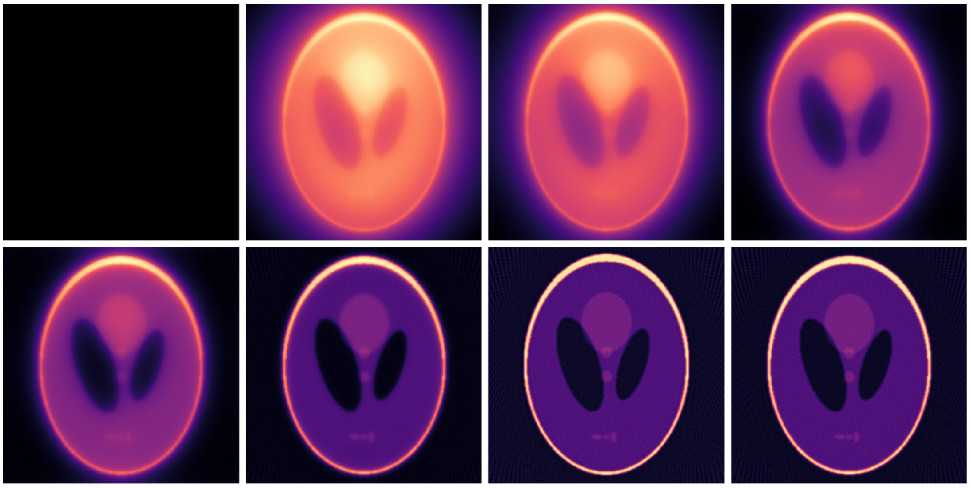
Image Reconstruction with Cimmino's Algorithm

Task M4.T1D - Project

Computed Tomography (CT) Scan and Reconstruction of the Shepp-Logan Phantom using Metal/C++



Kate Suraev (s224854029)

October 5, 2025

Contents

1 Introduction **2**

2 Project Structure **2**

3 Cimmino's Algorithm Overview **3**

4 Implementation Details **4**
4.1 Cimmino's Algorithm Implementation 7

5 Results **7**
5.1 Relative Error Norms and Execution Times 8
5.2 Comparing the Reconstructions 10
5.3 Sinogram Normalisation Results 11
5.4 Sinogram Computation Results 12

6 Final Thoughts **13**

Please see the GitHub repository for the complete code, as I am only able to upload one code file to OnTrack, and this does not reflect the application in its entirety. I have added my tutor, Daniel, and the unit chair, Maksym, to the repository as collaborators. This project is based on the SIT292 HD report I wrote this trimester, with permission from Maksym. The report is included in the repository for reference and focuses on the theoretical analysis of Cimmino's algorithm, proving its convergence as $k \to \infty$, and characterising the limit point.

1 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 and sparse 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 or an in-depth insight into CT reconstruction techniques, 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 parallelisation, and we also provide sequential and OpenMP implementations for comparison. The focus of this project is on the Metal program; therefore, the other implementations are kept simple.

2 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 perform the computations required by Cimmino's algorithm efficiently. The source files are located in the `metal-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 and phantom files 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.

3 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 weighted average of these reflections. The result is a convergent sequence of approximations that approaches either a solution of a consistent system or the weighted least-squares solution in the case of an inconsistent system [1]. Mathematically, Cimmino's algorithm is guaranteed to converge for any initial approximation x^0. This is rigorously proven in the accompanying SIT292 HD report.

Given an $m \times n$ matrix A with rows A_i, a vector $b \in \mathbb{R}^m$ and non-negative 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 non-negative 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:

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

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:

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

To explicitly measure the reconstruction quality and check for convergence, we compute the relative error norm E between the current approximation and the phantom P (original image) every 50 iterations. The relative error norm is defined as:

$$E = \frac{\|x^k - P\|_2^2}{\|P\|_2^2} \tag{5}$$

The convergence criteria are set to a relative error norm of less than 10^{-2} or until the maximum number of iterations is reached, as specified by the user. Mathematically,

$$E < 10^{-2} \quad \text{or} \quad k = \text{max\_iterations}$$

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 phantom to obtain a sinogram (a vector of measurements), and then using Cimmino's algorithm to reconstruct the original image from the sinogram.

