Neural Image Upsampling in God of War: Ragnarök

Xuanyi Zhou
Rendering Programmer
Sony Santa Monica Studio

Good afternoon. My name is Zhou Xuanyi and I'm a rendering programmer at Santa Monica Studio. Today I'll be presenting the neural network-based image upsampling system in God of War: Ragnarok.

Agenda

- Background
- System/network design and training details
- Implementation and optimization

I'm going to first talk about the motivations and goals for the system. I'll also be introducing BC7 image compression that our method works with.

Then, I'm going to cover how the system is designed. What we're going to focus on is how we adapt simple tools to fit our specific needs.

After that, I'm going to talk about how we implemented and optimized the system to run on a PS5. This will include high-level implementation choices as well as a fair amount of in-depth and low-level details that primarily focuses on optimization.

Design Goals

- Save disk space by storing lower-res textures on disc
- All texture assets can be upsampled
  - Most textures are BC compressed
- No extra cost for compression by outputting directly to BC
- High performance and adaptive

We started working on the system with this set of goals:

[click]

We hoped that upsampling textures at run-time would help us save disk space – Artists author textures for PS4, and we upsample these textures on PS5 while keeping roughly the same package size.

[click]

We hoped to upsample as many textures as we can,

[click]

and since most of them are BC compressed, we started out focusing on one of the most complex versions of BC which is BC7.

[click]

We wanted to use a single network to handle both upsampling and compression, and output directly to BC.

[click]

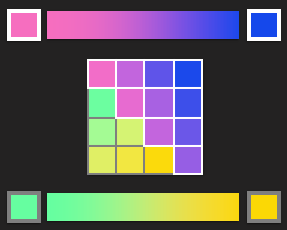
The player is always moving from one level to another, and upsampling must keep up.

We intended to use any excess GPU time to perform upsampling, so the method has to be adaptive as well.

We'll see during the presentation how much of these goals we were able to achieve, and to what extent.

BC7 ("Block Compression 7")

- Designed for `RGBA8Unorm/RGBA8Snorm` formats
- Operates on 4 × 4 blocks, 16 bytes per block
- Interpolation between two colors (endpoints) using indices
- 8 different modes
  - Each mode can have up to 64 partitions, or 4 rotations



Let's first talk about BC7, or Block Compression Seven.

[click]

It's a GPU-friendly compression method designed for 8 bits per channel, RGBA images,

[click]

and it operates on 4-pixel by 4-pixel blocks, where each compressed block is 16 bytes. Effectively, it compresses an image to one fourth of its original size.

[click]

The fundamental assumption behind BC compression is that a block contains similarly colored pixels. Each pixel can then be represented with linear interpolation between two colors that are shared across the block.

[click]

Each BC7 block can choose from 8 different modes.

[click]

Some modes can partition a block into sets of pixels that use different colors as endpoints.

Here’s an example of a mode with two subsets of indices. It has too much contrast to be compressed with decent quality with only one pair of endpoints.

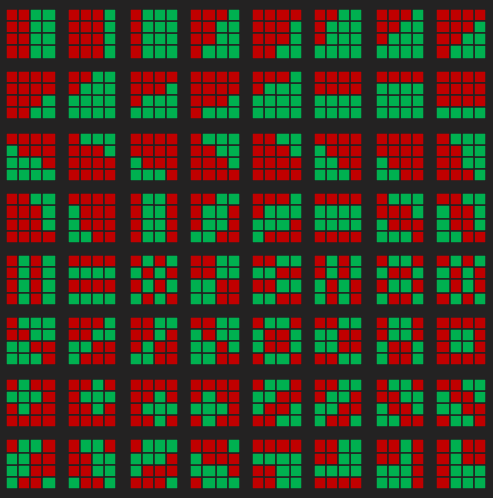
The choice of which pair of endpoint to use for a pixel is based on a hard-coded pattern that changes depending on which mode and partition is used.

Some modes also support rotation, where a color channel is swapped with the alpha channel after all other decoding has finished.

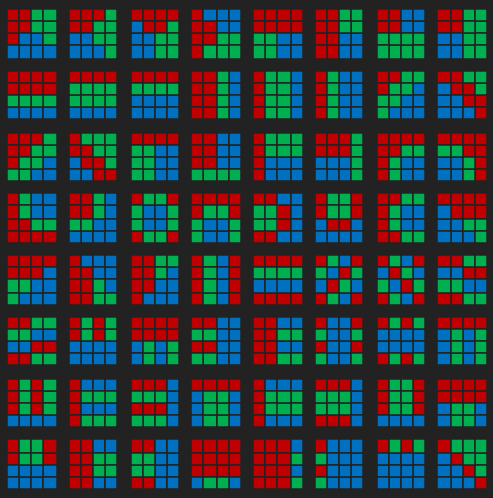
BC7 Partitions

https://rockets2000.wordpress.com/2015/05/19/bc7-partitions-subsets/

2 Subsets



3 Subsets



5

There are 64 partition patterns for modes with 2 sets of endpoints, and a different 64 patterns for modes with 3 sets. Each color denotes a set of pixels that use a different pair of endpoints. Each individual block that uses a mode that supports partition, can choose from one of these.

BC7 Modes

Mode	Channels	Index Bits	P-bit	Subsets	Partitions
0	4,4,4	3	√	3	16
1	6,6,6	3	Per Subset	2	64
2	5,5,5	2	-	3	64
3	7,7,7	2	√	2	64
4	5,5,5,6	2,3	-	1	-
5	7,7,7,8	2,2	-	1	-
6	7,7,7,7	4	√	1	-
7	5,5,5,5	2	√	2	64

This table shows the details of each BC7 mode. We're not going to dive deep into the specifics; rather, this is just for an intuitive understanding of how much the modes differ from each other, and the complexities it causes.

[click]

Here's an example. The "channels" column specify how many bits are used to represent each channel of the endpoint. Thus, mode 0 will be encoding endpoints with a much lower precision than mode 3, since it's using 3 less bits per channel.

[click]

Another example is that the last four modes support alpha channels while the first four modes don't.

Network Design

Let's look at how the neural networks are designed and trained.

First Version

- In partnership with R&D Center US Laboratory
- Multilayer perceptron: 512 – 512 – 512 – 512
- Network input and output: Block parameters
  - Example for mode 1:

64 Partitions	2 Sets of RGB Endpoints	16 Indices
Softmax	Sigmoid	Sigmoid

- Outputs 4 blocks at once

Since BC7 has 8 distinct modes, the natural first step is to separate them out into different networks.

[click]

The team at R&D Center US Laboratory kindly provided the implementation of this version.

[click]

For each mode, we use a multilayer perceptron with four hidden layers that are 512 wide.

[click]

We convert the input BC7 blocks into vectors of parameters. This includes the colors used as endpoints and indices used to interpolate between them. Depending on the mode, this also includes partition or rotation used by the block encoded as a one-hot vector.

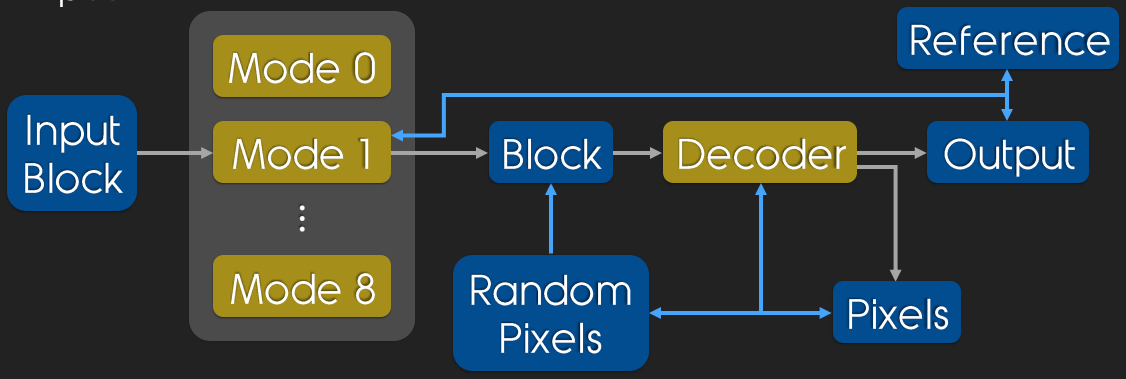
An example for mode 1 can be seen in the table.

[click]

Because each block in the low-resolution image corresponds to four blocks in the upsampled image, this method outputs 4 blocks at once.

First Version

- Uses a neural BC7 decoder trained in advance with random input



Comparing BC7 block parameters directly would be suboptimal because there are multiple ways to encode the same block.

Instead, we initially used an additional neural network to decode the outputs into 4x4 pixel blocks.

[click]

Here's what the process looks like: The input blocks go through one of the networks, then the result is decoded and compared with the reference.

[click]

The decoder networks are trained in advance with randomly generated inputs. Their parameters are then fixed when we train the upsample networks.

Initial Results

Bilinear

NN

Here are the results of this first version on the right. Compared with bilinear upsampling on the left, it does sharpen the image somewhat.

[click]

Initial Results

Bilinear

NN

11

But it also introduces lots of visible artifacts, like here where the grass blade is broken.

