Hello everyone! This talk is all about sharing our learnings and work on Neural Super Sampling for Mobile.

Before diving into the technical section the talk, I'd like to share our motivation for working on Neural Graphics for Mobile, which really stemmed from some key observations we've made in the space.

Neural Graphics for Mobile

- **Observations**
  - Trend towards neural workloads tightly coupled to graphics, e.g., NVIDIA DLSS
  - Mobile SoCs ➔ Increasing neural acceleration, inaccessible for graphics
- **Expectations**
  - Impactful neural graphics techniques advance to mobile
  - Future Mobile SoCs will have neural acceleration for graphics

arm

The first observation is likely one you have made yourselves, that is, Neural workloads are becoming a prevailing research topic in our field of graphics.

We believe their opportunity for impact cannot be understated, and in some cases, such as NVIDIA's DLSS technology, they are already transformative in production titles today.

At the same time, in the mobile space, we are seeing chips shipping with an unprecedented and growing level of silicon area dedicated to accelerating neural workloads. Only, the path to harness this acceleration today, in the context of a graphics application, is not an easy one. There's no unified API, there's a lack of shared resources, and accelerator dispatches can have incredibly high latency.

Our expectation though, is that any technology that is transformative and impactful for graphics, is so in equal measures for desktop and mobile. So, it's more than reasonable to expect that future mobile SoC vendors will align to enable neural graphics for mobile.

But what does that mean?

Neural Graphics for Mobile

Neural Graphics Tenets:

- Sustainable high-throughput acceleration (>10 TOPs/Watt)
- Part of the GPU ➔ Low latency, common resources…
- Unified graphics API

arm

Our position, is that neural acceleration for graphics requires (at least) three core ingredients:

Firstly, sustainable high-throughput acceleration, which we can translate to at least 10 TOPs of compute per watt of power spent.
The premise here is that 10TOPs is a good performance point for graphics networks, such as neural super sampling, whilst being an achievable extrapolation from what phones can do today.

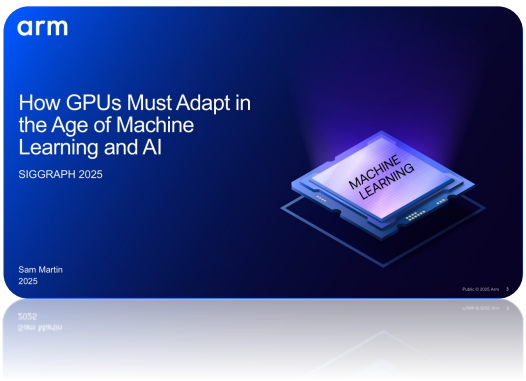
The 1-watt of power draw is necessary to stay within the TDP envelope of a mobile operating for extended periods.

Secondly, this acceleration either be, or behave as though it's part of the GPU, this means low-latency dispatches, full synchronization and common resources, such as shared memory.

Thirdly, this will require a unified graphics API, that can express workloads as combined neural and graphics pipelines.

Neural Accelerators Coming to Arm GPUs 2026

- NPU-class HW embedded in the GPU
  - High efficiency, high throughput
  - Same memory and work scheduling control systems as the rest of the shader core
- ML acceleration in Vulkan
  - Whole-tensor ISA for ML
  - Keep pipeline compile-dispatch approach
- Learn more in Sam Martin's talk!



arm

To address these tenets, we're taken two core actions:

Firstly, in our 2026 Arm GPUs, we are adding neural accelerators. Think NPU-class HW deeply embedded within the shader cores, enabling high efficiency high throughput ML compute. Sharing the same memory and work scheduling systems as the rest of the shader core.

And secondly, we have published ML extensions for Vulkan, which brings a unified API for dispatching work to the neural accelerators. This has added whole-tensor ISA added to SPIRV for describing graph pipelines. Alongside a familiar pipeline compile-dispatch approach for ML workloads.

Relevant links to ML Acceleration in Vulkan:

VK_ARM_tensors: https://github.com/KhronosGroup/Vulkan-Docs/blob/5d386163f25cca10d2af7be2bbea07d1e6fb52ba/chapters/VK_ARM_tensors/tensorops.adoc

VK_data_graphs: https://github.com/KhronosGroup/Vulkan-Docs/blob/5d386163f25cca10d2af7be2bbea07d1e6fb52ba/chapters/VK_ARM_data_graph/graphs.adoc

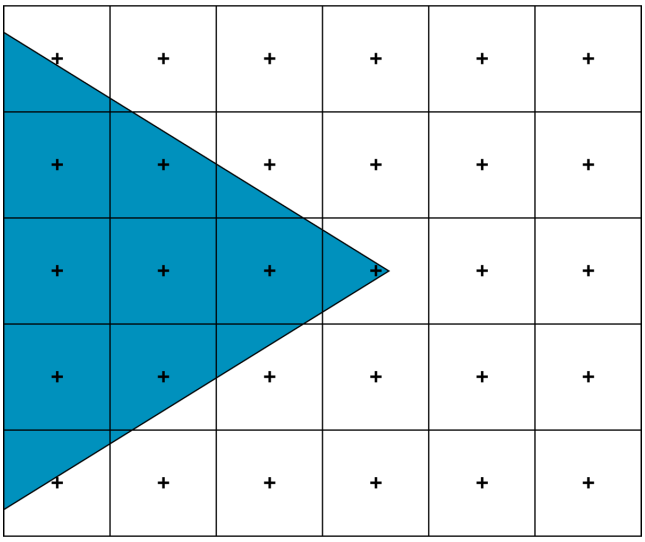
TOSA: https://www.mlplatform.org/tosa/tosa_spec.html

Recap: Super Sampling

arm

To put everyone on the same page, we'll start with a recap of the problem that super sampling is addressing.

Recap: Super Sampling

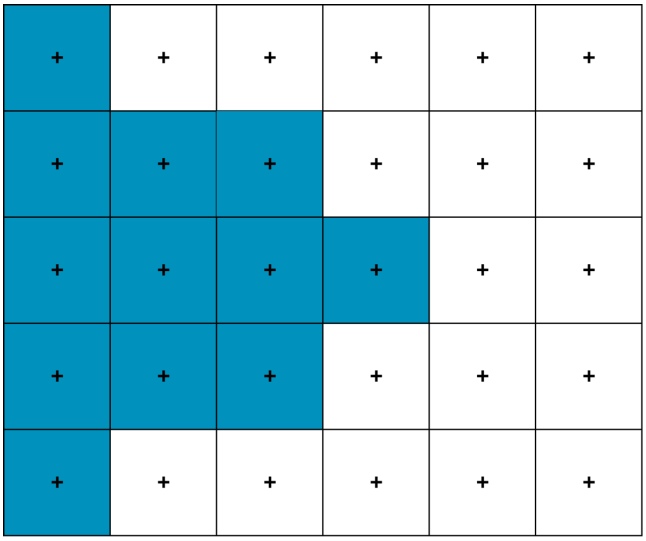


arm

As with most computer graphics stories, ours begins with triangles…

We want to take some continuous geometry, and discretize it onto a grid of pixels.

Recap: Super Sampling

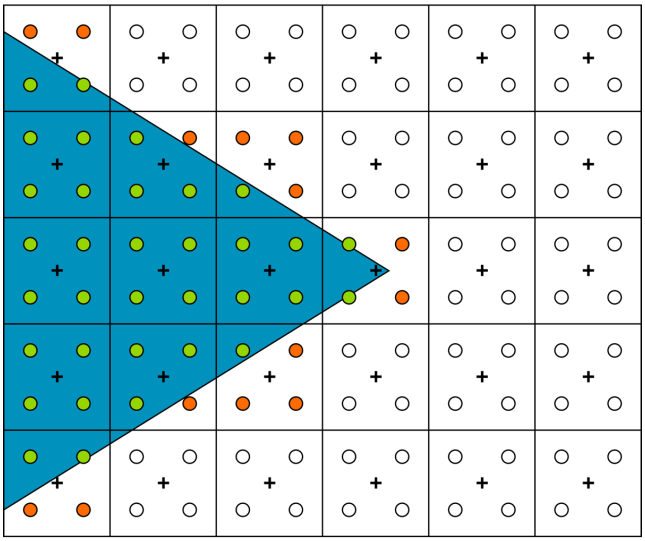


arm

Although when we do this, under sampling is often a problem.

This manifests as aliasing or “jaggy edges” forming stair-stepped edges.

Recap: Super Sampling

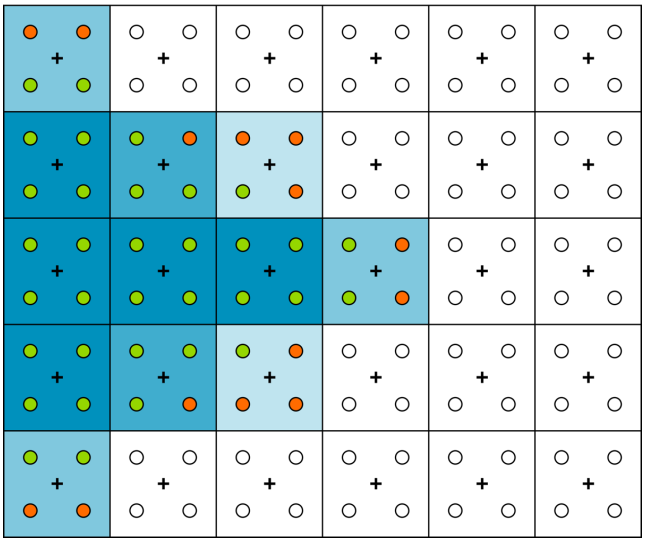


arm

The common approach to solve this is to take more samples per-pixel.

Here, we sample four evenly distributed positions within each pixel, green dots are sampling our triangle, red dots are sampling adjacent geometry.

Recap: Super Sampling

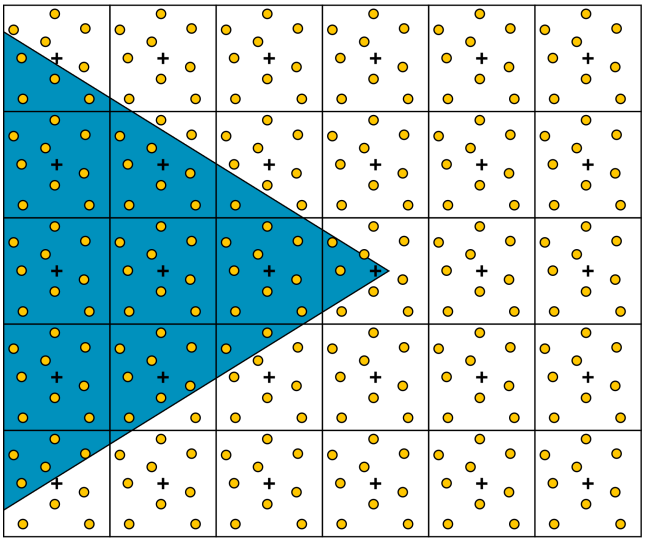


arm

By averaging the x4 samples, you integrate the color across the pixel

Zooming out at the macro level, this reduces aliasing removing jaggies and giving nice smooth transitions across the border of geometry.

Recap: Super Sampling

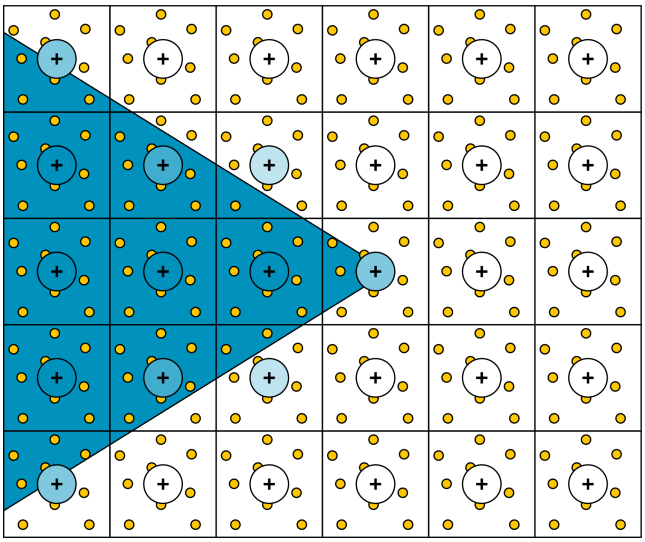


arm

In practice, for the highest quality, it’s desirable to take more than just 4 samples per pixel.

Usually distributed evenly in a pseudo-random “blue noise” pattern across the pixel area.

Recap: Super Sampling



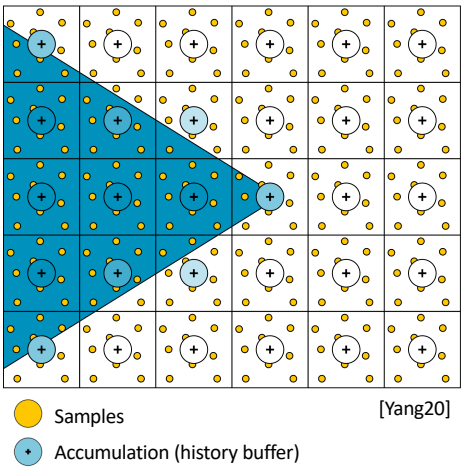
arm

Averaging all these samples gives gives an integral of the pixel and resolves aliasing in highest possible quality.

Although, doing this is highly undesirable for real-time graphics developers, as can involve shading an order of magnitude more samples, than the number of displayed pixels.