For this image reconstruction problem, we use the well-known Shepp-Logan phantom as our test image. The Shepp-Logan phantom is a commonly used synthetic image that models the cross-section of a human head and is widely employed in the field of computed tomography (CT) for testing and evaluating reconstruction algorithms [2]. Our objective is to reconstruct this phantom image from its projections (sinogram) using Cimmino's algorithm. The projections are obtained by simulating the passage of X-rays through the phantom at various angles [3].

Phantom (P): A phantom is a standard test image comprising various shapes and intensities, used in medical imaging to model the human body [4]. The Shepp-Logan phantom is our ground truth image, against which we aim to reconstruct and measure the quality of our reconstruction. The 256x256 Shepp-Logan phantom is generated in Python using ASTRA-Toolbox [5].

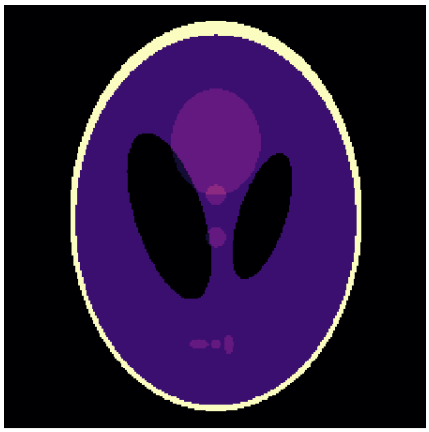


Figure 1: Shepp-Logan Phantom (256x256)

Projection Matrix (A): A matrix that transforms the image space into the projection 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]. In other words, it is complex mathematics that I am not confident in implementing myself. The projection matrix 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, as well as the other implementations.

Sinogram (b): A vector of measured projections obtained by simulating the CT scan of the phantom [8]. This can be thought of as b in the linear system $Ax = b$. The sinogram is computed by projecting it using the projection matrix A, i.e. $b = AP$, where P is the vectorised phantom image. To achieve better performance, a vectorised loop is used in the Metal kernel, so four elements are processed per thread at a time. This is computed in a Metal kernel function with the following thread configuration: Grid Size: (65536, 1), threadgroup Size: (1024, 1), number of threadgroups: 64.

As shown below, the sinogram does not resemble the original phantom, but it contains all the necessary information to reconstruct it using the projection matrix.

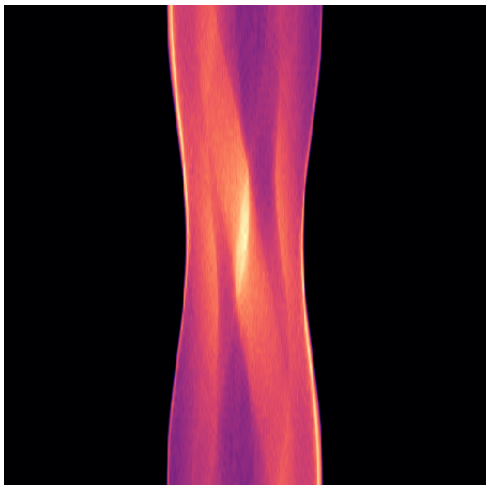


Figure 2: Sinogram (90 angles, 725 detectors)

We also use Metal kernel functions to normalise the sinogram texture values to a [0,1] range for better visualisation in the render window after computation. For this, a two-step process is used:

> **Step 1:** Find the maximum value in the sinogram. The `findMaxInTexture` kernel is designed to find the maximum value in a 2D texture (inputTexture) using atomic operations. This approach eliminates the need for a multi-step reduction process (e.g., partial reductions within threadgroups and a CPU-based final reduction). Instead, the kernel directly updates a single global maximum value stored in a device atomic_uint buffer. The thread configuration for our problem size in this kernel is as follows: Grid size: (725, 90, 1), thread group size: (32, 32, 1), number of thread groups: (23, 3, 1), total thread groups: 69.

Step 2: Normalise the sinogram using the maximum value found in the previous steps. The `normaliseKernel` divides each element in the sinogram by the maximum value to scale it to the [0,1] range. The thread configuration for our problem size in this kernel is as follows: Grid size: (725, 90, 1), thread group size: (32, 32, 1), number of thread groups: (23, 3, 1), total thread groups: 69.

Normalisation of the sinogram is not necessary for computation and is only performed for better visualisation in the render window, as well as to demonstrate other Metal/GPU computation capabilities. In addition, due to our small problem size, this normalisation step is quite efficient on CPU already and does not necessarily benefit from GPU parallelisation for a size of 90x725. However, we consider that this could be useful for larger-scale problems where the sinogram size is much larger and test this separately from the main reconstruction algorithm with larger sinogram sizes in Section 5.3. Since the normalisation functions dynamically assign thread groups based on the input texture size, they can handle larger sinograms or images without any changes.

