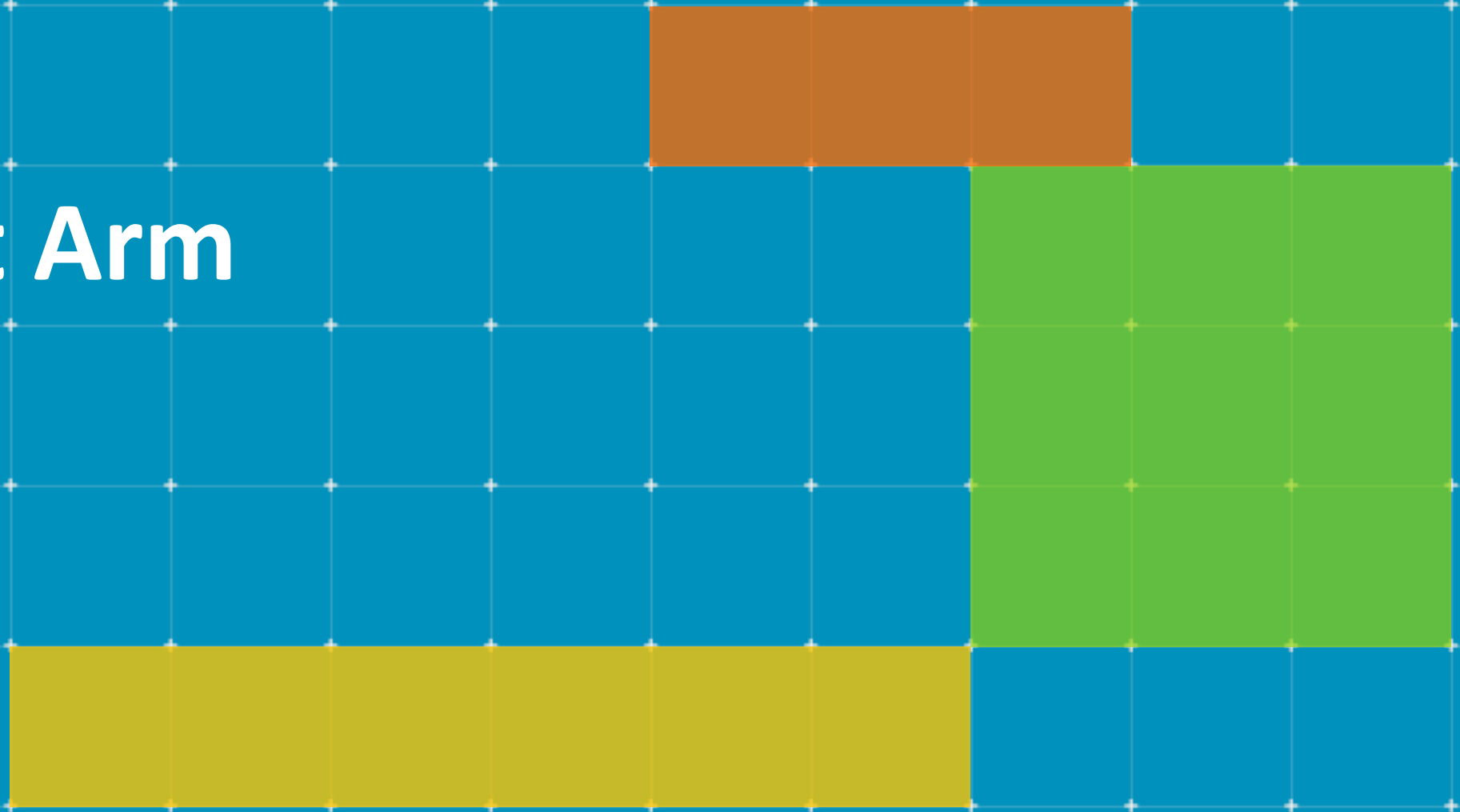
Hardware-Software Co-design at Arm GPUs

Johan Grönqvist

MCC 2017 - Uppsala

About Arm



arm

Arm Mali GPUs: The World's #1 Shipping Graphics Processor

151
Total Mali licenses

21
Mali video and display licenses

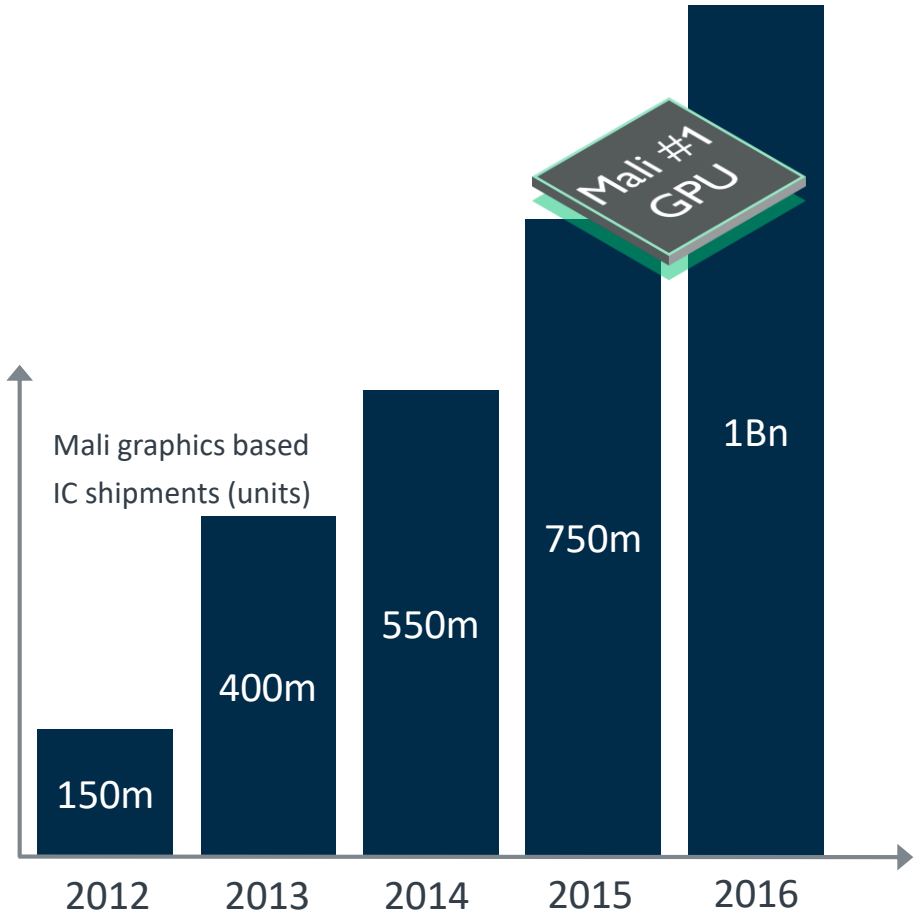
1Bn
Mali GPUs shipped in 2016

Mali GPUs are in:

