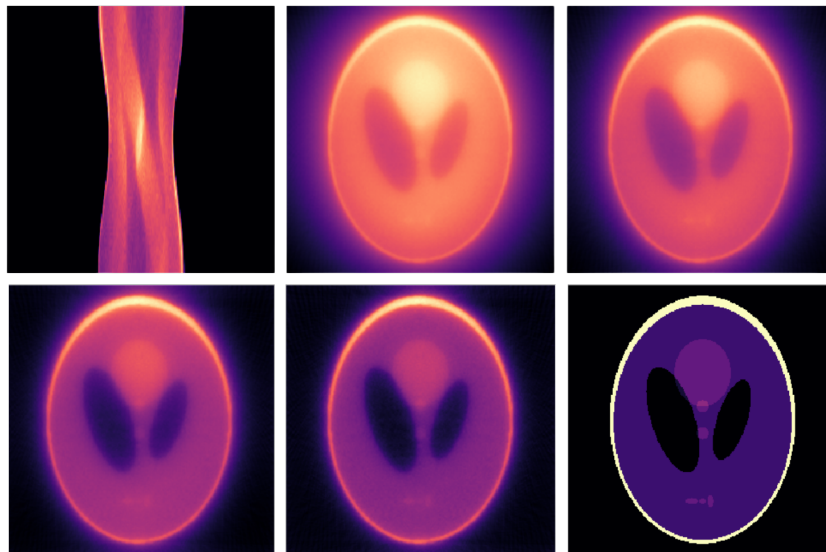# Image Reconstruction with Cimmino's Algorithm

Task M4.T1D - Project

Kate Suraev (s224854029)

September 16, 2025

Please see the GitHub repository for the complete code as I am only able to upload one code file to OnTrack. I have added my tutor Daniel and the unit chair Maksym to the repository as collaborators. The final project is in the FINAL_Project directory. The other directories have been left for reference.

Introduction

In this project, we explore the implementation of Cimmino's algorithm for image reconstruction problems, such as those encountered in computed tomography (CT). The algorithm is particularly useful for solving large-scale linear systems that arise in these applications due to its ability to handle inconsistent systems and its parallelisable nature. We do not aim to provide a perfectly reconstructed scan but rather to demonstrate the idea of using Cimmino's algorithm in a parallel computing environment to achieve a reasonable approximation of the original image. The main program uses Metal for GPU computing, and we also provide sequential and OpenMP implementations for comparison.

Project Structure

The project is structured as follows:

- **Metal Implementation:** The main program is written in C++ and Metal Shading Language (MSL) with some Objective-C++ for interfacing with Metal. This implementation leverages the parallel processing capabilities of the GPU to efficiently perform the computations required by Cimmino's algorithm. The source files are located in the `src` directory and the header files in the `metal-include` directory. The metal kernels are in the `metal-shaders` directory. I have split the Metal compute and render logic into two separate classes (`MTLComputeEngine` and `MTLRenderEngine`) for better organisation and modularity.

- **Sequential Implementation:** A simple C++ program that implements Cimmino's algorithm in a single-threaded manner for baseline performance comparison. This is in `Other-Implementations/Sequential/sequential.cpp`.

- **OpenMP Implementation:** A C++ program that uses OpenMP to parallelise the computation across multiple CPU cores, providing a middle ground between the sequential and Metal implementations. This is in `Other-Implementations/OpenMP/openmp.cpp`.

- **Python Scripts:** Python scripts are used to generate the projection matrix and phantom image using the Astra Toolbox, as well as to visualise the results outside of the Metal application. These can be found in the `metal-data` directory.

- **Data and Log Files:** The `metal-data` directory contains the projection matrix file as well as the reconstructed image files. The `metal-logs` directory contains log files generated during the execution of the Metal program for debugging and performance analysis.

- **CMake Build System:** The project uses CMake for building the C++ and Metal

code. It includes all the necessary frameworks and headers and automatically compiles the Metal shaders. The CMake configuration file is located in the root directory.

- **Instructions:** A README.md file is provided in the root directory with instructions on how to build and run the project.

