WHITE PAPER

ASYNCHRONOUS SHADERS

UNLOCKING THE FULL POTENTIAL OF THE GPU

INTRODUCTION

GPU technology is constantly evolving to deliver more performance with lower cost and lower power consumption. Transistor scaling and Moore's Law have helped drive this evolution for many years. Modern GPUs are very efficient, but software and hardware limitations have been keeping them from reaching their full potential.

Even during intensive rendering operations, a significant proportion of a GPU's processing capabilities are generally sitting idle. This paradoxical behavior presents a great opportunity to do even better through architectural improvements, and Asynchronous Shader technology represents the next important step in that process.

EXPLOITING PARALLELISM

Graphics has been referred to as an "embarrassingly parallel" problem, since it involves processing millions of polygons and pixels to create moving images. The faster these elements can be processed, the higher the quality of the resulting output. This characteristic is very amenable to specialized hardware that can handle many operations in parallel, which is why GPUs have become so critical in modern computing.

However, submitting all of that work to the GPU and scheduling it efficiently on the available processing resources is still a complex problem. GPUs rely heavily on pipelining to achieve their performance, and bottlenecks at any stage of the pipeline can leave other stages under-utilized. These stages include command processing, geometry processing, rasterization, shading, and raster operations. Each stage can make extensive use of parallel processing, but the throughput of each stage is also dependent on other stages that come before and/or after it.

Applications submit work to the GPU in the form of command buffers or command lists, which consist of a series of commands that execute on a set of data in parallel. Each command buffer is associated with a particular rendering task (shadow mapping, lighting, physics simulation, etc.). Ideally we want the GPU to be able to handle multiple tasks simultaneously, so they can share the available resources and improve utilization. Accomplishing this means frame rates can be improved and latency reduced without requiring more processing power. In practice though, there are many challenges that must be overcome to make this model work well.

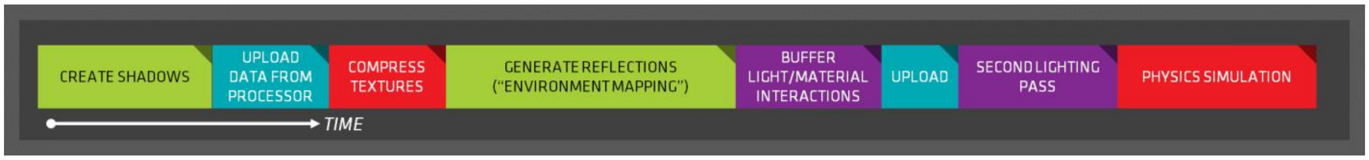

CREATE SHADOWS
UPLOAD DATA FROM PROCESSOR
COMPRESS TEXTURES
GENERATE REFLECTIONS ("ENVIRONMENT MAPPING")
BUFFER LIGHT/MATERIAL INTERACTIONS
UPLOAD
SECOND LIGHTING PASS
PHYSICS SIMULATION
TIME

