Achieving Console Quality Games on Mobile

ARM

DIGITAL LEGENDS ENTERTAINMENT

Peter Harris, Senior Principal Engineer, ARM
Unai Landa, CTO, Digital Legends
Jon Kirkham, Staff Engineer, ARM

GDC 2017

Agenda

- Premium smartphone in 2017
  - ARM Cortex CPU efficiency
  - ARM Mali GPU efficiency
- Best practises
  - Six principles of high performance rendering
  - Digital Legends Afterpulse case study
- Mali Tools overview

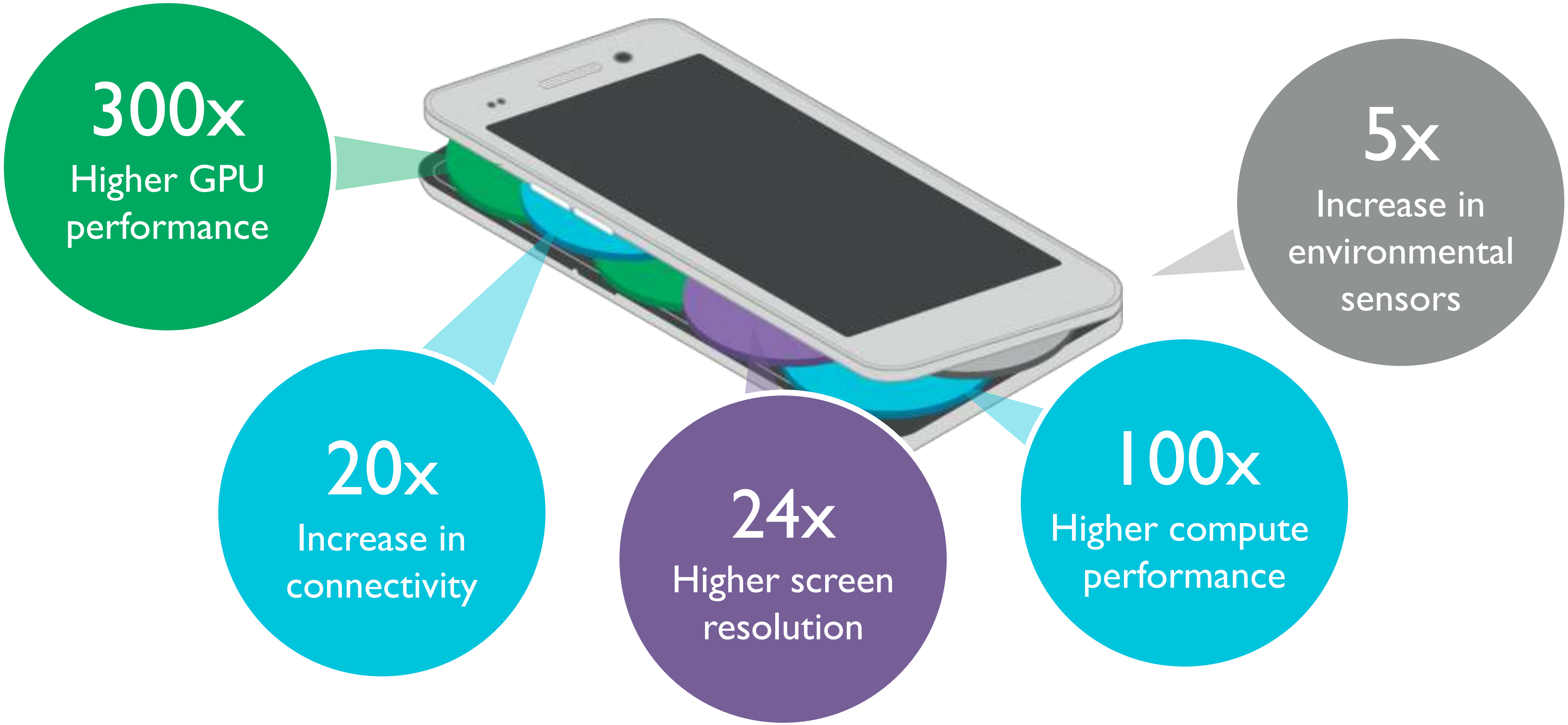
ARM

Premium smartphone in 2017

ARM

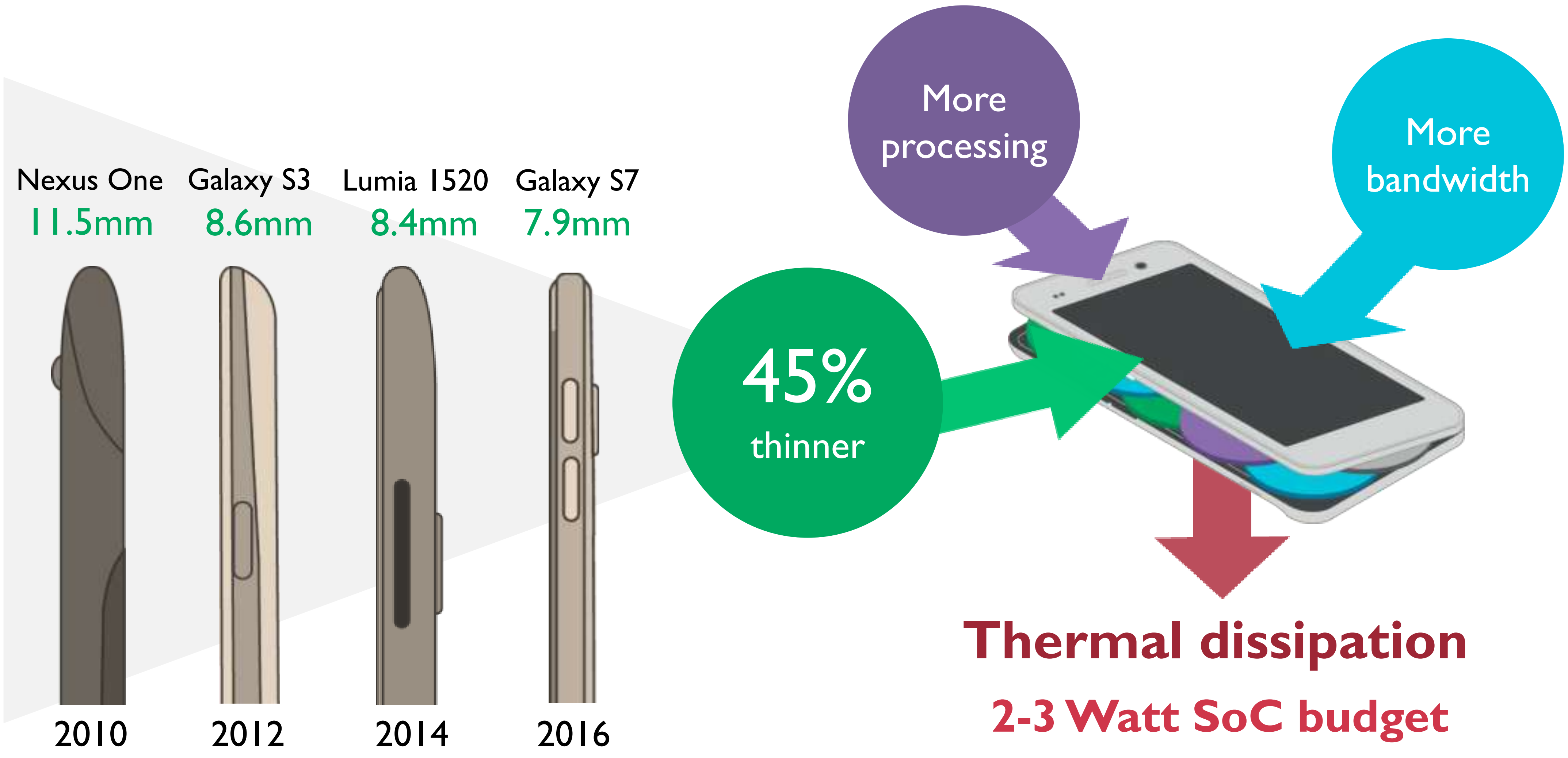
The premium smartphone

