Performance, Methods, and Practices of DirectX* 11 Multithreaded Rendering

By Guo Sheng (Intel), Mi Nan (NetEase Booming)

Abstract

Rendering is usually the main performance bottleneck of PC games on the CPU; multithreaded rendering is an effective way to eliminate the bottleneck. This article investigates the performance scalability of DirectX* 11 multithreaded rendering, discusses two basic methods for multithreaded rendering, and introduces the case of traditional multithreading deferred shading pipelines in a large-scale online game, *Conqueror's Blade**.

Background

Over the past 10 years, CPU chips in the PC market have shown great improvements. According to a software and hardware investigation by Steam [2], 4-core processors (usually 8 logical cores) have become mainstream in the current PC game market. The 6-core processor (usually 12 logical cores) is already on its way to become the mainstream next-generation CPU. For example, the Intel® Core™ i7-8700K processor with 8 or more physical cores has been available since late 2017. We expect this trend to continue. In the next few years, 6-core and 8-core CPUs will become the most popular processors for gamers.

In many PC games, rendering is usually single-threaded and easily becomes the biggest performance bottleneck. This makes it difficult for games to utilize extra idle cores in a multicore processor to improve game performance or enrich game content. Although DirectX 12 has been around a few years, most of the games currently under development—especially the most popular online games—are still using DirectX 11. DirectX 11 is designed to support multithreading from the beginning [1]. Therefore, investigating the performance scalability of DirectX 11 multithreaded rendering on current mainstream multicore platforms, and studying the methods of making full use of this feature have important reference value for the development and optimization of the majority of games.

DirectX* 11 Multithreaded Rendering Model

First, let's briefly review the DirectX 11 multithreaded rendering model (see Figure 1). DirectX 11 supports two types of rendering—immediate and

deferred, based on two Direct3D* 11 device contexts—the immediate context and the deferred context. Immediate rendering calls draw APIs through immediate context, and the generated command is immediately sent to the graphics processing unit (GPU). Deferred rendering calls draw APIs through deferred context, but only records the draw commands in a command list that is submitted to the GPU by the immediate context at another time point. DirectX 11 supports the use of different deferred simultaneous contexts in multithreading. This strategy allows the rendering of complex scenes to be divided into multiple concurrent tasks; that is, multithreaded rendering.

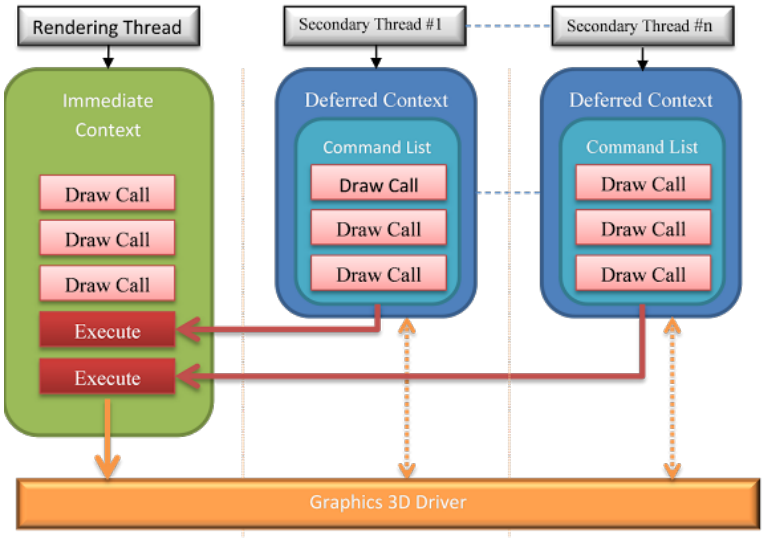


Figure 1: *DirectX* 11 multithreaded rendering model.*

Evaluate DirectX 11 Multithreading Performance Scalability

Based on the hardware and software configuration of Table 1, we evaluate the performance scalability of DirectX 11 multithreaded rendering on multicore CPUs.

