International Journal of Multiphysics
Volume 18, No. 2, 2024
ISSN: 1750-9548

A Novel Heterogeneous Multi-GPU Parallel Rendering Framework in UE4 Scene

Siyu Zhang, Yanfeng Wang, Jianjun Guo*

[1]Beijing Weiling Times Technology Co., Ltd., Beiijng, China

* Corresponding Author.

Abstract

Parallel rendering of heterogeneous multi-GPU in UE4 scene can be realized by explicitly calling graphics API, instead of relying on hardware limitations or drivers. Dual GPUs can bring better performance improvement. In this paper, we mainly design a novel multi-thread rendering framework in UE4 scene, which has a better improvement in performance. Under the current multi-GPU implementation, we have expanded the multi-threading architecture and added a new RHI Thread, which we call MRHI Thread, to handle the rendering instruction operation for the newly added GPU. Then, in the design of rendering architecture, rendering objects are divided into U Object main thread logic component object, F Render Object rendering thread object, and FRHI Render Object RHI thread object. Moreover, Because the requirement is to render two scenes, the Scene Capture component is used as the entrance of GPU1 rendering. In the test, we used two Nvida RTX3070, and realized parallel rendering of two GPUs. The performance improved by 150-160%, and the parallel efficiency was close to 95%. Under our proposed framework, the parallel efficiency of the GPU will be greatly improved compared with the previous version.

Keywords: Parallel rendering, UE4, heterogeneous, Multi-adapters

1. Introduction

Nowadays, with the increasing quality of games, more and more people are paying attention to the development of unreal engine [1-5]. At present, unreal engine is the most advanced real-time 3D creation platform in the world, which is applied in simulation, XR development, multimedia design, transportation, architecture, visualization, urban planning and other industries. Among them, the fourth generation unreal engine (UE4) is one of the most open and advanced real-time 3D creation platforms in the world [6-8]. It can use advanced technologies such as ray tracing to create images that are difficult to distinguish between true and false. Many products that can be seen in the market, such as Survival of the Jedi and Need for Speed, are from UE4 unreal engine. In recent years, UE4 is not only used in game apps, but also widely used in film and television production[9].

UE4 scene is usually combined with rendering technology [10,11]. Usually, the process of generating images from 3D models by software is called rendering, which is widely used in simulation, animation video and film and television production [12-14]. Graphics Processing Unit (GPU) is a kind of computing chip specially used for processing graphics and images. Compared with Central Processing Unit (CPU), GPU has more parallel processing units and can process a large amount of data at the same time, so it has higher performance in graphics and images. Rendering business scenes requires GPU graphics card to realize graphics acceleration and real-time rendering, and also requires a lot of calculation, memory or storage [15-17].

Lots of works have been done in the redering technology based on GPUs [18-22]. In [18], the authors address the integration of early ray termination and empty-space skipping into texture based volume rendering on graphical processing units (GPU). The authors exploit the early z-test to terminate fragment processing once sufficient opacity has been accumulated, and to skip empty space along the rays of sight. In [19], the authors consider the problem of real-time GPU rendering of algebraic surfaces defined by Bezier tetrahedra. It is proposed that the

general framework could be extended to higher order with numerical root finding. In [20], the authors present a reformulation of bidirectional path-tracing that adequately divides the algorithm into processes efficiently executed in parallel on both the CPU and the GPU. The proposed method is more than ten times faster than standard bidirectional path-tracing implementations, leading to performance suitable for production-oriented rendering engines. An algorithm to generate high quality images with a small number of slices by utilizing displaced pixel shading technique was given in [21]. A new 3D edge detector was given. In [22], A new rendering technology of GPU-accelerated radiosity is presented. Using new OpenGL extensions to realize texture traverse, classification and accumulation, the rendering results of hemi-cube method can be used directly on GPU.

In this paper, we design a novel framework to mainly realize the parallel rendering technology of heterogeneous multiple GPUs in UE4 scene. Under the current multi-GPU implementation, we have expanded the multithreading architecture and added a new RHI Thread, which we call MRHI Thread, to handle the rendering instruction operation for the newly added GPU. The whole paper is organized as follows. In section 2, we give out the design of the proposed multi-thread rendering framework to achieve parallel rendering in UE4 scene. In section 3, the proposed framework is test based the comparision results are given. It is conclued in Section 4.

2. Design of Multi-thread Rendering Framework

2.1 Introduction of MRHI threads

UE4' s rendering pipeline involves three layers of threads, such as main thread, rendering thread and RHI thread, and multiple threads exist at the same time. In the design of multi-thread rendering framework, as shown in Figure. 1, the rendering thread of UE is divided into three layers:

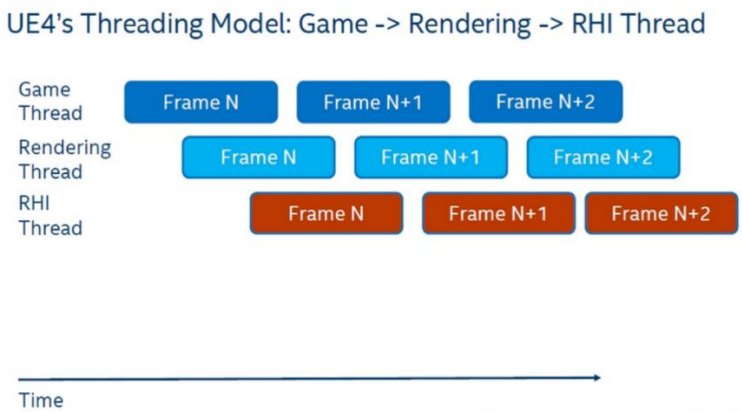

Figure 1 UE4's threading model

- Game Thread: which is used to handle the main game logic and running process, will be executed as a rendering thread at the same time when the engine starts the first frame, and there will also be some rendering tasks;
- Rendering Thread: which mainly handles rendering tasks and generates rendering instructions and rendering logic;
- RHI Thread: RHI thread is used to translate the rendering instructions passed by the rendering thread into the rendering commands of the corresponding platform API.

