End to End: Part 4

Earl Wong

Subjects

- Optics
- Sensor
- ISP
- **GPU**
- NPU

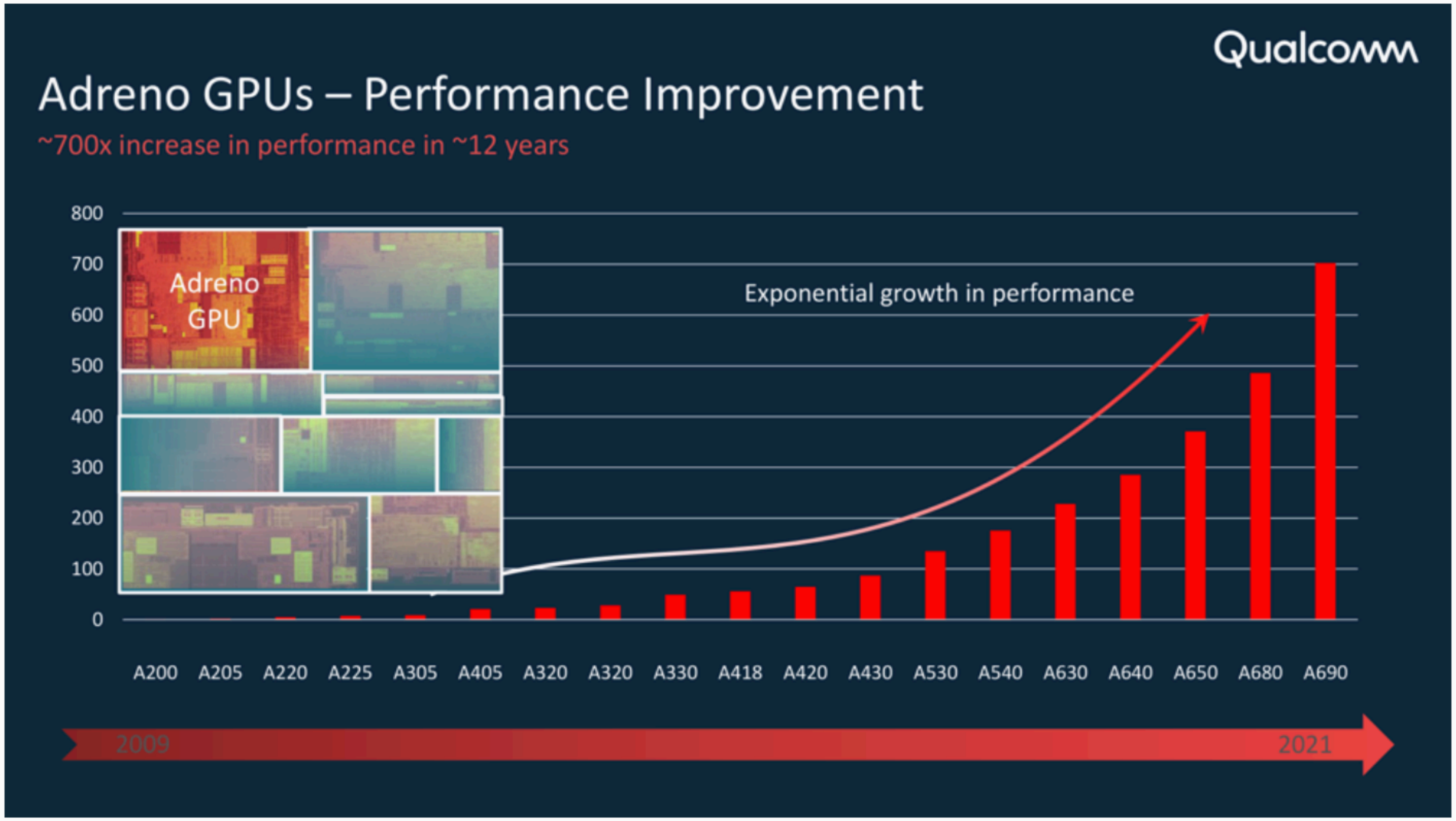
GPU

- The GPU on mobile devices can handle both image / video post processing and ML processing (**Transformers** and CNN based models).
- The NPU on mobile devices is highly optimized for ML processing - specifically, CNN based models.
- We will discuss the NPU in the next presentation.

GPU

- We will focus on the Qualcomm (QC) Adreno GPU, since Qualcomm is a major chip supplier for mobile devices.
- The Mediatek Mali GPU provides similar functionality for Mediatek mobile chipsets.
- Note: Micro level architectural descriptions for the QC Adreno GPU hardware do not exist.
- Hence, the Compute Unit (CU) and Shader Processor (SP) architectural diagrams are my “best guesses”, based on Snapdragon Profiler output fields, known approaches from other GPU manufacturers, etc.

GPU Evolution

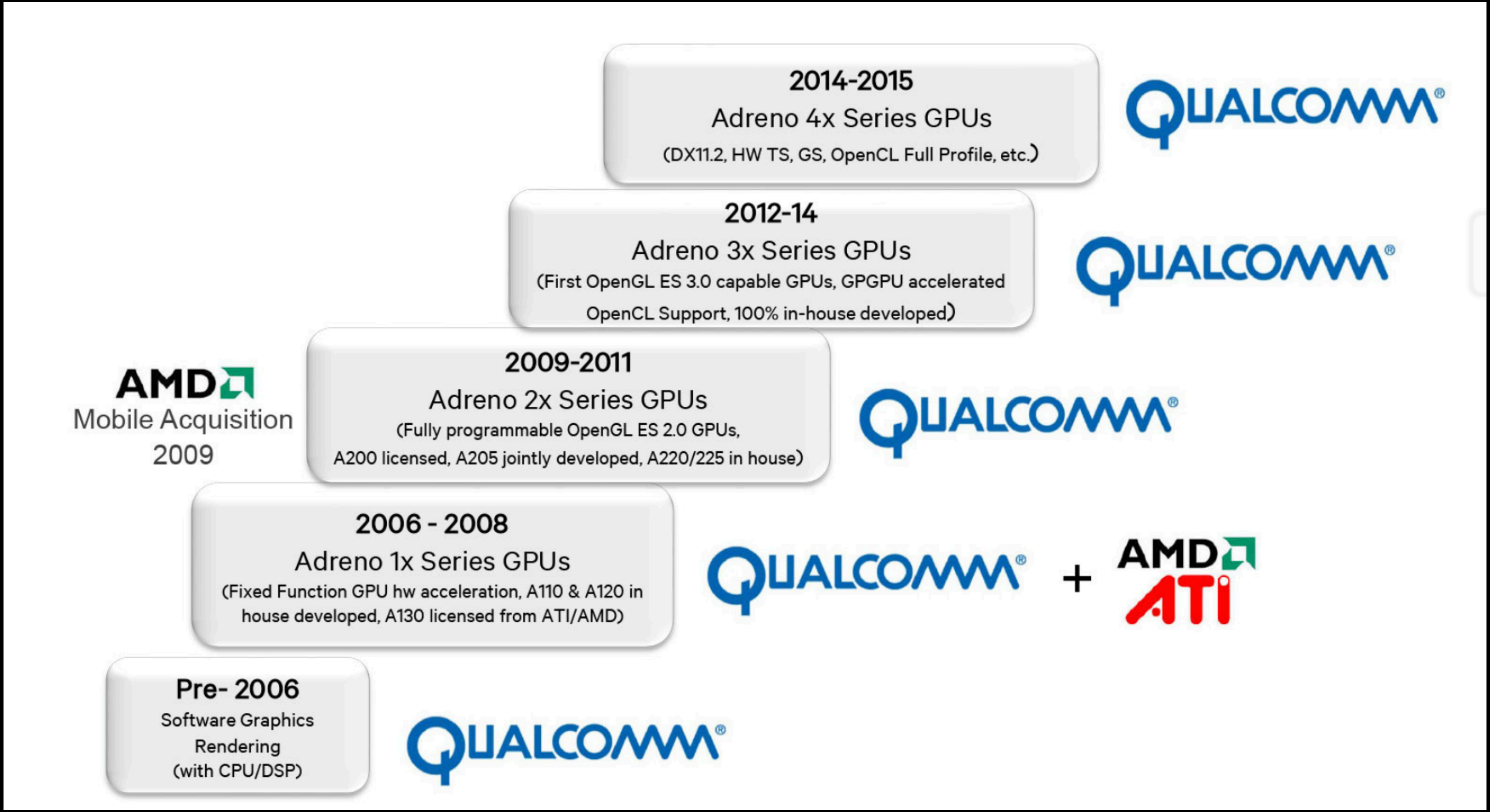


Graph showing Adreno's performance improvement over generations.