Without normalisation, we are unable 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.

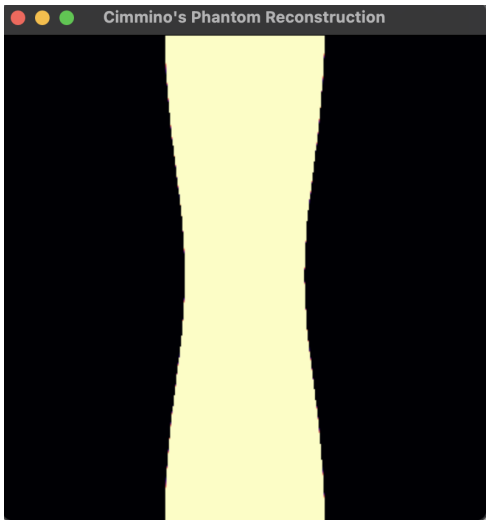


Figure 3: Non-normalised Sinogram (90 angles, 725 detectors)

Scanner Geometry: The configuration of the CT scanner, including the number of angles, the number of detectors, the detector spacing and total angle degree [9]. 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) [10]

Detector spacing: 1 unit

Total angle degree: 180 degrees

It is 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 relatively 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.

4.1 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 three 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. The thread configuration for this kernel is as follows: Grid Size: (65536, 1), thread group size: (1024, 1), number of thread groups: 64.

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. In this kernel, the relative update norm is also partially computed - specifically, the difference between the approximation and the phantom and the phantom norm. The thread configuration for this kernel is as follows: Grid Size: (65536, 1), threadgroup Size: (1024, 1), number of threadgroups: 64.

Relative Error Norm/Convergence Check: Every 50 iterations, the relative error norm E is calculated. The difference between each element of the current estimate and the phantom is squared and summed up in a Metal kernel with the following thread configuration: Grid Size: (65536, 1), threadgroup Size: (1024, 1), number of threadgroups: 64. The phantom norm is precomputed on the CPU since it does not change during the iterations. The relative error norm is then computed on the CPU using the differences and checked against the convergence criteria.

5 Results

We have three implementations of Cimmino's algorithm for comparison: a sequential C++ implementation, an OpenMP parallelised C++ implementation, and the main Metal implementation. All programs are run on a MacBook Air with an Apple M3 chip (8-core CPU, 10-core GPU, 8GB unified memory).

All programs are tested with the same parameters: 90 angles, 725 detectors, and a 256x256 image size, and use the same projection matrix generated with ASTRA-Toolbox.

The sinogram produced in the Metal application is saved to a .txt file and loaded into other applications for direct comparison. We are specifically looking at the computation time of the reconstruction algorithm itself as detailed in Section 4.1. The sequential and OpenMP implementations mimic the same steps as the Metal implementation, but without the GPU acceleration.

We are only comparing the reconstruction time (the reconstruction loop including reconstruction, update and convergence check) to have a fair comparison between the different programs. The time taken to generate the sinogram, load the projection matrix and render the images is not included in this comparison. Furthermore, while we should consider the total time taken for the Metal implementation (including data transfer to/from the GPU), even in our case, this time is negligible ($\approx 10 - 15$ms).

Let's take a look at the execution times for 10 to 1000 iterations. The times are averaged over multiple runs. We have included the final relative error norm E for all iteration counts and programs to give an analytical measure of the reconstruction quality and ensure each program is working correctly and consistently. We expect that each implementation will produce the same relative error norm for a given number of iterations.

All results are compiled and tabulated in Python (`plots.ipynb`) from the log files generated by each program.

5.1 Relative Error Norms and Execution Times

The key observation here is the consistent relative error norms across all implementations, indicating that they are all functioning correctly and producing the same results. This is important for validating the correctness of the parallel implementations against the sequential baseline - often parallel implementations can be deceivingly fast simply because they are not producing the correct results, and visually inspecting the reconstructed images may not always reveal subtle differences. We can confirm that for the same number of iterations, all implementations yield the same relative error norm, which is a good indication that they are all functioning consistently.

