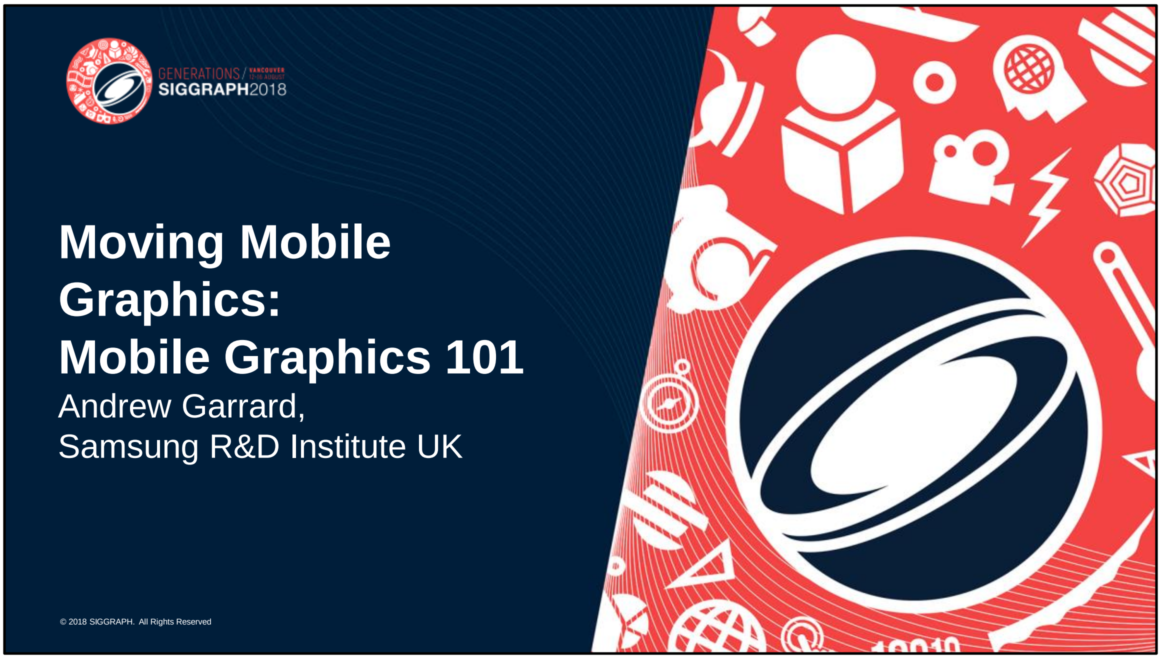
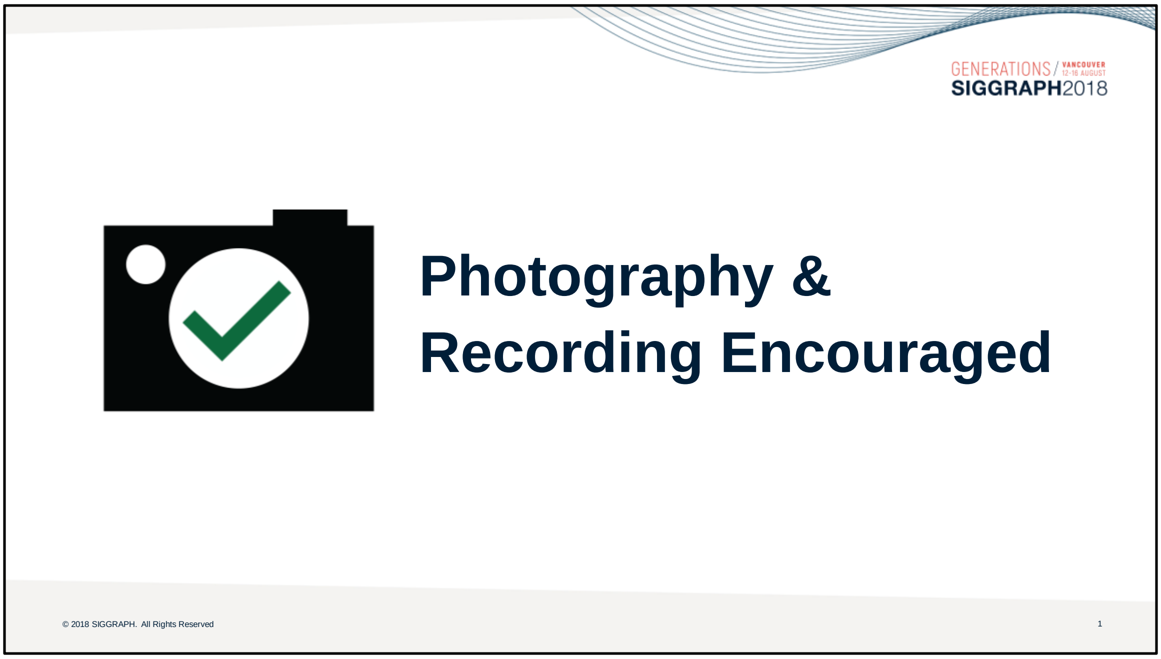


GENERATIONS / VANCOUVER 12-16 AUGUST
SIGGRAPH2018

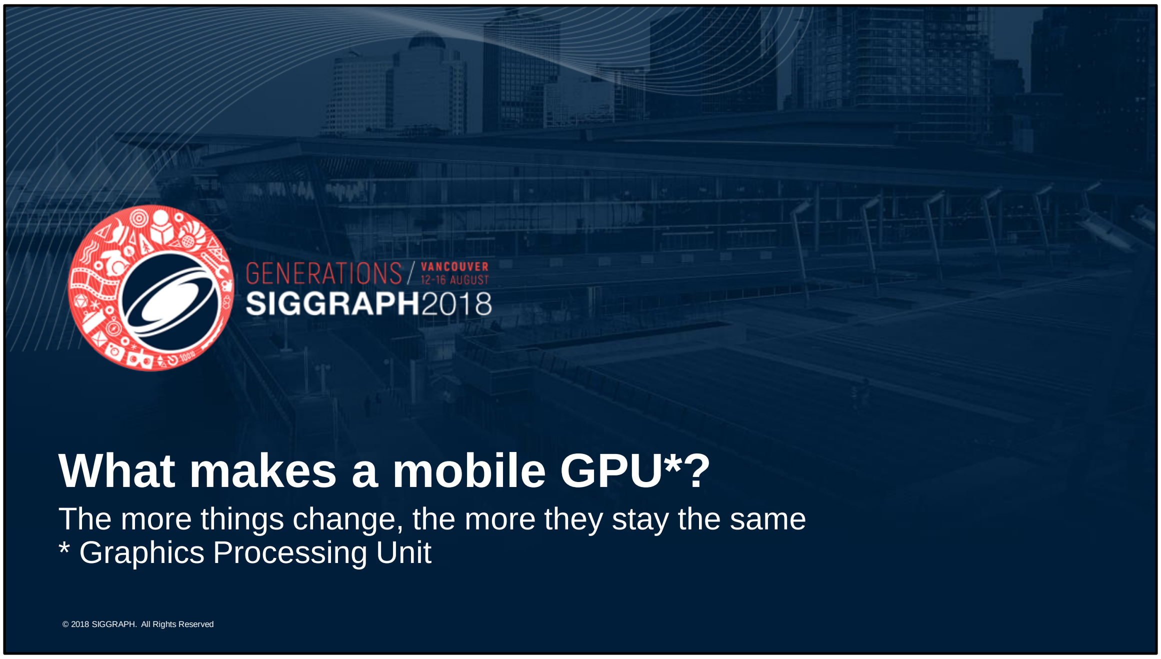
Moving Mobile Graphics: Mobile Graphics 101

Andrew Garrard,
Samsung R&D Institute UK



GENERATIONS / VANCOUVER 12-16 AUGUST
SIGGRAPH2018

Photography & Recording Encouraged

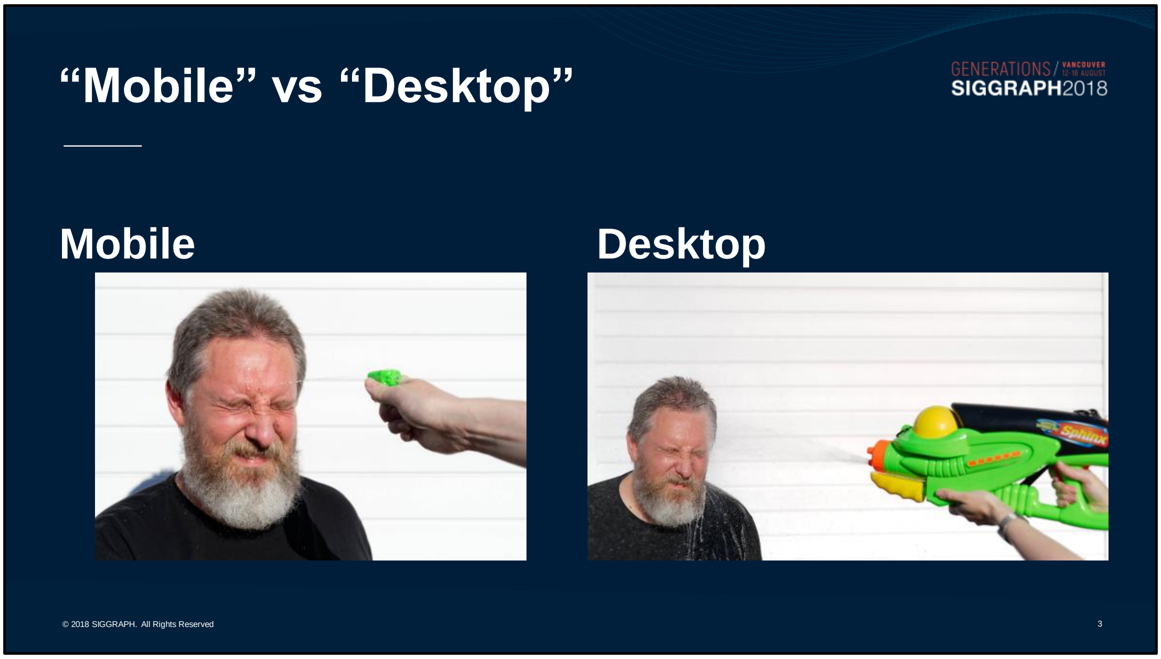


GENERATIONS / VANCOUVER 12-16 AUGUST
SIGGRAPH2018

What makes a mobile GPU*?

The more things change, the more they stay the same
* Graphics Processing Unit

This is an introductory talk about the differences (and similarities) between “mobile” and “desktop” GPUs, and how to target the former with a modern graphics API.
Just to be clear: “GPU” is “graphics processing unit”; talks like this have been given to very confused people who didn’t know that.

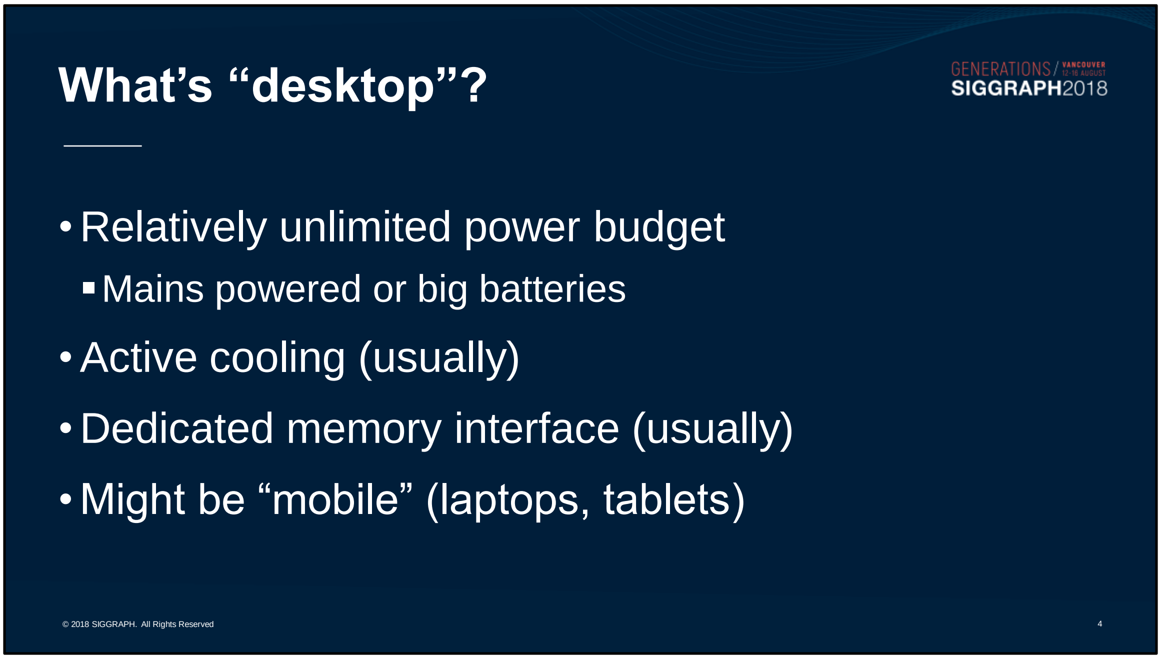


“Mobile” vs “Desktop”

Mobile

Desktop

Mobile and desktop GPUs are doing basically the same thing. The big difference is just one of power – both performance and current draw; this does affect what you can do with it. While there’s some cross-over, “mobile” architectures are designed for a very power-constrained environment, and “desktop” architectures have more elbow room for power.

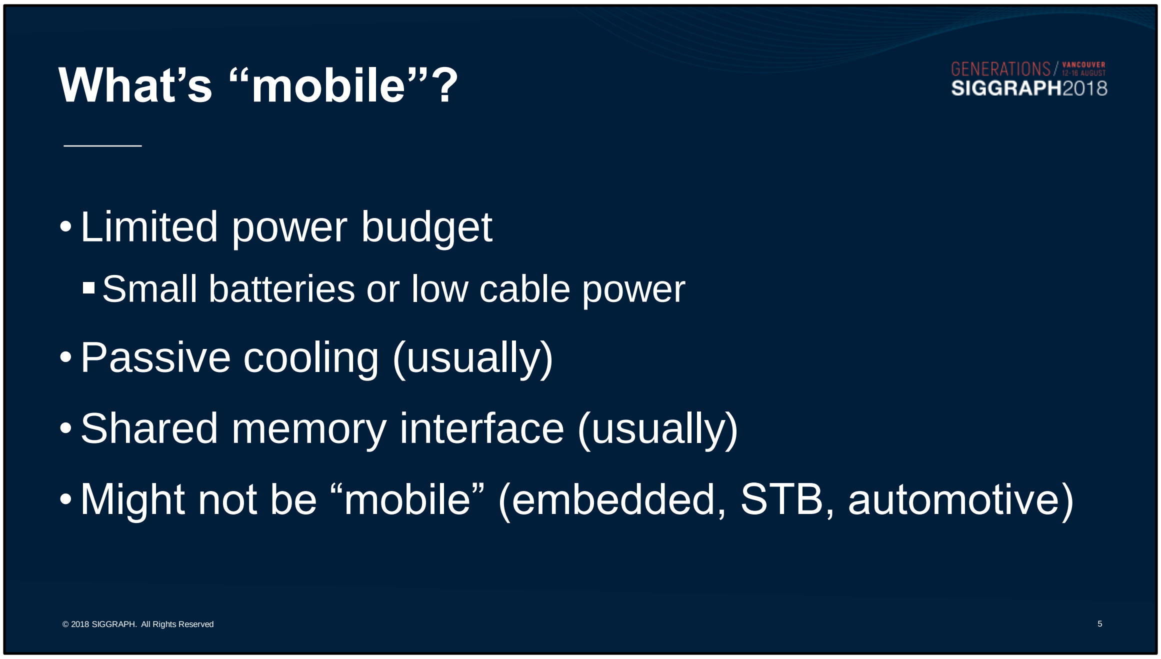


What’s “desktop”?

GENERATIONS / VANCOUVER 12-16 AUGUST
SIGGRAPH2018

- Relatively unlimited power budget
  - Mains powered or big batteries
- Active cooling (usually)
- Dedicated memory interface (usually)
- Might be “mobile” (laptops, tablets)

What do I mean by a “desktop GPU”?
The architectures are typically designed to be plugged into a PC, which means plenty of power is available (a bit less if you’re in a laptop), and plenty of cooling is available (unless the laptop or tablet is fanless).
Desktop graphics cards and some integrated graphics systems have a dedicated graphics memory interface with fast RAM optimised for GPU usage; some integrated systems just use the system memory.
The GPU architecture typically used by most laptops and PC-based tablets are derived from desktop systems, despite having more constraints.

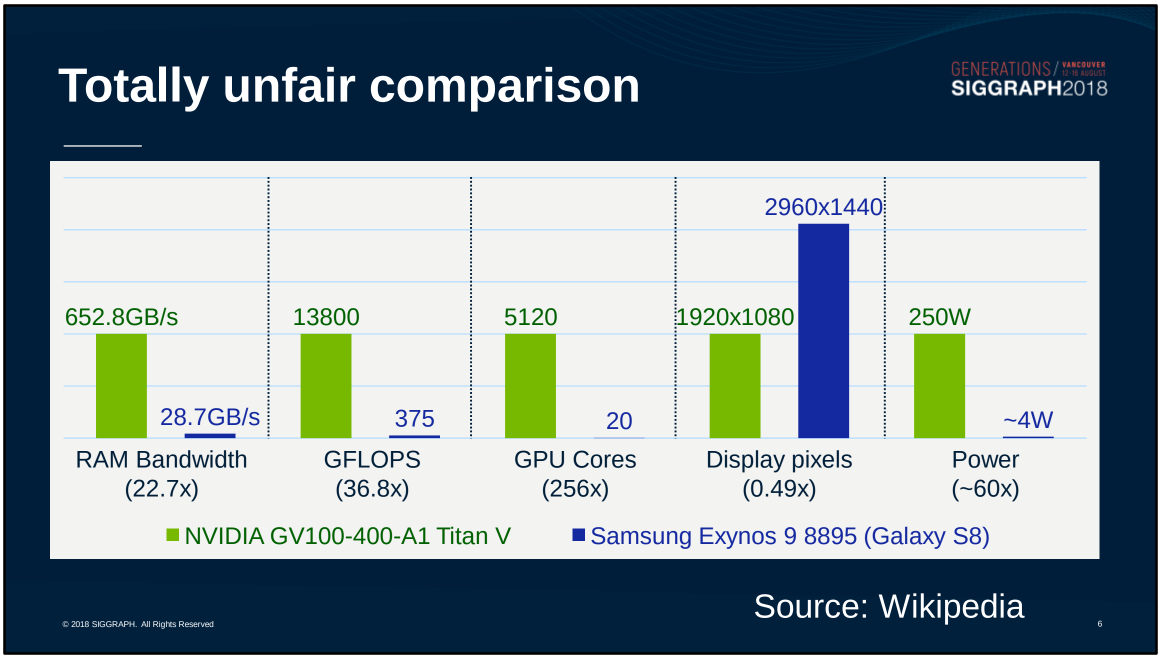


What’s “mobile”?

