

California State University, San Bernardino

CSUSB ScholarWorks

Theses Digitization Project

John M. Pfau Library

2009

Efficient proton computed tomography image reconstruction using general purpose graphics processing units

Scott Alan McAllister

Follow this and additional works at: <https://scholarworks.lib.csusb.edu/etd-project>



Part of the [Computer and Systems Architecture Commons](#)

Recommended Citation

McAllister, Scott Alan, "Efficient proton computed tomography image reconstruction using general purpose graphics processing units" (2009). *Theses Digitization Project*. 3753.
<https://scholarworks.lib.csusb.edu/etd-project/3753>

This Thesis is brought to you for free and open access by the John M. Pfau Library at CSUSB ScholarWorks. It has been accepted for inclusion in Theses Digitization Project by an authorized administrator of CSUSB ScholarWorks. For more information, please contact scholarworks@csusb.edu.

EFFICIENT PROTON COMPUTED TOMOGRAPHY IMAGE
RECONSTRUCTION USING GENERAL PURPOSE
GRAPHICS PROCESSING UNITS

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
Scott Alan McAllister

March 2009

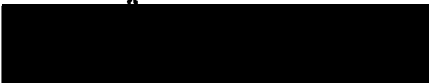
EFFICIENT PROTON COMPUTED TOMOGRAPHY IMAGE
RECONSTRUCTION USING GENERAL PURPOSE
GRAPHICS PROCESSING UNITS

A Thesis
Presented to the
Faculty of
California State University,
San Bernardino

by
Scott Alan McAllister

March 2009

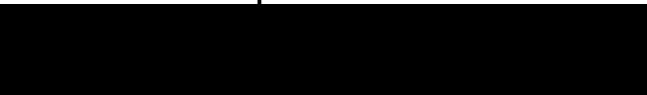
Approved by:



Keith Evan Schubert, Chair, Computer
Science and Engineering



Ernesto Gomez



Richard Botting



Reinhard Schulte

3/23/09
Date

© 2009 Scott Alan McAllister

ABSTRACT

Proton Computed Tomography (pCT) is an imaging modality that is based on the tracking of individual protons as they traverse an object. These paths deviate from a straight line due to the effects of multiple Coulomb scattering (MCS) and must be tracked using a formalism that models MCS. A sparse iterative solver, like the algebraic reconstruction technique (ART), is used to reconstruct the object using these non-linear paths. Because sparse iterative solvers are slow, parallel algorithms, executed simultaneously over multiple processing units are required. This thesis will investigate the use of general purpose graphics processing units (GPGPUs) for execution of these parallel algorithms as well as exploiting the structure of the data being processed. From the results, recommendations for new research directions will be suggested.

ACKNOWLEDGEMENTS

I would like to thank my advisor Dr. Keith Schubert for his time, knowledge, and patience while guiding me during these past years. I would also like to thank my committee members, Dr. Reinhard Schulte, Dr. Ernesto Gomez, Dr. Richard Botting. Finally, I would like to thank Scott Penfold and Dr. Yair Censor for their contributions and my wife Brittany for her patience this past year.

DEDICATION

To Bob

TABLE OF CONTENTS

<i>Abstract</i>	iii
<i>Acknowledgements</i>	iv
<i>List of Tables</i>	x
<i>List of Figures</i>	xi
1. <i>Introduction</i>	1
1.1 Background	1
1.2 Significance	2
1.3 Flow of Document	3
1.4 Processing the Data	4
2. <i>The Most Likely Path</i>	6
2.1 Σ Matrices	6
2.1.1 Proton Energy Loss as a Function of Depth	7
2.2 R Matrices	8
2.3 Naïve Most Likely Path Step Calculation	8
2.4 Most Likely Path Step Calculation Optimization	9
2.4.1 First Half of Most Likely Path Calculation	11
2.4.2 Second Half of Most Likely Path Calculation	12
2.4.3 Finalized Calculation	14

2.5	Further Optimizations	14
2.5.1	Precalculations	15
2.5.2	Component Separation with Precalculations	16
2.5.3	P Row Separations	17
3.	<i>Calculating the Convex Hull of the Object</i>	19
4.	<i>Integral Relative Electron Density</i>	23
4.1	Energy Calculation	23
4.2	General Purpose Graphics Processing Unit Speedup of Integral Relative Electron Density Calculation	25
5.	<i>A Bin Sort Based Fast Most Likely Path Algorithm</i>	27
5.1	Choosing Bins	27
5.2	Accuracy of Binning	29
5.3	Reconstruction Algorithms	29
5.4	On the Fly Most Likely Path Calculation With Block Iterative Reconstruction Algorithm	30
6.	<i>Summary of Findings</i>	31
6.1	Time Savings of Precalculations	31
6.2	Space Savings of Sparse Matrices	31
6.3	Relaxation Parameter as a Function of Depth	32
6.4	Future Work	34
6.4.1	Faster Most Likely Path Calculation	34
6.4.2	Bin Size Calculation and Relaxation Parameter Testing	36
6.4.3	Applying the Bin Sort Based Fast Most Likely Path Algorithm to More Algorithms	36

6.4.4	Research on Applications of General Purpose Graphics Processing Units in Clusters	36
6.4.5	On-Line Imaging	37
7.	<i>Conclusions</i>	38
	<i>Appendix A: Reconstruction Algorithms</i>	39
A.1	Fully Sequential Algorithms	40
A.1.1	The Fully Sequential Algebraic Reconstruction Technique (Kaczmarz)	40
A.2	Fully Simultaneous Algorithms	40
A.2.1	The Fully Simultaneous Algebraic Reconstruction Technique (Cimmino)	40
A.2.2	Fully Simultaneous Component Averaging (CAV)	41
A.3	Block Iterative Algorithms	41
A.3.1	Block Iterative Component Averaging (BICAV)	41
A.3.2	The Block Iterative Diagonally Relaxed Orthogonal Projections Algorithm (DROP)	42
A.3.3	The Ordered Subsets Simultaneous Algebraic Reconstruction Technique (OS-SART)	42
A.4	String Averaging Algorithms	42
A.4.1	The String Averaging Projections Algorithm (SAP)	43
A.4.2	Component Averaged Row Projection (CARP)	43
	<i>Appendix B: Matrix Operation Costs</i>	44
B.5	2×2 Matrix-Matrix Multiplication	45
B.6	2×2 Sparse Matrix-Matrix Multiplication	45
B.7	2×2 Matrix-Vector Multiplication	45

B.8 2×2 Matrix Inverse	46
<i>Appendix C: Source Code</i>	47
C.9 Reconstruction Code	48
C.10 Reconstruction Kernel Code	92
<i>References</i>	97

LIST OF TABLES

4.1 Graphics Processing Unit Speedup of Integral Relative Electron Density Calculation	26
---	----

LIST OF FIGURES

1.1	The path of the data as it passes through the image reconstruction program.	5
3.1	Top down view of the pCT detector system.	20
3.2	The phantom to be reconstructed.	21
3.3	Convex hull approximation of the object to be imaged using 0.300MeV as a tolerance value.	22
5.1	Proton history distribution as a function of depth.	28
6.1	Phantom reconstructed after ten cycles with a relaxation parameter of 3.0.	33
6.2	Phantom reconstructed after ten cycles with a relaxation parameter of 0.3 multiplied by the depth in cm.	35

1. INTRODUCTION

For accurate proton treatment planning of brain and head and neck tumors, we need to reconstruct a 3D data set of about 30 million voxels contained in a human head and neck. The CT system must deal with solving problems of the form $Ax = b$ for x , where A is a large sparse matrix on the order of 100 million (proton histories) by 30 million (voxels). Even larger matrices will arise when imaging and reconstructing other anatomical regions in the human body. Such large and sparse systems can only be solved with iterative reconstruction methods, which are known to be inherently slow. This work explores the possibility of efficiently reconstructing pCT images using general purpose graphics processing units (gp-gpu's).

1.1 Background

Clinical application of protons was first suggested over 60 years ago [13]. Proton radiation can deliver high doses of radiation to tumors or other targets close to critical structures, and thus is vitally important for modern 3D conformal radiation therapy. Currently proton dose calculations rely on x-ray computed tomography (xCT), which limits their accuracy due to the physical interaction differences of protons and x rays. To gain the maximum benefit from proton therapy, proton computed tomography (pCT) offers the opportunity to more accurately plan proton doses and to verify the correct proton beam delivery in the treatment position. This is accomplished by choosing the proton energy sufficiently high to penetrate the patient and by reconstructing density values based on energy loss measurements [3]. As an additional advantage, pCT achieves similar density resolution with lower dose than x-ray CT, because each proton is tracked individually. Despite these

advantages, a fully operational pCT system does currently not exist. This is, in part, related to the large amount of proton and object data that need to be acquired and reconstructed, respectively.

Preliminary work in proton CT has been performed and was published over the last four years [11, 5, 6, 7, 8, 10]. The published work includes a detailed analysis and description of the conceptual design of a proton CT scanner [10], an overview and direction towards reconstruction methods in proton CT [8], an analysis of the dose-contrast relationship in proton CT in comparison to an ideal x-ray CT system [9], and a demonstration of the feasibility of the algebraic reconstruction technique for proton CT reconstruction [7]. In addition, Williams published a paper on the most likely path of protons in a homogeneous medium estimated from known entry and exit parameters [12], which is important for all proton reconstruction methods. The published work has laid the foundation for subsequent studies into proton CT. It has pointed toward a dose advantage of proton CT based on single particle tracking and the superiority of the most likely path reconstruction versus a straight line based reconstruction. It has also demonstrated the need for numerical optimization of reconstruction algorithms.

1.2 Significance

Parallel algorithms designed specifically for gp-gpu computation can take advantage of the parallel nature of pCT reconstruction and may be able to provide an on-line imaging system for proton radiotherapy in the future. This would allow for dose tracking as well as beam tracking, providing a more precise dose for patients. It could, in turn, allow for fewer treatments and shorter overall treatment times.

NVIDIA GTX280 series GPUs contain 240 processing cores running at 1296Mhz each [1]. They can be programmed using a subset of C called CUDA (Compute Unified Device Architecture). Because the GPUs are accessed via a PCI Express 2.0 bus, they can handle up to 8GB/s in and 8GB/s out (500MB/s * 16 lanes). In the case of pCT, the PCI Express bus will be a bottleneck because memory bandwidths

on the motherboard as well as the GPU are much faster. Because of this bottleneck, algorithms need to be designed to maximize the number of calculations per data transfer.

With the way pCT data is gathered, it should be possible to design such algorithms. Proton entry and exit angle and location are taken from the detectors and calculations are done to find the most likely path of the proton [12, 9] through the object. If the most likely path is stored in an array for calculation of the image reconstruction it will cause much larger data transfers (100 million (proton histories) x 30 million (voxels) instead of 100 million x 4 (input/output angle and offset for a single plane) for two dimensional or 100 million x 8 (input/output angle and offset for two planes) for three dimensional image reconstruction). Reconstructing a two dimensional image from 100 million proton histories can be done with as few as two data transfers (GPU memory permitting), one to input the detector data and one to output the image. If more data transfers are required, e.g., in an iterative sequence of image reconstructions, the most likely path will be recalculated instead of stored because the time taken to calculate the path should be less than the time required to transfer it across the PCI Express bus. If the total reconstruction times are sufficiently low (less than 15 minutes), tests will be performed for the feasibility of an on-line image reconstruction. That is, starting the reconstruction process while the detectors are still gathering data. An on-line reconstruction would also be useful for beam tracking as well as dose deposit tracking. This would make the far-reaching goal to image the patient, plan the treatment, and then treat the patient in the same visit a realistic goal.

1.3 Flow of Document

In this thesis I will show the following:

- A convex hull was created to approximate the boundary of the image. This is shown in Chapter 3.

- Code migration from CPU to GPU can show significant performance increases with as much as a three order of magnitude difference. This is shown in Section 4.2
- The precalculations required for efficient GPU computation lead to a new sparse iterative reconstruction algorithm, are not GPU specific and can yield significant performance increases for CPU computations with as much as a two order of magnitude difference. This is shown in Section 2.5.1 and Figure 1.1.
- Reconstruction times were broughtg down from 1.5 hours per cycle to 1 minute per cycle. This speedup shows a two order of magnitude difference and is enough to perform a 10 cycle reconstruction in about 10 minutes, reaching the goal of reconstructing an image in less than 10 minutes. This is shown in Section 6.1.

1.4 Processing the Data

When processing the data for a pCT image reconstruction several steps must be performed in order. The following list shows the order of the data flow through the reconstruction process. While inputting data:

1. Remove large angle histories via standard deviation cuts (3σ tolerance)
2. Separate data with ≤ 0.300 MeV loss as straight line data for boundry approximation

After all data has been read in and processed:

1. Calculate integral relative electron density (IRED) on GPU for all protons with > 0.300 MeV loss
2. Perform boundry approximation with < 0.300 MeV data
3. Use boundry information to find maximum depth and calculate Σ , R and P matrices

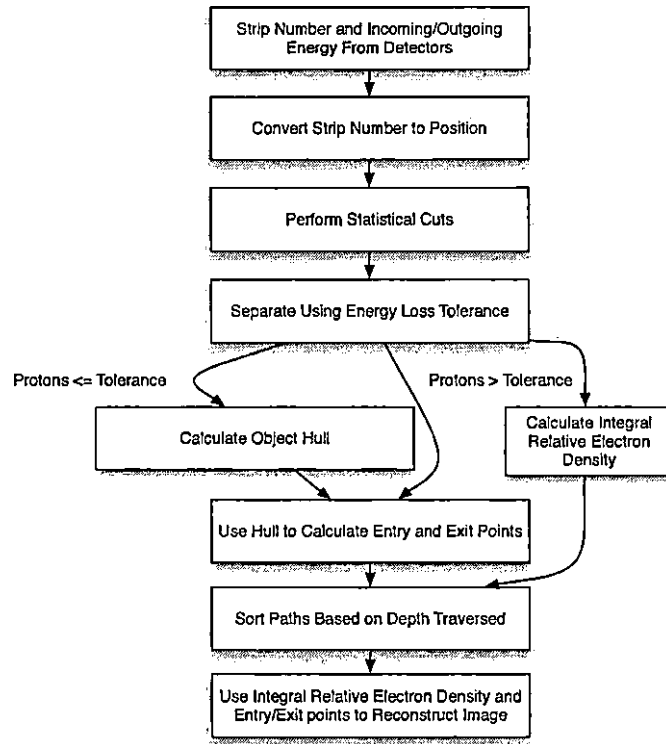


Fig. 1.1: The path of the data as it passes through the image reconstruction program.

4. Sort data into bins or "blocks" based on depth traversed (u_2) for MLP calculation
5. Send bin with associated P matrices, IREDs and gantry rotation to GPU

Do for each bin on GPU:

1. Calculate MLP for histories in the bin
2. Rotate via Givens rotation if needed
3. Give the data a pixel/voxel number
4. Reconstruct image for the bin with an iterative reconstruction algorithm

2. THE MOST LIKELY PATH

The path of a proton through an object can be represented as

$$y_{mlp} = (\Sigma_1^{-1} + R_1^T \Sigma_2^{-1} R_1)^{-1} (\Sigma_1^{-1} R_0 y_0 + R_1^T \Sigma_2^{-1} y_2)$$

Where y_{mlp} is a vector of varying size. It contains two components, t and θ , for as many steps as the most likely path needs. The number of steps varies based on the depth the proton traverses and the size of the step. Σ_1 and Σ_2 are energy loss variance matrices and R_0 and R_1 are the depth the proton has traveled through the object and the depth yet to be traveled respectively.

2.1 Σ Matrices

The matrices Σ_1 and Σ_2 can be represented by the following integrals:

$$\Sigma_1 = \begin{bmatrix} \sigma_{t_1}^2 & \sigma_{\theta_1}^2 \\ \sigma_{\theta_1}^2 & \sigma_{t_1\theta_1}^2 \end{bmatrix} \quad (2.1)$$

where

$$\sigma_{t_1}^2(u_0, u_1) = E_0^2 \left(1 + 0.038 \ln \frac{u_1 - u_0}{X_0} \right)^2 \int_{u_0}^{u_1} \frac{(u_1 - u)^2 du}{\beta^2(u) p^2(u) X_0} \quad (2.2)$$

$$\sigma_{\theta_1}^2(u_0, u_1) = E_0^2 \left(1 + 0.038 \ln \frac{u_1 - u_0}{X_0} \right)^2 \int_{u_0}^{u_1} \frac{1 du}{\beta^2(u) p^2(u) X_0} \quad (2.3)$$

$$\sigma_{t_1\theta_1}^2(u_0, u_1) = E_0^2 \left(1 + 0.038 \ln \frac{u_1 - u_0}{X_0} \right)^2 \int_{u_0}^{u_1} \frac{u_1 - u du}{\beta^2(u) p^2(u) X_0} \quad (2.4)$$

and

$$\Sigma_2 = \begin{bmatrix} \sigma_{t_2}^2 & \sigma_{\theta_2}^2 \\ \sigma_{\theta_2}^2 & \sigma_{t_2\theta_2}^2 \end{bmatrix} \quad (2.5)$$

where

$$\sigma_{t_2}^2(u_1, u_2) = E_0^2 \left(1 + 0.038 \ln \frac{u_1 - u_2}{X_0} \right)^2 \int_{u_1}^{u_2} \frac{(u_2 - u)^2}{\beta^2(u)p^2(u)} \frac{du}{X_0} \quad (2.6)$$

$$\sigma_{\theta_2}^2(u_1, u_2) = E_0^2 \left(1 + 0.038 \ln \frac{u_1 - u_2}{X_0} \right)^2 \int_{u_1}^{u_2} \frac{1}{\beta^2(u)p^2(u)} \frac{du}{X_0} \quad (2.7)$$

$$\sigma_{t_2\theta_2}^2(u_1, u_2) = E_0^2 \left(1 + 0.038 \ln \frac{u_1 - u_2}{X_0} \right)^2 \int_{u_1}^{u_2} \frac{u_2 - u}{\beta^2(u)p^2(u)} \frac{du}{X_0} \quad (2.8)$$

where β^2 is the squared velocity relative to the speed of light, $p^2(u)$ is the momentum of the proton at depth u , and $E_0 = 13.6 \text{ MeV}/c$ is a constant. In this case X_0 is the radiation length for water (36.08 cm).

2.1.1 Proton Energy Loss as a Function of Depth

$$\frac{1}{\beta^2(u)p^2(u)} = \frac{(E(u)+E_p)^2 c^2}{(E(u)+2E_p)^2 E^2(u)} \quad (2.9)$$

where $E(u)$ is the kinetic energy of a proton at depth u , E_p is the proton rest energy in MeV and c is the speed of light in m/s. $E(u)$ is estimated by a fifth degree polynomial (higher degree polynomials become oscillatory) of the form $a_0 + a_1u + a_2u^2 + a_3u^3 + a_4u^4 + a_5u^5$ with coefficients as follows for 200MeV entry energy:

$$\begin{aligned} a_0 &= 202.20574 \\ a_1 &= -7.6174839 \\ a_2 &= 0.9413194 \\ a_3 &= -0.1141406 \\ a_4 &= 0.0055340 \\ a_5 &= -0.0000972 \end{aligned}$$

2.2 R Matrices

The R matrices are the same size as the Σ matrices ($2 \times 2 \times$ number of steps) and are of the form

$$R_0 = \begin{bmatrix} 1 & u - u_0 \\ 0 & 1 \end{bmatrix}$$
$$R_1 = \begin{bmatrix} 1 & u_2 - u \\ 0 & 1 \end{bmatrix}$$

where u_0 is the proton entry point (usually 0), u_2 is the proton exit point and u is the incrementing step point between u_0 and u_2 . In other words, $u - u_0$ is the distance the proton has traversed into the object and $u_2 - u$ is the distance the object has yet to traverse.

2.3 Naïve Most Likely Path Step Calculation

Using the costs from Section A.4.2 we will calculate the number of multiplications in the naïve implementation of the MLP.

For the purpose of this implementation we will assume every matrix is dense with no elements equal to 1.

$$\begin{aligned}
y_{mlp} &= (\Sigma_1^{-1} + R_1^T \Sigma_2^{-1} R_1)^{-1} (\Sigma_1^{-1} R_0 y_0 + R_1^T \Sigma_2^{-1} y_2) \quad (2.10) \\
&= \left(\begin{array}{c} \left[\begin{array}{cc} S_{1,1} & S_{1,2} \\ S_{2,1} & S_{2,2} \end{array} \right]^{-1} + \\ \left[\begin{array}{cc} R_{1,1} & R_{1,2} \\ R_{2,1} & R_{2,2} \end{array} \right]^T \left[\begin{array}{cc} S_{2,1} & S_{2,2} \\ S_{2,1} & S_{2,2} \end{array} \right]^{-1} \left[\begin{array}{cc} R_{1,1} & R_{1,2} \\ R_{2,1} & R_{2,2} \end{array} \right] \end{array} \right)^{-1} \\
&\quad \left(\begin{array}{c} \left[\begin{array}{cc} S_{1,1} & S_{1,2} \\ S_{2,1} & S_{2,2} \end{array} \right]^{-1} \left[\begin{array}{cc} R_{0,1} & R_{0,2} \\ R_{0,1} & R_{0,2} \end{array} \right] \begin{bmatrix} t_0 \\ \theta_0 \end{bmatrix} + \\ \left[\begin{array}{cc} R_{1,1} & R_{1,2} \\ R_{2,1} & R_{2,2} \end{array} \right]^T \left[\begin{array}{cc} S_{2,1} & S_{2,2} \\ S_{2,1} & S_{2,2} \end{array} \right]^{-1} \begin{bmatrix} t_2 \\ \theta_2 \end{bmatrix} \end{array} \right)
\end{aligned}$$

There are two matrix-matrix multiplications, one matrix addition and one matrix inverse in the first part of the equation (The sigma matrices are inverted when they are calculated so those operations will not be counted). These calculations total 35 floating-point operations. There are two matrix-matrix multiplications, two matrix-vector multiplications and one vector addition in the second part of the equation. These calculations total 38 floating-point operations. Finally, multiplying the two parts requires one matrix-vector multiplication for 6 floating-point operations for a total of 79 floating-point operations per step per proton.