Iterations	Seq. (ms)	OMP (ms)	Metal (ms)	Relative Error Norm		
				Seq.	OMP	Metal
1	126.711	71.500	13.193	0.996	0.996	0.996
10	1231.700	732.940	67.532	0.965	0.965	0.965
100	20751.050	7805.349	546.398	0.808	0.808	0.808
500	129114.500	39806.083	2400.600	0.661	0.661	0.661
1000	226751.500	88153.525	4824.020	0.576	0.576	0.576

Figure 4: Relative Error Norms for Different Implementations up to 1000 Iterations

The relative error norm is relatively high, but this is expected given the limited number of angles and detectors used in the scan, as well as the lack of preconditioning, regularisation and other advanced techniques that are often employed in practical CT reconstruction scenarios [3].

Increasing the number of angles and detectors would likely improve the reconstruction quality but also increase the computational load, memory requirements and execution times. However, a decrease in the relative error norm is observed with an increasing number of iterations, indicating that the algorithm is converging towards a better approximation of the original image.

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

Iterations	Seq. (ms)	OMP (ms)	Metal (ms)	Speedup		
				OMP vs Seq	Metal vs Seq	Metal vs OMP
1	126.711	71.500	13.193	1.772	9.604	5.419
10	2154.880	735.737	67.532	2.929	31.909	10.895
100	20751.050	7805.349	532.843	2.659	38.944	14.648
500	129114.500	39806.083	2368.883	3.244	54.504	16.804
1000	226751.500	88153.525	4824.020	2.572	47.005	18.274

Figure 5: Execution Times for Different Implementations up to 1000 Iterations

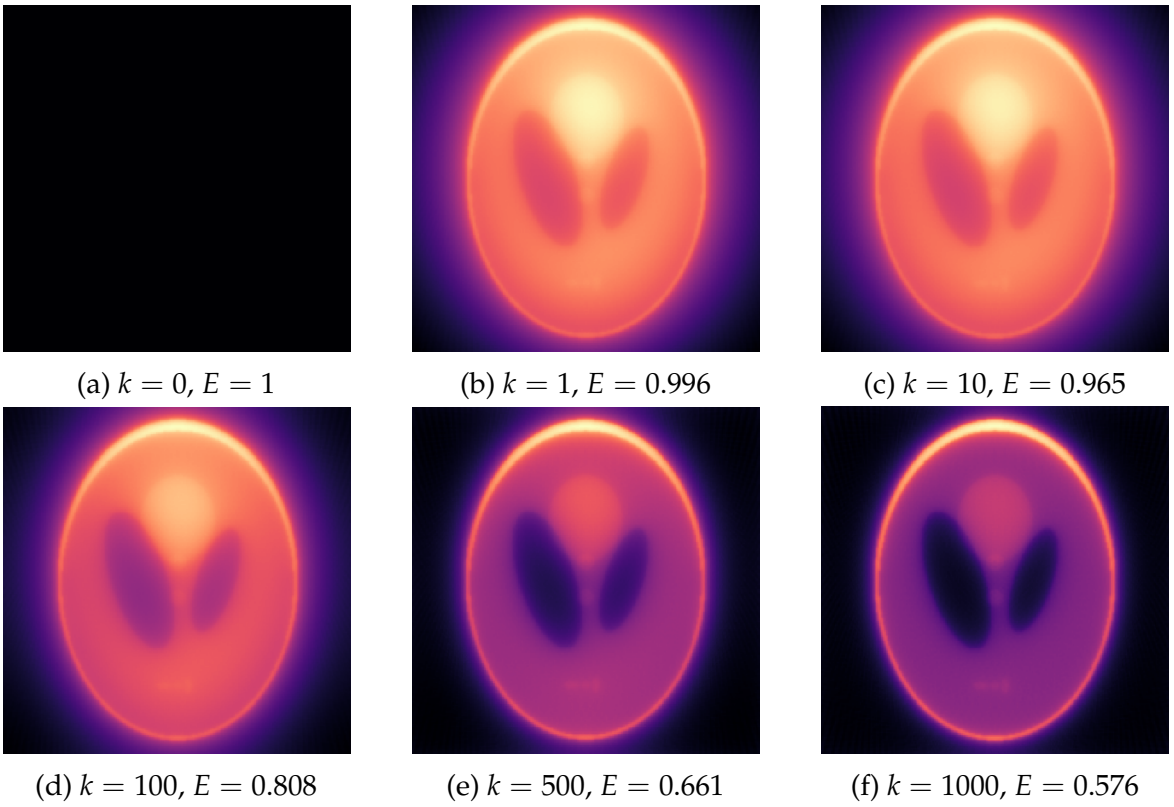
The OpenMP program, though simple, still offers a decent speedup ($\approx$ 3x) over the sequential version. However, for larger-scale image reconstruction problems, this is not sufficient. We are only performing 1000 iterations and have yet to achieve a good reconstruction in terms of the relative error norm. Therefore, the Metal implementation is the most impressive, achieving a speedup of over 50x compared to the sequential version and an 18x speedup compared to the OpenMP version for 1000 iterations.

