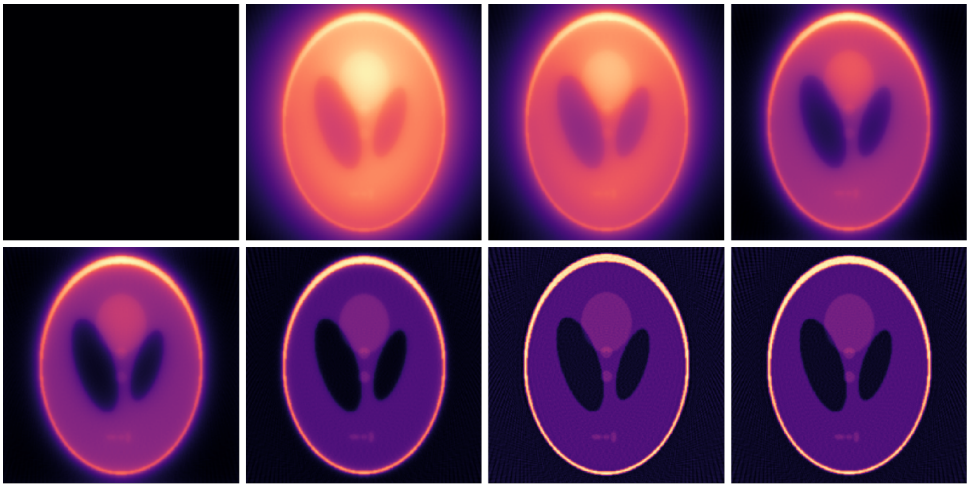
# Image Reconstruction with Cimmino's Algorithm

Task M4.T1D - Project

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



Kate Suraev (s224854029)

October 8, 2025

# Contents

*Please see the GitHub repository for the complete code. 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 and Simon (Unit chair for SIT292). 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 $\omega_i$ 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, $\omega_i$ 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].

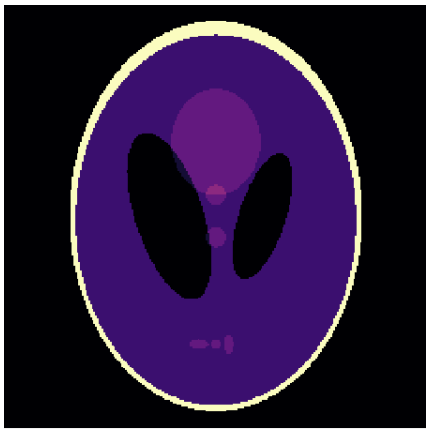


Figure 1: Shepp-Logan Phantom (256x256)

**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.

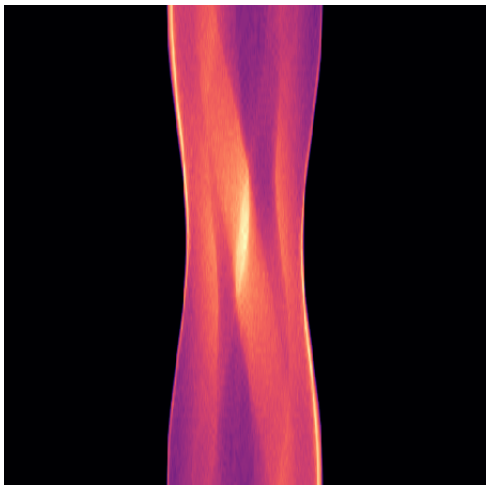


Figure 2: Sinogram (90 angles, 725 detectors)

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.

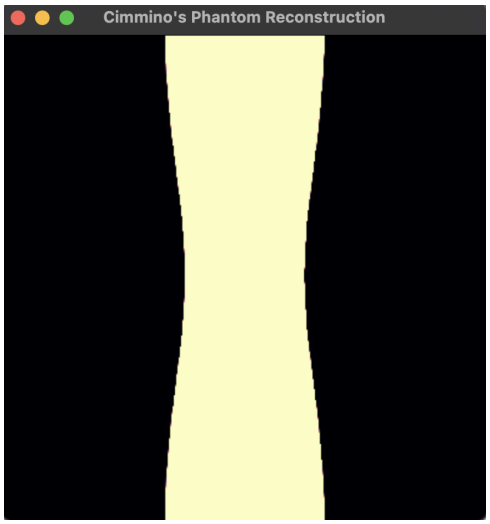



Figure 3: Non-normalised Sinogram (90 angles, 725 detectors)

**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 essential to note that the more angles and detectors we have, the better the reconstruction quality is likely to be; however, 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 $\approx 4.28$ 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. 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. 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 ensure 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

