Hardware acceleration of artificial X-ray image generation

by

Per Knops

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Wednesday November 29, 2023 at 14:00.

Student number:	4603192	
Project duration:	November 15, 2022 – November 29, 2023	
Thesis committee:	Dr. ir. Z. Al-Ars,	TU Delft, supervisor
	Dr. ir. R. Remis,	TU Delft
	BSc. R. de Jong,	Philips Medical Systems

Abstract

X-ray imaging systems play an important role in the diagnostic process of various medical conditions. Generating an accurate artificial X-ray image has multiple advantages. It allows for flexible configurations during generation. The resulting images can reduce testing time and cost, help the training of surgeons, and increase the amount of data for artificial intelligence model training. The generation of an X-ray image involves the simulation of a raytracing algorithm through a data model. In this research, a naive approach to this problem is examined. It was found that this approach can be improved by implementing model parallelization, data caching, and data compression. The resulting algorithm is simulated and validated in a software environment. This is then implemented for both an Ultrascale+ and a Versal FPGA. The results show that the algorithm can achieve real-time X-ray image generation, matching the performance of currently used detectors, provided that the required memory performance is achieved.

Contents

1 Introduction

1.1 Context

The increase in demand for artificial intelligence (AI) is rampant. To train artificial intelligence models, lots of data is needed. Currently, real images make up a big part of that data. The use of synthetic images will grow larger over time. If the true issues of compliance, regulatory requirements, customer privacy, bias, and performance with AI models are addressed, businesses are going to need to rely on tools such as synthetic data generators [6].

Synthetic image generation is the process of creating visual content, including images and animations, using computer software and algorithms. It involves generating images entirely in a digital environment, as opposed to filming them with cameras or other detection devices. In the context of this research, the goal is to develop a synthetic image generation algorithm tailored for a Philips X-ray system.

This research is part of the TASTI (Application-TAilored SynThetic Image generation) project [17]. The TASTI project is an EU-funded initiative with the primary objective of creating a framework of transferable technology to accelerate the development of novel solutions for synthetic image generation, tailored to their applications. The project aims to cater to various industries, including the automotive sector, material production, agriculture, and healthcare.

For the medical application, three specific goals are defined:

- **Testing:** The primary goal is to develop a virtual simulation of X-ray systems for testing purposes. Real X-ray system testing is time-consuming and involves logistical challenges. By creating an accurate virtual representation of X-ray systems, the testing process can be greatly accelerated and made more efficient. This can have a substantial impact on product development and quality assurance.
- **Surgeon training:** Training surgeons traditionally involves the use of physical X-ray systems and patients. The goal here is to use the virtualization of X-ray systems to create a realistic training environment for surgeons. This would provide a safe and controlled setting for surgeons to practice their skills, reducing the need for a real system during training.
- **AI training:** Training AI models, particularly for medical image analysis, requires a large volume of high-quality data. Generating synthetic X-ray images that closely resemble real ones can increase the amount of data available for AI training. This can lead to more accurate and effective AI systems for medical diagnosis and other applications.

1.2 Problem definition

The Philips Azurion is a series of advanced medical imaging systems designed for use in interventional procedures and surgery. These systems are used in the field of cardiology, vascular surgery, and other interventional medical specialties. The Azurion platform represents a range of high-end medical devices and software solutions for image-guided procedures [14]. An example of the Azurion 7 series can be seen in Figure 1.

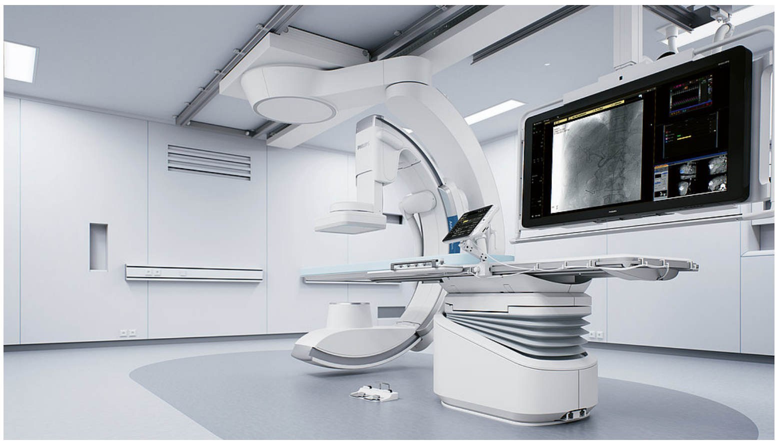


Figure 1: Philips Azurion 7 C20 with FlexArm

The goal of this research is to develop an efficient solution for generating synthetic X-ray images that closely mimic real-world X-ray images. The synthetic images should be close to indistinguishable from those produced by physical X-ray systems like the Azurion. They serve multiple purposes, including testing X-ray systems, training healthcare professionals, and training AI models for medical image analysis.

This project aims to develop a hardware-based algorithm for real-time X-ray image generation, and several key requirements for implementation have been established:

- **Image quality:** The algorithm should produce images that closely resemble real X-ray images in terms of quality. Even unwanted side effects of real X-ray images such as noise and scatter should be considered.
- **Position versatility:** The algorithm must be capable of generating images for all feasible positions of the X-ray system.
- **Large model support:** The algorithm should handle large 3D voxel models with dimensions of up to 1024 x 1024 x 1024 correctly.
- **Performance target:** The algorithm is expected to match the performance of existing detectors. For smaller detectors, up to 60 images per second should be generated. This can go down to 15 images per second for larger detectors.

Therefore the following research question is posed: Can a hardware algorithm for real-time X-ray image synthetization achieve the image generation rate of existing detectors? To address this question effectively, several fundamental sub-questions need to be answered:

- What are the primary bottlenecks and performance limitations in a naive implementation of the algorithm?
- What hardware-based solutions can be harnessed to overcome these bottlenecks?
- How can the initial naive algorithm be adapted to use the identified hardware solutions effectively?

1.3 Thesis outline

The rest of this thesis is organized into the following chapters, each contributing to the comprehensive understanding of the research. Chapter 2 offers essential background information that lays the foundation for comprehending the design choices and outcomes discussed throughout the thesis. Chapter 3 provides insight into the design process and considerations for the final algorithm. In Chapter 4, the details of the final algorithm, including its structure, components, and operational principles will be explained. The simulation and validation of this algorithm will be discussed in Chapter 5. Chapter 6 discusses the architectural and technological aspects of the hardware solution. In Chapter 7, the results will be discussed. Finally, Chapter 8 summarizes the achieved results and lists potential avenues for future investigations in the field.

2 Background

In this chapter, the required background knowledge is discussed. Section 2.1 describes an X-ray system. In Section 2.2 the mass attenuation coefficient is explained. Section 2.3 examines the functionality of FPGAs. Finally Section 2.4 discusses the related works.

2.1 X-ray system

The Azurion system is a family of X-ray systems. These systems are used by healthcare professionals to visualize the internal structures of the human body. It operates on the principles of X-ray attenuation of a patient. A small overview of an X-ray system is shown in Figure 2.

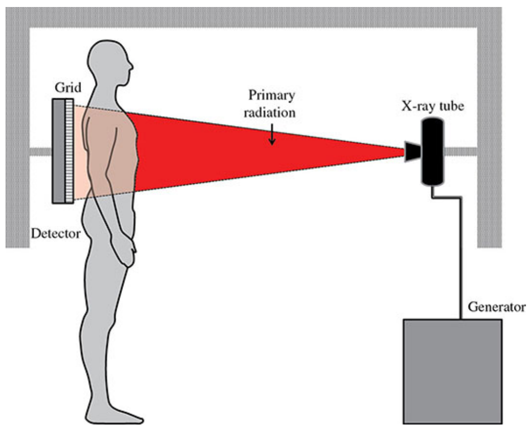


Figure 2: Overview of an X-ray system [15]

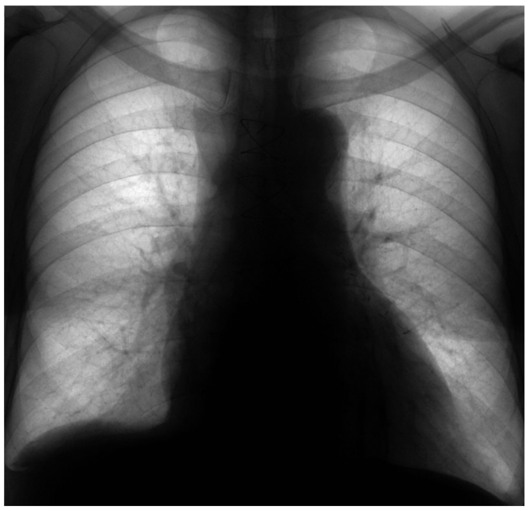


Figure 3: Example resulting X-ray image from the chest of a patient [11]

X-rays are a form of electromagnetic radiation, similar to visible light, except with much higher energy. They are generated by an X-ray tube (the X-ray source), which contains a cathode and an anode. Electrons inside this tube are accelerated from the cathode to the anode when a voltage is applied. When these electrons hit the anode, they are decelerated and deflected, and form X-rays.

These X-rays form a beam that interacts with the body of the patient. The materials in a body absorb this beam depending on their density. The variation in X-ray absorption results in a difference in the resulting beam intensity.

X-ray scatter is a phenomenon that occurs when X-ray photons interact with matter. A photon can potentially strike an electron, releasing it from its atom. The original X-ray photon loses some of its energy and changes direction. The scattered X-ray photon may then go on to interact with other atoms in the material. This interaction is a result of the X-ray photons interacting with atoms of the material they are passing through. An example of X-ray scatter is shown in Figure 4. X-ray scatter is undesirable in medical imaging since it degrades image quality. An X-ray scatter grid as shown in Figure 5 reduces the impact of scatter, but it does not fully eliminate scatter.

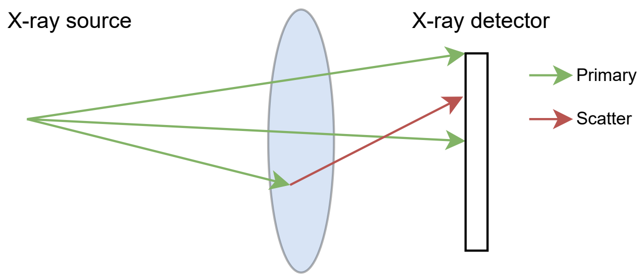


Figure 4: X-ray scatter

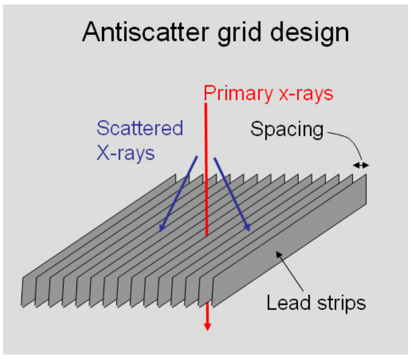


Figure 5: Scatter grid, reducing the impact of scatter [19]

There is also an X-ray detector on the other side of the body respective to the X-ray tube. This detector is positioned such that its central axis is perpendicular to the line connecting the X-ray source and the center of the detector. It captures the X-rays that pass through the body. The detector converts these X-rays into visible light. The targeted detectors are indirect conversion FPDs (Flat Panel Detectors). A layer of cesium iodide (CsI) is used to convert X-rays to light photons. The photons produced are then transformed into electric charges by a matrix of photodiodes. These charges are then converted to a digital representation [11].

The captured image is sent to a computer for thorough image processing. Different absorption levels are assigned different shades of gray, resulting in the well-known X-ray images, which show the internal structure of the body. The final X-ray image is then displayed on a monitor so doctors can analyze it to diagnose medical conditions [11]. An example result of an X-ray can be seen in Figure 3.

2.2 Mass attenuation coefficient

The mass attenuation coefficient describes how a material reduces the intensity of a beam of X-rays or other matter as they pass through it. It is denoted by the symbol μ/ρ, where:

- μ represents the linear attenuation coefficient, which is a measure of how much the intensity of radiation decreases as it passes through a material.
- ρ represents the density of the material.

In other words, the mass attenuation coefficient tells us how a material attenuates X-rays per unit mass. It is usually expressed as cm^2/g (centimeters squared per gram) and is used to model the interaction between radiation and matter. The mass attenuation coefficient depends on a few factors. The type and energy of the radiation is important, as well as the composition and density of the material through which the radiation is passing.

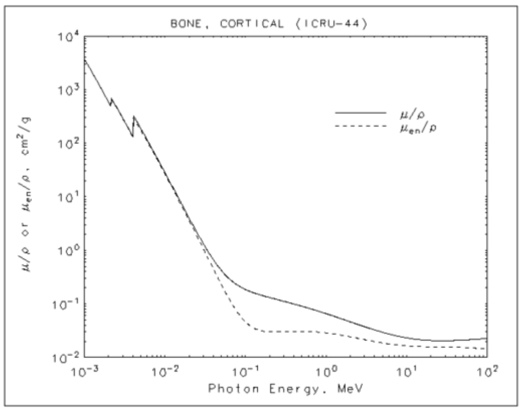


Figure 6: Values of the mass attenuation coefficient, μ/ρ, and the mass energy-absorption coefficient, μ_{en}/ρ, as a function of photon energy, for cortical bone [10]

2.3 FPGA

An FPGA (Field Programmable Gate Array) is a kind of Integrated Circuit (IC). Its main selling point is that it can be reprogrammed after manufacturing. This makes it perfect for implementing different IC designs. Reprogramming an FPGA is done using a Hardware Description Language (HDL), primarily Verilog or VHDL.

FPGAs consist of numerous configurable logic blocks (CLBs). These cells are interconnected to create custom digital logic circuits. Each cell contains look-up tables (LUTs), flip-flops, multiplexers, and other basic logic gates.

These cells are connected by a network of programmable routing resources on the FPGA. This allows users to establish connections between logic cells. These resources include programmable interconnects, switches, and routing channels. A visualization of an FPGA can be found in Figure 7.

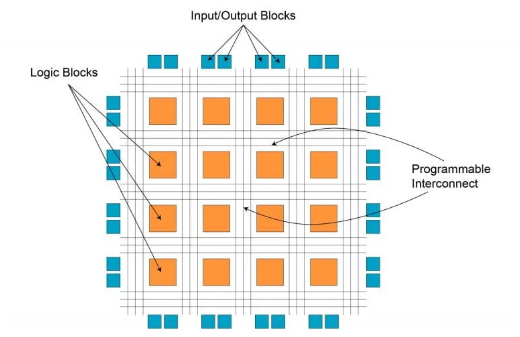


Figure 7: Simplified overview of an FPGA architecture [2]

The Versal series is a family of FPGAs developed by Xilinx, a leading manufacturer of programmable logic devices. Versal devices are part of the latest device family, and are designed to address the growing need for versatile processing platforms [24]. Its clock frequency is high at 400 MHz.

The older Ultrascale+ architecture is primarily designed for traditional FPGA applications and high-performance computing tasks [20]. It was released in 2016 and has a lower clock frequency of 300 MHz.

2.4 Related work

Gao et al.[8] investigated the impact of using synthetic data, as opposed to real data, for training artificial intelligence (AI) models. The study revealed that synthetic data can be a viable substitute for real data in the context of AI training. This finding shows the potential of synthetic data to deliver results comparable to those achieved with real-world data.

Prokopenko et al. [16] researched a way to generate CT images from MRI data by using a Generative Adversarial Network (GAN), to reduce the amount of radiation dose needed for a correct image. It was found that with more research, using a GAN could lead to a reduction of examinations for patients.
Teixeira et al. [18] studied a method to predict human anatomy from a body geometry surface. First, the position of specific markers on the body is estimated. These are then used as input for a GAN. It was shown to result in more accurate images than comparable methods. The quality of the result does however differ depending on the region of the body.

Dorgham et al. [4] proposed a parallel processing technique to create synthetic CT images from a model. This was done by computing the results for many rays in parallel. A speed-up of more than three times while using 4 CPU cores and more than 5 times while using 8 cores was found. This is still not fast enough for this research.

The same researchers (Dorgham et al.) [5] researched approximations to reduce the computational overhead of calculating a full image on a GPU. It resulted in images with sufficient quality for certain applications, however is not accurate enough for most other applications.

Dhont et al. [3] proposed a new framework called RealDDR. This framework starts with a raytracer that produces an initial 2D image, which is then further refined by a GAN to create authentic X-ray images. The raytracer was created for use on the graphics processing unit (GPU). After training the network, it achieves a speed of 100 images in 2 minutes.

3 Process towards solution

In this chapter, the computational challenges associated with X-ray image processing, as defined in earlier chapters, will be analyzed, and the alternative solutions used to address these challenges will be discussed. This chapter will first give some definitions of the problem in Section 3.1. A naive approach and its issues will be discussed in Section 3.2. These issues and possible solutions will be further examined in Section 3.3. At last an overview of these solutions will be given in Section 3.4.

