IMPLEMENTATION OF ITERATIVE SOLVERS FOR THE DIGITAL TOMOSYNTHESIS PROBLEM IN GPUS

VERONICA MEJIA BUSTAMANTE *

Abstract. Tomosynthesis imaging provides a viable alternative to computed tomography (CT) and has obtained significant interest from the medical community as a means for diagnostic radiology and radiation therapy. In digital tomosynthesis imaging, multiple projections of an object are obtained along a small range of different incident angles in order to reconstruct a 3D representation of the object. In this paper we discuss the implementation details of the polyenergetic digital breast tomosynthesis reconstruction algorithm in a GPU using OpenCL. We describe three different algorithm implementations: a serial implementation, a GPU implementation threaded by functionality of the model, and a GPU fused kernel implementation which is threaded to increase performance, throughput, and GPU utilization in the application. We show that the explicit kernel fusion achieves significant speed-up in the reconstruction process of a clinical size patient data set, from running over 100X faster than the version threaded by functionality to 200X faster than the serial approach.

Key words. tomography, tomosynthesis, gradient descent, maximum likelihood, OpenCL, GPU, fused kernel

AMS Subject Classifications: 65F20, 65F30

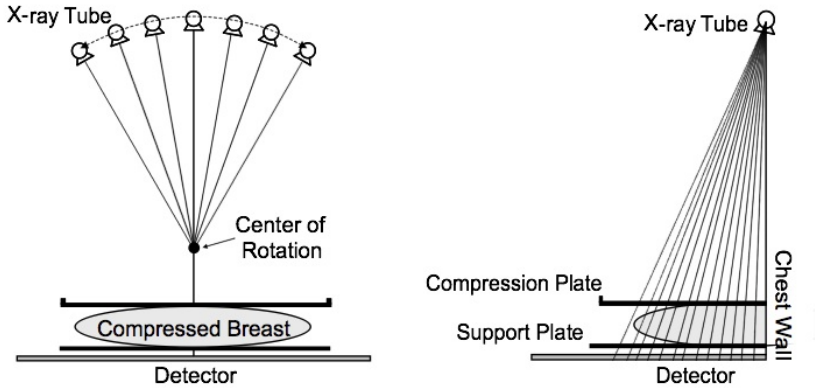
1. Introduction.

In digital tomosynthesis imaging, multiple projections of an object are obtained along a small range of different incident angles in order to reconstruct a 3D representation of the object. This technique is of relevant interest in breast imaging screening since it requires less radiation and a shorter acquisition period than a CT breast scan, which obtains projections of the object around a full 360° rotation [2, 3]. In addition, breast tomosynthesis imaging can ultimately provide more details as a screening technology than a standard mammography, since mammography takes a single 2D projection of the object and many features could be missed [12] . An example of the imaging device for breast tomosynthesis imaging can be seen in Figure 1. Here we have a rotating x-ray source that can take 2D projections of the breast at different incident angles, typically about 15 to 30 projections in a range of about ±15°. The size of the detector can be about 1280 by 2048 pixels and the size of the resulting reconstruction of the object is about 50 to 70 slices (depending on the height of the object) each slightly smaller in size than the detector.

The multiple 2D projections obtained as the source rotates allow for more information to be used in the reconstruction process and a more accurate representation of the object. However, more input data greatly increases the size of the problem and managing this amount of data in the reconstruction algorithm can be an issue, as most of the reconstruction process becomes memory bound. Direct and naive approaches to implement the tomosynthesis reconstruction can be costly in both memory and time. Since these results are meant to be used in a clinical setting, it is unfeasible to allow reconstruction algorithms to run for hours or days, physicians must be able to make a timely diagnosis to their patients.