~50%
of mobile VR…

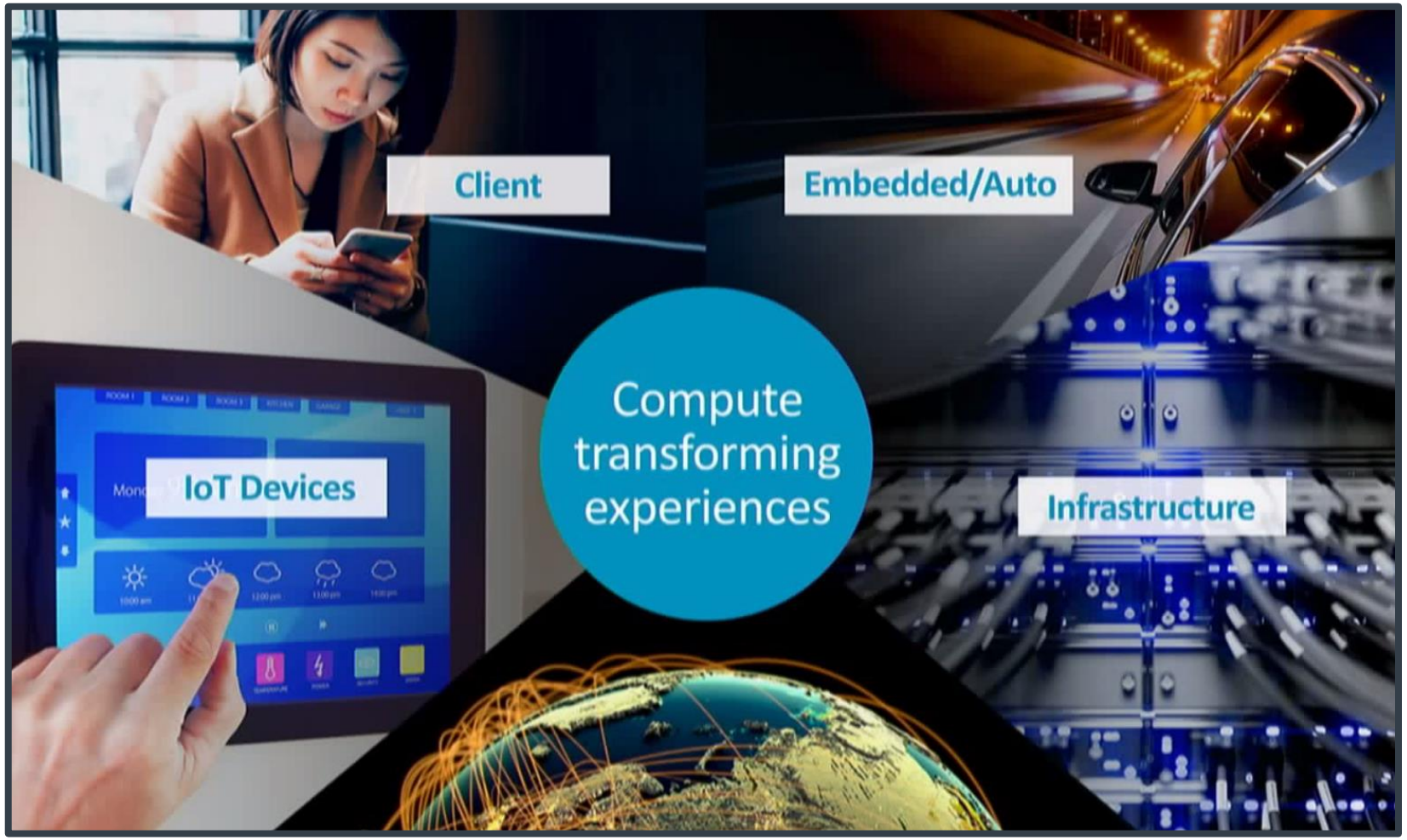
~80%
of DTVs…

~50%
of smartphones



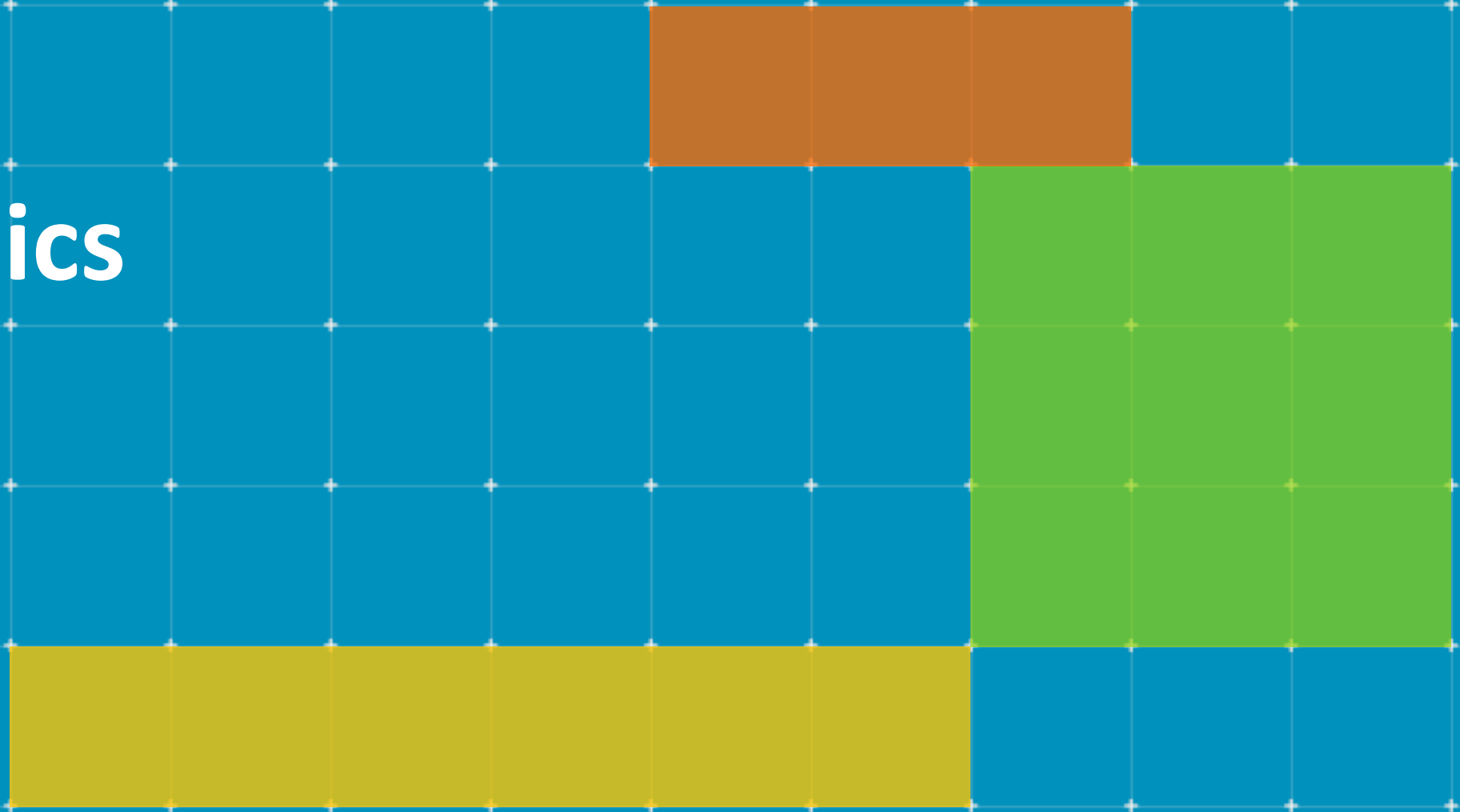
arm

Arm



arm

Graphics



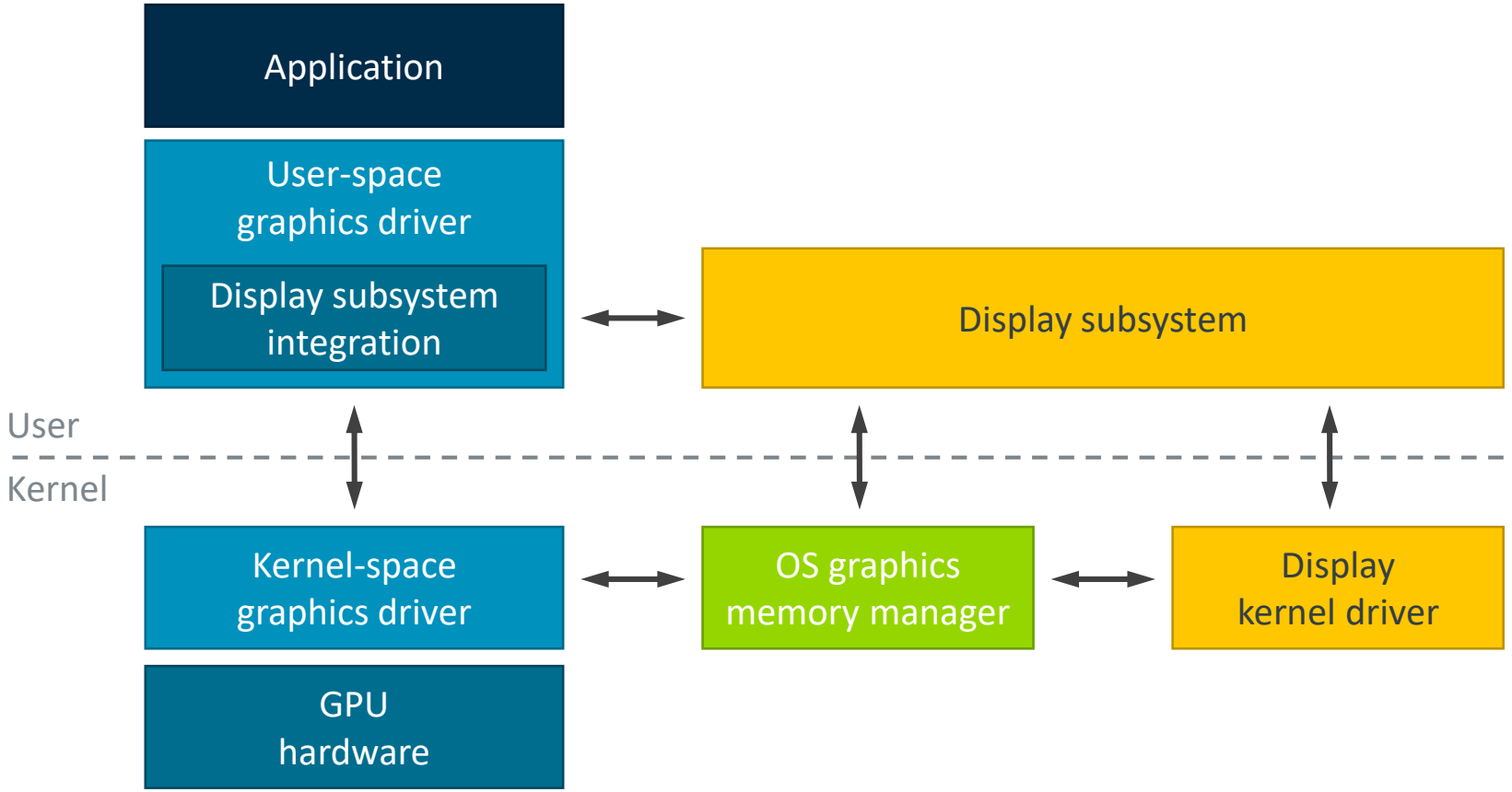
arm

Levels of Abstraction

