X-ray Computed Tomography using a GPU

Aditya Maheshwari
School of Computer Science
Carleton University
Ottawa, Canada K1S 5B6
adityamaheshwari@scs.carleton.ca

December 12, 2019

Abstract

X-ray Computed Tomography is a leading method for visualizing the inside of an object non-destructively. Iterative Reconstruction methods are continuing to grow in popularity because they are able to produce sufficiently high quality reconstructions with less data and therefore can reduce the radiation exposure of the object being reconstructed. However, these methods are very computationally intensive, making them hard to use in practise settings simply because reconstruction times are extremely high. Graphics Processing Units (GPUs) are extremely efficient at computationally intensive problems, and they can also work well at speeding up iterative methods for XCT reconstruction!

In this paper I implement a stochastic gradient descent (SGD) based solution for XCT reconstruction and parallelize it using a GPU. These results are compared in terms of reconstruction speed to a stochastic gradient descent based solution on CPUs. The results from SGD are also are compared to a CPU based iterative coordinate descent (ICD) implementation both visually and in terms of L1 error. The conclusions show that SGD is able to reach a lower L1 error and produces different results than ICD visually. Furthermore, the parallelized SGD (on a GPU) is much faster (gives a 4X speedup on smaller 82*82 pixel reconstructions with under 10,000 X-rays and over a 30X speedup on the largest 256*256 pixel reconstructions with over 50,000 X-rays) compared to sequential SGD (on a CPU). This speedup factor continues to increase as the reconstruction sizes and numbers of X-rays get bigger.

1 Introduction

X-ray Computed tomography is a non-destructive method of visualizing of the inside of an object by using the change in X-ray attenuation to approximate the different internal densities [5]. Non-destructive analysis of an object is especially important in applications including medical analysis, mammography, geology (crystallography), and even airport security. Minimizing radiation dose and increasing reconstruction speed while maintaining reconstruction quality in XCT are important especially in a medical domain, as this reduces wait times for results, minimizes energy required per scan, and also protects the object being reconstructed from extended radiation exposure [8].

XCT reconstruction consists of a source which emits X-rays with an initial intensity, an object which is to be visualized, and a detector that captures the intensity of the X-ray after it passes through the object. The detector takes one projection measurement which

represents how much the X-ray has attenuated passing through the object at that view (and angle). We assume X-rays are strong enough not to refract, and therefore if we divide the object into pixels we know exactly which pixels any particular X-ray went through. By sending X-rays from many views and at many angles, we can divide the slice being reconstructed approximate the attenuation coefficients of the different pixels. Typically the number of views can be 90 to 180 (meaning on average one every 3 to 4 degrees rotating around the object), and for each view there can be a few hundred X-rays sent in slightly different directions.

There are methods such Filtered Backprojection or the Feldkamp algorithm that can compute these coefficients directly [10]. However, for such methods to be accurate, one needs to meet a minimum number of initial X-rays from a minimum number of views, which is not often desired due to the ill-effects of extended radiation exposure, and also makes datasets that have too few views unusable. Proposed solutions to this problem are iterative methods, which can produce reasonably high quality reconstructions given lower numbers of X-rays [1]! The problem is that because these methods are iterative, they can take an extremely long time to converge and a very large amount of computation per iteration. In addition both the time and computation increases with the amount of X-ray measurements, to a point where some computations take so long they are infeasible for most sequential machines without dividing the problem, and dividing the problem can increase the number of iterations needed even more.

Therefore, the major problems in iterative reconstruction are how to reduce dosage and how to decrease reconstruction time, while still preserving quality of reconstructions. These are related problems, because when there are less X-rays the amount of iterations required to maintain a reconstruction quality increases, while in cases with more X-rays, the time of each iteration increases. To speed up this reconstruction process, one can use a GPU to significantly reduce the time within each iteration, and potentially even parallelize different iterations.

In this paper we use two publicly available datasets and develop a version of stochastic gradient descent (SGD) that can be used on a GPU to speedup reconstruction. We compare the results of SGD on the GPU for to sequential SGD in terms of reconstruction time, and then visually compare the results of SGD with an iterative coordinate descent solution (ICD), as well as show how the image improves in quality with increased numbers of iterations. This parallel algorithm gives a 4X speedup on smaller 82*82 pixel reconstructions with under 10,000 X-rays and over a 30X speedup on the largest 256*256 pixel reconstructions with over 50,000 X-rays. The datasets used for reconstruction are of a walnut slice [4], which comes with 3 resolutions (82*82, 164*164, and 328*328), and a Lotus root that has different objects stuck in it [2], which comes with 2 resolutions (128*128, and 256*256). The walnut is interesting because it has a symmetric skeleton, but on the inside has a membrane that somewhat simulates the structure of a brain. It is feasible in the case of the walnut to get a fairly high quality reconstruction for the 328*328 case. In contrast, the lotus root allows us to catch attenuation coefficients for different materials and because it is a much more complex object with many more X-ray measurements, reconstruction quality is harder to achieve.

In this paper we will first review how GPUs have been used so far in order to speed up XCT reconstruction in the literature in section, then explain the specific problem we are looking to solve, and finish with a discussion of algorithms used in parallel for the solution, a comparison in actual reconstruction images between ICD, SGD, and how SGD results improve with more iterations, and finally discuss speed results (both CPU and GPU).

2 Literature Review

Parallel XCT reconstruction has been a popular topic especially since 2011, with the majority of research done on parallelizing iterative methods [**?**]. In this review first the fundamentals of XCT reconstruction as an optimization problem are explained, then some background on GPUs and why they can be applied to speed up reconstruction is provided, and finally the two dominant approaches to solving the reconstruction problem iteratively, Stochastic Gradient Descent (SGD) and Iterative Coordinate Descent (ICD), are analyzed.

2.1 The Reconstruction Problem

XCT methods involve an X-ray source, and detectors which measure the extent an X-ray has been attenuated (diminished) by passing through the object. By sending X-rays through the object from a series of views (slices or planes through the object), and measuring how much the X-ray attenuates (decreases in intensity) through each slice, we can produce an approximation of the objects thickness and properties. Typically the source which sends X-rays will start from a certain location, send many x-rays at different angles on the same plane through the object, all of which are picked up by the detector, and then rotate around the object by a few degrees and repeat this process until it returns to where it started. The attenuation across the paths defined by the position of the X-ray source at every slice is characterized by Beer's Law, where μ is the material's linear attenuation coefficient, I_0 is the initial intensity, and x is the length of the X-ray path [5]:

$$I = I_0 exp(-\mu x)$$

We can define all the paths of the X-rays taken from different views and at different angles using a system matrix, A, where each column represents a different voxel (n-dimensional pixel - in the case of one slice reconstruction these are just pixels) of the input image, and each row represents one X-ray. Only the voxels $j \in J$ which X-ray i passes through have a non-zero coefficient at the indices $A_{ij}, j \in J$. For the voxels which an X-ray passes through, typically the coefficient represents the length of the path of the X-ray inside that voxel. This system matrix therefore has m rows and n columns, where n represents the total number of voxels being reconstructed and m represents the number of X-rays taken in total.

The projection measurements are collected in a vector, and because there is one projection measurement per X-ray sent, this vector is of length m. The reconstructed image, which is a vector holding the attenuation coefficient of each of the voxels above, has length n because there is one attenuation coefficient per voxel. Because only a fraction of voxels are intersected by any individual X-ray, the system matrix is extremely sparse and also grows very quickly in rows and columns for a reconstruction with more voxels and more views.