2.4 Most Likely Path Step Calculation Optimization

The naïve implementation of this type of equation works very well for test cases, but for the purpose of practical pCT applications it is simply too slow. The R matrices lend themselves to fast multiplications because three of the four values are constant and not only that, they are either zero or one which allows for the removal of several multiplications.

$$\begin{aligned}
y_{mlp} &= (\Sigma_1^{-1} + R_1^T \Sigma_2^{-1} R_1)^{-1} (\Sigma_1^{-1} R_0 y_0 + R_1^T \Sigma_2^{-1} y_2) \quad (2.11) \\
&= \left(\begin{array}{c} \left[\begin{array}{cc} S_{1,1} & S_{1,2} \\ S_{2,1} & S_{2,2} \end{array} \right]^{-1} + \\ \left[\begin{array}{cc} R_{1,1} & R_{1,2} \\ R_{2,1} & R_{2,2} \end{array} \right]^T \left[\begin{array}{cc} S_{2,1} & S_{2,2} \\ S_{2,1} & S_{2,2} \end{array} \right]^{-1} \left[\begin{array}{cc} R_{1,1} & R_{1,2} \\ R_{2,1} & R_{2,2} \end{array} \right] \end{array} \right)^{-1} \\
&\quad \left(\begin{array}{c} \left[\begin{array}{cc} S_{1,1} & S_{1,2} \\ S_{2,1} & S_{2,2} \end{array} \right]^{-1} \left[\begin{array}{cc} R_{0,1} & R_{0,2} \\ R_{0,1} & R_{0,2} \end{array} \right] \begin{bmatrix} t_0 \\ \theta_0 \end{bmatrix} + \\ \left[\begin{array}{cc} R_{1,1} & R_{1,2} \\ R_{2,1} & R_{2,2} \end{array} \right]^T \left[\begin{array}{cc} S_{2,1} & S_{2,2} \\ S_{2,1} & S_{2,2} \end{array} \right]^{-1} \begin{bmatrix} t_2 \\ \theta_2 \end{bmatrix} \end{array} \right)
\end{aligned}$$

There are two sparse matrix-matrix multiplications, one matrix addition and one matrix inverse in the first part of the equation (The sigma matrices are inverted when they are calculated so those operations will not be counted). These calculations total 19 floating-point operations. There are two sparse matrix-matrix multiplications, two matrix-vector multiplications and one vector addition in the second part of the equation. These calculations total 22 floating-point operations. Finally, multiplying the two parts requires one matrix-vector multiplication for 6 floating-point operations for a total of 47 floating-point operations per step per proton. Accounting for the sparsity of the R matrices allows for approximately 40% fewer floating-point operations.

These floating-point operation counts are assuming values are stored after every matrix operation which requires more memory and synchronization. The expanded form, requiring less memory, is shown below. The advantage of this form will be most apparent on the GPU where there is less memory. Using the expanded form below it may be possible to send a higher number of proton histories to the GPU per batch thus reducing the total number of batches and data transfers to the GPU. Because data transfers are so expensive, the time required to perform the extra calculations associated with the expanded form may be less than the time saved

from data transfers.

2.4.1 First Half of Most Likely Path Calculation

Substitute values for $R1$ and the first part of the equation becomes:

$$\begin{aligned}
 & \left(\begin{bmatrix} S1i_{1,1} & S1i_{1,2} \\ S1i_{2,1} & S1i_{2,2} \end{bmatrix} + \begin{bmatrix} 1 & 0 \\ R1_{1,2} & 1 \end{bmatrix} \begin{bmatrix} S2i_{1,1} & S2i_{1,2} \\ S2i_{2,1} & S2i_{2,2} \end{bmatrix} \begin{bmatrix} 1 & R1_{1,2} \\ 0 & 1 \end{bmatrix} \right)^{-1} \quad (2.12) \\
 = & \left(\begin{array}{c} \begin{bmatrix} S1i_{1,1} & S1i_{1,2} \\ S1i_{2,1} & S1i_{2,2} \end{bmatrix} + \\ \begin{bmatrix} R1_{1,1} * S2i_{1,1} + R1_{2,1} * S2i_{2,1} & R1_{1,1} * S2i_{1,2} + R1_{2,1} * S2i_{2,2} \\ R1_{1,2} * S2i_{1,1} + R1_{2,2} * S2i_{2,1} & R1_{1,2} * S2i_{1,2} + R1_{2,2} * S2i_{2,2} \end{bmatrix} \\ \begin{bmatrix} 1 & R1_{1,2} \\ 0 & 1 \end{bmatrix} \end{array} \right)^{-1}
 \end{aligned}$$

Accounting for the symmetry in the Σ matrices ($S1i_{1,2} = S1i_{2,1}$ and $S2i_{1,2} = S2i_{2,1}$), multiplication by one and zero from first R matrix multiplication and then performing the second R matrix multiplication:

$$\begin{aligned}
 & \left(\begin{array}{c} \begin{bmatrix} S1i_{1,1} & S1i_{1,2} \\ S1i_{1,2} & S1i_{2,2} \end{bmatrix} + \\ \begin{bmatrix} S2i_{1,1} & S2i_{1,2} \\ R1_{1,2} * S2i_{1,1} + S2i_{1,2} & R1_{1,2} * S2i_{1,2} + S2i_{2,2} \end{bmatrix} \begin{bmatrix} 1 & R1_{1,2} \\ 0 & 1 \end{bmatrix} \end{array} \right)^{-1} \\
 & = \begin{bmatrix} M_{1,1} & M_{1,2} \\ M_{2,1} & M_{2,2} \end{bmatrix}^{-1} \quad (2.13)
 \end{aligned}$$

where

$$\begin{aligned}
 M_{1,1} &= S2i_{1,1} + S1i_{1,1} \\
 M_{1,2} &= (S2i_{1,1} * R1_{1,2} + S2i_{1,2}) + S1i_{1,2} \\
 M_{2,1} &= (R1_{1,2} * S2i_{1,1} + S2i_{1,2}) + S1i_{1,2} \\
 M_{2,2} &= ((R1_{1,2} * S2i_{1,1} + S2i_{1,2}) * R1_{1,2} + (R1_{1,2} * S2i_{1,2} + S2i_{2,2})) + S1i_{2,2}
 \end{aligned}$$

By exploiting the symmetry of the Σ matrices and sparsity of the R matrices $M_{1,2}$ and $M_{2,1}$ are the same.

The inverse of a 2×2 matrix (from Section B.8) can be shown as

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix}^{-1} = \frac{1}{ad - bc} \begin{bmatrix} d & -b \\ -c & a \end{bmatrix}$$

where $ad - bc$ is the determinant of the matrix

But in this case b and c are the same, which gives

$$\begin{bmatrix} a & b \\ b & d \end{bmatrix}^{-1} = \frac{1}{ad - 2b} \begin{bmatrix} d & -b \\ -b & a \end{bmatrix}$$

$$\begin{aligned} \det = & (S2i_{1,1} + S1i_{1,1}) * \\ & (((R1_{1,2} * S2i_{1,1} + S2i_{1,2}) * R1_{1,2} + (R1_{1,2} * S2i_{1,2} + S2i_{2,2})) + S1i_{2,2}) - \\ & 2 * ((S2i_{1,1} * R1_{1,2} + S2i_{1,2}) + S1i_{1,2}) \end{aligned}$$

Substituting for 2.13 gives:

$$\frac{1}{\det} \begin{bmatrix} M_{2,2} & -M_{1,2} \\ -M_{1,2} & M_{1,1} \end{bmatrix} \quad (2.14)$$

$$\begin{bmatrix} \frac{M_{2,2}}{\det} & \frac{-M_{1,2}}{\det} \\ \frac{-M_{1,2}}{\det} & \frac{M_{1,1}}{\det} \end{bmatrix} \quad (2.15)$$

2.4.2 Second Half of Most Likely Path Calculation

$$\Sigma_1^{-1} R_0 y_0 + R_1^T \Sigma_2^{-1} y_2 \quad (2.16)$$

The first part of the second half of the MLP:

$$\begin{bmatrix} S1_{1,1} & S1_{1,2} \\ S1_{2,1} & S1_{2,2} \end{bmatrix}^{-1} \begin{bmatrix} R0_{1,1} & R0_{1,2} \\ R0_{2,1} & R0_{2,2} \end{bmatrix} \begin{bmatrix} t_0 \\ \theta_0 \end{bmatrix} \quad (2.17)$$

$$= \begin{bmatrix} S1i_{1,1} * R0_{1,1} + S1i_{1,2} * R0_{2,1} & S1i_{1,1} * R0_{1,2} + S1i_{1,2} * R0_{2,2} \\ S1i_{2,1} * R0_{1,1} + S1i_{2,2} * R0_{2,1} & S1i_{2,1} * R0_{1,2} + S1i_{2,2} * R0_{2,2} \end{bmatrix} \begin{bmatrix} t_0 \\ \theta_0 \end{bmatrix} \quad (2.18)$$

$$= \begin{bmatrix} S1i_{1,1} & S1i_{1,1} * R0_{1,2} + S1i_{1,2} \\ S1i_{2,1} & S1i_{2,1} * R0_{1,2} + S1i_{2,2} \end{bmatrix} \begin{bmatrix} t_0 \\ \theta_0 \end{bmatrix} \quad (2.19)$$

$$= \begin{bmatrix} S1i_{1,1} * t_0 + (S1i_{1,1} * R0_{1,2} + S1i_{1,2}) * \theta_0 \\ S1i_{2,1} * t_0 + (S1i_{2,1} * R0_{1,2} + S1i_{2,2}) * \theta_0 \end{bmatrix} \quad (2.20)$$

The second part of the second half of the MLP:

$$\begin{bmatrix} R1_{1,1} & R1_{1,2} \\ R1_{2,1} & R1_{2,2} \end{bmatrix}^T \begin{bmatrix} S2_{1,1} & S2_{1,2} \\ S2_{2,1} & S2_{2,2} \end{bmatrix}^{-1} \begin{bmatrix} t_2 \\ \theta_2 \end{bmatrix} \quad (2.21)$$

$$= \begin{bmatrix} R1_{1,1} * S2i_{1,1} + R1_{2,1} * S2i_{2,1} & R1_{1,1} * S2i_{1,2} + R1_{2,1} * S2i_{2,2} \\ R1_{1,2} * S2i_{1,1} + R1_{2,2} * S2i_{2,1} & R1_{1,2} * S2i_{1,2} + R1_{2,2} * S2i_{2,2} \end{bmatrix} \begin{bmatrix} t_2 \\ \theta_2 \end{bmatrix} \quad (2.22)$$

$$= \begin{bmatrix} S2i_{1,1} & S2i_{1,2} \\ R1_{1,2} * S2i_{1,1} + S2i_{2,1} & R1_{1,2} * S2i_{1,2} + S2i_{2,2} \end{bmatrix} \begin{bmatrix} t_2 \\ \theta_2 \end{bmatrix} \quad (2.23)$$

$$= \begin{bmatrix} S2i_{1,1} * t_2 + S2i_{1,2} * \theta_2 \\ (R1_{1,2} * S2i_{1,1} + S2i_{2,1}) * t_2 + (R1_{1,2} * S2i_{1,2} + S2i_{2,2}) * \theta_2 \end{bmatrix} \quad (2.24)$$

2.4.3 Finalized Calculation

Putting Equations 2.14, 2.20 and 2.24 together gives the final, though still a bit lengthy version of the optimized MLP calculation.

$$y_{mlp} = \frac{1}{\det} \begin{bmatrix} M_{2,2} & -M_{1,2} \\ -M_{2,1} & M_{1,1} \end{bmatrix} \left(\begin{array}{c} \left[\begin{array}{c} S1i_{1,1} * t_0 + (S1i_{1,1} * R0_{1,2} + S1i_{1,2}) * \theta_0 \\ S1i_{2,1} * t_0 + (S1i_{2,1} * R0_{1,2} + S1i_{2,2}) * \theta_0 \end{array} \right] + \\ \left[\begin{array}{c} S2i_{1,1} * t_2 + S2i_{1,2} * \theta_2 \\ (R1_{1,2} * S2i_{1,1} + S2i_{2,1}) * t_2 + (R1_{1,2} * S2i_{1,2} + S2i_{2,2}) * \theta_2 \end{array} \right] \end{array} \right) \quad (2.25)$$

2.5 Further Optimizations

Using the finalized calculation 2.25 above we can separate the most likely path formula into its two components y_t and y_θ by separating the matrix into rows.

$$y_{mlp} = \begin{pmatrix} t \\ \theta \end{pmatrix}$$

such that

$$y_t = \frac{M_{2,2}}{\det} \begin{pmatrix} S1i_{1,1} * t_0 + (S1i_{1,1} * R0_{1,2} + S1i_{1,2}) * \theta_0 + \\ S2i_{1,1} * t_2 + S2i_{1,2} * \theta_2 \end{pmatrix} - \frac{M_{1,2}}{\det} \begin{pmatrix} S1i_{2,1} * t_0 + (S1i_{2,1} * R0_{1,2} + S1i_{2,2}) * \theta_0 + \\ (R1_{1,2} * S2i_{1,1} + S2i_{2,1}) * t_2 + (R1_{1,2} * S2i_{1,2} + S2i_{2,2}) * \theta_2 \end{pmatrix}$$

and

$$y_\theta = \frac{-M_{2,1}}{\det} \begin{pmatrix} S1i_{1,1} * t_0 + (S1i_{1,1} * R0_{1,2} + S1i_{1,2}) * \theta_0 + \\ S2i_{1,1} * t_2 + S2i_{1,2} * \theta_2 \end{pmatrix} - \frac{-M_{1,1}}{\det} \begin{pmatrix} S1i_{2,1} * t_0 + (S1i_{2,1} * R0_{1,2} + S1i_{2,2}) * \theta_0 + \\ (R1_{1,2} * S2i_{1,1} + S2i_{2,1}) * t_2 + (R1_{1,2} * S2i_{1,2} + S2i_{2,2}) * \theta_2 \end{pmatrix}$$

Because, in this calculation we are looking for the "t-offset" to find a voxel number, we do not need the θ calculation. This allows us to remove several unnecessary

multiplications from the MLP. This simplifies to:

$$y_{mlp} = \begin{pmatrix} t \end{pmatrix}$$

2.5.1 Precalculations

If the depth fo the object is known precalculations can be used to remove redundant calculations. More specifically, R_0 , R_1 , Σ_1 and Σ_2 are known and all associated matrix-matrix multiplications and inverses can be calculated before transferring data and calculating the most likely path.

$$\begin{aligned} y_{mlp} &= (\Sigma_1^{-1} + R_1^T \Sigma_2^{-1} R_1)^{-1} (\Sigma_1^{-1} R_0 y_0 + R_1^T \Sigma_2^{-1} y_2) \quad (2.26) \\ &= \left(\begin{array}{c} \begin{bmatrix} S_{1,1} & S_{1,2} \\ S_{2,1} & S_{2,2} \end{bmatrix}^{-1} + \\ \begin{bmatrix} R_{1,1} & R_{1,2} \\ R_{2,1} & R_{2,2} \end{bmatrix}^T \begin{bmatrix} S_{2,1} & S_{2,2} \\ S_{2,1} & S_{2,2} \end{bmatrix}^{-1} \begin{bmatrix} R_{1,1} & R_{1,2} \\ R_{2,1} & R_{2,2} \end{bmatrix} \end{array} \right)^{-1} \\ &\quad \left(\begin{array}{c} \begin{bmatrix} S_{1,1} & S_{1,2} \\ S_{2,1} & S_{2,2} \end{bmatrix}^{-1} \begin{bmatrix} R_{0,1} & R_{0,2} \\ R_{0,1} & R_{0,2} \end{bmatrix} \begin{bmatrix} t_0 \\ \theta_0 \end{bmatrix} + \\ \begin{bmatrix} R_{1,1} & R_{1,2} \\ R_{2,1} & R_{2,2} \end{bmatrix}^T \begin{bmatrix} S_{2,1} & S_{2,2} \\ S_{2,1} & S_{2,2} \end{bmatrix}^{-1} \begin{bmatrix} t_2 \\ \theta_2 \end{bmatrix} \end{array} \right) \end{aligned}$$

Precalculating known matrix multiplications results in

$$y_{mlp} = P_1 (P_2 y_0 + P_3 y_2) \quad (2.27)$$

Using distributive property of matrix multiplication gives

$$y_{mlp} = P_4 y_0 + P_5 y_2 \quad (2.28)$$

Where P are the precalculated matrices

$$P_1 = (\Sigma_1^{-1} + R_1^T \Sigma_2^{-1} R_1)^{-1} \quad (2.29)$$

$$P_2 = \Sigma_1^{-1} R_0 \quad (2.30)$$

$$P_3 = R_1^T \Sigma_2^{-1} \quad (2.31)$$

$$P_4 = P_1 P_2 \quad (2.32)$$

$$= (\Sigma_1^{-1} + R_1^T \Sigma_2^{-1} R_1)^{-1} \Sigma_1^{-1} R_0 \quad (2.33)$$

$$P_5 = P_1 P_3 \quad (2.34)$$

$$= (\Sigma_1^{-1} + R_1^T \Sigma_2^{-1} R_1)^{-1} R_1^T \Sigma_2^{-1} \quad (2.35)$$

Expanding gives:

$$y_{mlp} = P_4 y_0 + P_5 y_2 \quad (2.36)$$

$$= \left(\begin{bmatrix} P_{41,1} & P_{41,2} \\ P_{42,1} & P_{42,2} \end{bmatrix} \begin{bmatrix} t_0 \\ \theta_0 \end{bmatrix} + \begin{bmatrix} P_{51,1} & P_{51,2} \\ P_{52,1} & P_{52,2} \end{bmatrix} \begin{bmatrix} t_2 \\ \theta_2 \end{bmatrix} \right) \quad (2.37)$$

which lowers the floating-point operation count to 14 (two matrix-vector multiplications and one vector addition) when storing the intermediate values.

2.5.2 Component Separation with Precalculations

Expanding the equations from Section 2.5.1 and separating the y_t from y_θ could yield further improvements and will not require the storage of intermediate results.

$$y_{mlp} = \left(\begin{bmatrix} P_{41,1} & P_{41,2} \\ P_{42,1} & P_{42,2} \end{bmatrix} \begin{bmatrix} t_0 \\ \theta_0 \end{bmatrix} + \begin{bmatrix} P_{51,1} & P_{51,2} \\ P_{52,1} & P_{52,2} \end{bmatrix} \begin{bmatrix} t_2 \\ \theta_2 \end{bmatrix} \right) \quad (2.38)$$

$$= \begin{bmatrix} (P_{41,1} * t_0 + P_{41,2} * \theta_0) + (P_{51,1} * t_2 + P_{51,2} * \theta_2) \\ (P_{42,1} * t_0 + P_{42,2} * \theta_0) + (P_{52,1} * t_2 + P_{52,2} * \theta_2) \end{bmatrix} \quad (2.39)$$

$$y_t = (P_{41,1} * t_0 + P_{41,2} * \theta_0) + (P_{51,1} * t_2 + P_{51,2} * \theta_2) \quad (2.40)$$

$$y_\theta = (P_{42,1} * t_0 + P_{42,2} * \theta_0) + (P_{52,1} * t_2 + P_{52,2} * \theta_2) \quad (2.41)$$

Calculating the expanded form will take the same number of floating-point operations at 14, but this form allows us to calculate only the y_t component (Eqn 2.40) which will take 7 floating-point operations per step per proton. This is a savings of 91% over the original 79 floating-point operations per step per proton, does not require as much memory and removes the synchronization requirement between steps. It also shows that only the first two rows of the $P4$ and $P5$ are needed. This reduction requires a new ordering of proton histories where they are grouped on depth traversed, (u_2), rather than the projection angle, θ .