GENERATIONS / VANCOUVER 12-16 AUGUST
SIGGRAPH2018

- Limited power budget
  - Small batteries or low cable power
- Passive cooling (usually)
- Shared memory interface (usually)
- Might not be “mobile” (embedded, STB, automotive)

A mobile GPU is designed for a lower power budget – although in an embedded environment such as running off a USB socket, that doesn’t necessarily mean a battery.
They usually don’t have fans, although some embedded systems do, and there’s talk of some gaming phones having active cooling attached.
Because of the size of the packaging, there’s usually no dedicated graphics memory. Mobile GPUs did once keep the whole framebuffer in dedicated memory, which was fast, but very restrictive.
And just because we’re calling them mobile, that doesn’t mean you can physically pick them up. These systems get used in cars, security cameras, TVs and set top boxes and so on.

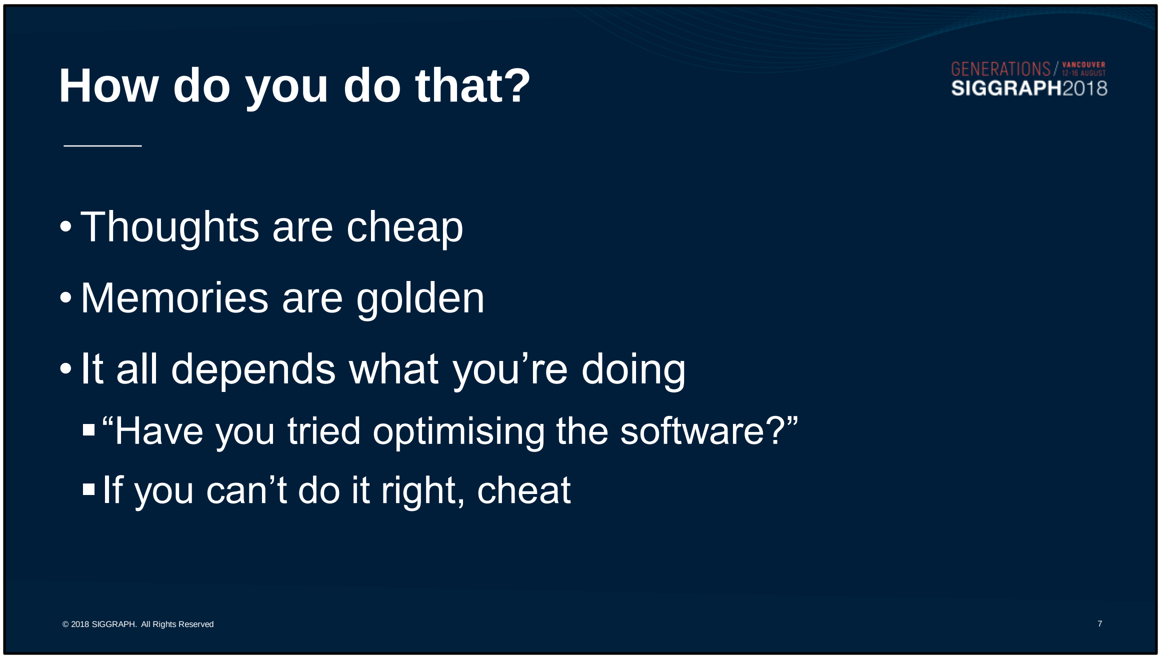


Totally unfair comparison

Source: Wikipedia

At the extremes, the comparison isn't close. Here's a high-end NVIDIA Titan V (green team) against the blue team of the Exynos from the Galaxy S8 – these are figures from Wikipedia, and the S9 ones weren't to hand.
Compared with the mobile chip, the Tegra has over twenty times the bandwidth to its main memory, over thirty times the compute, and over 200 times the cores.
The Volta die is also about 8x the size of the Exynos – and note that a relatively small proportion of the Exynos is GPU: there are 8CPU cores in there, for a start.

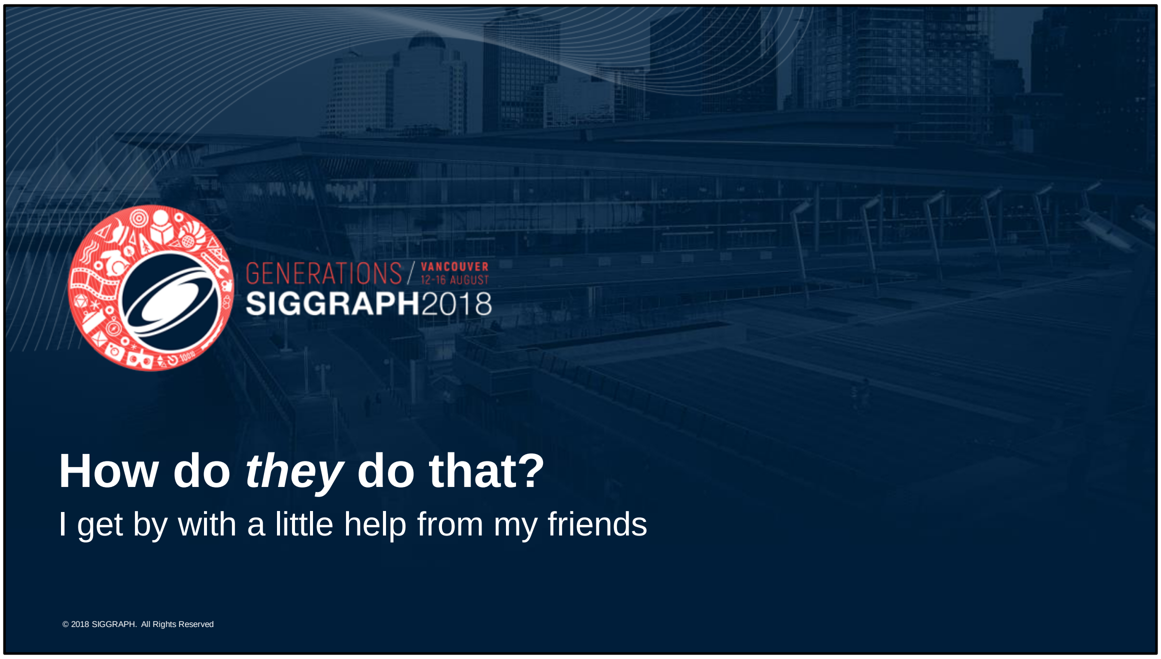
But the phone screen is roughly twice the resolution of a typical desktop.
And the Volta also has sixty times the power budget of a phone (I've heard between 3 and 5W) – so although it's slower, the mobile GPU has to be much more efficient.
Nothing like a challenge.



How do you do that?

- Thoughts are cheap
- Memories are golden
- It all depends what you’re doing
  - “Have you tried optimising the software?”
  - If you can’t do it right, cheat

Compute isn’t such a problem, in general – but memory access is a scarce resource, because waggling those external data pins takes a lot of power.
Whether the mobile GPU can keep up depends highly on the workload.
I’ve discussed faster hardware with managers in the past, and literally been told “have you tried optimising the software”. Yes, we did. That’s still mostly the solution.
It’s all atmospherics and reflectance maps – I mean smoke and mirrors. And you have to pick your fights. All of computer graphics is cheating, and in mobile, sometimes you have to cheat a lot.

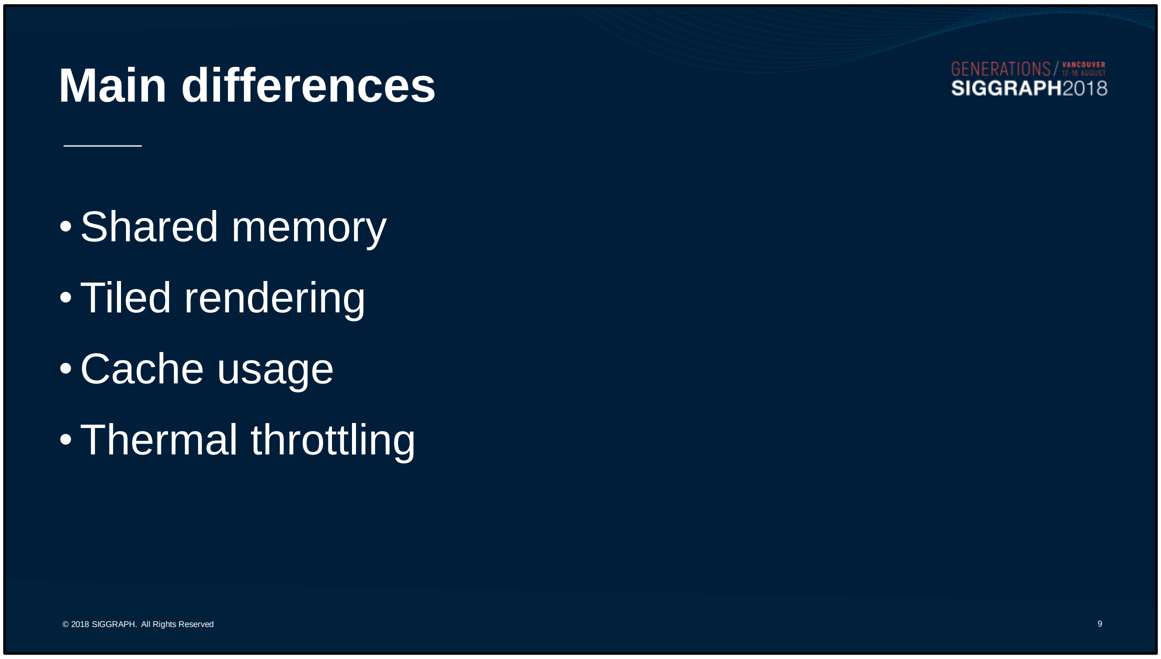


GENERATIONS / VANCOUVER 12-16 AUGUST
SIGGRAPH2018

How do *they* do that?

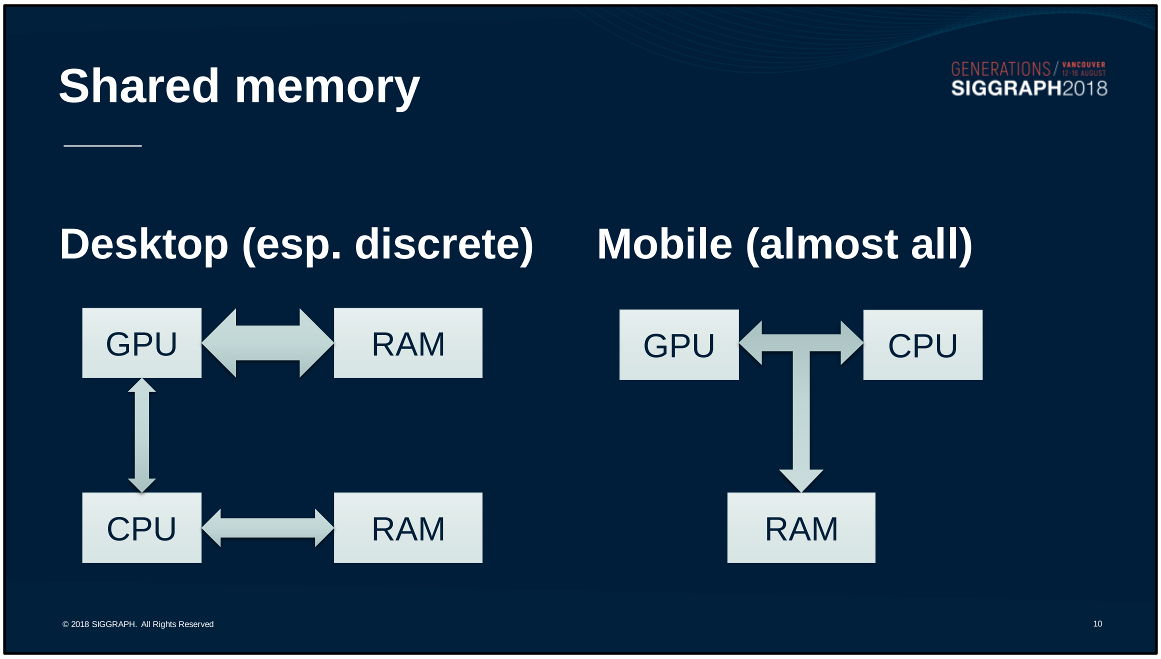
I get by with a little help from my friends

So what’s special about a mobile GPU that allows it to render efficiently compared with a desktop?
Several things – but they are things that impact on the software, so you need to work with them.



Main differences

- Shared memory
- Tiled rendering
- Cache usage
- Thermal throttling



Shared memory

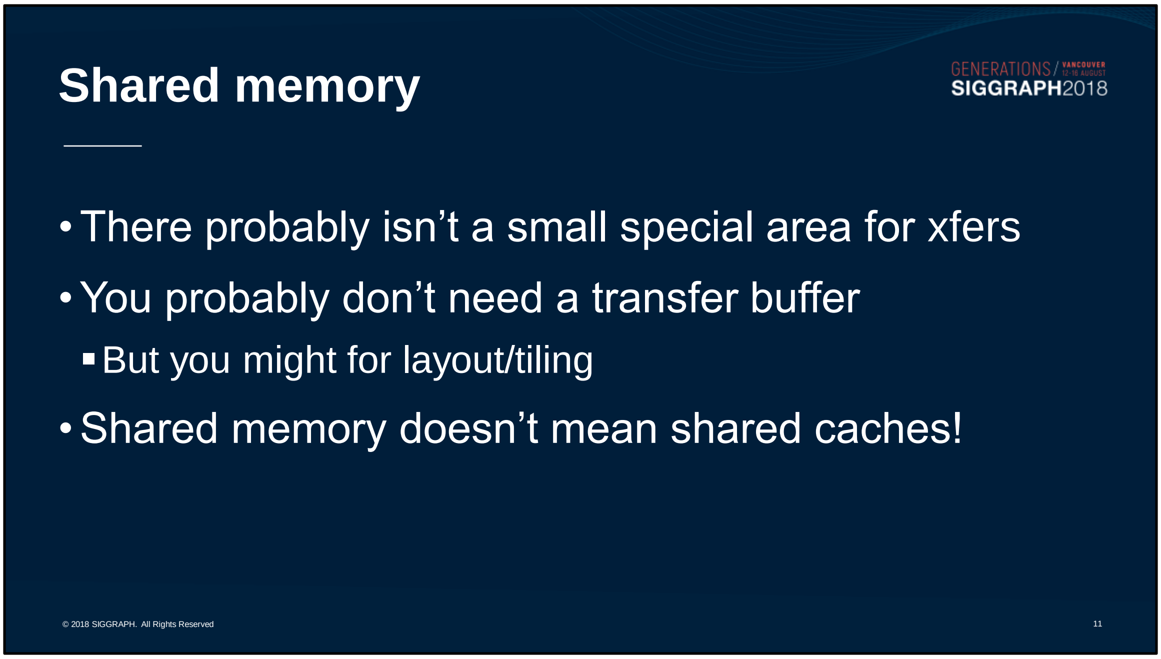
Desktop (esp. discrete)

GPU
RAM
CPU
RAM

Mobile (almost all)

GPU
CPU
RAM

Desktop graphics cards – especially discrete systems – tend to have dedicated graphics memory attached to the GPU, meaning that the GPU and CPU aren't contesting resources.
The GPU and CPU are connected via a relatively slow interface (such as PCI-e), but their RAM interfaces are separate, and usually faster.
On a mobile system, there's only one memory interface, saving power and space, but limiting the overall performance.
Technically this means if the CPUs and GPU are both doing a memory-heavy operations, there could be contention.



Shared memory

- There probably isn't a small special area for xfers
- You probably don't need a transfer buffer
  - But you might for layout/tiling
- Shared memory doesn't mean shared caches!

More importantly, on a next-gen API, there are some direct impacts of the memory architecture.

On a discrete memory system you may need a transfer area in the shared memory window. For example, you might copy textures there from the CPU, then copy them to dedicated GPU memory, which has more space and performance.

On a shared memory system this isn't an issue – but you might still need to do a copy to change the layout or image tiling of the image.