This demonstrates the power of parallel computing on GPUs for computationally intensive tasks, such as image reconstruction. However, it is 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 additional iterations. Even with only 1000 iterations, the sequential and OpenMP programs become impractical, taking over 3.5 minutes and 1.4 minutes, respectively.

Thus, this essentially eliminates the use case of the sequential and OpenMP programs for larger-scale image reconstruction.

5.2 Comparing the Reconstructions

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.



(a) $k = 0$, $E = 1$ (b) $k = 1$, $E = 0.996$ (c) $k = 10$, $E = 0.965$

(d) $k = 100$, $E = 0.808$ (e) $k = 500$, $E = 0.661$ (f) $k = 1000$, $E = 0.576$

Figure 6: Reconstructed Shepp-Logan phantoms using Cimmino's algorithm in C++/Metal for $k = 0, 1, 10, 100, 500, 1000$ iterations with relative error norms E.

Even with 1 iteration, we can already see some structure of the original image, and the phantom is identifiable. As the number of iterations increases, the reconstruction quality improves, with more details becoming visible. However, even with 1000 iterations, the reconstruction is still quite rough and lacks fine details. Regardless, the results are quite impressive.

Out of interest, and without comparing execution times to the sequential and OpenMP programs (since this would take too long), we also ran the Metal implementation for $k = 5000, 10000, 15000$, and 20000 iterations to see how the reconstruction quality improves with more iterations.

(a) $k = 5000$, $E = 0.337$, 23092.4 ms

(b) $k = 10000$, $E = 0.254$, 47546.7 ms

(c) $k = 15000$, $E = 0.222$, 74106.9 ms

(d) $k = 20000$, $E = 0.208$, 97672.7 ms

(e) Original Phantom

Figure 7: Reconstructed Shepp-Logan phantoms for $k = 5000, 10000, 15000$ and 20000 iterations with relative error norms E and execution times (ms). Original phantom shown for reference.

We can see sharper lines and more details as the number of iterations increases, particularly in the 15,000 and 20,000 iterations, but of course, there is still a lack of fine details and clarity. 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.

This level of reconstruction may well be sufficient for specific applications, such as preliminary scans or scenarios where high precision is not critical. However, for medical imaging applications where accuracy is paramount, further improvements and more sophisticated techniques would be necessary.

5.3 Sinogram Normalisation Results

In this section, we discuss the execution time results of the sinogram normalisation process. This is tested in isolation from the main reconstruction algorithm to evaluate its performance on larger sinogram sizes. Matching the projection matrix size to these sinograms would be impractical for our hardware. Therefore, the sinograms are generated in Python using ASTRA-Toolbox (`sinogram_astra.py`) and saved to `.txt` files. The sinograms are then loaded into a separate Metal application that uses a class that inherits from the MTLComputeEngine class to perform the normalisation, `NormalisationProfiler`. A similar approach is used in a sequential C++ program and

an OpenMP parallelised C++ program for comparison. Each program was run multiple times, and the max value was checked to ensure consistency and correctness.

	Image Size	Angles	Detectors	Sinogram Size	Seq. (ms)	OMP (ms)	Metal (ms)	Speedup: OMP vs Seq.	Speedup: Metal vs Seq.	Speedup: Metal vs OMP.
0	256	90	725	65250	0.790	0.740	5.681	1.067	0.139	0.130
1	256	180	725	130500	1.318	1.001	4.762	1.317	0.277	0.210
2	256	360	725	261000	2.426	1.462	5.532	1.659	0.439	0.264
3	512	90	1449	130410	2.919	0.750	4.748	3.892	0.615	0.158
4	512	180	1449	260820	2.447	1.314	6.565	1.862	0.373	0.200
5	512	360	1449	521640	4.424	2.164	5.114	2.045	0.865	0.423
6	1024	180	2897	521460	4.407	2.424	5.112	1.818	0.862	0.474
7	1024	360	2897	1042920	8.730	4.590	5.011	1.902	1.742	0.916
8	2048	360	5793	2085480	17.563	9.284	6.859	1.892	2.561	1.354
9	2048	720	5793	4170960	35.163	18.899	6.845	1.861	5.137	2.761
10	4096	180	11586	2085480	17.687	8.976	7.932	1.970	2.230	1.132
11	4096	360	11586	4170960	35.496	17.010	7.184	2.087	4.941	2.368
12	4096	720	11586	8341920	70.646	34.998	19.366	2.019	3.648	1.807

