University of California at Los Angeles

Electrical Engineering

C and CUDA Implementation for SIRT and SART Reconstruction Algorithms

Comprehensive Project Report for Master Degree

Non-thesis Option

by

Shan He

Advisor: Prof. Markovic, Dejan

Winter 2014

Abstract

Tomographic reconstruction techniques deserve studying because of its plenty of application in interdisciplinary fields. With outstanding features of no need to set of uniformly distributed projections for precise reconstruction, easy provide a priori knowledge about the reconstructed object, good image quality, we chose Simultaneous Iterative Reconstruction Technique (SIRT) and Simultaneous Algebraic Reconstruction Technique (SART) as our project research subjects. The main work of the project is to implement the SIRT and SART algorithms in C and CUDA respectively, aiming at improve the performance of iterative calculation. By utilizing the Sparse Level 2 subroutines in Inter MKL for C implementation and CUSPARSE library supported by CUDA SDK for GPU implementation, the performance is largely improved compared with original MATLAB code. Moreover, the times of speedup increases when increase the image size and number of projection angles. This phenomenon is obvious for GPU implementation, because larger data size can make more use of GPU. The maximum image size of C and CUDA implementation for both SIRT and SART can be 256*256 and the maximum number of projection angles of C and CUDA implementation for both SIRT and SART can be 179. For C implementation, SIRT slightly faster than MATLAB and SART is largely faster than MATLAB. For CUDA, loop time test matter. SIRT and SART are both much faster than C in the iterative

Keyword: SIRT, SART, C, CUDA,

CONTENT

1 Introduction

1.1 Tomography Problem

Tomography is a cross-sectional imaging technique in which an object is illuminated from many angles using any kind of penetrating wave. In common is the principle of reconstructing an object of interest from measurements or projections of the object, typically obtained bypassing a series of electromagnetic rays through the object. For example, Conventional X-ray tomography uses an X-ray source and a detector to measure the intensity of the X-ray beam after it has been passed through the object. This gives rise to a set of projections of the object and the problem is then how to put the information together to obtain a reconstruction of the object. This is the tomography problem referring to reconstruct an object from its projection and this is the subject of the project.

1.2 The overview of the report

The chapters of the report are organized in the following way:

Chapter 2: Introducing the basic reconstruction algorithm especially for algebraic reconstruction methods. Pointing out the meaningless of algebraic reconstruction methods and formulating it as linear inverse problem. And then describe the iterative techniques to solve this problem which are ART, SIRT and SART and compare their advantages and disadvantages.

Chapter 3: Describe the storage format for sparse matrix which is used in spare matrix-vector multiplication for solving the linear SIRT and SART algorithm. Introduce the GPU and its architecture difference with CPU as well as its CUDA programming model which serve as background for CUDA implementation

Chapter 4: Focus on the workflow and essential code structure of C and CUDA implementation and verification

Chapter 5: Present, compare and discuss the results from MATLAB code, C and CUDA implementation.

1.3 The Goal of the Project

The project is to implement SIRT and SART in C and CUDA respectively, aiming at accelerating calculation compared with MATLAB. The C implementation is expected to be faster than MATLAB code and CUDA implementation should be faster than C code.

2 Iterative Reconstruction Methods

2.1 Introduction

There are two main categories for reconstruction algorithm [1]:

(1) Analytical Reconstruction Methods: mainly includes back projection and filtered back projection, based on the Fourier Slice Theorem

(2) Conventional Algebraic Reconstruction Methods: search for solving the reconstruction problem by solving simultaneous linear equations system

For analytic reconstruction methods, only if all the (infinitely many) line integrals through the object are now, the object can be reconstructed perfectly which was proved by Radon [2]. In real life, we can't obtain an infinite number of measures and also we are limited to projection angles, so the image quality of analytic reconstruction methods is much poorer.

Although, the computation cost of algebraic reconstruction methods is higher than analytic reconstruction methods, algebraic reconstruction methods still stand out because of no need to complete set of uniformly distributed projections for precise reconstruction, easy provide a priori knowledge about the reconstructed object, good image quality, less requirement projections, easy implementation of different rays, etc. Therefore, we consider to solve the reconstruction problem by algebraic methods[3]. FPGAs also good for algebraic methods [4] [5].

For algebraic reconstruction method, the tomographic problem is viewed as a linear system, i.e. $Ax = b$ where A is an M by N^2 matrix representing the data acquisition process , x is a column vector of length N^2 representing the pixels of the original image and b is a column vector of length M representing the measured projection data. Our goal is to find a solution to $x = A_{\text{left}}^{-1}b$. Because M and N are usually very large, it is not practical to use the matrix-inversion based method to solve the problem. So an iterative

method is more appropriate to solve the problem to get the reconstructed 3D object. The iterative methods based on Algebraic Reconstruction Technique (ART), Simultaneous Iterative Reconstruction Technique (SIRT) and Simultaneous Algebraic Reconstruction Technique (SART) which model the problem using the same system equation but use different iterative method. I first derive the system of linear equations and then describe ART, SIRT and SART in detail.

2.2 Formulate as a Linear Inverse Problem

We have a projection set of an object taken from different projection angles on different detectors. The approximation of x can be got by sending rays and measuring the projection data. The object is considered to be in the d-dimension box: $\Omega \in [0, N]_d$, d can be 2 or 3. The process of obtaining the reconstructed image can be viewed as a function f: $\Omega \to \mathcal{R}$. Along each dimension the domain is split into N parts of unit length therefore N^d square subregions are generated, known as pixels for 2D, and voxels for 3D. Here for simplicity, we only consider 2D, so the discredited image is denoted by $X \in \mathcal{R}^{N*N}$ and by $x = \text{vector}(X) \in \mathcal{R}^{N^2*1}$ where x is obtained by stacking the columns of X. The data b is the collection of all the projection data. If in total M rays are used, then b will be of length M, viewed as one entry for each array. [6]

Now we need to consider how to set up Matrix A. Now consider the i^{th} ray, which we assume a perfect straight line with no width. On its way through the ray passes through a number of pixels, and the length of the i^{th} ray through j^{th} pixel is called a_{ij}. If the ray does not pass through a given pixel, $a_{ij} = 0$. In each pixel the value of f is assumed as constant, so the contribution to the i^{th} line integral from the j^{th} pixel is the path length multiplies by the pixel value, i.e.$a_{ij}x_j$. By summing over over all the pixels along the i^{th} ray we obtain the total projection data.[6] $\sum_{j=1}^{N^2} a_{ij} * x_j = b_i$, i=1,2……M, where M is the total number of rays, which equals total number of detectors times total number of projection angles We can write the discrete equation above in an extended form in Equation 2.1

$$a_{11}x_1 + a_{12}x_2 + a_{13}x_3 + \cdots + a_{1N2}x_{N2} = b1$$

$$a_{21}x_1 + a_{22}x_2 + a_{13}x_3 + \cdots + a_{2N2}x_{N2} = b2$$

…….

$$a_{M1}x_1 + a_{M2}x_2 + a_{M3}x_3 + \cdots + a_{MN2}x_{N2} = bM$$

In real life, matrix A is usually a sparse matrix meaning that the matrix is sparsely filled with nonzero data. Because large amount values are zeros, many storage formats for not accessing this zero values are derived for accelerating the calculation which will be described in Chapter 3.

2.3 Iterative Techniques

2.3.1 Derivation

Just as stated in introduction section, we use iterative techniques to solve this linear equation. So for computer implementation of this method, initial guess of $x_j^{(0)}$ is made, i.e $x_0^{(0)}, x_1^{(0)} \ldots\ldots x_{N^2}^{(0)}$. In most cases, we simply assign a value of zeros to all the i^{th}. The initial value is projected on the hyper plane represented by the first equation in (2.2) generating $x_j^{(1)}$. Similarly, $x_j^{(1)}$ is projected on the hyperplane represented by the second equation in (2.2) giving $x_j^{(2)}$. When $x_j^{(i-1)}$ is projected on the hyperplane represented by the j^{th} equation in (2.2), the $x_j^{(i)}$ is yielded. The process can be mathematically described by $\overrightarrow{x^I} = \overrightarrow{x^{I-1}} - \frac{\overrightarrow{x^{(I-1)}} * \overrightarrow{a_I} - \overrightarrow{b_I}}{\overrightarrow{a_I} * \overrightarrow{a_I}} * \overrightarrow{a_I}$. Assume k is the iteration number, after k^{th} iteration, the $\overrightarrow{x^{(k+1)}}$ can be obtained by the following equation:

$$x^{(k+1)} = x^{(k)} + \lambda A^T(b - Ax^k) \text{ -----Equation 2.2}$$

where λ is scalar for relaxation factor.

Here we introduce two diagonal matrices W and W for normalization:

V is the diagonal matrix of the inverse of the row sums:

$$V_i = V_{i,i} = \frac{1}{\sum_{j=1}^{N^2} a_{i,j}}$$

W is the diagonal matrix of the inverse of the column sums:

$$W_i = W_{i,i} = \frac{1}{\sum_{i=1}^{M} a_{i,j}}$$

So the equation (2.2) is transformed into:

$$x^{(k+1)} = x^{(k)} + \lambda V A^T W(Ax^{(k)} - b) \text{ ---Equation 2.3}$$

2.3.2 ART, SIRT and SART

The difference among ART, SIRT, and SART is their different iteration methods, that is when they update the value of j^{th} cell (i.e. x_j).

