec P1(gantryRad*cos(theta),gantryRad*sin(theta));
PointVec v1(cos(theta),sin(theta));
PointVec v2(-sin(theta),cos(theta));
float varLat,covariance,varAng;
float d1=MyUniformRand();
//calculate intersection of proton entry line with ellipse
PointVec P3(P1.x+d1*v2.x,P1.y+d1*v2.y);
float m=v1.y/v1.x; //slope of proton entry line
float B=P3.y-m*P3.x; //y-intercept of proton entry line
//to calculate the entry point Pin:
//the intersection test of line and ellipse --> solve for x
//in the quadratic equation:  alpha*x^2+beta*x+gamma=0
float alpha=1/pow(a,2)+pow(m/b,2);
float beta=2*m*B/pow(b,2);
float gamma=pow(B/b,2)-1;
float disc=pow(beta,2)-4*alpha*gamma; //discriminant
if ( disc<0 or equalf(disc,0) ) {
  // proton path missed the object -> return blank history
  History2d* T=new History2d(nullvec,0);
  return T;
}
else {
  float x1=(-beta+sqrt(disc))/(2*alpha);
  float x2=(-beta-sqrt(disc))/(2*alpha);
  float y1=m*x1+B;
  float y2=m*x2+B;
  float y11=b*sqrt(1-pow(x1/a,2));
  float y22=-b*sqrt(1-pow(x2/a,2));
  float norm1=normEu(PointVec(x1,y1),P3);
  float norm2=normEu(PointVec(x2,y2),P3);
  if (norm1<norm2) {
    Pin.set(x1,y1);
    Pt.set(x2,y2);
  }
  else {
    Pin.set(x2,y2);
    Pt.set(x1,y1);
  }
```

```
//depth is the distance proton traveled through object
//assuming it moves in a straight line
float depth=normEu(Pin,Pt);
//the lookup table is indexed by cm so divide by 10
//to convert mm to cm
int index=ceil(depth/10);
if (index>20) index=20; //avoid seg faults
varLat=scatParam[index].x;
covariance=scatParam[index].y;
varAng=scatParam[index].z;
PointVec R2=generateJointRandNorm(varLat,covariance,
 varAng);
float d2=R2.x; //exiting lateral displacement
float psi=R2.y; //exiting angular displacement
//generate temporary line to intersect with ellipse to
//find Pout
PointVec P4(Pt.x+d2*v2.x,Pt.y+d2*v2.y);
float m2=m; //slope
float B2=P4.y-m2*P4.x; //intercept
//line-ellipse interesection quadratic coefficients
float alpha2=1/pow(a,2)+pow(m2/b,2);
float beta2=2*m2*B2/pow(b,2);
float gamma2=pow(B2/b,2)-1;
float disc2=pow(beta2,2)-4*alpha2*gamma2; //discriminant
if (disc2<0) { // very rare circumstance
  History2d* T=new History2d(nullvec,0);
  return T;
}
else if (equalf(disc2,0)) {
  float x21=-beta2/(2*alpha2);
  float y21=m2*x21+B2;
  Pout.set(x21,y21);
}
else {
  float x21=(-beta2+sqrt(disc2))/(2*alpha2);
  float x22=(-beta2-sqrt(disc2))/(2*alpha2);
  float y21=m2*x21+B2;
  float y22=m2*x22+B2;
  float norm21=normEu(PointVec(x21,y21),P4);
  float norm22=normEu(PointVec(x22,y22),P4);
  if (norm21<norm22)
    Pout.set(x21,y21);
  else
    Pout.set(x22,y22);
}
//exiting angle relative to xy axis
```

```
thetaOut=theta+psi;
//calculate upper left corner coordinates of the entry
//voxel and exit voxel to find xmin and ymin
float xin0=CalculateX0(Pin.x,xstep);
float xout0=CalculateX0(Pout.x,xstep);
float xmin=min(xin0,xout0);
float xmax=max(xin0,xout0);
if (pathOption==1) {//straight-line between Pin and Pout
  float ms=(Pout.y-Pin.y)/(Pout.x-Pin.x); //slope
  float Bs=Pout.y-ms*Pout.x; //y-intercept
  for (float xx=xmin; xx<xmax; xx+=xstep) {
    float ys=ms*xx+Bs;
    float y0=CalculateY0(ys,ystep);
    int i=(int)((yshift-y0)/ystep);
    int j=(int)((xshift+xx)/xstep);
    //if the slope of the line is greater than 1 in abs
    //val then the line will intersect more voxels above
    //i (if positive slope) before the next xx so need
    //to mark all those voxels by calculating y at the
    //next xx
    float ysnext=ms*(xx+xstep)+Bs;
    float y0next=CalculateY0(ysnext,ystep);
    int inext=(int)((yshift-y0next)/ystep);
```