Under the current multi-GPU implementation, we have expanded the multithreading architecture as shown in Figure. 2, and added a new RHI Thread, which we call MRHI Thread, to handle the rendering instruction operation for the newly added GPU. The purpose of adding a new RHI Thread is to specify the GPU Adapter Index executed by the current rendering task through thread TLS as UID.

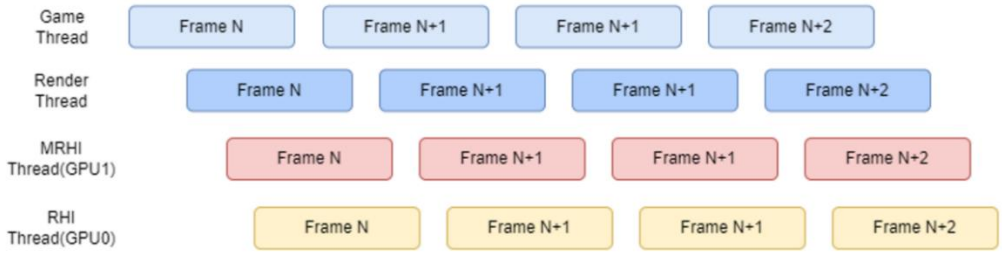

Figure 2 Multi-thread rendering model in UE

2.2 Command Dispatch

The UE mentioned above will record the rendering command in the rendering thread, and then the rendering command will be translated into the corresponding platform API command in the RHI thread. Here, the rendering thread stores the command through the FRHI Command List object, and then passes it to RHI Thread. Here, the RHI Command List is extended, bound with Adapter Index, and created as GPU1, and a global RHI Command List is also created. Then create a Command Dispatcher command dispatcher as a global object for rendering thread and game thread, which is used to control the current command to be recorded on those RHI Command List.

At present, there are three modes of command allocator as shown in Figure. 3: Dispatch To All is allocated to all adapters, Dispatch To Primary is allocated to Adapter0, and Dispatch To Code is allocated to Adapter1.

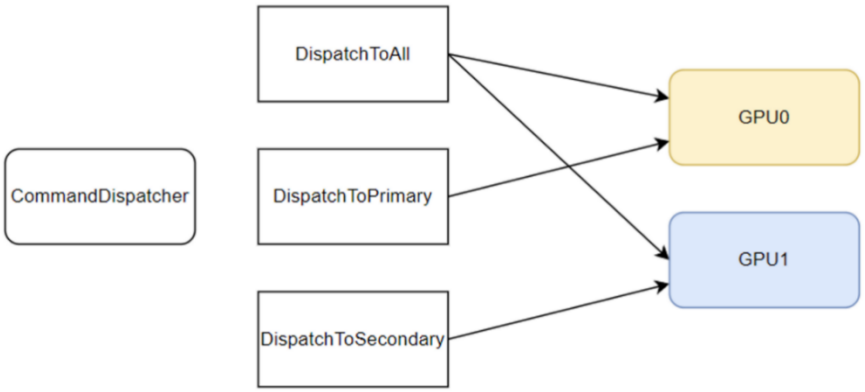

Figure 3 Command Dispatch

2.3 Resource object structure

In the design of UE rendering architecture, rendering objects are divided into U Object main thread logic component object, F Render Object rendering thread object, and FRHI Render Object RHI thread object.

These resource objects have a one-to-one correspondence between rendering threads and RHI threads. However, for different GPUs, separate objects should be created for them, and this object should be linked with rendering thread resources. Therefore, it is necessary to create a structure that can map multiple D3D12 resources in the resource object of RHI layer. The mapping relationship is shown in Figure. 4.

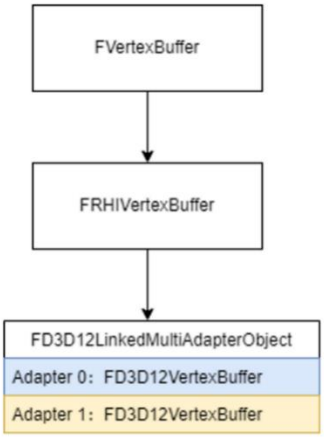

Figure 4 The mapping of Relationship

The resource objects that inherit this structure are: In addition to rendering resources, there are pipeline state object PSO objects. The design of PSO mechanism in UE is very complicated because it faces a variety of APIs. Simply put, it can be divided into three layers: first, Shader Pipeline Cache, and the cache of Pipeline parameters at the rendering thread level is essentially CSV table. Then there is the Pipeline State Cache, which is the cache of RHI PSO objects by the RHI layer. Finally, FD3D12PipelineStateCacheBase caches D3D12 RHI objects.