For ART, updating x_j after every equation is finished in Equation 2.2 for calculating. For SIRT, updating the value of cell after all rays are processed, which means before making any changes, we go through all the equations in Equation 2.2, and after that the cell values are updated. The change for each cell is the average value of all the computed changes for that cell. This constitutes an iteration of the algorithm. In the second iteration, we go back to the first equation in Equation 2.2 and the process is repeated. For SART, after we are through all the rays in a projection angel, we add the correction array to the image array. Their pseducode is shown in Fig2.1

ART

for k=1:k
  for i=1:M
    $x^{(i+1)} = x^{(i)} - \lambda V_i A_i^T W_i (A_i x^{(i)} - b_i)$
  end
end

SIRT

for k=1:k
  $x^{(k+1)} = x^{(k)} - \lambda V A^T W (Ax^{(k)} - b)$
end

SART

for k=1:k
  for θ=1: Θ
    $x^{(\theta+1)} = x^{(\theta)} + \lambda V_\theta A_\theta^T W_\theta (b_\theta - A_\theta x^{(\theta)})$
  end
end

Fig 2.1 The pesudocode for ART, SIRT and SART

2.3.4 Pros and Cons among ART, SIRT, SART

ART reconstructions usually suffer from salt and pepper noise. The effect of inconsistencies is exacerbated by the fact that as each equation corresponding to a ray in a projection is taken up, it changes some of the pixels just altered by the preceding equation in the same projection. The SIRT algorithm also suffers from these inconsistencies, but by eliminating the continual and competing pixel update as each new equation is taken up, it results in smoother reconstructions, though it has slower convergence. To further reduce the noise resulting from the unavoidable but now presumably considerably smaller inconsistencies with real projection data, the correction

terms are simultaneously applied for all the rays in one projection; this is in contrast with the ray-by-ray updates in ART.[7]

All in all, while ART converges fast but produces very noisy results. But at the expense of slower convergence usually leads to better looking images than those produced by ART. SART combine fast convergence of ART with good results obtained with SIRT.

So in order to get better reconstruction image quality, we chose SIRT and SART as our reconstruction algorithm which are implemented in C and CUDA respectively.

3 Preparation Knowledge for Implementation

The most time consuming part for SIRT and SART algorithm is the matrix-vector multiplication in Equation2.3. The goal of our project is to speed up calculating this equation by implementing it in C and CUDA respectively. Also as described in Chapter 2, the matrix A , A^T, diagonal W and V are all sparse matrix, so we can call the subroutines handling in sparse matrix-vector multiplication from MKL for C implementation and CUSPARSE Library for CUDA implementation to help acceleration. So in this chapter, firstly, I gave a brief description of matrix compressed format for sparse matrices and then introduce the architecture difference between CPU and GPU as well as CUDA programming.

3.1 Storage Format for Sparse Matrix

Sparse matrix means that the matrix is sparsely filled with nonzero data. Because large amount values are zeros, there is no need to take time to access these zeros. So, a new format could be utilized to only represent the nonzero values, the row and columns index showing where these nonzero values are in order to save time and memory space. The three basic sparse matrices are the Compressed Row Schemes (CRS), Compressed Column Schemes (CCS), and Coordinate Storage Schemes (CSS). Following example based on zero-indexing can easily explain the concept shown below:[8]

The matrix in Table 3.1 above would have the following representations for these three schemes based on zero-indexing.

Table 3.1 Sparse Matrix

10	0	4	-1	0	0
-4	9	0	0	-1	0
0	8	0	3	0	-1
1	0	-3	0	0	7
0	1	0	0	6	2
0	0	1	0	-2	5

(1) Compressed Row Scheme

Val(i) = (10,4,-1,-4,9,-1,8,3,-1,1,-3,7,1,6,2,1,-2,5)

Col(i) = (0,2,3,0,1,4,1,3,5,0,2,5,1,4,5,2,4,5)

Rowptr = (0,3,6,9,12,15,18)

(2) Compressed Column Scheme

Val(i) = (10,-4,1,9,8,1,4,-3,1,-1,3,-1,6,-2,-1,7,2,5)

Row(i) = (0,1,3,1,2,4,0,3,5,0,2,1,4,5,2,3,4,5)

Colptr(i) = (0,3,6,9,11,14,18)

(3) Coordinate Storage Scheme

Val(i) = (10,4,-1,-4,9,-1,8,3,-1,1,-3,7,1,6,2,1,-2,5)

Row(i) = (0,0,0,1,1,1,2,2,2,3,3,3,4,4,4,5,5,5)

Col(i) = (0,2,3,0,1,4,1,3,5,0,2,5,2,4,5,2,4,5)

In our project, I use CRS format for both C and CUDA implementation because it is the most popular one lending itself well to coding and memory access and its pointer index can be used to index large array directly which is very time-saving.[] For CRS format, Val(i) refers to an array that stores the nonzero value of sparse matrix. It has the length of number of nonzero values. Col(i) is an integer array indicates the column where the nonzero value in spare matrix exists, which also has the length of number of nonzero values. Rowptr is also an integer array indexing for first nonzero values in a row and the last element in this array equals the number of nonzero values plus one. So the number of elements in rowptr array is the number of rows of sparse matrix plus one.

3.2 GPU architecture

3.2.1 Overview of GPU architecture

Graphics Processing Units (GPUs) are highly parallel computing device designed for the task of graphics rendering. Moreover, the GPU has evolved to become a more general processor, allowing users to flexibly program certain aspects of the GPU to facilitate sophisticated graphics effects and even scientific applications. All of these make GPU become a powerful device for the execution of data-parallel, arithmetic intensive applications in which the same operations are carried out on many elements of data in parallel. Fig3.1 shows an overview of the GPU architecture.[9]

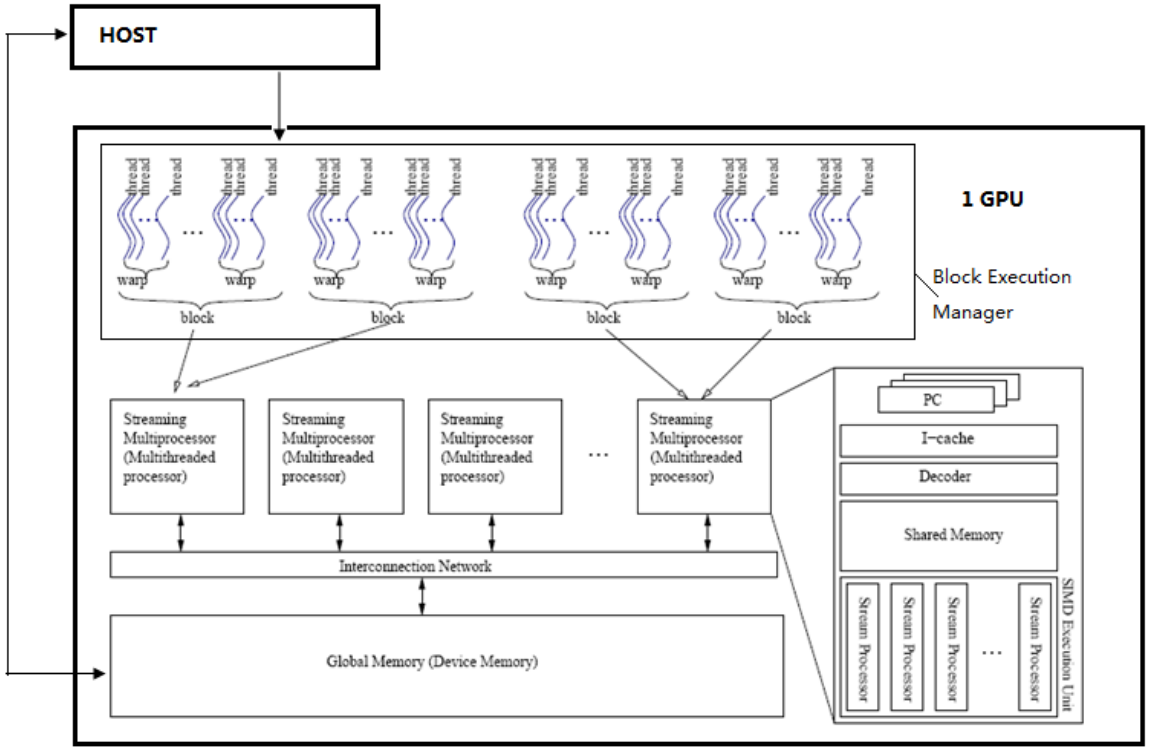

Fig 3.1 The overview of GPU architecture

The GPU architecture consists of a scalable number of streaming multiprocessors (SMs), each containing eight streaming processor (SP) cores, two special function units (SFUs), a multithreaded instruction fetch and issue unit, a read-only constant cache, and a read/write shared memory. The SM executes a batch of 32 threads together called a warp. Executing a warp instruction applies the instruction to 32 threads. All the threads in one block are executed on one SM together. One SM can also have multiple concurrently

running blocks. The shared memory is implemented within each SM multiprocessor as an SRAM and the global memory is part of the off-chip DRAM. The shared memory has very low access latency and high bandwidth. All multiprocessors have access to global device memory, which is not cached by the hardware. Memory latency is hidden by executing thousands of threads concurrently. Register and shared memory resources are partitioned among the currently executing threads.

3.2.2 The major architecture difference between GPU and CPU

The most essential design goal of a single core CPU and GPU is that the design goal of a CPU is to execute one stream of instructions as fast as possible, however on the other hand, design goal of a GPU is to execute many parallel streams of instructions as fast as possible, that is to optimize for the execution of massive number of threads.

Moreover, there all many other ways to look at their different architecture:[10]

(1) Transistor usage: for CPU, the transistor usage such as instruction reorder buffers, reservation stations, branch prediction hardware, and large on-die cache aims at speeding up the execution of a single thread. However, speed the GPU spends transistors in process arrays, multithreading hardware, shared memory, and multiple memory controllers. These features are not fixated on speeding up the execution of a particular thread, rather they allow the chip to support tens of thousands of threads concurrently on-chip, facilitating thread communication, and sustained high memory bandwidth.

(2) Role of cache: The CPU uses cache to improve performance by reducing the latency of memory accesses. The GPU uses cache (or software-managed shared memory) to amplify bandwidth so that multiple threads that access the same memory data do not need to all go to DRAM.