QUALCOMM

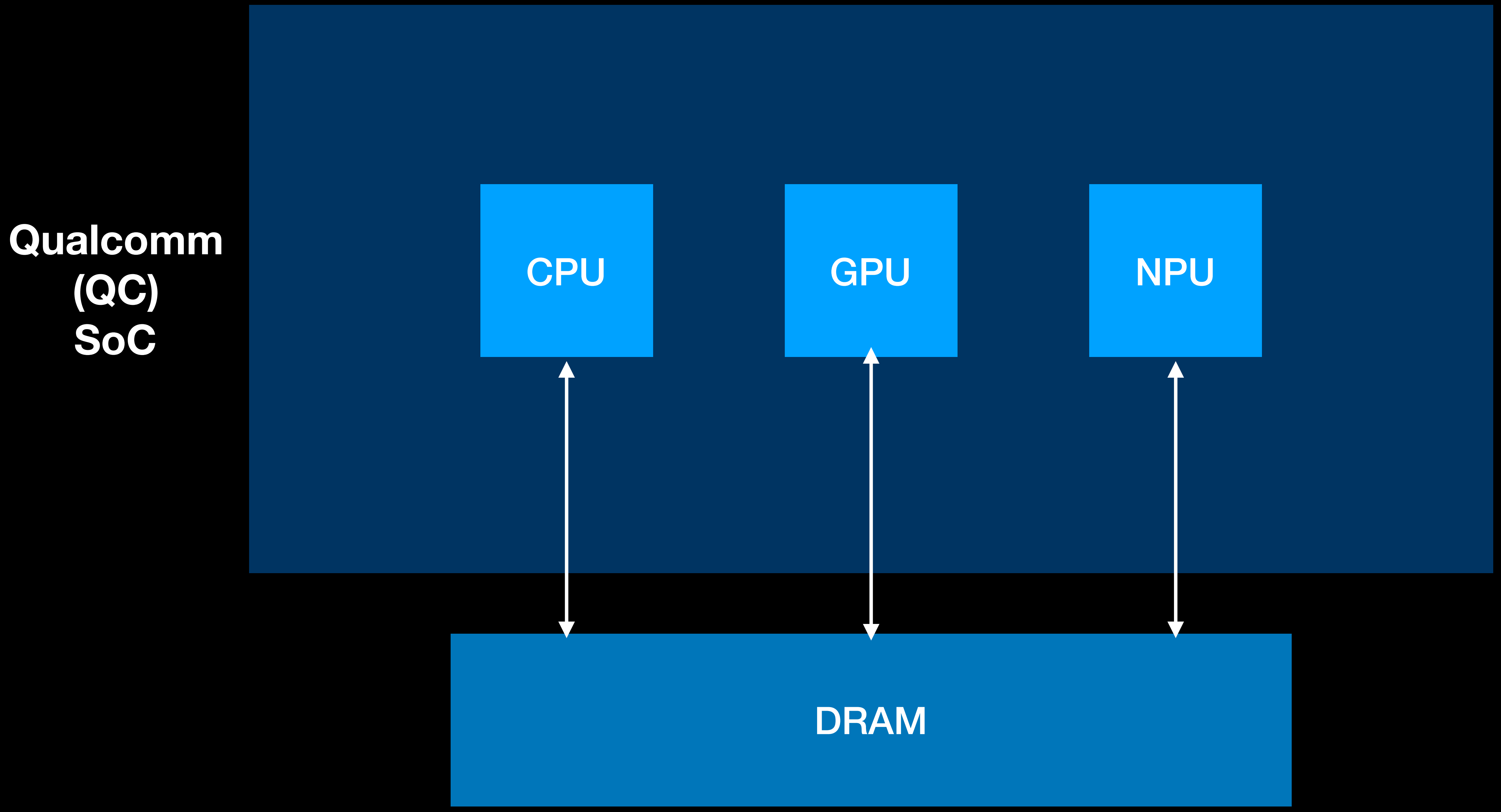
2024 to 2025 Flagship QC GPU - Adreno 800 Series

GPU - OpenGL support


2014-2015
Adreno 4x Series GPUs
(DX11.2, HW TS, GS, OpenCL Full Profile, etc.)
QUALCOMM
2012-14
Adreno 3x Series GPUs
(First OpenGL ES 3.0 capable GPUs, GPGPU accelerated OpenCL Support, 100% in-house developed)
QUALCOMM
AMD
Mobile Acquisition 2009
2009-2011
Adreno 2x Series GPUs
(Fully programmable OpenGL ES 2.0 GPUs, A200 licensed, A205 jointly developed, A220/225 in house)
QUALCOMM
2006 - 2008
Adreno 1x Series GPUs
(Fixed Function GPU hw acceleration, A110 & A120 in house developed, A130 licensed from ATI/AMD)
QUALCOMM + AMD ATI
Pre- 2006
Software Graphics Rendering (with CPU/DSP)
QUALCOMM

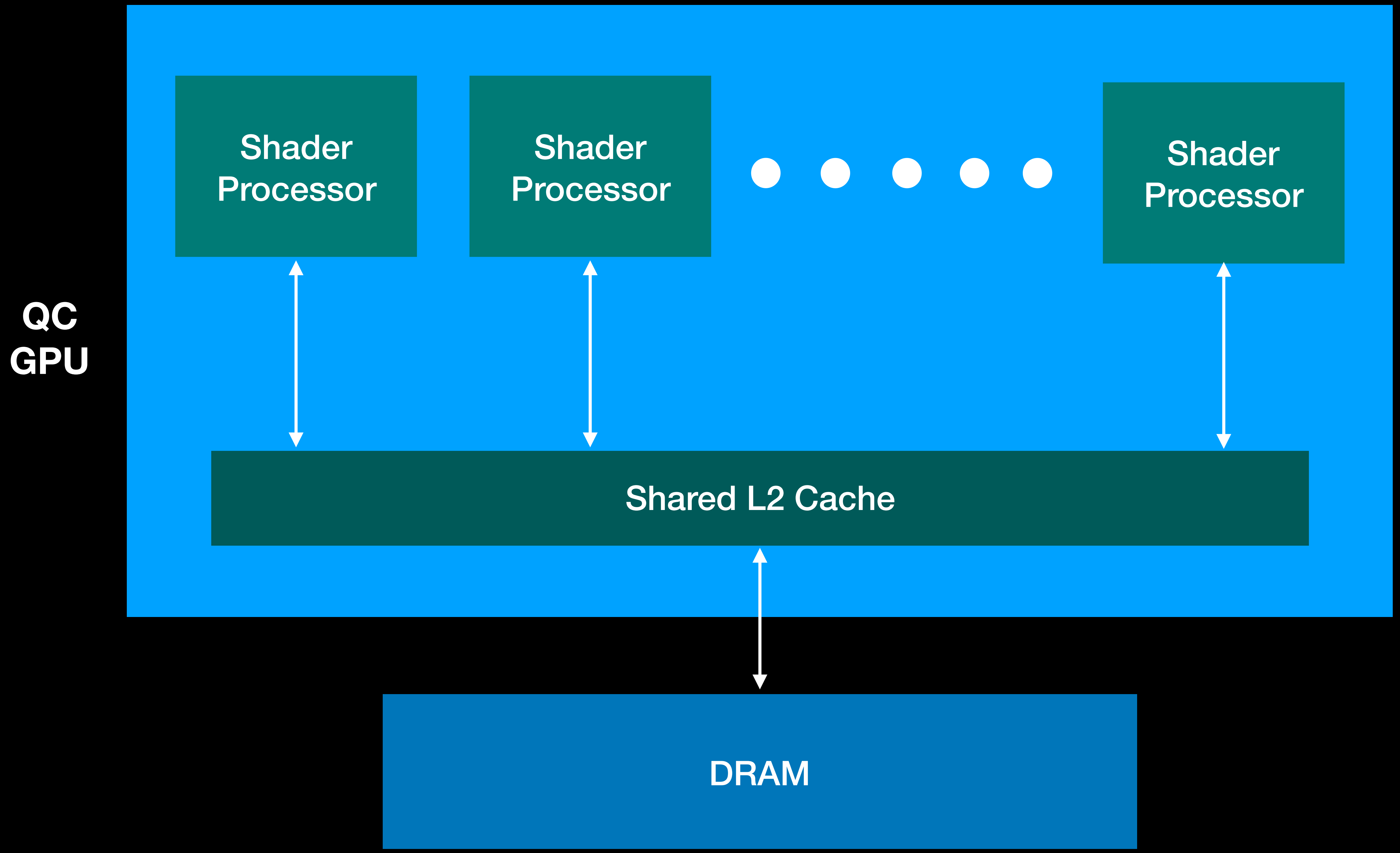
OpenGL ES 3.2: Full support in the 600 series to current (800 series)