Standardized APIs allow separation of concerns

Graphics APIs

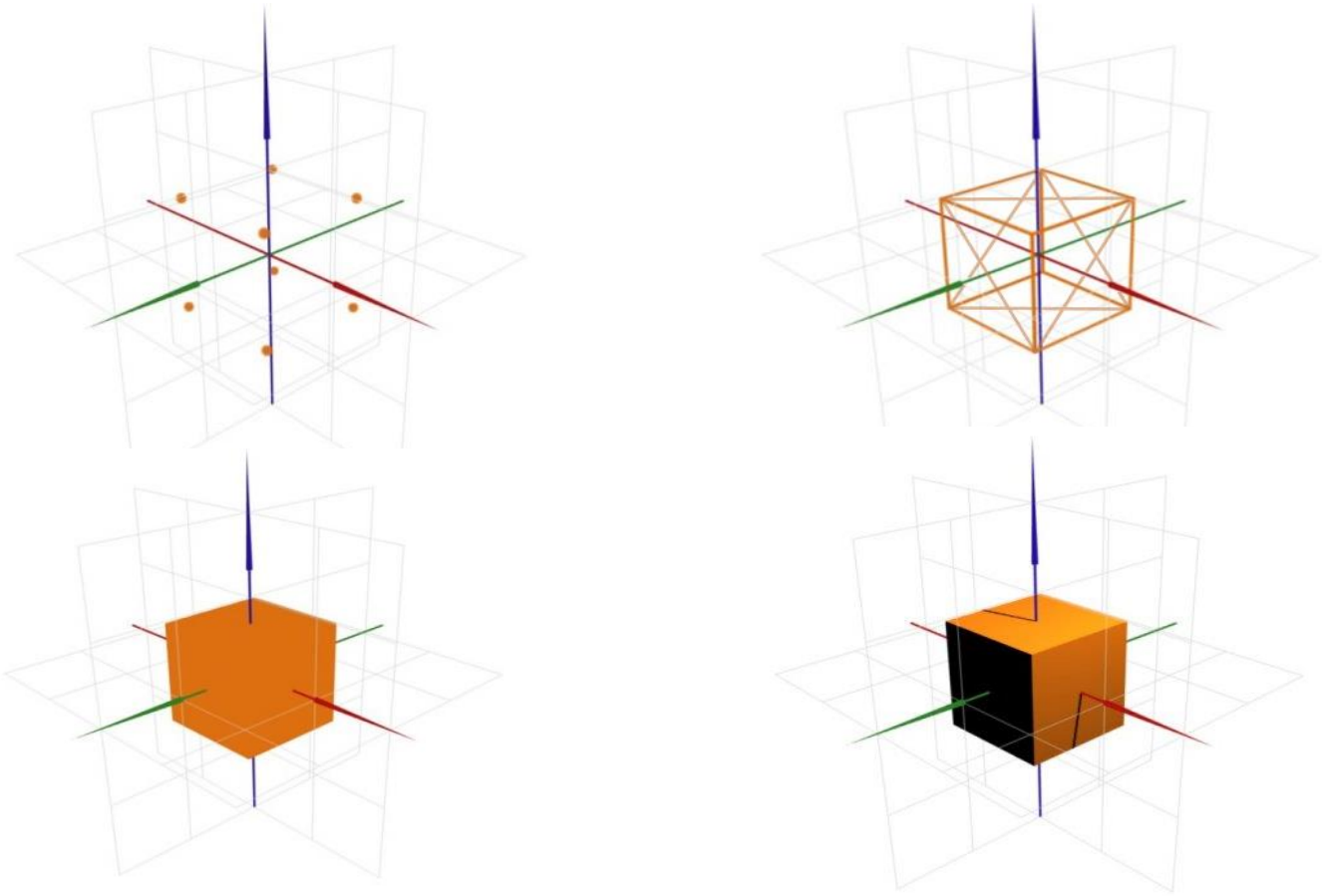
- HW hidden
  - ISA
  - Memory hierarchy
- Portability
- HW specific optimizations in the Application
- Application specific optimizations in the driver