3.1 Definitions

Before explaining any algorithm, it is essential to define the key components of the system: the X-ray source, the detector, and the model. These definitions provide the foundational information for understanding how the algorithm works and how it simulates X-ray interactions.

Definition of source and detector

The X-ray source position, the four corner positions of the detector, and the resolution of the detector are important parameters for simulating X-ray imaging. These parameters help create the geometry and characteristics of the imaging system. Here is a breakdown of each of these parameters:

- **X-ray source position:** The X-ray source position is defined as a vector with three values: its X, Y, and Z coordinates in 3D space. This position represents the location from which X-rays are emitted. An accurate location of the source is necessary for accurately simulating the trajectory of X-rays as they interact with the model.
- **Detector corners:** The four corners of the detector are also defined as vectors, each with X, Y, and Z coordinates. These vectors describe the positions of the four corners of the detector in 3D space. The detector is used to detect the X-rays after they have interacted with the model. Knowing the positions of the detector corners is important to determine how the X-rays hit the detector.
- **Detector resolution:** The detector resolution specifies how finely the detector is subdivided into pixels. It is defined as the number of pixels in the X and Y directions. For example, a resolution of 1024 x 1024 indicates that the detector consists of 1,048,576 individual pixels. A higher resolution provides more detailed images but requires more computational resources to process.

Definition of model

A voxel model is used for this algorithm. A voxel is a three-dimensional equivalent of a pixel. It has a position vector and a value. In this case, this value encodes the material of the voxel as a mass attenuation coefficient, as explained in Section 2.2. For the theoretical calculations in this chapter, one mass attenuation coefficient is assumed to take up 32 bits.

The voxel model is built up from these small voxels. For this research, a small model of a skull is used as an example. This model has 384 x 297 x 384 voxels, for a total of 43,794,432 voxels. A frontal view of the skull model can be seen in Figure 8.

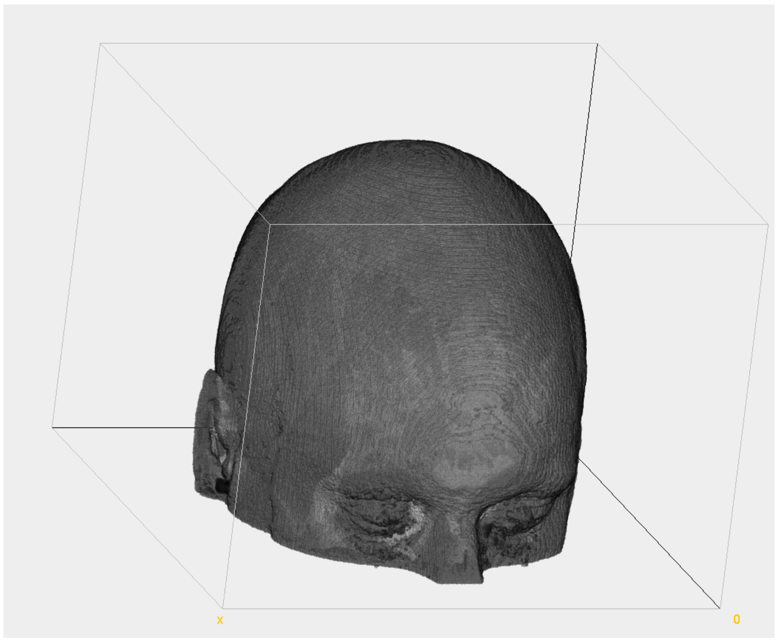


Figure 8: Small skull model, only voxels with nonzero value

Two other models are used to measure theoretical performance. These include a medium (512 x 512 x 512 voxels) model as well as a large (1024 x 1024 x 1024 voxels) model. These do not represent real models but are merely used for performance calculations, and thus do not result in a useful image.

3.2 Naive approach

A naive approach to creating a final image using a voxel model is to generate rays straight from the source to all pixels of the final image and calculate each intersection between the rays and the model. The complete pseudocode is depicted in Algorithm 1, and a simple visualization can be seen in Figure 9.

Algorithm 1 Naive approach to calculate intersection for each ray

```
1: for pixel $p$ of detector do
2:     $ray \leftarrow$ generate ray from source to $p$
3:     $p_{value} \leftarrow 0$
4:     for voxel $v$ in the model do
5:         if $ray$ intersects $v$ then
6:             $p_{value} \leftarrow p_{value} +$ intersection length $*$ value of $v$
7:         end if
8:     end for
9: end for
```

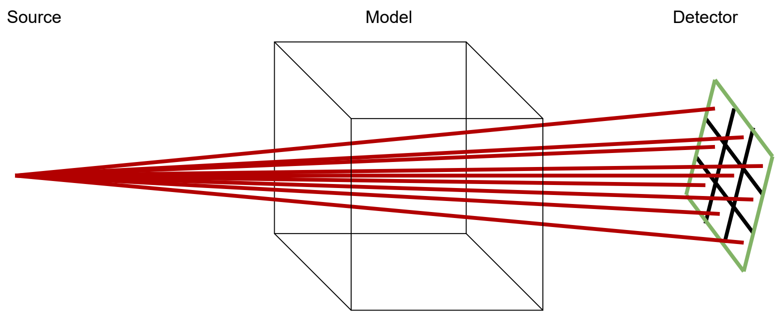


Figure 9: Naive approach: generating rays from source to detector, and computing all intersections

This approach is very simple, yet has a huge flaw: the performance of this naive approach is low. For each pixel of the final image, a loop through all voxels of the model is needed. The amount of intersection calculations can be calculated as follows, where *model_dim* represents the dimensions of the model:

$$num\_intersect\_calc = detector\_resolution_x * detector\_resolution_y * model\_dim_x * model\_dim_y * model\_dim_z$$

For each intersection calculation, the voxel data of that voxel needs to be accessed. As stated in Section 3.1, each voxel value is represented by 32 bits. From this information, the total model memory access volume needed per final image can be calculated. These can be seen in Table 1.

Model	Model size	Detector size	Intersection calculation	Total memory access volume
Small	384 x 297 x 384	220 x 220	2.1196505 teraOps	8.5 terabyte
Medium	512 x 512 x 512	220 x 220	6.496138e teraOps	25.9 terabyte
Large	1024 x 1024 x 1024	220 x 220	51.969104e+13 teraOps	208 terabyte

Table 1: Intersection calculations and total memory access size for 1 image

Bottlenecks of naive approach

It is important to address the two main problems in the naive algorithm: the large required memory bandwidth and the large number of intersection calculations. Both of these issues negatively impact the efficiency of the algorithm. Below is a brief overview of each problem:

Intersection computations
The first of two problems is the large number of intersection computations. Lots of comparisons and computations have to be done in this naive approach. Since all computations are very similar and independent of each other, a possible solution for this is parallel processing. Applying parallel processing techniques can distribute the computation load across multiple resources. This can significantly reduce the time required to perform a large number of intersection computations, which makes the algorithm more efficient. However, it is essential to design the algorithm and data structures with parallelism in mind and consider potential challenges such as load balancing and data synchronization to fully realize the benefits of parallel processing.

Memory bandwidth
The second problem of this naive algorithm is the large required memory bandwidth. As seen in Table 1 even for a small model and small detector, the total memory needed for only one final image is already 8.5 terabytes, which is orders of magnitude too large. If the desired 60 frames per second is achieved with this algorithm, this would amount to a total data flow of more than 500 terabytes per second. This high memory requirement is problematic. If the medium and large models are considered, the total memory access size and thus the required memory bandwidth only increases.

Since the focus of this research is on the algorithm itself, only reducing the required memory bandwidth is considered. To reduce the required memory bandwidth, the following two possible solutions were researched:

1. Data compression: If the voxel data allows for it, data compression techniques can reduce the memory footprint while maintaining essential information.
2. Data caching: In the naive approach each voxel can be accessed more than once. The algorithm could be reshaped in such a way that all voxels only need to be accessed once, and are cached for use thereafter.

3.3 Solution investigation

By examining the naive approach, three techniques to improve the algorithm were identified: parallel processing, data compression, and data caching. These techniques are further investigated in this section. They will be used to propose a new algorithm that is more efficient than the naive approach. This new algorithm is discussed in Section 4.1. As the target of this algorithm is an FPGA device, the investigations are specifically catered towards an FPGA implementation.

3.3.1 Parallel processing

Algorithms with loops are good candidates for parallelism, especially when the loop iterations are independent of each other. Parallelism can help exploit the inherent concurrency within these algorithms, making them more efficient and reducing execution time.

Implementing parallelism in an FPGA can be done by using so-called DSP slices (Digital Signal Processing slices). A DSP slice is a specialized hardware block designed to perform digital signal processing operations efficiently. These slices are a key feature of many modern FPGAs and are optimized for tasks that involve arithmetic calculations often found in signal processing and other computational workloads. DSP slices can be customized to perform a wide range of arithmetic and logic operations.

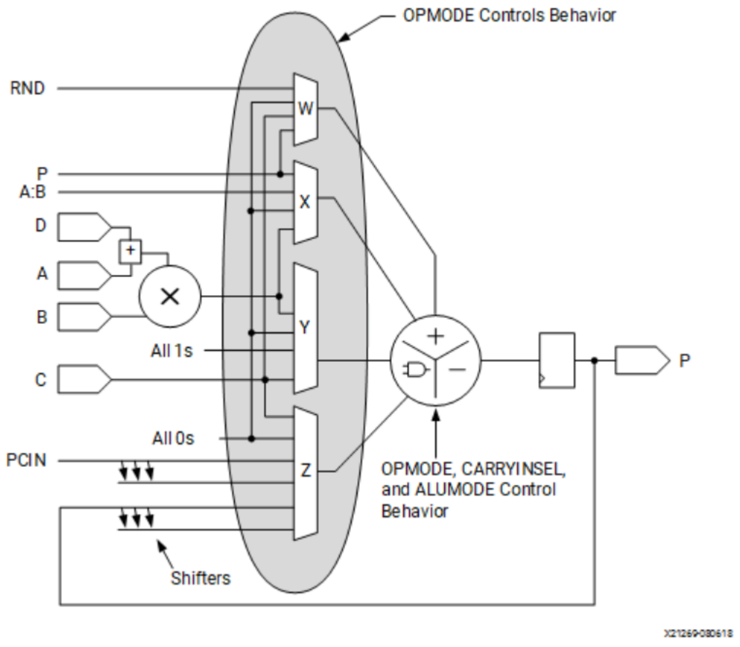


Figure 10: Simplified operations of a DSP58 slice [22]

A simplified version of a DSP slice can be seen in Figure 10. It can be used to do the multiply-accumulate operations present in the algorithm. This makes it a good candidate for parallelizing this algorithm.

Multiple DSP slices can be connected in series. This cascading technique allows for the chaining of DSP slices to create multi-stage pipelines that process data sequentially. Each DSP slice is responsible for a specific stage of the computation. The result of the computation in the first DSP slice is passed to the second DSP slice in the cascade. The intermediate results move from one stage to the next in a sequential fashion. The final DSP slice in the cascade performs the last stage of computation, and the output is produced.

Two different techniques of parallelizing the algorithm using DSP slices were considered:

- **Parallel ray calculations:** The ray calculations can be parallelized. This means that each DSP slice is responsible for calculating the result for one ray. This has two implications. Firstly there is a limit on the amount of DSP slices on an FPGA, which would either limit the final image size or would make each DSP slice responsible for calculating multiple rays sequentially. The second implication is a bigger problem: Each DSP slice still needs access to the whole model. If a voxel is traversed by multiple rays, the voxel data has to be accessed multiple times by multiple different DSP slices.
- **Parallel model calculations:** For this technique each DSP slice is responsible for computing all intersections between a part of the model and all rays. This has a few advantages. Firstly it means that each DSP slice only requires the voxel data of a specific part of the model. This does not inherently reduce the total memory access size. It does however allow for better methods of data caching locally, which in turn will reduce the total memory access size. This will be examined later.
  The cascading feature of the DSP slices can be used to accumulate the total value for a ray over all parts of the model. The cascading of DSP slices can be used for the accumulation of the total value for a ray as it traverses different parts of the voxel model. This technique allows the algorithm to process the interactions of the ray with the voxel data in a sequential manner, calculating and accumulating the values as the ray progresses through the model.

Model parallelization can be used more optimally for this algorithm. Therefore it was chosen to research further using this parallelization technique.

3.3.2 Data caching

Some form of data caching is necessary to lower the required memory bandwidth. To cache data for a DSP slice in the Versal FPGA series, UltraRAM (Ultra Random Access Memory, also abbreviated as URAM) can be used. The main advantages of UltraRAM are:

- High capacity: UltraRAM blocks provide a large amount of on-chip memory (288 kilobytes per UltraRAM block). This makes them suitable for storing large data structures like a partial model.
- Low latency: UltraRAM offers low-latency access. This makes it ideal for applications that require fast read and write operations.
- High bandwidth: UltraRAM can provide high memory bandwidth, which allows for rapid data transfers.
- Configurability: UltraRAM can be configured to match the specific requirements needed. It is for instance possible to use an UltraRAM as single-port or dual-port, or add parity bits.

Since a DSP slice needs regular data access, a one-to-one ratio of UltraRAMs and DSP slices was chosen. This limits the amount of parallelization somewhat, as there are fewer UltraRAMs available on a Versal device than DSP slices [23]. The details of the caching strategy will be further examined in Section 4.2.3.

3.3.3 Data compression

If 32-bit values are used for the voxel values, the small model has a size of $384*297*384*32/8 = 175$ megabyte. By using the data compression as shown in Figure 11, the model size can be reduced.

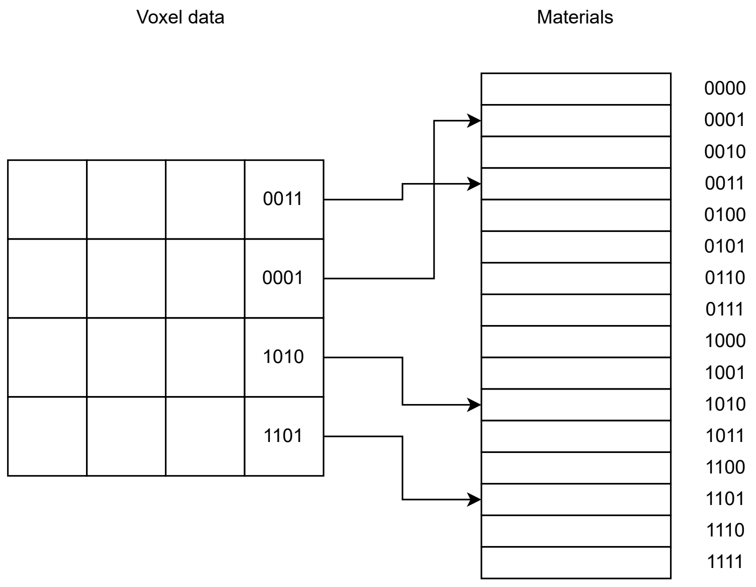


Figure 11: A 2-dimensional representation of a model, where each voxel can have 1 of 16 materials

Instead of each voxel having a specific material value, 16 different material values are defined. Each voxel has a value of 0 to 15 (4-bit binary value), defining which of the 16 possible material values apply to that voxel. The size of the model itself is now reduced to only $384 * 297 * 384 * 4/8 = 21.9$ megabyte. The material values only add $16 * 32/8 = 64$ bytes of data.

The drawback of using this technique is that only a limited amount of materials can be defined for a model. The current models require up to 16 materials, and as such a maximum of 16 materials is not a concern. If a model is however made up of more than 16 different materials, two different solutions exist:

- **Simplified model:** Simplifying the model by merging multiple materials with similar values is a practical approach to reduce the memory requirements of the representation.
- **Increase material amount:** If a model is made up of more than 16 different materials, this does not fit into a 4-bit value. Transitioning from 4 bits per voxel to 8 bits per voxel is a logical choice for these models. It allows for greater flexibility in material representation while aligning with common memory standards. This means that the size of the model increases by a factor of 2, but it does increase the number of possible materials per model to 256 (2^8).

3.4 Overview

All the above considerations come together in a global overview of the discussed improvements as seen in Figure 12. Both the parallelization and data caching strategies are incorporated in this design.