(3) Managing Latency: The CPU handles memory latency by using large caches and branch prediction hardware. These take up a large deal of die-space and are often power hungry. The GPU handles latency by supporting thousands of threads in flight at once. If a particular thread is waiting for a load from memory, the GPU can switch to another thread with no delay.

(4) Multithreading: CPUs support one or two threads per core. CUDA capable GPUs support up to 1,024 threads per streaming multiprocessor. The cost of a CPU thread switch is hundreds of cycles. GPUs have no cost in switching threads. GPUs typically switch threads every clock.

(5) Memory Controller: Intel CPUs have no on-die memory controllers. CUDA capable GPUs employ up to eight on-die memory controllers. As a result, GPUs typically have 10× the memory bandwidth of CPUs.

3.2.3 CUDA Programming Model

CUDA (Compute Unified Device Architecture) is a parallel computing platform and programming model, which has far been one of the most successful languages to create a GPU that would excel at computation in addition to performing well at traditional graphics tasks.

The GPU is a compute device, serving as a coprocessor for the host CPU, that executes data-parallel kernel functions. A kernel is actually a function called from the host CPU that runs on the GPU device which can execute in parallel across a set of parallel threads[9]. CUDA provides three key abstractions, a hierarchy of thread groups, shared memories, and barrier synchronization. Threads have a three level hierarchy. A grid is a set of thread blocks that execute a kernel function. Each grid consists of blocks of threads. Each block is composed of hundreds of threads, which is shown in Fig3.1. All threads within a block are executed concurrently on a multithreaded architecture.

When invoking a kernel, the programmer specifies the number of threads per block, and the number of blocks per grid. Each thread is given a unique thread ID number threadIdx within its thread block, for example, for 1D, it is numbered 0,1,2,…..blockDim-1 for 1D(It can be 1D,2D and 3D). And each thread is given a unique block ID number blockIdx within its grid. The 1D grid size and block size are shown in Fig 3.2.[11]

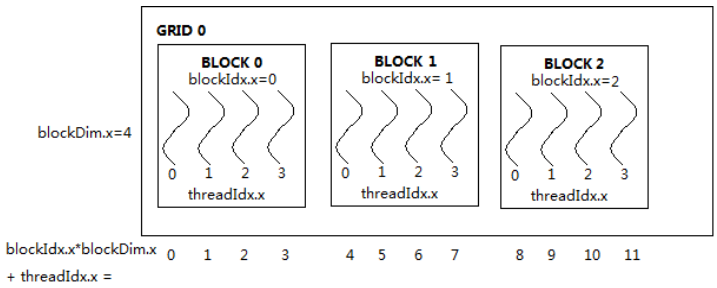

Fig 3.2 1D grid size and block size index example

In the CUDA parallel programming model, each thread has a per-thread private memory space used for register spills, function calls, and C automatic array variables. Each thread block has a per-Block shared memory space used for inter-thread communication, data sharing, and result sharing in parallel algorithms. Grids of thread blocks share results in Global Memory space after kernel-wide global synchronization. The host can read and write global memory but not shared memory.

4 C and CUDA implementation

Our job is to implement SIRT and SART algorithm in C and CUDA respectively. To be more specific, this project is to replace the MATLAB code for calculating$x^{(k+1)} = x^k + \lambda * V * A^T * W * (b - A * x^k)$ with my C and CUDA code integrated into MEX-files and call them in original MATLAB file for SIRT and SART for verification. This chapter mainly describes the workflow of implementation as well as the verification and code structure of implementation.

4.1 Working Environment

All the coding, compiling and testing is on Kodiak server where has an Intel Core i7-2600 Quad-Core HT 3.4GHz processor and NVIDIA GeForce GTX 660 graphics card for CUDA implementations.

4.2 Workflow of implementation

4.2.1 Essence of Implementations via MEX-Files

MEX stands for “MATLAB executable” which means that we can call our own C subroutines from the MATLAB command lines as if they were built-in functions. So essentially, our C or CUDA implementations is to write MEX-Files consisted of our own

C or CUDA code as well as gateway function to connect MATLAB and C/CUDA code, and use them as MATLAB built-in functions after compiling for our implementation testing and demonstration. Next session I will expand the workflow of implementation by writing MEX-Files, compiling and testing.

4.2.2 Workflow

The detailed descriptions of the steps stated in previous session are:

(1) Create source MEX-file which consists of the realization $x^{(k+1)} = x^k + \lambda * V * A^T * W * (b - A * x^k)$ in C or in CUDA code and the mexFunction module serving as gateway function to provide interface for C/CUDA and MATLAB input and output port.

(2) Compile the source MEX-file to generate binary mexfile. In a typical MATLAB workflow, MEX file compilation is handled by the MATLAB command mex, which compiles and links the C/MEX file.

(3) Test the result: Call my C/MEX and CUDA/MEX files in "sirt.m "and "sart.m" file, (E.G out_xk= sirt_cpu(A,b,x0,K,nonneng,A')) and modify the original "sirt.m" and "sart.m" and rename them as "sirt_cpu.m", "sart_cpu.m" for C implementation, "sirt_gpu.m" ,"sirt_gpu.m" for CUDA implementation. Also modify "example.m" files correspondingly and finally run it via MATLAB command line and the time of MATLAB, C and CUDA implementation and the correlation coefficient, root mean square error and signal to noise ration shows up. Fig 4.1 shows the steps described above.

For CUDA implementation, step (1) and step (3) are almost the same. The only difference lies in step (2). Because CUDA source file consists of host code and device code, after compiling, it generates both the CPU binary code as "filename.mexa64" file and GPU binary code as "filename.o file". Because CUDA source files are given the .cu file extension, as of R2009b, MATLAB does not support the .cu file extension or the CUDA compiler nvcc. To prepare a file for use in MATLAB, do the following: 1. Shell to the NVIDIA compiler to create the host object file. 2. Invoke mex with the object file, not the source file, as the parameter. A simple MATLAB program can manage both tasks.

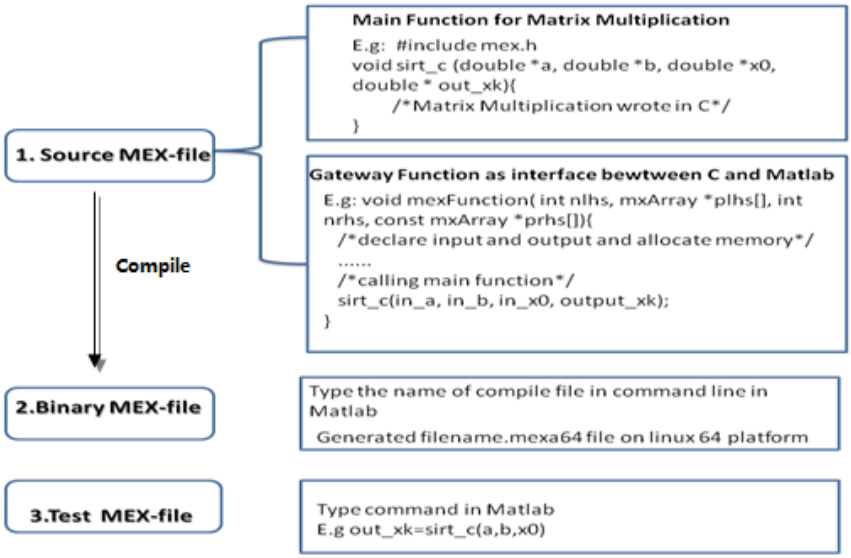

Fig 4.1 workflow of implementation and verify in Matlab

4.3 C Implementation

The following focuses on how to build MEX-Files, i.e step (1) stated above, which is the core of implementation.

4.3.1 MKL library

The Intel® Math Kernel Library (Intel® MKL) improves performance with math routines for software applications that solve large computational problems. Moreover, as stated before, the core calculation is to solve Xk iteratively where most the time consuming part is matrix-vector multiplication. But this multiplication is sparse matrix multiply dense vector which can be efficiently solved by subroutine in Sparse BLAS Level 2 and Level 3 in MKL. Moreover, Intel MKL is optimized for certain Intel processors including the Core i7-2600 Quad-Core HT 3.4GHz installed on our server. All in all, MKL is an efficient option for our C implementation

4.3.2 C code structure for SIRT

The MATLAB code for calculating $x^{(k+1)} = x^k + \lambda * V * A^T * W * (b - A * x^k)$ is replaced with my C code. The report only shows the basic idea and structure of code, the code structure for SIRT is shown in Fig 4.2