SoC View

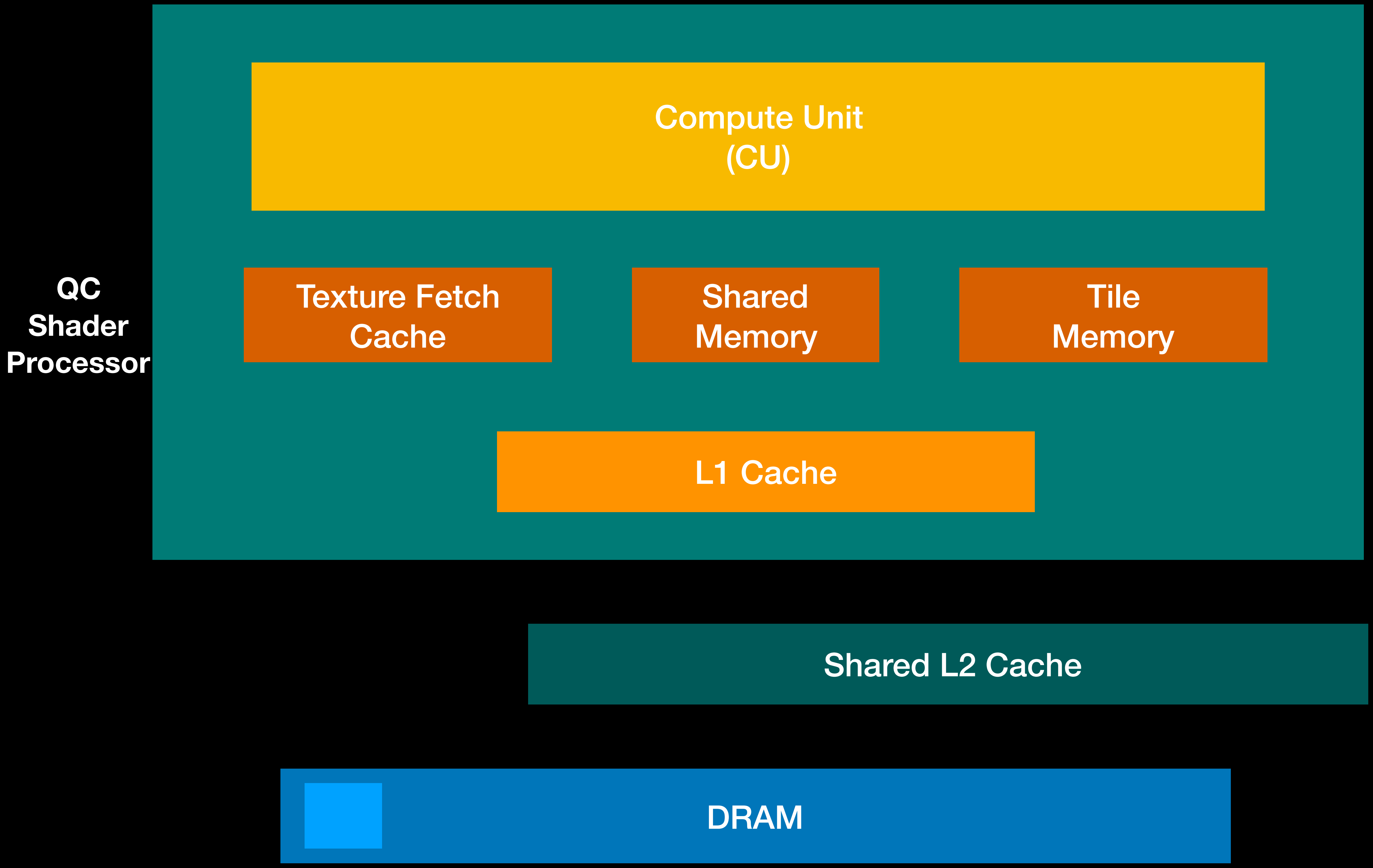


DRAM is LPDDR# - low power double data rate RAM for mobile devices.
[Clocking occurs on both the rising and falling edges.]