```
        if (i>=0 and inext>=0) {
          for (int ii=min(i,inext); ii<=max(i,inext); ii++) {
            localCol=ii*xparts+j;
            if (0<localCol and localCol<xparts*yparts) {
              col=startCol+localCol;
              colInd.push_back(col);
              wepl+=Xtrue[localCol];
            }
          }
        }
      }
    }
    else if (pathOption=2) { //cubic-spline
      float thetaIn=theta;
      float c=thetaIn*(Pout.x-Pin.x)-(Pout.y-Pin.x);
      float d=-thetaOut*(Pout.x-Pin.x)-(Pout.y-Pin.y);
      for (float xx=xmin; xx<xmax; xx+=xstep) {
        float t=(xx-Pin.x)/(Pout.x-Pin.x);
        float q=(1-t)*Pin.y+t*Pout.y+t*(1-t)*(c*(1-t)+d*t);
        float y0=CalculateY0(q,ystep);
        int i=(int)((yshift-y0)/ystep);
        int j=(int)((xshift+xx)/xstep);
        float tnext=(xx+xstep-Pin.x)/(Pout.x-Pin.x);
        float qnext=(1-tnext)*Pin.y+tnext*Pout.y+tnext*
         (1-tnext)*(c*(1-tnext)+d*tnext);
        float y0next=CalculateY0(qnext,ystep);
        int inext=(int)((yshift-y0next)/ystep);
        if (i>=0 and inext>=0) {
          for (int ii=min(i,inext); ii<=max(i,inext); ii++) {
            localCol=ii*xparts+j;
            if (0<=localCol and localCol<xparts*yparts) {
              col=startCol+localCol;
              colInd.push_back(col);
              wepl+=Xtrue[localCol];
            }
          }
        }
      }
    }
  }
  History2d* T=new History2d(colInd,wepl);
  return T;
}
```

Listing 2.3: bivGenCode

```
// each PointVec3  (varLat, cov, VarAng)
```

```
vector<PointVec3> scatteringParameters() {
  vector<PointVec3> output;
  output.push_back(PointVec3(0,0,0));
  output.push_back(PointVec3(0.00112,0.0001686,
   3.397*pow(10,-5)));
  output.push_back(PointVec3(0.009335,0.0007052,
   7.154*pow(10,-5)));
  output.push_back(PointVec3(0.0324,0.001638,0.0001117));
  output.push_back(PointVec3(0.07861,0.002992,0.0001542));
  output.push_back(PointVec3(0.1567,0.004793,0.0001994));
  output.push_back(PointVec3(0.2761,0.007067,0.0002472));
  output.push_back(PointVec3(0.4466,0.009843,0.0002979));
  output.push_back(PointVec3(0.6786,0.01315,0.0003519));
  output.push_back(PointVec3(0.9833,0.01703,0.0004094));
  output.push_back(PointVec3(1.372,0.0215,0.0004709));
  output.push_back(PointVec3(1.859,0.02663,0.0005368));
  output.push_back(PointVec3(2.456,0.03245,0.0006078));
  output.push_back(PointVec3(3.178,0.03902,0.0006847));
  output.push_back(PointVec3(4.041,0.0464,0.0007683));
  output.push_back(PointVec3(5.063,0.05467,0.0008599));
  output.push_back(PointVec3(6.261,0.06392,0.0009611));
  output.push_back(PointVec3(7.658,0.07425,0.001074));
  output.push_back(PointVec3(9.275,0.08579,0.001201));
  output.push_back(PointVec3(11.14,0.09871,0.001347));
  output.push_back(PointVec3(13.28,0.1132,0.001518));
  return output;
}

