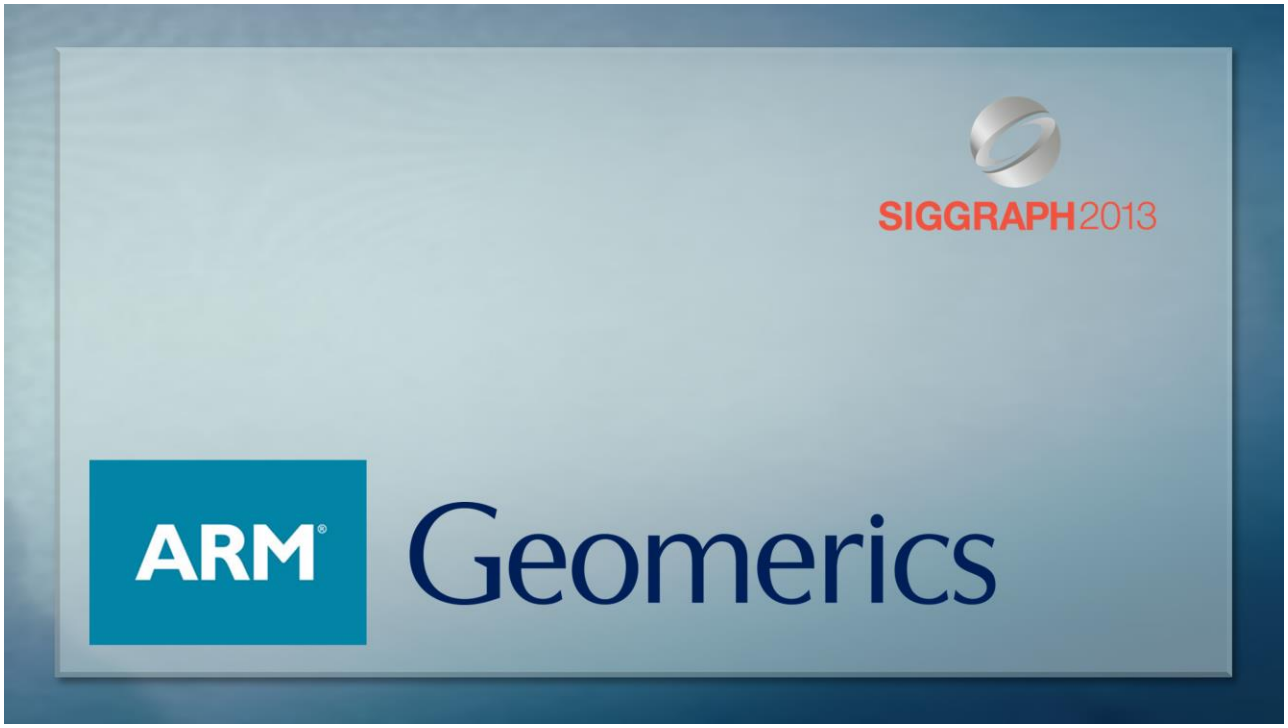
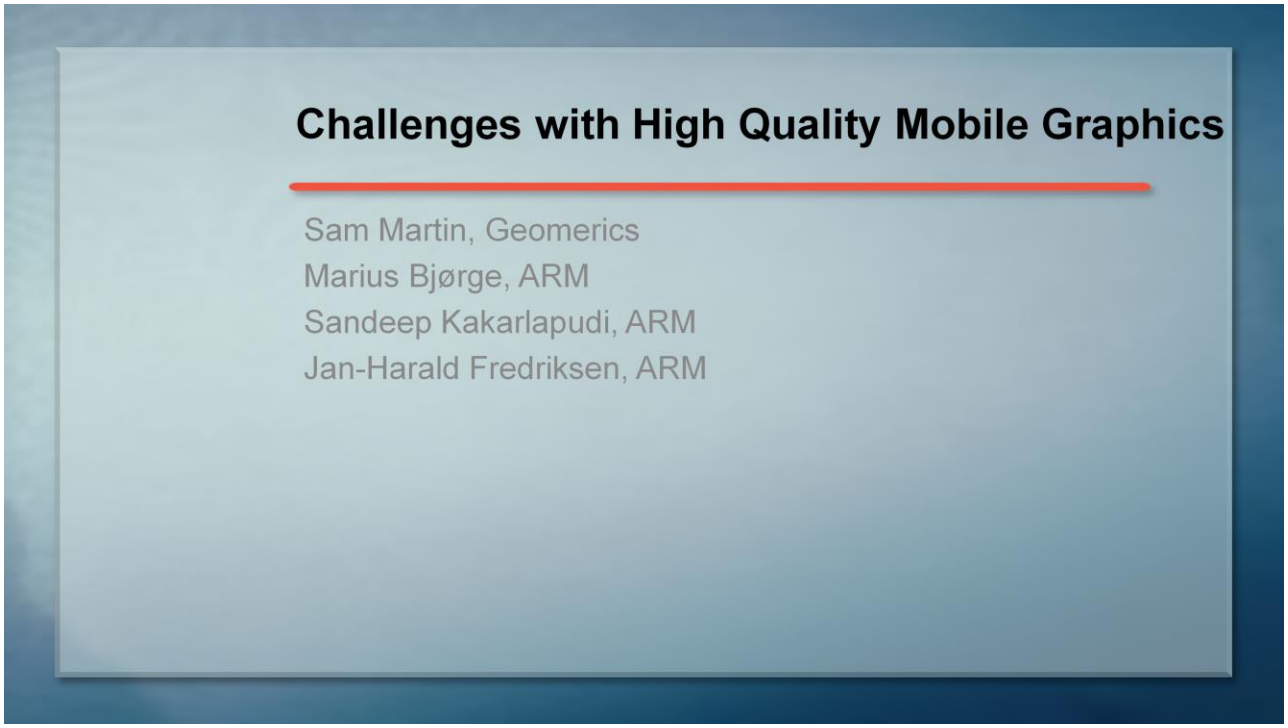
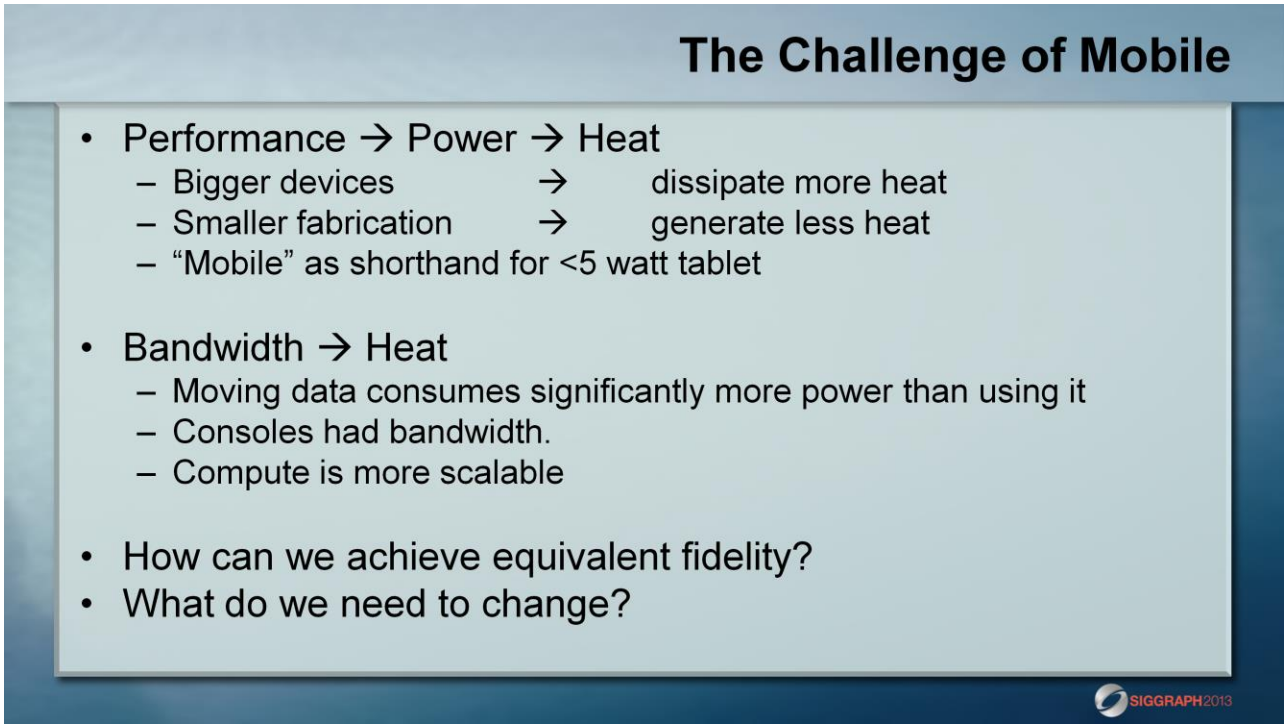

SIGGRAPH2013
ARM
Geomerics



Hi.

I'm Sam Martin, Head of Technology at Geomerics, and this is joint work with Marius Bjorge, Sandeep Kakalarpudi and Jan-Harald Fredriksen from ARM's offices in Trondheim.

This talk is about **one aspect** of our ongoing R&D on **how to achieve high quality graphics** – the kind we see on PC/consoles – on mobile devices.



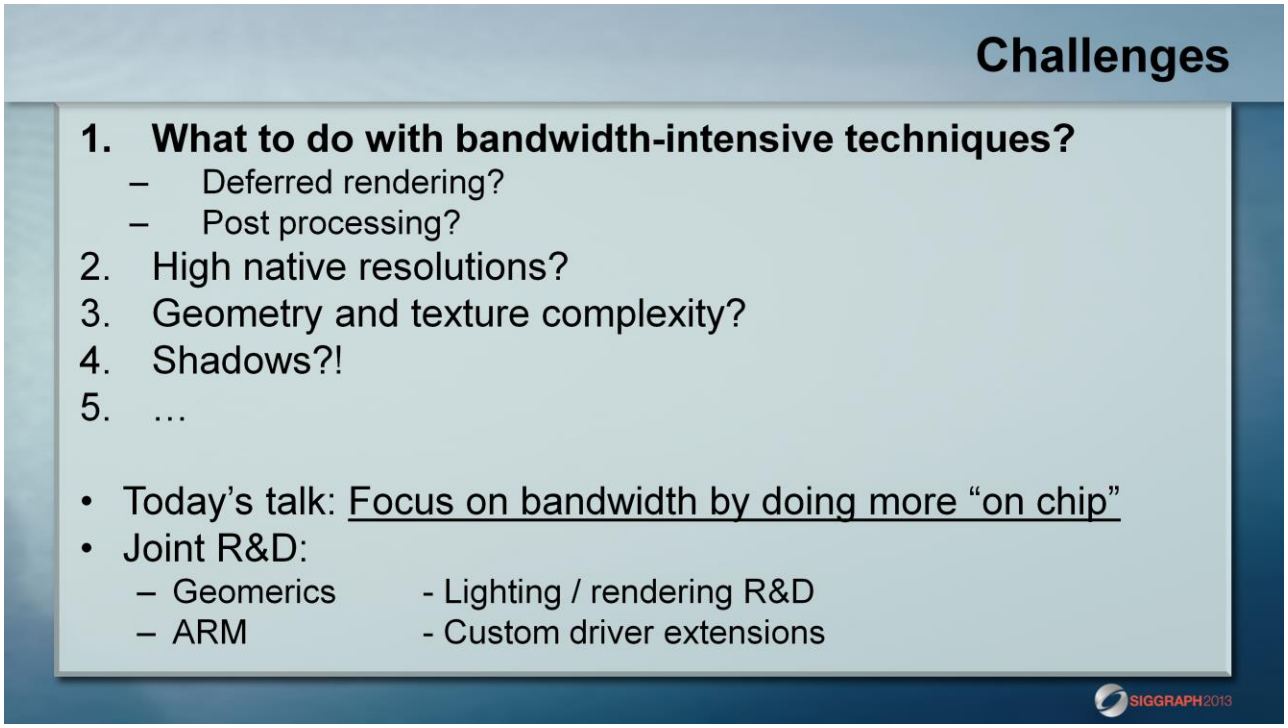
Mobile has it's own special challenges.

