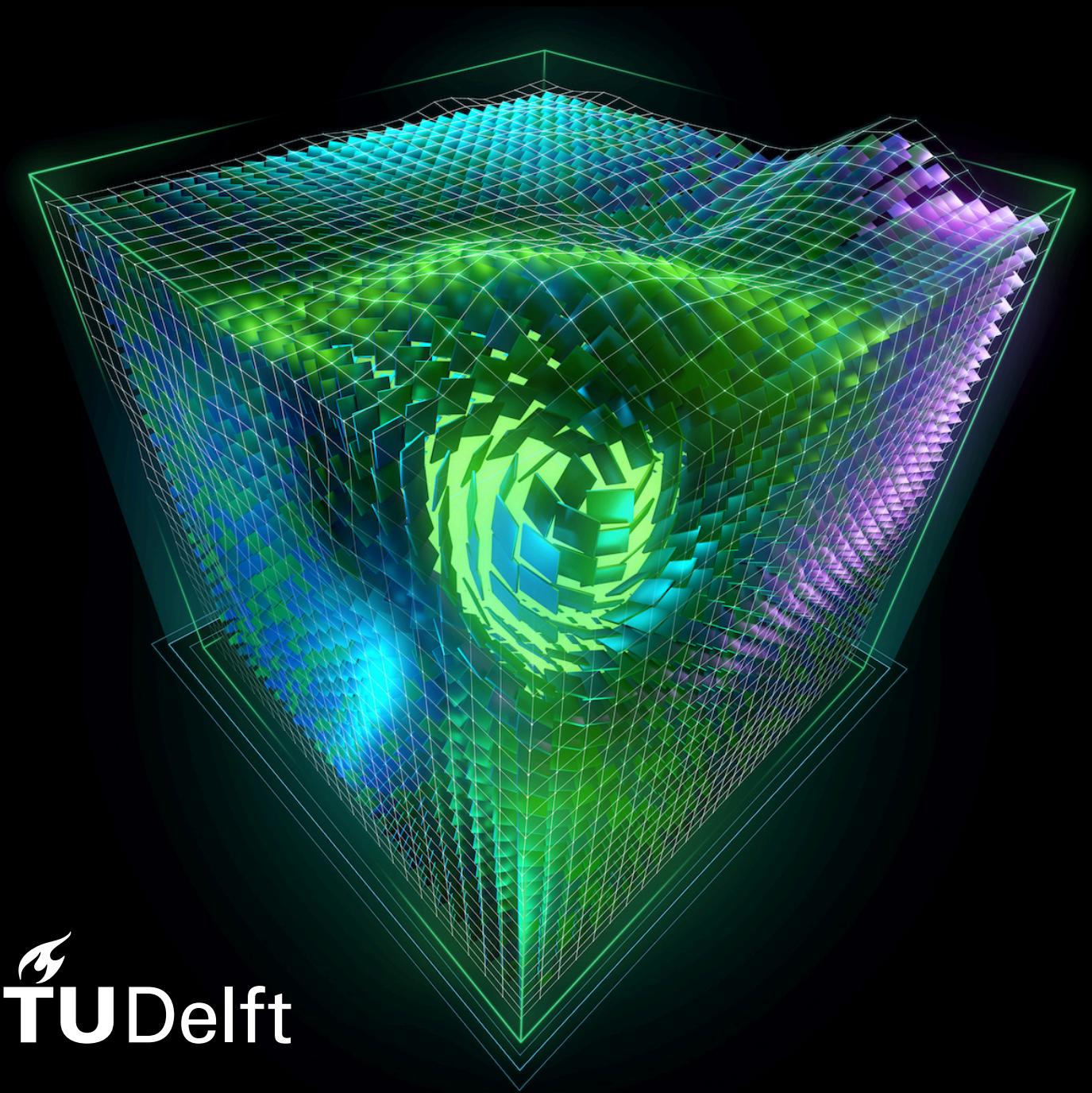
GPU Accelerated X-Ray Image Synthesis

Utkarsh Verma



Delft University of Technology

TU Delft

GPU Accelerated X-Ray Image Synthesis

by

Utkarsh Verma

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Tuesday February 24, 2026 at 11:00.

Student number:	5955424	
Project duration:	July, 2025 – February, 2026	
Thesis committee:	Prof. Dr. Harm Peter Hofstee	TU Delft, Supervisor
	Dr. rer. nat. Klaus Jürgen Engel	Philips, Supervisor
	Dr. Ir. Matthias Möller	TU Delft
	Dr. Ir. Zaid Al-Ars	External Advisor

Cover: “CUDA cube” by NVIDIA.

An electronic version of this thesis is available at http://repository.tudelft.nl.

TU Delft

Abstract

"There is no threshold dose below which radiation can be considered completely harmless."
— Hermann J. Muller

Although originally formulated for radiation in general, this statement applies to X-ray imaging as well, given its ionising nature. Even low-dose diagnostic procedures can damage DNA and carry cumulative cancer risk, which has led radiologists to minimise patient exposure whenever possible. These safety constraints limit the acquisition of large, diverse imaging datasets needed for developing, validating, and benchmarking modern medical imaging systems. They also restrict the number of projections that can be acquired in modalities such as computed tomography (CT), requiring accurate volumetric reconstruction from fewer scans and thereby increasing the technical demands on reconstruction algorithms.

Because acquiring realistic X-ray images is limited by patient safety, modern approaches first use sparse CT scans to reconstruct an accurate volumetric model of the object of interest. From this model, synthetic images can be generated through novel view synthesis, enabling large-scale offline datasets for machine learning, system testing, and model validation without additional radiation exposure. Beyond dataset creation, these synthetic projections can be produced in real time for interactive applications such as digital twins, used in virtual physician training and integration testing. However, generating high-fidelity synthetic images in real time remains challenging, given the substantial computational requirments of the algorithm.

This thesis investigates ways to accelerate X-ray simulation using graphics processing unit (GPU) implementations. Two techniques were developed: one based on voxelised models and another using Gaussian mixture models (GMMs). The approaches were evaluated in terms of visual fidelity and rendering performance, achieving ≈300 frames per second for voxel-based simulation and ≈40 frames per second for GMM-based simulation. Both techniques significantly reduce computation time compared to baseline CPU implementations, while maintaining realistic image quality suitable for virtual testing, physician training, and AI data generation.

These results demonstrate that GPU acceleration can enable real-time synthetic X-ray simulation, supporting scalable dataset creation and interactive applications while maintaining strict adherence to radiation safety principles.

Acknowledgements

I wish to thank my supervisors, Zaid, Peter, and Klaus (in no particular order), for their excellent supervision, their unconditional and unwavering support, and their willingness to engage deeply with all aspects of the project. While it is common for people to have a single supervisor, I feel fortunate to have benefited from three experienced individuals. Suffice it to say, I had all my bases covered. I am grateful for the opportunity to carry out my thesis project at Philips while enjoying full exploratory freedom. Thank you for your trust and making it possible, Klaus.

As with any project, there were ups and downs, and I am truly fortunate to have friends who helped me think clearly while also reminding me to experience all that life has to offer. For that, I would like to thank Nic, Arjun, and Junzhe (君哲).

Last, but not least, I want to express my gratitude my family for making it possible to be where I am today. I grew up watching my parents giving their all to ensure that no compromises were made in my upbringing. Having been firsthand witness to their efforts, I find it difficult to put into words how fortunate I feel to have them as my parents. I would also like to thank my elder brother for his guidance, encouragement, and steady presence throughout my academic journey.

To all these people, thank you once again. This work would not have been possible without your support.

Utkarsh Verma
Delft, Feb 2026

Contents

List of acronyms

3DGS – **3D Gaussian splatting** xi, xi, xi, 16, 47, 47, 47, 47, 48, 48, 49, 49, 51, 53, 53, 53, 53, 54, 55
AI – **arithmetic intensity** 21, 22
BVH – **bounded volume heirarchy** 44
CT – **computed tomography** iii, iii, 1, 1, 5, 11, 11, 12, 12, 12, 12, 12, 13, 15, 15, 15, 16, 16, 46, 46, 47, 63
DDA – **digital differential analyser** x, 32, 44
DRR – **digitally reconstructed radiograph** 15, 15, 15, 15, 15, 15, 46
FBP – **filtered back-projection** 12, 12
FLOPS – **floating-point operations per second** 21, 22, 22
FP32 – **single-precision floating point** vi, x, xi, 36, 36, 36, 37, 37, 37, 37, 37, 37, 37, 41, 41, 56, 61, 61, 61
FP64 – **double-precision floating point** x, xi, 36, 36, 37, 37, 40, 61, 61
FPS – **frames per second** 3, 4, 4, 15, 28, 28, 58
GMM – **Gaussian mixture model** iii, iii, vi, vi, xi, xi, 5, 5, 5, 5, 45, 45, 45, 46, 46, 46, 46, 46, 46, 46, 47, 47, 48, 49, 50, 51, 52, 53, 54, 55, 58, 58, 58, 59, 59, 63, 64
GPGPU – **general-purpose computing on graphics processing units** 16, 17
GPU – **graphics processing unit** iii, iii, v, v, v, vi, vi, vii, ix, 4, 4, 4, 5, 5, 15, 15, 16, 16, 16, 16, 16, 16, 17, 17, 17, 18, 18, 18, 20, 20, 20, 20, 21, 21, 21, 21, 21, 22, 25, 26, 26, 28, 28, 28, 29, 34, 34, 34, 34, 34, 34, 35, 35, 35, 35, 36, 36, 37, 38, 38, 39, 42, 43, 47, 61, 61, 63, 63, 63
LUT – **look-up table** vi, 41, 42, 42, 42, 42, 42, 42, 56, 56, 56, 61
MSE – **mean squared error** 23
PSNR – **peak signal-to-noise ratio** v, 22, 23, 23, 23, 23, 23, 23, 24, 25, 28, 29, 37, 37, 37, 57, 59, 60
SDD – **source-to-detector distance** 27
SIMT – **single instruction, multiple thread** ix, 16, 20, 20
SM – **streaming multiprocessor** ix, 17, 17, 17, 17, 17, 18, 18
SOD – **source-to-object distance** 27
SSIM – **structural similarity index measure** v, 22, 24, 24, 24, 24, 24, 25, 28, 29, 37, 37, 37, 57, 59, 60
TASTI – **Application-TAilored SynThetic Image generation** ix, 2, 2, 2
WSL2 – **Windows Subsystems for Linux 2** 26

List of figures

[1] https://commons.wikimedia.org/wiki/File:Color_X-ray_photogram.jpg
[2] https://commons.wikimedia.org/wiki/File:First_medical_X-ray_by_Wilhelm_R%C3%B6ntgen_of_his_wife_Anna_Bertha_Ludwig%27s_hand_-_18951222.jpg
[3] https://commons.wikimedia.org/wiki/File:Knee_plain_X-ray.jpg
[4] https://www3.ntu.edu.sg/home/ehchua/programming/opengl/CG_BasicsTheory.html
[5] https://www.arccompute.io/arc-blog/gpu-101-memory-hierarchy
[6] https://en.wikipedia.org/wiki/Peak_signal-to-noise_ratio#Quality_estimation_with_PSNR

[7]https://www.testdevlab.com/blog/full-reference-quality-metrics-vmaf-psnr-and-ssim
[8]https://developer.nvidia.com/blog/accelerating-openvdb-on-gpus-with-nanovdb/

1 Introduction

"Ionising radiation has sufficient energy to damage DNA within cells, potentially leading to mutations and an increased risk of malignancies" [13]. "Epidemiologic studies indicate that the radiation dose from even two or three CT scans results in a detectable increase in the risk of cancer, especially in children" [14]. These statements highlight "the" challenge in medical imaging: obtaining diagnostically useful X-ray scans while minimising patient exposure. As X-ray imaging remains a cornerstone of clinical diagnosis, from detecting fractures to identifying lung diseases and guiding interventions, there is a pressing need for high-quality imaging data to train and validate diagnostic tools safely.

High-quality X-ray datasets are essential for the development and evaluation of artificial intelligence-based models. Machine learning algorithms require large and diverse datasets to learn clinically relevant patterns, validate novel imaging techniques, and benchmark decision-support systems [15,16]. Beyond machine learning applications, such datasets are also widely used in industrial development workflows, where they support the design, calibration, and validation of new imaging systems. However, acquiring extensive real-world X-ray data is limited by radiation safety, patient availability, and privacy regulations. Consequently, conventional approaches to dataset collection are constrained, particularly when repeated scans pose potential harm or are ethically infeasible. CT faces similar limitations, creating a need to develop accurate models from fewer scans.

Several strategies have emerged to address these challenges. Low-dose imaging combined with AI reconstruction can improve image quality while reducing radiation exposure [15]. Synthetic X-ray image generation enables the creation of realistic, high-fidelity images from limited datasets, supporting model training, system validation, and physician training without additional patient risk [16]. Data augmentation and transfer learning further enhance dataset diversity, minimising the need for new scans [17,18]. Together, these approaches allow researchers to leverage large-scale imaging data safely, accelerating the development of AI tools while adhering to the principle of "as low as reasonably achievable" [19] in radiology practice.

1.1 Thesis scope

Figure 1.1: A hospital simulator with a C-arm machine for training physicians.

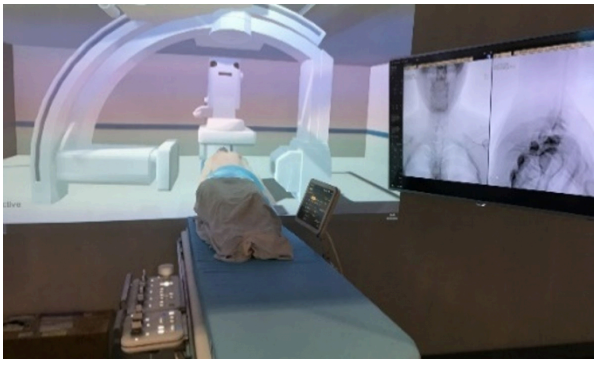


Figure 1.2: The virtual testing platform part of the TASTI project.

This thesis is part of an internship project at Philips Medical Systems and contributes to the European Application-TAilored SynThetic Image generation (TASTI) initiative [20], which aims to develop a modular framework for synthetic image generation across a diverse set of applications. For Philips, the TASTI project aims to create a virtual testing platform using synthetic image generation, with three primary applications:

1. **Physician training**: A virtual environment provides hands-on experience safely, without patient and trainee exposure, supporting risk-free and efficient training (see Figure 1.1).
2. **Equipment testing**: Synthetic images allow system integration tests without relying on an actual X-ray source or an X-ray system at all, reducing safety concerns and accelerating development.
3. **AI data generation**: Synthetic images enable large-scale dataset creation to train and validate computer vision algorithms for system evaluation and medical analysis.

1.1.1 The problem statement

Developing a fully physically realistic X-ray simulator that accurately models the non-idealities of the complete source-to-detector acquisition chain is an important and challenging research problem. However, given the limited availability of flexible X-ray simulation frameworks[9] and the practical constraints of this project, this thesis does not aim to model such physical complexities.

Instead, the focus lies on establishing a computationally efficient foundation for simulating an ideal conic-beam X-ray source governed by the Beer–Lambert law [1]. The primary objective is not to increase physical realism, but to investigate how such forward models can be engineered to achieve interactive performance on modern hardware.

This requirement is driven by the intended deployment context. The simulator operates as part of a digital twin used by Philips for testing systems in the Azurion series. The virtual detector must reproduce the temporal behaviour of a real imaging chain: control software and downstream applications expect an image stream with timing characteristics identical to those of physical hardware. Earlier simulation approaches were insufficiently fast to meet this constraint, resulting in timing mismatches and integration issues. Achieving true real-time performance, defined here as sustained

[9]There exist only a few publicly available forward projectors for X-rays and this topic is covered in detail in Section 2.3.

frame rates matching the physical detector, therefore becomes a functional requirement rather than merely a performance target.

With this focus, and taking into account additional project constraints from Philips, the following requirements were defined for the project:

- Render quality drops are acceptable as long as there are no *visible*[10] artefacts in the output.
- The input for the renderer does not have to be limited to voxelised models, and new file formats are permitted.
- The renderer must output an image in which each pixel represents the intensity perceived by the detector.
- The frames must be rendered at least as fast as the real X-ray machine being simulated[11], i.e., ≥ 60 frames per second (FPS).

1.2 What makes real-time X-ray simulation challenging?

Broadly speaking, simulating an X-ray image can be broken down into the following steps:

1. **Scene initialisation**: Positioning the source, object, and the detector, and specifying their geometries.
2. **Ray-casting**: Shooting a conical[12] X-ray beam towards the detector and accumulating the attenuation for each pixel of the detector according to Beer-Lambert law [1].
3. **Intensity calculation**: The accumulated attenuation can be applied to the source intensity to yield the perceived intensity for each pixel.

In this process, steps 2 and 3 are independent per-pixel operations and are the time-consuming spots for all algorithms. The render times are strongly dependent on the resolution of the detector and the level of detail encapsulated by the object being simulated.

Consider the simulation of a voxelised model[13] of $W \times H \times T$ voxels on a detector of $M \times N$ pixels. Such a simulation results in a total of $M \times N$ rays being cast from the X-ray source with each ray accessing $A = \min(W, H, T)$ or $A = \left\lfloor \sqrt{W^2 + H^2 + T^2} \right\rfloor$ voxels in their best (ray aligned to shortest axis) or worst (ray aligned to the diagonal) cases respectively. This highlights the following about the computation:

- Each render can be broken into $n_{\text{tasks}} = M \times N$ tasks that compute the intensity of each pixel.
- For a target frame rate F, this means $n_{\text{memory accesses}} = F \times A \times n_{\text{tasks}}$ memory accesses would be made per second.

[10] It is difficult to objectively quantify the visibility of artefacts. However, Section 2.5 later introduces image fidelity metrics that attempt to approximate perceptual quality.

[11] Machines from the Philips Azurion series. https://www.usa.philips.com/healthcare/brand/azurion-image-guided-therapy-system

[12] X-ray beams can also be parallel, but this thesis only consideres conical beams.

[13] Just 3D arrays of cubes called voxels. They are treated in detail later in Chapter 2 and Section 4.1.

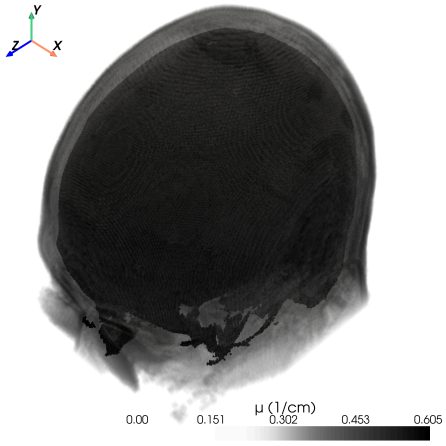


Figure 1.3: A voxelised model of the top half of a scanned human skull with $181 \times 178 \times 216$ voxels.

To put things into context, take the example of simulating the skull model from Figure 1.3 with $W = 181, H = 178, T = 216$ on a detector with $M = N = 1024$ for a target frame rate of $F = 60$ FPS. This means that:

- $n_{\text{tasks}} = 1024 \times 1024 = 1048576$.
- $n_{\text{memory accesses}} = 60 \times A \times 1048576$. In the best case, $A_{\text{best}} = 178$ which corresponds to a total of $n_{\text{memory accesses,best}} \approx 10$ GiB/s. In the worst case, $A_{\text{worst}} = 333$ which corresponds to at total of $n_{\text{memory accesses,worst}} \approx 20$ GiB/s.

Performing this simulation on the CPU without multithreading results in frame rates ≈ 0.8 FPS. To achieve the target frame rate of 60 would require a ≈ 76-fold speedup. For CPUs, a common optimisation would be to use multithreading and exploit the inherent parallelism of all n_{tasks} tasks. However, for the required speedup, even in the best case of a purely parallel task, a total of 76 cores would be needed which is impractical for the hardware used at Philips[14]. Looking at the memory accesses, it hard to estimate the potential speedup that can be achieved by optimising them. Still, it is known that they can't be optimised to the extent required for interactivity.

Recognising the algorithm's inherent parallel nature makes GPUs an obvious target for hardware acceleration, as they provide massively parallel execution and thrive on workloads where a large number of independent computations can be performed concurrently.

1.3 Research questions

Considering the criteria laid down in Section 1.1 and conducting a literature review helped identify the following research questions for this thesis:

1. What maximum performance can be achieved when porting the existing CPU-based X-ray simulation algorithm to the GPU?
2. What computational and memory bottlenecks arise in this GPU implementation, and how can alternative algorithmic designs or data representations alleviate them?
3. What trade-offs do different simulation approaches exhibit in terms of performance, memory consumption, and visual fidelity?

[14]Most developers only have workstations or laptops having 8-16 CPU cores

1.4 Contributions

To address the research questions from Section 1.3, this thesis makes the following contributions:

- A GPU-based implementation of an existing CPU X-ray forward projection pipeline, accompanied by a systematic performance characterisation. At the time of writing, TIGRE (CERN) [21] constitutes the only widely adopted open-source reference for voxel-based forward projection, making this work one of the few openly documented and industrially validated real-time GPU implementations in this domain.
- An in-depth analysis of computational and memory bottlenecks limiting real-time performance, including roofline-based evaluation and detailed profiling of global memory access behaviour.
- The development of two accelerated simulation approaches: one based on dense voxel traversal and another based on Gaussian mixture models (GMMs), each achieving speedups of $\approx$ 10x and $\approx$ 3x compared to their respective state-of-the-art implementations.
- The first investigation of GMMs as a primary 3D representation for X-ray forward simulation. While GMMs have been employed in CT reconstruction pipelines as intermediate or learned representations, their use as a first-class data representation for forward X-ray simulation has not been previously explored.
- A comparative evaluation of voxel-based and Gaussian-based approaches with respect to rendering performance, memory usage, compression characteristics, and visual realism.
- An assessment of the suitability of both representations for real-time digital twin applications in industrial X-ray simulation workflows.

1.5 Thesis outline

The remainder of this thesis is structured as follows. Chapter 2 provides an overview of X-ray imaging, relevant simulation techniques, and related work. Chapter 3 describes the benchmarking setup and profiling methodology used to evaluate performance. Chapter 4 and Chapter 5 detail the implementation of the voxelised and GMM-based simulation approaches, respectively. Chapter 6 presents a comparative evaluation of rendering performance and visual fidelity. Finally, Chapter 7 summarises the key contributions and discusses the implications of this work.

2
Background

This chapter discusses the required background knowledge for the thesis. It starts by explaining X-ray physics, the nomenclature, and imaging in Section 2.1. Then moving on to a brief introduction to computer graphics in Section 2.2 and the current state of X-ray simulation in Section 2.3. Thereafter, it concludes by covering the evaluation metrics in Section 2.5.

2.1 X-rays

Figure 2.1: A natural colour photogram where spectral assignments and sensitivity curves have been scaled and shifted from visible light to X-ray (12-55 pm, 22-103 keV). Courtesy: Wikimedia Commons[15].

X-rays are electromagnetic radiation and hence they transport energy through space using waves and photons just like radio waves, visible light and microwaves. As with all forms of light, X-rays are characterised by their frequency or wavelength. In literature, most sources define the wavelength range of X-rays to be between 10 picometres and 10 nanometres [1,22]. This corresponds to photon energies in the range of 100 eV up to 100 keV [1]. They were discovered in 1895 by Wilhelm Conrad Röntgen who named it X-radiation to signify an unknown type of radiation [23].

2.1.1 How they are generated

X-rays can be generated in two fundamentally different physical ways, namely as Bremsstrahlung and characteristic radiation.

2.1.1.1 Bremsstrahlung

Derived from the German words *bremsen* 'to brake' and *Strahlung* 'radiation', Bremsstrahlung typically involves the interaction of high-speed electrons with the anode of an X-ray tube. Electrons

[15] https://commons.wikimedia.org/wiki/File:Color_X-ray_photogram.jpg

are accelerated by the tube's acceleration voltage, moving them from the negative cathode to the positive anode. When these high energy electrons collide with the anode material, they are decelerated and deflected by the electric fields of the atoms in the anode. The deceleration produces X-rays with a continuous energy spectrum [1].

2.1.1.2 Characteristic X-rays

Characteristic X-rays, on the other hand, are produced when high-energy electrons eject inner-shell electrons from atoms in the anode material, creating vacancies. To stabilise, electrons from outer shells transition into these vacancies, releasing X-ray photons with energies equal to the difference between the two shell levels. These energies are unique to the atomic structure of the target material, resulting in a discrete spectrum with sharp peaks corresponding to specific transitions [1]. When the continuous and discrete spectra are combined, the spectrum shown in Figure 2.2 is produced. An X-ray beam consisting of radiation with multiple energy levels is referred to as polychromatic. Similarly, when only one energy level is present, it is called monochromatic.