2.5.3 P Row Separations

Because the calculation from Section 2.5.2 requires only the first row of $P4$ and $P5$ they can be separated to reduce time precalculating.

$$P_4 = (\Sigma_1^{-1} + R_1^T \Sigma_2^{-1} R_1)^{-1} \Sigma_1^{-1} R_0 \quad (2.42)$$

$$P_5 = (\Sigma_1^{-1} + R_1^T \Sigma_2^{-1} R_1)^{-1} R_1^T \Sigma_2^{-1} \quad (2.43)$$

Using Eq 2.15, Eq 2.19 and Eq 2.23, Eq 2.42 and Eq 2.43 become:

$$P_4 = \begin{bmatrix} \frac{M_{2,2}}{\det} & \frac{-M_{1,2}}{\det} \\ \frac{-M_{1,2}}{\det} & \frac{M_{1,1}}{\det} \end{bmatrix} \begin{bmatrix} S1i_{1,1} & S1i_{1,1} * R0_{1,2} + S1i_{1,2} \\ S1i_{1,2} & S1i_{1,2} * R0_{1,2} + S1i_{2,2} \end{bmatrix} \quad (2.44)$$

Multiplying for only the top row gives:

$$p4_{1,1} = \frac{M_{2,2}}{\det} * S1i_{1,1} + \frac{-M_{1,2}}{\det} * S1i_{1,2} \quad (2.45)$$

$$p4_{1,2} = \frac{M_{2,2}}{\det} * (S1i_{1,1} * R0_{1,2} + S1i_{1,2}) + \frac{-M_{1,2}}{\det} * (R0_{1,2} + S1i_{2,2}) \quad (2.46)$$

or

$$p4_{1,1} = \frac{1}{\det} (M_{2,2} * S1i_{1,1} - M_{1,2} * S1i_{1,2}) \quad (2.47)$$

$$p4_{1,2} = \frac{1}{\det} (M_{2,2} * (S1i_{1,1} * R0_{1,2} + S1i_{1,2}) - M_{1,2} * (R0_{1,2} + S1i_{2,2})) \quad (2.48)$$

and

$$P5 = \begin{bmatrix} \frac{M_{2,2}}{\det} & \frac{-M_{1,2}}{\det} \\ \frac{-M_{1,2}}{\det} & \frac{M_{1,1}}{\det} \end{bmatrix} \begin{bmatrix} S2i_{1,1} & S2i_{1,2} \\ R1_{1,2} * S2i_{1,1} + S2i_{1,2} & R1_{1,2} * S2i_{1,2} + S2i_{2,2} \end{bmatrix} \quad (2.49)$$

Multiplying for only the top row gives:

$$p5_{1,1} = \frac{M_{2,2}}{\det} * S2i_{1,1} + \frac{-M_{1,2}}{\det} * (R1_{1,2} * S2i_{1,1} + S2i_{1,2}) \quad (2.50)$$

$$p5_{1,2} = \frac{M_{2,2}}{\det} * S2i_{1,2} + \frac{-M_{1,2}}{\det} * (R1_{1,2} * S2i_{1,2} + S2i_{2,2}) \quad (2.51)$$

or

$$p5_{1,1} = \frac{1}{\det} (M_{2,2} * S2i_{1,1} - M_{1,2} * (R1_{1,2} * S2i_{1,1} + S2i_{1,2})) \quad (2.52)$$

$$p5_{1,2} = \frac{1}{\det} (M_{2,2} * S2i_{1,2} - M_{1,2} * (R1_{1,2} * S2i_{1,2} + S2i_{2,2})) \quad (2.53)$$

Removing unnecessary calculations within the precalculations will not have as much of an impact on the overall time as removing calculations in the iterative step, but it will contribute in reducing the total calculation time.

3. CALCULATING THE CONVEX HULL OF THE OBJECT

The object can be much more accurately reconstructed if its boundary is known. A good approximation for the boundary of the object can be obtained by calculating the convex hull of the object. The approximate location of the object is known to lie between the two inner detectors, shown in Figure 3.1. This system was modeled by the GEANT4 simulation [2]. The dotted circle represents the area covered by at least some of the proton paths while the dash-dotted circle represents the area receiving full beam coverage. The squares within the respective circles represent the possible discretized areas and the oval in the center represents the phantom.. According to the National Institute of Standards and Technology's stopping power and range tables for protons in various materials, a tolerance of about 0.300MeV should be sufficient to create a convex hull around the object. The paths of these protons are calculated using a straight line approximation from one inner detector to the other. The intersected voxels are then zeroed and the paths are discarded. The resulting hull is shown in Figure 3.3. Comparing this to the original phantom (Figure 3.2 [4]) it is found to produce a good approximation of the convex hull of the object.

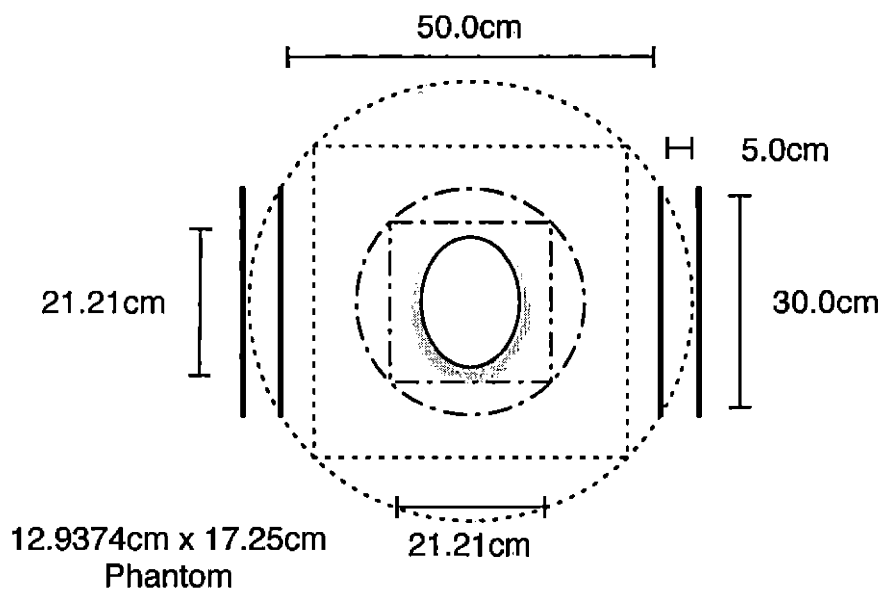


Fig. 3.1: Top down view of the pCT detector system.

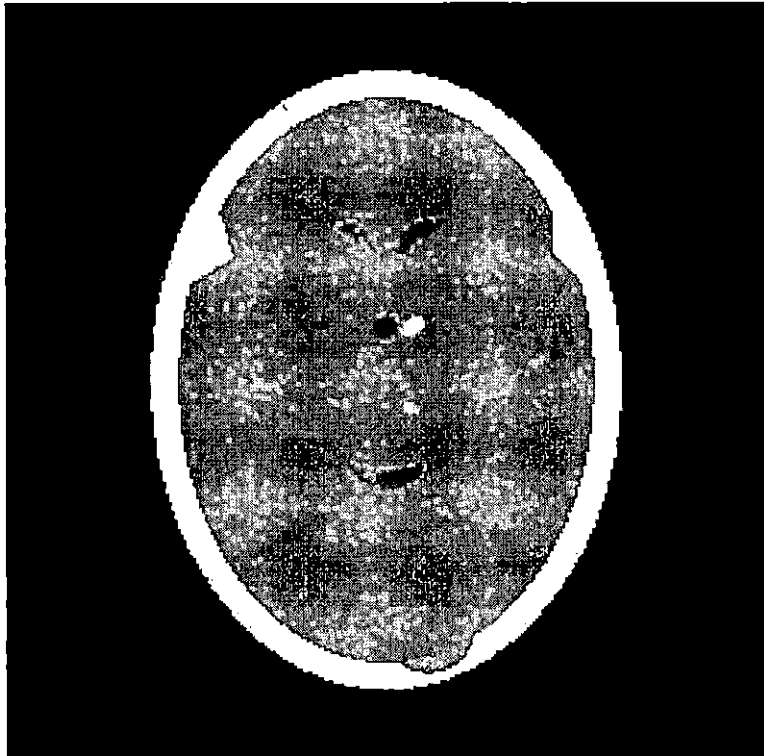


Fig. 3.2: The phantom to be reconstructed.

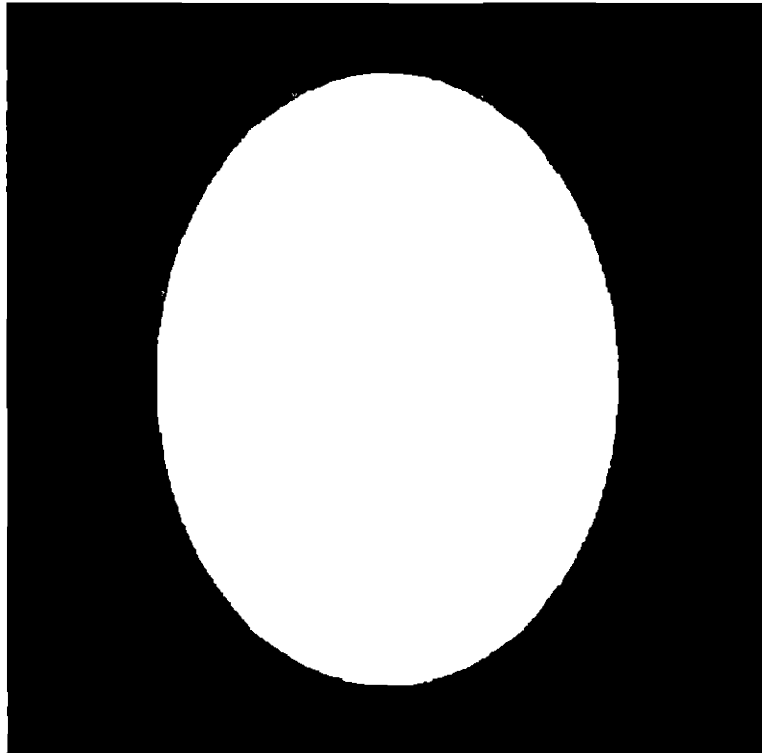


Fig. 3.3: Convex hull approximation of the object to be imaged using 0.300MeV as a tolerance value.

4. INTEGRAL RELATIVE ELECTRON DENSITY

When reconstructing an image x in the form $Ax = b$, A and b are required to be known to calculate x . In the case of pCT, A is the path matrix and b is the integral relative electron density. This chapter will show the process to convert from energy loss values to the integral relative electron density.

4.1 Energy Calculation

Given

$$\int_{E_{out}}^{E_{in}} \frac{dE}{F(E, I_{water})}$$

and

$$F(I, E(U)) = K \frac{1}{\beta^2(u)} \left[\ln \left(\frac{2m_e c^2}{I_{water}} \frac{\beta^2(u)}{1 - \beta^2(u)} \right) - \beta^2(u) \right]$$

gives

$$\int_{E_{out}}^{E_{in}} \frac{dE}{F(E, I_{water})} =$$

$$\int_{E_{out}}^{E_{in}} \frac{dE}{K \frac{1}{\beta^2(u)} \left[\ln \left(\frac{2m_e c^2}{I_{water}} \frac{\beta^2(u)}{1 - \beta^2(u)} \right) - \beta^2(u) \right]} \quad (4.1)$$

$$= \frac{1}{K} \int_{E_{out}}^{E_{in}} \frac{dE}{\frac{1}{\beta^2(u)} \left[\ln \left(\frac{2m_e c^2}{I_{water}} \right) + \ln \left(\frac{\beta^2(u)}{1 - \beta^2(u)} \right) - \beta^2(u) \right]} \quad (4.2)$$

where

$$\beta(u) = \sqrt{1 - \left(\frac{E_p}{E(u) + E_p} \right)^2}$$

and

$$\frac{1}{\beta^2(u)} = \frac{1}{1 - \left(\frac{E_p}{E(u) + E_p} \right)^2}$$

gives

$$\frac{1}{\beta^2(u)} = \frac{1}{1 - \left(\frac{E_p}{E(u) + E_p} \right)^2} \quad (4.3)$$

$$= \frac{1}{\left(1 + \frac{E_p}{E + E_p} \right) \left(1 - \frac{E_p}{E + E_p} \right)} \quad (4.4)$$

$$= \frac{1}{\left(\frac{E + E_p}{E + E_p} + \frac{E_p}{E + E_p} \right) \left(\frac{E + E_p}{E + E_p} - \frac{E_p}{E + E_p} \right)} \quad (4.5)$$

$$= \frac{1}{\left(\frac{E + 2E_p}{E + E_p} \right) \left(\frac{E}{E + E_p} \right)} \quad (4.6)$$

$$= \frac{1}{\frac{E^2 + 2EE_p}{(E + E_p)^2}} \quad (4.7)$$

$$= \frac{(E + E_p)^2}{E^2 + 2EE_p} \quad (4.8)$$

$$= \frac{E^2 + 2EE_p + E_p^2}{E^2 + 2EE_p} \quad (4.9)$$

$$= \frac{E^2 + 2EE_p}{E^2 + 2EE_p} + \frac{E_p^2}{E^2 + 2EE_p} \quad (4.10)$$

$$= 1 + \frac{E_p^2}{E^2 + 2EE_p} \quad (4.11)$$

Starting from Equation 4.8 above:

$$\beta^2(u) = \frac{E^2 + 2EE_p}{(E + E_p)^2} \quad (4.12)$$

$$= \frac{E^2 + 2EE_p + E_p^2 - E_p^2}{E^2 + 2EE_p + E_p^2} \quad (4.13)$$

$$= 1 - \frac{E_p^2}{(E + E_p)^2} \quad (4.14)$$

Adding in the $\beta^2(u)$ equations from above gives:

$$= \frac{1}{K} \int_{E_{out}}^{E_{in}} \frac{dE}{\left(1 + \frac{E_p^2}{E^2 + 2EE_p}\right) \left[\ln\left(\frac{2m_e c^2}{I_{water}}\right) + \ln\left(\frac{\left(1 - \frac{E_p^2}{(E+E_p)^2}\right)}{1 - \frac{E_p^2}{(E+E_p)^2}}\right) - \left(1 - \frac{E_p^2}{(E+E_p)^2}\right) \right]} \quad (4.15)$$

$$= \frac{1}{K} \int_{E_{out}}^{E_{in}} \frac{dE}{\left(1 + \frac{E_p^2}{E^2 + 2EE_p}\right) \left[\ln\left(\frac{2m_e c^2}{I_{water}}\right) + \ln\left(\frac{1}{\frac{E_p^2}{(E+E_p)^2}} - \frac{E_p^2}{(E+E_p)^2}\right) - \left(1 - \frac{E_p^2}{(E+E_p)^2}\right) \right]} \quad (4.16)$$

$$= \frac{1}{K} \int_{E_{out}}^{E_{in}} \frac{dE}{\left(1 + \frac{E_p^2}{E^2 + 2EE_p}\right) \left[\ln\left(\frac{2m_e c^2}{I_{water}}\right) + \ln\left(\frac{1}{\frac{E_p^2}{(E+E_p)^2}} - 1\right) - \left(1 - \frac{E_p^2}{(E+E_p)^2}\right) \right]} \quad (4.17)$$

$$= \frac{1}{K} \int_{E_{out}}^{E_{in}} \frac{dE}{\left(1 + \frac{E_p^2}{E^2 + 2EE_p}\right) \left[\ln\left(\frac{2m_e c^2}{I_{water}}\right) + \ln\left(\frac{(E+E_p)^2}{E_p^2} - 1\right) - \left(1 - \frac{E_p^2}{(E+E_p)^2}\right) \right]} \quad (4.18)$$

$$= \frac{1}{K} \int_{E_{out}}^{E_{in}} \frac{dE}{\left(1 + \frac{E_p^2}{E^2 + 2EE_p}\right) \left[\ln\left(\frac{2m_e c^2}{I_{water}}\right) + \ln\left(\frac{(E^2 + 2EE_p + E_p^2 - E_p^2)}{E_p^2} - \frac{E_p^2}{E_p^2}\right) - \left(1 - \frac{E_p^2}{(E+E_p)^2}\right) \right]} \quad (4.19)$$

$$= \frac{1}{K} \int_{E_{out}}^{E_{in}} \frac{dE}{\left(1 + \frac{E_p^2}{E^2 + 2EE_p}\right) \left[\ln\left(\frac{2m_e c^2}{I_{water}}\right) + \ln\left(\frac{(E^2 + 2EE_p + E_p^2 - E_p^2)}{E_p^2} - \frac{E_p^2}{E_p^2}\right) - \left(1 - \frac{E_p^2}{(E+E_p)^2}\right) \right]} \quad (4.20)$$

$$= \frac{1}{K} \int_{E_{out}}^{E_{in}} \frac{dE}{\left(1 + \frac{E_p^2}{E^2 + 2EE_p}\right) \left[\ln\left(\frac{2m_e c^2}{I_{water}}\right) + \ln\left(\frac{(E^2 + 2EE_p)}{E_p^2} - \left(1 - \frac{E_p^2}{(E+E_p)^2}\right)\right) \right]} \quad (4.21)$$

$$= \frac{1}{K} \int_{E_{out}}^{E_{in}} \frac{dE}{\left(1 + \frac{E_p^2}{E^2 + 2EE_p}\right) \left[\ln\left(\frac{2m_e c^2}{I_{water}}\right) + \ln\left(\frac{E(E+2E_p)}{E_p^2} - \left(1 - \frac{E_p^2}{(E+E_p)^2}\right)\right) \right]} \quad (4.22)$$

$$= \frac{1}{K} \int_{E_{out}}^{E_{in}} \frac{dE}{\left(1 + \frac{E_p^2}{E^2 + 2EE_p}\right) \left[\ln\left(\frac{2m_e c^2}{I_{water}}\right) + \ln(E) + \ln(E+2E_p) - 2\ln(E_p) - 1 + \frac{E_p^2}{(E+E_p)^2} \right]} \quad (4.23)$$

Equation 4.23 is now suitable for computation on a serial CPU program or a parallel GPU program. Because the data input and output in calculating the integral relative electron density is completely independent, it lends itself very well to GPU computation.

4.2 General Purpose Graphics Processing Unit Speedup of Integral Relative Electron Density Calculation

Shown below is a table of CPU and GPU times for the integral relative electron density equation (Equation 4.23). Times are in milliseconds and are averages of times for a given number of elements over 1000 iterations.

Table 4.1 shows that before one million proton energies the GPU doesn't have

Number of Elements	CPU	GPU	Speedup
100	0.034	0.100	0.34
1000	0.329	0.102	3.2254902
10k	3.803	0.105	36.219048
100k	38.131	0.185	206.11351
1M	384.776	0.860	447.41395
2M	767.827	1.423	539.58327
3M	1205.037	2.564	469.98323
4M	1527.700	3.057	499.73831
5M	1919.564	3.856	497.81224
6M	2293.294	4.550	504.02066
7M	3666.551	N/A	N/A

Tab. 4.1: Graphics Processing Unit Speedup of Integral Relative Electron Density Calculation

much, if any, advantage over the CPU. The reason for this is the GPU initialization time. After six million elements the GPU ran out of memory and returned no values. With batches of five million proton energies, 100 million proton energies could be calculated in 77.12ms while on a CPU, 100 million proton energies would take 38.39 seconds to calculate.

5. A BIN SORT BASED FAST MOST LIKELY PATH ALGORITHM

Removal of over 90% of MLP calculations (Equation 2.40) is only possible if the distance the proton traversed (u_2) is constant or varies no more than a given tolerance. Because u_2 needs to be calculated for every proton before the Σ or R matrices are calculated it would not be much more work to sort the proton histories based on their traversed depths.

5.1 *Choosing Bins*

The number of bins, and therefore number of different Σ and R matrices, depends ultimately on the rate of change of the covariance matrices (Σ_1^{-1} and Σ_2^{-1}). In this case the covariance changes enough to need an update every 0.5mm. Using this information it can be said that for an object with a maximum depth of 20.0cm there are 400 unique bins, and therefore Σ matrices. A depth step increment u will always be chosen to be smaller than the voxel width. In this case 0.125mm step increments are used. This will create 1600 unique R matrices, but because they can be binned with the Σ matrices only 400 bins, and thus P matrices, are required. For a large number of protons, 100 million or more, this type of binning would allow for a large reduction in calculations, requiring one set of Σ and R matrices per 250,000 proton histories instead of the current one set of Σ and R matrices per one proton history. A distribution of proton histories as a function of depth is shown in Figure 5.1.

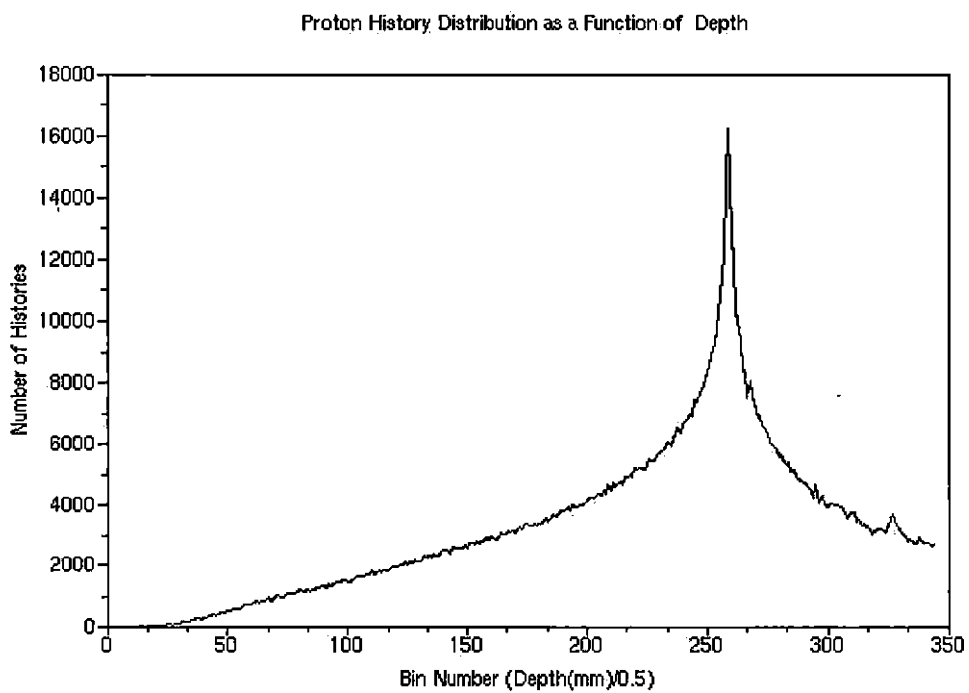


Fig. 5.1: Proton history distribution as a function of depth.

5.2 Accuracy of Binning

Currently there are no tests to show if proton history organization with this type of ordering has an effect on the accuracy of the reconstructions. There is a reconstruction algorithm that uses ordered subsets as blocks in the simultaneous algebraic reconstruction technique called Ordered Subsets Simultaneous Algebraic Reconstruction Technique (OS-SART). In their paper, Jiang and Wang showed that by creating ordered subsets based on projection angle they were able to deal well with noisy data.