Perhaps surprisingly, mobile is not really bottlenecked on battery life anymore. It's all about heat, and at least as far as heat is concerned, size matters. Bigger things can run faster simply because they can dissipate more heat. I don't see phones getting that bigger any time soon, so need to look at ways of reducing the sources of heat.

Different aspects of computing require different amounts of power. An important observation is that moving data about consumes dramatically more power than computing with it, so **memory bandwidth** is a major source of heat. Since the amount of heat you produce and can dissipate is constrained by the laws of physics, mobiles simply have much less bandwidth than consoles, even old ones.

While mobile devices will soon be in the low 100s Gflops – roughly speaking the same ballpark as current consoles - this is a very limited part of the picture. Bandwidth figures remain way behind and will **stay that way** for the predictable future. (Post talk update: See bonus slides for more details.)

So we have to wean ourselves off bandwidth and lean more on compute. Consoles had bandwidth and used it. If we want to get the same results on mobile we'll need to **find alternatives**.



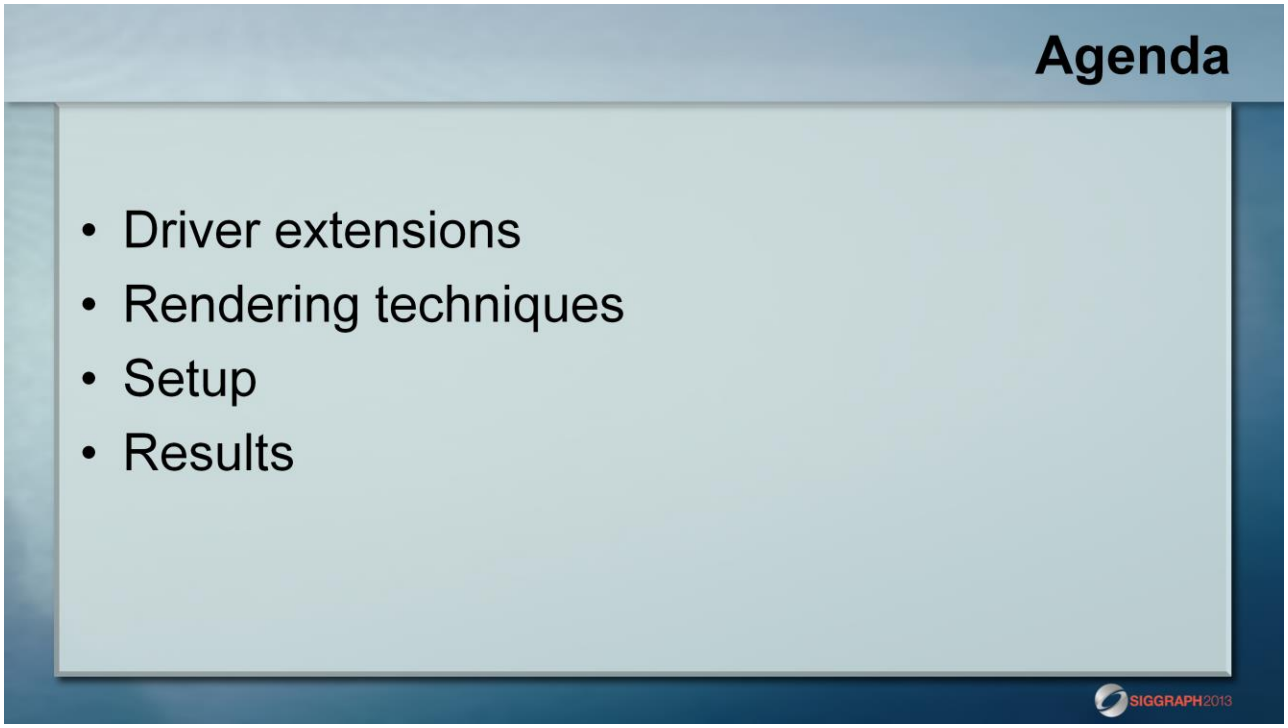
So this talk is about ways to do more with less bandwidth.

There are a lot of rendering challenges we could look at, but I'd like to **focus on deferred rendering** – a very popular means of rendering with many dynamic lights. IMO, deferred rendering and & post processing pretty much defined the look of current gen consoles. Both are bandwidth intensive, so are worth revisiting for mobile. You couldn't write a deferred renderer in OpenGL ES 2.0 (no MRT support), but OpenGL ES 3.0 (GLES3) recently fixed this.

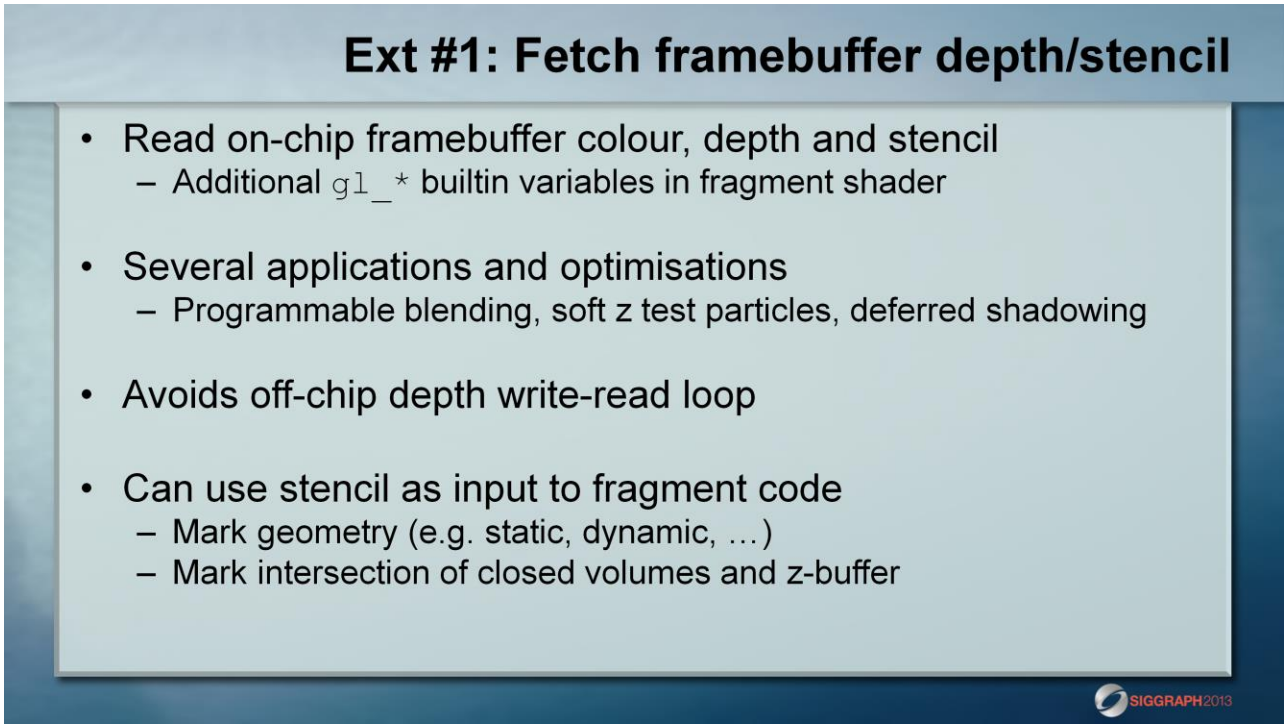
This was also intended to be forward looking R&D. By moving more work onto compute and away from bandwidth we can potentially upset the tuning of current hardware. It turns out that current devices are quite capable of efficient on-chip deferred rendering, but their performance was not the main focus.

The bandwidth issue with deferred lighting has been known for some time. We (Geomerics) expected a solution to come from a collaboration of hardware and software advances and so teamed up with ARM. We look at ways to do more work "on chip". By doing things on chip we would minimise bandwidth, moving the focus to compute, and produce a viable means of getting console fidelity lighting without melting any phones.

The on chip research is what we will discuss and present today.



Here's the agenda.



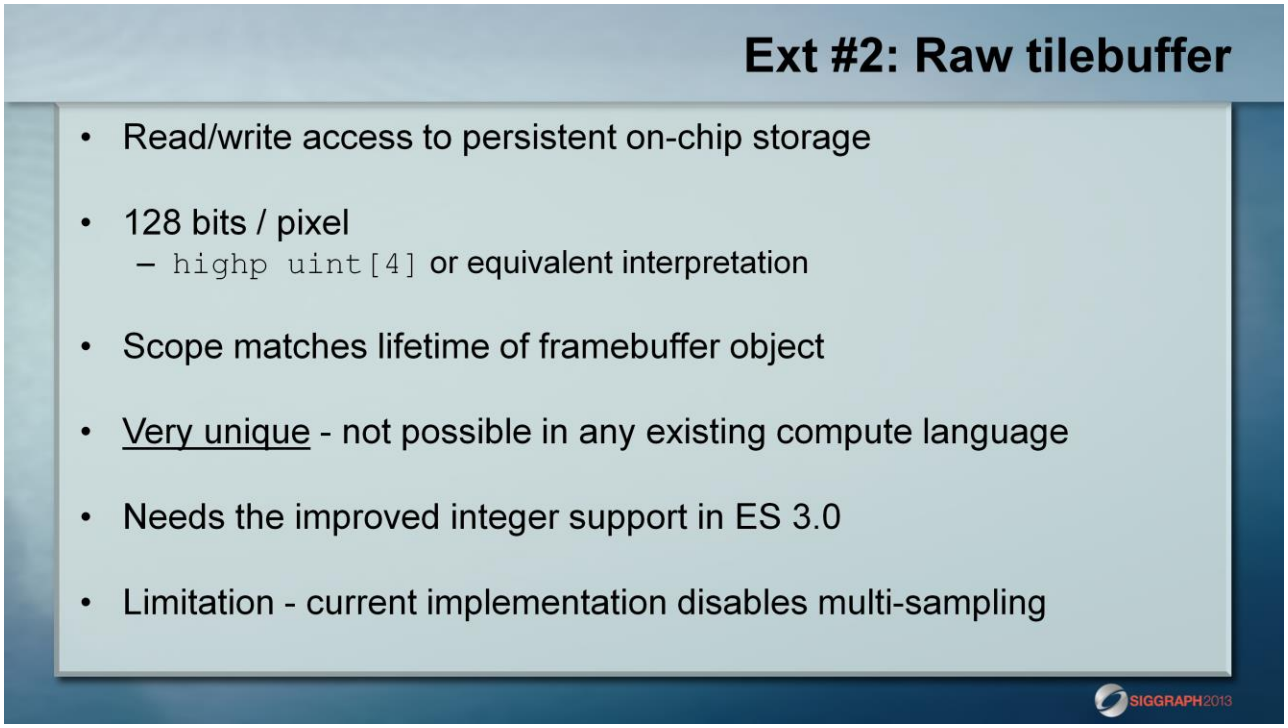
To enable on chip work, ARM added two experimental extensions. Here's the first.