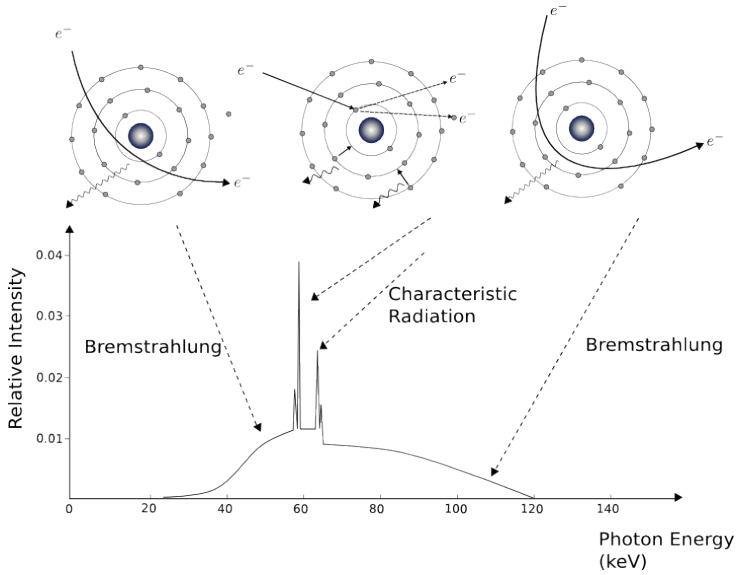


Figure 2.2: Energy spectrum produced by an X-ray tube with a tungsten target [1].

2.1.2 How they interact with matter

X-rays interact with matter through several physical mechanisms whose likelihood depends on both the photon energy and the atomic composition of the material. In the energy range relevant for diagnostic imaging, three interaction mechanisms are of primary importance: the photoelectric effect, Compton scattering, and Rayleigh scattering [1]. These interactions collectively determine the attenuation of an X-ray beam as it propagates through matter.

2.1.2.1 Photoelectric effect

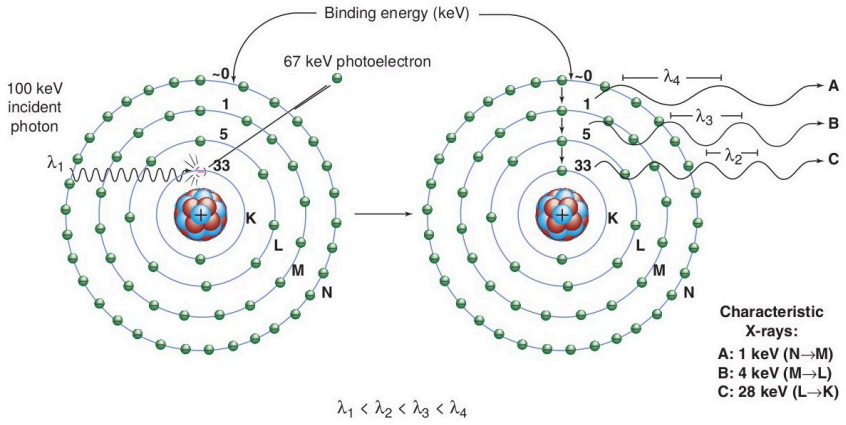


Figure 2.3: Photoelectric effect in action [2].

The photoelectric effect was originally described by Einstein [24] following the establishment of the quantised nature of light. It occurs when the energy of an incident X-ray photon exceeds the binding energy of an inner shell electron of an atom in the target material. In this interaction, the photon transfers its entire energy to the electron, which is subsequently ejected as a photoelectron. The incident photon ceases to exist.

The removal of an inner shell electron leaves the atom in an excited state. To restore stability, an electron from an outer shell transitions into the vacancy, releasing a photon with an energy equal to the difference between the two shell levels. This results in the emission of characteristic radiation. Consequently, the photoelectric effect produces a positive ion, a photoelectron, and a characteristic X-ray photon.

For tissue-like materials, the binding energy of K-shell electrons is relatively low compared to diagnostic X-ray energies. As a result, the photoelectron acquires nearly the full energy of the incident photon. The probability of photoelectric absorption strongly depends on the atomic number of the material and decreases rapidly with increasing photon energy.

2.1.2.2 Compton scattering

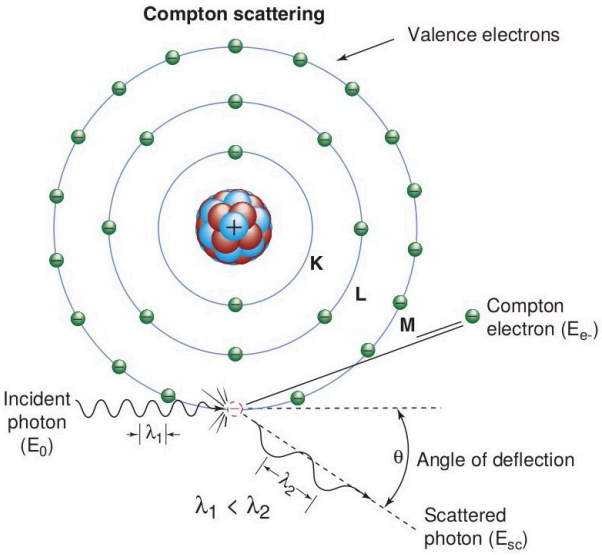


Figure 2.4: An illustration of Compton effect, where an incident photon strikes a valence electron, scattering the photon [2].

Compton scattering [25] is the dominant interaction mechanism for diagnostic X-ray energies in tissue-like materials. As illustrated in Figure 2.4, it occurs when an incident photon interacts with a weakly bound (valence) electron, transferring part of its energy to the electron and scattering the photon through an angle θ.

The scattered photon retains a significant fraction of its original energy, particularly for small scattering angles, while the "recoil" electron or Compton electron carries away the remaining energy. The interaction results in a positive ion, a recoil electron, and a scattered photon. The scattering angle can range from 0 to 180 degrees, with forward scattering being more probable at diagnostic energies.

Compton scattering does not remove photons from the beam entirely but redistributes their directions and energies, contributing to image degradation through scatter.

2.1.2.3 Rayleigh scattering

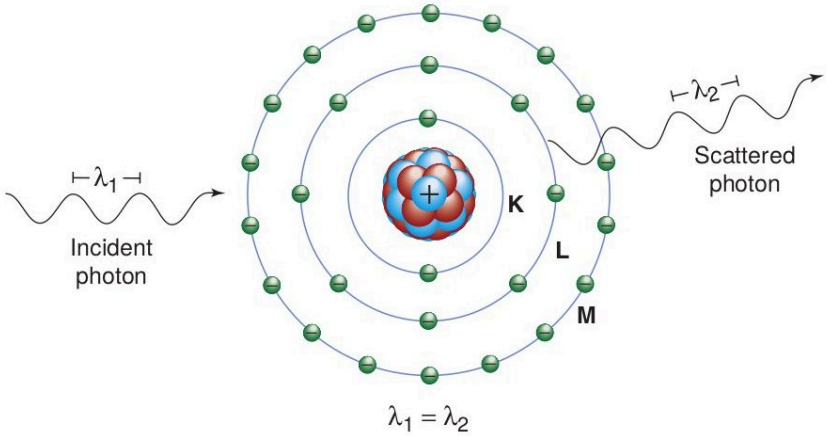


Figure 2.5: Rayleight scattering in action [2].

Rayleigh scattering is a coherent interaction that predominates at low X-ray energies. It arises from the interaction of an incident photon with bound electrons of an atom, typically involving multiple outer shell electrons simultaneously. In this process, no electron is ejected and no energy is transferred to kinetic energy.

Instead, the incident photon interacts coherently with the bound electron cloud of the atom. The oscillating electric field of the photon induces a collective polarisation of the outer electrons, creating a transient electric potential from which the photon is elastically scattered. The re-emitted photon retains the same energy as the incident photon but may propagate in a different direction. Because no ionisation or energy transfer occurs, Rayleigh scattering does not contribute to radiation dose in the same way as inelastic interactions. In diagnostic X-ray imaging, its contribution to total attenuation is generally small compared to photoelectric absorption and Compton scattering, and the scattered photons are predominantly emitted in the forward direction.

2.1.2.4 X-ray attenuation and Beer–Lambert law

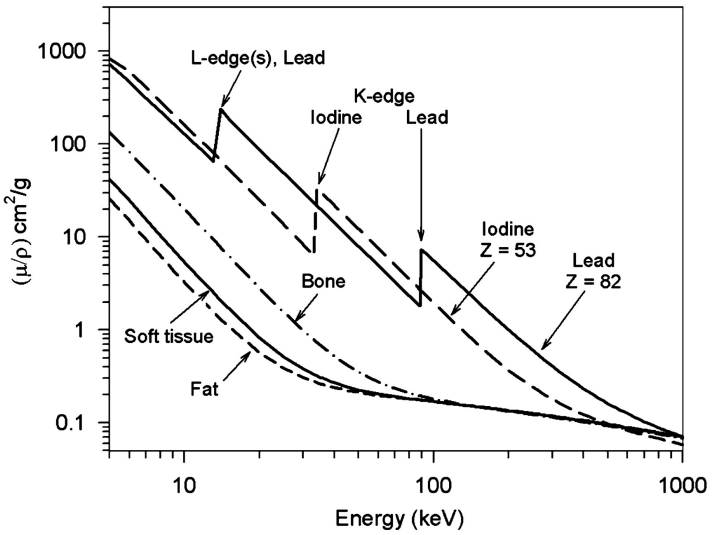


Figure 2.6: Mass attenuation coefficient μ/ρ of several materials as a function of energy [3]. From this plot, it can be determined that mass attenuation approximately $\propto \frac{Z^3}{E^3}$ where Z is the atomic number and E is the energy.

The combined effect of the interaction mechanisms described above leads to attenuation of an X-ray beam as it propagates through matter. This attenuation is quantified by the linear attenuation coefficient μ, which represents the cumulative probability per unit length of all relevant interaction processes. Figure 2.6 shows how the mass attenuation can change depending on the material and the X-ray photon energy.

For a monochromatic X-ray beam traversing a homogeneous material of thickness x, the transmitted intensity I is related to the incident intensity I_0 by the Beer–Lambert law [1]:

$$I = I_0 e^{-\mu x} \tag{2.1}$$

In heterogeneous objects, the attenuation coefficient varies spatially along the ray path. In this case, the transmitted intensity is given by the line integral of the local attenuation coefficient $\mu(x)$:

$$-\ln \frac{I}{I_0} = \int \mu(x)\,\mathrm{d}x \tag{2.2}$$

In X-ray CT, this logarithmic relationship forms the basis for reconstruction algorithms, where a large number of such line integrals are measured from different projection angles to recover the spatial distribution of μ.

2.1.3 Their use in medical imaging

Röntgen recognised the medical potential of X-rays almost immediately after their discovery in 1895. Within weeks of his initial experiments, he produced an image of his wife Anna Bertha Ludwig's hand (Figure 2.7), clearly revealing the bones and her wedding ring. This image is widely regarded as the first medical radiograph and demonstrated, for the first time, the ability to visualise internal anatomical structures non-invasively. Röntgen's rapid dissemination of his findings led to the adoption of X-rays in medical diagnostics within months, marking the birth of radiology as a clinical discipline.

Since then, X-rays have become a cornerstone of medical imaging, with applications spanning several distinct modalities, namely projectional radiographs, X-ray CT, and fluoroscopy.

Figure 2.7: First medical X-ray by Röntgen of his wife Anna Bertha Ludwig's hand. Courtesy: Wikimedia Commons[16].

2.1.3.1 Projectional radiographs and fluoroscopy

Figure 2.8: Plain radiograph of the right knee. Courtesy: Wikimedia Commons[17].

Projectional radiography is the most common use case of X-rays. It is the practice of producing a two-dimensional image (see Figure 2.8) by recording a single X-ray projection. This technique is routinely used for diagnosing fractures, detecting lung pathologies in chest radiographs, and identifying dental conditions. Differences in tissue density and atomic composition lead to varying attenuation of the X-ray beam, creating contrast between structures such as bone, soft tissue, and air-filled cavities.

Moving on, fluoroscopy extends projectional radiography by acquiring images continuously over time, effectively forming a dynamic time series. This enables real-time visualisation of anatomical motion and device interaction within the patient. Fluoroscopy is particularly valuable in interventional settings, where physicians must monitor the position of instruments during a procedure or visualize vascular structures following the injection of a contrast agent.

[16]https://commons.wikimedia.org/wiki/File:First_medical_X-ray_by_Wilhelm_R%C3%B6ntgen_of_his_wife_Anna_Bertha_Ludwig%27s_hand_-_18951222.jpg

[17]https://commons.wikimedia.org/wiki/File:Knee_plain_X-ray.jpg

2.1.3.2 Computed tomography

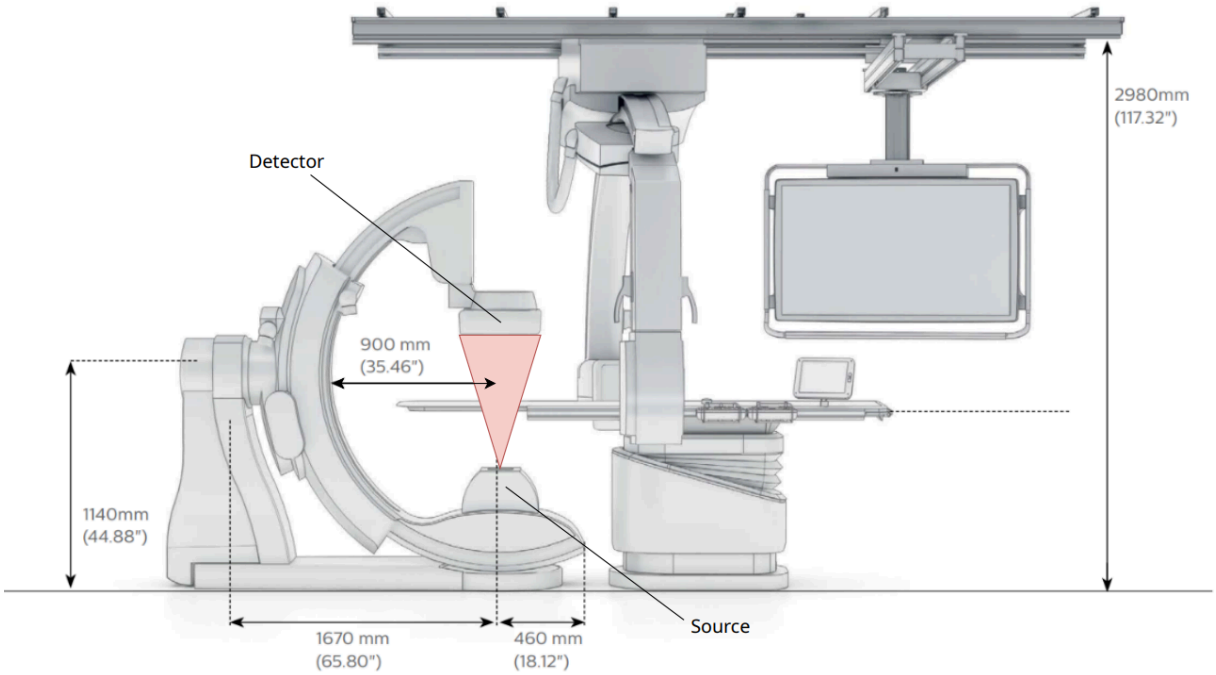


Figure 2.9: Philips Azurion 7 B20/15 biplane imaging system showing the location of the source and the detector, adapted from [4].

CT scanning is a medical imaging modality where tomographic images or slices of specific areas of the body are obtained from a large series of radiographs taken in different directions [26]. These cross-sectional images can be combined into a three-dimensional representation of the internal anatomy [27]. CT scans are a fast and cost-effective imaging modality used for both diagnostic and therapeutic purposes across a wide range of clinical disciplines [27].

A CT imaging system consists of an X-ray source and a detector arranged on opposite sides of the patient. During acquisition, the source–detector pair rotates around the patient, capturing a large number of X-ray projections at different viewing angles. Each projection records the line integrals of X-ray attenuation through the body along many ray paths. In modern systems such as the Philips Azurion 7 shown in Figure 2.9, this geometry supports both conventional fluoroscopy and cone-beam CT acquisition, enabling three-dimensional imaging in interventional environments.

The generation of three-dimensional images from these projection data is performed by tomographic reconstruction algorithms. One of the most widely used analytical methods is filtered back-projection (FBP), which reconstructs an image by first filtering each projection to compensate for frequency-domain distortions and then back-projecting the filtered data across the image domain [26,28]. FBP is computationally efficient and remains prevalent in clinical practice, particularly in applications where real-time or near-real-time reconstruction is required.

In addition to analytical methods, iterative reconstruction techniques are increasingly employed in modern CT systems. These methods formulate image reconstruction as an optimisation problem, iteratively refining an image estimate to minimise the difference between measured projections and simulated projections generated from the current estimate [26]. Iterative approaches can incorporate physical models of the imaging system, noise statistics, and prior knowledge, enabling improved image quality and reduced noise, particularly in low-dose imaging scenarios.

More recently, model-based and data-driven reconstruction methods, including those incorporating machine learning, have been explored to further improve reconstruction quality and reduce radiation exposure [27]. Regardless of the reconstruction approach, all CT algorithms rely fundamentally on accurate forward models of X-ray attenuation along rays. Consequently, efficient and physically accurate X-ray simulation plays a critical role not only in image formation but also in system design, algorithm validation, and the generation of synthetic projection data for research and development.

2.2 X-ray simulation: A computer graphics perspective

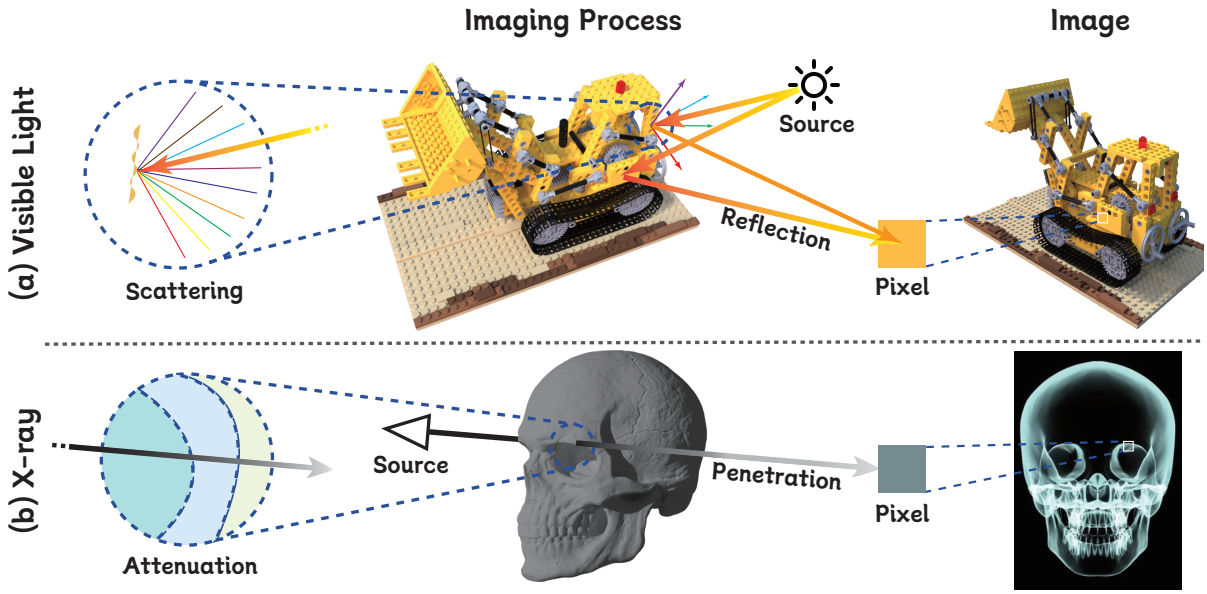


Figure 2.10: An illustration highlighting the difference between convention ray tracing (a) and X-ray tracing (b). Courtesy: X-Field [5].

Image synthesis in computer graphics is typically described as a simple scene abstraction: a camera observes a three-dimensional object and produces a two-dimensional image on a screen. Rays are cast from the camera centre through the image plane into the scene, where they interact with surfaces and materials to contribute to pixel values.

In conventional 3D rendering, the camera is decoupled from illumination as shown in Figure 2.10 (a). Light sources are positioned independently, and the pixel intensities are determined by how surfaces reflect or scatter light. As a result, the transport of light is typically modelled as a two-stage process: illumination from light sources to surfaces, followed by visibility from surfaces to the camera.

X-ray image simulation fits naturally into this geometric framework but differs in the physical interpretation of rays (see Figure 2.10 (b)):

- The light source is replaced by an X-ray source.
- The object encodes volumetric attenuation rather than surface reflectance.
- The image plane is replaced by a detector, typically a flat-panel detector.

This thesis considers only point X-ray sources emitting conical beams. Rays originate at the source, traverse the object, and terminate at the detector without intermediate surface interactions. Image formation is therefore governed by a single, unidirectional transport process, where each pixel records cumulative attenuation along its ray path.

The fundamental distinction from conventional rendering lies in the interaction model: optical rendering computes reflected light at surfaces, whereas X-ray simulation integrates attenuation through volumetric material. Recursive ray tracing and surface visibility tests are unnecessary, and effects such as reflection, refraction, or diffraction are negligible under typical diagnostic conditions (see Subsection 2.1.2).

In this context, X-ray imaging can be understood as a forward rendering problem: rays are cast from the source, and volumetric integration replaces surface shading. This correspondence establishes a geometric and computational foundation that is reused throughout the remainder of this thesis.

2.2.1 Ray generation

For each detector pixel, a ray is constructed that originates at the X-ray source and passes through the corresponding point on the detector plane. This mirrors the pinhole camera model commonly used in computer graphics, with the detector acting as the image plane.

Let $\vec{s}_w$ denote the source position in world coordinates and $\vec{p}_w$ the world-space position of a detector pixel. The ray direction is given by

$$\vec{d} = \frac{\vec{p}_w - \vec{s}_w}{\| \vec{p}_w - \vec{s}_w \|}, \tag{2.3}$$

and the ray is parameterised as

$$\vec{r}(t) = \vec{s}_w + t\vec{d}, \quad t \geq 0. \tag{2.4}$$

This construction yields exactly one ray per detector pixel.

2.2.2 Object-local transformations

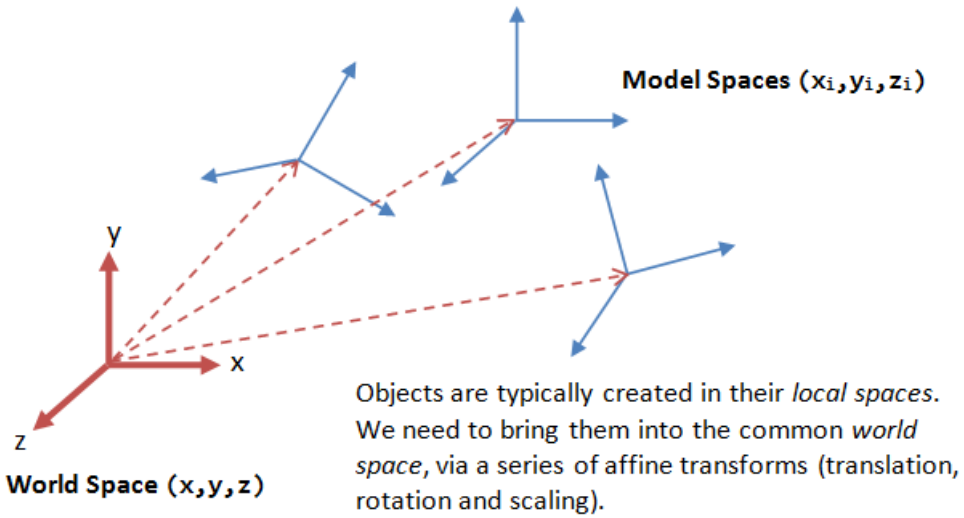


Figure 2.11: An illustration of the world vs model coordinate frames. Courtesy: yet another insignificant... programming notes[18].

The scanned object is defined in its own local coordinate frame, allowing it to be translated or rotated independently of the imaging system. Since the object's data is encoded in this local frame, rays or points defined in global coordinates must be transformed into object space using the inverse object transformation. As an example, consider the model spaces shown in Figure 2.11. To access

[18]https://www3.ntu.edu.sg/home/ehchua/programming/opengl/CG_BasicsTheory.html

a model's data, the model must be queried in its local coordinate frame $\vec{x}_l$. Hence, a point in the world space must $\vec{x}_g$ must first be converted as:

$$\vec{x}_l = T_{\text{obj}}^{-1} \vec{x}_g \tag{2.5}$$

where T_{obj} is the object-to-world transformation matrix. This formulation simplifies implementation and enables efficient evaluation of ray–object intersections and integrals, particularly when the object representation is static while the imaging geometry changes.

2.2.3 Volumetric integration along rays

As a ray traverses the object, it accumulates attenuation due to interactions with matter along its path. Under the Beer–Lambert law, the measured intensity I is related to the incident intensity I_0 by

$$-\ln \frac{I}{I_0} = \int \mu(x)\, \mathrm{d}x, \tag{2.6}$$

where $\mu(x)$ denotes the spatially varying linear attenuation coefficient.

