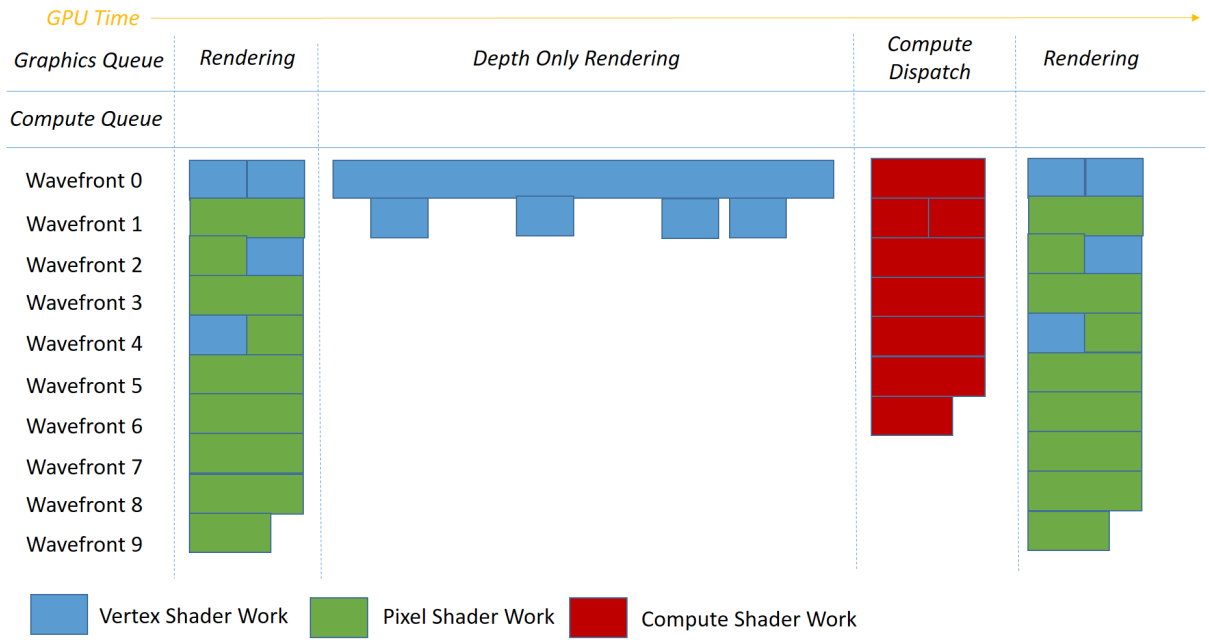
Async Compute Support (2017.3)

Async compute is a feature allowing for more efficient use of GPU resources when performing compute shader tasks. It is aimed at expert users who are issuing custom compute shader dispatches and is **supported on PS4 only**.

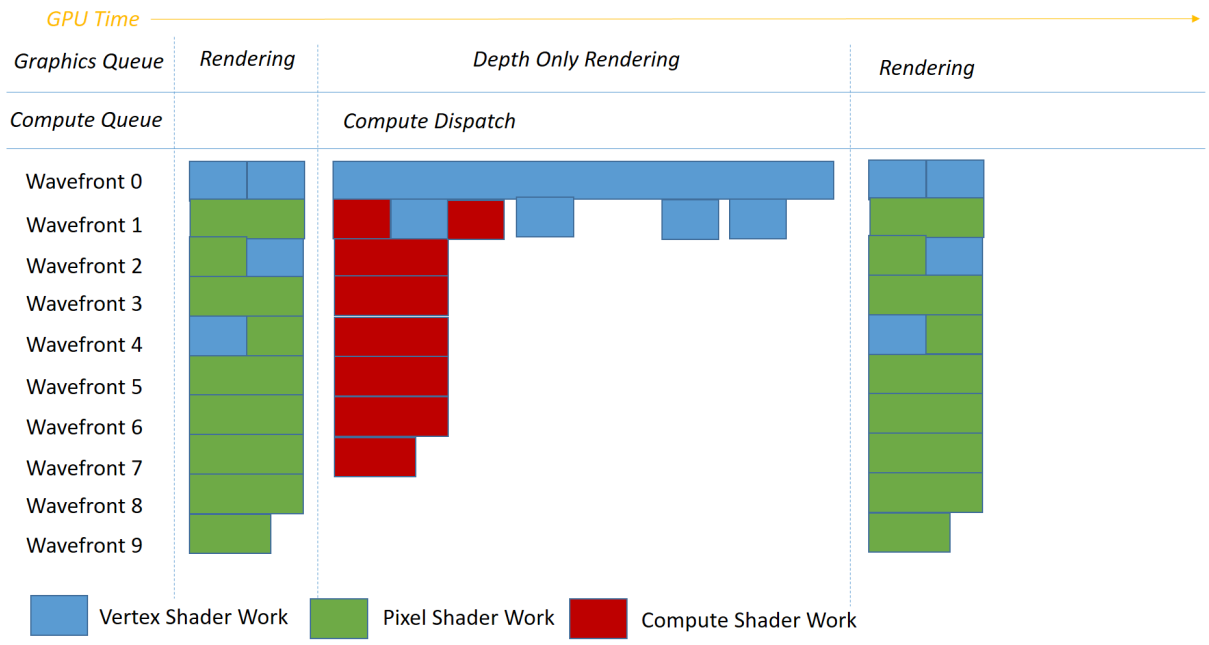
Update: As of Unity 2018.3, async compute is also supported on **Xbox One**.