Cimmino's Algorithm Overview

To give a high-level overview, Cimmino's algorithm is a row-iterative algorithm that simultaneously reflects the current approximation point (estimate) across all hyperplanes defined by the equations of the linear system. The subsequent approximation is then computed as the average of these reflections. The result is a convergent sequence of approximations that approaches either the exact solution in the case of a consistent system or the least-squares solution in the case of an inconsistent system [1].

Given an $m \times n$ matrix A with rows a_i, a vector $b \in \mathbb{R}^m$ and positive weights ω_i for $i = 1, 2, \ldots, m$, the algorithm can be expressed as follows:

$$x^{k+1} = \sum_{i=1}^{m} \frac{\omega_i}{\omega} y^{(k,i)} = x^k + 2 \sum_{i=1}^{m} \frac{\omega_i}{\omega} \frac{b_i - a_i^T x^k}{\|a_i\|^2} a_i \quad \text{for } k = 0, 1, 2, \ldots \tag{1}$$

Here, x^k is the current approximation, $y^{(k,i)}$ is the reflection of x^k across the hyperplane defined by the i-th equation, ω_i are positive weights assigned to each equation, and $\omega = \sum_{i=1}^{m} \omega_i$ [1].

In simpler terms, the algorithm can be expressed in matrix form as:

$$x^{k+1} = x^k + \frac{2}{\omega} A^T D^T D (b - Ax^k) \quad \text{for } k = 0, 1, 2, \ldots \tag{2}$$

where D is a diagonal matrix with entries $D_{ii} = \sqrt{\omega_i} / \|a_i\|$.

$$D = \begin{bmatrix} \frac{\sqrt{\omega_i}}{\|a_1\|} & 0 & \ldots & 0 \\ 0 & \frac{\sqrt{\omega_2}}{\|a_2\|} & \ldots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \ldots & \frac{\sqrt{\omega_m}}{\|a_m\|} \end{bmatrix} \tag{3}$$

For simplicity, we choose particular weights $\omega_i = \|a_i\|^2$ for $i = 1, 2, \ldots, m$. This choice simplifies the diagonal matrix D to the identity matrix I, leading to a more straightforward update rule:

$$x^{k+1} = x^k + \frac{2}{\omega} A^T (b - Ax^k) \quad \text{for } k = 0, 1, 2, \ldots \tag{4}$$

Implementation Details

We intend to perform a simulation of a CT scan by setting the geometry parameters of our 'scanner', generating a projection matrix that models the scanner geometry, performing a scan of a given image to obtain a sinogram (a vector of measurements), and then using Cimmino's algorithm to reconstruct the original image from the sinogram.

Phantom: A phantom image is a standard test image used in medical imaging to simulate the human body [2]. In this project, we use the Shepp-Logan phantom, a widely used test image in CT imaging that models a head cross-section. This is our ground truth image that we aim to reconstruct. The 256x256 Shepp-Logan phantom is generated in Python using Astra Toolbox [3].
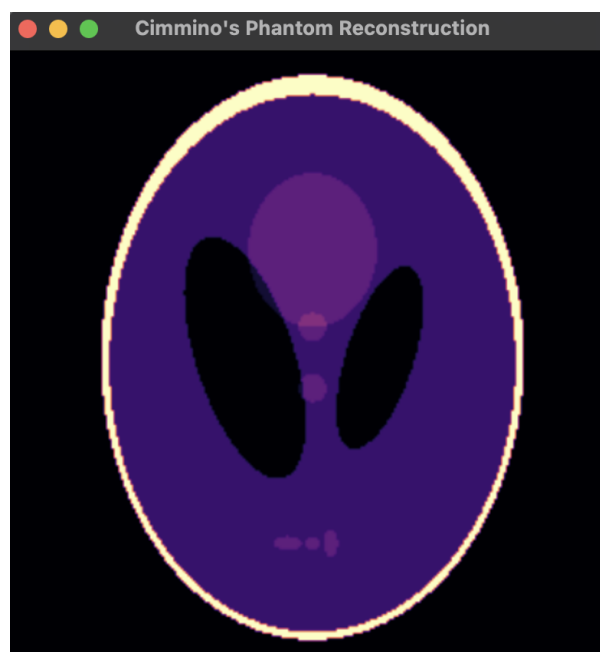

Figure 1: Shepp-Logan Phantom (256x256)