In this paper we describe the implementation details of the polyenergetic digital breast tomosynthesis reconstruction process using a single GPU to accelerate computations. We focus on efficient handling of memory communication and increasing computational performance to achieve the best results. The paper is organized as follows: Section 2 describes the polyenergetic mathematical model and the iterative solver used in the reconstruction process. This section recounts the work presented by

*Mathematics and Computer Science, Emory University. vmejia@emory.edu



(a) Front view (b) Side view with X-ray tube at 0°

FIG. 1.1. *Example of the imaging device for breast tomosynthesis imaging.*

Chung, Nagy and Sechopoulos in [1] and extends their process to a quadratic model approach. Section 3 discusses in detail three implementations of the reconstruction algorithm, a serial approach, a threading by functionality approach, and a fused kernel approach, all written in C++ and running on a GPU using OpenCL [10]. Section 4 presents numerical results of all three implementations and shows that a fused kernel approach is the most effective in our tomosynthesis reconstruction. Section 5 outlines our conclusions and future work on the subject.

2. Reconstruction Framework. In this section we discuss the mathematical model for the problem and the iterative solvers proposed to generate the 3D reconstructions. Tomosynthesis reconstruction is a nonlinear inverse problem, where the goal is to approximate the true volume from the given set of projection images of the object. In order to do this we must have a physically accurate model of the forward problem, like the polyenergetic model, and a well suited iterative solver. This section briefly outlines the work presented by Chung, Nagy and Sechopoulos in [1]

2.1. The Mathematical Model. In modeling the forward problem for the image acquisition process, it is important to take into account the polyenergetic source spectrum. Previous approaches assume that the x-ray source is monoenergetic, i.e. all incident photons have the same energy level (see [1] for a discussion of previous approaches). This assumption results in beam hardening, which occurs when there is absorption of low-energy photons from the x-ray by the object, thus changing the average energy of the x-ray beam. Severe artifacts can appear in the reconstruction using a monoenergetic model.

The image acquisition model for the breast tomosynthesis problem can be described as:

$$b_i^{(\theta)} = \int_{\varepsilon} \rho(\varepsilon) e^{-\int_{L_\theta} \mu(\varepsilon,x)dl} d\varepsilon + \eta_i^{(\theta)}, \ i = 1 : N_p \text{ and } \theta = 1 : N_\theta \tag{2.1}$$

where N_p is the number of pixels in the detector, N_θ is the number projections obtained by rotating the x-ray source to a new position, ε represents the spectrum of energies that are emitted by the source x-ray beam, which can range, for example,

from 10keV to 28keV, $\rho(\varepsilon)$ is the product of the x-ray energy with the number of incident photons at that energy, L_θ is the line on which the x-ray beam travels through the object, $\mu(x,\varepsilon)$ is the attenuation coefficient dependent on both energy of the x-ray beam and the material composition of the object at position x, and $\eta_i^{(\theta)}$ represents any additional noise during the acquisition process.

The continuous model can then be discretized using N_v voxels and N_ε discrete energy levels to obtain:

$$b_i^{(\theta)} = \sum_{e=1}^{N_\varepsilon} \rho(e)\, \exp\left(-\sum_{j=1}^{N_v} \mu(e)^{(j)} a_\theta^{(ij)} \right) + \eta_i^{(\theta)}, i = 1 : N_p \text{ and } \theta = 1 : N_\theta \tag{2.2}$$

The new quantity $a_\theta^{(ij)}$ represents the length of the x-ray beam that passes through voxel j and contributed to pixel i for incident angle θ. We can collect these $a_\theta^{(ij)}$ quantities to construct the $N_p \times N_v$ matrix A_θ to simplify notation.

Now consider the attenuation coefficients $\mu(e)$. Each voxel in the true object is interpreted as a composition of glandular tissue, adipose tissue, or the two combined. Thus we can use the percentage of glandular tissue present in the voxel to quantify the constitution of the reconstructed voxel. This quantity is known as the percentage glandular fraction and is related to the energy-dependent attenuation coefficients of the voxel through an algebraic transformation [4]. Having this quantification of the material composition of the object, we can model the attenuation coefficient for each voxel as:

$$\mu(e)^{(j)} = h(e)(g_{true}^{(j)})^2 + s(e) g_{true}^{(j)} + z(e) \tag{2.3}$$

for $j = 1 : N_v$, where $g_{true}^{(j)}$ represents the glandular fraction in voxel j of the "true" object and $h(e)$, $s(e)$ and $z(e)$ are known energy-dependent fit coefficients determined as described in [1]. This formulation of $\mu(e)$ combined with the discretization of the problem in equation (2.2) allows us to write each projection image acquired as:

