Understanding DirectX* Multithreaded Rendering Performance by Experiments

By Sheng Guo (Intel Corporation)

Abstract

The renderer of a game engine is often a performance bottleneck from the CPU side. Adding multithreading to the rendering step is an effective means to address the performance issue without losing content details. In this article, the performance of DirectX 11* and DirectX 12* multithreading APIs is evaluated on advanced multicore processors and typical graphics hardware, the primary factors affecting multithreaded rendering performance are analyzed, and the relevant optimization methods are explored as well.

1. Introduction

In the past ten years, PC processors have greatly improved. 4-core CPUs have become the mainstream configuration in the PC game market today. CPUs with more cores are rising in the market share. This trend is anticipated to continue, and in the next few years 6-core CPUs will likely grow in popularity for gamers.

Unfortunately, the renderers of most game engines are still single-threaded by now, which often results in a performance bottleneck on the CPU, preventing multicore computing resources from being leveraged to improve performance or enrich visual contents. For example, when rendering large-scale outdoor scenes with many visible objects, the single-threaded renderer often leads to individual CPU cores running fully loaded while other cores remain relatively idle, and the performance below the playable frame rate.

DirectX* started from DirectX 11 to formally support calling Direct3D*(D3D) application programming interfaces (APIs) in multiple threads. DirectX 11 multithreading supports two types of device contexts: immediate context and deferred context (Figure 1). Different deferred contexts can be used in different threads simultaneously, generating the command-lists that are to be executed in the immediate context. This multithreading strategy allows complex scenes to be broken up into concurrent tasks [1].