GPU View


Shader Processor
Shader Processor
Shader Processor
QC GPU
Shared L2 Cache
DRAM

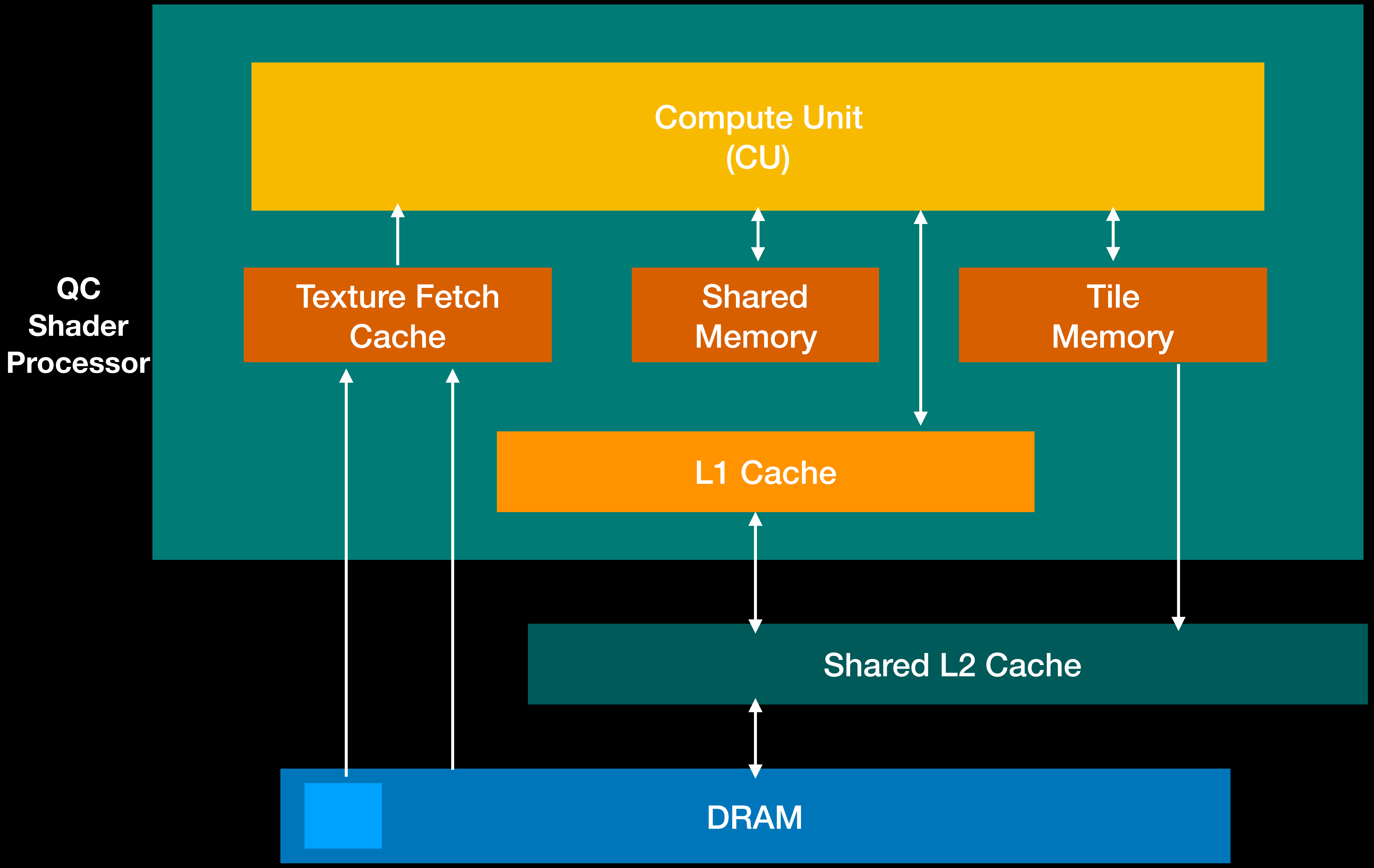
Shader Processor View



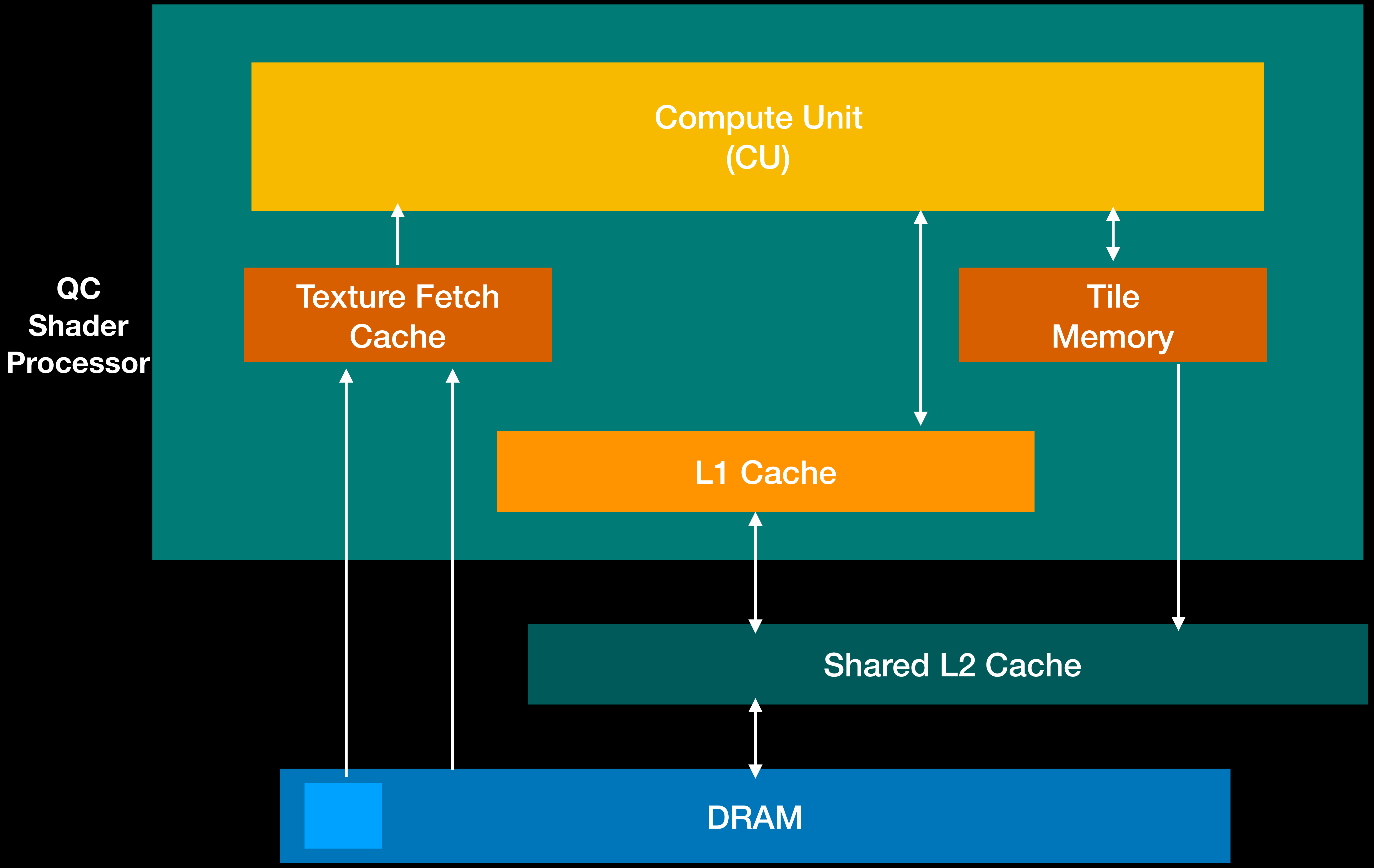
Shader Process View

- I will now add “the arrows”.
- Caveat emptor ….