The numerical evaluation of this line integral depends on how the object is represented internally and on the chosen integration strategy. These aspects are independent of the geometric formulation presented here and are addressed in subsequent chapters.

2.3 X-ray simulation: The current state

Early synthetic X-ray image generation techniques date back to at least the mid-1990s. The concept of digitally reconstructed radiographs (DRRs), that is, simulated transmission images produced by ray integration through a volumetric CT dataset, was established in the context of CT-based virtual simulation and treatment planning as early as 1994–1995, where fast trilinear interpolation algorithms were used to reformat CT scans into arbitrary DRR projections for clinical use [29].

Over the past two decades, DRRs have become a primary method for generating synthetic X-ray images from volumetric datasets obtained from CT or reconstructed rotational X-ray scans. Classical methods cast individual rays through the volume and integrate attenuation along each ray to produce projection images; while accurate and physically grounded, these techniques are computationally expensive and memory-bound for high resolution volumes. To address this, [30] introduced the use of attenuation fields to accelerate DRR generation, precomputing extensive light-field information to significantly reduce computation compared to conventional precomputed DRR tables. Later, GPU acceleration became a natural fit for DRR pipelines: [31] reported roughly 100 FPS for 512x512 renders using algorithmic simplifications and specialized ray-casting, and [32] further optimised DRR rendering on GPUs, achieving 190–370 images per second on commercial devices, though these figures correspond to limiting the region of interest to 1024 pixels.

At the time of writing, only a handful of open-source X-ray simulators are available. Most of these tools are personal or research projects that focus on polygonal or mesh-based geometries rather than voxel models. Examples include xraySimulator [33], which operates on STL meshes but is no longer maintained; Xray_Sim [34], a MATLAB-based personal project also relying on STL meshes; XRaysim [35], and gVirtualXray [36], both supporting polygonal mesh models; and SYRIS [37], which accommodates both geometric and mesh-based representations. While these simulators can generate realistic X-ray projections from surface meshes, none natively support voxelised volumetric models.

For volumetric X-ray simulation, the only widely used open routine is TIGRE [21], developed primarily as a toolbox for CT reconstruction at CERN. TIGRE allows users to perform forward

projection on voxelised volumes, but it is not a dedicated X-ray simulator: documentation is minimal, and users are generally expected to explore the source code and examples to understand its functionality. Despite these limitations, TIGRE remains a common choice in research for simulating projections from volumetric datasets.

In recent years, researchers have explored new ways to represent volumes that make X-ray simulation and CT reconstruction faster and more flexible. Traditional forward projection uses numerical integration over voxel grids, which can be slow and memory-intensive, especially for large datasets or real-time applications. Inspired by neural rendering, Gaussian-based volumetric representations have emerged as an alternative. Instead of storing values in fixed voxels, these methods represent the volume as a set of 3D Gaussian "blobs" that can be combined and projected efficiently.

Some key approaches include:

- **Radiative Gaussian Splatting** [38] – represents the volume with 3D Gaussian points and uses a differentiable rasteriser[19] to quickly generate X-ray projections. It is faster than earlier neural approaches and works well for sparse-view CT data.
- **R²-Gaussian** [9] – improves the standard Gaussian splatting by adjusting the shape of the Gaussians to reduce errors when projecting, enabling more accurate and efficient reconstruction from limited angles.
- **X-Field** [5] – models material-dependent X-ray attenuation using 3D ellipsoids and efficient path partitioning, achieving high-fidelity novel views and CT reconstruction.

Overall, these methods move away from rigid voxel grids and instead use continuous, flexible representations. This makes forward projection faster, more memory-efficient, and naturally compatible with differentiable reconstruction, opening the door for modern GPU-accelerated algorithms and machine-learning-based approaches.

Parallel work on volumetrically consistent 3D Gaussian rasterisation [39] has shown that analytic integration of Gaussian primitives can yield physically accurate transmittance computations suitable for tomography, bridging the gap between efficient rasterisation methods and volumetric physics-based projection integration.

Together, these developments illustrate a trend toward continuous, differentiable forward models that depart from discrete voxel grids, instead leveraging parametric or primitive-based representations to improve computational efficiency, enable differentiable reconstruction, and better exploit modern GPU architectures. In this thesis, forward projection is investigated for both traditional voxelised volumes and Gaussian mixture representations, highlighting the computational trade-offs, performance implications, and opportunities for real-time synthetic X-ray simulation.

2.4 Graphics processing unit (GPU)

A GPU is a specialised processor originally developed for graphics and now widely used for general-purpose computation. NVIDIA GPUs follow a single instruction, multiple thread (SIMT) execution model, where groups of threads execute the same instruction across many data elements while supporting multithreading to hide memory latency and expose parallelism [40]. This execution model is suitable for algorithms that can be massively parallelised and run on general-purpose GPUs or general-purpose computing on graphics processing units (GPGPUs). This section will

[19] A rasteriser is a 3D Gaussian splatting (3DGS) term for something that takes in a set of Gaussians and renders a 2D image.

briefly describe the general architecture of NVIDIA GPUs and how to use them as GPGPUs. We will also describe some practical techniques available through this interface.

2.4.1 GPU architecture

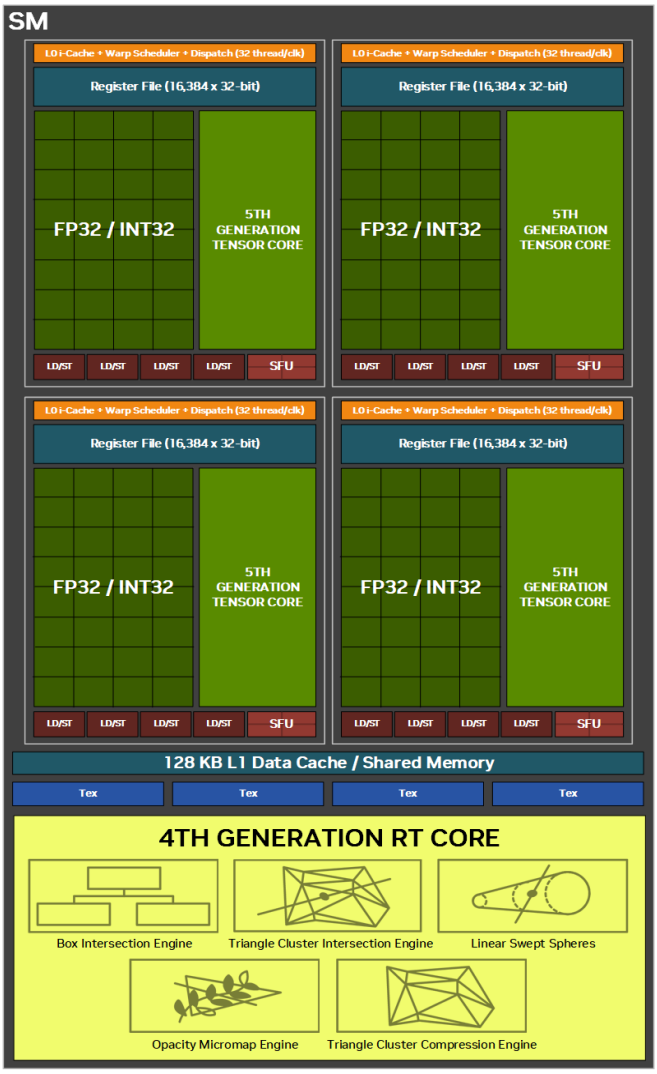


Figure 2.12: NVIDIA Blackwell streaming multiprocessor architecture [6].

At the core of GPUs lie many small, various cores (CUDA, Tensor, RT), which enable massive parallelism. The different cores have different specialties, but CUDA cores are the standard processing cores. These cores are grouped as streaming multiprocessors (SMs), with each having their own schedulers, register files, and caches. Figure 2.12 shows the architecture of SMs in NVIDIA's latest Blackwell chips.

The SMs can execute multiple threads simultaneously, achieving high throughput through parallelism. Threads running on an SM are grouped into thread blocks and into warps, which run in lockstep. This means all threads execute the same instructions, potentially leading to inefficiencies if there is divergence between threads in the same warp.

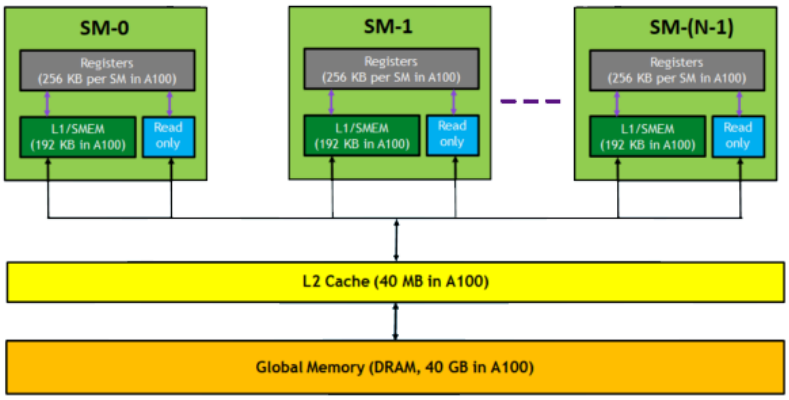


Figure 2.13: Memory heirarchy of an NVIDIA A100 40 GB GPU. Courtesy: Arc Compute[20].

Similar to a CPU, GPUs have a hierarchical memory architecture (see Figure 2.13). The largest and slowest type of memory is global memory. This memory is accessible to all threads and is relatively plentiful, but it has the largest access latency and stricter requirements for optimal bandwidth utilisation. The next layer is the L2 cache, which can reduce latency for frequently accessed memory. Then comes the L1 cache, which functions as shared memory within a thread block. This memory is located on the SMs themselves. It can be used to communicate between threads on the same SM and to cache intermediate results before issuing expensive instructions to global memory. Memory that is not required by the compute load can be used as a regular L1 cache. Finally, there is the register file, which is used directly by the threads.

2.4.2 Coalesced access

Since GPUs employ a hierarchical memory architecture, efficient utilisation of global memory bandwidth is critical for performance. In CUDA, threads are executed in groups of 32 known as warps. When threads within a warp access global memory, their memory requests are combined into as few memory transactions as possible. This process is known as *memory coalescing*.

Global memory is serviced in aligned memory segments (commonly referred to as sectors). On modern NVIDIA GPUs, memory transactions operate on 32-byte sectors, which may be combined into larger 128-byte requests depending on the access pattern. The number of memory transactions required for a warp depends on two factors:

- The size of the data type accessed by each thread.
- The distribution and alignment of memory addresses across the warp.

[20]https://www.arccompute.io/arc-blog/gpu-101-memory-hierarchy

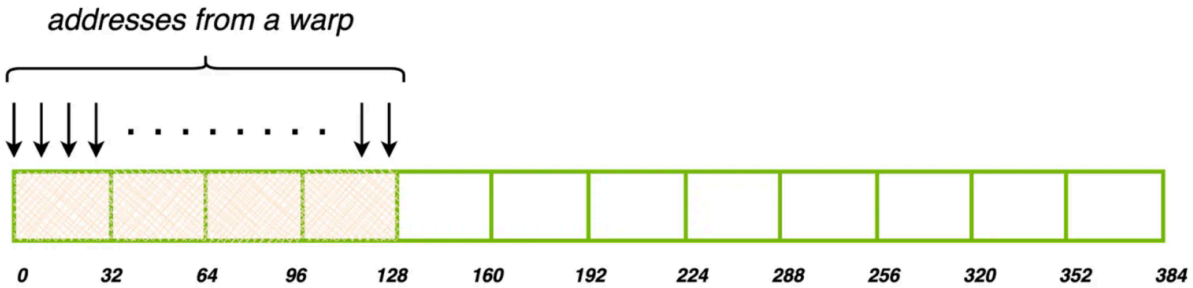
```
__global__ void uncoalesced_access(float* input, float* output, int n) {
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    if (tid < n) {
        // Access with a stride of 32 (128 bytes), wrapped around to stay within
bounds
        int scattered_index = (tid * 32) % n;
        output[tid] = input[scattered_index] * 2.0f;
    }
}
```

Listing 1: Uncoalesced memory access pattern showing each thread (arrow) accessing data in a separate 32-byte memory sector. Courtesy: NVIDIA Technical Blog[21].

In the uncoalesced example shown in Listing 2.1, each thread accesses memory with a large stride. As a result, the addresses requested by threads in a warp fall into different 32-byte sectors. Instead of servicing the warp with a small number of aligned transactions, the hardware must issue many separate memory transactions. For each 4-byte value requested by a thread, an entire 32-byte sector may be fetched, with most of the transferred data unused. This leads to poor bandwidth utilisation and increased memory latency.

By contrast, coalesced access occurs when consecutive threads access consecutive memory locations. The kernel in Listing 2.2 demonstrates this optimal access pattern:

```
__global__ void coalesced_access(float* input, float* output, int n) {
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    if (tid < n) {
        // Each thread accesses consecutive 4-byte words.
        output[tid] = input[tid] * 2.0f;
    }
}
```



Listing 2: Coalesced memory access pattern showing the threads (arrows) of a warp accessing a contiguous 128-byte memory region in four 32-byte sectors. Courtesy: NVIDIA Technical Blog[22].

[21] https://developer.nvidia.com/blog/unlock-gpu-performance-global-memory-access-in-cuda/
[22] https://developer.nvidia.com/blog/unlock-gpu-performance-global-memory-access-in-cuda/

In this kernel, consecutive threads in a warp access consecutive 4-byte elements of memory. A full warp therefore accesses 32×4 bytes $= 128$ bytes of contiguous data. This access can be serviced using four aligned 32-byte sectors, minimising the number of memory transactions. The hardware efficiently aggregates the memory requests, resulting in near-optimal global memory bandwidth utilisation.

For memory-bound workloads such as voxel traversal or forward projection through large attenuation volumes, ensuring coalesced access is often the single most important optimisation. Poor access patterns can degrade effective bandwidth by an order of magnitude, whereas properly aligned and contiguous accesses allow the GPU to approach peak memory throughput.

2.4.3 Thread divergence

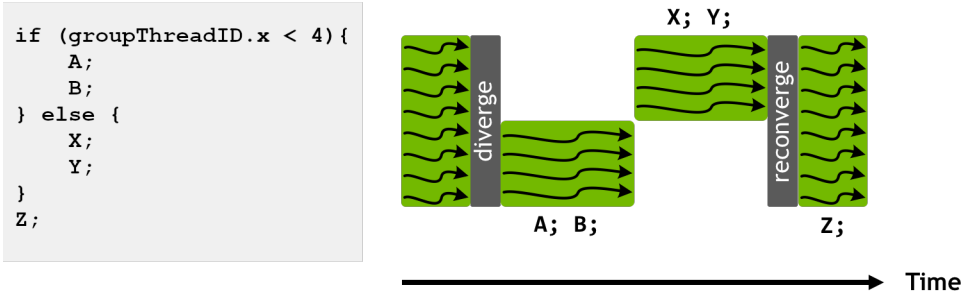


Figure 2.14: Illustration of warp execution and control-flow divergence on SIMT architectures. Divergence forces serialised execution of different paths, reducing effective throughput.

As mentioned earlier, GPUs execute threads in warps and all threads within a warp execute instructions in lockstep under the SIMT model. When all threads follow the same control flow (i.e. no branches or identical branch outcomes), the warp can execute efficiently with all lanes active.

Control-flow divergence occurs when threads within the same warp take different branches of a conditional. In such cases, the hardware must serialise execution of each unique path, masking out threads that are not on the currently executing path. Only once all divergent paths are executed do the threads reconverge. This behaviour is illustrated conceptually in Figure 2.14, where divergence forces the warp to execute multiple sequences of instructions sequentially.

The cost of divergence arises from several sources:

- **Serialised execution**: Divergent paths cannot be executed in parallel; each path is issued in turn, increasing total execution time.
- **Idle lanes**: Threads not on the active path remain idle while instructions for the other path are being issued.
- **Reconvergence overhead**: Additional logic is required to manage reconvergence points and track active thread masks.

In practice, divergent control flow has a pronounced effect when branch outcomes vary across threads within a warp. For example, in ray traversal and voxel intersection kernels, per-ray decisions (e.g. which axis to step next) can differ across neighbouring threads, leading to warp divergence. Divergence penalises effective throughput because the GPU can only execute one active control path at a time for the whole warp, reducing the number of useful operations issued per cycle.

Minimising divergence is therefore a critical optimisation on GPUs. Strategies include:

- Reformulating algorithms to eliminate or reduce conditional branches.
- Using *branchless* constructs (e.g. arithmetic masks or lookup logic) where possible.

- Ensuring that threads within a warp tend to follow the same control flow path.

These techniques improve both occupancy and utilisation of the arithmetic pipelines, leading to higher sustained performance, especially in memory-bound, control-dependent algorithms such as ray tracing and volume traversal.

2.4.4 Quantifying graphics processing unit (GPU) acceleration limits

The main strength of a GPU is in its massive parallel processing power. When a task can be executed in a parallel fashion, it conceptually makes sense that its overall performance is enhanced. However, there are limitations to what a GPU can achieve. There are two classes of constraints: fundamental limitations of the algorithm to be accelerated, and hardware limitations of the GPU.

The first limitation is described by Amdahl's law [41], which states that the overall performance improvement of accelerating an algorithm depends on how much of the algorithm can be parallelised. It states that

$$\text{Speedup} = \frac{1}{(1-P) + \frac{P}{N}} \tag{2.7}$$

where P is the part of the program that can be accelerated and N is the number of processors.

Following Amdahl's law, the most crucial limitation for GPU acceleration is that the overall acceleration potential is limited by the portion of the algorithm that can be accelerated.

While Amdahl's law dictates the theoretical limit, it is too simplistic to estimate the performance limit accurately. It is more important to consider the two main limitations of GPUs: (peak) memory bandwidth and (peak) computational performance. Memory bandwidth is measured as the number of bytes that can be transferred per second, and the computational performance is traditionally measured as the number of floating-point operations per second (FLOPS). Additionally, any kernel can be characterised by using *arithmetic intensity (AI)*, which is defined by the number of floating-point operations per memory operation. The arithmetic intensity can then be used to determine if a kernel is memory-bound or compute-bound, by comparing the compute bandwidth to the effective memory bandwidth (AI $\times$ BW).

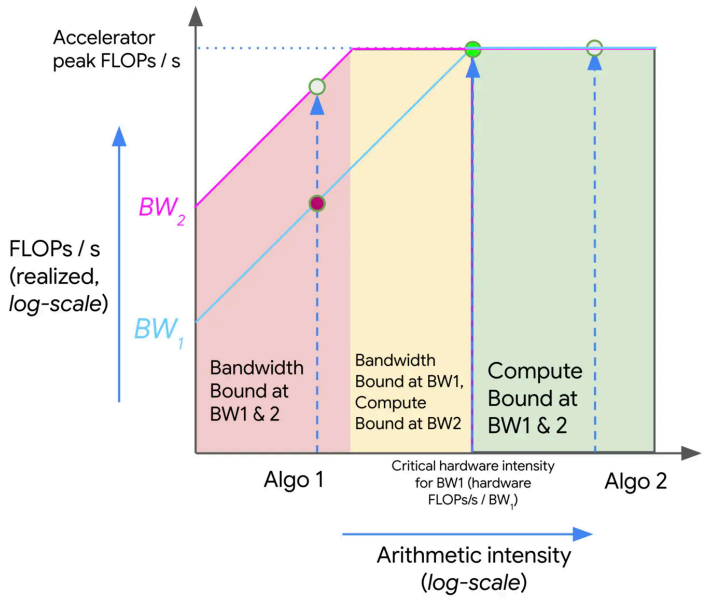


Figure 2.15: An example roofline plot showing two algorithms with different arithmetic intensities (Algo 1 and Algo 2) and their corresponding theoretical peak throughput under different bandwidths (BW1 and BW2).

The roofline model [42] can be used to compare these. Figure 2.15 illustrates two algorithm where in the red area, an algorithm is bandwidth bound at both bandwidths and is wasting some fraction of the hardware's peak FLOPS. The yellow area is bandwidth-bound only at the lower bandwidth (BW1). The green area is compute-bound at all bandwidths. Here, the algorithms use the peak FLOPS of the hardware and increasing bandwidth or improving intensity yield no benefit. Computational limits and memory limits define the two main bounds. The AI determines if the theoretical maximum performance is limited by memory bandwidth or overall compute. This model can be used to determine how to further improve a kernel within the limits of the underlying hardware, by modifying the AI.

2.5 Evaluation metrics

To assess the quality and performance of the proposed X-ray simulation methods, both *image fidelity* and *computational efficiency* are evaluated. Image fidelity is quantified using PSNR and SSIM, while performance is measured using kernel execution time on the GPU. Together, these metrics capture complementary aspects of simulation accuracy and practical usability.

2.5.1 Peak signal-to-noise ratio (PSNR)

(a) Original uncompressed image (b) Q = 90, PSNR = 45.53 dB (c) Q = 30, PSNR = 36.81 dB (d) Q=10, PSNR = 31.45 dB

Figure 2.16: Example luma PSNR values for a cjpeg compressed image at various quality level. Courtesy: Wikipedia[23].

PSNR is a widely used metric that quantifies the difference between two images on a per-pixel basis. It is derived from the mean squared error (MSE) between a reference image I_{ref} and a test image I_{test} of size $W \times H$:

$$\text{MSE} = \frac{1}{WH} \sum_{i=0}^{W-1} \sum_{j=0}^{H-1} \left(I_{\text{ref}}(i,j) - I_{\text{test}}(i,j)\right)^2 \tag{2.8}$$

PSNR is then defined as:

$$\text{PSNR} = 10 \log_{10}\left(\frac{I_{\text{max}}^2}{\text{MSE}}\right), \tag{2.9}$$

where I_{max} denotes the maximum possible pixel value (e.g. 255 for 8-bit images).

PSNR values are expressed in decibels (dB) and typically lie in the range $[0, \infty)$. In practice:

- Values above 40 dB indicate excellent agreement with almost imperceptible differences.
- Values between 30–40 dB suggest good quality with minor visible errors.
- Values below 30 dB generally correspond to noticeable image degradation.

Higher PSNR values correspond to closer agreement with the reference image. However, since PSNR treats all pixel differences equally, it does not account for perceptual or structural distortions. Small misalignments or localized artefacts can produce a disproportionately large PSNR penalty, even if the image looks acceptable visually.

[23] https://en.wikipedia.org/wiki/Peak_signal-to-noise_ratio#Quality_estimation_with_PSNR

2.5.2 Structural similarity index measure (SSIM)

SSIM was introduced to better align image quality assessment with human visual perception by explicitly modelling structural information in images [43].

SSIM compares a reference image I_{ref} and a test image I_{test} based on three components: luminance $l(x, y)$, contrast $c(x, y)$, and structure $s(x, y)$. The combined SSIM index is defined as:

$$\text{SSIM}(x, y) = l(x, y)^{\alpha} c(x, y)^{\beta} s(x, y)^{\gamma}, \tag{2.10}$$

where α, β, and γ are weighting parameters, typically set to 1.

SSIM values lie in the range $[-1, 1]$, with a value of 1 indicating perfect structural similarity. Unlike PSNR, SSIM is sensitive to local structural distortions (Figure 2.17) and is therefore better suited to assessing perceptually relevant differences in medical images.

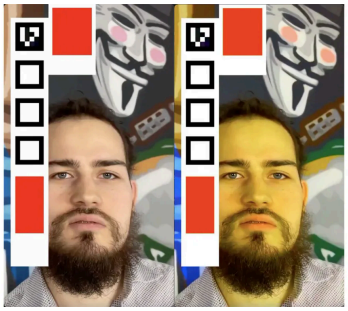


Figure 2.17: A demonstration of SSIM's structure-only sensitive nature. The colour-skewed image (right) still has an SSIM = 0.97, which indicates it is visually similar the original (left) although it is not. Courtesy: TestDevLab[24].

2.5.3 Kernel execution time

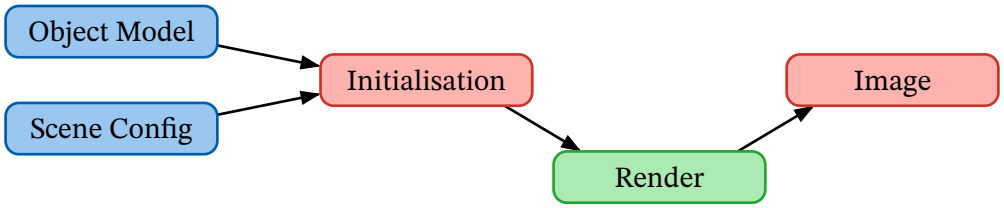


Figure 2.18: An architectural overview of the simulation pipeline consiting of data entities (blue), CPU (red) and gpu routines (green).