Configuration	Description	
CPU	Intel® Core™ i7-6950X processor @ 3.00GHz (10 Cores)	
Memory	2 x 16 GB RAM	
GPU	NVIDIA GeForce* GTX 1080	AMD Radeon* RX Vega 64
Driver Version	22.21.13.8494	22.19.677.257
Operating System	Windows® 10 Professional 64-bit	
Test Program	Microsoft DirectX* SDK (June 2010) Sample: MultithreadedRendering11.exe	

Table 1: *Hardware and software configurations for performance scalability evaluation.*

The evaluation uses the Intel Core i7-6950X processor (10 physical cores; that is, 20 logical cores) to simulate CPUs with different numbers of cores. To ensure that the GPU does not become a performance bottleneck for the test program, the test uses two high-performance discrete GPUs: NVIDIA GeForce* GTX 1080 and AMD Radeon* RX Vega 64. The test program uses the MultithreadedRendering11 routine in the Microsoft DirectX SDK [4], which is based mainly on the following considerations. First, the program performance is CPU-bound, and it is developed to demonstrate the DirectX 11 multithreaded rendering feature, which is conducive to maximizing the potential of performance scalability. Second, the main function of the program is rendering (each frame contains more than 4,000 draw calls), and there is no impact of animation, physical load, and so on. It can make scalability a result of DirectX 11 multithreaded rendering as much as possible. In addition, the program's scene complexity and rendering technology are pretty common in games, so that the test results are of representativeness. Last, but not least, the source code of the program is open, making it easy to analyze and understand the DirectX 11 multithreaded rendering methods, and the impact on scalability performance.

Figure 2: *Test program.*

When running the test program, we chose the MT Def/Chunk mode, because the scalability in this mode is not limited by the number of game rendering passes (or scenes), but only by the number of CPU cores. The workload of each thread is relatively balanced, which can make full use of the computing power of the multicore CPU. During the test, we adjusted the CPU's active

core number through the BIOS and tested the program's frame rate at each of these different core numbers. In order to compare the effects of different GPUs on DirectX 11 multithreaded rendering scalability, we divided the multithreaded frame rate on the same GPU by the single-threaded frame rate (immediate mode) under the same configuration, to obtain a normalized relative performance metric. The test results are shown in Figure 3.

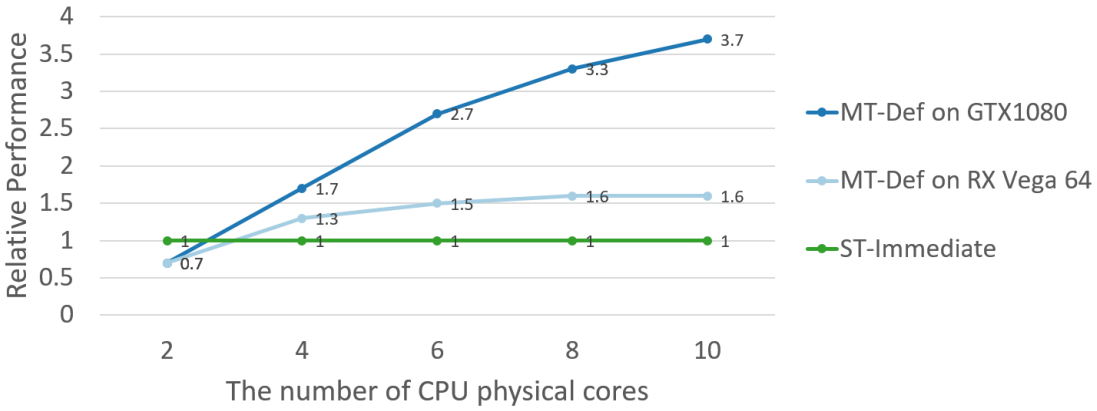


Figure 3: *Multicore performance scalability of DirectX* 11 multithreaded rendering.*

As we can see from Figure 3, with two CPU cores, no matter which GPU we use, multithreaded rendering (MT Def/Chunk mode) performance is lower than single-threaded rendering (immediate mode). What leads to this result? According to the source code of the test program, the number of working threads is the number of CPU physical cores minus one. In other words, on a two-core CPU, in multithreaded rendering mode, only one working thread processes all scene draw calls based on *deferred rendering*, while the main thread does not assume any scene draw calls. In the single-thread rendering mode, all draw calls are processed by the main thread based on *immediate rendering*. This means that the overhead of *deferred rendering* is slightly larger than that of *immediate rendering* on the basis of handling an equal number of draw calls.

However, when the number of CPU cores is greater than two, the DirectX 11 multithreaded rendering performance is significantly better than that of single-threaded rendering, regardless of which GPU is used, and the performance increases as the number of cores increases. When paired with the NVIDIA GeForce GTX 1080, multicore performance scales very well; performance increase is almost linear from 2 to 6 cores. Even from 6 to 10 cores, the performance increase is significant. When paired with AMD Radeon RX Vega 64, the scalability is worse than that; especially when the number of CPU cores exceeds 4, the performance increase is almost negligible.