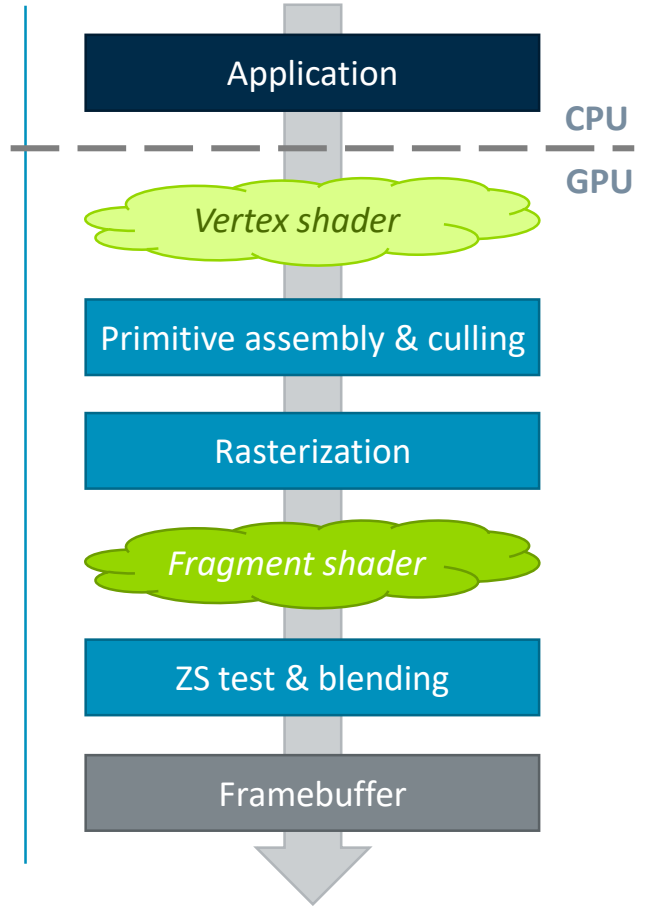


arm

A Graphics Pipeline

Vertices, triangles, fragments and pixels





arm

A Graphics Demo by Arm

Ice Cave, created by Arm in 2015 to show what kids of effects and level of detail a mobile GPU is capable of.