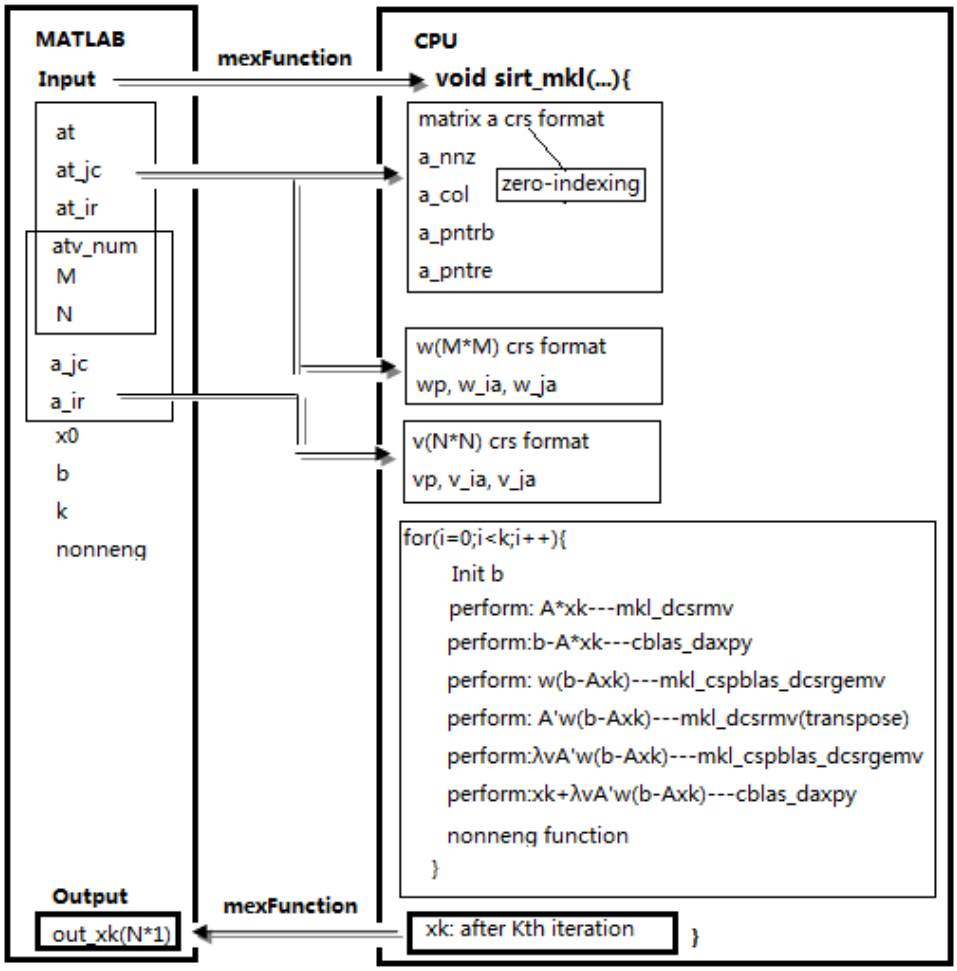

Fig 4.2 C code structure for SIRT

MexFunction, serving as gateway for C code and Matlab code, provides input at, at_jc, at_ir, atv_num, a_jc, a_ir, x0, b, k, nonneng for main function sirt_mkl. The way to get t hem is shown in Table 4.1

Table 4.1 The realization of input parameters in mexFunction for main function in C

a /at	mxGetPr	Pointers to the first elements of an array containing nonzero value for sparse matrix A and A’
x0/b	mxGetPr	Pointers to the first elements of an array

		containing the value for vector x0/b
k/nonneng	mxGetScalar	Get the real value
m_row/M	mxGetM	Get the number of rows of matrix A
n_col/N	mxGetN	Get the number of columns of matrix A
at/a_jc	mxGetJc	Pointer to the first element of an array containing the index for first nonzero element in the jth column of matrix A/A'
at/a_ir	mxGetIr	Pointer to the first element of an array containing the indicates a row at which a nonzero element can be found in matrix A or A'

Because matrix A and A' are read in sparse format from Matlab, input "a" and "at" is an array containing the number of nonzero values of matrix A and transpose matrix of A. "atv_num" is the number of nonzero values in matrix A which can be obtained by mxGetNzmax in mexFunction. "at/a_jc[i]" is an array that contains the index of the first nonzero value in the i^{th} column of matrix A' and the last element in jc is the number of nonzero values plus one. So its length is the number of columns plus one. "at/a_ir[i]" is an array that indicates a row at which a nonzero element can be found in matrix A'. So the length is the number of nonzero values. According to the description above, through array "at , "at_jc" and "at_ir", we can easily get the CSC format for matrix A'. Because CRS format is identical to the CSC format for the transposed matrix, getting the CSC format for matrix A' equals getting the CRS format for matrix A. So through input "at", "at_jc", "at_ir" which are obtained from transposed matrix of a, just like the arrow in

Fig4.2. Like verse, through input “a”, “a_jc”, “a_ir”, we can easily get the CRS format for matrix A’.

The vector W and V are also generated in C code. If they are generated in Matlab code and serve as input for main function, it would be very slow due to time consuming part of diagonal matrix multiplication in Matlab. Through input “at”,” at_jc” and M, N the vector W which containing nonzero values for diagonal W (M*M)(M is the number of rows of matrix A and N is number of columns in matrix A) could be obtained. By “at_jc” we can get the number of nonzero values in a row in matrix A and by using “at_jc” as index to “at”, we can get the corresponding nonzero value and then add them together and then get reciprocal of the sum. In order to use Sparse BLAS level 2 routines in MKL to perform W*(b-Axk), we also need to get the CRS format for W, that is w_ia and w_ja. W_ia is the index for first nonzero values in a row and the last element in array ia is the number of nonzero values plus one, the same definition as “_jc” array. Here for w_ia, the length is N+1(N is the number of columns in matrix A). W_ja indicates a row at which a nonzero element can be found, the same definition as “?_ir” array. The length is N for diagonal W. V is generated similar as W. Due to V is the diagonal matrix of the inverse of column sums so the sum of each column can be obtained through a_jc and a. The v_ia, v_ja are almost the same as w_ia, w_ja just change the size from M for W to N for V.

The iteration numbers in SIRT (N_sirt) is got from Matlab code and serve as input K for main function. Non negative operation in Matlab code, referring to after each iteration assign zero to negative xk to make sure the xk used for next iteration is nonnegative, is also got from Matlab code serving as input nonneng for main function.

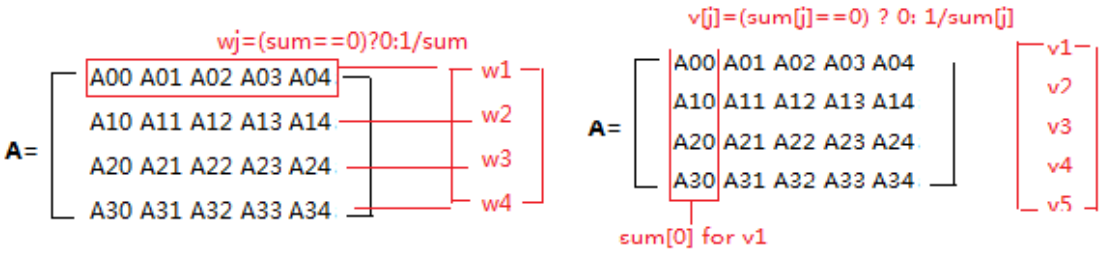

Fig 4.3 The flow of creating W and V

After all the initialization for matrix A, W and V CRS format, vector B and x0, calling sparse BLAS level 2 and BLAS level 1 function in MKL to calculate the $x^{(k+1)} = x^k + \lambda * V * A^T * W * (b - A * x^k)$ in loop just like drawn in Fig 4.2 . After K^{th} iteration, the N*1 vector xk is obtained and through mexFunction assign it to out_xk which is a N*1 mxArray allocated by Matlab.

4.3.2 C code structure for SART

The basic structure and idea behind the SRAT are almost the same as SIRT, the major difference is that the image is updated until we go through all rays in a projection angel. If the number of projection angles is N_theta, we go through number of D ($\frac{\text{The number of rows: M}}{\text{The number of projection angles:N\_theta}}$) equations in Equation 2.2 rather than M equations in SIRT algorithm, and then update the values in image cell and enter into next iteration for another projection angle. After N_theta iterations, i.e, going through all the equations in Equation 2.2 , the calculation enters into next iteration which performs all N_theta iterations to go through all the rays in all projection angles.

Essentially, I divide the input A, x0, B, W, V which are used to calculate $x^{(k+1)} = x^k + \lambda * V * A^T * W * (b - A * x^k)$ into N_theta and perform $x^{(\theta+1)} = x^{(\theta)} + \lambda V_\theta A_\theta^T W_\theta (b_\theta - A_\theta x^{(\theta)})$ in inner loop for N_theta iterations. So from code point of view, only change is that we deal with a smaller size of matrix A, V, W, A' and vector b in inner loop. So right now, for inner loop calculation, the A_θ is a D*N sparse matrix, b_θ is a D*1 vector, W_θ is a D*D diagonal matrix, V_θ a is N*N diagonal matrix and A_θ^T is transpose version of A_θ which is N*D.

In realization for this, I define two-dimension arrays for A_θ, W_θ, V_θ and their CRS format and input b_θ, respectively. So all of these 2D arrays has N_theta of rows, each row store the CRS format for A_θ, W_θ, V_θ and values in b_θ respectively. The subroutines in MKL called in inner loop are exactly the same as in SIRT algorithm, just replace A,W,V,b,A' for A_θ, W_θ, V_θ , A_θ'. A small example shown in Fig 4.3 could explain this. Defining a two dimension array a_theta[i][j], each a_theta[i](i referring to the N_theta iteration) store the nonzero values for i^{th} D*N sub matrix A. But because we read the sparse format of matrix A and A', so like SIRT we also need to use a/at_jc, a/at_ir to

obtain a_theta[i][j], wp_theta[i][j], vp_theta[i][j] and their CRS format. The basic code structure is in Fig 4.5 which is very similar to SIRT in Fig4.2. Next I only describe the different part.