Sinogram: A 2D array where each row represents the projection of the image at a specific angle [4]. This is our measurement data obtained from the simulated CT scan and can be thought of as b in the linear system $Ax = b$. The sinogram is generated in a Metal kernel using the phantom image and a ray marching algorithm. As seen below, the sinogram does not look like the original image, but it contains all the information needed to reconstruct it.
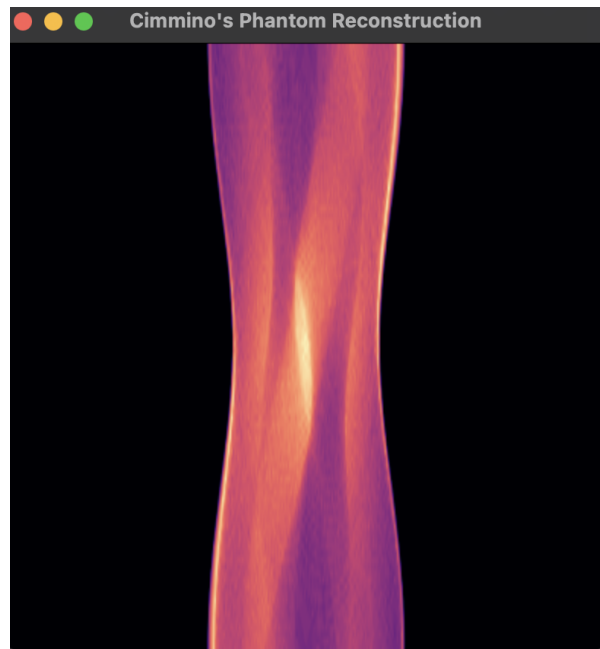
Figure 2: Sinogram (90 angles, 725 detectors)

We also use metal kernel functions to normalise the sinogram values to a 0-1 range for better visualisation and computation. For this, we use a SIMD parallel reduction algorithm, inspired by an example from the MSL specification [5], to find the maximum value in the sinogram and then normalise all values accordingly. Without normalisation, we are not able to properly visualise the sinogram as an image since the values are not in the 0-1 range. Below is the non-normalised sinogram for comparison.
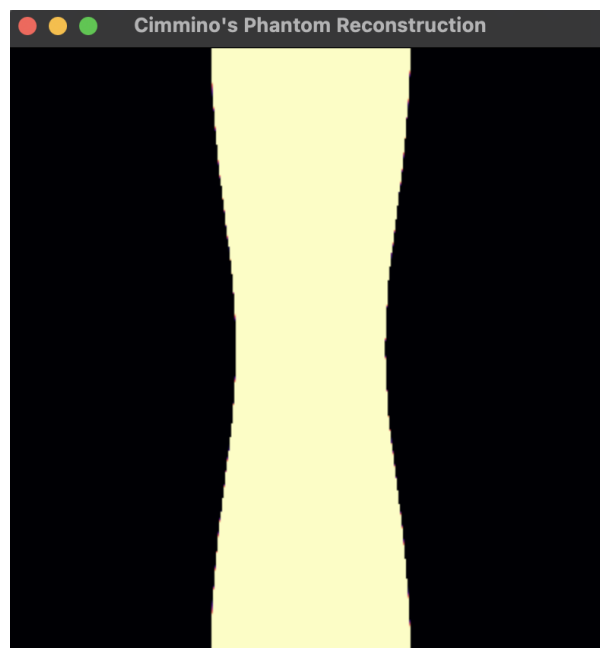

Figure 3: Non-normalised Sinogram (90 angles, 725 detectors)

Projection Matrix: A matrix that transforms the image space into the sinogram space based on the scanner geometry [6]. This matrix represents the system of equations we need to solve to reconstruct the image. In other words, it is A in the linear system $Ax = b$. The projection matrix is generated using Astra Toolbox in Python, which provides efficient methods for creating projection matrices based on specified scanner geometries [7]. This is then saved to a binary file as a sparse matrix in the Compressed Sparse Row (CSR) format to optimise memory usage and read into the Metal application.