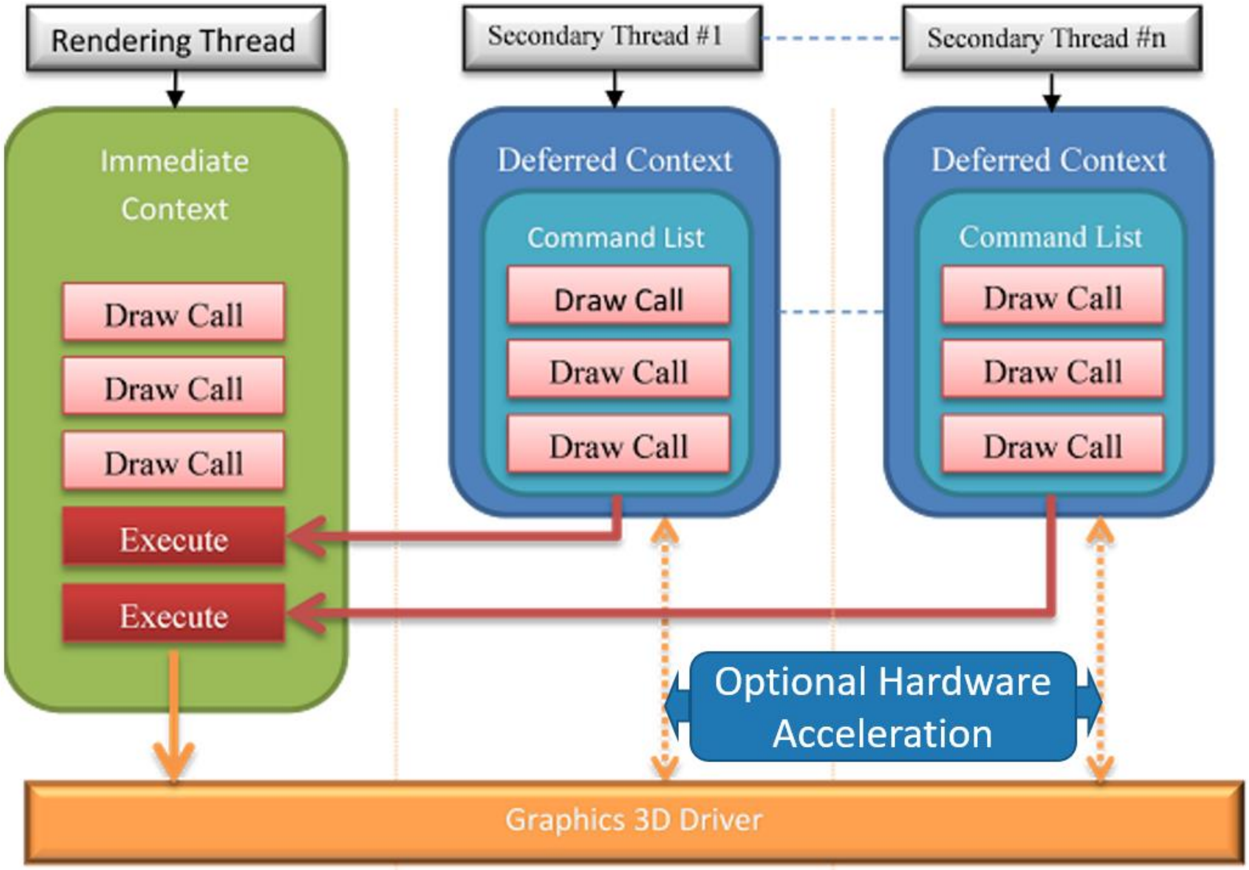
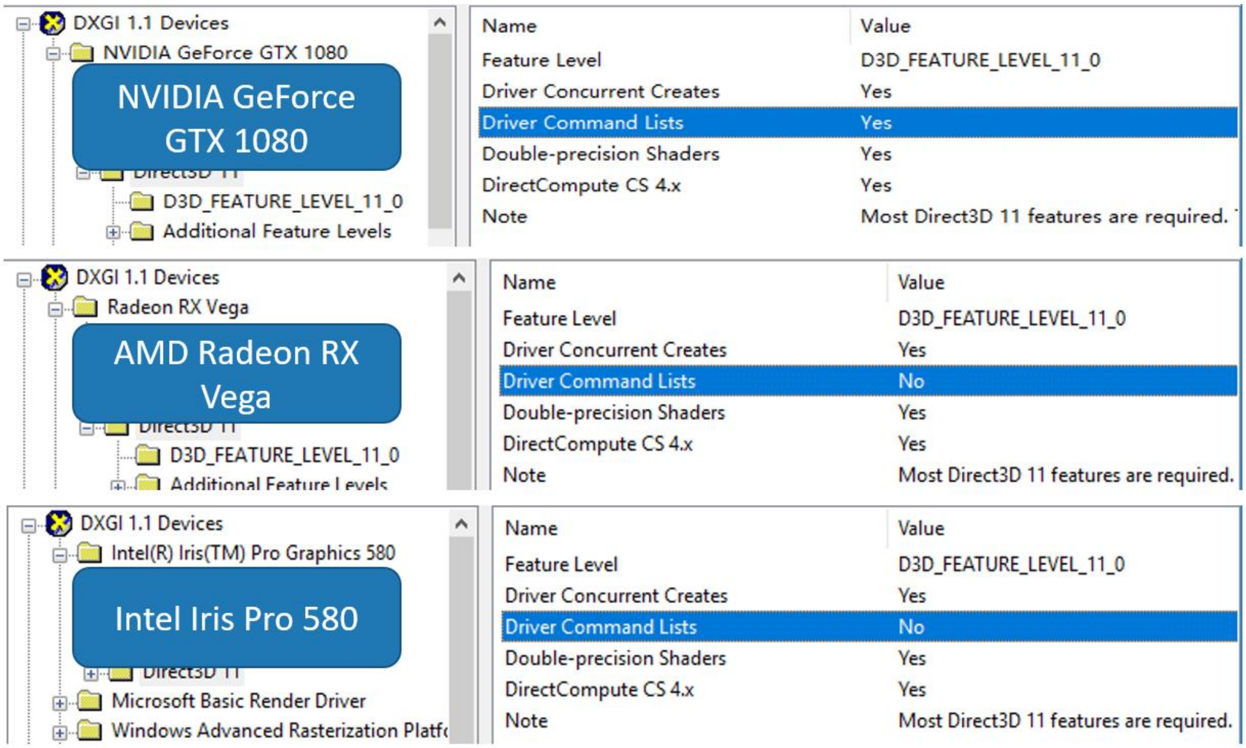


Figure 1. DirectX 11 multithreading model

Although DirectX 11 multithreading is supported by D3D runtime, the hardware acceleration for it is optional [2]. With hardware acceleration support, a part of the driver load can be parallelized along with the command-list build. Figure 2 shows the hardware acceleration capability in different graphics devices. Although all the graphics devices support “Driver Concurrent Creates”, only NVIDIA* graphics supports “Driver Command Lists”.