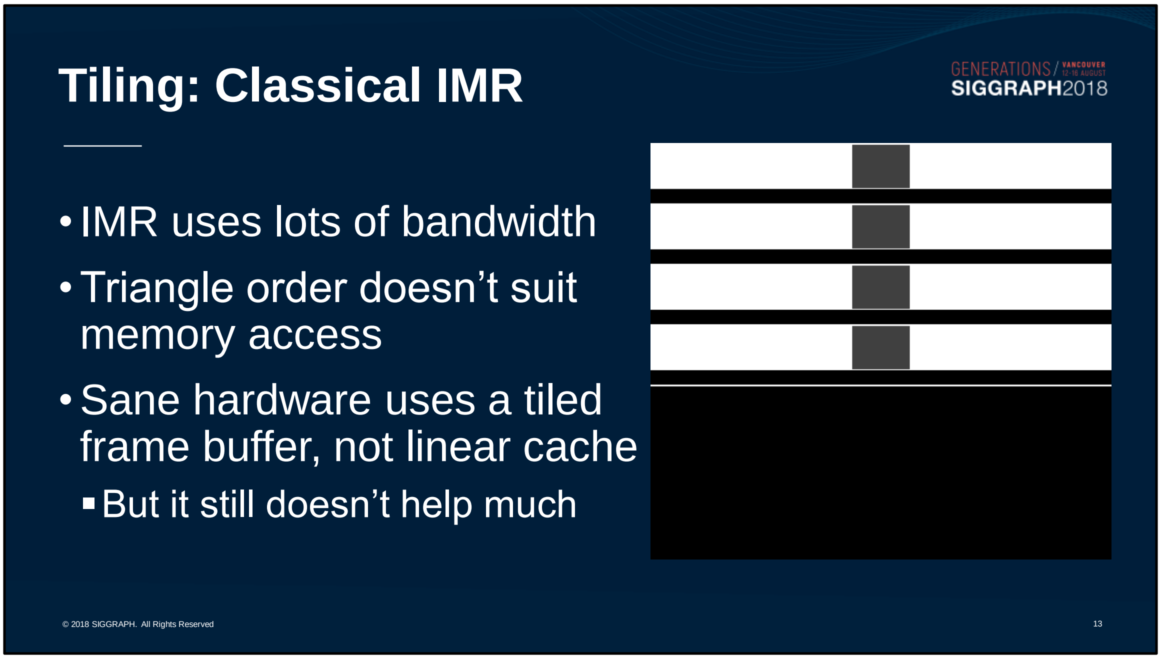
Bear in mind that just because the memory is shared doesn't necessarily mean the caches are synchronous.

Tiling: Classical IMR

GENERATIONS / VANCOUVER 12-16 AUGUST
SIGGRAPH2018

- “Immediate mode” rendering:
  triangles render in the order submitted
- Depth updated as you go
- It’s never this simple
  - We need more parallelism!

The most obvious architectural difference between mobile and desktop is that most mobile systems use tiled rendering, whereas historically desktop systems have been “immediate mode renderers”.
In an IMR, as each triangle is submitted its data enters the GPU pipeline, and all the pixels for this triangle (and its Z buffer values) pop out of the other end of the pipeline.
In practice this approach doesn’t expose as much parallelism as we’d like.

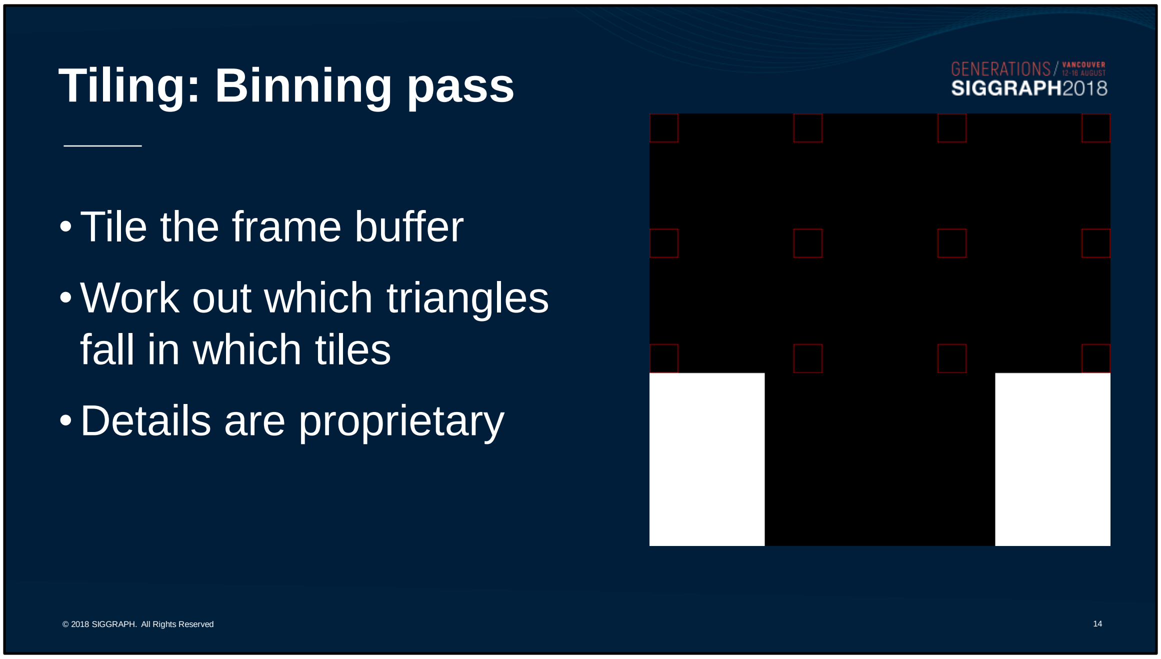


Tiling: Classical IMR

GENERATIONS / VANCOUVER 12-16 AUGUST
SIGGRAPH2018

- IMR uses lots of bandwidth
- Triangle order doesn't suit memory access
- Sane hardware uses a tiled frame buffer, not linear cache
  - But it still doesn't help much

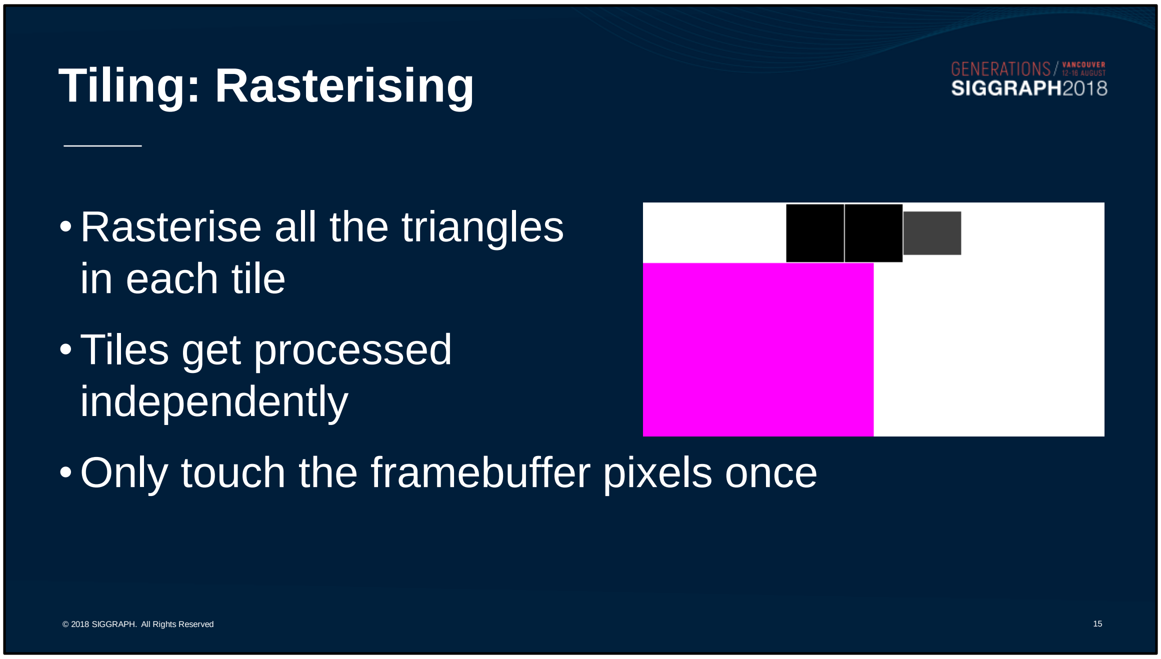
More importantly for a mobile GPU, rendering each triangle in order leads to fairly incoherent memory access.
Here's the same image rendered with some virtual cache lines – the grey boxes are showing where the cache line falls in the image.
The triangles aren't processed in a way that maximises cache access, and so the cache thrashes – there would be a lot of memory traffic.
(This example is simplified with horizontal cache lines, but no arrangement helps with unordered triangles).



Tiling: Binning pass

- Tile the frame buffer
- Work out which triangles fall in which tiles
- Details are proprietary

A solution is to sort the triangles so that they are spatially coherent.
A tiled renderer has the frame buffer broken down into rectangular tiles (4x3 in this example – usually a lot more).
A “binning” pass checks each triangle and determines whether it touches each tile.

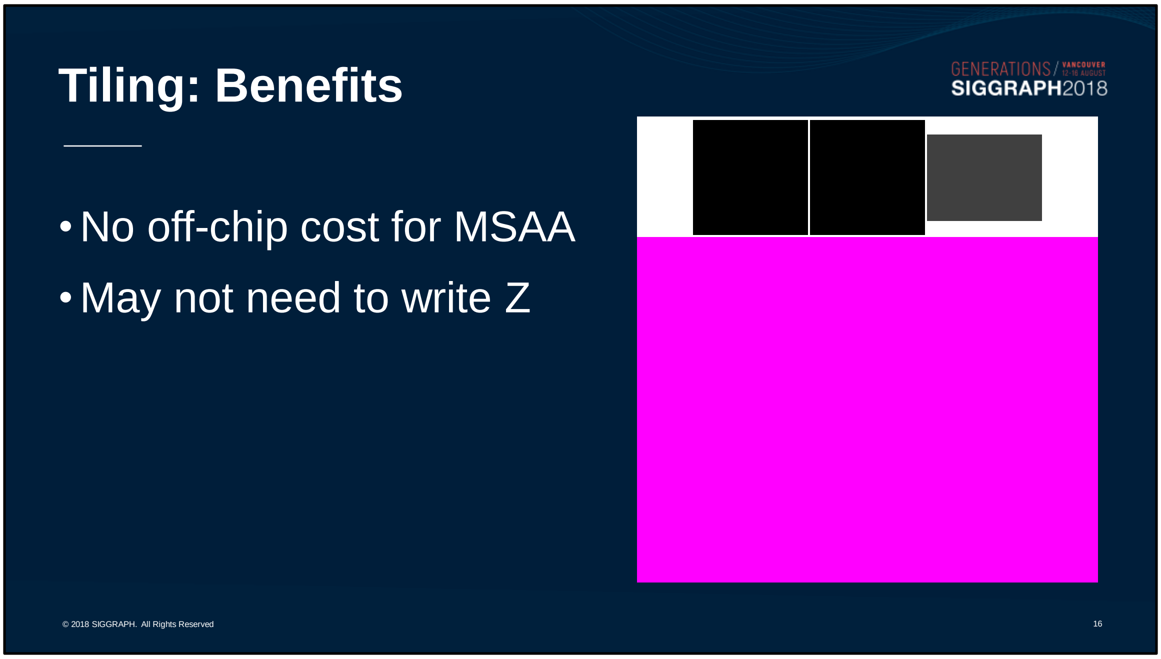


Tiling: Rasterising

GENERATIONS / VANCOUVER 12-16 AUGUST
SIGGRAPH2018

- Rasterise all the triangles in each tile
- Tiles get processed independently
- Only touch the framebuffer pixels once

Then a tiled renderer walks through each tile independently and draws only the triangles that have been binned as touching that tile.
By only rendering the area within the current tile, this area can be processed in fast on-GPU memory.
The tile then gets written out to main memory only when rendering is finished, so memory only gets touched once.
So long as walking over the scene description for each tile uses less bandwidth than writing to the frame buffer (and it very likely does), this saves bandwidth.

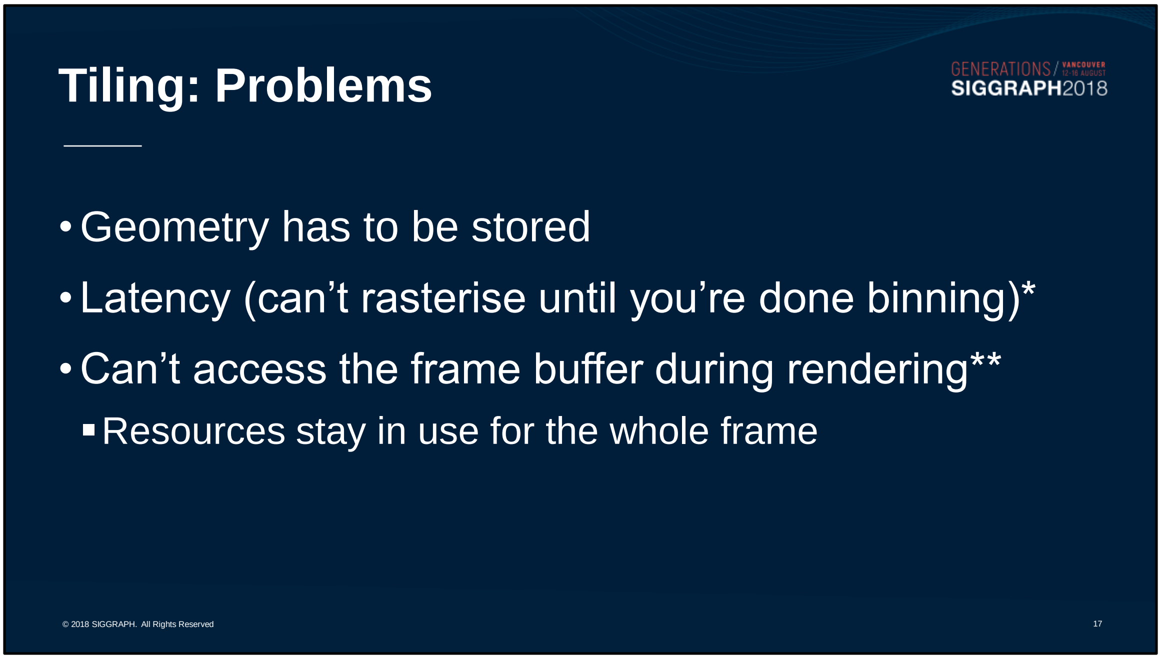


Tiling: Benefits

GENERATIONS / VANCOUVER 12-16 AUGUST
SIGGRAPH2018

- No off-chip cost for MSAA
- May not need to write Z

If we're tiling, there are a few other benefits.
One is that if we don't need the depth buffer (or other intermediate attachments) to be in main memory for anything, it may never need to be written off the chip.
Another is that we can render at higher resolution internally and downscale as the frame buffer is written out, which is a cheap way to get antialiasing.

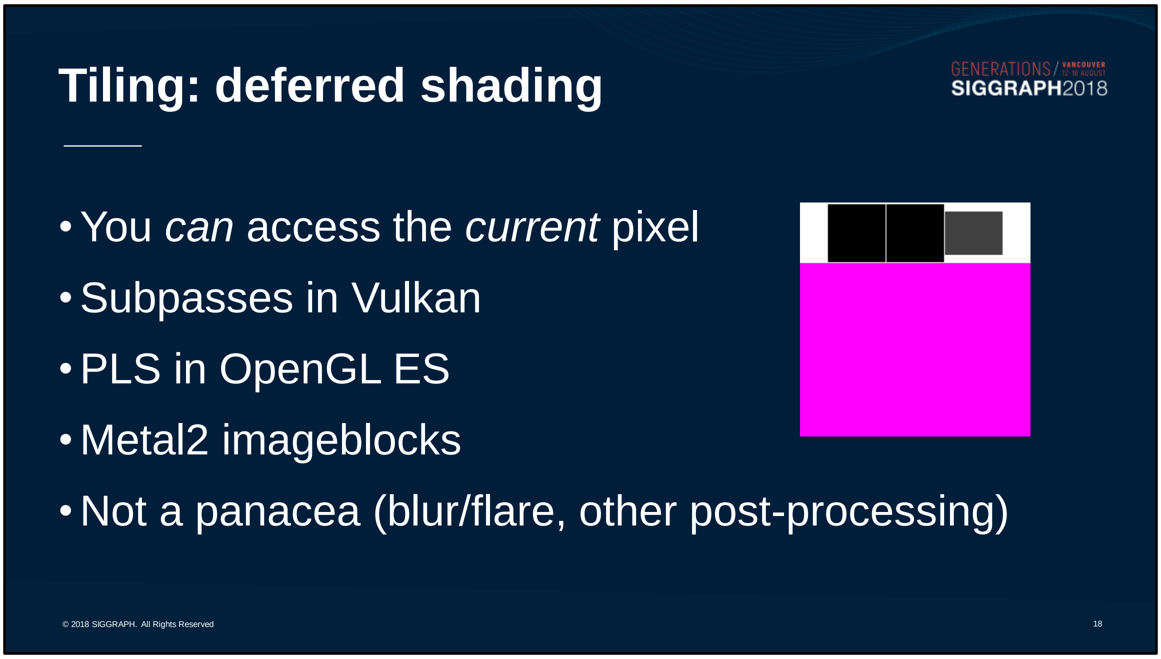


Tiling: Problems

GENERATIONS / VANCOUVER 12-16 AUGUST
SIGGRAPH2018

- Geometry has to be stored
- Latency (can’t rasterise until you’re done binning)*
- Can’t access the frame buffer during rendering**
    - Resources stay in use for the whole frame

Why doesn’t everything tile?
To process it for each tile, the scene description has to live somewhere. For a mobile game it shouldn’t be very large. If you’re rendering an enormous workstation model, it could be prohibitively large.
Rasterising can’t start until geometry is processed – which adds latency, although it might reduce the total frame time. There are solutions with “partial tiling” where a sliding window of geometry over the scene is processed in tile order, and it seems that the latest desktop cards are using this.
Because the image isn’t finished until all the tiles are rendered, you also can’t access arbitrary locations in the frame buffer during rendering – which hampers some rendering techniques. Fortunately you can know what’s happened at your pixel location.
The tiling also means resources stay in use for longer – for example if a texture is used on one piece of geometry, a later primitive can’t reuse the texture memory, because other tiles may still rely on its original value.
** Except the local tile.