Figure 1: A typical pipeline of graphics rendering tasks

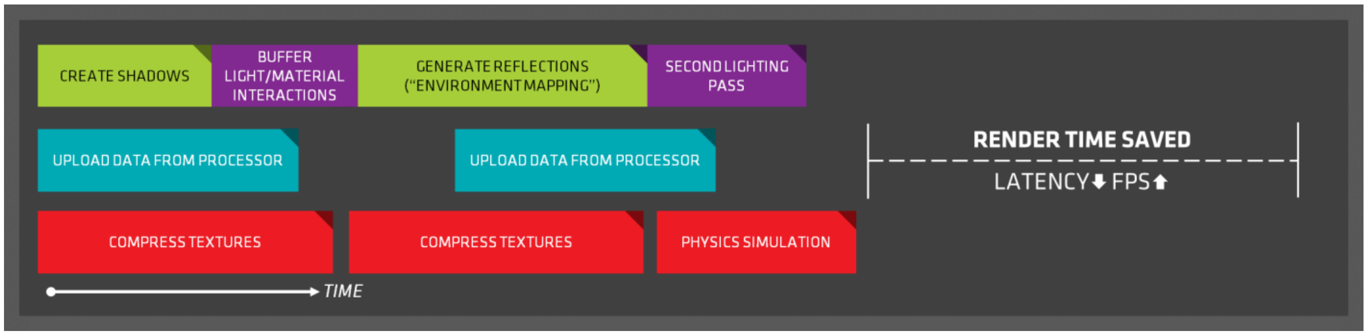

CREATE SHADOWS
BUFFER LIGHT/MATERIAL INTERACTIONS
GENERATE REFLECTIONS ("ENVIRONMENT MAPPING")
SECOND LIGHTING PASS
UPLOAD DATA FROM PROCESSOR
UPLOAD DATA FROM PROCESSOR
RENDER TIME SAVED
LATENCY⬇ FPS⬆
COMPRESS TEXTURES
COMPRESS TEXTURES
PHYSICS SIMULATION
TIME

Figure 2: Accelerated rendering pipeline making use of multiple queues executed in parallel

ASYNCHRONOUS COMPUTING

For many tasks in the graphics rendering pipeline, the GPU needs to know about ordering; that is, it requires information about which tasks must be executed in sequence (synchronous tasks), and which can be executed in any order (asynchronous tasks). This requires a graphics application programming interface (API) that allows developers to provide this information. This is a key capability of the new generation of graphics APIs, including Mantle, DirectX® 12, and Vulkan™.

In DirectX 12, this is handled by allowing applications to submit work to multiple queues. The API defines three types of queues:

- Graphics queues for primary rendering tasks
- Compute queues for supporting GPU tasks (physics, lighting, post-processing, etc.)
- Copy queues for simple data transfers

Command lists within a given queue must execute synchronously, while those in different queues can execute asynchronously (i.e. concurrently and in parallel). Overlapping tasks in multiple queues maximize the potential for performance improvement.

Developers of games for the major console systems are already familiar with this idea of multiple queues and understand how to take advantage of it. This is an important reason why those game consoles have typically been able to achieve higher levels of graphics performance and image quality than PCs equipped with a similar level of GPU processing power. However the availability of new graphics APIs is finally bringing similar capabilities to the PC platform.

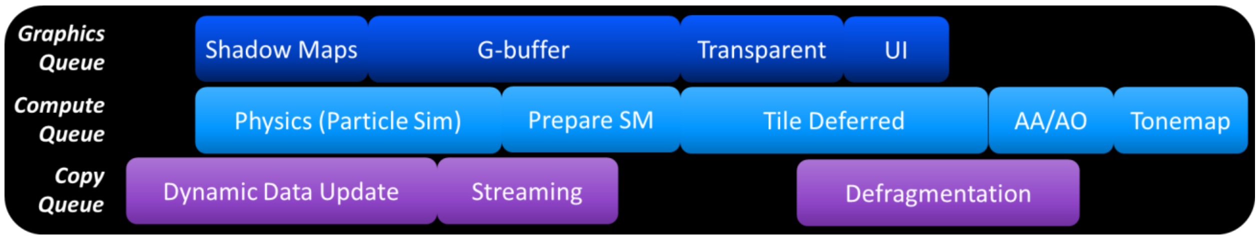


Figure 3: Example of graphics rendering tasks assigned to the different queue types available in DirectX 12

SCHEDULING

A basic requirement for asynchronous shading is the ability of the GPU to schedule work from multiple queues of different types across the available processing resources. For most of their history, GPUs were only able to process one command stream at a time, using an integrated command processor. Dealing with multiple queues adds significant complexity. For example, when two tasks want to execute at the same time but need to share the same processing resources, which one gets to use them first?

Consider the example below, where two streams of traffic (representing task queues) are attempting to merge onto a freeway (representing GPU processing resources). A simple way of handling this is with traffic signals, which allow one traffic stream to enter the freeway while the other waits in a queue. Periodically the light switches, allowing some traffic from both streams onto the freeway.