Iterative Reconstruction methods minimize an objective function by iteratively updating the attenuation coefficients [9]. Mathematically, this can be represented as minimizing an objective function. Given that A is the system matrix, f is the image being reconstructed, g is the projection measurements, $D(g, Af)$ is a distance measure between the reconstructed image and the desired projection measurements, α is a weight and $R(f)$ is a regularizer, this can be written as minimizing:

$$E(f) = D(g, Af) + \alpha R(f)$$

The goal is to find values for f such that when each X-ray (row i in A) is multiplied by f, we should get a value close to g_i. This minimization problem can be solved using

SGD or ICD, which are both iterative solvers that will be discussed in more depth later in the context of this problem. Outside of parallelizing each iteration, the main speedups for SGD or ICD involve determining the optimal amount of information that will be sent into the GPU (as this is a large time cost), how much time is reasonable for pre-processing the data, and once the desired information is in the GPU how to optimize each update.

Finally, for the scope of this project we ignore regularizers, which are in practise an important component for minimizing error as they take in to account prior information about the object being reconstructed to guarantee more smoothness of the image. They are ignored in this project because when comparing reconstruction times the bulk of the computation goes into minimizing the distance term. Also, in the datasets we are using, the regularizer is not explicitly stated [1]. Therefore, we focus on minimizing $E(f)$ where $E(f) = D(g, Af)$

2.2 GPUs

A GPU has several streaming microprocessors (SMs) connected by a global memory through an interconnection network. Each SM has many registers, some shared memory, and single instruction multiple data (SIMD) pipelines. GPUs were originally and still are largely designed to provide graphic effects for entertainment, are extremely efficient at handling streams of data which all need to be processed using a series of forward streaming non-atomic (and now some atomic) operations.

The global memory in a GPU consists of many fixed size segments. When a thread executed requires global memory, it coalesces (gets grouped together with other threads from that SM) so that the accesses of threads within that warp all transact with global memory at the same time. One main difference between a GPU and general purpose processor is that each thread performing computation is not expected to access many different segments of the systems memory. This way, there can be thousands of threads running at the same time, which is often the case in graphics. However, if multiple non-atomic instructions inside the same warp attempt to write to the same memory address for more than one of the threads, only one (randomly chosen) thread can finish this operation, which is known as a memory collision. A memory collision can also occur when threads from different warps try writing to the same global address at the same time.

On the other hand, if memory accesses within a warp reside in multiple segments of the global memory, this will require multiple memory transactions where on each transaction many threads are sitting idle. This problem is known as non-coalesced memory access. A high amount of non-coalesced memory accesses reduce the probability that there will be a memory collision and vice versa, meaning that minimizing one of these issues can still lead to inefficient results. Any memory transaction usually causes some latency, therefore irregular memory accesses are usually the largest factor in degrading performance.

While dealing with memory slows down performance, GPUs are popular and extremely fast for repetitive data processing seen in loops and apply statements general coding jobs because they are efficient at performing computations. GPUs in theory can work very well for situations like XCT reconstruction, because we are processing each X-ray projection in a similar way. However, a few general rules are important to follow when using GPUs to take advantage of improved performance [7]:

1. Each pipeline should focus on outputting a single data element (one voxel) rather than many output data elements (multiple voxels), so that many inputs are processed

in parallel to produce one output, rather than being forced to write across the entire memory into many outputs

2. Conditional statements significantly restrict the number of workers that can run so should be avoided

3. Loading sufficient amounts data to fill pipelines is more efficient as GPUs tend to be abundant in bandwidth instead of latency

4. Perform extra computation rather than look up values from a table as GPUs are extremely fast at computation and slow down significantly when they have to search across a lot of memory at each step

Modern GPUs also come with texture memory, which were originally designed for graphics applications where memory access patterns have a spatial locality [7]. Because all the measurements from tomographic data are read-only, these can sit in texture memory (or the L1 cache), and therefore can be accessed faster than they would be as a part of global memory, provided the system matrix is represented in a small enough format. Next we discuss the specifics of each solver (SGD and ICD), and finish with a discussion on how parallelize each solver and for what components.

2.3 Iterative Solvers

To find the local minimum of an objective function iteratively, the two most common approaches are stochastic gradient descent and iterative coordinate descent. Both algorithms take steps proportional to the negative gradient of the objective function from the current point. However, in traditional parallel gradient based algorithms, multiple workers will compute the gradient of each component of the full data (which sits in memory) and then synchronize before the start of each iteration [6]. Synchronizations, as explained above, cause significant performance degradation. Because the system matrix is often sparse, and the fidelity term $D(g, Af)$ is the most significant part of the computation within each iteration, there is a way to perform these computations without too much synchronization.

2.3.1 Stochastic Gradient Descent (SGD)

Stochastic gradient descent works by taking the gradient with respect to all the arguments for the objective function and then moving a pre-determined step size in the opposite direction of the gradient until the net change in position falls inside of some predetermined error value ϵ. In this case, this means SGD chooses a set of projection measurements to update at each step. Mathematically, on each iteration we repeat the following two steps until the image f converges, where γ is the learning rate (step size):

1. $delta \leftarrow \nabla(D(g, Af))$

2. $f \leftarrow f + \gamma * delta$

In most cases D is simply the L2-norm or mean squared error. In this case, the gradient is derived as follows:

$$D(g, Af) = \frac{1}{2}\|g - Af\|^2 = A^T(g - Af)$$

This is one matrix vector multiplication, followed by a vector-vector subtraction, followed by another matrix vector multiplication! Clearly we can do each of these operations in parallel using the GPU, so within each iteration we should get a significant speedup here especially for a larger matrix size.

2.3.2 Iterative Coordinate Descent (ICD)

Iterative coordinate descent finds which direction has the steepest descent and then moves until the gradient of that particular value is zero but only in one direction at one time. There are methods of parallelizing ICD in a sparse matrix as well [3]. For XCT reconstruction, this means updating one voxel at each step, once again using the negative gradient with respect to that voxel! To do this we repeat the following steps for each voxel until the image converges, where f_j is the current voxel being updated in the reconstruction vector f, and e represents the error of the reconstruction on each voxel ($e \leftarrow g - Af$):

1. $\hat{f}_j = f_j$
2. $\theta_1 = \sum_{i=1}^{M} A_{ij} e_i$
3. $\theta_2 = \sum_{i=1}^{M} A_{ij}^2$
4. $f_j \leftarrow argmin_{r \geq 0} \{\theta_1 r + \frac{\theta_2 (r - \hat{f}_j)^2}{2}\}$ (*)
5. $e \leftarrow e + A_{*j}(f_j - \hat{f}_j)$

The main step here is calculating the min value at step 4 [12], which, similar to above is derived as follows:

$$0 = \frac{\partial}{\partial r}(*) = \theta_1 + \theta_2 (r - \hat{f}_j)$$

$$r = max(\frac{-\theta_1}{r\theta_2}, -\hat{f}_j)$$

This is a much more complicated set of computations for a GPU, because at each step we have to send a new "column" of the matrix into a graph to then compute all of the above. This increases the time of each iteration significantly because we have to loop through each pixel which requires looking back and then sub setting a part of the memory into the computation graph each time. However, in a single-core implementation where memory accesses are not as significant in degrading performance, this will converge in much fewer iterations. One more big difference between SGD and ICD implementations are that in ICD, without a regularizer since each pixel is being optimized independently, the resulting image looks more like a mosaic, and also mathematically does not converge to as low of an L1 error. In SGD, on the other hand, the resulting image looks more "smooth". In Section 3, methods of evaluating reconstructions are further explained.