5.3 Reconstruction Algorithms

Because of the variety of reconstruction algorithms available (Algorithms A.1.1, A.2.1, A.2.2, A.3.1, A.3.2, A.3.3, A.4.1, A.4.2), one can be chosen that allows for different sized blocks. The algorithms that stand out are the String-Averaging Projections (SAP) algorithm and Block Iterative algorithms such as Block-Iterative Projections (BIP), Block-Iterative Component Averaging (BICAV), Component-Averaged Row-Action Projections (CARP) and Diagonally Relaxed Orthogonal Projections (DROP). An algorithm of note is the Block-Iterative Projections algorithm because of its simplicity. This algorithm does not use component averaging which means it does not require information on the number of protons passing through a particular voxel. However, this also means it might be more sensitive to non-uniform coverage by the proton beam. Another algorithm worth mentioning is the Ordered Subsets Simultaneous Algebraic Reconstruction Technique (OS-SART) algorithm which was developed by Anderson and Kak (SART) and Jiang and Wang (Ordered Subsets) for use with subsets made up of rays from single projection angles. This algorithm was found to reduce noise in the reconstructed images. The subsets can be changed from rays from a single projection angle to rays traversing a single depth. For example, instead of taking all rays with a 0° projection angle, we will take rays from all projection angles with u_2 of 20.0cm.

5.4 On the Fly Most Likely Path Calculation With Block Iterative Reconstruction Algorithm

The MLP and reconstruction algorithms are going to be merged to be done in one step to reduce data transfers to the GPU.

Example using Block Iterative Projections (BIP)

$$x^{k+1} = x^k + \lambda_k \sum_{i \in B_t} w^k(i) \frac{b_i - \langle a^i, x^k \rangle}{\|a^i\|_2^2} a^i \quad (5.1)$$

where $w^k(i) = 1/v_t$ with $v_t =$ the number of elements in block t gives

$$x^{k+1} = x^k + \lambda_k \sum_{i \in B_t} \frac{1}{v_t} \frac{b_i - \langle a^i, x^k \rangle}{\|a^i\|_2^2} a^i \quad (5.2)$$

adding the most likely path from Equation 2.40

$$y_{t_u} = (P_{4,1,1_u} * t_0 + P_{4,1,2_u} * \theta_0) + (P_{5,1,1_u} * t_2 + P_{5,1,2_u} * \theta_2) \quad (5.3)$$

$$a_{t_u}^i = y_{t_u} \quad (5.4)$$

$$\|a^i\|_2^2 = \frac{n_s}{w_v} \quad (5.5)$$

$$\langle a^i, x^k \rangle = x_{y_{t_u}}^k \quad (5.6)$$

$$x^{k+1} = x^k + \lambda_k \sum_{i \in B_t} \frac{1}{v_t} \frac{b_i - (x_{y_{t_u}}^k * w_v)}{\frac{n_s}{w_v}} \quad (5.7)$$

$$(5.8)$$

where $\{P_{4,1,1}\}_{u=0}^{B_{t_d}}$, $\{P_{4,1,2}\}_{u=0}^{B_{t_d}}$, $\{P_{5,1,1}\}_{u=0}^{B_{t_d}}$ and $\{P_{5,1,2}\}_{u=0}^{B_{t_d}}$ are the required elements from the Σ and R matrices to perform the MLP calculation and B_{t_d} is the maximum depth of the current block. In this case $\frac{n_s}{w_v}$ is equal to the number of steps taken through the object divided by the voxel width and, consequently, is the same for every history in the block. Using on the fly MLP calculation not only eliminates the creation and storage of the path matrix A , but it eliminates the creation of the entire row a_i . Instead, the path is created as a series of coordinates relative to the image x . This allows the coordinates calculated by y_{t_u} to be used as an index of x and, in turn, removes the need to multiply by a_i .¹

¹ $x_{y_{t_u}}^k$ is rotated via givens rotation according to the angle of the gantry.

6. SUMMARY OF FINDINGS

For this thesis I researched proton computed tomography image reconstruction using a most likely path approach. Research was conducted to calculate speed increases from algorithm optimization as well as hardware acceleration using general purpose graphics processors (GPGPUs). Through algorithm optimization in Chapter 3 many redundant calculations were removed. The phantom to be reconstructed 3.2 has several different densities as well as several different shapes

6.1 Time Savings of Precalculations

When 90% of the MLP is precalculated (Section 2.5.1) the time savings are easily apparent. One cycle using a naïve MLP takes approximately 90 minutes while one cycle using the optimized MLP with precalculations takes only one minute. This is almost 100 times faster or two orders of magnitude faster.

6.2 Space Savings of Sparse Matrices

In this thesis an on the fly most likely path calculation is suggested where the path for each proton is calculated as needed and then discarded. A different approach to the $Ax = b$ problem was introduced where A is never stored. Usually, when faced with a problem of this type ($Ax = b$, where A is very sparse) the matrix is stored in a compressed sparse format. Storing a matrix this way greatly reduces the amount of space required by the matrix. For example, a matrix with 5 million histories and a resolution of 512×512 would require $5 \times 10^6 \times 512 \times 512 \times 4$ or 5.243×10^{12} bytes for single precision floating point numbers (twice as much for double precision) or about 5 terabytes. With pCT data expected to reach sizes of 1

billion histories and 3-D resolutions of $1024 \times 1024 \times 512$, the space required grows to $1 \times 10^9 \times 1024 \times 1024 \times 512 \times 4$ or 2.147×10^{18} or about two exabytes for single precision floating point numbers. These numbers are assuming data is stored in dense matrix form. For sparse matrices of these same sizes, information is needed on how sparse the matrix is. In this case a good representation of the number of voxels a proton passes through is the square root of the number of voxels parallel to the beam. This would reduce the size of the aforementioned matrices to 1.024×10^{10} or 10 gigabytes and 4.096×10^{12} or about four terabytes respectively. While this is a significant reduction, transferring this much data can be very costly which is where the need for an on the fly most likely path algorithm arose. This algorithm, explained in Section 5.4, greatly reduces the space required for computing the MLP and the reconstructed image. The space savings of the on the fly algorithm requires more computation because the MLP is not stored, but the time saved by not having to move terabytes of data more than makes up for it.

6.3 Relaxation Parameter as a Function of Depth

The difference between the inner and outer sections of the reconstructed image using the depth based imaging algorithm (Figure 6.1) brings about the need for a depth based relaxation parameter. If the blurring of the center of the image can be controlled a more accurate image can be produced.

Working from the on the fly MLP algorithm in Section 5.4, an example using Block Iterative Projections (BIP) with a depth based relaxation parameter can be

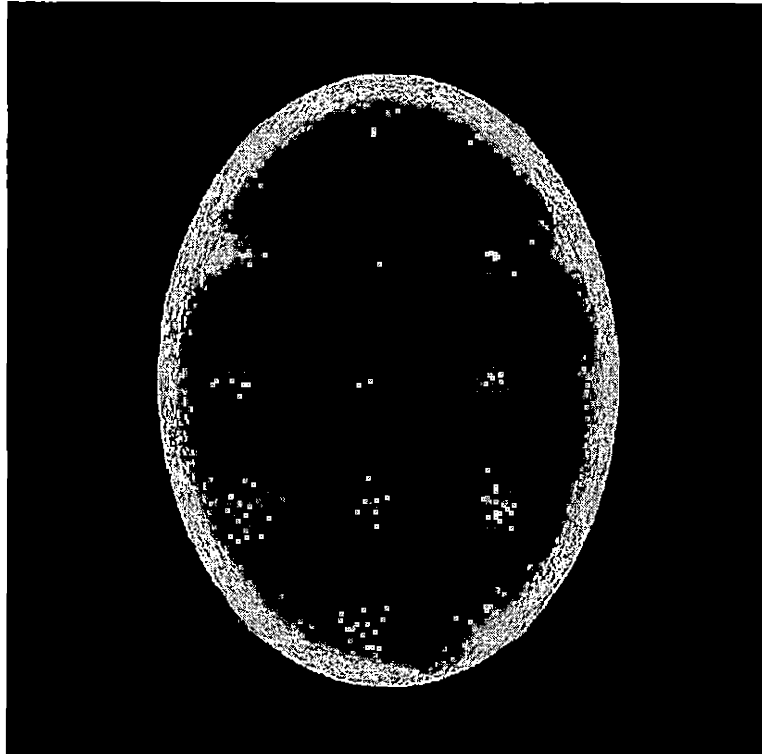


Fig. 6.1: Phantom reconstructed after ten cycles with a relaxation parameter of 3.0.

shown as:

$$y_{t_u} = (P4_{1,1_u} * t_0 + P4_{1,2_u} * \theta_0) + (P5_{1,1_u} * t_2 + P5_{1,2_u} * \theta_2) \quad (6.1)$$

$$a_u^i = y_{t_u} \quad (6.2)$$

$$\|a^i\|_2^2 = \frac{n_s}{w_v} \quad (6.3)$$

$$\langle a^i, x^k \rangle = x_{y_{t_u}}^k \quad (6.4)$$

$$x^{k+1} = x^k + \lambda_k d_i \sum_{i \in B_t} \frac{1}{v_t} \frac{b_i - (x_{y_{t_u}}^k * w_v)}{\frac{n_s}{w_v}} \quad (6.5)$$

where $\{P4_{1,1}\}_{u=0}^{B_{t_d}}$, $\{P4_{1,2}\}_{u=0}^{B_{t_d}}$, $\{P5_{1,1}\}_{u=0}^{B_{t_d}}$ and $\{P5_{1,2}\}_{u=0}^{B_{t_d}}$ are the required elements from the Σ and R matrices to perform the MLP calculation, B_{t_d} is the maximum depth of the current block and d_i is the maximum depth of the current proton. In this case $\frac{n_s}{w_v}$ is equal to the number of steps taken through the object divided by the voxel width and, consequently, is the same for every history in the block.

By making the relaxation parameter a function of depth simply by multiplying the proton path update by the maximum depth of that proton the edges of the image are smoothed while the center of the image in Figure 6.2 is made slightly sharper.

6.4 Future Work

There is still much work to be done in the field of pCT. The following subsections show work that is yet to be done as well as some new projects that have come about as a result of this research.

6.4.1 Faster Most Likely Path Calculation

The most likely path in this thesis was calculated using a single core of one CPU. Migrating this code to a GPU could yield significant improvements in speed. Small scale tests were performed on migrating the MLP to the GPU and it was found that one cycle took approximately 36.5ms. This is a further three orders of magnitude

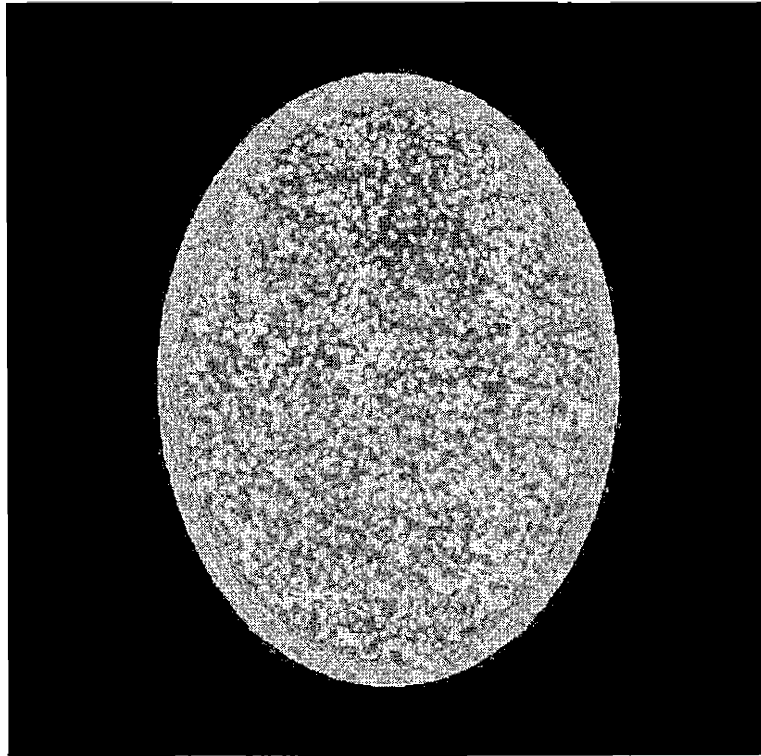


Fig. 6.2: Phantom reconstructed after ten cycles with a relaxation parameter of 0.3 multiplied by the depth in cm.

increase in speed. It is possible that data transfers and image updates will increase this time and thus decreasing the speedup.

6.4.2 Bin Size Calculation and Relaxation Parameter Testing

Although a bin sort based fast MLP algorithm was introduced in this thesis in Chapter 5, it will still require much more research to make it both faster and more accurate. The bin order can be looked at more closely to see if a random selection of bins would provide a better image or if ordering bins based on capacity instead of object traversal length.

6.4.3 Applying the Bin Sort Based Fast Most Likely Path Algorithm to More Algorithms

The images produced using the bin sort based fast MLP algorithm introduced in this thesis in Chapter 5 are not ideal, but show promising results. This algorithm could be applied to other reconstruction algorithms such as Kaczmarz, Cimmino, CAV, BICAV, DROP, OS-SART, SAP and CARP. Many of these algorithms have more tuning parameters such as component averaging (Algorithms A.2.2, A.3.1, A.3.2, A.3.3, A.4.2) and string averaging (Algorithms A.4.1 and A.4.2)

6.4.4 Research on Applications of General Purpose Graphics Processing Units in Clusters

The data parallel nature of GPU programming as well as the substantial speedup over CPU computation opens the possibility of having cluster nodes with graphics processors. With three orders of magnitude increase in speed and code very similar to C/C++, adding GPUs to a cluster could yield very good performance per dollar spent, especially if the cluster already has nodes with graphics card slots.

6.4.5 *On-Line Imaging*

While it has been shown that on-line imaging is certainly feasible according to the time required per reconstruction cycle, there is a potential problem. In order to perform data cuts on the large angle histories the standard deviation must be calculated which requires a complete dataset. This problem could be remedied by calculating the standard deviation of the angles of very large datasets and using this information as a basis for on-line data cuts. The image produced using this precalculated standard deviation may not be good enough for treatment planning, but it should be sufficient for patient location and beam alignment.

7. CONCLUSIONS

In this thesis it was shown that GPGPUs can be used to significantly reduce computation time in data parallel applications (Table 4.1). It was also shown that exploiting the structure of the data being processed in a sparse iterative solver, such as the algebraic reconstruction technique, the time it takes to complete a reconstruction can be significantly reduced. This reduction in reconstruction time brings pCT into clinically accepted time for reconstruction.

APPENDIX A
RECONSTRUCTION ALGORITHMS

Mentioned in Section 5.4 was the block iterative projections (BIP) method for image reconstruction. This method was chosen primarily for its simplicity and image quality. Below are some other algorithms sorted by type.

A.1 Fully Sequential Algorithms

A fully sequential algorithm will update the image after every proton history. They tend to be slow and produce a somewhat grainy image.

A.1.1 The Fully Sequential Algebraic Reconstruction Technique (Kaczmarz)

Given x^k compute x^{k+1} by:

$$x^{k+1} = x^k + \lambda_k \frac{b_i - \langle a^i, x^k \rangle}{\|a^i\|_2^2} a^i \quad (\text{A-1})$$

where λ_k are the user defined relaxation parameters, b_i is the integral relative electron density of the row being calculated and a^i is the vector of voxels the proton passed through.

A.2 Fully Simultaneous Algorithms

Fully Simultaneous algorithms are easy to parallelize, but suffer from a slow convergence rate. They update the image after all proton histories in a cycle.

A.2.1 The Fully Simultaneous Algebraic Reconstruction Technique (Cimmino)

Given x^k compute x^{k+1} by:

$$x^{k+1} = x^k + \frac{\lambda_k}{m} \sum_{i=1}^m \frac{b_i - \langle a^i, x^k \rangle}{\|a^i\|_2^2} a^i \quad (\text{A-2})$$

where λ_k are the user defined relaxation parameters, b_i is the integral relative electron density of the row being calculated, a^i is the vector of voxels the proton passed through and m is the total number of proton histories.

A.2.2 Fully Simultaneous Component Averaging (CAV)

Given x^k compute x^{k+1} for $j = 1, 2, \dots, n$ by:

$$x_j^{k+1} = x_j^k + \frac{\lambda_k}{s_j} \sum_{i=1}^m \frac{b_i - \langle a^i, x^k \rangle}{\|a^i\|_2^2} a_j^i \quad (\text{A-3})$$

where λ_k , b_i and m are the same as above, a_j^i is the j -th component of the i -th row of A , s_j is the number of non-zero elements in the j -th column of A . This version of CAV uses non-orthogonal projections, but they can be made orthogonal with the use of a "sparsity weight" $w(i)$ such that

$$w_i = \frac{1}{\sum_{j=1}^n s_j (a_{ij})^2} \quad (\text{A-4})$$

which gives the iterative step

$$x^{k+1} = x^k + \lambda_k \sum_{i=1}^m w_i (b_i - \langle a^i, x^k \rangle) a_j^i \quad (\text{A-5})$$

or

$$x^{k+1} = x^k + \lambda_k \sum_{i=1}^m \frac{b_i - \langle a^i, x^k \rangle}{\sum_{j=1}^n s_j (a_{ij})^2} a_j^i \quad (\text{A-6})$$

A.3 Block Iterative Algorithms

Block iterative algorithms share the speed of convergence of the fully sequential algorithms and the parallelizability of the fully simultaneous algorithms.

A.3.1 Block Iterative Component Averaging (BICAV)

This algorithm takes BIP and adds component averaging to speed convergence. Let $\mathbb{B} = \{B_1, \dots, B_t\}$ with t being the number of blocks. In this case w_i is

$$w_i = \frac{1}{\sum_{l=1}^t s_l^i (a_l^i)^2} \quad (\text{A-7})$$

Given x^k compute x^{k+1} by:

$$x^{k+1} = x^k + \lambda_k \sum_{i \in B_t} w_i (b_i - \langle a^i, x^k \rangle) a^i \quad (\text{A-8})$$

where λ_k , b_i , m and a_j^i are the same as above. It can also be shown as

$$x^{k+1} = x^k + \lambda_k \sum_{i \in B_t} \frac{b_i - \langle a^i, x^k \rangle}{\sum_{l=1}^n s_l^i (a_l^i)^2} a_j^i \quad (\text{A-9})$$

A.3.2 The Block Iterative Diagonally Relaxed Orthogonal Projections Algorithm (DROP)

The aim of the DROP algorithm is to improve the initial convergence with the use of component-dependent weighting.

$$x^{k+1} = x^k + \lambda_k U_{t(k)} \sum_{i \in I_{t(k)}} w_i \frac{b_i - \langle a^i, x^k \rangle}{\|a^i\|^2} a^i \quad (\text{A-10})$$

where $U_{t(k)} = \text{diag}(\min(1, 1/s_l^i))$ with $\{s_l^i\}_{l=1}^n$ is the number of non-zero elements $a_l^i \neq 0$ in the l -th column of the t -th block of the matrix A given by

$$A_t = \begin{pmatrix} a^{t_1} \\ a^{t_2} \\ \vdots \\ a^{t_m(t)} \end{pmatrix} \quad (\text{A-11})$$

A.3.3 The Ordered Subsets Simultaneous Algebraic Reconstruction Technique (OS-SART)

This algorithm was used to order the block by projection angle and found it reduced noise among the images produced.

$$x_j^{k+1} = x_j^k + \lambda_k \left(\frac{1}{\sum_{i \in I_{t(k)}} a_j^i} \right) \sum_{i \in I_{t(k)}} \frac{b_i - \langle a^i, x^k \rangle}{\sum_{j=1}^n a_j^i} a^i \quad (\text{A-12})$$

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

A.4 String Averaging Algorithms

Unlike block iterative algorithms, which are simultaneous within blocks, string averaging algorithms are sequential within blocks and are then averaged to create the reconstructed image.

