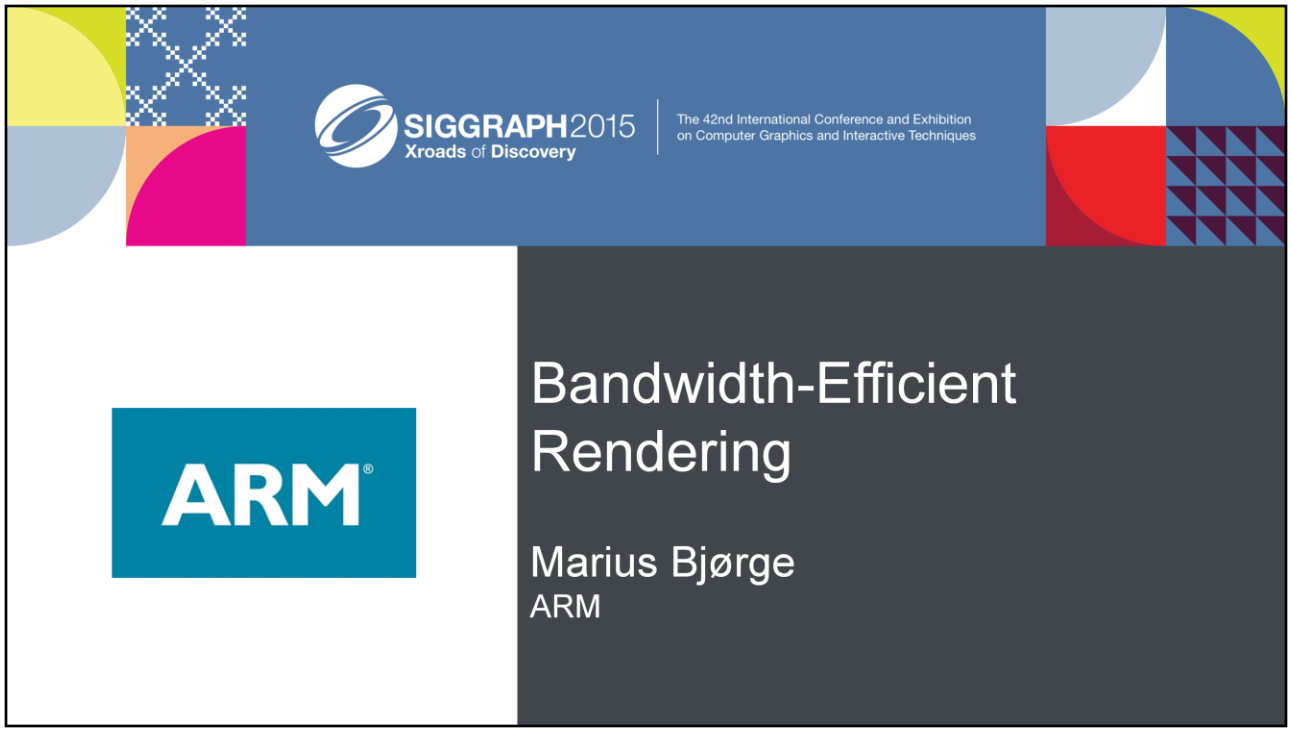

SIGGRAPH2015
Xroads of Discovery


SIGGRAPH2015
Xroads of Discovery
The 42nd International Conference and Exhibition
on Computer Graphics and Interactive Techniques
ARM
Bandwidth-Efficient Rendering
Marius Bjørge
ARM

Agenda

- Efficient on-chip rendering
- Post-processing
  - Bloom
  - Blur filters

This is the agenda for this talk.

The main focus on this talk is about bandwidth-efficient rendering. So first I'll talk a bit about on-chip rendering and how that can help reduce memory bandwidth and increase performance.

Then I'll talk about the challenges of post-processing in mobile, and will specifically talk about bloom and blur filters and how to optimize that with bandwidth-efficiency in mind.

Efficient on-chip rendering

- Extensions
  - Framebuffer fetch
  - Pixel Local Storage
- Why extensions?
  - Surely mobile GPUs are already bandwidth-efficient?

So, first on-chip rendering.
In order to enable more efficient on-chip rendering we have introduced a couple of extensions: framebuffer fetch and pixel local storage.

But why extensions? Aren't mobile GPUs already bandwidth efficient?

Historically the graphics APIs have been designed around the premise of immediate-mode architectures, and as such a lot of the benefits of tile-based architectures have been left out of the core APIs – that's why we've introduced these extensions, and that's why we encourage developers to use them.

Framebuffer fetch

- Read the current fragment's previous color value
- ARM also supports reading the previous depth and stencil values of the current fragment

- Useful for
  - Programmable blending
  - Programmable depth/stencil testing

Framebuffer fetch is a fragment shader extension that allows you to read the previous color value. We've also got a similar extension for reading the previous depth and stencil values.

This is useful for programmable blending and programmable depth/stencil testing.

Pixel Local Storage (PLS)

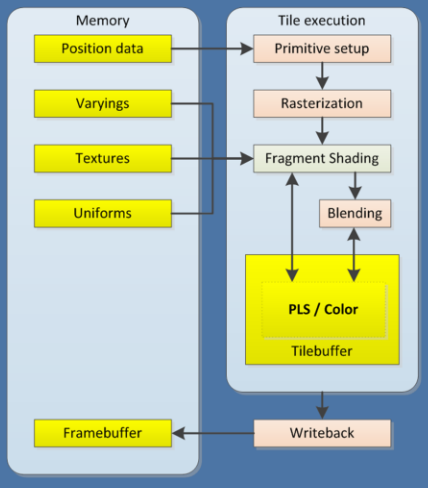
- Per-pixel storage that is persistent throughout the lifetime of the frame
  - Read/write access
  - Storage stays on-chip
  - Storage layout declared per fragment shader invocation – does not depend on framebuffer format
- Useful for
  - Deferred shading
  - Order Independent Transparency [1]
  - Volume rendering

Pixel Local Storage, or PLS, is a general purpose storage that is persistent throughout the lifetime of the frame. You can read and write to it, and the storage stays on-chip. Storage layout is declared per fragment shader, so there's no dependency on the current framebuffer format. This also makes it easy to re-interpret PLS values between fragment shader invocations.

PLS is useful for things like deferred shading, order independent transparency and also volume rendering.

Pixel Local Storage (PLS)