// return uniform random number from [0,1]
float ranf() {
  return rand()%1000001/(float)1000000;
}

// return uniform rv from [-125,125]
float MyUniformRand() {
  float rv=rand()%1000001/(float)1000000;
  rv=rv*250.0; //scale rv to be in [0,250]
  rv=rv-125.0; //shift rv to be in [-125,125]
  return rv;
}

// Marsaglia Polar Method ~ modified from Box-Muller
// to take 2 uniform rvs and transform them into
// 2 independent standard normal rvs
PointVec generate2RandStdNorm() {
  float u,v,x,y,s,t;
  s=1.0;
  while (s>=1.0 or equalf(s,0)) {
    u=2.0*ranf()-1.0;
    v=2.0*ranf()-1.0;
    s=u*u+v*v;
  }
  t=sqrt((-2.0*log(s))/s);
  x=u*t;
  y=v*t;
  return PointVec(x,y);
}

// hard coded calculation of determinant
float Determinant3x3(float* A) {
  float d=A[0]*(A[4]*A[8]+A[5]*A[7])
         -A[1]*(A[3]*A[8]+A[5]*A[6])
         +A[2]*(A[3]*A[7]+A[4]*A[6]);
  return d;
}

// solve linear 3x3 system of equations Ax=b
float* CramerSolve3x3(float* A,float* b) {
  float* output=(float*)malloc(3*sizeof(float));
```

```
float detA=Determinant3x3(A);
if (detA==0) {
  cout<<"error: zero determinant for J"<<endl;
  output[0]=0;
  output[1]=0;
  output[2]=0;
  return output;
}
float* tempM=(float*)malloc(9*sizeof(float));
for (int i=0; i<9; i++)
  tempM[i]=A[i];
//calculate the x of x=(x,y,z)
tempM[0]=b[0];
tempM[3]=b[1];
tempM[6]=b[2];
float x=Determinant3x3(tempM)/detA;
tempM[0]=A[0];
tempM[3]=A[3];
tempM[6]=A[6];
//calculate y
tempM[1]=b[0];
tempM[4]=b[1];
tempM[7]=b[2];
```

```
  float y=Determinant3x3(tempM)/detA;
  tempM[1]=A[1];
  tempM[4]=A[4];
  tempM[7]=A[7];
  //calculate z
  tempM[2]=b[0];
  tempM[5]=b[1];
  tempM[8]=b[2];
  float z=Determinant3x3(tempM)/detA;
  output[0]=x;
  output[1]=y;
  output[2]=z;
  return output;
}

// F is vector function for nonlinear system
// of equations, this function returns -F
// which is needed in newton method
// v = (var1,cov,var2)
float* MyF(float*x, float* v) {
  float* F=(float*)malloc(3*sizeof(float));
  F[0]=-(x[0]*x[0]+x[1]*x[1]-v[0]);
  F[1]=-(x[1]*x[1]+x[2]*x[2]-v[2]);
  F[2]=-(x[0]*x[1]+x[1]*x[2]-v[1]);
  return F;
}

// hard coded calculation of the jacobian
// for the nonlinear system of equation to
// solve for the entries in matrix to
// multiply to the vector of std norms to
// get the joint norms
float* MyJacobian(float* x) {
  float* J=(float*)malloc(9*sizeof(float));
  J[0]=2*x[0];
  J[1]=2*x[1];
  J[2]=0;
  J[3]=0;
  J[4]=2*x[1];
  J[5]=2*x[2];
  J[6]=x[1];
  J[7]=x[0]+x[2];
  J[8]=x[1];
  return J;
}
```

```
// J~jacobian of F, v is needed to calculate F
// x~solution vector
float* NewtonNonlinearSysMethod(float* v) {
  float *J,*F,*s;
  float* x=(float*)malloc(3*sizeof(float));
  float* xprevious=(float*)malloc(3*sizeof(float));
  x[0]=10; x[1]=0.1; x[2]=0.01; //intitial x guess
  for (int k=0; k<20; k++) { //20 iterations
    J=MyJacobian(x); // 3x3 jacobian
    F=MyF(x,v); //returns 3x1 vector
    //solve Js=-f for s using Cramer
    s=CramerSolve3x3(J,F);
    x[0]=x[0]+s[0];
    x[1]=x[1]+s[1];
    x[2]=x[2]+s[2];
    free(J); free(F); free(s);
  }
  return x;
}