$$\mathbf{b}^{(\theta)} = \sum_{e=1}^{N_\varepsilon} \varrho(e) \exp\left[-h(e) A_\theta \mathbf{g_{true}^2} + s(e) A_\theta \mathbf{g_{true}} + z(e) A_\theta \mathbf{1} \right] + \eta^{(\theta)}, \tag{2.4}$$

for $i = 1 : N_p$ and $\theta = 1 : N_\theta$. Here we have the exponential function applied component-wise, $\mathbf{g_{true}}$ and $\mathbf{g_{true}^2}$ are vectors whose corresponding jth entries are $(g_{true}^{(j)})^2$ and $g_{true}^{(j)}$ respectively, and $\mathbf{1}$ is a vector of all ones.

2.2. Iterative Reconstruction Process. The reconstruction of the 3D volume from the given 2D projections in tomosynthesis is a numerical optimization problem, where we find the maximum likelihood estimator (MLE) of the Poison-based likelihood function stemming from the image acquisition model shown in (2.4) see [1]. The expected value of the measured data is

$$\bar{b}_\theta^{(i)} + \bar{\eta}_\theta^{(i)} = \sum_{e=1}^{N_\varepsilon} \varrho(e) \exp\left(-h(e) \mathbf{a_i^T g^2} + s(e) \mathbf{a_i^T g} + z(e) \mathbf{a_i^T 1} \right) + \bar{\eta}_i^{(\theta)} \tag{2.5}$$

where $\mathbf{a_i^T}$ is the ith row of A_θ and $\bar{\eta}_i^{(\theta)}$ represents error which is assumed to follow a Poisson distribution whose statistical mean is known. This allows us to write the

negative log likelihood function as:

$$-L_\theta(\mathbf{g}) = \sum_{i=1}^{N_p} (\bar{b}_\theta^{(i)} + \bar{\eta}_\theta^{(i)}) - b_\theta^{(i)} \log(\bar{b}_\theta^{(i)} + \bar{\eta}_\theta^{(i)}) + c \tag{2.6}$$

for all θ where c is a constant and $\mathbf{g}$ is the unknown volume. We then use a gradient descent method [6,8] to solve:

$$g_{MLE} = \min_{\mathbf{g}} \left\{ \sum_{\theta=1}^{N_\theta} -L_\theta(\mathbf{g}) \right\} \tag{2.7}$$

3. Implementation. In this section we discuss the implementation and computational aspects of the reconstruction process in digital breast tomosynthesis. The reconstruction algorithms are all implemented in objective C++ using template code for single and double precision. The GPU implementation uses the OpenCL framework [10].

The implemented reconstruction process is a gradient descent algorithm [6,8] seeking to minimize the negative log likelihood function shown in equation (2.7). As we are looking for g_{MLE} it is clear that equation (2.7) must be evaluated several times in the reconstruction process. The most computationally intense part of the function evaluation are the dot products in equation (2.5) or, equivalently, the matrix-vector product in equation (2.4). The size of matrix A_θ is the number of pixels in all projections by the number of voxels in the reconstruction, which translates to millions of rows by billions of columns. Computing this matrix-vector product explicitly is not realistic, we need three matrix products per function evaluation and two matrix transpose-vector products per derivative evaluation. We discuss alternate ways to compute this product and ways to optimize the reconstruction process even further.

3.1. Matrix Product Optimizations. To compute the matrix-vector products with A_θ and A_θ^T we note that the matrix is large, sparse and each of its entries has a physical meaning in our reconstruction. Each $a_\theta^{(ij)}$ represents the length of the x-ray beam that passes through voxel j and contributes to pixel i for incident angle θ. This is a well-known matrix in radiation therapy, where computing the matrix-vector product with A_θ is known as a *raytrace* and computing the matrix-vector product with A_θ^T is known as a *back projection*. These products can be computed very efficiently using the algorithm developed by Siddon [13].

