Imperial College London

MEng Individual Project

Imperial College London

Department of Computing

3D Gaussian Splatting on a Graph Processor

Author:
Nicholas Fry

Supervisor:
Prof. Andrew Davison

Second Marker:
Prof. Edward Johns

June 17, 2024

Abstract

3D Gaussian Splatting is a form of radiance field; a method used to reconstruct 3D models of a scene from monocular RGB images. This report presents a novel 3D Gaussian Splatting renderer developed for Graphcore's intelligence-processing-units (IPUs). It leverages the architecture's unique parallel processing model and efficient memory system to render complex scenes from arbitrary viewpoints. The implementation is based on the recent work by Kerbl et al. titled "3D Gaussian Splatting for Real-Time Radiance Field Rendering," and aims to address the limitations of GPU-based algorithms, particularly in the context of mobile robotics where power efficiency and real-time performance are critical.

This report outlines the challenges inherent in 3D rendering for robotic vision, emphasising the need for real-time, high-fidelity 3D scene reconstruction using lightweight sensors such as RGB cameras. Traditional GPU-based radiance field methods, while effective, suffer from memory bottlenecks and inefficiencies that hinder their deployment in low-power environments. In contrast, IPU-based rendering exploits the high-speed access and distributed nature of static random-access memory (SRAM), eliminating the need for dynamic RAM (DRAM) and its associated power consumption and latency issues.

Acknowledgements

I am incredibly grateful to Prof. Andrew Davidson for being such a wonderful supervisor. Thank you for welcoming me into your lab, and always making time for me.

Thank you Mark Pupilli for all the help, time and patience you have had for me throughout the project. It would not have been possible without you.

A special mention to Ignacio Alzugaray, Riku Murai, Marwan Taher, Hidenobu Matsuki, Xin Kong, Eric Dexheimer, Kirill Mazur, and all the other members of the Dyson Robotics Lab. You guys are awesome.

I extend the same gratitude to my friends and family who supported me along the way. I don't know where I'd be without you.

In loving memory of Daniel, I miss you man.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Problem Statement

Rendering in computer graphics is the process of generating the pixels of an image displayed on a computer screen. In this work, we concern ourselves with 3D rendering: converting a virtual 3D scene into a 2D projection. It is an important step in robot vision pipelines, as it can enable the training of the robot's internal representation of the world. This project explores an alternative to the current status quo in graphics compute. More specifically, the work aims exploit parallelism in vision pipelines, by using many processing cores with highly distributed static random-access memory (SRAM). The end result is a novel 3D Gaussian Splatting renderer [7] for Graphcore's intelligence-processing-units (IPU).

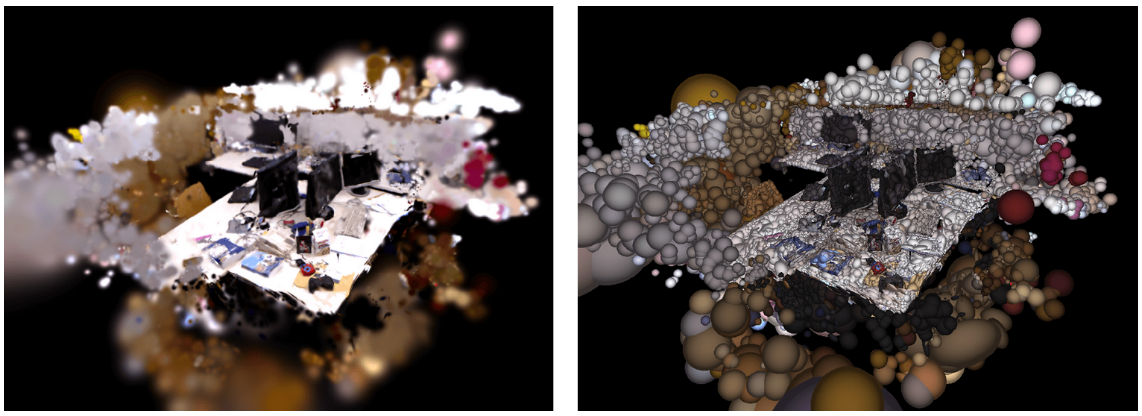


Figure 1.1: Novel view rendering and Gaussian visualizations on Real Scene
Taken from 3D Gaussian Splatting SLAM [1]

1.2 Motivation

Why can't robots do my laundry?

In order for robots to reliably perform everyday tasks, they must be able to perceive and navigate their environment. However, emulating abilities that humans have developed from over 6 million years of evolution [8] is a challenging task. The most important sense in the human body is sight, with over 50% of the human brain dedicated to visual processing [9]. If a robot were also able to build an accurate 3D representation of an environment in real-time, it could begin to plan and act in the same way humans do. Current state of the art (SOTA) methods that achieve this are too resource intensive for embedded deployment / mobile robotics. This motivates research into alternative compute strategies that may help overcome these bottlenecks.

Radiance Fields

Numerous methods exist to perform 3D reconstruction using sensors. One of the most ubiquitous and lightweight sensors is the RGB camera, a component present in nearly every smartphone. Recent advances in computer vision and graphics have enabled us to reconstruct scenes in photorealistic detail from just RGB images. One method of doing this is known as a *radiance field*.

A radiance field is a technique whereby a 3D scene is expressed as a function of position and direction [10]. Mathematically, a radiance field describes the amount of light that passes through a point in space, in a particular direction. Recent years have seen an explosion in methods targeting the real-time rendering of radiance fields [11, 7]. This technique is particularly interesting in the context of robotic vision. Building a 3D map of a scene on the fly would mark a tremendous feat of engineering, with myriad applications in various industrial sub-domains. One example of 3D reconstruction using a radiance field is shown in Figure 1.1 - work by Matsuki et al. [1]. The representation used to reconstruct this scene is known as 3D Gaussian Splatting (3DGS). Unfortunately, real-time radiance fields still have a number of limitations.

The goal of this project is to identify novel algorithms and alternative hardware that could improve real-time radiance field pipelines like the one in Figure 1.1. In this work we examine just one component of the pipeline: **rendering**.

GPUs, GPUs, GPUs?

The current paradigm for vision and graphics workloads is to use a graphics-processing-unit (GPU) for both training and rendering of a scene. NVIDIA's relentless rise in recent years has made NVIDIA Graphics Processing Units (GPU) and CUDA (the software stack that sits atop it) the only real choice for consumers [12].

Given time, the effective monopoly on AI/Graphics chips could result in inflated prices, stagnated architectural innovation and a potential dip in the quality of the product. The phenomenon can already be observed in the current market, and strongly motivates a paradigm shift towards alternative distributed architectures. In search for better access to compute, industry giants like Microsoft, Amazon and Google have already begun developing in-house AI solutions for data-center chips. However the same is not true of low-power embedded and portable processing. Exploration into other types of architecture is especially important considering Moore's Law may be at an end. In demonstrating viable alternatives to the GPU, we can promote a more diverse compute ecosystem, leading to better products and consumer targeted innovation.

1.3 Limitations of GPU

Only recent advances in computer vision have enabled high fidelity 3D reconstruction of shapes and textures in a real-time monocular context [1]. While faster and more accurate techniques exist, these most often rely on depth measurements [13, 14, 15], have significant memory bottlenecks, or do not run in real time [16]; factors which are prohibitive for deployment in low power/compute robotic systems. The same limitations exist in alternative mapping methods such as LiDAR and TSDF-fusion [17].

Despite significant progress in the field, monocular methods (without depth) still fall short of the performance and memory requirements necessary to be viable in mobile robotics. Current SOTA methods achieve a maximum framerates below 30fps on complex scene generation [18, 13]. One of the main bottlenecks is the rendering pipeline present in these systems. This is because high fidelity capture methods implement backpropagation to train the scene, the generated 3D model must therefore be differentiable - which GPU rendering pipelines were not originally designed for.

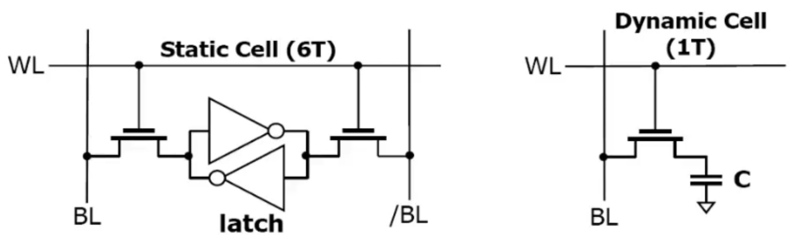


Figure 1.2: SRAM vs DRAM
(Taken from Imperial's Advanced Digital Systems course)

Dynamic random-access memory (DRAM) is a type of computer memory that uses a single capacitor to store electric charge representing a 1 or 0. Capacitance leaks over time and requires constant refreshing to maintain its charge. It is slow to access and is often used for main memory, or High Bandwidth Memory (HBM) in GPUs. In contrast, static random-access memory (SRAM) is extremely fast access and does not require frequent refreshing, making it much more energy efficient. However, SRAM takes up more space on-chip as the circuit uses 6 transistors to store a single bit. SRAM is typically reserved for smaller memories like cache.

One of the biggest program bottlenecks in modern GPUs is the access latency of its HBM/main-memory, combined with relatively small amounts of cache. If a program accesses HBM frequently then runtime becomes dominated by memory latency. This means developers have to carefully design algorithms to avoid using main memory, one famous example of this is FlashAttention [19].

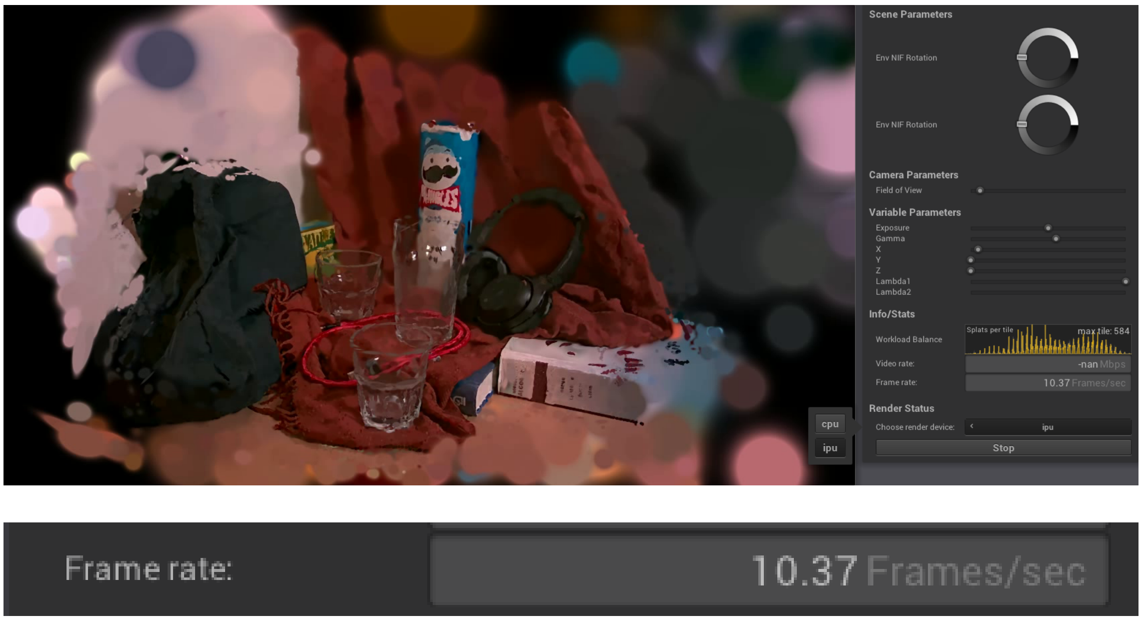
Instead of designing a radiance-field renderer to limit GPU main memory access, in this work we implement one on a processor that has no main memory. Graphcore's IPU is a distributed architecture like a GPU, but the programmer can only access large banks of SRAM, making it in theory more power efficient and faster in some scenarios. However, the chip does have some significant drawbacks which we attempt to overcome using various algorithms and programming techniques.

1.4 Contributions

This report outlines the implementation of a novel 3D Gaussian Splatting renderer for Graphcore's intelligence-processing-units. A realistic scene generated using this implementation is shown in Figure 1.3 running at 10fps. More scenes rendered on IPU are shown at the very end of this report - **see Figures 7.1, 7.2, 7.3.**

The following is a summary of the main contributions of this work:

- The first fully in-SRAM implementation of 3D Gaussian Splatting.
- A novel protocol for routing geometric data between SRAM banks of IPU.
- A full performance evaluation examining the tradeoffs between IPU and GPU.
- An investigation of optimisations and alternative methods which could be used for in-SRAM radiance field rendering.
- Render quality very close to that of the original implementation - see Figure 1.4



Video rate: -nan Mbps

Frame rate: 10.37 Frames/sec

Choose render device: ipu

Frame rate: 10.37 Frames/sec

Figure 1.3: Gaussian Splatting model rendered on IPU

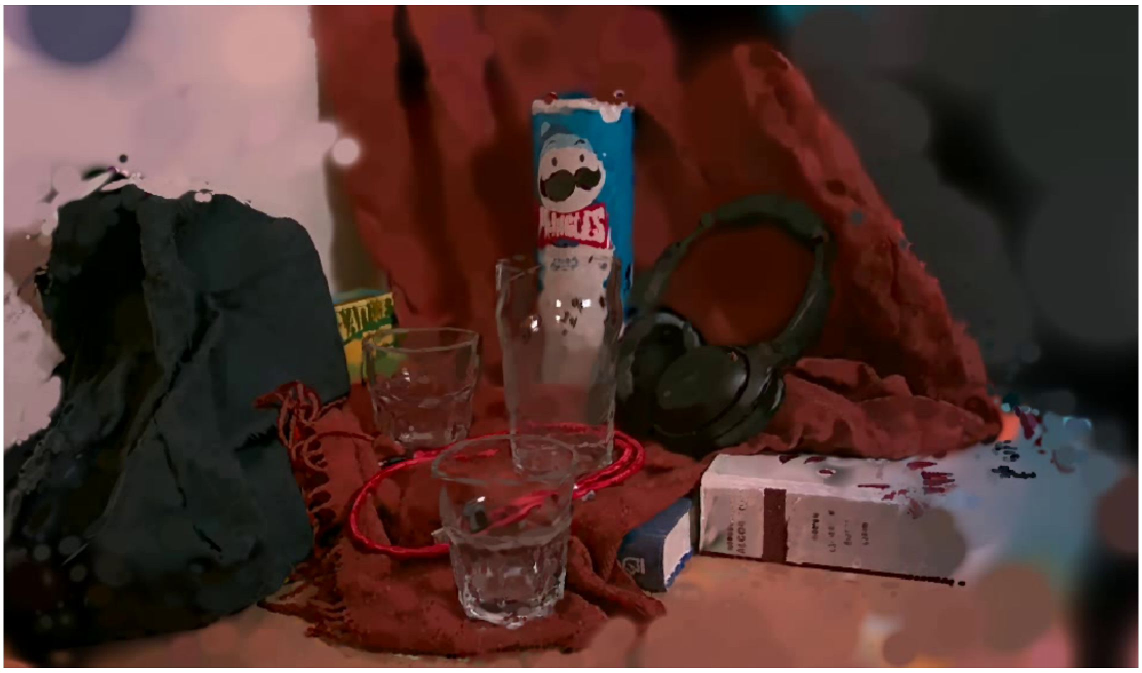


Figure 1.4: IPU 3D Gaussian Splatting

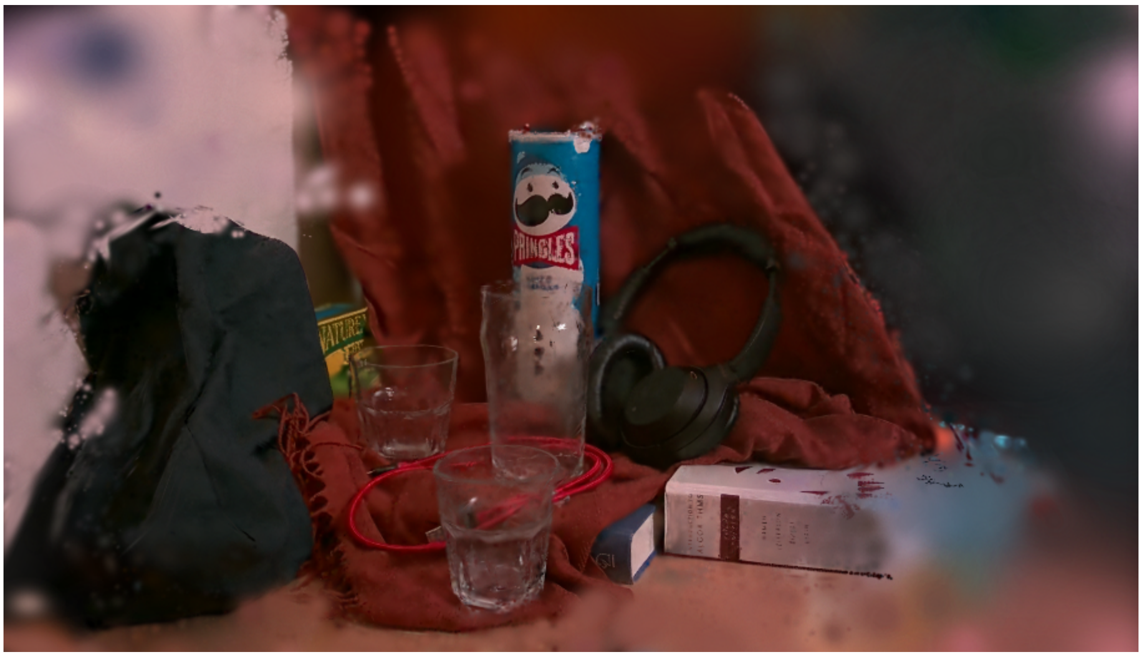


Figure 1.5: Original GPU 3D Gaussian Splatting

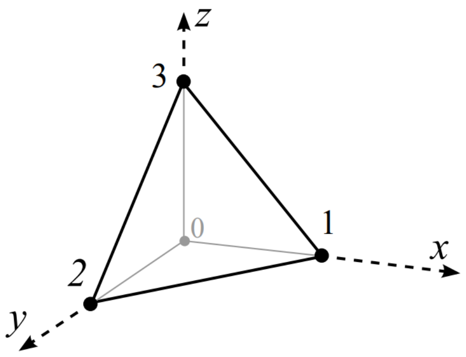
Chapter 2

Background

This chapter equips the reader with the background knowledge required to understand 3D Gaussian Splatting and IPU architecture. We begin by explaining 3D geometry, scene representation and projections. We then explore the process of rendering, and discuss graphics hardware and graphics processing on IPU.

2.1 Geometry & Scene Representation

There are several methods to represent a 3D environment in software. Geometry-based, or explicit scene representation is the expression of a virtual 3D scene using "objects". Classically these objects are made up of planar (flat) faces. Each face is defined by an ordered set of 3D point vertices, lying on one plane, which form a closed polygon [2], this is shown in Figure 2.1.



Vertex Data		Face Data
Index	Location	Vertices for face
0	$(0,0,0)$	0 1 3
1	$(1,0,0)$	0 2 1
2	$(0,1,0)$	0 3 2
3	$(0,0,1)$	1 2 3

Figure 2.1: "An example of location and topology data that can be given to describe a tetrahedron. Each vertex has a location and an index (starting from zero). Each face is defined by the indices of the vertices it uses [2]." - (Graphics Notes ICL)

Large collections of polygons can then used to represent complex geometry. In video games; landscapes, character models (such as in Figure 2.2) and other objects are typically created using 3D modeling software such as Blender.

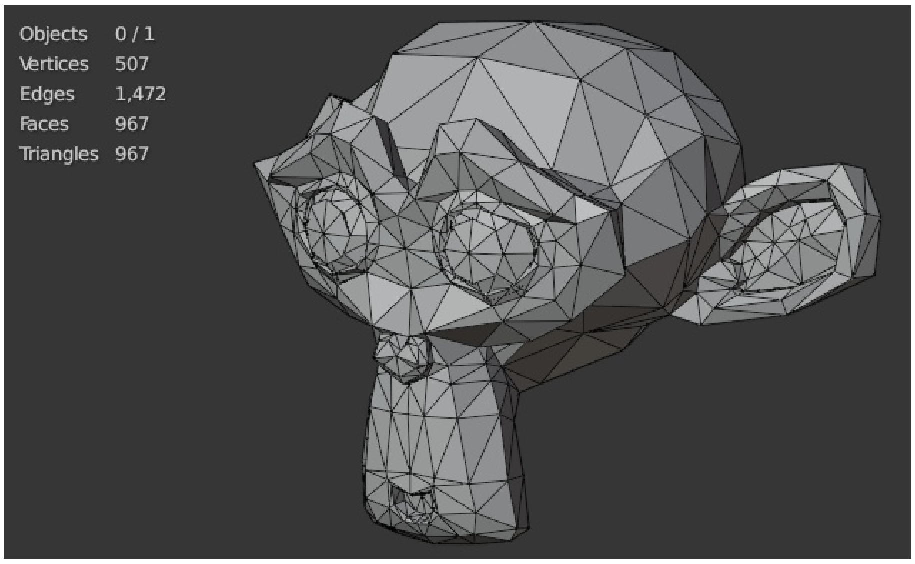


Figure 2.2: “Suzanne”, a famous model of a monkey in Blender

2.1.1 Coordinate Spaces

To simplify geometric manipulation, we define different coordinates systems relative to characteristics in our 3D world model. Terminology varies in the literature, however we will refer to the following spaces throughout this text:

- Object space (local space, or model space): The coordinate system of an object relative to its local origin (0,0,0). For example, given the tetrahedron shown in Fig 2.1. if its position were translated 10 in the x-axis, vertex 0 would lie at (10,0,0).
- World space: The coordinate system that defines where all objects in the virtual world live, relative to some origin. i.e. all objects transformed into one space.
- View space (sometimes eye space, or camera space): The coordinates of a model relative to the viewpoint in the scene. In other words, making the all objects centered around the camera.
- Clip space: 3D space containing vertex coordinates that will be shown on screen. Used to perform clipping operations on the objects in the scene. It is often defined by a cube, centered at the origin. Any object that falls outside this cube is clipped and not rendered.
- Screen space: This is the coordinate system that represents the final image that is displayed on the screen. It is a 2D coordinate system that maps the 3D objects in the scene into 2D pixels. The origin is typically the bottom left pixel.