Today’s high-end phone compared to 2009



ARM

The premium content challenge

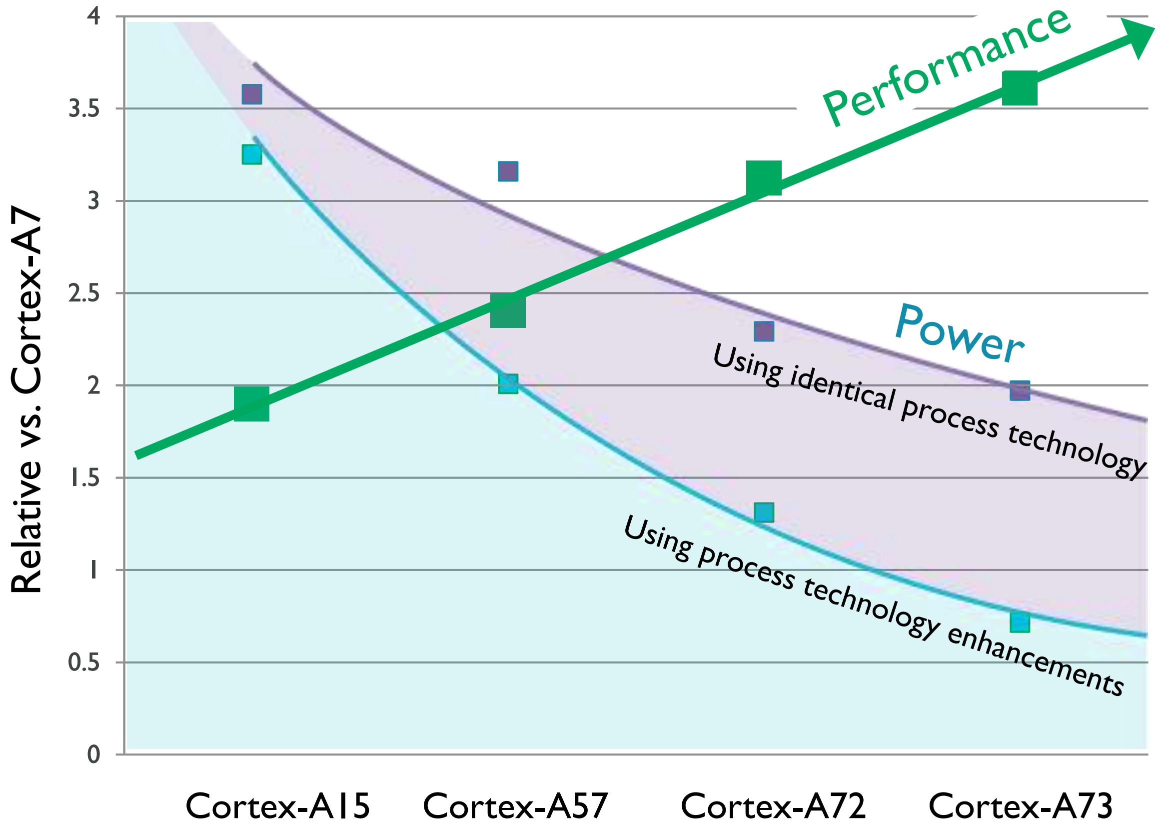


Thermal dissipation

2-3 Watt SoC budget

ARM

More performance, less power



Continuous growth in delivered performance

and

Continuous reduction in power consumption

Power efficiency contributing to longer battery life

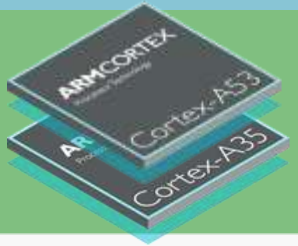
or

Power efficiency allowing available power budget to be reallocated

Power consumption measured in mW/unit performance
Performance measured as single thread at-speed