A.4.1 The String Averaging Projections Algorithm (SAP)

Given x^k , for each $t = 1, 2, \dots, M$, set $y^0 = x^k$ and calculate for $i = 0, 1, \dots, m(t)-1$,

$$y^{i+1} = y^i + \lambda_i \frac{b_i - \langle a^i, x^k \rangle}{\|a^i\|_2^2} a^i \quad (\text{A-13})$$

and let $y^t = y^{m(t)}$ for each $t = 1, 2, \dots, M$. Then, calculate the next iterate by:

$$x^{k+1} = \sum_{t=1}^M w_t y^t \quad (\text{A-14})$$

A.4.2 Component Averaged Row Projection (CARP)

This component averaged row projections algorithm is similar to the string averaging projections algorithm above (Algorithm A.4.1), but with component averaging included. Given x^k , for each $t = 1, 2, \dots, M$, set $y^0 = x^k$ and calculate for $i = 0, 1, \dots, m(t) - 1$,

$$y^{i+1} = y^i + \lambda_i \frac{b_i - \langle a^i, x^k \rangle}{\|a^i\|_2^2} a^i \quad (\text{A-15})$$

and let $y^t = y^{m(t)}$ for each $t = 1, 2, \dots, M$. Then, calculate the next iterate by:

$$x_j^{k+1} = \frac{1}{s_j^t} \sum_{t=1}^M y_j^t \quad (\text{A-16})$$

APPENDIX B
MATRIX OPERATION COSTS

B.5 2×2 Matrix-Matrix Multiplication

Multiplying two 2×2 matrices together requires 12 floating-point operations as shown below. Four additions and 8 multiplications.

$$\begin{aligned} \begin{bmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{bmatrix} &= \begin{bmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{bmatrix} \begin{bmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{bmatrix} \\ &= \begin{bmatrix} A_{1,1} * B_{1,1} + A_{1,2} * B_{2,1} & A_{1,1} * B_{1,2} + A_{1,2} * B_{2,2} \\ A_{2,1} * B_{1,1} + A_{2,2} * B_{2,1} & A_{2,1} * B_{1,2} + A_{2,2} * B_{2,2} \end{bmatrix} \end{aligned}$$

B.6 2×2 Sparse Matrix-Matrix Multiplication

Multiplying a 2×2 matrix and a 2×2 sparse matrix together requires four floating-point operations as shown below. Two additions and two multiplications.

$$\begin{aligned} \begin{bmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{bmatrix} &= \begin{bmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{bmatrix} \begin{bmatrix} 1 & B_{1,2} \\ 0 & 1 \end{bmatrix} \\ &= \begin{bmatrix} A_{1,1} * 1 + A_{1,2} * 0 & A_{1,1} * B_{1,2} + A_{1,2} * 1 \\ A_{2,1} * 1 + A_{2,2} * 0 & A_{2,1} * B_{1,2} + A_{2,2} * 1 \end{bmatrix} \\ &= \begin{bmatrix} A_{1,1} & A_{1,1} * B_{1,2} + A_{1,2} \\ A_{2,1} & A_{2,1} * B_{1,2} + A_{2,2} \end{bmatrix} \end{aligned}$$

B.7 2×2 Matrix-Vector Multiplication

Multiplying a 2×2 matrix and a vector together requires 6 floating-point operations as shown below. Two additions and four multiplications.

$$\begin{aligned} \begin{bmatrix} C_1 \\ C_2 \end{bmatrix} &= \begin{bmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{bmatrix} \begin{bmatrix} B_1 \\ B_2 \end{bmatrix} \\ &= \begin{bmatrix} A_{1,1} * B_1 + A_{1,2} * B_2 \\ A_{2,1} * B_1 + A_{2,2} * B_2 \end{bmatrix} \end{aligned}$$

B.8 2×2 Matrix Inverse

Taking the inverse of a 2×2 matrix requires 7 floating-point operations (three for the determinant and one for dividing each element of the matrix by the determinant). Taking the inverse also assumes the matrix is non-singular which is the case for this calculation.

The inverse of a 2×2 matrix can be shown as:

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix}^{-1} = \frac{1}{ad - bc} \begin{bmatrix} d & -b \\ -c & a \end{bmatrix}$$

where $ad - bc$ is the determinant of the matrix.

APPENDIX C
SOURCE CODE

This chapter contains the source code that was used to process the data as well as reconstruct the object.

C.9 Reconstruction Code

```
//Scott McAllister
//Thesis
//CUDA functions for pCT reconstruction problem

// includes , system
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>

// includes , project
#include <cutil.h>

// includes , kernels
#include <reconstruction_kernel.cu>

#define PROJANGLES 180
#define HISTORIES 18000
#define PI 3.14159265
FILE *input;
FILE *convex_hull;
FILE *bin_data;
FILE *reconstruction;

////////////////////////////////////
// declaration , forward
void runTest( int argc, char** argv);
double gaussian_quadrature( int n, double(*function)(double val),
    double a, double b);
void electron_density_CPU(int size, float *E_in, float *E_out,
    float *output, float *u_temp1, float *u_temp2, float *temp1,
    float *temp2);
void MLP_CPU(int size, float *E_in, float *E_out, float *output,
    float *u_temp1, float *u_temp2, float *temp1, float *temp2);
void mat.inverse(float A[2][2], float A_inv[2][2]);
void mat.mult(int t1, int t2, float A[2][2], float B[2][2],
    float output[2][2]);
```

```

void mat_vec_mult(float A[2][2], float B[2], float output[2]);
void mat_add(float A[2][2], float B[2][2], float output[2][2]);
void vec_add(float A[2], float B[2], float output[2]);
//Scattering matrix elements
double E(double u);
double beta_squared_p_squared_inv(double u);
double s1_beta2_p2_u2(double u);
double s1_beta2_p2(double u);
double s1_beta2_p2_u(double u);
double s2_beta2_p2_u2(double u);
double s2_beta2_p2(double u);
double s2_beta2_p2_u(double u);

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// Program main
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
int
main( int argc, char** argv)
{
    printf("Start\n");
    runTest( argc, argv);

    CUT_EXIT(argc, argv);
}

//variables
float u_0 = 0;
float u_2;
float X_0 = 36.083; // cm
float E_0 = 13.6;
double E_p = 938.272; // MeV
double C = 299792458; // m/s
double eV = 1.602176487e-19;
double MeV = 1e6*eV;

void g_rot(float x, float y, float theta, float x_p, float y_p)
{
    //performs a 2-D Givens rotation
    x_p = x*cos(theta)-y*sin(theta);
    y_p = x*sin(theta)+y*cos(theta);
}

void QR_solve(double M[2][2], double S[2][2], double x[2][2])

```

```

{
// Mx = S;
double R[2][2];
double z[2][2];
double v[2];
double t[2];
v[0] = M[0][0] - sqrt(pow(M[0][0],2)+pow(M[1][0],2));
v[1] = M[1][0];
double temp = sqrt(pow(v[0],2)+pow(v[1],2));
v[0] /= temp;
v[1] /= temp;
R[0][0] = M[0][0] - 2 * v[0] * (v[0] * M[0][0] + v[1] * M[1][0]);
R[0][1] = M[0][1] - 2 * v[0] * (v[0] * M[0][1] + v[1] * M[1][1]);
R[1][0] = 0;
R[1][1] = M[1][1] - 2 * v[1] * (v[0] * M[0][1] + v[1] * M[1][1]);
t[0] = v[0]*S[0][0]+v[1]*S[1][0];
t[1] = v[0]*S[0][1]+v[1]*S[1][1];
z[0][0] = S[0][0]-2*v[0]*t[0];
z[0][1] = S[0][1]-2*v[0]*t[1];
z[1][0] = S[1][0]-2*v[1]*t[0];
z[1][1] = S[1][1]-2*v[1]*t[1];
x[1][0] = z[1][0]/R[1][1];
x[1][1] = z[1][1]/R[1][1];
x[0][0] = (z[0][0]-R[0][1]*z[1][0]/R[1][1])/R[0][0];
x[0][1] = (z[0][1]-R[0][1]*z[1][1]/R[1][1])/R[0][0];
}

```

```

void MLP_CPU(float step_size, float voxel_size, int number_of_histories,
float *x, //x is the previous iteration's image
int width, float *rotation, float *x_in, float *y_in, float *theta_in,
float *x_out, float *y_out, float *theta_out, float *b, float *output,
float *depth, float *P4_11, float *P4_12, float *P5_11, float *P5_12)
{
//step size should be half the width of a voxel
//(eg. 0.125mm for 0.25mm voxel width)

int number_of_steps;
float a_norm_inv;
float lambda = 0.003; //From SAP Paper for Fully Sequential ART

for(int index=0;index<number_of_histories;index++)
{

```

```

voxel_size = 0.025; //cm
step_size = 0.0125; //cm
number_of_steps = floor((depth[index]/10)/step_size);
a_norm_inv = (float)(voxel_size/number_of_steps); /*(depth[index]/10);
//remove comment above for relaxation as a function of depth
//relaxation parameter in bin loop will
//need to be lowered by a factor of 10

int a_i_0[1600]; //x pixel numbers
int a_i_1[1600]; //y pixel numbers
float x_dot_a = 0;

float xin = (x_in[index]-106)/10;
float xout = (x_out[index]-106)/10;
float yin = (y_in[index]-106)/10;
float yout = (y_out[index]-106)/10;
float u_offset_in = ((xin) * cos(-rotation[index])
                    - (yin) * sin(-rotation[index])); //cm
float t_offset_in = ((xin) * sin(-rotation[index])
                    + (yin) * cos(-rotation[index])); //cm
float u_offset_out = ((xout) * cos(-rotation[index])
                    - (yout) * sin(-rotation[index])); //cm
float t_offset_out = ((xout) * sin(-rotation[index])
                    + (yout) * cos(-rotation[index])); //cm
float j_rot = rotation[index];
for(int j=0;j<number_of_steps;j++)
{
    float u = j * step_size + u_offset_in; // 'x' component (u)
    float t = ((P4_11[j]*t_offset_in+P4_12[j]*theta_in[index])
              +(P5_11[j]*t_offset_out+P5_12[j]*theta_out[index]));
    // 'y' component (t)
    float u2 = u * cos(rotation[index]) - t * sin(rotation[index]);
    float t2 = u * sin(rotation[index]) + t * cos(rotation[index]);
    float u3 = u2 + 10.6;
    float t3 = t2 + 10.6;
    a_i_0[j] = ceil(u3/voxel_size);
    a_i_1[j] = ceil(t3/voxel_size);
}
for(int j=0;j<number_of_steps;j++)
{
    if (a_i_0[j] && a_i_1[j] <= 849 && a_i_0[j] && a_i_1[j] >= 0)
    {
        x_dot_a = x_dot_a + x[a_i_1[j]*width+a_i_0[j]]; // * 10);
    }
}

```



```

    }
}
float update = a.norm_inv * (b[index] - x_dot_a * voxel_size);
for(int j=0;j<number_of_steps;j++)
{
  if (a_i-0[j] && a_i-1[j] <= 849 && a_i-0[j] && a_i-1[j] >= 0)
  {
    output[a_i-1[j]*width+a_i-0[j]] = output[a_i-1[j]*width+a_i-0[j]]
      + update; //BIP
    //x[a_i-1[j]*width+a_i-0[j]] = x[a_i-1[j]*width+a_i-0[j]]
      //+ update * lambda; //Fully Sequential ART
  }
}
}
}
}
} //end MLP_CPU

void mat_inverse(float A[2][2], float A_inv[2][2])
{
  double det = A[0][0]*A[1][1] - A[0][1]*A[1][0];
  A_inv[0][0] = 1/det*A[1][1];
  A_inv[0][1] = 1/det*-A[0][1];
  A_inv[1][0] = 1/det*-A[1][0];
  A_inv[1][1] = 1/det*A[0][0];
}

double E(double u)
{
  //SciLab code
  //function output=depth5(u)
  // output = a_0+a_1*u+a_2*u^2+a_3*u^3+a_4*u^4+a_5*u^5;
  //endfunction
  double a_0 = 202.20574; //coefficients of E(u) at 200MeV entry
  double a_1 = -7.6174839;
  double a_2 = 0.9413194;
  double a_3 = -0.1141406;
  double a_4 = 0.0055340;
  double a_5 = -0.0000972;

  return a_0+a_1*u+a_2*pow((double)u,2)+a_3*pow((double)u,3)+a_4*
    pow((double)u,4)+a_5*pow((double)u,5);
}
double beta_squared_p_squared_inv(double u)
{

```

```

//SciLab code
//function output=depth(u)//1/(beta^2(u)*p^2(u))
// temp = depth5(u)*MeV;///(c*c);
// output = ((temp+E-p)^2*C*C)/((temp+2*E-p)^2*temp^2)*(MeV/C)^2;
//endfunction

double temp = E(u)*MeV;
return (pow((double)(temp+E-p),2)*C*C)/(pow((double)(temp+2*E-p),2)*
      pow((double)temp,2))*pow((double)(MeV/C),2);

} //end beta_squared_p_squared_inv

//Sigma1 functions
double s1_beta2_p2_u2(double u)
{
//SciLab code
//function output=s1_beta2_p2_u2_1(u)
// output = ((u-u_0)^2) * depth(u) / X_0;
//endfunction

return (pow((double)(u-u_0),2)) * beta_squared_p_squared_inv(u) / X_0;
}

double s1_beta2_p2(double u)
{
//function output=s1_beta2_p2(u)
// output = depth(u) / X_0;
//endfunction

return beta_squared_p_squared_inv(u) / X_0;
}

double s1_beta2_p2_u(double u)
{
//function output=s1_beta2_p2_u(u)
// output = (u-u_0) * depth(u) / X_0;
//endfunction

return (u-u_0) * beta_squared_p_squared_inv(u) / X_0;
}

//Sigma2 functions

```

```

double s2_beta2_p2_u2(double u)
{
//function output=s2_beta2_p2_u2(u)
// output = ((u_2-u)^2) * depth(u) / X_0;
//endfunction
return (pow((double)(u_2-u),2)) * beta_squared_p_squared_inv(u) / X_0;
}

```

```

double s2_beta2_p2(double u)
{
//function output=s2_beta2_p2(u)
// output = depth(u) / X_0;
//endfunction
return beta_squared_p_squared_inv(u) / X_0;
}

```

```

double s2_beta2_p2_u(double u)
{
//function output=s2_beta2_p2_u(u)
// output = (u_2-u) * depth(u) / X_0;
//endfunction
return (u_2-u) * beta_squared_p_squared_inv(u) / X_0;
}

```

```

//Matrix multiplication for single precision 2 x 2
//matrices including transposed versions of either
void mat_mult(int t1, int t2, float A[2][2], float B[2][2],
              float output[2][2])
{
if(t1 == 0 && t2 == 0)
{
output[0][0] = A[0][0] * B[0][0] + A[0][1] * B[1][0];
output[0][1] = A[0][0] * B[0][1] + A[0][1] * B[1][1];
output[1][0] = A[1][0] * B[0][0] + A[1][1] * B[1][0];
output[1][1] = A[1][0] * B[0][1] + A[1][1] * B[1][1];
}
if(t1 == 1 && t2 == 0)
{
output[0][0] = A[0][0] * B[0][0] + A[1][0] * B[1][0];
output[0][1] = A[0][0] * B[0][1] + A[1][0] * B[1][1];
output[1][0] = A[0][1] * B[0][0] + A[1][1] * B[1][0];
output[1][1] = A[0][1] * B[0][1] + A[1][1] * B[1][1];
}
}

```

```

if(t1 == 0 && t2 == 1)
{
    output[0][0] = A[0][0] * B[0][0] + A[0][1] * B[0][1];
    output[0][1] = A[0][0] * B[1][0] + A[0][1] * B[1][1];
    output[1][0] = A[1][0] * B[0][0] + A[1][1] * B[0][1];
    output[1][1] = A[1][0] * B[1][0] + A[1][1] * B[1][1];
}
if(t1 == 1 && t2 == 1)
{
    output[0][0] = A[0][0] * B[0][0] + A[1][0] * B[0][1];
    output[0][1] = A[0][0] * B[1][0] + A[1][0] * B[1][1];
    output[1][0] = A[0][1] * B[0][0] + A[1][1] * B[0][1];
    output[1][1] = A[0][1] * B[1][0] + A[1][1] * B[1][1];
}
}

//Matrix multiplication for double precision 2 x 2
//matrices including transposed versions of either
void mat_mult_d(int t1, int t2, double A[2][2], double B[2][2],
                double output[2][2])
{
    if(t1 == 0 && t2 == 0)
    {
        output[0][0] = A[0][0] * B[0][0] + A[0][1] * B[1][0];
        output[0][1] = A[0][0] * B[0][1] + A[0][1] * B[1][1];
        output[1][0] = A[1][0] * B[0][0] + A[1][1] * B[1][0];
        output[1][1] = A[1][0] * B[0][1] + A[1][1] * B[1][1];
    }
    if(t1 == 1 && t2 == 0)
    {
        output[0][0] = A[0][0] * B[0][0] + A[1][0] * B[1][0];
        output[0][1] = A[0][0] * B[0][1] + A[1][0] * B[1][1];
        output[1][0] = A[0][1] * B[0][0] + A[1][1] * B[1][0];
        output[1][1] = A[0][1] * B[0][1] + A[1][1] * B[1][1];
    }
    if(t1 == 0 && t2 == 1)
    {
        output[0][0] = A[0][0] * B[0][0] + A[0][1] * B[0][1];
        output[0][1] = A[0][0] * B[1][0] + A[0][1] * B[1][1];
        output[1][0] = A[1][0] * B[0][0] + A[1][1] * B[0][1];
        output[1][1] = A[1][0] * B[1][0] + A[1][1] * B[1][1];
    }
    if(t1 == 1 && t2 == 1)

```

```

{
    output[0][0] = A[0][0] * B[0][0] + A[1][0] * B[0][1];
    output[0][1] = A[0][0] * B[1][0] + A[1][0] * B[1][1];
    output[1][0] = A[0][1] * B[0][0] + A[1][1] * B[0][1];
    output[1][1] = A[0][1] * B[1][0] + A[1][1] * B[1][1];
}
}

//small matrix-vector multiply
void mat_vec_mult(float A[2][2], float B[2], float output[2])
{
    output[0] = A[0][0] * B[0] + A[0][1] * B[1];
    output[1] = A[1][0] * B[0] + A[1][1] * B[1];
}

//Matrix Addition for 2 x 2
void mat_add(float A[2][2], float B[2][2], float output[2][2])
{
    output[0][0] = A[0][0] + B[0][0];
    output[0][1] = A[0][1] + B[0][1];
    output[1][0] = A[1][0] + B[1][0];
    output[1][1] = A[1][1] + B[1][1];
}

//Vector Addition for 2 x 1 or 1 x 2
void vec_add(float A[2], float B[2], float output[2])
{
    output[0] = A[0] + B[0];
    output[1] = A[1] + B[1];
}

//MLP_step_naive
void MLP_step_naive(float sig1_inv[2][2], float sig2_inv[2][2],
    float R_0[2][2], float R_1[2][2],
    float y_0[2], float y_2[2], float y_out[2])
{
    float temp1[2][2];
    float temp2[2][2];
    float temp_vec1[2];
    float temp_vec2[2];
    mat_mult(1,0,R_1,sig2_inv,temp1);
    mat_mult(0,0,temp1,R_1,temp2);
    mat_add(sig1_inv,temp2,temp1);
}

```

```

mat_inverse(temp1,temp2);//first half of MLP step held in temp2[[[
mat_mult(0,0,sig1_inv,R_0,temp1);
mat_vec_mult(temp1,y_0,temp_vec1);//first part of second half
mat_mult(1,0,R_1,sig2_inv,temp1);
mat_vec_mult(temp1,y_2,y_out);//second part of second half
vec_add(temp_vec1,y_out,temp_vec2);
mat_vec_mult(temp2,temp_vec2,y_out);//final output

}//end MLP_step_naive

```

```

void electron_density_CPU(int size, float *E_in,
    float *E_out, float *output,
    float *u_temp1, float *u_temp2,
    float *temp1, float *temp2)
{
    float K_inv = 1/0.17;
    float C = 299792458; // speed of light in m/s
    float I = 75;
    float eV = 1.602176487e-19; // eV
    float MeV = 1e6*eV;
    float keV = 1e3*eV;
    float m_p = 1.672621637E-27; // mass of proton
    float e_p = m_p*C*C/eV;
    float M_e = 9.1093826e-31; // mass of an electron
    float M_ec2 = M_e*C*C/eV;
    float x_0 = -0.57735026918963;
    float x_1 = 0.57735026918963;

    //scilab function to integrate
    //e=E*10^6;
    //temp = (1+(e_p^2/(e^2+2*e*e_p)))*(log(2*M_ec2/I)+log(e)+
        log(e+2*e_p)-2*log(e_p)-1+(e_p^2/(e+e_p)^2))
    //output = 1/temp;

    for (int index=0;index<size;index++)
    {
        u_temp1[index] = ((E_in[index]+E_out[index])/2)+((E_in[index]-
            E_out[index])/2)*x_0;
            //the two points to be added together
        u_temp2[index] = ((E_in[index]+E_out[index])/2)+((E_in[index]-
            E_out[index])/2)*x_1;

        temp1[index] = 1/((1+((e_p*e_p)/(u_temp1[index]*u_temp1[index]+

```

```

                2*u_temp1[index]*e_p)))*(log((double)2*M_ec2/I)+
                log((double)u_temp1[index])+log((double)u_temp1[index]+
                2*e_p)-2*log((double)e_p)-1+((e_p*e_p)/((u_temp1[index]+
                e_p)*(u_temp1[index]+e_p)))));

temp2[index] = 1/((1+((e_p*e_p)/(u_temp2[index]*u_temp2[index]+
                2*u_temp2[index]*e_p)))*(log((double)2*M_ec2/I)+
                log((double)u_temp2[index])+log((double)u_temp2[index]+
                2*e_p)-2*log((double)e_p)-1+((e_p*e_p)/((u_temp2[index]+
                e_p)*(u_temp2[index]+e_p)))));

output[index]=K_inv*((E_in[index]-E_out[index])/2)*(temp1[index]+
                temp2[index]);//+f(temp);
}
//end electron_densirty_CPU

double gaussian_quadrature( int n, double(*function)(double val),
                            double a, double b)
{
//performs a two to five point gaussian quadrature on a function
double c = 0;
double d = 0;

const unsigned int n_mem_size = sizeof(double) * n;
double *w = (double*) malloc(n_mem_size);
double *x = (double*) malloc(n_mem_size);

for (int i=0;i<n;i++)
{
w[i]=0;
x[i]=0;
}
//weights from table 6.1 of Keith on Numerical and J. Tafas code
if (n == 1)
{
x[0] = 0;
w[0] = 2;
}
else if (n == 2)
{
x[0] = -0.57735026918963;
x[1] = 0.57735026918963;
w[0] = 1;