- https://www.youtube.com/watch?v=gsyBMHJVhXA

Pipeline & Optimizations

arm

A Schematic OpenGL Implementation

```
def shade(vertices, vertex_shader, fragment_shader, blend_shader, vertex_data):
        (vertices', vertex_data') = ([], [])
        for (v, vd) in zip(vertices, vertex_data):
                v', vd' = vertex_shader(v, vd)
                vertices'.append(v')
                vertex_data'.append(vd')
        triangles, triangle_data = primitive_assembly(vertices', vertex_data')
        for (t, td) in zip(triangles, triangle_data):
                fragments, fragment_data = rasterize_triangle(t, td)
                for (f, fd) in zip(fragments, fragment_data):
                        depth, color = fragment_shader(f, fd)
                        if is_visible(depth, pos):
                                output[pos] = blend_shader(output[pos], color)
```


arm

A Schematic OpenGL Implementation

```
def shade(vertices, vertex_shader, fragment_shader, blend_shader, vertex_data):
    (vertices', vertex_data') = ([], [])
    for (v, vd) in zip(vertices, vertex_data):
        v', vd' = vertex_shader(v, vd)
        vertices'.append(v')
        vertex_data'.append(vd')
    triangles, triangle_data = primitive_assembly(vertices', vertex_data')
    for (t, td) in zip(triangles, triangle_data):
        fragments, fragment_data = rasterize_triangle(t, td)
        for (f, fd) in zip(fragments, fragment_data):
            depth, color = fragment_shader(f, fd)
            if is_visible(depth, pos):
                output[pos] = blend_shader(output[pos], color)
```


arm

A Schematic OpenGL Implementation

```
def shade(vertices, vertex_shader, fragment_shader, blend_shader, vertex_data):
    (vertices', vertex_data') = ([], [])
    for (v, vd) in zip(vertices, vertex_data):
        v', vd' = vertex_shader(v, vd)
        vertices'.append(v')
        vertex_data'.append(vd')
    triangles, triangle_data = primitive_assembly(vertices', vertex_data')
    for (t, td) in zip(triangles, triangle_data):
        fragments, fragment_data = rasterize_triangle(t, td)
        for (f, fd) in zip(fragments, fragment_data):
            depth, color = fragment_shader(f, fd)
            if is_visible(depth, pos):
                output[pos] = blend_shader(output[pos], color)
```


arm

A Schematic OpenGL Implementation

```
def shade(vertices, vertex_shader, fragment_shader, blend_shader, vertex_data):
        (vertices', vertex_data') = ([], [])
        for (v, vd) in zip(vertices, vertex_data):
            v', vd' = vertex_shader(v, vd)
            vertices'.append(v')
            vertex_data'.append(vd')
        triangles, triangle_data = primitive_assembly(vertices', vertex_data')
        for (t, td) in zip(triangles, triangle_data):
            fragments, fragment_data = rasterize_triangle(t, td)
            for (f, fd) in zip(fragments, fragment_data):
                depth, color = fragment_shader(f, fd)
                if is_visible(depth, pos):
                    output[pos] = blend_shader(output[pos], color)
```


arm

A Schematic OpenGL Implementation

```
def shade(vertices, vertex_shader, fragment_shader, blend_shader, vertex_data):
    (vertices', vertex_data') = ([], [])
    for (v, vd) in zip(vertices, vertex_data):
        v', vd' = vertex_shader(v, vd)
        vertices'.append(v')
        vertex_data'.append(vd')
    triangles, triangle_data = primitive_assembly(vertices', vertex_data')
    for (t, td) in zip(triangles, triangle_data):
        fragments, fragment_data = rasterize_triangle(t, td)
        for (f, fd) in zip(fragments, fragment_data):
            depth, color = fragment_shader(f, fd)
            if is_visible(depth, pos):
                output[pos] = blend_shader(output[pos], color)
```


arm

A Schematic OpenGL Implementation

```
def shade(vertices, vertex_shader, fragment_shader, blend_shader, vertex_data):
        (vertices', vertex_data') = ([], [])
        for (v, vd) in zip(vertices, vertex_data):
            v', vd' = vertex_shader(v, vd)
            vertices'.append(v')
            vertex_data'.append(vd')
        triangles, triangle_data = primitive_assembly(vertices', vertex_data')
        for (t, td) in zip(triangles, triangle_data):
            fragments, fragment_data = rasterize_triangle(t, td)
            for (f, fd) in zip(fragments, fragment_data):
                depth, color = fragment_shader(f, fd)
                if is_visible(depth, pos):
                    output[pos] = blend_shader(output[pos], color)
```


arm

Programmable Parts

Everything else is fixed function

```
def shade(vertices, vertex_shader, fragment_shader, blend_shader, vertex_data):
        (vertices', vertex_data') = ([], [])
        for (v, vd) in zip(vertices, vertex_data):
                v', vd' = vertex_shader(v, vd)
                vertices'.append(v')
                vertex_data'.append(vd')
        triangles, triangle_data = primitive_assembly(vertices', vertex_data')
        for (t, td) in zip(triangles, triangle_data):
                fragments, fragment_data = rasterize_triangle(t, td)
                for (f, fd) in zip(fragments, fragment_data):
                        depth, color = fragment_shader(f, fd)
                        if is_visible(depth, pos):
                                output[pos] = blend_shader(output[pos], color)
```


arm

Parallelization Opportunities

We have many shading jobs and compute jobs running at the same time

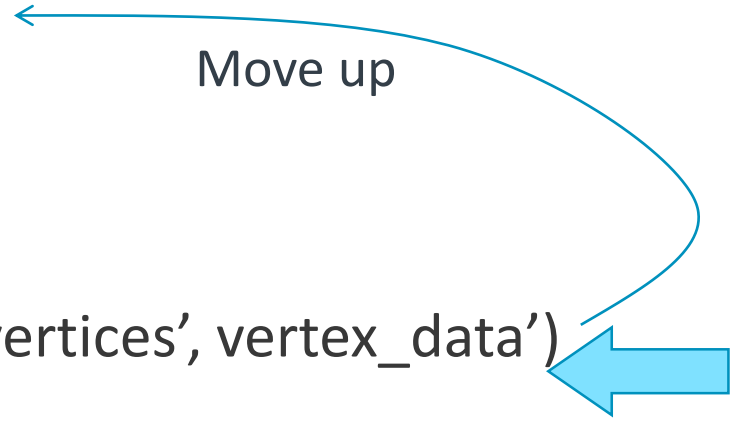

```
def shade(vertices, vertex_shader, fragment_shader, blend_shader, vertex_data):
        (vertices', vertex_data') = ([], [])
        for (v, vd) in zip(vertices, vertex_data):
                v', vd' = vertex_shader(v, vd)
                vertices'.append(v')
                vertex_data'.append(vd')
        triangles, triangle_data = primitive_assembly(vertices', vertex_data')
        for (t, td) in zip(triangles, triangle_data):
                fragments, fragment_data = rasterize_triangle(t, td)
                for (f, fd) in zip(fragments, fragment_data):
                        depth, color = fragment_shader(f, fd)
                        if is_visible(depth, pos):
                                output[pos] = blend_shader(output[pos], color)
```