The computational performance of the proposed X-ray simulation methods is evaluated using the *execution time of the rendering kernel*. This kernel performs the core computation of the simulation by tracing rays through the object representation and computing the resulting pixel intensities for the entire image.

To isolate the computational efficiency of the rendering algorithm itself, kernel execution time is measured under the following assumptions:

- The object's three-dimensional representation is already resident in device memory.
- The rendered image remains in device memory after kernel execution.

Under these conditions, the measured time reflects only the cost of ray traversal, numerical integration, and associated memory accesses performed by the kernel. Host-side overheads such as memory allocation, host–device data transfer, and kernel launch latency are explicitly excluded from the measurement. In terms of simulation pipeline shown in Figure 2.18, this means measuring the runtime of the `Render` routine alone.

This choice places emphasis on *algorithmic optimisations*, including memory access coalescing, arithmetic intensity, and instruction-level parallelism, rather than on system-level integration strategies such as streaming or asynchronous execution. While both aspects are important in a

[24]https://www.testdevlab.com/blog/full-reference-quality-metrics-vmaf-psnr-and-ssim

complete GPU-accelerated application, this thesis focuses on improving the efficiency of the core rendering algorithm.

Kernel execution time therefore provides a consistent and hardware-focused basis for comparing different object representations and integration strategies under identical conditions. This metric is particularly relevant for interactive and real-time X-ray simulation, where performance is commonly limited by memory bandwidth and floating-point throughput rather than by control or I/O overhead.

2.5.4 Limitations

While PSNR, SSIM, and kernel execution time provide a quantitative basis for comparison, they do not capture all aspects of image quality or system performance. In particular, these metrics do not assess clinical relevance, diagnostic confidence, or user-perceived latency.

Nevertheless, they offer reproducible and widely accepted measures that are sufficient for evaluating the numerical accuracy and computational characteristics of the proposed methods.

3 Methodology

This chapter describes the methodology used to evaluate the proposed X-ray simulation techniques. It details the benchmarking environment in Section 3.1, the simulation setup in Section 3.2, and the benchmarking approach used to assess both *image fidelity* and *computational performance* in Section 3.3.

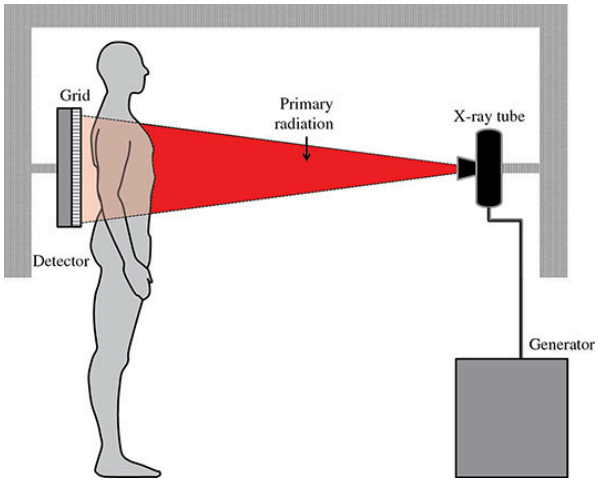
3.1 Benchmarking environment

All benchmarks were taken on a Lenovo ThinkPad P16 Gen 2 laptop with the following specifications:

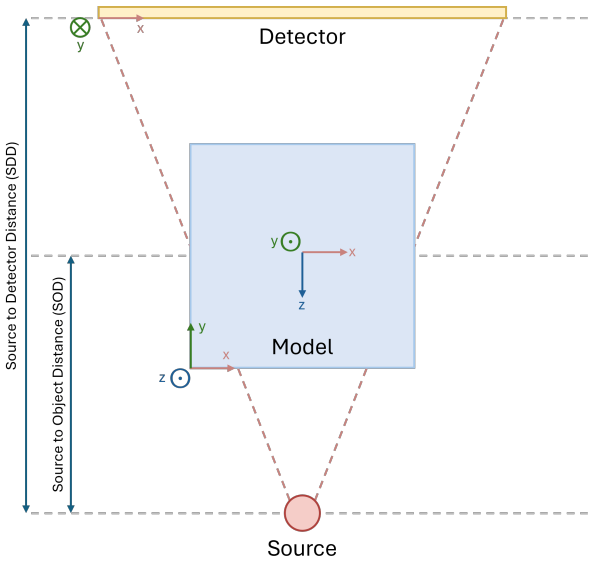
- **CPU**: Intel Core i7-13850HX
- **GPU**: NVIDIA RTX A1000 Laptop GPU (CUDA Compute 8.6), 6 GB
- **RAM**: 32 GB DDR5
- **Operating system**: Arch Linux under Windows Subsystems for Linux 2 (WSL2) on Windows 11 Pro, with CUDA Toolkit 13.1 and Python environment managed by the uv package manager[25].

[25] https://docs.astral.sh/uv/

3.2 Simulation setup



(a) A typical X-ray setup with the source, object, and the detector. Courtesy: Radiology Key[26].



(b) Schematic overview of the X-ray simulation scene The world coordinate frame and the model are centred on the isocentre.

Figure 3.1: Illustrations describing the X-ray simulation simulation scene.

For the simulation, the virtual scene is constructed as illustrated in Figure 3.1 (a). This scene models a cone-beam X-ray radiograph acquisition setup and is parameterised to closely resemble the Philips Azurion 7 configuration. These parameters have been defined as:

- Source-to-object distance (SOD): 80 cm
- Source-to-detector distance (SDD): 120.5 cm
- Detector size: 40 cm × 40 cm
- Detector resolution: 1024 × 1024 pixels

The rotation of the C-arm during image acquisition is modelled implicitly by rotating the object volume while keeping the source and detector fixed in space. This approach is mathematically equivalent to rotating the source–detector pair around a stationary object, but avoids repeated reconfiguration of the imaging geometry.

In the simulator, the object is rotated about a single principal axis corresponding to the C-arm sweep angle θ. The rotation is applied by updating the object-to-world transformation matrix used in the camera model described in Section 2.2. This formulation naturally supports the generation of multiple projection views by varying θ while preserving a consistent imaging geometry.

Optional out-of-plane tilts can be incorporated by extending the transformation with additional rotations; however, unless stated otherwise, all experiments in this work assume a single-axis rotational trajectory.

Image formation in the simulator follows the forward-projection process described in Section 2.1. For each detector pixel, a ray is cast from the source through the scene, intersecting the object volume. The total attenuation along each ray path is accumulated according to the material properties, yielding the detected intensity at the corresponding pixel.

[26]https://radiologykey.com/projection-x-ray-imaging/

3.3 Profiling and benchmarking

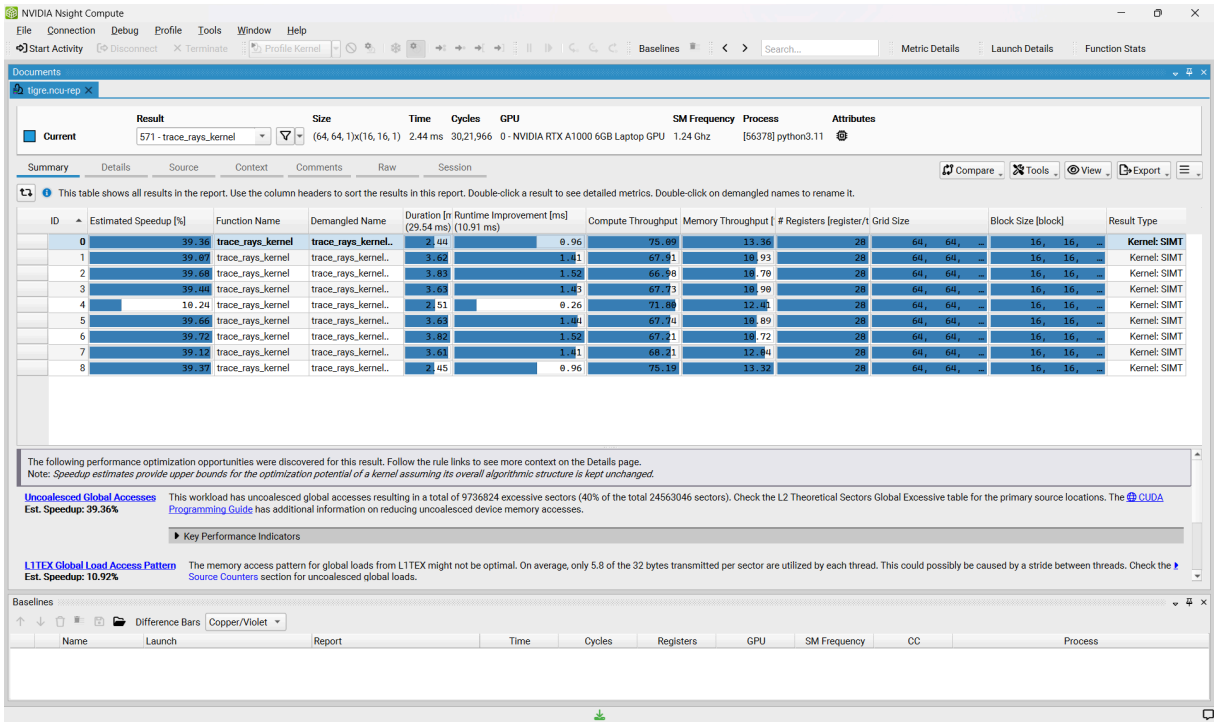


Figure 3.2: Screenshot of the NVIDIA Nsight Compute interface. It provides easy access to performance metrics such as execution time, floating-point utilisation, and memory access behaviour.

To accurately assess the computational performance and resource utilisation of the proposed algorithms, NVIDIA Nsight Compute[27] was used as the primary profiling tool. Nsight Compute provides fine-grained, kernel-level insight into GPU execution, exposing metrics such as kernel runtime, instruction mix, memory access patterns, cache hit and miss rates across the memory hierarchy, and overall hardware utilisation.

As the focus of this thesis is on accelerating the X-ray forward-projection computation itself, performance measurements were restricted to the execution time of the core GPU kernels. Data transfers between host (CPU) and device (GPU) memory were explicitly excluded from the measurements. This approach isolates the computational cost of the projection algorithms and allows for a fair comparison between different implementations.

To obtain stable and representative timing results, each kernel was executed multiple times under identical conditions, and the average runtime per frame was recorded. The effective frame rate (FPS) was then computed as the reciprocal of the average kernel execution time. For interactive applications such as virtual system testing and physician training, real-time performance is considered achievable at a target rate of 60 FPS.

In addition to performance, image fidelity was evaluated using established quantitative metrics, as described in Section 2.5:

- **PSNR**: It measures the pixel-wise difference between simulated and reference images in decibels. Higher PSNR values indicate closer numerical agreement.
- **SSIM**: It assesses perceptual similarity by comparing luminance, contrast, and structural information. SSIM values range from 0 to 1, with higher values indicating greater similarity.

[27] https://developer.nvidia.com/nsight-compute

3.4 Benchmarking procedure

To benchmark the proposed algorithms, a dedicated C++ driver program was written. This program performs the following steps:

1. Load the object representation into memory.
2. Initialise the X-ray source, detector, and scene parameters.
3. Launch the forward-projection GPU kernel for all rays corresponding to the detector pixels.
4. Transfer the rendered image from device to host memory and save it to disk.
5. Repeat steps 2-4 for multiple projection angles.

Kernel-level performance metrics were collected by executing this benchmarking program under the NVIDIA Nsight Compute command-line interface (`ncu`). The profiler was configured to capture a comprehensive set of metrics for the forward-projection kernel:

```
$ ncu --kernel-name=<name of render kernel> --set full \
      --export <name of report> ./bench
```

The resulting profiling reports were analysed using the Nsight Compute graphical interface to extract execution time, instruction mix, memory behaviour, and hardware utilisation metrics.

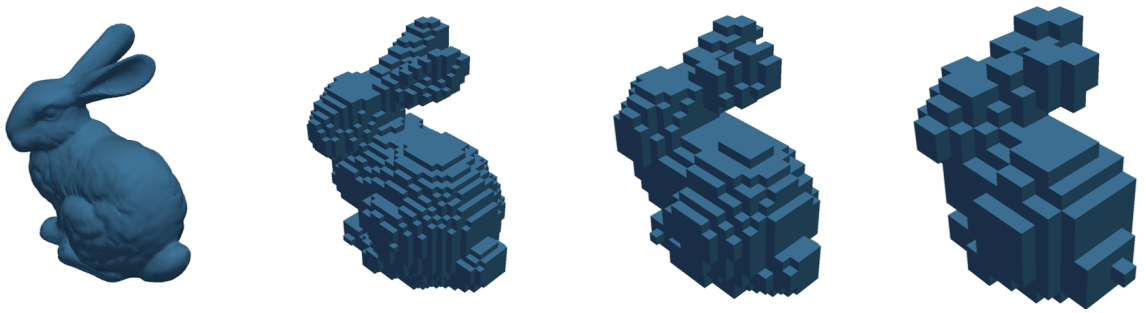
Image quality metrics, including PSNR and SSIM, were computed separately using a Python-based evaluation script. The rendered images produced by the benchmarking program were compared against reference projections to quantify numerical and perceptual fidelity.

For the sake of reproducibility and transparency, all benchmarking scripts, profiling reports, and evaluation outputs are provided in the code repository[28] .

[28]https://github.com/UtkarshVerma/gpu-accel-xray-sim

4

Simulation: Voxelised models



(a) Surface mesh. (b) Voxel size = 0.03. (c) Voxel size = 0.05. (d) Voxel size = 0.08.

Figure 4.1: An illustration of voxelising a rabbit mesh (Figure 4.2) with increasing voxel sizes from left to right.

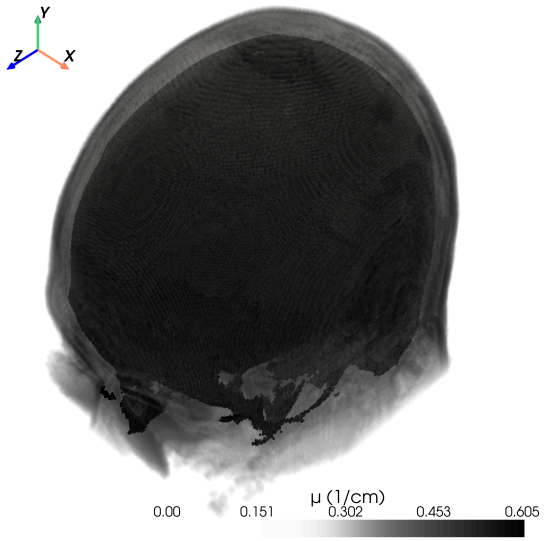
Voxelised models represent three-dimensional anatomy as a discrete volumetric field defined over a regular Cartesian grid. In practice, this field is stored as a dense 3D array of values, where each voxel corresponds to a sample of a spatially varying physical quantity at fixed spatial intervals. Instead of explicitly modelling object boundaries, the interior of the object is described through these regularly spaced samples. In the context of X-ray simulation, the stored quantity corresponds to either the mass attenuation density (μ/ρ) or the linear attenuation coeffecient (μ) distribution within the object.

As illustrated in Figure 4.1, voxelisation converts a continuous surface representation into a piecewise-constant volumetric approximation. Increasing the voxel size reduces geometric fidelity, particularly near object boundaries, while decreasing the voxel size improves spatial accuracy at the cost of increased memory usage and computational workload.

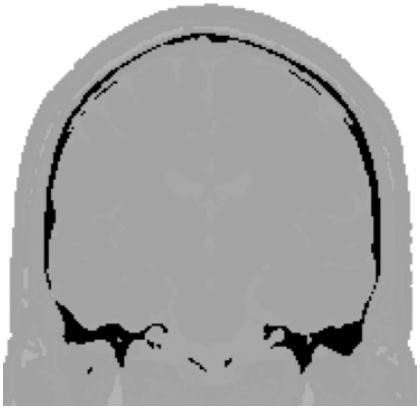
Since voxelised models directly encode volumetric attenuation properties, the overall simulation pipeline remains unchanged from the formulation described in Section 2.2 and Section 3.2. Rays are traced from the X-ray source to the detector, and attenuation is accumulated along each ray path. The distinction lies solely in how the volumetric properties are queried and integrated during ray traversal. To make this distinction precise, the data encoding of voxel models is described next.

4.1 Data encoding

(a) YZ slice at x = 90 mm.



(b) 3D volume.



(c) XY slice at z = 108 mm.

(d) XZ slice at y = 89 mm.

Figure 4.2: Planar slices (Figure 4.3, Figure 4.3, and Figure 4.3) and 3D representation (Figure 4.3) of the human skull model. Linear attenuation coffecient (μ) values have been color mapped such that dark regions correspond to dense materials and vice versa.

As previously stated, a voxelised model is stored as a dense three-dimensional array of cubic cells on a regular grid. Each voxel stores the linear attenuation coefficient (μ), assumed constant within its finite spatial extent.

Voxelisation can be interpreted as a discretisation of a continuous scalar field defined over three-dimensional space. Let $\mu(x, y, z)$ denote the continuous linear attenuation coefficient distribution of an object. As is the case with most techniques, there exist multiple ways to voxelise a continuous scalar field [44]. One of these approaches is to sample the field at discrete spatial locations determined by the voxel size Δ_v. Hence, a voxel grid can be expressed as:

$$\mu_{\{i,j,k\}} = \mu(x_i, y_j, z_k), \tag{4.1}$$

where

$$x_i = x_0 + i\Delta_v, \quad y_j = y_0 + j\Delta_v, \quad z_k = z_0 + k\Delta_v, \tag{4.2}$$

and (x_0, y_0, z_0) denotes the origin of the voxel grid in world coordinates.

Because each voxel represents a finite volume rather than an infinitesimal point, the stored value $\mu_{\{i,j,k\}}$ approximates the average μ over that voxel. This discretisation introduces an approximation error that depends on both the voxel size and the local geometric complexity of the object, as seen in Figure 4.1. For example, the skull voxel model used throughout this thesis uses a voxel size (Δ_v) of 1 mm.

During simulation, rays traverse the voxel grid and attenuation is accumulated by integrating these piecewise-constant voxel values along the ray path. As a result, voxel resolution directly influences both numerical accuracy and computational cost, making voxel size a central parameter in the evaluation of voxelised models.

4.2 Computing the path attenuation

The physics of X-ray attenuation along a ray path is governed by Eq. (2.6):

$$-\ln \frac{I}{I_0} = \int \mu(x)\,\mathrm{d}x \tag{4.3}$$

where $\mu(x)$ denotes the spatially varying linear attenuation coefficient.

When the object is represented as a voxelised volume, the continuous integral can be approximated by a discrete summation over the voxels intersected by the ray:

$$-\ln \frac{I}{I_0} = \sum_{i=0}^{N} l_i \mu_i \tag{4.4}$$

Here, l_i denotes the length of the ray segment inside the i^{th} voxel, μ_i is the linear attenuation coefficient associated with that voxel, and N is the total number of voxels intersected by the ray.

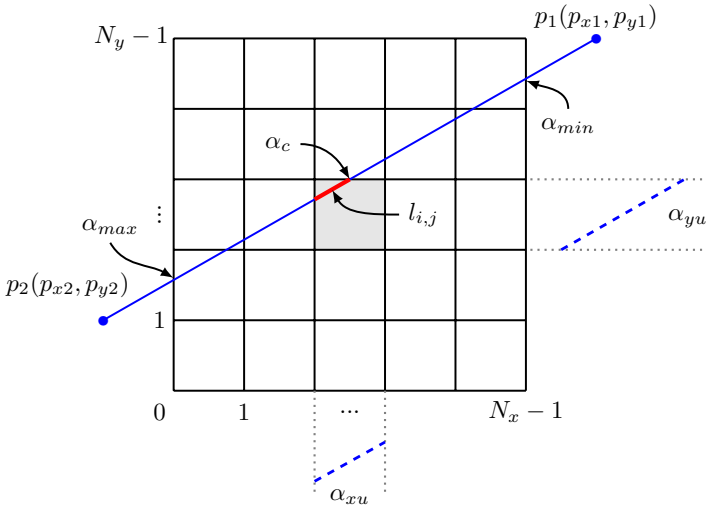


Figure 4.3: Illustration of voxel traversal along a ray in 2D where $l_{i,j}$ is the length intersected with (i, j)$^{\text{th}}$ voxel and α_{max} and α_{min} denote the line fraction for the entry and exit points respectively [7].

The quantities $\{l_i\}$ and $\{\mu_i\}$ are obtained by incrementally traversing the voxel grid along the ray direction. This traversal is commonly performed using digital differential analyser (DDA)–based algorithms in three dimensions or closely related variants, such as the fast voxel traversal algorithm by Amanatides and Woo [45], Siddon's algorithm [46], Jacob's algorithm [47], as well as later

refinements [48,49]. The principle of voxel traversal is illustrated in Figure 4.3 which uses Jacob's algorithm.

4.2.1 The algorithm

Expressing Eq. (4.4) using the notation of Figure 4.3 yields

$$d_{\mathrm{uv}} = \sum_{i<N_x, j<N_y} l_{\mathrm{uv},\{i,j\}} \mu_{\{i,j\}}, \tag{4.5}$$

where d_{uv} denotes the accumulated attenuation of ray uv, $l_{\mathrm{uv},\{i,j\}}$ is the intersection length inside voxel $\{i, j\}$, and N_x, N_y are the number of voxels along each axis. For clarity the derivation is presented in 2D; the 3D extension is direct.

Let the ray start at $\vec{p}_1 = \begin{pmatrix} p_{\mathrm{x1}} \\ p_{\mathrm{y1}} \end{pmatrix}$ and end at $\vec{p}_2 = \begin{pmatrix} p_{\mathrm{x2}} \\ p_{\mathrm{y2}} \end{pmatrix}$. Its parametric representation is

$$\vec{p}(\alpha) \coloneqq \vec{p}_1 + \alpha(\vec{p}_2 - \vec{p}_1), \quad \alpha \in [0,1]. \tag{4.6}$$

4.2.1.1 Entry and exit parameters

The parameter values at which the ray intersects vertical and horizontal voxel planes are

$$\alpha_x(i) \coloneqq \frac{i\Delta_v - p_{\mathrm{x1}}}{p_{\mathrm{x2}} - p_{\mathrm{x1}}}, \tag{4.7}$$

$$\alpha_y(j) \coloneqq \frac{j\Delta_v - p_{\mathrm{y1}}}{p_{\mathrm{y2}} - p_{\mathrm{y1}}}, \tag{4.8}$$

for $i \in [0, N_x]$ and $j \in [0, N_y]$ and where Δ_v is the voxel size.

The entry and exit parameters are obtained by restricting the ray to the intersection of the x- and y-slabs:

$$\alpha_{\mathrm{entry}} \coloneqq \max\big(\min(\alpha_x(0), \alpha_x(N_x)), \min\big(\alpha_y(0), \alpha_y(N_y)\big)\big), \tag{4.9}$$

$$\alpha_{\mathrm{exit}} \coloneqq \min\big(\max(\alpha_x(0), \alpha_x(N_x)), \max\big(\alpha_y(0), \alpha_y(N_y)\big)\big). \tag{4.10}$$

If $\alpha_{\mathrm{exit}} \leq \alpha_{\mathrm{entry}}$, the ray does not intersect the grid.

4.2.1.2 Initialisation

The total ray length $l_{\mathrm{tot}} \coloneqq \left\| \vec{p}_2 - \vec{p}_1 \right\|_2$, voxel step direction $\vec{\Delta} \coloneqq \mathrm{sign}(\vec{p}_2 - \vec{p}_1)$, and parameter increments per voxel crossing are

$$\alpha_{\mathrm{xu}} \coloneqq \frac{1}{|p_{\mathrm{x2}} - p_{\mathrm{x1}}|}, \quad \alpha_{\mathrm{yu}} \coloneqq \frac{1}{|p_{\mathrm{y2}} - p_{\mathrm{y1}}|}. \tag{4.11}$$

Initial voxel index at the entry point:

$$i \coloneqq \lfloor p_x(\alpha_{\mathrm{entry}}) \rfloor, \quad j \coloneqq \lfloor p_y(\alpha_{\mathrm{entry}}) \rfloor \tag{4.12}$$

Initialize the traversal state:

$$\alpha_c \coloneqq \alpha_{\mathrm{entry}}, \quad d_{\mathrm{ray}} \coloneqq 0 \tag{4.13}$$

The next intersection parameters α_x and α_y are initialized to the first grid crossings beyond α_{entry}.

4.2.1.3 Iterative traversal

At each step, the next crossed plane is determined by comparing α_x and α_y.

If $\alpha_x < \alpha_y$:

$$d_{\text{ray}} \mathrel{+}= (\alpha_x - \alpha_c) l_{\text{tot}} \mu_{\{i,j\}} \tag{4.14}$$

$$i \mathrel{+}= \Delta_x \tag{4.15}$$

$$\alpha_c = \alpha_x \tag{4.16}$$

$$\alpha_x \mathrel{+}= \alpha_{\text{xu}} \tag{4.17}$$

Otherwise:

$$d_{\text{ray}} \mathrel{+}= (\alpha_y - \alpha_c) l_{\text{tot}} I(i,j) \tag{4.18}$$

$$j \mathrel{+}= j\Delta_y \tag{4.19}$$

$$\alpha_c = \alpha_y \tag{4.20}$$

$$\alpha_y \mathrel{+}= \alpha_{\text{yu}} \tag{4.21}$$

The procedure continues until $\alpha_c \geq \alpha_{\text{exit}}$. Simultaneous crossings naturally produce zero-length updates and require no special handling.