2.1.2 Projection & Transformation

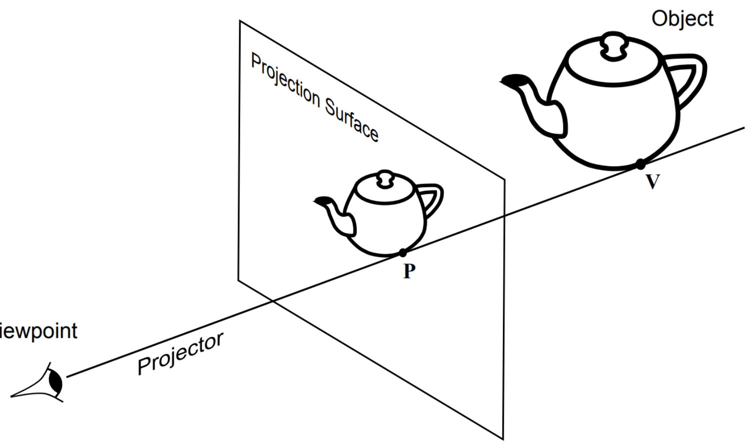


Figure 2.3: Projection of 3D geometry onto 2D viewing plane [2]

Since our target display device is 2D (a screen), we must define a transformation from 3D space to the 2D surface of the display device. This transformation is called a projection, and can be defined using matrix multiply operations. Ordinary cameras project 3D space onto a 2D image. To project an object onto a surface we first select a viewpoint and then define projectors or lines which link each vertex of the object to the viewpoint - this is illustrated in Figure 2.3 [2]. The 2D point on the surface where the line intersects defines the projection, and corresponds to the original vertex in the object.

2.2 Rendering

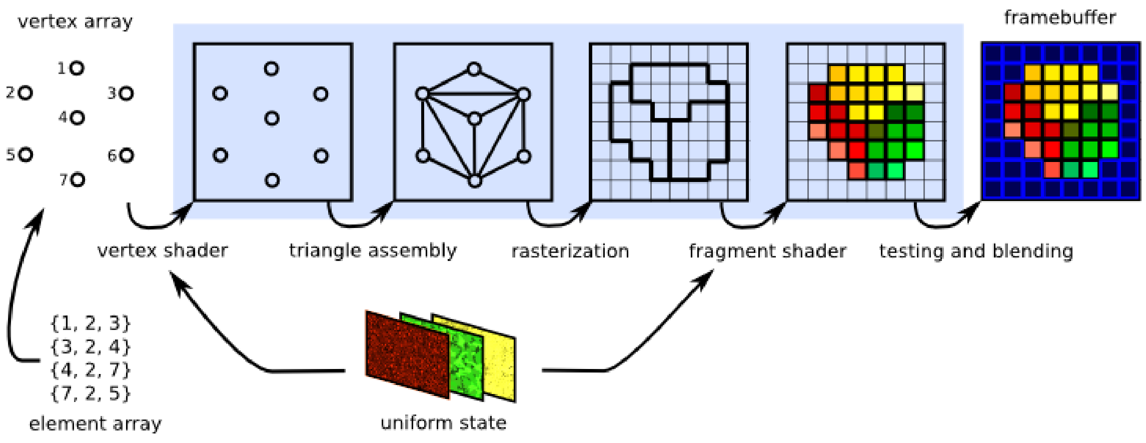


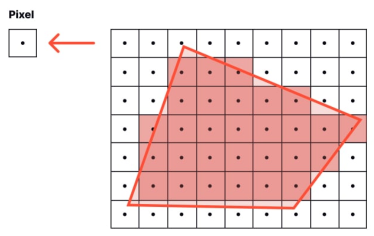
Figure 2.4: A typical rasterisation pipeline [3]

Figure 2.4. shows a pipeline that a graphics API such as OpenGL might use, and would be implemented in hardware on a graphics processing unit (GPU). As discussed, the first three steps involve assembling and projecting the geometry of the scene. Projection is typically defined in a programmable piece of code called a *vertex shader*, where view-dependant lighting may also be applied and stored for later use. In this section we will focus on the next three steps of the pipeline; converting the explicit scene representation we have built into something we can see. i.e. how can we capture a specific 2D view of the 3D scene, such as the picture of Suzanne in Fig 2.2. Note that in this work we implement a similar pipeline but in **software**.

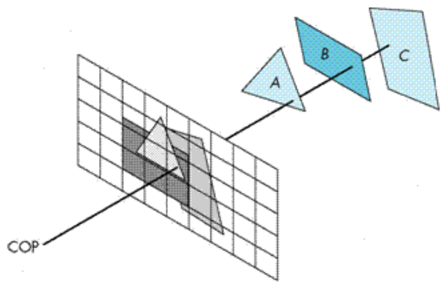
2.2.1 Rasterisation

After projection, the vertices of our polygons lie in screen space. The next goal is to find all the pixels that are inside the primitive (polygon) being rendered, so we can shade those pixels with its colour. To avoid checking every pixel in screen space, we first compute the bounding box of the polygon, and search within this [20]. We only colour each pixel who's center is inside the polygon, this is shown in Figure 2.5a. For each pixel, we also store the depth of the primitive.

We repeat this process for each polygon, unless a pixel has already been coloured; in which case we only recolour a pixel if the polygon is closer to the screen than the current depth of that pixel. In other words, we only update the pixel colour with the closest polygon that covers its center. This algorithm is known as the Z-buffer, or depth-buffer algorithm, depicted in Figure 2.5. Polygons may also store an α-value which represents the transparency of an object. This allows us to proportionally "α-blend" the colour of overlapping polygons into a pixel (e.g. for stain glass windows). The pixel data that is output from this process is known as a fragment, to which other effects like blur can be applied in a piece of code called a *fragment shader*.



(a) Pixel raster [21] (aka. Framebuffer)



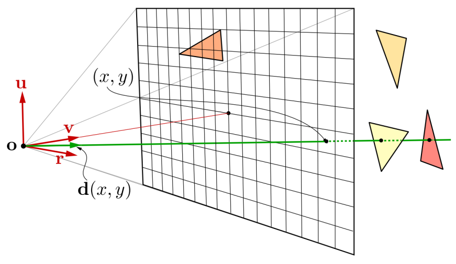
(b) Z-buffering [22] (axis denotes polygon depth)

Figure 2.5: Pixel painting via rasterisation

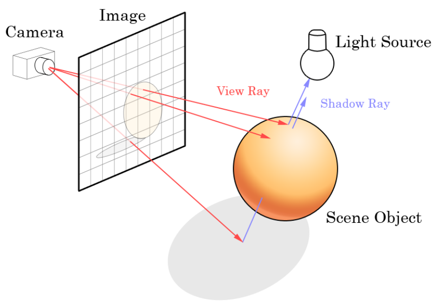
2.2.2 Ray Tracing

Our work focuses on rasterisation, but it is important to understand alternative rendering methods in order to compare them. A ray is defined by an origin o and a direction d. Ray tracing (see Fig 2.6) consists of constructing and casting one (or more) rays from the viewpoint through each pixel of the viewplane (framebuffer), and calculating whether the ray intersects the scene geometry. The pixel's colour is given by the weighted sum of the contributions from each ray intersect. Note; "in Figure 2.6b that the vectors r (right), u (up), and v (view) are used to construct a direction vector d(x, y) of a sample position (x, y) " - (Real-Time Rendering, Fourth Edition [20]). We can manipulate the casting of rays a number of different ways to implement phenomena like refraction, matte surfaces or soft shadows [23].

We will refer to ray *casting* as simply moving the end point p of a ray by some distance in a direction d from its origin o (i.e. $p = o + distance * d$). We term ray *tracing* as the process of rendering by calculating intersections of cast rays and proportionally summing the object surface contributions. We denote ray *marching* as casting a ray in small measured steps until the point p is within a distance threshold of an object - this is typically slower but avoids performing intersection calculations and is useful for some radiance field representations.



(a) The ray shown in this figure hits two triangles, but if the triangles are opaque, only the first hit is of interest [20]. Origin o is the viewpoint.



(b) If a view ray intersects with the scene, we cast "shadow rays", if they do not reach the light source, the point is obscured [24].

Figure 2.6: Ray Tracing - a physically based rendering (PBR) approach

2.2.3 Rasterisation vs. Ray Tracing?

In summary, rasterisation checks each object and paints the closest pixels in the framebuffer, ray tracing checks each pixel for the closest object. The two algorithms are complementary and can exist together in the same render. They do, however, have different memory and performance metrics depending on the hardware and structure of the code in which they are implemented. Generally speaking, ray-tracing helps us simulate light, resulting in more realistic images, while rasterisation is often orders of magnitude (OOM) faster.

2.2.4 Real-Time Rendering

"Offline" rendering refers to rendering high-quality images and movies that require a lot of detail and realism. Each frame may take hours or days to complete. In contrast, real-time rendering is concerned with creating interactive experiences, like in video games, web viewers, VR headsets and more. In this report we focus real-time rendering: generating high quality images on the fly without sacrificing performance.

2.3 Inverse Rendering

We can think of rendering as a function f of a scene $\mathbf{X}$, where y is the image produced: $y = f(\mathbf{X})$. In many cases, such as in computer vision, finding the inverse of this function $\mathbf{X} = f^{-1}(y)$ is much more interesting [25]. Here, from just an image y (i.e. a set of pixels) we wish to generate geometry, positions and textures of a scene. This is a such a difficult task that it is often not directly solvable. A classic method for generating a 3D scene from 2D images is to invert the 2D projection and fuse the resulting 3D geometry using weighted averages [26, 27]. However, these methods often result in overly smooth geometry and cannot capture photo-realistic details.

2.3.1 Differentiable Rendering

Instead of trying to find a direct solution, we can approximate photo-realism. Differentiable rendering is a novel field, where we can differentiate f and combine this with a gradient based optimiser to approximate the inverse iteratively. Gradient descent [28] is performed by taking steps in the $\frac{\partial}{\partial \mathbf{X}} f(\mathbf{X})$ direction to find the geometry that renders with minimal loss $\mathcal{L}(y, y')$, where y' is the input image. Classical rendering hardware was not developed with differentiable rendering in mind, so the scene representation must instead be tailored to the architecture. This makes the construction of differentiable representations complex and diverse in nature. This work focuses on the rendering of a type of differentiable scene known as a *radiance field*. Section 2.4 compares a few different types of these models, with a particular focus on the way in which they are rendered.

2.3.2 Novel-View Synthesis

The human brain extrapolates spatial features of our environment, helping us navigate and interpret distance, and there are many instances where we wish to emulate this. For instance, if a user wants to view a VR world from a different angle, or for a robot to make a decision based on a viewing position it has not expressly captured. Novel-view synthesis is the process of generating arbitrary new views of the world from a bounded set of input images and is a highly desirable feature of radiance fields.

2.4 Radiance Fields

Until now we have referred to the *colour* of geometric primitives, however this can be formalised as the energy leaving an area in 3D space. *Radiance* (specifically exitant radiance) L_r is defined as the amount of radiant flux per unit area, per solid angle [2]:

$$L_r = d^2\Phi / dA d\omega \quad [W/m^2 sr] \tag{2.1}$$

Radiance fields are a way of programmatically capturing radiance in a 3D scene, so we can later query it when rendering/pixel painting. We can write this as a function L : $R^5 \rightarrow R^+$ which takes a position (x, y, z) and a direction in spherical coordinates (θ, ϕ) and returns a non-negative radiance value [4, 29]. Radiance fields allow us to generate novel views of a scene with no additional reconstruction cost, while also providing high quality photorealistic rendering. They are an ever more important area of research, with new techniques constantly being developed that present various strengths and weaknesses.

2.4.1 Eulerian Representations

Implicit or Eulerian radiance fields are so named because they do not explicitly define the geometry in world space. Instead, the radiance at a given point is encoded in a continuous function of its coordinates and viewing direction. Typically this function is defined by a neural network [30, 31, 32], which is queried at render-time to blend the radiance at a point into a pixel of the framebuffer. This method was popularised by Neural Radiance Fields (NeRF) [11, 33] which achieve high quality image synthesis and benefit from a very compact volumetric representation [34]. Crutially, a Multi-Layer Perceptron (MLP) representation is differentiable, allowing us to train the scene in the manner described above. Variations of NeRF have been used effectively in several real-time vision pipelines [35, 13, 14], however are disadvantaged in a number of ways. In order to render an image, NeRF uses ray-marching to query the MLP at coordinates along a ray path. Not only is ray-marching an expensive approach to rendering, the MLP also imposes a “black-box" representation, making the scene very difficult to edit and relight [36, 37].

2.4.2 Lagrangian Representations

Explicit or Lagrangian methods follow a more traditional geometric representation, where radiance is captured in a discrete structure such as a point cloud, plenoxels or a surface function [38, 39, 40, 41]. A general form can be expressed as the following equation, taken from “A Survey on 3D Gaussian Splatting” [4]:

$$L_{explicit}(x, y, z, \theta, \phi) = \text{DataStructure}[(x, y, z)] \cdot g(\theta, \phi) \tag{2.2}$$

where querying DataStructure with coordinates (x, y, z) returns a radiance value for that primitive that is used for pixel painting as discussed in section 2.2.1. By **equation 2.1** radiance is dependant on viewing direction, so we must also scale the queried value by some function $g(\theta, \phi)$. A scene is typically sparse (mostly empty space); so having an explicit representation allows us to perform rasterisation, which keeps our render time in the order of the number primitives, i.e. cheaper than ray-marching [7, 20]. Unfortunately, faster render times and an editable representation come at the cost of higher memory usage. They also may struggle to capture details at a very fine granularity, which can be easily encoded in an MLP.

2.4.3 3D Gaussian Splatting

While traditional graphics pipelines are optimised for large triangle rasterisation, many other efficient primitives exist. Point clouds can be used as a convenient representation for sparse or unstructured geometry. However, point renderers must expand points into a unit square made of two pixel-length triangles [42] often resulting in holes and aliasing, while also forgoing the batch parallelism present when rendering larger polygons in hardware [43]. Many alternatives have been proposed such as discs, surfels or ellipsoids [44, 45, 46].

The focus of this report is a new technique called 3D Gaussian Splatting [7], a major breakthrough in real-time differentiable rendering. This work by GraphDeco, Inria, aims to combine the best of both explicit and implicit radiance fields. The method works by representing a scene as a cloud of 3D gaussian probability distributions (ellipsoids), each with its own colour and α-value. To render the scene, the 3D gaussians are projected into screen space - a process called "splatting" [45, 46], after which we perform alpha blending to colour the pixels that fall within the projected ellipse. This is shown in Figure 2.7 .

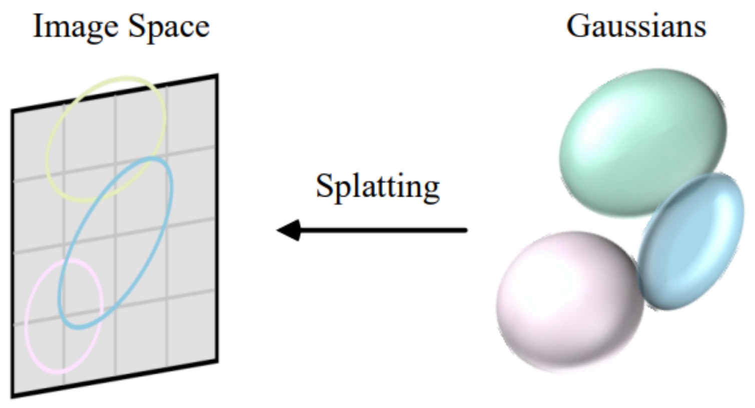


Figure 2.7: Projection of 3D Gaussians
(Taken from "A Survey on 3D Gaussian Splatting" [4])

The representation is highly flexible and tailors well to traditional GPU pipelines, but most importantly allows efficient scene optimisation via backpropagation. In order to generate a scene from a set of images, the algorithm only requires a point cloud input, generated using Structure-from-Motion (SfM) [47]. The gaussians are centered at each point and initialised with a colour, α and 3D covariance matrix. The training loop consists of rendering an image y (via splatting), computing the loss w.r.t. to the ground truth image y', and adjusting the gaussian parameters in the direction of minimal loss $\mathcal{L}(y, y')$ [7, 4]. This is much like an MLP, where network weights are updated based on the training loss via backpropagation. In order to address under or over filled areas of geometry, the authors also include adaptive gaussian density control during this loop.

3D Gaussian Splatting (3DGS) is a huge step forward for real-time rendering algorithms. It allows us to exploit the natural sparsity of a scene in a way that ray based methods do not. Dedicated hardware for real-time ray tracing has not had as much time to mature and is thus not as ubiquitous or fine-tuned. By having a differentiable and *rasterisable* representation, we can make far better use of existing parallel hardware. Furthermore, with the explosion of deep learning and neural rendering approaches [48], having dedicated hardware to compute ray intersections may be less desirable.

2.5 Massively Parallel Architectures

The history of parallel compute is rich and complex. To oversimplify; the development of massively parallel hardware was largely driven by consumerism in the video game and VFX industries [12, 49]. Hardware that processes many elements at once is necessary for real-time in "embarrassingly parallel" problems such as pixel computations. Here we explore different types of parallel architecture and the performance tradeoffs of each.

2.5.1 GPU - Graphics Processing Unit

Graphics hardware evolved from a fixed-function pipeline (implementing Figure 2.4) that was not easily programmable [12]. Today, there is huge interest in general purpose GPU (GPGPU) compute, the most notable use case being the training of neural networks (NN). Graphics workloads themselves are increasingly neural [5], or, like 3DGS, exhibit similar characteristics to that of a neural network.

While vertex and fragment shaders are highly optimised, they are bound to a specific stage in the graphics pipeline. General parallel compute can instead be achieved using a GPGPU API. *Compute shaders*, OpenCL, and CUDA all expose massively parallel hardware to the programmer [50] and are used extensively in a machine learning (ML) context. In fact, the original implementation of 3DGS was written in CUDA, and does not use the graphics pipeline. To determine what makes a GPU so good for hybrid ML-graphics workloads, we must first examine its architecture.