The first step we take to optimize the performance of the tomosynthesis reconstruction process is to implement an efficient version of the raytracing algorithm to compute the matrix-vector products. Siddon's algorithm [13] is much more efficient than an explicit matrix-vector product but it poses two major drawbacks: a sorting procedure at its core and a large amount of memory for temporal variables. These two issues can be a problem for CPU implementations but pose an even greater challenge for GPU computing. M. de Greef et al [7] devised a different version of Siddon's algorithm for radiation therapy applications that is GPU friendly, it eliminates the need for sorting and reduces the amount of memory for local variables to fit in the limited global memory of a GPU. We use a modified version of their approach to compute the matrix-vector products in our algorithm, reducing the amount of time in computing the products by at least a third in the CPU case (we did not implement Siddon's algorithm [9] on a GPU). In addition, we gain speed-ups in the computation of the matrix-vector products by performing them as $A_\theta[\mathbf{u}\ \mathbf{v}\ \mathbf{w}]$ and $A_\theta^T[\mathbf{x}\ \mathbf{y}]$ in one

function call as opposed to having three separate function calls in the first case and two function calls in the second case.

3.2. Serial Implementation. We begin by describing the serial implementation of the reconstruction algorithm as a point of comparison for our GPU approaches. The algorithm scheme for the serial case is described in Approach 1 of Figure 3.1. We begin each gradient descent iteration by a function evaluation (equation 2.7) which is a three step process. First we compute the the matrix-vector multiplications needed for equation (2.5), then we apply the quadratic fit using the coefficients $h(e)$, $s(e)$ and $z(e)$ from equation (2.5), and finally we perform the summations over all pixels and all angles in Equations (2.6) and (2.7). After we complete the function evaluation, we begin to compute the derivative of the function in two steps. First we compute the matrix transpose-vector products needed and then we compute the derivative using a linear combination of the two product results. We do a line search to guarantee descent of the function and then start the gradient iteration with a new starting $\mathbf{g}$. The process is repeated until we are unable to guarantee descent in the value of the function.

3.3. Functional Kernel Implementation. The first approach to accelerate the performance of the reconstruction algorithm using a single GPU is to thread the application for functionality, that is, create a kernel for each computationally intensive portion of the process and allow the device to run that portion before continuing the process. Here we use OpenCL to run the computations. The choice to use OpenCL as opposed to CUDA [9] was solely based on portability, we would like the code to be as accessible as possible to all clinical settings. We have a CUDA implementation in the works and all of our experiments are run using Nvidia cards.

Profiling the serial code shows that the reconstruction algorithm spends the majority of the time computing the matrix-vector products. Even as optimized by using the modification of Siddon's algorithm [7, 13], the matrix-vector product (raytracing) in serial is very computationally intensive. However, it can be noted that the raytracing algorithm is well suited for the threading capabilities of a GPU. Ultimately, we are following the geometry of the x-ray beam as it passes through the object and is contributing to each pixel in the detector. For a particular incident angle θ we can parallelize the raytrace by considering each pixel in the detector a single GPU thread. Each thread can then trace the angle from the source along the object and determine the extent of the contribution from the x-ray beam to its value in the detector. Each thread only needs to know the simple geometric set-up of the problem, the incident angle at which the beam approaches, and the attenuation of the voxels in the object (this is the vector by which A_θ is being multiplied), all other variables are local to the thread. There is no thread communication or synchronization needed, other than the starting geometry of the problem which can be loaded into global space and then moved to the shared space of each work group (or thread block). Since each thread will be ultimately computing a single output value for the pixel corresponding to its thread ID in the execution model, there is no chance of a race condition or data hazards.

Having the set-up described above of how to implement a single $A_\theta\mathbf{u}$ multiplication, we can modify it slightly to compute $A_\theta[\mathbf{u}\ \mathbf{v}\ \mathbf{w}]$. The only thing we need is enough memory in the global space of the GPU to fit all three vectors $\mathbf{u}$, $\mathbf{v}$ and $\mathbf{w}$. Then we can have each thread calculate three output values, as if we had the same ray going through three different volumes and landing in three different detectors. The thread is responsible for following the ray, take into account the attenuation of each

volume **u**, **v** and **w**, and report as output the value of the pixel corresponding to the thread ID at each of the three detectors.