Next, we quickly discuss differences between reconstructing one slice of an image and a full 3D image, as well as discuss further ways to parallelize within an iteration.

2.3.3 Thread Mapping Optimization

Different X-rays will intersect on some voxels, but we don't want multiple threads updating the same voxels simultaneously, if we are specifying exactly which thread updates which voxel. Threads in different warps generally will not attempt to update the same voxel, because the GPU has a scheduler which is fairly optimized and will avoid such collisions. This means the real challenges is preventing memory collisions from the same warp, meaning we need a way to group the X-rays so that two X-rays in the same group do not attempt to write to the same voxels at the same time. If we choose X-rays that are far away to update in the same warp, we will have many non-coalesced memory accesses which can also slow down performance. Therefore, a collision friendly scheme results on non-coalesced accesses, and the same vice versa.

We can make this a mathematical problem by creating a graph where each node represents an X-ray, and each edge between two nodes holds a weight representing how many voxels they share for updates. We then want to find a colouring to reduce inter-warp memory collisions as this causes the largest delay, and then find a partition within each colour minimizing the non-coalesced memory accesses. The initial colouring problem for inter-warped memory can be done with any graph colouring algorithm - in the paper [6] they use the Welsh-Powell algorithm. Each sub-graph with a different colour now needs to be optimized to reduce non-coalesced memory accesses, where the edge between two nodes represents the number of memory addresses that can be coalesced. For two X-rays, this is the same as the number of voxels that can reside in the same segments. We want to ideally partition nodes from the same colour into sub-graphs containing 32 nodes (representing 32 X-rays, meaning each thread in the warp is responsible for one X-ray). After all components have been processed, we can map the X-rays into a vector to form warps and then distribute warps into multiple thread blocks evenly (all of which can be done offline). This scheme gets transferred to the GPU and only needs to be transferred once as the whole algorithm is only related to the geometry of the XCT scanner and object size.

We can also use the same approach in ICD but combining voxels that do not appear in the same X-ray very frequently into the same update. In order to do this, we can once again use a graph algorithm similar to what is described above, or we can trace the sinogram information that has been stored to determine which projections involve each voxel. The second method is known as using a super voxel buffer. This method is unlikely to help in a single slice situation, because our scanner is a 1D array and so the patterns are not as clearly different from voxel to voxel. The more efficient partitioning methods for determining what voxels go in a super voxel buffer method rely on a third axis which is used to determine which voxels are unlikely to appear in the same scan [11].

A drawback to these approach which has not explicitly been mentioned in the literature is that for a problem of such size (10,000 to 60,000 X-rays), finding such a partition in groups of 32 is not trivial! When added to the time of training, while it is true that the GPU can train faster knowing which partitions can be parallelized and still get extremely comparable results, the time of creating such a partition is not worth it. Unless knowledge of the scanner behaviour is known ahead of time, it is not feasible to perform this reconstruction. Therefore, for a 3D object which has a much larger number of voxels to update and a low number of X-rays taken, this might be feasible to add to the computation, but is not very realistic for a single slice in either SGD or ICD with many projections.

3 Problem Statement

In this paper we want to parallelize iterative CT reconstruction methods for a 2D slice of an object. This is a variation of the above problems in the literature review, because instead of reconstructing a full 3D object using parallel ICD we focus on reconstructing 2D slices of the object using SGD and parallelize within each iteration. We then compare these results to the ones obtained with ICD in terms of image quality, and SGD on single cores for speed. Furthermore, we focus on minimizing the fidelity term rather than testing different regularizers because this is the main challenge and limitation of speed in computations. Because the distance term will result in a very similar reconstruction at a given number of iterations (whether they are done on GPU or CPU does not affect the result outside of differences in floating point calculations), the goal is to have a significant speed up in total time using a GPU. To more precisely define what a significant speed up time means, we will evaluate the total time it takes to train once all information is passed into the GPU, the time it takes to load data into the GPU, and also the time it takes to do pre-processing of the data, and compare this to the CPU which does not require loading but does also require some pre-processing. Finally, we aim to create an implementation in Tensorflow so that in the future we can integrate this approach with machine learning to further improve the quality of images in future research.

4 Solution, Implementation Details, and Reconstructed Images

In this section the solution for implementing SGD for XCT reconstruction on GPUs is provided. We discuss the datasets in some more detail, the pre-processing of the data for SGD on the GPU, evaluation metrics, and finally the reconstructed images are displayed both with the ICD implementation and the SGD implementation.

4.1 Datasets

We use 5 different tomographic datasets on two different items, 3 for the walnut slice reconstruction and 2 for lotus root reconstructions. In both datasets, a large number of X-rays are taken from one source but at different angles, after which the object is rotated by 3 degrees (or similarly the X-ray source and detector are rotated by 3 degrees), and once again a large number of X-rays are taken from the new view. Details about the size of the datasets are below:

1. The 82*82 pixel walnut slice reconstruction has 82 projections per view (120 views), meaning the system matrix A is 9870*6724
2. The 164*164 pixel walnut slice reconstruction has 164 projections per view (120 views), system matrix A is 19680*26896
3. The 328*328 pixel walnut slice reconstruction has 328 projections per view (120 views), system matrix A is 39360*107584
4. The 128*128 pixel lotus root reconstruction has 429 projections per view (120 views), system matrix A is 51480*16384

5. The 256*256 pixel lotus root reconstruction has 429 projections per view (120 views), system matrix A is 51480*16384

Therefore the Lotus reconstruction datasets have many more X-rays, but the third Walnut dataset has a higher pixel count.

4.2 Pre-processing

For smaller matrices it is possible both in a single core and in a GPU implementation to send the data as is, but because GPUs support sparse matrices, the most "heavy" step is reading and storing the data in Tensorflow as a sparse matrix. This step ranges from 1.5 seconds for the smallest dataset to around 30 seconds for the largest one. Loading this representation into the GPU can take anywhere from 6 to 120 additional seconds. Furthermore, because we never modify the system matrix, we can send this into the read-only memory of the GPU. Once this representation is on a GPU, we can run the loop described in the literature review for gradient descent to go through iterations, and then parallelize the computation inside each iteration because the only modifications in memory are the reconstruction vector. Because everything else is just read only and therefore can be accessed from the local caches of the GPU more frequently, and parallelizing the matrix multiplications gives a more significant speedup factor for larger matrix sizes (proportional to the problem size), this in theory should lead to a much faster reconstruction.

Because we are only minimizing the distance function and there is no randomness in terms of sample selection, both single core (CPU) and parallel (GPU) implementations lead to the same quality reconstruction at the same number of iterations; the difference is in the amount of time it takes to perform an iteration. These reconstructions (CPU and GPU) are compared for speed, the actual GPU reconstructions are also compared to see the difference in reconstruction quality across the number of iterations, and the GPU reconstruction times are also compared to ICD reconstruction times for the smaller datasets (1, 2, and 4 above) just to visually assess the difference. Next, the ICD solution is explained briefly.