Why does the test program have such a large scalability difference for multicore performance on different GPUs? We used Microsoft GPUView* to capture the multithreaded activities of the test program (see Figure 4), and find that the bottleneck of the test program is on the CPU with either the NVIDIA GeForce GTX 1080 or the AMD Radeon RX Vega 64 GPU. However, multithreaded concurrency is better with the NVIDIA GPU, and the main thread blocking working threads is significantly longer with AMD graphics cards.

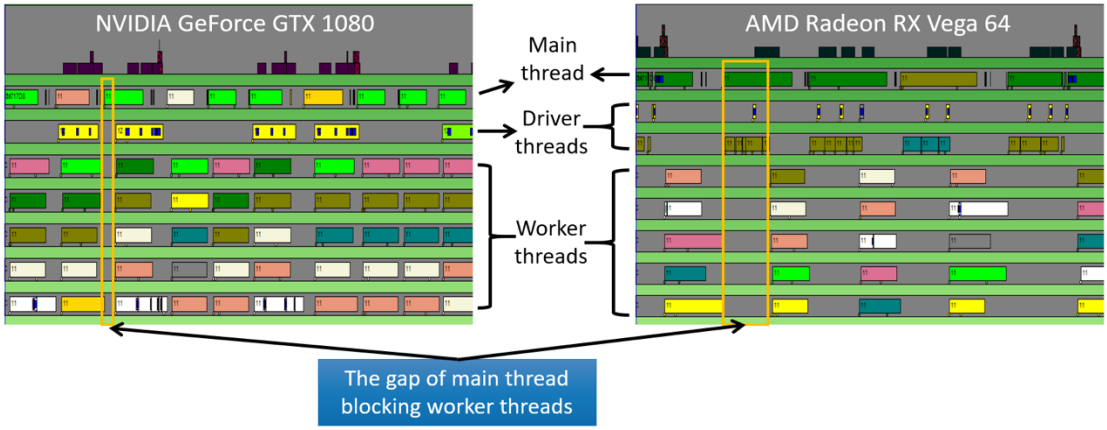


Figure 4: *DirectX* 11 multithreaded rendering parallelism with different GPUs.*

From the source code, we know that each working thread has a *deferred context*, and all draw calls for scene rendering are called by *deferred context*. The main thread contains an *immediate context* that is responsible for submitting the commands list generated in the *deferred context* to the GPU. Using Microsoft Windows* Performance Analyzer to further analyze the module called by the working thread, we find that, on the NVIDIA GPU, all the working threads call the graphics driver module (see Figure 5), which means that a number of *deferred context* operations share some of the driver load, and make the *immediate context* operations bear less driver load, thereby shortening the occurrences of the main thread blocking the working threads. On the AMD GPU, the graphics driver module does not appear in the working thread but is concentrated on the main thread (see Figure 6), which means that a single *immediate context* bears a large amount of driver load, thus increasing the time of the working threads waiting for the main thread.

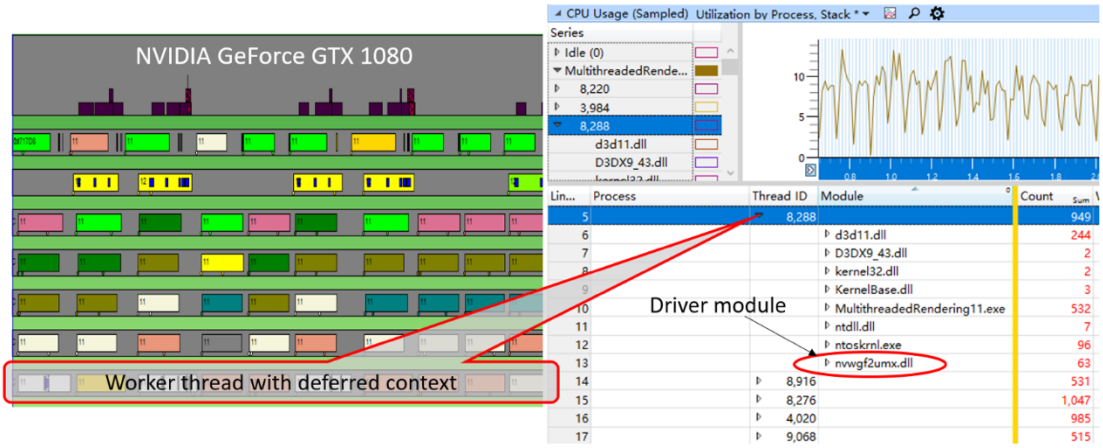


Figure 5: *Working thread (deferred context) represents some of the NVIDIA driver load.*