ARM

big.LITTLE: A technology that keeps improving

		→ 2017
High Performance “big”		
High Efficiency “LITTLE”		
Software	Intelligent Power Allocation* (IPA)	Energy Aware Scheduling** (EAS) (includes IPA)
Improvement	Thermal management	Mainline Linux, Generic energy models for task scheduling

* *Available on Linux today*

** *Ready for upstream*

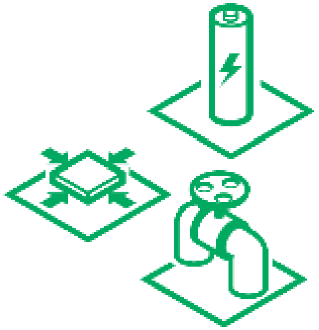
Vulkan™

- Reduces driver draw overhead
- Adds multi-threaded rendering support
- Reduces average per-core CPU load
- Allows more tasks to use LITTLE cores
- Improves overall task energy efficiency

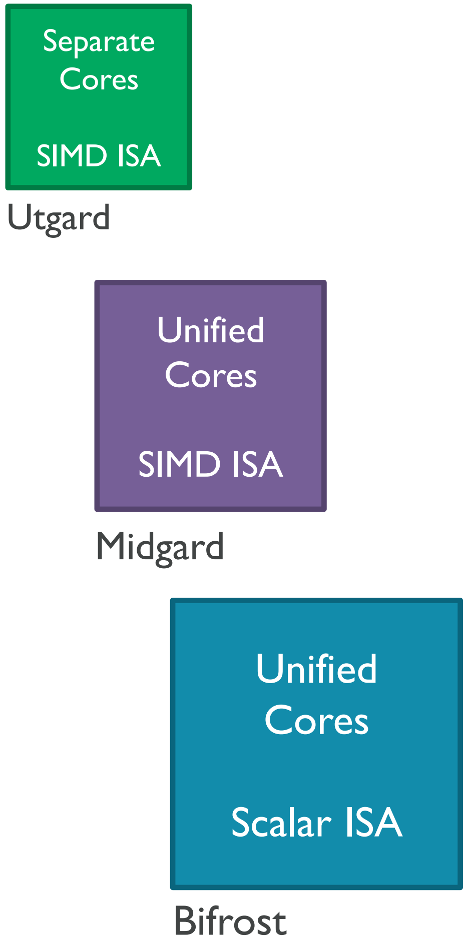
ARM

Introducing the Bifrost architecture

- 3rd generation programmable Mali GPU
  - **Energy efficiency:** more FPS per Watt
  - **Performance density:** more FPS per mm^2
  - **Bandwidth efficiency:** fewer bytes per frame



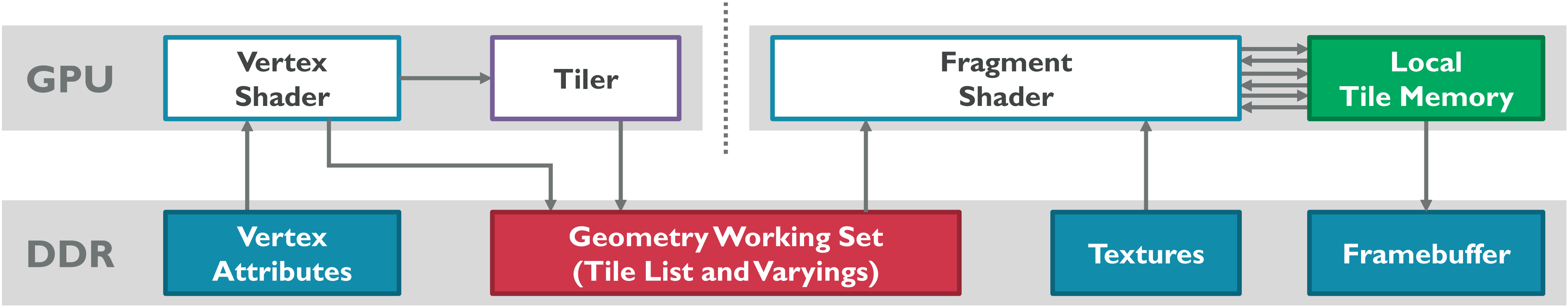
- New scalar ISA with quad-based arithmetic units
  - Maximize efficiency of the arithmetic hardware in the design
- New geometry data flow
  - Minimize vertex bandwidth related to culled triangles



ARM

Tile-based rendering pipeline

- All Mali GPUs are tile-based renderers
  - All geometry processed before fragment shading is started
  - Fragment shading processed as a stream of 16x16 pixel tiles



- **Pros:** Fragment shading intermediate state local to the GPU
- **Cons:** Geometry intermediate state sent via system memory

ARM

Mali best practices

ARM

“Efficiency is doing things right

Effectiveness is doing the right things”

- Peter Drucker

ARM

The Key Principle

Spend cycles where they make a visible difference to the final render

ARM

Principle one

Remove major redundancy in the application

ARM

Applications know more than the driver

- Graphics drivers are deliberately ignorant of overall scene state
  - Draw calls and triangles within them are processed in isolation
- Ignorance is pursued by design because it keeps thing fast
  - ... but means that drivers cannot apply high-level optimizations
- Only the application has any high-level knowledge of the scene
  - Exploit knowledge of the scene structure ruthlessly in your game engines
- The fastest mesh you'll ever draw is the one that you don't draw at all

ARM

Principle two

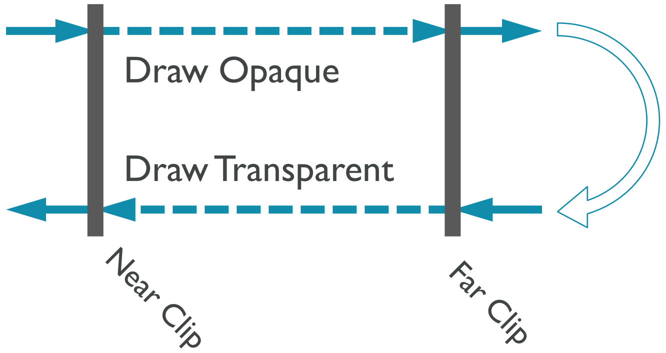
Help the hardware remove in-frustum redundancy