4.3 Iterative Coordinate Descent

We run iterative coordinate descent in a sequential form using the formulas stated in the literature review, so that we are able to contrast the different styles of images produced. ICD has been the most heavily studied parallel reconstruction algorithm in the 3D case, but in all of the most popular papers [3], [12], [11], [9] the solution depends on a third axis to create voxel groups. In the 2D case, ICD effectively converges within 20 iterations, but the size of each iteration tends to be longer as we have to update the voxel values sequentially. This still in theory is faster than SGD because SGD requires thousands of iterations to converge. ICD has been used extensively in the past, while SGD has only appeared as a solution more recently [6], and therefore it is important to check that the results are of a similar quality. Furthermore, the error converges at a higher number than SGD, and the image looks much closer to a mosaic, and because in ICD it is harder to implement regularizers that minimize the change between adjacent voxels without significantly degrading performance (especially in parallel voxel updates), for many medical applications it's not a practical solution [6]. Error metrics are discussed below, followed by the actual reconstructed images.

4.4 Error metrics

There are many different error metrics in this problem, though we use L1 norm explicitly to evaluate quality of image; lower L1 norm means that the reconstruction is of better quality on a pixel by pixel level. This L1 norm is the sum of the absolute difference between Af and g where g is the projection measurement vector, A is the system matrix, and f is the reconstructed image ($sum(abs(Af - g))$). While this is an absolute metric, we can also visually make an assessment of the reconstruction qualities. Because we are not incorporating regularizers, there is usually a lower bound for how good the reconstruction can get purely based on L1 norm. Furthermore, any errors in the actual X-ray machine or dataset are out of our control. This does not mean that being able to do more iterations in the same amount of time is useless just because the L1 norm does not decrease as significantly in the extra iterations, as in many real applications the smallest improvement can lead to completely different diagnosis. Therefore, we use the next sub-section to evaluate actual quality of images and the next Section, 5, to evaluate the speedups.

4.5 Reconstructions

We will start by examining ICD vs SGD, and then move on to viewing SGD reconstructions as the number of iterations increases. It is important to note that in the plots we do not need to evaluate the colour on the outsides of the object (which is all air), or the colour of any portions in the object that are supposed to be hollow (air). Instead we just look for a consistency in background colour, and then evaluate all the dark portions for reconstruction quality. The only reason the background colours appear different is because the plotting software looks to emphasize a contrast in attenuation coefficients; the goal therefore is to compare consistencies in the solid parts of the objects and also compare the contrasts between different materials.

4.5.1 ICD vs SGD Reconstructions

We show ICD vs SGD comparisons for the 82*82 Walnut, 164*164 Walnut, and 128*128 Lotus, as these are the only examples with comparable reconstruction times (for larger examples GPU SGD is much faster than sequential ICD and so it is not worth comparing).

ICD for the 82*82 Walnut reconstruction converges at 98 (L1 norm), and takes 19.36 seconds to get under 100. On the GPU, it takes 17.94 seconds of training to get under 100, plus an additional 14.76 seconds for processing and inputting the data. However, SGD can train the L1 norm down to as low as 85 in 113 seconds (plus the additional 15 for sending the data). The only apparent difference between the reconstruction qualities shown in Figure 1 is that the membrane in the middle of the Walnut membrane looks slightly more smooth. This suggests while ICD reaches a good reconstruction slightly faster, SGD can reach a "better" reconstruction given more time.

ICD for the 164*164 Walnut reconstruction converges at 185 (L1 norm), and takes on average 120.4 seconds to get under 190. On the GPU, it takes 76 seconds to get under 190, plus an additional 46.3 seconds for processing and inputting the data. However, SGD can go as low as 175 and does this in 150 seconds (plus the additional 76 for for sending the data). We see a much more apparent difference in the reconstructions shown in Figure 2, specifically the shell and inside shell are more clearly defined, and the inner membrane is much more smooth. This suggests that the GPU can reach a good solution slightly faster, but then can continue to improve as well, and that the loading time is the biggest difference.

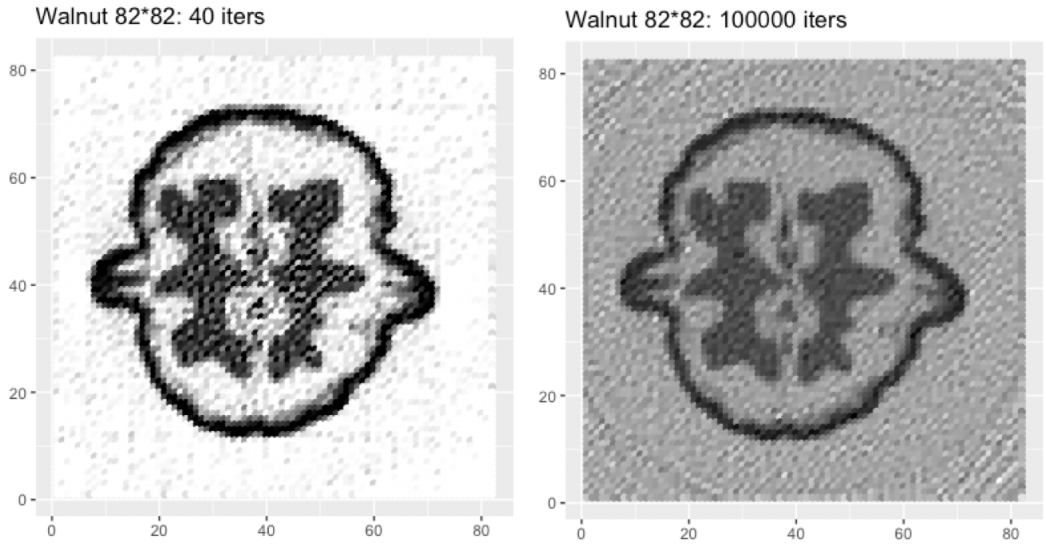

Figure 1: Walnut reconstructions for 82*82 with ICD (Left) and SGD (Right)

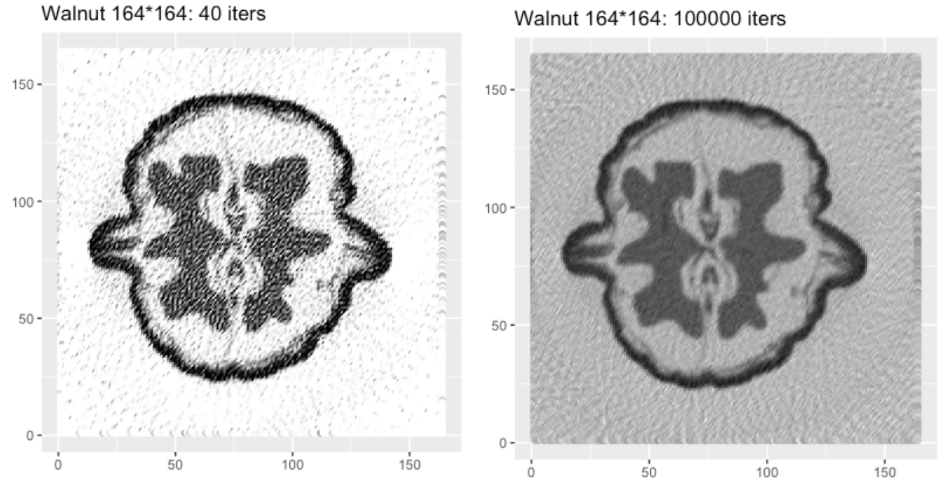

Figure 2: Walnut reconstructions for 164*164 with ICD (Left) and SGD (Right)