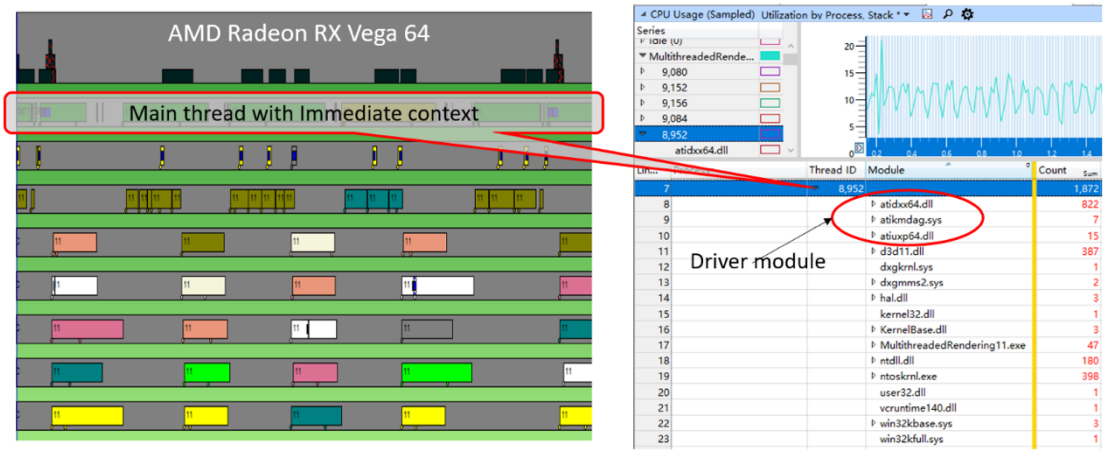


Figure 6: *The main thread (immediate context) represents a large amount of the driver load.*

By checking the GPU driver support for DirectX 11 multithreaded rendering features [3] (see Figure 7) through the DirectX Caps Viewer, we learn that the NVIDIA GPU driver supports driver command lists, while the AMD GPU driver does not support them. This explains why the driver modules on different GPUs appear in different contexts. When paired with the NVIDIA GPU, working threads can build driver commands in parallel in a *deferred context*; while when paired with the AMD GPU, the driver commands are all built in serial in the *immediate context* of the main thread.

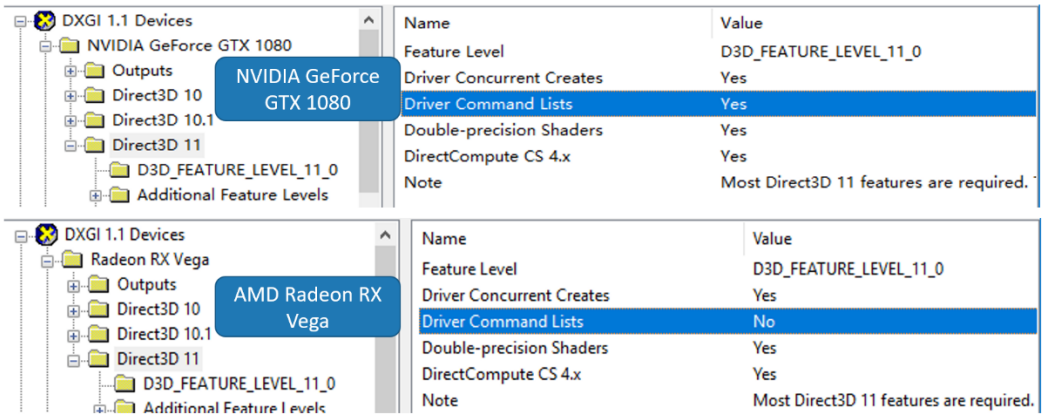


Figure 7: *Support for DirectX* 11 multithreaded rendering by different GPU drivers.*

Based on the above tests and analysis, we can draw the following conclusions:

- Although the indirect load of deferred rendering is larger than that of the immediate rendering, the performance of the DirectX 11 multithreaded rendering can be significantly higher than that of single-threaded rendering, especially on current mainstream 4-core or more-core CPUs when using the appropriate rendering task division method—evenly distributing draw calls to contexts of more than two Direct3D devices.