PointVec generateJointRandNorm(float var1,float cov,float var2) {
  PointVec U=generate2RandStdNorm();;
  float* v=(float*)malloc(3*sizeof(float));
  v[0]=var1; v[1]=cov; v[2]=var2;
  float* temp=NewtonNonlinearSysMethod(v);
  PointVec X;
  X.x=temp[0]*U.x+temp[1]*U.y;
  X.y=temp[1]*U.x+temp[2]*U.y;
  free(v); free(temp);
  return X;
}
```

APPENDIX C

RECONSTRUCTION CODE

Listing 3.4: osartCode

```
/*********************************************************/
/*
In step1 we assign each thread to a row in the block and do
parallel projections and in doing so we calculate residual
and scale every element by its appropriate row scalar ~
res/rowSum.  Therefore we only need to do column sum in the
update step so row indexes do not matter.  And so we can
sort scaledA and colInd by the values in ColInd without
concern for recalculating the values in rowPtr.
*/
__global__ void osartCsrCscStep1Ker(float *scaledA,
 int *colInd,int *rowPtr,float *B,float *X,int lengthAsub,
 int lengthA,int id) {
  int osartBlkStart=id*lengthAsub;
  int osartBlkEnd=osartBlkStart+lengthAsub;
  int tx=threadIdx.x;
  int bx=blockIdx.x;
  int rowBegin,rowEnd,Row,startRow;
  int totalThreads=gridDim.x*blockDim.x;
  int localRow=bx*blockDim.x+tx;
  float answer,res,rowSum;
  for (int m=0; m<ceil(lengthAsub/(float)totalThreads);
   m++) {
    startRow=osartBlkStart+m*totalThreads;
    Row=startRow+localRow;
    if (Row<lengthA and Row<osartBlkEnd) {
      answer=0;
      rowBegin=rowPtr[Row];
      rowEnd=rowPtr[Row+1];
      rowSum=rowEnd-rowBegin;
      for (int jj=rowBegin; jj<rowEnd; jj++)
        answer+=X[colInd[jj]];
      res=B[Row]-answer;
      //now scale each row into scaledA
      for(int jj=rowBegin; jj<rowEnd; jj++)
        scaledA[jj]=res/rowSum;
    }
  }
}
__global__ void osartCsrCscSetupKer(
 float *sortedColScaledASmall,float *scaledA,
 int *sortedColOrderSmall,int hitsInBlock,float* X) {
  int startIndexSmall,IndexSmall;
  int totalThreads=gridDim.x*blockDim.x;
  int localIndexSmall=blockIdx.x*blockDim.x+threadIdx.x;
  for (int m=0; m<ceil(hitsInBlock/(float)totalThreads);
   m++) {
    startIndexSmall=m*totalThreads;
    IndexSmall=startIndexSmall+localIndexSmall;
    if (IndexSmall<hitsInBlock)
      sortedColScaledASmall[IndexSmall]=
       scaledA[sortedColOrderSmall[IndexSmall]];
  }
}
__global__ void osartCsrCscUpdateKer(float *X,
 float *sortedColScaledASmall,int *colPtrSmall,
 int widthA,float lambda) {
  int tx=threadIdx.x;
  int bx=blockIdx.x;
  int colBegin,colEnd,Col,startCol;
  int totalThreads=gridDim.x*blockDim.x;
  int localCol=bx*blockDim.x+tx;
  float answer,colSum;
  for (int m=0; m<ceil(widthA/(float)totalThreads); m++) {
    answer=0;
    startCol=m*totalThreads;
    Col=startCol+localCol;
    if (Col<widthA) {
      answer=0;
      colBegin=colPtrSmall[Col];
      colEnd=colPtrSmall[Col+1];
      colSum=colEnd-colBegin;
      for (int ii=colBegin; ii<colEnd; ii++)
        answer+=sortedColScaledASmall[ii];
      if (colSum!=0)
        X[Col]+=answer*lambda/colSum;
    }
  }
}
/*******************************************************/