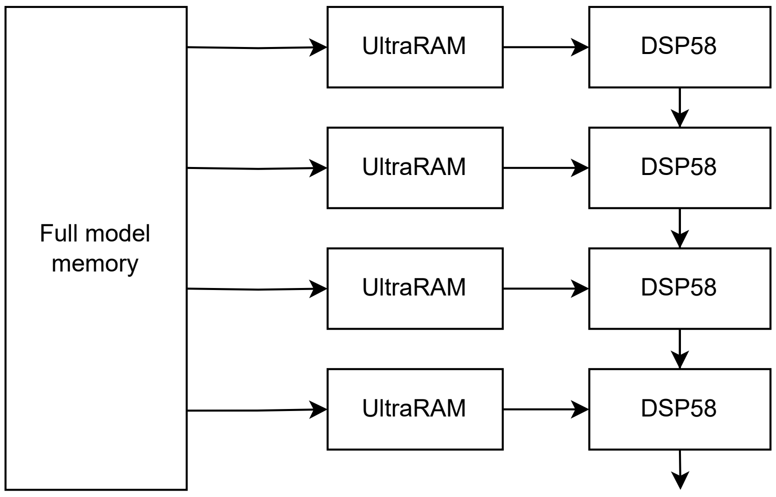


Figure 12: A global overview of implementing data caching and parallelization

The voxel model is first compressed to a smaller format using the data compression method. This model is then divided into smaller pieces. An UltraRAM is responsible for buffering and caching the data necessary for a piece, while the DSP slice does the actual computations. The data is then accumulated throughout the DSP chain. The last DSP slice in the chain produces the final values.

This overview outlines the thought process and key considerations that led to the design of the algorithm for X-ray imaging. A more detailed and specific explanation of the proposed algorithm will be given in the next chapter.

4 Proposed algorithm explanation

In this chapter, the workings of the final proposed algorithm will be explained. In Section 4.1 a general overview of the complete algorithm will be given. Sections 4.2 and 4.3 examine the projection step and detector step of the algorithm in more detail.

4.1 Overview of algorithm

The algorithm is divided into two distinct steps, the projection step and the detector step. This separation of tasks simplifies the design of the algorithm and allows for efficient parallelization and optimization. An overview of this division can be seen in Figure 13.

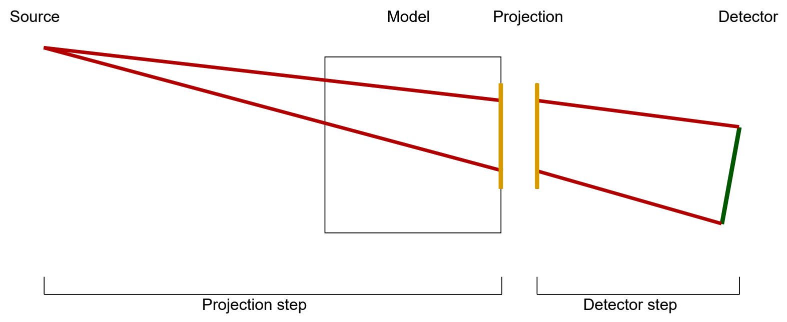


Figure 13: 2D overview of the two parts of the algorithm, projection step on the left, detector step on the right

The projection step involves calculating a projection image, which represents the interaction of X-rays with the voxel model as seen from the X-ray source. This step includes calculating the X-rays from the source through the model to the projection image and accumulating these interactions to calculate the pixel values. The result of the projection step is the projection image, a 2D representation of the X-ray attenuation as it passes through the voxel model.

The detector step takes the projection image generated in the previous step and transforms it into the final image. The output of the detector step is the final X-ray image.

An overview of the substeps involved in each step can be seen in Figure 14. The explanations of the projection step and detector step follow the same structure.

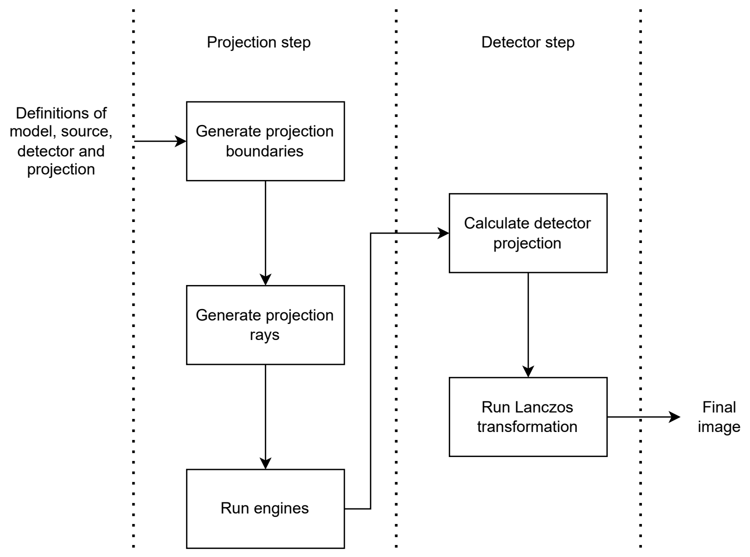

Projection step
Detector step
Definitions of model, source, detector and projection
Generate projection boundaries
Generate projection rays
Run engines
Calculate detector projection
Run Lanczos transformation
Final image

Figure 14: Overview of the two distinct steps

In addition to the definitions provided in Section 3.1, a specific set of projection parameters should be defined. These projection parameters are essential for configuring the X-ray imaging system and include parameters related to the intermediary product.

Definition of projection

Defining the resolution of the projection image and specifying the size of the pixel boundary are essential parameters for controlling the accuracy of the projection step. They play a critical role in determining the computational resources required. Below is an explanation of these parameters:

- **Resolution of the projection image:** The resolution of the projection image defines the level of detail captured in the X-ray projection. It specifies the number of pixels in the X and Y dimensions and affects the clarity of the resulting projection. A higher resolution provides more detailed images but requires additional computational resources and memory for processing and storage.
- **Size of the pixel boundary:** The size of the pixel boundary, also referred to as pixel padding, defines the region surrounding the edge of the projection image. This boundary is used to capture additional information and to accommodate interpolation and filtering processes. The size of the pixel boundary influences the precision of the final result and can affect the handling of pixel values near the edges of the projection.

4.2 Projection step

The projection step is the first of two steps in this algorithm. This step results in a projection image. Only in this step, the voxel model is needed. It follows the structure of Figure 14.

4.2.1 Projection boundaries

A first projection boundary is calculated by projecting the detector boundaries onto the model, as seen from the source. This projection has a quadrilateral shape. This projection is fit to a rectangular area based on the given pixel size of the projection image. The choice to approximate the projection boundary as a rectangle, while the original projection is a quadrilateral, allows for a more straightforward and hardware-friendly implementation. Rectangular regions are well-suited for memory access, calculations, and data processing in many hardware platforms, including FPGAs.

The additional step of padding the rectangular image after projecting the boundary is a common technique in image processing and computational geometry, for instance in machine learning [9]. It introduces a buffer of additional pixels around the rectangular image. This allows for more accurate interpolation around the edges during the second step.

Figure 15 shows the projection image relative to the model.

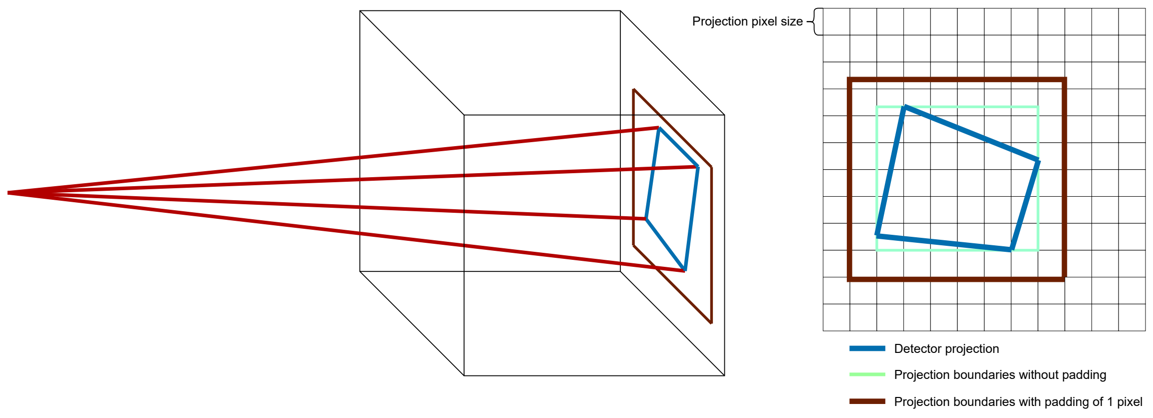

Projection pixel size
Detector projection
Projection boundaries without padding
Projection boundaries with padding of 1 pixel

Figure 15: Projection image relative to model on the left, projection boundary with a pixel padding of 1 on the right

4.2.2 Projection ray generation

Rays are then formed from the X-ray source towards all pixels of the projection image. The total amount of rays is thus equal to the amount of pixels in the projection image. This step involves calculating the end point of each ray and computing the slope of the rays.

Figure 16 shows a two-dimensional representation of the current state of the algorithm. It can be seen that the rays extend further than the edge rays to the detector. This is the aforementioned pixel boundary.

The length of the ray is also computed and saved. This allows for an optimization in the intersection calculation, which will be explained further in the next section.

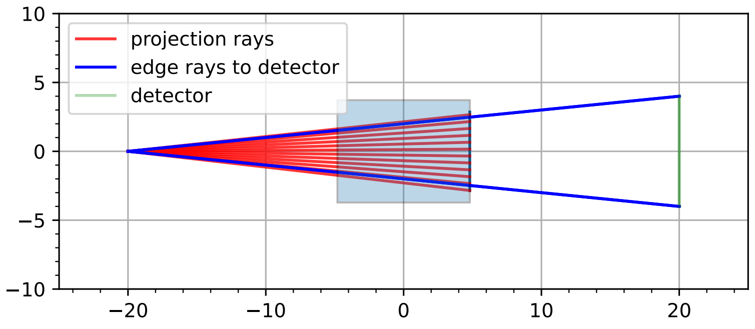


Figure 16: Visualization of algorithm state after ray generation

4.2.3 Engine

The approach of building the algorithm around parts of the model rather than individual rays is a significant optimization, already discussed in Section 3.3.1. Model layers were chosen to represent those parts. In Figure 17, the division of the model into model layers can be seen. The layers are parallel to the projection image. This alignment ensures that the rays intersect with the model layers consistently.

The property that the order of materials encountered by a ray does not matter is a significant simplification. This property allows the intersection computations between the ray and model layers to happen independently and in parallel. Further explanation of the projection step will therefore feature just one model layer, as this can be generalized to all the others.

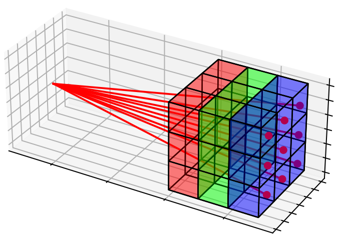


Figure 17: Different layers of a model

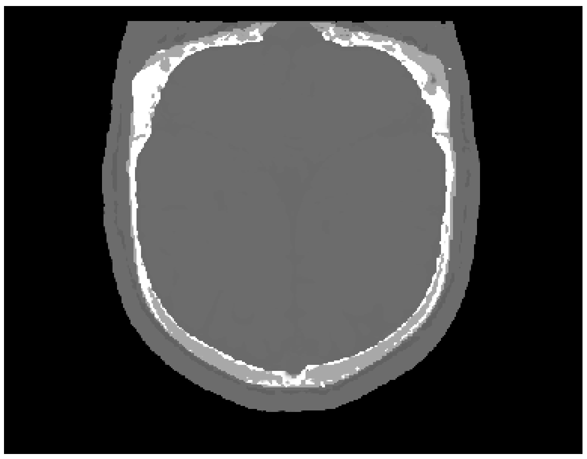


Figure 18: Example layer of the skull model

The concept of using an engine to perform the calculations for a set of assigned model layers is a practical approach to the algorithm's design. The key points about engines are listed below:

- **Parallelization with engines:** Each engine is responsible for calculating the intersection between its assigned model layers and each passing ray. This parallelization approach allows for the efficient distribution of computational work among multiple engines, improving performance.
- **Workload assignment:** The workload of one engine can differ per implementation. Further explanations assume a workload of 1 model layer per engine. This simplifies the explanation but provides a clear illustration of the concept. In practice, multiple model layers can be assigned to an engine, allowing for more efficient utilization of computing resources.
- **Consistent calculations:** Each engine performs the same set of calculations for its assigned model layers. This consistency ensures that the calculations can be executed in parallel and that the same operations are applied to all layers.

- **Scalability:** The design allows for scalability by adding or removing engines as needed, depending on the computational requirements and the available hardware resources.

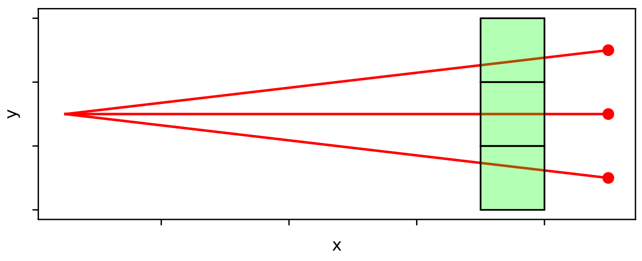


Figure 19: Orthographic view of a single layer from the z-axis; in the z direction the rays overlap

Rays that end on the same horizontal line in the projection overlap when viewed from the side. This is an important characteristic of the algorithm design. This overlap occurs because these rays share the same starting position and have the same Y coordinate for their endpoint in the projection image. This is shown in Figure 19. To ensure efficient traversal of voxels, the algorithm divides each model layer into lines. This division simplifies the calculations and provides a structured approach to processing the voxel model.

By calculating the intersections from top to bottom in the projection image, the voxel lines within each model layer can also be traversed from top to bottom. This consistent traversal direction simplifies the calculations and ensures that all model layers can be processed in the same manner. Figure 20 shows the division of a model into lines.

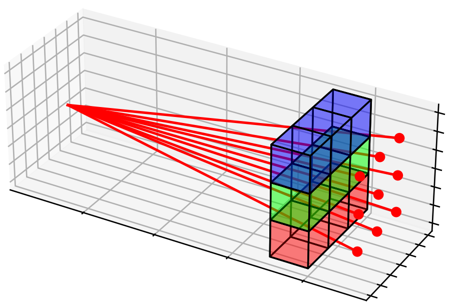


Figure 20: Division of model layer into lines; the engine will calculate intersections with the blue voxels first, then the green voxels, and at last the red voxels

Intersection calculation

Only one intersection calculation is executed per model layer. This is an important aspect of the algorithm and helps optimize the computation. It implies that when a ray traverses multiple voxels within a model layer, only the voxel intersected in the middle is taken into account. This is shown in Figure 21. The intersection is computed as the voxel material multiplied by the ray length inside that voxel.

$$intersection\_result = voxel\_material * ray\_length\_in\_voxel$$

Factoring out the constant ray length is an optimization strategy that simplifies the algorithm and accelerates calculations. By assuming a constant ray length of 1 within each voxel, the intersections with the voxel are normalized. After accumulating the result of all intersections for a ray, this result is then scaled by the total ray length.

$$intersection\_result = voxel\_material$$
$$final\_ray\_result = ray\_length\_in\_model * \sum_{l=0}^{num\_layers} intersection\_result_l$$

This optimization assumes that the detector is perfect, and no scatter is computed. If scatter is introduced, this optimization might not be suitable anymore.

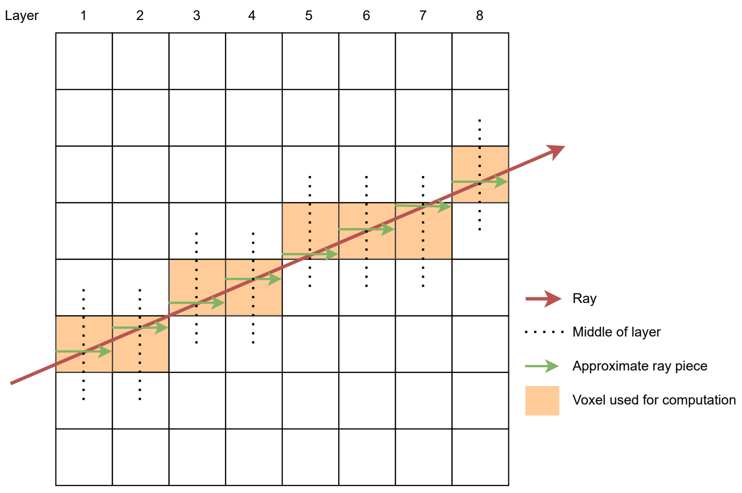


Figure 21: 2D simplification of ray intersection calculation, only the orange voxels are considered

A ray can traverse up to 4 voxels per layer in 3D. This computation acts as if it only traverses a single voxel. By ignoring this error factor, the algorithm is simplified and the computational complexity is reduced. However, it is crucial to be aware that this simplification introduces a potential source of error in the calculations. Future research can be done to find ways to reduce this error term. This could involve calculating each intersection as if the ray was a cylinder. In that case, the previously mentioned optimization is also not applicable.