For PSO objects, it is also necessary to expand multiple GPUs. At present, the operation of rendering thread on PSO is transferred to RHI thread, and a Map of PSO cache is created for each GPU, and the operation of the corresponding GPU is carried out in RHI thread.

2.4 Rendering application layer pipeline

Because the requirement is to render two scenes, the Scene Capture component is used as the entrance of GPU1 rendering. Extends the rendering mode of Scene Capture component, M Adapter Rendering. When the Scene Capture rendering mode is M Adapter Rendering, the rendering logic flow is as shown in Figure. 5.

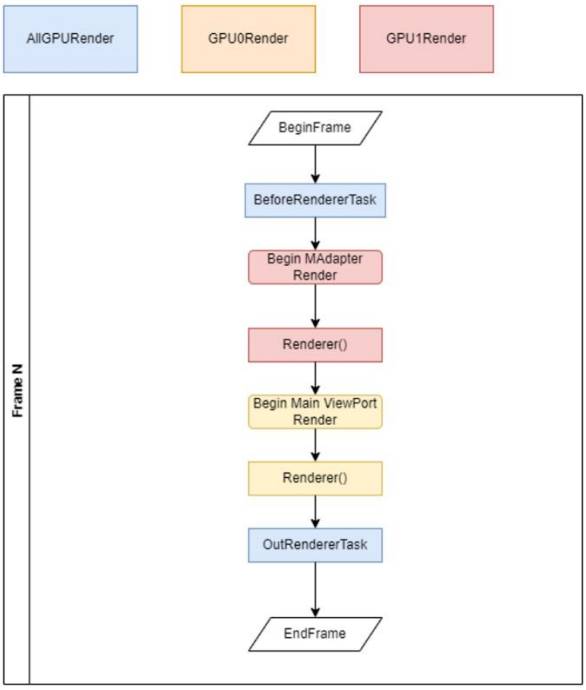

Figure 5 Rendering logic flow chart

The current implementation is divided into three stages, namely, GPU0 Renderer (), GPU1 Renderer () and Renderer (). The renderer here is the F Scene Renderer, and the pipeline for scene rendering in UE is the PC delay pipeline F Deferred Shading Scene Renderer. What is executed in the Renderer is the rendering of scene objects, and the input contains viewport information and information needed for scene rendering, so it should be executed separately for each GPU, and it is necessary to switch the Command Dispatcher to the corresponding task allocation mode during execution.

In addition to Renderer, we also have external rendering tasks such as particle simulation, Texture Steam and Environment Map. These rendering tasks usually prepare the necessary data information for the Renderer, so they need to be executed on multiple GPUs.

2.5 Rendering command delayed execution

When designing Cmdlist rendering commands, UE usually has two modes: immediate execution by bypass and delayed execution by pressing into the command queue, and some of them are directly executed on the rendering thread.

Bypass immediate execution mode is designed to make local synchronous calls to APIs that do not support Deferred Context, such as OpenGL and early Dx11. And open multi-thread rendering API such as Dx12. When the engine starts the first frame, Vulkan will also execute in Bypass mode. Another purpose is to facilitate debugging.

Push command queue mode, which is called more modes in multithreading rendering of modern API such as Dx12 and Vulkan.

Take Draw Primitive Indirect as an example, as shown in Figure. 6, and the process is as follows:

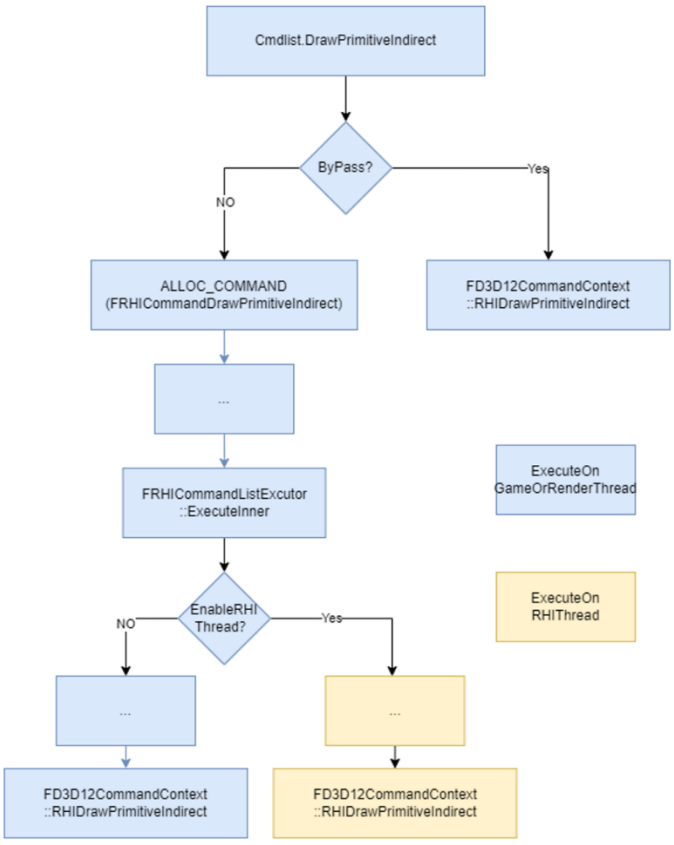

Figure 6 Push command queue mode

The calling process is through the member function of Cmdlist, and the object that Cmdlist usually obtains in rendering thread and game thread is GRHI CommandList.CommandList Immediate global object. The ALLOC_COMMAND macro is used to push commands into the command queue. What we need to change here is to

extend ALLOC_COMMAND to push the command into the corresponding Cmdlist according to the mode of Command Dispatcher. When FRHI CommandList Executor executes, different RHI threads are enabled for execution according to different Cmdlist. Therefore, the Bypass path and the execution path that does not enable RHI Thread will be discarded. The modified process is shown in Figure. 7.