SIMD Processing & Scheduling

GPUs use an execution model known as "single instruction, multiple thread" (SIMT), a variation of "single instruction multiple data" (SIMD). Within a host (CPU) program we can dispatch *kernels* to the GPU. A kernel defines a set of blocks of vector instructions (known as warps or wavefronts). Each element of the vector lives in a lane (or thread) in the wider context of the program. This is shown in Figure 2.8. Instructions are issued from each lane by a scheduler to one of a number of functional units called CUDA cores, aka streaming processors (SP) which are executed in lockstep [20] - see Figure 2.9. Note that the number of threads in a block may be larger than the number of cores, it is the scheduler that will assign work to each SP (or a different unit e.g. load-store unit)

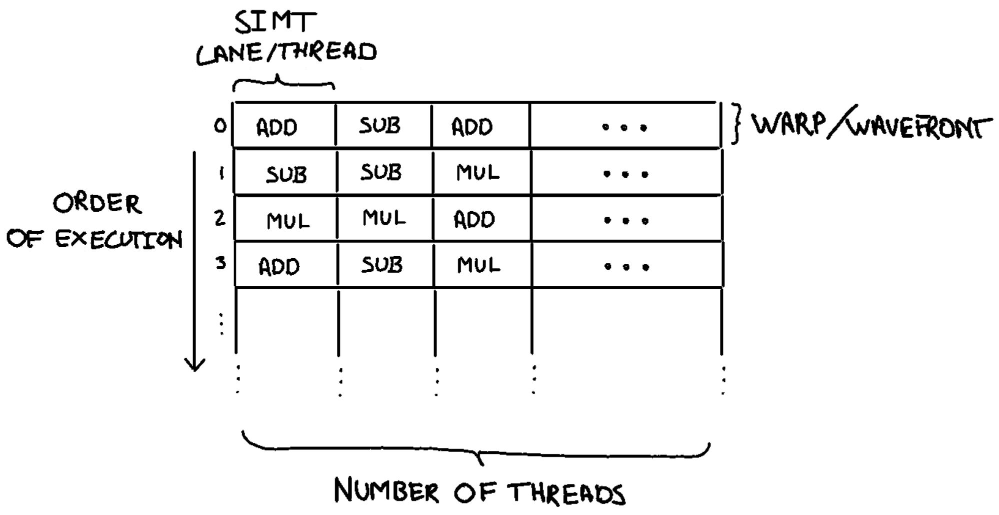


Figure 2.8: A "threadblock", each warp typically formed of 32 threads

Each streaming multiprocessor (SM) in Figure 2.9 shares a global L2 cache and main memory. They maintain a small internal L1 cache (SRAM). The nature of this model is that the memory latency of the whole warp is bound by the access time of the slowest thread. Most notably, real-time performance in many neural

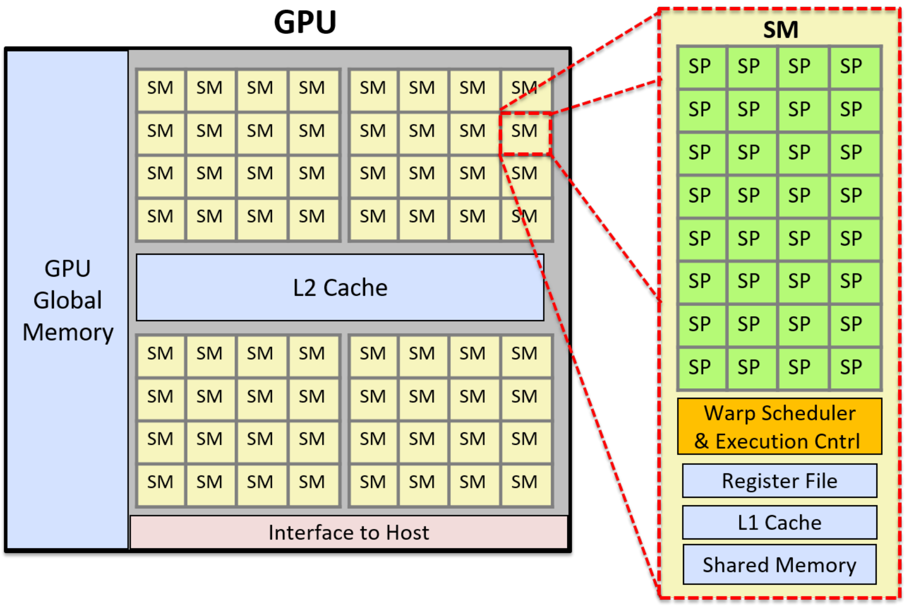


Figure 2.9: Simplified GPU architecture (taken from diveintosystems)

workloads is dependant on keeping data in L1 cache for as long as possible [51, 52, 19]. As discussed in the introduction, this must be done using a combination of clever programming, compilation and algorithm design.

2.5.2 IPU - Intelligence Processing Unit

Graphcore is a British semiconductor company that designs accelerators targeted at AI and machine learning workloads. Their intelligence-processing-unit (IPU) was in no way developed for rendering, but exhibits unique characteristics which make it a fascinating avenue for exploration. It is a massively parallel architecture, where each processor core (known as a *tile*, similar to SM) is tightly coupled to a huge bank of static memory (SRAM). Each tile holds six hardware threads which are "barrel-scheduled", i.e. have interleaved pipelines, which avoids warp synchronisation/thread masking. This paradigm is known as "multiple instruction, multiple data" (MIMD) and has some interesting tradeoffs compared to SIMD. Furthermore, IPU does not use a shared main memory. It only holds a special type of very high bandwidth dynamic random-acces memory chip (DRAM) known as streaming memory (*double-data* rate (DDR)) which is used to transfer data between tiles and the host.

2.5.3 GPU vs. IPU?

On a GPU, individual cores can address any part of global memory, and can thus share memory easily with other cores. This flexibility is extremely useful, but comes at the cost of high access times and synchronisation if the data is not in L1 cache. In contrast, an IPU tile can only access data in its own SRAM, but with very little latency. If it needs something from a different tile or external memory, that data must be copied to tile SRAM in a process called *exchange* [53]. Exchange is defined at compile time by the programmer, and therefore requires a completely different approach to algorithm design. IPUs are less flexible in this regard, but can hold

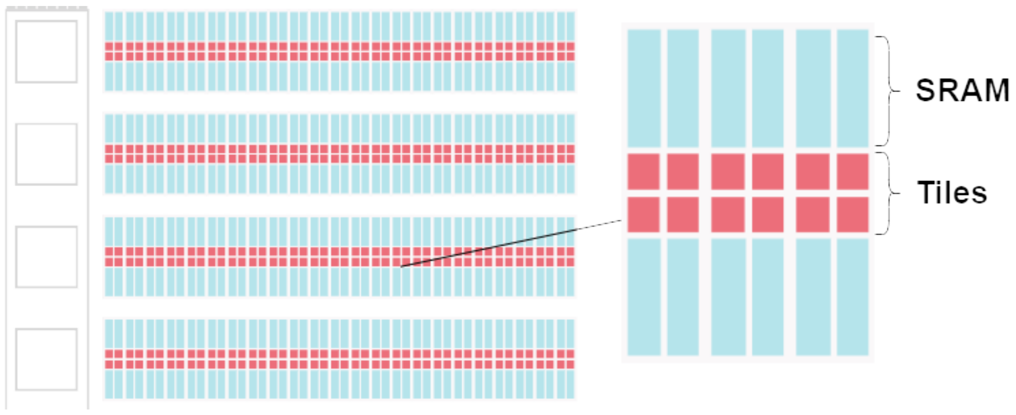


Figure 2.10: Simplified IPU architecture (adapted from Graphcore's website)

far more data on-chip than other parallel architectures. This feature is of particular interest in 3D Gaussian Splatting and neural rendering where training times are dominated by memory access latency [6]. The key architectural differences are condensed in Figure 2.11.

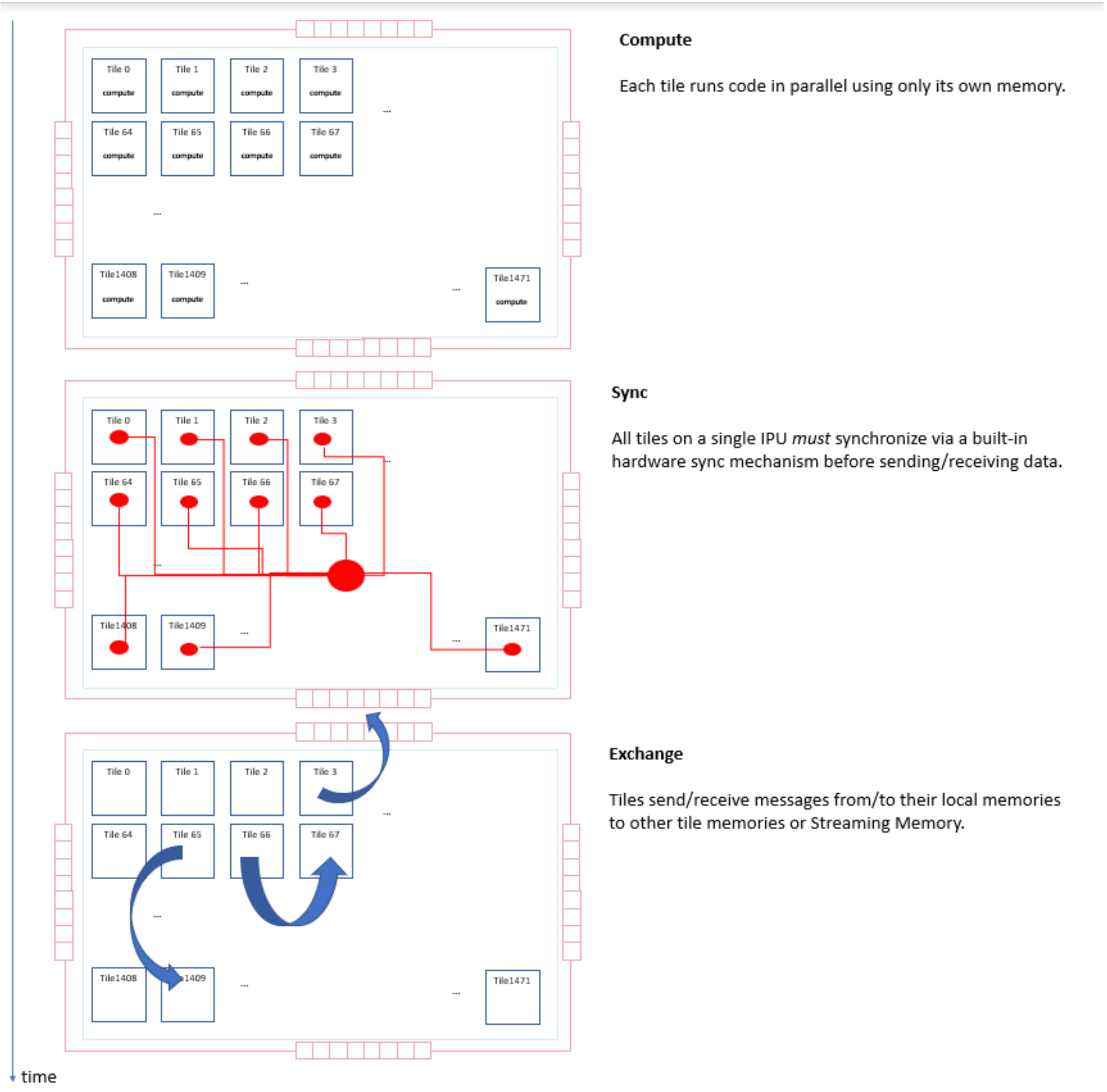
Property	IPU Bow GC200	GPU A100
Independent instruction streams (IIS)	1472 (tiles) x 6 (threads) = 8832	108 (SMs) x 64 (warps) = 6912
SRAM per core	624 KiB per tile	192 KiB (up to 164 KiB shared) per SM
Contention free SRAM/registers per IIS	624/6 = **104 KiB**	(192 + 256)/64 = **7 KiB**
SIMD vector width per IIS	2 float	32 float
	4 half	64 half
Integer compute	Non-vectorized but dual issue	Vectorized but no dual issue
Peak TFLOP/sec (half, non-zero)	349	312

Figure 2.11: "Comparison of IPU and GPU (A100) [5] with similar silicon area from the same process node (7nm)" - taken from "Towards Neural Path Tracing in SRAM" [6]

A number of differentiable rendering systems use *megakernels* (aka. *fully-fused* kernels) [25, 34], to avoid any access to main memory. This approach can be disadvantageous if data coherence is not carefully considered [54]. Programs must be written or compiled with a mind to the cache architecture of a particular system. Chip makers are addressing this issue with hardware extensions such as NVIDIA's shader execution reordering (SER) or Intel's thread sorting unit (TSU) [55, 56]. As the computing landscape becomes increasingly neural and programs ever larger and more complex; perhaps we should instead consider a shift towards architectures with larger amounts of on-chip SRAM. In recent work, Mubarik et al [57] design a "neural graphics processing cluster" (NGPC) with a possible performance improvement of 58.36× over traditional GPUs for radiance field rendering. The design features distributed banks of on-chip SRAM, much like an IPU.

IPU operates a different execution model compared to other parallel architectures. When a program runs on an IPU, the cores alternate between data exchange and executing operations on their local data. More specifically, the chip employs a "bulk-synchronous parallel" (BSP) execution model where all tiles perform the operations shown in Figure 2.12 [53]. During the compute phase, all tiles operate in parallel.

Once each tile completes its computation, it initiates a synchronization process with the other tiles. When all tiles have synchronized, the IPU enters the exchange phase, where data is transferred between tiles. The system can be thought of like many small CPUs working together on one silicon wafer.


Compute

Each tile runs code in parallel using only its own memory.

Sync

All tiles on a single IPU *must* synchronize via a built-in hardware sync mechanism before sending/receiving data.

Exchange

Tiles send/receive messages from/to their local memories to other tile memories or Streaming Memory.

Figure 2.12: Compute, Exchange and Global Sync Phases of IPU (Taken from GC Developer Website)

2.5.4 Gaussian Splatting on IPU

In response to the growing trend in megakernels, we explore whether a SOTA method for radiance field rendering is possible to implement on an *existing* chip with lots of distributed SRAM. While new radiance fields are constantly being published [58], 3D Gaussian Splatting remains one of the most versatile and efficient solutions. In this work we attempt to leverage the matrix and convolutional units of the IPU, combined with its large SRAM banks to build a gaussian renderer.

Chapter 3

Related Work

In this short chapter we outline the existing work that motivated and shaped our design for an IPU splat renderer.

3.1 Rendering on IPU

While IPUs are designed for general purpose compute, there some existing research published by Graphcore that explores rendering on IPU. The only published paper being "Towards Neural Path Tracing in SRAM" [6], which presents many key insights that heavily influenced the direction of this project. The paper demonstrates that IPU is capable of rendering high quality ray traced scenes such as those in Figure 3.1. It also shows IPUs act like many small CPU cores and are capable of efficiently running algorithms designed for distributed computing [59, 60].

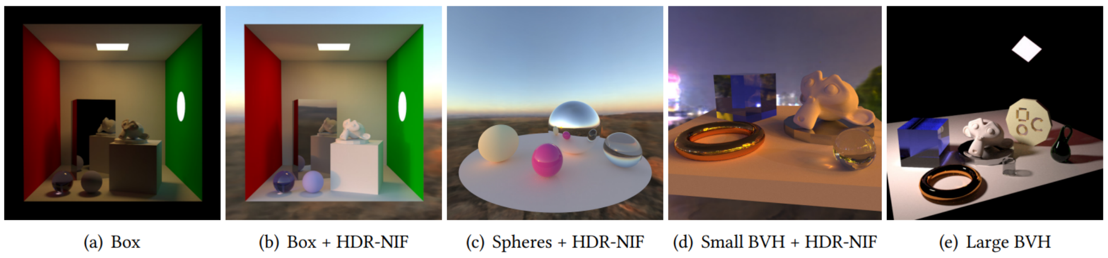


Figure 3.1: Scenes path traced on GC Bow-Pod-16. "The HDR environment light is compressed into 97 KiB of neural network weights. The neural-network weights, activations, and scene BVH reside entirely in on-chip SRAM." - Image taken from "Towards Neural Path Tracing in SRAM"

The paper identifies some key points: **1.** IPU MIMD threads do not suffer from data incoherence and so avoid the need for program reordering **2.** The ray tracer is limited by the size of the bounding volume heirarchy (BVH) and background (encoded as an MLP) - they must be sufficiently small to fit on one tile.

The nature of 3DGS is such that we do not need a BVH, this somewhat mitigates the constraint identified in **2**. Gaussians are also very lightweight and should therefore easily fit in tile memory, provided they are not all on one tile. Niedermayr et al's [61] work targeting compression of 3DGS for lightweight GPUs may also help significantly, allowing us to fit complex scenes fully on-chip.

3.2 Gaussian Splatting Hardware

As 3DGS is so new, there is little published work on implementing it on alternative hardware, especially since it was designed for GPU architectures. The original renderer (written in CUDA) is still being improved [62], and we may see significant speed-up just at the software level. Most of the focus on radiance field hardware has been directed towards neural radiance fields devices [63, 64]. At the time of writing, this is the first work on porting 3DGS to an alternative architecture. While 3D Gaussian Splatting uses an explicit representation, projecting and training a vector of gaussians can be thought of like a large single layer MLP. The same matrix-vector operations are involved in the projection and transformation of primitives. It effectively converts a pure rendering problem into an AI-like workload, something IPU has been proven to do extremely effectively.

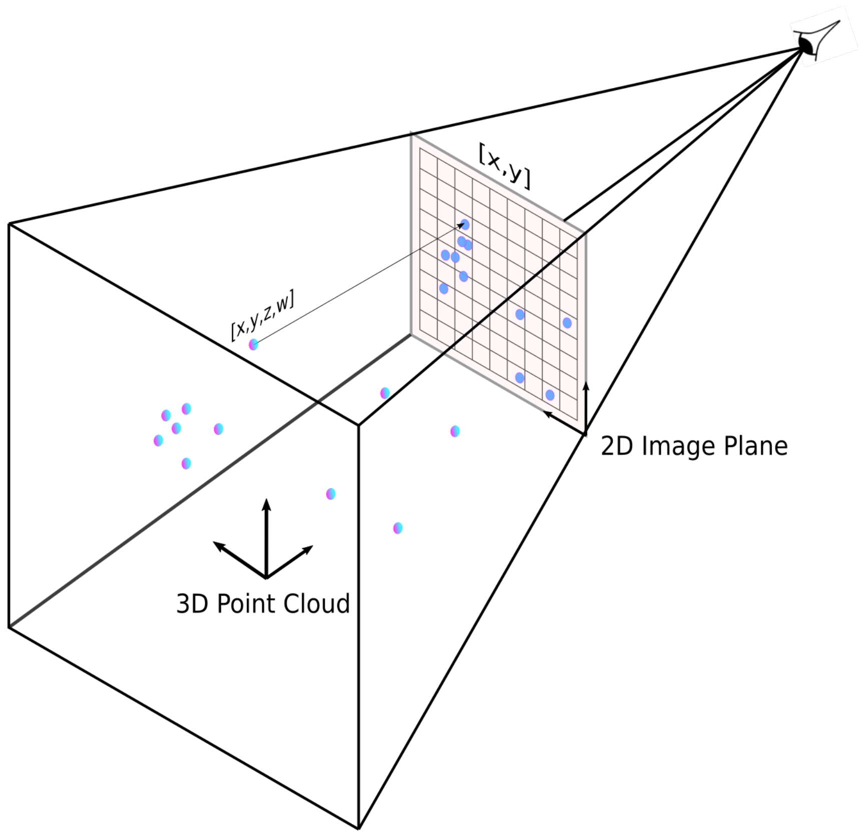


Figure 3.2: Point "splatting"

Chapter 4

Splatting in SRAM

This chapter outlines the implementation of IPU-3DGS. Section 4.1 presents some initial testing on IPU for real-time image processing. The subsequent sections detail the structure of the 3D Gaussian renderer, from the camera and framebuffer setup, to the 3D gaussian representation on IPU. Section 4.6 goes on to discuss volume splatting and α-compositing to generate the final rendered images.

4.1 On-Tile Pixel Computations

To test the feasibility of real-time pixel processing on IPU, we performed simple experiments using a Kinect 4 camera. The camera reads 30 frames per second (fps), and sends each frame via a TCP connection to a server running on a host CPU. The CPU is connected via PCIE to a MK2 IPU.

On the host, the captured image is divided into equal chunks that are pushed to each of the IPU tiles. We frequently refer to chunks of pixels as *slices* of the framebuffer. After an initial exchange, small hand-written code sections called *codelets* are then executed in parallel on the tiles of the IPU. We use the terms *vertices*, *kernels* and codelets interchangeably to describe these code sections. In this case, the codelets simply iterate over all the pixels in the local slice and apply a filter to their RGB values. This pipeline is shown in Figure 4.1.

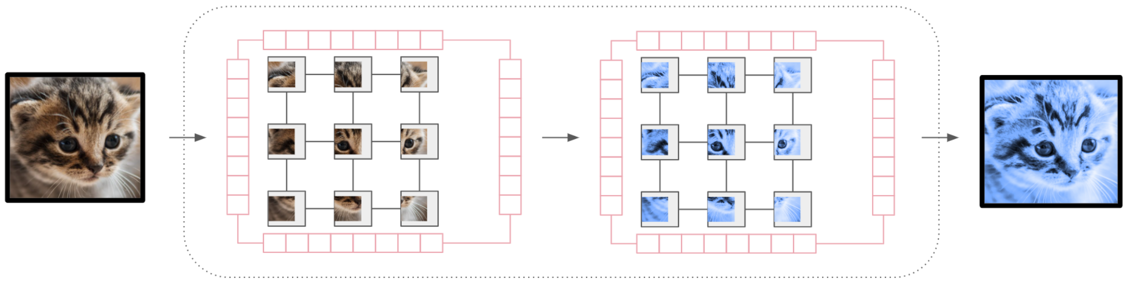


Figure 4.1: Distributed pixel processing on IPU

The single-threaded and unoptimised implementation of this executes at a lowly ∼3fps, which is due to inefficient reads and writes to the IPU. However, it shows promise as almost all the operations in this pipeline are parallelisable. We use a 1280x720 image and divide it into chunks of 32x20. This experiment was useful to develop the system of mapping between image chunks and IPU tiles, which is discussed in the next section. Using a real-time camera setup like this would only be useful in the context of optimising 3DGS, which is out of scope for this project. We therefore abandon the real camera for the rest of the implementation and focus solely on the forward pass presented in the 3D Gaussian Splatting paper.