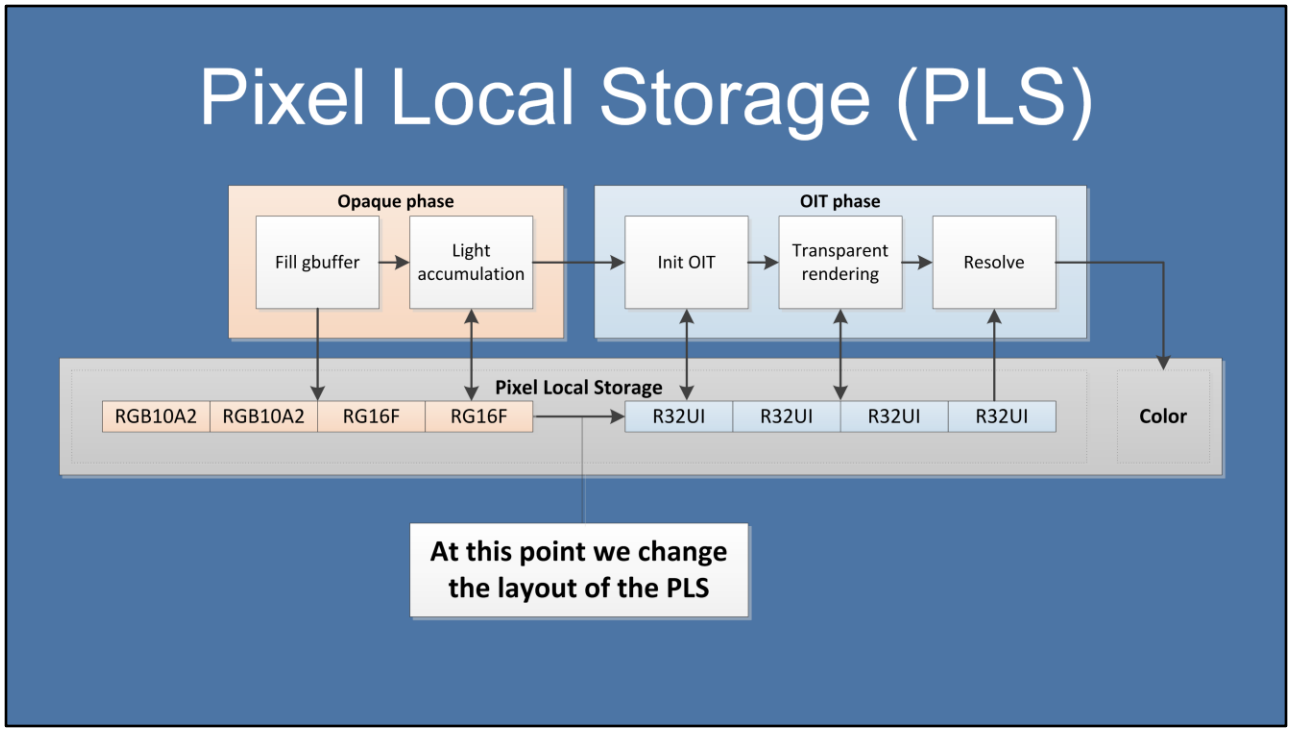
- Rendering pipeline changes slightly when PLS is enabled
  - Writing to PLS bypasses blending
- Note
  - Fragment order
  - PLS and color share the same memory location



A bit simplified, this is what the rendering pipeline looks like when using PLS.

There are two paths from the fragment shader to the tilebuffer - one through the fixed function blender and one that allows direct read/write access to the tilebuffer.

(Note that all operations on the PLS happen in fragment submission order. Also PLS and color share the same memory location – so writing to one of them will discard the other.)



One of the powerful features of PLS is that it allows you to re-interpret data between fragment shader invocations – so you can basically repartition your storage mid-frame. This makes it possible to chain things like deferred shading with order independent transparency.

But, one of the limitations of Pixel Local Storage is that access is restricted to the current pixel, so for doing complicated post-processing effects you need to rely on more conventional data flows. Which leads me to post-processing.

Post-processing

- High-end mobile devices typically have small displays with massive resolutions
- Rendering at native resolution is often out of the question, especially if you add post-processing to the mix

- Solution: mixed resolution rendering
  - Go as low as you can without sacrificing quality, and then upscale

High-end mobile devices typically have small displays with massive resolution, so rendering at native resolution is often out of the question – especially if you also want to do post-processing.

A solution to that is to do mixed resolution rendering. Render as low as you can without sacrificing visual quality, and then upscale.

Mobile post-processing

On-chip

- Color Grading
- Tonemapping

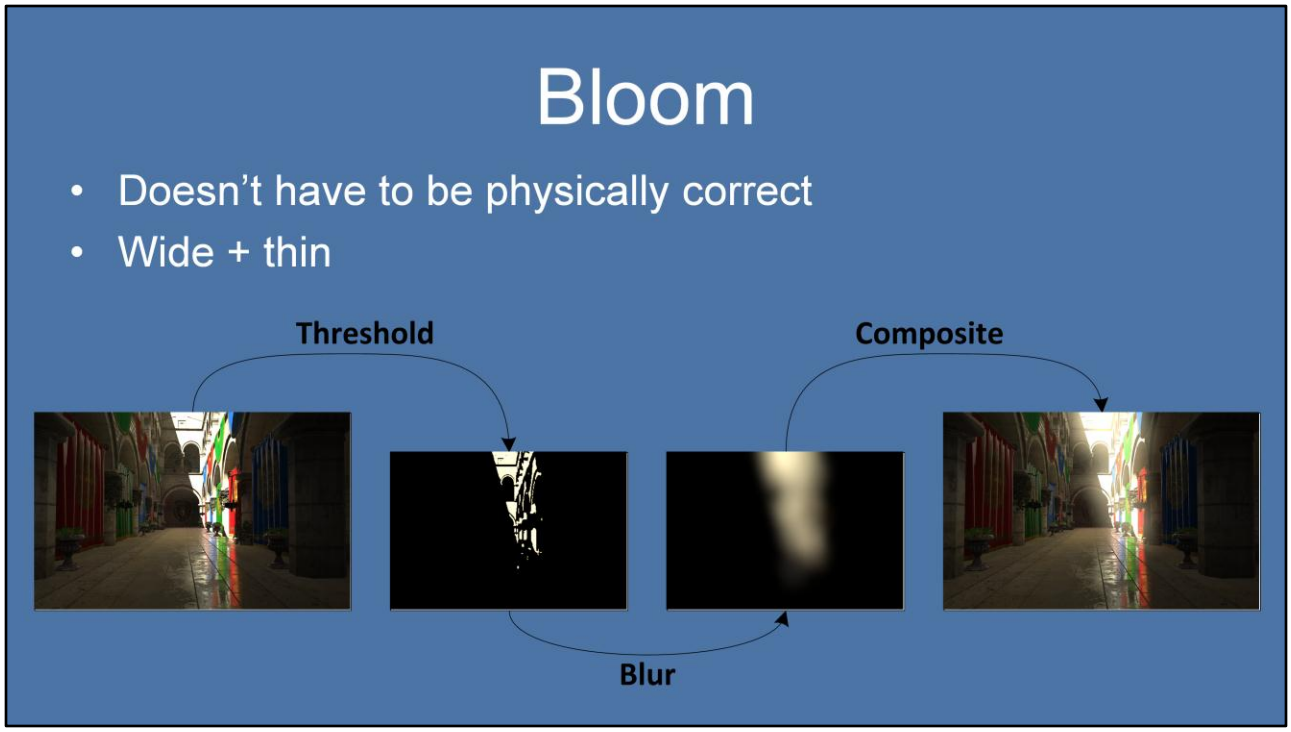
Off-chip

- Anti-aliasing
- Bloom
- Depth of Field
- Screen Space Ambient Occlusion
- Screen Space Reflections

These are the typical post-processing effects used in mobile content today.

Note that most AAA mobile games only use a subset of these post-processing effects, but we have seen engines that combine things like SSAO, anti-aliasing, bloom and depth of field in mobile.

Since PLS is limited to only accessing the current pixel, it will only benefit things like color grading and tonemapping. Other algorithms have more complex pixel dependencies which forces us to go beyond the current pixel.



So for this talk I chose to focus on Bloom, since it's quite the opposite of what you can do with PLS – it goes wide, it's bandwidth heavy and complex.

Bloom is often implemented in multiple passes by first extracting the brightest part of the rendered image, then blur this part and finally composite it all.

There's some overlap in these passes so we can combine for instance the threshold pass with the first blur pass in order to exchange some bandwidth for ALU. But it's the blur pass that's really interesting. Also, since the bloom itself doesn't really have to be physically correct we have some freedom to get creative with the blur algorithm.