- The performance scalability of DirectX 11 multithreaded rendering is GPU-related. When the GPU driver supports the driver command list, DirectX 11 multithreaded rendering can achieve good performance scalability, whereas performance scalability is easily constrained by the driver bottleneck. Fortunately, the NVIDIA GPU [2], with the largest share of the current game market, supports driver command lists.

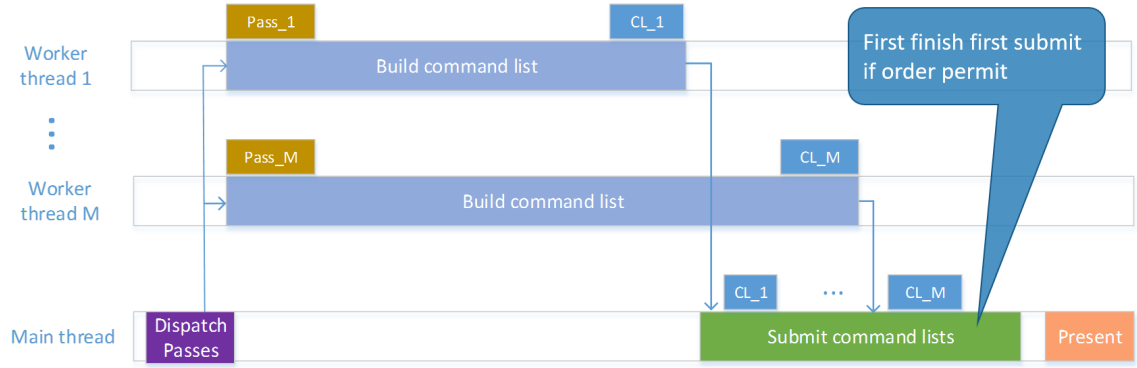
Multithreaded Rendering Method

The performance scalability evaluation of the above DirectX 11 multithreaded rendering shows that on the current mainstream multicore CPUs and GPUs, multithreaded rendering on DirectX 11 games may achieve significant performance improvement. So, how do you effectively use the performance potential of multithreaded rendering? The MultithreadedRendering11 routine demonstrates two basic methods for dividing a rendering task into multiple threads:

1) Assign each thread a rendering *Pass*.
2) Assign each thread an equal amount of *Chunk*.

It should be noted that the multithreaded rendering method described here is not only suitable for DirectX 11 but also for DirectX 12. In fact, we can take the DirectX 11 deferred context as a DirectX 12 command list, and the DirectX 11 immediate context as a combination of the DirectX 12 command list and the command queue.

Figure 8 shows a multithreaded rendering method that divides the rendering task by *Pass*. *Pass* is a relatively independent rendering task. The typical *Pass* includes the generation of pre-Z buffers, shadow maps, reflection maps, G buffers, UI, and the main *Pass* generating the final frame buffer. With this method, each *Pass* is assigned with a working thread. A command list of this *Pass* is built into this working thread. The main thread is responsible for distributing *Pass* and orderly submitting the command list completed by the working threads. In the MultithreadedRendering11 routine, the main thread will orderly submit after all the working threads complete the command list. Figure 8 shows a better way: When a command list is completed, it should be immediately submitted to the GPU as long as the rendering order permits. Since the submitted command list is usually serial and associated with some overhead, the earlier the submission, the more serial time that can be shielded, which allows GPU processing in advance.