4.2 Tiling the Framebuffer

The execution model of an IPU is radically different to most other architectures, and thus writing a program is also very different. A program is constructed using a language/execution model called Poplar (loosely equivalent to CUDA for an NVIDIA GPU). A Poplar program is defined by a computational graph which is mapped by the Poplar compiler to the tiles of the IPU. The graph is made of Variables (data) and Vertices (compute); this is shown in Figure 4.2. Variables are of type “Streamable Tensor”, and can be read/written to by the CPU during exchange or at compile-time. Vertices are defined by the Poplar codelets file, where we perform operations on our data.

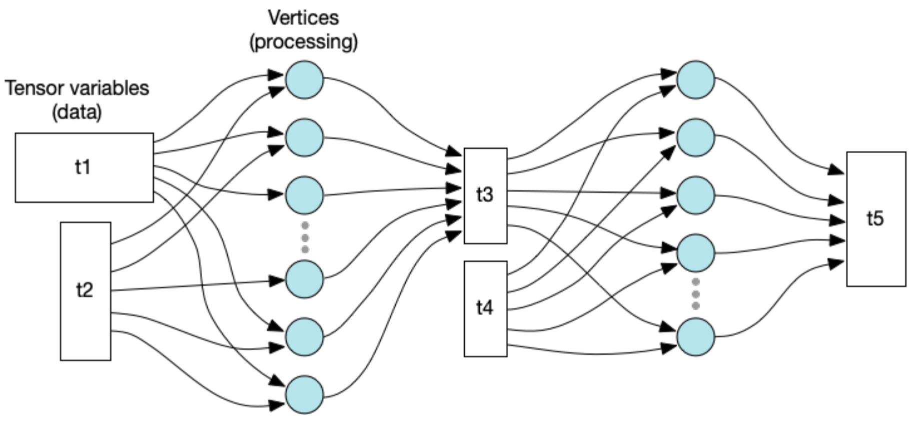
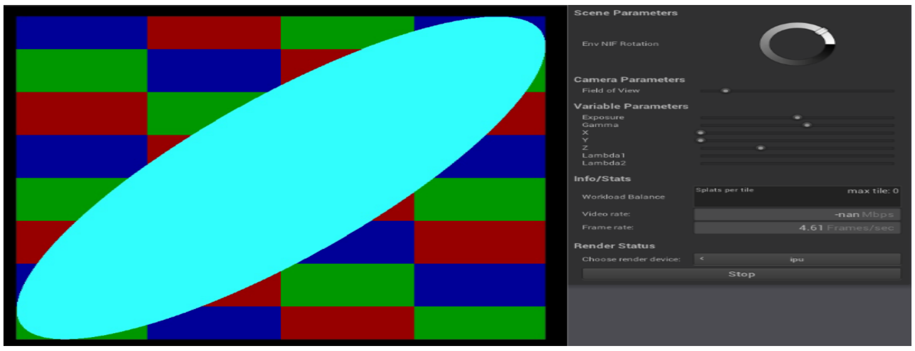


Figure 4.2: Computational Graph in Poplar (See Graphcore Developer Docs)

In a machine learning setting, we can imagine the Tensors to contain network weights/biases and data points, while the vertices might be weighted sums (e.g. matmul). In our case, we have a number of Tensors acting as slices of the Framebuffer and another set of slices used to store the 3D Gaussian cloud. The vertices implement the projection, rasterisation and blending of 3D Gaussians, which we will discuss in depth later. First, we must define a mapping between the pixels in the rendered image and the slices held on-tile. This mapping is shown in Algorithm 1. We also use a helper class “TiledFramebuffer” to handle the different coordinate transforms involved in distributed rendering, i.e. going from a coordinate in image-space to a local tile coordinate. This class also ensures that the pixels of the image are spread as evenly as possible between tiles, to achieve maximum parallelism. The result of the tiling is illustrated in Fig 4.3 where the tiles are coloured based on tileID. This is similar to what is known as a “tile renderer” GPU, however the slices are allocated and remain pinned in tile memory throughout the program - instead of being implemented in hardware.


Scene Parameters

Env NIF Rotation

Camera Parameters

Field of View

Variable Parameters

Exposure
Gamma
X
Y
Z
Lambda1
Lambda2

Info/Stats

Workload Balance: Splats per tile, max tile: 0
Video rate: -nan Mbps
Frame rate: 4.61 Frames/sec

Render Status

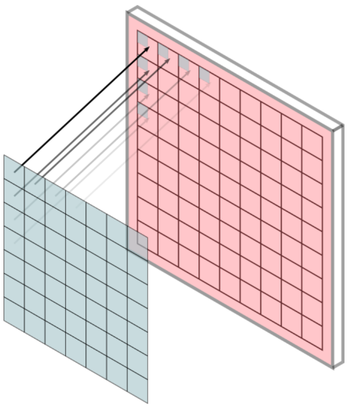
Choose render device: < ipu

Figure 4.3: Tile-Rendered Ellipse on IPU

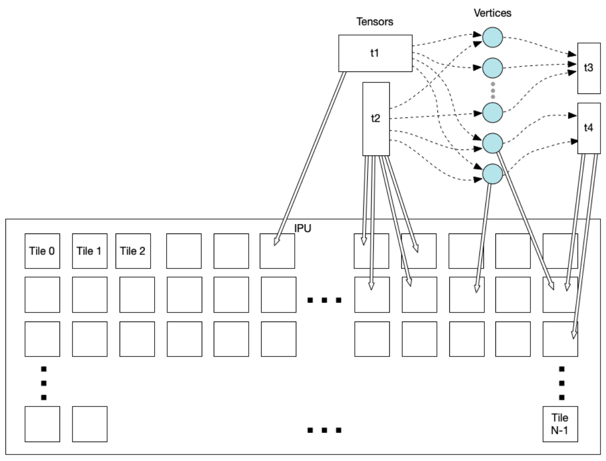
Algorithm 1 Tensor-to-Tile Mapping Algorithm

```
1: Input: Computational Graph $g$, Input Tensor $paddedInput$, Mapping Info $info$,
   Codelets $codelets$
2: $sliceStart \leftarrow 0$
3: $t \leftarrow 0$
4: for $t \leftarrow 0$ to $info.totalTiles$ do
5:     $sliceEnd \leftarrow sliceStart + info.elementsPerTile$
6:     $slice \leftarrow paddedInput.slice(sliceStart, sliceEnd)$
7:     $vertex \leftarrow g.addVertex(codelets[t])$
8:     $g.setTileMapping(slice, t)$
9:     $g.setTileMapping(vertex, t)$
10:    $sliceStart \leftarrow sliceEnd$
11: end for
```

Algorithm 1 is visually illustrated by Figure 4.4a. The algorithm is generic and can be used for anything represented as a tensor. We reuse the same mapping algorithm to distribute clouds of primitives (gaussians) across tiles.



(a) Pinning tensor slices to IPU Tiles.



(b) Tile Mapping with Poplar. For more information, see the Graphcore Developer Docs.

Figure 4.4: Tile Mapping on an Intelligence Processing Unit

4.3 Virtual Camera Setup

Instead of a camera image, the host sends only the camera position to the IPU tiles at each *exchange* phase. The initial camera position is determined on the host by creating a bounding volume around the SfM point cloud, initialising the viewpoint at its centroid, and translating it to look at the box. This is depicted in Figure 4.5. We reuse the GUI from the IPU ray-tracer [6] implementation, with some slight modifications to move the camera more freely.

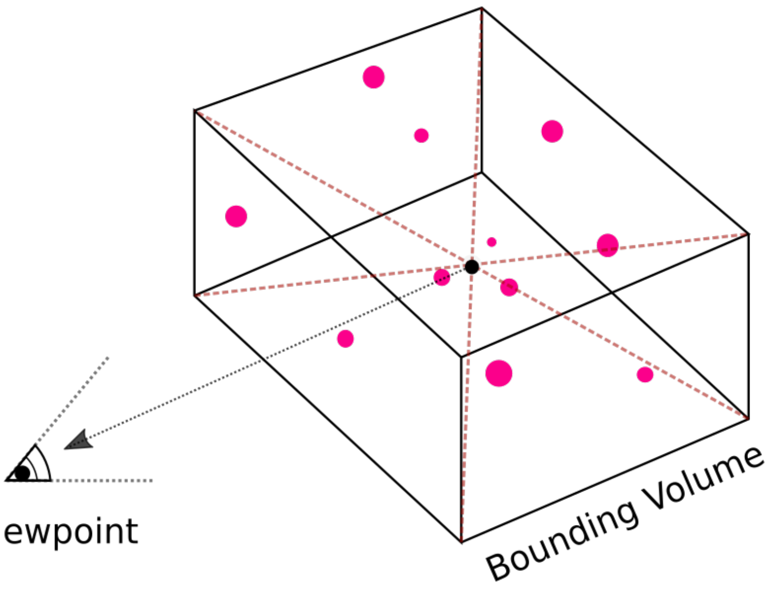


Figure 4.5: Initialising camera pose from SfM point cloud

Once the view and projection matrices are initialised, we send the raw float data to the IPU. After pushing these values we can reinstantiate glm matrices and directly perform vector dot products and other graphics operations on tile. This is thanks to some recent work to support - but not optimise - OpenGL Mathematics library (GLM) on IPU. In previous projects on IPU, vector products and other operations had to be manually implemented [65].

4.4 Exchange

In the original 3DGS paper, when the 2D extent of a projected Gaussian spans multiple tiles of the framebuffer (as illustrated in Figure 4.3), the Gaussians are loaded into the shared memory of each GPU SM that might render it. This is achieved by querying the address in GPU global memory. CUDA defines a global address space which multiple threads can access, and thus may easily share data between SMs.

IPU tile memory must also store any Gaussians that may be rendered during a frame. The IPU's greatest strength lies in the capacity of on-chip memory, each SRAM bank able to store thousands of gaussians at once. However, this advantage comes with a significant compromise: the IPU lacks a *global* address space. What this means in practice is that "dynamic exchange" cannot be performed, where a global memory address - only known at runtime - is queried (e.g. a Gaussian rendered on another tile). The absence of global memory necessitates a completely new algorithm to efficiently move Gaussians to their required locations, some of the algorithms we considered are listed below.

1. Dedicate a subset of IPU tiles to act as routing nodes. Misplaced gaussians would be sent to these to be rebroadcast to the correct tiles.

2. Use half the tiles for rendering, half for routing and sorting. Gaussians could migrate within the sort tiles while the other tiles render (i.e. “double buffer”), after which the new gaussians would be loaded into the render tiles.

3. All tiles participate in rendering and sending. Building a network between tiles, each one is responsible for passing on misplaced gaussians to their destination.

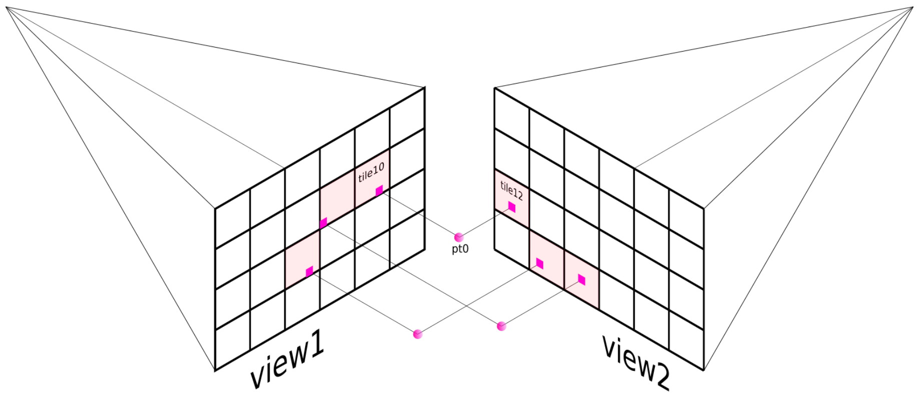


Figure 4.6: Point to tile mapping on viewpoint change.
pt0 must move from ***tile10*** to ***tile12***

These algorithms always **assume Gaussians are uniquely and uniformly distributed in the input set**, because we cannot make guarantees about the spacial distribution of gaussians in a scene. To simplify the problem, the mean of each Gaussian in image-space is treated as an "anchor" point and used to determine which tile the gaussian should be stored on for the current viewpoint. We also use the term anchor to refer to the tile that renders the mean. Figure 4.6 shows how when the viewpoint changes, the mean may project to a different location in the framebuffer and thus may need to change anchor tile.

Each routing strategy presents a number of interesting tradeoffs. *Method 1* assumes that rasterisation will consume the most processing time, and thus dedicates the majority of resources to pixel painting. In reality, we found that compute is dominated by routing. This is because changing viewpoints means all gaussians may be projected to a different tile and must be copied there to render to next frame (see Figure 4.6). In *method 1* the gaussians must all pass through a small number of nodes to be rerouted, which would likely be a huge bottleneck. It does remain an intriguing type of *small world* [66] problem, which we explore in more depth later.

Method 2 attempts to address the bottleneck by striking a balance between routing and rendering. It could work as a form of “scatter-gather” algorithm [67] where gaussians are gathered at a routing tile, and scattered to the correct nodes while the other tiles render the previous frame. The downside of this method is that it requires a complex network of connectivity, since the destinations are unknown until runtime. Scatter operations would need to send fixed size - and possibly empty - packets to all other tiles at every exchange phase.

We opt for *method 3* where all tiles work to reroute the gaussians to the correct destination, while also splatting to its local slice of framebuffer. The rest of this section describes the rendering of a point cloud such as the one shown in Figure 3.2 (just the gaussian means) using *method 3*. We focus mainly on the pattern of exchange between tiles. Section 4.5 outlines the additional steps necessary to render other primitives.

4.4.1 Topology

IPU allows fixed sized tensors to be broadcast between any number of tiles at each *exchange*. Ideally, a tile would sort the gaussians by destination, and then send each slice to the correct destination tile (see Figure 4.8). Unfortunately, the length of the slices is unknown at compile time, meaning we cannot construct this ideal routing pattern. Instead, we take inspiration from network-on-chip (NoC) literature [68] and construct a 2D Mesh Topology to move gaussians around the IPU. The structure is shown in Figure 4.7.

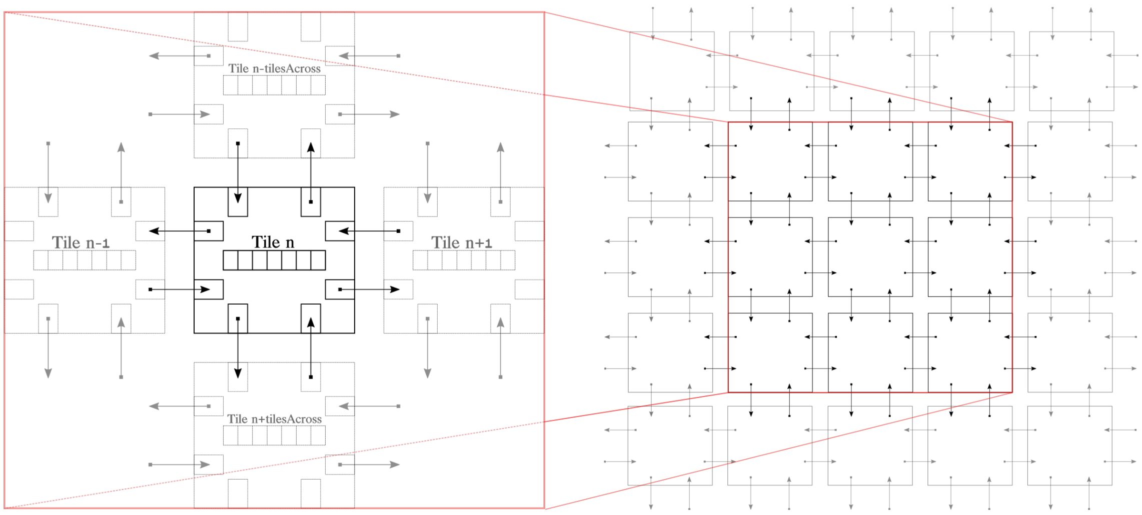


Figure 4.7: 2D Mesh Topology - Exchange Pattern for Rasteriser

In this setup, each tile allocates a fixed-size tensor to SRAM that is used to store points throughout the duration of the program. Separate in/out tensors are allocated for each cardinal direction, to avoid extra synchronisation overhead.

Figure 4.8 illustrates how a 2D Mesh allows a point to move to any tile without knowing its destination at compile time. Note that multiple equidistant paths are sometimes possible; in this case we choose arbitrarily which channel to write to. All points are given a unique ID at compile time, which is used to avoid storing multiple copies in a buffer. Setting this ID to 0 treated as an eviction.

4.4.2 Routing

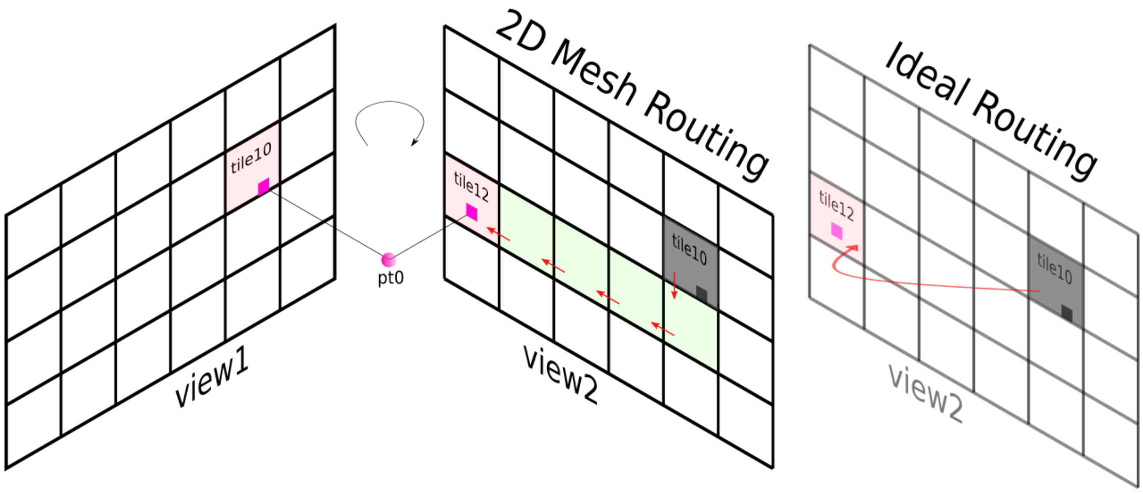


Figure 4.8: 2D Mesh vs Ideal Routing on viewpoint change

At compute time, tiles evict misplaced points and write them to the out channels that minimise the manhattan distance between it and a point's destination tile. At exchange, each out-buffer is copied to the corresponding in-buffer of the tiles in its halo region. In the following compute phase all tiles read from their in-channels and either write points to their internal buffer, or directly to one of the out channels. These steps are outlined in Figure 4.9. Using this method, all points can eventually migrate to their destination in a fixed number of exchanges, as shown in Figure 4.8.

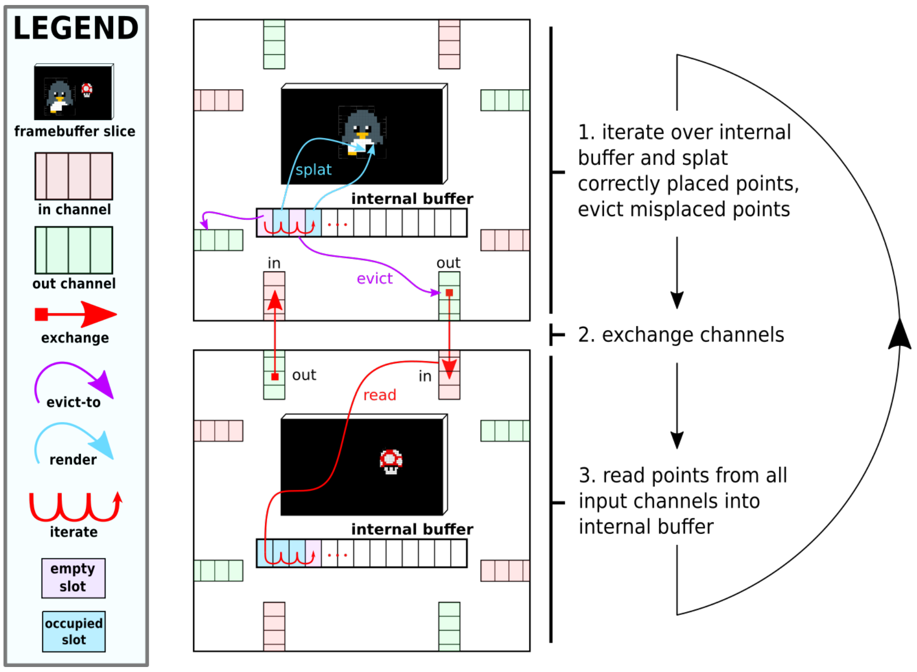


Figure 4.9: Order of IPU operations

Note that almost all operations remain entirely on-chip. The only data written from the host is the projection matrix, and the only data read are the pinned slices of the framebuffer, which can be accessed in parallel. Rendering just point clouds, this method achieves framerates of up to 300fps on models with ∼500000 points.