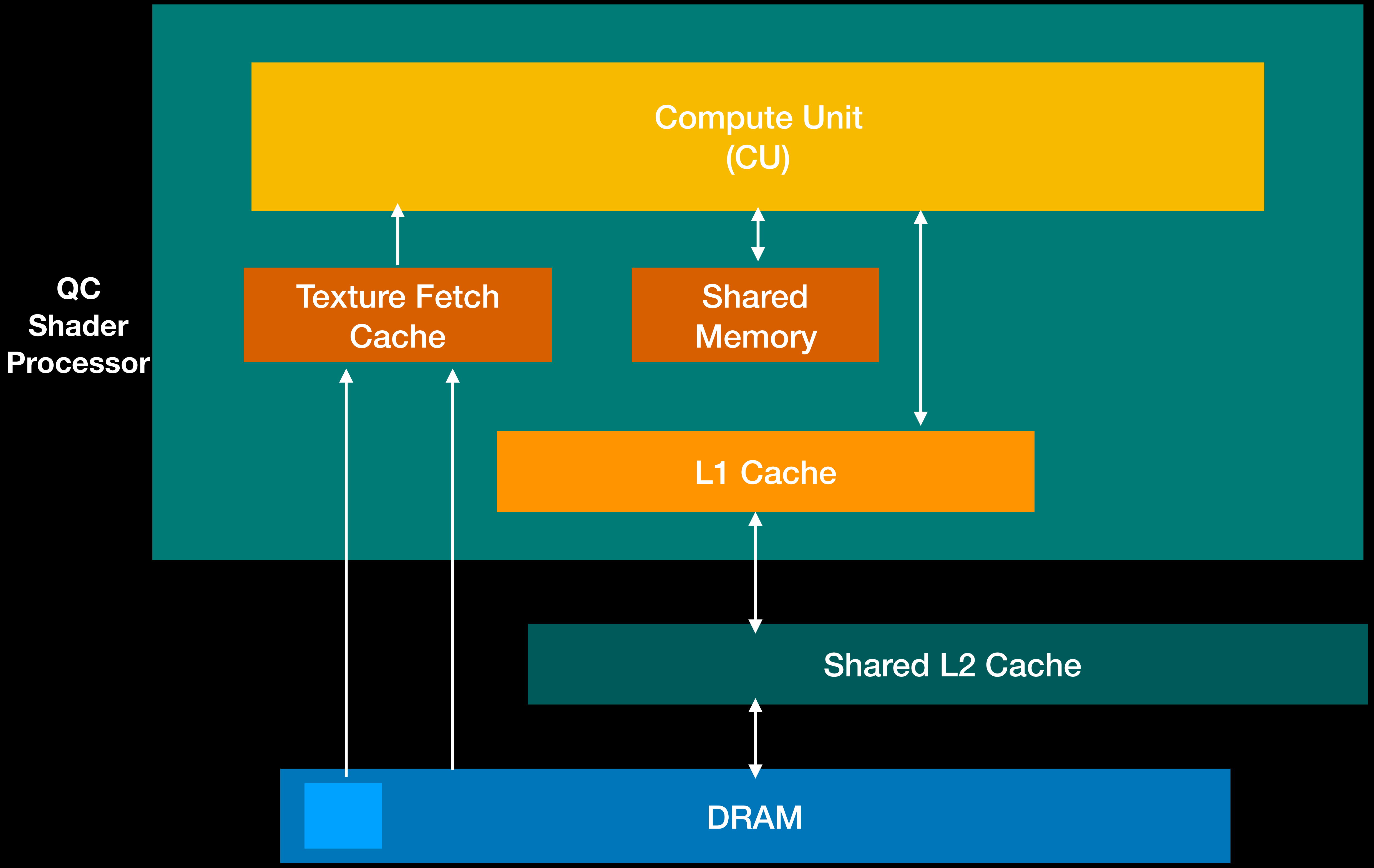
Shader Processor View



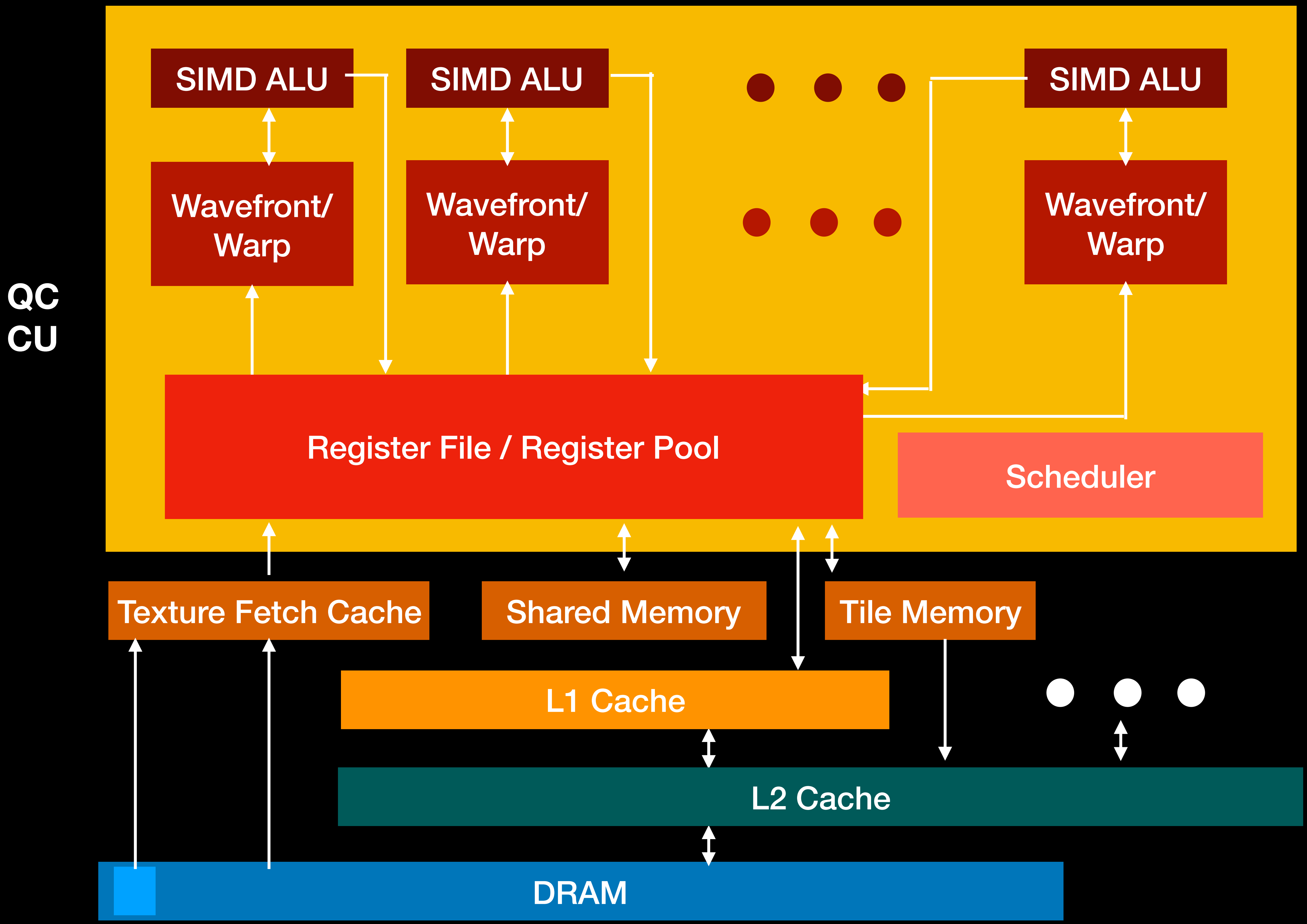
OpenGL Fragment Shader Path



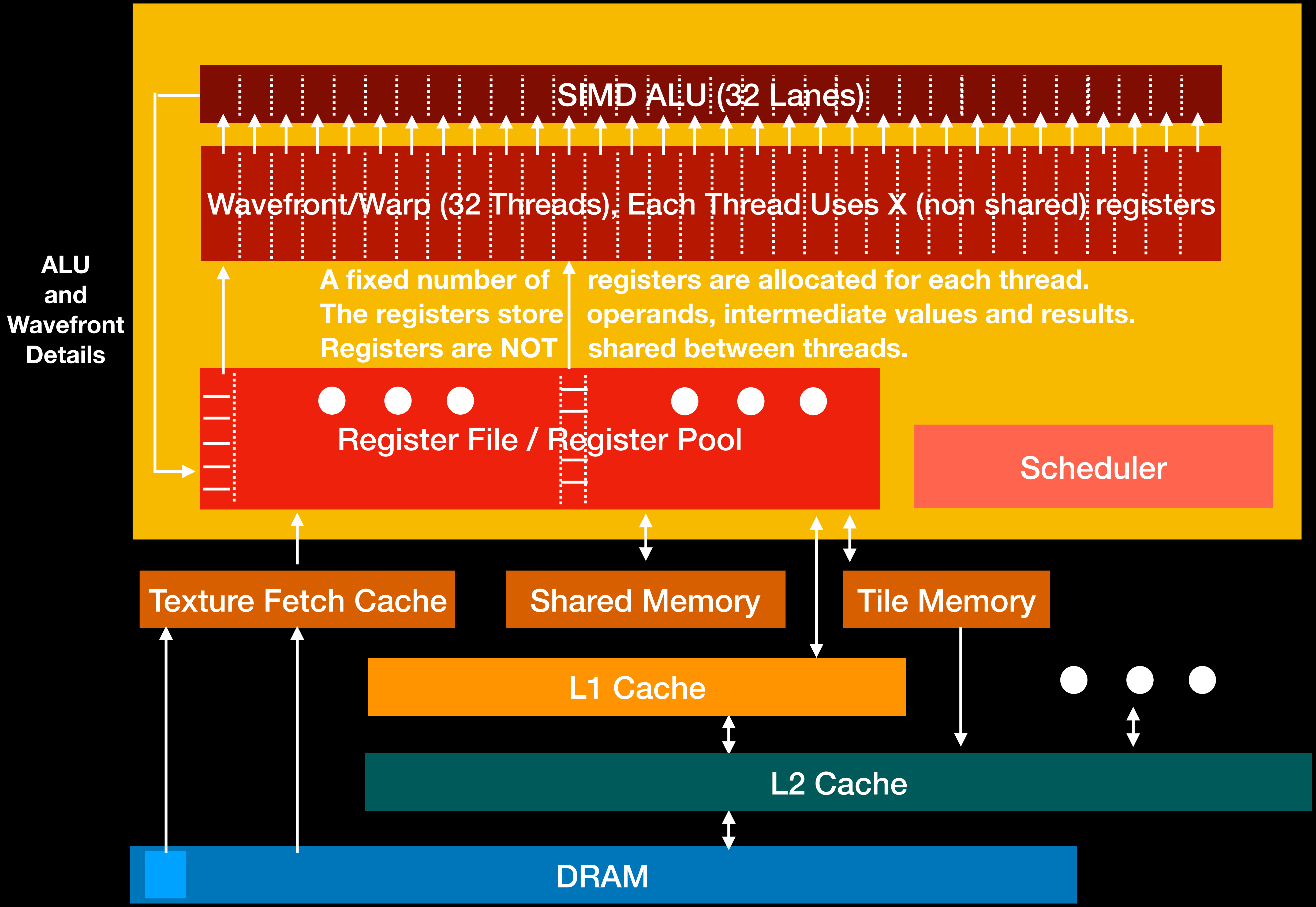
OpenGL Compute Shader Path



Compute Unit (CU) View



ALU and Wavefront Details


SIMD ALU (32 Lanes)
Wavefront/Warp (32 Threads), Each Thread Uses X (non shared) registers
ALU and Wavefront Details
Register File / Register Pool
Scheduler
Texture Fetch Cache
Shared Memory
Tile Memory
L1 Cache
L2 Cache
DRAM

A fixed number of registers are allocated for each thread.
The registers store operands, intermediate values and results.
Registers are NOT shared between threads.

The Details,
In Reverse Order

- CU
- Tile and Shared Memory
- Texture Cache

The CU Details

- The ALU contains 32 (or possibly 64) SIMD lanes.
- Each instruction is executed in SIMD fashion, applying the same instruction to all 32 (or 64) lanes simultaneously.
- During an instruction issue, each lane is assigned one thread from a wavefront.
- Each thread has its own private registers that store operands, intermediate values and results.
- The lanes act as temporary execution slots for threads and are reused across different wavefronts over time.

The CU Flow

- If a thread in a wavefront **“is not ready for processing” - the entire wavefront is evicted from the ALU, and a different resident wavefront is substituted in it’s place.
- Occupancy describes the availability of substitute resident wavefronts.
- Occupancy determines the effectiveness of latency hiding (which will be described shortly).
- **Reasons for not being ready: waiting on a texture fetch, waiting on a memory fetch, waiting on a specialized function unit result (e.g. sin, exp), waiting on a dependency from a prior instruction.

Tile and Shared Memory

- Tile memory is used by fragment shaders, while shared memory is used by compute shaders.
- ***Tile memory is hardware driven, and exists “under the hood”.
- Often, programmers / developers are not aware of its existence.
- Shared memory is configurable (via software) by the programmer / developer.
- Both tile memory and shared memory can be read from and written to.

A Tile Memory Example

- Tile size [GMEM]: 32x32
- Location (depth) [Max Color Attachments]: 8
- Each pixel and location is associated with vec4 [4 Bytes for Int8, 8 Bytes for FP16, 16 Bytes for FP32]
- 32x32x8x4: ~32 KB
- If **total tile memory per SP = 32 KB** => 1 tile can exist.
- If we specify FP16, **<u>the hardware will automatically reduce the tile size</u>, to say, 16x16.

Tile Memory

- Tile memory is very close to the CU registers.
- As a result, tile memory is extremely fast.
- Goal: When using a fragment shader, you want to perform all of your compute in an “ALU-register-tile memory” loop.
- Upon completion, tile memory is then written to DRAM.
- The latter minimizes system bandwidth and power usage.

A Shared Memory Example

- Filter size: 21x21
- Workgroup size (threads): 32x32
- Memory per pixel: 4 Bytes
- => Configure shared memory tile size as:
  (32 + 21 - 1)x(32 + 21 - 1) = 52x52
- 52x52x4 = ~10 KB
- If total shared memory = 64 KB => you can fit 6 workgroups into shared memory.

Shared Memory

- Shared memory is configured by the user.
- The user specifies the workgroup size and the tile size.
- The workgroup size represents the number of threads associated with the tile.
- Each thread has access to all of the data in the tile.
- For latency hiding reasons, shared memory should be configured so that multiple workgroups result.

Shared Memory

- In the current example, 6 different workgroups can be realized / running in parallel, using shared memory.
- These workgroups are independent of each other, producing independent wavefronts that are amenable for latency hiding.

Tile Memory vs Shared Memory

In OpenGL, this translates to:

- When should you use a fragment shader vs a compute shader?