Finally, to complete this implementation we note that we must compute $A_\theta[\mathbf{u}\ \mathbf{v}\ \mathbf{w}]$ for all θ (usually between 15 to 30 angles). This presents us with two alternatives: compute each $A_\theta[\mathbf{u}\ \mathbf{v}\ \mathbf{w}]$ individually in the GPU, or compute all products $A_\theta[\mathbf{u}\ \mathbf{v}\ \mathbf{w}]$ for $\theta = 1 : N_\theta$ at once in the GPU. Memory traffic between the CPU and GPU over the PCIe bus is expensive and we want to avoid the GPU sitting idle waiting for a new vector. The geometry details remain the same through all N_θ multiplications; the only value that changes is the single value for the incident angle at which the x-ray beam approaches. Here we can take advantage of the 3D execution topology that GPU computing offers. We can use the third dimension to represent each incident angle. Now each thread will query its incident angle after it obtains the geometry parameters and perform its required raytracing calculations.

With all these optimizations we can then compute three matrix-vector products for all incident angles with one single GPU function call. The matrix transpose-vector product (back projection) follows a similar approach. In this case, we let each voxel constitute an individual GPU thread responsible for determining its value based on the given projection images. Approach 2 of Figure 3.1 shows the new algorithm scheme, now moving all matrix-products to the GPU.

3.4. Kernel Fusion Implementation. After moving all matrix-products to the GPU we are able to attain some speed-ups in our reconstruction. However, we still can optimize the approach even further. Profiling the new reconstruction shows that the algorithm is now spending minimal time computing the matrix-vector products, but we have a new bottle neck as we apply the fit coefficients to the results of the GPU, see equation (2.5). Applying the fit coefficients, summing over all energies in equation (2.5), and summing over all pixels in equation (2.6) is a memory bound computation. We have a triple nested loop, the inner loop goes over all voxels (order of a billion), the outer loop goes over all energies (order of ten), the outside loop goes over all pixels (order of a million). Loading this triple loop into the GPU is not an option, since even within the core of the loop we still have memory bound operations to update the variables.

A viable approach is to enhance the raytracing kernel we have running, since its output consists of all three quantities $a_i^T\mathbf{g}^2$, $a_i^T\mathbf{g}$ and $a_i^T\mathbf{1}$. Keeping these quantities in the device, we can load the fit coefficients into the GPU (given we have sufficient memory available) and perform the fit inside the kernel. In addition, we can use the exponential function provided by the OpenCL standard to compute the exponential portion. Now the output of the kernel is the vector $\bar{b}_\theta^{(i)} + \bar{\eta}_\theta^{(i)}$ from equation (2.5). Once the kernel execution is completed, all we have is a single loop over all pixels in the CPU which can run significantly faster since it is only three lines of memory bound computations that use the CPU cache to hide memory latency. We call this a kernel fusion implementation because we have fused the nested loops into one kernel call as opposed to having two separate kernels for the matrix product and the application of the fit coefficients to the data.

One final optimization we have included in this approach is that CPU-GPU communication during the reconstruction process is absolutely minimal. The kernel calls for both the raytracing and back projection operation are almost identical since in both cases we just need to communicate to the geometry of the problem and the necessary vectors to the device. We have created a function call prior to the beginning of the reconstruction to load all necessary variables into the global memory of the GPU.

This leads to only having to transfer the necessary vectors to the device at the time of enqueueing the kernel. Now this approach spends most of the computation time running in the GPU and allows for much faster line search computations. There is no additional optimizations needed by profiling the code because, as can be seen in the next section, we are performing clinical size patient reconstructions in under five minutes.

4. Numerical Results. In this section we show timings for our numerical experiments using the three implementations described in the previous section. The reconstructions run in objective C++ and OpenCL for the GPU case. We use the Nvidia Tesla c2070, which has 6GB of global memory for the larger size problems. For the smaller reconstruction we use the Nvidia GeForce GT330M.