Tiling: deferred shading

GENERATIONS / VANCOUVER 12-16 AUGUST
SIGGRAPH2018

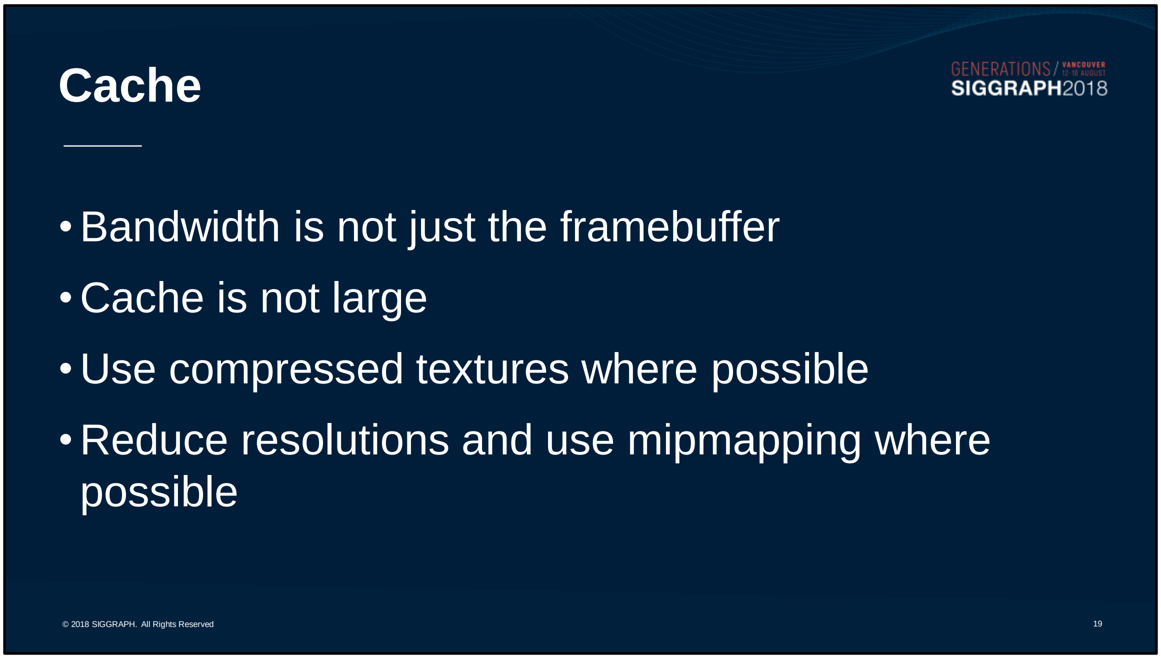
- You *can* access the *current* pixel
- Subpasses in Vulkan
- PLS in OpenGL ES
- Metal2 imageblocks
- Not a panacea (blur/flare, other post-processing)

One thing you can do with access just to the current pixel is screen-space shading. Game developers know this as deferred shading or deferred lighting, but those terms get used for lots of other things as well.

In this version you can write a scene to a geometry buffer at low cost, then shade the scene more efficiently once all the geometry is rendered.

In Vulkan you do this with subpasses; in GL ES you can use pixel local storage, and Metal has image blocks.

This doesn't always help – if you want to filter the image for motion blur, depth of field, etc. it won't help.



Cache

GENERATIONS / VANCOUVER 12-16 AUGUST
SIGGRAPH2018

- Bandwidth is not just the framebuffer
- Cache is not large
- Use compressed textures where possible
- Reduce resolutions and use mipmapping where possible

Tilers help with frame buffer bandwidth, but there are other demands on the memory, and caches aren’t big enough to get you out of jail.
It can make a significant difference to minimise the texture sizes – which includes using compression schemes and mipmapping.

GENERATIONS / VANCOUVER 12-16 AUGUST
SIGGRAPH2018

Thermal throttling

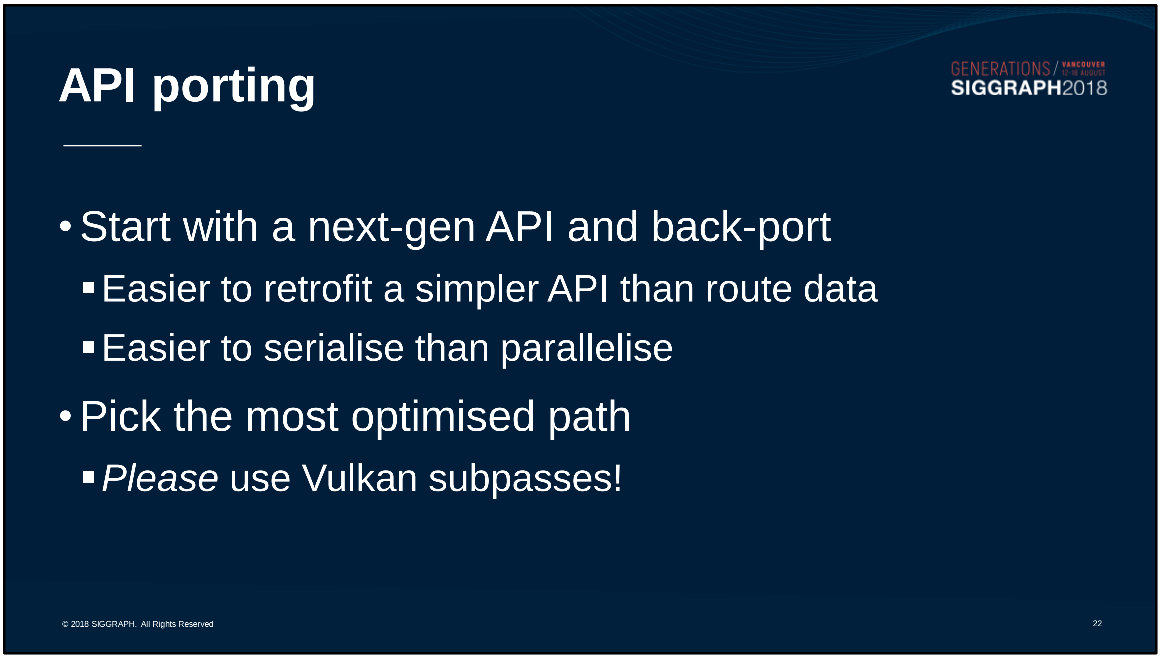
- Throttle back when it’s too hot
- Peak speed cannot be maintained
  - Passive cooling (usually)
  - Cases make it worse
- Degrade gracefully
- Consider bursty workloads

The final way that mobile GPUs handle their limitations is that they throttle the clock speed when things get too hot.
This doesn’t happen so much in desktop systems where a fan can just spin up to add cooling: your phone can’t do that (although there’s talk of gaming phones with fans).
A cell phone has to handle worst-case cooling conditions without being uncomfortable to touch.
Rather than assuming your app can maintain peak performance, you may want to be ready to manage the workload dynamically.
It can help to have bursts of action, with less load in between – for example, keep loading screens at a low GPU load.

Stay cool

It’s easy to be cool with enough airflow

Remember: a desktop or even laptop can blow air around, but you’re relying on much less efficient heat transfer in mobile – and a phone can’t even get as hot as would be efficient for dissipation without hurting the user.

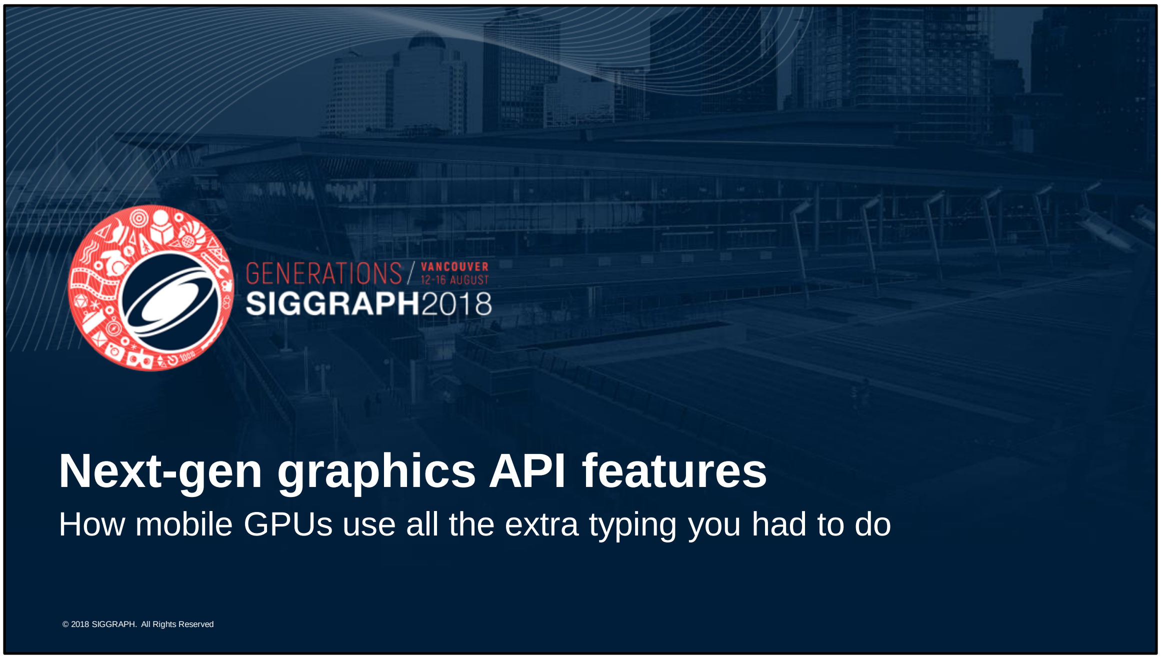


API porting

GENERATIONS / VANCOUVER 12-16 AUGUST
SIGGRAPH2018

- Start with a next-gen API and back-port
  - Easier to retrofit a simpler API than route data
  - Easier to serialise than parallelise
- Pick the most optimised path
  - *Please* use Vulkan subpasses!

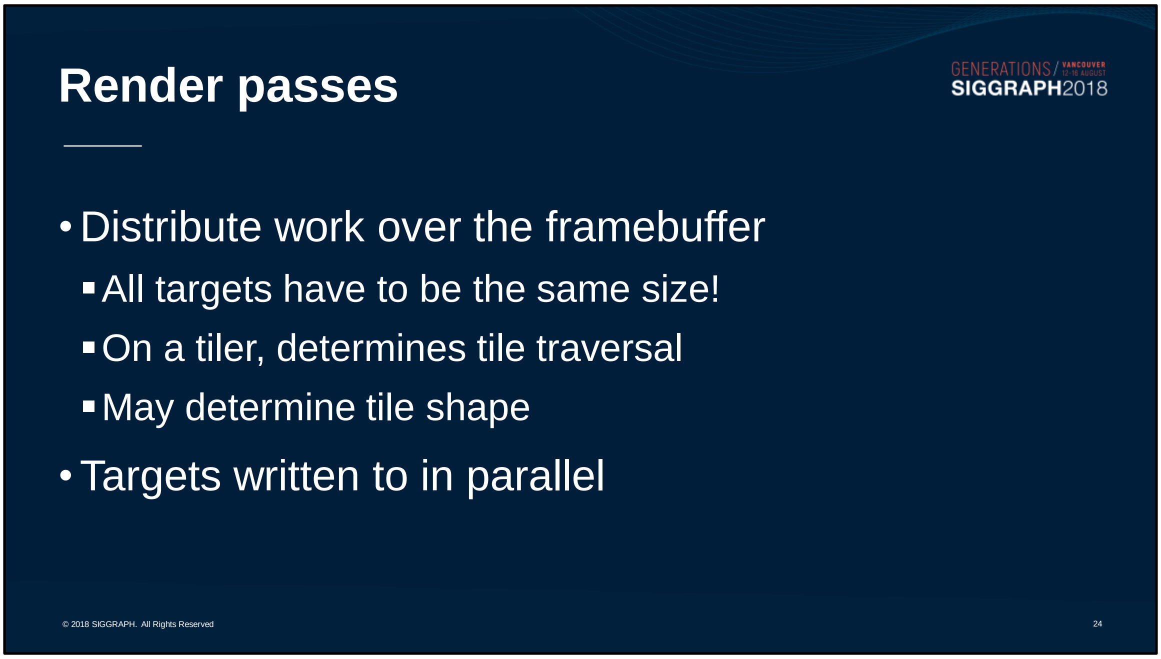
The more direct control over the GPU’s actions that you get with the newer graphics APIs can really help with using mobile GPUs efficiently.
If you’re ever likely to do this, it’s worth considering as you design your rendering engine – something that works fine in order APIs like OpenGL ES may need a lot of work to change to newer APIs because some information isn’t available when it’s needed.
Part of that message is that the newer APIs are designed to allow better use of CPU parallelism; there are often eight cores available on a cell phone GPU, and only using one discards performance. If you design around a single-threaded API, making the engine multi-threaded is a lot of work.
Ideally, when you’re designing an engine, look at the what features are available in each, and make sure you have the necessary arguments accessible, even if the API you’re currently targeting doesn’t need it. Features like subpasses may be hard to use if your engine can’t expose the data to use them, but these engines leave a lot of performance on the floor. If you don’t use the new APIs to their fullest extent, you may get no benefit from using them.


GENERATIONS / VANCOUVER 12-16 AUGUST
SIGGRAPH2018

Next-gen graphics API features

How mobile GPUs use all the extra typing you had to do

One problem of the new APIs is that they put a lot of work on the developer to provide the information that the drivers need. Some of this information seems redundant, so this section runs through what the features may actually do in hardware, and how to use them.



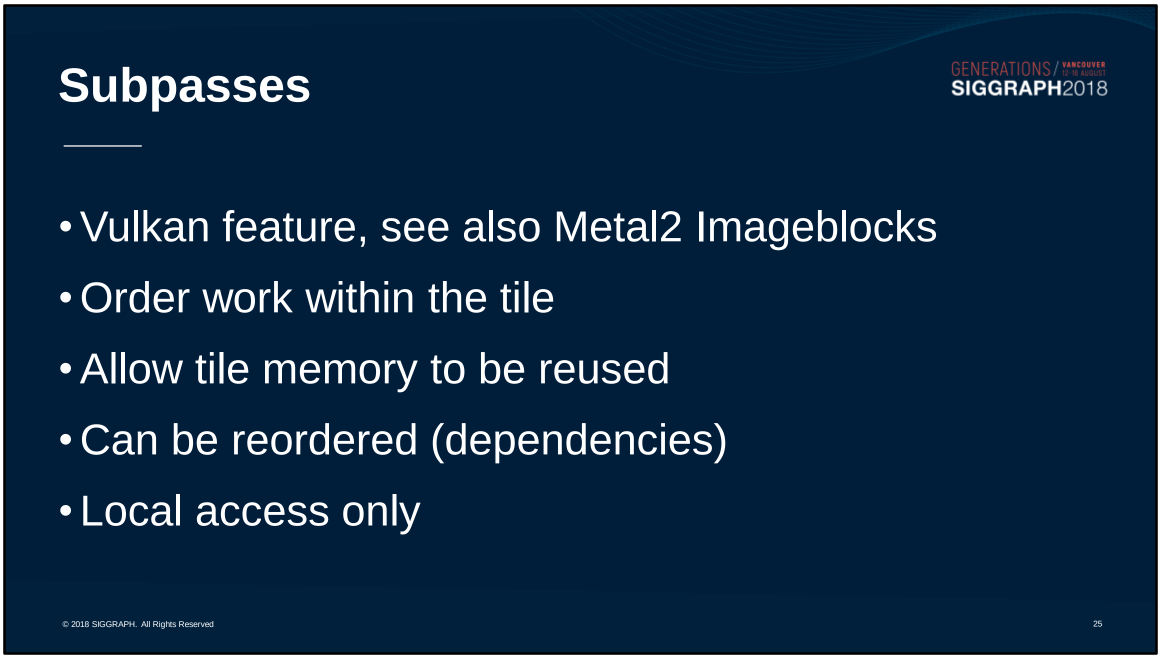
Render passes

GENERATIONS / VANCOUVER 12-16 AUGUST
SIGGRAPH2018

- Distribute work over the framebuffer
  - All targets have to be the same size!
  - On a tiler, determines tile traversal
  - May determine tile shape
- Targets written to in parallel

A render pass allows the rendering load to be configured to run on a framebuffer. Particularly, on a tiler, this is the granularity at which the framebuffer is split into tiles and work on them is scheduled. If you're writing to multiple attachments, this is why the sizes have to match. If you're doing some work that can't be run in tile order, it typically doesn't live inside a render pass.

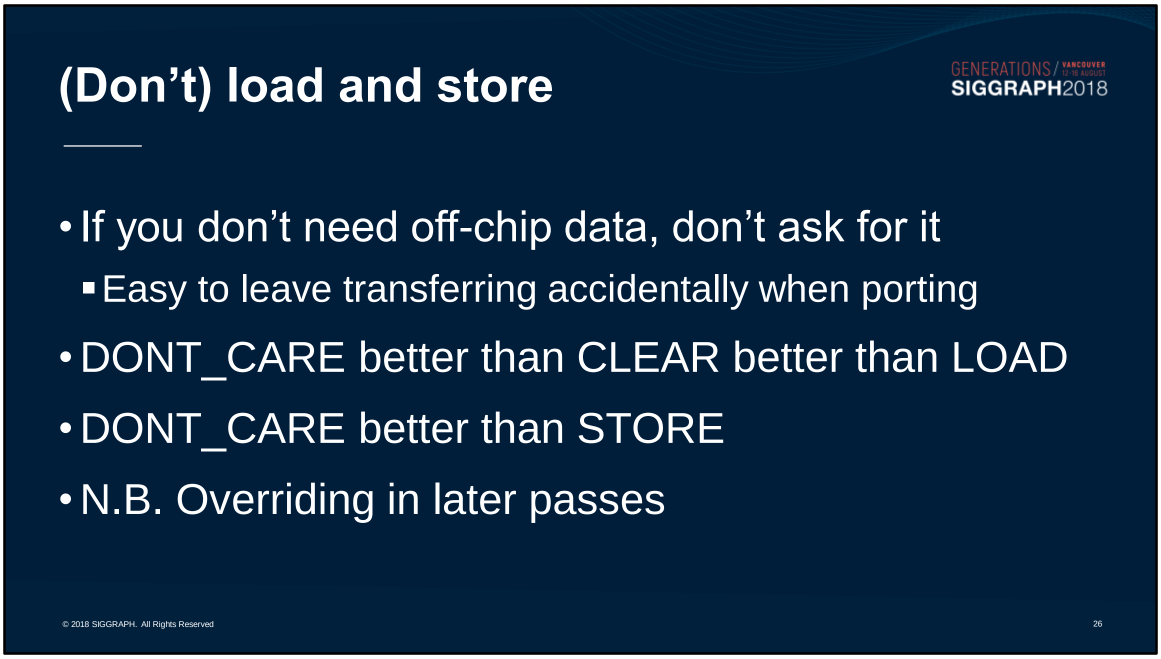
Within the processing of tiles, as much as the hardware is capable the framebuffer attachment render targets get written to in parallel, in tile order.