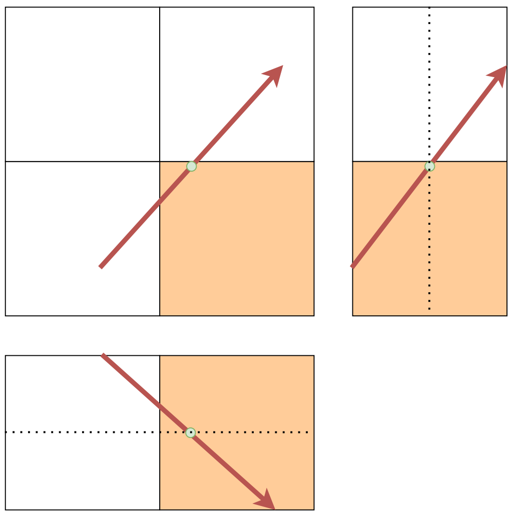


Figure 22: Top-down view and two side views of a model layer. Currently, the intersection calculation only considers the position of the ray in the middle of a model layer, while this ray intersects three different voxels

Caching/buffering

The use of a buffering/caching strategy is a practical approach in the design, and it is based on the top-to-bottom traversal of each model layer. This buffering strategy offers several advantages in terms of optimizing the performance of the algorithm:

- **Minimized memory access:** By caching voxel lines, the algorithm can reduce the frequency of memory accesses. Since the model is traversed top-to-bottom, any cached voxel lines that are not used anymore can be removed without having to access them again later.
- **Continuous computation:** With a buffer in place, the algorithm can continuously process and calculate intersections for a set of model lines without the need to access external memory. This minimizes computational interruptions. Since the model layer is traversed from top to bottom, it is always known which line of voxels needs to be cached next.
- **Pipeline efficiency:** The buffer contributes to pipeline efficiency by ensuring a steady supply of data for computation, minimizing idle times in the engine.

Multiple engines

The calculation for only one engine is not sufficient. Multiple engines can be stacked (cascaded) to calculate all intersections for a ray. When an engine is done with calculation for a specific ray this information is communicated to the next engine, which proceeds with this calculation. The original engine can then start computing the next ray. This is visualized in Figure 23.

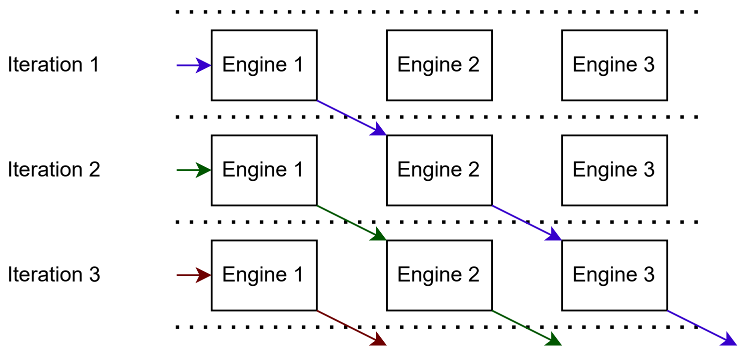


Figure 23: Engine communication: arrows represent data flow (rays)

The last engine in this chain calculates the final values. These values can be assembled to form a projection image. An example of a projection image can be seen in Figure 24.

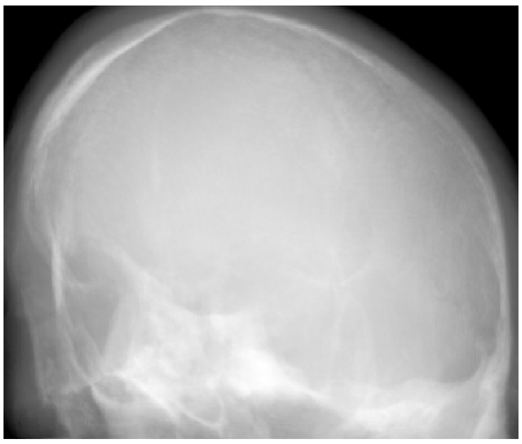


Figure 24: Example result of projection step

4.3 Detector step

The detector step, which follows the projection step in the algorithm, is focused on transforming the projection image into the final X-ray image. While the projection step involves generating a projection image that represents X-ray interactions with the 3D model, the detector step processes this data to create the final output image.

4.3.1 Detector projection

To figure out which known data points of the projection image to sample, the position of each final image pixel projected onto the projection needs to be calculated. An example result can be seen in Figure 25. It shows the rectangular projection image, with the detector and its pixels projected onto it. This mapping ensures that the final image accurately represents the X-ray interactions captured during the projection step.

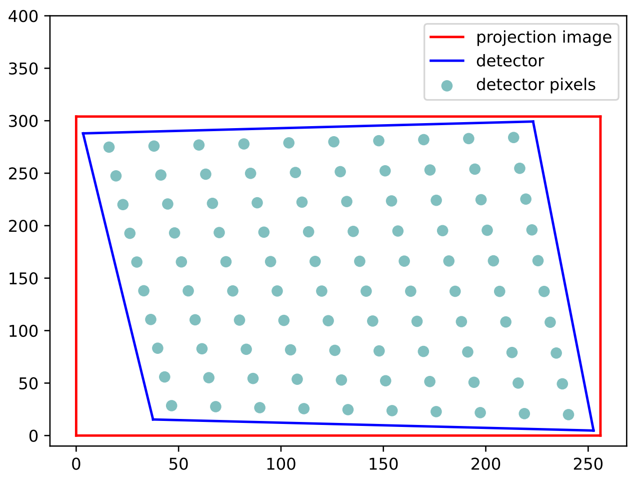


Figure 25: Example detector pixels (10x10 detector) relative to the projection image

4.3.2 Lanczos transformation

Lanczos resampling is a digital image processing technique used for rescaling or resizing images. It is a high-quality resampling method that aims to minimize aliasing artifacts and maintain image sharpness when scaling images to different dimensions. Lanczos interpolation is used in various situations in medical imaging, including three-dimensional interpolation [7]. In medical imaging, the accuracy and quality of reconstructed images are critical for diagnostic and research purposes. Figure 26 shows that the Lanczos interpolation method gives good results for most patients.

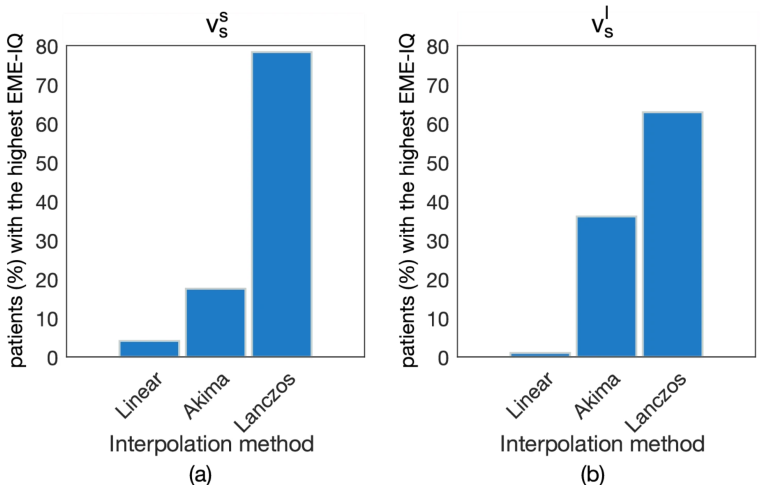


Figure 26: Comparison of linear, Akima, and Lanczos interpolation methods, based on the percentage of patients receiving the highest EME-IQ score (higher EME-IQ values indicate better preserved sharpness and edges), for both upsampling (a) and downsampling (b) [12]

Lanczos resampling is based on the mathematical concept of the sinc function, a well-known signal processing function. The resampling process involves applying a convolution filter to the image. This filter is a weighted sum of sinc ($\frac{sin(x)}{x}$) functions, and it uses a windowed version of the sinc function known as the Lanczos kernel. The Lanczos kernel has a parameter a, which determines the size of the kernel, and controls the trade-off between sharpness and suppression of interpolation artifacts. This is usually set to 2 or 3. To avoid artifacts in the final image, a value of 3 is used. The kernel is as follows:

$$Lanczos\_kernel(x) = \begin{cases} sinc(x) * sinc(\frac{x}{a}) & \text{if } -a < x < a \\ 0 & \text{otherwise} \end{cases}$$

The following equation describes how a continuous function is estimated from a discrete set of samples taken at integer positions using this Lanczos kernel.

$$Lanczos\_continuous\_interpolation(x) = \sum_{i=\lfloor x \rfloor - a + 1}^{\lfloor x \rfloor + a} Lanczos\_kernel(x - i)$$

This is only a one-dimensional kernel, which can be applied to the x or y coordinate of a detector pixel. To use a Lanczos filter kernel in both dimensions, all that is needed is to multiply both one-dimensional results [1].

$$Lanczos(x, y) = Lanczos\_continuous\_interpolation(x) * Lanczos\_continuous\_interpolation(y)$$

For each detector pixel, the Lanczos kernel is evaluated at its mapped position. This results in a final image, of which an example can be seen in Figure 28.

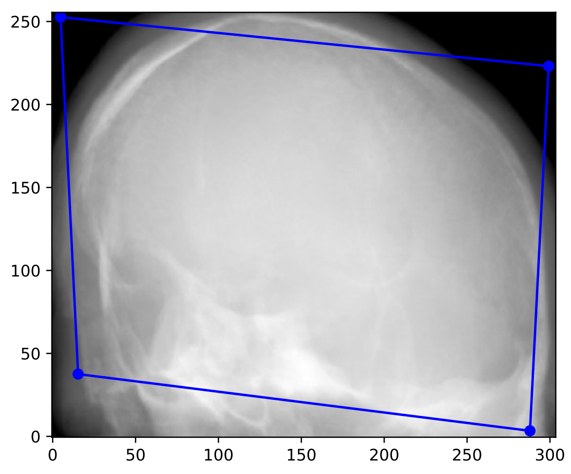


Figure 27: Projection image with detector boundary

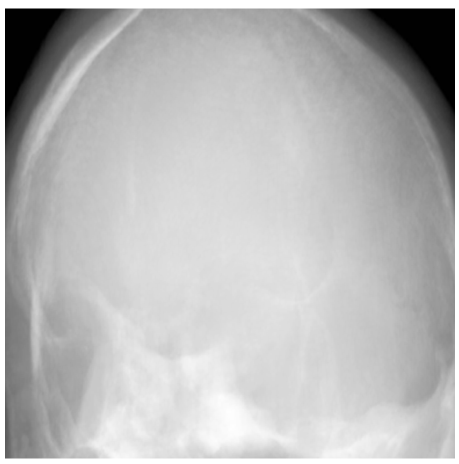


Figure 28: Final image on the detector

5 Algorithm simulation and validation

Both the projection step and the detector step are implemented in software using Python. This was done to verify the correctness of the algorithm before implementation in hardware. First, Section 5.1 will explain the setup of the software. Sections 5.2 and 5.3 explain the software implementation of the projection step and detector step respectively. Subsequently, the validation of the projection step is discussed in Section 5.4. Lastly, the possibility for rotation will be examined in Section 5.5.

5.1 Experimental setup

Jupyter notebook [13] was used to implement the algorithm in software. This makes it easy to change parts of the setup and see the results. Some parts of the code need to be executed only once for different pictures. Other parts can be executed at will to do different actions, like rotation. Furthermore, the NumPy library was used to do some mathematical operations. The experiments were executed on a Lenovo ThinkPad T14S. This system uses an Intel Core i5-10310U CPU and has 16 gigabytes of DDR4 RAM.

5.1.1 Initial parameters

The setup of the algorithm simulation has a few parameters. These are explained in Sections 3.1 and 4.1. The input parameters for the software are slightly different. Internally the same definitions are derived from these input parameters.

Model

The model is loaded as a 3-dimensional array. In this representation, each value in the array corresponds to a voxel, and the array indices define the spatial position of the voxels within the 3D space. The dimensions of the model should be specified as well. The model starts centered on the origin.

Source and detector

Both the source and the detector start on the X-axis. This alignment simplifies the initial configuration. The Source-Image Distance (SID) is the distance between the X-ray source and the detector and should be specified. The distance from the X-ray source to the model is often referred to as the Source-Object Distance (SOD) and should be set as well. These parameters coupled with the detector resolution make up the basic position of the source and detector.

The ability to rotate the source and detector pair allows for all conceivable positions of this pair around the model. Further details regarding the implementation of rotation will be explained in a later section.

Projection

The projection definition is changed slightly as well. The pixel boundary is defined the same as before. The resolution is however derived from a set projection pixel size. It allows for the same flexibility but is in another representation.

5.1.2 Initialization functions

The first two setup functions will execute: the *setup_parameters* and *generate_voxels* functions. These only need to be executed once, even when generating multiple images. Only a change of model or a reset of parameters requires re-executing these functions.

The *setup_parameters* function initializes the parameters that are required for calculations. This includes the source and detector positions, rotation, offsets of the model, and a logger. This logger is used to generate timing documents. These timing documents will be examined later. The model offsets (the distance from any edge of the model to the origin) are calculated as follows:

$$model\_offsets_x = dim_x * vox\_siz/2$$

$$model\_offsets_y = dim_y * vox\_siz/2$$

$$model\_offsets_z = dim_z * vox\_siz/2$$

Generating the voxels is done in the *generate_voxels* function. It reads voxel values from a preset *.npy* file, and returns a 3D array of voxels. The voxels are then scaled by the set voxel size and translated according to the model offsets.

5.2 Projection step

The projection step is subdivided into a few functions. These functions can be seen in Figure 29. First, the projection boundaries and projection rays are generated in the *generate_projection_boundaries* and *generate_projection_rays* functions. These are then used to calculate the projection image using the *run_engines* function. All these functions will be discussed in this section.

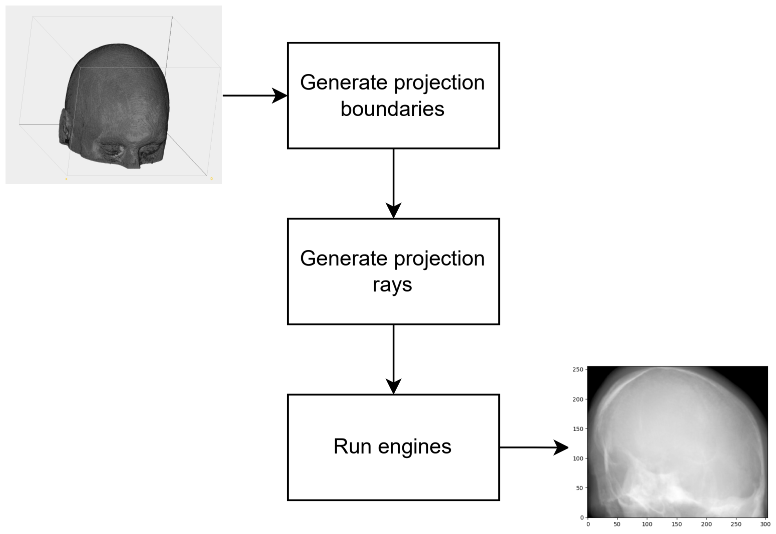


Figure 29: The three functions executed during the projection step

Generate projection boundaries

In the *generate_projection_boundaries* function the boundaries are calculated. For each corner of the detector, a projection point is calculated. The function determines the maximum and minimum values in both the Y and Z planes based on the calculated projection points. In other words, it identifies the Y and Z coordinates of the corners of the rectangular projection boundary. To accommodate any additional pixels (defined by *projection_extra_pixels*), the maximum and minimum values obtained are offset. These offsets expand the boundaries in both the Y and Z directions to include the extra pixel padding. The implementation can be seen in Algorithm 2.

Algorithm 2 Generate projection boundaries

```
1: $projection\_y\_coords \leftarrow$ empty array size 4
2: $projection\_z\_coords \leftarrow$ empty array
3: for Detector point $d$ do
4:     $projection\_detector\_scale \leftarrow (model\_offsets_x - src_x)/(d_x - src_x)$
5:     $dist\_src_y \leftarrow d_y - src_y$                                ▷ Calculate y distance to source
6:     $pos\_at\_projection_y \leftarrow dist\_src_y * projection\_detector\_scale + src_y$
7:     append $pos\_at\_projection\_y$ to $projection\_y\_coords$
8:     $dist\_src_z \leftarrow d_z - src_z$                                ▷ Calculate z distance to source
9:     $pos\_at\_projection_z \leftarrow dist\_src_z * projection\_detector\_scale + src_z$
10:    append $pos\_at\_projection\_z$ to $projection\_z\_coords$
11: end for
12: $boundary\_size \leftarrow projection\_pixel\_size * projection\_extra\_pixels$
13: $boundary_y \leftarrow [min(projection\_y\_coords) - boundary\_size, max(projection\_y\_coords) + boundary\_size]$
14: $boundary_z \leftarrow [min(projection\_z\_coords) - boundary\_size, max(projection\_z\_coords) + boundary\_size]$
```

Generate projection rays

Now that the boundaries of the projection image have been established, the rays can be generated. Each ray is defined by a destination point and its corresponding detector coordinate. This detector coordinate is used solely for visualization purposes and does not play a role in subsequent calculations.