Issues

1. No way to determine which mode to use
2. Need many networks for combinations of input and output modes
3. Networks need to learn all the partition patterns
4. No context information
5. Inaccuracy introduced by the NN decoder

So, what went wrong? If we look, there are a few apparent issues.

[click]

First of all, we don't know which mode to use for each output block. This version uses the mode of the input block for all 4 output blocks, which is suboptimal.

[click]

For 1 input block and 4 output blocks with 8 modes each, there are over 30,000 combinations.

[click]

Regarding upsampling quality, since partition information is encoded in the inputs and outputs, the network essentially has to learn the hard-coded partition patterns. This increases the number of parameters needed and introduces discontinues in the problem space.

[click]

We don’t have a means to provide any context information – the networks are unable to see any pixels that are out of the input 4x4 chunk.

[click]

And finally, the neural network decoder also introduces a small amount of error.

Splitting the Problem

- No need to use raw BC7 block as input
- "Mode predictor" network selects which network to use
- Submode: A partition/rotation within a mode
  - Different networks for different submodes
    - 281 submodes
      - Ran a search to find the mode/partition/rotation combinations that give the least amount of error

As a first step to solving these issues, we tried splitting the problem into smaller ones.

[click]

Firstly, since the GPU has dedicated hardware for BC7 decoding, we can use the pixel values directly for network input. This also means that we don't need to output 4 blocks at a time.

[click]

To determine which mode to use, we add a small neural network which we call the "mode predictor".

[click]

For convenience, we call partitions or rotations within one mode "submodes".

[click]

We separate different submodes into different networks that no longer

need to care about partition or rotation. However, this re-introduces the issue of requiring a lot of networks;

[click]

Specifically, there are a total of 281 submodes that we need to consider.

When observing the outputs of a conventional compressor, only a few submodes would be frequently used, and would often produce better results than most other modes despite not being optimal.

[click]

Therefore, we created a dataset of blocks and ran them through the compressor using every single submode, giving us compression errors. We then ran a beam search on this data to find a combination of a small number of submodes that produce high-quality results when used in conjunction.

We selected 4 submodes in the end.

[NOTES]

16 + 64 + 64 + 64 + 4 + 4 + 1 + 64 = 281

Beam search:
Since we have a lot of data, we quantized all output PSNR values to 8 bits so that it's easier to fit the entire dataset into memory. The reason we're using a beam search is also to reduce memory usage.
It turned out to be a bit of an overkill – just counting the frequencies of modes chosen by the compiler yields pretty much the same results. Running a search on a subset of the data will likely work as well.

Network Details

- Mode predictor: 64 – 64
  - Outputs expected error for all candidate modes
  - Account for both compression error and error introduced by upsampling networks
- Upsample networks: 128 – 128 – 64 – 64
- Input: box downsampled and compressed textures
  - 2 × 2 block in the low-res image corresponding to the high-res 4 × 4 block, with an additional one-pixel border for context information

[click]

The mode predictor is significantly smaller than regular upsample networks and is much cheaper to evaluate.

[click]

The job of the mode predictor is not to classify a block into one specific mode, but rather to predict how well each mode can encode a block. Therefore, instead of using a softmax to predict the single best mode, it outputs expected error for all modes. This means that even when it does not select the best mode, it will still select one that's good enough.

[click]

This network is trained with errors obtained from blocks produced by the upsample networks to account for both compression error and error introduced by upsampling.

[click]

For upsample networks, we now only need endpoints and indices for

one block, which means that we can also get away with a much smaller network.

[click]

We use downsampled textures as training input for all these networks. These are downsampled with a box filter, which matches how we generate mipmaps in the build system. The inputs are also BC7 compressed and decompressed to mimic how textures will be compressed in the game.

[click]

Each 4x4 output block corresponds to 2x2 pixels in the input image. In addition, we provide context information with pixels around the 2x2 block. Based on our experimentation, a one-pixel border strikes a good balance between quality and model size.

Eliminating the NN Decoder

- The BC7 decompression process is *mostly* differentiable
  - Interpolation is differentiable
  - Quantization is *not* differentiable
    - Use a custom layer to quantize the outputs, but leave its gradients unchanged

And now, we'd like to solve the last issue as well: we want to get rid of the neural network decoder.

[click]

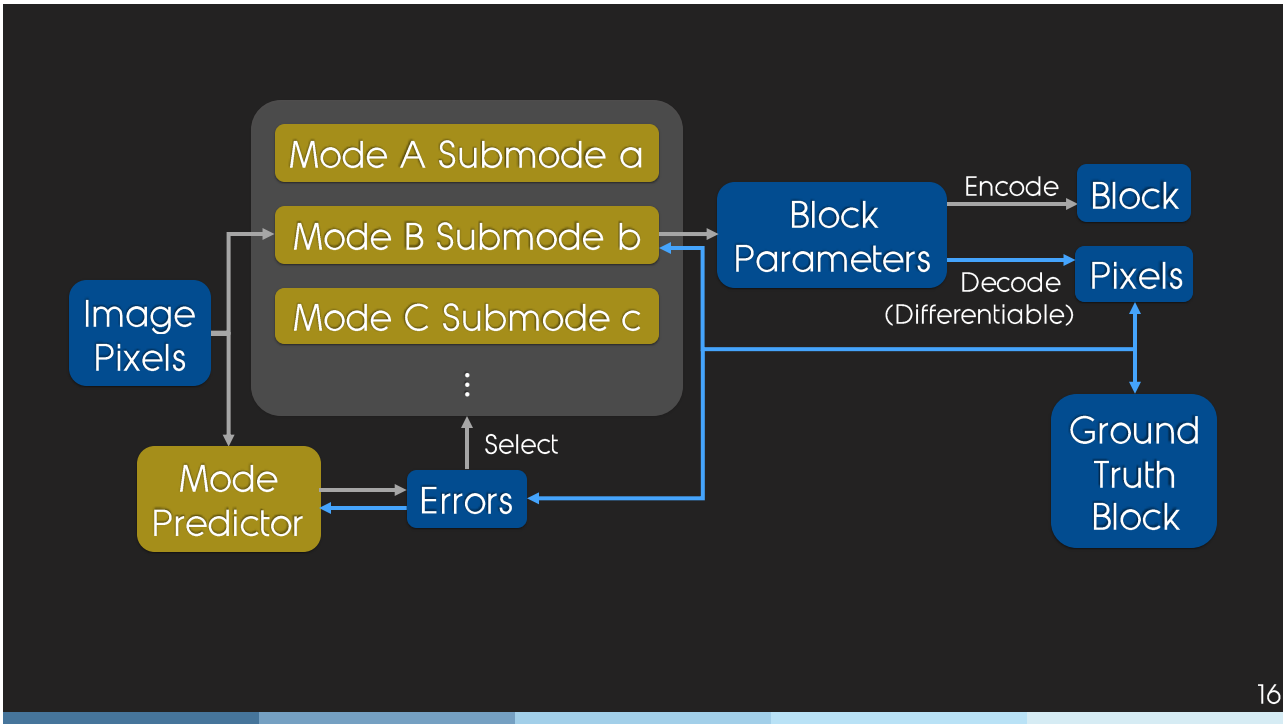
An important observation is that the BC7 decompression process is mostly differentiable, which means that we can replicate the process in a differentiable manner and gradients will flow through naturally.

[click]

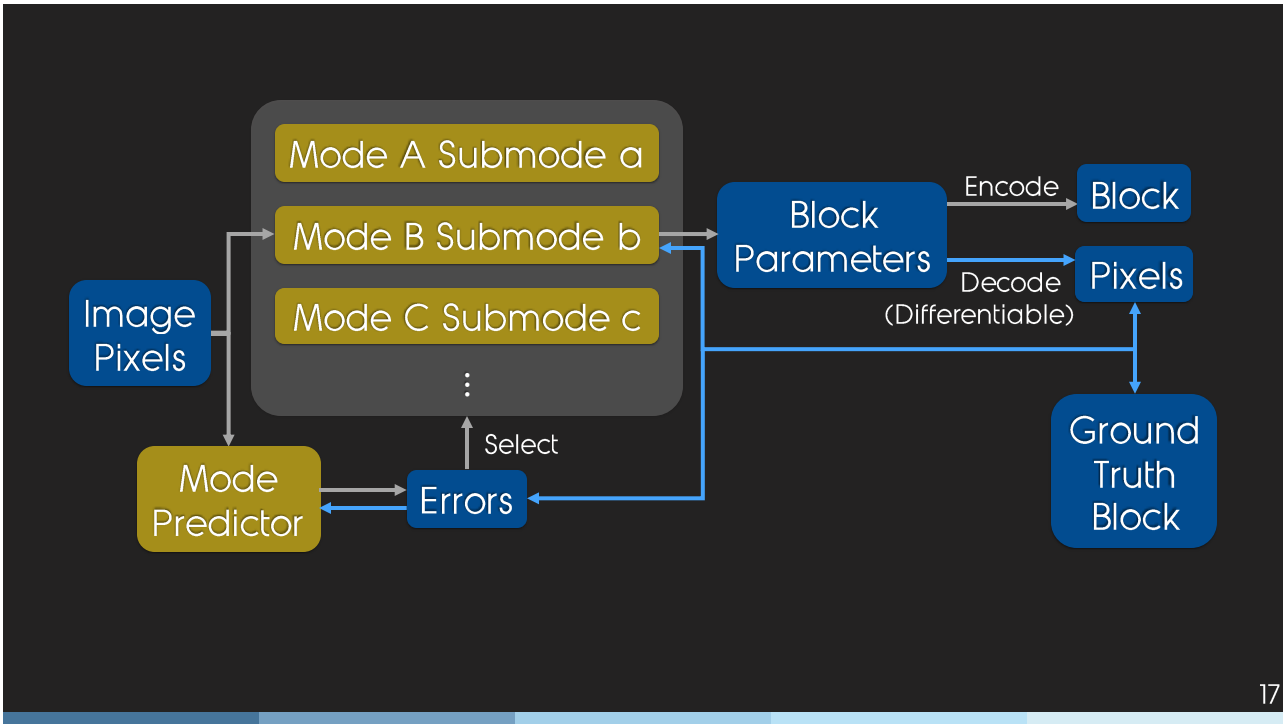
It consists of mostly interpolation and quantization steps, of which only quantization is not differentiable.