Recap: Temporal Super Sampling

- Super Sampling ➔ amortize cost temporally
  - Apply sub-pixel offset to ensure novel samples, called “jitter”
  - Shade 1 new sample per-pixel
  - Accumulate new sample into “history”
- There’s a lot to like...
  - Addresses all types of aliasing
  - Cost-effective for deferred rendering
  - Extensible to upscaling
- … but comes with challenges



[Yang20]

arm

Which is where temporal super sampling comes in:

In spirit, it is just like regular super sampling, where we resolve aliasing by accumulating multiple samples per-pixel, only that work is amortized across the time dimension.

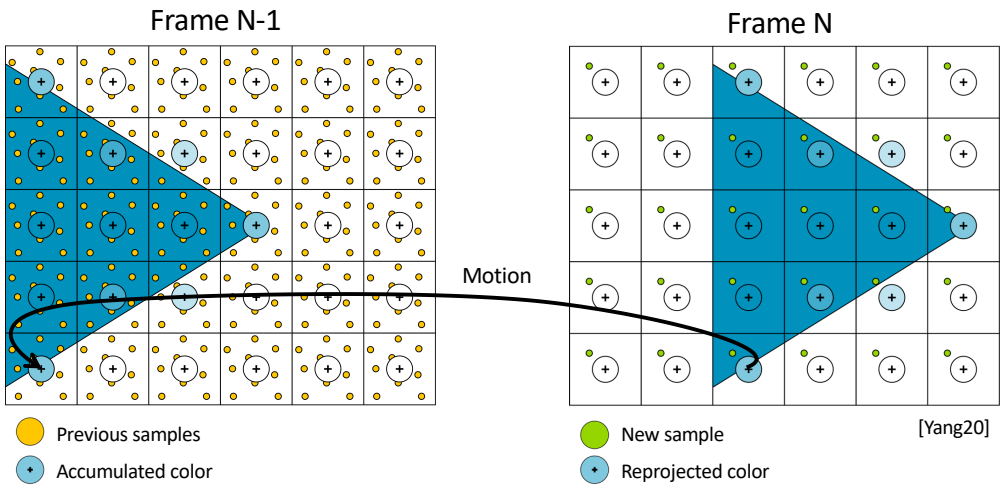
This means that for each new frame we render:

- Apply sub-pixel jitter, so that each new frame receives novel samples, even when the camera is static
- We shade a single sample
- Accumulate this sample into a history buffer, reconstructing the sub-pixel detail

This approach has grown to significant popularity in the last decade, as it addresses all types of aliasing (i.e., shading too, not just around geometric boundaries like MSAA), it's cost-effective with a deterministic runtime, and can be leveraged for upscaling, by accumulating into a larger output history.

But... it does come with its challenges.

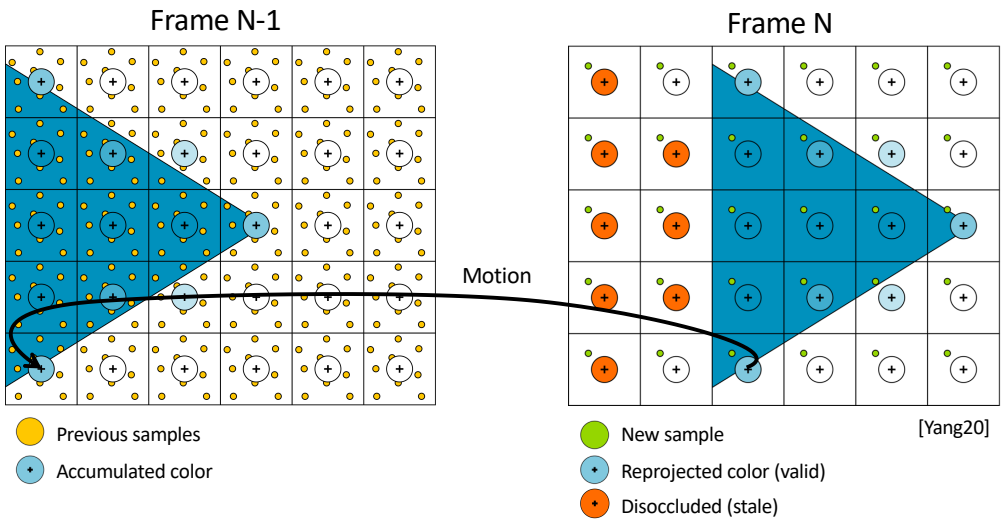
Recap: Temporal Super Sampling



arm

Accurate dense motion vectors are essential to reproject previous temporal accumulation into the correct spatial position in the current frame of reference.

Recap: Temporal Super Sampling



arm

But because of this, the temporally integrated history can frequently become invalid, e.g., due to disocclusions. So, the approach must be robust and capable of detecting such events and resetting the buffer, to avoid artefacts

Challenges: Ghosting

arm

If you fail to reset the history ➔ you get ghosting!

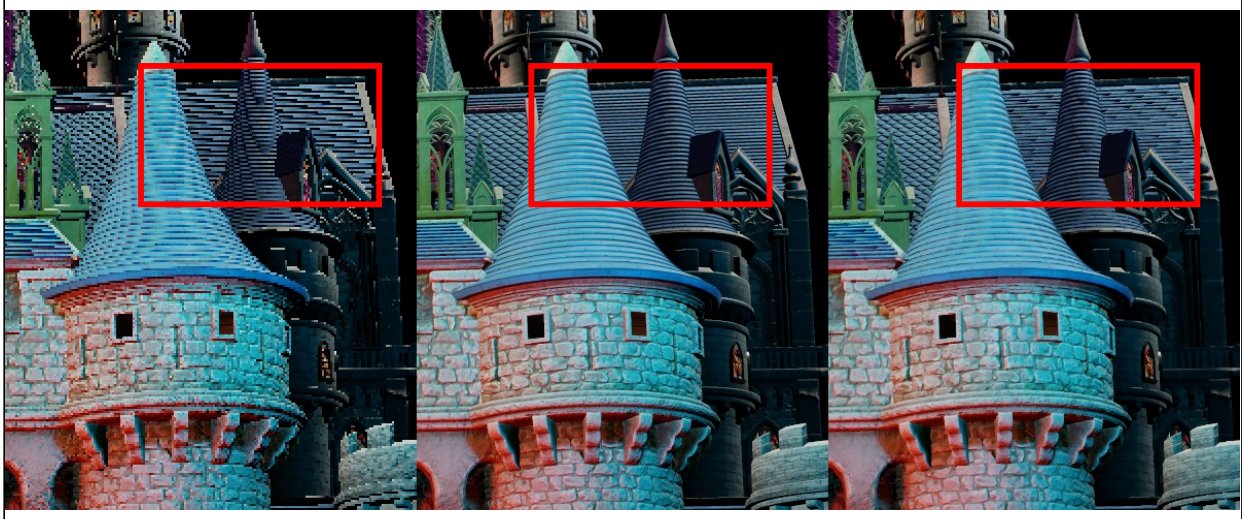
Challenges: Rectification Bias

Low Resolution Input | History | Rectification Bias

arm

But if you reset the history too aggressively, you can induce bias towards the current sample, re-introducing aliasing.

Challenges: Rectification Bias



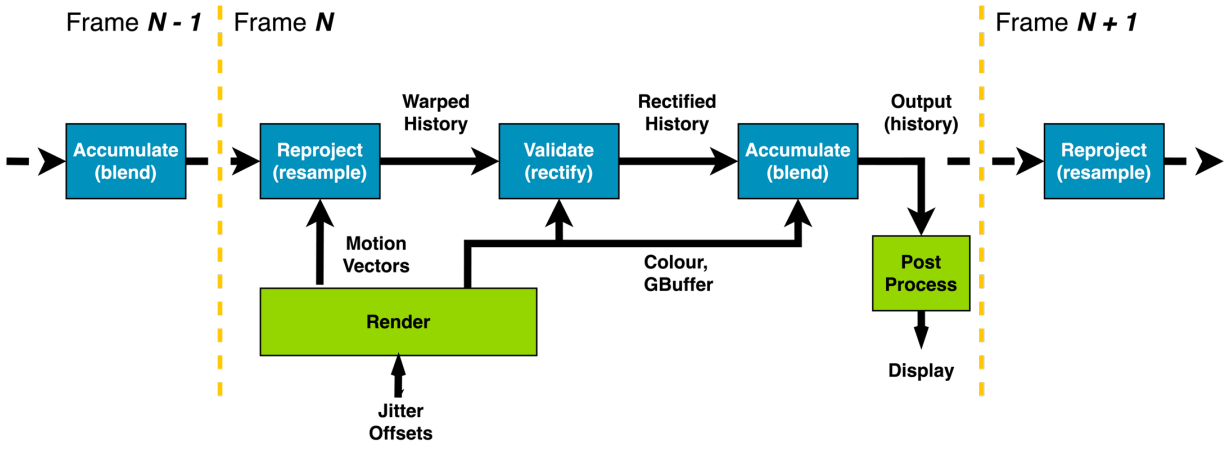
Low Resolution Input | History | Rectification Bias Example

arm

This can be seen when applying an AABB clamp on a nicely accumulated history, with a very aliased input, aliasing returns to the output frame.

Recap: Temporal Super Sampling

[Yang20]



arm

To address these issues, temporal super sampling typically follows a 3-step approach, of:

Reprojection ➔ resampling the history into the current frame of reference

Validation ➔ removal of artefacts that occur during the reprojection / updating the history due to shading changes,

Accumulation ➔ where we blend some amount of the current sample into the history to enable the super sampling.

The pipeline and overall image quality is heavily dependent on the Validation stage, which ascertains whether the history is valid or not, and there is all manor of clever heuristics that do this.

Initial work outline

- Explore the advantages of neural approaches for TSS...
  … whilst being mindful of mobile constraints:
  - **Quantization** ➔ Neural Accelerator favors INT8
  - **Memory bandwidth** ➔ shared across CPU/GPU/Neural Accelerator
  - **Power** ➔ stay within ~10TOPs/Watt budget

arm

Temporal Super Sampling (TSS) is no doubt a powerful technique.

However, as many have found, turning the dials to get the best perceptual quality out of the technique in a specific engine can be time consuming.

Whilst neural approaches are often touted for offering the promise of better image quality, they also offer a means for automated re-tuning of heuristics, learning them automatically for specific content, which can be advantageous too.

From the outset of this work aimed to explore neural approaches for TSS, whilst being mindful of the constraints outlined for mobile, namely:

Quantization ➔ to get the best performance out of mobile accelerators INT8 is the precision of choice.

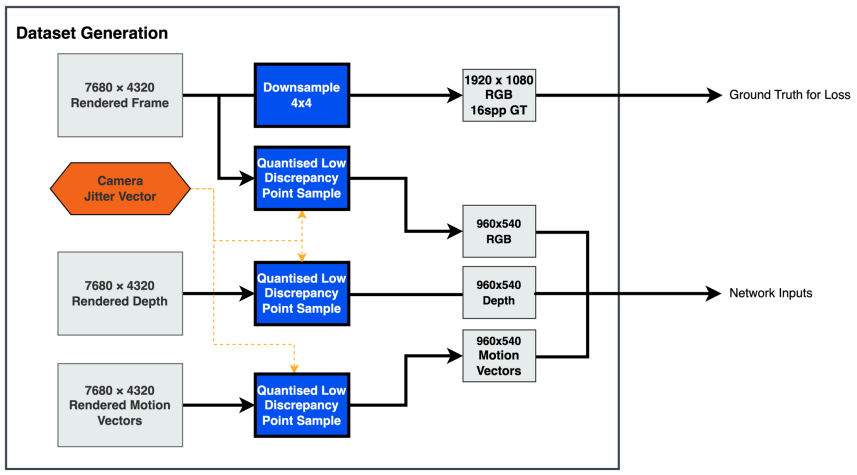
Memory Bandwidth ➔ shared across the system, correlated with power, so best kept at a minimum when possible.

Power ➔ in the scope of this work, this meant not exceeding the position we outlined of the reasonable future 10TOPs acceleration.

Dataset Generation

arm

Dataset Generation Overview



arm

We'll start by giving a brief overview of our dataset generation process for this work.

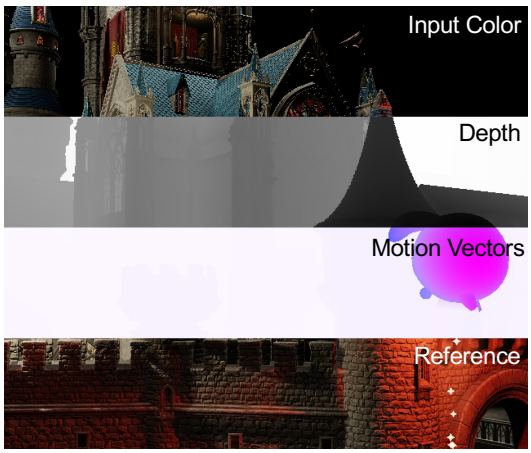
A single 7680 × 4320 render generates both input and ground-truth data, so geometry, lighting, and all screen-space effects stay aligned, and no second deterministic pass is required.

Ground truth frames can be created by applying a 4x4 box/average filter on the high-resolution render to produce an anti-aliased ground truth.

Jittered inputs are formed by point-sampling the high-resolution data with a camera-jitter vector drawn from a quantised Halton sequence, using the same sample for RGB, depth, and motion.