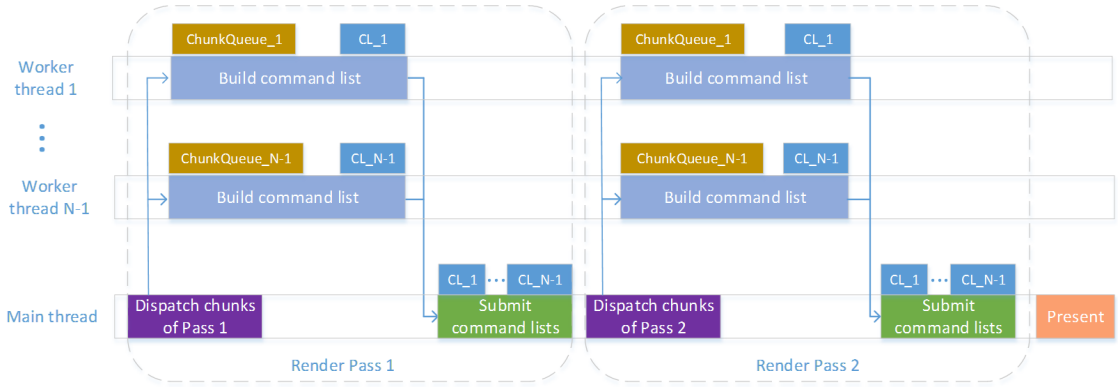
M: the number of passes

Figure 8: *Divide the rendering task by Pass.*

Dividing rendering tasks by *Pass* is easy to apply to the multiple-pass rendering technology commonly used in modern games. As long as *Pass* contains a relatively large amount of rendering load (draw call number), using this method in games is usually effective in improving performance. The shortcoming is that the performance scalability is limited by the number of *Passes*, and it is not easy to achieve load balance between *Passes*.

Figure 9 shows a multithreaded rendering method that divides rendering tasks by *Chunk*. *Chunk* is a granularity rendering task that is smaller than *Pass*. A typical *Chunk* can be a set of draw calls, a mesh, or a larger rendering unit such as a separate rendering object containing multiple meshes. In this

method, each *Pass* is divided into *Chunks*, which are evenly distributed by the main thread to multiple working threads. Each working thread is responsible for building a command list. After each command list is completed, the main thread is responsible for submitting them in order. The number of working threads is determined based on the number of physical cores, rather than the number of logical cores, in order to avoid excessive command list submissions resulting in excessive overhead. The *Pass* as the unit of submitting the command list is conducive to unifying the rendering status of the command list, advancing GPU processing, and multiplexing the command list between *Passes*.



N: the number of physical cores

Figure 9: *Dividing rendering tasks by Chunk.*

The multithreaded rendering method that divides rendering tasks by *Chunk* can achieve a significant performance improvement, and the performance is not affected by the number of passes and increases with the increase of the number of CPU cores. The shortcoming is that for certain situations that require orderly rendering (such as rendering semi-transparent objects), the strategy of distributing *Chunks* is limited, and it is easy to lose the load balance among the threads, thereby affecting the performance scalability.

No matter which of the above multithreaded rendering methods is used, the following points should be noted:

- Since the submitted command list is serial and with a certain amount of overhead, the command list should be executed immediately after it is completed and allowed by rendering order, rather than waiting for the other command lists. The former helps shield the serial time and relieves the GPU from burst load pressure.

- To shield the overhead of using deferred contexts, each deferred context, whether for *Pass* or *Chunk*, should contain enough draw calls. If the

number of draw calls processed by the deferred context is too small, you should consider handling these draw calls in the immediate context or combining them.

- Try to balance the load between different contexts to maximize the advantages of multithreaded rendering.

Case Study

Here we introduce multithreaded rendering methods and effects achieved in a real DirectX 11 game. *Conqueror's Blade* [5] is a large-scale online game developed by NetEase. The game has large-scale outdoor battle scenes, a large number of characters on the same screen, and rich visual effects. These characteristics make the game demand more CPU resources. To enable players on the low-end CPU platforms to have a smooth gaming experience, developers continue to optimize the game engine for multithreading optimization in order to improve performance by fully utilizing CPU resources, or to improve game details.