4.5 Model Primitives

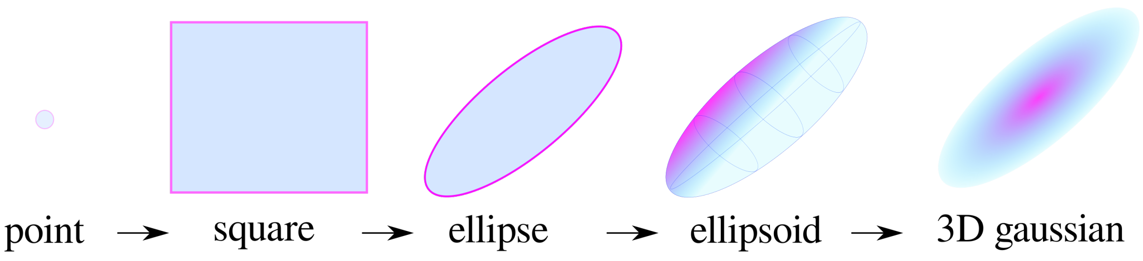


Figure 4.10: Increasing primitive complexity.

The rasterizer was incrementally developed using the primitives shown in Figure 4.10. As this was the first instance of a software rasterizer on an IPU, an obvious starting point was rendering a point cloud. The next step is to render a primitive with an extent that could span multiple tiles. This section outlines the challenges involved in rendering more complex primitives on IPU. The testing was predominantly conducted with just one large primitive in the scene; Section 4.7 describes the added complexity of ordering multiple gaussians based on their distance from the viewpoint.

4.5.1 Squares & Rectangles

We first augmented the points with a 2D square extent applied in clip-space. This extent is independent of the field of view (FOV) and was initially introduced to develop a protocol for rendering a primitive across multiple tiles simultaneously. The approach is especially helpful as it models the bounding box of a 2D ellipse; we go on to reuse much of the logic developed around squares in the final implementation.

The squares are represented by two points defining the top left and bottom right corners. Other representations are possible but this is most convenient for clipping. After the square is initialised in clip-space, the bounding coordinates are converted to tile-space. If the x or y values of either point lie outside the local coordinate frame, then we know that either the square must be copied onto multiple tiles, or some/all of it lies outside the viewing frustum. In both cases the square must be clipped to fit inside the current tile and the remaining local rectangle rendered. The red lines in Figure 4.11 indicate where the square is clipped. The green sub-rectangles formed by the clipping are each rendered on a separate tile.

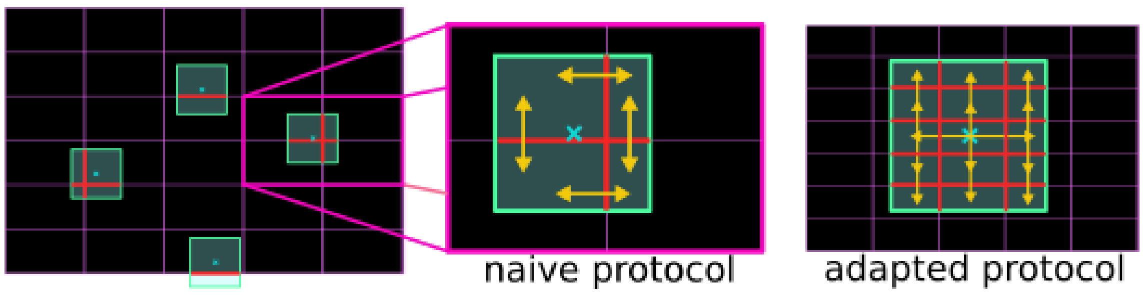


Figure 4.11: Protocols for broadcasting a square primitive

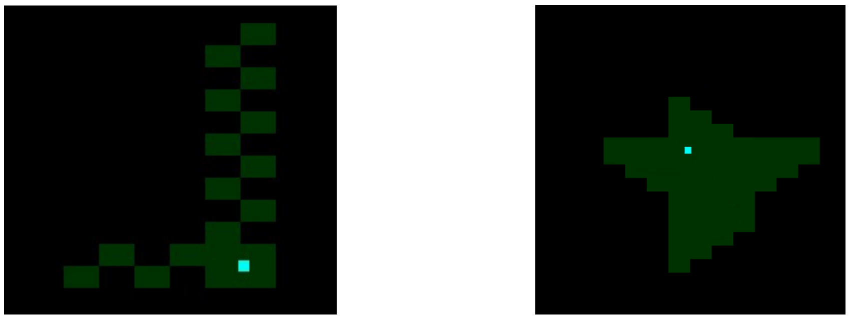

(a) Cyclic behaviour (b) Multiple copies converging

Figure 4.12: Communication of a square using naive protocol. Green tiles show where a copy of the square exists.

It is while clipping that a tile determines which channels a square should be copied to; to ensure the square is rendered on all the slices of framebuffer it covers. A naive protocol for this would be to have each tile send a copy in all clipped directions, regardless of the section of square it renders. However, this can lead to many copies being in-flight at the same time on a view change, or cyclic/reciprocal patterns such as those shown in Figure 4.12. While these side effects do not cause immediate visual artifacts, the fixed-sized out-channels saturate more quickly in a scene with more copies, which leads to data loss when trying to write to a full buffer.

To mitigate this issue, a copy is only sent out to tiles parallel to the square's principle axis, first in horizontal beams and then forwarded vertically. When a tile finds a primitive in its internal buffer that is no longer rendered, it is evicted. Only the tile that did hold the anchor is in charge of sending on the data to its new destination, e.g. tile10 in Figure 4.8. Imposing an ordering like this removes cycles and ensures that only a single copy traverses the mesh on a large view change (see Figure 4.8). Both protocols are illustrated in Figure 4.11. The result of rendering a model with square primitives is shown in Figure 4.13. Completely unoptimised it runs at ~30fps compared to ~5fps on CPU.

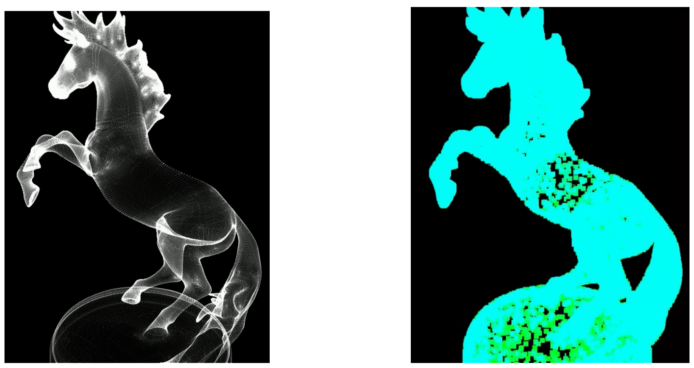

(a) Points splatted on CPU (b) Squares splatted on IPU

Figure 4.13: Horse statue comprised of ~700000 points

4.5.2 Ellipses

Rendering a 2D ellipse is significantly more challenging than a square, but a necessary stepping stone towards rendering Gaussians. In order to reuse the adapted protocol from Figure 4.11, the bounding box of the ellipse needs to be calculated. Finding the minimal axis-aligned bounding box involves taking the derivative of the ellipse equation $\frac{x^2}{a^2} + \frac{y^2}{b^2} = 1$ with respect to x and y, and solving for the max/min values. The result is shown in Figure 4.14.

In the case of a rotated ellipse, this requires computing trigonometric functions which must be simulated in software on the IPU. Testing this implementation revealed it was too slow to render anything in real time (∼3fps). One way to mitigate this is to use a lookup-table implementation which avoids the use of double precision calculations (Cephes Math Library Release 2.1, credit to Stephen L. Moshier), which increased the framerate to about 30fps.

The faster method - and the one used in the original 3DGS - is to use the bounding radius of the gaussian (i.e. 2 standard deviations) as the diagonal of the box, which brings the framerate to ∼40fps. However, since this bound is not minimal, it comes at the cost of more data sharing across tiles. On IPU this can lead to visual artifacts due to channel saturation, especially for very long and thin gaussians. Non axis-aligned boxes are also possible, however these are too slow to compute in practice. The three variations are shown in Figure 4.14.

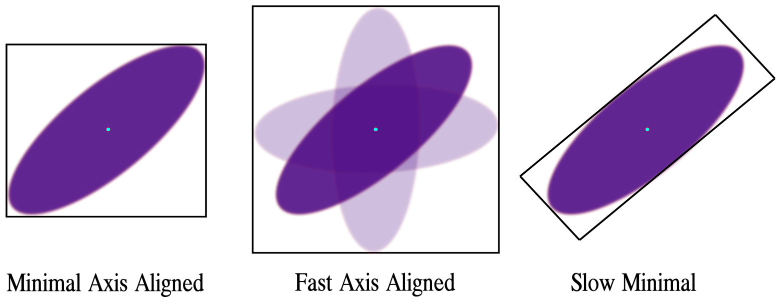


Figure 4.14: Bounding box implementations

We opt for the minimal axis-aligned approach, as it exhibits the best balance between speed and memory. We reuse the protocol from section 4.6.1 and treat the ellipse as a rectangle until render-time, however it is possible to approximate the minimal bounding box (far right in Figure 4.14) by extending the protocol, which would result in a tile mapping similar to Figure 4.15. This is discussed further in Chapter 5.

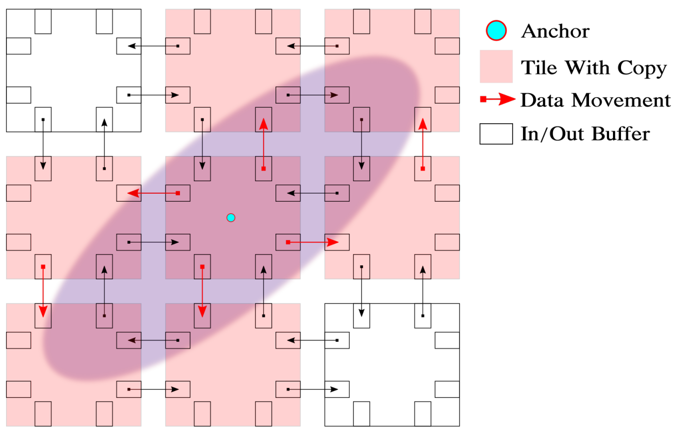


Figure 4.15: Optimal data path on 2D Mesh

4.5.3 3D Gaussians

The next step is to go from a 2D representation to a 3D one. The only distinction we make between 3D gaussians and ellipsoids is that ellipsoids do not contribute to every pixel but are rendered with constant radiance. This helped us to effectively debug the z-buffer algorithm described in Section 4.7.

3D Gaussians are represented by a mean, 3D covariance matrix, opacity (α) and view dependant colour **c**. The original 3DGS paper opt to use spherical harmonics to represent c.

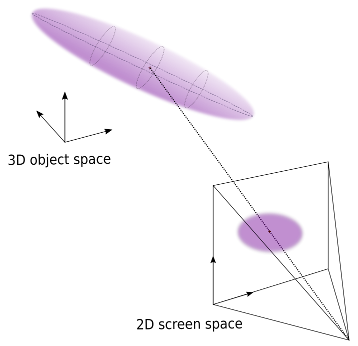
$$\boldsymbol{\mu} = \begin{bmatrix} \mu_x \\ \mu_y \\ \mu_z \end{bmatrix}, \quad \boldsymbol{\Sigma} = \begin{bmatrix} \sigma_{xx} & \sigma_{xy} & \sigma_{xz} \\ & \sigma_{yy} & \sigma_{yz} \\ & & \sigma_{zz} \end{bmatrix}, \quad \mathbf{c} = SH, \quad \alpha$$

However, using the 3D covariance matrix directly makes the optimisation step difficult. Instead, the authors represent the gaussians with three scaling values and a rotation (quaternion) and reparameterise the covariance as the following:

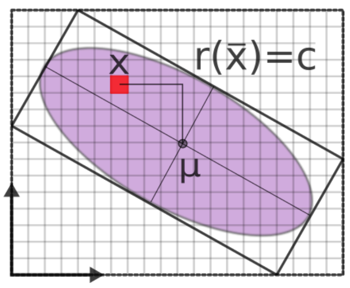
$$\Sigma = RSS^TR^T$$

In this way, the rotation and scale can be optimised independently. We only focus on the rendering step, but use the same representation as the original paper. Equation 4.1 shows how the radiance of a 3D point **x** is influenced by the gaussian covariance. After projecting into screen space, **x** becomes a 2D pixel coordinate, and we represent a gaussian's contribution by a weighted function of distance to **x**, this is described in Figure 4.16b.

$$f(\mathbf{x}) = \sigma(\alpha_i) \exp\left(-\frac{1}{2}(\mathbf{x} - \boldsymbol{\mu})^T \Sigma^{-1} (\mathbf{x} - \boldsymbol{\mu})\right) \tag{4.1}$$



(a) 3D to 2D covariance



(b) Rasterisation using resampling filter **r**

Figure 4.16: Combined caption for both subfigures

The colour at each pixel is calculated using resampling filter **r** from the EWA Splatting algorithm [46]:

$$r(\overline{x}) = \overline{x}^T Q \overline{x} \quad \text{where} \quad \overline{x} = x - \mu \tag{4.2}$$

where Q is the 2x2 inverse of the variance matrix - known as the **conic** matrix. Given the distance from the mean, the resampling filter returns the exponential falloff radiance. This gives us the soft edges shown in Figure 4.17 and is part of what makes gaussian splatting end-to-end differentiable.

Byte Alignment

The format of the representation matters especially for the IPU codelets, since the structure should be byte aligned to the nearest 4B. This ensures that IPU operations can execute as fast as possible by fetching a single memory element per read. If the structure is misaligned, then its possible that a value straddles two memory elements and requires two reads per gaussian. We use the following layout to ensure the gaussians are byte aligned in memory:

```
struct Gaussian3D {
    float4 mean;    // In world space
    float4 colour; // RGBA colour space
    float4 rot;     // Local rotation of gaussian (real, i, j, k)
    float3 scale;  // S matrix
    uint32_t gid;     // Unique gaussian ID
}
```

We also use custom types to hold vectors of 32bit float values, which removes extra metadata such as function pointers in the data structure. This way, the struct is exactly 64B wide and easily indexable in the SRAM of the IPU tiles.

As discussed, the algorithm for splatting 3D gaussians remains the same as the one proposed in the paper. A 2D covariance is computed from the values above using the viewmatrix and field of view. This is then used to calculate the eigenvalues for the bounding box (as in section 4.6.2), and the conic opacity for alpha compositing.

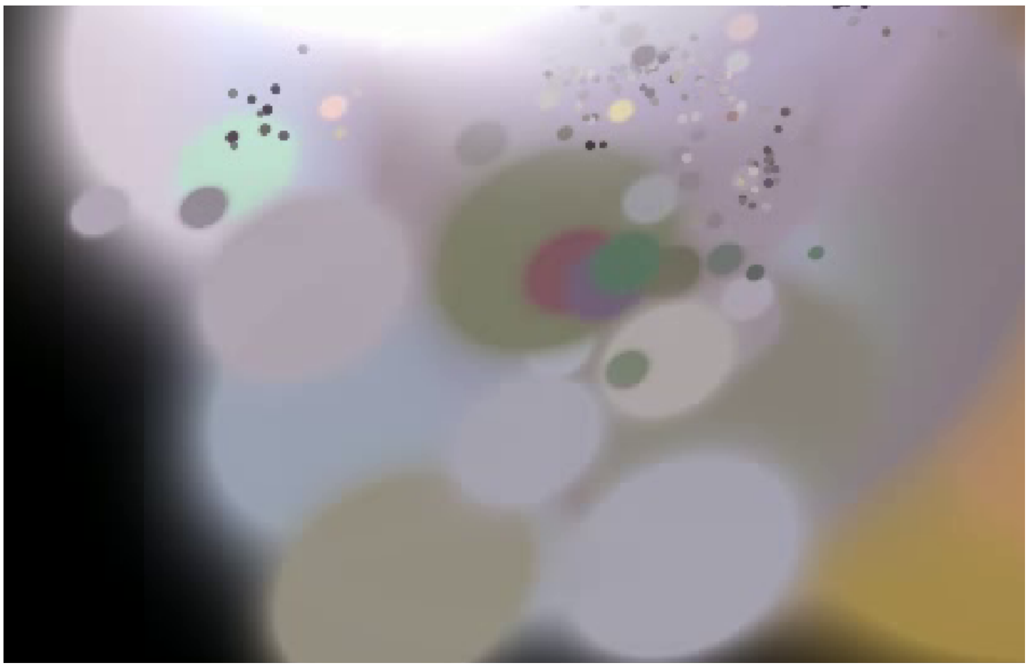


Figure 4.17: Blended gaussians on multiple IPU tiles

4.6 Z-Buffering & Volume Splatting

In the original 3DGS, a global sort of the gaussians is performed, based on depth and tile ID. This is done in global memory, using a fast GPU radix sort - after which, the sorted sections are loaded into GPU shared memory of each SM (according to tile ID). The threads of each SM then iterate through the section of the sorted list in the local shared memory and accumulate the contributions for the pixel they are assigned to.

In the case of IPU, there is no global memory. This means the sort must be either be done locally on-tile, or the gaussians must be read back to the CPU, sorted, and then rebroadcast to the tiles. The latter is too slow and defeats the purpose of keeping everything on-chip.

Gaussians visible to the on-tile slice of framebuffer are determined during the initial pass over the internal memory buffer, as shown by step 1. in Figure 4.9. A separate array is allocated that holds only these gaussians to render, which is subsequently sorted. We use a custom iterative merge sort because IPU tiles cannot dynamically determine the stack size of recursive functions.

4.6.1 α-compositing

Once the gaussians are sorted in depth order, the tile iterates over the pixels in its framebuffer slice. For each pixel, the contributions of each 2D gaussian in the sorted array is accumulated using the formation model shown in Figure 4.18. This returns the accumulated radiance for each pixel given by Equation 2.1. The result of this blending can be seen in Figure 4.17 where the gaussians are shared over many tiles, but all tiles coordinate to evenly blend them.

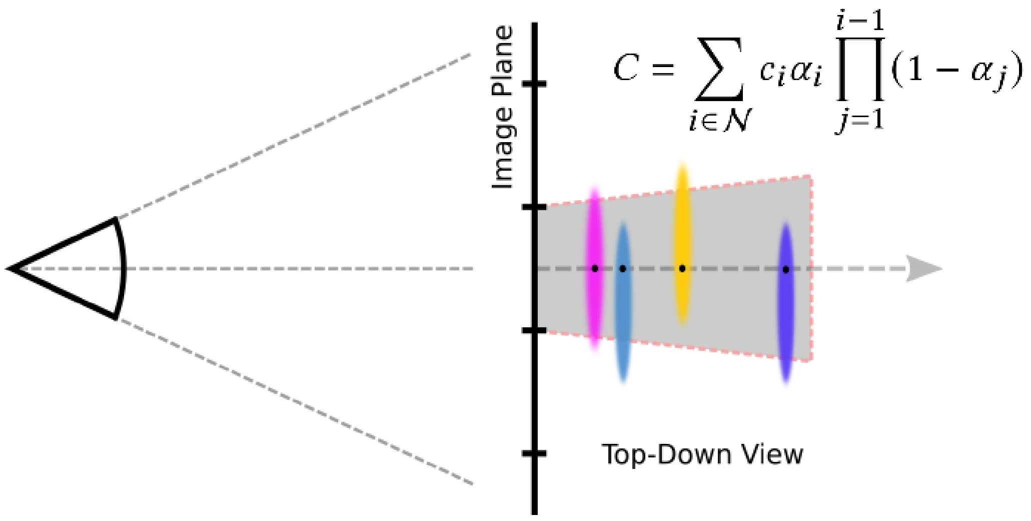


Image Plane
$C = \sum_{i \in \mathcal{N}} c_i \alpha_i \prod_{j=1}^{i-1} (1 - \alpha_j)$
Top-Down View

Figure 4.18: Alpha blending volumes in screen space

Chapter 5

Further Applications

5.1 Grid Networks & Small Worlds

Some of the algorithms we considered for our exchange network present interesting properties that may be exploitable in other related domains. In this chapter, we discuss how our work links to these domains and why novel architectures like Graphcore's IPU are worth exploring.

The protocols we developed could be applied to many other representations, one example is NeRF. vMAP [14] and KiloNeRF both optimise many small MLPs and aggregate their contributions. We could map a number of MLPs to different tiles and optimise them in parallel, and then compose the distinct parts of the scene using the same method as our 3DGS implementation. This could be especially effective as MLPs are a far more compact representation than 3D gaussians, meaning we would be less likely to lose information due to channel saturation.