Dataset Constitution

- Target super sampling 540p ➔ 1080p
- Capture each sequence @ 30fps:
  - Jittered inputs (1 spp): color, depth, motion
  - Ground truth color (16 spp)
  - Metadata (jitter vectors, camera matrices, etc.)
- Training Sequence length: ~100 frames (~3 s); adequate for temporal accumulation
- Testing Sequence length: >200 frames (>6s); suitable to watch in playback



arm

For this work, we've targeted the 2x2 super sampling use case, capturing low-res aliased inputs at 540p and super sampling them to 1080p

We've built plugins to popular game engines such as unity and unreal engine to capture our training datasets.

We typically capture the datasets at 30fps, where each frame constitutes a dump of all the typical inputs to a super sampling algorithm, this means:

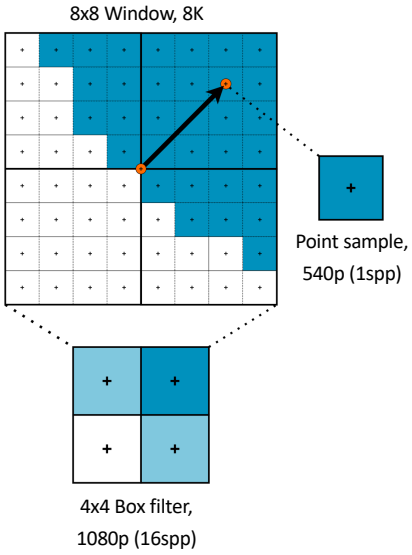
- Low-res jittered textures like color, depth, and motion
- Ground truth color, which is 16spp
- Metadata from the engine – notably the camera jitter vector, view projection matrices, and depth planes

For training content, we typically capture ~3 second long clips, which are adequate to view back and allow sample accumulation to the maximum 64spp.

For testing, we typically capture sequences double the length which is better to

A Note on Ground Truth

- Found consistency between ground truth and input to be crucial
- Avoid two-pass renders by using one 8K frame for both; ensures ground truth is sum of inputs
- Mip bias +1 applied; nice compromise between ground truth fidelity and realistic runtime input
- **Ground truth:** 4x4 box filter to 1080p (16spp), tone-mapped before averaging [Karis14]
- **Inputs:** point-sample within 8 × 8 window to 540p, using discretized Halton(2,3) positions



arm

Consistency between the ground truth and the inputs is vital – we're training a reconstruction algorithm, so ensuring the input samples accurately integrate to form the ground truth is important to avoid unnecessary error in the loss calculation.

In practice found that this can be difficult to achieve in 2 separate render passes, e.g., engine determinism is required, some effects e.g., SSAO can be resolution dependent.

Instead, we generate ground truth and inputs in a single pass, by rendering a single 8k image, and constructing the ground truth and inputs from this.

To get an effective mip bias of -1 for the decimated input, we would need to set a mip bias of +2 when rendering at 8K. We found this would give a 1080p downsampled AA GT would then be too blurry. So, we bumped it by +1 instead when rendering the 8K image to give a good compromise between GT quality and decimated input bias (-2).

The mip bias would typically be calculated as Log2(render res/target res). for a 2x upscale from 540 -> 1080 that's a bias of -1.

We construct the ground truth by applying a box filter, giving 1080p 16spp.

We construct the input using a decimating point sample from each 8x8 tile, giving

540p low-resolution aliased inputs. The decimated position is driven using a quantized Halton sequence.

Dataset Summary

- Split across Train / Test / Validation:
  - Train ➔ ~30K frames
  - Test ➔ ~16K frames
  - Validation ➔ ~8k frames
- Dataset diversity is more important than dataset size, e.g.,
  - Static scenes
  - Thin high frequency features
  - Missing motion vectors (particles)
  - Fast motion
- Looking at where existing approaches struggle helps inform where to focus

arm

We're now working with roughly 50K frames captured across a wide variety of sequences.

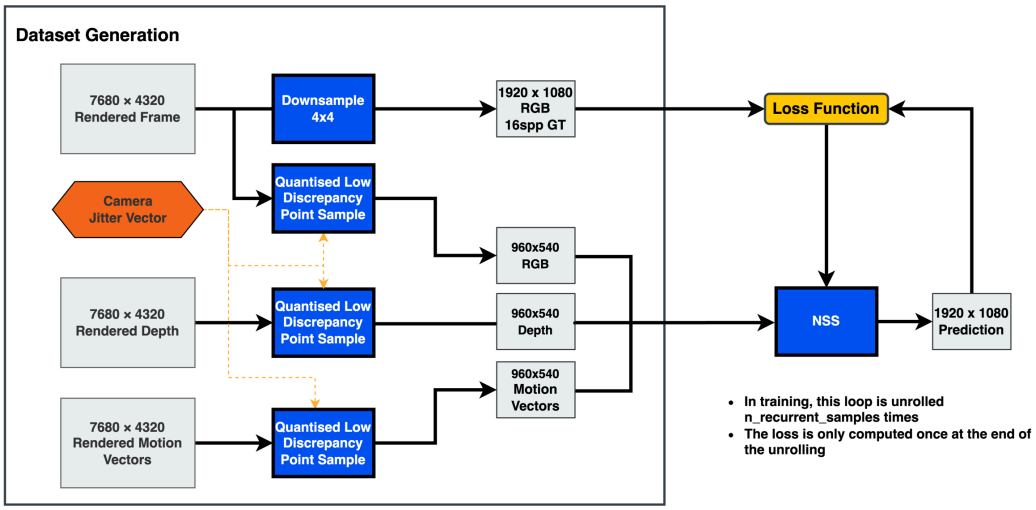
Key observation - We've found tailor making sequences that capture specific content we care about, e.g., thin flickering features, static scene/camera, etc. is much more effective than just arbitrarily capturing lots of content.

Reviewing other temporal super sampling algorithms and where they fail can give a good impression on the type of content that is important to capture and highlight in the training dataset.

Training Methodology

arm

Training Methodology Overview



arm

Next, before diving into the approach we use for NSS, let’s cover the methodology that we use to train a model to solve this problem.

Training Strategy

- Networks authored in PyTorch; Slang is used for authoring compute passes
- ExecuTorch is used for Quantization Aware Training (QAT)
- Typical approach followed for training:
  - Adam optimizer
  - Augmentations: random flip, rotate, shuffling, etc.,
  - Cosine annealing learning rate schedule, with warmup
- As NSS is a temporal algorithm, we train recurrently…

arm

The network is authored and trained in PyTorch, we use Slang, and it's automatic differentiation, to author the pre and post processing passes – mostly as it makes for easier deployment of these passes after training.

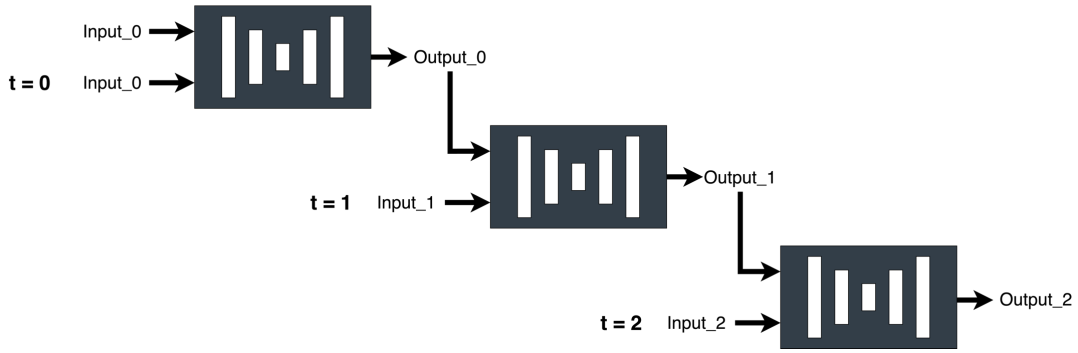
The ExecuTorch library is used to insert the quantization nodes for Quantization Aware Training and exporting the deployment artefacts.

We're using very industry standard approach w.r.t, training methodology, Adam Optimizer, typical augmentations and learning rate schedules.

As NSS is a temporal algorithm we are training it recurrently…

Recurrent Training Loop

- Trained in a recurrent loop ➔ multiple forward passes per backward pass
- Allows for gradients to back-propagate through time
- Loop length between [8, 32] seems to work well; we use 16



arm

Recurrently training just means that before each backward pass to update the model weights, we perform multiple forward passes.

This allows the gradients to back-propagate through time, and the network to learn temporal sample convergence.

The number of recurrent steps you take before doing a backward pass is a tunable variable – trade-off between training time and quality (usually temporal stability) – we typically perform 16 inferences before each backward pass.

Loss Function(s)

- Loss calculated in a tone-mapped domain; errors better align to perception
- We employ a temporally-weighted (w_t) spatiotemporal loss:

$$L = \frac{1}{T}\sum_{t=1}^{T}(1 - w_t) \cdot L_T\ (t) + w_t \cdot L_S\ (t)$$

- With a combined L_1 + perceptual ($LPIPS$) spatial-term:

$$L_S = L_1\left(\hat{Y}_t, Y_t\right) + \rho \cdot LPIPS(\hat{Y}_t, Y_t)$$

- And an exponentially weighted temporal term on reprojected past frames:

$$L_T = e^{\alpha(|\hat{Y}_t - W(\hat{Y}_{t-1})| - |Y_t - W(Y_t)|)} - 1$$

- Large trade-off space to explore between fidelity and temporal stability

arm

We use a spatiotemporal loss function to train the network – where all losses are computed in a perceptual tone-mapped domain.

Each of the outputs within the recurrent window receive a different spatiotemporal weighting, we increase the spatial loss weighting over time, as the output should be more representative of a converged ground truth frame later in the window.

For the spatial loss, which encourages fidelity, we employ a combination of a typical L1 mean absolute error across prediction and ground truth pixels, in addition to the LPIPS metric, which is a neural network-based image quality metric that can be used as a perceptual loss, and provides closer alignment to human perception.

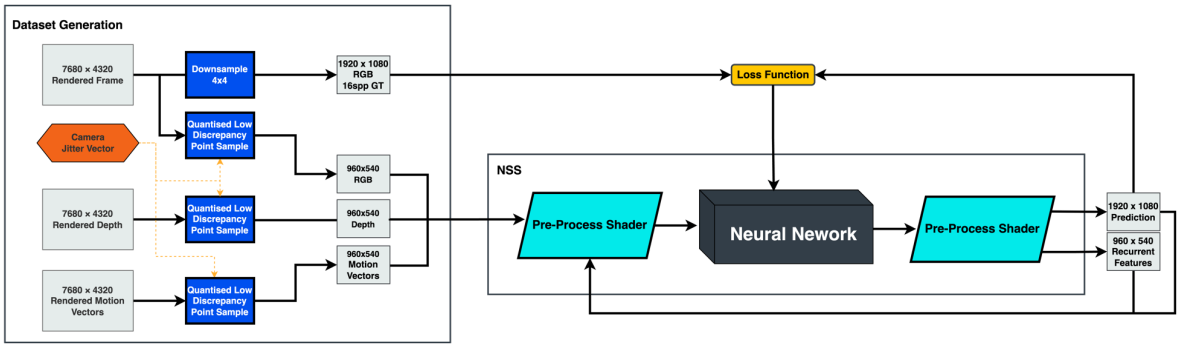
For the temporal term, we employ a L1 loss between the temporal residual of the prediction and truth frames, where the temporal residual is the difference between frame N and N-1 after reprojection, we then apply an exponential function to this error, which we've found helps to penalize instability more harshly.

Ultimately, often find that optimal temporal stability, such as reduced flickering and best spatial fidelity, such as no ghosting are contradicting goals – where tuning the loss is a process of finding a desirable balance between them.

Learnings on Approach

arm

Approach Overview

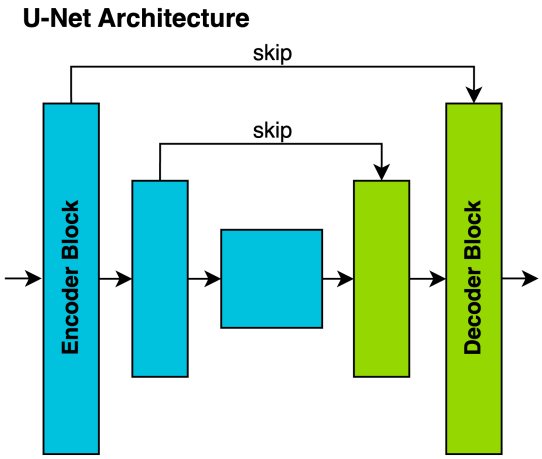


arm

With the dataset and training methodology explained, we can now speak to the underlying network architecture, the approaches we have experimented with and the design we are currently using.

Backbone

- Adopted UNet architecture
- Network inputs:
  - Color (1spp)
  - History color
  - Depth difference
  - Motion vectors
  - Temporal feedback
    (more on that later…)
- Network output(s) depend on the approach...



arm

For network architecture, we adopted the UNet backbone, which is commonly employed in imaging tasks.