Some of the function operations corresponding to Cmdlist only provide Bypass immediate execution mode in design. Because of the need for multi-thread delay execution, this part of the command needs to be extended and changed so that it can be executed in the RHI layer.

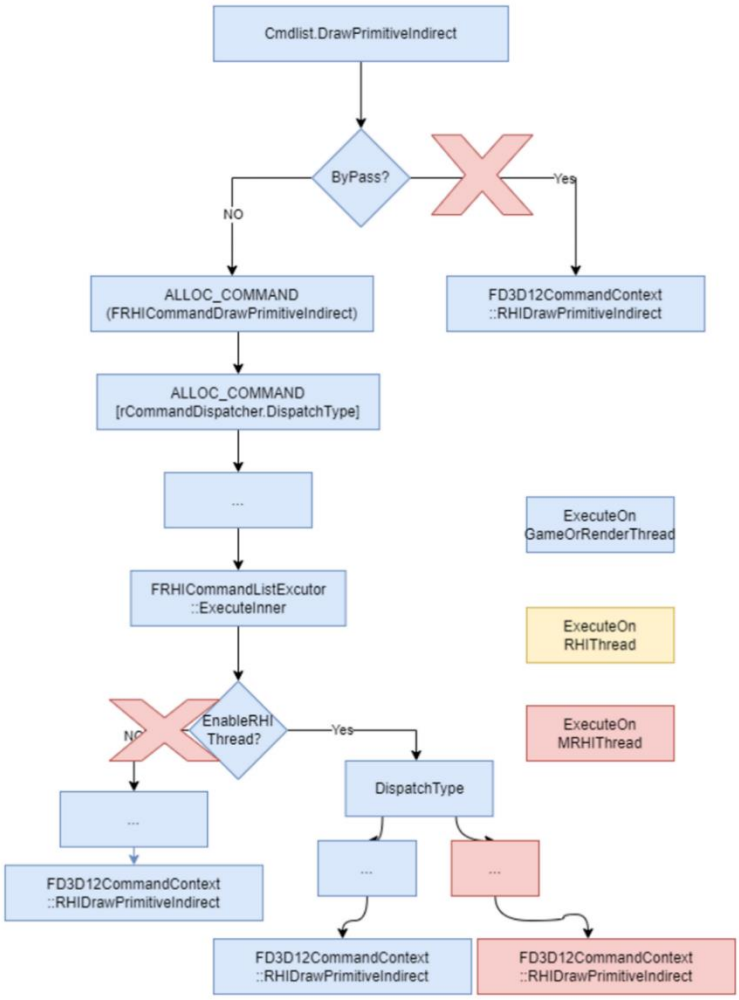

Figure 7 Modified push command queue mode

2.6 Multi-GPU synchronization logic

In the current platform, only the GPU is connected to the realtor for output. If you want to get the rendering structure of GPU1, you need to copy the data across GPUs. Cross-GPU copy operation depends on API to create cross-GPU resources. There are two ways to get cross-GPU resources: Copy, DX12. One is that if the GPU supports Feature CrossdapterRowMajortextSupported, the operation can be directly completed through the statements Copy Resource (GPUTexute [1], GPUTexute [2]), and there will also be hardware driver optimization. However, there are very few GPUs supported by this Feature, basically only a few Intel core displays and Intel ARC single displays are supported.

For most vendor, there is no support. Therefore, we can only rely on another Copy method: explicitly control the Copy operation. The copy process is GPU1-Copy→GPU1Shared, GPU0Shared-copy→GPU0. The process of GPU1-copy→GPU1Shared is essentially a memory read-back process, which consumes a lot of bandwidth and has a slow performance. And this process will interrupt the GPU and CPU processes.

The specific process is shown in Figure. 8.

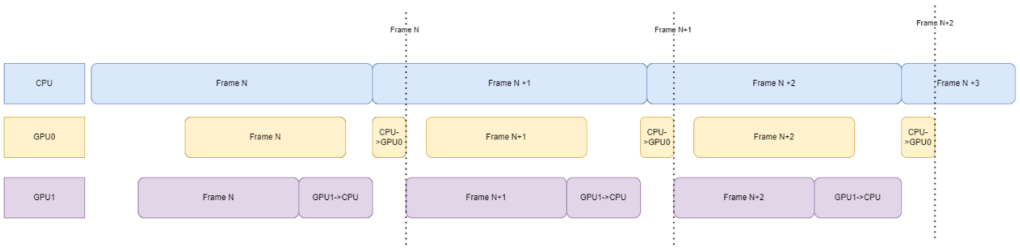

Figure 8 Explicitly control the Copy operation

It may be seen that the CPU will be interrupted, resulting in a long overall wait, so it is necessary to design a synchronization scheme to reduce the delay. The synchronization scheme is to design a Back Buffer for the rendering result of GPU1. By copying the rendering result asynchronously, the Back Buffer of GPU 1 is set to 3 frames like GPU0. GPU1→CPU processes the current frame, and CPU-GPU0→ processes the previous frame. In this way, the synchronization delay is reduced, at the cost that the test result needs a frame delay, as shown in Figure. 9.