ARM

Hardware tools

- **Do:** Remember to enable the facing test to kill back-facing triangles

- **Do:** Maximize use of early depth and stencil "ZS" testing
  - **Render order:** opaque front-to-back then transparent back-to-front



- **Do:** Maximize potential use of Mali Forward Pixel Kill hidden surface removal*
  - Opaque fragments can cull occluded fragments even if not in front-to-back order
  - **Opaque:** no blending, no shader discard, no alpha-to-coverage
  - **Occluded:** any fragment without side-effects

** Present in Mali-T620 onwards*

Principle three

Amortize software overheads

ARM

Draw call batching

- Committing draw operations to the command stream is not free
  - CPU setup cost setting up the state and emitting the commands
- **Do:** Batch draws for multiple objects into a single larger draw
  - Use texture atlases to merge distinct render states into a single batch
  - Use static batching for stationary objects
  - Use runtime batching for objects which move
- **Do:** Batching is still worth while on Vulkan
- **Beware:** Trade-off between optimal batching and optimal culling/depth sorting

Principle four

Optimize your data streams

ARM

Geometry streams

- Effective geometry encoding aims to minimize the geometry bandwidth
  - **Vertex shader bandwidth:** Attribute reads, Varying writes
  - **Fragment shader bandwidth:** Varying reads
- **Do:** Use appropriate geometry level of detail and triangle density
  - Dynamic mesh LoD based on view-distance if large range of depth values used for a mesh
- **Do:** Ensure good spatial locality and data density in attribute encoding
  - Aim for contiguous index ranges for each draw without holes (for all LoD levels)
  - Use fp16 "mediump" inputs as much as possible
  - Minimize padding and unused fields in any input structures
- **Do:** Interleave non-position attributes in one buffer and position in another
  - Reduces data bandwidth for culled triangles in Bifrost; only need position data before culling

Texture streams

- **Do:** Use texture compression
  - OpenGL ES 3.0 and 3.1 mandates ETC2 + EAC
    - Standard support for alpha channel compression
  - OpenGL ES 3.2 mandates ASTC 2D LDR profile
    - Extremely flexible texture compression in terms of both formats and bitrates
  - Mali supports all ASTC extensions: 2D LDR, 2D HDR, and 3D volumetric textures

- **Do:** Use mipmapping:
  - Looks better *and* goes faster; no reason not to use it for 3D content

- **Beware** Trilinear (`GL_*_MIPMAP_LINEAR`) filtering is half throughput
  - If texture unit limited just use bilinear (`GL_*_MIPMAP_NEAREST`) filtering

Principle five

Play to the strengths of the underlying GPU

ARM

Play to architecture strengths

- Tile memory in a tile-based renderer provides some useful features

- Low cost 4x and 8x multi-sample anti-aliasing
  - **Do:** Use **`EXT_multisampled_render_to_texture`** to get free resolve for off-screen renders

- Direct access to the tile-buffer for in-tile deferred rendering schemes
  - Structure-like access: **`EXT_shader_pixel_local_storage`**
  - Framebuffer-like access: **`EXT_shader_framebuffer_fetch`**
    - Also: **`ARM_shader_framebuffer_fetch_depth_stencil`**
  - Vulkan support via subpass functionality exposed in the API
  - **Do:** aim for maximum of 128-bits per pixel of storage

ARM

Principle six

Optimize your shader code

ARM

Shaders

- **Do:** Optimize the most significant shaders
  - It's time consuming so you don't want to do it for all shaders
- **Do:** Optimize what you can by hand in the shader source
  - Developers often over-estimate what a compiler is able to optimize
  - If you get it right in the source then its guaranteed to be right in the binary
- **Do:** Use fp16 "mediump" where possible for both data feeds and computation
- **Do:** Write vector code as it's a more natural fit for existing Mali devices
- **Don't:** Reinvent the ESSL built-in function library in hand-written code
  - It's very well optimized and often backed by dedicated hardware

Afterpulse

A Digital Legends case study

ARM

Our motivations

- Heat
- Heat
- Heat
- Heat

ARM

Principle One: Engine redundancy removal

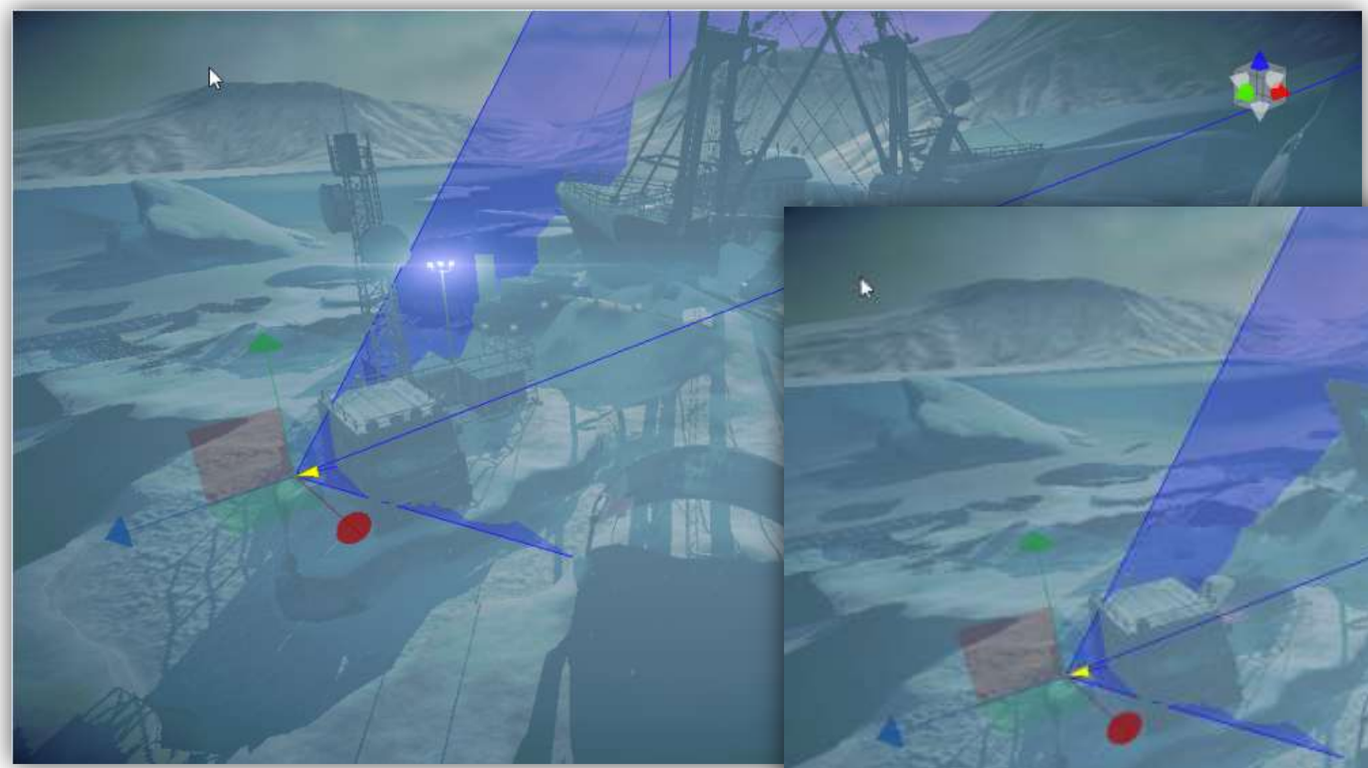
- Shadow proxy meshes
- Frustum culling
- Occlusion culling
- Level of detail
- Contribution culling