Scanner Geometry: The configuration of the CT scanner, including the number of angles, the number of detectors, the detector spacing and total angle degree [8]. This defines how the projection matrix is constructed and influences the quality of the reconstruction. In this project, we set the scanner geometry parameters as follows:

- Number of angles: 90 (i.e., projections taken every 2 degrees over 180 degrees)

- Number of detectors: $\lceil 2 * \sqrt{2} * 256 \rceil = 725$ (to ensure full coverage of the image width)

- Detector spacing: 1 unit

- Total angle degree: 180 degrees

It's important to note, that the more angles and detectors we have, the better the reconstruction quality likely will be, but this also increases the computational load and memory requirements. For example, using the parameters we have chosen (90 angles and 725 detectors based on a 256x256 image) which are rather modest, the projection matrix becomes quite large ($256 \cdot 256 \times 90 \cdot 725 = 65536 \times 65250$), with 18737864 non-zero elements and ≈ 4.28 billion total elements if stored as a dense matrix. This is why we opted to use sparse matrix representation and parallel processing to handle the computations efficiently.

Cimmino's Algorithm Implementation: The core of the project is the implementation of Cimmino's algorithm in a parallel computing environment using Metal. The algorithm iteratively updates the image estimate by reflecting it across the hyperplanes defined by each equation in the linear system and averaging these reflections. The algorithm implementation is decomposed into two steps/kernels:

- **Reconstruction Kernel:** Each thread is responsible for one row of the projection matrix. It computes the dot product of the current estimate and the corresponding row of the projection matrix, subtracts this from the corresponding measurement in the sinogram to obtain the residual, and then computes that ray's contribution to the reflection. These results are stored in an update buffer.

- **Update Kernel:** This kernel reads the update buffer and adds the contributions from all rays to the current estimate, effectively averaging the reflections to produce the new estimate. Each thread is responsible for one pixel in the image.

Results

All programs are tested with the same parameters: 90 angles, 725 detectors and 256x256 image size and use the same projection matrix generated with Astra Toolbox. The sinogram produced in the Metal application is saved to a .bin file and loaded into other

applications for direct comparison. We are only comparing the reconstruction time (the reconstruction loop) to have a fair comparison between the different implementations - sequential, OpenMP and Metal. The time taken to generate the sinogram, load the projection matrix and render the images is not included in the comparison.

Let's have a look at the execution times for 10 - 2000 iterations. The times are averaged over multiple runs. We have included the final error (norm of the difference between the original phantom and the reconstructed image) for all iteration counts and programs to give an idea of the reconstruction quality and ensure each program is working correctly and consistently. We expect that each implementation will produce the same error norm for a given number of iterations.

The error norm is calculated as follows:

$$\text{Error Norm} = \|x_{\text{reconstructed}} - x_{\text{original}}\|_2^2 \tag{5}$$

	Sequential_Time	OpenMP_Time	Metal_Time	Sequential_Norm	OpenMP_Norm	Metal_Norm
NumIterations						
10	2247.61	671.86	71.25	61.05	61.05	61.05
100	24452.89	7492.12	473.54	51.16	51.16	51.16
500	127594.17	36421.52	2185.59	41.93	41.93	41.93
1000	200628.20	75266.26	4309.20	36.64	36.64	36.64
2000	513818.00	163745.00	8963.58	30.65	30.65	30.65

Figure 4: Reconstruction Times for Different Implementations and Final Error Norms

Firstly, we can see that all implementations produce the same final error norm, which indicates that they are all functioning correctly and consistently. The error norm is quite high, but this is expected given the limited number of angles and detectors used in the scan. Increasing these parameters would likely improve the reconstruction quality but also increase the computational load, though, we still decrease from 61 to 30 with 2000 iterations.

Secondly, we can see huge performance improvements using GPU computing with Metal. This is further reflected in the following speedup table.