arm

Early Visibility Testing

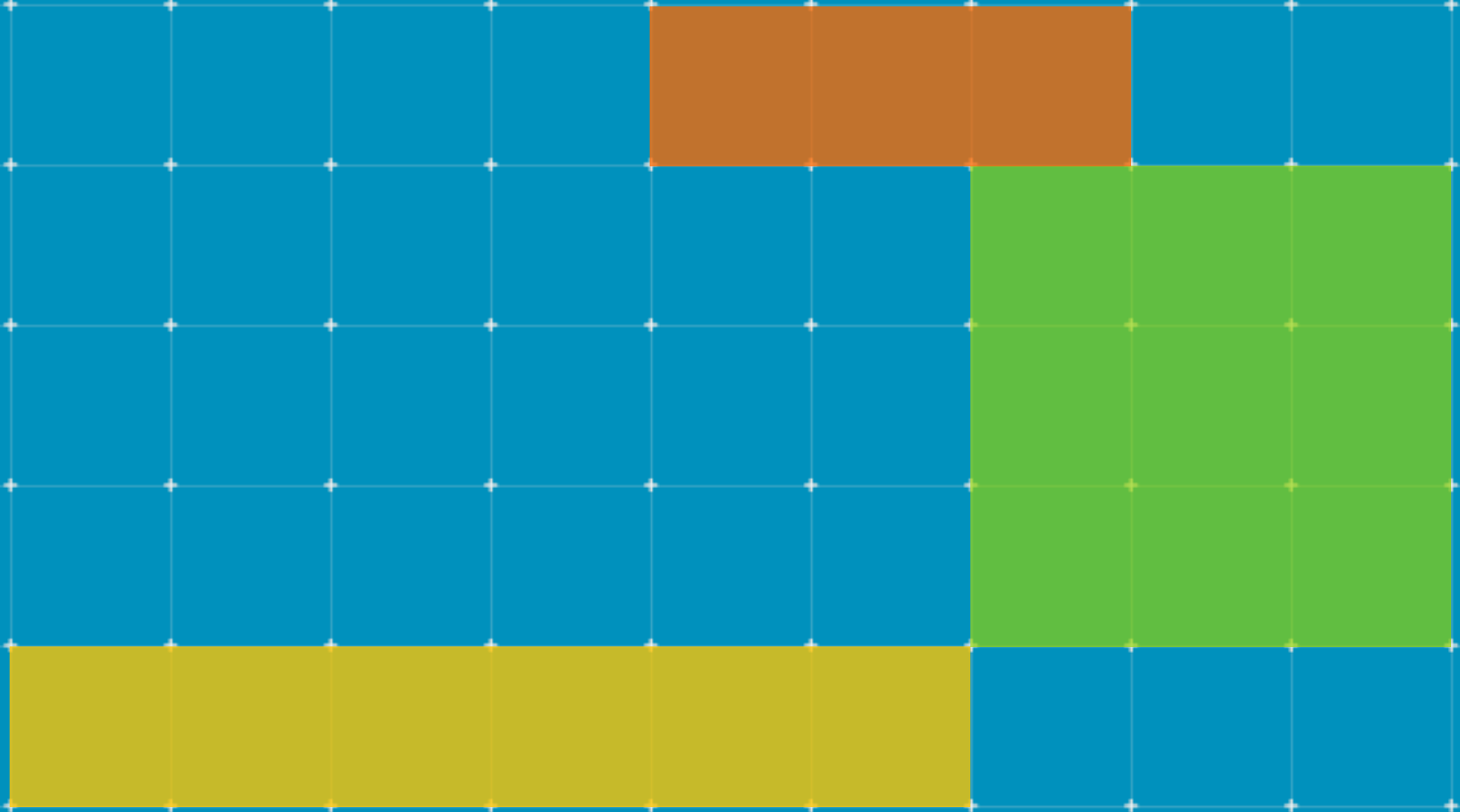
Dependency tracking becomes tricky with visibility tests in many places

```
def shade(vertices, vertex_shader, fragment_shader, blend_shader, vertex_data):
        (vertices', vertex_data') = ([], [])
        for (v, vd) in zip(vertices, vertex_data):
                v', vd' = vertex_shader(v, vd)
                vertices'.append(v')
                vertex_data'.append(vd')
        triangles, triangle_data = primitive_assembly(vertices', vertex_data')
        for (t, td) in zip(triangles, triangle_data):
                fragments, fragment_data = rasterize_triangle(t, td)
                for (f, fd) in zip(fragments, fragment_data):
                        depth, color = fragment_shader(f, fd)
                        if is_visible(depth, pos):
                                output[pos] = blend_shader(output[pos], color)
```