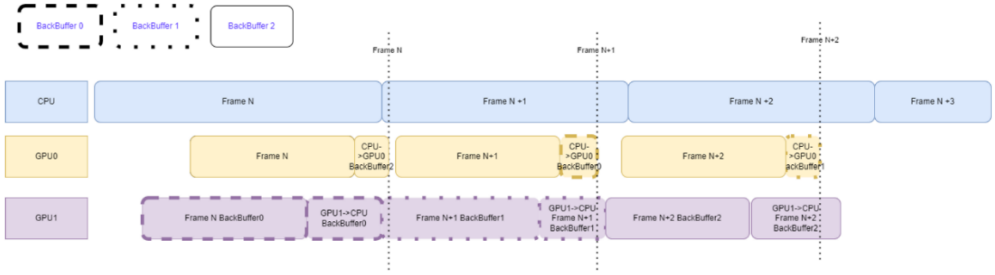

Figure 9 Synchronization scheme

2.7 Multi-thread rendering framework process

To sum up, the specific process of multithreading rendering framework is shown in Figure. 10.

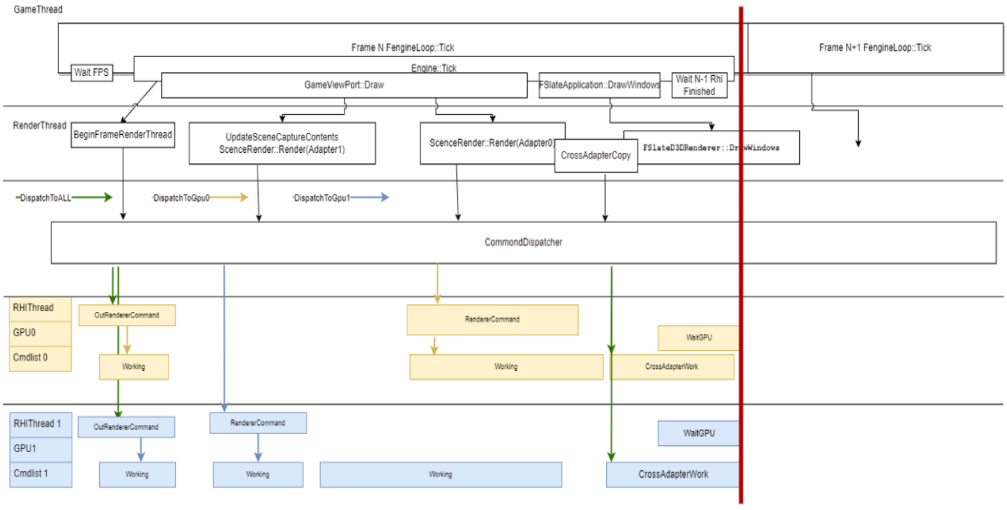

Figure 10 Multi-thread rendering framework

3. Simulations and Tests

Based on UE4, this paper implements a parallel real-time rendering framework for heterogeneous GPUs. Heterogeneous GPUs refer to calling graphics API through display, rather than relying on hardware limitations or drivers to achieve parallel rendering. However, when the workload of the two GPUs is unbalanced or the performance difference is large, there will be inevitable delays. In order to solve this problem, the framework of this

paper theoretically supports heterogeneous GPUs, but the current rendering task needs GPUs with similar performance. In the test, we used two Nvida RTX3070, and realized parallel rendering of two GPUs. The performance improved by 150-160%, and the parallel efficiency was close to 95%.

Specifically, we use two GPUs to render two images respectively, and realize the rendering within one frame. As shown in Figure. 11, compared with the previous sharing of Value on GDC 2016, dual GPU brings 30-35% performance improvement. The version we have implemented has further improved in performance.

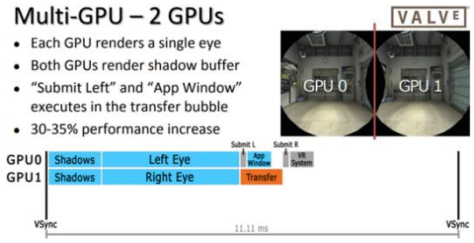

Figure 11 GPU rendering

3.1 Initial test

The test configuration in this paper mainly includes: CPU configuration i7-12700k, System Memory：64g DDR5 4800Mhz, GPU1 and GPU 2: RTX 3070.

The test scenario in this paper is a free resource in the mall: [SCANS] Abandoned Factory Buildings - Day/Night Scene]. Temporarily turn off Occlusion Culling in the project settings, because there is an exception in the implementation of Occlusion Query, which may cause wait blocking. Set the RT Fotmat RGBA8 with the resolution of 1920*1080 as shown in Figure. 12 below from another perspective of Scene Capture, and temporarily turn off the rendering of Scene Capture Particle Sprites.