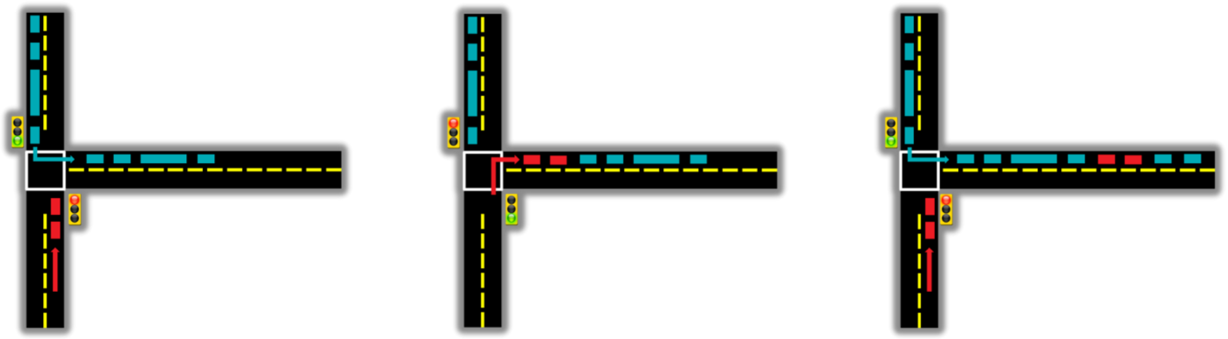


Figure 4: Representation of a simple task switching mechanism

To get the GPU to switch from working on one task to another, a number of steps are required:

- Stop submitting new work associated with the current task
- Allow all calculations in flight to complete
- Replace all context data from the current task with that for the new task
- Begin submitting work associated with the new task

Context (also known as "state") is a term for the working set of data associated with a particular task while it is being processed. It can include things like constant values, pointers to memory locations, and intermediate buffers on which calculations are being performed. This context data needs to be readily accessible to the processing units, so it is typically stored in very fast on-chip memories. Managing context for multiple tasks is central to the scheduling problem.

An alternative way to handle scheduling is by assigning priorities to each queue, and allowing tasks in higher priority queues to pre-empt those in lower priority queues. Pre-emption means that a lower priority task can be temporarily suspended while a higher priority task completes. Continuing with the traffic analogy, high priority tasks are treated like emergency vehicles – that is, they have right-of-way at intersections even when the traffic light is red, and other vehicles on the road must pull to the side to let them pass.

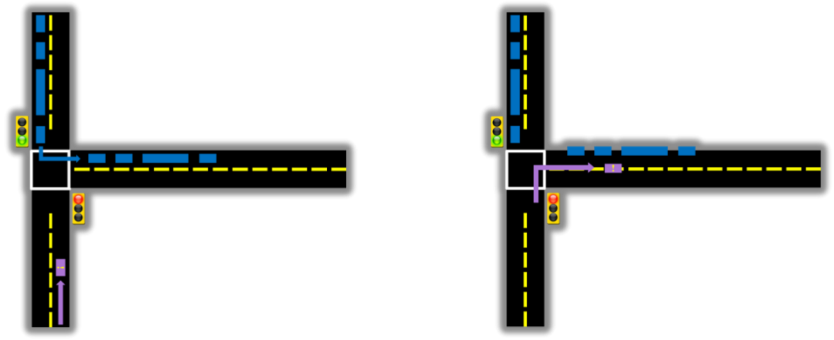


Figure 5: Pre-emption mechanism for handling high priority tasks

This approach can reduce processing latency for tasks that need it most, however it doesn't necessarily improve efficiency since it is not allowing simultaneous execution. In fact, it can actually reduce efficiency in some cases due to context switching overhead. Graphics tasks can often have a lot of context data associated with them, making context switches time consuming and sapping performance.

A better approach would be to allow new tasks to begin executing without having to suspend tasks already in flight. This requires the ability to perform fine-grained scheduling and interleaving of tasks from multiple queues. The mechanism would operate like on-ramps merging on to a freeway, where there are no traffic signals and vehicles merge directly without forcing anyone to stop and wait.

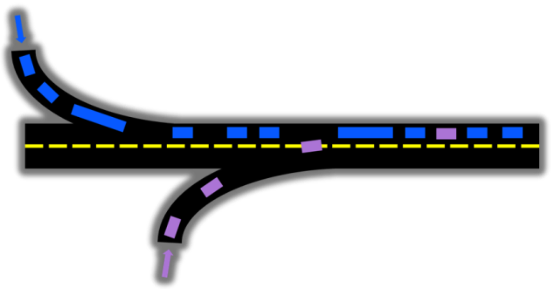


Figure 6: Asynchronous compute with fine-grained scheduling

The best case for this kind of mechanism is when lightweight compute/copy queues (requiring relatively few processing resources) can be overlapped with heavyweight graphics queues. This allows the smaller tasks to be executed during stalls or gaps in the execution of larger tasks, thereby improving utilization of processing resources and allowing more work to be completed in the same span of time.