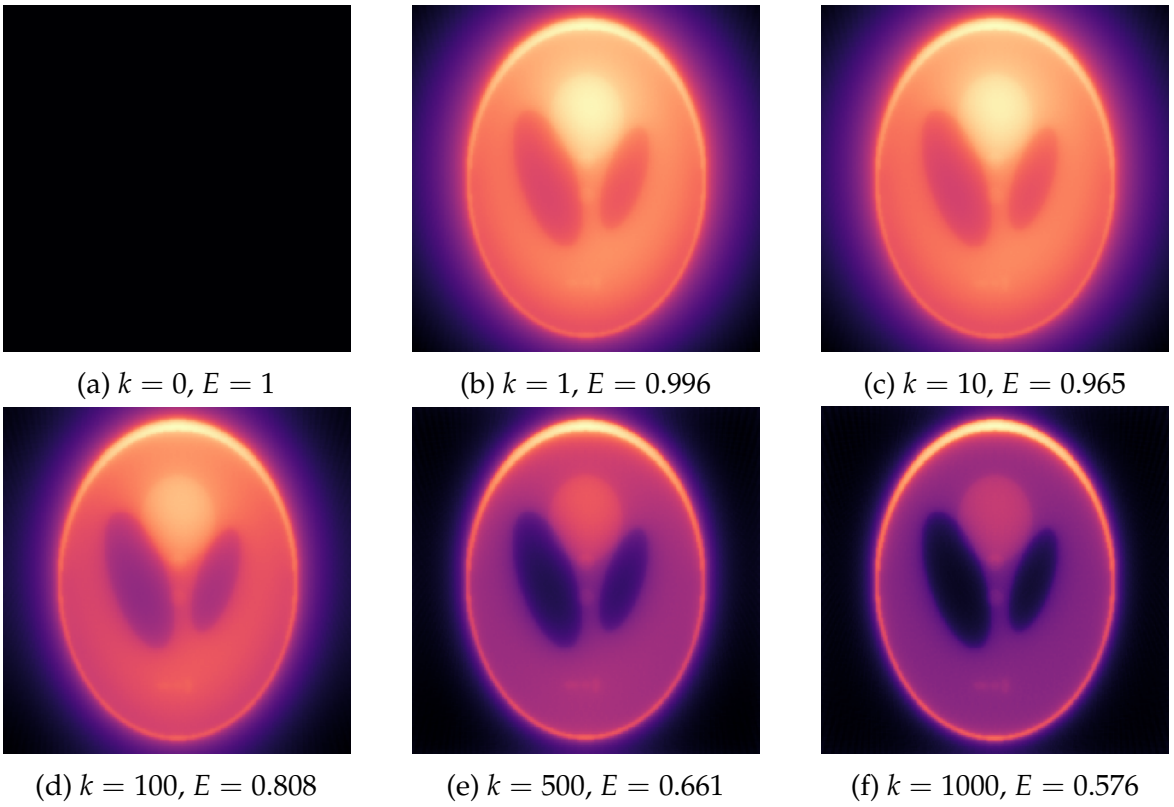
The OpenMP program, though simple, still offers a decent speedup ($\approx$ 3x) over the sequential version. However, for larger-scale image reconstruction problems, this is not sufficient. We are only performing 1000 iterations and have yet to achieve a good reconstruction in terms of the relative error norm. Therefore, the Metal implementation is the most impressive, achieving a speedup of $\approx$ 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 overall performance, could be helpful for larger sinograms or images, and demonstrates how this can be achieved in Metal.

## 5.4 Sinogram Computation Results

The sinogram computation is performed in a Metal kernel function. 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 fine 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 Testing Algorithm Accuracy

To ensure the correctness of the algorithms during development, a small test system is used with a known solution, computed and saved in `R`. `R` is a high-level language allowing us to directly use the definition of Cimmino's algorithm in Equation 4 to compute the solution for a specific number of iterations and the sinogram computation $b = AP$.

Since the sequential, OpenMP, and Metal implementations produce the same relative error norm, we know they are working consistently, so we mainly want to test that the Metal application is working correctly, since it is the most complex and has the highest potential for implementation errors.

A class `AlgorithmTester` is written as a child class of MTLComputeEngine to facilitate this testing. The sinogram computation and reconstruction algorithms are tested and compared against the solutions saved as `.txt` files. The test system is as follows:

**Projection matrix:** A projection matrix for image size 32x32, 90 angles and 91 detectors, generated using ASTRA-Toolbox in Python and saved as a sparse matrix binary file.