It's a straightforward extension that provides read access to the current frame buffer, notably the depth and stencil value. There are existing extensions, e.g. EXT_shader_framebuffer_fetch, which allow read access to the framebuffer colour but not depth or stencil. So it's the depth/stencil aspect that is new.

On tile-based renderers the framebuffer resides on chip anyway, so providing read access to it is free from a bandwidth point of view.

Probably the most well known application of this extension is proper programmable blending – a long standing feature request from the games and graphics industry. This request has been blocked on desktop hardware for years because they have fixed function blending units.

It's also worth noting that if you have depth and the camera matrix, you can reconstruct 3d positions for every pixel in the framebuffer. So read access to depth provides a wealth of information. For example, you can use this extension to modulate shadows into the framebuffer, and many other depth-based computations – perhaps deferred fog or volumetric effects.



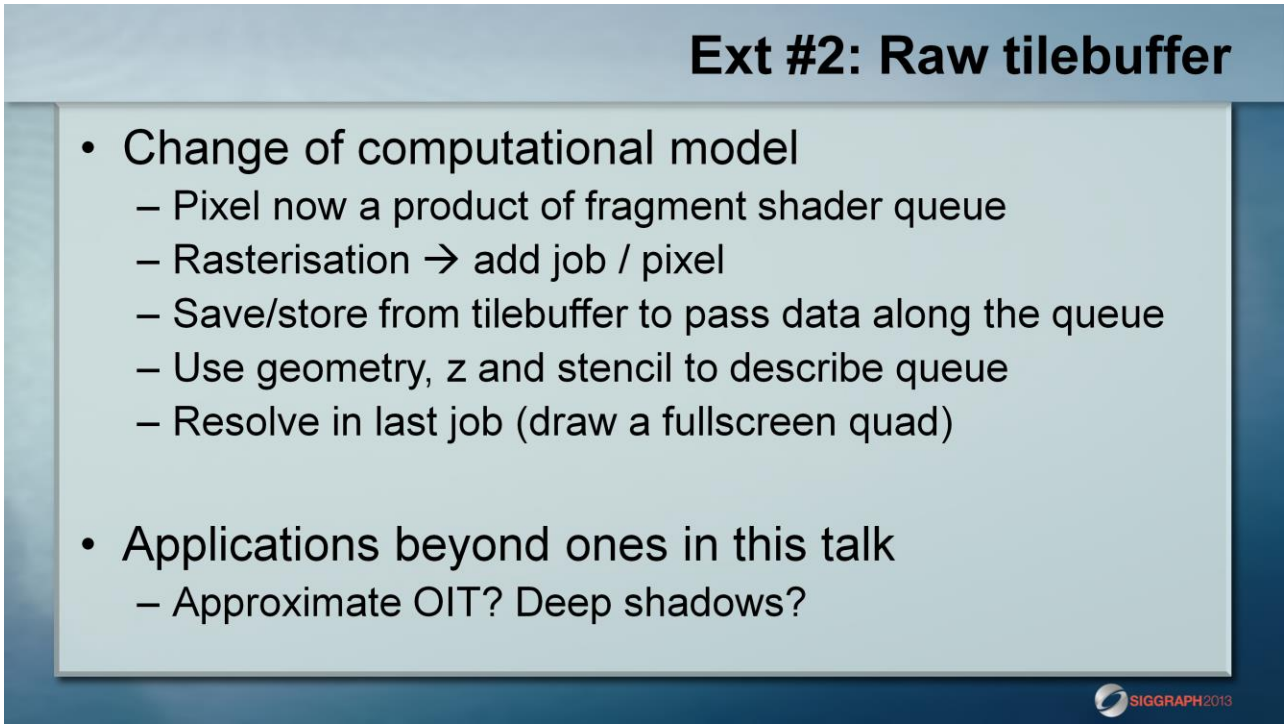
The second extension is the one I'm most excited about.

It turns out that the Mali T604 has a small amount of per-pixel storage on chip – **128bits/pixel** – which is normally used for multi-sampling. In the version of the extension we used it's exposed as 4 highp uints you can just read and write to, but this is not final and likely to change.

The really interesting thing about this storage is that it's **persistent throughout the lifetime of the framebuffer** – so across draw calls and shader changes.
You just read and write from your fragment shader, so it's also very simple and minimally invasive.

I really want to emphasis how uniquely expressive this extension is. No other production compute language I know of has any kind of persistent on chip storage. They have the concept of on chip memory, but it's always cleared down at the end of each kernel and any data for subsequent kernels must be passed back through main memory. On mobile, where you really want to do everything in a single pass, this extension is an extremely powerful tool for reducing bandwidth.

Unfortunately, the current implementation disables multi-sampling, which is not an uncommon problem for deferred rendering.

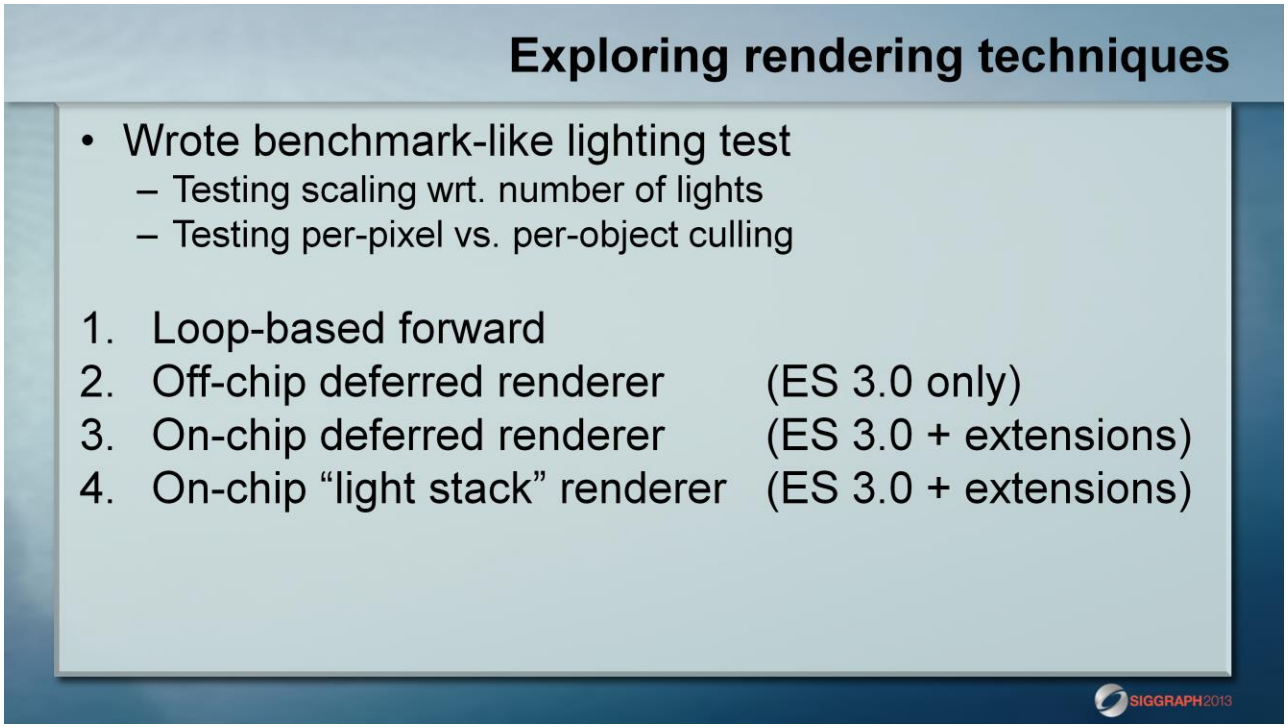


In case it wasn't yet clear exactly why I think this is such an exceptional advance, let me illustrate how it allows you to think differently about how computations are performed:

You no longer generate a pixel per fragment shader invocation.
Instead, each pixel is the end result of a queue of fragment shaders, which build up the result progressively, passing data between them.
Rasterisation is then the means for describing the queue of fragment work. E.g.:
- A full screen quad queues a job everywhere.
- Geometry, together with the usual z and stencil logic can be used to describe regions of affect.

As a worked example, we might start by rendering a full screen quad to initialise the on chip storage. We then render geometry to describe the work we want to do, then finish by drawing another full screen quad to 'resolve' the storage to the draw attachments. We would only write to the draw attachments in the last shader. Otherwise you are free to party on the bits as much as you want.

It'd just like to note that this extension, and the change of model it brings, is significant for many aspects of rendering, not just our focus on deferred rendering.



