Get the most from Vulkan in Unity with practical examples from Infinite Dreams

ARM

Roberto Lopez Mendez, Senior Software Engineer

Marek Wyszyński, VP & Co-Founder

Mikko Strandborg, Vulkan Lead

GDC 2017

Agenda

- The benefits of the Vulkan graphics API (Roberto)
- Sky Force Reloaded with Vulkan and Unity (Marek)
- Vulkan in Unity - under the hood (Mikko)

The benefits of the Vulkan graphics API

Multi-threading / multicore efficiency

- Multi-threading responsibility moved to application level
  - The application has better visibility

- Efficient utilization of multiprocessor architecture
  - Spread work out faster to multiple cores. Lower CPU load and energy consumption
  - Able to schedule and migrate tasks between ARM ® big.LITTLE™ cores according to the load

Multi-pass rendering

- Very performant in tiled GPUs such as ARM Mali GPUs
  - Each pixel in a sub-pass can access the result of the previous sub-pass
  - All data can be contained on the fast on-chip memory, saving bandwidth

- Example of use-cases:
  - Deferred rendering
  - Soft-particles
  - Tone-mapping

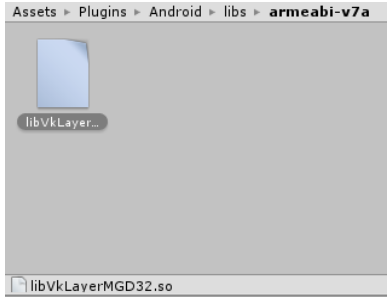
Vulkan benefits in Lofoten demo

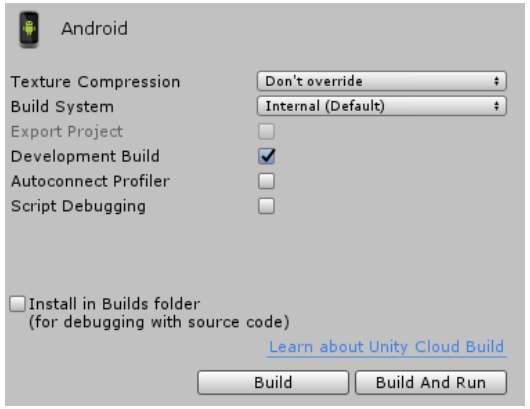
SCENE INFO

- ~ 100 lights with shadow maps
- 3M primitives (reduced to 500K with very efficient occlusion culling)
- 500 draw calls
- Sun light with cascade shadow map
- ~ 10 reflection probes
- FFT compute in ocean rendering
- Deferred shading using multi-pass
- 10x less load on CPU with multithreading

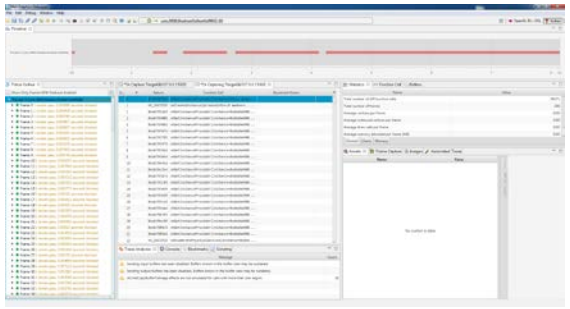
Mali Graphics Debugger

Vulkan

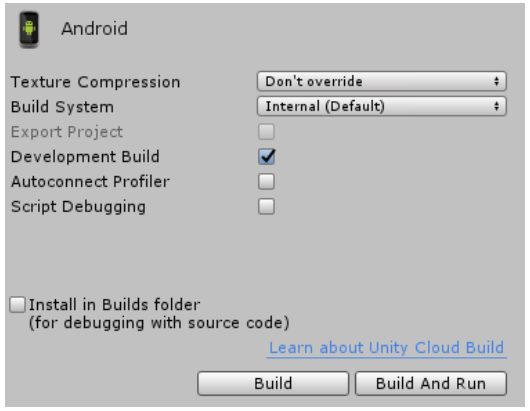




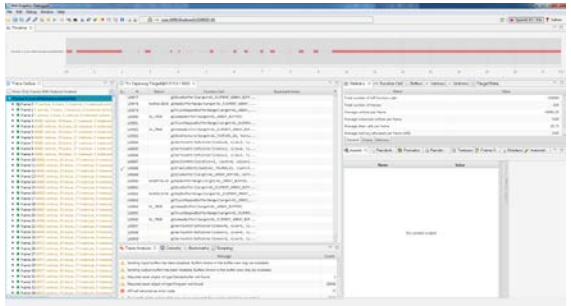

```
adb install -r MGD.apk
adb forward tcp:5002 tcp:5002

adb shell
setprop debug.vulkan.layers
VK_LAYER_ARM_MGD
```