Principle One: Occlusion culling example

Player View

ARM

Principle One: Occlusion culling example



Culling off

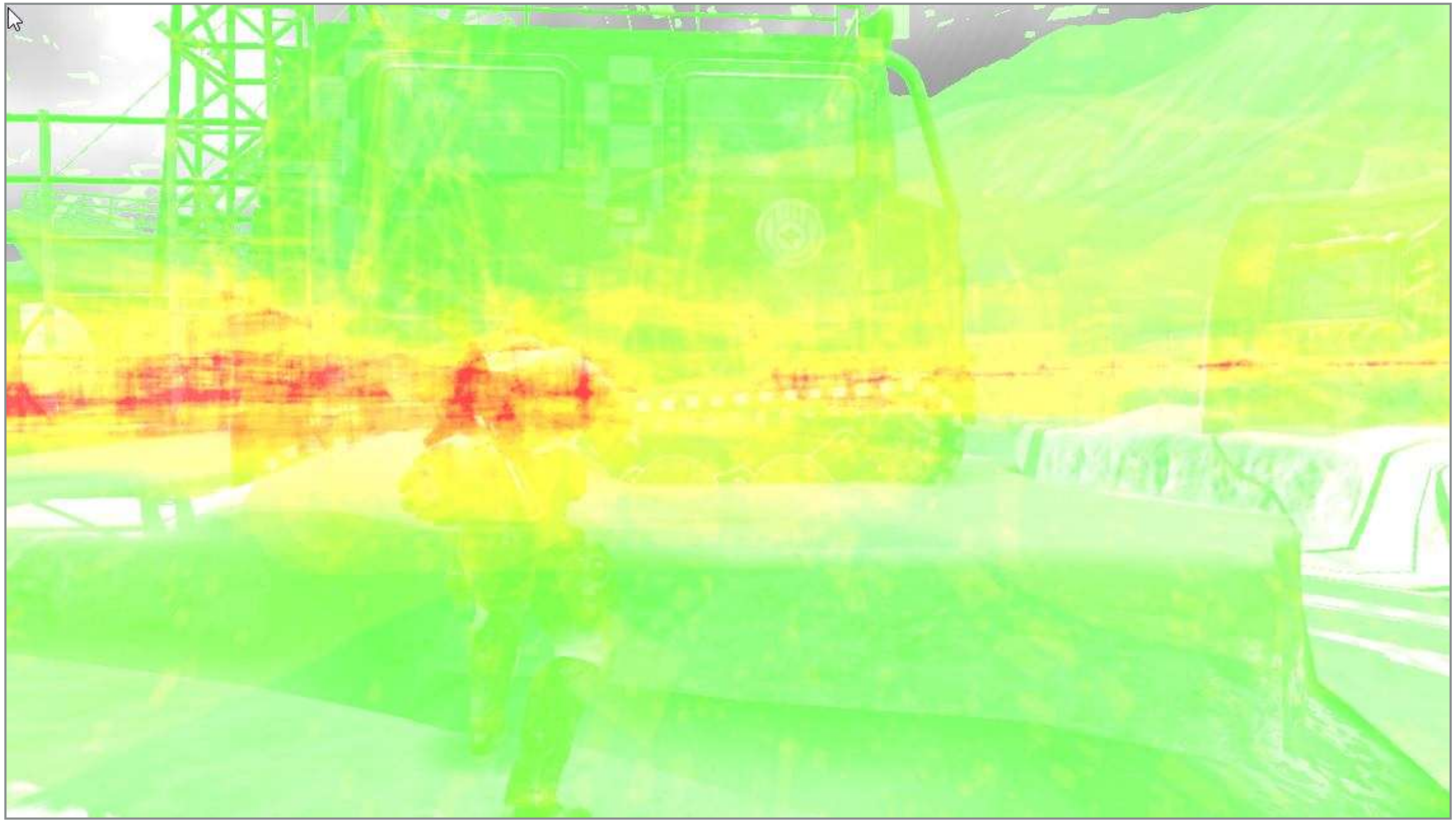
Culling on

Principle Two: Assist overdraw removal

- Draw Opaque, then alpha-test, then… no don't draw alpha.
  - Unless you really need it
  - Avoid **`discard`** in shaders
- Use **`layout(early_fragment_tests)`** in fragment shaders
  - Forces early-zs testing in situations where engine knows it is safe, but the driver might not
- Do "loose" front to back sort of object batches
  - Efficiency of batching tested on a per-game level basis