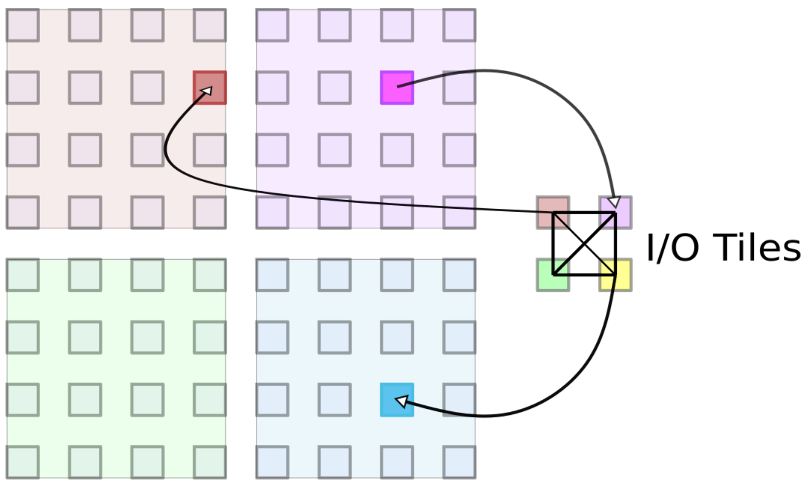


Figure 5.1: Routing between clusters using a small subset of tiles

We originally wanted our communication topology to exhibit certain characteristics that bear resemblance to "small world" networks. Small world networks are characterized by a high degree of local clustering (nodes tend to form tightly-knit groups) and short average path lengths (any node can be reached from any other node in a small number of steps) [66]. These networks contrast with random networks and regular lattices, such as the one we construct in Figure 4.7. In the case of IPU3DGS we frame the problem in a spatial way, where each gaussian must navigate to the correct region of the framebuffer to be rendered. In section 4.4, we discussed various methods of moving data around the chip and opted for a regular grid; however, we may be able to exploit the IPU's flexible exchange pattern far more effectively. Figure 5.1 shows a small-world network we create using left-over tiles to route data in a small number of exchange cycles.

We originally dismissed this algorithm as it would likely bottleneck a large discrete representation like 3DGS. However, it may work well for algorithms like Gaussian Belief Propagation (GBP). For instance, we could use a hierarchical structure of nodes to perform coarse-to-fine optimisations - similar to existing system-on-chip architectures like SPIN [69]. Bundle Adjustment has been proven to work well on IPU [65], provided the factor graph is mapped to certain tiles at compile-time. Perhaps we could use a generic tree of precompiled routing nodes and then prune and add various factors using a similar send/evict policy to our gaussian routing.

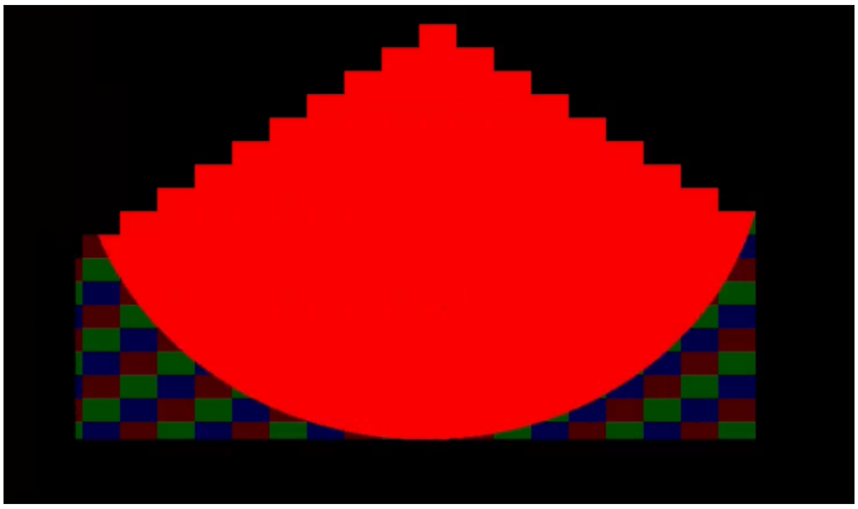


Figure 5.2: Blooming artifact with sending protocol

Minor adjustments to our lattice could also provide a number of benefits. Currently, large gaussians take time to propagate from the centre point, which means changing viewpoints causes a visual artifacts like the one seen in 5.2. We could add some skip connections to the mesh, so as to propagate the gaussians in just a small number of hops from the centre.

Another interesting application is multi-device optimisation. In our system, we do all computation entirely locally, however the same methods could be used to render/optimise scenes on multiple different chips. If the viewpoints are known on all devices then we could aggregate the information by passing messages in the same way we pass gaussians using the IPU exchange fabric. In this setting, the exchange pattern would likely be far more flexible than an IPUs precompiled pattern.

Chapter 6

Evaluation

In this section we discuss the performance of the system and takeaways from this project. This work aims to answer key questions about scene representations on a distributed processor like a Graphcore IPU.

6.1 Performance Evaluation

We successfully construct an end-to-end software rasteriser on a graph processor. While rendering on IPU has been done before - as demonstrated by the IPU ray-tracing implementation - never has scene information been dynamically moved around based on viewpoints only known at runtime.

All data in our implementation is kept entirely in on-chip SRAM. In the ray-tracer, a pretrained NeRF model is copied to all tiles and the ray data is streamed to each tile from external DRAM. The DRAM latency is hidden by using a double buffer; accumulating contributions of the previous rays, while simultaneously fetching new ray data from the host. While this allows for extremely high resolution images to be rendered, it involves constantly pushing data from DRAM. We avoid this entirely and only load data onto the chip at compile time. While there are improvements to be made in terms of visual quality, this initial implementation may serve as the groundwork for other on-chip scene representations.

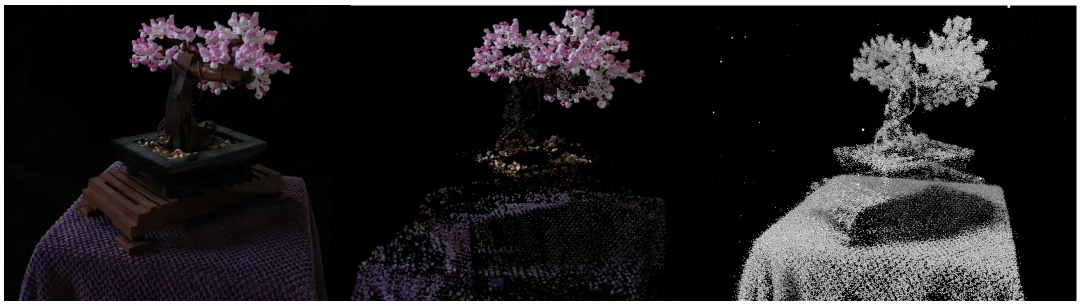
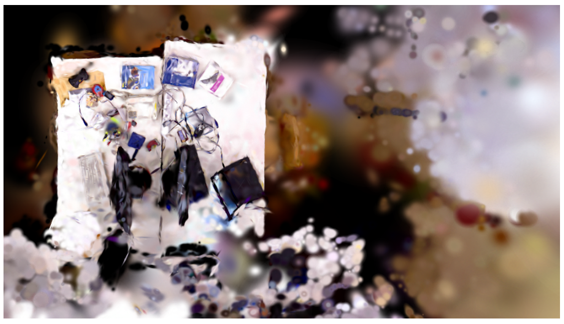
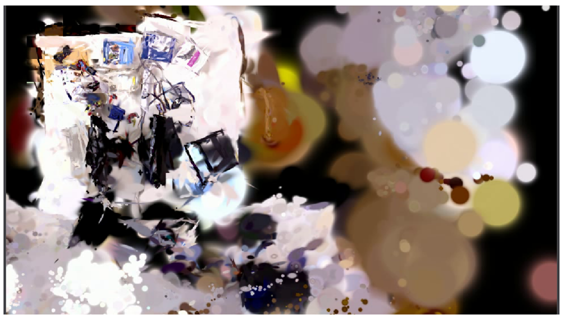

(a) Original 3DGS (b) IPU 3DGS (c) Isotropic Gaussians

Figure 6.1: Comparison of different IPU and GPU 3DGS implementations

Figure 6.1 shows a comparison between a scene rendered with 3DGS, vs the IPU implementation. For an arbitrary viewpoint, the IPU render approaches the quality of the original implementation in some areas of the scene, namely the leaves. The scene is made up of 272956 gaussians, which we render at 8fps.

The geometry is fully described by the gaussians shown in 6.1c, however the density of the model varies enormously in different areas. In the IPU implementation this causes regions where the channels between tiles saturate, meaning some parts are not rendered - we will discuss this problem in more depth later. In the case of smaller models and point clouds we can efficiently render most scenes below 300000 primitives.


(a) 3DGS TUM Desk


(b) IPU 3DGS TUM Desk

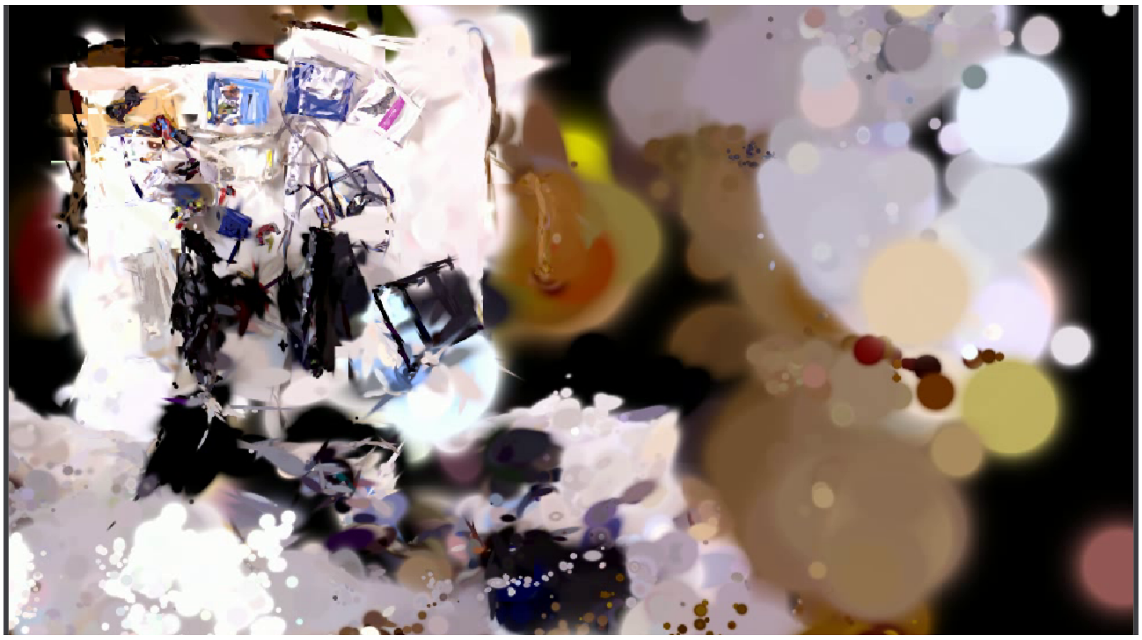


Figure 6.3: Comparison of TUM Desk scene with original 3DGS

Figure 6.3 shows a scene rendered on IPU in which fine details such as cables, books and computers can be seen. The quality is almost the same as that of original implementation. The protocol for sending gaussians works well and can handle view changes. However, copies take time to propagate, which is especially noticeable for gaussians with large bounding boxes. Furthermore, an abundance of copies may cause the internal buffers to saturate, which leads to holes in the model and square artifacts where large gaussians have failed to propagate to all regions of the scene they cover. With more time, one might explore optimal space filling algorithms such as Jump Flood or Hilbert Curves, however the exchange topology constructed in section 4.5.1 may need to be modified. As discussed, this kind of problem bares resemblance to algorithms such as GBP where factor graphs are dynamically updated using message passing between nodes [65].

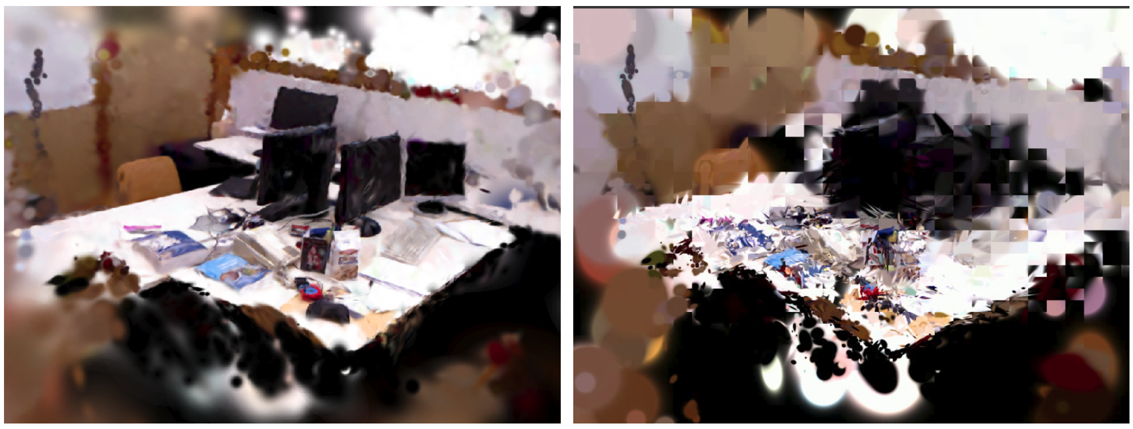

(a) 3DGS TUM Desk (b) IPU-3DGS TUM Desk

Figure 6.4: Effect of small channels on IPU implementation

Figure 6.4 shows a comparison of the same room-scale scene where the channel size has been made too small. It displays the tiling artifacts and missing geometry which larger models may exhibit.

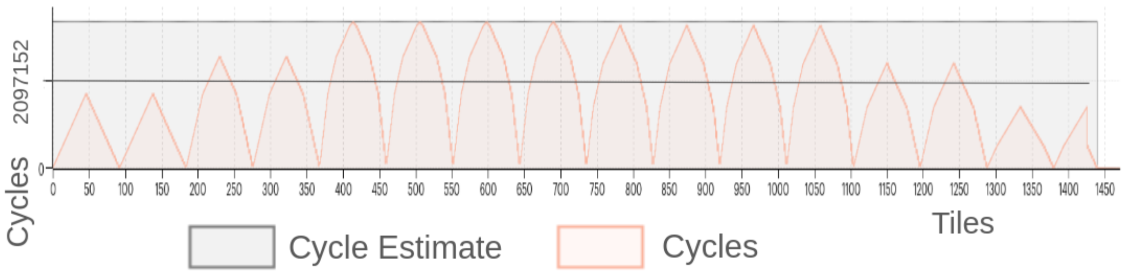
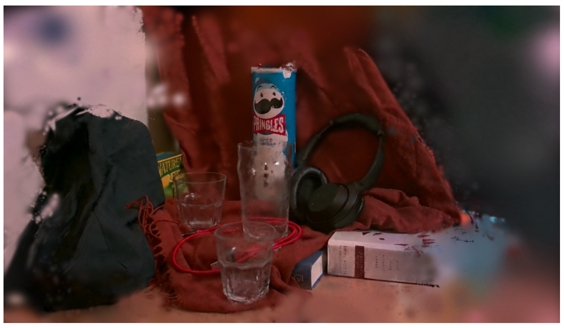
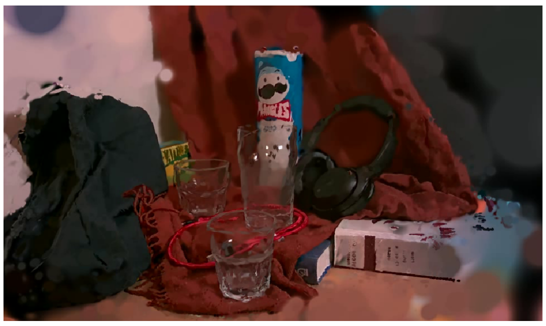


Figure 6.5: Cycles taken to read framebuffer

We analysed the time spent communicating with the host and found that our implementation keeps reads from IPU memory to a minimum. Figure 6.5 depicts the number of cycles taken to read each frame back to the host. The bands in the graph show that reading from tile memory does not take a uniform amount of time. This may be due to predefined memory access patterns on streaming memory, or possibly the layout of pixels in each section of framebuffer. Overall, total time reading/writing from CPU to IPU SRAM takes a fraction of the total cycles at only 1.6%. This is shown by the execution trace in Figure 6.8 (“StreamCopyMidHost”) meaning almost 100% of computation remains local and on-chip. This demonstrate that our method successfully evaluates scenes *without* main-memory.

FlashAttention and other related algorithms are extremely important for transformer models, in order to reduce the number of reads from High Bandwidth Memory (GPU main memory) to on-chip SRAM [19]. They use tiling to achieve a 3x speedup of models like GPT-2. We show that for a rendering system like 3D Gaussian Splatting, it is possible to completely avoid main memory by leveraging IPU's inter-tile exchange fabric. Taking a new approach to algorithm design enables us to perform local computations but migrate data between cores.


(a) 3DGS Pringles


(b) IPU-3DGS Pringles

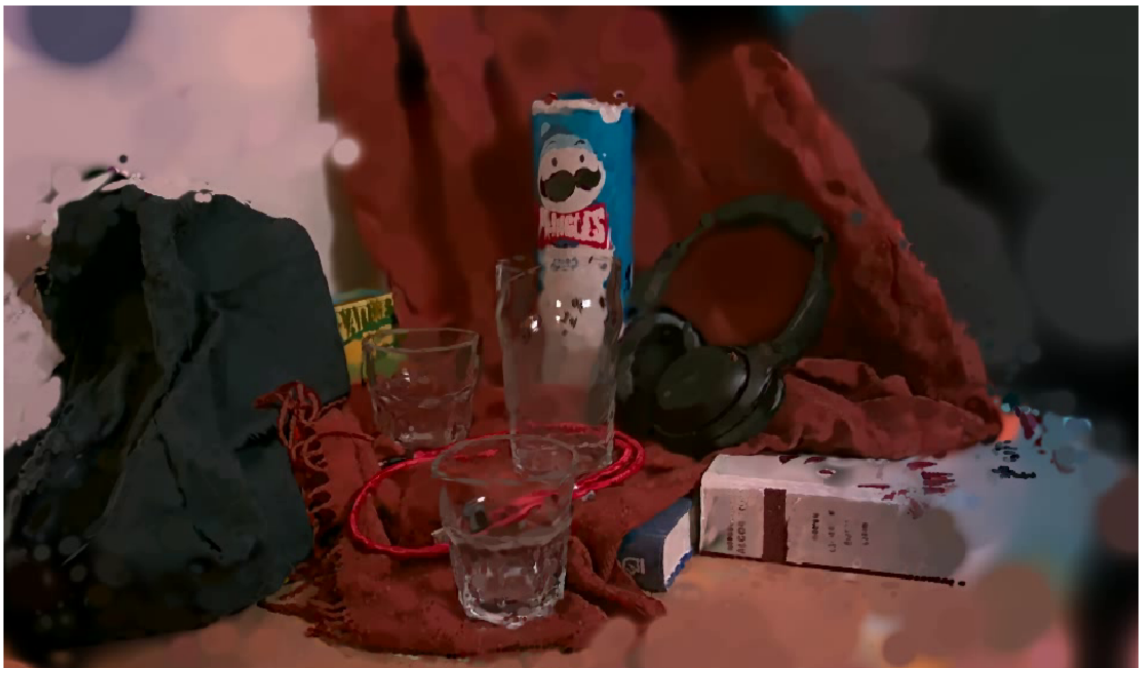
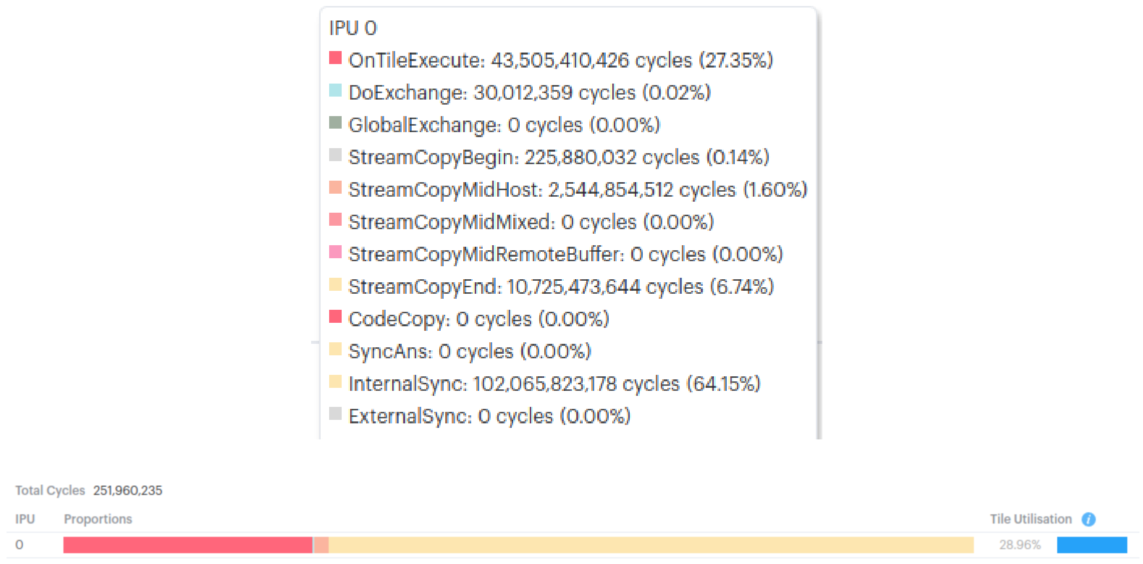


Figure 6.7: Comparison of Pringles scene with original 3DGS

Another example of our reconstruction is shown in 6.7. In this scene, objects and even words are clearly visible. The reconstruction is almost indistinguishable from the original model. With some tuning of the channel size (as discussed in section 6.2.1) we should be able to reconstruct a photorealistic picture entirely in SRAM.