extern "C" void osart_CsrCsc(float *X,int *colInd,
 int *rowPtr,float *B,float *scaledA,
 vector<ColOrderBlock*> C,int lengthA,int lengthAsub,
 int widthA,int totalHits,float lambda,int iter) {
  int sizeX=widthA*sizeof(float);
  int sizecolInd=totalHits*sizeof(int);
  int sizerowPtr=(lengthA+1)*sizeof(int);
  int sizeB=lengthA*sizeof(float);
  int sizescaledA=totalHits*sizeof(float);
  int sizecolPtrSmall=(widthA+1)*sizeof(float);
  float *Xd,*Bd,*scaledAd,*sortedColScaledASmalld;
  int *colIndd,*rowPtrd,*colPtrSmalld,hitsInBlock;
  cudaMalloc((void**)& Xd,sizeX);
  cudaMemcpy(Xd,X,sizeX,cudaMemcpyHostToDevice);
  cudaMalloc((void**)& colIndd,sizecolInd);
  cudaMemcpy(colIndd,colInd,sizecolInd,
   cudaMemcpyHostToDevice);
  cudaMalloc((void**)& rowPtrd,sizerowPtr);
  cudaMemcpy(rowPtrd,rowPtr,sizerowPtr,
   cudaMemcpyHostToDevice);
  cudaMalloc((void**)& Bd,sizeB);
  cudaMemcpy(Bd,B,sizeB,cudaMemcpyHostToDevice);
  cudaMalloc((void**)& scaledAd,sizescaledA);
  for (int k=1; k<=iter; k++) {
    for (int bid=0; bid< ceil(lengthA/(float)lengthAsub);
     bid++) {
      osartCsrCscStep1Ker<<<GRID_SIZE,BLOCK_SIZE>>>(scaledAd,
       colIndd,rowPtrd,Bd,Xd,lengthAsub,lengthA,bid);
      hitsInBlock=C[bid]->sortedColOrder.size();
      cudaMalloc((void**)& sortedColScaledASmalld,
       hitsInBlock*sizeof(float));
      int* colPtrSmall=CopyVecToArray(C[bid]->colPtr);
      cudaMalloc((void**)& colPtrSmalld,sizecolPtrSmall);
      cudaMemcpy(colPtrSmalld,colPtrSmall,sizecolPtrSmall,
       cudaMemcpyHostToDevice);
      int* sortedColOrderSmall=
       CopyVecToArray(C[bid]->sortedColOrder);
      int* sortedColOrderSmalld;
      cudaMalloc((void**)& sortedColOrderSmalld,
       hitsInBlock*sizeof(int));
      cudaMemcpy(sortedColOrderSmalld,sortedColOrderSmall,
       hitsInBlock*sizeof(int),cudaMemcpyHostToDevice);
      osartCsrCscSetupKer<<<GRID_SIZE,BLOCK_SIZE>>>
       (sortedColScaledASmalld,scaledAd,sortedColOrderSmalld,
        hitsInBlock,Xd);
      osartCsrCscUpdateKer<<<GRID_SIZE,BLOCK_SIZE>>>(Xd,
       sortedColScaledASmalld,colPtrSmalld,widthA,lambda);
      cudaFree(colPtrSmalld);
      cudaFree(sortedColOrderSmalld);
      cudaFree(sortedColScaledASmalld);
      freeArrayI(colPtrSmall);
      freeArrayI(sortedColOrderSmall);
    }
  }
```

```
  cudaMemcpy(X,Xd,sizeX ,cudaMemcpyDeviceToHost);
  cudaFree(Xd);          cudaFree(colIndd);
  cudaFree(rowPtrd);     cudaFree(Bd);
}

class ColOrderBlock {
public:
  ColOrderBlock(vector<int> s,vector<int> p,int i) :
   sortedColOrder(s),colPtr(p),AsubId(i) {};
  vector<int> sortedColOrder;
  vector<int> colPtr;
  int AsubId;
};
```

Listing 3.5: sapCode