DXGI 1.1 Devices
NVIDIA GeForce GTX 1080
NVIDIA GeForce GTX 1080
Direct3D 11
D3D_FEATURE_LEVEL_11_0
Additional Feature Levels

Name	Value
Feature Level	D3D_FEATURE_LEVEL_11_0
Driver Concurrent Creates	Yes
Driver Command Lists	Yes
Double-precision Shaders	Yes
DirectCompute CS 4.x	Yes
Note	Most Direct3D 11 features are required.

DXGI 1.1 Devices
Radeon RX Vega
AMD Radeon RX Vega
Direct3D 11
D3D_FEATURE_LEVEL_11_0
Additional Feature Levels

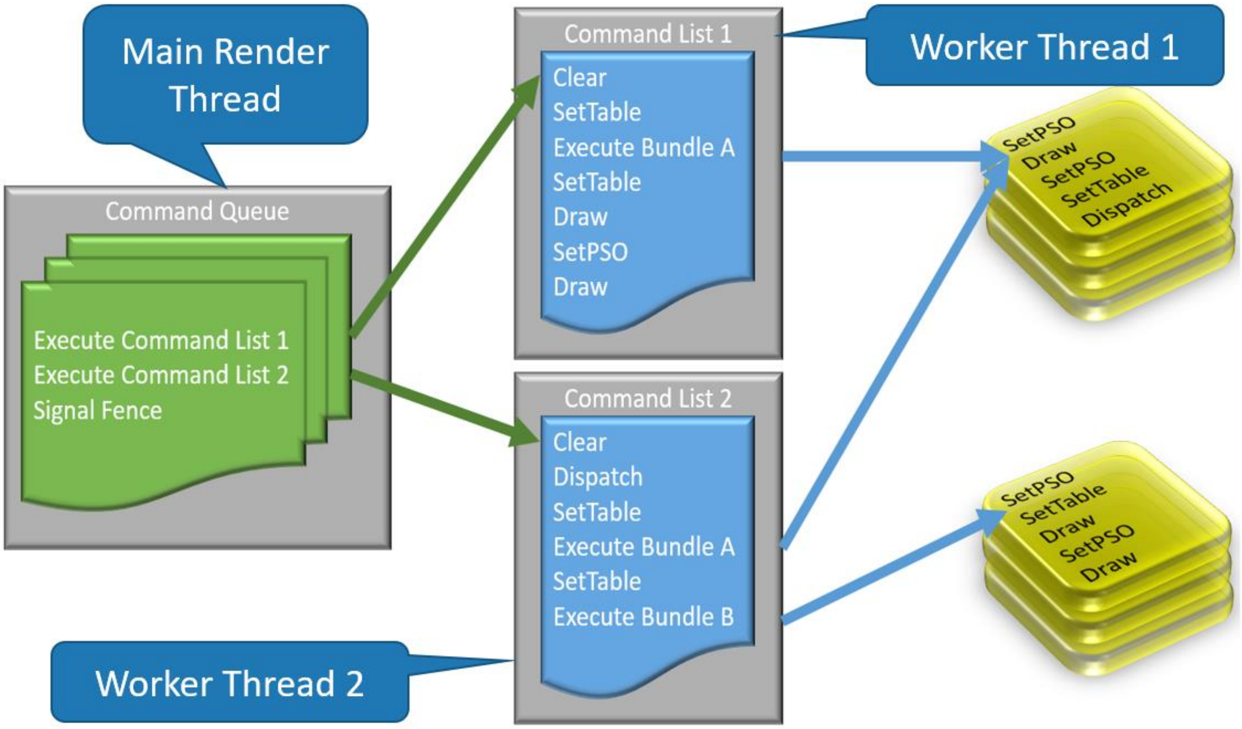
Name	Value
Feature Level	D3D_FEATURE_LEVEL_11_0
Driver Concurrent Creates	Yes
Driver Command Lists	No
Double-precision Shaders	Yes
DirectCompute CS 4.x	Yes
Note	Most Direct3D 11 features are required.