Extension to 3D follows directly by introducing the z-direction and selecting the minimum among α_x, α_y, and α_z at each iteration.

4.3 GPU implementation

The renderer was developed over a span of several iterations through profile guided optimisation principles using Nsight Compute. These iterations are detailed below.

4.3.1 Baseline

The baseline rendering pipeline proceeds as follows:

1. Allocate a *tile buffer* of fixed size (16×16 pixels) on the GPU.
2. Project the model's bounding box onto the image plane to identify the tiles that intersect the volume, marking these tiles as *active*.
3. Process each active tile sequentially[29]:
   1. Dispatch the tile buffer, along with the relevant volume and geometry data, to the GPU kernel.
   2. Compute the contributions for each pixel in the tile on the GPU.
   3. Copy the completed tile back to the CPU, insert it into the corresponding location in the final image, and launch the kernel for the next tile.

Attenuation along each ray is computed using the algorithm described in Section 4.2.

4.3.1.1 Metrics

Table 1: Frame render times (ms) for the baseline implementation[30].

Projection	0°	30°	45°	60°	90°	120°	135°	150°	180°	Average
Time	931.88	1132.57	1148.59	1102.2	796.03	1098.41	1147.19	1112.56	777.27	1027.41 ± 151.33

Running the baseline implementation yields the execution times listed in Table 4.1. The measured performance is only marginally better than the single-core CPU implementation, which renders a frame in approximately 1.3 ms. This indicates that the GPU is not being effectively utilised.

[29] In Philips' codebase, tiling was handled on the CPU to reduce memory footprint and enable multithreading. Early design decisions made it difficult to implement a GPU pipeline without tiling, so this approach was used even if it was less efficient for GPU execution.

[30] These times are not measured using Nsight Compute since the baseline implementation spawns too many kernels.

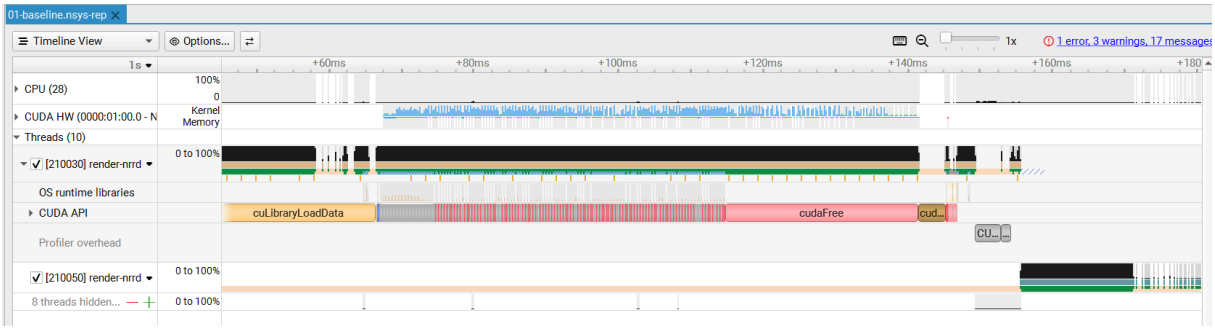


Figure 4.4: Timeline view from NVIDIA Nsight Systems for the baseline implementation. Frequent `cudaMemcpy()` calls (thin red strips) dominate execution time.

Time	Total Time	Num Calls	Avg	Med	Min	Max	StdDev	Name
43.1%	245.338 ms	1	245.338 ms	245.338 ms	245.338 ms	245.338 ms	0 ns	cuLibraryLoadData
42.3%	240.792 ms	3	80.264 ms	398.671 µs	12.842 µs	240.380 ms	138.665 ms	cudaMalloc
8.7%	49.757 ms	3138	15.856 µs	4.742 µs	2.880 µs	3.632 ms	81.607 µs	cudaMemcpy
4.9%	27.981 ms	3	9.327 ms	624.125 µs	506.358 µs	26.850 ms	15.176 ms	cudaFree
0.7%	3.806 ms	1	3.806 ms	3.806 ms	3.806 ms	3.806 ms	0 ns	cudaDeviceSynchronize
0.3%	1.611 ms	196	8.217 µs	4.814 µs	2.478 µs	520.248 µs	37.004 µs	cudaLaunchKernel
0.1%	435.964 µs	1	435.964 µs	435.964 µs	435.964 µs	435.964 µs	0 ns	cudaMemset
0.0%	20.410 µs	196	104 ns	74 ns	62 ns	540 ns	71 ns	cuKernelGetName
0.0%	9.036 µs	1	9.036 µs	9.036 µs	9.036 µs	9.036 µs	0 ns	cuModuleGetLoadingMode
0.0%	764 ns	1	764 ns	764 ns	764 ns	764 ns	0 ns	cuLibraryGetKernel

Figure 4.5: CUDA API summary from Nsight Systems indicating that ≈ 50 ms is spent on host-device memory transfers.

Profiling with NVIDIA Nsight Systems reveals that the majority of execution time is spent in repeated `cudaMemcpy()` calls, as shown in Figure 4.4 and summarised in Figure 4.5. Approximately 50 ms of runtime is attributed to host–device memory transfers, dwarfing the actual kernel execution time.

This behaviour is expected: in the baseline design, tiling and orchestration logic remain on the CPU. As a result, intermediate data must be repeatedly transferred between host and device memory, incurring significant PCIe latency and bandwidth overhead. Because these transfers are synchronous and frequent, they effectively serialise execution and prevent the GPU from operating at high throughput.

The primary bottleneck is therefore not arithmetic performance, but data movement across the CPU–GPU boundary. The most direct optimisation is to eliminate unnecessary memory copies by migrating tiling and control logic to the GPU. By keeping intermediate data resident in device memory and allowing CUDA kernels to handle tile scheduling directly, host–device transfers can be minimised, enabling substantially higher effective performance.

4.3.2 Move tiling logic to CUDA

To eliminate slow CPU ↔ GPU transfers over the PCIe bus, the library was restructured to support algorithms without tiling. In the CUDA implementation, the following changes were applied:

- Tiling was removed entirely. A single kernel now renders the entire image.
- Projection logic was removed to simplify the initial implementation; it can be reintroduced later if needed.

CPU↔GPU traffic is reduced to just two transfers: the volume data (CPU → GPU) and the final image (GPU → CPU), significantly improving throughput.

4.3.2.1 Metrics

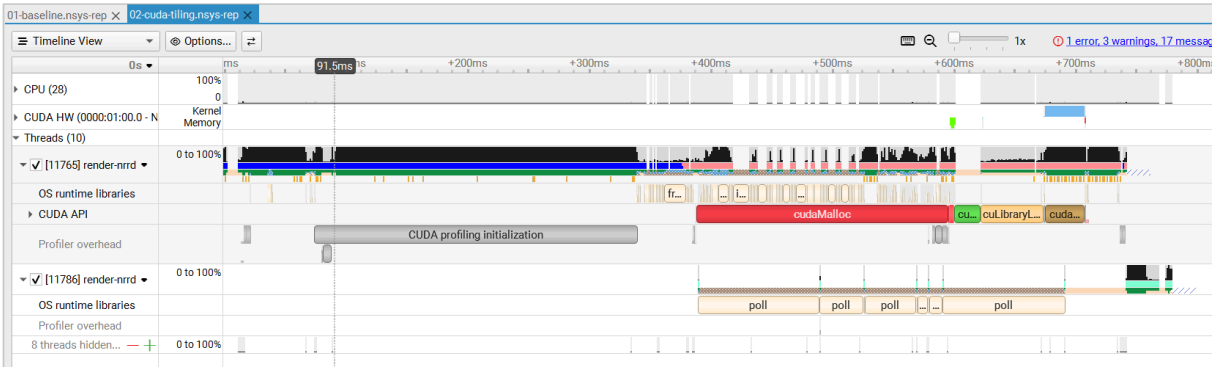


Figure 4.6: Timeline view from NVIDIA Nsight Systems for the CUDA tiling implementation. Excess `cudaMemcpy()` calls have been eliminated.

Time	Total Time	Num Calls	Avg	Med	Min	Max	StdDev	Name
64.5%	207.068 ms	2	103.534 ms	103.534 ms	338.933 µs	206.729 ms	145.940 ms	cudaMalloc
15.8%	50.776 ms	1	50.776 ms	50.776 ms	50.776 ms	50.776 ms	0 ns	cuLibraryLoadData
10.2%	32.633 ms	1	32.633 ms	32.633 ms	32.633 ms	32.633 ms	0 ns	cudaDeviceSynchronize
6.9%	22.145 ms	1	22.145 ms	22.145 ms	22.145 ms	22.145 ms	0 ns	cudaMemset
1.8%	5.745 ms	2	2.872 ms	2.872 ms	1.065 ms	4.679 ms	2.556 ms	cudaMemcpy
0.4%	1.415 ms	2	707.547 µs	707.547 µs	482.887 µs	932.207 µs	317.717 µs	cudaFree
0.3%	1.017 ms	1	1.017 ms	1.017 ms	1.017 ms	1.017 ms	0 ns	cudaLaunchKernel
0.0%	3.479 µs	1	3.479 µs	3.479 µs	3.479 µs	3.479 µs	0 ns	cuModuleGetLoadingMode
0.0%	647 ns	1	647 ns	647 ns	647 ns	647 ns	0 ns	cuLibraryGetKernel
0.0%	422 ns	1	422 ns	422 ns	422 ns	422 ns	0 ns	cuKernelGetName

Figure 4.7: CUDA API summary for Nsight Systems confirming that only two `cudaMemcpy()` calls happen now.

Running the profiler (see Figure 4.6 and Figure 4.7) revealts that only two `cudaMemcpy()` transfers take place now instead of 3138 calls from before. This is well reflected in the kernel runtime resulting in a 16.74x speedup.

Table 2: Kernel runtime (ms) before and after moving tiling logic to CUDA.

Projection	0°	30°	45°	60°	90°	120°	135°	150°	180°	Average
Before	931.88	1132.57	1148.59	1102.2	796.03	1098.41	1147.19	1112.56	777.27	1027.41 ± 151.33
After	46.66	68.99	71.79	68	46.52	67.99	71.79	68.98	46.66	61.93 ± 11.57
Speedup	19.97	16.42	16	16.21	17.11	16.15	15.98	16.13	16.66	16.74 ± 1.27

4.3.3 Use single-precision floating point (FP32) numbers

On the NVIDIA RTX A1000 GPU (Ampere architecture), the peak throughput for single-precision floating point (FP32) arithmetic is 64 times higher than that of double-precision floating point (FP64) arithmetic [50]. This disparity reflects a broader architectural trend across NVIDIA GPUs, where significantly more execution resources are allocated to single-precision arithmetic than to double-precision arithmetic. Given this, it was natural to evaluate how rendering performance changes when switching from FP64 to FP32.

To perform this optimisation, the kernel was modified as follows:

- All `double` variables were replaced with `float`.
- Floating-point literals such as `1.0` were replaced with `1.0f`, since the C and C++ standards treat unsuffixed floating-point constants as `double` by default [51].

4.3.3.1 Metrics

Table 3: kernel runtime (ms) before and after switching to FP32 arithmetic.

Projection	0°	30°	45°	60°	90°	120°	135°	150°	180°	Average
Before	46.66	68.99	71.79	68	46.52	67.99	71.79	68.98	46.66	61.93 ± 11.57
After	4.21	5.65	5.85	5.59	4.16	5.58	5.85	5.64	4.21	5.19 ± 0.76
Speedup	11.08	12.21	12.26	12.18	11.19	12.18	12.27	12.23	11.08	11.85 ± 0.55

Switching to FP32 resulted in a substantial reduction in kernel runtime. Averaged across all projection angles, the kernel achieved an approximate speedup of ≈ 12x, as shown in Table 4.3.

Several factors explain this performance improvement:

- The number of registers per thread (`launch__registers_per_thread`) decreased from 66 to 48, significantly reducing register pressure.
- Reduced register usage enabled higher occupancy, with achieved occupancy (`sm__warps_active.avg.pct_of_peak_sustained_active`) increasing from 47.82% to 81.22%.
- The elimination of FP64 arithmetic, confirmed by the pipeline utilisation data in Figure 4.8, shifted execution toward FP32 units. On Turing GPUs, each SM provides 64 FP32 units but only 2 FP64 units, leading to significantly higher instruction throughput for single-precision arithmetic [52].

Table 4: SSIM and PSNR metrics comparing FP32 renders against the baseline.

Angle (°)	SSIM	PSNR (dB)
0	1.0000	87.84
30	1.0000	75.16
45	1.0000	73.77
60	1.0000	73.87
90	1.0000	83.99
120	1.0000	74.68
135	1.0000	74.68
150	1.0000	74.02
180	1.0000	87.84

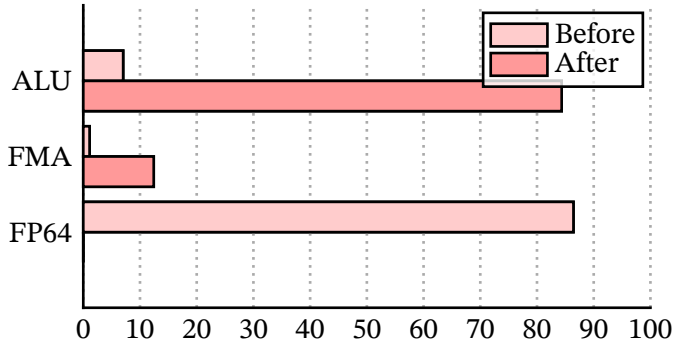


Figure 4.8: Pipeline utilisation (% of elapsed cycles) before and after switching to FP32 for the 0° projection. The total elapsed cycles decreased from 57746707.5 to 5209277, corresponding to a reduction of −90.98%.

Finally, image quality metrics indicate that the use of FP32 precision does not introduce any measurable degradation in visual fidelity. As shown in Table 4.4, the PSNR exceeds 70 dB for all projection angles, which is substantially higher than the 40 dB threshold for excellent image quality defined in Subsection 2.5.1. This margin suggests that numerical differences between precisions are negligible in practical terms.

Similarly, the SSIM values remain consistently close to 1, indicating near-perfect structural similarity between the reference and reduced-precision projections. Together, these results confirm that single-precision computation provides sufficient numerical accuracy for forward projection in this setting, while offering the associated performance benefits.

4.3.4 Reduce branching

As discussed in Subsection 2.4.3, GPU performance strongly depends on uniform and predictable control flow. On NVIDIA GPUs, threads within a warp execute in lockstep, and any control-flow divergence forces the warp to serialise execution paths. This directly reduces effective throughput.

This effect is visible in the warp execution efficiency metrics, that is, `smsp__thread_inst_executed_per_inst_executed` ≈ 24.9 instead of the ideal value of 32. Such values indicate frequent divergence within warps, primarily caused by axis-dependent branching in the ray traversal and intersection logic.

4.3.4.1 Initial implementation

```cpp
// Get the index of the minimum element in a 3D vector.
template <typename T, glm::qualifier Q>
__host__ __device__ uint8_t arg_min(const glm::vec<3, T, Q> v) {
    const uint8_t xy = v.y < v.x;
    const uint8_t xz = v.z < v.x;
    const uint8_t yz = v.z < v.y;

    return (xy & ~yz) + ((xz & yz) << 1);
}

// Get the index of the maximum element in a 3D vector.
template <typename T, glm::qualifier Q>
__host__ __device__ uint8_t arg_max(const glm::vec<3, T, Q> v) {
    const uint8_t xy = v.y > v.x;
    const uint8_t xz = v.z > v.x;
    const uint8_t yz = v.z > v.y;

    return (xy & ~yz) + ((xz & yz) << 1);
}
```

Listing 1: Branchless helper functions `arg_min()` and `arg_max()` that return the axis containing the minimum or maximum component of a 3D vector.

```cpp
// Before
================================
if (a.x >= a.y && a.x >= a.z) {
  // X-axis handling
} else if (a.y >= a.z) {
  // Y-axis handling
} else {
  // Z-axis handling
}


// After
================================
const uint8_t hit_axis = arg_max(a);
do_something_for_axis(hit_axis);
```

Listing 2: Replacing axis-dependent branching with branchless axis selection.

```cpp
// Before
================================
if (a.x > 0.0f) {
  b.x = -c.x / a.x;
} else if (a.x < 0.0f) {
  b.x =  c.x / a.x;
} else {
  b.x = FLT_MAX;
}
// Repeated for other axes

// After
================================
b = -glm::sign(a) * c / a;
```

Listing 3: Eliminating conditional assignments using arithmetic identities.

To mitigate this, the kernel was refactored to eliminate most conditional branches. Since the kernel is not compute-bound, additional arithmetic operations were preferred over divergent control flow. The refactoring applied the following transformations:

- Axis selection logic was rewritten using branchless `arg_min()` and `arg_max()` functions (see Listing 4.1).
- All axis-dependent operations were rewritten to use indexed vector access, e.g. `next_beam_fractions[hit_axis]`, instead of explicit `if-else` chains (see Listing 4.2).
- Conditional assignments were made branchless using arithmetic identities (see Listing 4.3).

4.3.4.1.1 Metrics

Table 5: Kernel runtime (ms) before and after reducing branching (v1).

Projection	0°	30°	45°	60°	90°	120°	135°	150°	180°	Average
Before	4.21	5.65	5.85	5.59	4.16	5.58	5.85	5.64	4.21	5.19 ± 0.76
After	7.45	10.35	10.61	9.99	7.43	9.97	10.57	10.32	7.44	9.35 ± 1.45
Speedup	0.57	0.55	0.55	0.56	0.56	0.56	0.55	0.55	0.57	0.56 ± 0.01

Examining the runtimes in Table 4.5 reveals an unexpected result: the branch-reduced kernel actually shows a slowdown of approximately 1.8x.

Profiling reports clarify the cause. On NVIDIA GPUs, registers are extremely fast but *not dynamically addressable*[53]. When a variable must be indexed dynamically, the compiler cannot place it in registers and instead allocates it in *thread-private local memory*. Despite its name, local memory is *backed by global memory*, making it roughly 500x slower than registers [54].

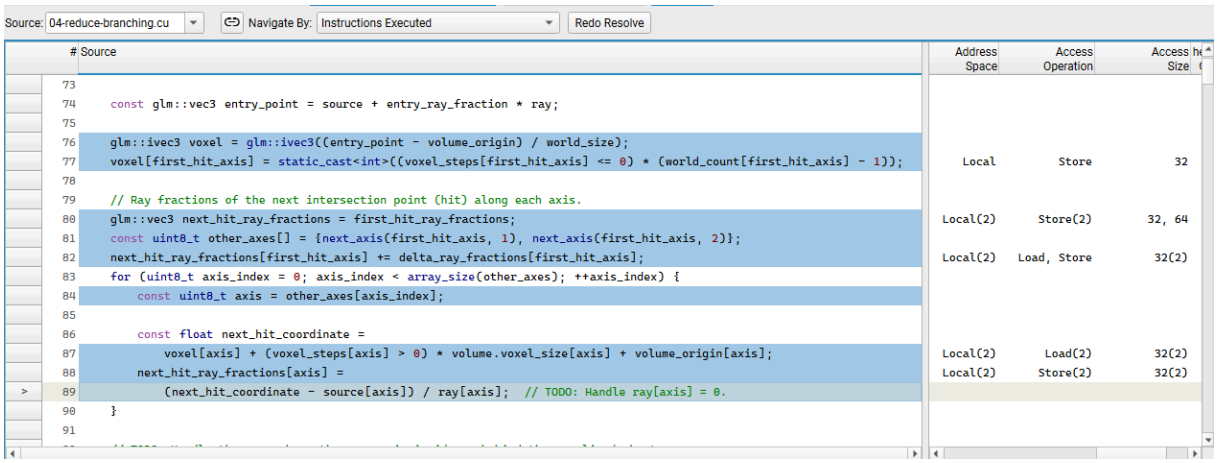


Figure 4.9: Nsight Compute source inspector where lines containing local-memory backed variables are selected.

This behaviour is also reflected in the metrics. Local memory loads (`l1tex__t_requests_pipe_lsu_mem_local_op_ld.sum`) increased from 0 to 30657588 following the branchless transformation. These additional local memory accesses explain why the kernel ran slower despite reducing branching.

4.3.4.2 Eliminating the local memory accesses

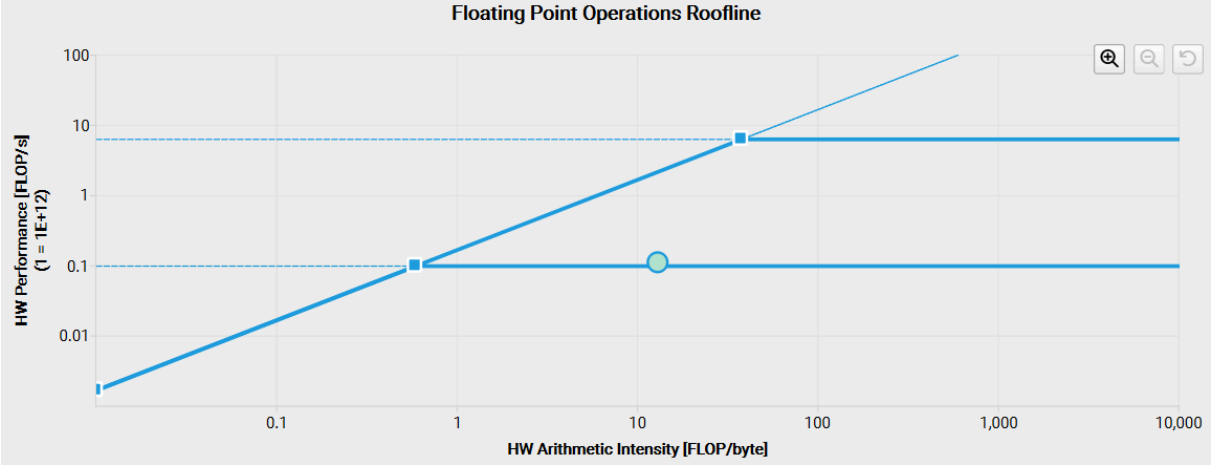


Figure 4.10: Roofline model for the first attempt at branch reduction. The lower roofline is for FP64 and must be ignored.

Knowing the cause of the slowdown also points to the remedy, that is, avoid dynamically indexing variables that must reside in registers.

In the original kernel, dynamic indexing was used to reduce the number of computations along certain axes. However, since the kernel is not compute-bound (see Figure 4.10), it is preferable to trade a few extra arithmetic operations again for register-friendly code. This will avoid slow local memory accesses.

```
// Before ================================
const uint8_t axis = arg_min(a);
a[axis] = b[axis];


// After =================================
const uint8_t axis = arg_min(a);
const auto axis_mask = glm::bvec3(
                         axis == 0,
                         axis == 1,
                         axis == 2);
a = axis_mask * b + (1 - axis_mask) * a;
```

```
// Before ================================
bool axis_in_bounds =
  voxel[axis] >= 0 &&
    voxel[axis] < bounds[axis]


// After =================================
const auto axis_mask = glm::bvec3(
                         axis == 0,
                         axis == 1,
                         axis == 2);
const glm::bvec3 in_bounds =
  glm::greaterThanEqual(voxel,
    glm::ivec3(0)) *
      glm::lessThan(voxel,
        glm::ivec3(bounds));
bool axis_in_bounds =
  (in_bounds.x * axis_mask.x +
    in_bounds.y * axis_mask.y +
    in_bounds.z * axis_mask.z);
```

Listing 4: Arithmetic transforms using vector interpolation and masking to avoid dynamic indexing of variables.

To implement this, assignments and bounds checks were rewritten using vector interpolation and masking, as shown in Listing 4.4. These transformations preserve the original logic while eliminating dynamic indexing, allowing the compiler to keep variables in registers and thereby improve throughput.

4.3.4.2.1 Metrics

Table 6: Kernel runtime (ms) before (using FP32) and after reducing branching (v2).

Projection	0°	30°	45°	60°	90°	120°	135°	150°	180°	Average
Before	4.21	5.65	5.85	5.59	4.16	5.58	5.85	5.64	4.21	5.19 ± 0.76
After	2.66	3.31	3.9	3.27	2.64	3.27	3.4	3.31	4.19	3.33 ± 0.5
Speedup	1.58	1.71	1.5	1.71	1.57	1.71	1.72	1.7	1.01	1.58 ± 0.23

As shown in Table 4.6, the results are now much more promising. The kernel achieves a speedup of 1.58x compared to the baseline using FP32.

Furthermore, local memory accesses have returned to 0, confirming that the branch reduction and vector masking optimisations successfully eliminated the local memory accesses.