Fragment Shader vs Compute Shader

- Q: If I wanted to implement XYZ on the GPU, should I use a fragment shader or a compute shader?

- A: What is XYZ more amenable to?

Fragment Shader vs Compute Shader

- Q: Can XYZ be easily implemented using either shader?
- A: If no, the compute shader will be the shader of choice.
- The compute shader was introduced, to address the implementation needs of various tasks, not amenable using a fragment shader.
- Fragment shaders are targeted for a graphics pipeline.
- Compute shaders turn the GPU from a graphics-centric processor into an explicitly programmable general-purpose parallel processor.

Shader History - A Brief Segue

- Graphics were originally performed on the CPU.
- However, this proved to be unacceptably slow, for many rendering tasks.
- The GPU (specialized graphics hardware) was introduced, providing parallel compute / vectorization capabilities.
- OpenGL was originally introduced to provide an API to abstract graphics hardware for a fixed graphics pipeline.
- Later, OpenGL added vertex and fragment shaders, to allow a programmable graphics pipeline.

Shader History - A Brief Segue

- The image / video processing community recognized the compute power of the GPU, and adopted OpenGL’s fragment shader for their own needs.
- The fragment shader did an excellent job at various per pixel, or, small neighborhood image / video based tasks.
- However, fragment shaders were restrictive: algorithms requiring large spatial neighborhoods or arbitrary memory access were difficult to implement efficiently.

Shader History - A Brief Segue

- Other domains also recognized the GPU's compute potential, and faced similar constraints.
- This led to the introduction of the compute shader, and enabled enabled general-purpose GPU computing.
- The compute shader allowed flexible memory access, workgroup cooperation, and arbitrary thread organization.

Fragment Shader vs Compute Shader

- Assume XYZ be easily implemented using either shader.
- Q: Which shader do we now choose?
- A: When using a fragment shader, does the shader need to frequently (and repeatedly) fetch identical data from DRAM?
- A: Is the overhead for using a compute shader exceedingly high for the task at hand?

Fragment Shader vs Compute Shader

An additional consideration:

- When writing to DRAM, fragment shaders perform a single “burst” tile write (32x32), after all 1024 results are available.
- In contrast, when writing to DRAM, compute shaders perform 1024 individual writes - when each result becomes available.
- => Compute shaders use more power than fragment shaders.

Classic Example A

- You are performing an operation using a 21x21 filter kernel - say averaging.
- With a fragment shader, each thread will require its own 21x21 copy of the data.
- Since there is no such thing as shared memory, the SoC hardware quickly gets overwhelmed.

Classic Example A

- 1) The same data is re-fetched from DRAM numerous times, increasing system bandwidth pressure.
- 2) To address the slowness of the DRAM fetches, the compiler pre-fetches the data by unrolling the averaging / summation loop - increasing register pressure.
- a0) If there are an insufficient number of registers, L1 cache spillover occurs.
- a1) If the spillover is large enough, L1 cache thrashing occurs.
- a2) Because of the increased register pressure, occupancy drops, reducing the effectiveness of latency hiding.

Classic Example A

- With a compute shader, data is shared amongst the threads in a workgroup.
- 1) Bandwidth pressure is reduced, since each thread does not need an individual copy of the data.
- 2) Loop unrolling does not occur, because pre-fetching is not needed.
- i.e The data is nearby (1-2 cycles vs 100’s of cycles).
- 3) Register usage remains low, increasing occupancy and the effectiveness of latency hiding.

Classic Example A

- The Compute Shader is the clear winner.