Blur

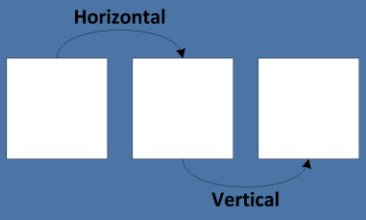
- What makes a good blur filter?
- Goal:
  - High quality
  - Stable
  - High performance

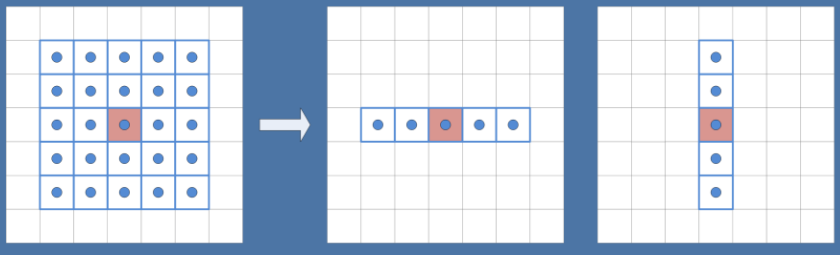
So what makes a blur filter good? Quality obviously plays an important role, but also stability. We don't want a blur filter that is unstable with slight changes in rendered image.

So the goal was "simple": come up with a high quality and stable blur filter with the least amount of samples. Quality for the bloom can also be highly subjective, so there is some headroom here.

Box blur

- 5x5 box blur = 25 samples
- Separate the blurs
  - 5 + 5 = 10 samples



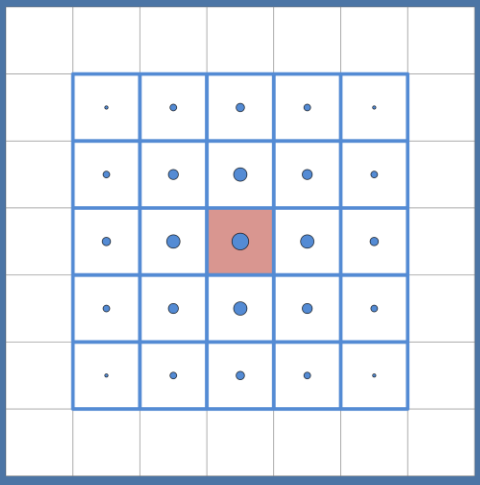


So first some quick background of different blur filters. The simplest one of all is the box filter. This filter basically just sums up all samples inside a “box” and stores the average.

One step further, we have the separable version of the box filter. This works by doing horizontal and vertical sums in separate passes, and this can greatly reduce the number of samples required. A 5x5 blur can for instance be reduced from 25 samples to just 10 – but at the extra cost of more write-out bandwidth.

Gaussian blur

- Convolve a gaussian function over the image
- Separable just like the box filter

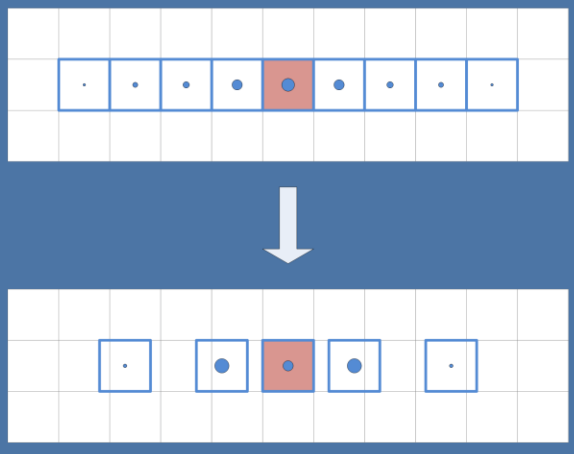


One of the problems with the box filter is that, since all the samples are given equal weight, the shape of the blurred output appears blocky. One way around that is to apply the box filter twice, but that requires more processing and more bandwidth. Another way could be to apply a circular shaped filter, but that wouldn't be separable, so you would have to do a lot of samples.

A better solution is to use gaussian blur. Gaussian blur works by convolving a gaussian function over the image – and just like the box filter, this filter is separable so you can reduce the number of samples by doing separate horizontal and vertical passes.

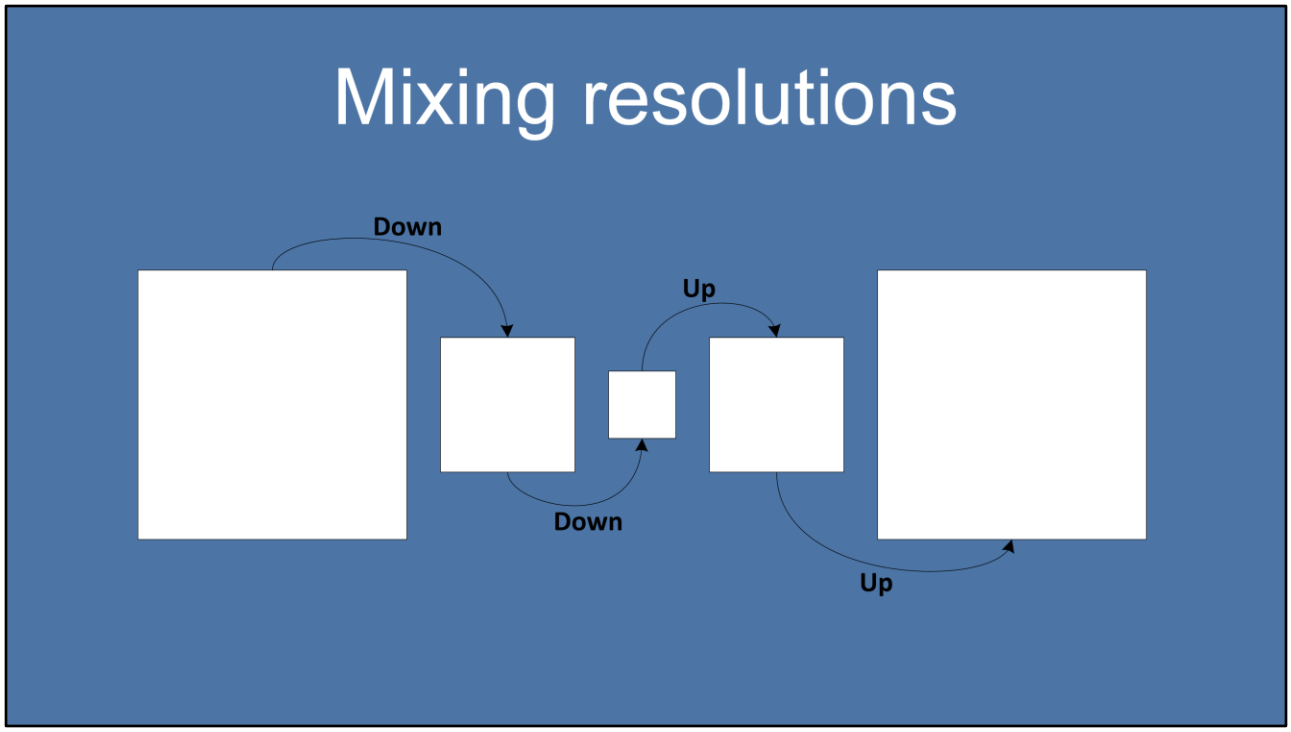
Linear sampling optimization [2]

- Reduce number of texture lookups by exploiting the HW texture unit
  - Modify sample offsets and gaussian weights
- Get 9x9 at similar cost as 5x5



One of the neat things with modern GPUs is that they have fixed function support for doing bilinear filtering. This is something we can exploit when filtering by offsetting the sample positions as well as the gaussian weights.