4.3.5 Using look-up tables (LUTs)

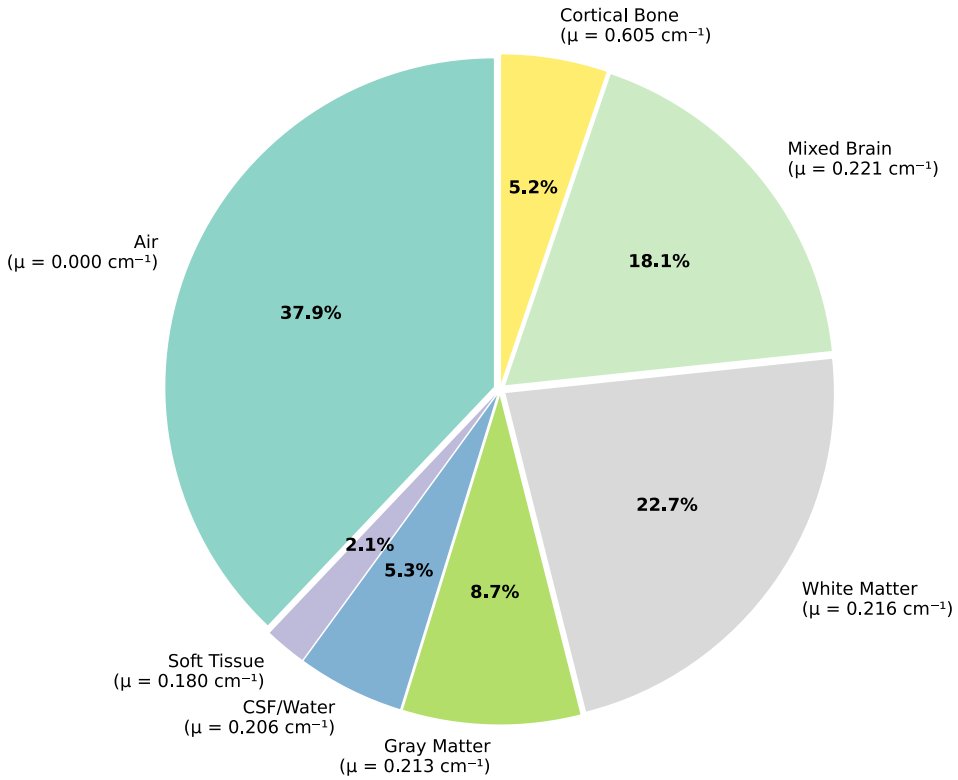


Figure 4.11: Distribution of linear attenuation coefficient (μ) values in the skull voxel model. Only seven distinct values are present and their materials have been labelled using NIST's X-ray mass attenuation database [8].

Voxelised models that represent biological anatomy often exhibit a highly sparse distribution of linear attenuation coefficients (μ). This sparsity arises because biological volumes are composed of a finite and typically small set of constituent materials (e.g. air, soft tissue, bone). As a result, large spatial regions share identical attenuation properties.

This behaviour is clearly visible in the skull model used throughout this thesis. As shown in Figure 4.11, the entire volume contains only seven distinct μ values.

This observation motivated the use of a small look-up table (LUT) to compress the voxel representation. Instead of storing a full-precision floating-point attenuation coefficient per voxel, each voxel stores an index into a LUT containing the corresponding material coefficient. A LUT of 256 entries was used, enabling the voxel data to be stored as `uint8_t` values. This reduces the memory footprint by a factor of four compared to the original `float` representation (1 byte versus 4 bytes per voxel).

The LUT itself was placed in constant memory to exploit caching and broadcast behaviour, since many neighbouring voxels reference the same material entry.

It is important to note that this optimisation primarily targets memory capacity rather than execution speed. The dominant performance bottleneck of the voxel ray casting kernel, that is, uncoalesced global memory accesses, remains unchanged. Consequently, no substantial runtime improvement is expected from this approach. The principal benefit is the reduced memory footprint, which enables significantly larger volumes to fit within the available GPU memory.

4.3.5.1 Metrics

Table 7: Kernel runtime (ms) before and after using look-up tables (LUTs).

Projection	**0°**	**30°**	**45°**	**60°**	**90°**	**120°**	**135°**	**150°**	**180°**	**Average**
Before	2.66	3.31	3.9	3.27	2.64	3.27	3.4	3.31	4.19	3.33 ± 0.5
After	2.77	3.47	3.57	3.42	2.74	3.43	3.57	3.47	2.77	3.24 ± 0.37
Speedup	0.96	0.96	1.09	0.95	0.97	0.95	0.95	0.95	1.51	1.03 ± 0.18

As expected, the use of a LUT does not yield a statistically significant speedup. However, the reduced memory footprint allows rendering of voxel models up to four times larger than before, which is a practical and meaningful improvement when working with high-resolution anatomical data.

4.4 Limitations

Even after implementing the final iteration, this approach suffers from the following limitations inherent to the data format.

4.4.1 Non-uniform render times

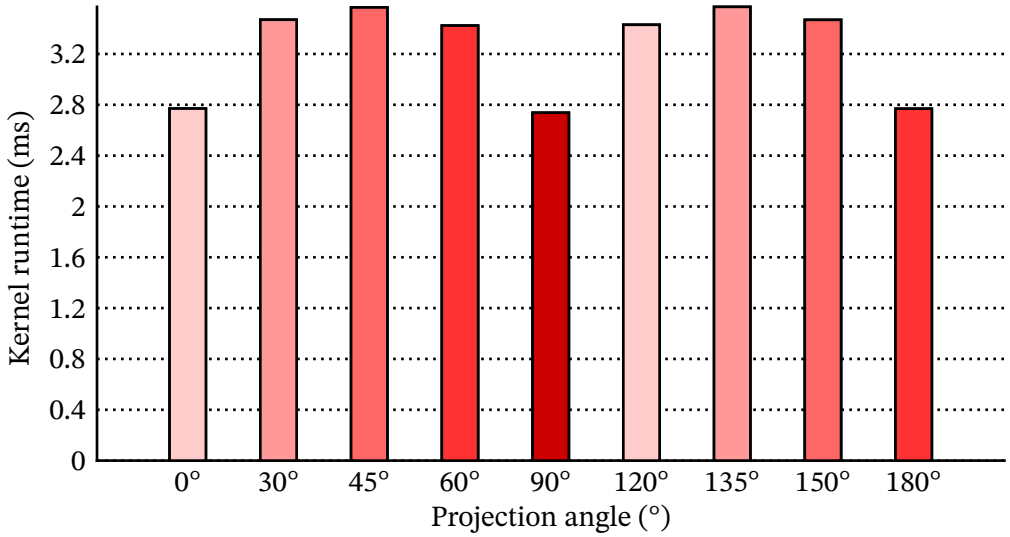


Figure 4.12: Kernel runtime (ms) for each projection angle. They are view-dependent and symmetric around 90°.

As discussed in Subsection 2.4.2, GPU performance strongly benefits from coalesced memory accesses, where consecutive threads access consecutive memory locations. In the voxel-based rendering kernel, however, this condition is not always satisfied. The sequence of voxels intersected by a ray depends on the viewing direction and the orientation of the object, which can lead to irregular and strided memory access patterns.

This view-dependent access skew is reflected in the kernel execution times reported in Table 6.1 and Figure 4.12. Performance is highest for projection angles close to 0 degree, where memory accesses are largely coalesced and fewer voxels are intersected. In contrast performance degrades as the rays hit the model diagonally (45 °), where voxel-ray intersections increase and memory access patterns become increasingly irregular.

4.4.2 Inefficient memory usage

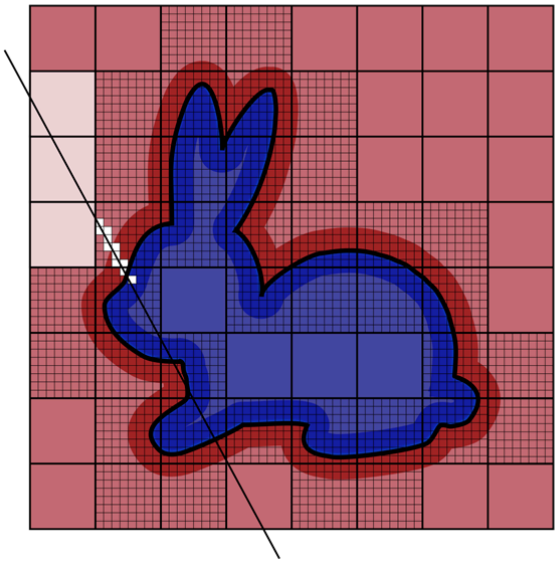


Figure 4.13: Illustration of hierarchical digital differential analyser (DDA) in OpenVDB. Note how uniform regions are clustered into a single datapoint. Courtesy: NVIDIA Technical Blog[31].

Using voxelised models directly for X-ray simulation consumes significantly more memory than necessary. In a dense voxel grid, every voxel occupies storage regardless of whether the region contains meaningful variation. Large homogeneous regions, such as a chunk of soft tissue or bone, are redundantly replicated across many voxels. This lack of compression leads to wasteful memory usage and can make large anatomical models, like a full human body, extremely costly to store.

Furthermore, raw voxel grids do not support level of detail. To faithfully capture fine anatomical features, every voxel must be as small as the finest detail in the model. As a result, the memory requirement grows cubically with the desired resolution, quickly becoming infeasible for high-resolution simulations.

More advanced data structures such as octrees or bounded volume heirarchies (BVHs) can mitigate these issues by encoding empty or homogeneous regions efficiently and enabling multi-resolution traversal. Libraries like OpenVDB [55] or NanoVDB [56] implement these strategies, compressing sparse voxel data and providing fast access for ray-based simulation. As shown in Figure 4.13, leveraging such sparse representations is a natural avenue for improving both memory usage and computational performance.

[31] https://developer.nvidia.com/blog/accelerating-openvdb-on-gpus-with-nanovdb/

5

Simulation: Gaussian mixture models (GMMs)

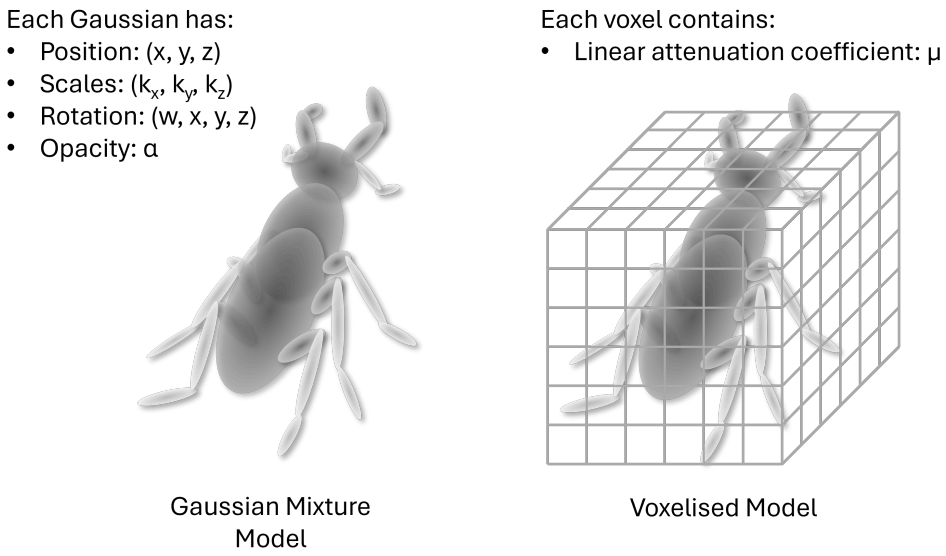


Figure 5.1: An illustration comparing GMMs to voxelised models. Courtesy: Adapted from R²-Gaussian [9].

As discussed in Subsection 4.4.2, voxelised models, particularly the raw voxel grids used throughout this thesis and still prevalent in industrial pipelines, encode volumetric data very inefficiently. Large regions of uniform or slowly varying material properties are represented using dense, regularly sampled grids, leading to excessive memory usage and bandwidth pressure during rendering.

An intuitive analogy can be drawn from image storage formats. Using a raw voxel grid to represent a volume is akin to storing images as uncompressed bitmaps, even when lossless formats such as PNG or TIFF can represent the same visual information far more compactly. While voxel grids are simple and convenient, they fail to exploit the underlying structure and redundancy present in real-world data.

Since the performance of the baseline renderer is largely limited by memory access rather than arithmetic throughput, adopting a more compact data representation is a natural direction for optimisation. A representation that encodes the same volumetric information using fewer memory accesses can directly translate into higher rendering performance.

This observation motivates the use of Gaussian primitives, and more generally, GMM, as an alternative volume representation. Gaussian functions offer several properties that make them well suited for this task:

- They naturally support level-of-detail control, as individual Gaussians can vary in scale and spatial extent.
- They provide a continuous representation of matter, in contrast to the discrete sampling imposed by voxel grids, making them better aligned with the underlying physical interpretation of volumetric attenuation[32].
- They allow large homogeneous or smoothly varying regions to be represented compactly using a small number of primitives.

The remainder of this chapter explores how Gaussian mixture models can be used to encode volumetric data efficiently, and how such representations can be integrated into the rendering pipeline to reduce memory traffic while preserving image quality.

5.1 Converting voxelised models to GMMs

GMMs are a relatively recent development in computer graphics [57] and are even more novel in medical imaging and CT applications [38]. To date, no published work proposes their use as a first-class storage representation for CT reconstruction models or for DRR generation. Instead, existing approaches employ GMMs as an intermediate representation within reconstruction pipelines [5,9,38].

In these pipelines, a limited number of projections are first generated from a voxelised model. A GMM is subsequently fitted to the projection data through an optimisation or training procedure. Once trained, the GMM becomes the primary representation for subsequent rendering or reconstruction tasks. The resulting models are typically exported as point clouds[33] consisting of Gaussian primitives each parameterised by position, density, scale, and rotation.

Training such models constitutes a substantial research topic in its own right and lies beyond the scope of this thesis. Consequently, this work assumes that a Gaussian point cloud[33] has been trained using methods available in the literature (e.g., R^2-Gaussian). The trained model is then converted into a custom binary format tailored to the rendering pipeline developed in this thesis using the provided conversion scripts.

The binary layout of the resulting `.gmm` file is defined as follows:

```
/// The file must be stored in little-endian byte order.
struct GaussianMixtureModel {
    size_t count = 0;  ///< Number of Gaussians.

    glm::vec3* positions;  ///< Gaussian positions (means) in world space.
    float* densities;      ///< Pre-activated scalar density for each Gaussian.
    glm::vec3* scales;     ///< Pre-activated per-axis scales for each Gaussian.
    glm::quat* rotations;  ///< Normalized rotation quaternions (w, x, y, z).
};
```

Listing 1: C struct defining the data layout of the GMM file format (.gmm) used for rendering.

[32] Gaussians impose smooth basis functions, which makes modelling sharp attenuation discontinuities (e.g., organ or bone interfaces) inherently challenging compared to piecewise-constant voxel grids. However, they may capture fine-scale structure more compactly and offer stronger perceptual compression, so whether they constitute a superior anatomical representation remains an open question.

[33] The term “point cloud” is used because the models are commonly stored in `.ply` format, even though each element represents an anisotropic Gaussian primitive rather than a geometric point.

5.2 Data encoding

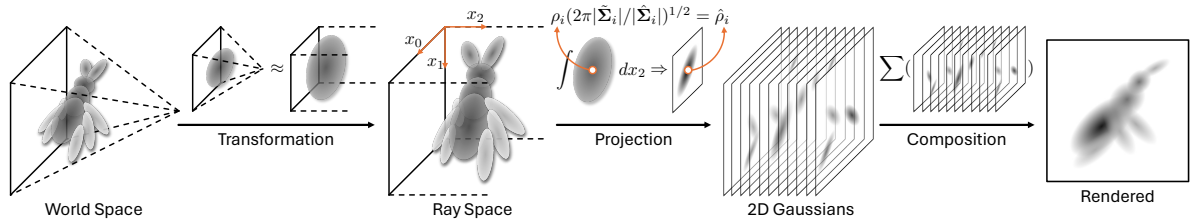


Figure 5.2: An overview of the forward pass of the R²-Gaussian algorithm. Courtesy: R²-Gaussian [9].

Unlike voxelised models, where the Beer–Lambert law Eq. (2.2) manifests as a discrete summation of voxel contributions along a ray, GMMs do not require explicit numerical evaluation of a line integral through a sampled grid.

Given a set of Gaussian primitives, each Gaussian is first *splatted* onto the projection plane, analogous to the procedure in 3DGS. The contribution of each projected Gaussian, modulated by its density or opacity, is accumulated in image space to determine the final pixel intensity. Rather than stepping along a ray and sampling attenuation values at discrete voxel locations, rendering with Gaussian primitives evaluates the analytic projection of continuous volumetric functions.

This constitutes a fundamental difference in the rendering process. Voxel-based methods approximate the volume integral through discrete sampling, whereas Gaussian-based approaches exploit the continuous nature of the primitives and their closed-form projection behaviour. Figure 5.2 illustrates this rendering pipeline in detail.

5.3 GPU implementation

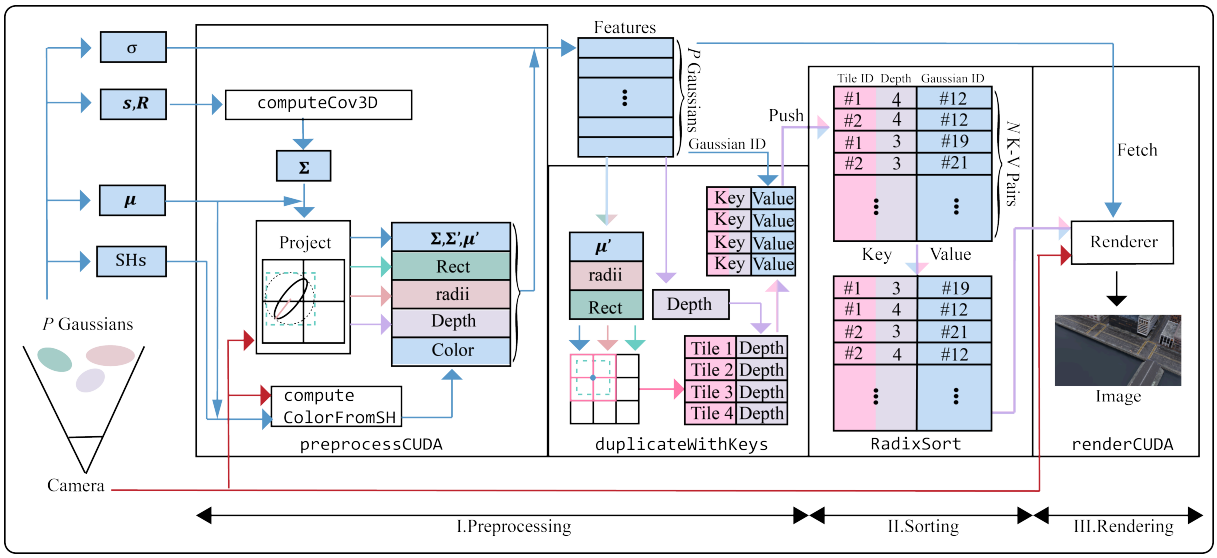


Figure 5.3: An overview of the forward pass of the 3DGS algorithm. Courtesy: FlashGS [10].

The rendering pipeline used in this work is based on the R²-Gaussian algorithm [9], which itself builds upon X-Gaussian [38]. X-Gaussian was the first work to adapt the 3DGS framework [57] for X-ray computed tomography.

As in 3DGS, the forward pass of the pipeline (see Figure 5.3) can be decomposed into three high-level stages:

- Preprocessing
- Sorting
- Rendering

However, while 3DGS is designed for conventional optical rendering, where light reflects off surfaces and radiance is view-dependent, R^2-Gaussian models X-ray transmission, where radiation penetrates the volume and is attenuated along the ray path. This fundamental difference in imaging physics leads to several important deviations from the original 3DGS pipeline, beyond the structural overview shown in Figure 5.3:

- View-dependent appearance modeling via spherical harmonics is omitted, as X-ray attenuation is isotropic and independent of viewing direction [38].
- Gaussian primitives represent radiative attenuation rather than surface radiance, and therefore contribute additively along the ray according to a transmission model instead of alpha compositing.
- The rendering stage integrates contributions from all intersected Gaussians along each ray, rather than terminating at the first visible surface.
- Camera and ray geometry are defined by known CT scanner parameters, eliminating the need for learned camera poses or structure-from-motion initialisation.
- The rasterisation step is adapted to compute line integrals of attenuation, aligning the forward pass with the physics of X-ray image formation.

These differences give rise to the algorithm illustrated in Figure 5.2.

5.3.1 Baseline

For the baseline implementation, the existing code from R^2-Gaussian algorithm was extracted out from the project and ran as it is with the trained skull model.

5.3.1.1 Metrics

Table 1: Kernel runtime (ms) for each projection angle in the baseline implementation.

0°	30°	45°	60°	90°	120°	135°	150°	180°	Average
70.99	71.67	73.54	73.68	73.67	74.76	75.22	76.61	76.33	74.05 ± 1.91

5.3.2 Remove depth sorting

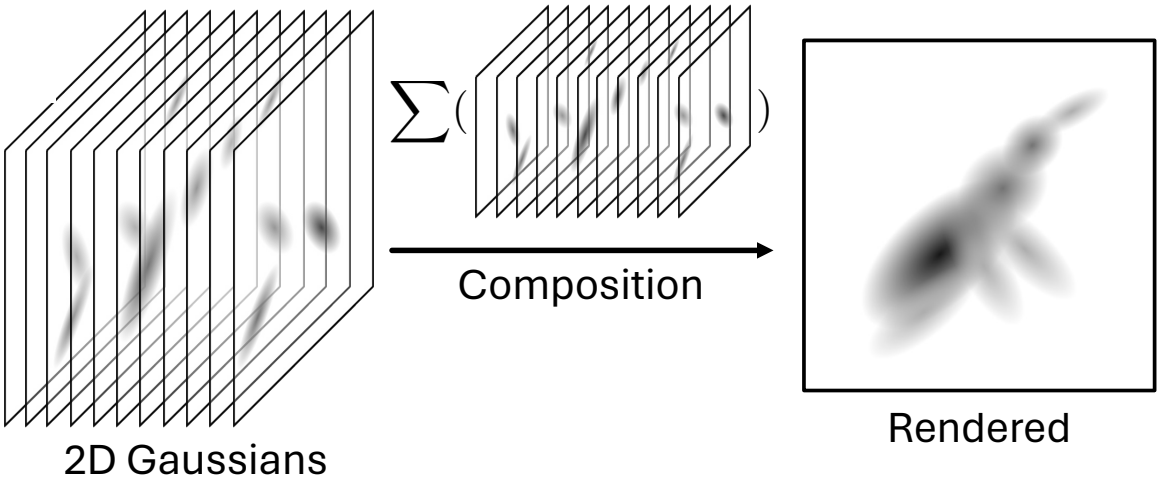


Figure 5.4: An illustration of the final accumulation step. Since it is just a sum, it is commutative, and hence, order-independent. Courtesy: R²-Gaussian [9].

Unlike 3DGS, tile contributions in R²-Gaussian do not require depth sorting prior to accumulation (see Figure 5.4). Since the formulation models purely transmissive radiative transport rather than alpha compositing of semi-transparent surfaces, the ordering of primitives along the ray does not affect the final result. Although the original authors were aware of this property, as indicated by a TODO comment in the implementation, they retained the depth-sorting step to remain closely aligned with the 3DGS pipeline.

For the performance-critical use case considered in this thesis, this additional ordering step introduces unnecessary overhead. Consequently, depth sorting was removed (see Listing 5.2), simplifying the pipeline and reducing per-tile computation without affecting numerical correctness.

```
// Before ============================================================
// For each tile that the bounding rect overlaps, emit a key/value pair
// Key is |  tile ID  |      depth      |
// Sorting yields Gaussians sorted by tile, then by depth
for (int y = rect_min.y; y < rect_max.y; y++) {
    for (int x = rect_min.x; x < rect_max.x; x++) {
        uint64_t key = y * grid.x + x;
        key <<= 32;
        key |= *((uint32_t*)&depths[idx]);
        gaussian_keys_unsorted[off] = key;
        gaussian_values_unsorted[off] = idx;
        off++;
    }
}

// After =============================================================
// For each tile that the bounding rect overlaps, emit a key/value pair
// Key contains only tile ID — no depth sorting within tiles
for (int y = rect_min.y; y < rect_max.y; y++) {
    for (int x = rect_min.x; x < rect_max.x; x++) {
        uint64_t key = y * grid.x + x;
        key <<= 32;
        // No depth in key — Gaussians within tile are unsorted
        gaussian_keys_unsorted[off] = key;
        gaussian_values_unsorted[off] = idx;
        off++;
    }
}
```

Listing 2: Changes made to `duplicate_with_keys` for depth sorting removal.