Figure 8: Execution Times for Sinogram Normalisation using Sequential C++ and Metal Implementations

The results are far less dramatic than the reconstruction times, but we still see a notable speedup using Metal for larger sinogram sizes compared to the sequential and OpenMP programs. However, whether this speedup is worth the added complexity of using Metal is debatable. The sequential implementation is quite efficient for smaller sinograms and would be a suitable choice for our problem size (90x725 = 65,250 elements). However, as the size increases, the OpenMP and Metal implementations become more advantageous.

The Metal normalisation is left in the main Metal application, because it does not have a detrimental effect on the overall performance and could be helpful for larger sinograms or images and demonstrates how this could be achieved in Metal.

5.4 Sinogram Computation Results

The sinogram computation is performed in a Metal kernel function as described in Section 4. Again, due to the relatively small problem size, the performance gain is not as significant as the reconstruction times; however, we still observe a speedup using Metal compared to a sequential and OpenMP implementation. The following sinogram computation times are averaged over multiple runs, performed for various projection matrix sizes.

Image Size	Angles	Det.	Nonzero Elements	Seq. Scan (ms)	OMP Scan (ms)	Metal Scan (ms)	OMP vs Seq.	Metal vs Seq.	Metal vs OMP
256	90	725	18737864	58.115	33.145	29.437	1.753	1.974	1.126
512	90	1449	74912415	238.268	109.627	64.384	2.173	3.701	1.703
1024	90	2897	299563448	1467.300	763.292	412.117	1.922	3.560	1.852

Figure 9: Execution Times for Sinogram Computation ($A * P$) using Sequential C++, OpenMP and Metal Implementations

Again, for our problem size, the sequential implementation is quite efficient and would be a suitable choice; however, for larger problem sizes, we certainly see the benefits of parallelisation. The OpenMP implementation is quite efficient and would be a good compromise between complexity and performance for larger problem sizes, but the Metal implementation still outperforms it. One thing to note is that the projection matrix for the image size of 1024x1024 with 90 angles and 2897 detectors is $approx$2.4 GB, even stored as a sparse matrix binary file. This is quite large for our hardware and would likely be impractical for even larger problem sizes. However, this is a limitation of our hardware rather than the algorithm or implementation itself.

6 Final Thoughts

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.

The Metal implementation showcases significant performance improvements over traditional CPU-based approaches, making it a promising avenue for image reconstruction applications. We must consider that image reconstructions of this nature are generally not performed on consumer-grade hardware, but rather on high-performance workstations or clusters equipped with powerful GPUs and large amounts of memory. Consequently, our results will be limited.

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] H. Gach, C. Tanase, and F. Boada, "2d and 3d shepp-logan phantom standards for mri," in *19th International Conference on Systems Engineering*, Sep. 2008, pp. 521–526. DOI: 10.1109/ICSEng.2008.15.

[3] A. C. Kak and M. Slaney, "Algorithms for reconstruction with nondiffracting sources," in *Principles of Computerized Tomographic Imaging*, ch. 3, pp. 49–112. DOI: 10.1137/1.9780898719277.ch3.

[4] 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..

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

[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] T. Peters, *Ct image reconstruction*, 2002. [Online]. Available: https://www.aapm.org/meetings/02am/pdf/8372-23331.pdf.
[9] scikit-image, *Radon transform*. [Online]. Available: https://scikit-image.org/docs/stable/auto_examples/transform/plot_radon_transform.html.
[10] G. Mahmoudi, M. R. Ay, A. Rahmim, and H. Ghadiri, "Computationally efficient system matrix calculation techniques in computed tomography iterative reconstruction.," *J. Med. Signals Sens.*, vol. 10, 2020. DOI: 10.4103/jmss.JMSS_29_19.