```
//Copy the current x-vector to every row of tempY
//(global-mem). tempY has a row to hold the update vector for
//each string
__global__ void setTempY(float* tempY,float* X,int M,
 int widthA) {
  int tx=threadIdx.x;
  int bx=blockIdx.x;
  int totalThreads=gridDim.x*blockDim.x;
  int localCol=bx*blockDim.x+tx;
  int Col,startColGrid;
  for (int m=0; m<ceil(widthA/(float)totalThreads); m++) {
    startColGrid=m*totalThreads;
    Col=startColGrid+localCol;
    if (Col<widthA) {
      for (int i=0; i<M; i++)
        tempY[i*widthA+Col]=X[Col];
    }
  }
}
//A thread is assigned to a string, and projects
//sequentially on all the rows within its string. After
//projecting the last row of its string each thread writes
//its update vector to its correct row in tempY
__global__ void sapProjectKer(float* tempY,int* colInd,
 int* rowPtr,float* B,int lengthA,int widthA,int M,
 float lambda) {
  int gtx=blockIdx.x*blockDim.x+threadIdx.x; //global thread
  //index
  if (gtx<M) {
    // threadRange ~ number of rows in a string
    int threadRange=ceil(lengthA/(float)M);
    int globalStartRow=gtx*threadRange;
    int Row,rowBegin,rowEnd,RowSum;
    float answer,Res;
    for (int i=0; i<threadRange; i++) {
      Row=globalStartRow+i;
      if (Row<lengthA) {
        answer=0;
        rowBegin=rowPtr[Row];
        rowEnd=rowPtr[Row+1];
        RowSum=rowEnd-rowBegin;
        for (int jj=rowBegin; jj<rowEnd; jj++)
          answer+=tempY[gtx*widthA+colInd[jj]];
        Res=B[Row]-answer;
        if (RowSum!=0) {
        for (int jj=rowBegin; jj<rowEnd; jj++)
          tempY[gtx*widthA+colInd[jj]]+=lambda*Res/RowSum;
        }
      }
    }
  }
}
// Do a column sum of tempY into X, averaging them.  Each
// element of the weight vector equals 1/M
```

```
__global__ void sapAverageKer(float* X,float* tempY,int M,
 int widthA) {
  int tx=threadIdx.x;
  int bx=blockIdx.x;
  int totalThreads=gridDim.x*blockDim.x;
  float answer;
  int Col,startColGrid;
  int localCol=bx*blockDim.x+tx;
  for (int m=0; m<ceil(widthA/(float)totalThreads); m++) {
    startColGrid=m*totalThreads;
    Col=startColGrid+localCol;
    if (Col<widthA) {
      answer=0;
      for (int i=0; i<M; i++)
        answer+=tempY[i*widthA+Col];
      X[Col]=answer/(float)M;
    }
  }
}

extern "C" void sapCsr(float* X,int* colInd,int* rowPtr,
 float* B,float* tempY,int lengthA,int widthA,int M,
 int totalHits,float lambda,int iter) {
```

```
int sizeX=widthA*sizeof(float);
int sizecolInd=totalHits*sizeof(int);
int sizerowPtr=(lengthA+1)*sizeof(int);
int sizeB=lengthA*sizeof(float);
int sizetempY=M*widthA*sizeof(float);

float *Xd,*Bd,*tempYd;
int *colIndd,*rowPtrd;

cudaMalloc((void**)& Xd,sizeX);
cudaMemcpy(Xd,X,sizeX,cudaMemcpyHostToDevice);
cudaMalloc((void**)& colIndd,sizecolInd);
cudaMemcpy(colIndd,colInd,sizecolInd,
 cudaMemcpyHostToDevice);
cudaMalloc((void**)& rowPtrd,sizerowPtr);
cudaMemcpy(rowPtrd,rowPtr,sizerowPtr,
 cudaMemcpyHostToDevice);
cudaMalloc((void**)& Bd,sizeB);
cudaMemcpy(Bd,B,sizeB,cudaMemcpyHostToDevice);
cudaMalloc((void**)& tempYd,sizetempY);

for (int k=1; k<=iter; k++) {
   setTempY<<<GRID_SIZE,BLOCK_SIZE>>>(tempYd,Xd,M,widthA);
```