OpenGL ES

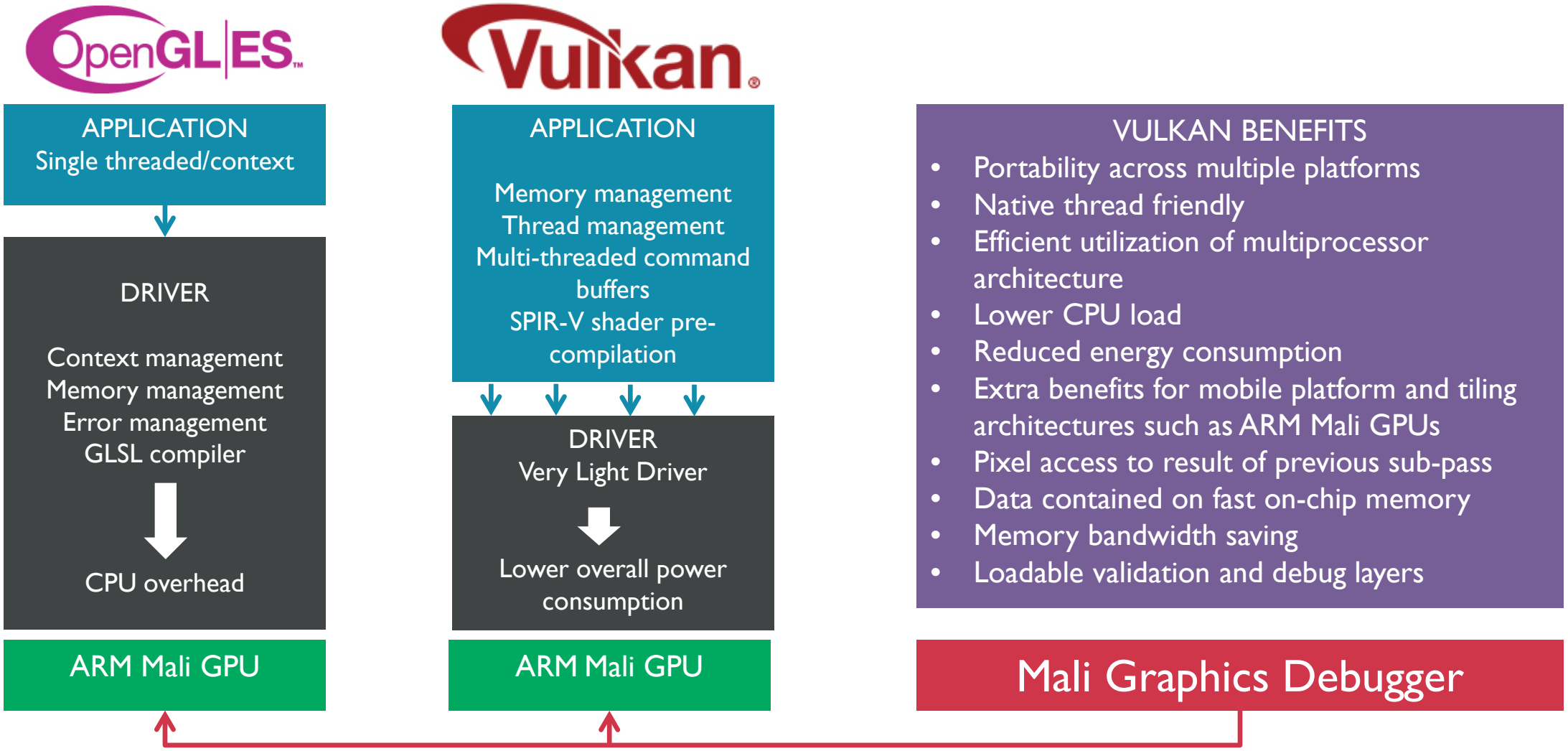



```
adb install -r MGD.apk
adb forward tcp:5002 tcp:5002
```



ARM

Wrap Up



Sky Force Reloaded with Vulkan and Unity

Marek WYSZYŃSKI
INFINITE DREAMS

WHAT IS SKY FORCE RELOADED?