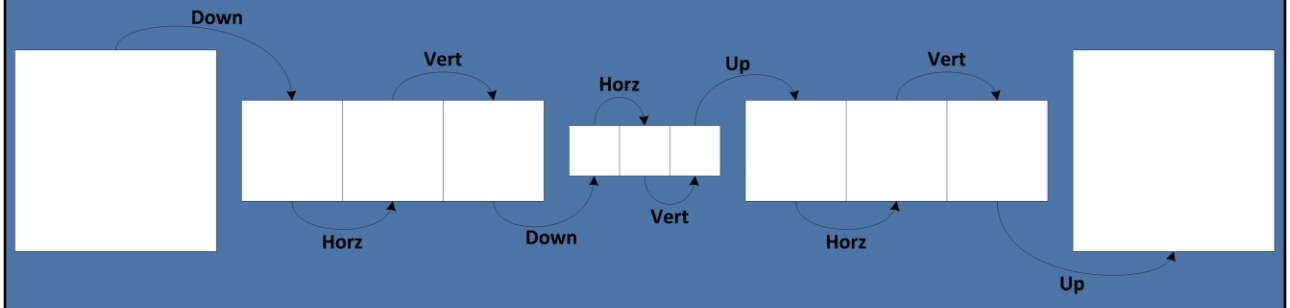
In this sample we have effectively reduced the number of required samples from 9 to 5, while at the same time keeping the visual quality.



But we can't do all that filtering at high resolution, so we need a way to downsample and upsample the image. The basic idea is to first downsample, and at each step make sure you propogate all pixel values to avoid instability – then apply the blur at reduced resolution before finally upsampling the image.

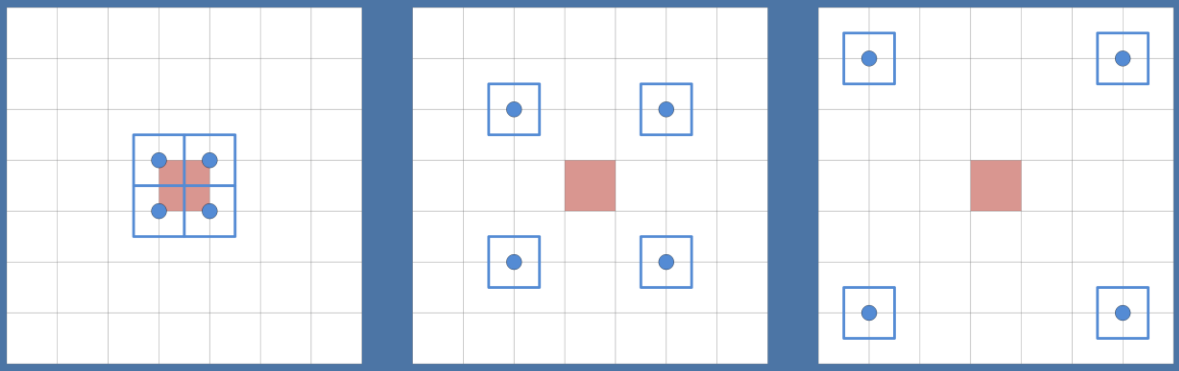
Mixing resolutions

- Gets increasingly complicated when using separable kernels

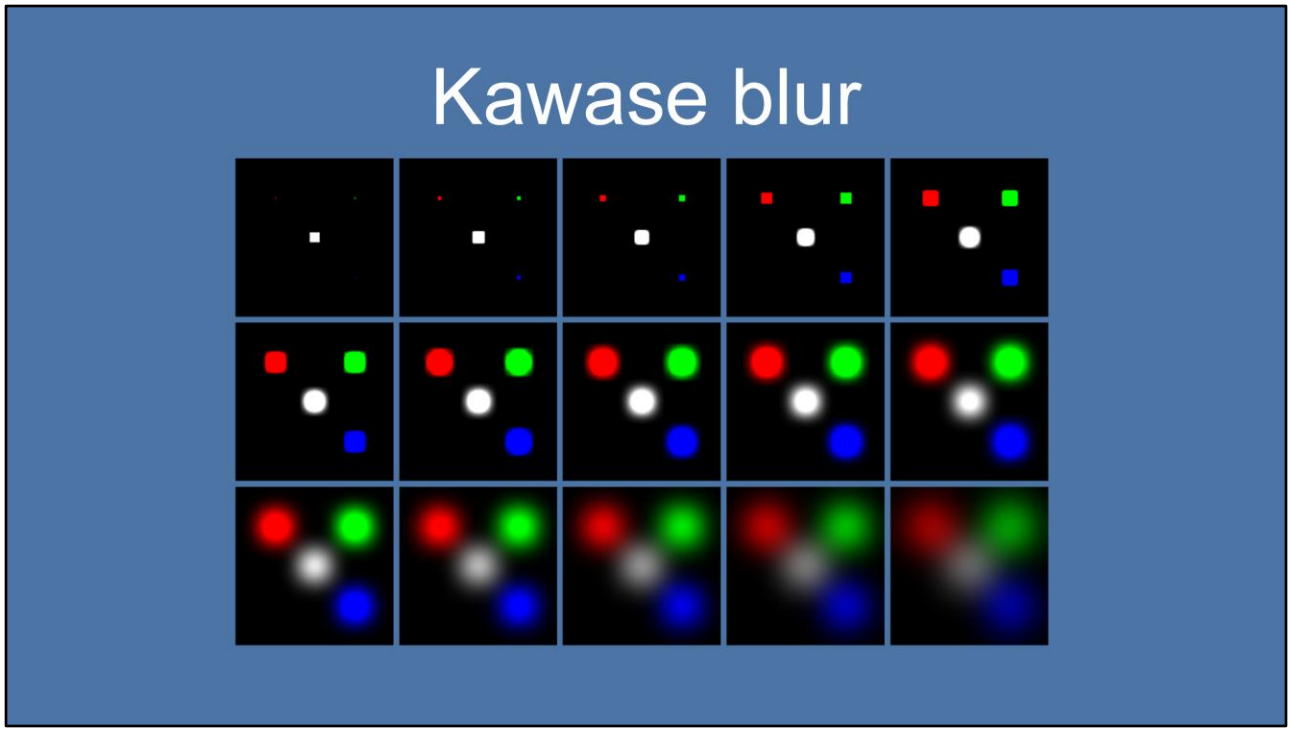


But this gets very complicated when using separable kernels as can be seen by this diagram.

Kawase blur [3]



The Kawase blur is a solution to that problem. It works in multiple passes, preferably at reduced resolution, by sampling four corners at increasing distance from the current pixel – basically ping-ponging between two equally sized textures.

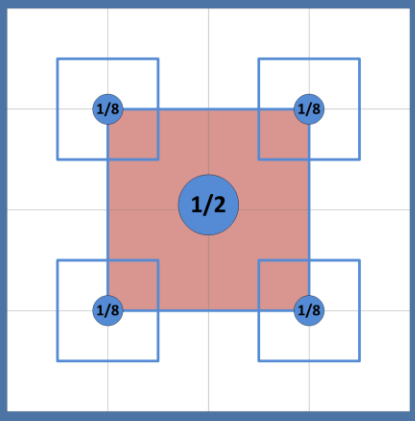


And this produces some very nice results. Here we have an input image in the top left corner – note that the red, green and blue pixels are really bright in this case.