Getting the most out of the GPU

Typically during a modern rendering pipeline there will be times where the full capacity of the GPU is not utilised. Getting the highest utilization possible from the available compute units can be a challenge but can also be a key factor in obtaining acceptable GPU performance from complex scenes. A typical lull in the utilisation of the compute units occurs during depth only rendering, such as shadow map creation or depth prepasses. During these stages of the rendering pipeline there is often no pixel shader bound, vertex shader shader wavefronts cannot be created fast enough to fully make use of all those available, and consequently the computational power of the GPU is under used. If we imagine a GPU where we have ten wavefronts available per SIMD, then the wavefront distribution on a single SIMD unit in this scenario could be loosely represented by the diagram below. Here we see the GPU performing some vertex and pixel shader rendering, then switching to perform depth only work, then performing some compute work, and finally switching back to vertex and pixel shader draws.



Async compute allows us to schedule compute shader work on **compute queues** to run simultaneously with tasks being scheduled from the **graphics queue** to utilise GPU resources being under used by the rendering process**.** In the case described above custom compute shader dispatches could be specifically scheduled on one or more compute queues to coincide with the depth only rendering on the graphics queue. These compute shader dispatches would make good use of the computational resources of the GPU which are being neglected during the depth only pass. The diagram below shows the scenario this time with async compute applied:



Comparing the two diagrams there are a few key points to observe:

- With async compute in use the overall time to complete all of the work is reduced constituting a performance gain.
- With async compute in use the time taken to complete only the compute processing is actually longer than if it was dispatched on the graphics queue.
- Moving the compute work to coincide with either of the two vertex and pixel shader enabled draw operations may not have been an optimisation as all wave fronts were in use during these items. However pairing async compute operations that have different bottlenecks to the tasks running on the graphics queue (e.g. ALU vs bandwidth) can still result in performance wins in these circumstances. A degree of trial and error may be needed when trying to determine where async compute operations can overlap with graphics queue tasks and still yield performance gains.

Using Async Compute

Unity's async compute scripting interface is best utilised as part of a Scriptable Render Pipeline as this provides the most flexibility for scheduling when async compute work will occur relative to the graphics queue. However, async compute can still be used in projects that are not using Scriptable Render Pipelines.

Work is issued to the async compute queues using the Rendering Command Buffer interface. Construct a command buffer containing only compute queue compatible work, then submit it using *Graphics.ExecuteCommandBufferAsync* or *ScriptableRenderContext.ExecuteCommandBufferAsync*, all of the commands within the submitted buffer will be executed on the same compute queue.

The following commands are valid within command buffers for async compute execution. Attempting to execute command buffers asynchronously that contain any command not included below will generate errors visible in the editor console and in the log at runtime.

- *CopyCounterValue*
- *CopyTexture*
- *CreateGPUFence*
- *DispatchCompute*
- Any of the *SetCompute...Param* commands
- *WaitOnGPUFence*

Async execution of command buffers on platforms that do not support async compute will result in the work being submitted to the graphics queue. Support for async compute on a given platform can be checked using *SystemInfo.supportsAsyncCompute.*

Scheduling relative to graphics tasks is accomplished using *GPUFences*. GPU Fences can be created using *Graphics.CreateGPUFence* or *CommandBuffer.CreateGPUFence*. Fences created in this fashion will be passed when the last clear, draw, dispatch or copy operation issued by unity's graphics processing prior to the processing of the creation of the fence has completed on the GPU. This may have been as a result of graphics related command issued from user script or from unity's own internal processing of rendering the scene.

Graphics.WaitOnGPUFence or *CommandBuffer.WaitOnGPUFence* can be used to have either the GPU's graphics queue or an async compute queue wait for a given fence to have passed before proceeding. Note neither of these functions will stall the CPU. As described in the above example, considering when async compute work should start relative to the work being done on the graphics queue is important for ensuring optimal use of GPU resources. Consequently, most command buffers designed for async compute use will have *WaitOnGPUFence* as their first

entry. Executing command buffers asynchronously that do not wait on a fence created from the graphics queue will be executed on the GPU at an indeterminate point during that frames GPU processing possibly occurring before any graphics queue work has commenced.