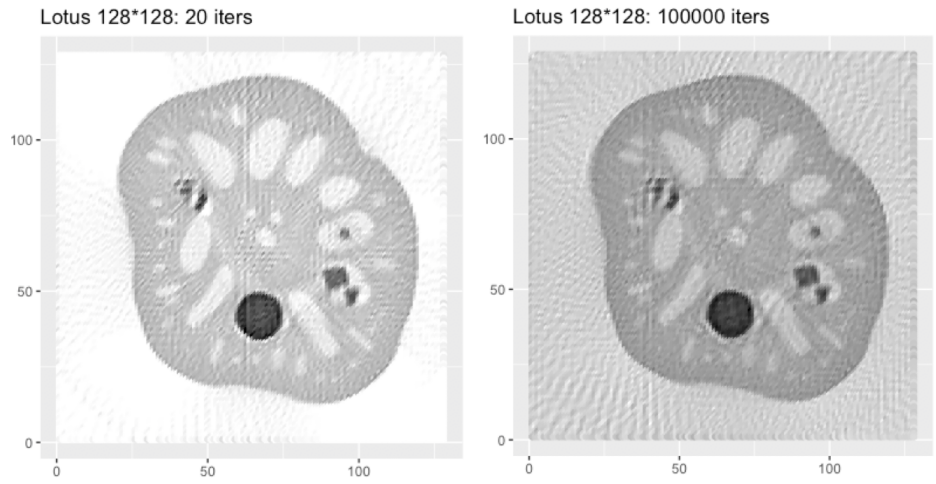

Figure 3: Lotus reconstructions for 128*128 with ICD (Left) and SGD (Right)

ICD for the 128*128 Lotus reconstruction converges at 1420 (L1 norm), and takes 180 seconds to get under 1450. On the GPU, it takes less than 12 seconds to get under 1420, plus an additional 159 seconds for processing and inputting the data. However, the GPU can go as low as 1377 and does this in 139 seconds (plus the additional 159 for convergence). The actual reconstruction quality is a little harder to determine visually in the reconstructions shown in Figure 3. However, there is a significant difference in training time when the data is in the GPU, to reach a better quality of reconstructions. This example shows both that a different L1 error does not always mean the image is clearly better, and also that ICD is not always fast at reaching a good solution compared to a GPU, especially as the reconstruction size and number of X-rays grows larger.

4.5.2 Best Quality Reconstructions

We next show a difference between best quality reconstructions visually across different numbers of pixels. For the Walnut dataset, clearly the image quality between 82*82 and 164*164 is drastically different, and 328*328 looks slightly better especially at defining the difference between hollow and filled spaces in the slice as seen in Figure 4. We can also see the increase in reconstruction times even on the GPU; this does not including loading and initialization time (that will be discussed in section 5).

As a contrast, in the case of lotus root reconstructions, it does not seem like there as significant a difference between lower and higher quality reconstructions between 128 and 256 in Figure 5! The only difference is that the hollow portions inside the Lotus have a slightly more clearly defined boundary in the 256*256, while the difference between hollow and not hollow in 128*128 are more blended together.

4.5.3 Progression of Reconstruction Quality With Increased Iterations

We see in the progression of iterations for the 82*82 Walnut in Figure 6, that after 15000 iterations the image stays fairly clear. The difference between 10000 and 15000 iterations in terms of L1 norm is a difference of 16, and 15000 to 20000 is 3. For 164*164 the main change in clarity happens between 20000 and 25000 (difference of 112), as shown in Figure

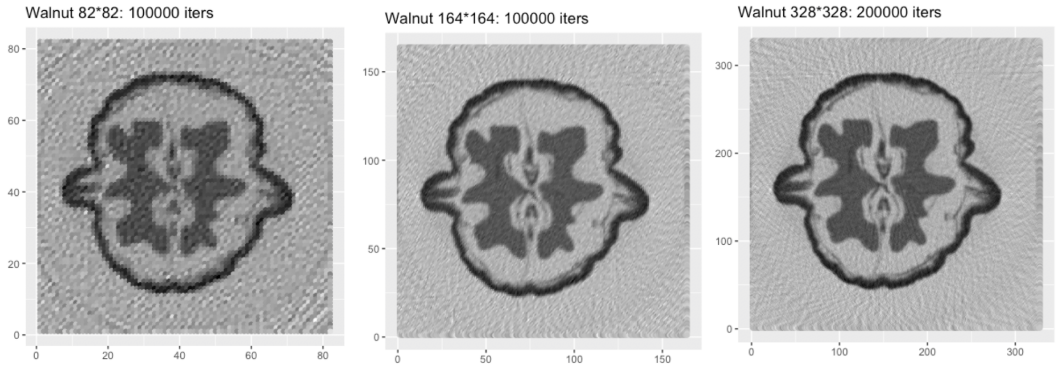

Figure 4: Walnut Reconstructions for 82*82 (left) which has L1 error 86 and was trained in 113 seconds, 164*164 (center) which has L1 error 174 and was trained in 150 seconds, and 328*328 (right) which has L1 error 359 and was trained in 716 seconds

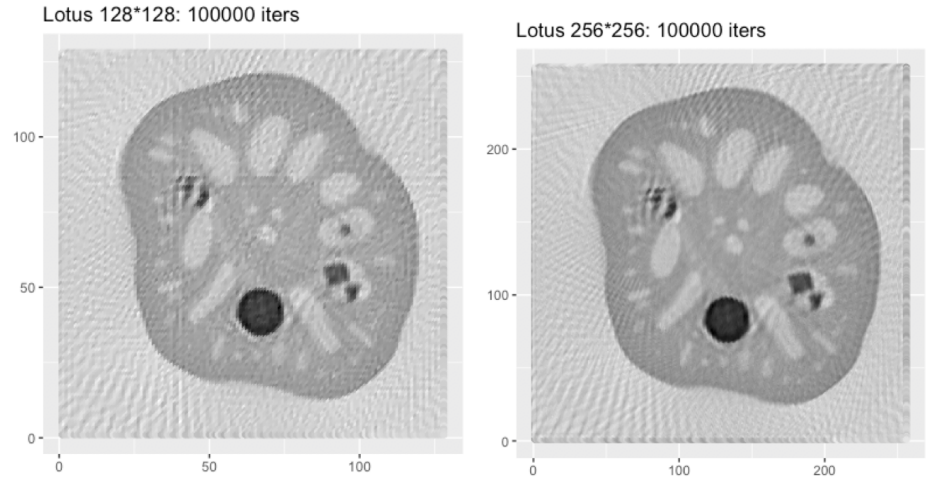

Figure 5: Lotus Reconstructions for 128*128 (left) which has L1 error 1377 and was trained in 139 seconds, 256*256 (right) which has L1 error 1346 and was trained in 667 seconds