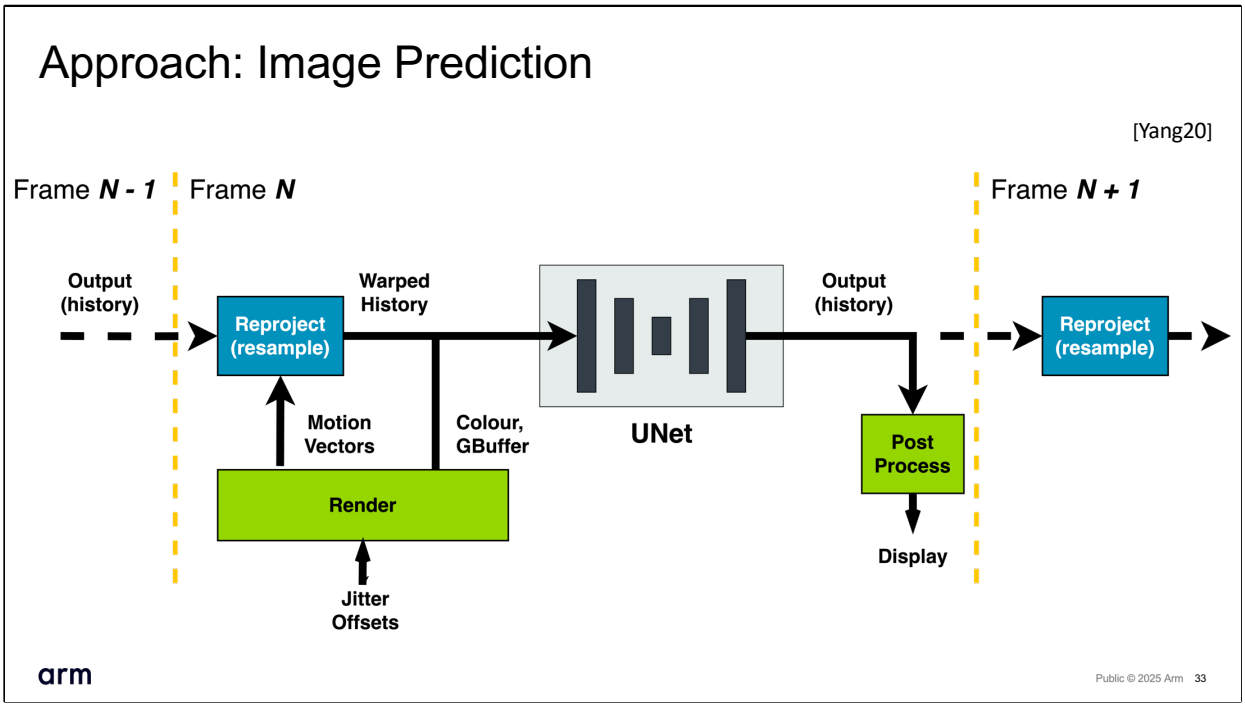
This architecture consists of encoder blocks which consist of multiple convolution operations and a downsampling (such as max pooling),
and decoder blocks which consists of more convolution operations and an upsampling operator (such as nearest neighbor).

In our work, we've favored concatenating the skip connections to bring back high-frequency detail in feature space.

The architecture has a nice property of quickly reducing the resolution of the input tensors, which allows for more favorable parameter-operations ratio.

The network inputs we used consisted of: ... (listed in the above slide)

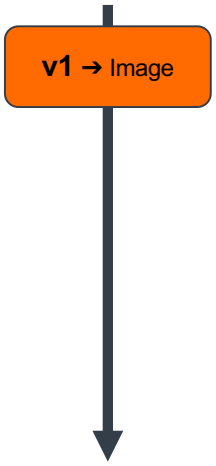
The output we asked the network to produce was the avenue we have spent most time exploring, and really forms the discovery arc of this talk.



Our first approach, was perhaps the most obvious starting point, which was to replace the entire TSS operation with a Unet, asking the network to directly predict the high resolution, anti-aliased output image.

Approach: Image Prediction

- **Pros**
  - Simple to implement
  - Unconstrained to learn the entire TSS function
  - Capable of hallucinating new details from priors



arm

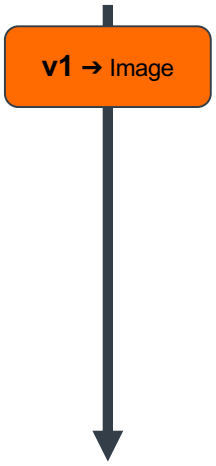
This approach has the benefit of being quick to get off the ground and running inside a deep-learning framework – it's a well trodden path in super resolution.

And following this approach, it's possible to train the network in a generative-style, whereby it can hallucinate details in the output frame, through priors in the training dataset.

the learning in this approach is "unconstrained" in the sense the entire temporal super sampling function is learnt by the network, which gives a high degree of freedom over the output.

Approach: Image Prediction

- **Pros**
  - Simple to implement
  - Unconstrained to learn the entire TSS function
  - Capable of hallucinating new details from priors
- **Cons**
  - Artefacts: over-sharpening, color-shift and ghosting
  - Slow to learn and requires large dataset for generalization
  - Poor quality under INT8 quantization, especially for HDR

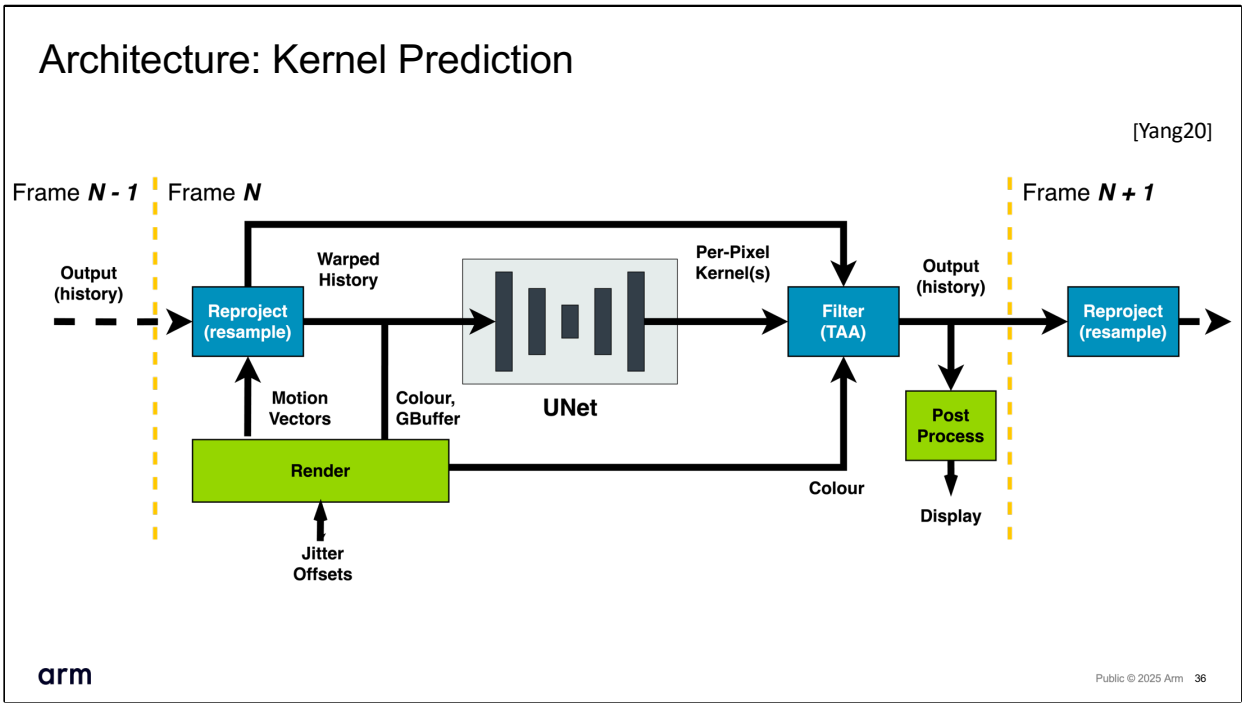


arm

This can be an advantage, but also leads on to the difficulties we experienced, which were ultimately artefacts which were difficult to resolve, such as: over-sharpening, colour-shift and ghosting. In this instance, the only mitigation is through loss function and network architecture changes, balancing the loss term to address this is not impossible, but also there were other issues.

Secondly, whilst this can be worked around, it's worth noting that image-to-image networks do require a large and diverse training dataset to generalize, and can suffer from overfitting greatly if given a smaller dataset.

Lastly, we observe poor quality under quantization, for this approach it became impractical to quantize the network down to 8-bit, without banding artefacts, and overall loss of quality, in the output images. Ultimately, for this work, the poor behaviors under quantization motivated the search for a different approach.



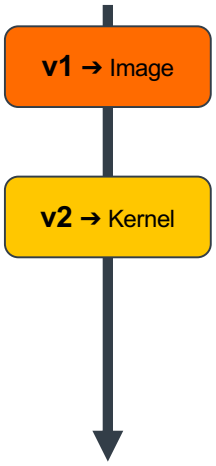
This led us towards Kernel Predication approach... this approach is popular in deep learning for graphics today.

In this approach the strategy is using the network to predict a series of kernel coefficients, to filter the input frames, as opposed to directly outputting an image.

The input images are then filtered in a separate pass to the network dispatch, using the estimated kernels, it's common to perform a large number of these filtering stages, estimating several kernels. It is assumed the GPU, rather than the neural acceleration will fulfil the role of this filtering.

Approach: Kernel Prediction

- **Pros**
  - Quantizes well, kernels don't need HDR precision
  - Can prevent color-shift / over-sharpening
  - Trains quickly and generalizes well



arm

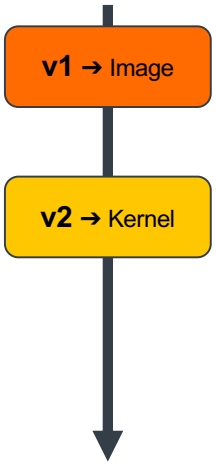
This approach is incredibly robust to quantization, as the kernels themselves do not require such high precision.

Additionally, through a choice of activation functions, it's possible to constrain this approach to avoid either colour shifting or over sharpening.

And lastly, training a model this way is very quick, and largely generalizable, as instead of asking the network to generate a radically new output image, we are asking it to inspect the input image and determine a suitable kernel to reduce aliasing.
We find the results from such networks fail less harshly, than that of an image predicting network, where the failure cases can be substantially incorrect, as opposed to slightly blurry

Approach: Kernel Prediction

- **Pros**
  - Quantizes well, kernels don't need HDR precision
  - Can prevent color-shift / over-sharpening
  - Trains quickly and generalizes well
- **Cons**
  - On Mobile: Significant bandwidth concerns (!)
    e.g., 1080p 3x3 filter ➔ 1080x1920x9 output
  - Significant filtering compute cost



arm

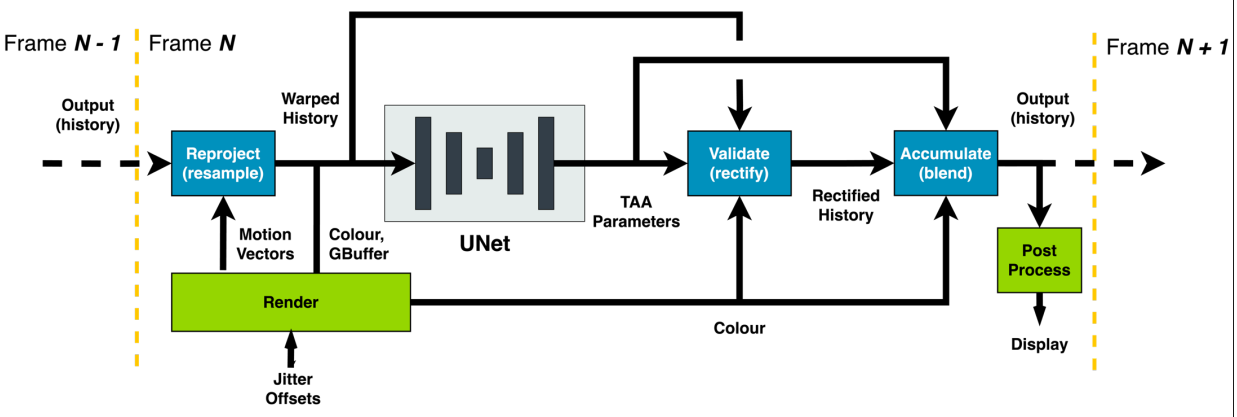
However, our primary observation is memory bandwidth, both writing and reading a 1080x1920x9 sized output buffer is costly in terms of memory bandwidth, which is pressure on a mobile platform.

Additionally, assuming we use the same backbone network, the compute cost becomes expensive, due to the requirement for additional stages after dispatching the network.

In summary, we found kernel prediction networks to be more inline with our desired outputs, but unreasonably costly in terms of memory bandwidth for mobile, when used in their research context.

Approach: Parameter Prediction

[Yang20]



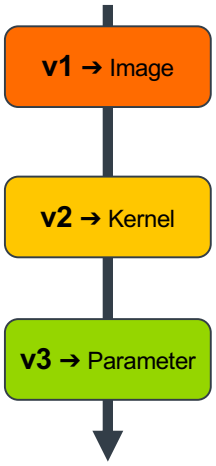
In the pursuit of optimizing Kernel predicting networks, we arrived at parameter prediction networks... Which almost, but not quite, arrive full circle at where we started with temporal super sampling.

The idea here, is instead of estimating many kernels, we estimate a handful of parameters which can be used to drive a pre-determined image processing function, such as those used in temporal super sampling today.

However, instead of hand tuning heuristics for such functions, we allow the network to do this, by estimating per-pixel local parameters which are dependent on both past at current context of the input frames.

Approach: Parameter Prediction

- **Pros**
  - Like KPN ➔ avoid artefacts, quantize & generalize well
  - Lower memory bandwidth, substantially fewer outputs
  - Parameters robust to low resolution prediction
    …then linearly upsampled (more bandwidth saving!)



arm

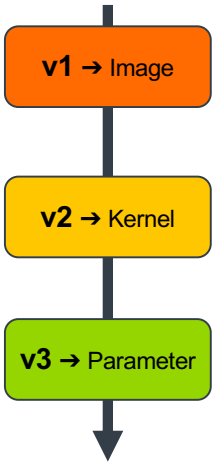
In this formulation, we receive all the benefits of the aforementioned kernel prediction approach.