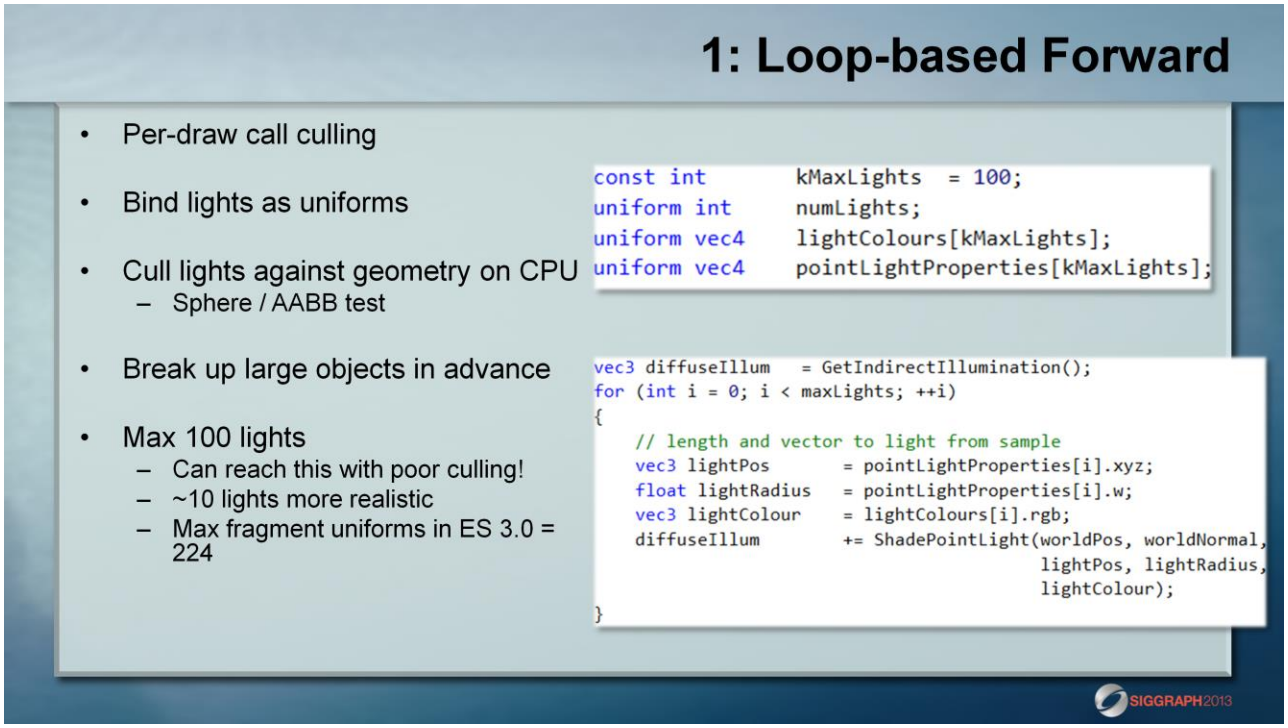
Armed with these extensions we revisited popular rendering techniques.

We implemented a regular forward renderer, that loops over lights in uniforms, and a ‘traditional’ deferred renderer using the MRT support in gles3. These provide the baselines for our comparisons.

Then using the extensions we implemented an on-chip variant of deferred rendering, and an on-chip variant of something similar to “forward+”, which are calling a lightstack renderer.

Deferred rendering was the defacto solution for large numbers of lights because it solved the culling problem. Through deferred rendering you could cull tightly per-pixel or per-tile. Forward renderers were culling per object.

The last 3 techniques can tightly cull lights at the pixel level, where as the loop-based renderer can only cull crudely and per-draw call.

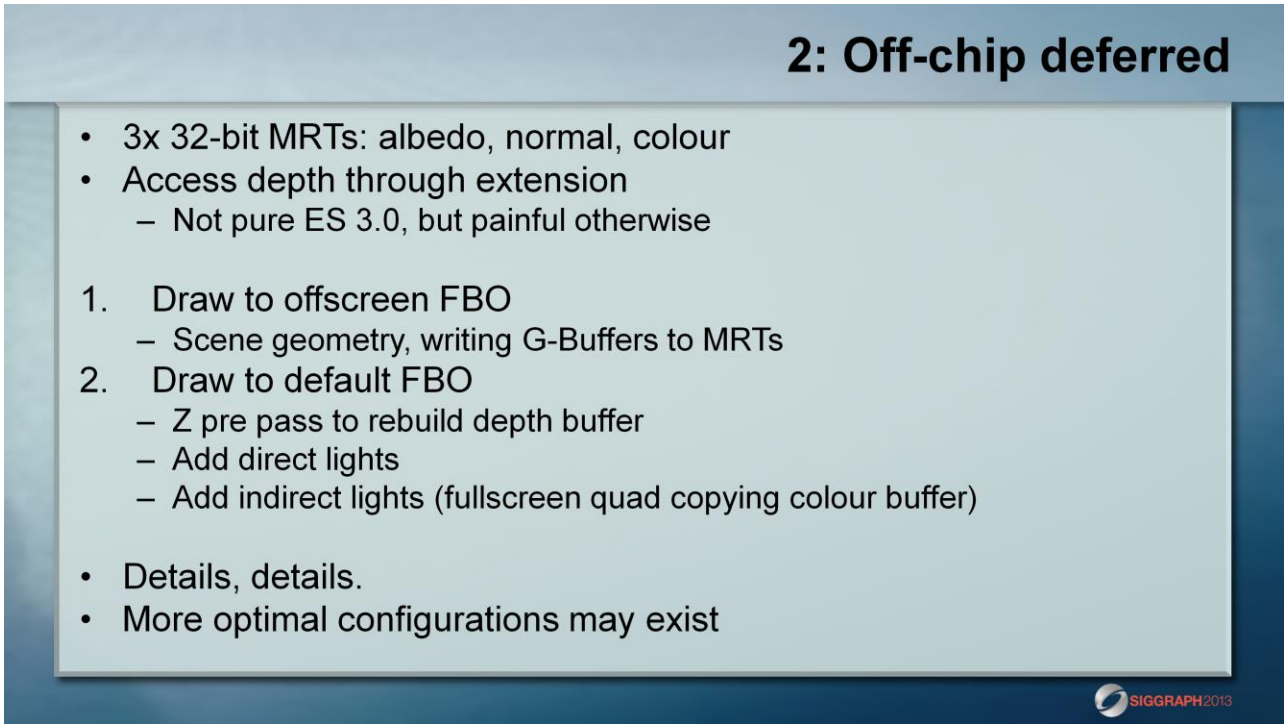


So, to explain the 4 renderers in a bit more detail.

The first is the usual forward renderer. In this we simply declare a large number of uniforms and then per draw call we intersect the bounding box of the geometry with each of the lights. We add all the lights that intersect to the uniform array and set the ‘numLights’ uniform appropriately. The fragment shader then just loops over them.

Key thing about this approach is that performance depends heavily on how successful the culling was. When this fails the technique fails.

Doing everything in a single shader can be useful, particularly as it allows arbitrary coupling of lights to the material shading model. The downside is that having a large shader tends to increase register pressure, which affects occupancy, which costs performance. This is a real issue on consoles and desktop GPUs, and also exists on mobile where you tend to have even fewer registers. We did not look in detail at the trade off between large shaders verses a sequence of smaller shaders (assuming data transfer through on chip memory) as it would depend on too many details of the current hardware. But I wanted to note it as a property to be aware of when investigating performance.

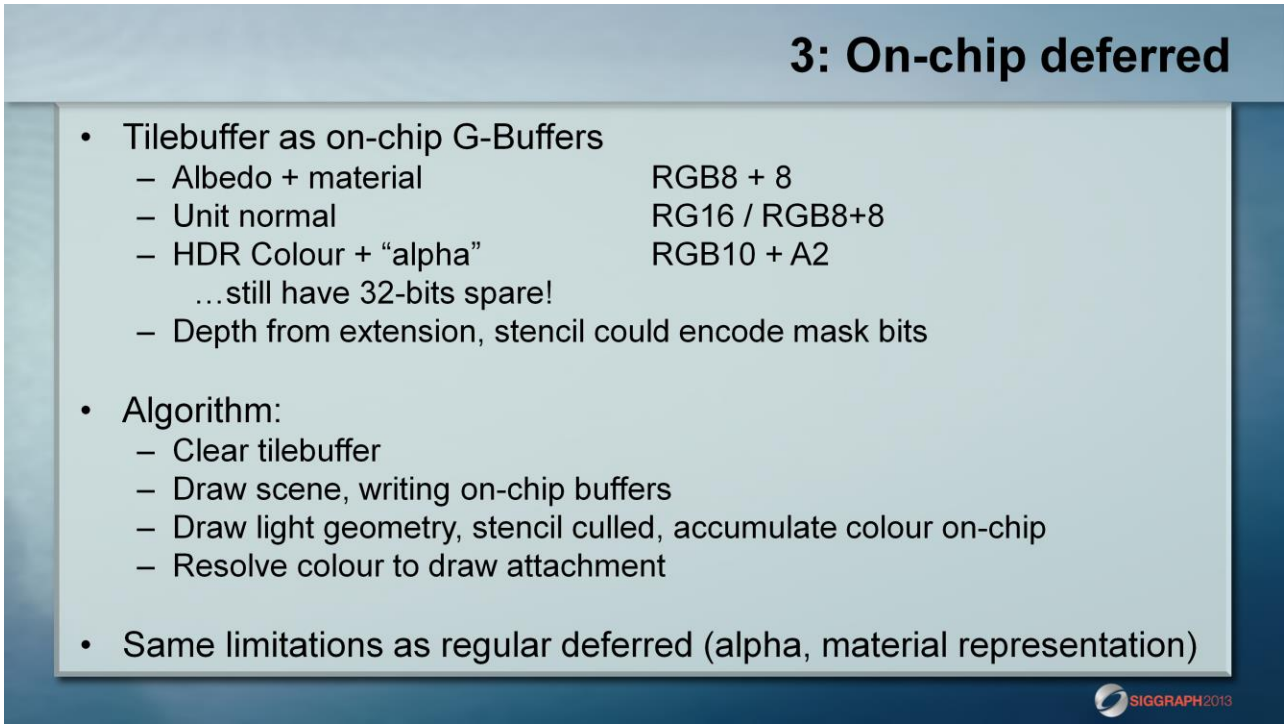


Our next renderer is an almost-standard deferred renderer. I say “almost” because we use 3 MRTs rather than the usual 4 and cheat a bit by grabbing depth from the framebuffer read extension. Without this extension we would have to use the 4th attachment point to store depth, as then read it back in through a sampler, which will certainly have a non-trivial bandwidth hit. However, we do need to prime the z-buffer after the FBO change so we have to render scene positions a second time. This may sound bad, but it appears to be the best of a bad set of options.

There are some api issues that surround the use of depth in this setting. In particular:

- You cannot really use the default (egl-created) depth and colour buffer effectively as you can’t bind them your own offscreen FBOs. This tends to force an additional framebuffer copy on you to get the results back into the default framebuffer, or an extra geometry pass (the option we chose).
- The api will allow you bind a depth buffer to read (via a texture) and potentially write (via the depth attachment) but the behaviour of this potential feedback loop is undefined. To complicate matters, we write to the stencil. It was unclear to me whether this behaviour is well defined in the spec. You could create another copy of the depth texture to avoid this, but then you are burning yet more bandwidth.

There are too many details to cover today – just contact me for more info if you are interested. Boiled down, gles3 is sufficient for deferred rendering, but it is not efficient or pretty.



Feature-wise our on-chip implementation of deferred lighting is very similar to the previous off-chip one.