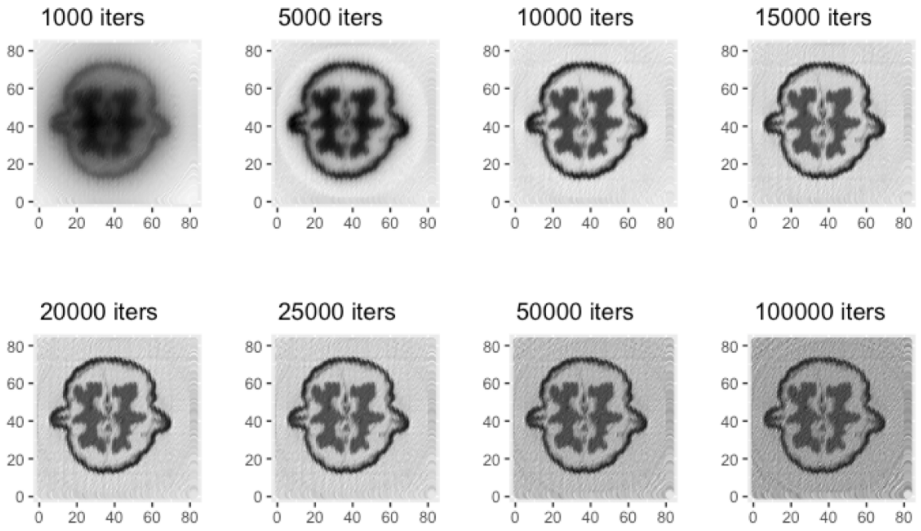

Figure 6: W82 progression across iterations; clarity between 10000 and 15000

7. For 328*328, the main change is from 50000 to 100000 (difference of 538), shown in Figure 8.

For the Lotus Root slice, on the other hand, for 128*128 the main change in clarity is from 5000 to 10000 (difference in L1 norms of 4 units), shown in Figure 9, and for 256*256 it happens from 10000 to 15000 (difference of 9 units), shown in Figure 10.

5 Experimental Evaluation

In this section we compare the different speed curves for reconstructions on all 5 datasets. Here are a few key points to help read each of the graphs shown:

- The horizontal (x) axis represents elapsed time (training time)
- The vertical (y) axis represents L1 error (also can be called absolute error)
- Blue lines represent GPU times
- Red lines represent CPU times
- The dotted line represents purely training time (not including pre-processing, loading data into the GPU, or initializing Tensorflow
- The solid line represents full reconstruction time (pre-processing + loading + initializing + training). Please note that there is a pre-processing step in the CPU case as well, though it is not as long as the GPU case!
- The size of any points represents the number of iterations that have elapsed at that point, with the largest circle representing the largest amount of iterations elapsed. The progression of circles (which increase as a line is followed), is 1000, 5000, 10000, 15000, 20000, 25000, 50000, 100000, and sometimes 200000!

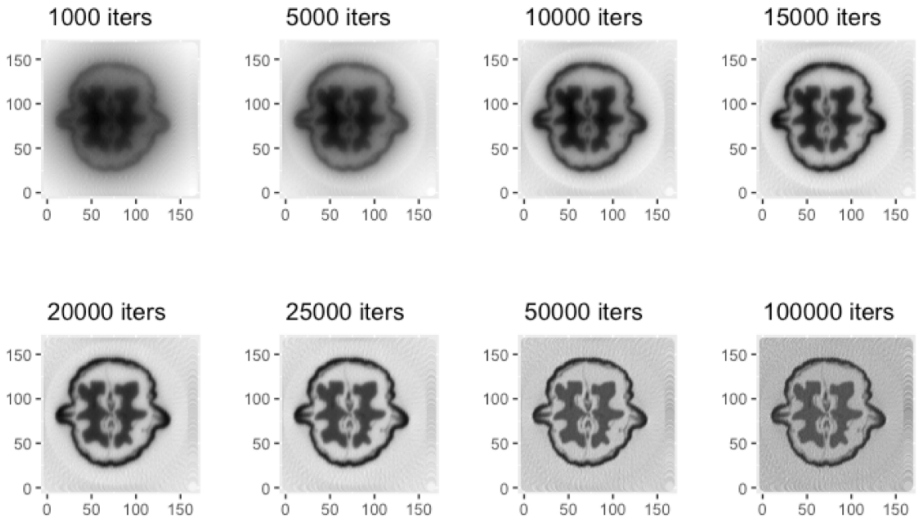

Figure 7: W164 progression across iterations; clarity between 20000 and 25000

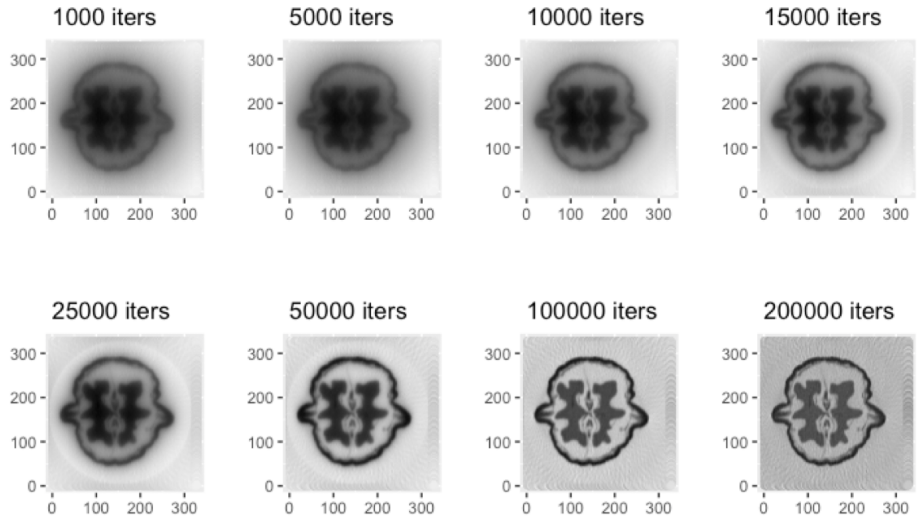

Figure 8: W328 progression across iterations; clarity between 50000 and 100000

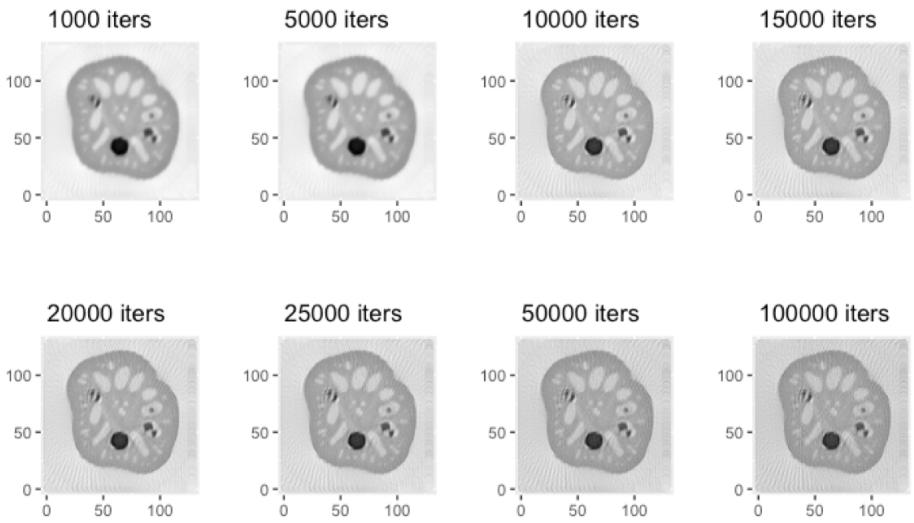

Figure 9: Lotus 128 Progression Across Iterations; clarity between 5000 and 10000