Subpasses

- Vulkan feature, see also Metal2 Imageblocks
- Order work within the tile
- Allow tile memory to be reused
- Can be reordered (dependencies)
- Local access only

Subpasses in Vulkan (and image blocks in Metal) allow you to submit some work that will be applied to each tile of the image.
By moving to a new subpass, you’re indicating some implicit synchronisation between operations.
While it’s verbose, the dependency system for attachments lets the driver optimise the use of on-chip memory, which is a limited resource.
This includes changing the order of (independent) subpasses.
But, as described above, the things you do in a subpass can only access the framebuffer information stored on chip.

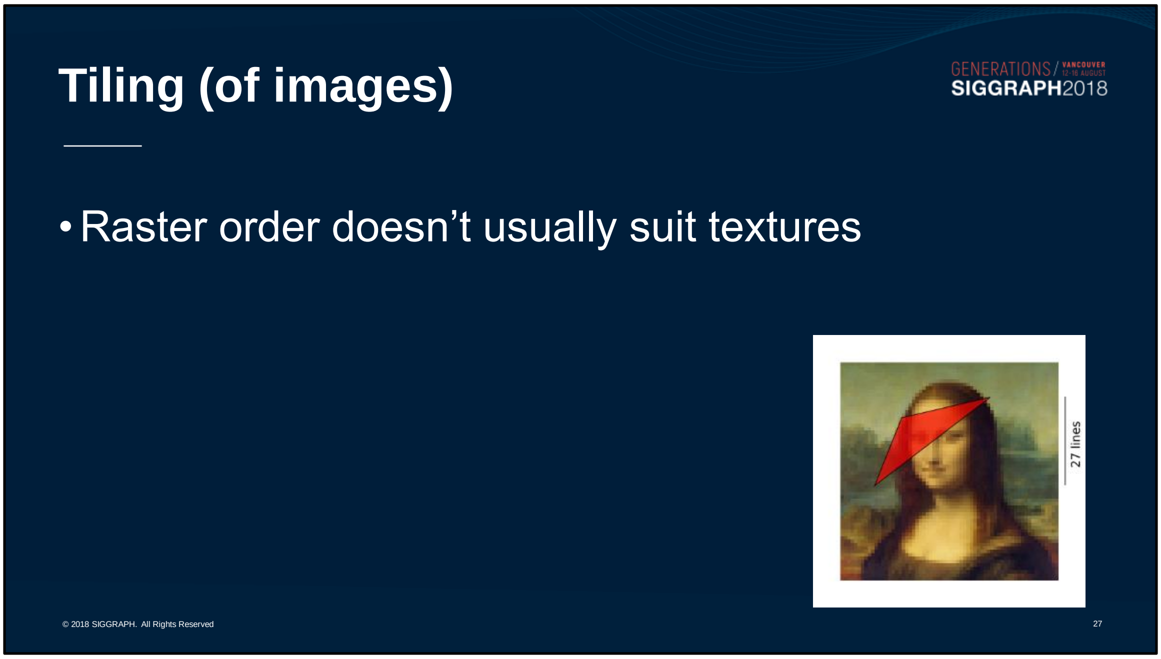


(Don’t) load and store

GENERATIONS / VANCOUVER 12-16 AUGUST
SIGGRAPH2018

- If you don’t need off-chip data, don’t ask for it
  - Easy to leave transferring accidentally when porting
- DONT_CARE better than CLEAR better than LOAD
- DONT_CARE better than STORE
- N.B. Overriding in later passes

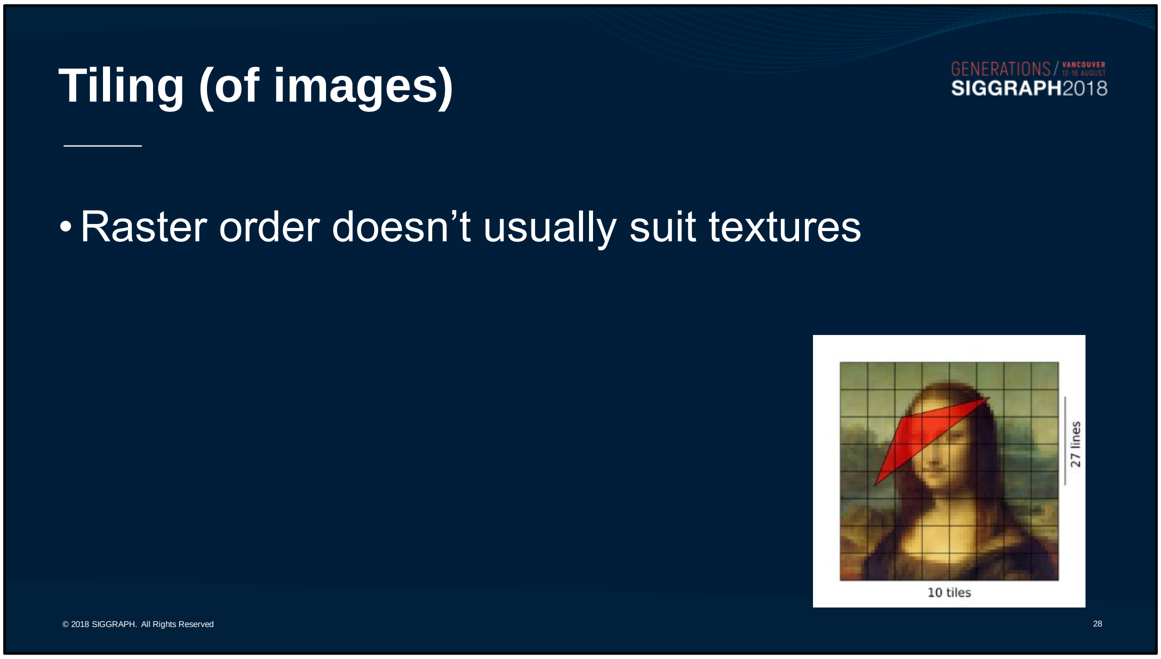
In the new APIs, you explicitly say what memory transfers you want to happen with your attachments – this is more implicit in APIs like OpenGL.
Since memory transfers are so important, the onus is on you to get this right.
For load operations, if you will write over the whole attachment (for example a colour framebuffer), the most efficient thing is to say you don’t care what was there before.
If you need it (e.g. for depth), a clear is still efficient on most tillers.
If you need to preserve the previous render target, you can – but since you’ll have to transfer the framebuffer into memory, don’t unless you need to.
Similarly only write out attachments that you want to keep – don’t preserve depth or multisampled buffers unless you really need them.
Heads up: one render pass using “STORE_OP_STORE” followed by another that reads and never modifies the attachment might think to use STORE_OP_DONT_CARE (because the second pass doesn’t write anything). That will override the STORE_OP_STORE!



Tiling (of images)

- Raster order doesn't usually suit textures

The order in which the pixels in a triangle are processed are unlikely, if the triangle has arbitrary orientation, to suit a simple linear image layout.
This may well be more data than would nicely fit in cache.



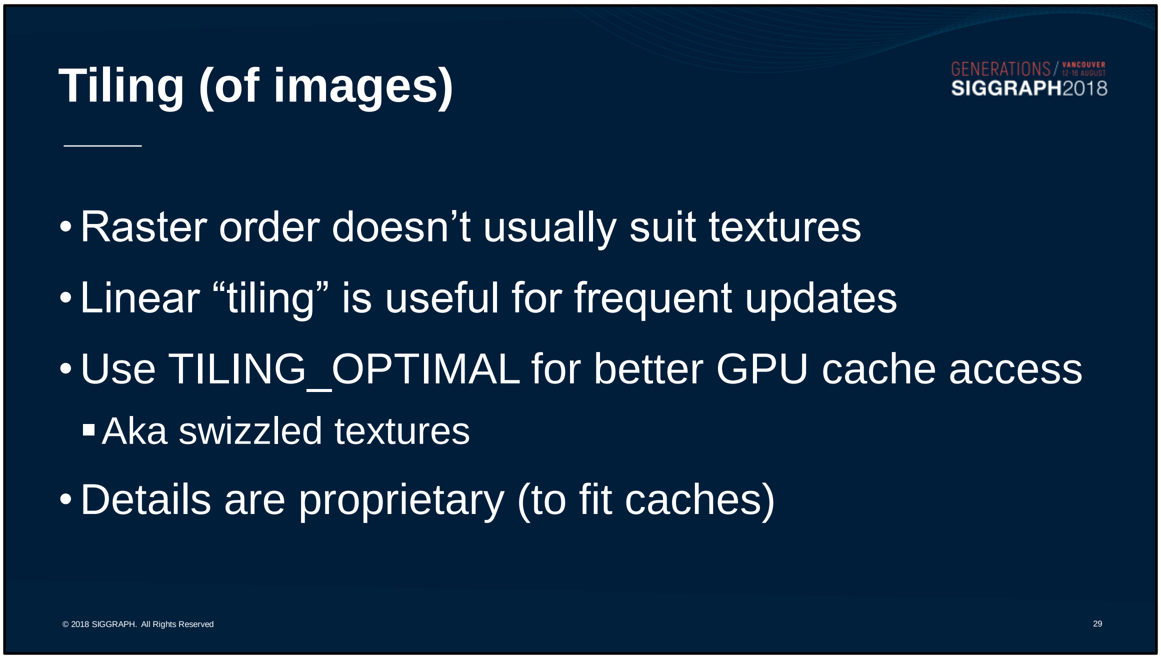
Tiling (of images)

GENERATIONS / VANCOUVER 12-16 AUGUST
SIGGRAPH2018

- Raster order doesn't usually suit textures

27 lines
10 tiles

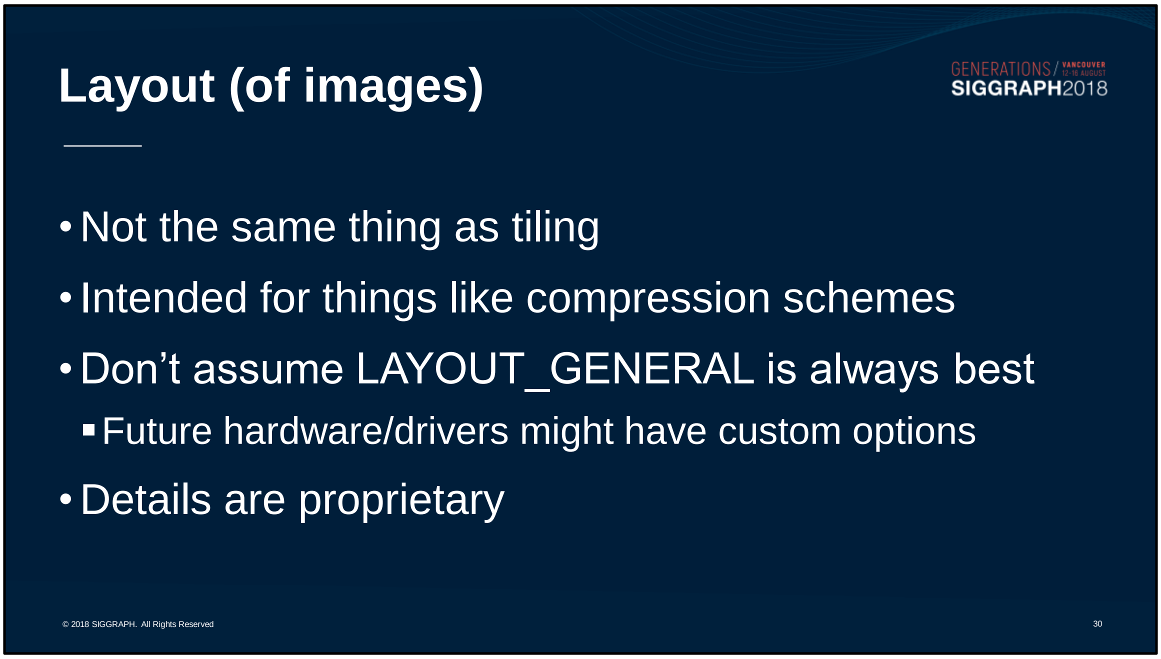
Storing 2D tiles of the image consecutively in memory gives a better chance that there will be good locality of reference when accessing the texture.
Tiling or linear is under user control in Vulkan.



Tiling (of images)

- Raster order doesn't usually suit textures
- Linear "tiling" is useful for frequent updates
- Use TILING_OPTIMAL for better GPU cache access
  - Aka swizzled textures
- Details are proprietary (to fit caches)

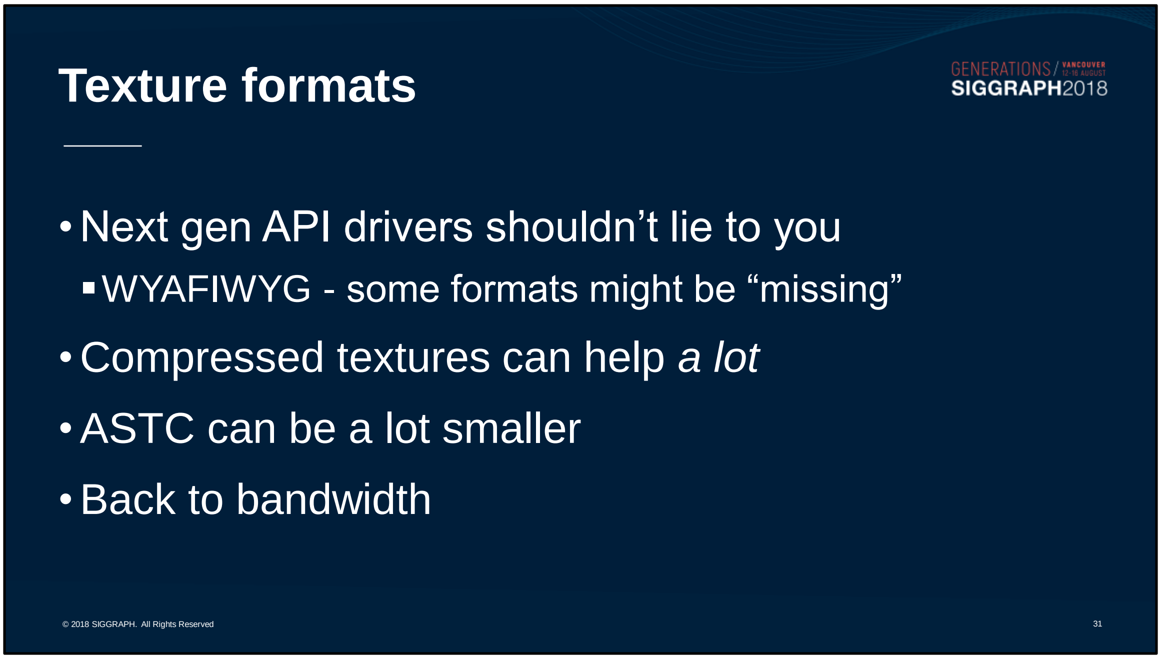
You need a linear layout for CPU access, and if the GPU can texture from it, it might be useful to use this despite the performance hit if the texture is being update every frame.
If you're not doing this, TILING_OPTIMAL is usually more friendly to the GPU.
The tiling algorithm is proprietary to each GPU, in part because it depends on cache sizes and internal details of the memory controller.



Layout (of images)

- Not the same thing as tiling
- Intended for things like compression schemes
- Don’t assume LAYOUT_GENERAL is always best
  - Future hardware/drivers might have custom options
- Details are proprietary

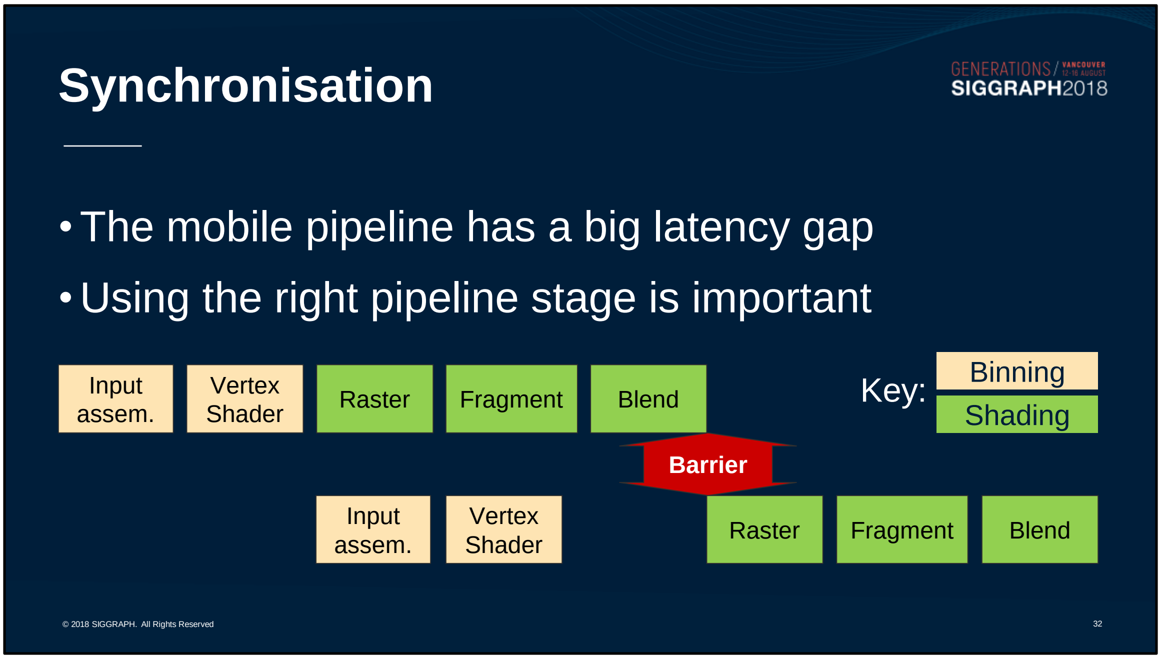
Vulkan has the concept of “layout”, which isn’t the same thing as tiling, even though it sounds like it.
It’s intended to be more specific about how images will be used in order to let hardware use techniques like framebuffer compression and hierarchical Z.
It’s best to use the most specific layout that’s appropriate, even if you don’t see a difference on your test hardware – new hardware or drivers might start using it.