HARDWARE DESIGN

The next consideration is designing a GPU architecture that can take full advantage of asynchronous shading. Ideally we want graphics processing to be handled as a simultaneous multi-threaded (SMT) operation, where tasks can be assigned to multiple threads that share available processing resources. The goal is to improve utilization of those resources, while retaining the performance benefits of pipelining and a high level of parallelism.

AMD's Graphics Core Next (GCN) architecture was designed to efficiently process multiple command streams in parallel. This capability is enabled by integrating multiple Asynchronous Compute Engines (ACEs). Each ACE can parse incoming commands and dispatch work to the GPU's processing units. GCN supports up to 8 ACEs per GPU, and each ACE can manage up to 8 independent queues.

The ACEs can operate in parallel with the graphics command processor and two DMA engines. The graphics command processor handles graphics queues, the ACEs handle compute queues, and the DMA engines handle copy queues. Each queue can dispatch work items without waiting for other tasks to complete, allowing independent command streams to be interleaved on the GPU's Shader Engines and execute simultaneously.

This architecture is designed to increase utilization and performance by filling gaps in the pipeline, where the GPU would otherwise be forced to wait for certain tasks to complete before working on the next one in sequence. It still supports prioritization and pre-emption when required, but this will often not be necessary if a high priority task is also a relatively lightweight one. The ACEs are designed to facilitate context switching, reducing the associated performance overhead.

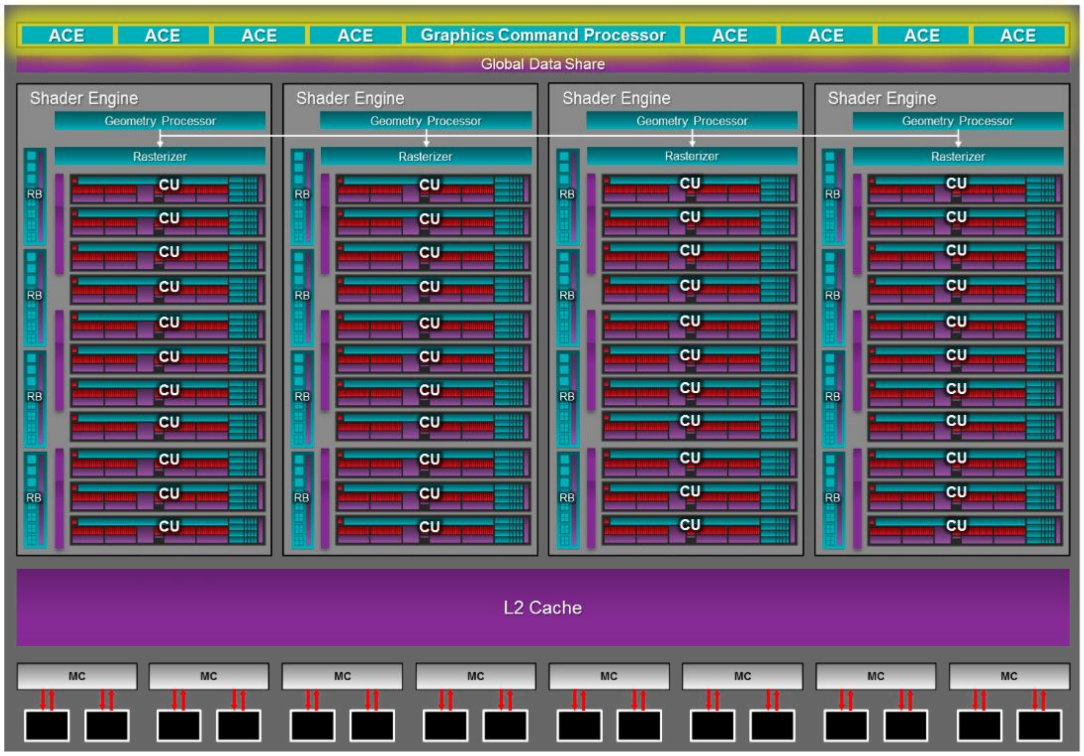


Figure 7: Block diagram of the GCN architecture as implemented in the AMD Radeon R9 390X GPU