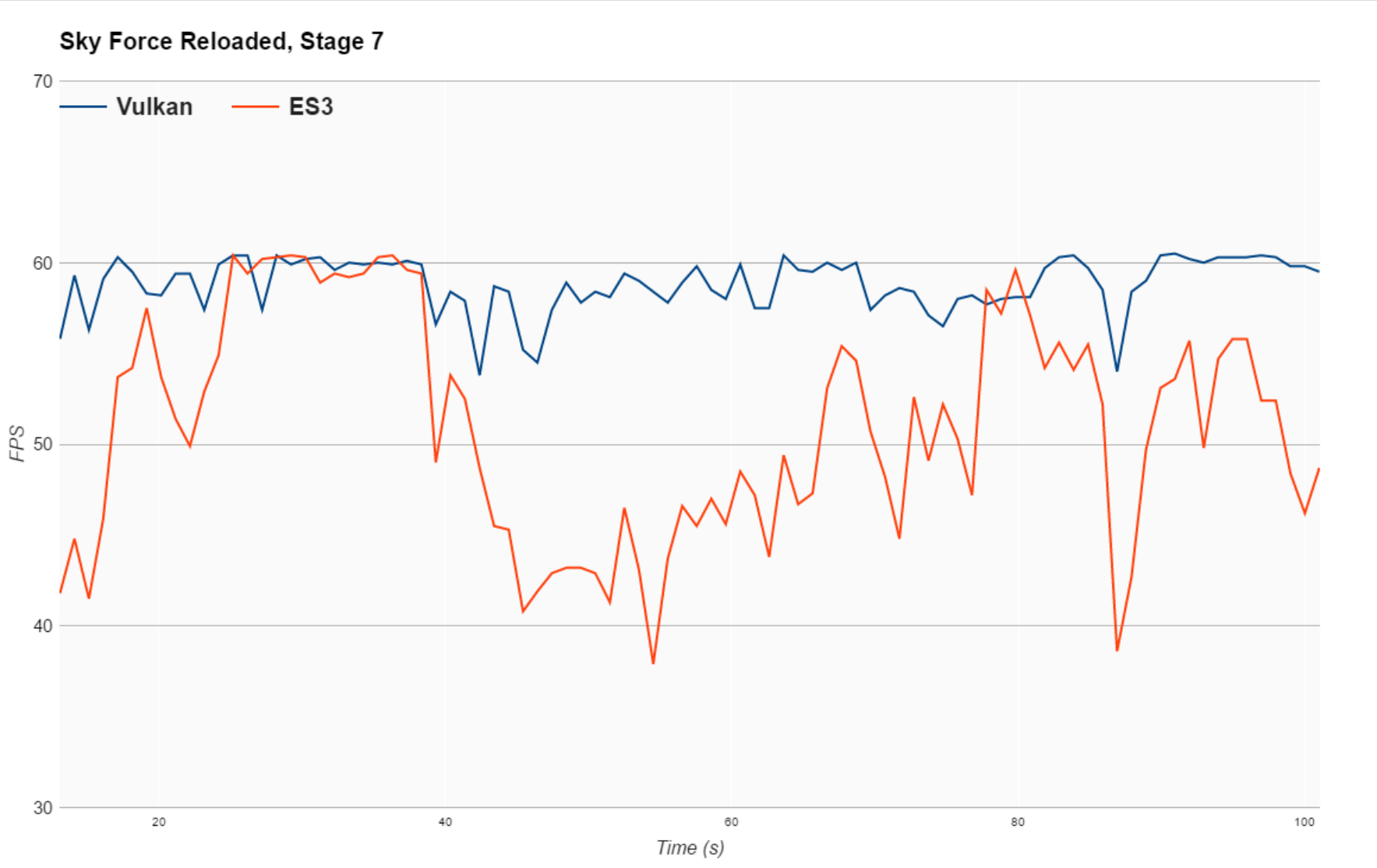
- modern **shoot’em up** experience
- intense **action**, very rich graphics
- pushing **GPU & CPU** to their limits

BIGGEST PERFORMANCE ISSUES IN SKY FORCE RELOADED

- **fill rate** is not a bottleneck
- up to **1000 draw calls** per frame
- CPU is spending a lot of time **preparing data** for GPU