[click]

We used a naïve solution which is to simply quantize the values and leave the gradients intact. This worked reasonably well in practice.



Finally, our updated process looks like this.



The image pixels are first fed into the mode predictor for errors.

[click]

Which is used to select the best network to use.

[click]

The pixels are then passed through the selected network.

[click]

And the resulting parameters are encoded into the block.

[click]

When training, we instead use the differentiable decoder and compare the result with the reference.

[click]

And finally, we also use these errors to train the mode predictor.

Results

Bilinear

New

Old

Here are what the results now look like in the middle, compared to bilinear upsampling on the left and the old results on the right.

There are less artifacts, and small features are better preserved.

This is the method that we used in the final game.

Results

Bilinear

Neural Network

Here's another example for normal maps. In this example, we're using a different set of networks trained specifically for normals.

Production

- Comparison with non-NN-based upsampling and compression solution provided by Tom Madams at SIE Tools and Technology

Bilinear | SIE | Neural Network

We have a non-neural-network-based upsampling and compression solution provided by Tom Madams at SIE Tools and Technology.

Here is an example of a normal map being upsampled and compressed using both methods. The labels below these images show how they are upsampled.

Compared to the approach from SIE, the main benefit of our method is that it is better at enhancing details such as edges. You can see sharper edges in the image on the right compared to the image in the middle.

On the other hand, our method is more unpredictable and prone to artifacts such as color shifts and block artifacts, which you can also see in this example. For this reason, we preferred the SIE method for diffuse textures and used neural networks primarily for normal maps.

Production

- Our NN-based method is primarily used for normal maps

Texture Type	Upsample Method
Diffuse BC1	SIE (Falls back to NN)
Diffuse BC7	SIE (Falls back to NN)
Normal	NN

- In 10 minutes of gameplay, the whole system produced ~760,000,000 BC1 and BC7 blocks (~10 GB uncompressed)
  - ~445,000,000 BC7 blocks (~7GB uncompressed) from NN

This table shows what methods we ended up using for the final game for different types of textures.

The SIE method has certain limitations on the input, in which case we fall back to our method.

[click]

We ran a test by progressing normally through the game and monitoring how much data is being produced. In 10 minutes, the system produced 760 million BC1 and BC7 blocks with both our method and the method provided by Sony, which is around 10 gigabytes of BC1 and BC7 blocks if no other form of compression is used. Around 70% of the 10 gigabytes of data is produced by neural networks, which is equivalent to around 42 4k textures per minute.

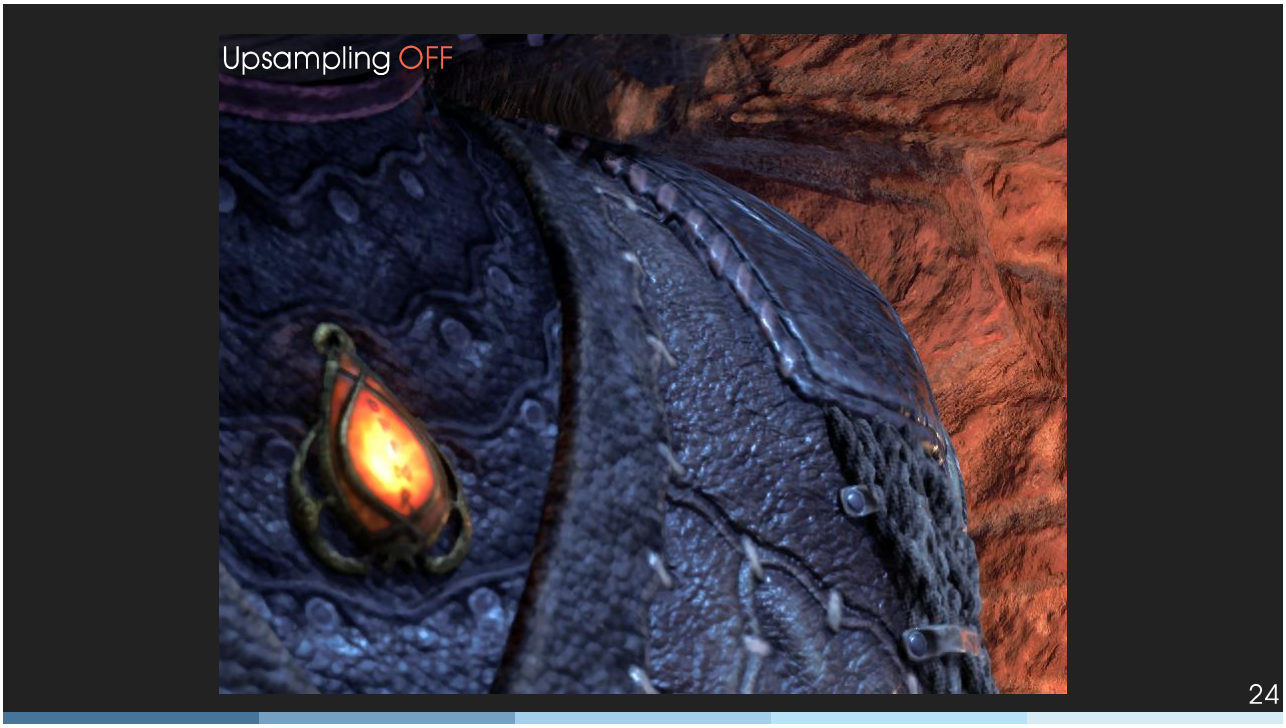
[NOTES]

Alfheim with Tyr, 10 min:
4,092,592,128 BC1 pixels with Tom‘s upscaler (255,787,008 blocks,

2,046,296,064 bytes)
878,706,688 BC7 pixels with Tom‘s upscaler (54,919,168 blocks, 878,706,688 bytes)
7,114,588,160 BC7 pixels with NN (444,661,760 blocks, 7,114,588,160 bytes)

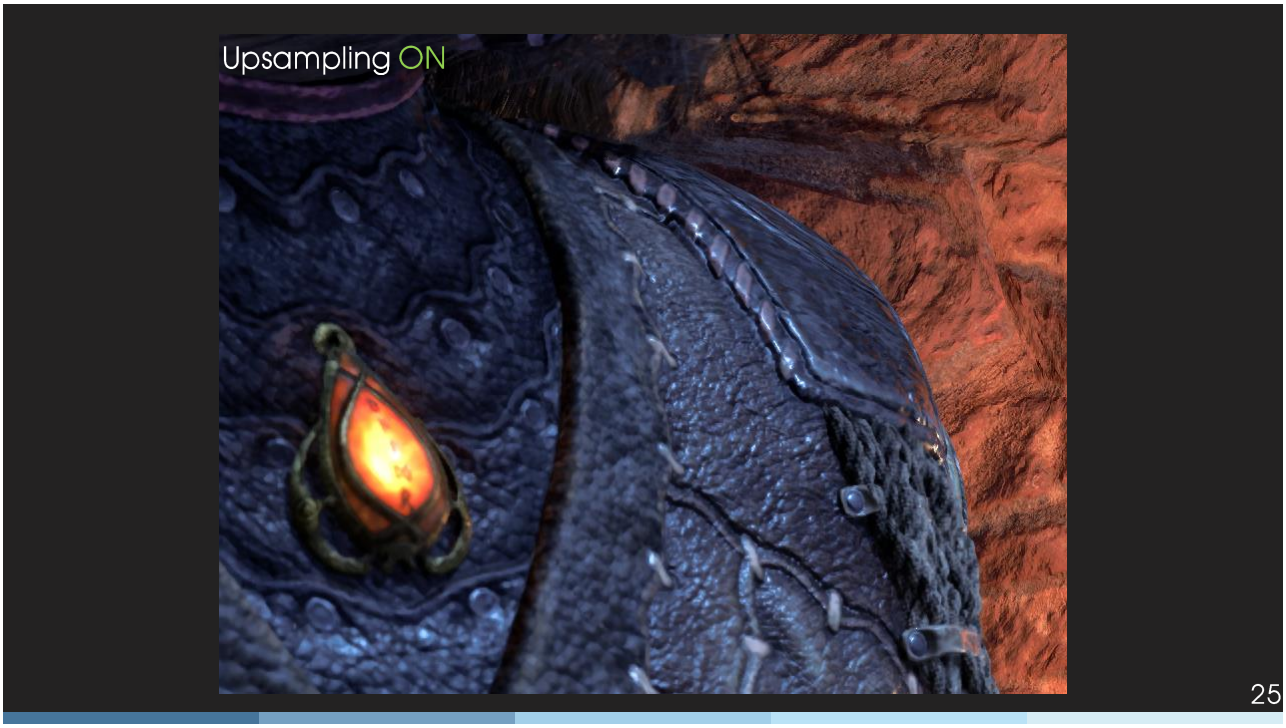
Here’s an example of in-game visual improvements. These can be fairly subtle, but you can see higher resolution patterns on the rock on the right, and more detailed specular highlights on the left.