**Sinogram:** For the reconstruction test, a sinogram of size 90x91 is set to a vector of ones.

**Phantom:** For the reconstruction test, a phantom of size 32x32 is set to a vector of ones.

**Initial guess:** A zero image (all black) of size 32x32.

This class is mainly to ensure that changes to the main Metal application do not break the algorithm, particularly as we investigate optimisations and performance improvements.

The R code is as follows:

```
imageSize <- 32
angles <- 90
detectors <- 91

A <- load_sparse_matrix_binary("projection_32.bin")
P <- rep(1, imageSize * imageSize)
b <- rep(1, angles * detectors)
```

```r
x <- rep(0, imageSize * imageSize)

AP <- as.vector(A %*% P)
sinogramMatrix <- matrix(AP, nrow = angles, ncol = detectors, byrow = TRUE)
write.table(sinogramMatrix, file = paste0("sino_sol_", imageSize, ".txt"), row.names = FALSE, col.names = FALSE)

iterations <- 100
xnew <- cimminos(A, b, iterations, 1e-2, x)
xnew <- as.matrix(xnew)
write.table(xnew, file = paste0("solution_", iterations, ".txt"), row.names = FALSE, col.names = FALSE)
```

# 7 Overrelaxing Cimmino's Algorithm

To improve the convergence rate of Cimmino's algorithm, we experimented with overrelaxation. This involves introducing a relaxation parameter $\lambda$ into the update rule, modifying Equation 4 to:

$$x^{k+1} = x^k + \lambda \frac{2}{\omega} A^T (b - Ax^k) \quad \text{for } k = 0, 1, 2, \ldots \tag{6}$$

The question is: can we get the relative error norm to decrease faster with a suitable choice of $\lambda$?

Indeed, we can.

| | Iterations | 0.0 | 2.0 | 4.0 | 8.0 | 10.0 | 20.0 | 50.0 | 75.0 |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 100 | 0.808276 | 0.749635 | 0.713795 | 0.604739 | 0.575547 | 0.475902 | 0.337270 | 0.284308 |
| 1 | 500 | 0.661313 | 0.575818 | 0.519068 | 0.370266 | 0.337758 | 0.253989 | 0.200792 | 0.193961 |
| 2 | 1000 | 0.575851 | 0.476315 | 0.414078 | 0.277379 | 0.254043 | 0.207980 | 0.191972 | 0.190859 |
| 3 | 5000 | 0.337868 | 0.254086 | 0.222335 | 0.193385 | 0.191975 | 0.190538 | 0.190059 | 0.189898 |
| 4 | 10000 | 0.254091 | 0.208006 | 0.196955 | 0.190769 | 0.190539 | 0.190153 | 0.189791 | 0.189642 |

Figure 10: Relative Error Norms for Different Relaxation Parameters $\lambda$ using Metal with 90 Angles

Interestingly, we find that we can achieve essentially the same relative error norm with 500 iterations and $\lambda = 75.0$, as we can with 10000 iterations. This represents a significant improvement in convergence rate; however, it is specific to our particular problem and scanner geometry.

We can see that the relative error norm does not improve significantly beyond 1.898, and this remains true for higher values of $\lambda$ and larger $k$. This indicates that this is approximately the best reconstruction we can achieve with this scanner geometry. Needing a significant relaxation factor also tells us that the step size is relatively small, and the relaxation is compensating for this - not necessarily a bad thing, but something to consider. We can modify the algorithm to use a lower relaxation factor, but this introduces more computations with similar results to using a higher relaxation factor.

So, how can we achieve a better reconstruction?

The most straightforward way would be to increase the number of angles and detectors in the scan, which would provide more information for the reconstruction. Increasing the number of angles to 180 increases the projection matrix dimensions to 130,500 x 65,536 with 37,476,784 non-zero entries. Naturally, this increases the execution time; however, using the relaxation factor allows us to achieve better results in fewer iterations. In addition to the relaxation factor, the projection matrix is also normalised before computations.

| | Iterations | 0.0 | 20.0 | 50.0 | 75.0 | 100.0 | 150.0 | 175.0 | Time (ms) |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 100 | 0.808495 | 0.475955 | 0.335407 | 0.281439 | 0.245670 | 0.206905 | 0.210340 | 861.470375 |
| 1 | 500 | 0.661577 | 0.246116 | 0.173419 | 0.156764 | 0.147991 | 0.140167 | 0.138121 | 4279.275714 |
| 2 | 1000 | 0.576111 | 0.186324 | 0.148005 | 0.140376 | 0.136686 | 0.133764 | 0.133070 | 8568.631818 |
| 3 | 5000 | 0.336006 | 0.136691 | 0.131962 | 0.131091 | 0.130529 | 0.129693 | 0.129352 | 42017.342857 |

Figure 11: Relative Error Norms for Different Relaxation Parameters $\lambda$ using Metal with 180 Angles

We can see some improvement in the relative error norm with 180 angles, achieving a relative error norm of $\approx 0.129$ for 5000 iterations and $\lambda = 175.0$. We achieve a similar relative error norm of $\approx 0.133$ with 1000 iterations and $\lambda = 175.0$, but with a much lower execution time of $\approx 8.5$ seconds compared to $\approx 42$ seconds.