5.3.2.1 Metrics

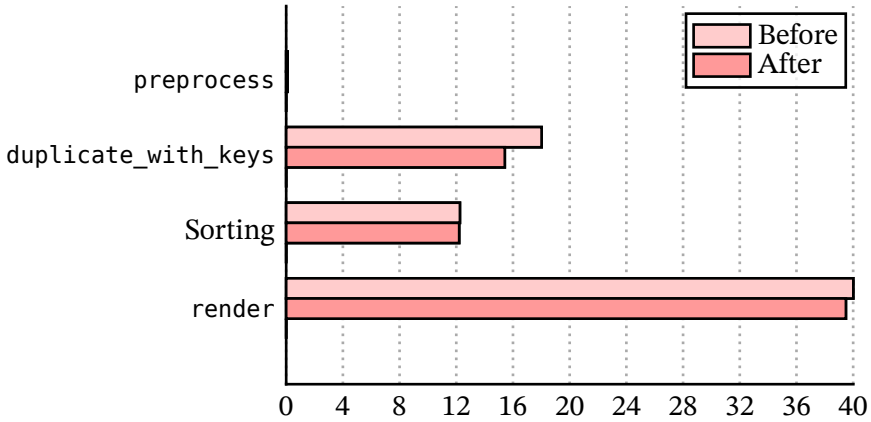


Figure 5.5: Runtime breakdown for each kernel before and after depth sorting removal.

As shown in Figure 5.5, the `duplicate_with_keys` now runs faster while the other kernels remain unaffected. This is expected because the only the depth bookkeeping was removed from this `duplicate_with_keys`. Table 5.2 shows that this optimisation helps speed up the application by 1.04x.

Table 2: Kernel runtime (ms) before and after removing depth sorting.

Projection	0°	30°	45°	60°	90°	120°	135°	150°	180°	Average
Before	70.99	71.67	73.54	73.68	73.67	74.76	75.22	76.61	76.33	74.05 ± 1.91
After	67.82	68.02	69.34	69.59	73.74	74.88	72.13	73.3	72.73	71.28 ± 2.63
Speedup	1.05	1.05	1.06	1.06	1	1	1.04	1.05	1.05	1.04 ± 0.02

5.3.3 Coalesce `duplicate_with_keys()` global accesses

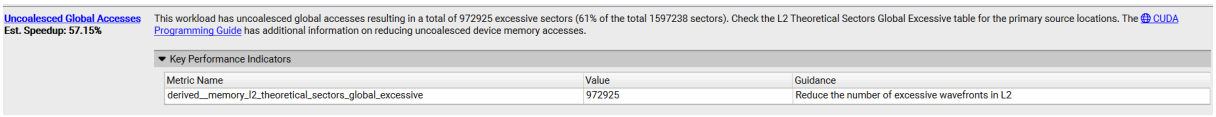


Figure 5.6: NSight Compute's suggestion to improve the kernel's memroy access pattern.

Looking at the profile for the previous iteration revealed theat the `duplicate_with_keys()` kernel, was performing a huge amount of uncoalesced memory accesses (see Figure 5.6). The original 3DGS `duplicate_with_keys()` kernel assigns one thread per Gaussian. Each thread emits N_g output entries, where N_g is the number of tiles overlapped by that Gaussian. Because different Gaussians overlap different numbers of tiles, the number of writes per thread varies significantly. As a result, adjacent threads within a warp write to non-contiguous memory locations. For example, thread 0 may write to indices 0–5, while thread 1 writes to 6–20. Across a full warp, this leads to highly scattered global memory writes and poor coalescing.

This issue can be resolved by inverting the parallelisation strategy. Instead of launching one thread per Gaussian that produces many output entries, we launch one thread per output entry. Thread i writes exclusively to output[i], ensuring that adjacent threads in a warp write to consecutive memory addresses. This guarantees fully coalesced global memory stores.

To determine which Gaussian owns a given output entry, each thread performs a binary search over the prefix-sum offsets of per-Gaussian tile counts. This requires $O(\log G)$ reads, where G is the number of Gaussians, followed by simple arithmetic to compute the tile index within the Gaussian's bounding region. Although this introduces additional reads, they are highly cache-friendly: neighbouring threads typically search similar regions of the prefix-sum array, leading to strong spatial locality.

This transformation effectively trades scattered global writes for coalesced writes combined with cached read operations. On modern GPUs, where global memory bandwidth is often the dominant bottleneck, this exchange is advantageous. The result is improved memory efficiency, higher effective throughput, and better overall kernel performance.

5.3.3.1 Metrics

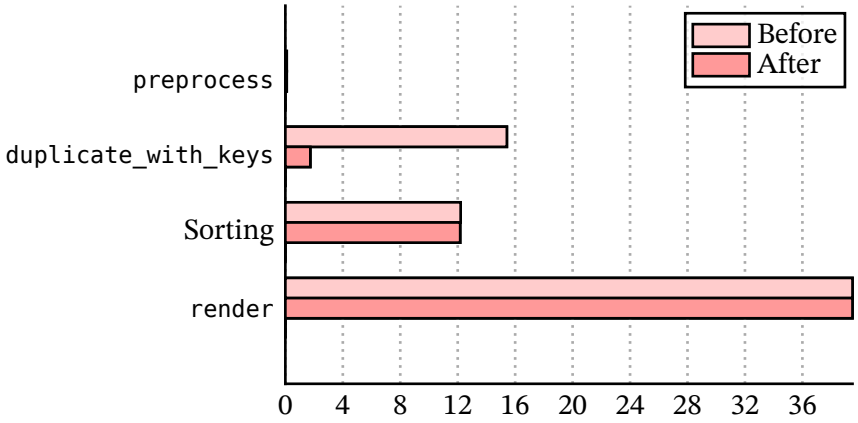


Figure 5.7: Runtime breakdown of different kernels before and after coalescing global memory accesses.

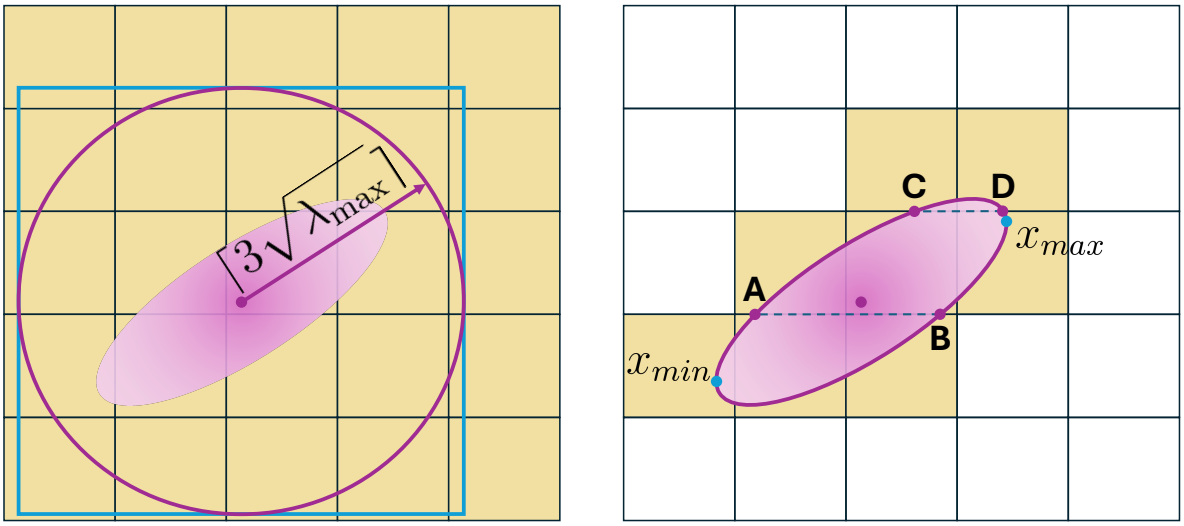
Figure 5.7 shows a significant speedup of 8.87x for `duplicate_with_keys`, while the remaining kernels are unaffected, as expected. As summarised in Table 5.3, this optimisation yields an average speedup of approximately 1.27x compared to the previous iteration.

The performance improvement is directly attributable to the revised memory access pattern. By replacing scattered global writes with fully coalesced stores, global memory latency is reduced and effective bandwidth utilisation increases. Consequently, kernel execution time decreases substantially without introducing additional computational overhead.

Table 3: Kernel runtime (ms) before and after coalescing `duplicate_with_keys()` lookups.

Projection	0°	30°	45°	60°	90°	120°	135°	150°	180°	Average
Before	67.82	68.02	69.34	69.59	73.74	74.88	72.13	73.3	72.73	71.28 ± 2.63
After	54.1	54.32	55.17	55.27	56.11	57.03	57.38	58.73	58.28	56.26 ± 1.68
Speedup	1.25	1.25	1.26	1.26	1.31	1.31	1.26	1.25	1.25	1.27 ± 0.03

5.3.4 Use AccuTile



(a) Tile–Gaussian intersection test used in 3DGS.

(b) AccuTile intersection test from SpeedySplat [11].

Figure 5.8: Comparison between the tile intersection strategy in 3DGS and the AccuTile method proposed in SpeedySplat [11].

During rendering, a tile-to-Gaussian mapping is constructed to determine which Gaussians contribute to which screen-space tiles. This mapping phase is critical, as it determines the workload of all subsequent rasterisation stages.

Inspection of the 3DGS implementation revealed that tile intersection is determined by approximating each projected Gaussian as a circle in screen space (see Figure 5.9). While simple and computationally inexpensive, this approach is conservative: the circular bound overestimates the true footprint of an anisotropic 2D Gaussian. Consequently, many tiles are marked as active even though their actual contribution is negligible. This overestimation increases the number of tile–Gaussian pairs, inflates memory traffic, and negatively impacts rendering performance.

To address this inefficiency, existing optimisation work on 3DGS was considered. A more accurate alternative is provided by the AccuTile algorithm introduced in SpeedySplat [11] (see Figure 5.9). Instead of relying on a coarse circular approximation, AccuTile performs a tighter intersection test based on the elliptical support of the projected Gaussian. This reduces the number of falsely activated tiles, leading to a smaller working set and improved computational efficiency.

5.3.4.1 Metrics

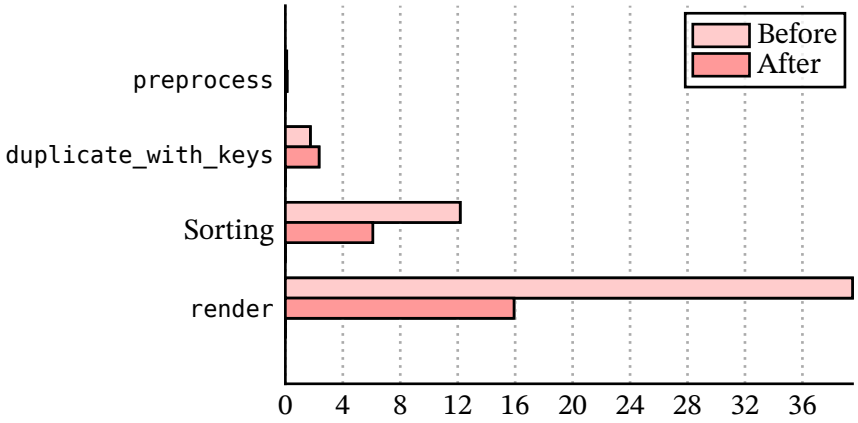


Figure 5.9: Runtime (ms) breakdown across kernels before and after the AccuTile optimisation.

As shown in Figure 5.9, the AccuTile optimisation substantially reduces the overall rendering workload, decreasing the time spent in `render` from 39.49 ms to 15.92 ms. This reduction is primarily due to the aggressive pruning of tile–Gaussian pairs, which lowers the number of primitives processed in subsequent rasterisation stages, i.e. sorting and `render`.

The additional pruning logic introduces a modest overhead of approximately 1 ms in `duplicate_with_keys`. However, this cost is negligible compared to the downstream savings achieved by reducing the working set size.

Table 5.4 corroborates this observation, showing that the optimisation yields an average speedup of 2.19x relative to the previous iteration.

Table 4: Kernel runtime (ms) before and after applying AccuTile for tile–Gaussian intersection.

Projection	0°	30°	45°	60°	90°	120°	135°	150°	180°	Average
Before	54.1	54.32	55.17	55.27	56.11	57.03	57.38	58.73	58.28	56.26 ± 1.68
After	24.73	23.85	24.95	25.09	27.03	26.09	26.3	26.8	26.62	25.72 ± 1.1
Speedup	2.19	2.28	2.21	2.2	2.08	2.19	2.18	2.19	2.19	2.19 ± 0.05

5.4 Limitations

Even after implementing the final iteratinon of the algorithm, some limitations still remain.

5.4.1 Sorting and binning overhead

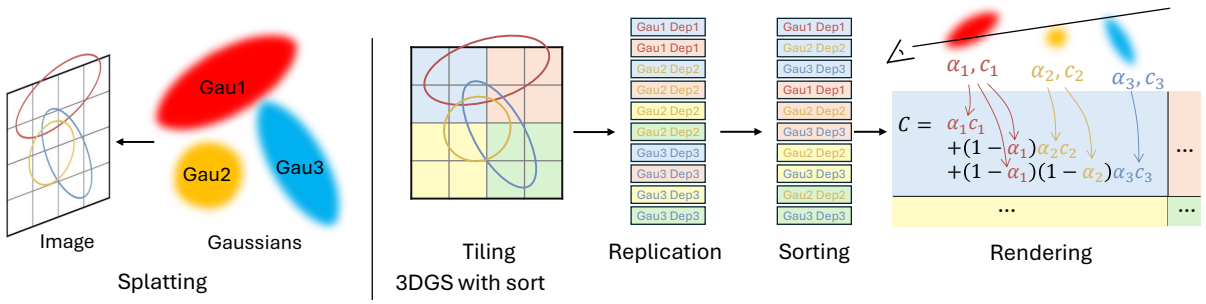


Figure 5.10: An overview of the 3DGS rendering pipeline. The sorting step is necessary to convert the Gaussian→Tile mapping to Tile→Gaussian mapping. Courtesy: Sort Free Gaussians [12].

Although the proposed implementation removes explicit depth sorting, the method still inherits a strctural sorting step from 3DGS as shown in Figure 5.10. In tile-based rasterisation, Gaussians are first assigned to screen-space tiles. This typically requires sorting or reordering primitives and computing prefix sums to determine tile ranges before rasterisation.

While this operation is parallelisable on the GPU, it introduces non-negligible overhead and additional memory traffic, particularly for large numbers of Gaussians or high-resolution projections. Thus, even without depth ordering, a global reorganisation of primitives per view remains necessary.

Recent work such as *Sort-free Gaussian Splatting via Weighted Sum Rendering* [12] suggests alternative accumulation strategies that reduce or remove sorting-based dependencies. Adapting similar ideas to this work may further improve scalability.

6 Results

Table 6.1 shows the kernel runtimes for each iteration of the voxel-based renderer, along with the average runtime and speedup compared to the baseline. The results indicate that each optimisation step contributed to a significant reduction in rendering time, culminating in a final speedup of approximately 316x compared to the original baseline implementation. As shown in Figure 6.1, the presented algorithm also outperforms the state-of-the-art TIGRE implementations, with a 10.07x speedup when compared to its Siddon implementation and a 7.75x speedup when compared to its interpolated algorithm with an accuracy of 0.5 voxels per sample[34]. Moreover, the presented algorithm allows rendering of volumes 4 times larger than what TIGRE does by using a LUT.

Table 1: A summary of kernel runtimes (ms) of various projection angles and speedup for each iteration.

Iteration	0°	30°	45°	60°	90°	120°	135°	150°	180°	Average	Speedup
TIGRE (Siddon)	22.65	36.63	41.54	36.16	23.15	36.25	38.38	36.6	22.65	32.67 ± 7.57	N/A
TIGRE (Interpolated)	24.97	25.05	25.08	25.1	25.05	25.09	26.1	25.05	24.97	25.16 ± 0.36	N/A
Baseline	931.88	1132.57	1148.59	1102.2	796.03	1098.41	1147.19	1112.56	777.27	1027.41 ± 151.33	N/A
Move tiling logic to CUDA	46.66	68.99	71.79	68	46.52	67.99	71.79	68.98	46.66	61.93 ± 11.57	16.74 ± 1.27
Use FP32 numbers	4.21	5.65	5.85	5.59	4.16	5.58	5.85	5.64	4.21	5.19 ± 0.76	11.85 ± 0.55
Reduce branching	2.66	3.31	3.9	3.27	2.64	3.27	3.4	3.31	4.19	3.33 ± 0.5	1.58 ± 0.23
Use LUTs	2.77	3.47	3.57	3.42	2.74	3.43	3.57	3.47	2.77	3.24 ± 0.37	1.03 ± 0.18

[34]Subsequent discussions only consider the Siddon algorithm from TIGRE since it algorithmically similar to the voxel rendering algorithm in this thesis. Additionally, the interpolated variant is not a fair comparison since it does not work with LUTs and also trades off image quality for render times through the accuracy parameter.

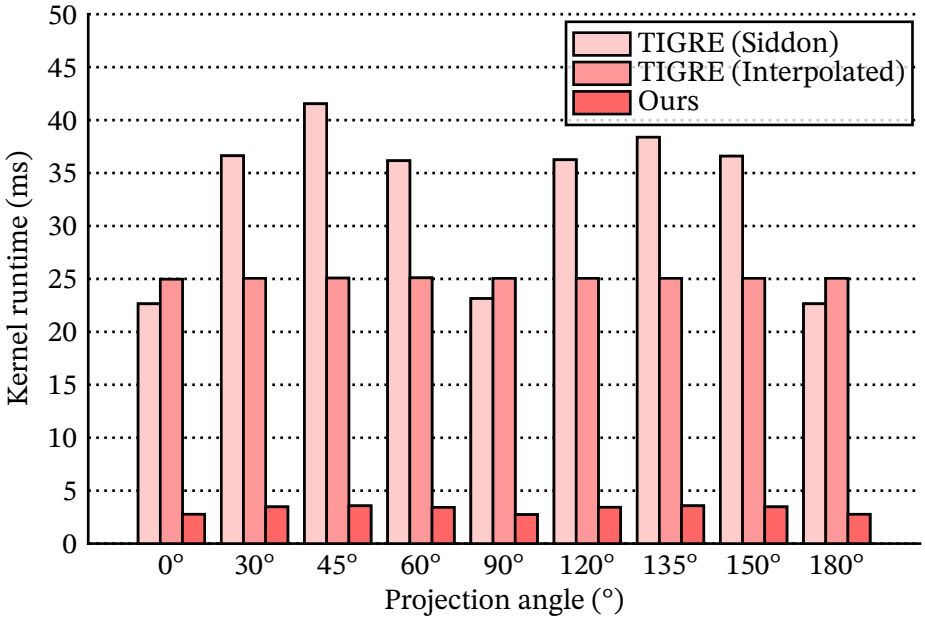


Figure 6.1: Comparing our voxelised algorithm with the state of the art implementation: TIGRE.

Table 2: Image fidelity metrics compared to the baseline and TIGRE.

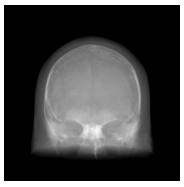
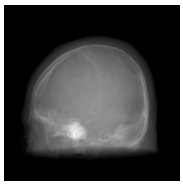
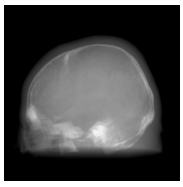
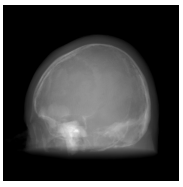
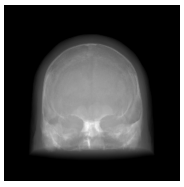
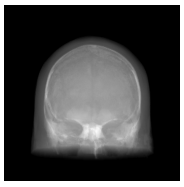
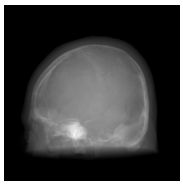
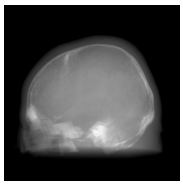
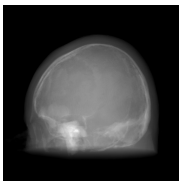
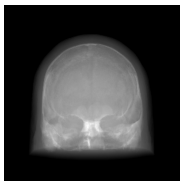
Projection (°)	**0 °**	**45 °**	**90 °**	**135 °**	**180 °**
TIGRE (Siddon)					
Our algorithm					
SSIM	1.00	1.00	1.00	1.00	1.00
PSNR	86.78 dB	75.69 dB	85.62 dB	76.29 dB	86.01 dB

Table 6.2 shows that the proposed voxel rendering algorithm achieves image quality equivalent to TIGRE, the state of the art, as indicated by an SSIM of 1 (structurally identical images) and PSNR values well above 45 dB, which corresponds to excellent visual fidelity.

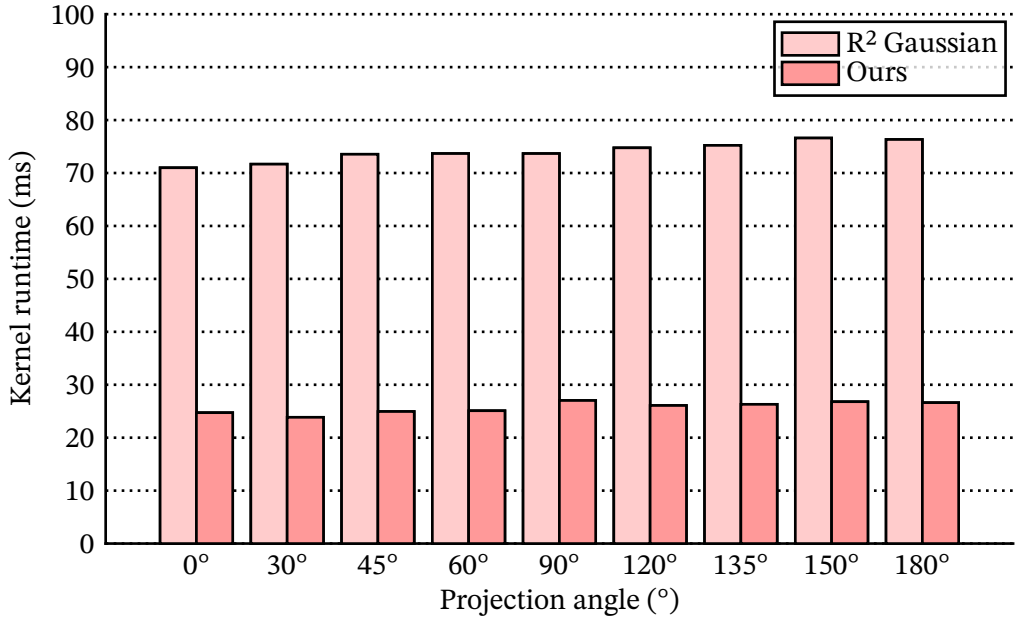


Figure 6.2: Comparing our GMM algorithm with the state of the art implementation: R² Gaussian.

On the Gaussian side of things, the results are also promising. As shown in Figure 6.2, our implementation of the R²-Gaussian algorithm achieves real-time performance on the trained skull model, with frame rates in the 40 FPS regime, which is a 2.88x improvement over the reference implementation, as shown in Table 6.3. The use of Gaussian primitives allows for a more compact representation of the volume, reducing memory bandwidth requirements and enabling faster rendering.

Table 3: A summary of kernel runtimes (ms) for different stages of the GMM rendering pipeline for each optimisation.

Iteration	Preprocess	DuplicateWithKeys	Sorting	Render	Total
R² Gaussian (Baseline)	0.1	18.02	12.26	40.01	70.99
Remove depth sorting	0.09	15.42	12.2	39.49	67.82
Coalesce DuplicateWithKeys	0.09	1.74	12.17	39.49	54.1
Use AccuTile	0.12	2.35	6.09	15.92	24.73

As shown in Figure 6.2, GMMs eliminate the pronounced view-dependent runtime skew observed in the voxel-based implementation. The voxel renderer exhibits significant angle-dependent variation, with projections near 90° taking considerably longer due to increased voxel–ray intersections and reduced memory coherence. In contrast, the Gaussian representation yields nearly uniform rendering times across projection angles, as the workload depends on a fixed set of primitives rather than direction-dependent grid traversal.

This is achieved while providing an approximately 26x compression ratio compared to the original dense voxel grid, substantially reducing memory bandwidth requirements and contributing to the overall performance gains.

The optimisation techniques introduced in this work, that is, coalescing `duplicate_with_keys`, removal of depth sorting, and tighter tile–primitive intersection testing, are not limited to a single

implementation. They directly apply to related Gaussian-based frameworks such as X-Field [5] and Radiative Gaussian Splatting [38], which share similar rendering pipelines.

Table 4: Image fidelity metrics comparing TIGRE with our GMM algorithm.