Based on these initial values, the slopes in both the Y and Z directions are calculated, along with the length of the ray segment inside the voxel model. This segment length serves as a parameter for the optimization process explained in Section 4.2.3. These rays are generated in a loop as seen in Algorithm 3.

Algorithm 3 Generate projection rays

```
1: $y \leftarrow boundary\_min_y + 0.5 * projection\_pixel\_size$
2: while $y < boundary\_max_y$ do
3:     $z \leftarrow boundary\_min_z + 0.5 * projection\_pixel\_size$
4:     while $z < boundary\_max_z$ do
5:         Create new ray with destination $[model\_offsets_x, y, z]$
6:     end while
7: end while
```

Run engines

The *run_engines* function serves as the central component where the core of the algorithm is executed. Notably, it adopts a specific data flow strategy that differs from completing one engine before moving to the next. Instead, the data flow is followed, as depicted in Figure 30.

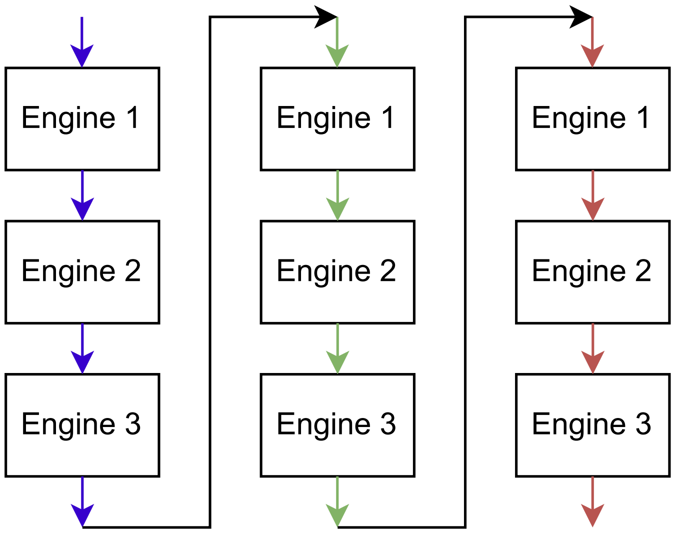


Figure 30: Order of engine calculations in software implementation, with each color representing a different ray

Each engine is uniquely identified with an ID and is tasked with processing a workload consisting of 8 individual layer objects. Each layer is responsible for its part of the final calculation in the engine. The object representing a layer has three associated functions:

- **Initialization:** The layer is initialized with a global ID (its layer number in the complete model), a local ID (its layer number in the engine), and a buffer. The buffer size can be configured. A buffer size of 4 lines per layer is used throughout this research.
- **Cycle buffer:** This function cycles the buffer. It removes the first line of voxels in the cyclical buffer and appends a new line of voxels.
- **Calculate intersection:** This function does the actual intersection computation. This process unfolds in several stages. Initially, the coordinates of the ray are adjusted to align with its position within the current layer. Subsequently, the buffer is cycled whenever necessary. Finally, the computation concludes with determining the index of the intersected voxel within the current line and the subsequent retrieval of the final material value, which is later incorporated into the overall ray total. This can be seen in Algorithm 4.

Algorithm 4 Calculate intersection of one ray and one engine layer

```
1: $ray_y \leftarrow ray_y + ray\_slope_y * voxel\_size$
2: $ray_z \leftarrow ray_z + ray\_slope_z * voxel\_size$                ▷ Update position of ray
3: if $ray_y$ or $ray_z$ outside of model bounds then
4:     Return 0                                                       ▷ Act as if voxel is air
5: end if
6: if $ray_y > buffer[0]_y + voxel\_size$ and $engine\_global\_index < dim_y$ then ▷ Cycle if first item in buffer not in use
7:     Cycle the buffer once
8: end if
9: $z\_index \leftarrow \lfloor \frac{ray_z + model\_offsets_z}{voxel\_size} \rfloor$          ▷ Z index in line
10: $voxel \leftarrow buffer[0][z\_index]$                               ▷ Correct voxel in buffer
11: Return material coefficient of $voxel$
```

In this way, each ray is calculated from start to finish. The start of a ray requires some additional work. To determine the starting position of a ray within the model, the following equations are employed:

$$ray_y = (-src_x - model\_offsets_y - 0.5 * voxel\_size) * ray\_slope_y + src_y$$

$$ray_z = (-src_x - model\_offsets_z - 0.5 * voxel\_size) * ray\_slope_z + src_z$$

Once the final engine completes its computation for a given ray, the last step entails scaling the outcome based on the length of a ray. This scaling factor has already been computed during the generation of projection rays.

5.3 Detector step

The detector step is subdivided into two functions. These functions can be seen in Figure 31. First, the detector projection is calculated. Afterward, the Lanczos transformation is done, resulting in a final image.

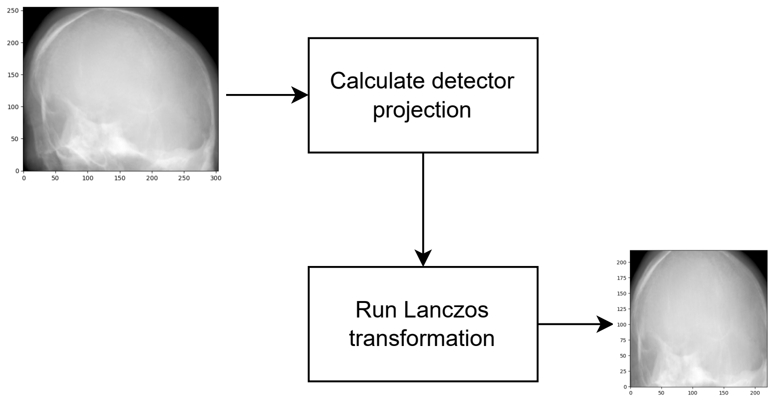


Figure 31: The two functions executed during the detector step

5.3.1 Calculate detector projection

The computation of the detector projection is optimized for hardware implementation and utilizes a technique that incorporates a total of four counters, two for each dimension. To illustrate this concept, the computations for one dimension are shown in Figure 32. The first counter in each dimension is tasked with tallying the number of added segments for each point. The first point in each horizontal line has only one added segment, and this count increases for subsequent points. The second counter keeps track of the length of a segment. The segment length is increased by a constant for each following line. This counter is incremented for each subsequent line.

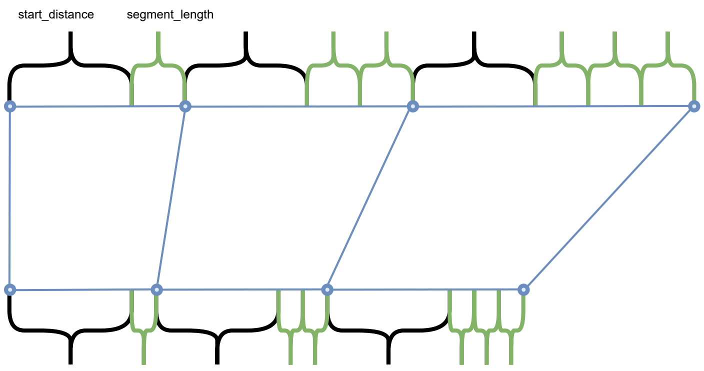


Figure 32: The distance between each following detector point increases by a constant per detector line. This constant increases regularly between lines. The resulting detector image is represented by the blue grid.

An example result can be seen in Figure 33. The intersections of lines represent the middle of each final pixel. These intersection points serve as the locations of which the values will be estimated in the subsequent function.

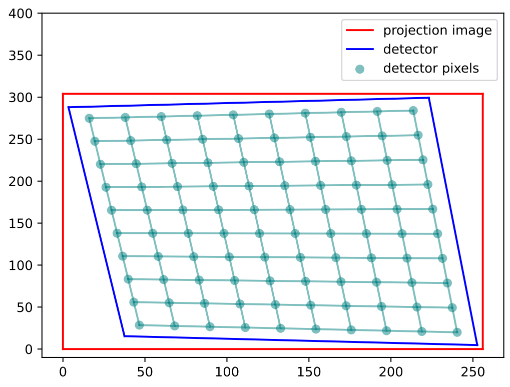


Figure 33: Result of the detector projection calculation

5.3.2 Run Lanczos transformation

The final phase of the algorithm involves applying the Lanczos transformation to each of the positions that were computed in the preceding function. This function implements the Lanczos function discussed in Section 4.3.

The primary distinction between the software implementation of the 2D Lanczos interpolation and the earlier equation lies in the order of calculation. In the current implementation, the algorithm does not perform two separate interpolations over the dimensions. Their kernels are multiplied instead. This approach yields an identical image, of which an example is demonstrated in Figure 34.

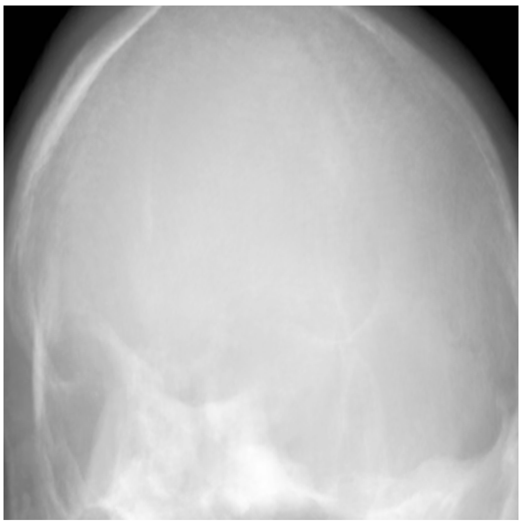


Figure 34: Example result of the proposed algorithm

5.4 Validation

The validation of the projection step in the algorithm is carried out using a specific test scenario. In this scenario, the source and detector points are intentionally positioned far apart from each other. As a result, the rays passing through the model become nearly parallel. Additionally, the spacing between each ray is set to match the height of a voxel. This validation configuration is visualized in Figure 35.

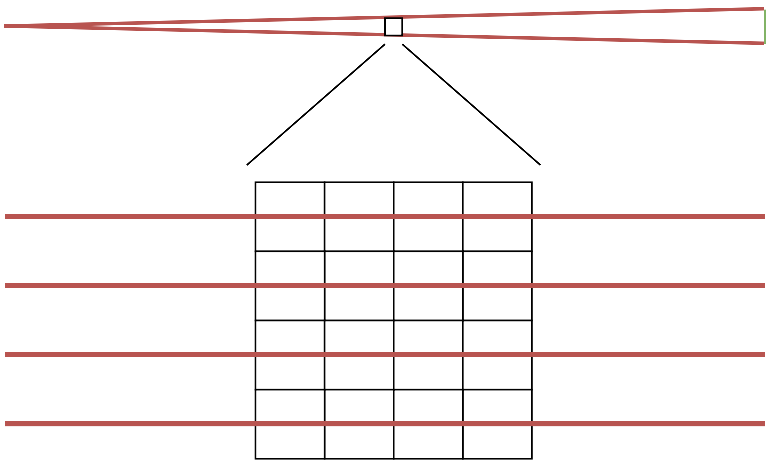


Figure 35: Overview of validation of projection step, rays are nearly parallel

The same result can be generated using an alternative approach. In this method, the test image is generated by aggregating values from all voxels sharing the same Y and Z coordinates as the corresponding projection image pixel. The pseudocode for this process is outlined in Algorithm 5.

Algorithm 5 Trivial approach to calculate validation image

```
1: $verified\_image \leftarrow$ 2D array of only zeroes
2: for $x$ from 0 to $dim_x$ do
3:     for $y$ from 0 to $dim_y$ do
4:         for $z$ from 0 to $dim_z$ do
5:             $verified\_image[y, z] \leftarrow verified\_image[y, z] + voxels[x, y, z]_{mat}$
6:         end for
7:     end for
8: end for
```

The resulting test and validation images are shown in Figures 36 and 37. The differences between both images are shown in Figure 38. As the source and detector are positioned farther apart, the discrepancies in results progressively diminish, approaching zero. Notably, these differences are negligible near the center of the image and become more significant towards the edges. This behavior is attributed to the greater deviation from perfectly parallel lines that occurs towards the edges.

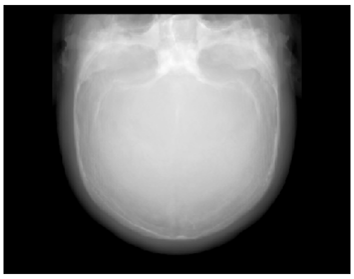


Figure 36: Result image of test scenario, values scaled to fill black/white spectrum

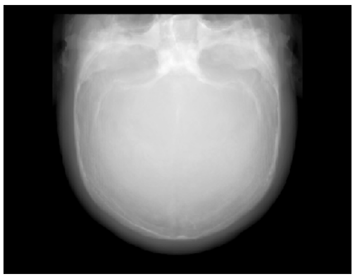


Figure 37: Resulting validation image, values scaled to fill black-/white spectrum

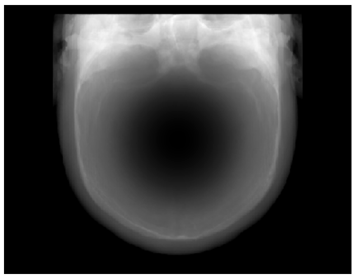


Figure 38: Difference between test and validation image, values scaled to fill black/white spectrum

When using a SID of 10000 and a SOD of 5000, with the model size being close to 10 x 10 x 10 (voxel size of $\frac{1}{40}$) the maximum value of the difference image in Figure 38 is only $1.6E - 5$ This is negligibly low, and thus this is a strong indicator of the validity and accuracy of the implementation.

5.5 Rotation

The source point and detector can rotate around the model. These rotations can be described using three coefficients: horizontal, vertical, and self-rotation. These three types of rotation are depicted in Figure 39 and are relative to the default positions of the source and detector.

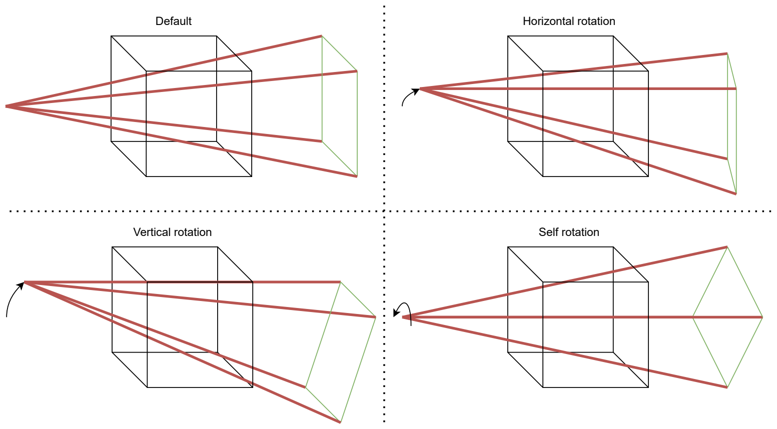


Figure 39: Horizontal, vertical, and self-rotation

An observation to note is that a maximum rotation of 45 degrees from the starting position can be imposed. If any of the rotations exceed 45 degrees, it is possible to adjust the orientation of the model in a manner that ensures the final coefficient for each rotation remains below 45 degrees. This constraint allows for effective control of the rotations while preserving the practical limitations of the system.

Different configurations for a single model can be computed before starting this algorithm. This involves rotating the model 45 degrees around any axis and recalculating the voxels such that they align with the axes. This ensures that, for any source and detector position, a model configuration exists where all rays traverse the model with an angle of less than 45 degrees. The maximum rotation for the source and detector can therefore be set to 22.5 degrees, as further rotation would lead to a rotation of the entire model. Since these models can be computed beforehand, the performance of the algorithm does not change.

6 Hardware implementation

This chapter describes the implementation of the projection step in hardware. It first explains the tools used in Section 6.1. Next, an overview of the complete architecture will be given in Section 6.2. Subsequently, the working of a single engine is explained in Section 6.3. Section 6.4 examines the communication between multiple engines. Finally Section 6.5 shows the validation tests that were done.

6.1 Experimental setup

Xilinx Vivado version 2021.2 was used to implement the algorithm in hardware, with the files coded in VHDL. The design is tailored for Ultrascale+ and Versal FPGAs.

6.2 Architecture

The implemented system is divided into several parts. The engines perform the actual computations, and they are the main focus of this research. However, the engine itself is not responsible for gathering the correct voxel data from the model memory. Each engine sends requests to an RRM (Read Request Manager), which communicates with an NMU (Network Management Unit) to deliver the correct model data via the NoC (Network-on-Chip). This data is then returned to the engine by the RRM. The complete architecture can be seen in Figure 40.