Here’s an example of in-game visual improvements. These can be fairly subtle, but you can see higher resolution patterns on the rock on the right, and more detailed specular highlights on the left.



Here’s another example on the in-game character, Freya. Notice how her shoulder plates now have more definition.

[pause]



Here's another example on the in-game character, Freya. Notice how her shoulder plates now have more definition.

[pause]

Pros & Cons

https://en.wikipedia.org/wiki/Peak_signal-to-noise_ratio

- Relatively easy to extend to other BC modes
- Upsampling quality is relatively low – PSNR = 26

Original Image

PSNR = 31.45

[click]

This method can be easily extended to other BC modes or any other compression schemes that are based on interpolation. We have an experimental version for BC1 that we ended up using only as a fallback.

[click]

One drawback with this approach is the quality of the results. We have a peak signal-to-noise ratio of around 26 which is relatively low compared to even some of the earliest convolutional neural network based upsampling methods.

[click]

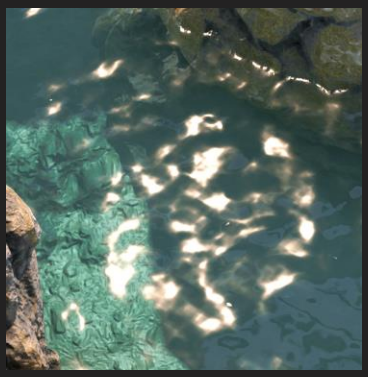
For context, here's an example of an image with a PSNR of 31 on the right, where you can already see a lot of banding. Banding is rare for our method, but still, there are plenty of room for improvements.

Convolutional networks have much larger reception fields instead of our tiny 4x4 window. This gives them the opportunity to react to global characteristics of the image and add in more details.

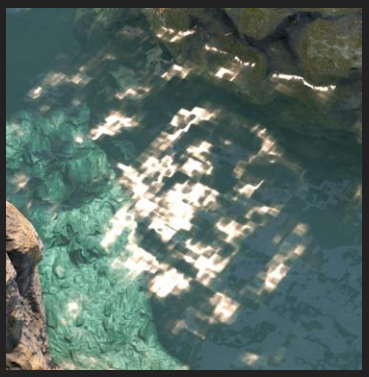
Unfortunately, switching to convolutional networks requires extensive changes to how we evaluate them at run time, which we'll introduce shortly.

Pros & Cons

- Block artifacts
- Table of known bad textures that we don't process



Without Upsampling



With Upsampling

You can see some of the most visible cases of artifacts in the image on the right.

Since BC7 works with 4x4 pixel blocks, the networks sometimes produce block artifacts. This is especially visible for highly specular surfaces under direct lighting.

We attempted to fix this by training with four neighboring blocks at the same time and adding first and second order gradients to the loss function. These helped somewhat but the issue was still visible.

[click]

We ended up creating a table of known bad textures that are rejected by the upsampling system.

Network Evaluation and Implementation Details

So far, everything we've been doing has been with PyTorch on a PC.
We need a way to evaluate these networks on a PS5.

Can't embed PyTorch into the game!

- Perform full network evaluation in a single shader
- Early attempts
  - Process one block per thread
  - Not enough registers or LDS to store intermediate results while keeping high occupancy
  - Writing intermediate results back to memory caused cache thrashing

PyTorch is too large to directly embed into our engine. It would take too much effort to trim the package down, and the result would likely not run optimally either.

We decided to implement network evaluation ourselves specifically for the hardware that we'll be running on.

[click]

We also decided to try evaluating the entire network in a single shader and see how far conventional optimization methods take us.

[click]

In our early attempts, we started out processing one block per thread, since we thought that our networks are small enough. Unfortunately, it didn't work out.

[click]

We don't have enough space in registers or LDS to store intermediate results while keeping high occupancy.

[click]

And writing intermediate results back to memory caused cache thrashing.

[NOTES]

Embedding constants in the shader caused instruction cache thrashing. FMA instructions with embedded constants take 3 DWORDs. Regular VALU instructions are 1 or 2 DWORDs.

Analysis

- Processing one block per thread requires too much storage for intermediate data
- What if we process one block per wavefront?
  - What if we store vector across the lanes of one or more VGPRs?
    - Elementwise operations are trivial
    - Read: `WaveReadLaneAt()` or equivalent
    - Inefficient to write to a specific lane

[click]

Since we cannot process data on the finest granularity – one block per thread – we move on to the next level of granularity, which is wavefronts. Can we process one block per wavefront? Upon investigation, this opens up another way of storing the intermediate vectors,

[click]

Which is to store them **across** the lanes of one or more VGPRs. This means that we store the first element of the vector in the first lane of the VGPR, and the second element in the second lane of the same VGPR, and so on. We go to the next VGPR when there are more elements in the vector than lanes in a wave. Because the widest part of our networks are 128-wide, assuming 64 lanes per wave, we can store the intermediate results in just 2 VGPRs.

We don't need to worry about matrices since they are constants that we read from buffers, which means that we can store and read them however we like.

[click]

With this setup, elementwise vector arithmetic is trivial. Matrix-vector multiplication usually requires reading and writing to arbitrary elements.

[click]

Reading from a lane is simple and fast: We can use WaveReadLaneAt() or equivalent, but it's limited to the same lane for the entire wave.

[click]

It's inefficient to write to a specific lane. But, as we'll see, these drawbacks can be easily worked around.

[NOTES]

There are two ways to formulate the multiplication:

1. Each element of the output vector is the dot product of one *row* of the matrix with the input vector.

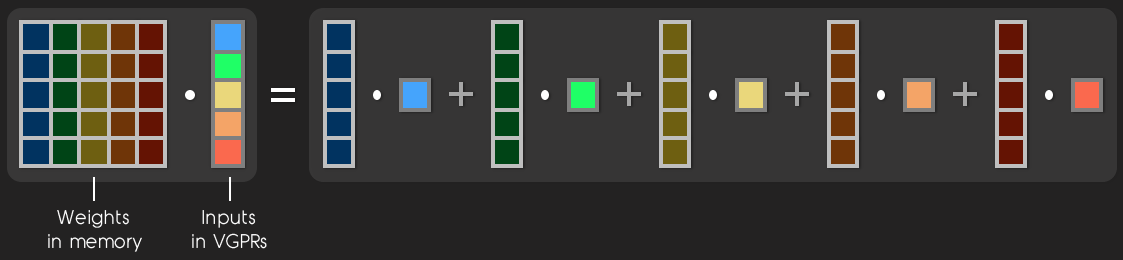
We'll need to use CrossLaneAdd() and write the result back to a specific lane which is not efficient.

2. The output vector is the sum of each *column* of the matrix times the corresponding element in the input vector.

Each element of the input vector can be easily and efficiently extracted using ReadLane(), and the summation maps naturally to the addition of regular variables that are stored in VGPRs.

Matrix Multiplication

- Sum of vector-scalar products



We can reformulate matrix multiplication by evaluating it as the sum of a series of vector-scalar products.

For each iteration, we read one column of the matrix from memory, and fetch the corresponding element of the input vector by reading the corresponding lane. It's then trivial to multiply them together and accumulate into a new VGPR.

Matrix Multiplication: Pseudocode

```
1  static const uint kNumLanes = 64;
2  VgprPack multiply(Matrix mat, VgprPack vec) { // Assumes mat.width == vec.size
3      VgprPack result = (VgprPack)0;
4      const uint laneID = WaveGetLaneIndex();
5      for (uint elemIn = 0; elemIn < vec.size; ++elemIn) {
6          float input = WaveReadLaneAt(vec[elemIn / kNumLanes], elemIn % kNumLanes);
7          for (uint vgprOut = 0; vgprOut * kNumLanes < mat.height; ++vgprOut) {
8              float parameter = mat[/*x=*/elemIn][/*y=*/vgprOut * kNumLanes + laneID];
9              result[vgprOut] += parameter * input;
10         }
11     }
12     return result;
13 }
```

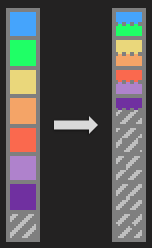

And here is the pseudocode for it.