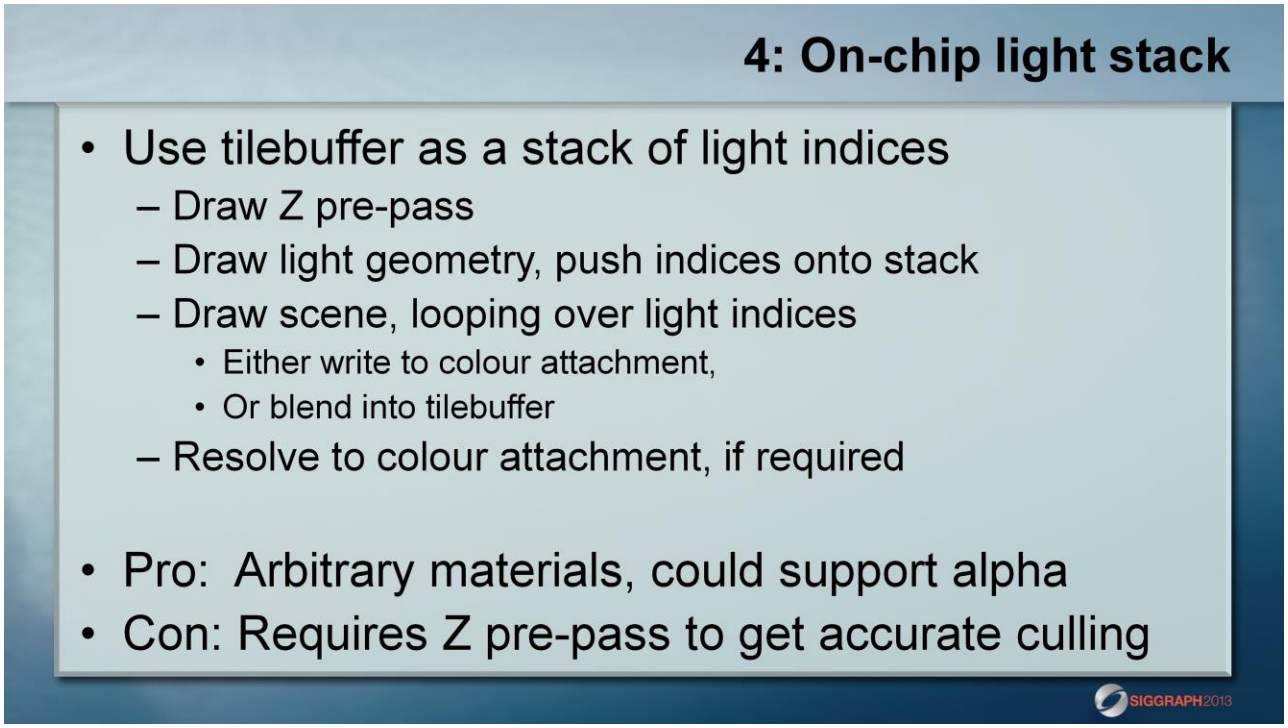
Instead of writing out the g-buffer data we instead pack it into the on-chip tilebuffer. There’s also no need to store depth as we can use the depth/stencil fetch. Because we never change from the default FB, depth data is always available so we avoid all the depth-related headaches of the off-chip version. The separate storage of depth actually gives us more available data than we had with the off chip approach. I couldn’t think of a use for the entire thing in our benchmark environment, but if you have a use for it you may as well! The 5 bytes we didn’t use provide enough room for a specular colour (e.g. metals), roughness and spec scale, for example.

The main benefits are:
- Single pass: no FBO changes, fussing about with depth, etc.
- Larger and more flexible g-buffers without the bandwidth cost.

Not to mention it’s actually really simple.

A likely corollary of removing the bandwidth cost is that tile-based variations that helped reduce bandwidth on PC/consoles are less likely to be a win on-chip. When on-chip you may as well cull more tightly using the GPU. No more CPU rasterisation.

The downside of deferred rendering, of any kind, is that it doesn’t support alpha…

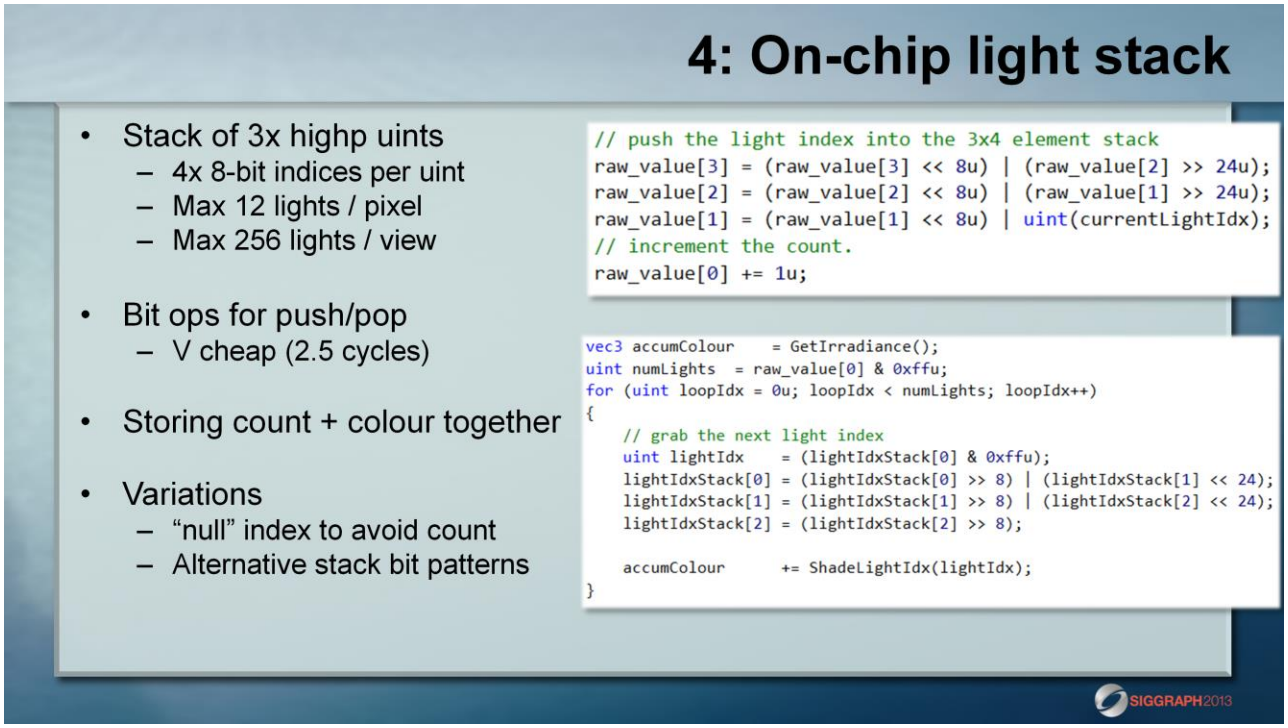


On PCs/consoles, “forward+” is probably the most well known technique that supports both per-pixel culling and arbitrary materials, including transparency. It involved doing a z pre-pass of sorts, then drawing the light geometry which added the light to a per-pixel linked list of lights. You then drew the scene properly, looping over the lights in the linked list much the same way we did in our loop-based forward renderer. Theoretically, forward+ used less bandwidth than regular deferred but if we are on-chip the reverse is actually true as you now need a z pre-pass.

However, forward+ also had the benefit that you didn’t have to distill your materials into g-buffers. G-buffers limit the interaction between the rest of the material shader (which could be artist authored) and the lighting. G-buffers also only store opaque geometry, but with care forward+ can also support alpha.

It’s a very nice idea, but I must admit, whenever I see a linked list, I also see an opportunity for optimisation ☺

So we tried out a forward+-like algorithm we are calling a lightstack renderer. We also build a per-pixel set of lights before rendering the scene, but do so by interpreting an area of the tilebuffer as a stack of ints.

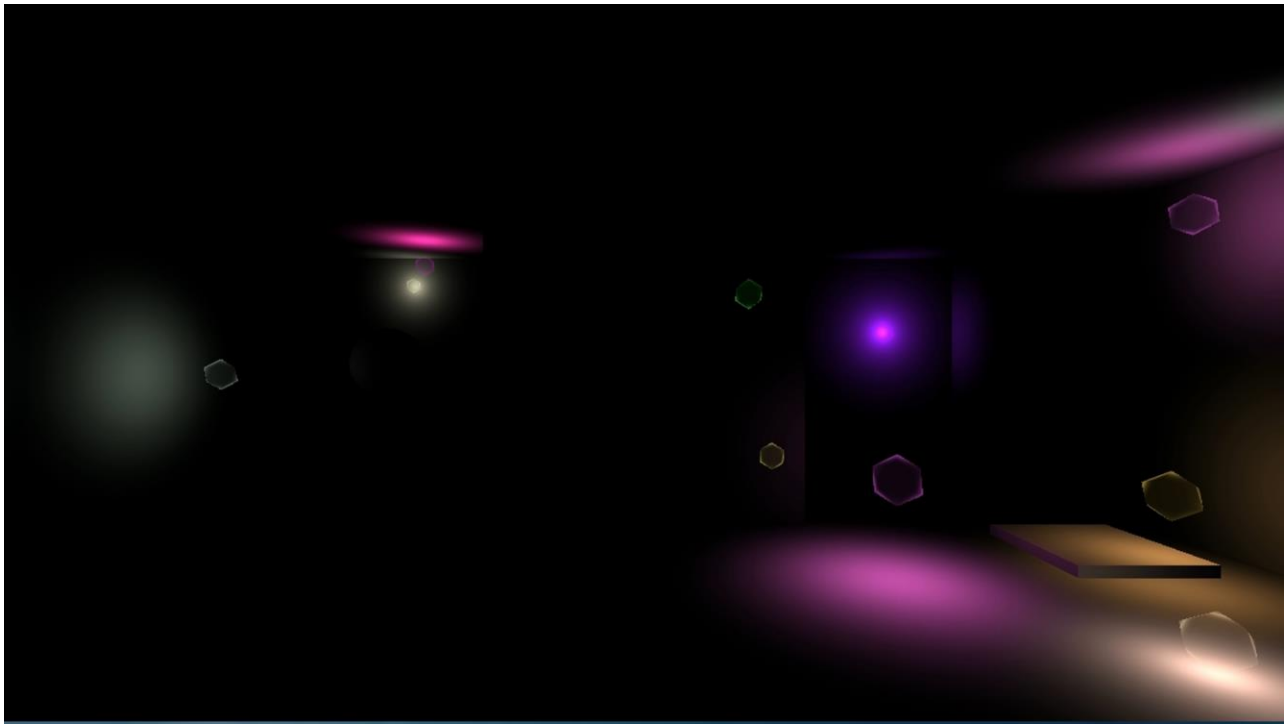


There are many ways you can shake this out but the variation we plumped for uses 3/4s of the tilebuffer as a stack of byte indices.

To push/pop indices we just shuffle the bits about - see code snippet. This turns out to be pretty cheap on Mali as the resulting fragment shader to push a light is just 2.5 cycles (cheaper than a z pre pass vertex for instance).

