High Performance Computing for Neutron Tomography Reconstruction

A Parallel Approach to Filtered Backprojection

Zongpu Li[1], Cain Gantt[2], and Rick Archibald[3*]

[1]Department of Physics and Materials Science, City University of Hong Kong
[2]Department of Mathematics, Georgia College & State University
[3]Oak Ridge National Laboratory
[*]Mentor

August 4, 2017

Abstract

For model reconstruction in neutron tomography and laminography applications, Filtered Backprojection (FBP) is a commonly used and reliable method. Long program execution times and large memory requirements are obstacles to practical use of these algorithms in a laboratory setting. During the Research Experiences in Computational Science, Engineering, and Mathematics (RECSEM) program, we explored implementations of a number of approaches to the parallelization of FBP algorithms with the goal of reducing program execution time. Specifically, we created a program to perform FBP on supercomputing clusters, and investigated both the use of the Message Passing Interface (MPI) library and parallel code execution on a Graphics Processing Unit (GPU). These are both increasingly available resources on High Performance Computing platforms, and our insights can be used for improvements to image reconstruction.

Background

Filtering is necessary for any backprojection method to create a useful reconstructed volume[1]. When backprojection is performed without filtering, most fine details are lost to

blurring. Even a simple ramp filter, commonly used in tomography, could be used to improve image quality. We set out to test a number of filters particularly the inverse filter function described by Myagotin, Voropaev, Helfen, *et al.*, with the equation outlined below:

$$\bar{H}(k_x, k_y, k_z) = \begin{cases} \frac{\sin\phi}{2}\sqrt{k_x^2 + k_y^2 - k_z^2 \cdot \cot^2\phi}, & |k_z| \leq \frac{\sqrt{k_x^2+k_y^2}}{\cot\phi}; \\ 0, & \text{otherwise}, \end{cases} \tag{1}$$

where k_x, k_y, k_z represent coordinates in the Fourier domain, and ϕ is the laminography angle.

This inverse filter function is three-dimensional and lies in the Fourier domain. Ideally, the filter is applied by first calculating its inverse Fourier transform, then computing its three-dimensional convolution with the volume after backprojection. The application of the Fourier Slice theorem allows the 3D convolution to be replaced by a set of 2D convolutions, and the convolution theorem allows the computation of the convolution in the spatial domain to be replaced by multiplication in the Fourier domain [2]. The 2D filter function derived from 1 for a particular projection is given by

$$\bar{H}(k_x, k_y) = \frac{\sin\phi}{2}|k_y|. \tag{2}$$

which is dependent only on the column index and the geometry of the laminography experiment. This allows further reduction to a set of 1D Fourier transforms of each column in a projection, which is then filtered by multiplication in the Fourier domain with the inverse filter function. This approach allows the filtering step to be completed with less intensive computation than with the 3D convolution, improving program execution time and memory requirements. The MATLAB program provided to us by our mentor Rick Archibald (ORNL) utilized this approach to projection filtering.

Objectives

We set out to accomplish a number of goals towards the overall reduction of computation and memory requirements without compromising the quality of the quality of the reconstructed model. We first needed to determine which inverse filter function to use in our implementation, balancing the computation time required and the image quality that it produced. Next, we wanted to implement the inverse radon transform algorithm in C, which would allow the program to run on a wider range of high performance computation (HPC) platforms. Once the program was functioning in C, we explored a number of approaches to execute different parts of the code simultaneously through parallelization on the CPU and on the GPU.

Inverse Filter Selection

To begin, we tested the performance of the MATLAB code without filtering. Then, we examined the filtered image and make a comparison. With the only filter stated in original

material, it is necessary to research into other type of filters, try to apply them in Matlab code, and analyze their difference in performance. If possible, we will do some experiments in the combinations of these filters.

With the simulated data, we have tested the Matlab code with or without the filtering section. This helps us to understand the algorithm of the filtering and the reconstruction process. Ramp Filter in the form of line chart, the image output with Ramp Filter, the image output without Ramp Filter. A noticeable change in image quality was detected when comparing the filtered output with an unfiltered one. We also run the Matlab code with industrial data, which was gathered in ORNL. The visual effect is even more sharper. The five dark areas can be clearly recognized in the filtered image. We can even see the tape used to stabilize the rotated plate. The difference in color indicates the different materials inside the object.