```

```

    w[1] = 1;
}
else if (n == 3)
{
    x[0] = -0.77459667;
    x[1] = 0;
    x[2] = 0.77459667;
    w[0] = 0.55555555;
    w[1] = 0.88888889;
    w[2] = 0.55555555;
}
else if (n == 4)
{
    x[0] = -0.86113631;
    x[1] = -0.33998104;
    x[2] = 0.33998104;
    x[3] = 0.86113631;
    w[0] = 0.34785485;
    w[1] = 0.65214515;
    w[2] = 0.65214515;
    w[3] = 0.34785485;
}
else // n == 5
{
    x[0] = -0.90617985;
    x[1] = -0.53846931;
    x[2] = 0;
    x[3] = 0.53846931;
    x[4] = 0.90617985;
    w[0] = 0.23692689;
    w[1] = 0.47862867;
    w[2] = 0.56888889;
    w[3] = 0.47862867;
    w[4] = 0.23692689;
}

c = (b+a)/2;
d = (b-a)/2;

double integral = 0;
double temp = 0;
for (int i=0;i<n;i++)
{

```



```

        integral = integral + w[i] * (*function)(c + d * x[i]);
    }

    return d * integral;

}

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// Run the reconstruction
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
void
runTest( int argc, char** argv)
{

    float max_depth = 20; // cm
    float step_size = 0.1; // cm

    unsigned int timer;
        cutCreateTimer(&timer);

        CUT_DEVICE_INIT(argc, argv);

{
    unsigned int timer;
    cutCreateTimer(&timer);
    float x1, y1, x2, y2, x3, y3, x4, y4, E_in, E_out, rot, A, B, C;
    float i9, i10, i11=0;
    float loss = 0;
    printf("\n\nStarted\n");
    input = fopen("F:\G4_Herman_Head_200_MeV_p-2D_29-12-08.txt", "r");
    int numRows, numCols, numBoundaryRows, numBoundaryCols, numPathRows;
    int numPathCols, numBinRows, numBinCols;
    int *bincount;
    int *binindex;
    int *bin_hist;
    float **array;
    float **boundary_array;
    float **path_array;
    float **bin_cut_path;
    float **image;
    float **image_update;
    int rowIndex, colIndex, i, j, k, l;
    float energy_tolerance = 0.300;
    float mean = 0;

```

```

float meany = 0;
float mean2 = 0;
float mean3 = 0;
float mean4 = 0;
float temp = 0;
float tempy = 0;
float temp2 = 0;
float temp3 = 0;
float temp4 = 0;
float std_dev = 0;
float std_devy = 0;
float std_dev2 = 0;
float std_dev3 = 0;
float std_dev4 = 0;
int energy_count=0;
int history_count=0;
int max=PROJANGLES*HISTORIES;
float image_width = 21.2; //cm, actually 21.21
        //reduced because of pixel width
float image_height = 21.2; //cm, same as above
float voxel_size = 0.25; //mm
int voxels_w = ceil((image_width*10)/voxel_size)+1;
int voxels_h = ceil((image_height*10)/voxel_size)+1;

numRows = 3240000; //Number of projections per angle * number of angles
numCols = 15;
numBoundaryCols = 9;
numPathCols = 14;
numBinCols = 10;
numBoundaryRows = 0;
numPathRows = 0;
numBinRows = 0;
printf("number of voxels = %i\n", voxels_w*voxels_h);
printf("numRows = %i\nnumCols = %i\nTotal number of elements = %i\n",
        numRows, numCols, numRows*numCols);

array = (float **) malloc( numRows * sizeof(float*));
if( array == NULL)
{
    printf("Out of memory");
}
for(i = 0; i < numRows; i++)
{

```

```

array[i] = (float * )malloc(numCols * sizeof(float));
if( array[i] == NULL)
{
    printf("Out of memory");
}
}

printf(" Allocated Memory\n");

printf("Importing data file\n");
i=1;
j=1;
//start timer
cutStartTimer(timer);
for(i=0;i<max;i++)
{
    fscanf(input, "%f %f %f %f %f %f %f %f %f %f", &x1, &y1, &x2, &y2,
        &x3, &y3, &x4, &y4, &E_in, &E_out, &rot);

//input is strip number and we want the location
array[i][0]=x1*0.01 - 15 + 0.005;
array[i][1]=y1*0.01 - 15 + 0.005;
array[i][2]=x2*0.01 - 15 + 0.005;
array[i][3]=y2*0.01 - 15 + 0.005;
array[i][4]=x3*0.01 - 15 + 0.005;
array[i][5]=y3*0.01 - 15 + 0.005;
array[i][6]=x4*0.01 - 15 + 0.005;
array[i][7]=y4*0.01 - 15 + 0.005;
array[i][8]= E_in;
array[i][9]= E_out;
array[i][10]=rot;
array[i][11]=atan(((x2*0.01 - 15 + 0.005)-(x1*0.01 - 15 + 0.005))/5);
    //angle between the first two x coordinates
array[i][12]=atan(((x4*0.01 - 15 + 0.005)-(x3*0.01 - 15 + 0.005))/5);
    //angle between the second two x coordinates
array[i][13]=atan(((y2*0.01 - 15 + 0.005)-(y1*0.01 - 15 + 0.005))/5);
    //angle between the first two y coordinates
array[i][14]=atan(((y4*0.01 - 15 + 0.005)-(y3*0.01 - 15 + 0.005))/5);
    //angle between the second two y coordinates
if (E_in-E_out<=energy_tolerance)
{
    energy_count++;
}
}

```

```

    history_count++;
    mean = mean + (array[i][12] - array[i][11]);
    meany = meany + (array[i][14] - array[i][13]);
    mean2 = mean2 + array[i][11];
    mean4 = mean4 + array[i][12];
    mean3 = mean3 + (array[i][8] - array[i][9]);
}
//stop and reset timer
cutStopTimer(timer);
float File_Load = cutGetTimerValue(timer);
cutResetTimer(timer);
printf("\nFile load time: %0.3f ms\n\n", File_Load);

printf("File loaded successfully\n");
mean = (float)mean/history_count;
meany = (float)meany/history_count;
mean2 = (float)mean2/history_count;
mean3 = (float)mean3/history_count;
mean4 = (float)mean4/history_count;
//calculate the standard deviation
for(i=0;i<max;i++)
{
    temp = temp + pow(((array[i][12] - array[i][11]) - mean), 2);
    tempy = tempy + pow(((array[i][14] - array[i][13]) - meany), 2);
    temp2 = temp2 + pow((array[i][11] - mean2), 2);
    temp4 = temp4 + pow((array[i][12] - mean4), 2);
    temp3 = temp3 + pow(((array[i][8] - array[i][9]) - mean3), 2);
}
std_dev = sqrt((float)temp/(history_count - 1));
std_devy = sqrt((float)tempy/(history_count - 1));
std_dev2 = sqrt((float)temp2/(history_count - 1));
std_dev3 = sqrt((float)temp3/(history_count - 1));
std_dev4 = sqrt((float)temp4/(history_count - 1));
printf("Standard Deviation Calculated\n");

//this is to calculate the number of histories
//in the boundary and path arrays
int _3sigma = 0;
int _3sigma_e = 0;
int _3sigma_y = 0;
int _3sigma_total = 0;
for(i=0;i<max;i++)
{

```

```

if (fabs(array[i][12] - array[i][11]) >= 3*std_dev)
{
    _3sigma++;
}
if (fabs(array[i][1]) >= 1 ||
    fabs(array[i][3]) >= 1 ||
    fabs(array[i][5]) >= 1 ||
    fabs(array[i][7]) >= 1)
{
    _3sigma_y++;
}
if (fabs(array[i][8] - array[i][9]) >= 3*std_dev3)
{
    _3sigma_e++;
}
if (fabs(array[i][12] - array[i][11]) >= 3*std_dev ||
    fabs(array[i][1]) >= 1 ||
    fabs(array[i][3]) >= 1 ||
    fabs(array[i][5]) >= 1 ||
    fabs(array[i][7]) >= 1 ||
    fabs(array[i][8] - array[i][9]) >= 3*std_dev3)
{
    _3sigma_total++;
}
if (fabs(array[i][12] - array[i][11]) < 3*std_dev &&&
    fabs(array[i][8] - array[i][9]) < 3*std_dev3 &&&
    array[i][8] - array[i][9] <= energy_tolerance &&&
    0.1 < array[i][8] - array[i][9] &&&
    fabs(array[i][14] - array[i][13]) < 3*std_dev3 &&&
    fabs(array[i][1]) < 1 &&&
    fabs(array[i][3]) < 1 &&&
    fabs(array[i][5]) < 1 &&&
    fabs(array[i][7]) < 1)
{
    numBoundaryRows++;
}
if (fabs(array[i][12] - array[i][11]) < 3*std_dev &&&
    fabs(array[i][8] - array[i][9]) < 3*std_dev3 &&&
    array[i][8] - array[i][9] > energy_tolerance &&&
    fabs(array[i][14] - array[i][13]) < 3*std_dev3 &&&
    fabs(array[i][1]) < 1 &&&
    fabs(array[i][3]) < 1 &&&
    fabs(array[i][5]) < 1 &&&

```

```

    fabs(array[i][7]) < 1)
    {
        numPathRows++;
    }
}
printf("number of energies less than %f MeV loss = %i\n",
        energy_tolerance, energy_count);
loss = (float)energy_count/history_count;
printf("Total number of proton histories = %i\n", history_count);
printf("Percentage of removed histories = %f\n", loss);
printf("Mean angle difference = %f\n", mean);
printf("Mean energy loss = %f\n", mean3);
printf("Standard Deviation angle = %f\n", std_dev);
printf("Standard Deviation energy = %f\n", std_dev3);
printf("Histories removed due to 3 sigma x-angle cut\t= %i\n", _3sigma);
printf("Histories removed due to 3 sigma y-angle cut\t= %i\n", _3sigma_y);
printf("Histories removed due to 3 sigma energy cut\t= %i\n", _3sigma_e);
printf("Total histories removed due to 3 sigma cuts\t= %i\n", _3sigma_total);
printf("Number of histories in boundary matrix\t\t= %i\n", numBoundaryRows);
printf("Number of histories in path matrix\t\t= %i\n", numPathRows);

//allocate memory for boundary array
boundary_array = (float **) malloc( numBoundaryRows * sizeof(float*));
if( boundary_array == NULL)
{
    printf("Out of memory");
}
for(i = 0; i < numBoundaryRows; i++)
{
    boundary_array[i] = (float *) malloc(numBoundaryCols * sizeof(float));
    if( boundary_array[i] == NULL)
    {
        printf("Out of memory");
    }
}

//allocate memory for path array
path_array = (float **) malloc( numPathRows * sizeof(float*));
if( path_array == NULL)
{
    printf("Out of memory");
}
for(i = 0; i < numPathRows; i++)

```

```

{
  path_array[i] = (float * )malloc(numPathCols * sizeof(float));
  if( path_array[i] == NULL)
  {
    printf("Out of memory");
  }
}

//fill boundary and path arrays
k = 0;
l = 0;
for(i=0;i<max;i++)
{
  if ( fabs(array[i][12] - array[i][11]) < 3*std_dev &&
      fabs(array[i][8] - array[i][9]) < 3*std_dev3 &&
      array[i][8] - array[i][9] <= energy_tolerance &&
      0.1 < array[i][8] - array[i][9] &&
      fabs((array[i][3] - array[i][1]) - (array[i][7] -
      array[i][5])) < 3*std_dev &&
      fabs(array[i][1]) < 1 &&&
      fabs(array[i][3]) < 1 &&&
      fabs(array[i][5]) < 1 &&&
      fabs(array[i][7]) < 1)
  {
    boundary_array[k][0] = array[i][0]; //x1
    boundary_array[k][1] = array[i][2]; //x2
    boundary_array[k][2] = array[i][4]; //x3
    boundary_array[k][3] = array[i][6]; //x4
    boundary_array[k][4] = array[i][10]; //rot
    boundary_array[k][5] = array[i][8];
    boundary_array[k][6] = array[i][9];
    boundary_array[k][7] = 0;
    boundary_array[k][8] = 0;
    k++;
  }
  if (fabs(array[i][12] - array[i][11]) < 3*std_dev &&
      fabs(array[i][8] - array[i][9]) < 3*std_dev3 &&
      array[i][8] - array[i][9] > energy_tolerance &&
      fabs(array[i][14] - array[i][13]) < 3*std_dev &&
      fabs(array[i][1]) < 1 &&&
      fabs(array[i][3]) < 1 &&&
      fabs(array[i][5]) < 1 &&&
      fabs(array[i][7]) < 1)

```

```

{
    path_array[1][0]=array[i][0]; //x1
    path_array[1][1]=array[i][2]; //x2
    path_array[1][2]=array[i][4]; //x3
    path_array[1][3]=array[i][6]; //x4
    path_array[1][4]=array[i][8]; //E_in
    path_array[1][5]=array[i][9]; //E_out
    path_array[1][6]=array[i][10]; //rot
    path_array[1][7]=array[i][11]; //x1, x2 angle
    path_array[1][8]=array[i][12]; //x3, x4 angle
    path_array[1][9]=0;
    path_array[1][10]=0;
    path_array[1][11]=0;
    path_array[1][12]=0;
    path_array[1][13]=-1; //-1 will assign no bin
    l++;
}
}
printf("Filled Boundary and Path Arrays\n");
free(array);

//Create image and image-update
image = (float **) malloc( voxels_h * sizeof(float*));
if( image == NULL)
{
    printf("Out of memory");
}
for(i = 0; i < voxels_h; i++)
{
    image[i] = (float * )malloc(voxels_w * sizeof(float));
    if( image[i] == NULL)
    {
        printf("Out of memory");
    }
}
image_update = (float **) malloc( voxels_h * sizeof(float*));
if( image == NULL)
{
    printf("Out of memory");
}
for(i = 0; i < voxels_h; i++)
{
    image_update[i] = (float * )malloc(voxels_w * sizeof(float));

```



```

if( image_update[i] == NULL)
{
    printf("Out of memory");
}
}
for (i=0; i<voxels_h; i++)
{
    for(j=0; j<voxels_w; j++)
    {
        image[i][j] = voxel_size; // 0.25 is unit length in this case
        image_update[i][j] = 0;
    }
}
for (i=0; i<voxels_h; i++)
{
    image[i][1] = 0;
    image[i][0] = 0;
}
for(j=0; j<voxels_w; j++)
{
    image[1][j] = 0;
    image[0][j] = 0;
}

```

```

//boundary_array[k][0]=array[i][0];//x1
//boundary_array[k][1]=array[i][2];//x2
//boundary_array[k][2]=array[i][4];//x3
//boundary_array[k][3]=array[i][6];//x4
//boundary_array[k][4]=array[i][10];//rot

```

```

printf("Processing image data\n");
float A_11, A_12, A_13, A_14, B_11, B_12, B_13, B_14;
float C_11, C_12, C_13, C_14;
float A1_left, B1_left, C1_left;
float A1_right, B1_right, C1_right;
float A1_top, B1_top, C1_top;
float A1_bottom, B1_bottom, C1_bottom;
float x1_left, y1_left, x2_left, y2_left;
float x1_right, y1_right, x2_right, y2_right;
float x1_top, y1_top, x2_top, y2_top;
float x1_bottom, y1_bottom, x2_bottom, y2_bottom;
//calculate the four line segments that

```

```

//make up the boundry of the image area
//A = y2-y1
//B = x1-x2
//C = A*x1+B*y1

//left segment
x1_left = -10.6;
y1_left = 10.6;
x2_left = -10.6;
y2_left = -10.6;
A1_left = y2_left-y1_left;
B1_left = x1_left-x2_left;
C1_left = A1_left*x1_left+B1_left*y1_left;
//right segment
x1_right = 10.6;
y1_right = 10.6;
x2_right = 10.6;
y2_right = -10.6;
A1_right = y2_right-y1_right;
B1_right = x1_right-x2_right;
C1_right = A1_right*x1_right+B1_right*y1_right;
//top segment
x1_top = -10.6;
y1_top = 10.6;
x2_top = 10.6;
y2_top = 10.6;
A1_top = y2_top-y1_top;
B1_top = x1_top-x2_top;
C1_top = A1_top*x1_top+B1_top*y1_top;
//bottom segment
x1_bottom = -10.6;
y1_bottom = -10.6;
x2_bottom = 10.6;
y2_bottom = -10.6;
A1_bottom = y2_bottom-y1_bottom;
B1_bottom = x1_bottom-x2_bottom;
C1_bottom = A1_bottom*x1_bottom+B1_bottom*y1_bottom;

int count_left = 0;
int count_right = 0;
int count_top = 0;
int count_bottom = 0;
int count_errors = 0;

```

```

for (i=0; i<numBoundaryRows; i++)
{
float x1prime, y1prime, x2prime, y2prime;
float A1, A2, B1, B2, C1, C2, det;
float x_intersect, y_intersect;
int j = 5;

//rotation =
//x' = x*cos(theta)-y*sin(theta)
//y' = x*sin(theta)+y*cos(theta)

//known points on the line segments (in this case, endpoints)
//(x1,y1) and (x2,y2) give:
//A = y2-y1
//B = x1-x2
//C = A*x1+B*y1

//calculating values for discretized
//area using the corners of the area

//calculating values for proton path
x1 = -25.0; //cm
x2 = 25.0; //cm
y1 = boundary_array[i][1];
y2 = boundary_array[i][2];
x1prime = x1*cos(boundary_array[i][4])-y1*sin(boundary_array[i][4]);
y1prime = x1*sin(boundary_array[i][4])+y1*cos(boundary_array[i][4]);
x2prime = x2*cos(boundary_array[i][4])-y2*sin(boundary_array[i][4]);
y2prime = x2*sin(boundary_array[i][4])+y2*cos(boundary_array[i][4]);
A2 = y2prime-y1prime;
B2 = x1prime-x2prime;
C2 = A2*x1prime+B2*y1prime;

// ** //check the left segment first *****
det = A1_left*B2 - A2*B1_left;
if((int)det == 0)
{
//Lines are parallel, do nothing
}
else
{
x_intersect = x1_left;

```

```

y_intersect = (A1_left*C2 - A2*C1_left)/det;

if (y2_left < y_intersect && y_intersect < y1_left)
{
    boundary_array[i][j] = x1_left; // using known value to reduce error
    j++;
    boundary_array[i][j] = y_intersect;
    j++;
    count_left++;
}
}

// ** //check the right segment second *****
det = A1_right*B2 - A2*B1_right;
if((int)det == 0)
{
    //Lines are parallel, do nothing
}
else
{
    x_intersect = x1_right;
    y_intersect = (A1_right*C2 - A2*C1_right)/det;

if (y2_right < y_intersect && y_intersect < y1_right)
{
    boundary_array[i][j] = x1_right; // using known value to reduce error
    j++;
    boundary_array[i][j] = y_intersect;
    j++;
    count_right++;
}
}

// ** //check the top segment third *****
det = A1_top*B2 - A2*B1_top;
if((int)det == 0)
{
    //Lines are parallel, do nothing
}
else
{
    y_intersect = y1_top;
    x_intersect = (B2*C1_top - B1_top*C2)/det;

```

```

// this value is known, there is no reason to calculate it

if (x1_top < x_intersect && x_intersect < x2_top)
{
    boundary_array[i][j] = x_intersect;
    j++;
    boundary_array[i][j] = y1_top; // using known value to reduce error
    j++;
    count_top++;
}
}

// ** //check the bottom segment last *****
det = A1_bottom*B2 - A2*B1_bottom;
if((int)det == 0)
{
    //Lines are parallel, do nothing
}
else
{
    y_intersect = y1_bottom;
    x_intersect = (B2*C1_bottom - B1_bottom*C2)/det;
    // this value is known, there is no reason to calculate it

    if (x1_bottom < x_intersect && x_intersect < x2_bottom)
    {
        boundary_array[i][j] = x_intersect;
        j++;
        boundary_array[i][j] = y1_bottom; // using known value to reduce error
        j++;
        count_bottom++;
    }
}

if ((j-5)>4)
{
    count_errors++;
}
} //end boundary calculation

int count_boundary_zero = 0;
for (i=0; i<numBoundaryRows; i++)
{

```

```

if((int)boundary_array[i][5] && (int)boundary_array[i][6] &&
(int)boundary_array[i][7] && (int)boundary_array[i][8] != 0)
{
    count_boundary_zero++;
}
}
printf("Number of histories hitting the left segment\t= %i\n",
    count_left);
printf("Number of histories hitting the right segment\t= %i\n",
    count_right);
printf("Number of histories hitting the top segment\t= %i\n",
    count_top);
printf("Number of histories hitting the bottom segment\t= %i\n",
    count_bottom);
printf("Total number of hits on all segments\t\t= %i\n",
    count_left+count_right+count_top+count_bottom);
printf("Number of histories in boundary matrix\t\t= %i\n",
    numBoundaryRows);
printf("Number of errors\t\t\t\t= %i\n",count_errors);
printf("Number of nonzero boundary histories\t\t= %i\n\n",
    count_boundary_zero);

//now zero the voxels passed through by coordinates created above
printf("Calculating Convex Hull\n");
float slope, d, theta, x_p, y_p;
int steps, x_pixel_number, y_pixel_number;
float step_size = 0.125;
int count_path = 0;
int count_path2 = 0;
for (i=0; i<numBoundaryRows; i++)
{
    if (count_path2 == 100000)
    {
        count_path++;
        count_path2 = 0;
        printf("%i00000 paths\n", count_path);
    }
    count_path2++;
    if((int)boundary_array[i][5] && (int)boundary_array[i][6] &&
(int)boundary_array[i][7] && (int)boundary_array[i][8] != 0)
    {
        x_pixel_number = 0;
        y_pixel_number = 0;