What about 256 angles? This yields a projection matrix of size 185,600 x 65536 with 53,301,791 non-zero entries.

| | Iterations | 0.0 | 200.0 | 210.0 | 215.0 | 220.0 | Time (ms) |
|---|---|---|---|---|---|---|---|
| 0 | 100 | 0.822784 | 0.186925 | 0.183508 | 0.184034 | 2.815880 | 1233.326667 |
| 1 | 500 | 0.659316 | 0.119339 | 0.118379 | 0.117935 | 2794.250000 | 6201.966667 |
| 2 | 1000 | 0.573908 | 0.109868 | 0.109460 | 0.109271 | 15602400.000000 | 12245.666667 |

Figure 12: Relative Error Norms for Different Relaxation Parameters $\lambda$ using Metal with 256 Angles

With 256 angles, we achieve a relative error norm of $\approx 0.109$ for 1000 iterations and $\lambda = 215.0$ with an execution time of $\approx 12.2$ seconds. Here, we also see the risk of overrelaxation, as evidenced by the significant divergence for $\lambda = 220.0$.

What about 360 angles? This gives us a projection matrix of size 261000 x 65536 with 74955609 non-zero entries.

| | Iterations | 0.0 | 200.0 | 210.0 | 215.0 | 220.0 | Time (ms) |
|---|---|---|---|---|---|---|---|
| 0 | 100 | 0.822772 | 0.186113 | 0.182624 | 0.183205 | 2.867880 | 1689.970000 |
| 1 | 500 | 0.659305 | 0.112066 | 0.110798 | 0.110205 | 3062.760000 | 8641.653333 |
| 2 | 1000 | 0.573893 | 0.098171 | 0.097449 | 0.097109 | 18744200.000000 | 16682.100000 |

Figure 13: Relative Error Norms for Different Relaxation Parameters $\lambda$ using Metal with 360 Angles

With 360 angles, we achieve a relative error norm of $\approx 0.097$ for 1000 iterations and $\lambda = 215.0$ with an execution time of $\approx 16.7$ seconds. Again, we see divergence for $\lambda = 220.0$.

This is a classic example of the trade-off between accuracy and performance. Yes, we have improved the reconstruction quality by increasing the number of angles, but this comes at the cost of increased execution time and memory usage.

### 7.0.1 Accuracy vs Speed Trade-off

From our results, we can see that improved accuracy comes at the cost of speed. This is more clearly visualised in the following plot. The best relaxation parameter for each number of angles is chosen based on the lowest relative error norm achieved for each iteration count and number of angles (since the relaxation factor does not affect execution time).