Texture formats

- Next gen API drivers shouldn't lie to you
  - WYAFIWYG - some formats might be "missing"
- Compressed textures can help *a lot*
- ASTC can be a lot smaller
- Back to bandwidth

Picking texture format is important to trade bandwidth and quality.
In older APIs you may appear to have some formats supported which are actually just emulated. Vulkan has the philosophy of not doing that, so if you seem to have some missing formats, they may never have been there!
Pick compressed textures (especially newer formats) as much as you can – they can save a lot of bandwidth, but check what the hardware supports.



Synchronisation

GENERATIONS / VANCOUVER 12-16 AUGUST
SIGGRAPH2018

- The mobile pipeline has a big latency gap
- Using the right pipeline stage is important

Input assem. | Vertex Shader | Raster | Fragment | Blend

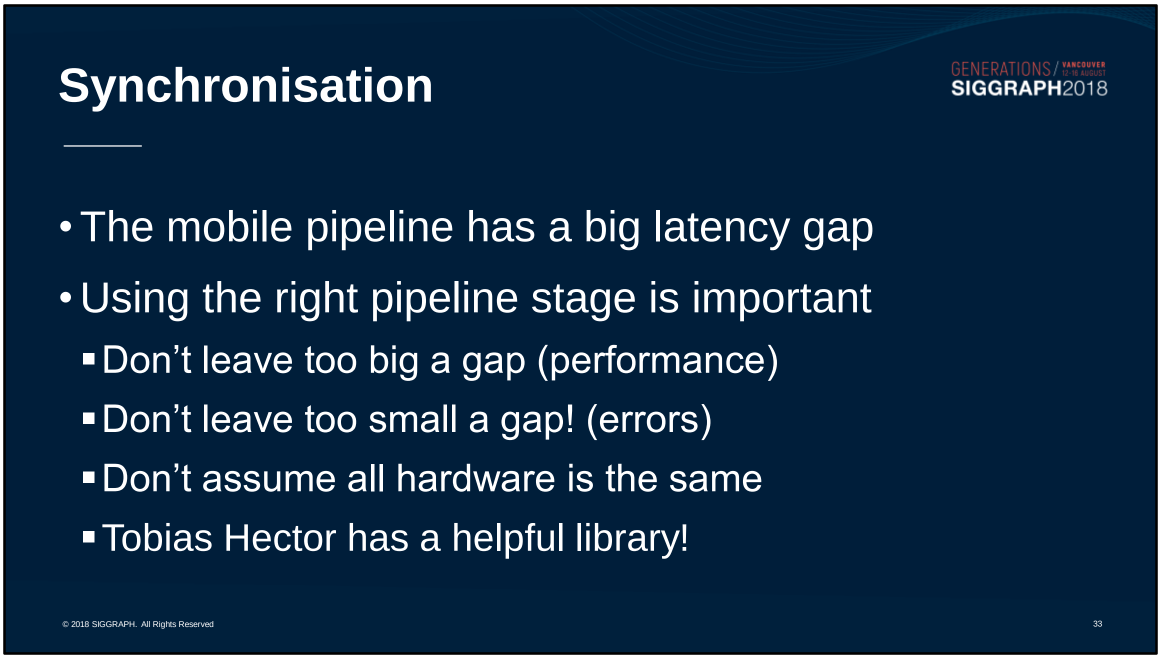
Barrier

Input assem. | Vertex Shader | Raster | Fragment | Blend

Key:
Binning
Shading

The conceptual graphics pipeline – each triangle walking through each stage of the pipeline – is interpreted differently by a tiler.
On a tiled GPU, all the primitives in the scene go through input assembly and vertex shading as part of the binning pass before any rasterization happens.
This means there’s a big difference between “start to render” and “start to rasterise” on a tiled GPU.
In Vulkan synchronisation is up to the developer – resources aren’t automatically tracked in the driver like in OpenGL, because there’s a cost to doing this unnecessarily.
That puts the onus on the developer to synchronise the right pipeline stages.
For example, consider rendering to an environment map.
You need synchronisation between the task of rendering to the environment map and then using it as a texture texture (you need to ensure that the rasterization has finished before you can use the image).
If you block the start of the graphics pipeline (in Vulkan speak, “TOP_OF_PIPE_BIT”) until the texture has been rendered, the binning pass is kept waiting, which can take a long time – and it probably doesn’t need to be.
Just blocking the rasterising of the second render target (“FRAGMENT_SHADER_BIT”) minimises the wasted time, since only the shading and texturing actually depends on

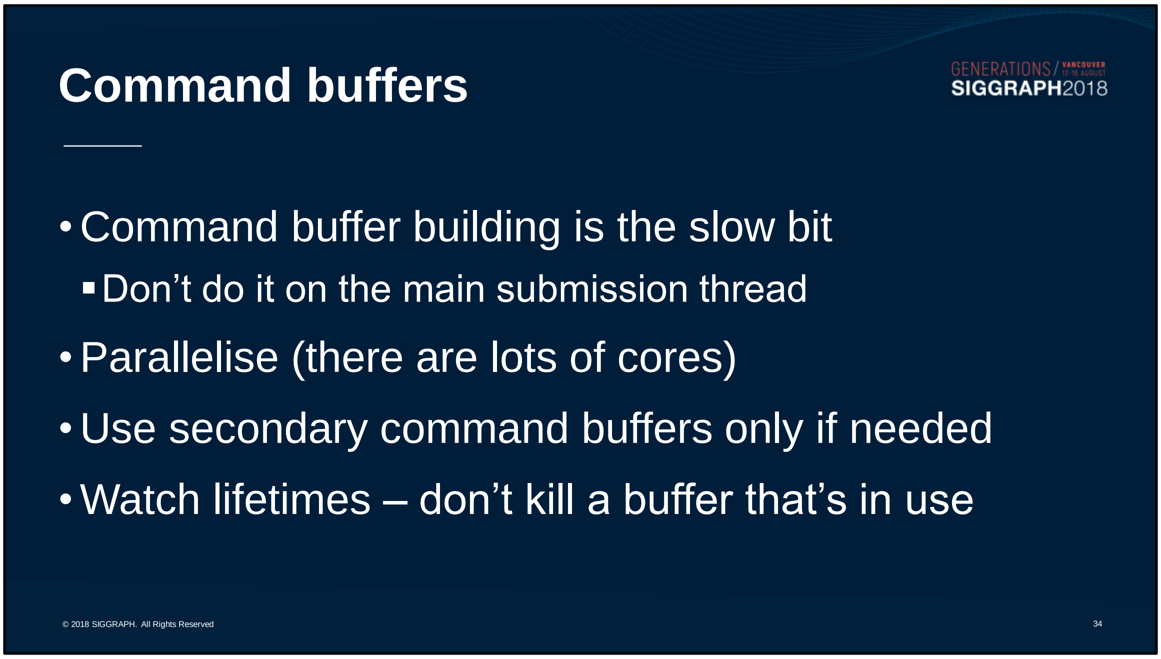
the environment map contents.



Synchronisation

- The mobile pipeline has a big latency gap
- Using the right pipeline stage is important
  - Don’t leave too big a gap (performance)
  - Don’t leave too small a gap! (errors)
  - Don’t assume all hardware is the same
  - Tobias Hector has a helpful library!

Check what the pipeline stages are doing, and don’t leave a bigger gap than you need to, or you’ll introduce bubbles that may be much more visible than on a desktop GPU. But remember it’s also up to you to wait as much as you need, so don’t fail to put the synchronisation in there.
This is worth stating because hardware is so different – just because it seems to render correctly on your test system doesn’t mean it will everywhere.
There are lots of ways to hook up synchronisation in Vulkan, and only a subset of them make sense; Tobias wrote a useful library to simplify synchronisation.



Command buffers

- Command buffer building is the slow bit
  - Don’t do it on the main submission thread
- Parallelise (there are lots of cores)
- Use secondary command buffers only if needed
- Watch lifetimes – don’t kill a buffer that’s in use

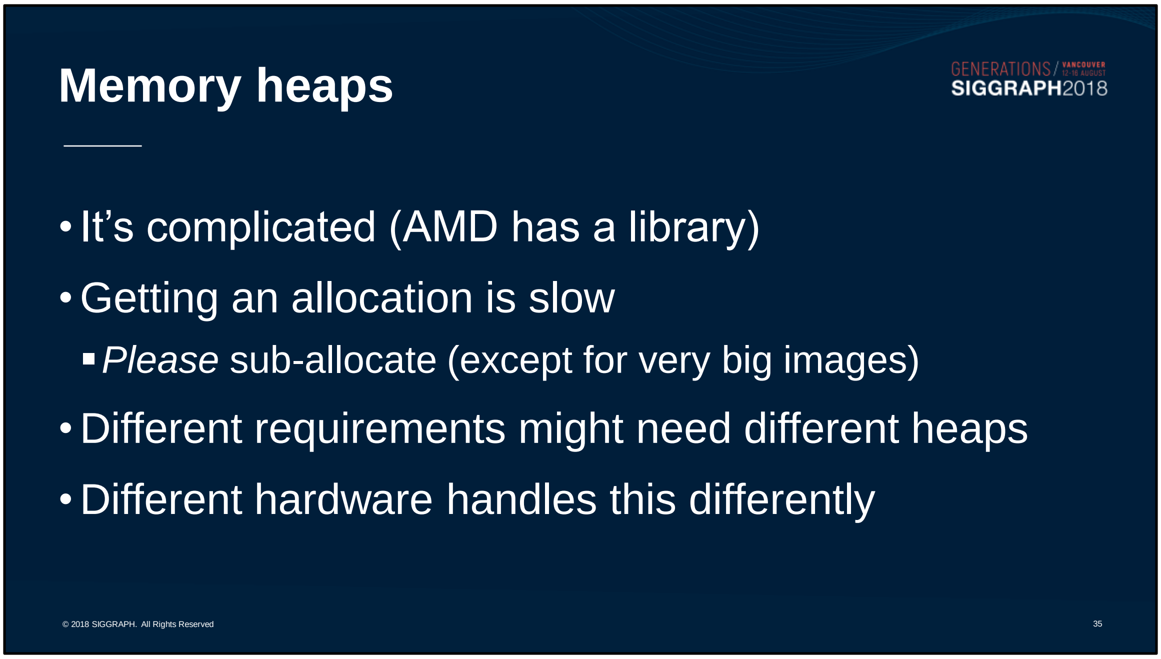
Command buffers are intended to represent blocks of work ready to run on the GPU – so any compilation, configuration or resource wrangling should happen while the command buffer is being built.

Vulkan is designed to avoid the single-threaded nature of GL, and the developer is expected to distribute work rather than having the driver do this implicitly.

If you can use the extra CPUs to build command buffers in the background, you can make better use of the CPU – but the main submission to a queue is synchronous, so it’s better to keep submission to a single thread.

Secondary command buffers let you split up work within a single render pass if you have a complex frame – but some architectures have an overhead in using them, so it’s better to keep parallelism at the render pass level if you can.

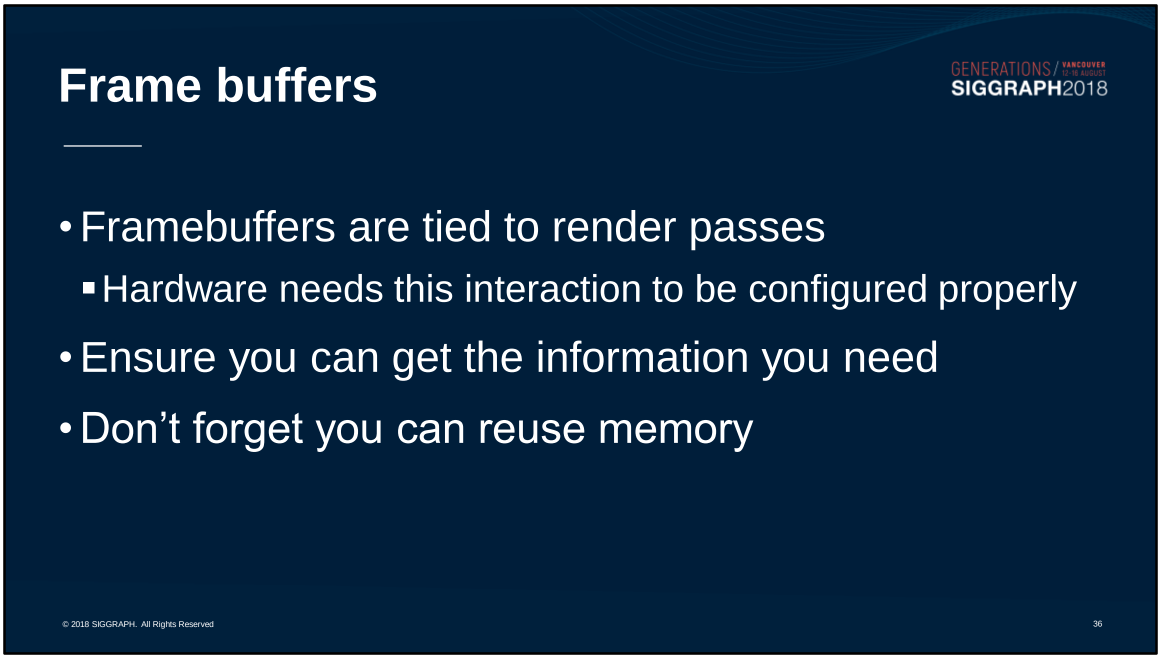
Do keep an eye on command buffer management – check memory usage, and don’t delete resources that are still in flight.



Memory heaps

- It’s complicated (AMD has a library)
- Getting an allocation is slow
  - *Please* sub-allocate (except for very big images)
- Different requirements might need different heaps
- Different hardware handles this differently

Different hardware has different kinds of memory available to it.
The Vulkan memory heap system allows you to get access to this memory, and it’s up to you to ensure you’re using memory with the features you need.
Doing this is a bit complicated, but AMD has made a helpful library available to simplify things.
Getting hold of a memory allocation is slow, so don’t do it for every tiny image: the intent is that you should be requesting large blocks of memory and sub-allocating out of it yourself.
Don’t assume that the heap settings on our development machine will be available everywhere.



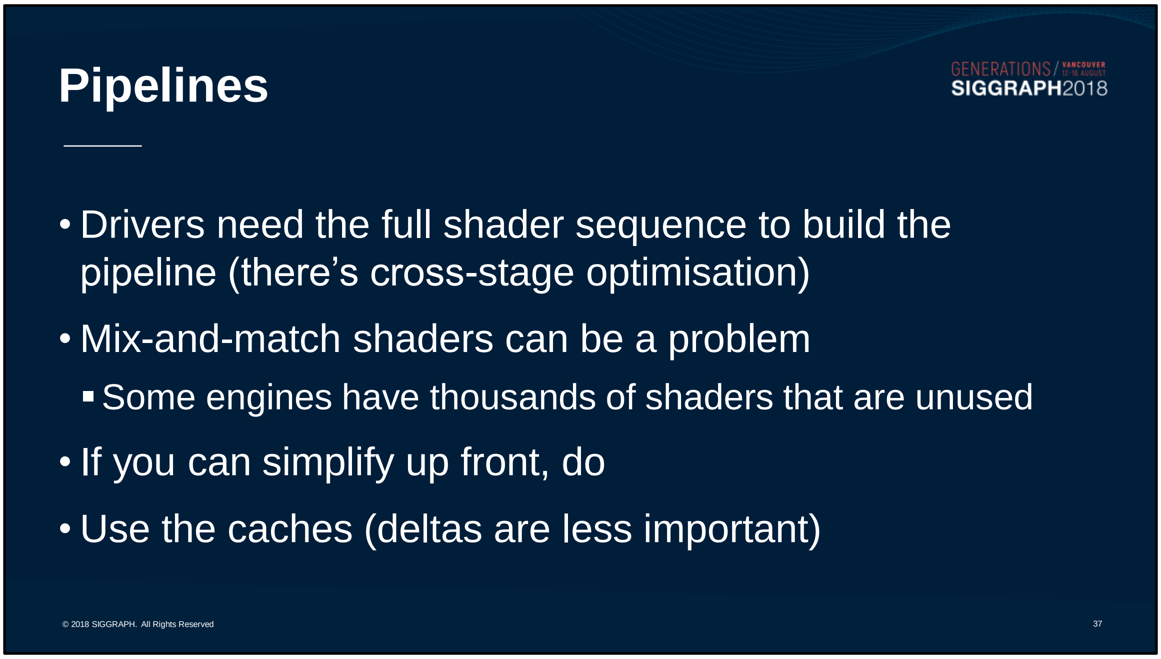
Frame buffers

- Framebuffers are tied to render passes
  - Hardware needs this interaction to be configured properly
- Ensure you can get the information you need
- Don’t forget you can reuse memory

Framebuffers and Render Passes are tied together in Vulkan – the render pass needs to know about the framebuffer in order to be configured correctly.