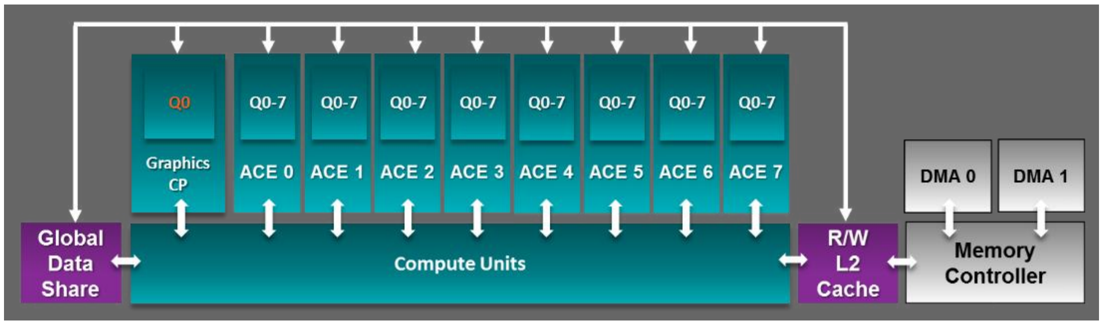


Figure 8: GCN command processing architecture

USING ASYNCHRONOUS SHADERS

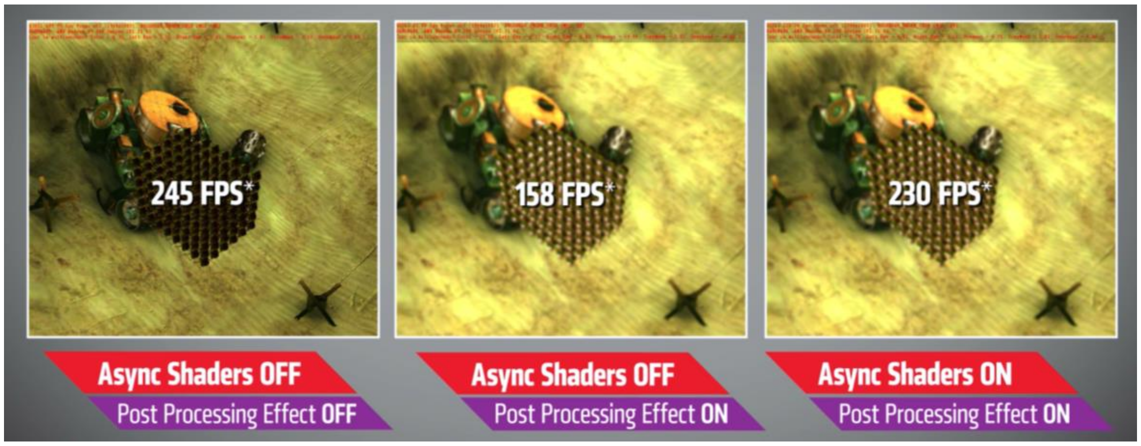
The ability to perform shading operations asynchronously has the potential to benefit a broad range of graphics applications. Practically all modern game rendering engines today make use of compute shaders that could be scheduled asynchronously with other graphics tasks, and there is a trend toward making increasing use of compute shaders as the engines get more sophisticated. Many leading developers believe that rendering engines will continue to move away from traditional pipeline-oriented models and toward task-based multi-threaded models, which increases the opportunities for performance improvements. The following are examples of some particular cases where asynchronous shading can benefit existing applications.

Post-Processing Effects

Today's games implement a wide range of visual effects as post-processing passes. These are applied after the main graphics rendering pipeline has finished rendering a frame, and are often implemented using compute shaders. Examples include blur filters, anti-aliasing, depth-of-field, light blooms, tone mapping, and color correction. These kinds of effects are ideal candidates for acceleration using asynchronous shading.

The following example shows a blur filter applied to a rendered scene. Using asynchronous shaders for the blur effect improves performance by 45%[1].