Classic Example B

- You are performing an operation using a 3x3 filter kernel - say averaging.
- With a fragment shader, each thread will require its own 3x3 copy of the data.
- Although there is no such thing as shared memory, the SoC hardware is sufficient for the task.

Classic Example B

- 1) System bandwidth pressure is greatly reduced, due to the reduced kernel size.
- 2) Loop unrolling is not needed, because the data can be found in the L1 cache lines.
- => Occupancy is high, and latency hiding is effective.

Classic Example B

- With a compute shader, data is shared amongst the threads in a workgroup.
- The bandwidth pressure advantage is now greatly reduced.
- Occupancy now is similar to the fragment shader, providing no latency hiding advantage.
- **But**, the overhead for shared memory management now becomes significant.
- **But**, the time needed for thread synchronization now becomes significant.

Classic Example B

- The Fragment Shader is now the clear winner, since it does not have the associated overhead of shared memory.

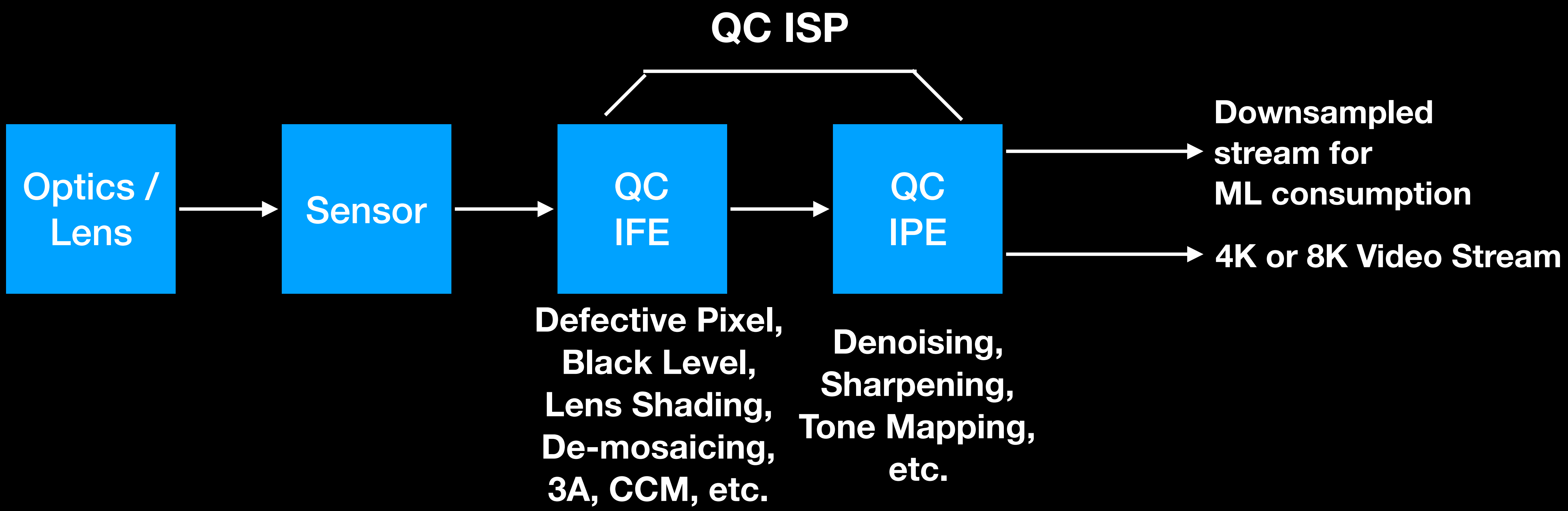
Classic Example C

- At some point in-between 3x3 and 21x21, a crossover point exists.

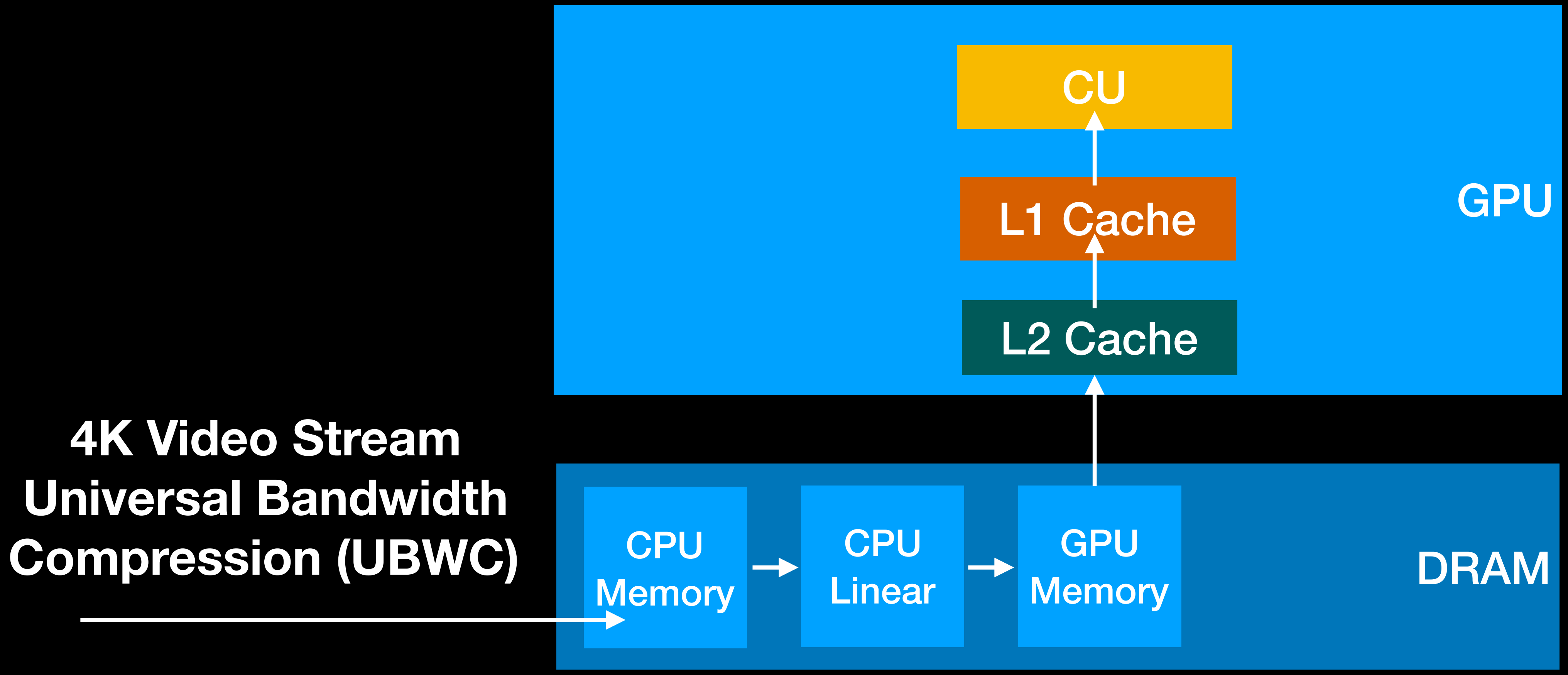
Texture Cache - Data Fetching

- There are different data fetching paths for the Adreno GPU.
- Traditional (non-native graphics buffer)
- Fast (native graphics buffer)
- We will now illustrate how data migrates from the ISP, using these two paths.

Signal / Video Stream


QC ISP
Optics / Lens
Sensor
QC IFE
QC IPE
Downsampled stream for ML consumption
4K or 8K Video Stream
Defective Pixel, Black Level, Lens Shading, De-mosaicing, 3A, CCM, etc.
Denoising, Sharpening, Tone Mapping, etc.