When shading we just loop, popping the indices as we go, grabbing lights from a big list of uniforms.

We also opted to keep a count of the lights in the low bits of the remaining uint. You can instead use a while loop with a special null index. This just turned out to be very slightly longer when compiled, but I suspect the difference is unlikely to be significant in practice. The count actually overlaps with the (unused) alpha channel of the accumulated colour in the tilebuffer, so if we were to add alpha support we would have to use the null index approach instead.



The easiest way to show you what we are doing is to show a quick video.

We have a simple test setup that uses the standard programmer lighting solution of lots of point lights flying about in space.

We also use Enlighten to provide the bounce lighting. It just adds a small portion of CPU work and an extract texture sample to the shader. Everyone should be doing real-time radiosity these days anyway ☺
<video shows with/without>

Now, to get per-pixel culling we use the stencil buffer to mask in the pixels where the light geometry intersects the z buffer. It's a fairly standard optimisation, and can all be done efficiently on chip.
<video illustrates stencil clipping operation>

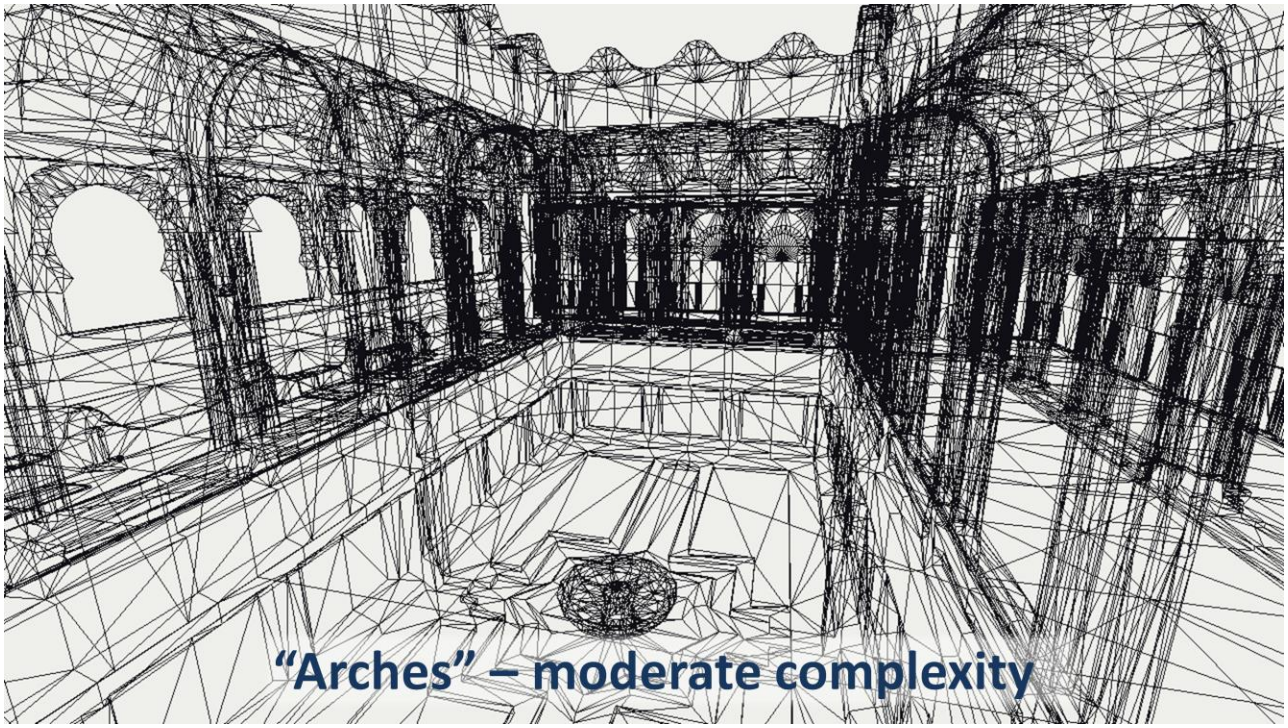
If we disable the stencil test, and just shade the affected area on screen you get some idea of how much of a saving the stencil optimisation gets us.
<video illustrates shading without stencil clipping>

We use a couple of test scenes, chosen primarily for their range of geometric complexity.

The scene in the video was the ‘ubox’. It’s really simple, so in these scenes the geometric complexity is essentially insignficiant.

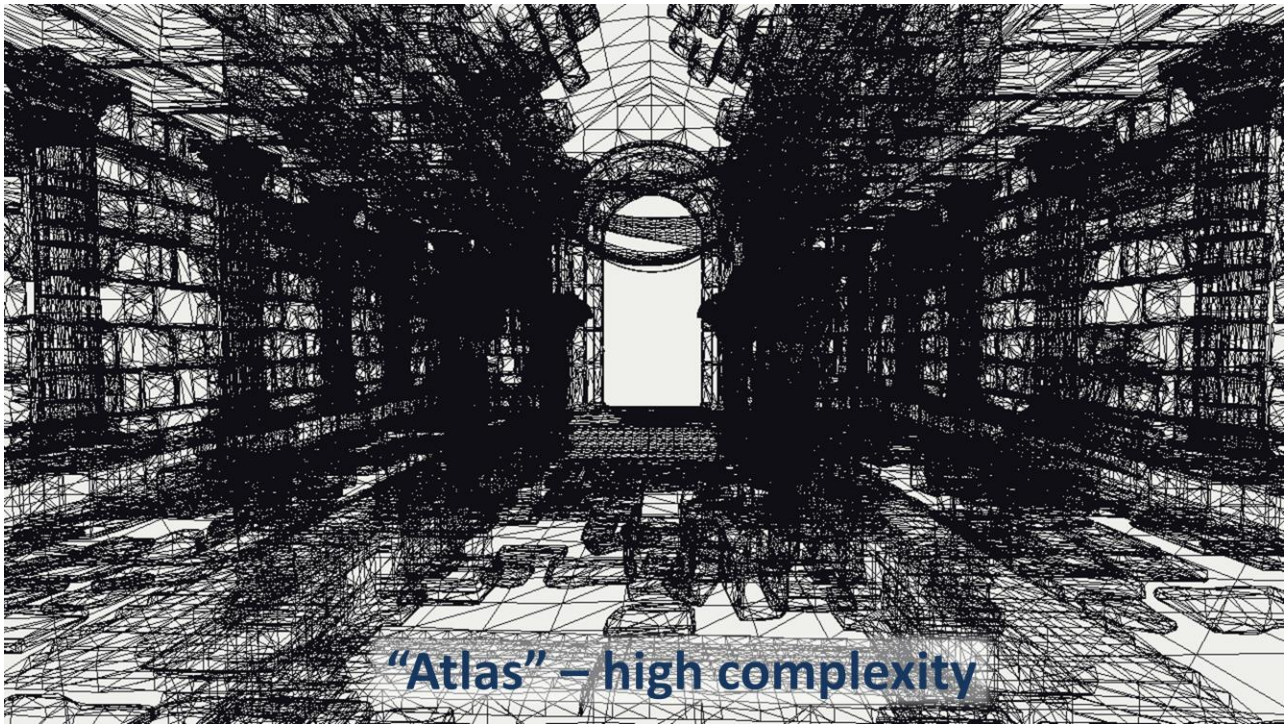
Our ‘moderate complexity’ scene is the Arches. It’s about 50k verts, which is still on the low side, but is hard to sort and cull against.

This is the geometry.



And the same scene but in wireframe so you can see the tri density.

Our high complexity scene is “atlas”. It’s a section from our (Geomerics) next-gen lighting demo. It’s about 500k verts and tris. The scene was sculpted in z-brush and then baked out with normal maps. It runs at 30fps on desktop, so it’s not really been designed with mobile in mind ☺



In particular it has a lot of very small tris. Areas of it have a very high tri density. Particularly the statues, which are around1 vert/pixel from typical viewpoints. We don’t use LODs in our benchmark, so this is a fairly pathological case for a tile-based renderer.

[Post talk update – see the bonus slides for results on other assets]

R&D test bed

- Test setup
  - "Arndale" board
  - Linux-on-ARM OS
  - ARM Mali T604 GPU
  - Similar spec to Google Nexus 10
- Custom OpenGL ES 3.0 driver
- Custom renderer
- Test assets from PC/consoles
- Enlighten real time radiosity ☺

SIGGRAPH2013

Sadly, you can't just go do all of this on your favourite tablet! Instead we run on a linux-based system, full of personality. The hardware is similar in spec to a Nexus 10, but otherwise custom everything.

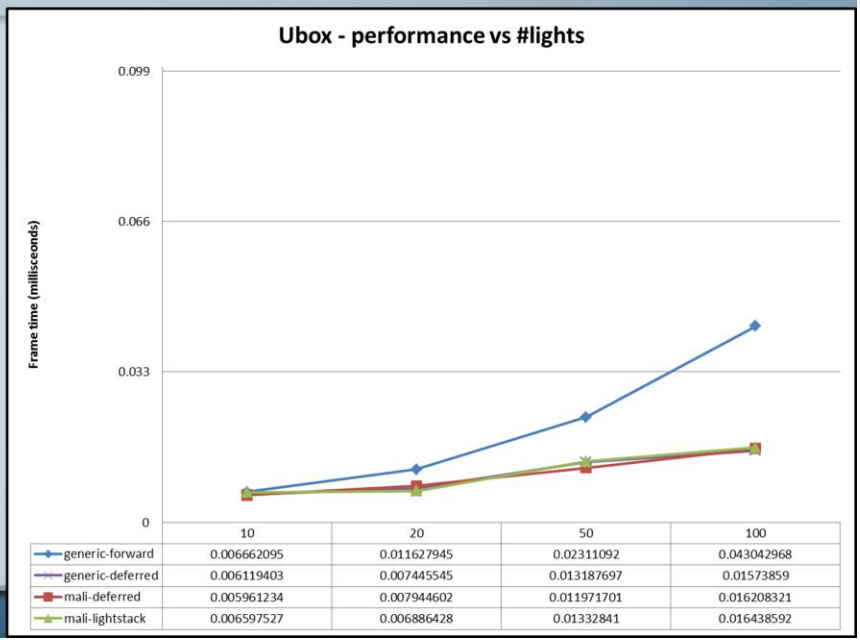
Frankly, we'd rather not have a completely custom stack as custom everything tends to come with some really impressive custom bugs ☺ The main reason was to get the custom gles3 driver with the experimental extensions, for which you have to go off-piste.

There are upsides, such as a working debugger, ssh access, apt-get, bash scripts and terminal prompt, etc.

The downsides did bite us though and we couldn't get reliable bandwidth figures until after this talk was presented – these are now in the bonus slides. We (and when I say "we", I really mean Sandeep ☺) spent a long time going over a complete breakdown of the memory bandwidth to ensure we were confident in them.