It is not difficult to apply a single filter to projections, though choosing a proper filter and implementing it in code is more challenging. Most of the time, we find a filter function that has already been built by other scientists, so it is vital for us to understand the meaning of mathematical equations and parameters that appear in their paper. That is definitely a time-consuming task. Other than this, another complex part in filter application is to figure out the data layout in every part of the code. For example, the data layout in Matlab is column-major, while in C, it is row-major. It is very easy to be lost in the data layout, the direction of the matrix, fourier transform, and the filtering. To deal with this situation, we first used a small dataset, and run through the Matlab code step by step, to see what happened in every single step. Then, we do the same thing with C code. We record them in our notebook, and make a comparison between them. After figuring out how they differ from each other, it is much clearer how we should arrange our data allocation to achieve the same output.

The Discrete Fourier Transform algorithms provided by the FFTW library are especially fast for input sizes equal to a power of 2. In our tests, we increased the transform size from 1000 to 1024, and did 1,000 times of transform. The average running time of the size 1,000 is 34.56 second per 1,000 times, while the running time of size 1,024 is 28.14 second per 1000 times. The size of transform is increased by 2.4%, but the running time is reduced by 18.6%. This would help us to further reduce the program running time.

Serial Program Development

We wanted to increase the portability of the program to other machines with a C implementation of the projection filtering and the inverse radon transform. C compilers allow a greater degree of optimization for particular computers hardware configuration, so we expected to see improvements in the execution time and memory usage once the program was written in C[3].

Since both the data generated by the MATLAB simulation and the data collected from neutron tomography at Oak Ridge National Laboratory were formatted for use in MATLAB, we needed a way to convert this data into a format that the C program could read. We

chose to write the variables and arrays from MATLAB into a binary file. This prevented erroneous modification of the data or its precision, which would complicate comparison with the MATLAB program. By first writing the dimensions of the projection, we are able to determine how much memory to allocate for each array, then to read directly from the binary file into the memory for the array.

Our next step was to implement the laminographic ramp filter. Since the inverse filter is applied in the Fourier domain, we used the FFTW libraries to transform each column of the projection [4]. The filter array is longer than each column of the projection, so the projections are zero-padded before calculation to prevent the transform from being periodic. The filter is then applied with element-wise multiplication of the transformed column with the filter array, and their product is processed with an inverse transform to create the filtered projection.

Once the projections are filtered, they are back projected into the volume to create the model. Each volume pixel (voxel) in the 3D array is given a triplet of coordinates in space, which is then used to calculate where the voxels shadow lies on the detector plane as determined by the rotation and laminographic angles. For the voxels that fall on the filtered projection, the value at that point is interpolated from the filtered projection; this calculation is not performed for voxels that land outside the bounds of the projection. We used bilinear interpolation for its balance between accuracy and computation time, though other algorithms could be used for improvements on either of those fronts.

The MATLAB codes implementation of the backprojection calculations involve the creation of a meshgrid to keep track of the coordinates, i.e. three 3-dimensional matrices, each the size of the volume holding the x, y, or z component of the coordinate triplet for each point in the voxel. Since the rotation on each of the voxels is independent from the others, we were able to reduce the memory footprint in this step by only holding in memory the coordinates for a single voxel at a given time. This greatly reduces the memory requirements for the backprojection calculation, which can become prohibitively large at even a modest resolution.

Parallel Program Development

To further reduce program execution time, we set out to structure portions of our program to execute simultaneously. As Myagotin, Voropaev, Helfen, *et al.* have proposed, the backprojection algorithm can be run in parallel one of two ways: computation distribution by projections, or memory distribution by portions of the reconstructed volume. We chose to implement the former through the use of Message Passing Interface (MPI).

A single process is assigned to read the binary data from storage and to initialize the filter array. It first broadcasts the dimensions of the data to all other processes, which then allocate an appropriate amount of memory to then receive the projection data and the filter array. Each process performs filtering and backprojection on only a portion of the projections. Using an `MPI_Reduce` call, the volume from each process is summed into a single array to be written as output.