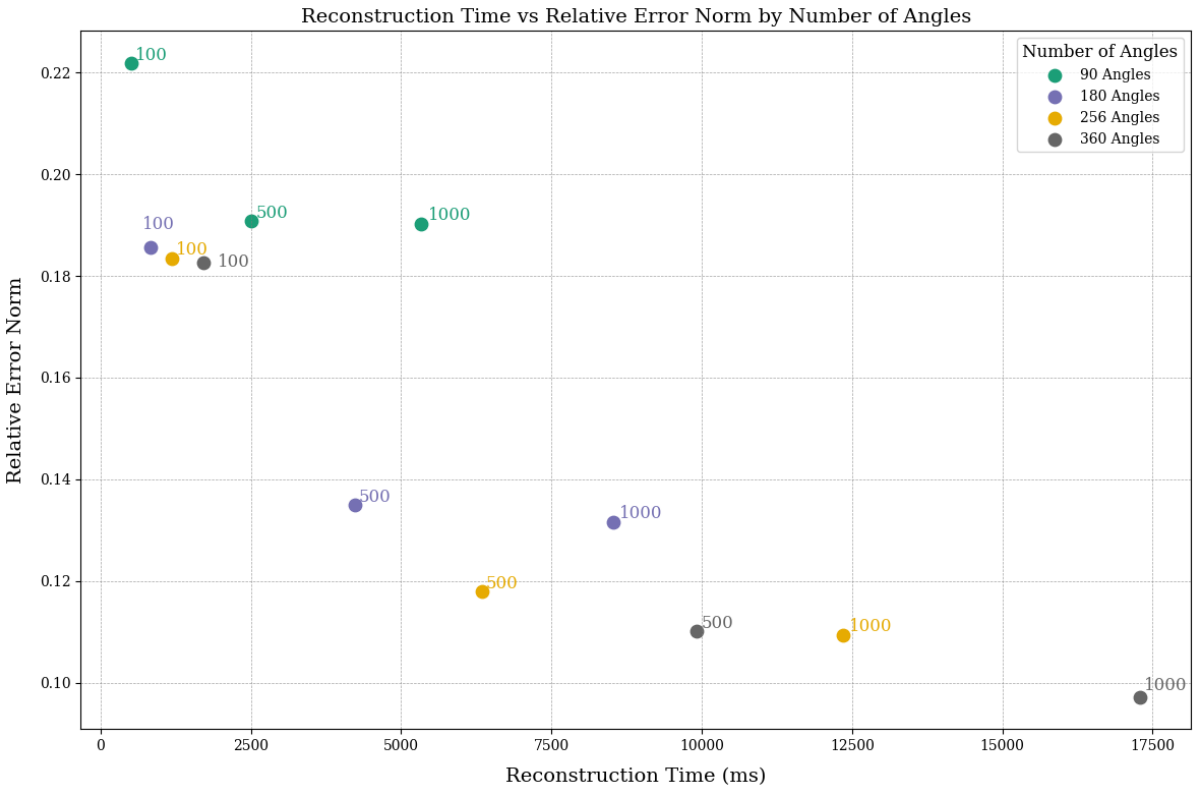



Figure 14: Accuracy vs Speed Trade-off for Different Numbers of Angles

The choice is ultimately up to the user and the specific application. In practice, you would likely need even better accuracy than what we have achieved here.

# 8 Different Projector Types

Now that we have shown we can achieve some reasonable results by incorporating a relaxation factor, normalising the projection matrix and increasing the number of angles, we can also investigate different projector types. Up until this point, we have

been using a 'strip' projector type. For 360 angles, the 'strip' type projection matrix size is 261,000 x 65536, with 74,955,609 non-zero entries. The strip projector type models the X-ray beam as a strip with a finite width, which is more realistic than a line but also more computationally intensive [11].

Now, we test the 'linear' projector type with 360 angles to see if we can further improve the reconstruction quality and perhaps achieve a better accuracy versus speed trade-off. For 360 angles and a 256x256 image size, the 'linear' type projection matrix size is 261,000 x 65536, with 42,482,426 non-zero entries. The linear type is a ray tracing approach that uses bilinear interpolation to compute the intersection of the rays with the image grid [11]. We can already see that the 'linear' type is more efficient in terms of memory usage, with significantly fewer non-zero entries. So, we expect to see improvements in execution time at the very least.