Results – Trivial geometry

- Simple geometry
- Limited overdraw
- Per-pixel culling dominant effect
- Forward slightly faster for <= 10 lights



	10	20	50	100
generic-forward	0.006662095	0.011627945	0.02311092	0.043042968
generic-deferred	0.006119403	0.007445545	0.013187697	0.01573859
mali-deferred	0.005961234	0.007944602	0.011971701	0.016208321
mali-lightstack	0.006597527	0.006886428	0.01332841	0.016438592

Thankfully, we can reliably measure performance.

In each graph we will plot the performance, measured in milliseconds of total frame time, of the 4 techniques against 10,20,50 and 100 lights respectively. This helps show how the techniques scale against lights.

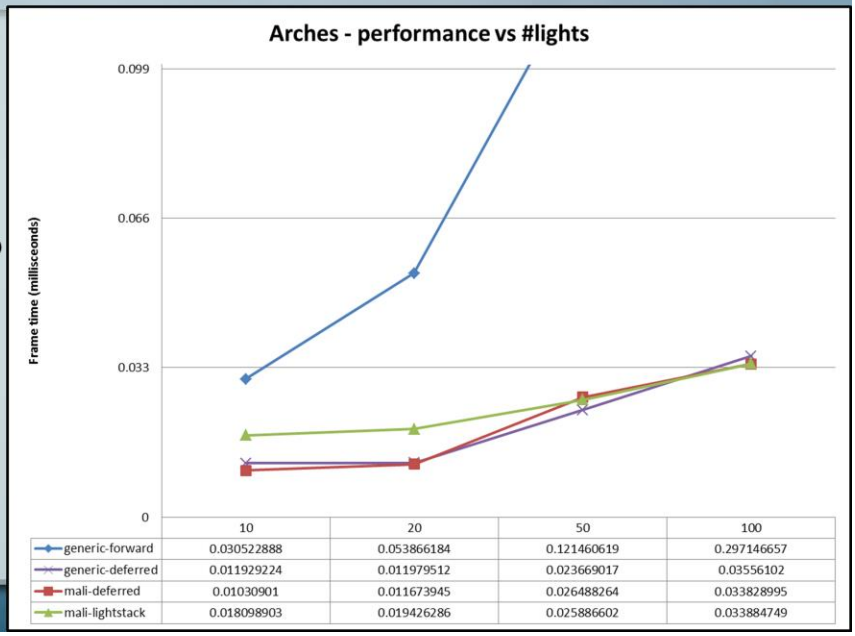
I'll show the ubox results first, then go through the other scenes.

Note that the first horizontal line equates to 30fps. We have a simple benchmark, so to be viable in production on this hardware, where there will be additional overheads, we'd have to see results well below this line.

The blue line in this graph is the forward renderer. The other 3 techniques are effectively on top of each other. So we are basically just showing the improvements that result from better per-pixel culling. At least on this simple geometry, there's no substantial difference in cost between the 3 per-pixel techniques.

Results – Moderate geometry

- More realistic geometry
- Forward suffers from poor culling
  - breaking up geometry would help
- Lightstack appears to be more expensive for less lights
  - Z pre-pass?
  - Otherwise unclear



	10	20	50	100
generic-forward	0.030522888	0.053866184	0.121460619	0.297146657
generic-deferred	0.011929224	0.011979512	0.023669017	0.03556102
mali-deferred	0.01030901	0.011673945	0.026488264	0.033828995
mali-lightstack	0.018098903	0.019426286	0.025886602	0.033884749

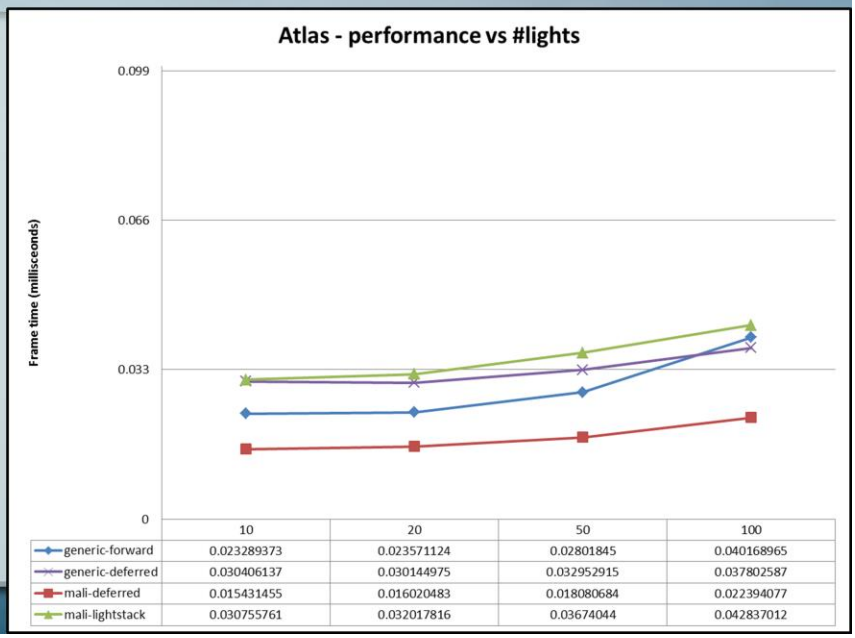
The results on the arches highlight the impact poor culling can have. A reasonable proportion of the geometry in the arches is connected around the central open area. This is great from the point of view of reducing draw calls, but terrible from a culling perspective as most draw calls have a bounding box that's a good proportion of the entire scene. You could break up the geometry further, but it's reasonable example of how badly per-object culling can fail.

We see a slightly different performance profile for the other techniques. I believe the increased cost for larger numbers of lights just comes from the different distribution of geometry. It's just more likely that with the extra lights you will get some lights that cover geometry near the camera - and therefore a lot more onscreen pixels. It's quite a confined space, so there's often some geometry that is in close view in this scene.

We also see an increased cost for the lightstack approach. The reason for this is unclear, but the additional overhead is reliably measurable. We have speculated that the memory access patterns the lightstack algorithm produces when reading the indexed uniforms may be a part of this, but have not investigated in sufficient detail to prove this.

Results – Complex geometry

- Very complex geometry
- Geometry dominates
- Reasonable per-object culling
- Expensive Z pre-pass
  - mali-lightstack & generic-deferred



	10	20	50	100
generic-forward	0.023289373	0.023571124	0.02801845	0.040168965
generic-deferred	0.030406137	0.030144975	0.032952915	0.037802587
mali-deferred	0.015431455	0.016020483	0.018080684	0.022394077
mali-lightstack	0.030755761	0.032017816	0.03674044	0.042837012

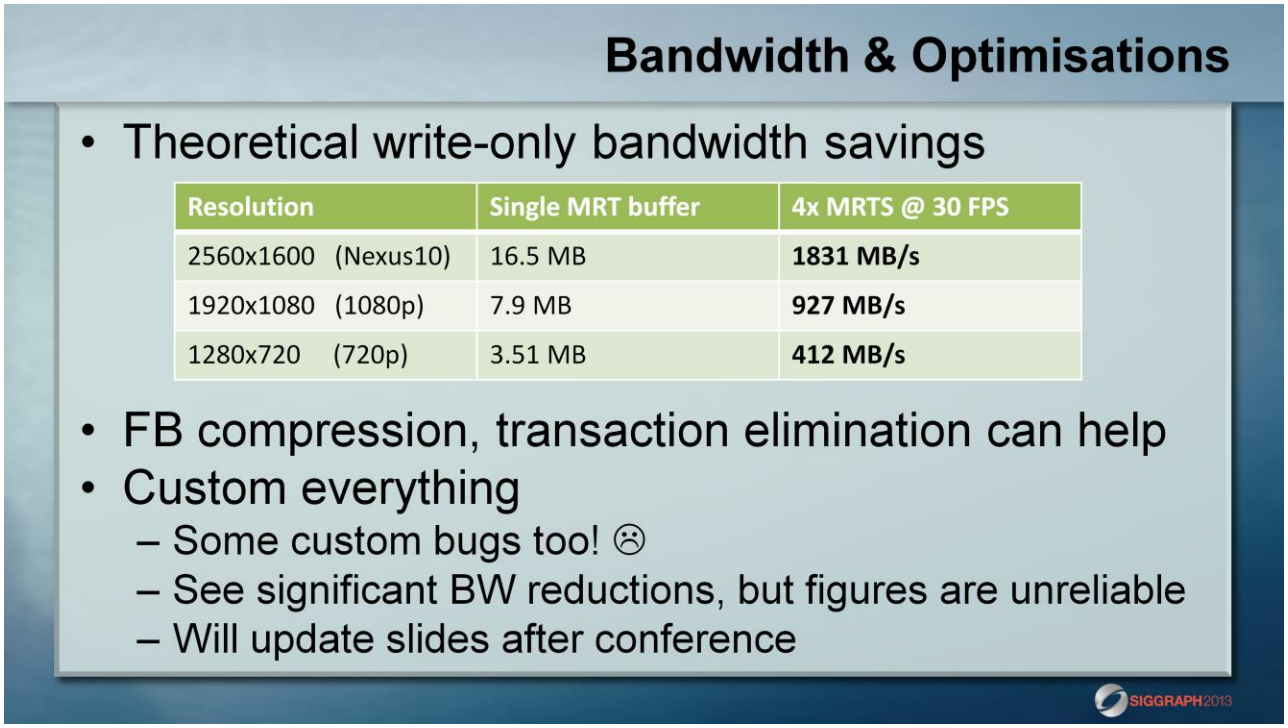
The final graph shows the results on the Atlas scene. Here the geometry cost dominates.

The clear winner here is the on-chip deferred rendering approach.

Both the off-chip deferred approach and lightstack renderer suffer from requiring a second geometry pass.

The forward renderer does better, but shows a less-than-desirable scaling pattern with larger numbers of lights. This scene is actually particularly well suited to per-object culling as it's composed of a lot (several hundred) small compact objects which give good cpu culling results. But we still see a significant overhead above on chip deferred rendering.

It's also worth noting that we can do 100 lights, including real time GI, on ½ million verts with headroom, which is very encouraging!