```
    sapProjectKer<<<GRID_SIZE,BLOCK_SIZE>>>(tempYd, colIndd,
     rowPtrd,Bd, lengthA, widthA,M, lambda);
    sapAverageKer<<<GRID_SIZE,BLOCK_SIZE>>>(Xd, tempYd,M,
     widthA);
  }
  cudaMemcpy(X,Xd, sizeX, cudaMemcpyDeviceToHost);
  cudaFree(Xd); cudaFree(colIndd); cudaFree(rowPtrd);
  cudaFree(Bd); cudaFree(tempYd);
}
```

APPENDIX D

GPGPU-TESLA M2090

Listing 4.6: Tesla

```
Device 0: "Tesla M2090"
  CUDA Driver Version / Runtime Version
    5.0 / 5.0
  CUDA Capability Major/Minor version number:
    2.0
  Total amount of global memory:
    5375 MBytes (5636554752 bytes)
  (16) Multiprocessors x ( 32) CUDA Cores/MP:
    512 CUDA Cores
  GPU Clock rate:
    1301 MHz (1.30 GHz)
  Memory Clock rate:
    1848 Mhz
  Memory Bus Width:
    384-bit
  L2 Cache Size:
    786432 bytes
  Max Texture Dimension Size (x,y,z)
    1D=(65536), 2D=(65536,65535), 3D=(2048,2048,2048)
  Max Layered Texture Size (dim) x layers
    1D=(16384) x 2048, 2D=(16384,16384) x 2048
  Total amount of constant memory:
```

```
  65536 bytes
Total amount of shared memory per block:
  49152 bytes
Total number of registers available per block:
  32768
Warp size:
  32
Maximum number of threads per multiprocessor:
  1536
Maximum number of threads per block:
  1024
Maximum sizes of each dimension of a block:
  1024 x 1024 x 64
Maximum sizes of each dimension of a grid:
  65535 x 65535 x 65535
Maximum memory pitch:
  2147483647 bytes
Texture alignment:
  512 bytes
Concurrent copy and kernel execution:
  Yes with 2 copy engine(s)
Run time limit on kernels:
  No
```