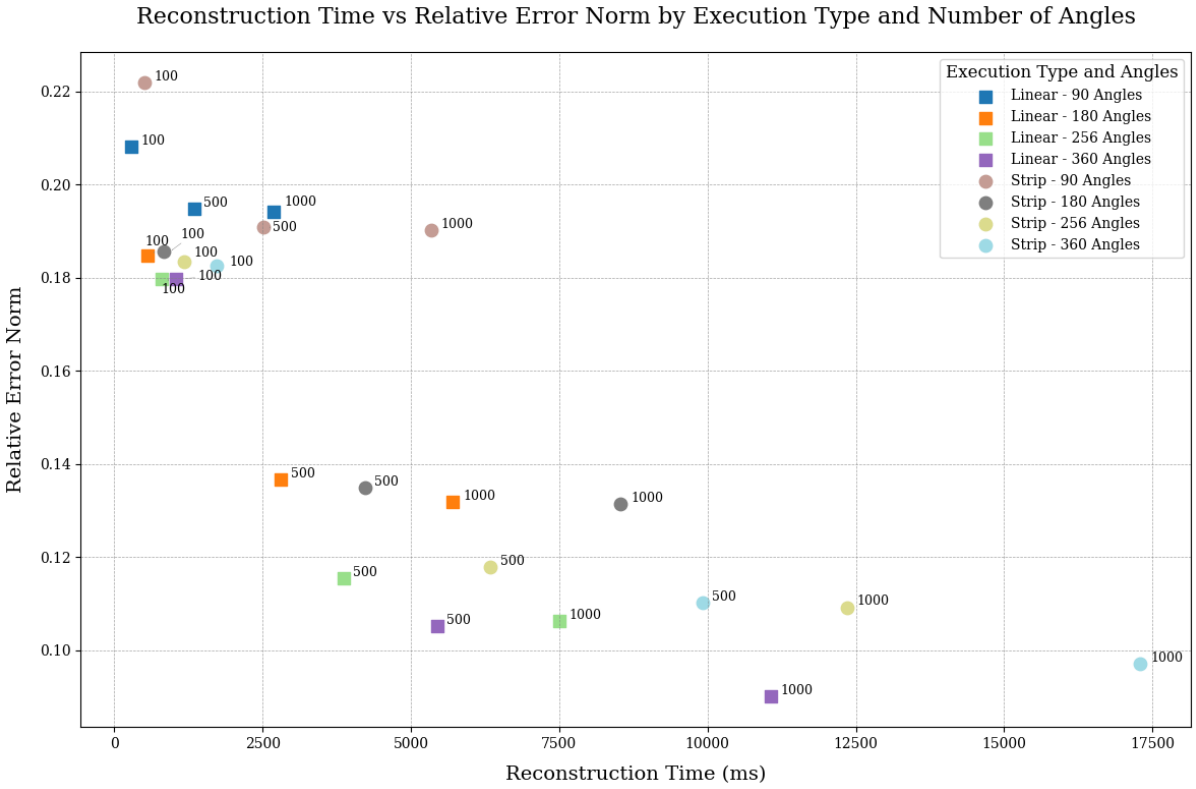



Figure 15: Relative error norms for different angles and iteration counts using the 'strip' and 'linear' projector types

Just by simply changing the projector type in the Python script, for 360 angles, we achieve a relative error norm of $\approx 0.09$ for 1000 iterations, with an execution time of $\approx 11$ seconds. This is a notable improvement over the 'strip' projector type, which achieved a relative error norm of $\approx 0.097$ for the same number of iterations but with a longer execution time of $\approx 16.7$ seconds. In fact, we see improvements across all angles and iteration counts.

## 8.1 Line Projector Type

Finally, we tested the 'line' projector type and found that it produced even better results. For 360 angles, the 'line' type projection matrix size is 261,000 x 65536, with 30,079,522 non-zero entries. So, even fewer non-zero entries than the 'linear' type, which we expect to translate to even better execution times. The 'line' projector type models the X-ray beam as an infinitely thin line [11].

For 1000 iterations, we achieved a relative error norm of 0.775 for $\lambda = 250.0$ with an execution time of $\approx 6.7$ seconds. This is our best result so far, achieving the lowest relative error norm with the shortest execution time. This behaviour is consistent across all angles and iteration counts.