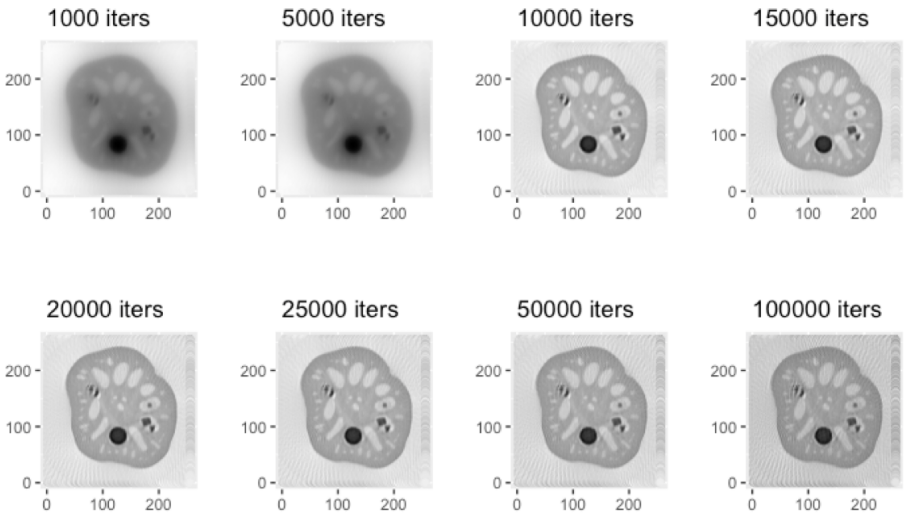

Figure 10: Lotus 256 Progression Across Iterations; clarity between 10000 and 15000

Figure 11: 82*82 Walnut; Time vs L1 Error vs Processor Type (CPU/GPU) vs Iterations Elapsed

- Often the CPU is not trained for as many iterations as the GPU. This is because the actual L1 or absolute error will be very similar at the same number of iterations, the only difference is the speed of reconstruction.

These comparisons are between a CPU implementation on tensorflow and a GPU still on tensorflow, which in both cases is able to handle large sizes of data but should train faster on a GPU in theory, because of the reasons explained in Section 2.

5.1 82*82 Walnut Reconstruction

For the 82*82 Walnut speed vs error curves shown in Figure 11, we see that initially when factoring in all the costs (loading data etc), the training on a CPU is faster at 1000 iterations, and also that the CPU reaches 10000 iterations close around 5 second before the GPU reaches 50000 iterations (73.35 seconds total). Overall, the GPU is able to run 100000 iterations before the single core can run 25000. Overall, this suggests that the GPU has around a 4x speedup converges.

5.2 164*164 Walnut Reconstruction

For the 164*164 Walnut speed vs error curves shown in Figure 12, we see that initially when factoring in all the costs (loading data etc), the training on a CPU is once again faster at 1000 iterations. However, the GPU is able to run 100000 iterations before the CPU can reach 10000 iterations. This suggests that the GPU has around a 10x speedup prior to convergence.

5.3 328*328 Walnut Reconstruction

For the 328*328 Walnut speed vs error curves shown in Figure 13, the GPU remains faster by a significant margin from start to finish. In this case the GPU can run 200000 iterations

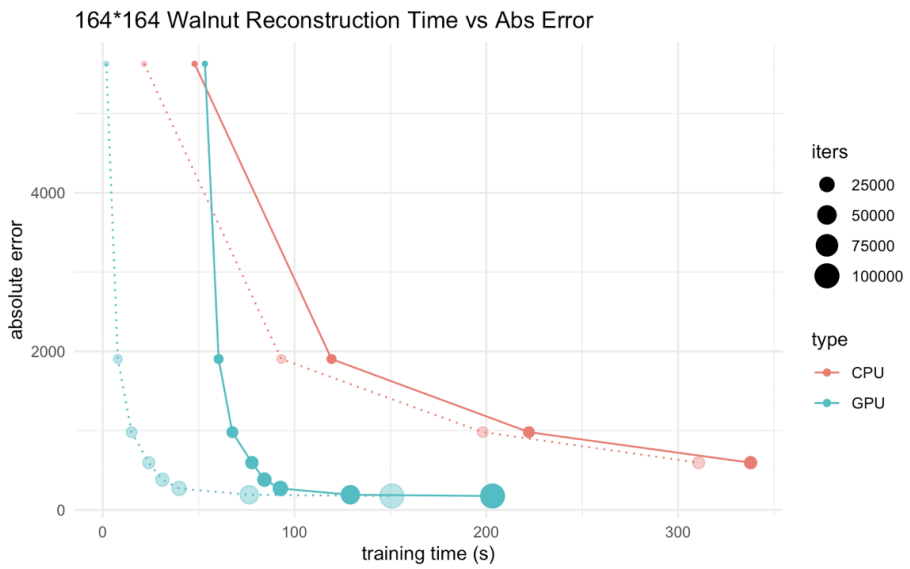

Figure 12: 164*164 Walnut; Time vs L1 Error vs Processor Type (CPU/GPU) vs Iterations Elapsed

in faster time than the CPU runs 10000, which suggests more than a 20X speedup. Keep in mind that the pre-computation costs are done on a single core for both machines, but the GPU has a higher loading cost because sending the data is more expensive (it takes 85 more seconds to send the data), but this is offset because the size of the matrix is large enough that the operation inside each iteration is much faster.

5.4 128*128 Lotus Reconstruction

For the 128*128 Lotus speed vs error curves shown in Figure 14, the GPU remains faster from start to finish. In this case the GPU can run 100000 iterations in around the same time that the CPU runs 5000, which suggests more than a 20X speedup. The precomputation is still large for this size, however it is important to note that because there are 51480 X-rays (projections) sent in this dataset, the matrix size remains large, which makes each iteration much faster on the GPU.

5.5 256*256 Lotus Reconstruction

For the 256*256 Lotus speed vs error curves shown in Figure 15, the GPU remains faster from start to finish. In this case the GPU can run 100000 iterations in around 2/3 of the time that the CPU runs 5000, which suggests more than a 30X speedup. The precomputation is still large for this size, however it is important to note that because there are 51480 X-rays (projections) sent in this dataset, the matrix size remains large. A good sign is that for larger reconstructions, the speedup factor is much higher which is a very good sign.

5.6 Instructions

The code attached contains an RmD file, which is used to create all the reconstruction pictures and parallel plots. It also contains an ipynb used to run experiments on a CPU,

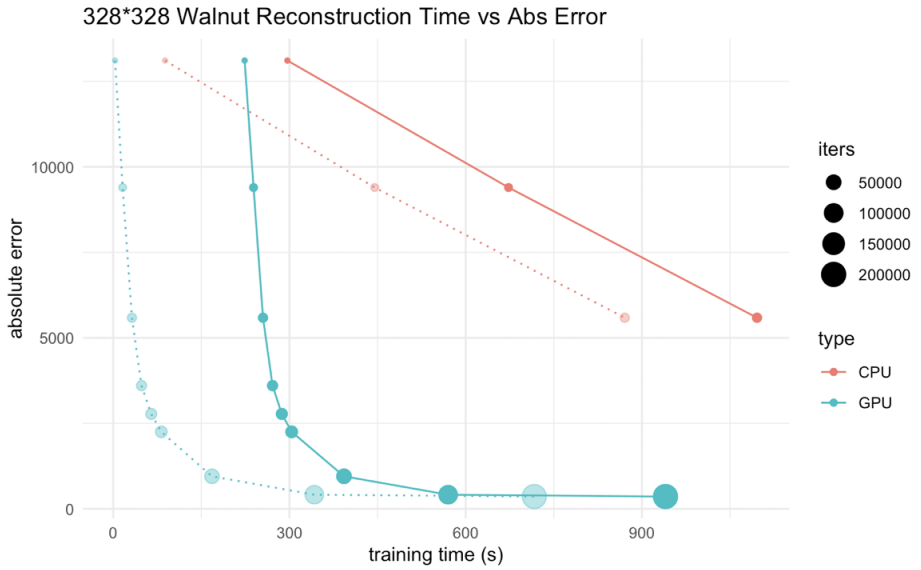

Figure 13: 328*328 Walnut; Time vs L1 Error vs Processor Type (CPU/GPU) vs Iterations Elapsed