DXGI 1.1 Devices
Intel(R) Iris(TM) Pro Graphics 580
Intel Iris Pro 580
Direct3D 11
Microsoft Basic Render Driver
Windows Advanced Rasterization Platf

Name	Value
Feature Level	D3D_FEATURE_LEVEL_11_0
Driver Concurrent Creates	Yes
Driver Command Lists	No
Double-precision Shaders	Yes
DirectCompute CS 4.x	Yes
Note	Most Direct3D 11 features are required.

Figure 2. Optional hardware acceleration for DirectX 11 multithreading

DirectX 12 multithreading is improved much by significantly reducing API calling overhead. It eliminates the concept of device context of DirectX 11, instead using the command-list to invoke D3D APIs and then submit command-lists to the GPU through the command queue (Figure 3). All the DirectX 12 graphics hardware supports hardware acceleration for DirectX 12 multithreading.



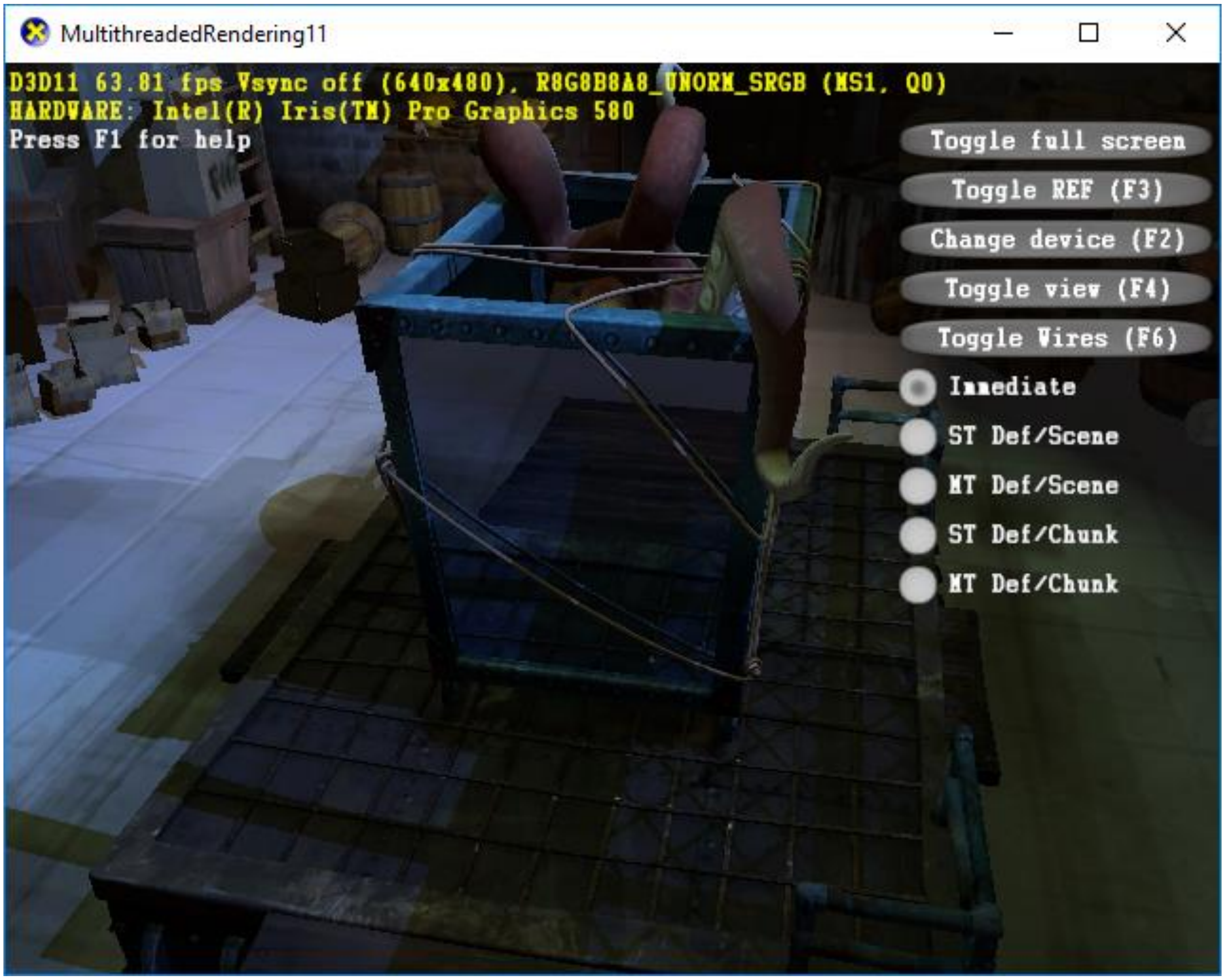
*Reference to "Direct3D 12 API Preview" presented by Max McMullen, Microsoft