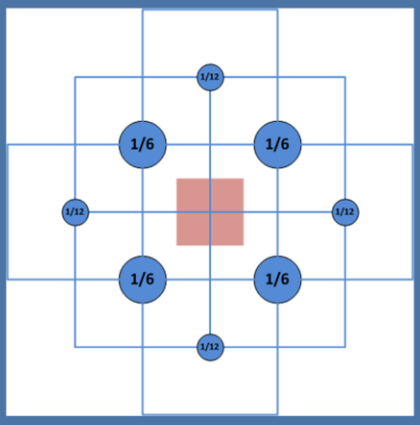
For each pass we sample further and further away from the center until we get the result we want.

“Dual filtering”

Downsample filter



Upsample filter



For lack of a better name, Dual filter, is something I came up with when playing with different downsampling and upsampling patterns. It is derived from the Kawase filter, but where Kawase ping-pongs between two equally sized textures, this new filter does downsampling and upsampling. “Dual filtering” uses different filtering kernels for the downsample and upsample passes – but the sampling distance is constant.

The downsample filter works by sampling four pixels covering the target pixel. Four samples on the corners are sampled in order to smudge in some information from all neighbouring pixels. This makes it a total of 5 samples - we can make use of bilinear filtering HW by sampling between the 4 samples.

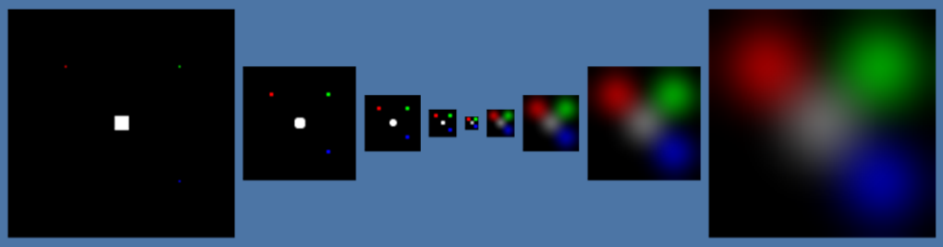
The upsample filter works by reconstructing information from the downsample pass. This pattern was chosen in order to get a nice smooth circular shape. Other shapes can also be used if you want a more artistic look. You can also tweak the distance from the target pixel to the sampling points to get a bigger or smaller blur.

```
vec4 downsample(vec2 uv, vec2 halfpixel)
{
  vec4 sum = texture(tex, uv) * 4.0;
  sum += texture(tex, uv – halfpixel.xy);
  sum += texture(tex, uv + halfpixel.xy);
```

```
  sum += texture(tex, uv + vec2(halfpixel.x, -halfpixel.y);
  sum += texture(tex, uv – vec2(halfpixel.x, -halfpixel.y);
  return sum / 8.0;
}

vec4 upsample(vec2 uv, vec2 halfpixel)
{
  vec4 sum = texture(tex, uv + vec2(-halfpixel.x * 2.0, 0.0));
  sum += texture(tex, uv + vec2(-halfpixel.x, halfpixel.y)) * 2.0;
  sum += texture(tex, uv + vec2(0.0, halfpixel.y * 2.0));
  sum += texture(tex, uv + vec2(halfpixel.x, halfpixel.y)) * 2.0;
  sum += texture(tex, uv + vec2(halfpixel.x * 2.0, 0.0));
  sum += texture(tex, uv + vec2(halfpixel.x, -halfpixel.y)) * 2.0;
  sum += texture(tex, uv + vec2(0.0, -halfpixel.y * 2.0));
  sum += texture(tex, uv + vec2(-halfpixel.x, -halfpixel.y)) * 2.0;
  return sum / 12.0
}
```

“Dual filtering”



And here’s the result of this filter. This algorithm gives us a blurred output that is equivalent to the output from the Kawase filter, but at much less cost.

Comparing filters

So next we're going to compare different aspects of these different filters.

Comparison setup

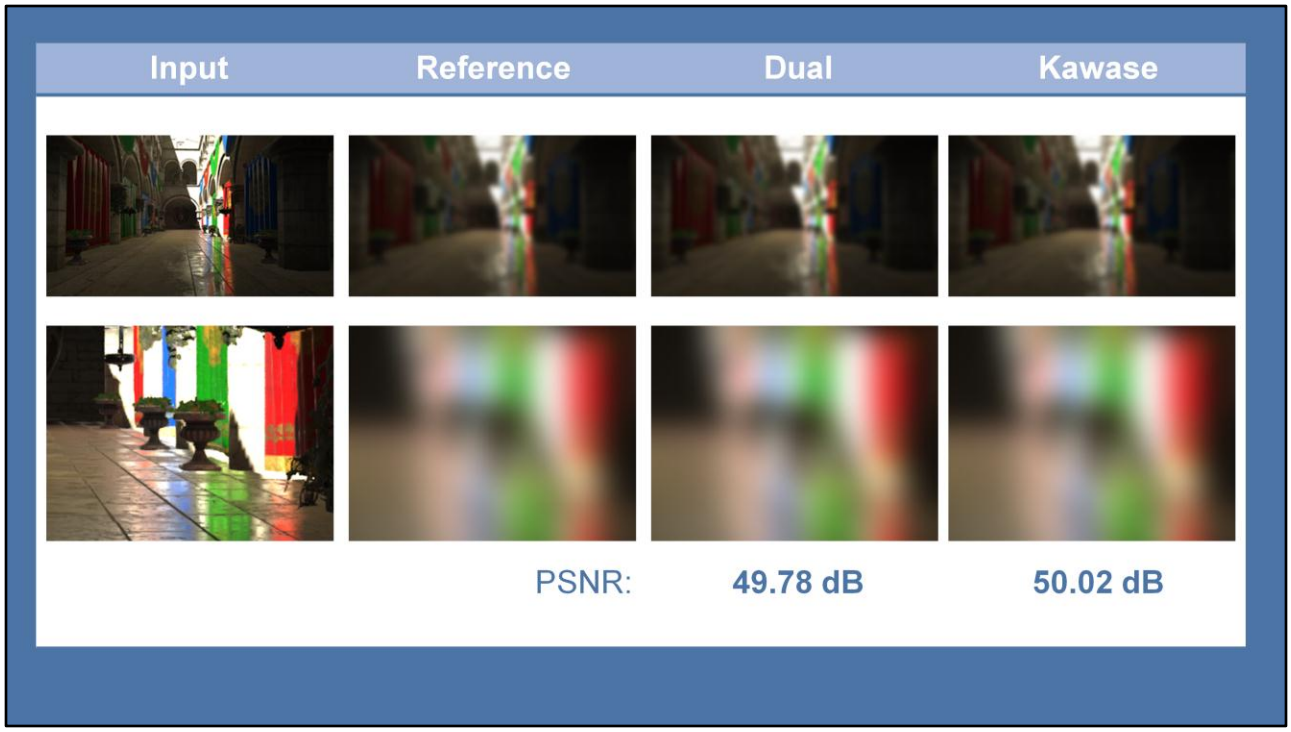
- 97x97 blur
- Gaussian used as reference
- Kawase
  - First downsample to 1/16th resolution
  - Setup with 0, 1, 2, 3, 4, 4, 5, 6, 7 distances passes
- “Dual filtering” setup with 8 passes
- Naïve method which relies on glGenerateMipmap

For this comparison I set up all the filters to do a 97x97 or equivalent blur, and gaussian is used as a reference.