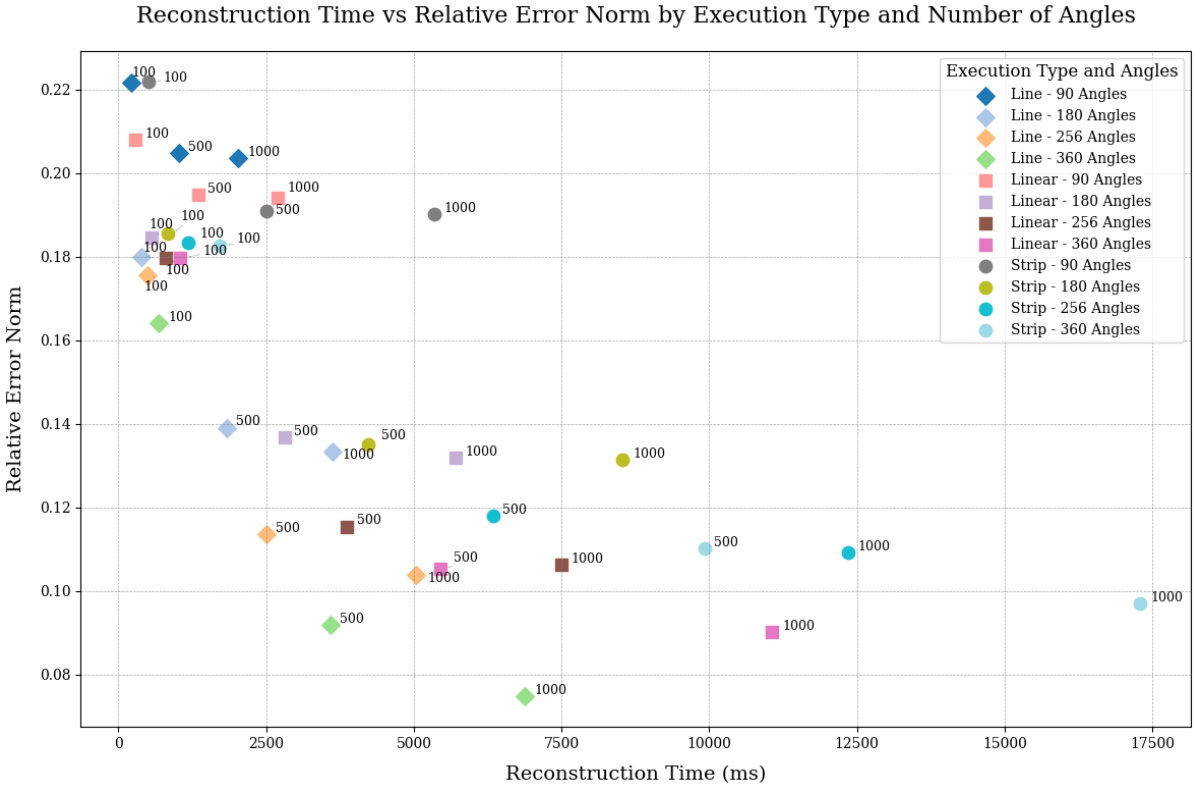



Figure 16: Relative Error Norms for different angles and iteration counts using the 'strip', 'linear', and 'line' projector types

# 9 Non-Negativity Constraint

To further improve reconstruction quality, we added a non-negativitiy constraint to the update kernel, preventing negative values in the image estimate. Examining the first few values in the reconstructed data, we observe some almost zero negative values, which are not physically meaningful in the context of CT imaging and are one of the reasons why our reconstructed images appear discoloured.

```
-8.25384e-06 0.000228852 0.00011652 -5.25288e-05
-9.2267e-05 0.000491378 0.000735923 0.000375223
-0.000538355 -4.74896e-05 0.00177381 0.000535843
```

```
-0.00091517 -0.00218309 0.000600351 -0.00134019
0.000860494 -0.00161219 0.00188096 0.00223338
```

We add a simple check in the update kernel to set any negative values to zero.

```
if (reconstructedBuffer[gid] + updateBuffer[gid] < 0.0f) {
    reconstructedBuffer[gid] = 0.0f;
} else {
    reconstructedBuffer[gid] += updateBuffer[gid];
}
```

We obtain yet another improvement in the relative error norm, achieving $\approx 0.0266$ for 1000 iterations, $\approx 0.0431$ for 500 iterations and $\approx 0.135$ for 100 iterations with the constrained 'line' projector type, 360 angles and $\lambda = 350$. The execution times remain the same since this is a very simple operation.