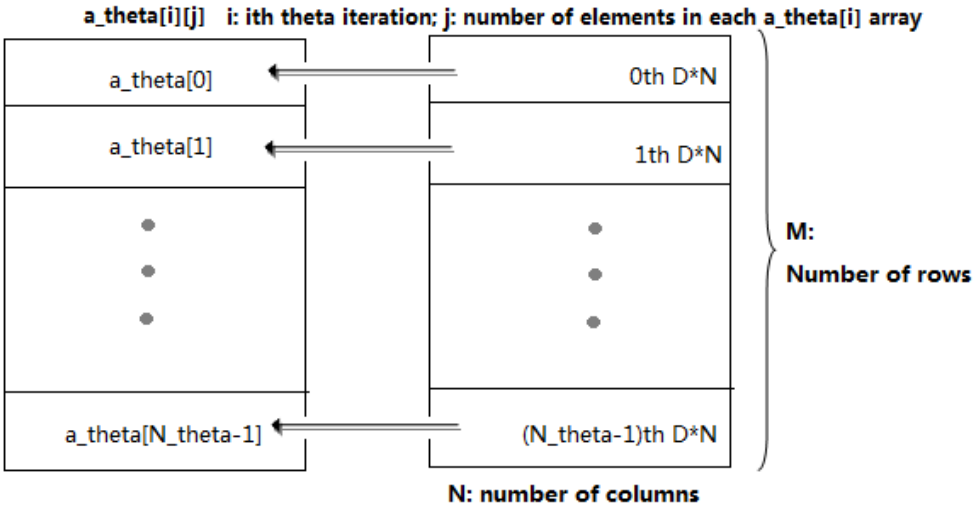

Fig 4.4 Example of defining 2D array to store a_theta[i][j]

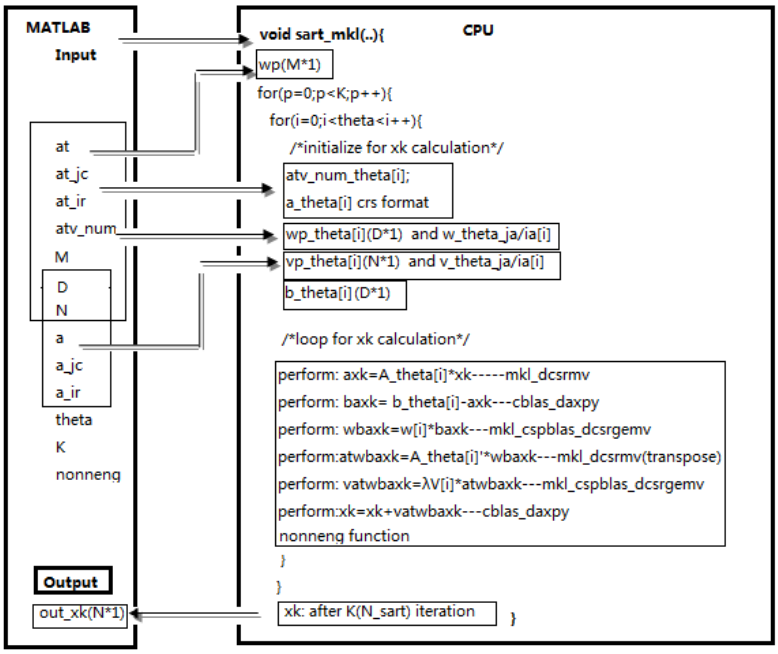

Fig 4.5 C code structure for SART

Theta is the number of projection angles, the same definition as N_theta in Matlab code. Atv_num_theta[i] is the number of nonzero values in i^{th} D*N matrix A which can be get by at_jc. Via atv_num_theta[i], at, at_ir, at_jc we can get number of i arrays

containing the nonzero values in i^{th} D*N sub matrix of M*N matrix A which are storing in a_theta[i][j], and j has the length of atv_num_theta[i]. a_theta_ja[i][j] is the index for which column the nonzero value is in i^{th} D*N sub matrix of matrix A. a_theta_pntrb[i][j] indicates the index of first element of nonzero value in each row in i^{th} D*N sub matrix of matrix A, a_theta_pntre[i][j] refers to the one plus the index of last element of nonzero value in each row in i^{th} D*N sub matrix of matrix A.

W is generated in the same way as in SIRT as M*1 wp, but is split into i^{th} (theta iteration times) D*1 wp_theta[i][j] in inner loop. However, another subroutine is coded to generate vp_theta[i][j] instead of using the way of generating V in SIRT. Because V is calculated from the sum of values in each column, the division of matrix A into D*N sub matrix changes the number of nonzero values in each column. By getting the index of which row the nonzero value is in each column in N*M matrix A' and by restricting to only take the row index in i^{th} N*D sub matrix of matrix A' and then by using this row index to index the corresponding nonzero value in N*M matrix A', we can get the nonzero value in each column in i^{th} D*N sub matrix of matrix A. After initializing all these variables needed to used in MKL, all the operation in inner loop is the same as SIRT just dealing with smaller size of input. Non negative is operated after every theta iteration. And similarly, after K^{th} iteration, the N*1 vector xk is obtained and through mexFunction assign it to out_xk which is a N*1 mxArray allocated by Matlab.

4.4 CUDA Implementation

CUDA implementations are very similar to C implementations just except for calculating $x^{(k+1)}$on GPU resulting in the need of memory copy from host to device and for calling another library supported by CUDA software for sparse matrix-vector multiplication.

4.4.1 Basic CUDA Code structure

CUDA programming is just an extension of C language with new syntax and built in variables and API/Library, which consists of host code and device code. Host code runs on CPU and device code runs on device. And all the management of the memory and execution of the devices are coded in host code.

The basic structure for CUDA is very similar to C implementation. The major difference is that due to kernels and CUSPARSE subroutines are operated on GPU, their input and output data are required to reside in GPU memory. So it is user's responsibility to allocate memory and to copy data between GPU memory and CPU memory by using standard CUDA runtime API routines, such as cudaMalloc(), cudaFree(), cudaMemcpy(), which should be done in host code.

4.4.2 CUSPARSE Library

The CUDA includes a GPU device driver, a runtime system that serves as an abstraction over the driver, and also runtime libraries that CUDA applications may link to in order to provide CUSPARSE support. CUSPARSE library is just a standard C library written by Nvidia to do sparse matrix linear algebra used for handling sparse matrices on the GPU and ideally it should be 32 faster than MKL. The level 2 functions in CUSPARSE library can efficient solve spare matrix-vector multiplication. And the CUSPARSE library allows developers to access the computational resources of the NVIDIA GPU.

4.4.3 CUDA Code Structure for SIRT

Fig 4.6 shows the structure for SIRT. Just like in C implementation, through mexFunction we can get input a, at, at_jc, at_ir, a_jc, a_ir, atv_num, M, N, x0, b, k, nonneng to main function in C. And through operation in main function, we can get the CRS format for matrix A and matrix A' on host. Here I mainly describe the difference. Because the CUSPARSE is operated on GPU and because I design to calculate W CRS format and V CRS format on GPU too, the"cudaMemcyHostToDevice" is needed to make the input required for GPU operation reside on device. So we need to copy the CRS format of matrix A and A', input x0, b from host to device. The calling of kernels to generate W and V CRS format and to operate non negative function as well as the calling of CUSPARSE library are coded in host code. After k^{th} iterations calculated on GPU, the output d_xk on device should be copied back from device to host and then send it to Matlab through mexFunction.

Unlike C implementation we use the transpose operation option to perform sparse matrix A' times vector, in CUDA implementation this transpose operation in

CUSPARSE library is very time consuming which is proved through our trial and errors. So instead of operating transpose matrix vector multiplication by CUSPARSE, we get the CRS format of matrix A' and do nontranspose sparse matrix-vector multiplication here which is way too much faster.

Besides memory copy, calling kernels or CUSPARSE, other things should also be included in host code for GPU operation: (1) setting the block size and grid size, after setting the block size, the grid size = (problem size+blocksize-1)/block size (2) Initialize CUSPARSE library and create and setup matrix descriptor (3) Use CUDA helper function for tracking the realization of cuda malloc, cuda memory copy and CUSPARSE function operations. All of these three things are the same through CUDA implementation and will not be stated in the following content.

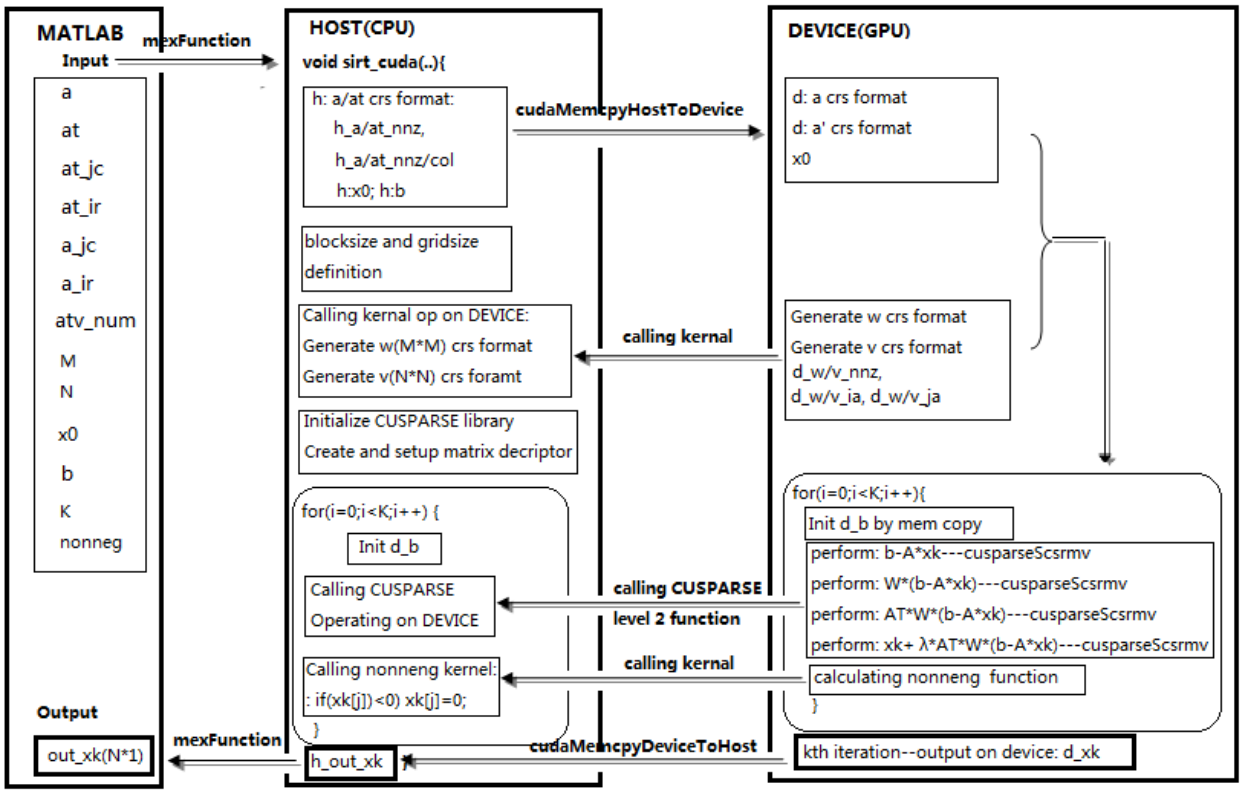

Fig 4.6 CUDA code structure for SIRT

4.4.4 CUDA Code Structure for SART

The major difference between SART CUDA implementation and SART C implementation is that in CUDA implementation, after initializing the a_theta[i][j] CRS

format, a_theta[i][j]' CRS format and input x0, b as well as wp_theta[i][j] and vp_theta[i][j] on host, they should also be copied to device to serve as input for kernel and CUSPARSE calculation which can be seen in Fig 4.7.