Our first implementation for parallelization faces a number of drawbacks. Since each process must record values for the entire volume, the total amount of memory required grows with both the resolution of the volume and the number of processes used. This could be remedied by using a shared memory configuration, so long as I/O operations can be synchronized to prevent collisions.

GPU Acceleration

The major purpose of the GPU implementation is to reduce the running time of our program. Compared with a Central Processing Unit (CPU), Graphics Processing Units (GPUs) have heavily parallelized architecture, and are built upon thousands of smaller cores, which are optimized for multitasking simultaneously. This feature can boost our program since our code consists of three calculation-intensive sections, most of them are independent and thus can be run at the same time. For example, there are 1094 projections in our input, each of which undergoes element-wise multiplication by a vector. In our current algorithm, the CPU will loop every element inside a column, loop every column inside a page, and loop every page in the whole array. In this process, we waste a big amount of time in waiting for the former loop to finish. So if this could be done in parallel, the total running time will reduce significantly.

There are two major approach to use GPU to accelerate a CPU code. The first one is to use drop-in libraries such as MAGMA (Matrix Algebra on GPU and Multicore Architectures), cuBLAS (Nvidia's CUDA Basic Linear Algebra Subroutine Library), or cuFFT (Nvidia's CUDA FFT library). These libraries are more like C language extensions, which means they have similar structure as CPU code, and are relatively easier to learn. The second method is to generate accelerator code as a variant of CPU source. It is possible to deeply access the hardware and thereby achieve high performance. However, this might require one to have adequate knowledge in both hardware and coding. In consideration of the program duration and our backgrounds, we decided to use the first approach to accelerate our code.

There are three parts in our project that can benefit from GPU resource: fourier transform, filter application, and bilinear interpolation. So far we have had two attempts in the first two of them. For fourier transform, we decided to use cuFFT library, in which there is a function performs exactly the same as Matlab does. In the simulated data testing, the running time of serial code is 31 times that of the parallel code. These testing was doing the fourier transform and inverse fourier transform to a 1024 by 1000 matrix, where the transform direction was along the columns. An interesting condition to mention is that, when further increasing the matrix size, the running time of serial code increases dramatically. However, the performance of GPU code seems not to be affected. For the filter application part, we have made good progress in cuBLAS code, and it works well in our testings. However, in our final code, cuBLAS function cannot generate expected result.

Although CUDA has huge enormous technical resource on the internet, its documentation is also well written, we still encountered many problems when we try to use their function

in our code. For example, in the filter application part, we have to redesign the layout of our filter to fit the structure of cuBLAS function. In addition, the data computed by cuFFT function has to be in the form of cufftDoubleComplex, while in cuBLAS function, it is set to be cuDoubleComplex. In our attempt, we transferred the output of cuFFT back to host memory, converted the data type, then transferred them into GPU again for cuBLAS function. In this process, we would waste a lot of time in data transfer and it will reduce the efficiency of GPU computing.

The bottleneck of data transfer is worse than we expected. As mentioned above, in our testing fourier transform codes, GPU code runs significantly faster than CPU code. However, in our programs for laminography, GPU code can barely run faster than a CPU code. This can be explained by too many data transfer functions, as well as too many loops in our GPU code. We might improve this in the future.

Data

We measured the time that Matlab took to run one part of the code, the"iradon transform" section. In this way, we can better compare it with the running time of our Real Data Code, which basically only contains the "iradon transform" section. The table of running time is as follows:

Data Type	Size	Filter	Average Running Time	Variance
Phantom	32 X 32	N/A	4.6517	0.0366
Phantom	32 X 32	Ramp Filter	4.5633	0.0124
Phantom	64 X 64	Ramp Filter	45.6303	1.8069
Phantom	128 X 128	Ramp Filter	IP	IP
Real	1501 X 1501	N/A	IP	IP
Real	1501 X 1501	Scaled Ramp Filter	740.7562	IP

We collected run time data with varying data sizes and using a number of different processes. The testing was performed on the Bridges system at the Pittsburgh Supercomputing Center[5], [6].