IPU 0

■ OnTileExecute: 43,505,410,426 cycles (27.35%)
■ DoExchange: 30,012,359 cycles (0.02%)
■ GlobalExchange: 0 cycles (0.00%)
■ StreamCopyBegin: 225,880,032 cycles (0.14%)
■ StreamCopyMidHost: 2,544,854,512 cycles (1.60%)
■ StreamCopyMidMixed: 0 cycles (0.00%)
■ StreamCopyMidRemoteBuffer: 0 cycles (0.00%)
■ StreamCopyEnd: 10,725,473,644 cycles (6.74%)
■ CodeCopy: 0 cycles (0.00%)
■ SyncAns: 0 cycles (0.00%)
■ InternalSync: 102,065,823,178 cycles (64.15%)
■ ExternalSync: 0 cycles (0.00%)

Total Cycles 251,960,235

IPU	Proportions	Tile Utilisation
0		28.96%

Figure 6.8: IPU 3DGS Execution Trace

6.2 Optimisations

6.2.1 Tile-Level Parallelism

Our IPU implementation does not exploit tile-level parallelism in any way. This means we only effectively utilise 1440 threads and still achieve framerates around 10fps. IPU tiles have a total of 6 worker threads that can be utilised with very little memory contention. As almost all operations we perform are parallelisable, we can in theory achieve much higher fps. One easy optimisation would be to compute the contributions for each pixel in parallel - we currently perform all pixel operations sequentially. The fact that we still achieve moderate framerates is testament to the speed of on-chip computation and is very promising for any future on-chip scene representations.

6.2.2 AMP Engine

Another optimisation that we do not exploit is the IPU's Accumulating Matrix Product (AMP) engine. A matrix-vector product can be interpreted as taking a linear combination of the columns of the matrix. I.e. a matrix which projects a vector into its "column space": the vector space spanned by its columns. The AMP engine works in exactly the same way. It is a "column scaling" engine (a systolic array) where partially scaled columns are fed to the next unit in the array and results accumulated until the values drop out of the end of the pipeline. Typically machine-learning applications on the IPU are written in one of the packages that sit above Poplar in the stack, e.g. PopTorch (essentially a PyTorch wrapper for IPU), which are compiled to utilise the AMP engine effectively. In our case, we write the Poplar vertices by hand, which means they are not automatically compiled to use the AMP engine. To make effective use of optimised matrix-vector products we would need to load the AMP engine at the assembly level, which was out of the scope of this project.

6.2.3 Load Balancing

As the IPU is as a bulk-synchronous device, all tiles perform a global sync after they finish executing their vertices. There is therefore a variable time between exchange phases, as shown in Figure 6.9. What this means in practice is that execution time is bounded by the slowest tile. In this section we demonstrate how certain viewpoints affect the distribution of data on chip, and thus the framerate as well.

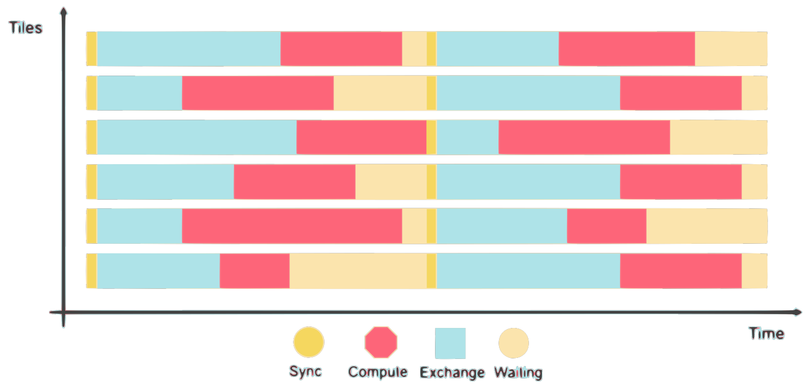
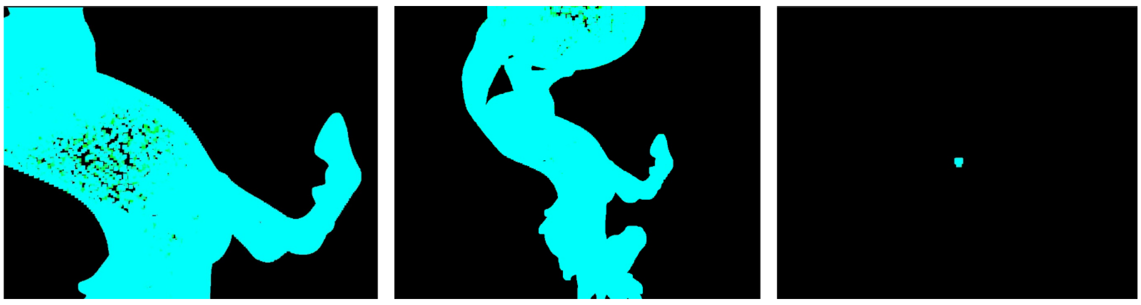


Figure 6.9: Variable times between sync

Figure 6.10 shows the distribution of primitives according to the viewpoints shown. The framerate varies according to the number of points each tile has to render. The graph on the far left shows how a close up view of the model leads points to be more evenly distributed across tiles. The same is true of the middle figure, but this has more, densely clustered, points and thus the framerate decreases - since the tiles have to iterate over a larger number of points in their buffers. The graph on the far right is the distribution when we zoom out very far. This leads all the primitives to cluster in the centre of the image plane, saturating two of the central tiles. Despite rendering a fraction of the total points, this view is almost as slow as the middle pose, which demonstrates the importance of load balancing on an IPU. This shows that to improve on the baseline implementation we should find ways of utilising the empty tiles.

One way in which we could balance the IPU more effectively would be to detect empty tiles at runtime and offload some of the projection and geometry calculations to them. We could do this by keeping a copy of the bounding volume on all tiles and detecting if any tiles are outside this boundary after projection - these will be unused. Gaussians could then be offloaded to these unused tiles.



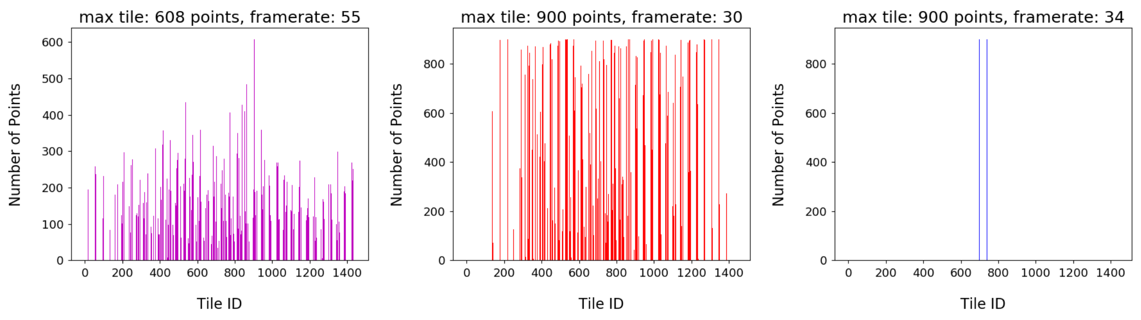


Figure 6.10: Distribution of primitives on tiles of IPU

The imbalance of primitives could also indicate that a different, more continuous representation such as KiloNeRF [70] would be better suited to distributed compute, as we could better interpolate between tiles rather than sending discrete packets.

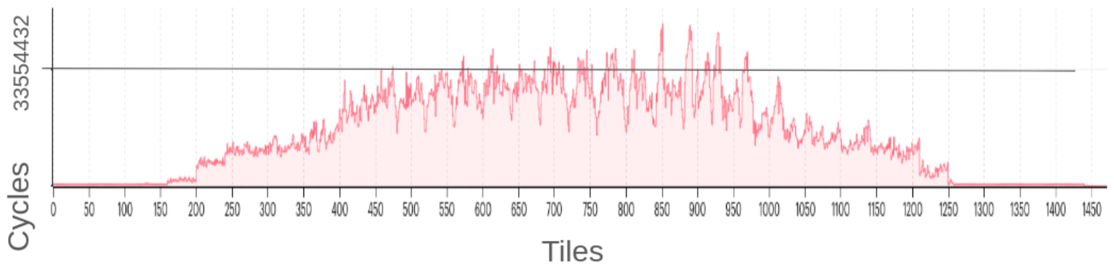


Figure 6.11: Number of cycles per vertex

Further investigation using Graphcore's Poplar Graph Analyser reveals the number of program cycles on each tile. This graph displays the same characteristics as Figure 6.10 showing how the central tiles are utilised far more than those on the edge. We can see exact numbers in Figure 6.8, that show 64% of the time is spent waiting for the slowest tile to finish. Given that most of the tiles are underutilised, if we find ways to balance the load more effectively, the implementation may be orders of magnitude faster. This is further evidence that distributed radiance fields are an extremely promising avenue of exploration.

6.2.4 Memory Usage

Different configurations of tile memory are made possible in our implementation using the TiledFramebuffer class. For evaluation we used a 1280x720 image, divided into 1440 equal slices of 640 pixels. Each slice is 32x20 pixels. We do not make optimal use of tile memory, since we store the pixels as 32bit float values as opposed to 8bit char. Furthermore, we store use a RGBA colour space; the alpha channel is not needed as the alpha values of the scene are taken into account during the blending of gaussians. Removing the alpha channel and converting to char would reduce the memory footprint of the framebuffer significantly and allow us to fit more gaussians on each tile. Table 6.1 shows how these small changes can reduce the memory footprint of the framebuffer by 80%. These changes are easy to make, but were not critical for a first prototype.

	Current	**Compressed**
Total Pixels	921600	921600
Total Memory (KB)	14745.6	2764.8
Pixels Per Tile	640	640
Memory Per Tile (KB)	10.24	1.92

Table 6.1: Framebuffer compression

After storing exchange code and other necessary variables (framebuffer, thread stacks, fixed array for indexing etc.) we are left with just under 500KB of memory on each tile. We use this to allocate the internal gaussian storage buffers and communication channels.

	TUM Desk	**Pringles**	**Bonsai**
Number of Gaussians	62040	91450	272956
Model Size (KB)	3970.56	5852.80	17469.18
Per-Tile Model Capacity (%)	12.6	8.5	2.9

Table 6.2: Model memory footprint

It is important to consider the tradeoff between channel and internal buffer size. Having larger channels means more gaussians can be shared between tiles which reduces the number of square artifacts like those in Figure 6.13. These artifacts happen because large gaussians are constantly broadcast to neighbouring tiles. In areas of dense geometry, if too many gaussians spill over to other tiles, the out channels become saturated and some copies may not reach their destination. The problem is illustrated in Figure 6.12. While large channels mitigate this problem, if we don't dedicate enough memory to storing gaussians then we will lose a lot of detail in the blended image.

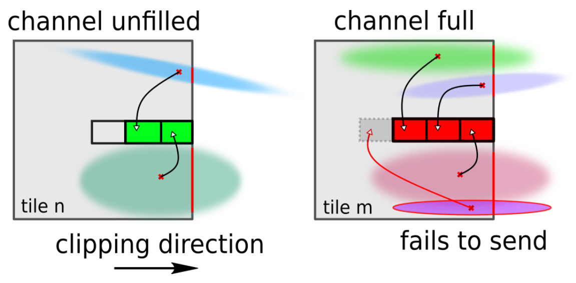


Figure 6.12: Failing to propagate gaussian when clipping

We tested different configurations and found that using channels of 400 gaussians (200KB Total) and internal buffers of around 4700 gaussians (300KB Total) provided the best balance between rendering and sending for most models. Table 6.2 compares some different model sizes; note that Bonsai is especially dense and thus we can only store a maximum of 3% of the model on one tile, compared to 12% with TUM Desk. An added benefit of our design is that the ratio of channel to storage size is configurable at compile time. Models with regions of small densely packed gaussians will render more faithfully if we dedicate less memory to channel size.

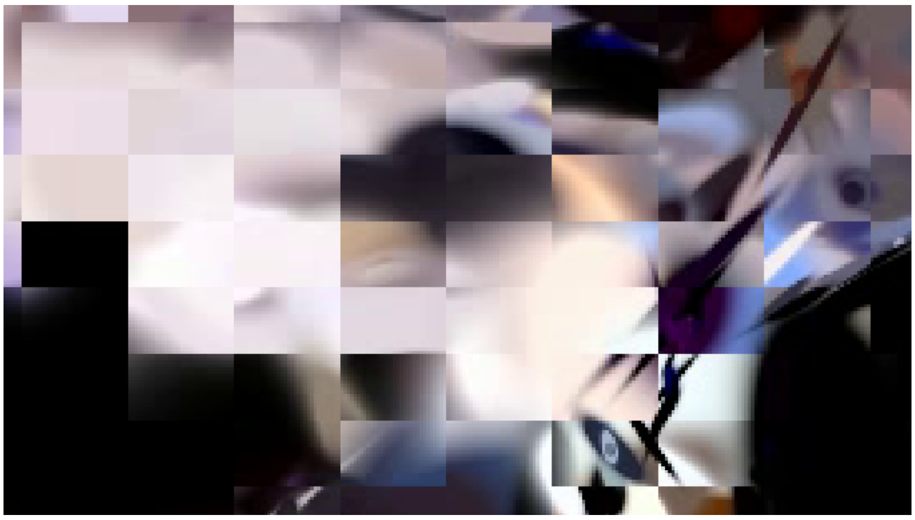


Figure 6.13: Tiling artifacts caused by channel saturation

6.3 Exchange

In this section we evaluate our method of moving gaussians around the chip. We also consider alternative implementations and discuss their tradeoffs.

6.3.1 Protocols

As discussed in earlier chapters, we develop custom protocols to send gaussians between tiles and to propagate their extents out from an anchor point. While this works and achieves good visual results for static scenes, there are many side effects which are less desirable for a future real-time system. One example is the blooming effect shown in Figure 6.14 which is visible after a sudden view change. This effect is present because each exchange cycle corresponds to a frame. To hide it, we must either render frames fast enough for it to not be visible, or develop alternative protocols to handle faster propagation.

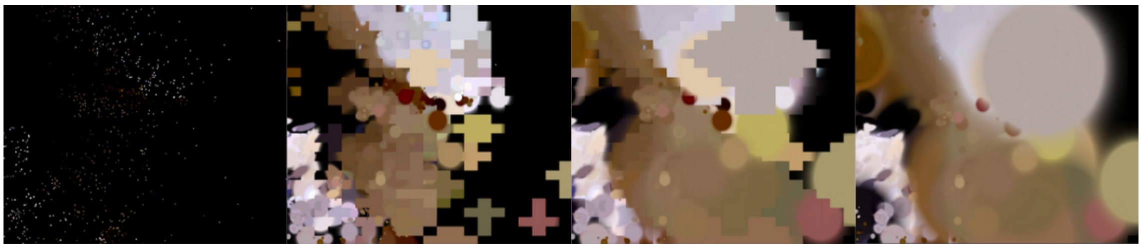


Figure 6.14: Blooming side effect of protocol on 3D gaussians

An alternative method would be to keep gaussians that have a large extent on many tiles, and not evict them on a view change. It may be possible to develop heuristics where only small foreground gaussians migrate using the protocols developed in section 4.5.1. A straightforward way of doing this on a view change would be to calculate if the both the current and new anchor tile lie within the gaussian's reprojected 2D extent. If so, then the gaussian need not be evicted from its current anchor tile. This was a promising avenue of exploration, but developing a first prototype was more critical in the time frame.

6.3.2 JIT Compilation & Dynamic Exchange

This work was done under the assumption that the pattern of exchange is completely fixed during runtime. While this is mostly true, there exists a highly experimental method to perform dynamic exchange using Just-In-Time (JIT) compilation. JIT is a process whereby high-level languages are compiled into machine-code at runtime rather than prior to execution. Typically, it is used to strike a balance between interpreted languages like Python and compiled languages like C++. Optimisations can be applied to hotspots at runtime by profiling code during execution and injecting context specific operations that improve performance, e.g. loop unrolling, function inline etc.

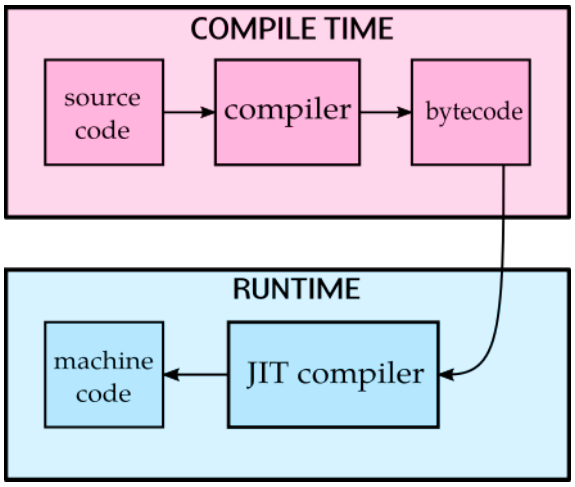


Figure 6.15: JIT compilation

In the context of IPU, an experimental package has recently been developed known as JIT Dynamic Lookup (JDL). The prototype exposes an operation which makes Just-In-Time modifications to Ahead-Of-Time compiled code; which in this case is the precompiled exchange program. This enables dynamic lookup of data on different tiles at runtime, subject to several caveats. Using this package would enable us to query slices of memory from other tiles only known at runtime. JIT compiling would remove the constraints imposed by a protocol and likely lead to a better implementation.

While there are numerous benefits to using JDL, it only supports lookups from slices that do not straddle multiple tiles. In practice this means all tiles would need to query all other tiles for gaussians, which may be slower than the protocols used in our implementation. Alternatively, a system of routing tiles could be used to pull data from all other tiles, after which these tiles request their respective packets from the routing nodes. However this strategy bares resemblance to the bottlenecked approach proposed in section 4.4. Futhermore, JDL is not currently available in the latest Poplar SDK, which means it has not be tested extensively.

While JDL provides a promising route to more flexible exchange, we wished to solve 3DGS using a system of local message passing, as the algorithms are not constrained to the specific hardware architecture. Instead the implementation is generic and could be applied to other architectures and many-device systems.

6.4 Tuning

There are a number of other small modifications that could be made to tune the performance of the system and render well above 10fps. Below is a list of these changes, none of which are difficult to implement but were less critical during the development of the current prototype.

- Use reduced precision in floating point computations, since scene representations are often approximate anyway.
- Make more efficient use of gaussian ID bits (don't need 32 bit unsigned)
- Fuse multiple loops in codelets. The current implementation passes over long arrays several times when a one-pass method is preferable.
- Maintain a sorted array of gaussians rather than evicting and inserting randomly and then sorting. Keeping the array sorted would open up a number of optimisations at the codelet level.

Many of the issues with the current implementation come from an under-optimised system. However, this work is a baseline prototype and demonstrates that we are capable of rendering scene representations on IPU even with very little tuning.

6.5 Concluding Remarks

Ultimately, we have proven that other general-purpose parallel architectures are viable for radiance field rendering. The data collected in this evaluation shows the extent to which we leverage the multicore capabilities of IPU. We outline which parts of the implementation can be optimised and propose solutions to the current limitations imposed by our method. Evaluating the implementation is especially important for any future work on low-power distributed compute. The hope is that the observations from this work provide a platform for future projects involving distributed radiance field rendering.

Chapter 7

Conclusion

Exploring alternatives to GPU for AI and parallel processing is crucial given the current technological landscape. GPU, originally designed for graphics processing, has evolved to become increasingly general-purpose and is now the dominant architecture for parallel processing. The current trend at NVIDIA is to push a new chip to market every year, with a focus on optimising the existing architecture for training models, rather than addressing fundamental weaknesses. One such weakness is High Bandwidth Memory (HBM) latency, which necessitates the use of fully-fused kernels and sophisticated algorithms like FlashAttention to mitigate. Recent versions of NVIDIA GPUs such as Grace and Blackwell boast phenomenal interconnect and memory bandwidths, which also helps to hide the problem. However, in low-power applications such as robotics, this strategy is not an option.

Recent work at Oxford University [71] shows that LLM performance will begin to plateau with each new iteration, as performance is dependant on an exponentially increasing amount of data. We should eventually pivot towards spatial computing and zero-shot learning to develop systems with true understanding of the world. Low-power, heterogeneous and distributed architectures may become just as important as large GPU platforms for pretraining models. It is therefore imperative to investigate and develop new hardware solutions like IPU, that can potentially offer more efficient and specialised processing capabilities. Research into alternative architectures is vital for the continued advancement of AI and parallel computing, ensuring that future innovations are not constrained by the limitations of current GPU architectures.