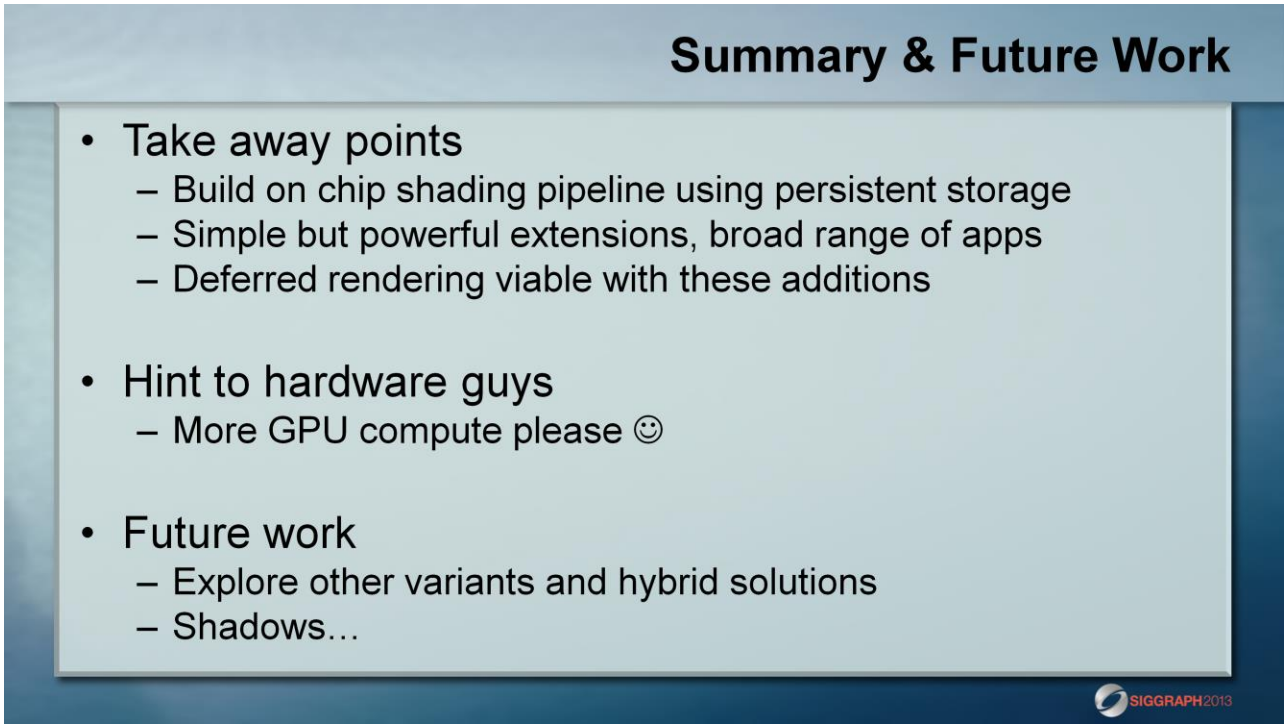
However, it is all about the bandwidth in the long run. It's important to note that **we are not bandwidth bound** in any of our test scenes, so the amount of bandwidth the techniques consume does not influence their performance.

The theoretical savings of on-chip deferred rendering verse off-chip deferred rendering are very significant. The exact amount you will save depends upon how many times you shade each pixel, but you can estimate a lower bound.

At 720p (the low end of mobile resolutions) the write-only bandwidth of 4 G-buffers is 412 MB/s assuming 30 fps. I think it's fairly safe to assume you will read this back in at least once, plus write an updated colour buffer, not to mention potentially an extra buffer or depth copy or two, or geometry pass to work around api hurdles.
So I'd estimate **1 GB/s** is likely to be the minimum saving you'll make, potentially much more. If you are working at a higher resolution – and I think 1080p is more likely in the long run – then this jumps up dramatically.

Hardware can do some things to help though, such as transaction elimination and framebuffer compression.

[Post talk edit – see bonus slides for more on this]

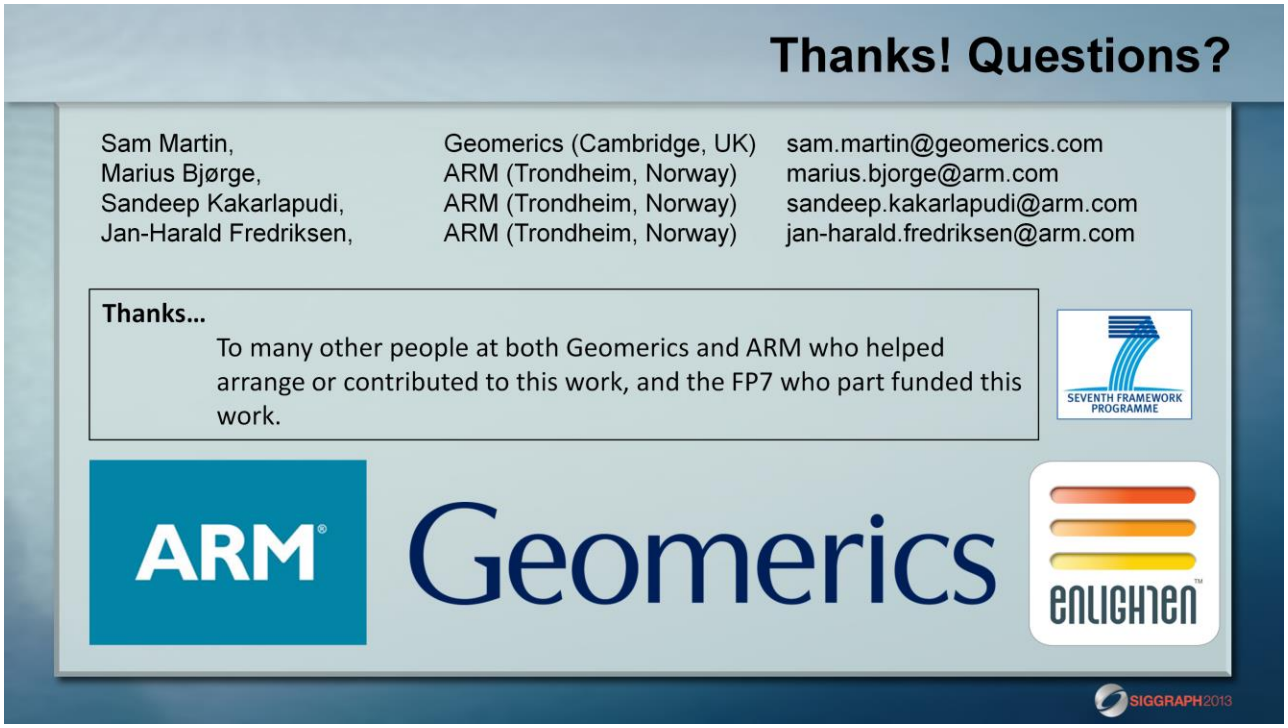


So, the key takeaways I'd like to note from our talk today are:

- The complete awesomeness that is persistent on-chip memory. Tile-based renderers should be looking to make the most out of this, and we should collectively be looking at ways to build new on-chip shading pipelines that take advantage of this new computation model. Deferred rendering is not the only possibility but we are time constrained today.
- These extensions are simple, minimally invasive but very powerful.
- On chip deferred rendering is viable even on current hardware with these extensions, offering a significant quality boost for no extra bandwidth cost, meeting the "more with less" criteria for improving graphics on mobile.

While the current extensions depend on Mali-specifics, the on-chip rendering principles apply to other tile-based architectures. I would encourage other vendors help expose this functionality.

And as a final plea – if you are a hardware vendor and capable of increasing your compute capacity, please see if you can do so! After optimising for bandwidth, we (Geomerics) are not yet drowning in compute resource ☺



Thanks!

I'd particularly like to note that a large number of people (too many to realistically list) from both ARM and Geomerics were involved in helping setting up, contributing to early discussions, and encourage this joint work. This would not have been possible without everyone's input.

Post Talk Update

- Measured bandwidth matches theory!
  - Arches, 720p, 30fps

Technique	TE bytes (60 frames)	BW Read bytes (60 frames)	BW Write bytes (60 frames)	BW MB/s @ 30fps
Off-chip deferred	454,459,392	1,604,044,000	1,114,820,208	**1513.158607**
On-chip deferred	53,551,104	470,340,480	529,007,952	**502.0616226**
Lightstack	46,759,936	831,814,496	643,712,592	**725.8830185**

- Some context for memory bandwidth figures and power consumption

SIGGRAPH2013

So, post talk, we managed to confirm the measured bandwidth saving after going through a fairly detailed breakdown.

One detail that's important to note with the Mali 6xx series is that the hardware includes transaction elimination (TE). This is a hardware optimisation that will skip the write bandwidth for tiles that have not changed since they were last rendered. It requires completely static contents per-tile to kick in, but can be a huge saving for static cameras or UI elements. Our benchmark has a static camera and no textures, and so kicks in all the time, particularly during off-chip G-buffer creation. This would not be the case in general, particularly for a first person shooter. It would also be very hard to budget for if the user can move the camera as you couldn't rely on it producing a consistent saving. So for our purposes we remove the effect of transaction elimination.

We show measured bandwidth for the Arches at 720p, which is still very much on the conservative end of the scale, and demonstrates the expected 1 GB/s saving of on-chip verse off-chip deferred rendering, and further confirms that the cost of the z-pre pass is an unfortunate overhead for the lightstack renderer.

It may be useful to provide some context to this figure. I mentioned some of this verbally during the presentation at SIGGRAPH, and include it here in the notes for completeness.

The amount of bandwidth you have available to use on a **sustained basis** varies between device, and tends to be very different to the headline peak memory bandwidth figures. Peak figures are not possible on the sustained basis that would be required by a game. At Geomerics, we use 2-4 GB/s as a reasonable safe limit for current phone and tablet devices respectively. We understand that 6 GB/s for bleeding edge tablets is possible* and other industry partners have confirmed this. But there is no expectation that significant rises in memory bandwidth will occur any time soon **, and further advances will be incremental.

In terms of power, a reasonable rule of thumb is that 1 GB/s of memory bandwidth costs approximately 150mW of power with current tech. I understand this figure is the result of an average, and the discarded details may swing this figure by perhaps 50% but not an order of magnitude. Industry partners appear to agree on the figure as being a reasonable estimator. Shebanow used the same figure in his HPG keynote.

Using this 150mW estimate, if you then work back from the amount of available power for a device you can produce a second estimate for an upper limit on the available bandwidth. The power available for the entire device (including the screen and radio!) is typically 1 Watt for a phone, 4-5 Watts for a tablet, sometimes more for larger devices. In this context, off-chip deferred rendering at 1080p or Nexus 10 resolutions are likely to consume such a significant portion of the available bandwidth and power as to make the technique completely impractical.

* Michael Shebanow, in “An Evolution of Mobile Graphics”, HPG keynote.
http://highperformancegraphics.org/wp-content/uploads/Shebanow-Keynote.pdf

** Ed Plowman et al, in “Optimized Effects For Mobile Devices” includes a graph estimating future memory bandwidth scaling.
http://www.gdcvault.com/play/1018165/Optimized-Effects-for-Mobile-Devices

And finally, as a last bonus slide, we've since tried the on-chip deferred technique with a slightly more realistic shader and scene.

This is a section of another next-gen demo asset ported to mobile, with around 300k verts, but now includes high res albedo and normal maps everywhere, a baked AO channel texture, Enlighten, filmic tonemapping, blinn-phong specular everywhere, about 20 lights, and some other bells and whistles. The albedo has been desaturated to show the lights. It runs between 33-40 fps depending on where the lights are.