ARM

Principle Two: Assist overdraw removal



ARM

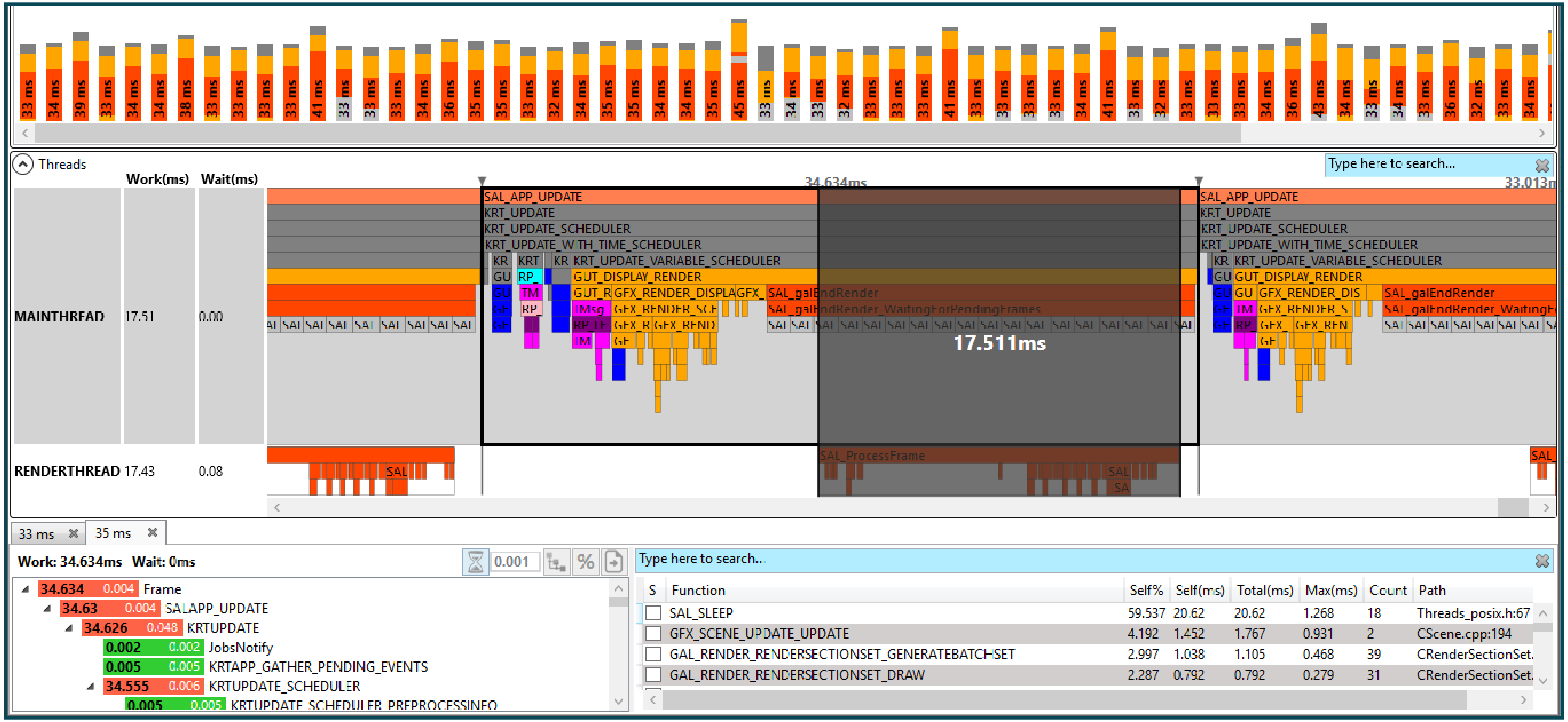
Principle Three: Amortize driver overheads

- Engine aims to minimize the number of driver calls
  - Avoid frame buffer changes and reuse them if possible, build some kind of draw graph and optimize it
  - Group by geometry, textures and parameters
  - Use instances

- OpenGL API calls are offloaded to dedicated CPU dispatch thread
  - Main game logic thread is not limited by the driver times