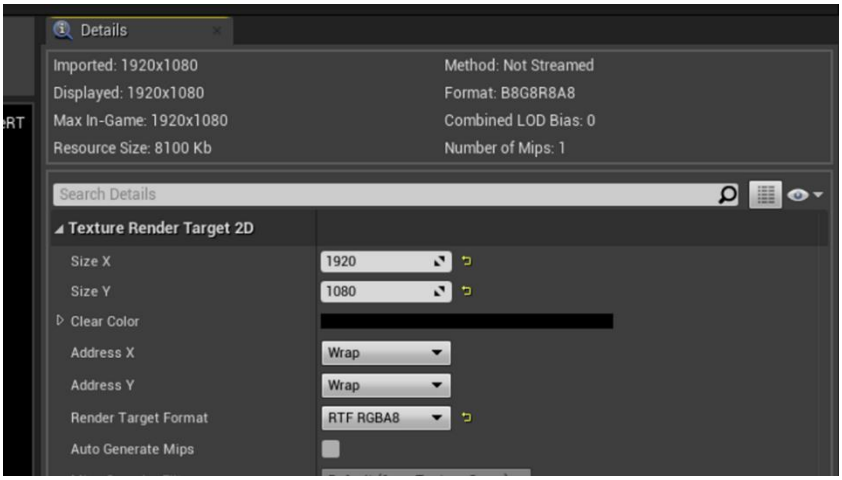

Figure 12 Scene Capture setup diagram

- Test results of Unmodified engine:

The unopened Scene Capture is compared with the opened Scene Capture, and the results are shown in Table 1:

Table 1 The comparison of RT turned on and not turned on of the unmodified engine

Test scenario	Total Time	GPU Time	Memory Usage	Video Memory Usage
RT not enabled	8.33ms	6.93ms	2076mb	2245mb
Turn on RT	12ms	11.9ms	2076mb	2245mb

- Test results of Current version of engine:

The unopened Scene Capture is compared with the opened Scene Capture, and the results are shown in Table 2:

Table 2 The comparison of RT turned on and not turned on of the current engine

Test scenario	Total Time	GPU Time	Memory Usage	Video Memory Usage	Notes
RT not enabled	8.33ms	7.64ms	1928Mb	1.6Gb	
Turn on RT GPU0 Rendering	12.58ms	12.27ms	1928Mb	1.6Gb	
Turn on RT GPU1 Rendering Backward read	13.00ms	20.7ms	2110Mb	1.6Gb	CrossCopy:5.35ms
Turn on RT GPU0 Rendering No back read	8.33ms	15.93ms	1725Mb	1.6Gb	CrossCopy:0ms Constant total time

According to the above test results of the unmodified engine and the test results of the current version of the engine, we can find that:

- For the Current version found under test:

(1) After starting cross-GPU readback, it takes as long as 5ms, which slows down the overall time, which is different from the expected performance. This is related to the need for Fence waiting synchronization in the current implementation of CrossMAdapterCopy operation.

(2) After the GPU is turned off to read back, the time per frame is exactly the same as that when the SceneCapture is not turned on.

- Reasons for the decrease of efficiency caused by CrossCopy:

(1) The operation itself is inefficient. Although API implementation is provided, the principle implementation should still be GPU0-CPU, and CPU-GPU1 should read back big RT twice in a row. It is not excluded that the API is inefficient.

(2) Another point may be that the current Fence synchronous waiting process interrupts parallel execution. Resulting in a long wait for the overall GPU0.

3.2 Update test

According to the test, we found that the format, size and bandwidth of RT map will cause great delay. Therefore, we set the delayed rendering result BackBuffer with the SceneCapture rendering result to 3 frames, which will delay the output of the result by 1 frame on the display RT. After adding delayed frame rendering, after repeated tests, it is found that cross-GPU copy operation itself is time-consuming.

In the format of 1920*1080 resolution RGBA16, the time consumption of a single GPU is 5ms, which is consistent with the previously observed CrossCopy consumption, while in the format of 1920*1080 resolution RGBA16, the time consumption of a single GPU is 5ms, which is consistent with the previously observed CrossCopy consumption. In the format of 1920*1080 resolution RGBA8, when the bandwidth pressure is reduced by half, when the output is delayed by one frame, the time to complete the whole rendering frame is within 9 ms.. Compared with 12.40ms when a single graphics card renders two RT's, it optimizes 3 ms-4 ms.

Then we test the distribution rate of 4K, and set the rendering resolution of the main ViewPort as 200%, the resolution of SceneCapture as 3840*2160, and the rendering of the main ViewPort alone as 16ms. The test results show that the efficiency is about 32ms； when Scene Capture is turned on and GPU0 is used for rendering. When Scene Capture is turned on to render with GPU1, the efficiency is about 25ms, and the extra time here actually comes from the copy bandwidth.

Carry out the CrossCopy unit test, as shown in Figure. 13, Figure. 14 and Figure. 15. We set the RT format as RGBA8, and it takes 2.5ms to copy when the resolution is 1920*1080. Setting the RT format to RGBA16, it takes 5ms to copy when the resolution is 1920*1080; Set the RT format to RGBA8, and it takes 10ms to copy when the resolution is 4k 3840*2160.