This is the most basic version of what can be achieved using this method: we're reading one element on line 6 and updating one element on line 9. This method allows for a lot of flexibility in how many elements can be processed in one iteration – we can independently adjust line 6 and line 9 to process 'blocks' of the matrix. We'll see how this helps with optimization later.

This initial naïve implementation upsamples one 2k by 2k image to 4k in 42.7 milliseconds.

Float16 (half)

- Pros
  - Reduces VGPR usage
  - Reduces the size of the weights buffer
  - Speeds up computation with packed 16-bit operations
    - Must process 2 elements per instruction
- Cons?
  - We didn't run into any precision issues for our use case
  - More wasted computation
    - Reducing wave size (switching to wave32) helps mitigate this issue



One of the simplest optimization is adopting 16-bit floats, also known as half-precision floats or halves.

[click]

This change reduces VGPR usage…

[click]

…and reduces the size of the weights buffer and relieves cache thrashing.

[click]

It also speeds up computation with packed 16-bit operations because they effectively double the amount of computation we can perform per cycle.

[click]

Though, packed operations must process 2 elements at the same time. We'll come back to this later.

[click]

Using halves does bring some precision concerns,

[click]

But in our case, we didn't observe significant changes in the results.

[click]

Another potential downside is wasting computation. This is an inherent issue with this approach but using halfves compounds it. Assuming that wave size stays at 64, the network width now needs to be a multiple of 128 to avoid wasting computation. And that's the widest part of our networks.

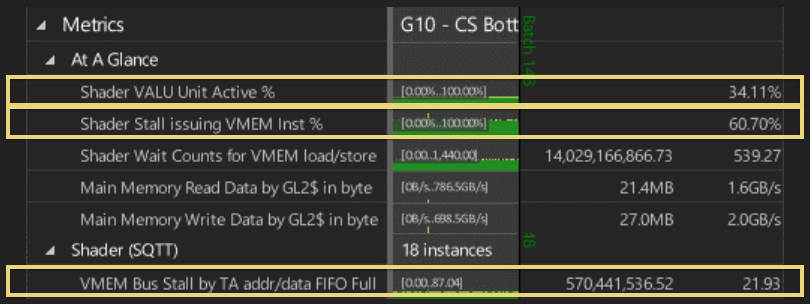
You can see an illustration on the right, where the stripe pattern indicates unused registers.

[click]

However, we can reduce the number of lanes by half by switching to wave32, and achieve roughly the same amount of waste as before.

Overall, there's very little downside to switching to halves.

Profiling Results



Low VALU usage

I/O request queue full

This version upsamples one 2k image to 4k in 37.1 milliseconds. It's a decent improvement but not as large as it theoretically should.

[click]

ALU utilization turned out to be lower than the previous version.

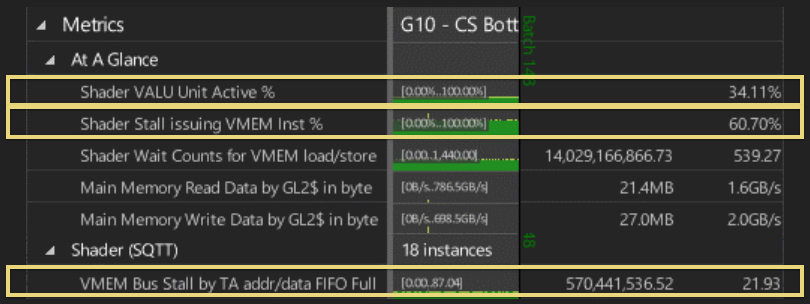
[click]

We can see what's wrong: we have a lot of stalls from waiting to *issue* memory loads,

[click]

because the shader is frequently stalled due to the queue for memory requests being full.

Profiling Results



Waiting to **issue** load instructions

I/O request queue full

This version upsamples one 2k image to 4k in 37.1 milliseconds. It's a decent improvement but not as large as it theoretically should.

[click]

ALU utilization turned out to be lower than the previous version.

[click]

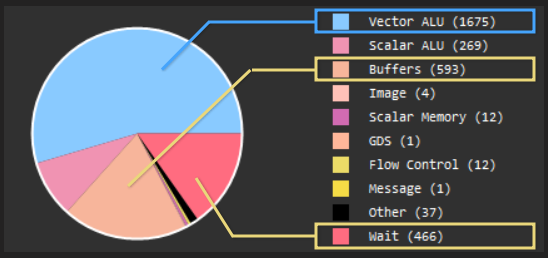
We can see what's wrong: we have a lot of stalls from waiting to *issue* memory loads,

[click]

because the shader is frequently stalled due to the queue for memory requests being full.

Bandwidth Issues: Memory Read Requests

- We're issuing too many load instructions



- Use explicit `buffer_load_dwordx4` intrinsics to load 8 `half`'s at once
  - Accommodate this by processing 4 rows per loop

From a macroscopic perspective, this is confirmed by the instruction statistics chart of the shader.

[click]

Only a bit over a half of the instructions are vector arithmetic – as indicated by the blue area – and that includes the instructions used for address computation.

[click]

At the same time, buffer loads and waits make up a third of the shader.

The assembly shows that the shader is only loading 32-bits per instruction.

[click]

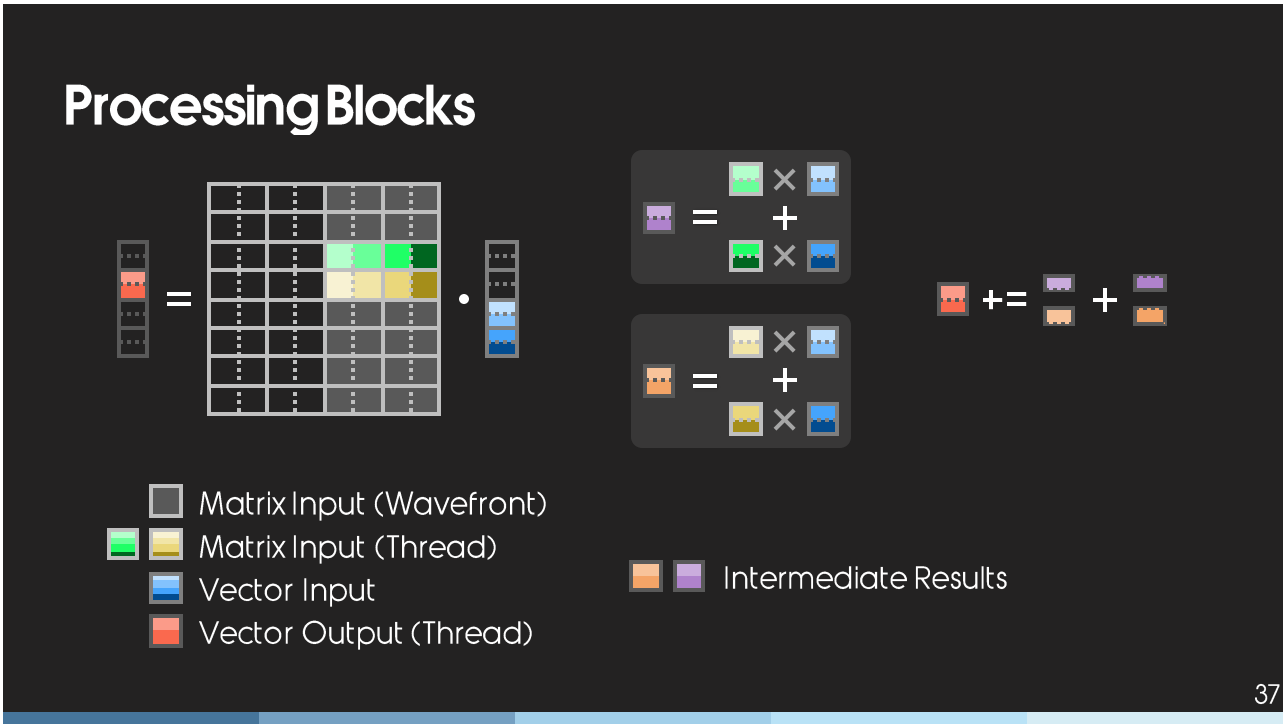
We can drastically cut down the number of load requests by loading as much data per instruction as possible.

On a PS5, we can use the buffer_load_dwordx4 instruction which loads

128 bits or 8 half's at once.

[click]

We accommodated this by switching to processing 4 rows and 2 columns per loop.



Here's how the matrix multiplication procedure is adjusted to accommodate these optimizations.

The light gray area in the matrix represents all computation performed by a wavefront in a single iteration. In this example it spans the entire height of the matrix, but it's not always the case when the matrix is large. The colored area represents computation performed by a single thread.

For clarity I'm representing the matrix with its conventional layout, but in practice it's converted to halves and packed in a way that the highlighted elements are adjacent in memory.

[click]

In the middle are packed operations for computing intermediate results.