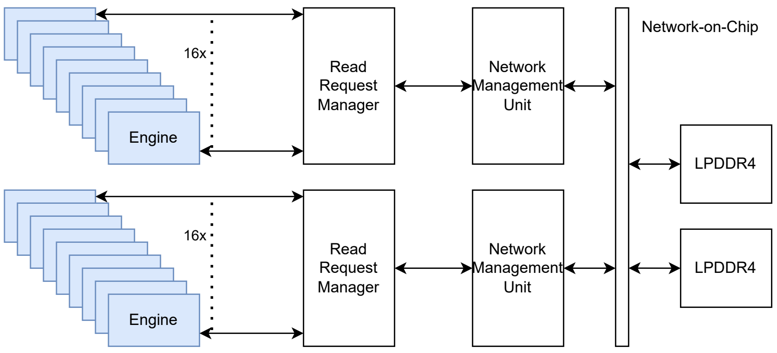


Figure 40: Complete architecture of the system

6.3 Engine overview

The engine is built out of a few different components. The URAM (UltraRAM) serves as the cache of the engine. The URAM manager computes the correct voxel address for the URAM and sends the received voxel data to the parameter RAM. This decodes the voxel data into a mass attenuation coefficient. The DSP manager accumulates the results. These components and their connections can be seen in Figure 41. A complete overview of the engine can be seen in Appendix A. Just as in the software implementation 8 layers per engine were chosen. As a result, a full computation of the skull model (384 layers), utilizes 48 engines.

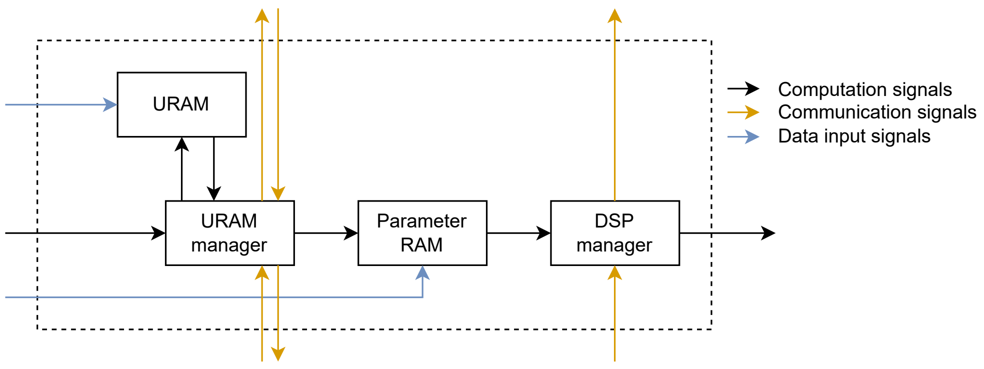


Figure 41: Simple overview of an engine

6.3.1 Ports

Each engine is equipped with external ports, as illustrated in Figure 42. These ports serve various purposes, including connections to other engines for cascading multiple engines. Some ports are designated for writing data to the URAM and Parameter RAM. Additionally, input ports are provided for the first engine in a cascade, while output ports are available for the final engine. Communication with the RRM is facilitated through the *rrm_* signals. Lastly, there are dedicated clock and reset ports.

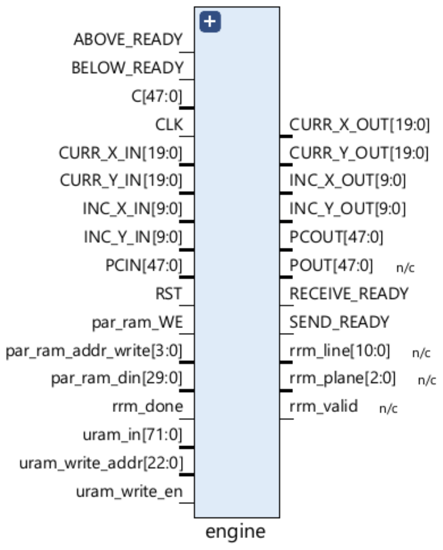


Figure 42: Engine ports

The specific function of each port will be elaborated on in the subsequent sections that delve into the components of the engine where these ports are utilized.

6.3.2 URAM

The URAM functions as the local memory of the engine, responsible for storing the buffers filled with model data. It is configured as a True Dual-Port RAM, allowing it to perform write operations and read operations for different addresses in parallel. An overview of the ports associated with the URAM can be found in Figure 43.

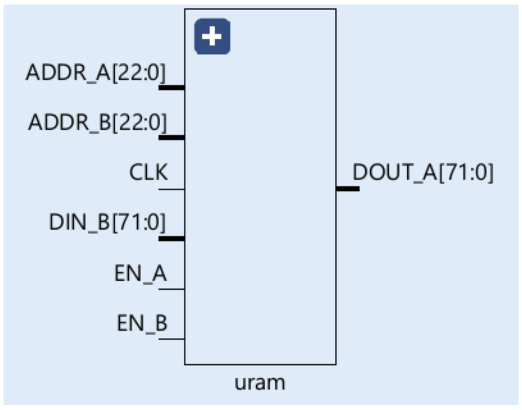


Figure 43: URAM ports

The writing operation to this memory is performed externally to the engine, as specified in the implementation details. On the other hand, the reading operation from this memory is managed by the URAM manager within the engine.

A single URAM module is 64 bits wide (or 72 bits if parity bits are active) and 2^{12} bits deep (12 address lines). This results in a total size of 262,144 bits ($64 * 2^{12}$). This size is twice the maximum possible memory requirement for the URAM module, ensuring it can accommodate the necessary data for the engine's operations.

This maximum memory requirement is computed in the following way:

$$4\frac{\text{bits}}{\text{voxel}} * 1024\frac{\text{voxels}}{\text{buffered line}} * 4\frac{\text{buffered lines}}{\text{layer}} * 8\frac{\text{layers}}{\text{engine}} = 131072\frac{\text{bits}}{\text{engine}}$$

6.3.3 URAM manager

The URAM manager communicates with both the URAM and the DSP manager and can be seen as the brains of the engine. It manages the signals to other engines and the RRM as well. The external ports are shown in Figure 44. These tasks are further detailed below, except for engine communication, which will be discussed separately in the following section.

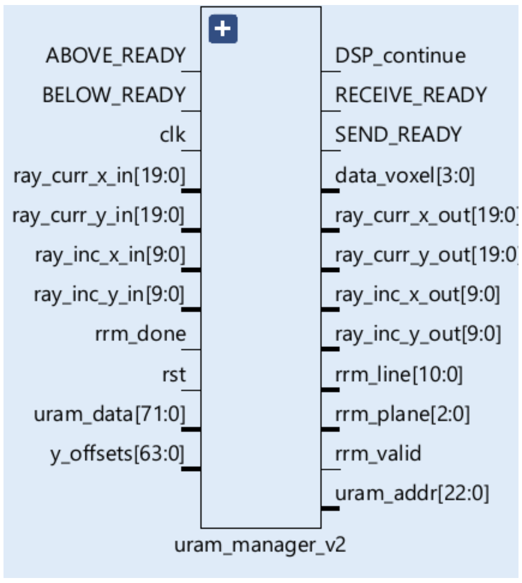


Figure 44: URAM manager ports

State of the engine

The URAM manager, and consequently the engine, can exist in several distinct states. These states include the reset state (when the reset input is high), the rec_prev state (when waiting for the previous engine to receive its output), the computation state (when reading data from URAM and actively engaged in computations), and the send_next state (when awaiting the opportunity to send data to the next engine). A visual representation of this state machine is depicted in Figure 45.

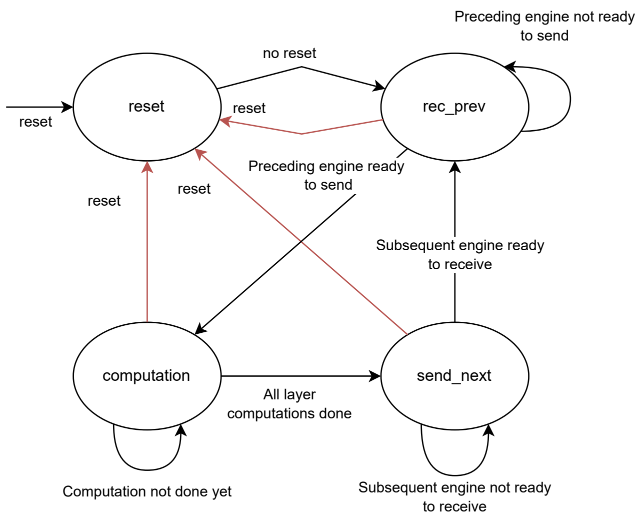

Preceding engine not ready to send
no reset
reset
reset
rec_prev
reset
reset
reset
Preceding engine ready to send
Subsequent engine ready to receive
computation
send_next
All layer computations done
Computation not done yet
Subsequent engine not ready to receive

Figure 45: State machine of the engine

Generation of read requests

When the first entry of any buffer is no longer required, a new read request is generated by configuring the *rrm_plane* signal to the layer number in need of a new entry and setting the *rrm_line* signal to the line number requested. To signal the validity of the request, the *rrm_valid* signal is momentarily set high for one cycle, informing the RRM that the request is valid and should be processed. To effectively manage the circular buffer, the engine uses buffer read pointers and buffer write pointers.

Only a single read request will be outstanding at a time. Only when this request is fulfilled by the RRM (when the *rrm_done* signal is set to high) a new request generation will be considered.

Read from URAM

The address for reading data from the URAM is determined by three variables: the X and Y coordinates of the ray and the current layer number. As reading from the URAM introduces a one-cycle delay, the data is available one cycle later. Given that the URAM is 64 bits wide excluding the unused parity bits, one entry contains data for 16 voxels. The appropriate voxel is selected based on the X coordinate of the ray, enabling precise data retrieval. This method ensures the correct data is obtained for further processing within the engine. The data is subsequently transmitted to the Parameter RAM through the *data_voxel* signal.

Communicate to DSP manager

The communication between the URAM manager and the DSP manager is relatively straightforward, as the DSP manager only requires a single bit of information to determine whether it should continue its operations or halt. When the DSP can proceed, the *DSP_continue* signal is set high, indicating that it can continue with its calculations. This minimalistic communication approach simplifies the coordination between these components.

Ray position calculation

To maintain accuracy during computations for each layer, the URAM manager updates the position of the ray after each layer computation. It receives the current position of the ray and its slopes (increments) from the previous engine via the *CURR_X_IN*, *INC_X_IN*, *CURR_Y_IN* and *INC_Y_IN* signals. For each next layer, the X and Y positions are incremented by their respective slopes. This ensures correct ray positions, which are then used to read from the URAM.

6.3.4 Parameter RAM

The Parameter RAM stores the fixed-point values for each possible voxel material. This serves as the implementation of the data compression method mentioned in Section 3.3. The Parameter RAM is designed using Look-Up Tables (LUTs) as RAM elements. Each LUT has a width of 6 bits, and multiple LUTs can be linked together to represent a wider range of values. A full Parameter RAM is shown in Figure 46.

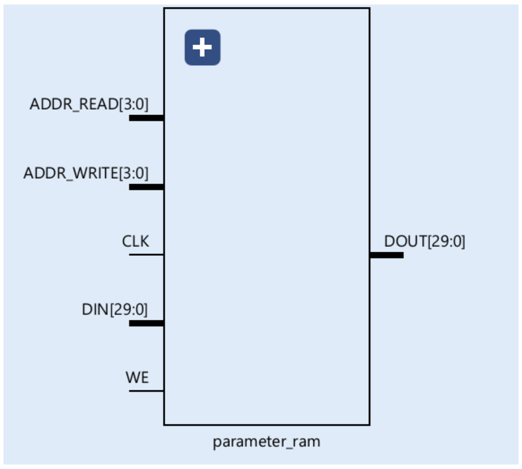


Figure 46: Parameter RAM ports

Unlike the URAM, a LUTRAM cannot be used as a True Dual Port RAM. It is designed as a Simple Dual Port RAM instead, but still has a read and write port. The write port allows external sources to set specific material values, while the read port is used internally to convert the *data_voxel* signal provided by the URAM manager into a mass attenuation coefficient for the DSP manager.

6.3.5 DSP manager

The DSP manager handles the actual computation. Its job is to correctly control the DSP slice. This is done by setting its operation mode to the right setting each cycle. The ports are shown in Figure 47.

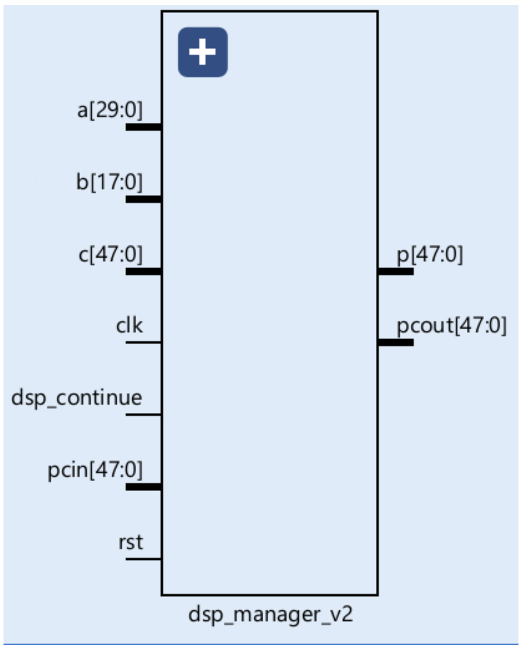


Figure 47: DSP manager ports

The DSP manager receives the value of the preceding engine via the internally routed *pcin* signal. An exception can occur when the DSP slice is the first slice in a chain. This will be discussed in Section 6.4. The material value enters via the *a* signal.

The continuation of computation depends on the *DSP_continue* signal, given by the URAM manager. One of three different computations are executed by the DSP slice:

1. **First layer when data ready:** Each engine is responsible for 8 layers. Therefore it needs to accumulate the result of the previous engine at some point during the computation. This is done during the first layer computation, which only happens if the required data for that calculation is available.
2. **Other layers when data ready:** If the voxel data is present, the intermediate result is incremented by the output of the current layer calculation.
3. **Not ready:** If the engine is waiting for new voxel data to arrive or for the preceding or subsequent engine the current value should be kept as is.

The final computed value will be communicated via the internal *pcout* signal, except when the engine is the last in a chain. In this case, the result is communicated via the *p* signal. A *data_valid* signal will be set when the *p* signal communicates a final pixel value.

6.3.6 Engine timing

One iteration of an engine without delays takes 10 clock cycles. In the URAM manager, these are represented as follows: First, a cycle to receive the values from the preceding engine, next 8 cycles to accumulate the final value of the engine, and then another cycle to communicate these values to the next engine. This iteration is shown in Figure 48.

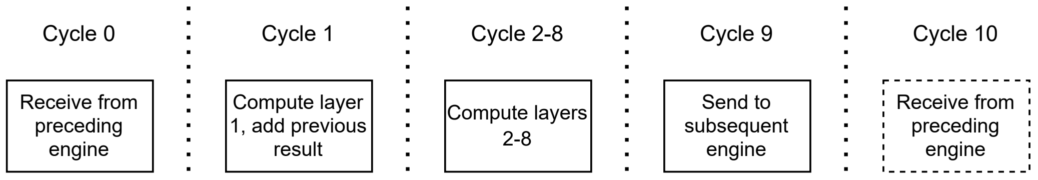

Cycle 0
Receive from preceding engine
Cycle 1
Compute layer 1, add previous result
Cycle 2-8
Compute layers 2-8
Cycle 9
Send to subsequent engine
Cycle 10
Receive from preceding engine

Figure 48: One iteration of an engine, consisting of 10 clock cycles if no delays are encountered

These cycles are repeated until either a delay is incurred (previous engine not ready, next engine not ready, voxel data not ready) or all rays are computed.

6.4 Multi-engine communication

Multiple engines have to communicate with each other. This is done via the *_READY* signals, of which a small overview can be seen in Figure 49. These are also further explained below:

- *ABOVE_READY:* This signal indicates that the subsequent engine is ready to receive the values from this engine.
- *SEND_READY:* If both this signal and the aforementioned *ABOVE_READY* signal are active, the data from this engine is communicated to the next engine.
- *RECEIVE_READY:* This signal indicates that the preceding engine is ready to send the values to this engine.
- *BELOW_READY:* If both this signal and the aforementioned *RECEIVE_READY* signal are active, the data from the previous engine is communicated to the current engine.

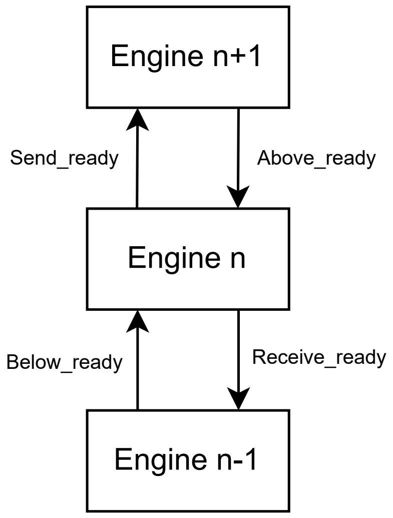


Figure 49: Communication ports of an engine

Since the receiving state of an engine corresponds to the sending state of the preceding engine, any next engine in a chain is exactly 9 clock cycles behind its predecessor.