Figure 3. DirectX 12 multithreading model

Both DirectX 11 and DirectX 12 supports make multithreading an attractive alternative to address the performance bottleneck of rendering. However, due to the complexity of both the renderer and multithreaded programming, it's necessary to carefully understand the characteristics of DirectX multithreading in detail to avoid the performance penalty of misusing. On multithreaded rendering performance, developers often concern about its actual pros and cons, multicore scalability and relevant key influence factors, and practical multithreading methods, among other things. Well understanding these questions will lead to more efficient implementation of a properly configured multithreaded renderer, which is the focus of the experiments detailed below.

2. Performance Evaluation of DirectX Multithreading APIs

The workloads to evaluate the performance of multithreading APIs of DirectX11 and DirectX12 are from DirectX official samples, which are respectively "MultithreadedRendering11" (Figure 4) in DirectX SDK (June 2010) [3] and "D3D12Multithreading" (Figure 5) in DirectX 12 Graphics samples [4]. Both samples are intended to demonstrate the advantages and methods of DirectX multithreaded rendering, which render the same "squid room" scene with thousands of draw calls per frame and are both CPU-bound on the test platforms. However, due to some programming tricks or defects, both samples do not comprehensively reflect the actual performance pros and cons of calling DirectX APIs in multithread threads. Thus, they are optimized and extended for the performance evaluation purpose.



```
D3D11 63.81 fps Vsync off (640x480), R8G8B8A8_UNORM_SRGB (MS1, Q0)
HARDWARE: Intel(R) Iris(TM) Pro Graphics 580
Press F1 for help
```

Figure 4. “MultithreadedRendering11” sample in DirectX SDK (June 2010) [3]