Figure 4.1 shows the results of using a single function call for three matrix-vector products versus each product performed at a time. The bottom row shows the results for two matrix transpose-vector products performed in a single function call as opposed to one individual product performed. There is a significant speedup in computing $A_\theta[\mathbf{u}\ \mathbf{v}\ \mathbf{w}]$ versus $A_\theta\mathbf{u}$, $A_\theta\mathbf{v}$, and $A_\theta\mathbf{w}$ on their own. Computing $A_\theta[\mathbf{u}\ \mathbf{v}\ \mathbf{w}]$ takes longer than computing $A_\theta\mathbf{u}$ because we have additional overhead costs when we reference three different objects in three separate detectors as described in Section 3.3. The same results holds for the back projection computation.

Figure 4.2 shows the computation time for each portion of a single gradient iteration. Note that the GPU time is computed using the fused kernel approach. For the fused kernel approach implementation, we consider the ray trace plus function evaluation loop as one kernel call as described in Section 3.4. In this case, the ray trace timing is shown to compare with the function evaluation loop timing reported. The function evaluation loop accounts for the ray trace in its computation time. This table shows how both bottle necks in the reconstruction process have successfully been eliminated.

Figure 4.3 shows the results of the a full reconstruction for three different problem sizes. This reconstruction takes about three gradient iterations. Note that the most significant speed-up is achieved in the last case which is the full size patient reconstruction. We would explain the greater speedup in the larger case as enough computational need for the GPU to compensate for the memory transfer between the CPU and GPU in the reconstruction algorithm. A slice from the reconstruction is shown in Figure 4.4. The reconstruction is that of a phantom used to simulate a breast.

5. Conclusions. In this paper we have discussed the major implementation details for the digital breast tomosynthesis problem. We have developed three approaches to solve the problem, a serial implementation, an implementation threaded for functionality, and a fused kernel approach using OpenCL [10]. We have shown that the bottle necks of the application are the matrix-vector products and the triple nested loop in the function evaluation. Avoiding an explicit matrix-vector product and using an alternative to Siddon's algorithm [13] we can compute fast matrix-vector products on a GPU. By fusing the function evaluation loops and extending the fuctionality of the raytracing kernel we reduce the bottle neck created by the memory bound computations of the function evaluation. Allowing minimal communication between CPU and GPU also reduces the time the GPU sits idle. Using all these performance optimizations, we have shown that fused kernel approach attains the best performance, as it reconstructs the full clinical size data 135X faster than the threaded for functionality approach and 200X faster than the serial implementation.