The Kawase blur is setup to first downsample to 1/16th resolution, then it does 9 passes with increasing distances. You can notice that 4 is duplicated – I found that to produce better quality than just doing increments.

Dual filtering is setup to do 8 passes, so that means 4 downsample passes and 4 upsample passes.

The naïve method relies on mipmap generation to produce the blurred output.



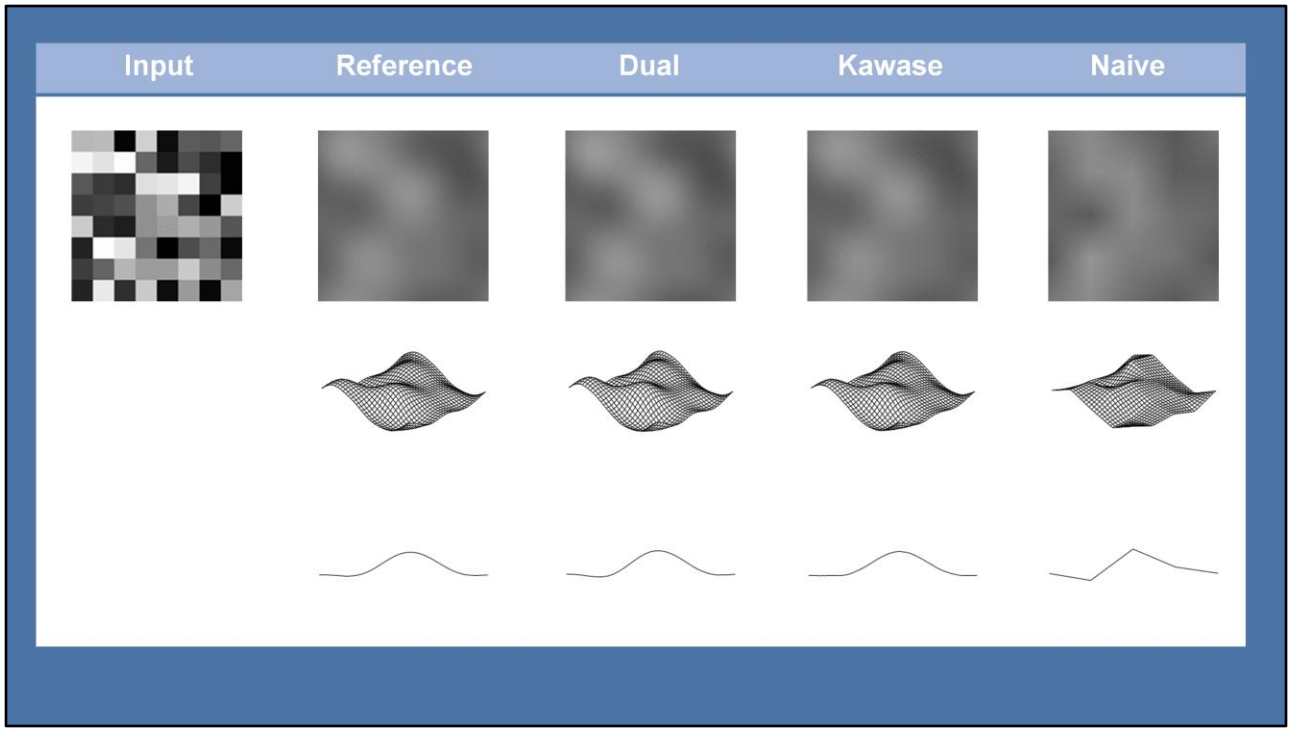
The input image here is a 1080p screenshot from our internal graphics engine.

The top row shows the whole image, while the bottom row shows a zoomed in part of the image. As you can see the output is very similar – only very small variations exist.

The peak-signal-to-noise ratio show that the Kawase filter is slightly better than “Dual” filtering compared to the reference – but these are very minor differences.

Stability comparison

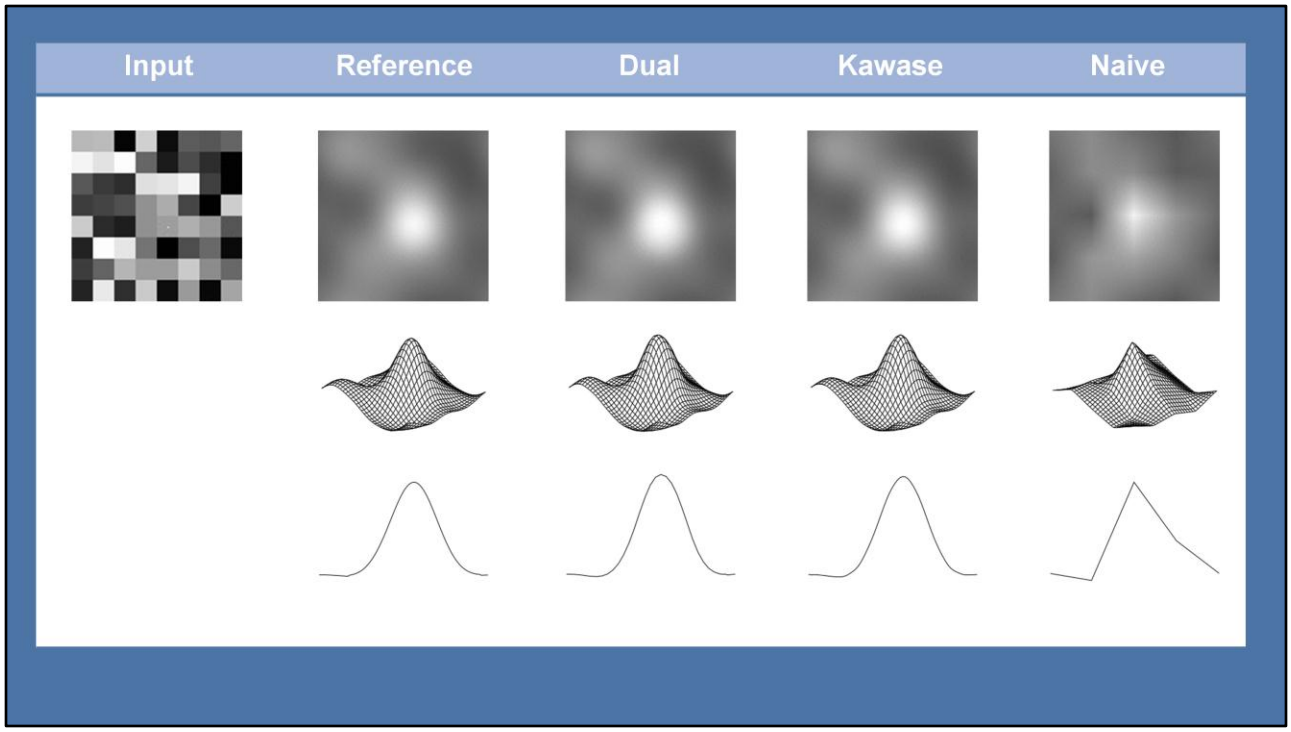
Next up is a stability comparison of the different filters.



The input image here is a blocky grayscale image that is translated diagonally using sub-pixel increments.