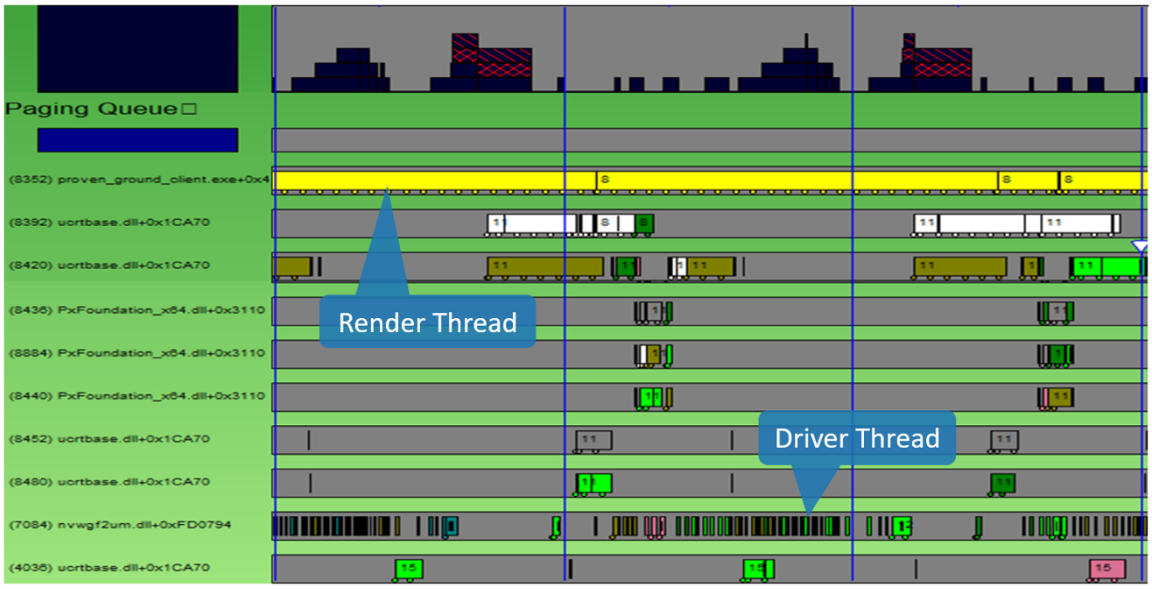


Figure 10: *Single-threaded rendering causes CPU performance bottlenecks.*

Before the performance optimization of the game, the engine has achieved a certain amount of multithreading: some CPU-intensive tasks, such as game logic, physics, particles, animation, and other calculations use separate threads for execution. The rendering thread is mainly responsible for visibility detection and running the entire rendering pipeline. Nevertheless, the rendering thread is still a performance bottleneck for the game (see Figure 10). A typical combat scene with more than 5,000 draw calls per frame also results in considerable Direct3D runtime and driver overhead.

The game uses the DirectX 11 API and a typical deferred shading pipeline. The task pipeline of the rendering thread is shown in Figure 11.

Figure 11: *The game's task pipeline of rendering thread.*

Based on some considerations such as limitation of game legacy code and implementation time, the game chooses a multithreading optimization method that divides rendering tasks according to *Pass*, which is a relatively easier implementation choice in a limited time. The specific implementation scheme is shown in Figure 12.

In the optimization scheme, Visibility is removed from the rendering thread and divided into two jobs: eye visibility and light visibility. The GBuffer generation, the shadow map generation, and the forward and transparent *Passes* that have or may dynamically have a large number of draw calls are also moved out of the rendering thread and encapsulated as a job that dispatches working threads. *GBuffer generation* has been further divided into three jobs: *GBuffer Terrain*, *GBuffer Static*, and *GBuffer Dynamic*, because there are too many draw calls. The rendering thread only retains the Scaleform UI. Deferred Shading and Post Process Passes must use *immediate context* rendering or have only a few draw calls.

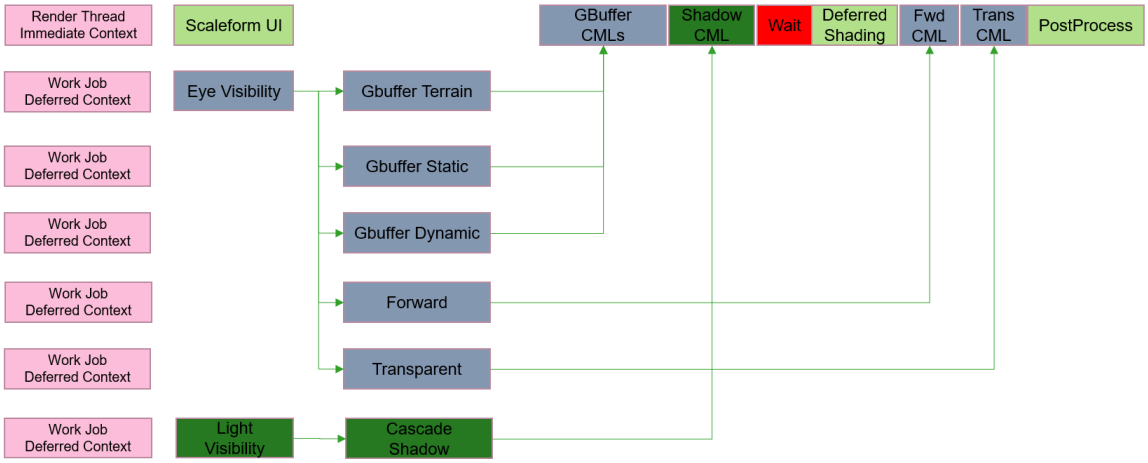


Figure 12: *Multithreaded rendering flowchart after game optimization.*

During the operation process, the working thread first processes the two visibility check jobs in parallel, then these two jobs derive six jobs rendering passes, and the working thread builds a related *Pass* DirectX 11 command list in the deferred context. The rendering thread orderly executes the *Passes* left in the rendering thread and the command list that has been completed by the working threads in the immediate context.