But advantageously, the bandwidth is substantially lower, as the parameters themselves can be constrained to a lower dimensionality.

In addition to the bandwidth saving from dimensionality, we notice it's also possible to estimate these parameters at lower than output resolution and simply upscale them, without entailing significant image quality penalties.

Approach: Parameter Prediction

- **Pros**
  - Like KPN ➔ avoid artefacts, quantize & generalize well
  - Lower memory bandwidth, substantially fewer outputs
  - Parameters robust to low resolution prediction
    …then linearly upsampled (more bandwidth saving!)
- **Cons**
  - Accuracy upper-bound set by the TSS implementation
  - Exploring parametrization can be time consuming



arm

However, whilst this approach is without a doubt more efficient than kernel prediction networks, it is more constrained and final quality is determined by the choice of image processing functions chosen, which in our case was a heavily modified implementation of temporal super sampling.

Exploring parameterization of this approach can be time consuming also, as it is dependent on the TSS implementation that you start with.

With that said, in this talk, I'd like to give some options that we have explored that others might find applicable also.

Options: Parameters

- **Accumulation (α)**
  - Control rate of convergence

$$Y_t = \alpha x_t + (1 - \alpha) * Y_{t-1}$$

arm

Firstly, and perhaps most obviously, we can delegate the accumulation of new samples into the history buffer to the network.

This is substantially better than using a fixed accumulation term, as convergence can now be sped up, such as in the case of a reset event, or slowed down, such as static camera instance, by the network.

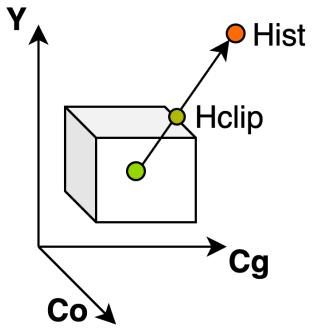
Ultimately this enables optimally learnt convergence.

Options: Parameters

- **Accumulation (α)**
  - Control rate of convergence
- **Rectification (θ)**
  - History clipping point
  - Expressivity to avoid thin-feature flicker

$$Y_t = \alpha x_t + (1 - \alpha) * Y_{t-1}$$

$$Y_t = \theta H_{clip} + (1 - \theta) * Hist$$



arm

Secondly, we can delegate control of when, or when not, the history buffer is rectified.

This enables the network to respond to disocclusions and shading changes, whilst giving it the ability to avoid rectification bias when sub-pixel features are not present within the current sample.

In general, we found this parameter change to have the biggest overall impact in the quality of our network's outputs.

But you must be mindful that the final output quality is a function of the clipping function that is employed for rectification.

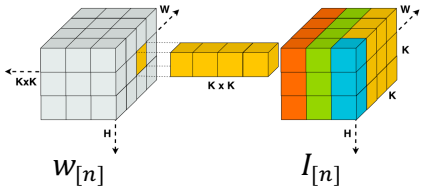
Options: Parameters

- **Accumulation (α)**
  - Control rate of convergence
- **Rectification (θ)**
  - History clipping point
  - Expressivity to avoid thin-feature flicker
- **Targeted Kernel Prediction (w_n)**
  - Estimate single-set of per-pixel kernels
  - Domain-specific; not a general KPN
  - Reduces aliasing and rectification bias

$$Y_t = \alpha x_t + (1 - \alpha) * Y_{t-1}$$

$$Y_t = \theta H_{clip} + (1 - \theta) * Hist$$

$$Y_{[n]} = \sum_{k=0}^{N} w_{[n][k]} \cdot I_{[n+k]}$$



$w_{[n]}$ $I_{[n]}$

arm

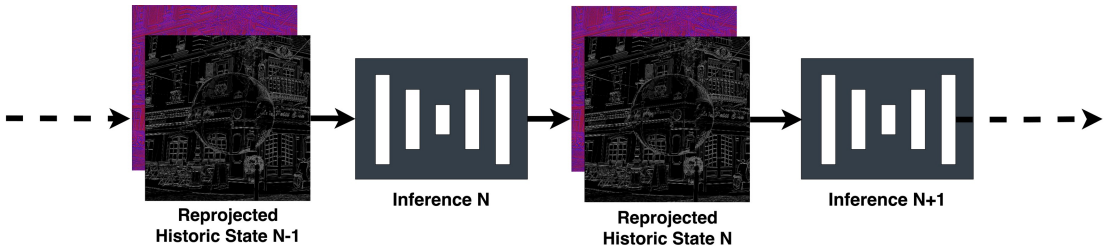
Finally, we estimate a single set of kernel weights to filter the aliased low-resolution image.

This is similar to state-of-the-art kernel prediction methods, but instead of estimating many kernels and using a generic filtering architecture. We just estimate a single set that is targeted to solve the singular problem of reducing aliasing in the low-resolution aliased input frame as we upscale it.

This helps to reduce rectification bias in the instances where the history goes stale and the network chooses to flush the history.

Temporal Feedback

- Temporal stability is paramount ➔ model must make congruent decisions
- Solution: expose historic state information to the model
- Options that have stuck:
  - Hidden features: propagate internal feature maps over time
  - Luma derivative: accentuate flickering fine details, provide as an input feature



arm

Finally, I'd like to briefly touch on temporal stability, which is of great importance to any temporal technique.

Ultimately, it is a necessity to expose temporal state information to the network, in order for the heuristics that it produces to be stable in the temporal dimension.

Again, there are many options to explore here, but we have currently settled on a couple which bring a good impact on the stability of the solution:

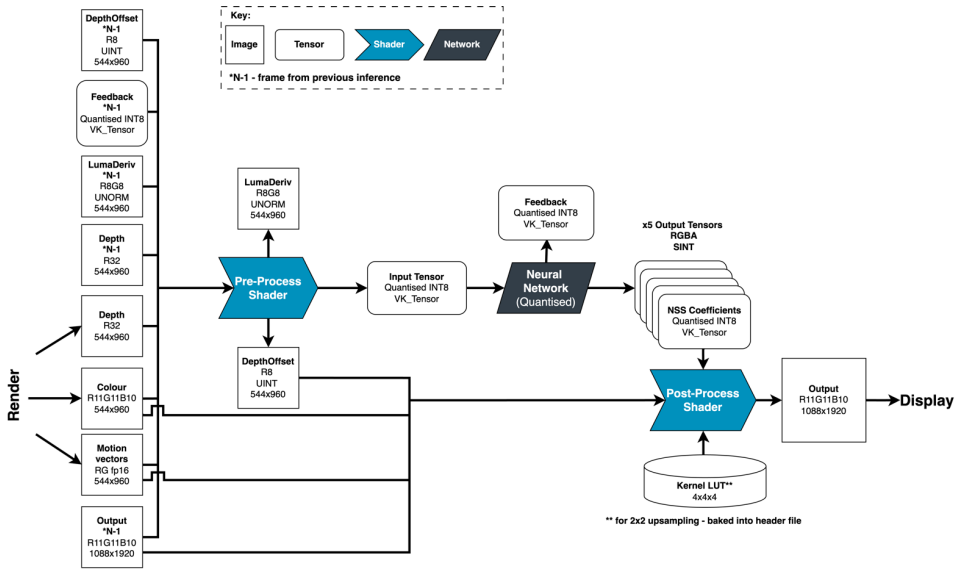
Firstly, a hidden state of temporal features, gives the network the freedom to learn it's own state that it can feed forward to itself over time.

Secondly, we found, especially for high-contrast thin flickering features, calculating the temporal variance of the luma, aided in the networks decision making for such features, reducing flicker that was heavily biased by jitter.

In summary, again there's limitless potential on what can be provided to the network to aid it's temporal decision making, but we found these approaches addressed specific issues that we were concerned with.

Current Approach

Mapping to ML Extensions for Vulkan



We map the approach to the new ML extensions to Vulkan as two compute dispatches, that sandwich a graph dispatch which runs the network.

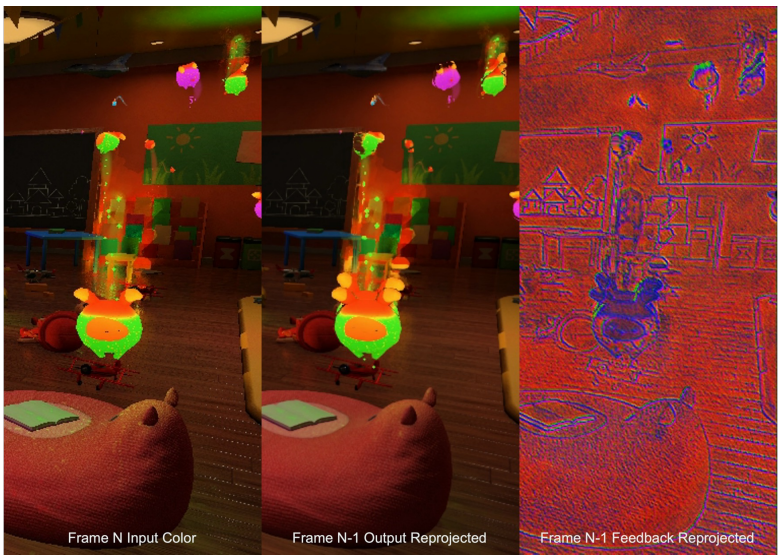
I'll now go into a bit more detail as to what each pass is responsible for.

For reference:

- The Vulkan data graph extension: https://registry.khronos.org/vulkan/specs/latest/man/html/VK_ARM_data_graph.html
- There is a quick start guide available: https://learn.arm.com/learning-paths/mobile-graphics-and-gaming/vulkan-ml-sample/

Pre Process – Computes Network Inputs

- Inputs derived from the current and previous frame buffers
- Frame N-1 buffers reprojected using motion vectors, for pixel alignment with Frame N
- Inputs comprise of:
  - Color (1spp)
  - History (previous output)
  - Feedback (hidden features)
  - Disocclusion Mask
  - Luma Derivative



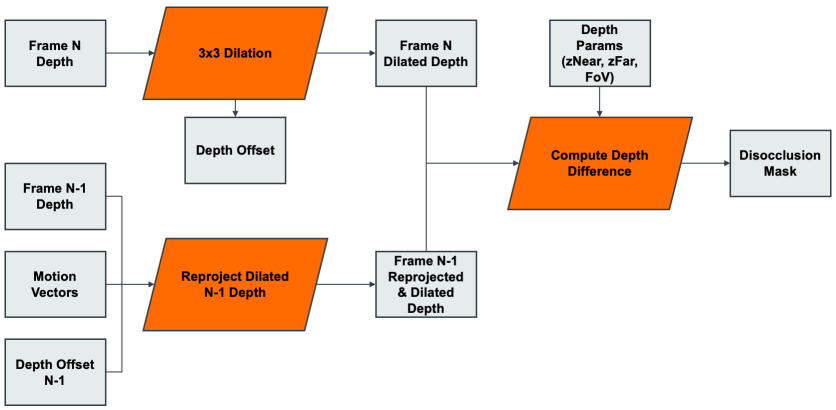
arm

The pre-process pass is all about computing the input tensor for the network – which is built up from rendered content in the current frame and previous buffers produced or used in the previous frame.

The inputs comprise of the 1spp color, the reprojected history, the reprojected hidden features, in addition to a calculated disocclusion mask and an input we call the “luma derivative”.

Pre Process – Disocclusion Mask

- Disocclusion mask provides hint to network on stale history
- Follow an approach similar to Arm ASR



arm

The disocclusion mask is provided as a hint to the network to flag regions of the frame that have been geometrically disoccluded in the current frame.

It's computed in a very similar way to Arm ASR, it dilates the depth and motion vectors, then computes an absolute difference in view space, forming a per-pixel flag, hinting where the history has gone stale.

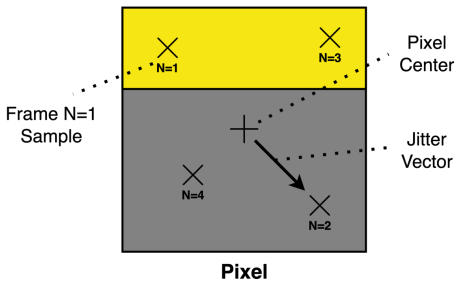
Pre Process – Luma Derivative

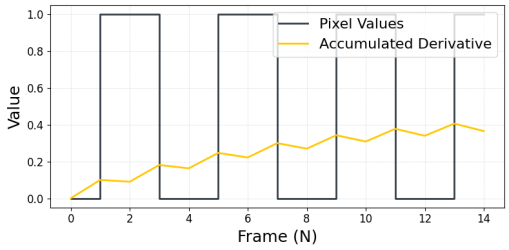
- High contrast thin features are challenging
- Successive jittered samples on/off high contrast edges can appear like ghosting
- Solution ➔ highlight regions as input feature
- Compute per-pixel luma derivative:

$$D_n(p) = |Y_n(p) - Y_{n-1}(p)|$$

- Then accumulate with exponential smoothing:

$$L_n(p) = \alpha D_n(p) + (1-\alpha) L_{n-1}(p)$$





arm

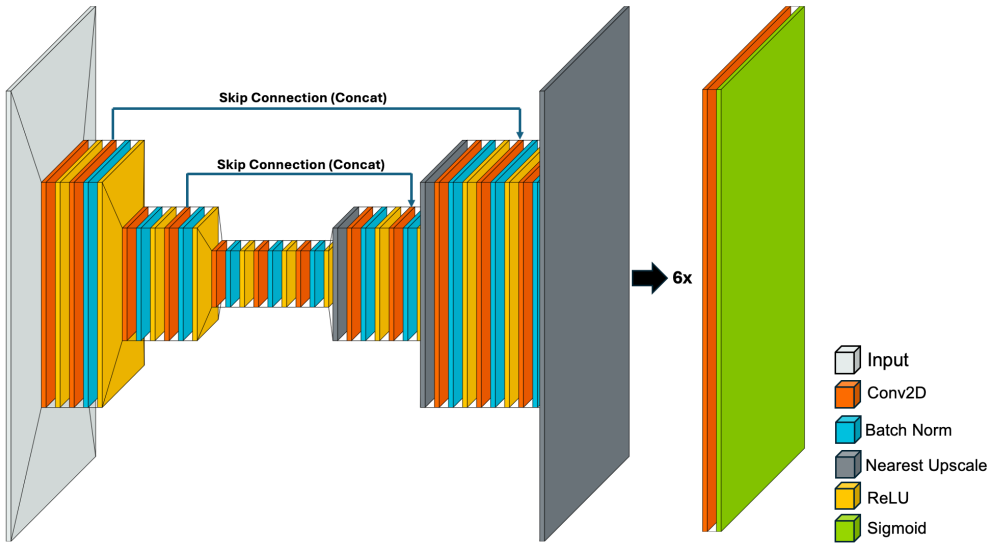
For improved temporal stability, we calculate an additional explicit input feature we call the luma derivative.