ARM

Principle Three: Amortize driver overheads



ARM

Principle Four: Optimize data streams

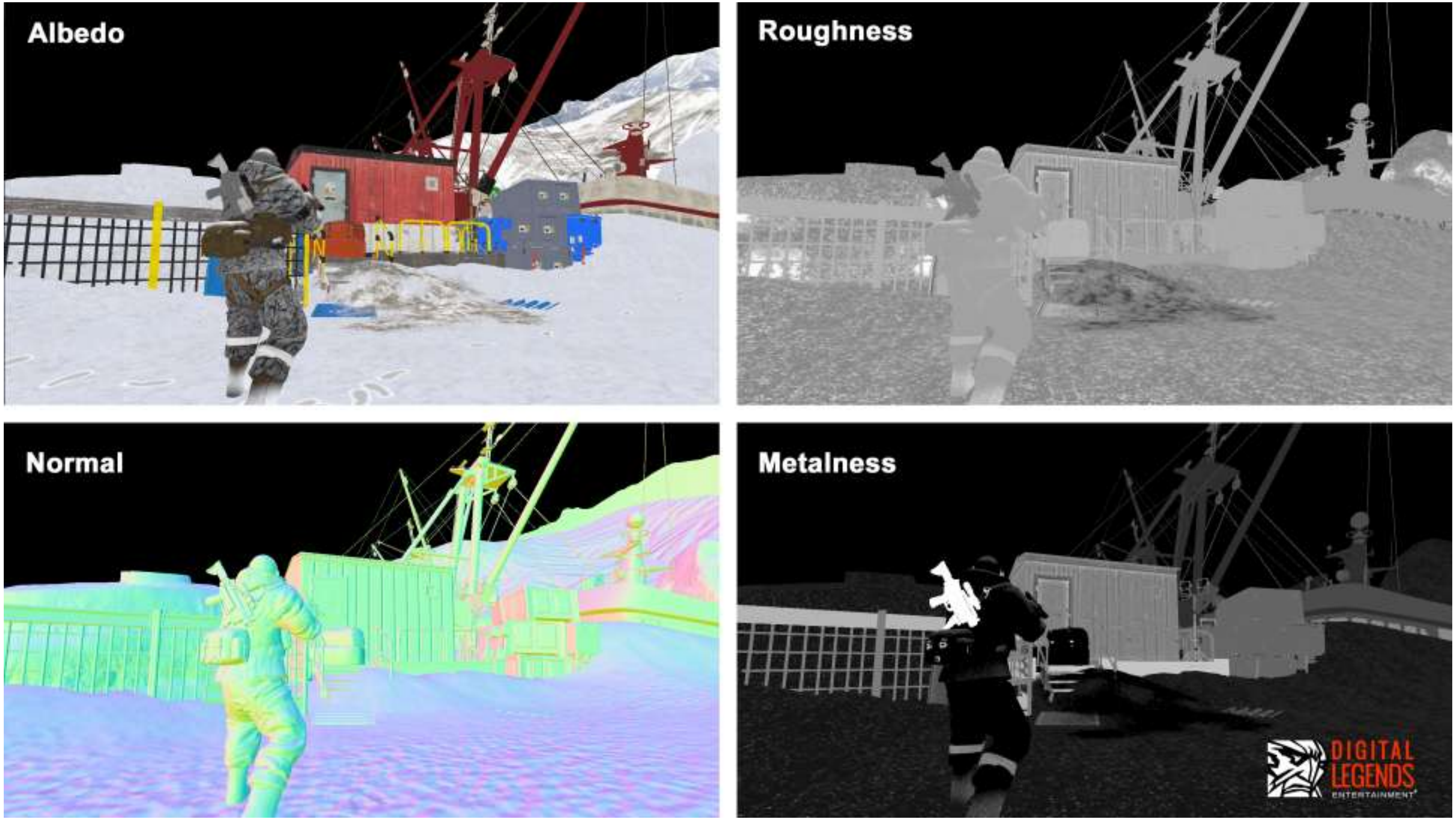
- Geometry streams
  - Use "compact" formats like **GL_INT_2_10_10_10_REV** for tangents and normals
  - Use half float for object texture coordinates
  - RGBA8 **GL_BYTE** vectors for colors
- Vertex Interpolators:
  - In our experience they are expensive if they are big
- Texture
  - Use ASTC formats as much as you can
- Use uniform blocks
  - Avoid redundant parameter updates to GPU, hash and track draw call parameters
  - Split shader data at least into local and global buffers
  - Promote "static" data from dynamic buffers to static ones if not changed in several frames

Principle Five: Play to strengths of the GPU

- Use of the PLS and/or `frame_buffer_fetch` is key to the pipeline
  - Reducing bandwidth and heat, and saving battery
- Also use `GL_ARM_shader_framebuffer_fetch_depth_stencil`
  - Avoid the z-write on the deferred pass and optimise the deferred lighting pass.
- Deferred lighting G-Buffer in pixel local storage looks like this:

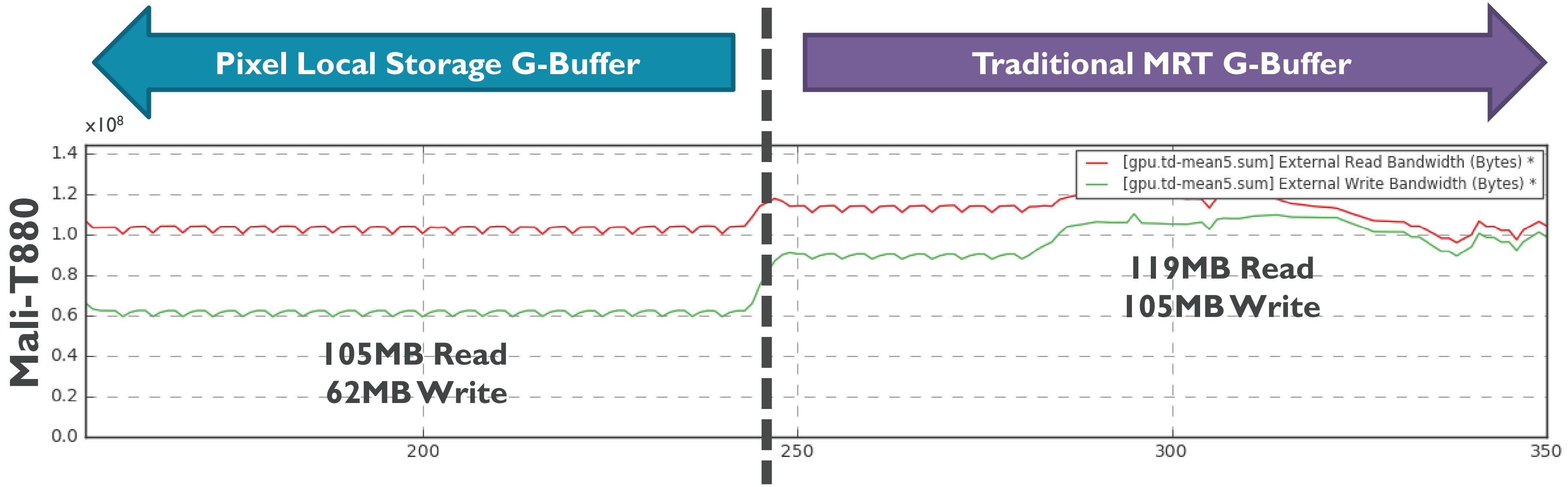
```
__pixel_localEXT FragLocalData {
    layout(r11f_g11f_b10f) krmFloat3 buff_0;
    layout(rgba8) krmFloat4 normals_gloss;
    layout(rgba8) krmFloat4 albedo_mtl;
} Storage;
```