	NumIterations	Sequential_Time	OpenMP_Time	Metal_Time	Speedup_OMP_vs_Seq	Speedup_Metal_vs_Seq
0	10	2247.61	671.86	71.25	3.35	31.55
1	100	24452.89	7492.12	473.54	3.26	51.64
2	500	127594.17	36421.52	2185.59	3.50	58.38
3	1000	200628.20	75266.26	4309.20	2.67	46.56
4	2000	513818.00	163745.00	8963.58	3.14	57.32

Figure 5: Speedup of OpenMP and Metal Implementations Compared to Sequential

The OpenMP program, though simple, still offers a decent speedup ($\approx$ 3x) over the sequential version. However, for image reconstruction this is not sufficient. We are only performing 2000 iterations and are yet to achieve a good reconstruction. Therefore, the Metal implementation is the most impressive, achieving a speedup of over 50x

compared to the sequential version for 2000 iterations. This demonstrates the power of parallel computing on GPUs for computationally intensive tasks like image reconstruction though, it's important to note that we would likely need significantly more iterations to achieve a high-quality reconstruction, especially since the error norm reduces more slowly with more iterations. Thus, this eseentially eliminates the use case of the sequential and OpenMP programs as they would take too long to be practical.

Comparing the Reconstructed Images

Finally, let's compare the reconstructed images after each iteration count. Since we have already established that all implementations produce the same results (same error norm), we will only show the Metal reconstructions here. Recall that we begin with an initial guess of a zero image (all black) and iteratively improve it. The final image on the bottom right is the original Shepp-Logan phantom for reference.
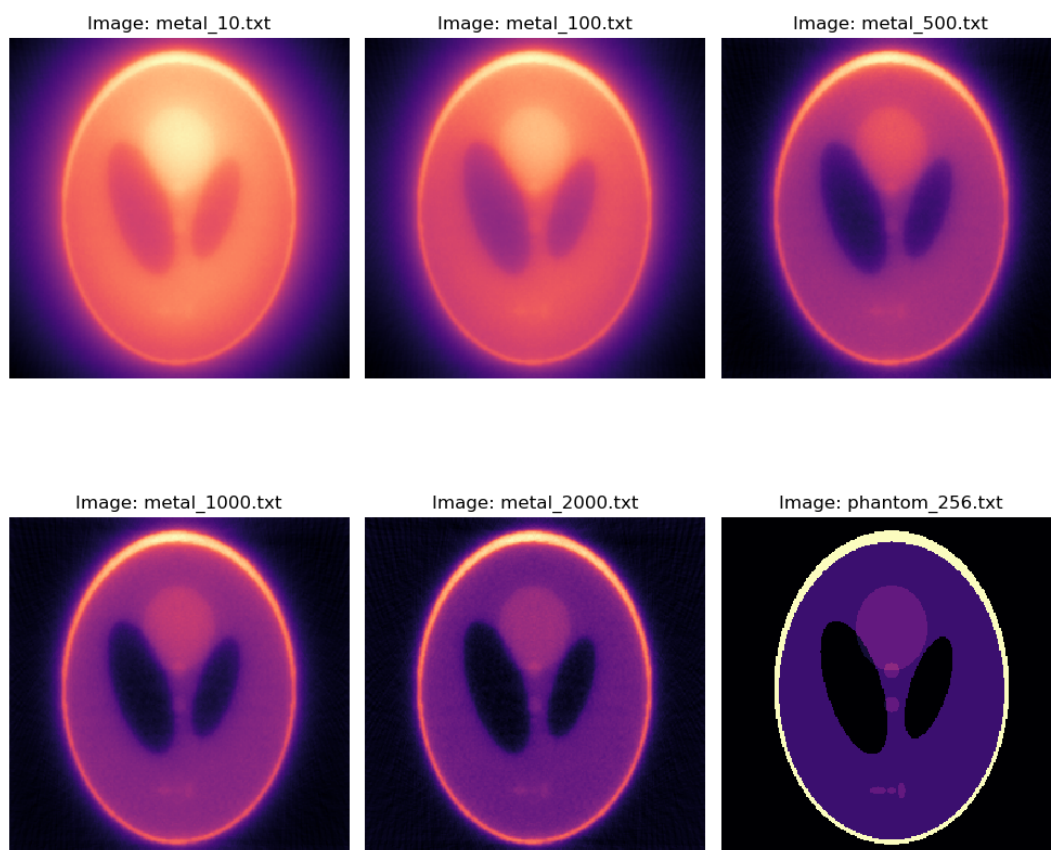

Figure 6: Reconstructed Images after Different Iteration Counts (Metal Implementation)

Even with 10 iterations, we can already see some structure of the original image. As the number of iterations increases, the reconstruction quality improves, with more details becoming visible. However, even after 2000 iterations, the reconstruction is still quite rough and lacks fine details. This is likely due to the limited number of angles and detectors used in the scan, as well as the inherent limitations of our simplified implementation. Regardless, the results are quite impressive.