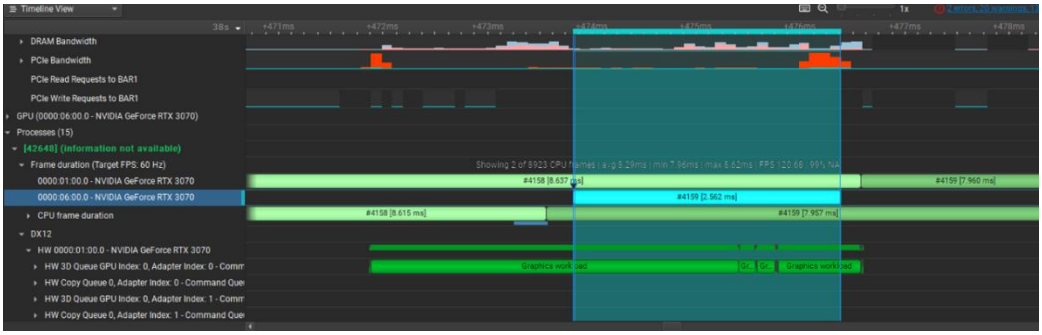

Figure 13 RT format setting as RGBA8 when the resolution is 1920*1080

Figure 14 RT format setting as RGBA 16 when the resolution is 1920*1080

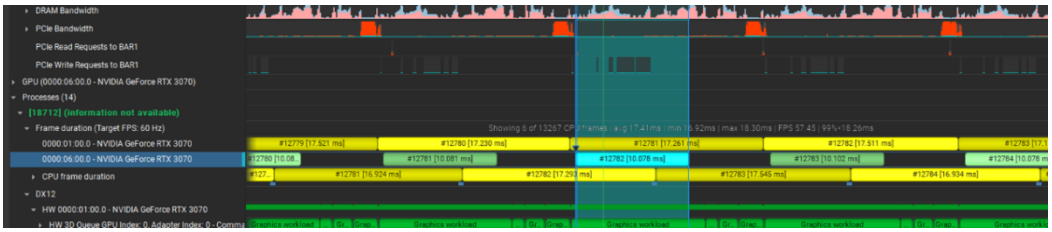

Figure 15 RT format setting as RGBA8 when the resolution is 4k 3840*2160

Based on the results in Table 3, we find taht the time consumption and bandwidth increase linearly.

Table 3 Time-consuming and bandwidth test results

RT Format	Resolution Ratio	Time-consuming
RGBA8	1920*1080	2.5ms
RGBA16	1920*1080	5ms
RGBA8	3840*2160	10ms

In Table 4, we can see the overall efficiency. After GPU1 is delayed by one frame, when the loads of two GPUs are close, there will be no consumption of waiting for the result of the current frame, and all the consumption is concentrated on the cross-GPU data copy.

Table 4 The comparison of overall efficiency test results

Test scenario	Total Time
Unmodified engine	
RT not enabled 1080p	8.33ms
Turn on RT GPU0 Rendering 1080p	12.58ms
Turn on RT GPU1 Rendering Backward read 1080p	13.00ms
Updated	
Turn on RT GPU1 Rendering Backward read 1080p	9ms
RT not enabled 4k	16ms
Turn on RT GPU0 Rendering 4k	32ms
Turn on RT GPU1 Rendering Backward read 4k	25ms

4. Conclusions

In this paper, we mainly design a novel multi-thread rendering framework in UE4 scene, which has a better improvement in performance. Under the current multi-GPU implementation, we have expanded the multithreading

architecture and added a new RHI Thread, which we call MRHI Thread, to handle the rendering instruction operation for the newly added GPU. In the test, we used two Nvida RTX3070, and realized parallel rendering of two GPUs. The performance improved by 150-160%, and the parallel efficiency was close to 95%. By moving the task dispatch mode to the render thread and specifying a separate RHIThread for each GPU to execute the CommandList, GPU parallelization efficiency is improved and wait time is reduced. Under our design, the GPU parallel efficiency can be improved by switching the dispatch mode in different stages and threads, and the organization and management can achieve more efficient rendering performance under the premise of ensuring task dependency.

For the future design, we need to expand more Adapter, which will involve great changes. Meanwhile, according to the logical distinction of different threads, it is necessary to make corresponding changes with the added new RHI Thread, which will be a heavy workload.

Acknowledgments

The paper was funded by the "New Generation Information and Communication Technology Innovation" program under the Beijing City Science and Technology Development Plan of Beijing City Science and Technology Commission. Name of Project: Development and Demonstration Application of PaaS Platform for Multi-graphics Card Distributed Rendering. Project Number: Z221100007722006.

References

[1]. Kim W Y . The Design and Implementation of an Elevator 3D Model Simulator Framework based on Unreal Engine. Jouranl of Information and Security, vol. 23, no. 2, pp. 67-74, 2023. DOI:10.33778/kcsa.2023.23.2.067.

[2]. Torres-Ferreyros C M , Festini-Wendorff M A , Shiguihara-Juarez P N . Developing a videogame using unreal engine based on a four stages methodology. *2016 IEEE ANDESCON*, Arequipa, Peru, 2016, pp. 1-4. DOI:10.1109/ANDESCON.2016.7836249.

[3]. Ratican J , Hutson J , Wright A . A Proposed Meta-Reality Immersive Development Pipeline: Generative AI Models and Extended Reality (XR) Content for the Metaverse. Journal of Intelligent Learning Systems and Applications, vol. 15, no. 1, pp. 24-35, 2023.