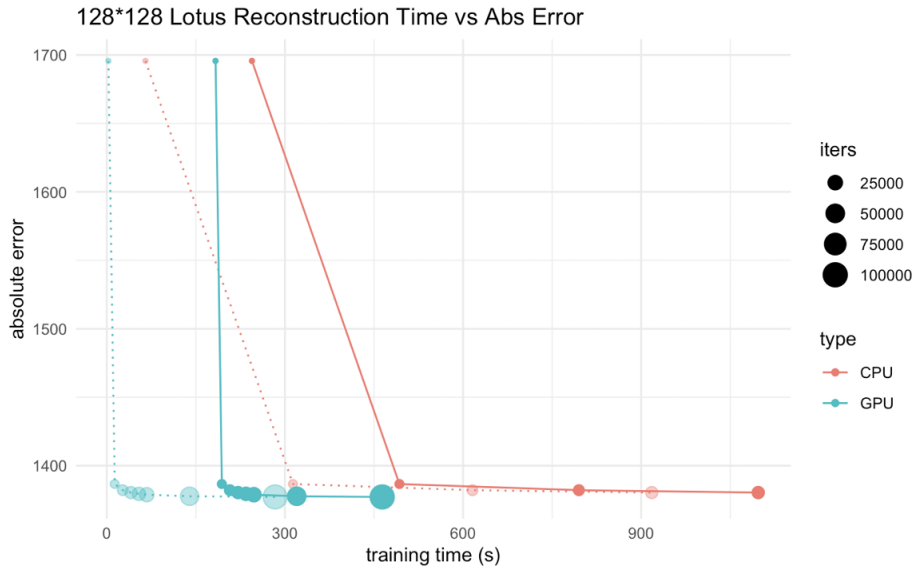

Figure 14: 128*128 Lotus; Time vs L1 Error vs Processor Type (CPU/GPU) vs Iterations Elapsed

Figure 15: 256*256 Lotus; Time vs L1 Error vs Processor Type (CPU/GPU) vs Iterations Elapsed

and another file used for the GPU. Instructions on how to reproduce results are contained there. Links to the dataset can be found at [2] and [4].

6 Conclusions

In conclusion, we see that running SGD with all the projections on each iteration can be specifically sped up using a GPU vs CPU, and that SGD trained for a long time can produce better reconstructions (if the evaluation criteria is the L1 norm or absolute error) than ICD, which theoretically and in practise requires less iterations. For the smaller reconstructions, this speedup is roughly 4X, but there is not a huge difference in terms of reconstruction quality (visually) in the later iterations. For larger reconstructions, we definitely notice a significant difference in quality between the later iterations, and so a speedup of 30X is far more prevalent and noticeable! Furthermore, SGD does produce smoother images, and also images where the boundaries between different materials are much more clear.

The main contributions or findings are as follows:

1. SGD converges to more accurate results than ICD when purely minimizing the distance term in reconstructions for a single slice
2. It is worthwhile to use a GPU because of the significant improvements in speedup and increasing speedup factor for larger reconstructions; 30X for 256*256 with 51480 X-rays, and 20X for 328*328 with 39360 X-rays

Moving forward, future contributions and extensions of the project can include:

1. Extending this approach to 3D reconstructions (it's possible that we can do this more efficiently than just extending the number of columns in a system matrix for additional voxels, maybe we can use the information from one slice's reconstruction to more efficiently recreate the next slice)

2. Adding regularizers with prior knowledge and testing results on more datasets as well as using X-ray hardware specification to parallelize ICD multi-voxel updates

3. It's possible that combining ICD and SGD can lead to even more accurate reconstructions without a regularizer, though this can be hard to do in parallel

4. We can combine the results of many reconstructions with deep learning methods to get a more smooth and realistic final image

References

[1] Marcel Beister, Daniel Kolditz, and Willi A Kalender. Iterative reconstruction methods in x-ray ct. *Physica medica*, 28(2):94–108, 2012.

[2] Tatiana A Bubba, Andreas Hauptmann, Simo Huotari, Juho Rimpeläinen, and Samuli Siltanen. Tomographic x-ray data of a lotus root filled with attenuating objects. *arXiv preprint arXiv:1609.07299*, 2016.

[3] Sungsoo Ha and Klaus Mueller. A gpu-accelerated multivoxel update scheme for iterative coordinate descent (icd) optimization in statistical iterative ct reconstruction (sir). *IEEE Transactions on Computational Imaging*, 4(3):355–365, 2018.

[4] Keijo Hämäläinen, Lauri Harhanen, Aki Kallonen, Antti Kujanpää, Esa Niemi, and Samuli Siltanen. Tomographic x-ray data of a walnut. *arXiv preprint arXiv:1502.04064*, 2015.

[5] Richard A Ketcham and William D Carlson. Acquisition, optimization and interpretation of x-ray computed tomographic imagery: applications to the geosciences. *Computers & Geosciences*, 27(4):381–400, 2001.

[6] Xiuhong Li, Yun Liang, Wentai Zhang, Taide Liu, Haochen Li, Guojie Luo, and Ming Jiang. cumbir: An efficient framework for low-dose x-ray ct image reconstruction on gpus. In *Proceedings of the 2018 International Conference on Supercomputing*, pages 184–194. ACM, 2018.

[7] Klaus Mueller, Fang Xu, and Neophytos Neophytou. Why do commodity graphics hardware boards (gpus) work so well for acceleration of computed tomography? In *Computational Imaging V*, volume 6498, page 64980N. International Society for Optics and Photonics, 2007.

[8] Michael J Paulus, Shaun S Gleason, Stephen J Kennel, Patricia R Hunsicker, and Dabney K Johnson. High resolution x-ray computed tomography: an emerging tool for small animal cancer research. *Neoplasia (New York, NY)*, 2(1-2):62, 2000.

[9] Amit Sabne, Xiao Wang, Sherman J Kisner, Charles A Bouman, Anand Raghunathan, and Samuel P Midkiff. Model-based iterative ct image reconstruction on gpus. *ACM SIGPLAN Notices*, 52(8):207–220, 2017.

[10] Jean-Baptiste Thibault, Ken Sauer, Charles Bouman, and Jiang Hsieh. Three-dimensional statistical modeling for image quality improvements in multi-slice helical ct. In *Proc. International Conference on Fully 3D Reconstruction in Radiology and Nuclear Medicine*, pages 271–274, 2005.

[11] Xiao Wang, Amit Sabne, Sherman Kisner, Anand Raghunathan, Charles Bouman, and Samuel Midkiff. High performance model based image reconstruction. In *ACM SIGPLAN Notices*, volume 51, page 2. ACM, 2016.

[12] Xiao Wang, Amit Sabne, Putt Sakdhnagool, Sherman J Kisner, Charles A Bouman, and Samuel P Midkiff. Massively parallel 3d image reconstruction. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 3. ACM, 2017.