This can be awkward for some software that doesn’t have the render target to hand at the point of building the command buffer; if you’re writing a new engine, it’s worth planning ahead.

Remember that with proper use of synchronisation, you can keep track of exactly when memory is in use, meaning you can alias the same memory for multiple tasks.



Pipelines

- Drivers need the full shader sequence to build the pipeline (there’s cross-stage optimisation)
- Mix-and-match shaders can be a problem
  - Some engines have thousands of shaders that are unused
- If you can simplify up front, do
- Use the caches (deltas are less important)

GENERATIONS / VANCOUVER 12-16 AUGUST
SIGGRAPH2018

Pipelines determine what shaders are being run on a primitive and the data being passed between stages – which means the driver needs to know the whole pipeline to compiler shaders for the GPU properly.

This is fine if you know about it, but some engines have a lot of shaders which can be mixed and matched by the developer, and it can be hard for these to keep the pipeline overhead under control.

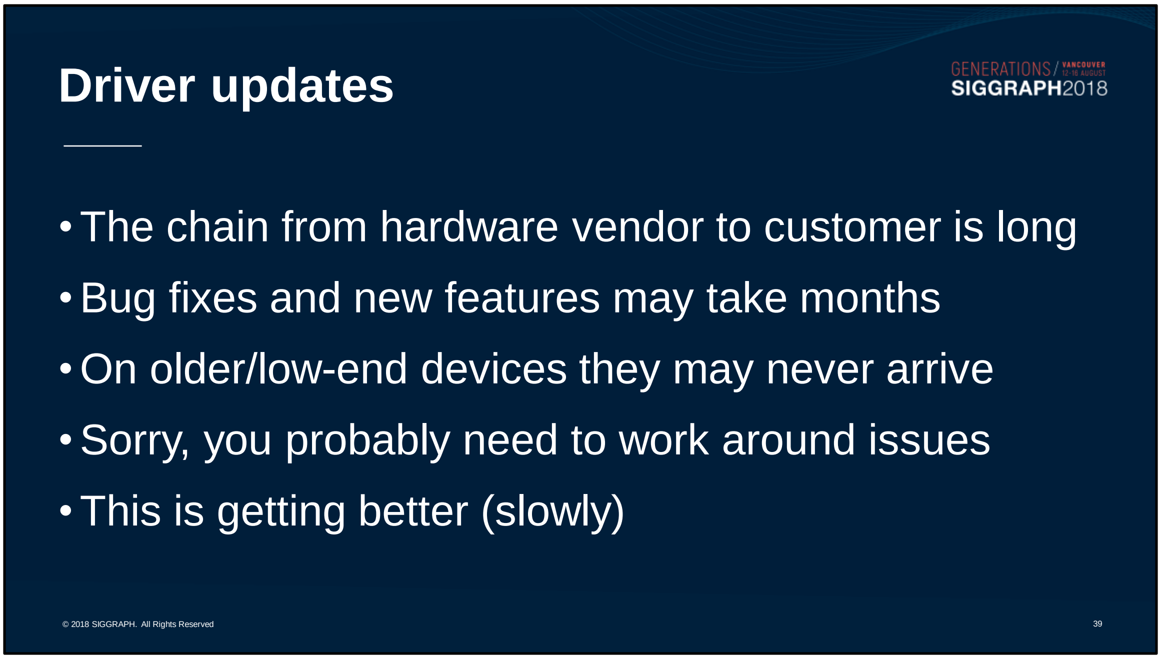
If you know exactly what shaders are going to be run, build the pipeline up front and warm the cache. The pipeline cache is your friend. Pipeline deltas tend to be less useful, although they can still help some hardware.

GENERATIONS / VANCOUVER 12-16 AUGUST
SIGGRAPH2018

Problems

Opportunities!

So what issues are you likely to have to deal with in moving to mobile?

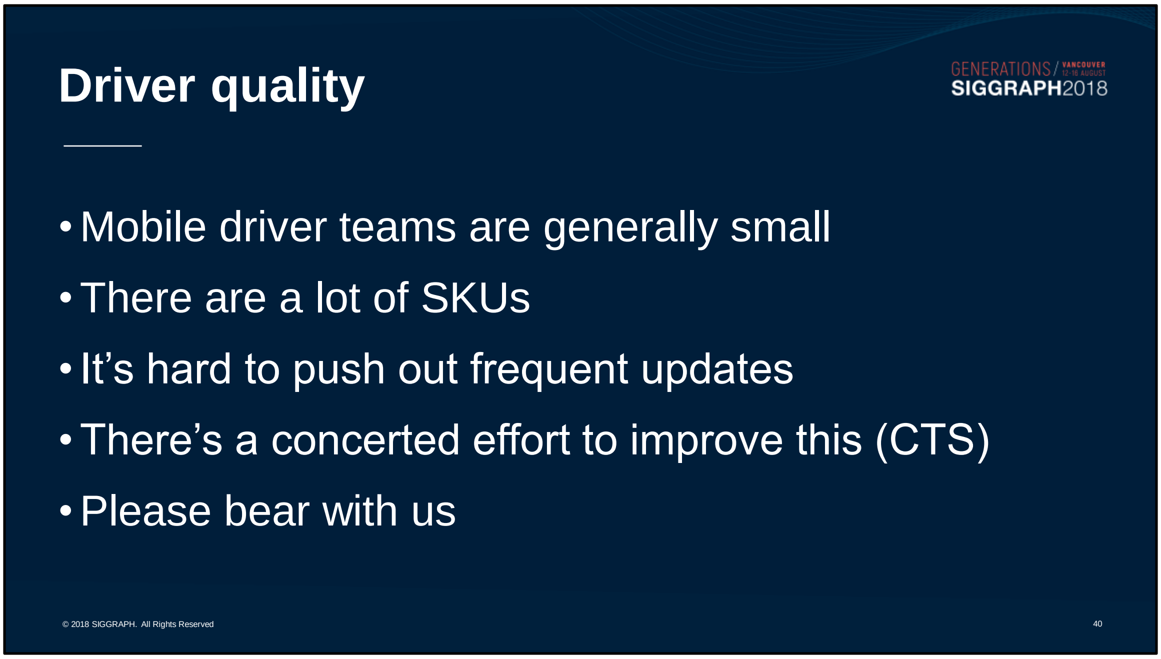


Driver updates

GENERATIONS / VANCOUVER 12-16 AUGUST
SIGGRAPH2018

- The chain from hardware vendor to customer is long
- Bug fixes and new features may take months
- On older/low-end devices they may never arrive
- Sorry, you probably need to work around issues
- This is getting better (slowly)

On desktop you can ask a user to update their drivers, and it’s relatively easy to do so.
On mobile, drivers need certification at multiple stages and it’s very expensive and slow to do so.
This means a bug fix might take a long time to get into users’ hands, and for anything but recent high-end hardware, it may never arrive.
Efforts are being made to improve this, but if you see a driver bug, you probably need to be prepared to work around it.



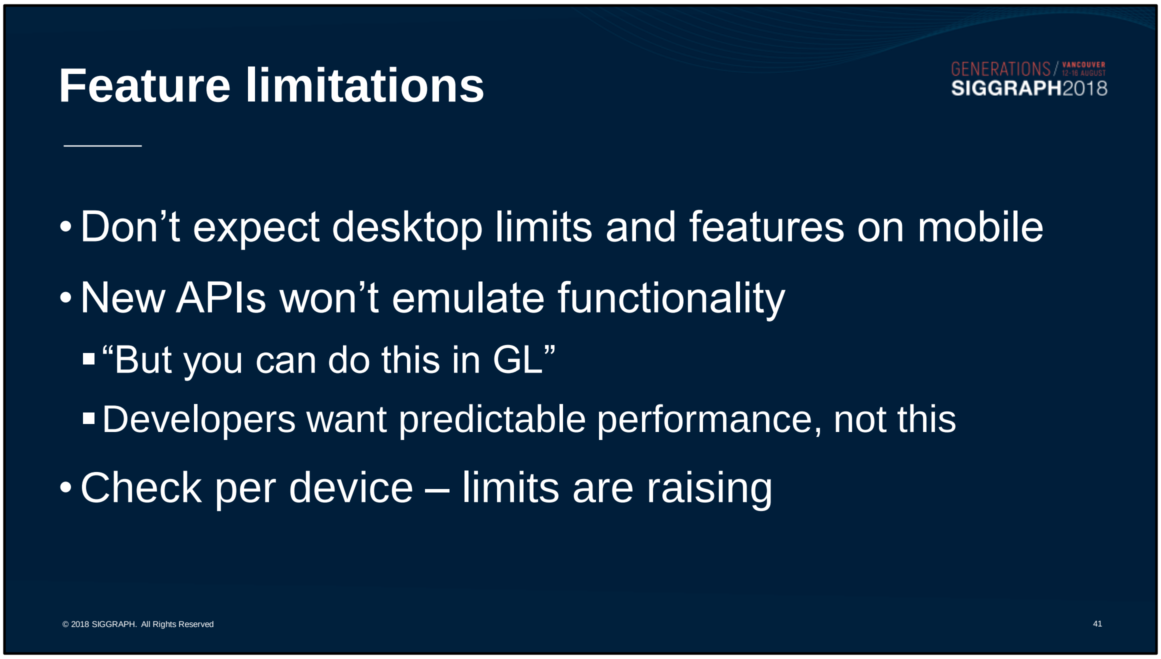
Driver quality

GENERATIONS / VANCOUVER 12-16 AUGUST
SIGGRAPH2018

- Mobile driver teams are generally small
- There are a lot of SKUs
- It’s hard to push out frequent updates
- There’s a concerted effort to improve this (CTS)
- Please bear with us

Speaking of driver bugs, the teams working on Vulkan drivers in mobile are generally not as big as in desktop, and there are a lot of embedded variants of GPUs to test. This means the quality of the drivers has been lagging a bit. Add the delay in shipping a fix and seeing whether it works or introduces other problems, and it’s not surprising that driver quality is an issue.

As conformance tests improve, drivers are becoming better tested, and the hardware vendors in Khronos are very much aware that this is a high priority.



Feature limitations

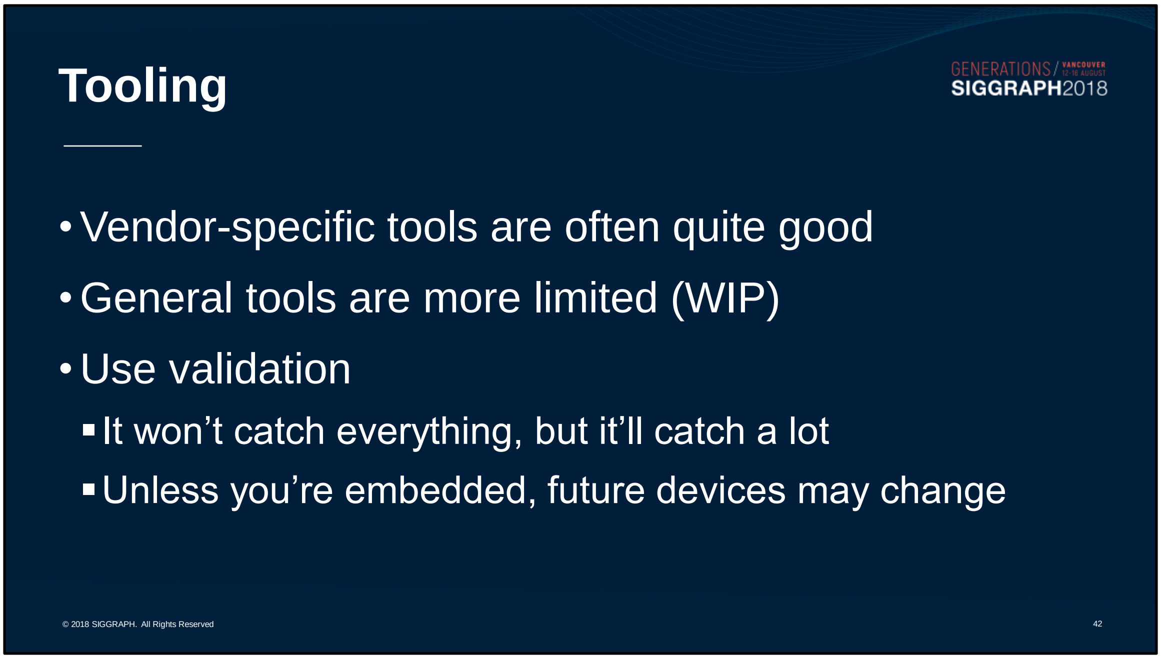
GENERATIONS / VANCOUVER 12-16 AUGUST
SIGGRAPH2018

- Don’t expect desktop limits and features on mobile
- New APIs won’t emulate functionality
  - “But you can do this in GL”
  - Developers want predictable performance, not this
- Check per device – limits are raising

Mobile devices have less area for dedicated silicon, and the architecture can make some techniques harder to implement – so features and resource limits that don’t seem to be useful in mobile may not have earned their place on the silicon. You’ll have to check what’s available on the hardware you care about.

This also applies to things which seem to be supported in other APIs, where they may have been emulated; however this means any feature you do have should be performant.

Do check the device you’re running on rather than assuming the Vulkan minimum – as newer generations of hardware and drivers are rolled out, higher limits and more features are becoming available, especially on flagship devices.



Tooling

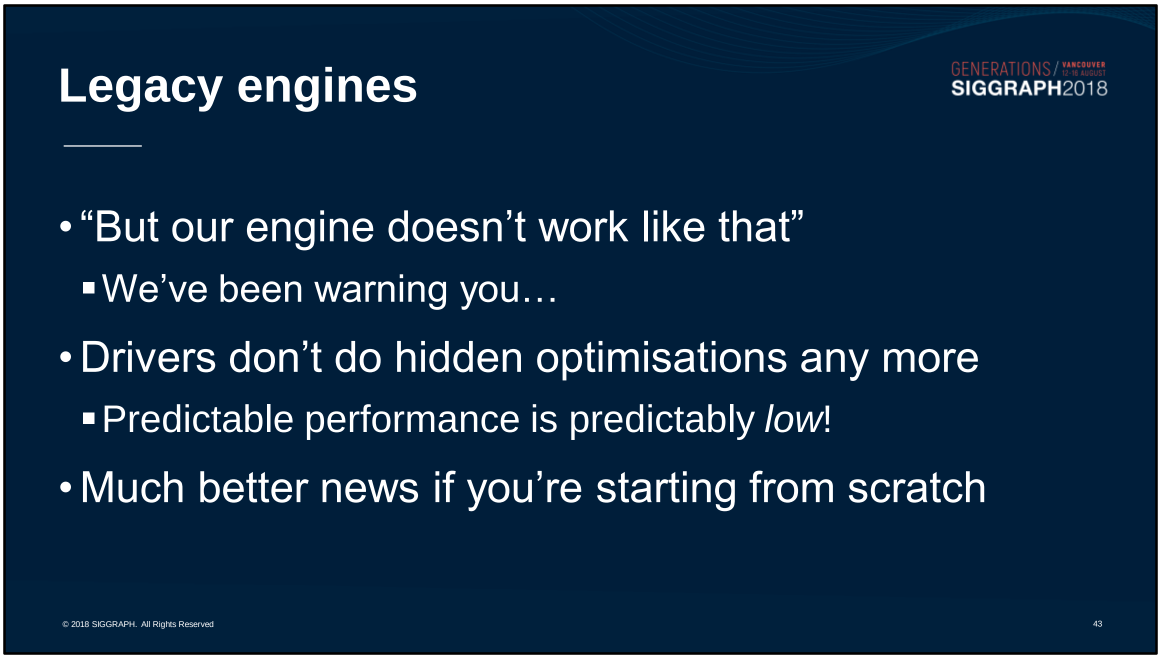
GENERATIONS / VANCOUVER 12-16 AUGUST
SIGGRAPH2018

- Vendor-specific tools are often quite good
- General tools are more limited (WIP)
- Use validation
    - It won’t catch everything, but it’ll catch a lot
    - Unless you’re embedded, future devices may change

The situation in mobile has historically been that each hardware vendor provides its own tools – and because the details of performance monitoring is often tied closely to the intellectual property of the rendering system, it’s harder to make these portable. Questions like “how long did this triangle take to render?” can be complicated on a tiler.

More general tools are coming.

Bear in mind that the error reporting in Vulkan is deliberately minimal, because adding debug reduces performance. The validation layer can be a useful way to ensure that the API usage of your application is correct, even if it seems to be working on your current hardware. Hardware vendors treat the validation rules as API guarantees, and the assumption that if something works now it must always work would restrict the development of new, faster GPUs.