Splitting cascade of engines

Every FPGA has a limit on the number of DSP engines that can be cascaded in a chain. If this limit is lower than the required limit of the model, the engines may not all fit in one continuous chain. In this case, the chain stops and continues at a different position. In this configuration, the cascading ports of the DSP engine cannot be used. All communication between the last engine before the split and the first engine after the split happens in the same way as explained before, except for the output value. This value is not communicated via the internally connected *PCOUT/PCIN* ports, but via the *POUT/C* ports instead. An example can be seen in Figure 50.

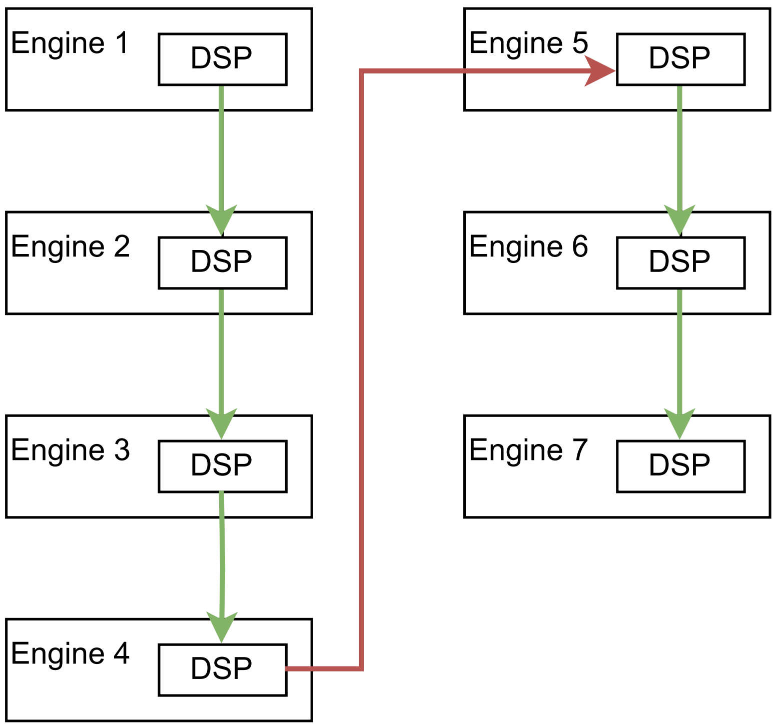


Figure 50: Engine configuration example with 7 engines; Green arrows represent the internal data bus, red arrows represent external data lines

6.5 Validation

To validate the hardware implementation, all configurations of engines were tested. Figure 51 shows these three test scenarios, which are also discussed below:

- **Start configuration:** This configuration validates communication between the first engine and a default engine. It ensures that the output of the first engine is correctly received by the next engine. This tests the initial part of the engine chain.
- **End configuration:** In the end configuration, the communication between a default engine and an end engine is tested. This verifies the proper functioning of the final engine and its ability to process data correctly. It also ensures that the final results are correctly generated.
- **Split configuration:** The split configuration tests communication between two engines that communicate the result via the $POUT/C$ ports.

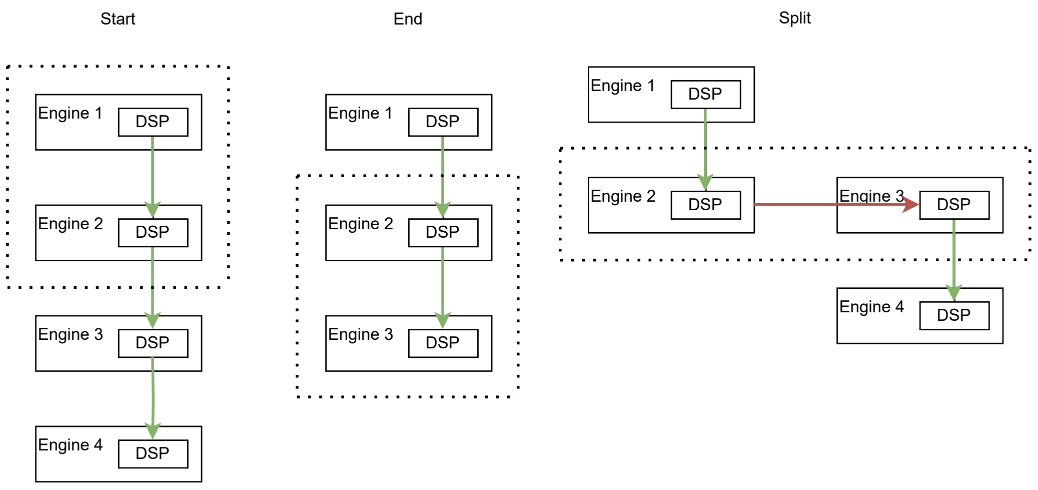


Figure 51: The three different test scenarios for validating engine functionality

Overall, these configurations provide a comprehensive validation of the hardware implementation, covering different scenarios and potential issues that may arise during operation. The results for the end configuration can be seen in Figure 53. This is a simulation of three engines, which means a model with 24 layers. The materials given to the voxels range from 1 to 16. This model and its materials can be seen in Figure 52.

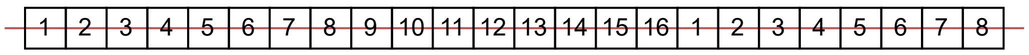
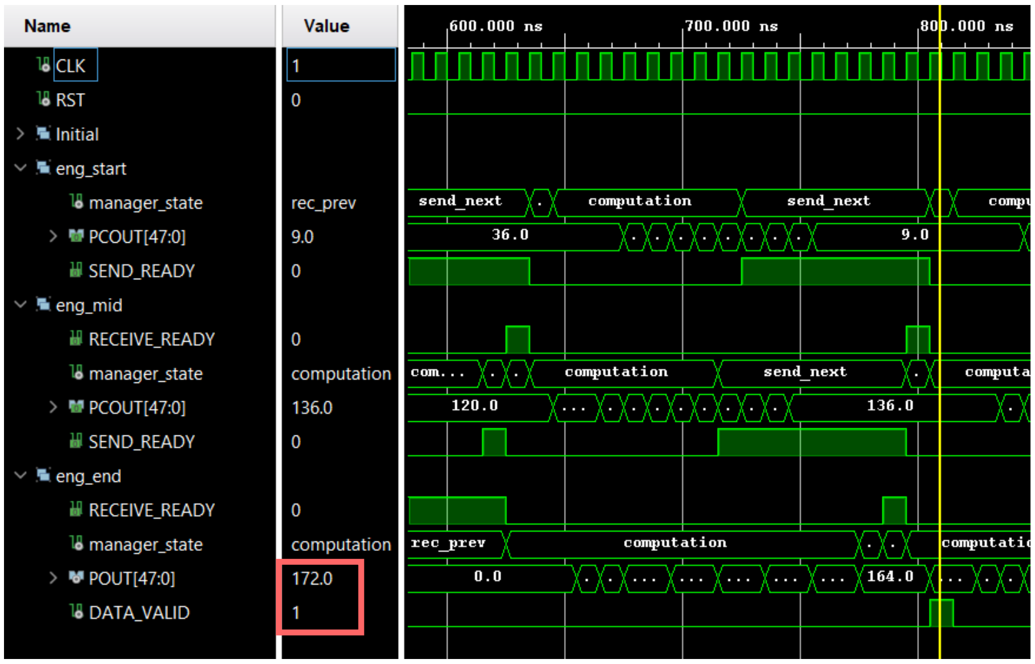


Figure 52: Model for engine simulation including the simulated ray: the materials add up to 172



Name	Value
CLK	1
RST	0
Initial	
eng_start	
manager_state	rec_prev
PCOUT[47:0]	9.0
SEND_READY	0
eng_mid	
RECEIVE_READY	0
manager_state	computation
PCOUT[47:0]	136.0
SEND_READY	0
eng_end	
RECEIVE_READY	0
manager_state	computation
POUT[47:0]	172.0
DATA_VALID	1

Figure 53: Simulation of the end configuration, where the final output of 172 is correct

6.6 Timing generation

The engine simulation is separate from the model memory simulation, due to practical reasons. Timing files are used to keep track of the interaction between the simulations. Three types of timing files are used: request files, access files, and acknowledgment files. Here is a summary of the content of each file:

- **Request files:** These files record when each engine issues a request to access data from the memory. It helps in tracking when each engine requires data. This information is used for simulating memory access patterns and analyzing potential delays in fulfilling requests.
- **Access files:** Access files provide information about when data is accessed for the first time by each engine. This reflects the actual latency experienced by each engine in retrieving data. Understanding the time it takes for data to be used after a request is made is essential for evaluating memory performance.
- **Acknowledgment files:** These files record the times when a request is acknowledged by the RRM, and the requested data is ready to be accessed by the requesting engine.

These files are structured as lines of timings. Each line contains comma-separated values which follow the following structure:

1. **Cycle number:** The timestamp in clock cycles that the action happens.
2. **Engine number:** The global engine ID of the engine that is involved. This can be computed from the global layer number
3. **Layer number:** The global layer ID of the layer that is involved.
4. **Line number:** The line number that is requested, acknowledged, or accessed.

The structured format of these timing files ensures that there are detailed records of when requests are made, when the requested data is available, and when data is accessed. These files are used as shown in Figure 54. First, the request and access files are generated using the aforementioned model simulation. The request files serve as input for the memory simulation. This results in acknowledgment files. These can be compared to the previously generated access files. If all data arrives on time, the simulation is complete. If however data is accessed by the engine simulation before it is available, the engines can be simulated again using these delays.

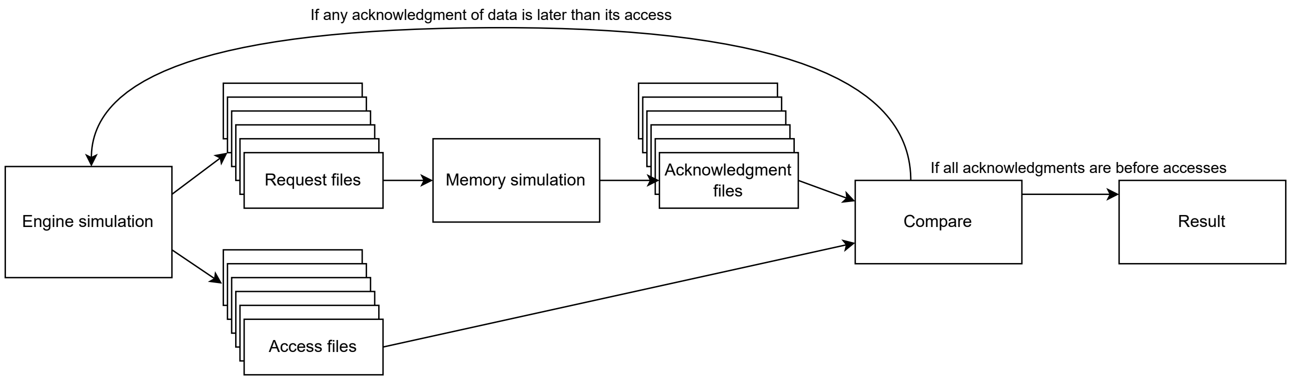

If any acknowledgment of data is later than its access
Engine simulation
Request files
Memory simulation
Acknowledgment files
If all acknowledgments are before accesses
Compare
Result
Access files

Figure 54: The structure of how the generated files are used to test for memory

To expedite the simulation process, a dedicated Python program was developed. This Python-based tool runs an algorithm closely resembling the main implementation but focuses exclusively on generating request and access times. It omits the computation of final values, streamlining the simulation process. The key objective of this program is to accurately derive the request and access time values, while significantly reducing simulation time. Validation was conducted to ensure that the timing values obtained from this Python program align with those produced by the full algorithm.

7 Measurement results

This chapter describes the achieved results. Section 7.1 examines the results of the algorithm simulation. In Section 7.2 the final theoretical performance is discussed. At last Section 7.3 describes the engine-memory interactions.

7.1 Simulation results

The software algorithm has been applied to the small skull model. An overview of the initial setup can be seen in Figure 55. Two different configurations have been tested. A projection pixel boundary of 4 was used for both configurations.

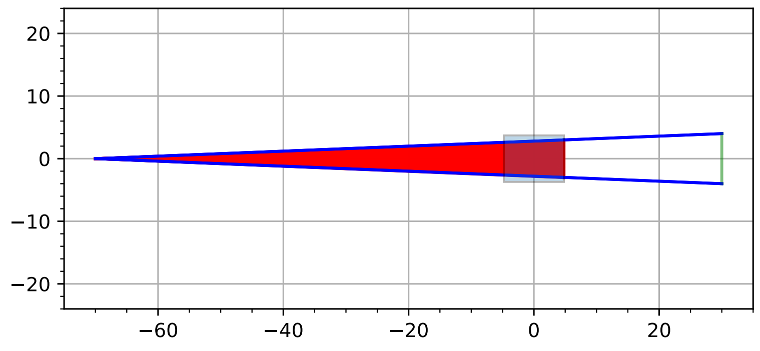


Figure 55: Overview of the simulation measurement setup

The difference between the configurations lies in the projection pixel size and detector resolution, as seen in Table 2. Due to the projection pixel size being different, the projection resolution changes as well.

	Configuration 1	Configuration 2
Projection pixel size	1/40	1/20
Projection resolution	247 x 247	487 x 487
Detector resolution	220 x 220	512 x 512

Table 2: Setup of both configurations

The software implementation is divided into a few substeps. The measured time for each substep for simulation of the small skull model is detailed in Table 3. Besides the times for each substep, the partial and complete pipeline times are also shown. The partial pipeline is defined as the time it takes to generate an image without the initial setup steps (setup of parameters and generation of voxels), as these setup steps are performed only once. The complete pipeline does include these setup steps.

Substep	Configuration 1	Configuration 2
Setup parameters	21μs	21μs
Generate voxels	95s	95s
Generate projection boundaries	9μs	9μs
Generate projection rays	138ms	540ms
Run engines	47s	173s
Calculate detector projection	130ms	153ms
Run Lanczos transformation	9.4s	80s
Partial pipeline for 1 image	57s	254s
Complete pipeline for 1 image	152s	349s

Table 3: Time needed per substep of the algorithm simulation for the small model

The timing generation algorithm can efficiently produce the required timing files. The corresponding times are listed in Table 4. To determine the time required for generating timing files for the original algorithm, the initialization and projection step were considered. The medium and large models are configured similarly to configuration 2, with a projection resolution of 487 x 487.

Model size	Configuration	Original algorithm	Timing generation time
384 x 297 x 384	1	142 s	0.59 s
384 x 297 x 384	2	269 s	0.68 s
512 x 512 x 512	-	-	0.90 s
1024x 1024 x 1024	-	-	1.78 s

Table 4: Time needed for the generation of timing files for the original algorithm versus timing generation algorithm

7.2 Hardware architecture results

The final engine design was synthesized and implemented on an Ultrascale+ FPGA. This process was conducted for the three configurations that were used for validation as explained in Section 6.5, along with a singular engine. The parameter RAM is configured to use 30 bits per material value. The final device resources utilized per configuration are depicted in Table 5.

Resource	Start configuration	End configuration	Split configuration	Single engine
LUTs	584	438	584	127
LUTRAMs	81	61	81	20
Flipflops	524	393	524	131
URAMs	4	3	4	1
DSP slices	4	3	4	1

Table 5: Device resources needed per configuration

Each engine uses exactly 20 LUTRAMs. This is due to using 30 bits per mass attenuation coefficient. Each 4 LUTRAMs can together save 6 bits of information, and thus a total of 20 LUTRAMS are needed per engine. A lower precision for this mass attenuation coefficient would save LUTRAMs. Additionally, one last LUTRAM is allocated for implementing the delay for the *data_valid* signal in the last engine of a chain. Most flip-flops and LUTs are used in the URAM manager, as that is the most complex component of the engine. Splitting an engine chain does not matter for the resource use. Only the routing is different. This aligns with the expectations.

The bottleneck for this implementation is the amount of URAMs needed. The Virtex Ultrascale+ series [21] and almost all Versal devices [23] have more than the required 128 URAMs available. Other resources are abundantly available on those FPGAs.

Theoretical performance

The initial output will arrive after the engine chain has completed computation for the first projection ray (9 cycles multiplied by the number of engines). After this initial setup every 10 cycles a new pixel of the projection result is generated. Using this information, the maximum theoretical performance for a specific configuration can be computed:

$$cycles\_per\_image = num\_proj\_pixels * 10 + num\_engines * 9$$

$$images\_per\_second = \frac{clock\_frequency}{cycles\_per\_image}$$

Table 6 shows the performance of the algorithm in frames per second (fps). This performance is calculated for both the Ultrascale+ (300 MHz) as well as the Versal (400 MHz) FPGAs. Three different projection resolution sizes were chosen, corresponding to a small, medium, and large detector.