Processes	Volume Resolution	Trials	Avg. Run Time (s)	Standard Deviation
1	65x65x65	30	1.3347	0.0359
12	129x129x129	30	2.9256	0.0123
12	257x257x257	10	29.6125	0.0840
16	129x129x129	30	2.6369	0.0157
16	257x257x257	15	26.1720	0.0578
20	129x129x129	30	2.5259	0.0169
20	257x257x257	20	27.3555	0.1506
24	129x129x129	30	2.4983	0.0310
24	257x257x257	20	38.5717	0.3838
28	129x129x129	30	3.3688	0.2424
28	257x257x257	20	47.8398	1.5117

Future Works

Currently, our objective is to convert the Matlab code into C code. After this, we will begin to parallelize the C code. In our plan, there are three stages of the parallelization. In the first stage, we are going to use MPI, doing the data decomposition, broadcasting them to several workers. Then each of the workers will do one part of the reconstruction. In the end, we will gather these together to obtain the final image of reconstruction. In the second stage, we will further parallelize the code with MPI, such as using the "ring" operation in MPI. We might also try some better methods to decompose the data. In the third stage, we will try to parallelize the codes that are sent to the nodes. We will enable GPUs in each nodes and do the parallel computing.

Acknowledgements

We would like to thank our home institutions, Georgia College and State University and City University of Hong Kong for their support with our research careers. We would also like to thank the University of Tennessee, Knoxville and to Dr. Kwai Wong for leading the Research Experiences in Computational Sciences, Engineering, and Mathematics (RECSEM) program, a Research Experiences for Undergraduates (REU) program that is funded and made possible by the National Science Foundation. Our work used the Extreme Science and Engineering Discovery Environment (XSEDE), which is supported by National Science Foundation grant number OCI-1053575. Specifically, we used the Bridges system, which is supported by NSF award number ACI-1445606, at the Pittsburgh Supercomputing Center (PSC).

References

[1] G. L. Zeng, "Revisit of the Ramp Filter", *IEEE Transactions on Nuclear Science*, vol. 62, no. 1, pp. 131–136, Feb. 2015, ISSN: 0018-9499. DOI: 10.1109/TNS.2014.2363776.

[2] A. Myagotin, A. Voropaev, L. Helfen, D. Hänschke, and T. Baumbach, "Efficient volume reconstruction for parallel-beam computed laminography by filtered backprojection on multi-core clusters", *IEEE Transactions on Image Processing*, vol. 22, no. 12, pp. 5348–5361, Dec. 2013, ISSN: 1057-7149. DOI: 10.1109/TIP.2013.2285600.

[3] V. Menon and A. E. Trefethen, "Multimatlab integrating matlab with high performance parallel computing", in *Supercomputing, ACM/IEEE 1997 Conference*, Nov. 1997, pp. 30–30. DOI: 10.1109/SC.1997.10011.

[4] M. Frigo and S. Johnson, "The design and implementation of FFTW3", *Proceedings of the IEEE*, vol. 93, no. 2, pp. 216–231, 2005, Special issue on "Program Generation, Optimization, and Platform Adaptation".

[5] J. Towns, T. Cockerill, M. Dahan, I. Foster, K. Gaither, A. Grimshaw, V. Hazlewood, S. Lathrop, D. Lifka, G. D. Peterson, R. Roskies, J. R. Scott, and N. Wilkins-Diehr, "Xsede: Accelerating scientific discovery", *Computing in Science and Engineering*, vol. 16, no. 5, pp. 62–74, 2014, ISSN: 1521-9615. DOI: doi.ieeecomputersociety.org/10.1109/MCSE.2014.80.

[6] N. A. Nystrom, M. J. Levine, R. Z. Roskies, and J. R. Scott, "Bridges: A uniquely flexible hpc resource for new communities and data analytics", in *Proceedings of the 2015 XSEDE Conference: Scientific Advancements Enabled by Enhanced Cyberinfrastructure*, ser. XSEDE '15, St. Louis, Missouri: ACM, 2015, 30:1–30:8, ISBN: 978-1-4503-3720-5. DOI: 10.1145/2792745.2792775. [Online]. Available: http://doi.acm.org/10.1145/2792745.2792775.