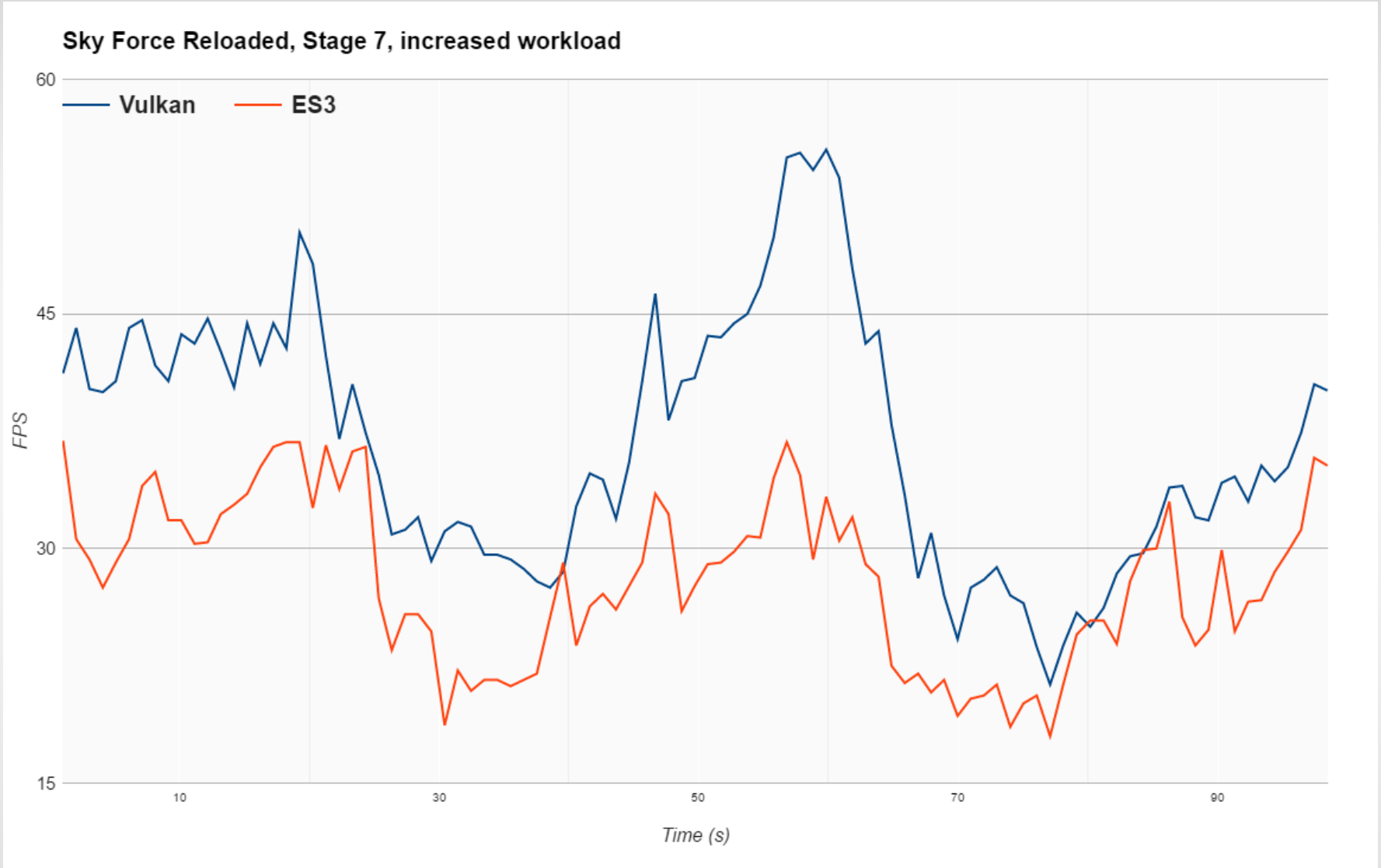
PERFORMANCE BENCHMARK

- **draw calls** are expensive
- OpenGL ES **driver** is not optimal
- perhaps **Vulkan** can help?
- Vulkan is supported by **Unity**!



Best case **21%** faster using Vulkan.

On average **15%** faster using Vulkan.



Best case **82%** faster using Vulkan.

On average **32%** faster using Vulkan.

MORE CONTENT

- **add** more particles, objects or animations
- keep the same FPS with **richer graphics**

POWER CONSUMPTION TEST

- power consumption is a **problem**
- players are **not happy**
- console-like quality games consume a lot of **power**
- can **Vulkan** help?

POWER CONSUMPTION TEST

- Vulkan consumed 10 to 12% **less power** in our game,
- majority of savings come from the **CPU**
- **extra minutes** of playtime with Vulkan!