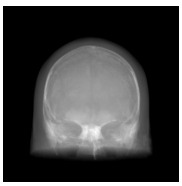
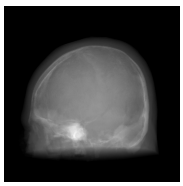
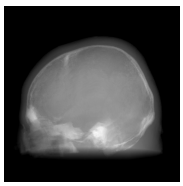
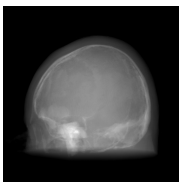
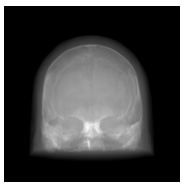
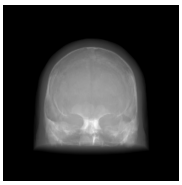
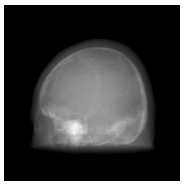
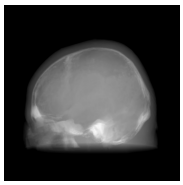
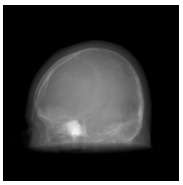
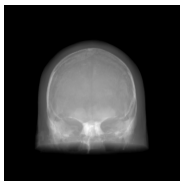
Projection (°)	**0 °**	**45 °**	**90 °**	**135 °**	**180 °**
TIGRE (Siddon)					
Our algorithm					
SSIM	0.87	0.84	0.86	0.86	0.84
PSNR	24.07 dB	24.04 dB	22.28 dB	22.81 dB	22.66 dB

Table 5: Image fidelity metrics comparing the final GMM algorithm with R^2 Gaussian.

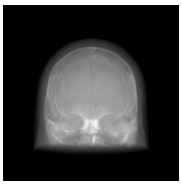
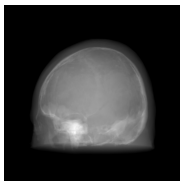
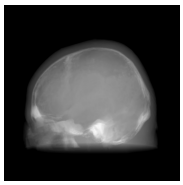
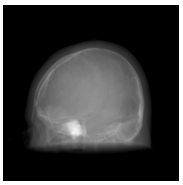
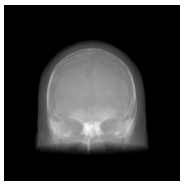
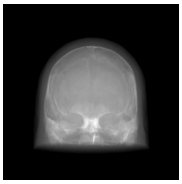
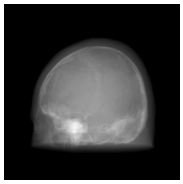
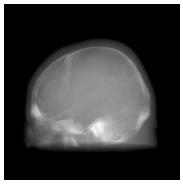
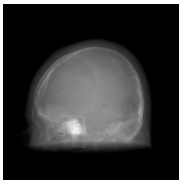
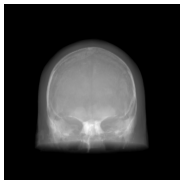
Projection (°)	**0 °**	**45 °**	**90 °**	**135 °**	**180 °**
R^2 Gaussians					
Our algorithm					
SSIM	1.00	1.00	1.00	1.00	1.00
PSNR	71.00 dB	71.67 dB	71.79 dB	72.02 dB	71.67 dB

Table 6.4 compares renders produced by the proposed Gaussian-based renderer with those generated by TIGRE. The quantitative metrics indicate a reduction in fidelity in this direct comparison, confirming the presence of a quality gap relative to the voxel-based reference.

To assess whether this loss originates from the rendering implementation itself, the proposed renderer is additionally compared against the baseline R^2-Gaussian implementation (see Table 6.5). In this case, no measurable difference is observed: SSIM $= 1$ and PSNR values well above 45 dB indicate structurally identical images and excellent numerical agreement. This demonstrates that the proposed renderer introduces no additional fidelity loss beyond that inherent to the trained Gaussian model.

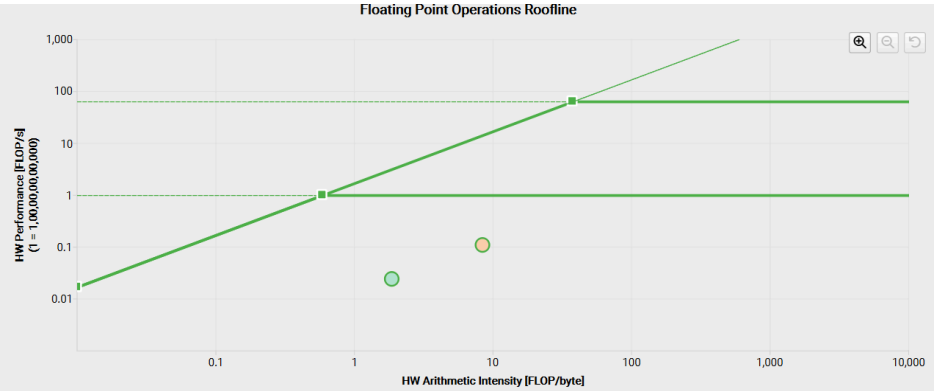
The R²-Gaussian paper [9] reports SSIM values of approximately 0.959 and PSNR values of 38.88 dB for models trained on synthetic datasets with 75 views, indicating that reconstruction quality depends strongly on training configuration. The observed fidelity gap relative to TIGRE can therefore be attributed to the limitations of the trained Gaussian model and may be reduced through improved training strategies, increased view coverage, or enhanced optimisation procedures.

7 Conclusion

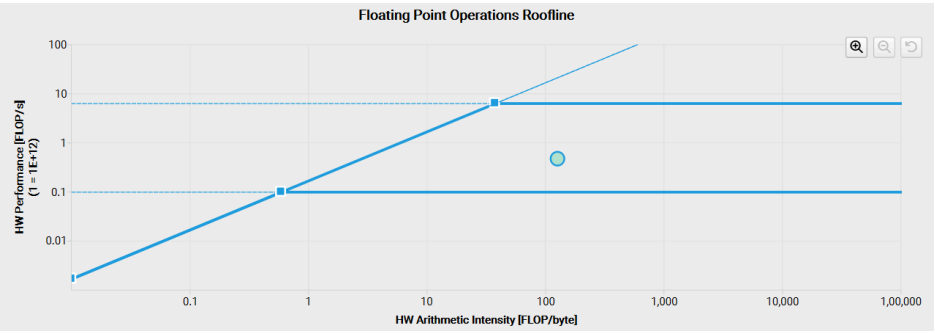
7.1 Addressing the research questions

The research questions posed in Section 1.3 aimed to evaluate both the feasibility and broader impact of GPU-accelerated X-ray simulation. Based on the results presented in this thesis, they can now be addressed as follows.

7.1.1 What maximum performance can be achieved when porting the existing CPU-based X-ray simulation algorithm to the graphics processing unit (GPU)?



(a) CUDA Tiling; FP32 (green) operations are memory bound but FP64 (yellow) operations are compute bound.



(b) Use LUTs The kernel is only having FP32 operations and is compute bound.

Figure 7.1: Roofline charts for initial and final iterations showing rooflines for FP32 (higher) and FP64 (lower).

Porting the voxel-based ray traversal algorithm from CPU to CUDA resulted in an acceleration of approximately 317x over the single-core baseline. At typical projection resolutions, the implementation achieves more than the requirement for real-time performance, even outpacing the state of the art forward projection algorith TIGRE by $\approx$ 10x.

Importantly, this acceleration was achieved without modifying the physical image formation model. The discrete voxel traversal and Beer-Lambert attenuation formulation were preserved, yielding numerically equivalent results up to floating-point precision differences. The observed performance gain therefore reflects architectural acceleration rather than approximation.

As shown in Figure 7.1, Nsight Compute profiling indicates that kernel execution lies in the compute-bound region of the roofline model. However, achieved memory throughput reaches only a small fraction of the device peak. This discrepancy arises from irregular, view-dependent voxel access during ray traversal, which prevents effective memory coalescing. As a result, performance is limited by inefficient global memory transactions and memory access latency. Consequently, further low-level tuning yields diminishing returns without altering the underlying data access pattern.

7.1.2 What are the bottlenecks this approach suffers from, and how can better algorithms be designed to alleviate them?

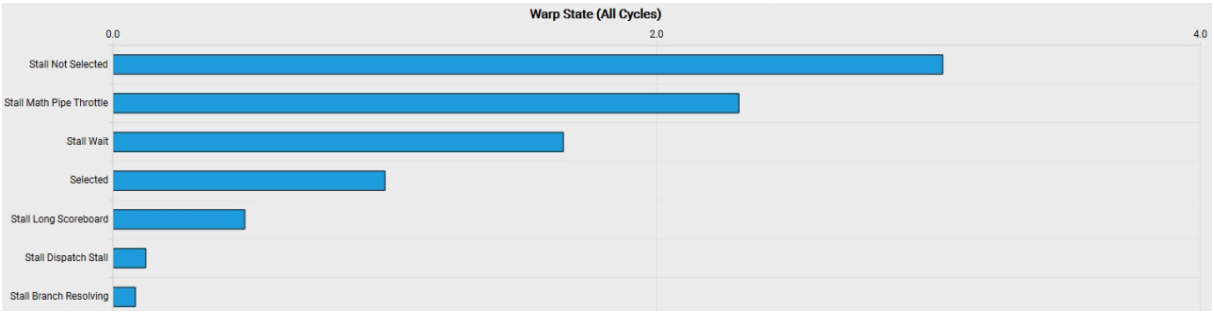


Figure 7.2: Truncated view of the warp state statistics from Nsight Compute showing the number of cycles per instruction the state the warp was in.

Performance analysis indicates that the dominant limitation arises from irregular global memory accesses caused by ray-dependent voxel traversal. These accesses introduce latency due to reduced coalescing. Since this behaviour is intrinsic to ray-based traversal, it cannot be fundamentally eliminated within the current representation.

Alternative plane-wise attenuation accumulation schemes were attempted to improve memory coherence. However, these approaches required usage of shared memory which hurt the occupancy, and resulted in approximately 20x slower execution.

Although the kernel exhibits high ALU pipeline utilisation, the warp state statistics in Figure 7.2 show that warps are not stalled by pipeline back-pressure. The kernel therefore remains compute-bound, with latency largely hidden by available parallelism.

Given these structural characteristics, further performance gains are unlikely to result from incremental kernel refinements alone. Instead, representational changes that improve traversal coherence or reduce per-ray work appear more promising.

Two algorithmic directions emerge:

- Hierarchical sparse voxel representations (e.g., NanoVDB [56] with HVDB path traversal) to enable empty-space skipping and reduce unnecessary traversal.

- Continuous Gaussian mixture representations to replace discrete voxel stepping with analytic primitive evaluation.

7.1.3 What are the trade-offs for each approach in terms of performance, memory usage, and visual fidelity?

The voxel-based representation offers:

- Exact correspondence to discretised CT data
- Direct physical interpretability
- No preprocessing or training requirement

However, it requires dense memory storage, high bandwidth, and exhibits projection-angle-dependent runtime skew.

The Gaussian mixture representation achieves approximately 26x compression on the skull dataset while maintaining high visual fidelity. Rendering times become more consistent across projection angles, and optimisations such as coalesced output writes and AccuTile pruning substantially reduce kernel runtime.

The trade-offs include:

- Additional preprocessing or training cost
- Approximation error dependent on mixture complexity
- Increased implementation complexity

Overall, the voxel method remains preferable when exact discretised fidelity is mandatory. The Gaussian representation is advantageous when scalability, memory efficiency, and architectural alignment with modern GPU hardware are primary concerns.

These results demonstrate that substantial performance gains arise not only from accelerating existing kernels, but from selecting representations that reduce memory pressure and increase arithmetic intensity.

7.2 Contributions

The main contributions of this thesis can be summarised as follows:

- A GPU-accelerated voxel-based X-ray rendering pipeline achieving approximately 316x speedup over the single-core CPU baseline while preserving numerical fidelity.
- A detailed performance analysis demonstrating that voxel-based simulation is fundamentally memory-bound, with global memory bandwidth and irregular access patterns constituting the primary bottlenecks.
- The introduction of a Gaussian mixture representation for X-ray simulation, reducing the limitations of dense voxel storage and achieving at least $\approx$ 26x compression on the skull dataset with a minimal loss in image quality.
- A GPU-accelerated rendering pipeline for GMMs that outperforms prior state-of-the-art implementations by 2.88x.
- A complete C++ implementation with Python bindings[28], released as open source and integrated into industrial workflows at Philips.

With these considerations, the proposed approach may also accelerate iterative CT reconstruction, where forward projection accounts for a substantial fraction of runtime. Any improvement in forward model efficiency therefore directly reduces overall reconstruction time.

7.3 Conclusion and outlook

This thesis demonstrates that substantial performance gains in X-ray simulation can be achieved through both architectural acceleration and representational redesign. The voxel-based CUDA implementation establishes a strong real-time baseline, but profiling shows that it is fundamentally memory-bound, limiting further gains from low-level kernel optimisation alone.

For voxel rendering, the most immediate improvement lies in adopting sparse hierarchical structures such as NanoVDB [56] with HVDB traversal. Since memory bandwidth and unnecessary voxel accesses were identified as the primary bottlenecks, enabling empty-space skipping via hierarchical traversal represents a clear low-hanging optimisation that could yield significant additional speedups with moderate implementation effort.

The Gaussian mixture representation explored in this work highlights a complementary direction: improving performance through more efficient scene representations. By replacing dense grid traversal with analytic primitives, rendering becomes more coherent and less view-dependent. Within this paradigm, integrating physically grounded formulations such as X-Field [5] represents the most direct next step toward more realistic rendering while retaining the performance advantages of primitive-based approaches.

Future work should also target the training process itself. Current Gaussian models are often over-parameterised; improved optimisation, pruning, or regularisation could produce leaner models with lower memory and runtime costs. Additionally, GMMs demonstrated an approximately 26x compression ratio for the skull dataset with acceptable image quality. While not yet a practical replacement for large-scale clinical storage, this suggests potential for compact dataset representation in simulation and research contexts.

Overall, the results indicate that meaningful advances in GPU-accelerated X-ray simulation arise not only from faster kernels, but from choosing representations that reduce memory pressure, exploit sparsity, and better align with modern hardware.

Bibliography

[1] M. Berger, Q. Yang, and A. Maier, *X-Ray Imaging*, in *Medical Imaging Systems: An Introductory Guide*, edited by A. Maier, S. Steidl, V. Christlein, and J. Hornegger (Springer International Publishing, Cham, 2018), pp. 119–145.

[2] J.-B. Letang, Physics, X-Ray Imaging: Physics, Instrumentation and Applications (n.d.).

[3] J. A. Seibert and J. M. Boone, X-Ray Imaging Physics for Nuclear Medicine Technologists. Part 2: X-Ray Interactions and Image Formation, Journal of Nuclear Medicine Technology **33**, 3 (2005).

[4] Philips Healthcare, Philips Azurion Bi-Plane System: Specifications (7 B20/15), technical report, 2015.

[5] F. Wang, J. Tao, J. Wu, H. Wang, B. Duan, K. Wang, Z. Yang, and Y. Yan, *X-Field: A Physically Grounded Representation for 3d X-Ray Reconstruction*, https://arxiv.org/abs/2503.08596.

[6] NVIDIA Corporation, *NVIDIA RTX Blackwell GPU Architecture*, https://images.nvidia.com/aem-dam/Solutions/geforce/blackwell/nvidia-rtx-blackwell-gpu-architecture.pdf.

[7] A. Biguri, Iterative Reconstruction and Motion Compensation in Computed Tomography on Gpus, Doctoral dissertation, 2017.

[8] J. Hubbell and S. Seltzer, *Tables of X-Ray Mass Attenuation Coefficients and Mass Energy-Absorption Coefficients (Version 1.4)*, http://physics.nist.gov/xaamdi.

[9] R. Zha, T. J. Lin, Y. Cai, J. Cao, Y. Zhang, and H. Li, *R2-Gaussian: Rectifying Radiative Gaussian Splatting for Tomographic Reconstruction*, in *Proceedings of the 38th International Conference on Neural Information Processing Systems* (Curran Associates Inc., Vancouver, BC, Canada, 2024).

[10] G. Feng, S. Chen, R. Fu, Z. Liao, Y. Wang, T. Liu, Z. Pei, H. Li, X. Zhang, and B. Dai, *Flashgs: Efficient 3d Gaussian Splatting for Large-Scale and High-Resolution Rendering*, https://arxiv.org/abs/2408.07967.

[11] A. Hanson, A. Tu, G. Lin, V. Singla, M. Zwicker, and T. Goldstein, *Speedy-Splat: Fast 3d Gaussian Splatting with Sparse Pixels and Sparse Primitives*, https://arxiv.org/abs/2412.00578.

[12] Q. Hou, R. Rauwendaal, Z. Li, H. Le, F. Farhadzadeh, F. Porikli, A. Bourd, and A. Said, *Sort-Free Gaussian Splatting via Weighted Sum Rendering*, https://arxiv.org/abs/2410.18931.

[13] A. Clement David-Olawade, D. B. Olawade, L. Vanderbloemen, O. B. Rotifa, S. C. Fidelis, E. Egbon, A. O. Akpan, S. Adeleke, A. Ghose, and S. Boussios, AI-Driven Advances in Low-Dose Imaging and Enhancement—A Review, Diagnostics **15**, (2025).

[14] D. J. Brenner and E. J. Hall, Computed tomography—an increasing source of radiation exposure, New England Journal of Medicine **357**, 2277 (2007).

[15] S. Zhang, Z. Zhu, Z. Yu, H. Sun, Y. Sun, H. Huang, L. Xu, and J. Wan, Effectiveness of AI for Enhancing Computed Tomography Image Quality and Radiation Protection in Radiology: Systematic Review and Meta-Analysis, J Med Internet Res **27**, e66622 (2025).

[16] L. R. Koetzier et al., Generating Synthetic Data for Medical Imaging, Radiology **312**, e232471 (2024).

[17] F. Garcea, A. Serra, F. Lamberti, and L. Morra, Data augmentation for medical imaging: A systematic literature review, Computers in Biology and Medicine **152**, 106391 (2023).

[18] G. Ayana, K. Dese, A. M. Abagaro, K. C. Jeong, S.-D. Yoon, and S.-w. Choe, Multistage transfer learning for medical images, Artificial Intelligence Review **57**, 232 (2024).

[19] A. Yeung, The'As Low as Reasonably Achievable'(ALARA) principle: a brief historical overview and a bibliometric analysis of the most cited publications, Radioprotection (2019).
[20] *TASTI – Application-Tailored Synthetic Image Generation Project*, https://tasti-project.eu/.
[21] A. Biguri et al., *TIGRE V3: Efficient and Easy to Use Iterative Computed Tomographic Reconstruction Toolbox for Real Datasets*, https://arxiv.org/abs/2412.10129.
[22] D. Tafti and C. V. Maani, X-ray Production, Statpearls (2025).
[23] R. Novelline, *Squire's Fundamentals of Radiology*, 5th edn (Harvard University Press, 1997).
[24] A. Einstein, Über einen die Erzeugung und Verwandlung des Lichtes betreffenden heuristischen Gesichtspunkt, Annalen Der Physik **17**, 132 (1905).
[25] T. Hoang and A. Goel, *Compton Effect*, https://doi.org/10.53347/rid-30308.
[26] G. T. Herman, *Fundamentals of Computerized Tomography: Image Reconstruction from Projections*, 2nd edn (Springer, 2009).
[27] S. Hermena and M. Young, CT-scan Image Production Procedures, Statpearls (2025).
[28] A. C. Kak and M. Slaney, *Principles of Computerized Tomographic Imaging* (Society for Industrial, Applied Mathematics, Philadelphia, PA, 2001).
[29] J. M. Galvin, C. Sims, G. Dominiak, and J. S. Cooper, The use of digitally reconstructed radiographs for three-dimensional treatment planning and CT-simulation, International Journal of Radiation Oncology, Biology, Physics **31**, 935 (1995).
[30] D. B. Russakoff, J. M. Fitzpatrick, C. R. Maurer, R. J. Maciunas, and B. M. Dawant, Fast generation of digitally reconstructed radiographs using attenuation fields with application to 2D-3D image registration, IEEE Transactions on Medical Imaging **24**, 1441 (2005).
[31] J. Spoerk, A. Khamene, W. Birkfellner, D. Georg, and H. Bergmann, High-performance GPU-based rendering for real-time, rigid 2D/3D-image registration and motion prediction in radiation oncology, Zeitschrift Für Medizinische Physik **22**, 13 (2012).
[32] G. J. Tornai, G. Cserey, and I. Pappas, Fast DRR generation for 2D to 3D registration on GPUs, Medical Physics **39**, 4795 (2012).
[33] jstnmchl, *Xraysimulator*, https://github.com/jstnmchl/xraySimulator.
[34] M. Behr, *Xray_sim*, https://github.com/MichaelBehr/Xray/_Sim.
[35] Koushik Viswanathan, *Xraysim: Open-source X-ray Imaging Simulator*, https://xraysim.sourceforge.net/.
[36] F. P. Vidal, M. Garnier, N. Freud, J. M. Létang, and N. W. John, *Simulation of X-Ray Attenuation on the GPU*, in *Proceedings of Theory and Practice of Computer Graphics 2009* (Eurographics Association, Cardiff, UK, 2009), pp. 25–32.
[37] ufo-kit, *SYRIS: GPU Accelerated X-ray and Projection Simulation Tools*, https://github.com/ufo%E2%80%91kit/syris.
[38] Y. Cai, Y. Liang, J. Wang, A. Wang, Y. Zhang, X. Yang, Z. Zhou, and A. Yuille, *Radiative Gaussian Splatting for Efficient X-Ray Novel View Synthesis*, https://arxiv.org/abs/2403.04116.
[39] C. Talegaonkar, Y. Belhe, R. Ramamoorthi, and N. Antipa, Volumetrically Consistent 3D Gaussian Rasterization, Arxiv Preprint (2024).
[40] NVIDIA Corporation, *CUDA C++ Programming Guide, Version 12.4.0*, https://docs.nvidia.com/cuda/archive/12.4.0/cuda-c-programming-guide/index.html.
[41] J. D. Bakos, Chapter 2 – Multicore and data-level optimization: OpenMP and SIMD, Embedded Systems 49 (2016).

[42] S. W. Williams, A. Waterman, and D. A. Patterson, Roofline: An Insightful Visual Performance Model for Floating-Point Programs and Multicore Architectures, technical report No. UCB/EECS-2008–134, 2008.

[43] Z. Wang, A. Bovik, H. Sheikh, and E. Simoncelli, Image quality assessment: from error visibility to structural similarity, IEEE Transactions on Image Processing **13**, 600 (2004).

[44] M. Aleksandrov, S. Zlatanova, and D. J. Heslop, Voxelisation Algorithms and Data Structures: A Review, Sensors **21**, 8241 (2021).

[45] J. Amanatides and A. Woo, *A Fast Voxel Traversal Algorithm for Ray Tracing*, in *Eurographics 1987 Technical Papers* (Eurographics Association, 1987).

[46] R. L. Siddon, Fast calculation of the exact radiological path for a three-dimensional CT array, Medical Physics **12**, 252 (1985).

[47] G. Han, Z. Liang, and J. You, *A Fast Ray-Tracing Technique for TCT and ECT Studies*, in *1999 IEEE Nuclear Science Symposium. Conference Record. 1999 Nuclear Science Symposium and Medical Imaging Conference (Cat. No.99ch37019)*, Vol. 3 (1999), pp. 1515–1518vol.3.

[48] A. S. Glassner, *Principles of Digital Image Synthesis* (Morgan Kaufmann, 1989).

[49] J. Graetz, High performance volume ray casting: A branchless generalized Joseph projector, (2016).

[50] NVIDIA Corporation, *CUDA Programming Guide — Appendix C: Compute Capabilities (Table 30)*, https://docs.nvidia.com/cuda/cuda-programming-guide/05-appendices/compute-capabilities.html.

[51] W3cubDocs, *Floating Constant – C Language Reference*, https://docs.w3cub.com/c/language/floating_constant.html.

[52] NVIDIA Corporation, *Turing Tuning Guide: Tuning CUDA Applications for Turing*, https://docs.nvidia.com/cuda/turing-tuning-guide/index.html.

[53] M. Milakov, *GPU Pro Tip: Fast Dynamic Indexing of Private Arrays in CUDA*, https://developer.nvidia.com/blog/fast-dynamic-indexing-private-arrays-cuda/.

[54] H. Wong, M.-M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos, *Demystifying GPU Microarchitecture Through Microbenchmarking*, in *2010 IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS)* (2010), pp. 235–246.

[55] K. Museth, VDB: High-Resolution Sparse Volumes with Dynamic Topology, ACM Transactions on Graphics **32**, 1 (2013).

[56] K. Museth, *Nanovdb: A GPU-Friendly and Portable VDB Data Structure for Real-Time Rendering and Simulation*, in *ACM SIGGRAPH 2021 Talks* (2021).

[57] B. Kerbl, G. Kopanas, T. Leimkühler, and G. Drettakis, 3D Gaussian Splatting for Real-Time Radiance Field Rendering, ACM Transactions on Graphics **42**, (2023).