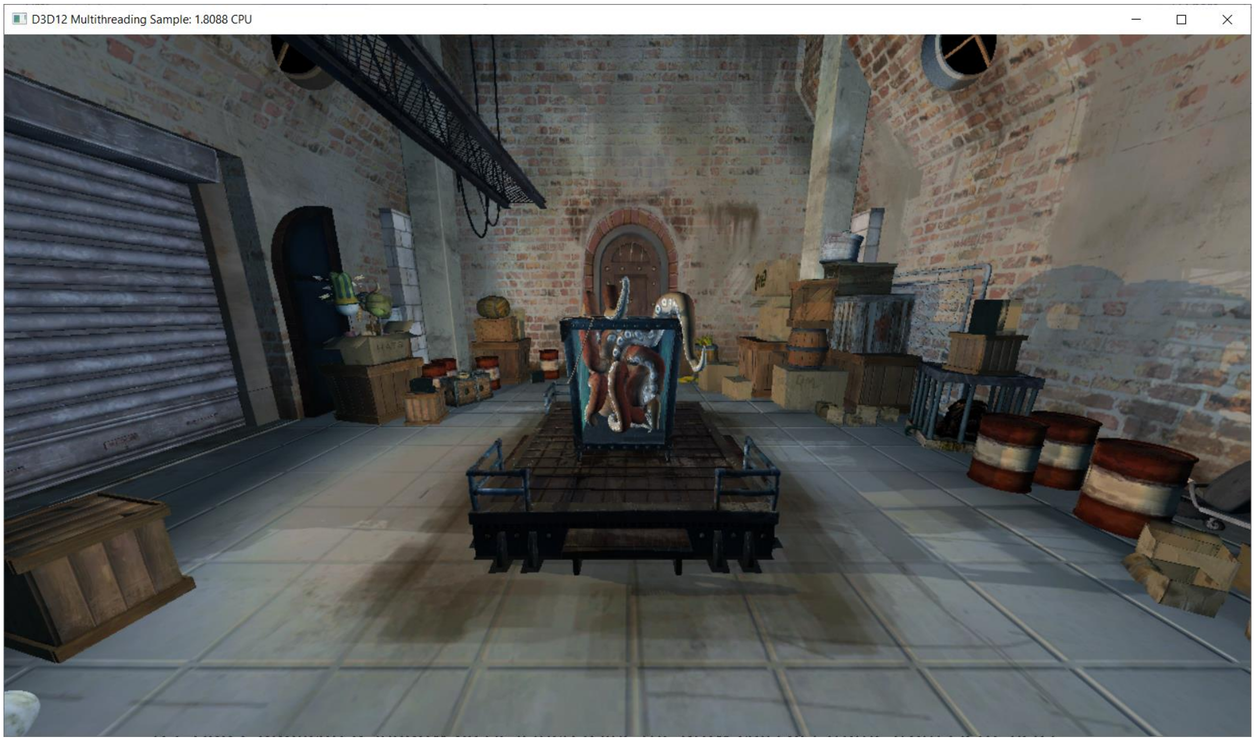


Figure 5. “D3D12Multithreading” in DirectX 12 Graphics samples [4]

This performance evaluation of DirectX multithreading APIs was conducted on three platforms with different configurations, as shown in Table 1. The configurations include new-generation multicore CPUs and the graphics devices from three major suppliers. To avoid GPU bottlenecks, the graphics devices used are at the mid to high end of the spectrum of products for each brand, and all support DirectX 11 and DirectX 12. To measure the multicore scalability of DirectX multithreading performance, the 10-core Intel® Core™ i7-6950X CPU is used to facilitate varying active CPU core count via BIOS. Since i7-6950X CPU doesn’t contain processor graphics, a 4-core Intel® Core™ i7-6770HQ CPU is also used for evaluation.

Config	Platform A	Platform B	Platform C
CPU	Intel® Core™ i7-6950X CPU @ 3.00GHz	Intel Corei7-6950X CPU @ 3.00GHz	Intel Core i7-6770HQ CPU @ 2.60GHz
Memory	4x8GB RAM	4x8GB RAM	2x8GB RAM
Graphics	NVIDIA* GeForce GTX 1080	Radeon* RX Vega 64	Intel® Iris™ Pro Graphics 580

Driver	22.21.13.8494	22.19.677.257	22.20.16.4749
OS	Windows® 10 Enterprise 64-bit (10.0, Build 17134)	Windows 10 Enterprise 64-bit (10.0, Build 17134)	Windows 10 Enterprise 64-bit (10.0, Build 17134)

Table 1. Test platform configurations

2.1 DirectX 11 Experiments

“MultithreadedRendering11” (Figure 4) is a sample of DirectX 11 multithreaded rendering. To objectively evaluate the performance and overhead of calling DirectX 11 APIs in multiple threads, it is optimized by reducing thread synchronization overhead and application-layer load. The optimization methods are detailed in section 3. The optimized and rebuilt workload is entitled “MTR11_Benchmark”, which implements five rendering modes, shown in Table 2, with more than 4k draw calls per frame.

Mode	**Description**
ST-Immediate	Single-threaded. The main thread uses immediate context to render all scenes.
MT-Scene	Multithreaded. Each scene is assigned a deferred context and a worker thread to render it. The main thread finally submits the command-lists of all deferred contexts. (command-list submission # per frame = scene #)
ST-Scene	Single-threaded. The serialized version of MT-Scene. Main thread sequentially executes all tasks of worker threads of MT-Scene.
MT-Chunk	Multithreaded. The meshes of each scene are evenly divided into N (Core#-1) chunks. Each chunk is assigned a deferred context and a worker thread to render it. The main thread submits the command-lists of all deferred contexts at the end of rendering each scene. (command-list submission # per frame = (Core#-1) * “scene #”)

ST-Chunk	Single-threaded. The serialized version of MT-Chunk. Main thread sequentially executes all tasks of worker threads of MT-Chunk.