After the multithreading optimization is complete, the bottleneck of the rendering thread is eliminated, the multicore utilization is significantly improved (see Figure 13), and the frame rate is increased by an average of 1.7 times than before the optimization. It has achieved the set performance target.

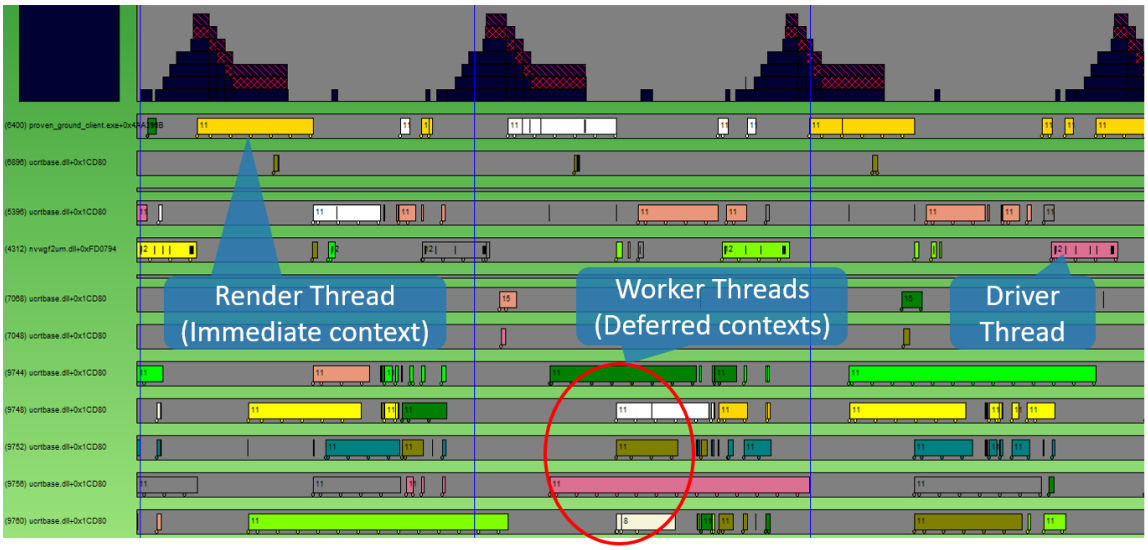


Figure 13: *Eliminate bottlenecks after multithreaded rendering to improve multicore utilization.*

Although the current solution significantly improves performance, there is still a lot of room for improvement in CPU utilization due to an uneven load between *Passes*. Therefore, in order to make the idle CPU further enhance performance details in games, the developers will rebuild the rendering code of the game engine and try to divide the multithreading optimization of the rendering task by *Chunk*.

Summary

On the multicore CPUs and GPUs with the largest share of the current game market, and for those DirectX 11 games on CPUs with performance bottlenecks in rendering, achieving multithreaded rendering may help realize significant performance improvements. Although the multicore performance scalability of DirectX 11 multithreaded rendering is limited for some GPUs with limited driver support, under the condition of reasonable implementation the performance of multithreaded rendering will be better than that of single-threaded rendering. The key to the advantage of multithreaded rendering is the division and scheduling of rendering tasks. For this reason, this article introduces the methods based on *Pass* and *Chunk*. These two methods are not only applicable to DirectX 11, but also to DirectX 12, so that multithreaded rendering optimization of DirectX 11 games can be easily ported to future DirectX 12 games. In the game

Conqueror's Blade, a *Pass*-based multithreaded method is successfully applied to the traditional deferred shading pipeline, proving the effectiveness of DirectX 11 multithreaded rendering.

Reference

1. Introduction to Multithreading in Direct3D 11：

   https://msdn.microsoft.com/en-us/library/windows/desktop/ff476891(v=vs.85).aspx

2. Steam Hardware and Software Survey：

   http://store.steampowered.com/hwsurvey

3. How To: Check for Driver Support: https://msdn.microsoft.com/en-us/library/windows/desktop/ff476893(v=vs.85).aspx

4. Microsoft DirectX SDK (June 2010)：

   https://www.microsoft.com/en-us/download/details.aspx?id=6812

5. Conquerors' Blade official website: http://z.163.com