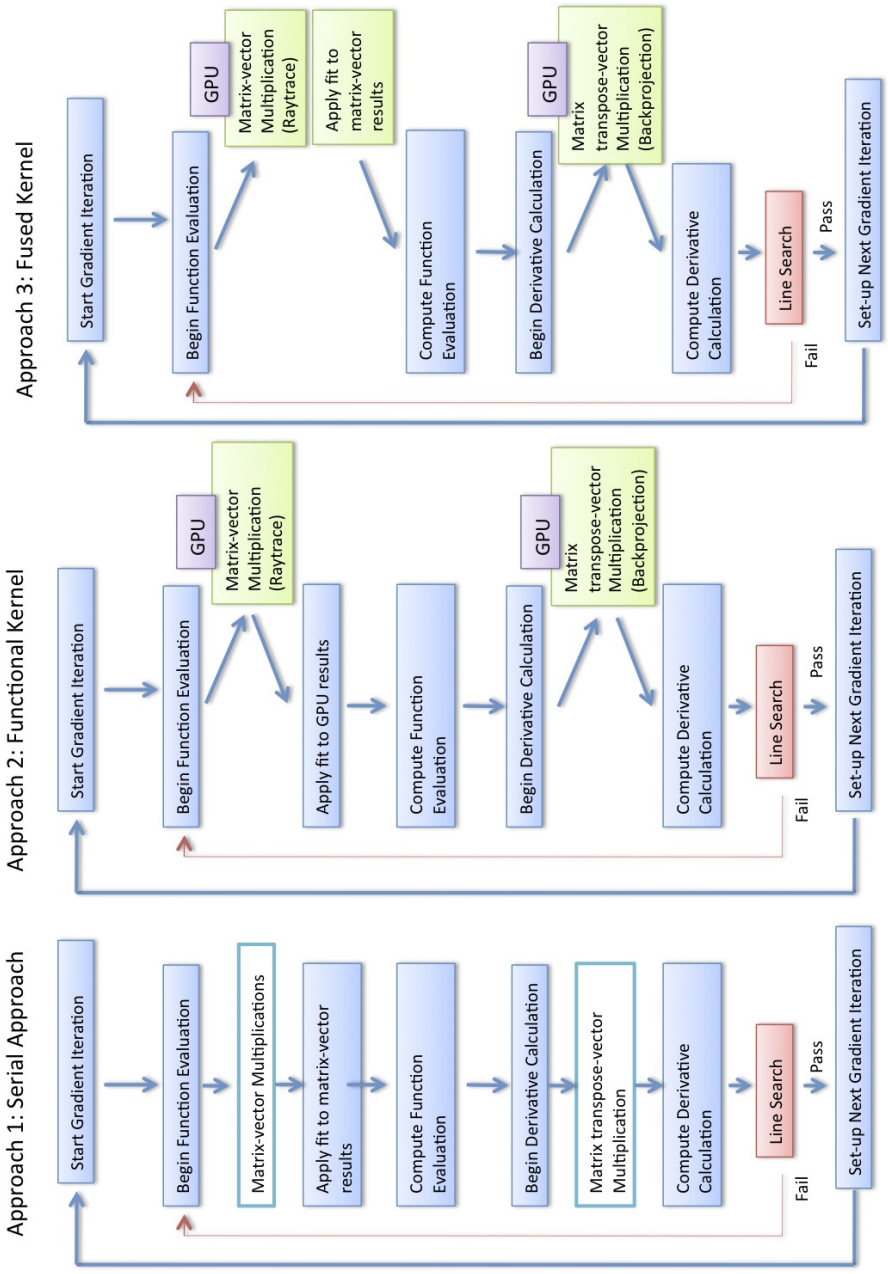


FIG. 3.1. *Algorithm Reconstruction Schemes*

Future work on the subject includes finishing a CUDA [9] implementation of the application for additional portability and further exploration of the mathematical model for better image quality results.

6. References.

[1] J. Chung, J. Nagy, and I. Sechopoulos. *Numerical algorithms for polyener-*

Size	Serial Product		GPU Kernel Product	
	$A_\theta\ \mathbf{u}$	$A_\theta\ [\mathbf{u}\ \mathbf{v}\ \mathbf{w}]$	$A_\theta\ \mathbf{u}$	$A_\theta\ [\mathbf{u}\ \mathbf{v}\ \mathbf{w}]$
810x1608x15 Pixels 400x1600x39 Voxels	85.4 sec	110.1 sec	4.6 sec	8.3 sec
	$A_\theta^T\ \mathbf{x}$	$A_\theta^T\ [\mathbf{x}\ \mathbf{y}]$	$A_\theta^T\ \mathbf{x}$	$A_\theta^T\ [\mathbf{x}\ \mathbf{y}]$
810x1608x15 Pixels 400x1600x39 Voxels	35.16 sec	38.93 sec	2.16 sec	2.90 sec

FIG. 4.1. *Matrix-vector and Matrix transpose-vector Product Computation Time*

Operation:	GPU Time	CPU Time
One Grad iteration:	12 sec	2.99 min
Ray Trace: $A_\theta\ [\mathbf{u}\ \mathbf{v}\ \mathbf{w}]$	1.8 sec	19.2 sec
Ray trace plus Function Evaluation Loop: (triple nested loop)	2.3 sec	37.2 sec
Back projection: $A_\theta^T\ [\mathbf{x}\ \mathbf{y}]$	2.8 sec	23.8 sec

FIG. 4.2. *Timings for Each Computationally Intensive Portion of Reconstruction Iteration. The size of the problem is* $360 \times 648 \times 15$ *pixels reconstructing* $467 \times 648 \times 39$ *voxels. GPU iteration timings using the Fused Kernel approach.*