The idea here is simple, high-contrast thin sub-pixel features can flicker on/off in successive series of frames, due to the jitter vector sampling on/off thin geometry within the pixel.

This manifests as flicker and can trigger the network to believe a sample is ghosting, when it isn't.

To help decision making, we found maintaining a temporal buffer that calculates the temporal derivative of the luma for each frame and accumulates this via exponential smoothing helps to highlight such regions, and aids network prediction, by providing a hint as to where pixels are flickering.

Neural Network - Architecture



arm

As mentioned before, we're using a relatively simple UNet architecture as our backbone.

It has 4 levels of resolution, so uses three internal downsampling and upsampling operations, which are performed through strided convolutions, and nearest neighbor upsampling operations, respectively.

The architecture follows typical practice of getting deeper as the resolution decreases and reducing depth as spatial resolution increases. Currently the network has two skip connections for preserving high-frequency information.

Neural Network – Control Parameter Outputs

Kernel Weights Part 1 (4 channels)	Form 4x4 kernel filter for upscaling of low-resolution input frame
Kernel Weights Part 2 (4 channels)	
Kernel Weights Part 3 (4 channels)	
Kernel Weights Part 4 (4 channels)	
Temporal Coefficients (4 channels)	Theta and alpha for learnt rectification and accumulation
Temporal Feedback (4 channels)	Additional information to feed network temporally

- All output tensors aliased as images for upscaling through texture unit

arm

There are three key outputs from the network, as mentioned previously, these are:

- Kernel weights
- Temporal coefficients
- Hidden Features (called temporal feedback)

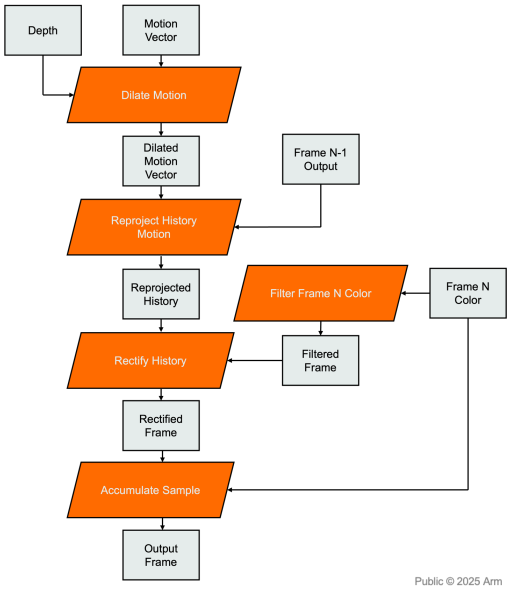
The kernel weights are 16ch in total and form a 4x4 filter for the low-resolution frame.

The temporal coefficients are employed for rectification and sample accumulation.

The Feedback is this hidden state tensor, noted previously, that is fed into the next future inference to carry forward state information on the networks decision making.

Post Process – Calculates Output

- Dilate motion with closest depth (3x3)
  - Reduces aliasing in motion vectors
- Reproject Color with Catmull-Rom Filter
  - Reduces blur from repeated reprojection
- Then, using network-controlled outputs
  - Anti-aliasing filter on 1spp color
  - History rectified
  - Fresh Sample accumulated
- Folded into a single compute pass



The final pass is the post process compute dispatch, which is responsible for producing the final output image.

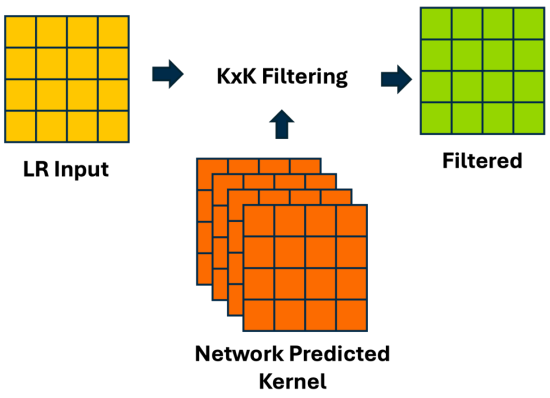
This involves dilating the motion vectors – which is commonly done in TSS to prevent reintroducing aliasing that is present in the motion vectors themselves

The dilated motion vectors are then used to reproject the previous output history with a Catmull-Rom filter – which is also commonly used in TSS to avoid blur from successive reprojections.

Finally, filtering of the low-res color is performed, and then the history is rectified and the new sample accumulated.

Post Process – Filtering

- 1spp color can be heavily aliased
- Unsuitable for direct replacement of stale history – causes rectification bias
- Targeted KPN applied to the input color to reduce aliasing
- KxK kernel weights predicted per-pixel by the network; currently use 4x4 size
- Further expanded upon using jitter-aware sparse filtering strategy for upscaling…



arm

Filtering to the low-res color is applied to reduce aliasing, reducing rectification bias on history reset events.

This is employed to alleviate cases where the low-res frame is too heavily aliased to directly replace stale history.

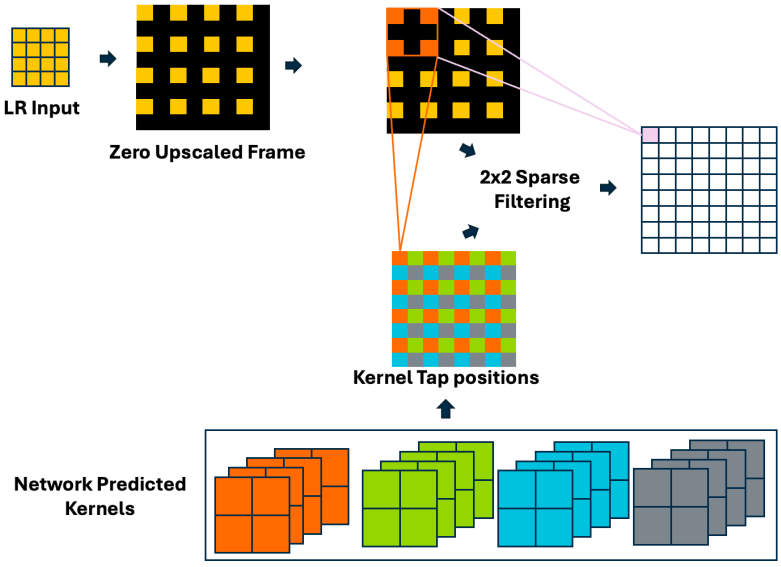
The network acts as a targeted KPN in this instance, predicting a single set of per-pixel filter coefficients that are spatially localized.

In the current approach, we're predicting a coefficients for a 4x4 kernel window.

For upscaling we augment this filtering strategy slightly to respect jitter positions on the high-res pixel grid.

Post Process – Sparsity in Upscaling

- Jittered low-res samples have varying distances to high-res pixels
- Zero-filled where samples miss the high-res grid
- Sparse filtering preserves jitter
- Estimated kernels interpolate between valid samples; like demosaicing



arm

In the upscaling case, we distribute the jittered low-resolution samples across the high-resolution grid.

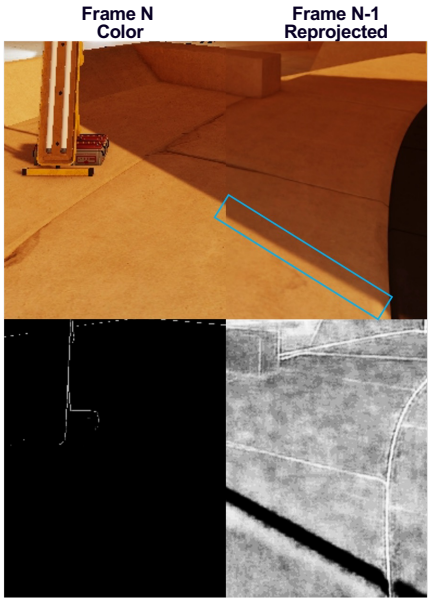
We apply zero-stuffing for the high-res pixels that are missing a sample on the grid.

We then apply the same kernel weights, but due to the zero stuffing, this becomes a a sparse filter. Which we can optimize to skip processing the empty taps.

This is similar to demosaicing in image processing, where the valid high-resolution samples are interpolated to fill in the missing pixels.

Post Process – Rectification

- Driven by a control parameter we call theta (θ)
  - Per-pixel network output
- Theta decides when the history is stale
- Resets to filtered / anti-aliased color under such an event
- Takes hints from the disocclusion mask
- But also, the contextual difference in 1spp color and reprojected history
  - Allows rectification when history becomes stale in content not represented by geometry, e.g., particle effects, shadows, reflections, etc.



arm

After filtering/upscaling the low-res color we rectify the reprojected history – this is driven by a control parameter the network estimates, that we call theta.

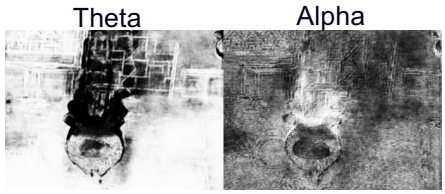
Theta decides when the history is stale, if it's stale, we lerp towards the filtered color, otherwise we keep the history.

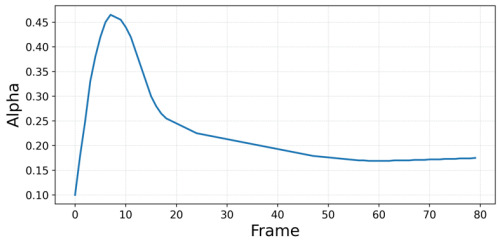
It's worth noting that whilst Theta is heavily correlated with the disocclusion mask, it is not the same – this is because unlike the disocclusion mask, theta can also rectify stale history based on content that is not representable by geometry

Like shadows, such as the example on the right, where we can see a shadow is moving across the floor, the depth-based disocclusion mask didn't flag this difference, but the network inferred the stale history from introspecting the contextual difference in the color.

Post Process - Accumulation

- Driven by a control parameter we call alpha (α)
  - Per-pixel network output
- Current sample accumulated into history
  - Scaling-aware ➔ jitter vector determined
- Observe learned alpha values over time:
  - Start high to accelerate convergence
  - Reduce to allow for higher sample count
  - On history rectification, trend repeats
  - Typically, inverse of theta
- Accumulated in tone mapped domain using cheap invertible Karis method
  - Avoids being largely altered by fireflies in HDR color





arm

Finally, once the history is reprojected and stale values are rectified (discarded), we can perform sample accumulation.

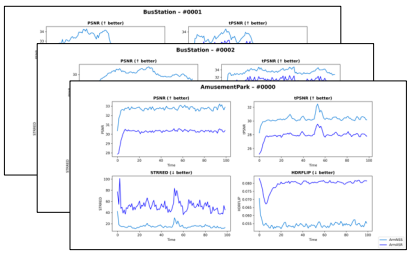
To do this, we estimate a parameter from the network we call “alpha” - this decides the rate at which new samples are accumulated into the history, in practice at a per-pixel granularity you tend to observe:

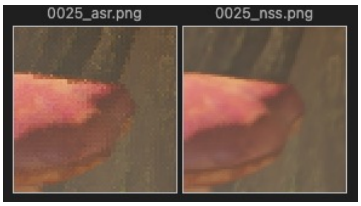
- Alpha starts as a relatively high value to accelerate convergence, it then reduces over time to allow for the maximum samples to be accumulated through exponential smoothing.
- When the history is reset, you’ll see this trend repeat, where in general alpha is following the inverse of theta.
- It’s worth noting we follow common TSS practice for sample accumulation, and accumulate new samples in tone-mapped domain, using the color preserving karis operator, to avoid fireflies heavily biasing the accumulation.

Results

Image Quality Analysis

- Subjective analysis is always the best
- Metrics can't (yet!) faithfully represent the Human Visual System
  ➔ But they can give good hints
- We compare our implementations against
  - Native Rendering @ 16spp
  - Desktop grade Neural Super Sampling techniques
  - State of the art non-neural techniques (e.g., ArmASR)
- We use a variety of tooling for comparisons
  - Objective analysis using CI systems
  - Robust subjective analysis using comparison tools





arm

We use a wide variety of image quality metrics, spanning from PSNR to state-of-the-art neural network-based approaches to attempt to validate quality

In practice, we haven't found anything quite as reliable as an expert's human eye yet – but the metrics serve as a useful guide and aid to find frames where the approach is particularly struggling.