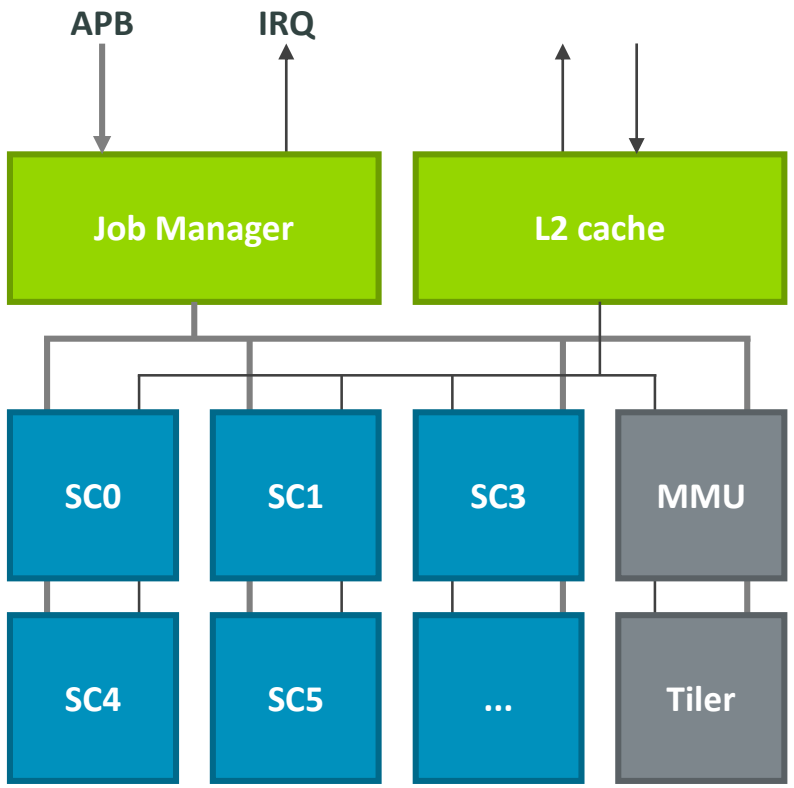
arm

GPUs



arm

Arm Bifrost GPU Architecture



arm

A Bifrost Core

Thread management

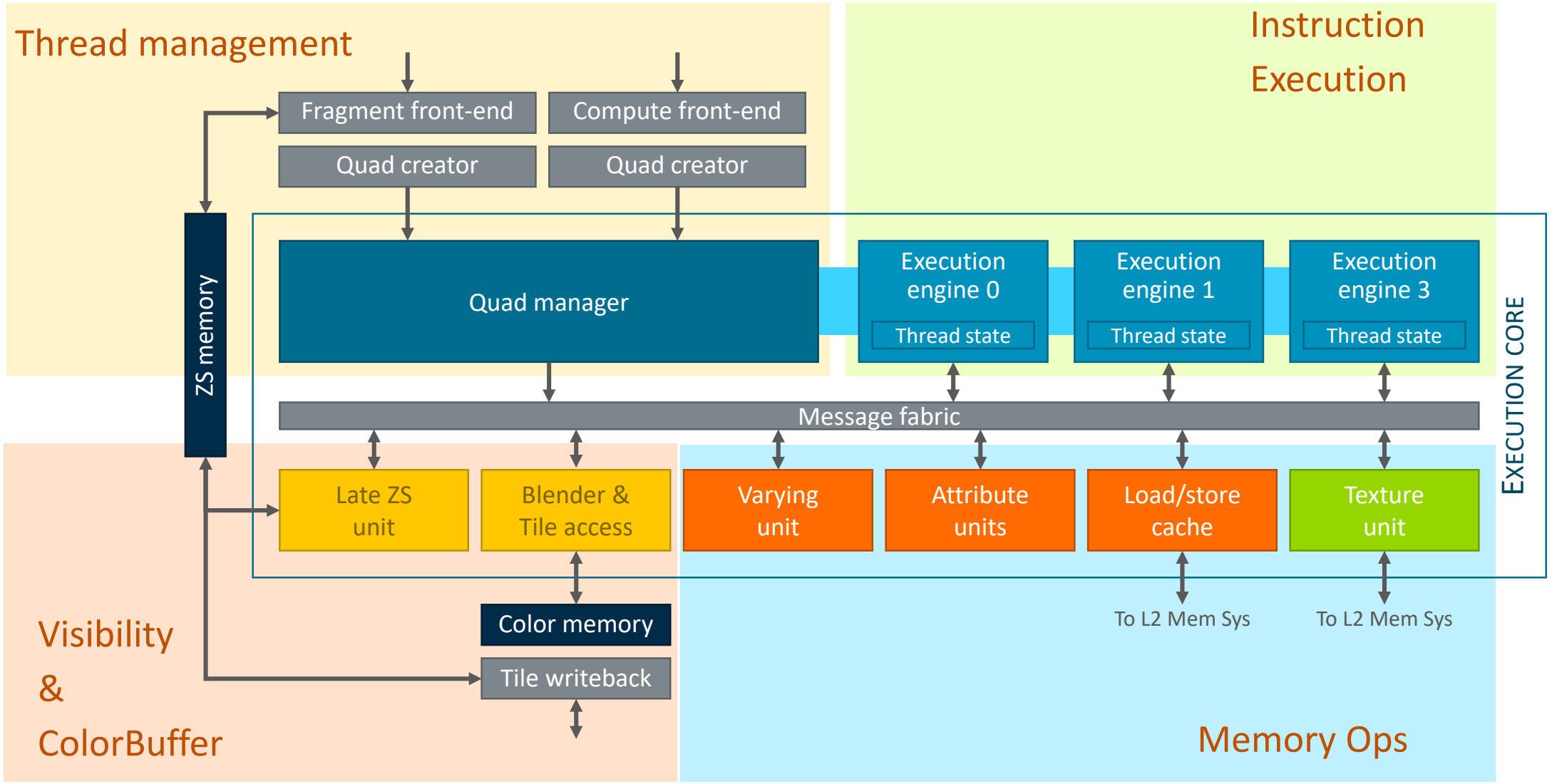
Instruction
Execution

Visibility
&
ColorBuffer

Memory Ops

arm

A Bifrost Core



arm

Fixed Use-Case and Standardized APIs

Enabling hardware-software co-design

A GPU implementation uses

- Fixed function hardware for triangle operations, rasterizations, interpolations, and more
- Hardware to ensure serialization when necessary
- Buffers to reorder threads to enable more parallelism, where serialization is not needed
- Fixed function hardware for killing hidden pixels as early as possible

Fixed Use-Case and Standardized APIs

Enabling hardware-software co-design

A GPU implementation uses

- Fixed function hardware for triangle operations, rasterizations, interpolations, and more
- Hardware to ensure serialization when necessary
- Buffers to reorder threads to enable more parallelism, where serialization is not needed
- Fixed function hardware for killing hidden pixels as early as possible

Tuning hardware becomes very content dependent

- Complexity of the Geometry, Number of buffers used, the amounts of data load per thread
- Software and hardware evolve together over time

Fixed Use-Case and Standardized APIs

Enabling hardware-software co-design

A GPU implementation uses

- Fixed function hardware for triangle operations, rasterizations, interpolations, and more
- Hardware to ensure serialization when necessary
- Buffers to reorder threads to enable more parallelism, where serialization is not needed
- Fixed function hardware for killing hidden pixels as early as possible

Tuning hardware becomes very content dependent

- Complexity of the Geometry, Number of buffers used, the amounts of data load per thread
- Software and hardware evolve together over time

The API allows large and invasive hardware changes that are transparent to the application, due to the driver mediating between the two.