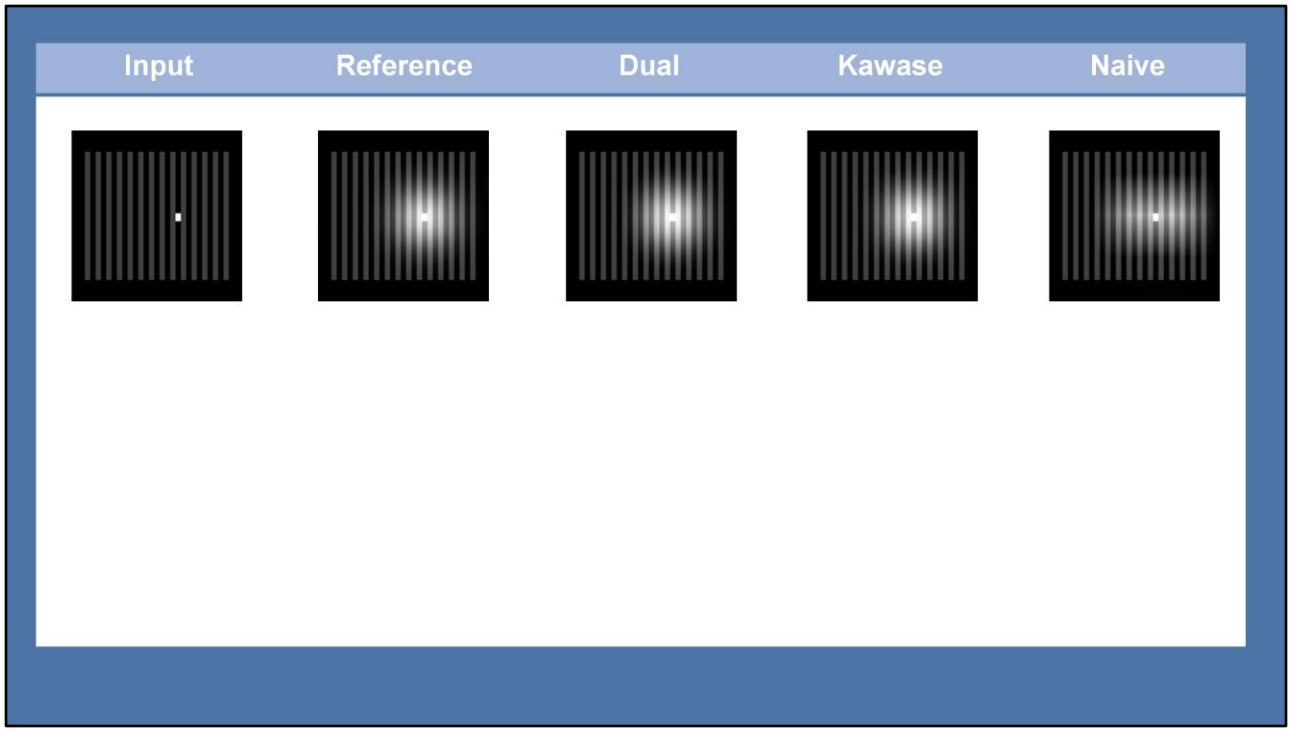
From the result of the different filters you can see they all perform really well in this case – all fluid movement.

Notice that the naïve mipmap generation implementation suffers from blocky artefacts here.



Next we have the same input image as the previous slide, but we've also added a single really bright pixel in the center of one of the blocks.

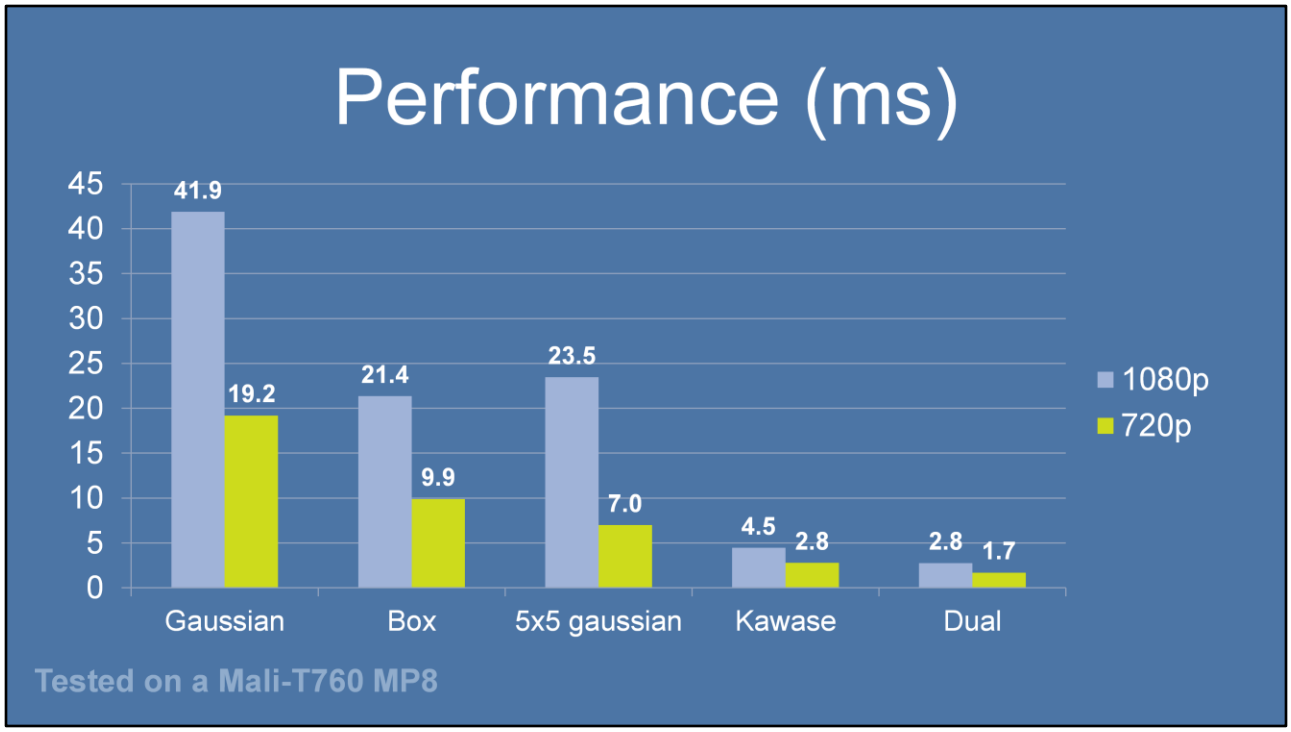
As you can see there are slight differences in the output of the different filters, but the stability is very good – except for the naïve implementation which really suffers here.



Finally we have a more complete setup. In this test we have an input image with a bright square rotating behind some grey bars. We extract the brightest parts of the image, blur it and finally composite it.