[4]. Lewis J , Brown D , Cranton W ,et al. Simulating visual impairments using the Unreal Engine 3 game engine. *2011 IEEE 1st International Conference on Serious Games and Applications for Health (SeGAH)*, Braga, Portugal, 2011, pp. 1-8. DOI:10.1109/SeGAH.2011.6165430.

[5]. Herrlich M , Meyer R , Malaka R ,et al. Development of a Virtual Electric Wheelchair - Simulation and Assessment of Physical Fidelity Using the Unreal Engine 3. 2010 Entertainment Computing ICEC, Springer Berlin Heidelberg, pp.286-293. DOI:10.1007/978-3-642-15399-0_29.

[6]. Su, Y.; Seng, K.P.; Smith, J.; Ang, L.M. Efficient FPGA Binary Neural Network Architecture for Image Super-Resolution. Electronics, vol.13, no.2, pp. 1-16, 2024. https://doi.org/10.3390/electronics13020266

[7]. Deng,Siyu,Che,et al. GPU-based 3D model rendering of chinese ink painting. Computer Aided Drafting, Design and Manufacturing, vol.38, No.351, pp.3-8, 2017. DOI:CNKI:SUN:CADD.0.2017-01-001.

[8]. Wang P, Yu Z. RayBench: An Advanced NVIDIA-Centric GPU Rendering Benchmark Suite for Optimal Performance Analysis. *Electronics*, vol. 12, no.19, pp.4124, 2023. https://doi.org/10.3390/electronics12194124

[9]. Whencheng Z , Chengyi W . Development of Real-time Rendering Technology for High-Precision Models in Autonomous Driving. Arxiv.org, 2023. DOI:10.48550/arXiv.2302.00291.

[10]. Chen Z , Ma Y , Wen H ,et al. Sonographic demonstration of sulci and gyri on the convex surface in normal fetus using 3D-ICRV rendering technology. Ultraschall in der Medizin, vol. 44, no. 6, pp.284-295, 2023. DOI:10.1055/a-2122-6182.

[11]. Liubogoshchev M, Korneev E, Khorov E. EVeREst: Bitrate Adaptation for Cloud VR. *Electronics*. vol.10, no.6, pp.678, 2021. https://doi.org/10.3390/electronics10060678

[12]. G. -H. Lin, C. -H. Chang, M. -C. Chung and Y. -C. Fan. Self-driving Deep Learning System based on Depth Image Based Rendering and LiDAR Point Cloud. *2020 IEEE International Conference on Consumer Electronics - Taiwan (ICCE-Taiwan)*, Taoyuan, Taiwan, 2020, pp. 1-2. DOI:10.1109/ICCE-Taiwan49838.2020.9258010.

[13]. Bin Mohamad Saifuddin MR, Ramasamy TN, Qi Tong WP. Design and Control of a DC Collection System for Modular-Based Direct Electromechanical Drive Turbines in High Voltage Direct Current Transmission. *Electronics*. vol. 9, no. 3, pp.493, 2020. https://doi.org/10.3390/electronics9030493

[14]. Deng S., Che, et al. GPU-based 3D model rendering of chinese ink painting.Computer Aided Drafting, Design and Manufacturing, vol.38, no.351, pp.3-8, 2017. DOI:CNKI:SUN:CADD.0.2017-01-001.

[15]. Weng X , Guo B . A method of airport runway dataset construction for the visual detection algorithm. 2022 International Conference on Intelligent Equipment and Data Processing, IOP Publishing Ltd, 2435, 2023. DOI:10.1088/1742-6596/2435/1/012016.

[16]. Dong Y , Peng C . Multi-Gpu Multi-Display Rendering of Extremely Large3d Environments. SSRN Electronic Journal, vol. 39, pp. 6473-6489, 2022. DOI:10.2139/ssrn.4144322.

[17]. Zheng A , Liu T , Song Y ,et al. Minimum dose path planning method for virtual nuclear facilities based on navigation mesh. Annals of nuclear energy, vol. 195, pp. 1-12, 2024. DOI: 10.1016/j.anucene.2023.110104.

[18]. Kruger J , Westermann R . Acceleration techniques for GPU-based volume rendering. *IEEE Visualization, 2003. VIS 2003.*, Seattle, WA, USA, 2003, pp. 287-292, doi: 10.1109/VISUAL.2003.1250384.

[19]. Charles,Loop,Jim,et al. Real-time GPU rendering of piecewise algebraic surfaces. ACM Transactions on Graphics, vol. 25, no. 3, pp.664-670, 2026. DOI:10.1145/1179352.1141939.

[20]. Pajot A , Barthe L , Paulin M ,et al. Combinatorial Bidirectional Path-Tracing for Efficient Hybrid CPU/GPU Rendering. Computer Graphics Forum, vol. 30, no. 2, pp.315-324, 2011. DOI:10.1111/j.1467-8659.2011.01863.x.

[21]. Zhang H , Choi J . High quality GPU rendering with displaced pixel shading. Proceedings of SPIE - The International Society for Optical Engineering, vol. 6141, pp.1-9, 2006. DOI:10.1117/12.652307.

[22]. Hu W . A New Rendering Technology of GPU-Accelerated Radiosity. Journal of Computer Research and Development, vol. 42, no. 6, pp.945-950, 2005. DOI:10.1360/crad20050607.