[click]

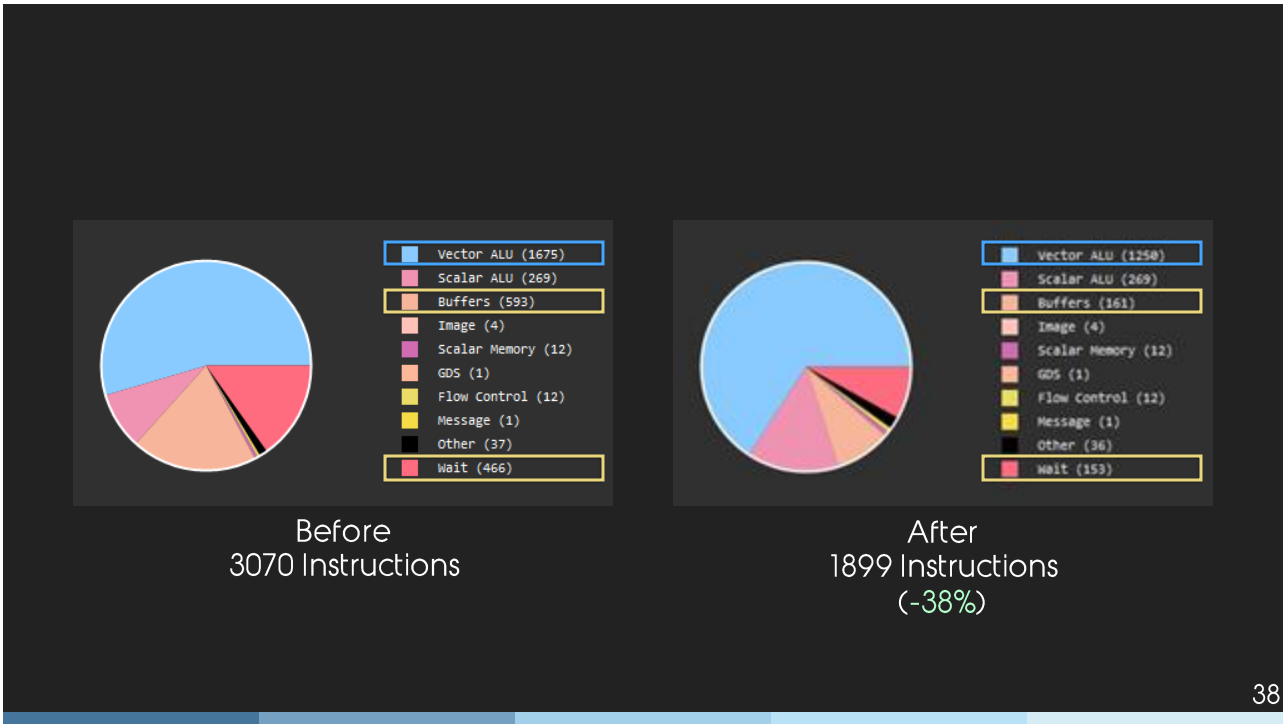
And on the right are the non-packed operations required to compute the final result.

These are inefficient, so instead of performing the accumulation every iteration, we use double the number of registers to store results from the second column until the entire sum has finished, before doing the non-packed sums on the third column.

[NOTES]

This setup of processing a 4-by-2 block of the matrix every thread is a result of trying to simultaneously minimize:

- The number of ReadLane()s that we need to do (to load the input vector)
- The amount of non-packed operations
- Number of registers needed to store intermediate results
- Number of parameter loads (max 8 halfs at a time)



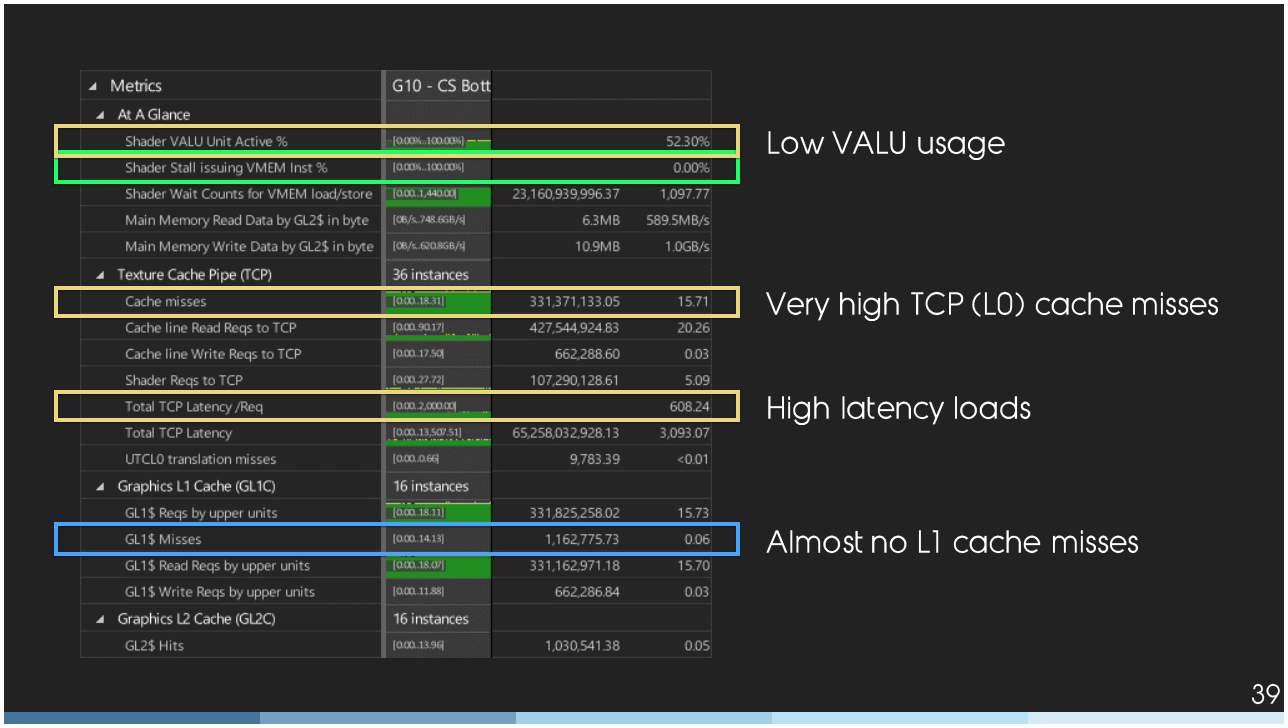
Looking at the stats again, we now have a much healthier portion of arithmetic instructions, shown in blue,

[click]

and a lot less waits and buffer loads.

Not only that, but the shader is also around 38 percent shorter than the previous version.

This version upsamples one 2k image to 4k in around 18.5ms.



If we profile this version, the stalls are gone, but two things are apparent:

[click]

Firstly, ALU usage is still low at just over fifty percent.

[click]

And second, the reason why it's low is because of memory loads: On average each load costs around 600 cycles.

[click]

In addition, we can see that we have a lot of L0 misses, but not a lot of L1 misses, which is an indication of L0 cache thrashing.

Bandwidth Issues: L1 Thrashing

- Use LDS to offload a portion of the bandwidth
  - LDS is cleared between thread groups
  - LDS is only shared within a thread group
- Use large, long-running thread groups
- Bonus: Splitting the network between LDS and L0 improved performance

[click]

And we can reduce the amount of data loaded from L1 by loading part of the weights into LDS beforehand. This presented some subtle challenges that we had to solve.

[click]

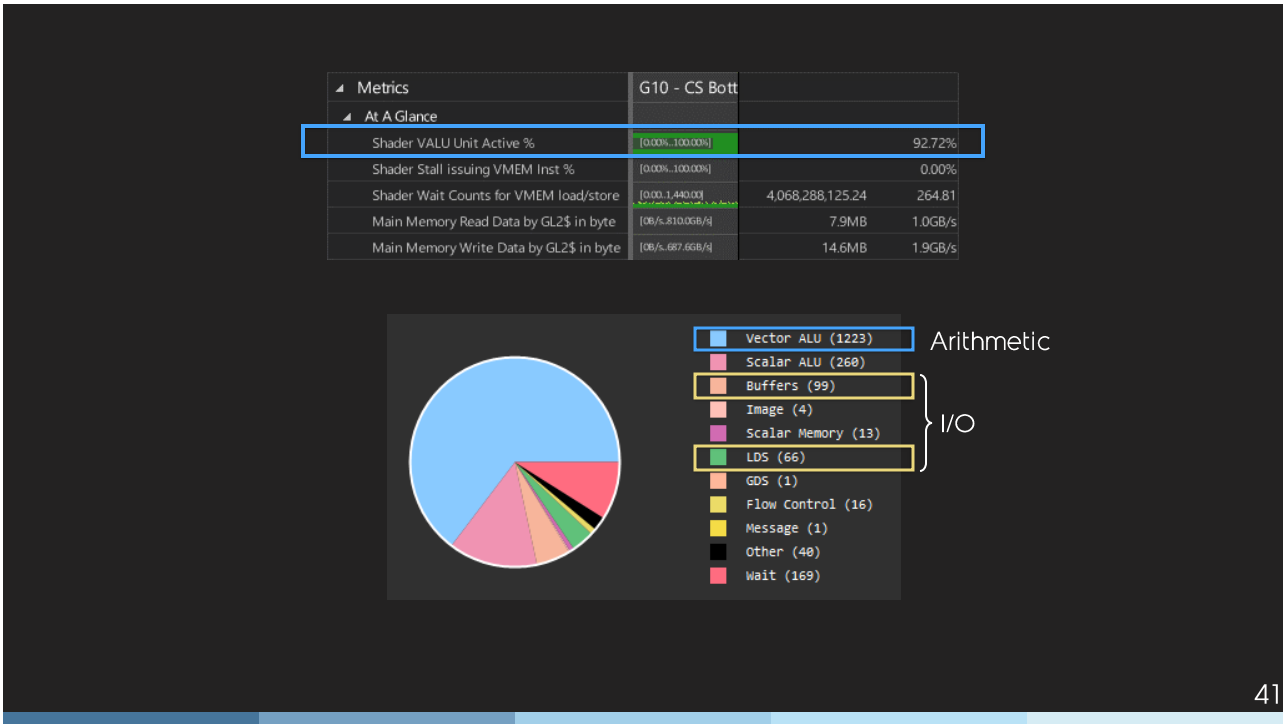
LDS is cleared between thread groups, and is only shared within thread groups and not between them. This means that our initial approach of launching new thread groups each containing a single wavefront every time we evaluate the network will not work.