	Small model (48 engines)		Medium model (64 engines)		Large model (128 engines)	
Projection resolution	Ultrascale+	Versal	Ultrascale+	Versal	Ultrascale+	Versal
220 x 220	619 fps	825 fps	618 fps	824 fps	618 fps	824 fps
512 x 512	114 fps	152 fps	114 fps	152 fps	114 fps	152 fps
1024 x 1024	29 fps	38 fps	28 fps	38 fps	28 fps	38 fps

Table 6: Achieved frames per second for all three models

Changing the model impacts the resulting performance only slightly, as this only determines the number of engines, affecting the startup delay until the first-pixel output. On the other hand, the projection resolution is a more significant factor in determining the resulting performance, as it defines the amount of iterations each engine has to go through.

7.3 Engine-memory interaction

The generated request files for the validation of the skull model were used to simulate the read request managers together with the network-on-chip. The current memory implementation does not perform well enough to achieve the timings necessary for this simulation. For preliminary results, these request files were relaxed by 2 (requests arriving at half the original rate). Subsequently, the resulting acknowledgment files were parsed, and the delays between requests and their acknowledgments are shown in Figure 56. It can be seen that at startup a lot of data is needed since the buffers of all engines need to be filled.

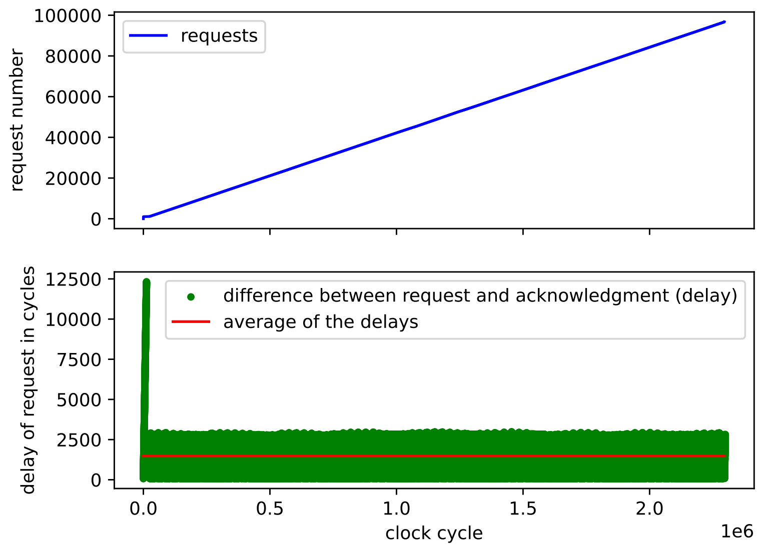


Figure 56: Request number and acknowledgment number over time (top), the difference between request time and acknowledge time (bottom)

Figure 57 illustrates a histogram representing the delays between requests and their corresponding acknowledgments. When only the requests after the initial setup are considered, the histogram looks like Figure 58.

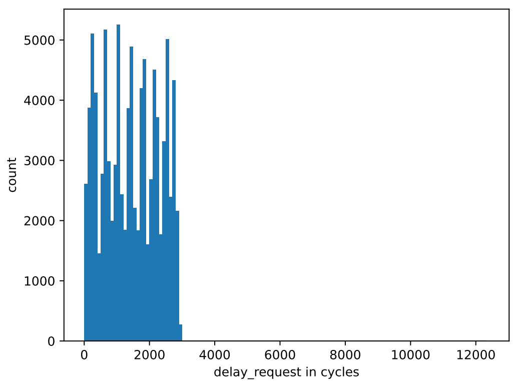


Figure 57: Histogram of acknowledgment delay including startup

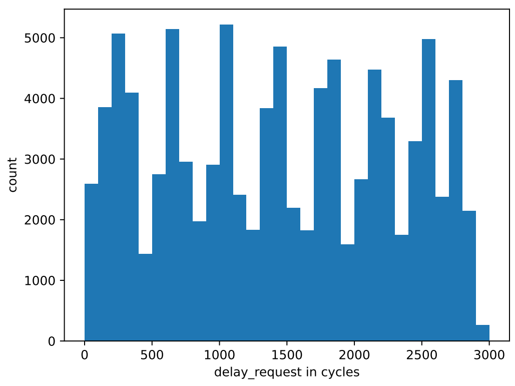


Figure 58: Histogram of acknowledgment delay excluding startup

Since the delay between requests and acknowledgments remains consistent over time, it can be concluded that running the engines with a relaxation of 2 does not overload the memory. The time between requests being sent and their corresponding data being accessed is 5940 cycles in the validation model (2 * 2970 cycles, 10 cycles multiplied by model width of 297 voxels). As the maximum request delay excluding the startup is smaller than the required 5940 cycles it indicates that the current state of the memory implementation is sufficient to run the engines at half speed.

8 Conclusions and recommendations

In this work, a hardware algorithm for synthetic X-ray image generation was proposed. This algorithm involves the division of calculation into a projection step and a detector step.

First, a naive approach to the X-ray image generation problem has been examined. Its two bottlenecks were identified: the large amount of intersection computations as well as the high required memory bandwidth. Next, the hardware-based solutions for these bottlenecks were investigated. This includes parallel processing, data compression, and data caching. Implementing these solutions led to a substantial improvement in the performance of the algorithm.

This algorithm was then simulated in software, where the algorithm itself was validated. A simplified and optimized version of the algorithm was used to generate the necessary timing files. At last a hardware implementation was developed and tested.

The first of four requirements that were set was image quality close to a real detector. It has been shown that this algorithm produces results that are close to a perfect detector. However, real detectors experience additional effects like noise and scatter, which are not currently integrated into the final algorithm design. Nevertheless, the design allows for the future inclusion of noise and scatter effects.

It was shown that all feasible positions are possible using this algorithm. The algorithm is limited in the possible configurations it can handle for a certain model. In cases where rotations surpass these limitations, an alternative rotated version of the same model can be used. As long as sufficient rotated versions of the model are available beforehand, all feasible positions of the system can be computed.

Large model support was another requirement. The algorithm has demonstrated support for large models. It was shown that the largest model requires 128 engines. Fortunately, the resources needed for these 128 engines are available on both the Ultrascale and Versal FPGAs.

At last the performance. This is closely related to the final research question that was posed: Therefore the following research question is posed: Can a hardware algorithm for real-time X-ray image synthetization achieve the image generation rate of existing detectors? The proposed algorithm can achieve this performance. Its maximum performance easily exceeds the required 60 images per second for a small and medium detector. Even when creating an image for a large detector, the algorithm can achieve an image rate exceeding 28 images per second.

Moving forward, the primary bottleneck for implementing this algorithm lies in the necessary memory bandwidth. Currently, even the smallest model requires its timings to be doubled for the memory to keep up. Increasing the memory performance will be a key focus for future research. Additionally, modeling scatter and noise in the algorithm to recreate the effects experienced by a real detector requires additional investigation.

References

[1] Wilhelm Burger and Mark J. Burge. *Principles of Digital Image Processing: Core Algorithms.* Springer, 2009.

[2] Punia Dheeraj. *FPGA design, architecture and applications [2023].* 2021. URL: `https://www.logic-fruit.com/blog/fpga/fpga-design-architecture-and-applications/`.

[3] Jennifer Dhont et al. "RealDRR – Rendering of realistic digitally reconstructed radiographs using locally trained image-to-image translation". In: *Radiotherapy and Oncology* 153 (2020). Physics Special Issue: ESTRO Physics Research Workshops on Science in Development, pp. 213–219. ISSN: 0167-8140. DOI: `https://doi.org/10.1016/j.radonc.2020.10.004`. URL: `https://www.sciencedirect.com/science/article/pii/S0167814020308409`.

[4] Osama Dorgham, Mark Fisher, and Stephen Laycock. "Accelerated generation of digitally reconstructed radiographs using parallel processing". In: Jan. 2009, pp. 239–243.

[5] Osama M. Dorgham, Stephen D. Laycock, and Mark H. Fisher. "GPU Accelerated Generation of Digitally Reconstructed Radiographs for 2-D/3-D Image Registration". In: *IEEE Transactions on Biomedical Engineering* 59.9 (2012), pp. 2594–2603. DOI: `10.1109/TBME.2012.2207898`.

[6] *Four Major Synthetic Data and AI Trends for 2022.* Mar. 2022. DOI: `10.1287/lytx.2022.02.06`. URL: `https://doi.org/10.1287/lytx.2022.02.06`.

[7] Thiago Franco de Moraes et al. "Medical image interpolation based on 3D Lanczos filtering". In: *Computer Methods in Biomechanics and Biomedical Engineering: Imaging & Visualization* 8 (Nov. 2019), pp. 1–7. DOI: `10.1080/21681163.2019.1683469`.

[8] Cong Gao et al. "Synthetic data accelerates the development of generalizable learning-based algorithms for X-ray image analysis". In: *Nature Machine Intelligence* 5.3 (2023), pp. 294–308. DOI: 10.1038/s42256-023-00629-1.

[9] Mahdi Hashemi. "Enlarging smaller images before inputting into convolutional neural network: zero-padding vs. interpolation". In: *Journal of Big Data* 6 (Nov. 2019). DOI: 10.1186/s40537-019-0263-7.

[10] J. H. Hubbell and S.M. Seltzer. In: *X-Ray Mass Attenuation Coefficients* (2004). DOI: https://dx.doi.org/10.18434/T4D01F.

[11] Andreas Maier et al. *Medical Imaging Systems: An introductory guide.* Springer open, 2018.

[12] Margherita Mottola et al. "Reproducibility of CT-based radiomic features against image resampling and perturbations for tumour and healthy kidney in renal cancer patients". In: *Scientific Reports* 11 (June 2021). DOI: 10.1038/s41598-021-90985-y.

[13] Fernando Pérez and Brian E. Granger. "IPython: a System for Interactive Scientific Computing". In: *Computing in Science and Engineering* 9.3 (May 2007), pp. 21–29. ISSN: 1521-9615. DOI: 10.1109/MCSE.2007.53. URL: https://ipython.org.

[14] Philips. *Azurion image-guided therapy systems.* URL: https://www.usa.philips.com/healthcare/resources/landing/azurion.

[15] *Projection X-ray Imaging.* URL: https://radiologykey.com/projection-x-ray-imaging/.

[16] Denis Prokopenko et al. "Unpaired Synthetic Image Generation in Radiology Using GANs". In: *Artificial Intelligence in Radiation Therapy.* Ed. by Dan Nguyen, Lei Xing, and Steve Jiang. Cham: Springer International Publishing, 2019, pp. 94–101. ISBN: 978-3-030-32486-5.

[17] Tasti. *Application-Tailored Synthetic Image Generation.* URL: https://tasti-project.eu/.

[18] Brian Teixeira et al. "Generating Synthetic X-ray Images of a Person from the Surface Geometry". In: *CoRR* abs/1805.00553 (2018). arXiv: 1805.00553. URL: http://arxiv.org/abs/1805.00553.

[19] Upstate Medical University. *Scatter Removal Grids.* URL: https://www.upstate.edu/radiology/education/rsna/radiography/scattergrid.php.

[20] Xilinx. *UltraScale: Managing Power and Performance with the Zynq UltraScale+ MPSoC.* URL: https://docs.xilinx.com/v/u/en-US/wp482-zu-pwr-perf.

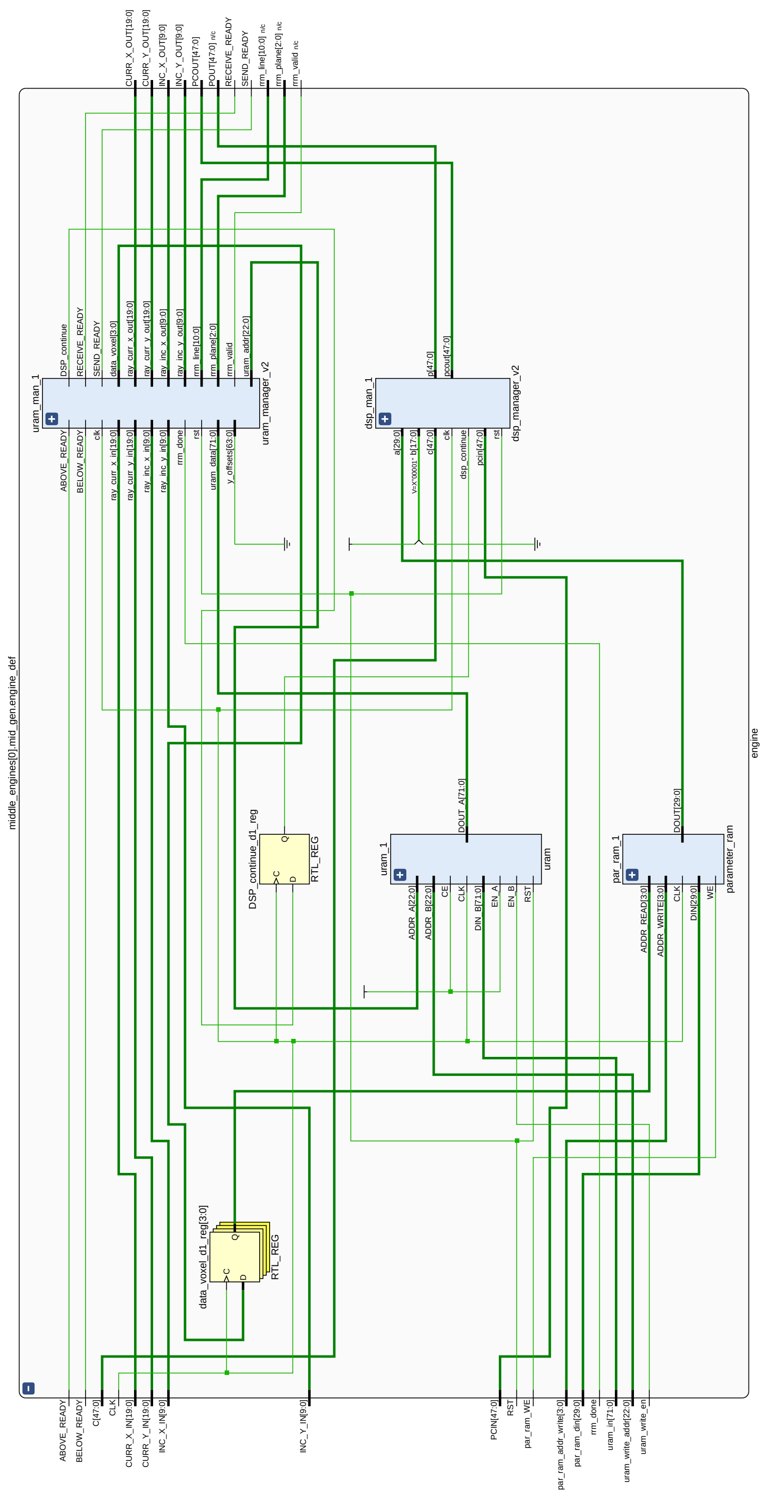
[21] Xilinx. *UltraScale+ FPGAs Product Selection Guide (XMP103).* URL: https://docs.xilinx.com/v/u/en-US/ultrascale-plus-fpga-product-selection-guide.

[22] Xilinx. *Versal ACAP DSP Engine Architecture Manual.* URL: https://docs.xilinx.com/r/en-US/am004-versal-dsp-engine.

[23] Xilinx. *Versal Architecture and Product Data Sheet: Overview.* URL: https://docs.xilinx.com/v/u/en-US/ds950-versal-overview.

[24] Xilinx. *Versal: The First Adaptive Compute Acceleration Platform.* URL: https://docs.xilinx.com/v/u/en-US/wp505-versal-acap.

A Expanded overview of engine



middle_engines[0].mid_gen.engine_def
engine
uram_man_1
uram_manager_v2
dsp_man_1
dsp_manager_v2
uram_1
uram
par_ram_1
parameter_ram
DSP_continue_d1_reg
RTL_REG
data_voxel_d1_reg[3:0]
RTL_REG
CURR_X_OUT[19:0]
CURR_Y_OUT[19:0]
INC_X_OUT[9:0]
INC_Y_OUT[9:0]
PCOUT[47:0]
POUT[47:0]
RECEIVE_READY
SEND_READY
rrm_line[10:0]
rrm_plane[2:0]
rrm_valid
ABOVE_READY
BELOW_READY
C[47:0]
CLK
CURR_X_IN[19:0]
CURR_Y_IN[19:0]
INC_X_IN[9:0]
INC_Y_IN[9:0]
PCIN[47:0]
RST
par_ram_WE
par_ram_addr_write[3:0]
par_ram_din[29:0]
rrm_done
uram_in[71:0]
uram_write_addr[22:0]
uram_write_en