[1] Measured in AMD Internal Application – Asynchronous Compute. Test System Specifications: AMD FX 8350 CPU, 16GB DDR3 1600 MHz memory, 990 FX motherboard, AMD R9 290X 4GB GPU, Windows 7 Enterprise 64-bit


245 FPS*
Async Shaders OFF
Post Processing Effect OFF
158 FPS*
Async Shaders OFF
Post Processing Effect ON
230 FPS*
Async Shaders ON
Post Processing Effect ON

* Average frames per second

Figure 9: Example of a post-process blur effect accelerated with asynchronous shaders

Lighting

Another common technique in modern games is deferred lighting. This involves performing a pre-pass over the scene with a compute shader before it is rendered, in order to determine which light sources affect each pixel. This technique makes it possible to efficiently render scenes with a large number of light sources.

The following example, which uses DirectX 12 and deferred lighting to render a scene with many light sources, shows how using asynchronous shaders for the lighting pre-pass improves performance by 10%[2].

```
<t> to capture timing data.
<r> to toggle async compute usage: async OFF
Rasterization:
Compute:
Frametime:
```

Figure 10: Demonstration of deferred lighting using DirectX 12 and asynchronous shaders

[2] Measured in AMD internal application – D3D12_AsyncCompute. Test System Specifications: Intel i7 4960X, 16GB DDR3 1866 MHz, X79 motherboard, AMD Radeon R9 Fury X 4GB, Windows 10 v10130

Virtual Reality

Virtual reality represents a new frontier for graphics. However, when rendering images to virtual reality headsets, low latency is critical to avoiding nausea and motion sickness. One of the challenges associated with reducing latency is ensuring that the GPU has access to the latest head position information when it is rendering each frame. Even if the information is fresh when rendering begins, it will already be stale by the time the frame completes.

One approach to dealing with this is to use a technique called Time Warp. This involves obtaining fresh head tracking information after each frame finishes rendering, and using it to warp the frame to appear as if it was rendered from the new viewpoint.

This warping makes use of a compute shader, and needs to be executed with high priority to avoid adding latency back into the pipeline. Executing it asynchronously allows latency to be minimized and eliminates stuttering, since context switching and pre-emption overhead can be avoided.

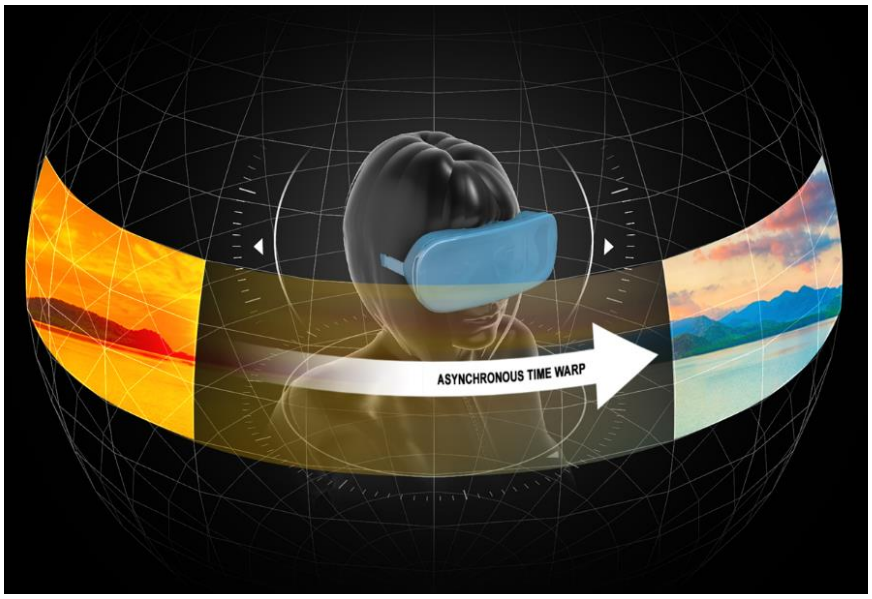


Figure 11: Asynchronous Time Warp is one of the features enabled by AMD's LiquidVR technology

SUMMARY

Hardware, software and API support are all now available to deliver on the promise of asynchronous computing for GPUs. The GCN architecture is perfectly suited to asynchronous computing, having been designed from the beginning with this operating model in mind. This will allow developers to unlock the full performance potential of today's PC GPUs, enabling higher frame rates and better image quality.