ARM


Albedo
Roughness
Normal
Metalness
DIGITAL
LEGENDS
ENTERTAINMENT

Mali Pixel Local Storage bandwidth savings

- PLS avoids needs to read and write the G-Buffer via system RAM



- Total savings average 60MB of bandwidth a frame
  - Rough rule of thumb is an energy cost of 100pJ per byte of DDR memory access
  - 60MB * 30FPS * 100pJ = 180mW of power saving at the system level

ARM

Principle Six: Optimize your shaders

- Engine builds all the shader variations offline to avoid logic inside the shader

- All shaders moved to `mediump` precision by default
  - Be aggressive, spend time to fix visible precision issues later

- Tweaking required to find and fix the issues, but it pays

ARM

ARM

Mali analysis tools

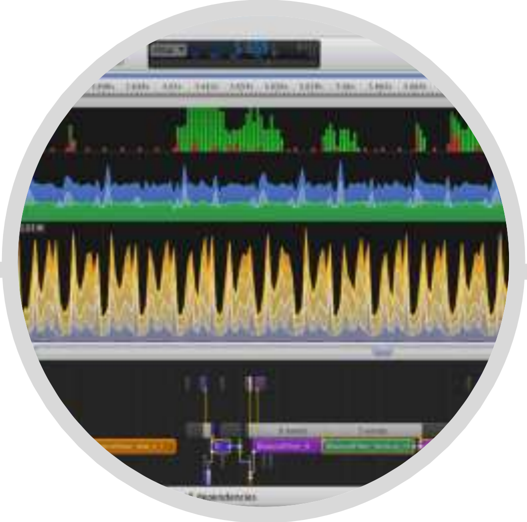
ARM

Tools workflow

Analyze

DS-5 Streamline

- Profile CPUs and Mali GPUs
- Timeline
- HW counters
- OpenCL visualizer



Debug

Mali Graphics Debugger

- API trace & debug
- OpenGL ES, OpenCL
- Debug and improve performance at frame level

Optimize

Mali Offline Compiler

- Analyze shader performance
- Command line tool
- Number of cycles
- Registers utilization

ARM

ARM DS-5 Streamline

Drill down to the source code

Speed up your code

OpenCL™ visualizer

Mali GPU support

Customize it for your system

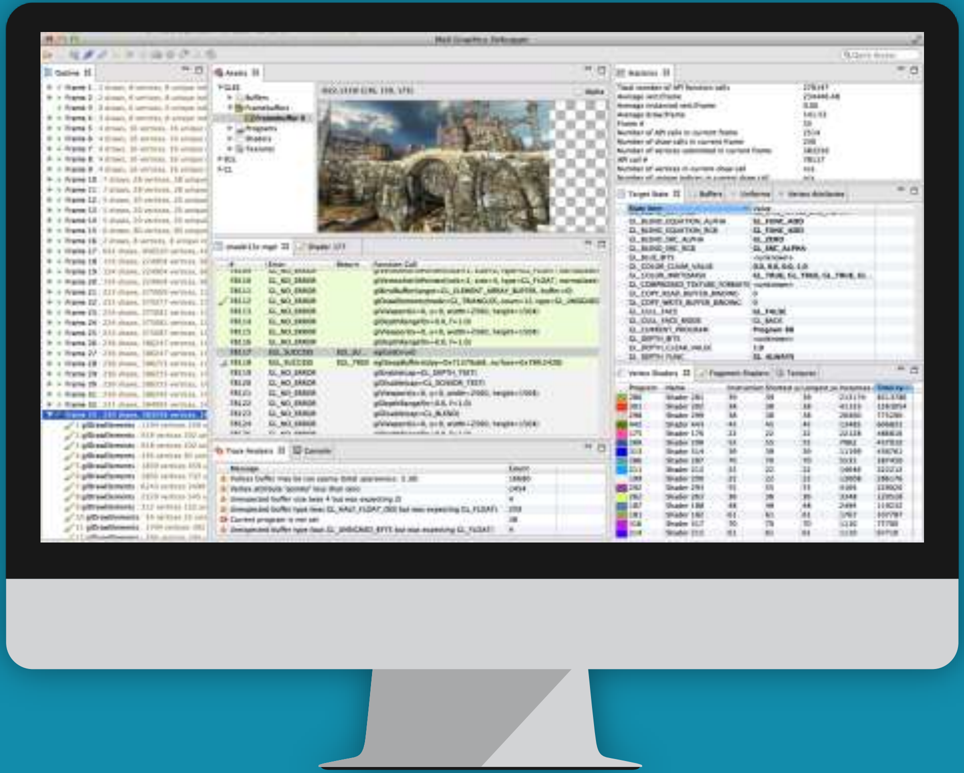
ARM

Mali Graphics Debugger (MGD)

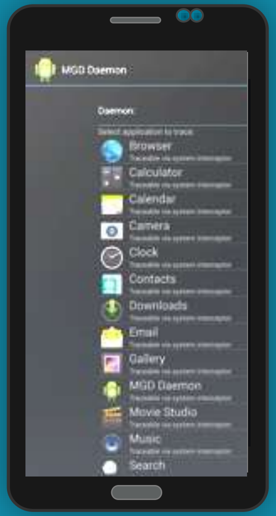
Frame analyzer

Advanced drawing modes

Advanced API debugger



Android application



- Graphics state visibility
- Analyze shaders and kernels
- Flexible and cross platform

ARM

New for GDC 2017

- Root access is no longer required for ARM DS-5 Streamline

- MGD can be used easily from:
  - Android™ Studio
  - Unity®
  - Unreal® Engine

Want to know more?

ARM Stand:

South Hall #1924

ARM Mali Developer Guides & Tools:

https://developer.arm.com/graphics

Don’t miss these other sessions and three ways to win cool prizes

Thur. March 2, 10:00-11:00 AM Moscone West – Rm. 3022	Get the most from Vulkan in Unity with practical examples from Infinite dreams *Joint with Unity and Infinite Dreams*

Daily prize draw at 5 PM Thursday at ARM booth #1942
See the postcard for more details.

ARM

ARM