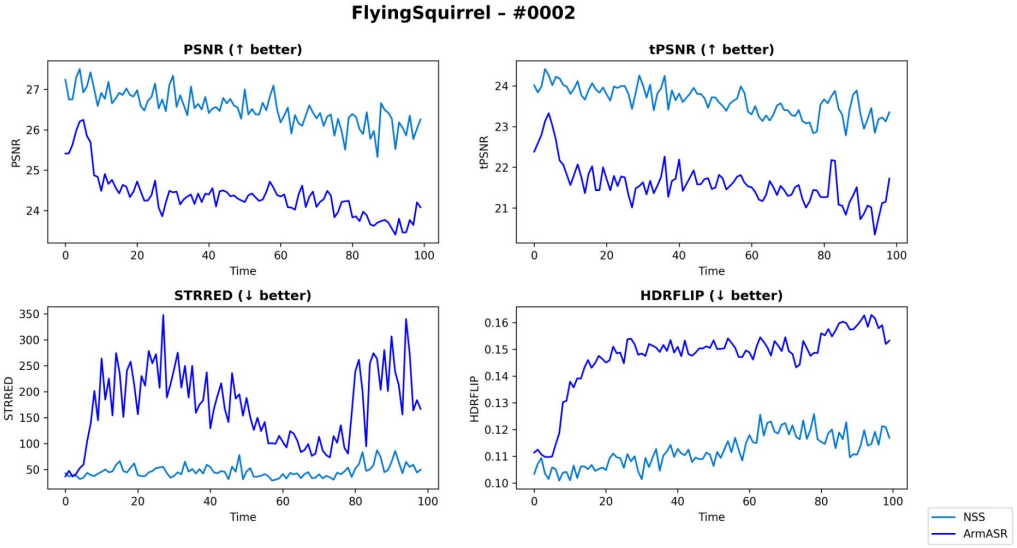
Typically, we perform our evaluations and compare against:

- The native anti-aliased ground truth
- Desktop grade neural approaches
- State-of-the-art non-neural techniques

We've built this into a CI workflow that allows large scale replaying of the test dataset through "replayer applications" and allows data-analysis.

For subjective analysis we make use of a variety of open-source tools to compare differences between mp4's and image files.

Results: Objective, 540 ➔ 1080



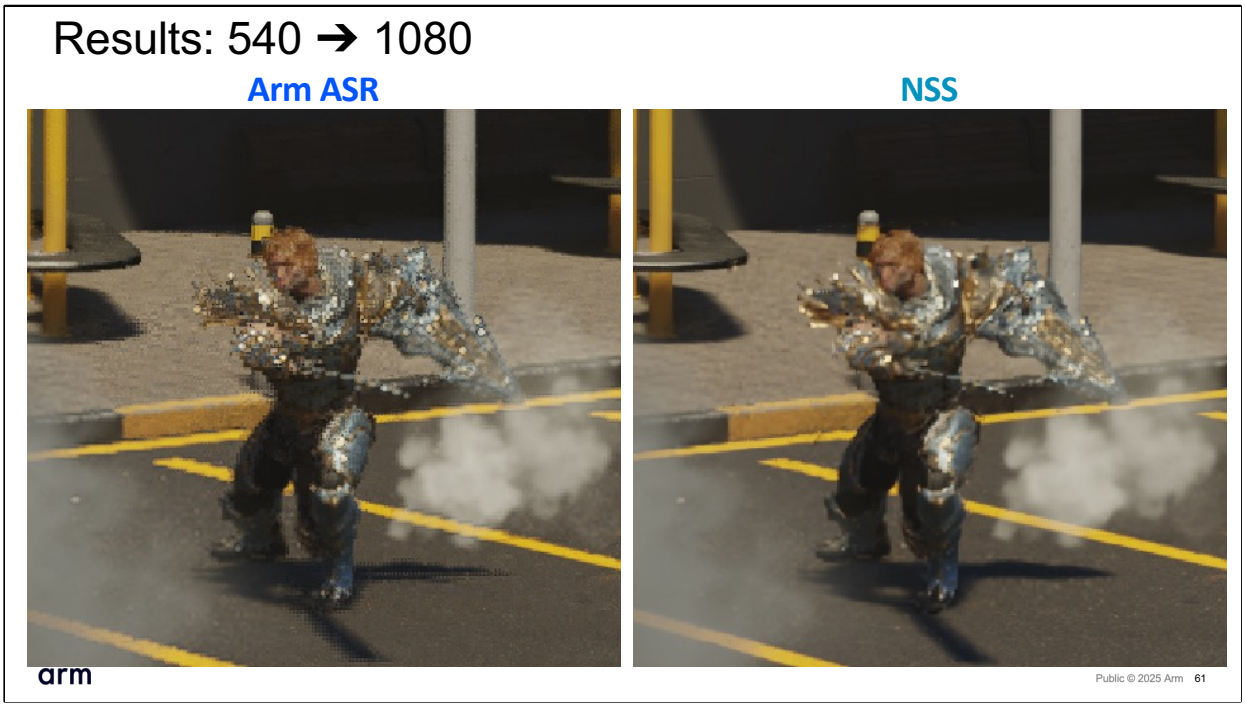
arm

For comparison we fed our entire unseen test dataset through our model and ASR.

ASR is Arm's mobile optimized non-neural upscaler and is based upon AMD's FSR2. It is representative of the state of the art of non-neural upscalers. This gives a direct comparison between the learnt heuristics and those coded manually.

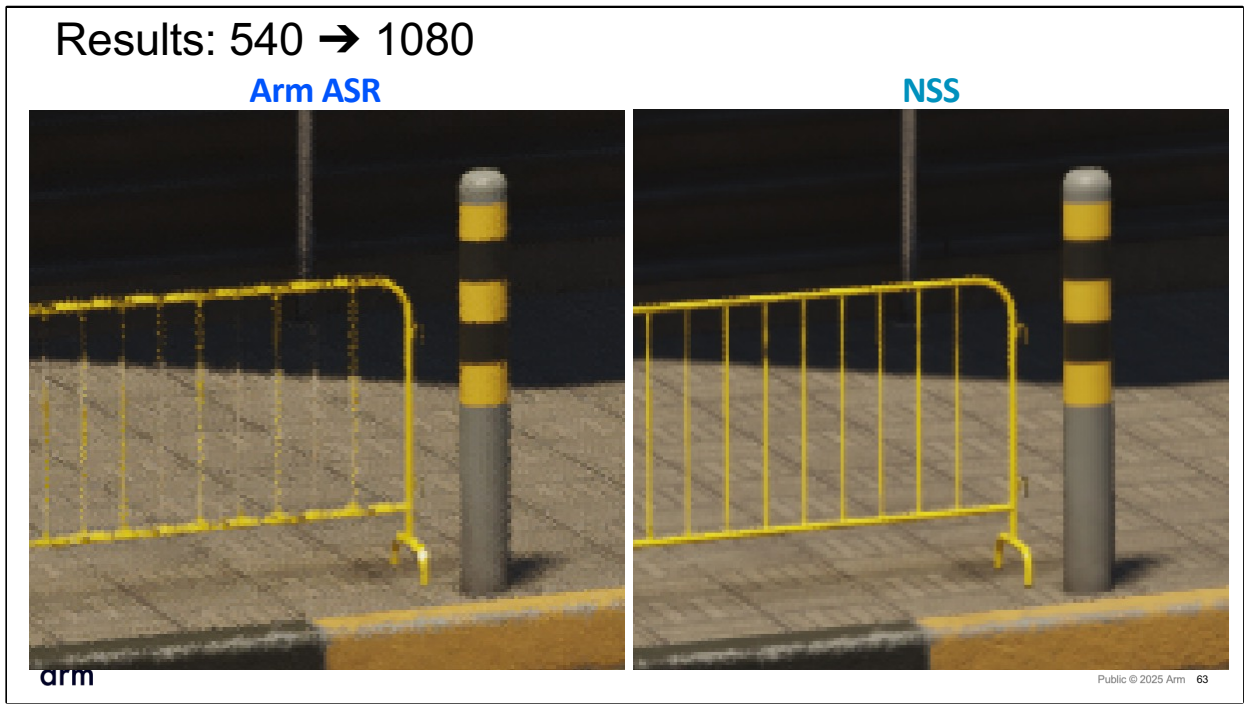
Specifically, this is the metric FLIP, that is a full-reference quality metric to compare the output frames against a ground truth. The lower the error the better. We found that we consistently out-performed ASR here, in some cases we note quite substantially.

I'll show a few examples digging into these instances.

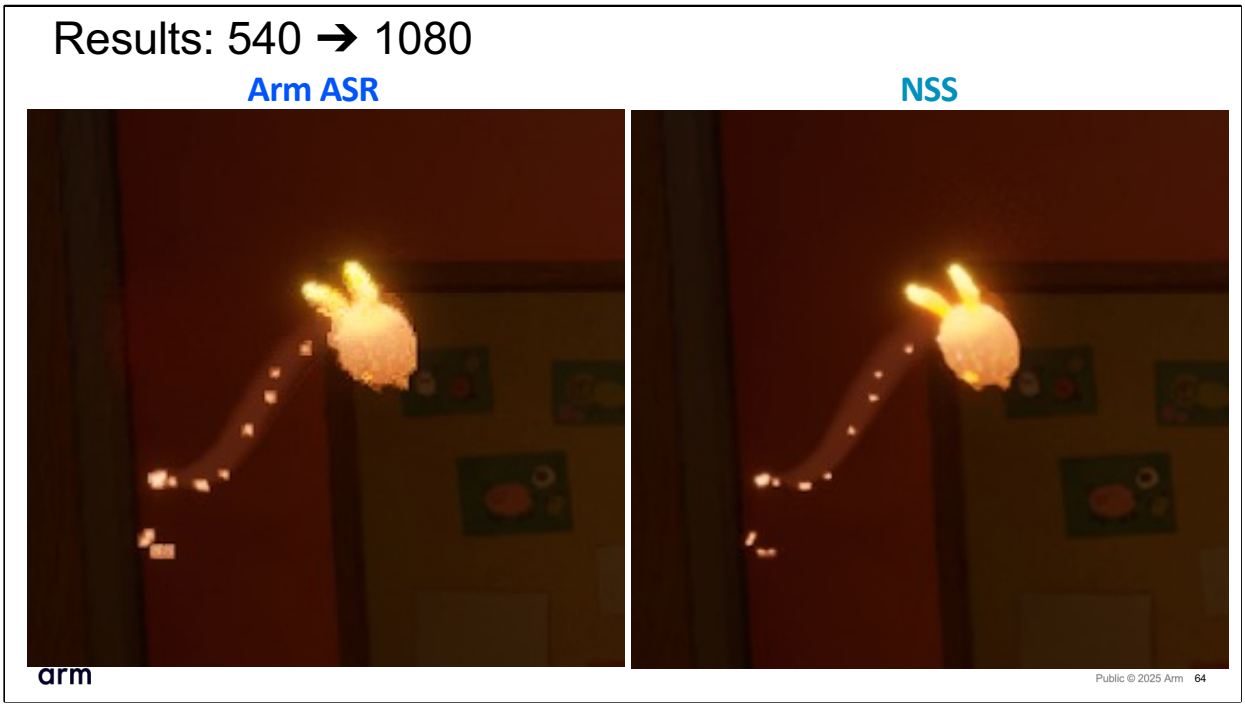


In general, we find NSS to produce temporally smoother and sharper results in fast motion than ASR and other non-neural upscalers.

NSS responds well to complex cases, with partially occluded character and shadow motion.



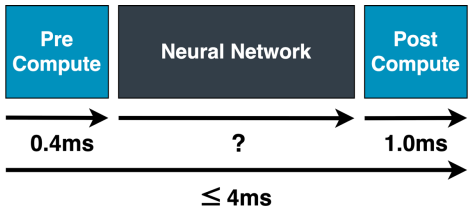
Under static conditions, thin features converge faster and result in more stable outputs.



NSS can also handles particle effects well, without need of a dedicated reactive mask, like Arm ASR and AMD FSR2 require.

Performance Framework

- $LatencyTarget \quad \leq 4ms \ @ \ 60fps$
- $ShaderLatency \quad \sim = 1.4ms$
- $AcceleratorTOPs \ \geq 10 \ TOPs/Watt$



$$MaxNetworkTOPs = MaxLatency * AcceleratorTOPs * Efficiency$$

arm

Whilst we do not have silicon today to accurately measure the performance of our work, we can estimate whether it is feasible under a napkin math performance model.

This is actually not unreasonable, as neural networks are highly deterministic workloads of consistent per-frame compute.

In order to estimate the cost, we do require some assumptions, here we decided on:

A target latency for mobile neural super sampling of less than or equal to 4ms per-frame at 60fps, which under our estimations should guarantee speed-up even under sub-linear pipeline speed up, when resolution is reduced by a factor of 4.

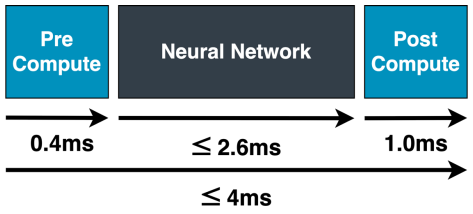
With this in mind, we optimized and measured the shaders on a low-clock frequency GPU (emulating thermally limited device). These are expected to dispatch before and after the network, like a sandwich, and measured to be around ~1.4ms.