Size	Serial	Functional Kernel	Fused Kernel
400x701x15 Pixels 400x701x50 Voxels	8.07 min	2.77 min	0.43 min
600x801x15 Pixels 600x801x50 Voxels	16.12 min	5.62 min	0.94 min
1280x2048x15 Pixels 1000x1748x50 Voxels	4+ hours	2.5+ hours	1.11 min

FIG. 4.3. *Timings for a Full Reconstruction of a Clinical Data Set.*

getic digital breast tomosynthesis reconstruction. SIAM J. Imaging Sci., 3(1): 133-152, 2010.

[2] J. T. Dobbins III and D. J. Godfrey, *Digital x-ray tomosynthesis: current state of the art and clinical potential*, Phys. Med. Biol., 48 (2003), pp. R65-R106.

[3] D. G. Grant, *Tomosynthesis: A three-dimensional radiographic imaging technique*, IEEE Trans. Biomed. Eng., 19 (1972), pp. 20-28.

[4] G. R. Hammerstein, D. W. Miller, D. R. White, M. E. Masterson, H. Q. Woodard, and J. S. Laughlin, *Absorbed radiation dose in mammography*, Radiology, 130 (1979), pp. 485-491.

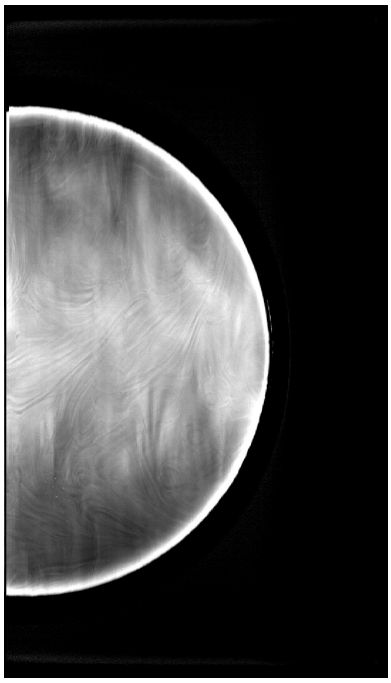


FIG. 4.4. *Slice 27/50 of Reconstruction with Phantom Data Set*

[5] A. Karellas, J. Lo, and C. Orton, *Point/counterpoint: Cone beam x-ray CT will be superior to digital x-ray tomosynthesis in imaging the breast and delineating cancer*, Med. Phys., 35 (2008), pp. 409-411.
[6] C. T. Kelley, *Iterative Methods for Optimization*, SIAM, Philadelphia, 1999.
[7] M. de Greef, J. Creeze, J.C. van Eijk, R. Pool, and A. Bel. *Accelerated Ray Tracing for Radiotherapy Dose Calculations on a GPU.* Med. Phys 36 (9) September 2009.
[8] J. Nocedal and S. Wright, *Numerical Optimization*, Springer, New York, 1999.
[9] NVIDIA CUDA Compute Unified Device Architecture Programming Guide, NVIDIA Corporation, 4.0 edition 2011
[10] OpenCL (Open Computing Language), Khronos Group, version 1.1. http://www.khronos.org/opencl/
[11] I. Sechopoulos, *Investigation of physical processes in digital x-ray tomosynthesis imaging of the breast*, PhD thesis, Georgia Institute of Technology, 2007.
[12] I. Sechopoulos and C. Ghetti, *Optimization of the acquisition geometry in digital tomosynthesis of the breast*, Med. Phys., 36 (2009), pp. 1199-1207.
[13] R. Siddon, *Fast calculation of the exact radiological path for a three dimensional CT array.* Med. Phys. 12: 252-255, 1985.
[14] Y. Zhang, H. Chan, B. Sahiner, J. Wei, M. M. Goodsitt, L. M. Hadjiiski, J. Ge, and C. Zhou, *A comparative study of limited-angle cone-beam reconstruction methods for breast tomosynthesis,* Med. Phys., 33 (2006), pp. 3781-3795.