CONCLUSIONS

- **great** to use “out of the box”
- **improves** your FPS
- adds some extra minutes of **playtime**
- you can add some **more graphics** and make your game look better

THANK YOU!

INFINITE DREAMS Inc.
www.idreams.pl
office@idreams.pl

Vulkan in Unity – under the hood

Mikko Strandborg – Vulkan Lead, Unity

@m_strandborg

Talk outline

- A quick intro on how Unity renders things in general, and in Vulkan
- Optimizations and tricks we do to extract maximum performance
- Going multicore

Overview of Unity rendering abstraction

- Abstraction API is mostly from DX9 / OpenGL ES 2.0 era ☹
  - Mostly because we still have to support those APIs for a good while
  - Improvements incoming!
- Worst-case simplified rendering sequence (no instancing, no batching):
  - "Hey, set a new shader program (here), with all the parameters it's going to need (here) and a serialized buffer containing the values for all those parameters (here)"
  - Update the world matrix
  - "Draw me N vertices using these vertex buffers and this index buffer with offsets X, Y and Z etc."
  - "Hey, use the shader program you already have bound, but here's a bunch of new parameters and their values, override the old ones with these, leave the rest of them intact"
  - Update the world matrix
  - Draw again

Hey, none of this maps to Vulkan!

- Extra smarts needed
- Problems:
  - What is the expected lifetime / possible reuse of constant buffers?
  - Partial updates effectively mean creating a copy of the constant buffer
  - Because of the matrix updates being separate, we'll only know the real final parameters at draw time.
  - Pro tip #1: GPUs really really hate switching between buffer bindings. Changing offsets is almost free.
- A naïve implementation would be very slow.

Some building blocks

- VulkanResource base class
  - MarkUsedInCurrentFrame()
  - IsBusy()
- GPU fence at each Present, get last frame number completed by the GPU
- Delayed delete facility
  - Delay delete until IsBusy() == false
- Reuse all the things
  - Even if you're rolled your own allocators and memory managers.

Tooling considerations

- Vulkan validation layers are awesome. Use them.
  - Caveat: Object IDs generated by the layers are monotonically increasing, may hide your bugs.
- RenderDoc is da real MVP
  - Android remote support coming!
- Keep your main development cycle on the desktop
  - Build-deploy-test cycle is a lot shorter
  - Wider range of debugging and profiling tools available
- Don't forget to periodically test on target device as well!
  - The Vulkan implementations are different and have different characteristics
- Most of our optimizations we did help both desktop and mobile GPUs!

SPIR-V generation

- Our SPIR-V compilation pipeline:
  - [ShaderLab + HLSL] -> CgBatch -> [HLSL] -> D3DCompiler.dll -> [DX bytecode] -> HLSLcc -> [GLSL] -> glslang -> [SPIR-V] -> SMOL-V
- Glslang doesn't do automatic descriptor set / binding slot allocations -> we do it in HLSLcc
- Descriptor set / binding namespace is the whole shader program, not separate shader stages.
- Reflection data comes from glslang

Descriptor set objects

- In OpenGL and DX11 each shader resource is bound separately
  - layout(location=X) decoration in GLSL
- In Vulkan they are grouped into descriptor sets
  - layout(set = X, binding = Y) decoration in GLSL
  - A first-class citizen in Vulkan
  - Allocated from VkDescriptorPool
- Our approach:
  - In HLSLcc, put all constant buffers into descriptor set 1, everything else into descriptor set 0
  - Separate VkDescriptorPool for each shader program, no individual release of descriptor sets
  - Own reuse pool (VulkanResources!)
- Pro Tip #2: Descriptor set objects may get consumed at bind time.

Constant buffers

- Pro tip #3: Mobile GPUs don't do any magic on constant buffers! They're just pointers to main RAM.
- Pro tip #4: GPUs have caches for memory access, typically 64- or 32-byte cache lines
- A larger constant buffer for "rarely accessed" parameters is a bad idea.
  - Thrashes cache.