Just like C implementation in SIRT and SART, the major difference between SIRT and SART CUDA implementation is that we define two dimension array for a_theta, a_theta', w_theta, v_theta, b_theta, and initialized the their device variable by memory copy coded in host code theta loop.

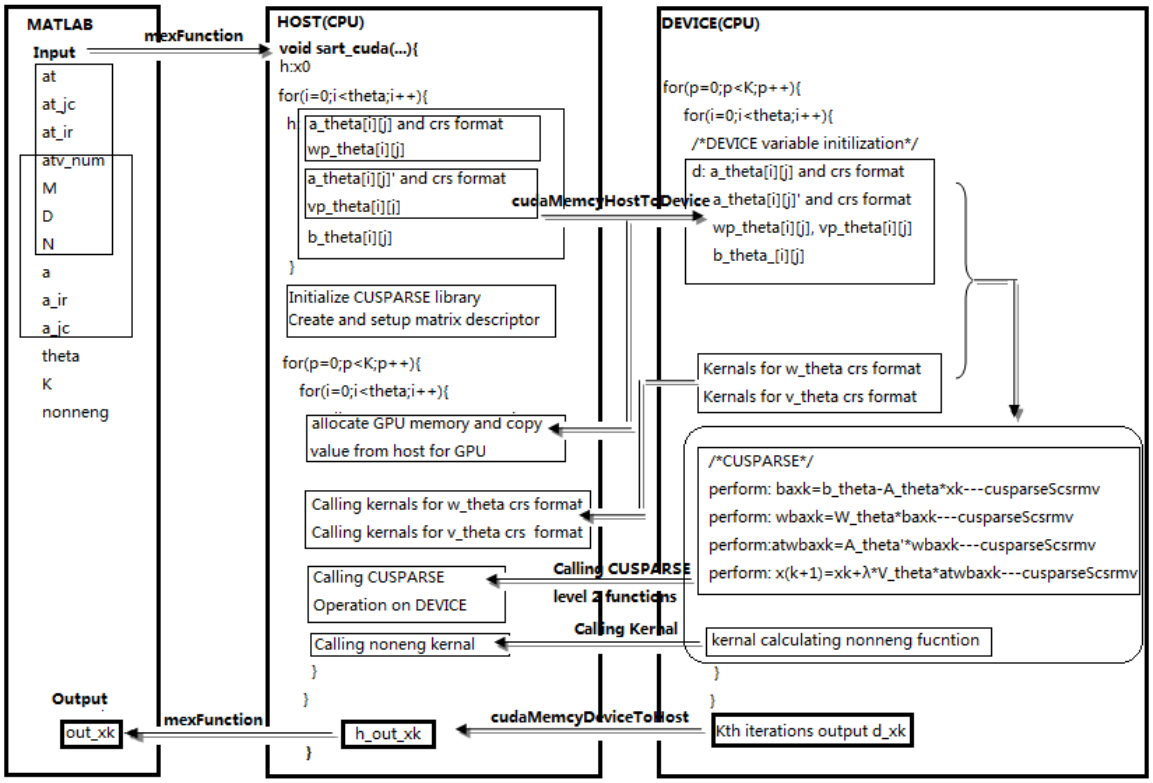

Fig 4.7 CUDA code structure for SART

5 Results and Discussion

5.1 Verification of C and CUDA implementation

Fig 5.1 shows the correlation coefficient, root mean square, signal to noise ration, and reconstruction time for Matlab, C and CUDA implementation when N=50 and N_theta=24. Firstly, it can be seen that the correlation coefficient, root mean square, signal to noise ratio for C and CUDA are the same with MATLAB result which verify my implementation. Secondly, for SIRT, the reconstruction time for MATLAB is

0.01406s, and for C is 0.01052s, which C is around 1.336s times faster. For SART, the reconstruction time for MATLAB is 0.09568s, and for C is 0.01498s, which C is 6.387s times faster. And speed will increase if we increase N and N_theta which will be shown latter. For CUDA implementation, it is obvious that the reconstruction time is slower than C and MATLAB, which is 0.73171s and 1.21614s respectively. This is because the data we deal with is relatively small and much of the time is spent on memory copy which has been proved through our tests. Considering the time we really matter is to calculate $x^{(k+1)} = x^k + \lambda * V * A^T * W * (b - A * x^k)$ in loop rather than prepare for these inputs, we tested the loop time for C and CUDA code respectively. It can be seen from Fig 5.1 that regarding loop time, the C and CUDA implementation for SIRT is 0.01s and 0.01s respectively. Although it is the same for this problem size, as we increase the problem size the CUDA implementation is faster than C in terms of loop time which will be shown next. This is because the GPU is not fully utilized for smaller problem size. For SART, the loop time for C and CUDA is 0.03s and 0.02s respectively, where CUDA is 1.5 times faster.

The method for testing loop time is described as follows. We include <time.h> in our code and put “begin=clock()” and “end=clock()” at the beginning and at the end of the loop respectively and code time_spent =(double)(end - begin) / CLOCKS_PER_SEC which returns to second as floating point. The specification is in code file. And then print out this time shown in the Fig 5.1

```
LOOP TIME: SIRT with C: 0.010000
LOOP TIME: SIRT with CUDA: 0.010000
LOOP TIME: SART with C: 0.030000
LOOP TIME: SART with CUDA: 0.020000
--------------------------
RECONSTRUCTED IMAGE QUALITY
--------------------------

SIMULTANEOUS ITERATIVE RECONSTRUCTION TECHNIQUE
  Number of Iterations:    30
  Correlation Coefficient: 0.92851
  Root Mean Square Error:  4.18289
  Signal to Noise Ratio:   9.38318
  Reconstruction Time:     0.01409

SIMULTANEOUS ITERATIVE RECONSTRUCTION TECHNIQUE BY C IMPLMENT
  Number of Iterations:    30
  Correlation Coefficient: 0.92851
  Root Mean Square Error:  4.18289
  Signal to Noise Ratio:   9.38318
  Reconstruction Time:     0.01052

SIMULTANEOUS ITERATIVE RECONSTRUCTION TECHNIQUE BY CUDA IMPLMENT
  Number of Iterations:    30
  Correlation Coefficient: 0.92851
  Root Mean Square Error:  4.18289
  Signal to Noise Ratio:   9.38318
  Reconstruction Time:     0.73171

SIMULTANEOUS ALGEBRAIC RECONSTRUCTION TECHNIQUE
  Number of Iterations:    2
  Correlation Coefficient: 0.93297
  Root Mean Square Error:  4.16636
  Signal to Noise Ratio:   9.41759
  Reconstruction Time:     0.09568

SIMULTANEOUS ALGEBRAIC RECONSTRUCTION TECHNIQUE BY C IMPLMENT
  Number of Iterations:    2
  Correlation Coefficient: 0.93297
  Root Mean Square Error:  4.16636
  Signal to Noise Ratio:   9.41759
  Reconstruction Time:     0.01498

SIMULTANEOUS ALGEBRAIC RECONSTRUCTION TECHNIQUE BY CUDA IMPLMENT
  Number of Iterations:    2
  Correlation Coefficient: 0.93297
  Root Mean Square Error:  4.16636
  Signal to Noise Ratio:   9.41759
  Reconstruction Time:     1.21614
```

Fig 5.1 Results for MATLAB, C and CUDA implementation for SIRT and SART,

when N=50, N_theta=24

5.2 The maximum N and N_theta for C and CUDA implementation

For C and CUDA implementation, both SIRT and SART can support N=256 and N_theta=179, but for MATLAB code, SART fails to support N=256 and N_theta=179 where the error comes from out of memory for line At = reshape(full(A'), N*N, D, N_theta) and ph_SART = sart(A, B2, N_sart, X0, options3). Fig 5.2 shows the result of

this size. Fig 5.2 shows the result of MATLAB, C and CUDA for N=256 and N_theta=179.