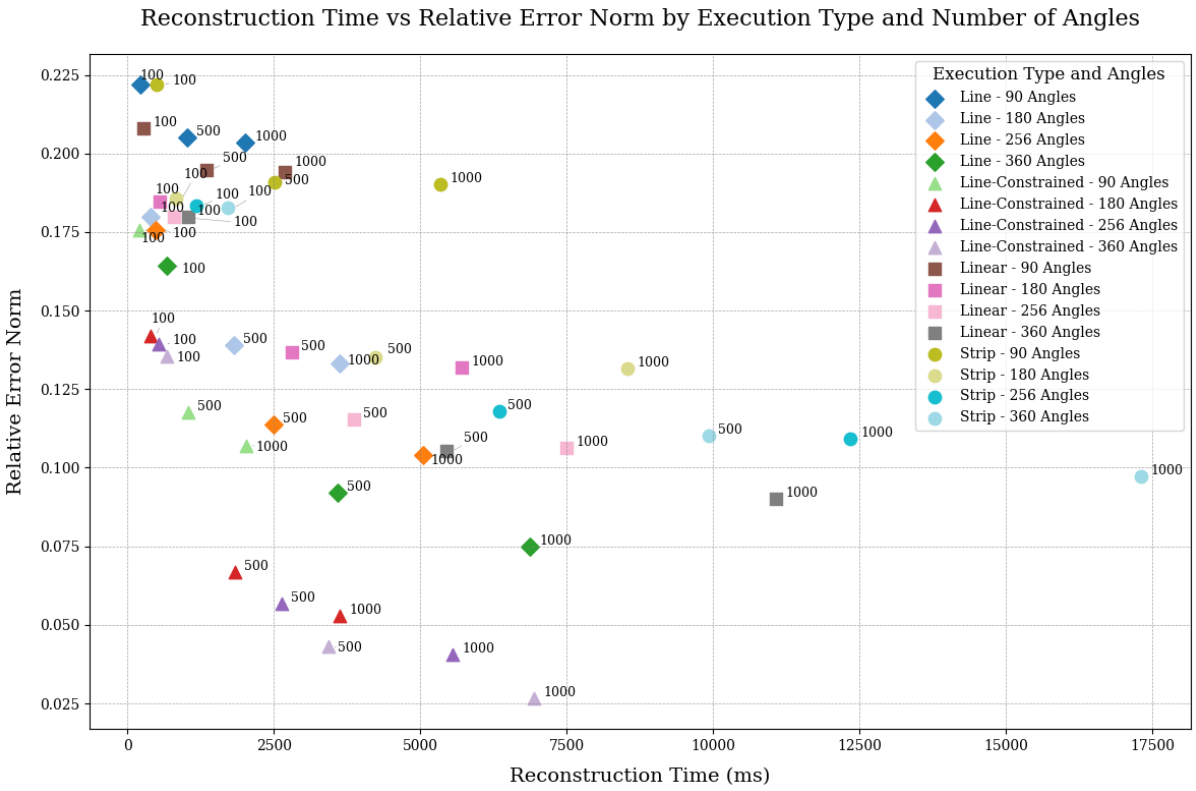



Figure 17: Relative Error Norms for different angles and iteration counts using the 'line' projector type with and without the non-negativity constraint

This gives us our best results yet, and we now have an even better accuracy versus speed trade-off.

For example, suppose we need a quick but accurate reconstruction. In that case, we can use the constrained 'line' projector type with 360 angles and 500 iterations, achieving a relative error norm of $\approx$ 0.0431 in $\approx$ 3.4 seconds. If we need a more accurate reconstruction, we can use the same projector type and number of angles but increase the iterations to 1000, achieving a relative error norm of $\approx$ 0.0266 in $\approx$ 6.7 seconds. If we need more speed over accuracy, we could opt for 180 angles and 500 iterations,

achieving a relative error norm of $\approx 0.0668$ in $\approx 1.8$ seconds or 360 angles and 100 iterations, achieving a relative error norm of $\approx 0.135$ in $\approx 0.7$ seconds.

## 10 Best Reconstructions

Finally, let's compare the best reconstructed images we achieved using the constrained 'line' projector type for 100, 500, and 1000 iterations with our previous best reconstruction using the 'strip' projector type for 20000 iterations.

(a) Prev. best $k = 20000$, $E = 0.208$, 97672.7 ms

(b) $k = 100$, $E = 0.135$ 695.135 ms

(c) $k = 500$, $E = 0.043$ 3422.26 ms

(d) $k = 1000$, $E = 0.027$ 6780.72 ms

(e) Original Phantom

Figure 18: New best reconstructed Shepp-Logan phantoms using Cimmino's algorithm in C++/Metal for $k = 100, 500, 1000$ iterations with relative error norms $E$ and execution times (ms). Using 360 angles and 'line' projector type in projection matrix. Original phantom and previous best reconstruction (20000 iterations) shown for reference.

Although the difference may be subtle from afar, we can see a much clearer, more defined, and smoother reconstruction with just 1000 iterations using the 'line' projector type, compared to our previous best reconstruction with 20000 iterations using the 'strip' projector type. The improvements are much more pronounced when zoomed in. In the previous best reconstruction with 20000 iterations, the rays are pretty visible, and the image is quite rough, with a noticeable colour discrepancy. Essentially, we have achieved a $\approx 87.02\%$ reduction in the relative error norm with 1000 iterations, accompanied by a $\approx 14.4\times$ speedup compared to our previous best result with 20000

iterations. We stress that this configuration works for this problem but may not be optimal for other problems or scanner geometries.

# 11 Final Thoughts

In practice, CT reconstructions often require more sophisticated techniques and preprocessing steps to achieve high-quality results for much larger-scale problems. However, this project demonstrates the feasibility of using Cimmino's algorithm with GPU parallelism for image reconstruction tasks. In particular, we demonstrated that parallelising the computation is only half the battle, and that manipulating the scanner geometry, projector type, and algorithm parameters can have a significant impact on reconstruction quality and performance.

The Metal implementation showcases significant performance improvements over traditional CPU-based approaches, making it a promising avenue for image reconstruction applications. 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, though nonetheless impressive, and we can confidently say that we have achieved our project goals, achieving a near perfect reconstruction in a matter of seconds on consumer-grade hardware.

# 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] ASTRA-Toolbox. "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.

[11] ASTRA-Toolbox. "2d projectors." (), [Online]. Available: https://astra-toolbox.com/docs/proj2d.html.