Constant buffers, or lack thereof

- Get rid of constant buffers completely!
  - Just the bare minimum of one tightly packed cbuffer per shader stage.
  - Tightly packed, contains no unused data.
  - Cache friendly: fragment shader only has the data it needs, no vertex uniforms polluting the cache.
- Write constant buffer data into HOST_VISIBLE scratch buffer
  - Rotate buffers when previous one fills up
- No memcmp's for checking reuse, always feed the data to the scratch buffer
- Render using the scratch buffer area directly (no GPU-mem copy on desktops)
- Use VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_DYNAMIC so can reuse descriptor set object
  - Remember: all constant buffers are in descriptor set #1

Descriptor set cache

- Realization: There is only ever a handful of different descriptor set objects!
- Cache all descriptor set objects!
  - Using VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_DYNAMIC means very little combinations per shader
  - Eliminates most vkUpdateDescriptorSet calls mid-frame
- Use dense_hash_map for caching
- Map key is pretty large: contains everything you'd need to build the VkDescriptorSet from scratch
- Tricks to speed up hashing:
  - Add data member in key struct that tells how many bytes to hash
    - Also speeds up key comparison!
  - Cache the hash value in the key struct

Memory management

- Scratch buffer is persistently mapped
  - Manual Flush/Invalidate when needed
  - Flush is a syscall, so only do it once right before job submission
- Use buddy allocator
  - Manages offsets into VkMemory
  - Used for allocating small textures and buffers
  - Shared between threads
  - Mainly used to avoid hitting maxMemoryAllocationCount (4096 on Adreno)

Push constants

- ARM Mali GPU engineers figured out that we were Load/Store bound on the ARM Mali GPUs
- On OpenGL ES, the driver can pin shader uniforms into GPU registers
  - Vulkan only has constant buffers so the driver cannot do that automatically
- Push constants to the rescue!
  - On ARM Mali GPUs push constants are automatically pinned to GPU registers
- Load-time decision:
  - Identify sufficiently small cbuffer (that's used in fragment shader) in SPIR-V bytecode
  - Transform bitcode on-the-fly to declare a push constant block instead of a cbuffer.
- Massive perf improvement on ARM Mali GPUs

Specialization constants

- DX11 aggressively recompiles shaders on the fly to get rid of static branching
  - Faster on GPU-bound work loads
- Good thing we control the shader compiler (HLSLcc), so:
  - Identify all conditional branches whose dependency tree only contains uniforms.
  - Transform them into specialization constants, encode condition expression into spec. constant name.
  - At runtime, evaluate the expression, pass values to vkCreateGraphicsPipelines() (or fetch from cache)
- Up to 30% perf improvement in GPU-bound cases
  - Real-world benefits smaller

Going multicore

- Multicore rendering in Unity
  - Generate batches of ~100 draw calls each
  - Call GfxDevice::ExecuteAsync() with an array of batches
  - GfxDevice spawns a job for each batch
    - Job itself calls back to the actual renderer to perform the rendering
    - Each job has its own GfxDevice object
    - After jobs are done, submit the results

Vulkan-specifics

- The Vulkan API sets some limitations on how we can approach the problem. Options:
- Primary command buffer per job:
  - Pros: No limitations on what the render job is allowed to do (barriers, copy operations, RT switches)
  - Cons: Renderpass has to begin and end within the same command buffer
- Secondary command buffer per job:
  - Pros: Can continue a renderpass from the parent command buffer
  - Cons: Cannot do anything else.
- Pro tip #5: Don't reuse secondary command buffers. Bad idea on many GPUs.

Our approach

- Each render job builds 1-n secondary command buffers plus a list of tasks
  - Tasks include things like "Submit this secondary command buffer", "Add a render barrier"
- Separate Task Executor thread
  - Waits for job completion
  - Builds the primary command buffer
  - Barriers, copies, resource uploads
  - Begin/End Renderpass
  - Executes the tasks
  - Can reorder things for efficiency

Making it fast

- Scratch buffer is shared between threads
  - Lockless allocation unless need to switch VkBuffers
  - Fast path is a single atomic add
- Descriptor set cache is also shared between threads
  - User-mode RWLock way too slow
  - Having descriptor set cache per shader program helps
  - Ended up making dense_hash_map re-entrant for reading (mutex-protected for writing)
    - Detect if insertion would cause the table to be resized
    - Create completely new dense_hash_map, copy contents over
    - Delay deletion until end of frame

Summary

- Vulkan is pretty fast. Use it!
- Vulkan support is shipping in Unity 5.6
  - Android
  - Linux
  - Windows
- Enable it from the Player Settings inspector
  - Uncheck "Automatic Graphics API" checkbox
  - Add Vulkan to the API list, and drag it to top
- Feedback and bug reports are welcome!

Don’t miss the chance to win cool prizes

Thur. March 2nd, 2:45 PM Unity Booth #1402	Improving mobile gaming experience with Vulkan *Unity and Samsung*

Prize draw at 5 PM Thursday at ARM booth #1942
See the postcard we handed out for more detail.

Thank you!

ARM