[click]

The solution is to saturate the GPU with a batch of large thread groups, where we load the weights first, then loop through all inputs and repeatedly execute the same network. We need to avoid idle wavefronts or reduced occupancy caused by having too many threads in a group, which requires careful balancing and is highly hardware dependent.

[click]

Most of our networks don't fit into LDS, but the mode prediction networks turned out to be small enough to fit. Interestingly, when we tried loading all their parameters into LDS, it caused a noticeable performance drop, compared to when the parameters are split between LDS and cache. Our theory is that data can be loaded from LDS and L0 concurrently by different waves.



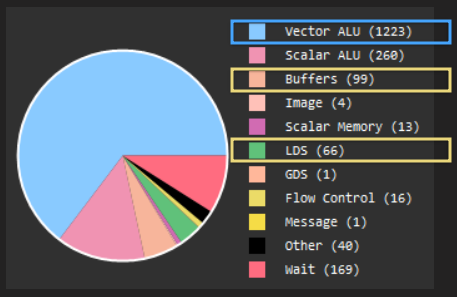
At this point, the shaders have over 92% VALU utilization, and upsampling a 2k image to 4k takes around 10.8 milliseconds. It's in a decent state.

[click]

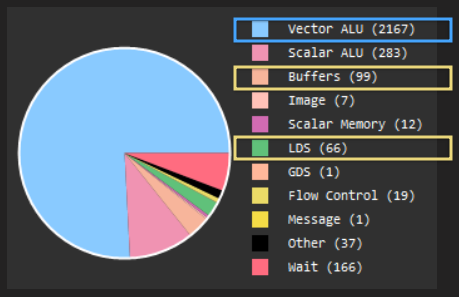
Looking at the composition of the shader, we have a lot of arithmetic instructions and very little I/O, which is exactly what we want.

More Performance

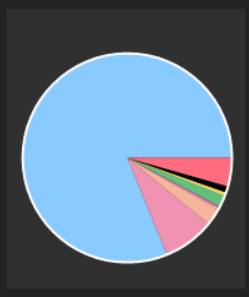
- Increase the number of blocks per network evaluation



1 Block
92.72% VALU
10.82 ms



2 Blocks
96.6% VALU
9.53 ms



3 Blocks
97.79% VALU
9.14 ms

42

But we're not done here.

Since our method is VGPR-efficient, it turned out that we can process more blocks at a time without affecting occupancy. This means that we'll be reading the weights once but using it to process multiple blocks.

[click]

When processing 2 blocks at a time, it upsamples one 2k texture to 4k in around 9.5 milliseconds, shaving off over a whole millisecond.

[click]

We have the same number of I/O instructions,

[click]

but almost double the number of VALU instructions. This is what we shipped in the final game.

[click]

As I was preparing this presentation, I realized that it's possible to bump this number up to 3 and get yet another small performance gain of around 0.4 milliseconds. But as shown in the stats, we're starting to get diminishing returns.

[NOTES]

VALU numbers here don't include warming up – i.e., loading things into LDS and warming up the cache. Compared to the upsampling process it usually takes a negligible amount of time.

Implementation

- Use SALUs for encoding "in parallel" with network evaluation
  - SALUs have little floating-point capability, so quantize beforehand
- Generated shaders
  - Perform basic optimizations on the network ahead of time
- The texture streaming system sends requests to the upsampling system and retrieves finished requests every frame
- CPU determines how many blocks to upsample
  - Track running averages of excess frame time and duration of one evaluation of any specific network

[click]

We encode network evaluation results into BC7 blocks with scalar ALUs to exploit the fact that we're operating in waves rather than threads.

[click]

The floating point results are quantized to integers with vector ALUs beforehand to work around the fact that scalar ALUs barely have any floating-point capabilities.

Thanks to the system being designed to handle each submode with a unique shader, almost all constants and table lookups can be inlined, and loops can be unrolled.

This results in extremely optimized encoding code that runs "in parallel" with network evaluation.

[click]

We use code generation to produce the shaders from dumped PyTorch models.

[click]

We disassemble network nodes into more primitive operations. We also define rules of optimization and run them over this intermediate representation.

[click]

Every frame, the texture streaming system detects textures that require upsampling, and sends them over to the upsampling system. It also retrieves finished requests from previous frames.

[click]

The way we determine how much BC7 blocks to upsample is by tracking running averages of excess frame time and the duration of one evaluation of any specific network. We also try to be conservative by clamping excess frame time down to last frame's value, and by only using a percentage of the budget.

This simple approach worked well enough in practice.

[NOTES]

If an elementwise multiply and an elementwise add immediately follows a matrix multiplication and an elementwise add, the multiply-add can be folded into it.
$\mathbf{C}(\mathbf{M}\boldsymbol{x} + \boldsymbol{b}) + \boldsymbol{d} = (\mathbf{CM})\boldsymbol{x} + (\mathbf{C}\boldsymbol{b} + \boldsymbol{d})$, where $\mathbf{C} = \mathrm{diag}(\boldsymbol{c})$

Although performant, the way we evaluate the networks has a few limitations.

[click]

Firstly, as we've already mentioned, when the width of the network is not a multiple of wave size, some computation is wasted. It's easy to enforce this for hidden layers but input/output layers usually have a fixed size.

[click]

And secondly, when evaluating larger models, we quickly hit limits imposed by cache size and VGPR count. It may be possible to work around this issue by creating a hybrid where some intermediate results are saved to memory/LDS.

[click]

The system is heavily optimized for PS5 and will likely perform poorly without adjustments on different hardware.

Conclusion & Future Work

- Room for quality improvements
  - Assessing quality in-game is important
- Used primarily for normal maps, but extensible to other texture types and compression methods
- No additional compression step with the help of the mode predictor network
- High performance and adaptive.

Let's conclude this talk by looking back at the initial list of goals that we started with and look at how we may improve the system.

[click]

The system succeeds in increasing texture resolution on PS5, but we would like to make quality improvements in the future. We also learn the lesson that it's important to assess quality in-game early in production, in addition to looking at raw numbers and textures.

[click]

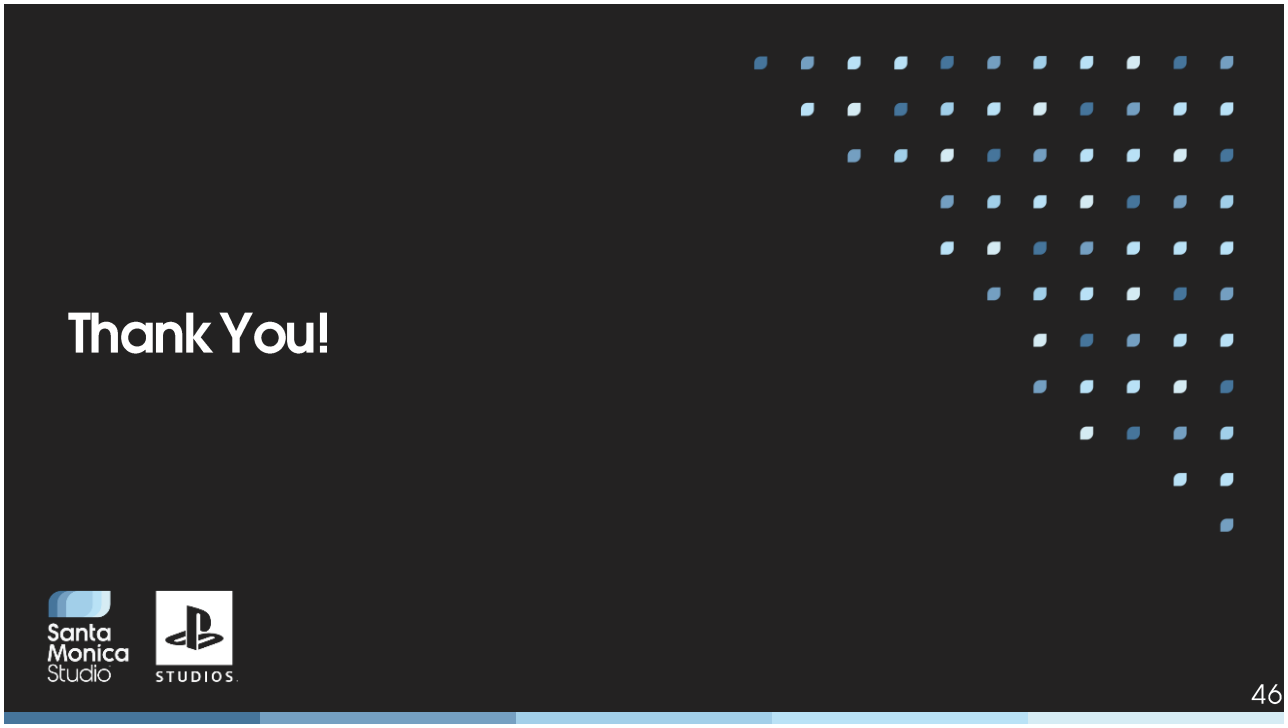
We primarily use this method for normal maps which are compressed exclusively using BC7, but it's easy to extend this method to other texture types and compression methods.