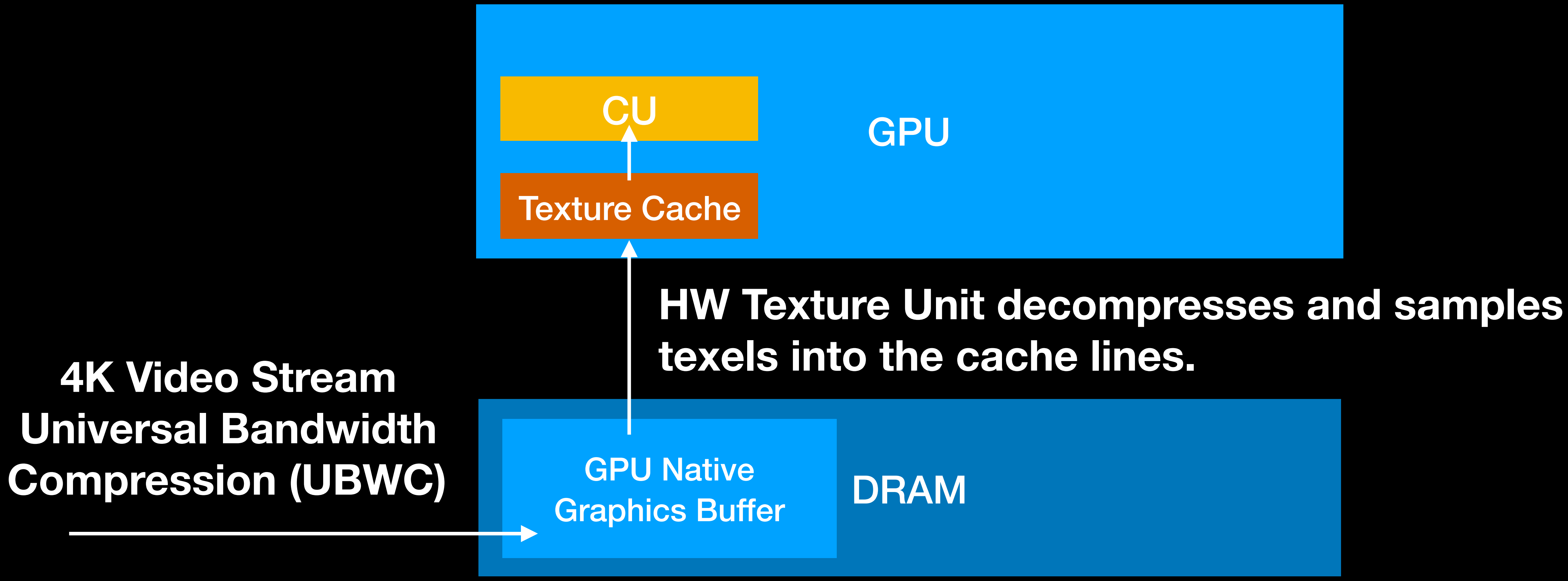
Video Stream (Traditional): Non Surface Path



Multiple copies occur.

1) **The UBWC tiles are written to cpu memory.**
2) **The tiles are decompressed, and written to standard linear format.**
3) **The linear format data is written to GPU specific memory.**
4) **The data migrates to the CU through the L2 and L1 caches.**

Video Stream (Fast): Surface Path


CU
Texture Cache
GPU
HW Texture Unit decompresses and samples texels into the cache lines.
4K Video Stream Universal Bandwidth Compression (UBWC)
GPU Native Graphics Buffer
DRAM

1) A surface / surface texture is defined, to create the native graphics buffer.
2) A hardware texture unit (TU) samples the data directly into the texture cache by filling the 64 Byte cache lines.

No copy / memcpy is involved.

Profiling

- We now have a solid understanding of the Adreno GPU.
- This understanding provides a solid foundation to address the next issue: power and performance.
- Specifically, what are the key idea(s) for optimizing / maximizing this criteria?

Profiling

- The ALU is the fundamental compute engine in the GPU.
- The objective is to fully utilize all of the ALU’s.
- However, this may not be occurring because the ALU’s are stalled, waiting for data.

Profiling

Where does the data need to reside?

- Registers

Where could the data reside?

- Texture Cache
- Tile Memory, Shared Memory
- L1 Cache
- L2 Cache
- DRAM

Profiling

- How long does it take to move the data into the registers? (Ballpark / back of envelope approximations)
- From Texture Cache: (let’s assume) 1 clock cycle
- From Tile or Shared Memory: ~1-2 clock cycles
- From L1 Cache: ~2-3 clock cycles
- From L2 Cache: ~10’s clock cycles
- From DRAM: ~100’s clock cycles

Profiling

- Hence, it pays to have the needed data, as close to the registers as possible.

- The search for the data begins locally and moves outward - texture fetch cache, shared memory, L1, L2, and finally, DRAM.

- Profiling parameters like ALU utilization, texture fetch stalls, L1 hit rate, etc. provide valuable insight into potential bottlenecks.

Profiling

- How large are the different memory units?
- Memory units are kept small on mobile devices, to reduce power consumption.
- The scaling might look something like:

Profiling

- Register: 16 B
- Register File (Main reason occupancy is limited): 64 KB
- Cache Line: 64 B
- Texture Cache (Read Only): 8 KB
- Tile memory: 32-64 KB, Shared Memory: 64 KB
- L1 Cache: 16-64 KB
- L2 Cache: 512 KB-8 MB
- DRAM: 8 GB

Profiling

- Understanding memory usage is important.
- Memory becomes a problem, when there is “not enough” where it is needed.
- Classical Example A (fragment shader) provides one such example.

Two Design Alternatives

- For a fixed and known workflow, one could scope the problem, and construct the optimal configuration to ensure that the ALU’s have a continuous stream of usable data.
- If one needed a robust workflow that supported a variety of different use case scenarios, the latency hiding trick can serve as an alternative.

Latency Hiding Trick

- The ALU SIMD is fed by a wavefront / wave consisting of 32 (or 64) threads with associated registers and data.
- Suppose there was a thread in the wavefront that did not have the needed data, in one (or several) of it’s registers.
- If this was the only wavefront, the ALU would need to wait, until the data arrived. BAD!
- Suppose there was another wavefront that was “ready to go”.
- The “ready to go” wavefront could (then) be swapped in, resulting in little ALU downtime = LATENCY HIDING!

Latency Hiding Trick

- Ideally, if there are M ALU’s, we would like to have N wavefronts, where N is significantly greater than M.
- That way, we could greatly exploit latency hiding.
- However, the number available wavefronts depends on the chip resources - such as the number of total registers.

Latency Hiding Trick

- i.e. The register file / register pool is finite.
- If each thread uses a large number of registers, fewer wavefronts would be possible, negating the latency hiding trick.
- Also, when using a compute shader, the shared memory should be configured to encourage multiple workgroups.
- This is because the threads in each workgroup will (then) be independent of the threads in a different workgroup.

In The Limit

- Suppose that a single wavefront consisting of 32 (or 64) threads, used up the entire register pool.
- Then, latency hiding would be impossible, because there would be no other wavefront to “swap in”.
- In profiling jargon, occupancy is low / near zero.
- Ideally, you want to have a lot of available wavefronts that are ready to go / ready to “swap in” = high occupancy.

In The Limit

- Suppose that the shared memory in your compute shader was configured so that a single tile consumed all of the shared memory.

- Although you now have a large workgroup with many threads resulting in many resident wavefronts, the resident wavefronts are not independent.

- As a result, latency hiding is ineffective, because a problem existing in one wavefront will be replicated in every other wavefront.

Free Gains To Reduce Hardware “Pressure"

- Tried and true tricks can be applied to ameliorate issues that prevent latency hiding - assuming the output quality remains acceptable.

The underlying theme: process less data

- 1) Downsample, process the reduced data, upsample
- 2) Decrease the precision, where possible: FP32 -> FP16 -> INT8

Both allow the creation of additional independent wavefronts, increasing the effectiveness of latency hiding.

Summary

- The goal is to fully utilize the ALU’s on a GPU.
- However, this can only be achieved, if the ALU’s are fed a continuous stream of data.

At this point, there are 2 paths.

- 1) Optimize the pipeline for the given use case, to make this statement as true as possible.
- 2) Employ the latency hiding trick.

Summary

- Fragment shaders and compute shaders both have utility for different cases.
- Understanding the strengths and weakness of each, arms you with the necessary knowledge to make the right choice.