```

```

//distance between two points
d=sqrt(pow((boundary_array[i][7]-boundary_array[i][5]),2)+
    pow(boundary_array[i][8]-boundary_array[i][6],2)); //cm
steps = (int)floor(d*10/step_size)-1;
// use atan instead of atan2 because
//we make sure the path goes from left to right
theta = atan(((boundary_array[i][8])-(boundary_array[i][6]))/
    ((boundary_array[i][7])-(boundary_array[i][5])));
for (j=0;j<steps;j++)
{
    //rotate the stepped coordinate and add it to the starting coordinate
    //rotation =
    //x' = x*cos(theta)-y*sin(theta)
    //y' = x*sin(theta)+y*cos(theta)

    x-p = (j*step_size)*cos(theta);//-0*sin(theta);
    y-p = (j*step_size)*sin(theta);//+0*cos(theta);

    if (boundary_array[i][5]<boundary_array[i][7])
    {
        x_pixel_number = ceil(((boundary_array[i][5]+10.6)*10+(x-p))/0.25);
        //x pixel number
        y_pixel_number = ceil(((boundary_array[i][6]+10.6)*10+(y-p))/0.25);
        //y pixel number
    }
    else //start from the other point
    {
        x_pixel_number = ceil(((boundary_array[i][7]+10.6)*10+x-p)/0.25);
        //x pixel number
        y_pixel_number = ceil(((boundary_array[i][8]+10.6)*10+y-p)/0.25);
        //y pixel number
    }
    image[x_pixel_number][y_pixel_number]=0; //update image matrix
}
}
}

convex_hull = fopen( "F:\convex_hull.txt", "w" );

for (i=0;i<voxels_h;i++)
{
    for(j=0;j<voxels_w;j++)
    {

```

```

    fprintf(convex_hull, "%1.2f ",image[i][j]);
}
fprintf(convex_hull, "\n");
}
printf("Hull calculation complete\n\n");

// ***** Path location calculation *****

printf("Calculating entry and exit points\n");
count_path = 0;
count_path2 = 0;
int miss = 0;
int hit = 0;
int d1_hit = 0;
int d2_hit = 0;
int x1_hit = 0;
int y1_hit = 0;
int x2_hit = 0;
int y2_hit = 0;
float MeV_Loss = 0;
float min_d = 1000000;
float max_d = 0;
float minn = 1000000;
float maxx = 0;
int max_bin = -1;
for (i=0; i<numPathRows; i++)
{
    float x1prime, y1prime, x2prime, y2prime;
    float x3prime, y3prime, x4prime, y4prime;
    float A1, A2, B1, B2, C1, C2, det;
    float x_intersect, y_intersect;
    float theta;
    int j = 5;
    int x1_pos = 0;
    int x2_pos = 0;
    int y1_pos = 0;
    int y2_pos = 0;

    if (count_path2 == 100000)
    {
        count_path++;
        count_path2 = 0;
        printf("%i00000 paths\t%i missed\tmin u_2 (mm)= %f\tmax u_2 (mm)= %f\n",

```



```

        count_path, miss, min_d, max_d);
if (min_d<minn)
{
    minn = min_d;
}
if (max_d>maxx)
{
    maxx = max_d;
}
min_d = 1000000;
max_d = 0;
miss= 0;
MeV.Loss = 0;
d1.hit = 0;
d2.hit = 0;
x1.hit = 0;
y1.hit = 0;
x2.hit = 0;
y2.hit = 0;
}
count_path2++;

//path_array[l][0]=array[i][0]; //x1
//path_array[l][1]=array[i][2]; //x2
//path_array[l][2]=array[i][4]; //x3
//path_array[l][3]=array[i][6]; //x4
//path_array[l][4]=array[i][8]; //E-in
//path_array[l][5]=array[i][9]; //E-out
//path_array[l][6]=array[i][10]; //rot
//path_array[l][7]=array[i][11]; //x1, x2 angle (angle-in)
//path_array[l][8]=array[i][12]; //x3, x4 angle (angle-out)
//path_array[l][9]=0;
//path_array[l][10]=0;
//path_array[l][11]=0;
//path_array[l][12]=0;

//rotate the two points
//take the atan (-atan for the third and fourth detectors)
//to get the angle/slope
//step down a line and rotate it
//check if any of the points hit !=0 elements of the image matrix
//(making sure they're still in the image matrix)
//do this forward for the left side and backward for the right side

```

```

//store these coordinates for MLP calculation

theta = path_array[i][6]+path_array[i][7];
xlprime = (((-25.0)*cos(theta)-path_array[i][1]*
            sin(path_array[i][6]))+10.6)*10;
ylprime = (((-25.0)*sin(theta)+path_array[i][1]*
            cos(path_array[i][6]))+10.6)*10;
for (j=1152; j<2500; j++) // 2000 is half the number of steps between
    // the detectors (25.0cm / 0.125mm/step)
    // or about the center of the image area
    // 1152 is 10.6cm from the center of the image
{
    //path_array[i][7]; //angle between first two x coordinates and axis
    //path_array[i][6]; //system rotation
    //need system rotation + angle between the first two points
    x_p = (j*step_size)*cos(theta); //-0*sin(theta);
    y_p = (j*step_size)*sin(theta); //+0*cos(theta);
    x_pixel_number = ceil((xlprime+(x_p))/0.25); //x pixel number
    y_pixel_number = ceil((ylprime+(y_p))/0.25); //y pixel number

    if (x_pixel_number > 0 && x_pixel_number < voxels_w &&
        y_pixel_number > 0 && y_pixel_number < voxels_h &&
        image[x_pixel_number][y_pixel_number]>0)
    {
        //assign point to path array
        xl_pos = x_pixel_number;
        xl_hit++;
        yl_pos = y_pixel_number;
        yl_hit++;
        j=4000; //break out of loop
    }
}
j=0;
theta = path_array[i][6]+path_array[i][8];
xlprime = (((25.0)*cos(theta)-path_array[i][2]*
            sin(path_array[i][6]))+10.6)*10; //mm
ylprime = (((25.0)*sin(theta)+path_array[i][2]*
            cos(path_array[i][6]))+10.6)*10; //mm
for (j=1152; j<2500; j++)
    // 4000 is the number of steps between
    //the detectors (50.0cm / 0.125mm/step)
{

```

```

//path_array[i][7]; //angle between first two x coordinates and axis
//path_array[i][6]; //system rotation
//need system rotation + angle between the first two points
x_p = (j*step_size)*cos(theta); //-0*sin(theta);
y_p = (j*step_size)*sin(theta); //+0*cos(theta);
x_pixel_number = ceil((x1prime-(x-p))/0.25); //x pixel number
y_pixel_number = ceil((y1prime-(y-p))/0.25); //y pixel number

if (x_pixel_number > 0 && x_pixel_number < voxels_w &&
    y_pixel_number > 0 && y_pixel_number < voxels_h &&
    image[x_pixel_number][y_pixel_number]>0)
{
    //assign point to path array
    x2_pos = x_pixel_number;
    x2_hit++;
    y2_pos = y_pixel_number;
    y2_hit++;
    j=4000; //break out of loop
}
}

if (x1_pos && y1_pos && x2_pos && y2_pos !=0)
{
    path_array[i][9] = (float)x1_pos;
    path_array[i][10] = (float)y1_pos;
    path_array[i][11] = (float)x2_pos;
    path_array[i][12] = (float)y2_pos;
    //x2-x1 + y2-y1
    d=sqrt(pow((path_array[i][11]-path_array[i][9])*0.25,2)+
        pow((path_array[i][12]-path_array[i][10])*0.25,2))+0.25;
    //mm, +0.25mm because this is counting from voxel centers
    path_array[i][13] = floor(d/0.5); // bin assignment
    hit++;
    if (path_array[i][13]>max_bin)
    {
        max_bin = (int)path_array[i][13];
    }
    if (d<min_d)
    {
        min_d = d;
    }
    if (d>max_d)
    {

```

```

    max_d = d;
}
}
else
{
    miss++;
    MeV_Loss = MeV_Loss + (path_array[i][4] - path_array[i][5]);
}

} //end path location calculation

printf("Assigning bins\n");
bincount = (int *) malloc( (max_bin + 1) * sizeof(int*));
if( bincount == NULL)
{
    printf("Out of memory");
}
binindex = (int *) malloc( (max_bin + 1) * sizeof(int*));
if( binindex == NULL)
{
    printf("Out of memory");
}
bin_hist = (int *) malloc( (max_bin + 1) * sizeof(int*));
if( bin_hist == NULL)
{
    printf("Out of memory");
}
for (i=0; i<=max_bin+1; i++)
{
    bincount[i] = 0;
    binindex[i] = 0;
    bin_hist[i] = 0;
}
for (i=0; i<numPathRows; i++)
{
    if (path_array[i][9] && path_array[i][10] &&
        path_array[i][11] && path_array[i][12] > 0)
    {
        int temp = path_array[i][13];
        binindex[temp]++;
        bin_hist[temp]++;
    }
}
}

```

```

printf("Minimum u-2 distance in mm = %f\n", minn);
printf("Maximum u-2 distance in mm = %f\n", maxx);
printf("Number of hits = %i\n", hit);
printf("Number of bins = %i\n", max_bin);

bin_data = fopen( "bin_data.txt", "w" );
int max_bin_size = 0;
for (i=0; i<max_bin; i++)
{
    fprintf(bin_data, "%i %i\n", i, binindex[i]);
    if (binindex[i]>max_bin)
    {
        max_bin_size = binindex[i];
    }
}
printf("max_bin = %i\n", max_bin);

bincount[1] = binindex[0];
for (i=2; i<max_bin+1; i++)
{
    bincount[i] = binindex[i-1]+bincount[i-1];//to add up all the bins
}
printf("\n\n");
numBinRows = hit;
for (i=0; i<max_bin+1; i++)
{
    binindex[i]=0;
}

//allocate memory for bin cut array
bin_cut_path = (float **) malloc( numBinRows * sizeof(float*));
if( bin_cut_path == NULL)
{
    printf("Out of memory");
}
for(i = 0; i < numBinRows; i++)
{
    bin_cut_path[i] = (float * )malloc(numBinCols * sizeof(float));
    if( bin_cut_path[i] == NULL)
    {
        printf("Out of memory");
    }
}
}

```

```

//fill bin cut array
int bin_cut_count = 0;
printf("Filling bin cut array\n");
for (i=0; i<numPathRows; i++)
{
  if (path_array[i][9] && path_array[i][10] &&
      path_array[i][11] && path_array[i][12] > 0)
  {
    // path_array[l][4]; //E.in
    // path_array[l][5]; //E.out
    // path_array[l][6]; //rot
    // path_array[l][7]; //x1, x2 angle
    // path_array[l][8]; //x3, x4 angle
    // path_array[i][9] = (float)x1_pos;
    // path_array[i][10] = (float)y1_pos;
    // path_array[i][11] = (float)x2_pos;
    // path_array[i][12] = (float)y2_pos;
    // path_array[l][13]=-1; //-1 will assign no bin
    int bin_number = (int)path_array[i][13];
    int bin_offset = bincount[bin_number] + binindex[bin_number];
    binindex[bin_number]++;
    bin_cut_path[bin_offset][0] = path_array[i][9]; //x1 pixel number
    bin_cut_path[bin_offset][1] = path_array[i][10]; //y1 pixel number
    bin_cut_path[bin_offset][2] = path_array[i][11]; //x2 pixel number
    bin_cut_path[bin_offset][3] = path_array[i][12]; //y2 pixel number
    bin_cut_path[bin_offset][4] = path_array[i][7]; //pos1, pos2 angle
    bin_cut_path[bin_offset][5] = path_array[i][8]; //pos3, pos4 angle
    bin_cut_path[bin_offset][6] = path_array[i][4];
    //E.in //later, b (Integral Relative Electron Density)
    bin_cut_path[bin_offset][7] = path_array[i][5]; //E.out
    bin_cut_path[bin_offset][8] = path_array[i][6]; //rotation
    bin_cut_path[bin_offset][9] = path_array[i][13]; //bin number
    bin_cut_count++;
  }
}

printf("numBinRows    = %i\n", numBinRows);
printf("bin_cut_count = %i\n", bin_cut_count);

//free not needed memory
free(boundary_array);
free(path_array);

```

```
// ***** Calculate Integral Relative Electron Density *****
```

```
float *Ein, *Eout, *b, *d_E.in, *d_E.out;  
float *d_output, *d_u.temp1, *d_u.temp2, *d.temp1, *d.temp2;  
float *b2, *u.temp1, *u.temp2, *temp1, *temp22;  
Ein = (float *) malloc((bin_cut_count + 1) * sizeof(float));  
Eout = (float *) malloc((bin_cut_count + 1) * sizeof(float));  
b = (float *) malloc((bin_cut_count + 1) * sizeof(float));  
//temp variables to check accuracy of GPU  
b2 = (float *) malloc((bin_cut_count + 1) * sizeof(float));  
u.temp1 = (float *) malloc((bin_cut_count + 1) * sizeof(float));  
u.temp2 = (float *) malloc((bin_cut_count + 1) * sizeof(float));  
temp1 = (float *) malloc((bin_cut_count + 1) * sizeof(float));  
temp22 = (float *) malloc((bin_cut_count + 1) * sizeof(float));
```

```
//assign block and grid sizes for the GPU
```

```
dim3 threads(100,1,1);  
dim3 grid((bin_cut_count)/100+1);
```

```
CUDA_SAFE_CALL( cudaMalloc( (void**) &d_E.in ,  
    (bin_cut_count + 1) * sizeof(float));  
CUDA_SAFE_CALL( cudaMalloc( (void**) &d_E.out ,  
    (bin_cut_count + 1) * sizeof(float));  
CUDA_SAFE_CALL( cudaMalloc( (void**) &d_output ,  
    (bin_cut_count + 1) * sizeof(float));  
CUDA_SAFE_CALL( cudaMalloc( (void**) &d_u.temp1 ,  
    (bin_cut_count + 1) * sizeof(float));  
CUDA_SAFE_CALL( cudaMalloc( (void**) &d_u.temp2 ,  
    (bin_cut_count + 1) * sizeof(float));  
CUDA_SAFE_CALL( cudaMalloc( (void**) &d.temp1 ,  
    (bin_cut_count + 1) * sizeof(float));  
CUDA_SAFE_CALL( cudaMalloc( (void**) &d.temp2 ,  
    (bin_cut_count + 1) * sizeof(float));
```

```
for (i=0; i<bin_cut_count; i++)  
{  
    Ein[i] = bin_cut_path[i][6]*1e6; //convert to eV for calculation  
    Eout[i] = bin_cut_path[i][7]*1e6;  
}
```

```
//copy information to GPU memory
```

```
CUDA_SAFE_CALL( cudaMemcpy( d_E.in, Ein, (bin_cut_count + 1)*  
    sizeof(float), cudaMemcpyHostToDevice) );
```

```

CUDA_SAFE_CALL( cudaMemcpy( d_E_out, Eout, (bin_cut_count + 1)*
                          sizeof(float), cudaMemcpyHostToDevice) );

//calculate
relative_electron_density <<<grid, threads>>>(d_E_in, d_E_out,
        d_output, d_u_temp1, d_u_temp2, d_temp1, d_temp2);
//CPU calculation to compare results
//electron_density_CPU(bin_cut_count, Ein, Eout, b2,
        u_temp1, u_temp2, temp1, temp22);

//copy result to host memory
CUDA_SAFE_CALL( cudaMemcpy( b, d_output, (bin_cut_count + 1)*
                          sizeof(float), cudaMemcpyDeviceToHost) );

//copy result back to bin_cut_array
for (i=0; i<bin_cut_count; i++)
{
    bin_cut_path[i][6] = b[i]*1e-6;
    bin_cut_path[i][7] = 0;
}

//free GPU memory
CUDA_SAFE_CALL(cudaFree(d_E_in));
CUDA_SAFE_CALL(cudaFree(d_E_out));
CUDA_SAFE_CALL(cudaFree(d_output));
CUDA_SAFE_CALL(cudaFree(d_u_temp1));
CUDA_SAFE_CALL(cudaFree(d_u_temp2));
CUDA_SAFE_CALL(cudaFree(d_temp1));
CUDA_SAFE_CALL(cudaFree(d_temp2));
//free host memory
free(Ein);
free(Eout);
free(b);

// ***** Reconstruct Image *****
//zero image and image.update
for (i=0; i<voxels_h; i++)
{
    for(j=0; j<voxels_w; j++)
    {
        image[i][j] = 0;
        image_update[i][j] = 0;
    }
}

```



```

}

printf("Begining Image Reconstruction\n");
int num_cycles = 5;
float lambda = 0.5; //relaxation parameter used to update the image
float sig1 [2][2];
float sig2 [2][2];
float sig-temp [2][2];
float sig1_inv [1600][2][2];
float sig2_inv [1600][2][2];
float R_0 [1600][2][2];
float R_1 [1600][2][2];
double M_11;
double M_12;
double M_22;
double det_i;
float *v_image, *v_image_update;
int *v_output_index;
v_image = (float *) malloc((2*voxels_w * voxels_h + 1)*sizeof(float));
v_image_update = (float *) malloc((2*voxels_w*voxels_h+1)*sizeof(float));
v_output_index = (int *) malloc((2*voxels_w*voxels_h+1)*sizeof(int));
//vectorize the image for CUDA computation
for (i=0; i<voxels_h; i++)
{
    for (j=0; j<voxels_w; j++)
    {
        v_image[i*voxels_w+j]=image[i][j];
    }
}

//begin iterative cycle
for (int cycle=0; cycle<num_cycles; cycle++)
{
    printf("cycle %i\n", cycle);

    //for (int bin=max_bin-1; bin>=0; bin--)
    for (int bin=0; bin<max_bin; bin++)
    {
        //copy last image into the GPU
        float *d_image, *d_image_update;
        CUDA_SAFE_CALL( cudaMalloc( (void**) &d_image,
            (voxels_w * voxels_h + 1) * sizeof(float)));
        CUDA_SAFE_CALL( cudaMemcpy( d_image, v_image,

```

```

        (voxels_w * voxels_h + 1) *
        sizeof(float), cudaMemcpyHostToDevice) );

float *h_x1_pos, *h_y1_pos, *h_x2_pos, *h_y2_pos;
float *h_theta_in, *h_theta_out, *h_rotation, *h_b, *h_depth;
float *d_x1_pos, *d_y1_pos, *d_x2_pos, *d_y2_pos;
float *d_theta_in, *d_theta_out, *d_rotation, *d_b, *d_depth;
float *h_P4_11, *h_P5_11, *d_P4_11, *d_P5_11;
float *h_P4_12, *h_P5_12, *d_P4_12, *d_P5_12;
int *d_output_index;
printf("bin number %i\tbin_hist[bin] = %i\n", bin, bin_hist[bin]);
int curr_bin_size = bin_hist[bin];
int pad = 100;
h_x1_pos = (float *) malloc((curr_bin_size + pad) * sizeof(float));
h_y1_pos = (float *) malloc((curr_bin_size + pad) * sizeof(float));
h_x2_pos = (float *) malloc((curr_bin_size + pad) * sizeof(float));
h_y2_pos = (float *) malloc((curr_bin_size + pad) * sizeof(float));
h_theta_in = (float *) malloc((curr_bin_size + pad) * sizeof(float));
h_theta_out = (float *) malloc((curr_bin_size + pad) * sizeof(float));
h_rotation = (float *) malloc((curr_bin_size + pad) * sizeof(float));
h_b = (float *) malloc((curr_bin_size + pad) * sizeof(float));
        //integral relative electron density
h_depth = (float *) malloc((curr_bin_size + pad) * sizeof(float));
CUDA_SAFE_CALL( cudaMalloc( (void**) &d_x1_pos,
        (curr_bin_size + pad) * sizeof(float)));
CUDA_SAFE_CALL( cudaMalloc( (void**) &d_y1_pos,
        (curr_bin_size + pad) * sizeof(float)));
CUDA_SAFE_CALL( cudaMalloc( (void**) &d_x2_pos,
        (curr_bin_size + pad) * sizeof(float)));
CUDA_SAFE_CALL( cudaMalloc( (void**) &d_y2_pos,
        (curr_bin_size + pad) * sizeof(float)));
CUDA_SAFE_CALL( cudaMalloc( (void**) &d_theta_in,
        (curr_bin_size + pad) * sizeof(float)));
CUDA_SAFE_CALL( cudaMalloc( (void**) &d_theta_out,
        (curr_bin_size + pad) * sizeof(float)));
CUDA_SAFE_CALL( cudaMalloc( (void**) &d_rotation,
        (curr_bin_size + pad) * sizeof(float)));
CUDA_SAFE_CALL( cudaMalloc( (void**) &d_b,
        (curr_bin_size + pad) * sizeof(float)));

int index = 0;
for (i=0; i<curr_bin_size; i++)

```

```

{
  index = i + bincount[bin]; //to get data from the correct bin
  h_x1_pos[i] = bin_cut_path[index][0] * 0.25; //x1 pixel number * 0.25mm
  h_y1_pos[i] = bin_cut_path[index][1] * 0.25; //y1 pixel number * 0.25mm
  h_x2_pos[i] = bin_cut_path[index][2] * 0.25; //x2 pixel number * 0.25mm
  h_y2_pos[i] = bin_cut_path[index][3] * 0.25; //y2 pixel number * 0.25mm
  h_theta_in[i] = bin_cut_path[index][4]; //pos1, pos2 angle
  h_theta_out[i] = bin_cut_path[index][5]; //pos3, pos4 angle
  h_b[i] = bin_cut_path[index][6]; //E_in
  h_rotation[i] = bin_cut_path[index][8]; //rotation
  h_depth[i] = sqrt(pow((h_x2_pos[i]-h_x1_pos[i]),2)+
                    pow((h_y2_pos[i]-h_y1_pos[i]),2)); //mm
}