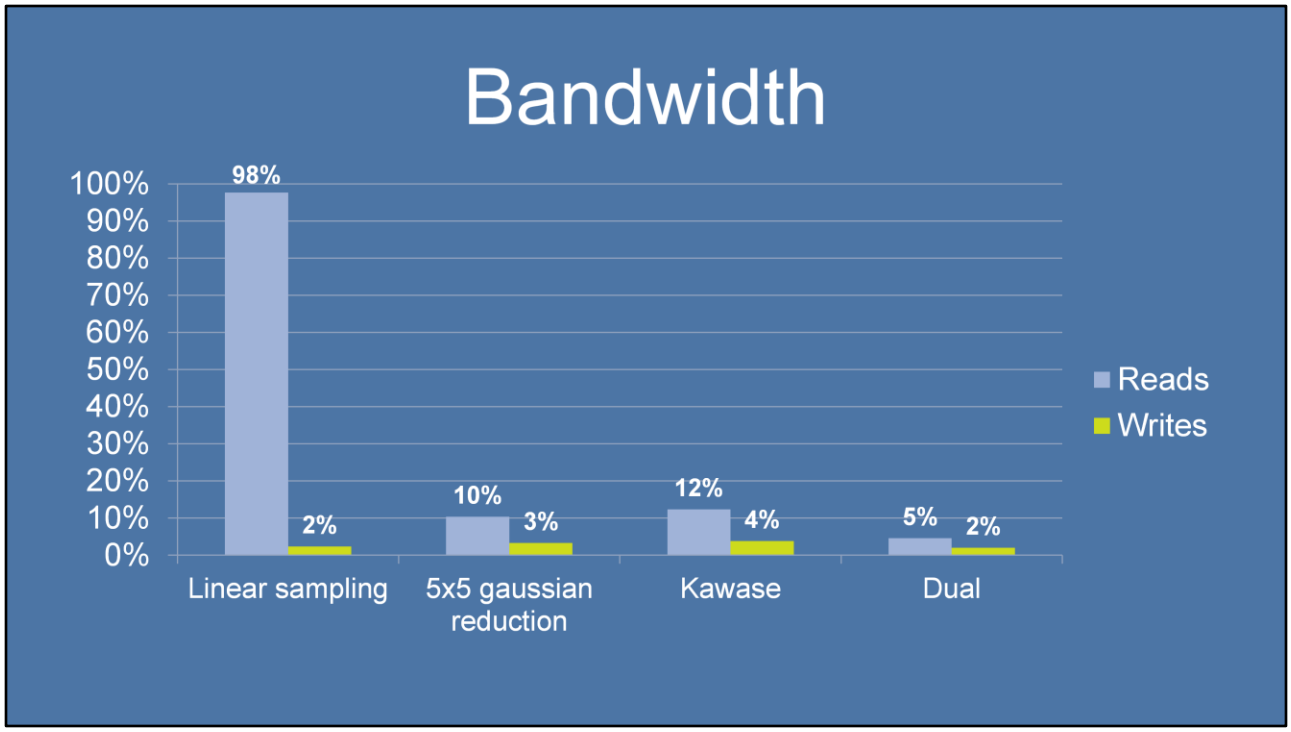
As you can see they all produce excellent result except for the naïve implementation which again suffer from bad aliasing.

Performance comparison



These are frametimes of the different filter implementations. This was tested on a Mali-T760 MP8 device. As you can see the dual filter is the fastest, closely followed by the kawase filter.

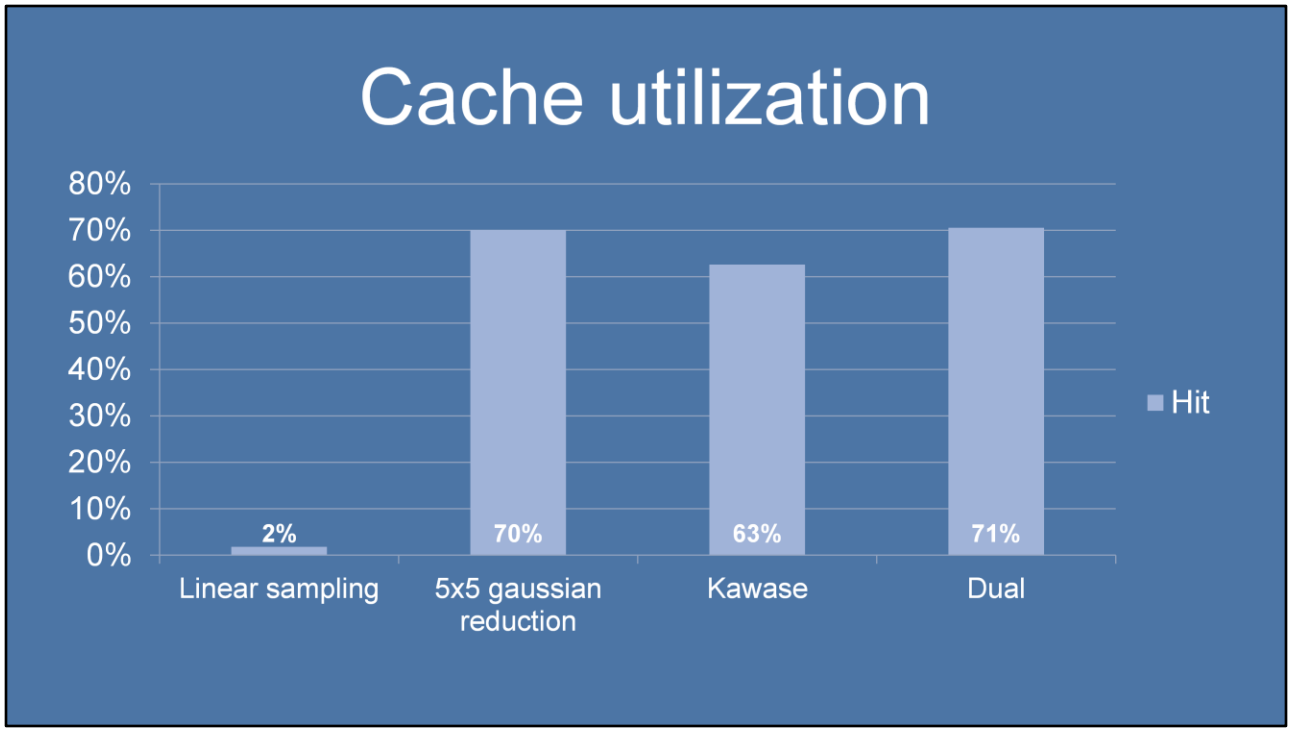
The 5x5 gaussian implemented here is implemented with 4 downsample and upsample passes – since it's a separable blur, that gives a total of 16 render target switches, which directly affects performance.



Here we have bandwidth comparison numbers between the different filters.

As you can see, most of the bandwidth for doing linear sampling goes into actual sampling. 5x5 gaussian is much better with only 10% read bandwidth.

The dual filter only needs 7% of the totalt bandwidth compared to the linear sampling filter, and less than half compared to the kawase filter. The dual filter is also has very balanced read/write bandwidth usage compared to the other approaches – which is good.



And finally we have a comparison of the cache utilization of the different implementations. And this is all inline with what we can expect. Smaller kernels benefit from cache locality, while larger kernels will quickly start thrashing the cache.

On mobile due to constraints on die area, the texture cache is quite small – which makes it even more important to use smaller kernels.

Summary

- On-chip rendering
  - Please use the extensions
- Bloom
  - Multi-pass mixed resolution
  - “Dual filter” blur
- Next steps
  - Work on getting on-chip rendering into future core APIs
  - Look into alternative data flows for doing blurs

So, just to sum up the talk. As mentioned, mobile GPUs are already bandwidth-efficient, but on-chip rendering with the use of extensions helps reduce bandwidth usage further – so please use these. The next step here is to work on getting this kind of on-chip rendering into future core APIs.

“Dual filter” blur shows that it’s possible to implement a high quality bloom filter on mobile without killing performance. It may sound unintuitive, but using multiple-passes and taking advantage of cache locality is a huge performance win. The next step here is to further look into alternative data flows.

Thanks!

- Questions?
  - Marius.Bjorge@arm.com

- References

1. Efficient Rendering with Tile Local Storage [Siggraph 2014]
2. http://rastergrid.com/blog/2010/09/efficient-gaussian-blur-with-linear-sampling/
3. Frame Buffer Postprocessing Effects in DOUBLE-S.T.E.A.L [GDC 2003]