```
Integrated GPU sharing Host Memory:
  No
Support host page-locked memory mapping:
  Yes
Alignment requirement for Surfaces:
  Yes
Device has ECC support:
  Enabled
Device supports Unified Addressing (UVA):
  Yes
Device PCI Bus ID / PCI location ID:
  3 / 0
Compute Mode:
   < Default (multiple host threads can use ::
  cudaSetDevice() with device simultaneously) >
```

REFERENCES

[1] S. Agostinelli, et al., “GEANT4-A simulation toolkit”, *Nuclear Instruments and Methods in Physics Research A*, vol. 506, pp. 250-303, 2003.

[2] N. Bell and M. Garland, *Efficient Sparse Matrix-Vector Multitplication on CUDA*. NVIDIA Technical Report NVR-2008-004. December 2008.

[3] G.E.P. Box, M.E. Muller, “A note on the generation of random normal deviates,” *The Annals of Mathematical Statistics*, Vol. 29-2, 610-611 (1958).

[4] Y. Censor, T. Elfving, and G.T. Hernam, “Averaging strings of sequential iterations for convex feasibilty problems,” *Inherently Parallel Algorithms in Feasibility and Optimization and their Applications*, Esevier Science Publications, Amsterdam, The Netherlands, D. Butnnariu, Y. Censor, and S. Reich (Ed), 101-114 (2001).

[5] G. Herman, *Fundamentals of Computerized Tomography*, Springer, 2009.

[6] G.T. Herman and L.B. Meyer. Algebraic reconstruction techniques can be made computationally efficient. *IEEE Transactions on Medical Imaging*, 12(3): 600-609,1993.

[7] V. Giacometti, S. Guatelli, A. Rosenfeld, R. Schulte (2013) High-Resolution Head Phantom for Geant4 Proton Simulations. Oral presentation at the Geant4 2013

International User Conference at the Physics-Medicine-Biology frontier, October 7-9, 2013 Bordeaux, France.

[8] V.L. Highland, “Some practical remarks on multiple scattering”, *Nuclear Instruments and Methods*, vol. 129, pp. 497-499, 1975.

[9] R. F. Hurley, R. W. Schulte, V. A. Bashkirov, A. J. Wroe, A. Ghebremedhin, H. F.-W. Sadrozinski, V. Rykalin and G. Coutrakon, P. Koss, B. Patyal, “Water-equivalent path length calibration of a prototype proton CT scanner,” *Med. Phys.*vol. 39, pp. 2438-2246, 2012.

[10] S. Kaczmarz, “Angenaherte Auflosung von Systemen Linearer Gleichungen,” *Bulletin Internatioanl de l’Academie Polonaise des Sciences et des Lettres.* Classe des Sciences Mathematiques et Naturelles. Serie A, Sciences Mathematiques, vol. 35, pp.355-357, 1937.

[11] D. Kirk, W. Hwu, “Programming Massively Parallel Processors: A Hands-on Approach,” Morgan Kaufman Publishers, 2010.

[12] G.R. Lynch, O.I. Dahl, “Approximations to multiple Coulomb scattering,” *Nuclear Instruments and Methods in Physics Research Section B: Beam Interactions with Materials and Atoms*, 58-1, 6-10, (1991).

[13] G. Marsaglia, T.A. Bray, “A convenient method for generating normal variables,” *SIAM*, Rev. 6, 260-264 (1964).

[14] K. Nakamura, et al. (Particle Data Group). “Review of Particle Physics, 27.3. Multiple scattering through small angles”, *J. Phys.*, vol. G 37, pp. 290-291, 2010.

[15] NIST, National Institute of Standards and Technology PSTAR Database Program, http://physics.nist.gov/PhysRefData/Star/Text/PSTAR.html.

[16] Particle Therapy Co-operative group, http://ptcog.web.psi.ch/ptcentres.html, 24/11/09.

[17] S. Penfold, *Image Reconstruction and Monte Carlo Simulations in the Development of Proton Computed Tomography for Applications in Proton Radiation Therapy*. Doctoral Thesis. University of Wollongong, Australia. 2010.

[18] S.N. Penfold, R.W. Schulte, K.E. Schubert, A.B. Rosenfeld, "A more accurate reconstruction system matrix for quantitative proton computed tomography", *Medical Physics*, vol. 36, pp. 4511-4518, 2009.

[19] R.W. Schulte, S.N. Penfold, J.T. Tafas, and K.E. Schubert, "A maximum likelihood proton path formalism for application in proton computed tomography", *Medical Physics*, vol. 35, pp. 4849-4856, 2008.

[20] R.W. Schulte, W. Bashkirov, R.F. Hurley, P. Johnson, S. Macafee, T. Plautz, H.F.W. Sadrozinski, A. Zatserklaniy, B. Schultze, M. Witt, K.E. Schubert, V. Giacometti, S. Guatelli, A. Rosenfeld, "Development of a CLinical Head Scanner for Proton CT,“ *IEEE Nuclear Science Symposium - Medical Imaging Conference* N 19-1 (2013).

[21] B. Schultze, M. Witt, K. Schubert, R.F. Hurley, V. Bashkirov, R.W. Schulte, E. Gomez, "Space Carving and Filtered Back-Projection as Preconditioners for Proton Computed TomographyReconstruction," *IEEE Nuclear Science Symposium - Medical Imaging Conference* (2012).

[22] J.M. Slater, J.O. Archambeau, D.W. Miller, M.I. Notarus, W. Preston, and J.D. Slater, "The proton treatment center at Loma Linda University Medical Center," *International Journal of Radiation Oncology, Biology, Physics*, 22, 383-389 (1991).

[23] C.A. Tobias, J.H. Lawrence, J.L. Born, R.K. McCombs, J.E. Roberts, H.O. Anger, V.V.A. Low-Beer, and C.B. Huggins, "Pituitary irradiation with high-energy proton beams: a preliminary report," *Cancer Research*, 18, 121-134 (1958).

[24] C. Tschalar, "Straggling distributions of extremely large energy losses," *Nuclear Instruments and Methods*, 61, 141-156 (1968).

[25] R.R. Wilson, "Radiological use of fast protons," *Radiology*, 47, 487-491 (1946).

[26] M. Witt, B. Schultze, R. Schulte, K.E. Schubert, E. Gomez, "A Proton Simulator for Testing Implementations of Proton CT Reconstruction Algorithms on GPGPU Clusters," *IEEE Nuclear Science Symposium - Medical Imaging Conference* HT 4-1 (2012).