From Fig 5.2, it is obvious that for SIRT, the loop time of C is 1.38s and CUDA is 0.23s respectively, which CUDA is 6 times faster than C compared with no faster when N=50 and N_theta=24, this further proves that when the size increases, the GPU is much more utilized so the speed is much faster. Similarly, for SART, the loop time of C is 7.4s and CUDA is 0.48s, which CUDA is 15.4 times faster C and compared with 1.5 times faster when N=50, N_theta=24 shown in Fig 5.1. Moreover due to large memory copy from host to device, the SART in CUDA implementation is terribly time consuming.

```
LOOP TIME: SIRT with C: 1.380000
LOOP TIME: SART with C: 7.400000
LOOP TIME: SIRT with CUDA: 0.230000
LOOP TIME: SART with CUDA: 0.480000
---------------------------
RECONSTRUCTED IMAGE QUALITY
---------------------------

SIMULTANEOUS ITERATIVE RECONSTRUCTION TECHNIQUE
  Number of Iterations:    30
  Correlation Coefficient: 0.95076
  Root Mean Square Error:  18.05447
  Signal to Noise Ratio:   10.86067
  Reconstruction Time:     2.16516

SIMULTANEOUS ITERATIVE RECONSTRUCTION TECHNIQUE BY C IMPLMENT
  Number of Iterations:    30
  Correlation Coefficient: 0.95076
  Root Mean Square Error:  18.05447
  Signal to Noise Ratio:   10.86067
  Reconstruction Time:     1.84875

SIMULTANEOUS ITERATIVE RECONSTRUCTION TECHNIQUE BY CUDA IMPLMENT
  Number of Iterations:    30
  Correlation Coefficient: 0.95076
  Root Mean Square Error:  18.05448
  Signal to Noise Ratio:   10.86067
  Reconstruction Time:     1.64179

SIMULTANEOUS ALGEBRAIC RECONSTRUCTION TECHNIQUE BY C IMPLMENT
  Number of Iterations:    2
  Correlation Coefficient: 0.96002
  Root Mean Square Error:  16.39235
  Signal to Noise Ratio:   11.69954
  Reconstruction Time:     11.28091

SIMULTANEOUS ALGEBRAIC RECONSTRUCTION TECHNIQUE BY CUDA IMPLMENT
  Number of Iterations:    2
  Correlation Coefficient: 0.96002
  Root Mean Square Error:  16.39234
  Signal to Noise Ratio:   11.69954
  Reconstruction Time:     7149.33824
```

Fig 5.2 Results for SIRT and SART in MATALB and C

When N=256, N_theta=179

5.3 Results for other problem sizes

(1) The Fig 5.3 shows the image quality for SIRT and SART with N/N_theta increasing with 50/24,100/48,150/72,200/96,256/179.

(2) Fig 5.4 shows how much faster the C implementation of SIRT and SART than Matlab code with N/N_theta increasing with 50/24,100/48,150/72,200/96,256/179. It is evaluated by diving Matlab simulated reconstruction time by the result of C implementation, i.e. $\frac{\text{Reconstructed time for MATLAB}}{\text{Reconstructed time for C implementation}}$. From graph, it is easily shown that with the increase of size, this ratio also increases for SART algorithm, however this ration increase first and then decreases for SIRT algorithm. It is probably because the input matrix and vector size used in sparse level 2 routines is smaller in SART than in SIRT. With the problem size increase, the CPU is over utilized in SIRT when performs sparse level 2 routines which slows down the calculation, however the quad core CPU can be efficiently utilized and data got in more parallel for SART because its smaller input size for sparse level 2 routines, resulting in speed up the calculation.

(3)Fig 5.5(a) shows the absolute loop time for C and CUDA implementation for SIRT and SART with N/N_theta increasing with 50/24,100/48,150/72,200/96,256/179. Fig 5.5(b) shows how much time faster for CUDA implementation of SIRT and SART compared with C implementation for loop time.

From Fig 5.5(a) it is straightforward that with the problem size increase the absolute loop time increases and especially C implementation for SART increase fastest. From Fig 5.5(b), it can been seen that for SIRT, with problem size increase the loop time ratio increases either because larger problem size makes more use of parallel calculation on GPU. However for SART, with problem size increase the loop time ratio increases first and then decreases and stays almost the same and then increases again. Ideally it should present the same trend as SIRT, however this phenomemon probably from my code. According to Chapter 4 Fig 4.6, I only define an one dimension device variable for a_theta, a_theta', w_theta, v_theta CRS format and b_theta and still initialize these device variable by copy them from host in theta(inner) loop, however just as state before, the memory copy is very consuming. So when problem size increases to a certain level, memory copy accounts for major time consumption and at the same time the Quad-core CPU for C calculations are still efficiently , so the loop ration presents decrease. However when problem size larger to another certain level where Quad-core CPU for C calculation

is saturated and can't parallelize anymore and at the same time the GPU is most utilized, resulting in large increase in loop time ration.

Performance can be improved for CUDA implementation by initializing device variable before entering into loop for iterative X^k calculation.

Graphs from Fig 5.3-5.5 are obtained by simulated results which are shown in Appendix serving as reference and demonstration.

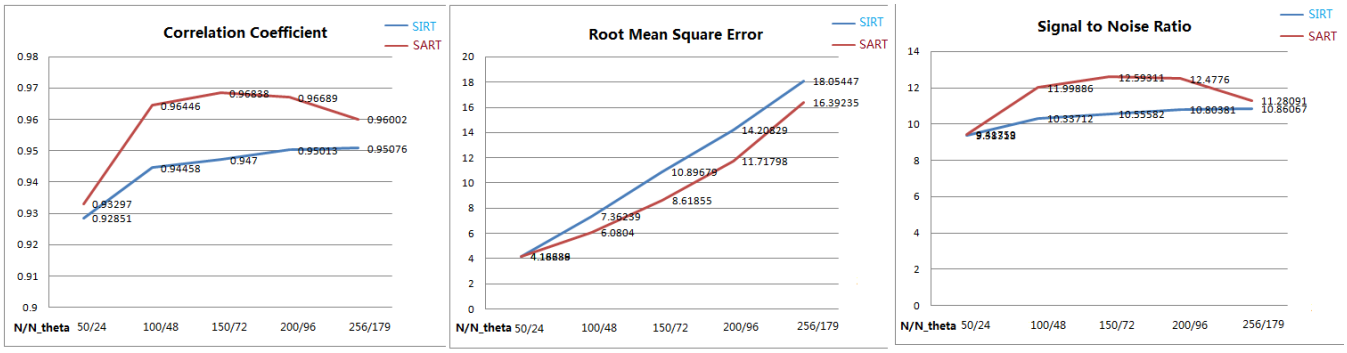

Fig 5.3 Correlation Coefficient, Root Mean Square Error, Signal to Noise Ration for SIRT and SART with N/N_theta increasing with 50/24,100/48,150/72,200/96,256/179

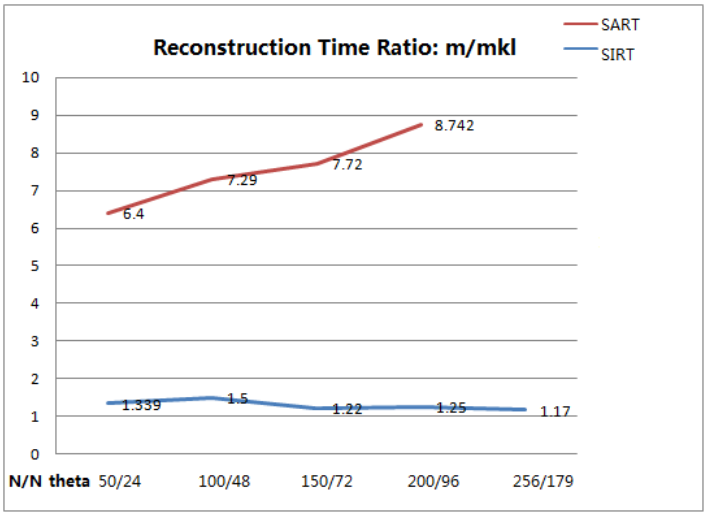

Fig 5.4 Reconstruction Time Ratio for SIRT and SART with N/N_theta increasing with 50/24,100/48,150/72,200/96,256/179

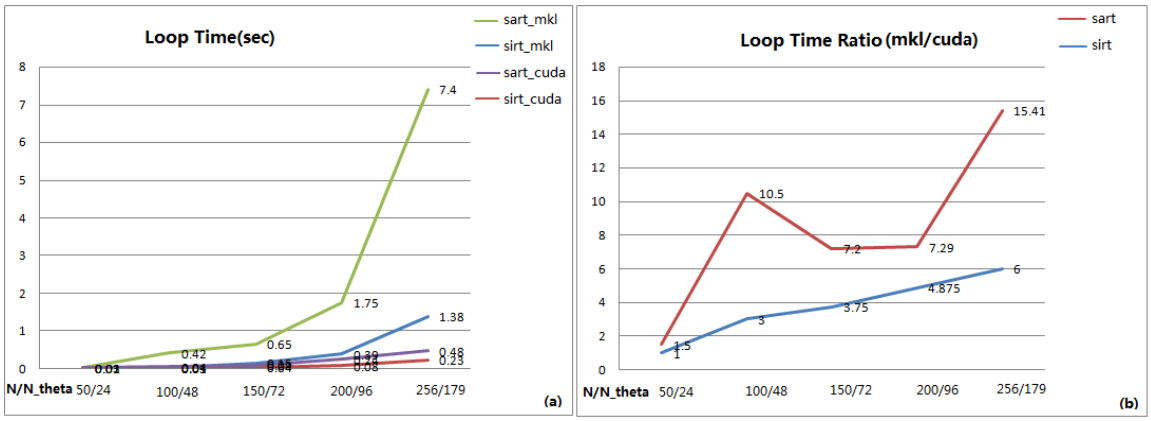

Fig 5.5 With N/N_theta increasing with 50/24,100/48,150/72,200/96,256/179 (a) shows the absolute loop time for C and CUDA implementation for SIRT and SART (b) shows how much time faster for CUDA implementation of SIRT and SART compared with C implementation for loop time

Conclusion

This project realizes the implementation for SIRT and SART reconstruction methods. For C and CUDA implementation, both SIRT and SART can support N=256 and N_theta=179, but for MATLAB code, SART fails to support N=256 and N_theta=179 where the error comes from out of memory.

By comparing the MATLAB and C code with N/N_theta increasing with 50/24,100/48,150/72,200/96,256/179, SIRT in C is just slightly faster than SIRT in MATLAB, however SART in C is largely faster than MATLAB. This is mainly because for SART, the input for matrix calculation is smaller than SIRT which can speed up the matrix calculation. With the problem size increase, the ratio of MATLAB speed to C speed in SART increases, but for SIRT it increases first and then stay the same or even a little bit decrease. It is probably because the input matrix and vector size used in sparse level 2 routines is smaller in SART than in SIRT. With the problem size increase, the CPU is over utilized in SIRT when performs sparse level 2 routines which slows down the calculation, however the quad core CPU can be efficiently utilized and data got in more parallel for SART because its smaller input size for sparse level 2 routines, resulting in speed up the calculation.