```

```

step_size = 0.0125; //cm
max_depth = (float)bin * 0.05 + 0.05; //cm
int P_steps = max_depth/step_size;
h_P4_11 = (float *) malloc((P_steps + 1) * sizeof(float));
h_P4_12 = (float *) malloc((P_steps + 1) * sizeof(float));
h_P5_11 = (float *) malloc((P_steps + 1) * sizeof(float));
h_P5_12 = (float *) malloc((P_steps + 1) * sizeof(float));
CUDA_SAFE_CALL( cudaMalloc( (void**) &d_P4_11,
                             (P_steps + 1) * sizeof(float));
CUDA_SAFE_CALL( cudaMalloc( (void**) &d_P5_11,
                             (P_steps + 1) * sizeof(float));
CUDA_SAFE_CALL( cudaMalloc( (void**) &d_P4_12,
                             (P_steps + 1) * sizeof(float));
CUDA_SAFE_CALL( cudaMalloc( (void**) &d_P5_12,
                             (P_steps + 1) * sizeof(float));

```

// ***** Sigma matrices

//Sigma 1 matrix

float u_2 = max_depth;

float u=step_size;

int j=0;

while (u<=max_depth+step_size)

{

sig1[0][0] = (pow((double)E_0,2)*pow((double)1+0.038*
log(abs(u-u_0)/X_0),2))*gaussian-quadrature(2,
s1_beta2_p2-u2, u_0, u);

sig1[0][1] = (pow((double)E_0,2)*pow((double)1+0.038*
log(abs(u-u_0)/X_0),2))*gaussian-quadrature(2,
s1_beta2_p2-u, u_0, u);

```

sig1 [1][1] = (pow((double)E_0,2)*pow((double)1+0.038*
                log(abs(u-u_0)/X_0),2))*gaussian_quadrature( 2,
                s1_beta2_p2 , u_0, u);
sig1 [1][0] = sig1 [0][1];
mat_inverse(sig1 , sig_temp);
sig1_inv [j][0][0] = sig_temp [0][0];
sig1_inv [j][0][1] = sig_temp [0][1];
sig1_inv [j][1][0] = sig_temp [1][0];
sig1_inv [j][1][1] = sig_temp [1][1];
R_0[j][0][0] = 1;
R_0[j][0][1] = u-u_0;
R_0[j][1][0] = 0;
R_0[j][1][1] = 1;
u=u+step_size;
j++;
}
//Sigma 2 matrix
j=0;
u=0;
while ( u<max_depth )
{
sig2 [0][0] = (pow((double)E_0,2)*pow((double)1+0.038*
                log(abs(u_2-u)/X_0),2))*gaussian_quadrature( 2,
                s2_beta2_p2_u2 , u, u_2);
sig2 [0][1] = (pow((double)E_0,2)*pow((double)1+0.038*
                log(abs(u_2-u)/X_0),2))*gaussian_quadrature( 2,
                s2_beta2_p2_u , u, u_2);
sig2 [1][1] = (pow((double)E_0,2)*pow((double)1+0.038*
                log(abs(u_2-u)/X_0),2))*gaussian_quadrature( 2,
                s2_beta2_p2 , u, u_2);
sig2 [1][0] = sig2 [0][1];
mat_inverse(sig2 , sig_temp);
sig2_inv [j][0][0] = sig_temp [0][0];
sig2_inv [j][0][1] = sig_temp [0][1];
sig2_inv [j][1][0] = sig_temp [1][0];
sig2_inv [j][1][1] = sig_temp [1][1];
R_1[j][0][0] = 1;
R_1[j][0][1] = u_2-u;
R_1[j][1][0] = 0;
R_1[j][1][1] = 1;

u=u+step_size;
j++;
}

```

```

}

// ***** P multiplications ***** //
for(i=0; i<P.steps; i++)
{
    step_size = 0.0125; //cm
    u = i * step_size;
    double M[2][2];
    double S1[2][2];
    double S2[2][2];
    double R0[2][2];
    double R1[2][2];
    double x[2][2];
    double y[2][2];
    double S1R0[2][2];
    double R1tS2[2][2];
    R0[0][0] = 1;
    R0[0][1] = u; // -u_0, but u_0 = 0.0mm
    R0[1][0] = 0;
    R0[1][1] = 1;
    R1[0][0] = 1;
    R1[0][1] = u_2 - u;
    R1[1][0] = 0;
    R1[1][1] = 1;
    M[0][0] = sig2_inv[i][0][0] + sig1_inv[i][0][0];
    M[0][1] = (sig2_inv[i][0][0]*R1[0][1] + sig2_inv[i][0][1]) +
                sig1_inv[i][0][1];
    M[1][1] = ((R1[0][1]*sig2_inv[i][0][0] + sig2_inv[i][1][0])*
                R1[0][1] + (R1[0][1]*sig2_inv[i][0][1] +
                sig2_inv[i][1][1])) + sig1_inv[i][1][1];
    M[1][0] = M[0][1];
    S1[0][0] = sig1_inv[i][0][0];
    S1[0][1] = sig1_inv[i][0][1];
    S1[1][0] = sig1_inv[i][1][0];
    S1[1][1] = sig1_inv[i][1][1];
    S2[0][0] = sig2_inv[i][0][0];
    S2[0][1] = sig2_inv[i][0][1];
    S2[1][0] = sig2_inv[i][1][0];
    S2[1][1] = sig2_inv[i][1][1];
    mat_mult_d(0, 0, S1, R0, S1R0);
    mat_mult_d(1, 0, R1, S2, R1tS2);
    QR_solve(M, S1R0, x);
    QR_solve(M, R1tS2, y);
}

```

```

    h_P4_11[i] = x[0][0];
    h_P4_12[i] = x[0][1];
    h_P5_11[i] = y[0][0];
    h_P5_12[i] = y[0][1];
}

if (curr_bin_size <= 100)
{
    dim3 threads(curr_bin_size, 1, 1);
    dim3 grid(1, 1, 1);
}
else
{
    dim3 threads(100, 1, 1);
    dim3 grid((curr_bin_size)/100+1);
}
voxel_size = 0.025; //cm
step_size = 0.0125; //cm

//copy data to GPU memory
CUDA_SAFE_CALL( cudaMemcpy( d_x1_pos, h_x1_pos,
    (curr_bin_size + 1) * sizeof(float),
    cudaMemcpyHostToDevice));
CUDA_SAFE_CALL( cudaMemcpy( d_y1_pos, h_y1_pos,
    (curr_bin_size + 1) * sizeof(float),
    cudaMemcpyHostToDevice));
CUDA_SAFE_CALL( cudaMemcpy( d_x2_pos, h_x2_pos,
    (curr_bin_size + 1) * sizeof(float),
    cudaMemcpyHostToDevice));
CUDA_SAFE_CALL( cudaMemcpy( d_y2_pos, h_y2_pos,
    (curr_bin_size + 1) * sizeof(float),
    cudaMemcpyHostToDevice));
CUDA_SAFE_CALL( cudaMemcpy( d_theta_in, h_theta_in,
    (curr_bin_size + 1) * sizeof(float),
    cudaMemcpyHostToDevice));
CUDA_SAFE_CALL( cudaMemcpy( d_theta_out, h_theta_out,
    (curr_bin_size + 1) * sizeof(float),
    cudaMemcpyHostToDevice));
CUDA_SAFE_CALL( cudaMemcpy( d_rotation, h_rotation,
    (curr_bin_size + 1) * sizeof(float),
    cudaMemcpyHostToDevice));
CUDA_SAFE_CALL( cudaMemcpy( d_b, h_b,

```

```

        (curr_bin_size + 1) * sizeof(float),
        cudaMemcpyHostToDevice));
CUDA_SAFE_CALL( cudaMemcpy( d_depth, h_depth,
        (curr_bin_size + 1) * sizeof(float),
        cudaMemcpyHostToDevice));
CUDA_SAFE_CALL( cudaMemcpy( d_P4_11, h_P4_11,
        (P_steps + 1) * sizeof(float),
        cudaMemcpyHostToDevice));
CUDA_SAFE_CALL( cudaMemcpy( d_P5_11, h_P5_11,
        (P_steps + 1) * sizeof(float),
        cudaMemcpyHostToDevice));
CUDA_SAFE_CALL( cudaMemcpy( d_P4_12, h_P4_12,
        (P_steps + 1) * sizeof(float),
        cudaMemcpyHostToDevice));
CUDA_SAFE_CALL( cudaMemcpy( d_P5_12, h_P5_12,
        (P_steps + 1) * sizeof(float),
        cudaMemcpyHostToDevice));
CUDA_SAFE_CALL( cudaMemcpy( d_image_update, v_image,
        (voxels_w * voxels_h + 1) * sizeof(float),
        cudaMemcpyHostToDevice));
CUDA_SAFE_CALL( cudaMemcpy( d_output_index, v_output_index,
        (voxels_w * voxels_h + 1) * sizeof(int),
        cudaMemcpyHostToDevice));

//Perform calculation
MLP_GPU<<< grid, threads >>>(step_size, voxel_size, curr_bin_size,
        d_image, voxels_w, d_rotation, d_x1_pos, d_y1_pos,
        d_theta_in, d_x2_pos, d_y2_pos, d_theta_out, d_b,
        d_image_update, d_output_index, d_depth, d_P4_11,
        d_P4_12, d_P5_11, d_P5_12);
CUT_CHECK_ERROR("Kernel execution failed");
cudaThreadSynchronize();

//Copy data to CPU memory
CUDA_SAFE_CALL( cudaMemcpy( v_image_update, d_image_update,
        (voxels_w * voxels_h + 1) * sizeof(float),
        cudaMemcpyDeviceToHost) );

//////// MLP_CPU ////////////
//Perform MLP/Reconstruction calculation on CPU
MLP_CPU(step_size, voxel_size, curr_bin_size, v_image, voxels_w,
        h_rotation, h_x1_pos, h_y1_pos, h_theta_in, h_x2_pos, h_y2_pos,
        h_theta_out, h_b, v_image_update, h_depth,

```

```

        h_P4_11, h_P4_12, h_P5_11, h_P5_12);

for (i=0;i<voxels_h;i++)
{
    for(j=0;j<voxels_w;j++)
    {
        v_image[i*voxels_w+j] = v_image[i*voxels_w+j] +
            v_image_update[i*voxels_w+j] * lambda;
    }
}

```

```

//free arrays from above
free(h_x1_pos);
free(h_y1_pos);
free(h_x2_pos);
free(h_y2_pos);
free(h_theta_in);
free(h_theta_out);
free(h_rotation);
free(h_b);
free(h_depth);
free(h_P4_11);
free(h_P5_11);
free(h_P4_12);
free(h_P5_12);
CUDA_SAFE_CALL(cudaFree(d_x1_pos));
CUDA_SAFE_CALL(cudaFree(d_y1_pos));
CUDA_SAFE_CALL(cudaFree(d_x2_pos));
CUDA_SAFE_CALL(cudaFree(d_y2_pos));
CUDA_SAFE_CALL(cudaFree(d_theta_in));
CUDA_SAFE_CALL(cudaFree(d_theta_out));
CUDA_SAFE_CALL(cudaFree(d_rotation));
CUDA_SAFE_CALL(cudaFree(d_b));
CUDA_SAFE_CALL(cudaFree(d_P4_11));
CUDA_SAFE_CALL(cudaFree(d_P5_11));
CUDA_SAFE_CALL(cudaFree(d_P4_12));
CUDA_SAFE_CALL(cudaFree(d_P5_12));
CUDA_SAFE_CALL(cudaFree(d_image));
CUDA_SAFE_CALL(cudaFree(d_image_update));
} // END BIN CYCLE LOOP
} //end cycle loop
printf("Cycles Completed\n");

```



```

//put image back to matrix form for output
for (i=0; i<voxels_h; i++)
{
    for (j=0; j<voxels_w; j++)
    {
        image[i][j]=v_image[i*voxels_w+j];
    }
}
printf("Printing image\n");
reconstruction = fopen( "reconstruction.txt", "w" );
for (i=0;i<voxels_h;i++)
{
    for(j=0;j<voxels_w;j++)
    {
        fprintf(reconstruction, "%f ",image[i][j]);
    }
    fprintf(reconstruction, "\n");
}
printf("Reconstruction complete\n\n");

//close files and free arrays
fclose(convex_hull);
fclose(input);
fclose(reconstruction);
free(bincount);
free(v_image);
free(image);
free(image_update);
free(bin_cut_path);
CUTSAFE_CALL( cutDeleteTimer(timer));
printf("Finished\n\n");
}
} //END

```

C.10 Reconstruction Kernel Code

```

// Scott McAllister
/* MLP and Integral Relative Electron Density Calculations with Cuda
 * Device code.
 */

```

```

#ifdef _RECONSTRUCTION_KERNEL_H_
#define _RECONSTRUCTION_KERNEL_H_

__global__ void
relative_electron_density(float *E_in,
                          float *E_out,
                          float *output,
                          float *u_temp1,
                          float *u_temp2,
                          float *temp1,
                          float *temp2)
{
    //integral relative electron density function using 2pt gaussian quadrature
    //calculating just a 2pt quadrature allows some simplifications
    //n is the number of points of the quad, but is not needed here
    //f is the name of the function to integrate
    //output is the output
    //u_temp1, u_temp2, temp1 and temp2 are workspace for the GPU

    int index = blockIdx.x * blockDim.x + threadIdx.x;

    float K_inv = 1/0.17;
    float C      = 299792458; // speed of light in m/s
    float I      = 75;
    float eV     = 1.602176487e-19; // eV
    float MeV    = 1e6*eV;
    float keV    = 1e3*eV;
    float m_p    = 1.672621637E-27; // mass of proton
    float e_p    = m_p*C*C/eV;
    float M_e    = 9.1093826e-31; // mass of an electron
    float M_ec2  = M_e*C*C/eV;

    //w = [1 ; 1]; //don't need w for 2pt quadrature
    float x_0 = -0.57735026918963;
    float x_1 = 0.57735026918963;

    //scilab function to integrate
    //e=E*10^6;
    //temp = (1+(e_p^2/(e^2+2*e_p)))*(log(2*M_ec2/I)+log(e)+log(e+2*e_p)-
    //      2*log(e_p)-1+(e_p^2/(e+e_p)^2))
    //output = 1/temp;

    u_temp1[index] = ((E_in[index]+E_out[index])/2)+((E_in[index]-

```

```

        E_out[index])/2)*x_0;
        //the two points to be added together
u_temp2[index] = ((E_in[index]+E_out[index])/2)+((E_in[index]-
        E_out[index])/2)*x_1;

temp1[index] = 1/((1+((e_p*e_p)/(u_temp1[index]*u_temp1[index]+2*
        u_temp1[index]*e_p)))*(log((double)2*M_ec2/I)+
        log((double)u_temp1[index])+log((double)u_temp1[index]+2*
        e_p)-2*log((double)e_p)-1+((e_p*e_p)/((u_temp1[index]+e_p)*
        (u_temp1[index]+e_p))))));

temp2[index] = 1/((1+((e_p*e_p)/(u_temp2[index]*u_temp2[index]+2*
        u_temp2[index]*e_p)))*(log((double)2*M_ec2/I)+
        log((double)u_temp2[index])+log((double)u_temp2[index]+2*
        e_p)-2*log((double)e_p)-1+((e_p*e_p)/((u_temp2[index]+e_p)*
        (u_temp2[index]+e_p))))));

output[index]=K_inv*((E_in[index]-E_out[index])/2)*(temp1[index]+
        temp2[index]); //+f(temp);
}

--global-- void MLP_GPU(float step_size, float voxel_size,
        int number_of_histories, float *x, //x is the previous iteration's image
        int width, float *rotation, float *x_in, float *y_in, float *theta_in,
        float *x_out, float *y_out, float *theta_out, float *b, float *output,
        int *output_index, float *depth,
        float *P4_11, float *P4_12, float *P5_11, float *P5_12)
{
    //step size should be half the width of a voxel
    //(eg. 0.125mm for 0.25mm voxel width)

    int index = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;

    int number_of_steps;
    float a_norm_inv;
    float lambda = 0.005; //From SAP Paper for Fully Sequential ART

    if(index<number_of_histories)
    for(int index=0;index<number_of_histories;index++)
    {
        voxel_size = 0.025; //cm
        step_size = 0.0125; //cm

```

```

number_of_steps = floor((depth[index]/10)/step_size);
a_norm_inv = (float)voxel_size/number_of_steps;

int a_i_0[1600]; //x pixel numbers
int a_i_1[1600]; //y pixel numbers
float x_dot_a = 0;

float xin = (x_in[index]-106)/10;
float xout = (x_out[index]-106)/10;
float yin = (y_in[index]-106)/10;
float yout = (y_out[index]-106)/10;
float u_offset_in = ((xin) * cos(-rotation[index])-
                    (yin) * sin(-rotation[index]));//cm
float t_offset_in = ((xin) * sin(-rotation[index])+
                    (yin) * cos(-rotation[index]));//cm
float u_offset_out = ((xout) * cos(-rotation[index])-
                    (yout) * sin(-rotation[index]));//cm
float t_offset_out = ((xout) * sin(-rotation[index])+
                    (yout) * cos(-rotation[index]));//cm
float j_rot = rotation[index];
float sin_rot = sin(rotation[index]);
float cos_rot = cos(rotation[index]);
if(j<number_of_steps)
{
float u = j*(step_size)+u_offset_in; // 'x' component (u)
float t = ((P4_11[j]*t_offset_in+P4_12[j]*theta_in[index])+
          (P5_11[j]*t_offset_out+P5_12[j]*theta_out[index]));
          // 'y' component (t)
float u2 = u*cos(rotation[index])-t*sin(rotation[index]);
float t2 = u*sin(rotation[index])+t*cos(rotation[index]);
float u3 = u2 + 10.6;
float t3 = t2 + 10.6;
a_i_0[j] = ceil(u3 / voxel_size);
a_i_1[j] = ceil(t3 / voxel_size);
}
__syncthreads();
if(j<number_of_steps)
{
if (a_i_0[j] && a_i_1[j] <= 849 && a_i_0[j] && a_i_1[j] > 0)
{
x_dot_a = x_dot_a + x[a_i_1[j]*width+a_i_0[j]] * (voxel_size);
}
}
}

```

```

__syncthreads();
output[4] = 0.1;
float update = a_norm_inv * (b[index] - x_dot_a);
if(j<number_of_steps)
for(int j=0;j<number_of_steps;j++)
{
  if (a_i_0[j] && a_i_1[j] <= 849 && a_i_0[j] && a_i_1[j] > 0)
  {
    if (index != output_index[a_i_1[j]*width+a_i_0[j]])
    {
      output[a_i_1[j]*width+a_i_0[j]] = output[a_i_1[j]*width+a_i_0[j]]+
      a_norm_inv * (b[index] - x_dot_a); //BIP
      output_index[a_i_1[j]*width+a_i_0[j]] = index;
    }
  }
}
__syncthreads();
}
} //end MLP_GPU

#endif // _RECONSTRUCTION_KERNEL.H.

```

REFERENCES

- [1] Geforce gtx 280, 2008. http://www.nvidia.com/object/geforce_gtx_280.html.
- [2] S. Agostinelli and others. Geant4 — a simulation toolkit. *Nuclear Instruments and Methods in Physics Research*, A(506):250–303, 2003.
- [3] K. M. Hanson, J. N. Bradbury, T. M. Cannon, R. L. Hutson, D. B. Laubacher, R. J. Macek, M. A. Paciotti, and C. A. Taylor. Computed tomography using proton energy loss. *Physics in Medicine and Biology*, 26:965–983, November 1981.
- [4] Gabor T. Herman. *Image Reconstruction from Projections: The Fundamentals of computerized Tomography*. Academic Press INC, London, 1st edition, 1980.
- [5] Johnson L., Keeney B., Ross G., Sadrozinski H.F.-W., Seiden A., Williams D.C., Zhang L., Bashkirov V., Schulte R.W., and Shahnazi K.
- [6] T. Li, Z. Liang, K. Mueller, J. Heimann, L. Johnson, H. Sadrozinski, A. Seiden, D. Williams, L. Zhang, S. Peggs, T. Satogata, V. Bashkirov, and R. Schulte. Reconstruction for Proton Computed Tomography: A Monte Carlo Study. In *IEEE Medical Imaging Conference*, page 3, October 2003.
- [7] T. Li, J. Singanallur, T. Satogata, D. Williams, and R. Schulte. Reconstruction for Proton Computed Tomography by Tracing Proton Trajectories: A Monte Carlo Study. *JOURNAL American Association of Physicists in Medicine*, February 2006.
- [8] K. Mueller, Z. Liang, T. Li, F. Xu, J. Heimann, L. Johnson, H. Sadrozinski, A. Seiden, D. Williams, L. Zhang, S. Peggs, T. Satogata, V. Bashkirov, and

- R. Schulte. Reconstruction for Proton Computed Tomography: A Practical Approach. In *IEEE Medical Imaging Conference*, page 3, October 2003.
- [9] Schulte RW, Bashkirov V, Klock MC, Li T, Wroe AJ, Evseev I, Williams DC, and Satogata T. Density resolution of proton computed tomography. *Med Phys.*, 32(4):1035-46, Apr 2005.
- [10] R. Schulte, V. Bashkirov, T. Li, J. Z. Liang, K. Mueller, J. Heimann, L. R. Johnson, B. Keeney, H. Sadrozinski, A. Seiden, D. C. Williams, L. Zhang, Z. Li, S. Peggs, T. Satogata, and C. Woody. Design of a Proton Computed Tomography System for Applications in Proton Radiation Therapy. *IEEE Transaction on Nuclear Science*, 51(3):866-872, June 2004.
- [11] Schulte R. W., Penfold S. N., Tafas J. E., and Schubert K. E. A maximum likelihood proton path formalism for application in proton computed tomography. *Med Phys.*, 2008.
- [12] D. C. Williams. The Most Likely Path of an Energetic Charged Particle Through a Uniform Medium. 49:2899-2911, 2004.
- [13] Robert Wilson. Radiological Use of Fast Protons. *Radiology*, (47):487-91, 1946.