[click]

We're able to avoid an additional compression step, although we do need help from the extra mode predictor network. We considered switching to a rule-based mode predictor for the specific submodes that we use but never got around to it.

[click]

And by optimizing specifically for the PS5, we were able to almost fully utilize the hardware while keeping the system extremely adaptive.



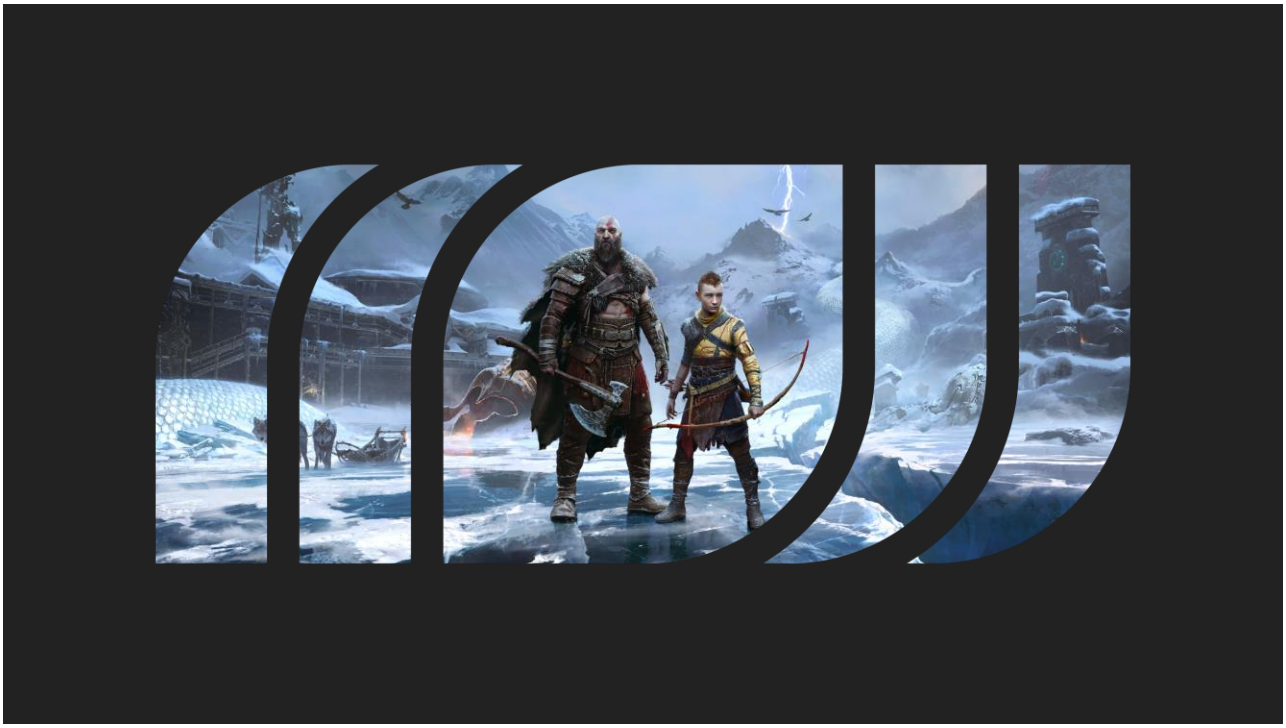
That's it! Thank you for attending my talk.

I would like to thank our technical director Josh Hobson and our rendering lead Stephen McAuley for providing ideas and feedback for this project, my advisor Olivier Pomarez for helping with the presentation, and all members of the rendering team for their help and support.

Here are all talks from Santa Monica Studio at this year's GDC. Feel free to check them out.

And finally, we're hiring!

I will be happy to take any questions you may have.



In-Game Results

Upsampling OFF

Upsampling ON

Upsampling OFF

Upsampling ON

Upsampling OFF

Upsampling ON

Upsampling OFF

Upsampling ON

Upsampling Normals

- Two approaches for representing normal: (x, y, α) or (x, y, z)
- Second approach resulted in higher PSNR in our testing
  - Stretch endpoints to at least unit length to produce valid normal maps

We can either represent normals with a direction in the X-Y plane and its angle against the Z axis, or a basic 3D vector.

The first approach guarantees that the endpoints are normalized while the second version doesn't. We eventually went with the second version since it resulted in higher PSNR. We manually stretch the endpoints to slightly longer than unit length before encoding.

P-bit Handling

- Previously we've set all P-bits to 0
  - Makes it impossible to have pixels valued 255
- We already have higher-precision floating point values
  - Quantize using more bits
  - Try different P-bit values to see which one results in the least error

Not being able to have 255 values has a significant implication for normal maps, since our shading depends on the length of the normal vectors.

In mode 1, normals are quantized using 6 bits. (0, 1, 0) will be converted to (0.5, 1, 0.5), which will become (63/127, 126/127, 63/127) assuming all P-bits are set to 0, which will be decoded as (-1/127, 125/127, -1/127), which has length 15627/16129≈0.968876.

The SALU trick still applies during the process – we're quantizing first before doing all the testing with integers. We ended up not losing any performance.

P-bit Handling: Quantization

- Suppose we have an n-bit quantized unsigned integer a_n, that we want to convert into an m-bit quantized unsigned integer a_m $(m < n)$
  - We'd like to do this without floating-point operations, and without overflowing
- $a_m = \left\lfloor \frac{2^m-1}{2^n-1} a_n + \frac{1}{2} \right\rfloor = \left\lfloor \frac{(2^{m+1}-2)a_n + 2^n - 1}{2^{n+1}-2} \right\rfloor$
  - Approximations:
    - $a_m = \left\lfloor \frac{(2^m-1)a_n + 2^{n-1}}{2^n} + \frac{(2^m-1)a_n}{2^n(2^n-1)} \right\rfloor = \left\lfloor \frac{a_n + 2^{n-m-1}}{2^{n-m}} - \frac{(2^{n-m}-1)a_n}{2^{n-m}(2^n-1)} \right\rfloor$
      - $\frac{(2^m-1)a_n}{2^n(2^n-1)} < \frac{1}{2^{n-m}}$, $\frac{(2^{n-m}-1)a_n}{2^{n-m}(2^n-1)} < 1$

57

Care must be taken when converting from quantizing using m bits and quantizing using n bits: Because we're handling ranges [0, 1] instead of [0, 1-1/2^k], it's not a simple matter of truncating the binary representation or padding with zeros.

For example, if we were to requantize a unorm represented by 2 bits to 3 bits by padding with zeros, we would get the results shown in yellow, and the last 2 are obviously incorrect (correct values are shown in green).

Since integer division rounds down, and both the numerator and the denominator are integers, we can avoid involving floating-point operations. Unfortunately, there's the possibility of overflowing roughly when $m + n \geq 32$.

I've shown two ways to transform the expression, from which approximations can be obtained by taking only the first term. Also shown are the bounds of their errors. Both approximations avoid the division, but only the second one helps avoid overflows.

Additionally, a lot of n and m pairs cover small enough values that we can iterate over all possible values and check if the approximations are

exact.

Experiment: Gloss Adjustment

- Apply the inverse of Toksvig's method to upsampling
  - https://developer.download.nvidia.com/whitepapers/2006/Mipmapping_Normal_Maps.pdf
- Store alpha gain in the alpha channel of the texture
- Did not make it into the final game

Towards the end of the project, we experimented with adjusting gloss together with upsampling normals. We use Toksvig's method to adjust gloss when using mipmapped normals based on the lengths of the normals, and the inverse should apply when upsampling the image.

It would be difficult to modify gloss directly because it's packed together with other channels. Thus, we decided to encode it in the alpha channel of the normal map. We were not previously using the channel and we're going to sample normal anyway, so this was an obvious choice. Modifications to the network are minimal – we're just outputting an additional channel.

Due to time constraints this did not make it into the final game.