Out of interest and without comparing execution times to the sequential and OpenMP programs (since they would take too long), we also ran the Metal implementation for 4000 and 10000 iterations. The results are shown below.

4000 iterations: 18060.5 ms with error norm 24.7619

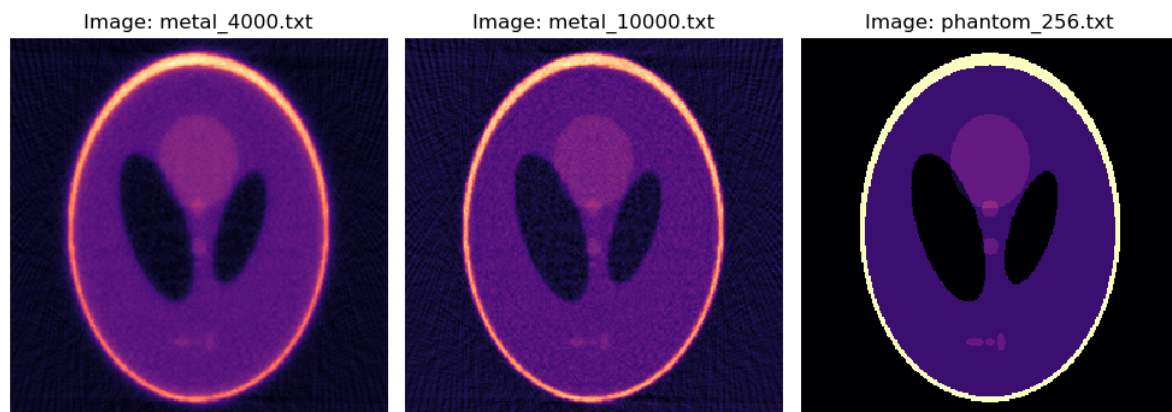10000 iterations: 44254.2 ms ms with error norm 20.2173

Figure 7: Reconstructed Images after 4000 and 10000 Iterations (Metal Implementation)

We can see some sharper lines and more details in the 10000 iteration reconstruction, but it is still quite far from the original image. This further emphasises the need for efficient computing for image reconstruction tasks, as achieving high-quality results may require a large number of iterations and more memory-intensive operations.

In practice, CT reconstructions often require many more iterations and more sophisticated techniques to achieve high-quality results. However, this project demonstrates the feasibility of using Cimmino's algorithm with GPU parallelism for image reconstruction tasks.

References

[1] S. Petra, C. Popa, and C. Schnörr, "Extended and constrained cimmino-type algorithms with applications in tomographic image reconstruction," Nov. 2008. DOI: 10.11588/heidok.00008798.

[2] NIST, *What are imaging phantoms?* 2024. [Online]. Available: https://www.nist.gov/health/what-are-imaging-phantoms#:~:text=In%20the%20biomedical%20research%20community,human%20body%20are%20operating%20correctly..

[3] MathWorks. "2D Data Objects." (), [Online]. Available: https://astra-toolbox.com/docs/data2d.html#shepp-logan.

[4] T. Peters, *Ct image reconstruction*, 2002. [Online]. Available: https://www.aapm.org/meetings/02am/pdf/8372-23331.pdf.

[5]   Apple. "Metal shading language specification." (2025), [Online]. Available: `https://developer.apple.com/metal/Metal-Shading-Language-Specification.pdf`.

[6]   X. Zhou, Q. Xu, and C. Wei, "Projection matrix based iterative reconstruction algorithm for robotic ct," *IEEE Access*, vol. 11, pp. 37 525–37 534, 2023. DOI: `10.1109/ACCESS.2023.3266989`.

[7]   A. Skorikov. "S009_projection_matrix.py." (2025), [Online]. Available: `https://github.com/astra-toolbox/astra-toolbox/blob/master/samples/python/s009_projection_matrix.py`.

[8]   scikit-image, *Radon transform*. [Online]. Available: `https://scikit-image.org/docs/stable/auto_examples/transform/plot_radon_transform.html`.