On an IPU, the biggest problem with having no global memory is that the vast majority of parallelisable programs contain a large reduction at the end, where global information is aggregated. On a GPU, global memory enables this reduction, but at the cost of other things. Arguably one of the biggest questions in the field of AI is how to consolidate local, low-power compute with a this concept of global knowledge aggregation. The two concepts seem antagonistic and yet this is likely the key to unlocking true human-level understanding in machines.

Our work successfully demonstrates the first fully functional 3DGS renderer on an alternative parallel architecture. Regardless of how future hardware adapts, the implementation remains a useful experiment to enable better understanding of in-SRAM radiance field rendering.

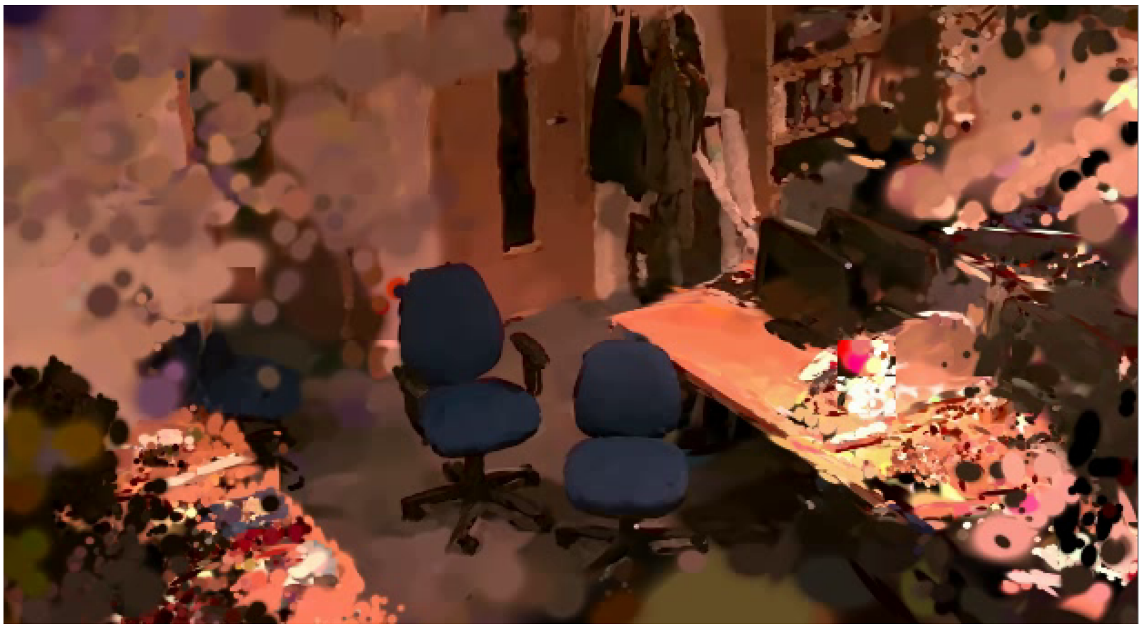


Figure 7.1: Office Chairs

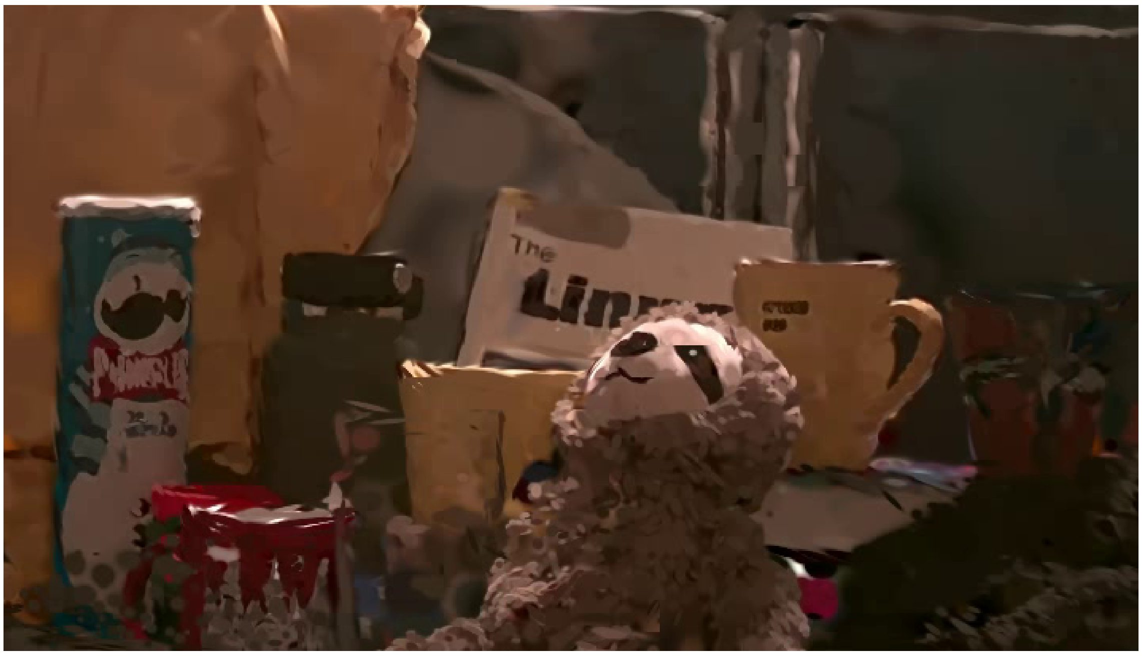


Figure 7.2: Sloth

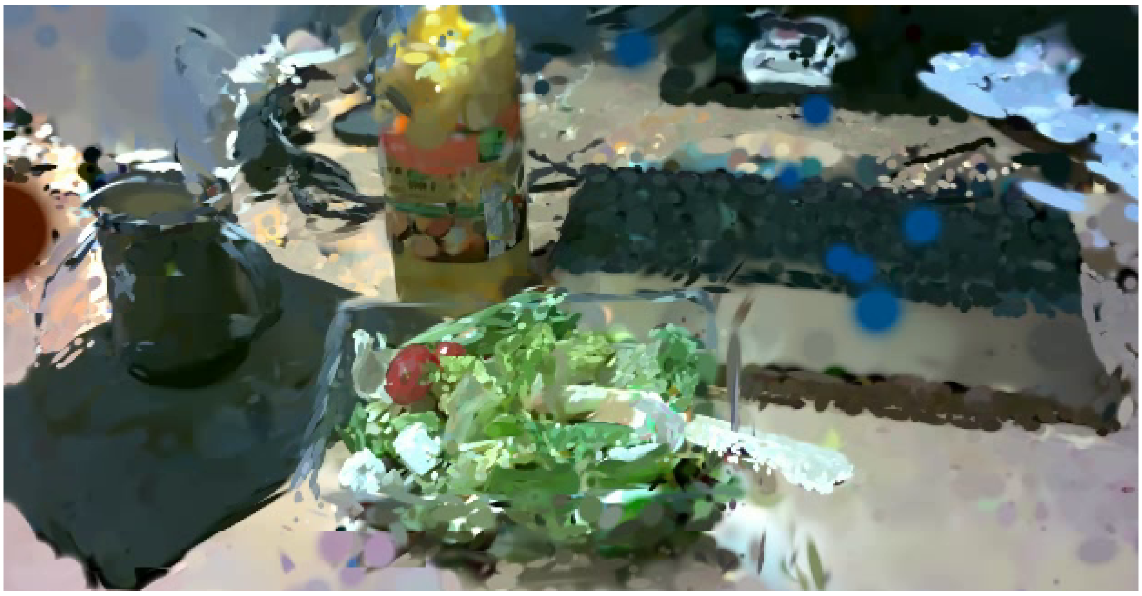


Figure 7.3: Salad

Figure 7.4: L'Inondation à Port-Marly 1876 - Alfred Sisley
(Not IPU!! Nothing beats human rendering)

Bibliography

[1] Hidenobu Matsuki, Riku Murai, Paul H. J. Kelly, and Andrew J. Davison. Gaussian splatting slam, 2023.

[2] Bernhard Kainz and Abhijeet Ghosh. 3rd and 4th year graphics course notes, imperial college london. 2023.

[3] Graphics Compendium. Introduction to computer graphics, 2021. Accessed: 2024-01-09.

[4] Guikun Chen and Wenguan Wang. A survey on 3d gaussian splatting. *arXiv preprint arXiv:2401.03890*, 2024.

[5] NVIDIA. A100 tensor core gpu - `https://images.nvidia.com/aem-dam/Solutions/geforce/ada/ada-lovelace-architecture/nvidia-ada-gpu-science.pdf`. 2020.

[6] Mark Pupilli. Towards neural path tracing in sram, 2023.

[7] Bernhard Kerbl, Georgios Kopanas, Thomas Leimkühler, and George Drettakis. 3d gaussian splatting for real-time radiance field rendering, 2023.

[8] Trevor D. Lamb. Evolution of the eye, 2011.

[9] Image Think. Does vision rule the brain: True or false?, 2012.

[10] Ujjayanta Bhaumik. Introduction to neural radiance field or nerf. *Analytics Vidhya*, 2021.

[11] Ben Mildenhall, Pratul P. Srinivasan, Matthew Tancik, Jonathan T. Barron, Ravi Ramamoorthi, and Ren Ng. Nerf: Representing scenes as neural radiance fields for view synthesis, 2020.

[12] William J. Dally, Stephen W. Keckler, and David B. Kirk. Evolution of the graphics processing unit (gpu). *IEEE Micro*, 41(6):42–51, 2021.

[13] Antoni Rosinol, John J. Leonard, and Luca Carlone. Nerf-slam: Real-time dense monocular slam with neural radiance fields, 2022.

[14] Xin Kong, Shikun Liu, Marwan Taher, and Andrew J. Davison. vmap: Vectorised object mapping for neural field slam, 2023.

[15] Erick P. Herrera-Granda, Juan C. Torres-Cantero, Andrés Rosales, and Diego H. Peluffo-Ordóñez. A comparison of monocular visual slam and visual odometry methods applied to 3d reconstruction. *Applied Sciences*, 13(15), 2023.

[16] Zhaoshuo Li, Thomas Müller, Alex Evans, Russell H Taylor, Mathias Unberath, Ming-Yu Liu, and Chen-Hsuan Lin. Neuralangelo: High-fidelity neural surface reconstruction. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2023.

[17] Florian Sauerbeck, Benjamin Obermeier, Martin Rudolph, and Johannes Betz. Rgb-l: Enhancing indirect visual slam using lidar-based dense depth maps, 2022.

[18] Chi Yan, Delin Qu, Dong Wang, Dan Xu, Zhigang Wang, Bin Zhao, and Xuelong Li. Gs-slam: Dense visual slam with 3d gaussian splatting, 2023.

[19] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness, 2022.

[20] Tomas Akenine-Mller, Eric Haines, and Naty Hoffman. *Real-Time Rendering, Fourth Edition.* A. K. Peters, Ltd., USA, 4th edition, 2018.

[21] 3d design foundations | uxcel, 2022. Accessed: 2024-01-09.

[22] Loren Rhodes. Vertices to fragments, 2024. Accessed: 2024-01-09.

[23] Turner Whitted. An improved illumination model for shaded display. *Commun. ACM*, 23(6):343–349, jun 1980.

[24] bitWise Academy. How to locate images using ray tracing?, 2022. Accessed: 2024-01-09.

[25] Wenzel Jakob, Sébastien Speierer, Nicolas Roussel, and Delio Vicini. DR.JIT, jul 2022.

[26] Richard A. Newcombe, Shahram Izadi, Otmar Hilliges, David Molyneaux, David Kim, Andrew J. Davison, Pushmeet Kohi, Jamie Shotton, Steve Hodges, and Andrew Fitzgibbon. Kinectfusion: Real-time dense surface mapping and tracking. In *2011 10th IEEE International Symposium on Mixed and Augmented Reality*, pages 127–136, 2011.

[27] John McCormac, Ankur Handa, Andrew Davison, and Stefan Leutenegger. Semanticfusion: Dense 3d semantic mapping with convolutional neural networks, 2016.

[28] Sebastian Ruder. An overview of gradient descent optimization algorithms, 2017.

[29] Yiheng Xie, Towaki Takikawa, Shunsuke Saito, Or Litany, Shiqin Yan, Numair Khan, Federico Tombari, James Tompkin, Vincent Sitzmann, and Srinath Sridhar. Neural fields in visual computing and beyond, 2022.

[30] Michael Niemeyer, Lars Mescheder, Michael Oechsle, and Andreas Geiger. Differentiable volumetric rendering: Learning implicit 3d representations without 3d supervision, 2020.

[31] Jeong Joon Park, Peter Florence, Julian Straub, Richard Newcombe, and Steven Lovegrove. Deepsdf: Learning continuous signed distance functions for shape representation, 2019.

[32] Lars Mescheder, Michael Oechsle, Michael Niemeyer, Sebastian Nowozin, and Andreas Geiger. Occupancy networks: Learning 3d reconstruction in function space, 2019.

[33] Jonathan T. Barron, Ben Mildenhall, Dor Verbin, Pratul P. Srinivasan, and Peter Hedman. Mip-nerf 360: Unbounded anti-aliased neural radiance fields, 2022.

[34] Thomas Müller, Alex Evans, Christoph Schied, and Alexander Keller. Instant neural graphics primitives with a multiresolution hash encoding. *ACM Transactions on Graphics*, 41(4):1–15, jul 2022.

[35] Edgar Sucar, Shikun Liu, Joseph Ortiz, and Andrew J. Davison. imap: Implicit mapping and positioning in real-time, 2021.

[36] Chong Zeng, Guojun Chen, Yue Dong, Pieter Peers, Hongzhi Wu, and Xin Tong. Relighting neural radiance fields with shadow and highlight hints. In *Special Interest Group on Computer Graphics and Interactive Techniques Conference Conference Proceedings*, SIGGRAPH '23. ACM, July 2023.

[37] Chengwei Zheng, Wenbin Lin, and Feng Xu. Editablenerf: Editing topologically varying neural radiance fields by key points, 2023.

[38] Cheng Sun, Min Sun, and Hwann-Tzong Chen. Direct voxel grid optimization: Super-fast convergence for radiance fields reconstruction, 2022.

[39] Sara Fridovich-Keil, Giacomo Meanti, Frederik Warburg, Benjamin Recht, and Angjoo Kanazawa. K-planes: Explicit radiance fields in space, time, and appearance, 2023.

[40] Leonid Keselman and Martial Hebert. Approximate differentiable rendering with algebraic surfaces, 2022.

[41] Alex Yu, Sara Fridovich-Keil, Matthew Tancik, Qinhong Chen, Benjamin Recht, and Angjoo Kanazawa. Plenoxels: Radiance fields without neural networks, 2021.

[42] Microsoft. Rasterization rules - win32 apps | microsoft docs, 1 2021.

[43] Cem Cebenoyan. Chapter 28. graphics pipeline performance - nvidia. https://developer.nvidia.com/gpugems/gpugems/part-v-performance-and-practicalities/chapter-28-graphics-pipeline-performance. Accessed: 2024-01-20.

[44] Hanspeter Pfister, Matthias Zwicker, Jeroen van Baar, and Markus Gross. Surfels: surface elements as rendering primitives. In *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '00, page 335–342, USA, 2000. ACM Press/Addison-Wesley Publishing Co.

[45] M. Botsch, A. Hornung, M. Zwicker, and L. Kobbelt. High-quality surface splatting on today's gpus. In *Proceedings Eurographics/IEEE VGTC Symposium Point-Based Graphics, 2005.*, pages 17–141, 2005.

[46] Liu Ren, Hanspeter Pfister, and Matthias Zwicker. Object space ewa surface splatting: A hardware accelerated approach to high quality point rendering. *Computer graphics forum*, 21(3), 2002-09.

[47] Noah Snavely, Steven M. Seitz, and Richard Szeliski. Photo tourism: exploring photo collections in 3d. *ACM Trans. Graph.*, 25(3):835–846, jul 2006.

[48] Ayush Tewari, Justus Thies, Ben Mildenhall, Pratul Srinivasan, Edgar Tretschk, Yifan Wang, Christoph Lassner, Vincent Sitzmann, Ricardo Martin-Brualla, Stephen Lombardi, Tomas Simon, Christian Theobalt, Matthias Niessner, Jonathan T. Barron, Gordon Wetzstein, Michael Zollhoefer, and Vladislav Golyanik. Advances in neural rendering, 2022.

[49] Wikipedia. Massively parallel - wikipedia, 1 2024.

[50] David Tarditi, Sidd Puri, and Jose Oglesby. Accelerator: using data parallelism to program gpus for general-purpose uses. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XII, page 325–335, New York, NY, USA, 2006. Association for Computing Machinery.

[51] Thomas Müller. tiny-cuda-nn. 2021.

[52] Thomas Müller, Fabrice Rousselle, Jan Novák, and Alexander Keller. Real-time neural radiance caching for path tracing. *ACM Transactions on Graphics*, 40(4):1–16, July 2021.

[53] Graphcore. Mk2 c600 ipu - https://docs.graphcore.ai/projects/ipu-programmers-guide/en/latest/index.html. 2022.

[54] Samuli Laine, Tero Karras, and Timo Aila. Megakernels considered harmful: wavefront path tracing on gpus. In *Proceedings of the 5th High-Performance Graphics Conference*, HPG '13, page 137–143, New York, NY, USA, 2013. Association for Computing Machinery.

[55] Joshua Barczak and Holger Gruen. Intel® arc™ graphics developer guide for real-time ray tracing in games - https://www.intel.com/content/www/us/en/developer/articles/guide/real-time-ray-tracing-in-games.html.

[56] Matthew Rusch and Evan Hart. Improve shader performance and in-game frame rates with shader execution reordering - https://developer.nvidia.com/blog/improve-shader-performance-and-in-game-frame-rates-with-shader-execution-reord
2022.

[57] Muhammad Husnain Mubarik, Ramakrishna Kanungo, Tobias Zirr, and Rakesh Kumar. Hardware acceleration of neural graphics, 2023.

[58] Linus Franke, Darius Rückert, Laura Fink, and Marc Stamminger. Trips: Trilinear point splatting for real-time radiance field rendering, 2024.

[59] Sadjad Fouladi, Brennan Shacklett, Fait Poms, Arjun Arora, Alex Ozdemir, Deepti Raghavan, Pat Hanrahan, Kayvon Fatahalian, and Keith Winstein. R2e2: low-latency path tracing of terabyte-scale scenes using thousands of cloud cpus. *ACM Transactions on Graphics*, 41:1–12, 07 2022.

[60] Chi-kwan Chan, Dimitrios Psaltis, and Feryal Özel. Gray: A massively parallel gpu-based code for ray tracing in relativistic spacetimes. *The Astrophysical Journal*, 777(1):13, October 2013.

[61] Simon Niedermayr, Josef Stumpfegger, and Rüdiger Westermann. Compressed 3d gaussian splatting for accelerated novel view synthesis, 2023.

[62] nerfstudio team. gsplat 0.1.2 - https://docs.gsplat.studio/ - https://github.com/nerfstudio-project/gsplat, 2023.

[63] Chaojian Li, Sixu Li, Yang Zhao, Wenbo Zhu, and Yingyan Lin. Rt-nerf: Real-time on-device neural radiance fields towards immersive ar/vr rendering. In *Proceedings of the 41st IEEE/ACM International Conference on Computer-Aided Design*, ICCAD '22, New York, NY, USA, 2022. Association for Computing Machinery.

[64] Yuanfang Wang, Yu Li, Haoyang Zhang, Jun Yu, and Kun Wang. Moth: A hardware accelerator for neural radiance field inference on fpga. In *2023 IEEE 31st Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 227–227, 2023.

[65] Joseph Ortiz, Mark Pupilli, Stefan Leutenegger, and Andrew J. Davison. Bundle adjustment on a graph processor. *CoRR*, abs/2003.03134, 2020.

[66] Jon Kleinberg. The small-world phenomenon: an algorithmic perspective, 2000.

[67] Bingsheng He, Naga K. Govindaraju, Qiong Luo, and Burton Smith. Efficient gather and scatter operations on graphics processors. In *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, SC '07, New York, NY, USA, 2007. Association for Computing Machinery.

[68] Edward G. Chron, Gene Kishinevsky, Brandon Nefcy, and N. V. Patil. Routing algorithms for 2-d mesh network-on-chip architectures. 2007.

[69] L. Benini and G. De Micheli. Networks on chips: a new soc paradigm. *Computer*, 35(1):70–78, 2002.

[70] Christian Reiser, Songyou Peng, Yiyi Liao, and Andreas Geiger. Kilonerf: Speeding up neural radiance fields with thousands of tiny mlps. *CoRR*, abs/2103.13744, 2021.

[71] Vishaal Udandarao, Ameya Prabhu, Adhiraj Ghosh, Yash Sharma, Philip H. S. Torr, Adel Bibi, Samuel Albanie, and Matthias Bethge. No "zero-shot" without exponential data: Pretraining concept frequency determines multimodal model performance, 2024.