And finally, to reiterate our position on future acceleration, we believe it to be reasonable to expect future devices to achieve 10TOPs of compute sustainably.

It's worth noting, we have disguised all manor of PPA complications under the guise of "efficiency", which we leave as a floating variable to explore.

Performance Framework

- $LatencyTarget \quad \leq 4ms \; @ \; 60fps$
- $ShaderLatency \quad \sim = 1.4ms$
- $AcceleratorTOPs \; \geq 10 \; TOPs/Watt$



$$MaxNetworkTOPs = (4.0ms \; - 1.4ms) \; * \; 10 \; TOPs/Watt = \sim \mathbf{26GOPs}$$

- **10GOPs** UNet Backbone ➔ feasible at 40% efficiency!

arm

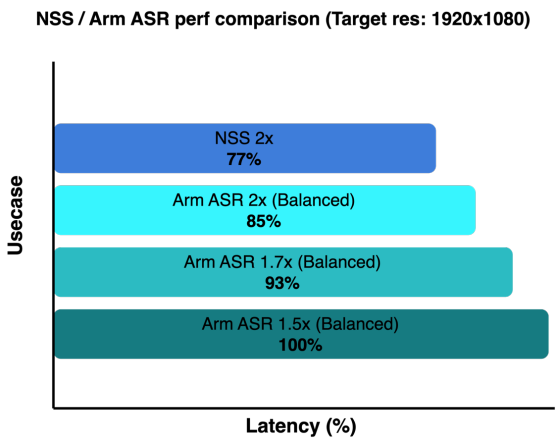
By populating this equation, we can see a neural super sampling network under this formulation should be no more than 27 Giga Operations in compute cost.

The parameter prediction approach that we have shared today, is currently utilizing a 10Giga Ops network backbone, which is achievable even when the black box neural accelerator is operating at 40% efficiency.

For us, this demonstrates adequate feasibility that an approach for neural super sampling such as the method we have shared today, is more than feasible within the scope of our future position on neural graphics acceleration.

Projected Relative Performance

- No silicon yet — insights based on early cycle-accurate simulations
- NSS shows efficiency gains over Arm ASR, with higher fidelity
- Improvements driven by neural-accelerated heuristics replacing complex shader-based ones



arm

To give a rough perspective of where this sits from cycle accurate simulation, it's costing roughly 77% of the ASR x1.5 upscaling runtime, in balanced mode.

And is projected to be relatively faster than ASR for x2 upscaling too.

The main reason for this is that many of the complex heuristics are being replaced by the neural dispatch, which is running very efficiently.

Conclusions

Future Work

- Not (yet!) a solved problem:
  - Reconstruction of heavily under sampled input frames ➔ moiré patterns
  - Sub-pixel particles lacking motion vectors
  - Striking the balance on temporal stability ⇆ spatial fidelity
- Main improvement vectors:
  - Finetuning on unseen content
  - Improve fall-downs in approach, e.g., new input features, extra filtering, etc.
  - Specialization ➔ domain-specific approaches, e.g., for manga art-style
- Challenging content motivates improvement
  - Can we create a consortium for open datasets for super sampling?

arm

To close out, super sampling is still not yet a solved problem, from our perspective.

There's still challenging failure cases we would like to address in future work, notably we would like to focus on:

- Heavily aliased input frames, which can produce moire patterns
- Sub-pixel features without motion vectors, such as tiny particle effects
- And deeper exploration into the loss function and finer-grained control of the temporal stability / spatial fidelity trade-off

In general, there's two primary improvement vectors we can explore for model improvement:

- The easiest is fine-tuning on unseen content, this typically yields improvement on content-specific failure cases
- Then for harder failure cases, such as those I spoke to above, we can look at modifying the approach, such as providing additional input features, such as the luma derivative, or the processing of the frames, such as adding additional filtering passes

In some cases, we pre-empt there might even be content-specific neural upscalers, that can exploit specific nuances of an art-style like manga, for a more efficient

content-specific neural upscaling.

The pre-dominant observation from this work has been that many challenging failure cases are not unique to the NSS solution we have arrived at, there's definitely value in collating such cases in open datasets for the industry to benchmark against – maybe this will eventually become a consortium for open datasets?

More Information

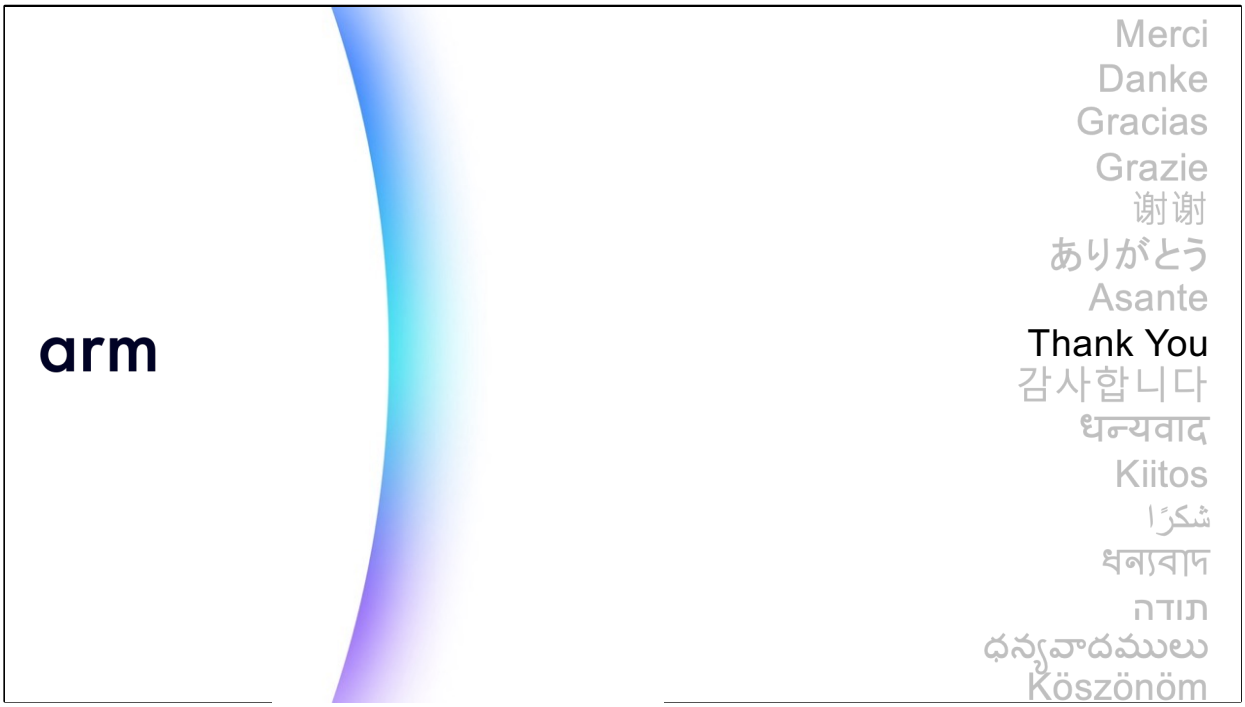
- NSS model on Hugging Face:
  - https://huggingface.co/Arm/neural-super-sampling
- Neural Graphics Development Kit
  - https://developer.arm.com/mobile-graphics-and-gaming/neural-graphics

For more information on NSS, you can read more from several blogs we've published, or find the source code for inference and training in the Hugging Face link.

Acknowledgements

- Thank you to the key technical contributors:
  - *Josh Sowerby*
  - *Matt Wash*
  - *Ridhwanul Haque*
  - *Yanxiang Catherine Wang*
  - *Sam Martin*
- Thanks to those who supported integration and analysis:
  - *Sergio Alapont*
  - *Pedro Boechat de Almeida Germano*
  - *Douwe van Gijn*

Special thanks to the team who have contributed to this work.

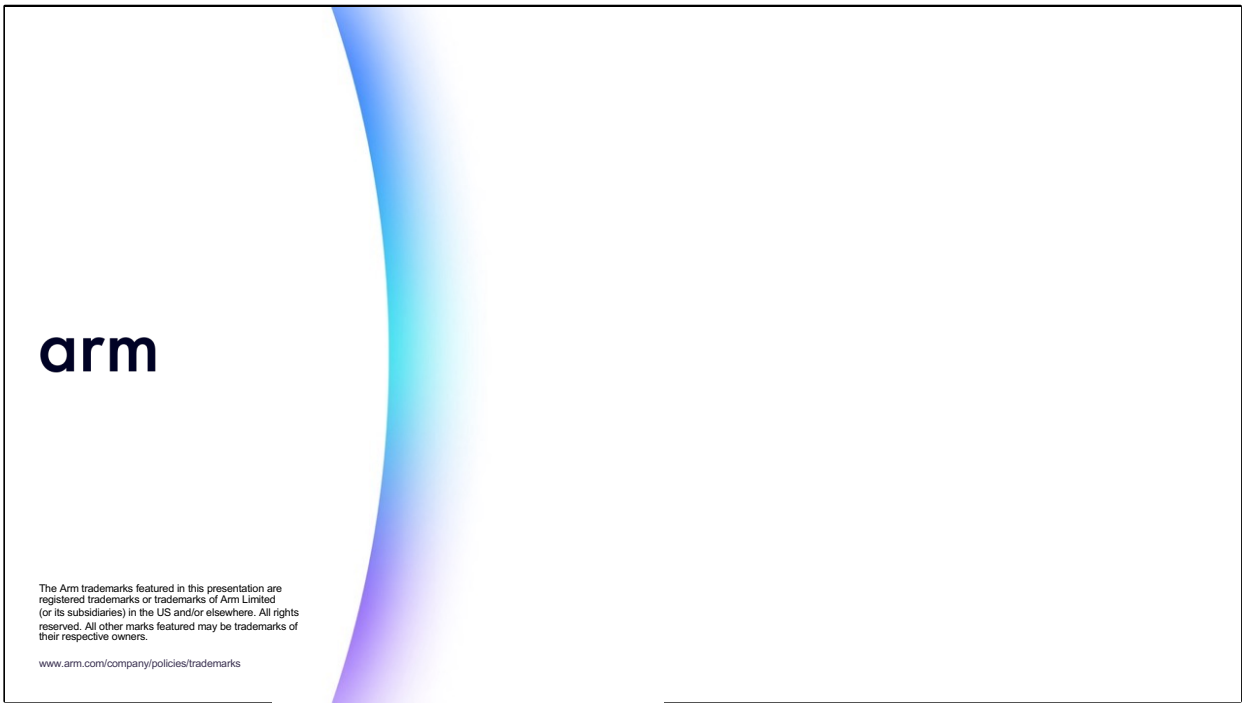

arm
Merci
Danke
Gracias
Grazie
谢谢
ありがとう
Asante
Thank You
감사합니다
धन्यवाद
Kiitos
شكرًا
ধন্যবাদ
תודה
ధన్యవాదములు
Köszönöm

References: Bibliography

- [Karis14] High Quality Temporal Supersampling
  - https://de45xmedrsdbp.cloudfront.net/Resources/files/TemporalAA_small-59732822.pdf
- [Yang20] A Survey of Temporal Antialiasing Techniques
  - https://diglib.eg.org/server/api/core/bitstreams/53732e70-b64d-46f4-bbae-865eb7673a35/content
- [Salvi17] Deep Learning The Future of Real Time Rendering
  - https://slidetodoc.com/deep-learning-the-future-of-realtime-rendering-marco/
- [Pedersen16] Temporal Reprojection Anti-Aliasing in INSIDE
  - https://s3.amazonaws.com/arena-attachments/655504/c5c71c5507f0f8bf344252958254fb7d.pdf?1468341463
- [Salvi16] An Excursion in Temporal Supersampling
  - https://developer.download.nvidia.com/gameworks/events/GDC2016/msalvi_temporal_supersampling.pdf
- [Thomas20] A Reduced-Precision Network for Image Reconstruction
  - https://creativecoding.soe.ucsc.edu/QW-Net/
- [Andersson20] ꟻLIP: A Difference Evaluator for Alternating Images
  - https://research.nvidia.com/publication/flip
- [AMD22] It's time for AMD FidelityFX Super Resolution 2.0
  - https://gpuopen.com/fsr2-announce/
- [Wronski21] Procedural Kernel Networks
  - https://arxiv.org/abs/2112.09318
- [Jimenez17] Dynamic Temporal Antialiasing and Upsampling in Call of Duty
  - https://research.activision.com/publications/2020-03/dynamic-temporal-antialiasing-and-upsampling-in-call-of-duty
- [Andersson21] Visualizing Errors in Rendered High Dynamic Range Images
  - https://research.nvidia.com/publication/2021-05_visualizing-errors-rendered-high-dynamic-range-images

Licenses: content

- Mori, Steel Arms, and Enchanted Castle
  - Arm Strategy and Ecosystems content creation team
- San Miguel 2.0 and Teapot (CC BY 3.0):
  - Morgan McGuire, Computer Graphics Archive, July 2017
    - https://casual-effects.com/data
- Bus Station
  - ArtcoreStudios, Unreal Engine Market Place, Jun 22, 2022
    - https://www.unrealengine.com/marketplace/en-US/product/bus-station-01
- Realistic Starter VFX Pack Vol 2
  - FX Cat UA, Unreal Engine Market Place, Feb 15, 2022
    - https://www.unrealengine.com/marketplace/en-US/product/realistic-starter-vfx-pack-vol


arm
The Arm trademarks featured in this presentation are registered trademarks or trademarks of Arm Limited (or its subsidiaries) in the US and/or elsewhere. All rights reserved. All other marks featured may be trademarks of their respective owners.
www.arm.com/company/policies/trademarks