Mutual Adaptation Between GPUs and Applications

Both evolve over time

Better Hardware → Heavier Applications

More complex geometries and more complex computations.

- More triangles per object
- More objects
- More complex light effects
- More particle effects

Application Changes → Rebalanced GPUs

HW buffers, memory hierarchy and compute density must be balanced for the use-case.

- Larger buffers to enable more out-of order and more parallelism
- Higher compute performance
- Large caches and more complex memory hierarchy

arm

Index-Driven Position Shading

arm

IDVS – Introduction

Discarding triangles early

A vertex shader typically computes two kinds of things, a position and some data

- A coordinate transformation from the “model space” to the “real space”
- A coordinate transformation as objects move around
- Data transformations for, e.g., lighting calculations
- Often, the coordinate transformations are simple.
- Sometimes, we can discard triangles based only on the position data.
- Executing the data transformations is then useless work that we want to avoid.

IDVS – Introduction

Splitting the vertex shader

We want to split the vertex shader into two parts

- The original vertex shader transformed both position and data

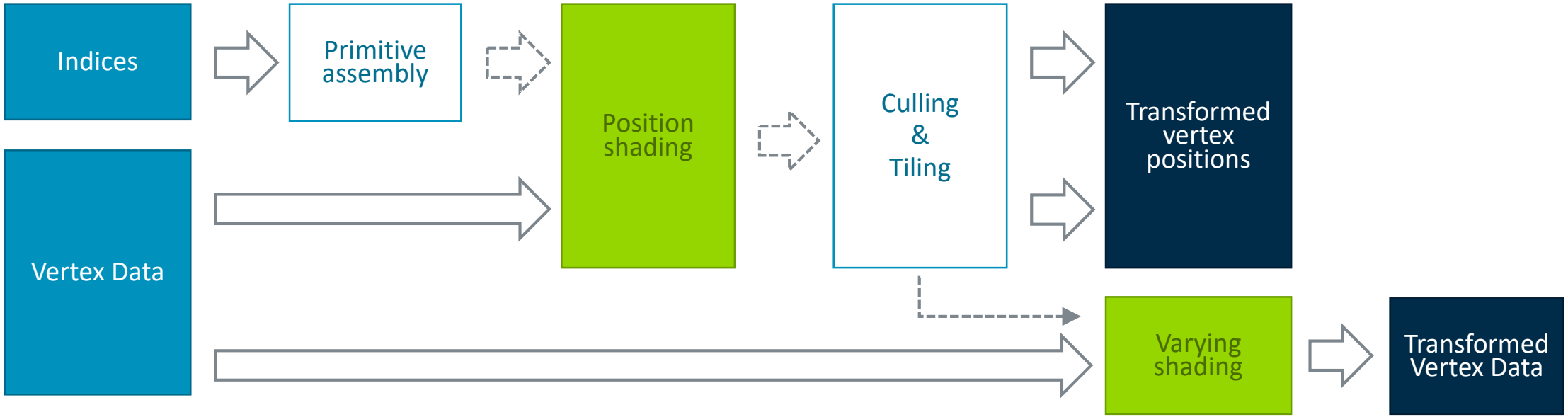
  ```
  out_position, out_data = vertex_shader(position, data)
  ```

- Split into two parts, computing one piece each

  ```
  out_position = position_shader(position, data)
  out_data = data_shader(out_position, data)
  ```

- Discard triangles between the two shaders
- Implementation
  - Let the compiler split the shader into two
  - Let the hardware cull triangles between position_shader and data_shader
  - Let the driver manage memory buffers to ensure correctness

IDVS: Hardware



arm

A Simple Example

Basic vertex shader

Input data in two parts, for position and data.

- A simple transform to position the object
- A separate computation for, e.g., lighting

```
def vertex_shader(position, data):
        transform, buffer = data
        out_position = transform(position)
        out_data = data_transform(out_position, buffer)
        return (out_position, out_data)
```

A Simple Example

Splits nicely into two parts

Input data in two parts, for position and other data.

- A simple transform to position the object
- A separate computation for, e.g., lighting
- Easily split into two parts

```
def vertex_shader(position, data):
        transform, buffer = data
        out_position = transform(position)
        out_data = data_transform(out_position, buffer)
        return (out_position, out_data)
```

A Simple Example

Splits nicely into two parts

Input data in two parts, for position and other data.

- A simple transform to position the object
- A separate computation for, e.g., lighting
- Easily split into two parts

```
def position_shader (position, data):
        transform, _ = data
        out_position = transform(position)
        return out_position

def data_shader(position, data):
        out_data = data_transform(out_position, data)
        return out_data
```

A More Complex Example

Transformation needed to compute data

Input data in two parts, for position and data.

- Complex dependencies on input data

```
def vertex_shader(position, data):
        transform_data = compute_transform(data)
        out_position = transform(transform_data, position)
        out_data = lighting_calculation(
                                        out_position,
                                        data,
                                        transform_data)
        return (out_position, out_data)
```

A More Complex Example

Common subexpression

Input data in two parts, for position and data.

- Complex dependencies on input data
- Common subexpressions for position and data computations

```
def vertex_shader(position, data):
        transform_data = compute_transform(data)
        out_position = transform(transform_data, position)
        out_data = lighting_calculation(
                                        out_position,
                                        data,
                                        transform_data)
        return (out_position, out_data)
```

A More Complex Example

Recomputing the same function twice

Input data in two parts, for position and data.

- Complex dependencies on input data
- Common subexpressions for position and data computations
- Naïve split requires computing transform twice

```
def vertex_shader(position, data):
        transform_data = compute_transform(data)
        out_position = transform(transform_data, position)
        return out_position

def data_shader(out_position, data):
        transform_data = compute_transform(data)
        out_data = lighting_calculation(
                                        out_position,
                                        data,
                                        transform_data)
        return out_data
```


arm

Directions for Solution

Intermediate results common for both position and data should not be recomputed

- Identify such intermediate results
- Store them to additional buffers
- Only split shaders when we expect a performance improvement

Needs active content aware hardware-software co-design to obtain good performance in general

Summary

- Graphics APIs give device designers a HW/SW co-design opportunity
- Understanding the use-case is important for successful HW/SW co-design
- HW/SW co-design is important for obtaining good performance
- Applications and GPUs evolve together

Thank You!
Danke!
Merci!
谢谢!
ありがとう!
Gracias!
Kiitos!
감사합니다
धन्यवाद