Legacy engines

- “But our engine doesn’t work like that”
  - We’ve been warning you…
- Drivers don’t do hidden optimisations any more
  - Predictable performance is predictably *low*!
- Much better news if you’re starting from scratch

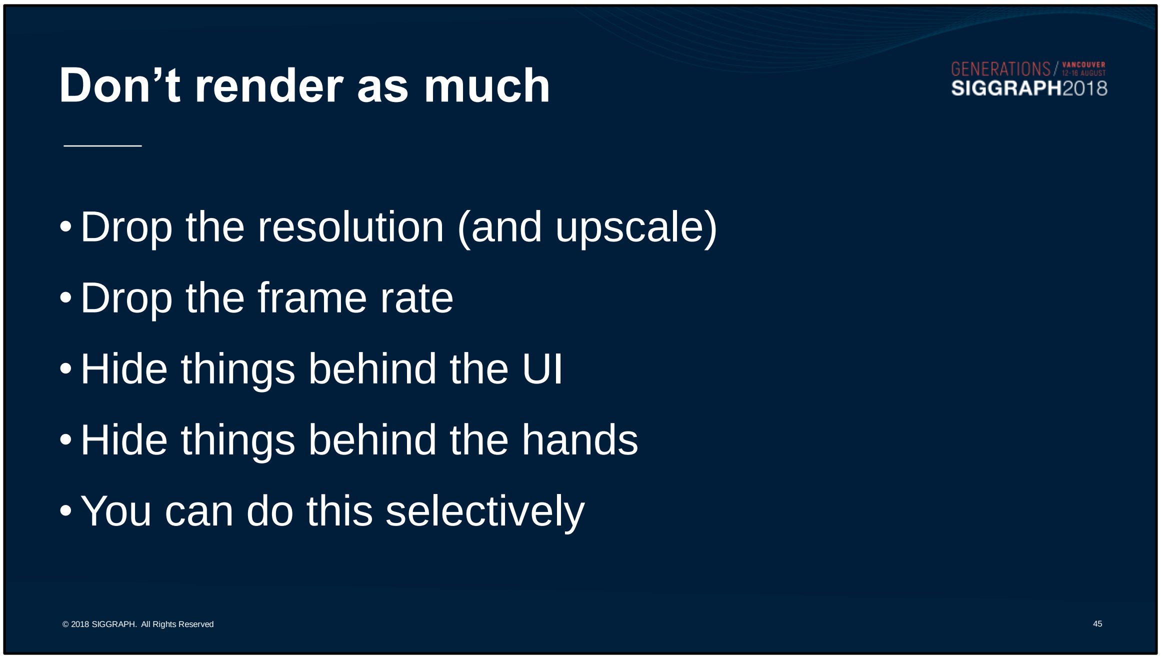
Developers starting a new project with a new API have an advantage; the new graphics APIs present more of a problem to large installed bases of software.
We’ve been saying for a while that structuring the rendering flow so you have the information you need in the right order is much easier to do at the beginning than to retrofit.
On top of that, engines can be written relying on the driver optimisations which older APIs performed to try to get good performance. Now the software developer has control – which makes performance predictable and ensures that the correct optimisations are being applied, but does mean that it’s up to the developer to make sure those optimisations are there.

GENERATIONS / VANCOUVER 12-16 AUGUST
SIGGRAPH2018

Design for the target

Inconsolable: *adj., won’t run on PlayStation, Xbox or Switch*

There are a few things you can do in game design to help.

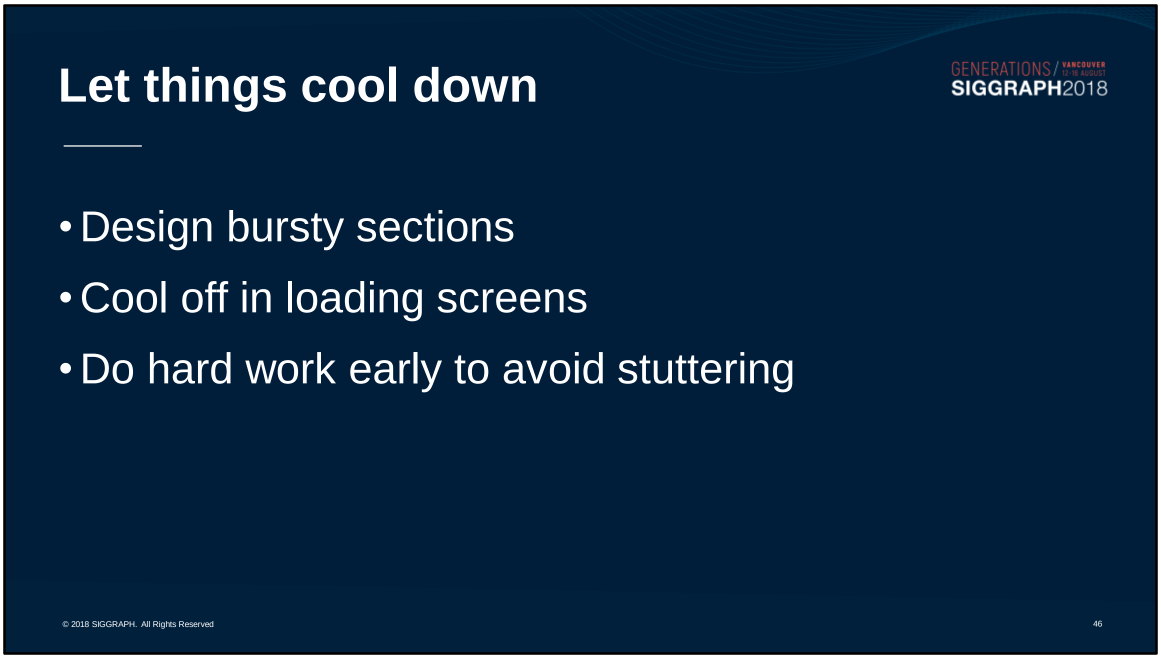


Don’t render as much

GENERATIONS / VANCOUVER 12-16 AUGUST
SIGGRAPH2018

- Drop the resolution (and upscale)
- Drop the frame rate
- Hide things behind the UI
- Hide things behind the hands
- You can do this selectively

The first thing to consider doing is restricting your workload. With a very high DPI screen, it is rarely necessary to render at full resolution; dropping to a fraction of the resolution and scaling the image up can be much more cost-effective.
The frame rate often doesn’t need to be maximised either, depending on the application. You might drop the refresh rate for menus or loading screens, or for calmer sections of the game, to let the device cool.
Don’t waste time rendering content that the UI will cover – consider rendering the UI first with a suitable Z so that the actual game geometry only appears where needed.
The same applies to the interface scheme – if your game relies on the user having thumbs covering on-screen paddles, think about not rendering areas that you believe will be covered by them.

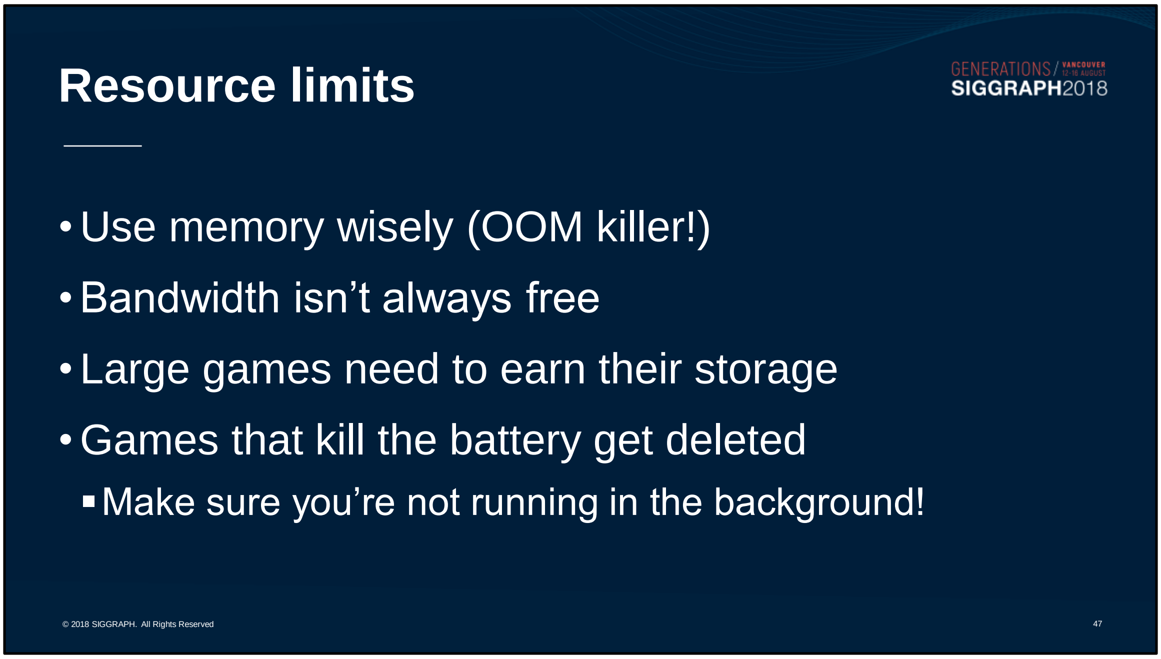


Let things cool down

- Design bursty sections
- Cool off in loading screens
- Do hard work early to avoid stuttering

If you do have high load elements of a game, see whether you can design in sections where the device can cool down – either less active sections or loading screens. If you do have long sequences of heavy rendering, make sure you can sustain the rendering load and degrade gracefully.

To avoid stuttering, try to prepare content as early as possible – get pipeline caches warmed outside the critical path, for example.



Resource limits

- Use memory wisely (OOM killer!)
- Bandwidth isn't always free
- Large games need to earn their storage
- Games that kill the battery get deleted
    - Make sure you're not running in the background!

Mobile devices are also limited in resources, obviously. A typical modern phone may have 4GB of RAM, but it's unlikely all to be available to you – and when a mobile app runs out of space, it doesn't page to disk, it gets killed. Stay within safe limits, and look at paging and level-of-detail to degrade on devices with less memory.

While it's less of a problem than it used to be, bear in mind that some phone contracts are limited in the amount of data available. If your game requires a gigabyte of transfer every time it starts up, people will notice.

But it is worth loading content incrementally. If your game takes up several GB of limited on-device storage before the user sees any benefit, it's going to be a candidate for deletion when the player next wants to take some more selfies. They may be much more willing to keep it on the phone once they've got more addicted.

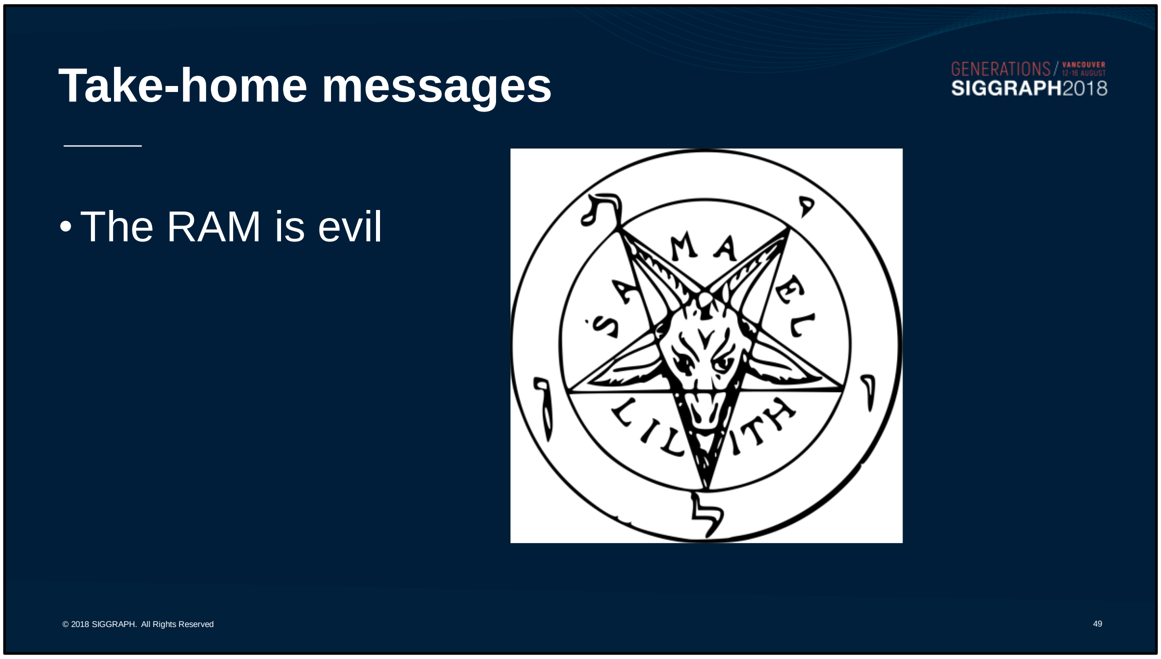
On a similar note, make sure you're running right – some games seem to run in the background, using up battery, and they'll be the first to go when the user cleans out their device even if the actual impact is small.

GENERATIONS / VANCOUVER 12-16 AUGUST
SIGGRAPH2018

Summary

Wake up now

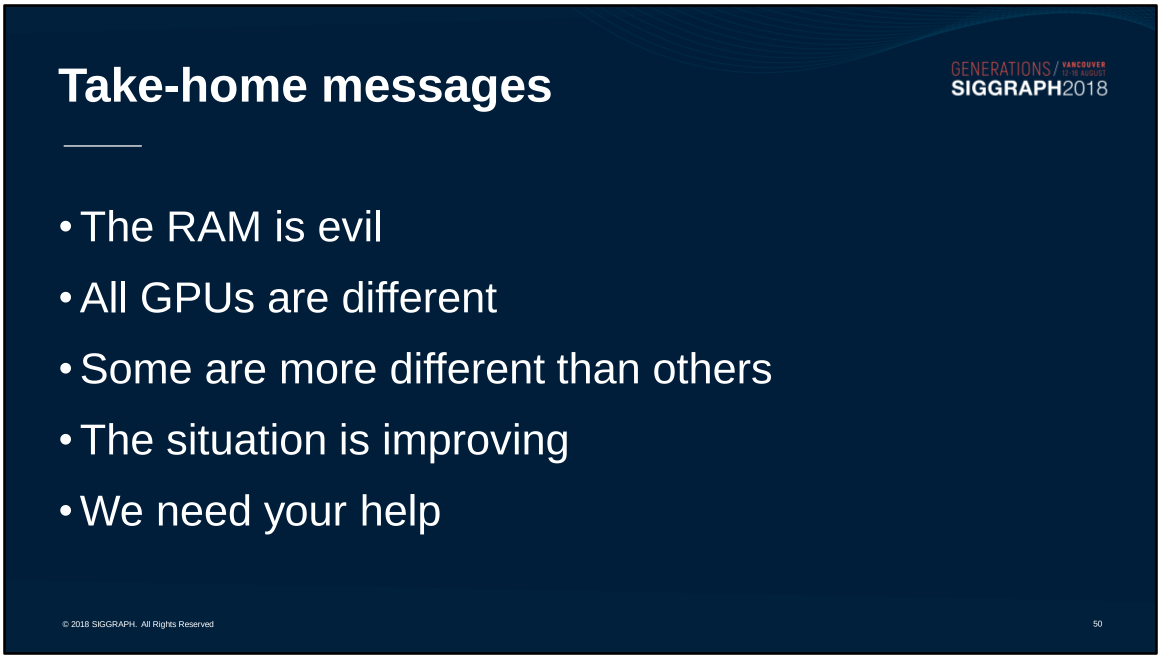
So, finally…


GENERATIONS / VANCOUVER 12-16 AUGUST
SIGGRAPH2018
SAMAEL
LILITH

Take-home messages

- The RAM is evil

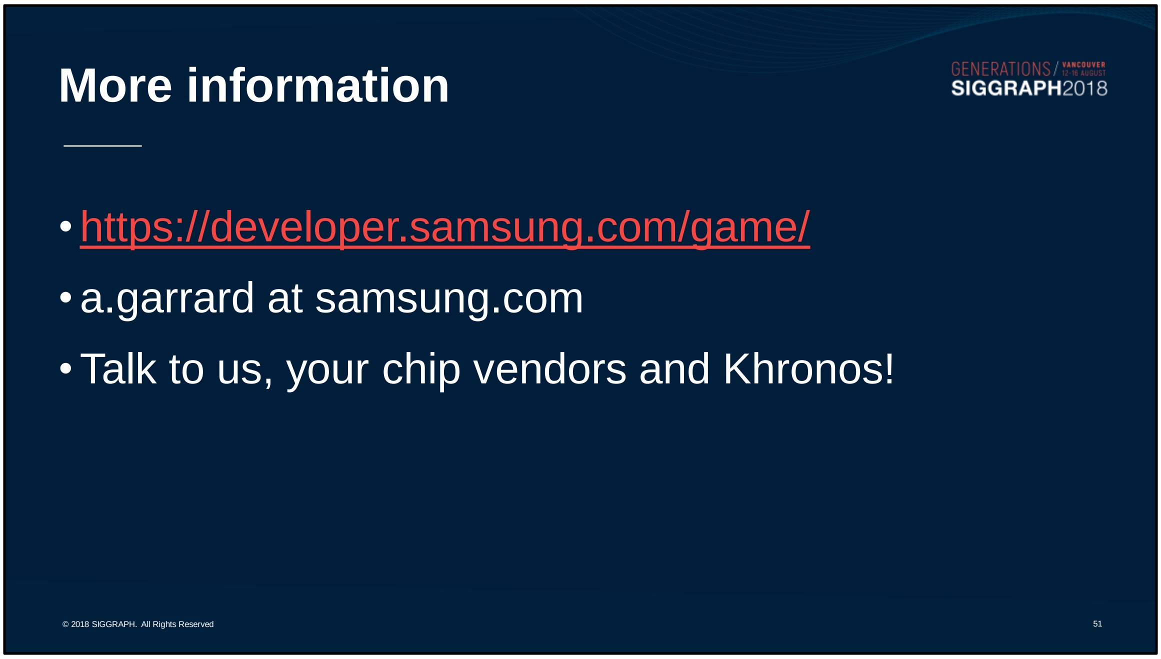
The RAM is a necessary evil. (Apologies to Satanists for the pun.) Don’t use it more than you have to, and prepare for things to get hot when you do.



Take-home messages

- The RAM is evil
- All GPUs are different
- Some are more different than others
- The situation is improving
- We need your help

There are lots of ways to design a GPU, and these are more visible in the modern APIs. Just because something works on your development machine, don’t assume it’s portable.
This is especially true in mobile, where the differences from a desktop GPU can be quite significant.
Driver quality, tools and the capabilities of mobile devices are all improving; please don’t get put off forever by a bad experience.
But to get good mobile performance, your software needs to make the best of the hardware.



More information

- https://developer.samsung.com/game/
- a.garrard at samsung.com
- Talk to us, your chip vendors and Khronos!