By comparing the CUDA and C code, test for loop time is introduced because too much time is consumed in memory copy in cuda code. With N/N_theta increasing with 50/24,100/48,150/72,200/96,256/179, the loop time ration for SIRT increases because larger problem size makes more use of parallel calculation on GPU. But for SART, the ratio increase first and then decrease and the increase, it is mainly because I do initialization in loops in CUDA, memory copy is very consuming. So when problem size increases to a certain level, memory copy accounts for major time consumption and at the same time the Quad-core CPU for C calculations are still efficiently , so the loop ration presents decrease. However when problem size larger to another certain level where Quad-core CPU for C calculation is saturated and can't parallelize anymore and at the same time the GPU is most utilized, resulting in large increase in loop time ration.

With the description above, for my code, improvements can be got by initializing all the variables outside of the loop.

Reference

[1] K. Mueller, R. Yagel, and J. J. Wheller. "Fast implementations of algebraic methods for 3d reconstruction from cone-beam data." *IEEE Transactions on Medical Imaging*, Sept 1999.

[2] J. Radon, Über die Bestimmung von Funktionen durch ihre Integralwerte Längs Gewisser Mannigfaltigkeiten, *Math.-Phys. Kl.* 69 (1917), 262-267.

[3] B. Turonova. "Simultaneous Algebraic Reconstruction Technique for Electron Tomography using OpenCL." *Master Thesis*, Jun.2011

[4] R. Dorrance, F. Ren, and D. Marković, "A scalable sparse matrix-vector multiplication kernel for energy-efficient sparse-blas on FPGAs." *2014 ACM/SIGDA international symposium on Field-programmable gate arrays.* pp. 161-170. Feb 2014.

[5] Fengbo Ren; Dorrace, R.; Wenyao Xu; Markovic, D., "A single-precision compressive sensing signal reconstruction engine on FPGAs," *Field Programmable Logic and Applications (FPL), 2013 23rd International Conference on* , vol., no., pp.1,4, 2-4 Sept. 2013

[6] J.H.Jorgensen. "Knowledge-Based TomographyAlgorithms." *Kongens Lyngby* 2009

[7] W.Chlewicki. "3D Simultaneous Algebraic Reconstruction Technique for Cone-Beam Projections." *MS Thesis.* 2011

[8] K.A.Baugher. "Sparse Matrix Sparse Vector Multiplication using Parallel and Reconfigurable Computing." MS Thesis. May 2004

[9] S.Hong, H.Kim. "Memory-level and Thread-level Parallelism Aware GPU Architecture Performance Analytical Model." *ACM SIGARCH Computer Architecture.* June 2009

[10] Web"http://benchmarkreviews.com/index.php?option=com_content&task=view&id=187&Itemid=38&limit=1&limitstart=3"

[11] J.Nickolls, I.Buck,etc. "Scalable Parallel Programming with CUDA." *ACM.* Mar.2008

Appendix

```
LOOP TIME: SIRT with C: 0.030000
LOOP TIME: SIRT with CUDA: 0.010000
LOOP TIME: SART with C: 0.420000
LOOP TIME: SART with CUDA: 0.040000
---------------------------
RECONSTRUCTED IMAGE QUALITY
---------------------------

SIMULTANEOUS ITERATIVE RECONSTRUCTION TECHNIQUE
  Number of Iterations:    30
  Correlation Coefficient: 0.94458
  Root Mean Square Error:  7.36239
  Signal to Noise Ratio:   10.33712
  Reconstruction Time:     0.07770

SIMULTANEOUS ITERATIVE RECONSTRUCTION TECHNIQUE BY C IMPLMENT
  Number of Iterations:    30
  Correlation Coefficient: 0.94458
  Root Mean Square Error:  7.36239
  Signal to Noise Ratio:   10.33712
  Reconstruction Time:     0.05177

SIMULTANEOUS ITERATIVE RECONSTRUCTION TECHNIQUE BY CUDA IMPLMENT
  Number of Iterations:    30
  Correlation Coefficient: 0.94458
  Root Mean Square Error:  7.36239
  Signal to Noise Ratio:   10.33711
  Reconstruction Time:     0.77331

SIMULTANEOUS ALGEBRAIC RECONSTRUCTION TECHNIQUE
  Number of Iterations:    2
  Correlation Coefficient: 0.96446
  Root Mean Square Error:  6.08040
  Signal to Noise Ratio:   11.99886
  Reconstruction Time:     1.22152

SIMULTANEOUS ALGEBRAIC RECONSTRUCTION TECHNIQUE BY C IMPLMENT
  Number of Iterations:    2
  Correlation Coefficient: 0.96446
  Root Mean Square Error:  6.08040
  Signal to Noise Ratio:   11.99886
  Reconstruction Time:     0.13675

SIMULTANEOUS ALGEBRAIC RECONSTRUCTION TECHNIQUE BY CUDA IMPLMENT
  Number of Iterations:    2
  Correlation Coefficient: 0.96446
  Root Mean Square Error:  6.08040
  Signal to Noise Ratio:   11.99886
  Reconstruction Time:     35.77111
```

Fig Results for N=100, N_theta=48

```
LOOP TIME: SIRT with C: 0.150000
LOOP TIME: SIRT with CUDA: 0.040000
LOOP TIME: SART with C: 0.650000
LOOP TIME: SART with CUDA: 0.090000
--------------------------
RECONSTRUCTED IMAGE QUALITY
--------------------------

SIMULTANEOUS ITERATIVE RECONSTRUCTION TECHNIQUE
  Number of Iterations:    30
  Correlation Coefficient: 0.94700
  Root Mean Square Error:  10.89679
  Signal to Noise Ratio:   10.55582
  Reconstruction Time:     0.24280

SIMULTANEOUS ITERATIVE RECONSTRUCTION TECHNIQUE BY C IMPLMENT
  Number of Iterations:    30
  Correlation Coefficient: 0.94700
  Root Mean Square Error:  10.89679
  Signal to Noise Ratio:   10.55582
  Reconstruction Time:     0.19872

SIMULTANEOUS ITERATIVE RECONSTRUCTION TECHNIQUE BY CUDA IMPLMENT
  Number of Iterations:    30
  Correlation Coefficient: 0.94700
  Root Mean Square Error:  10.89679
  Signal to Noise Ratio:   10.55582
  Reconstruction Time:     0.82278

SIMULTANEOUS ALGEBRAIC RECONSTRUCTION TECHNIQUE
  Number of Iterations:    2
  Correlation Coefficient: 0.96838
  Root Mean Square Error:  8.61855
  Signal to Noise Ratio:   12.59311
  Reconstruction Time:     5.18025

SIMULTANEOUS ALGEBRAIC RECONSTRUCTION TECHNIQUE BY C IMPLMENT
  Number of Iterations:    2
  Correlation Coefficient: 0.96838
  Root Mean Square Error:  8.61855
  Signal to Noise Ratio:   12.59311
  Reconstruction Time:     0.67069

SIMULTANEOUS ALGEBRAIC RECONSTRUCTION TECHNIQUE BY CUDA IMPLMENT
  Number of Iterations:    2
  Correlation Coefficient: 0.96838
  Root Mean Square Error:  8.61855
  Signal to Noise Ratio:   12.59311
  Reconstruction Time:     283.80243
```

Fig Results for N=150, N_theta=72

```
LOOP TIME: SIRT with C: 0.390000
LOOP TIME: SIRT with CUDA: 0.080000
LOOP TIME: SART with C: 1.750000
LOOP TIME: SART with CUDA: 0.240000
--------------------------
RECONSTRUCTED IMAGE QUALITY
--------------------------

SIMULTANEOUS ITERATIVE RECONSTRUCTION TECHNIQUE
  Number of Iterations:    30
  Correlation Coefficient: 0.95013
  Root Mean Square Error:  14.20829
  Signal to Noise Ratio:   10.80381
  Reconstruction Time:     0.62538

SIMULTANEOUS ITERATIVE RECONSTRUCTION TECHNIQUE BY C IMPLMENT
  Number of Iterations:    30
  Correlation Coefficient: 0.95013
  Root Mean Square Error:  14.20829
  Signal to Noise Ratio:   10.80381
  Reconstruction Time:     0.49856

SIMULTANEOUS ITERATIVE RECONSTRUCTION TECHNIQUE BY CUDA IMPLMENT
  Number of Iterations:    30
  Correlation Coefficient: 0.95013
  Root Mean Square Error:  14.20829
  Signal to Noise Ratio:   10.80381
  Reconstruction Time:     0.96697

SIMULTANEOUS ALGEBRAIC RECONSTRUCTION TECHNIQUE
  Number of Iterations:    2
  Correlation Coefficient: 0.96689
  Root Mean Square Error:  11.71798
  Signal to Noise Ratio:   12.47760
  Reconstruction Time:     15.93039

SIMULTANEOUS ALGEBRAIC RECONSTRUCTION TECHNIQUE BY C IMPLMENT
  Number of Iterations:    2
  Correlation Coefficient: 0.96689
  Root Mean Square Error:  11.71798
  Signal to Noise Ratio:   12.47760
  Reconstruction Time:     1.82226

SIMULTANEOUS ALGEBRAIC RECONSTRUCTION TECHNIQUE BY CUDA IMPLMENT
  Number of Iterations:    2
  Correlation Coefficient: 0.96689
  Root Mean Square Error:  11.71798
  Signal to Noise Ratio:   12.47760
  Reconstruction Time:     1149.81980
```

Fig Results for N=200, N_theta=96