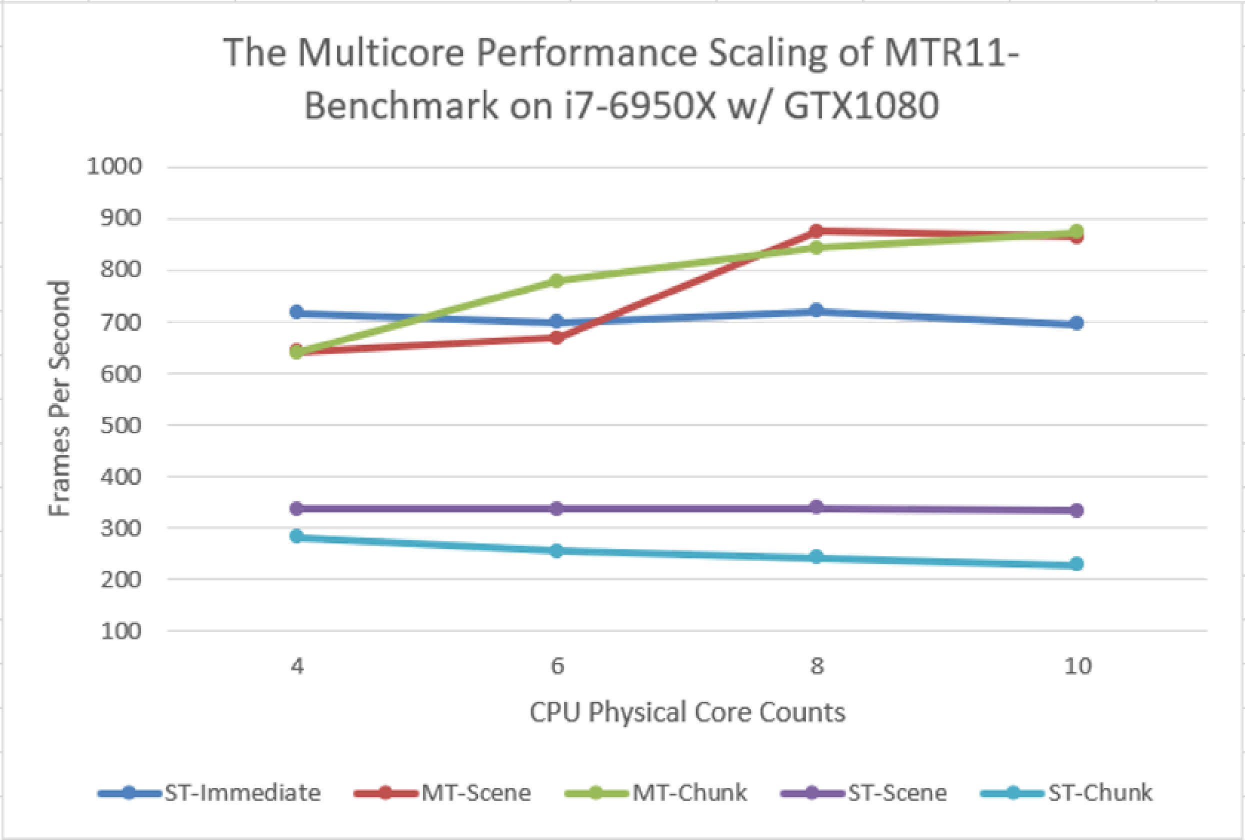
Table 2. The rendering modes of MTR11-Benchmark

The average frame rate of MTR11_Benchmark running on different CPU and GPU configurations are shown in Figure 6. The data revealed some surprising findings.

First, comparing the single-thread mode “ST-Immediate” with the multithread modes “MT-Scene” and “MT-Chunk”, Figure 6 shows that: on all test platforms, when with less CPU cores, single-threaded immediate rendering (“ST-Immediate”) has better performance than multithreaded deferred rendering (“MT-Scene” and “MT-Chunk”). However, when with more CPU cores, the performance of multithreaded deferred rendering is much better on Nvidia graphics, and a little better on Intel graphics, but worse on AMD graphics than single-threaded immediate rendering. Note that Nvidia graphics support hard acceleration for DirectX 11 multithreading.

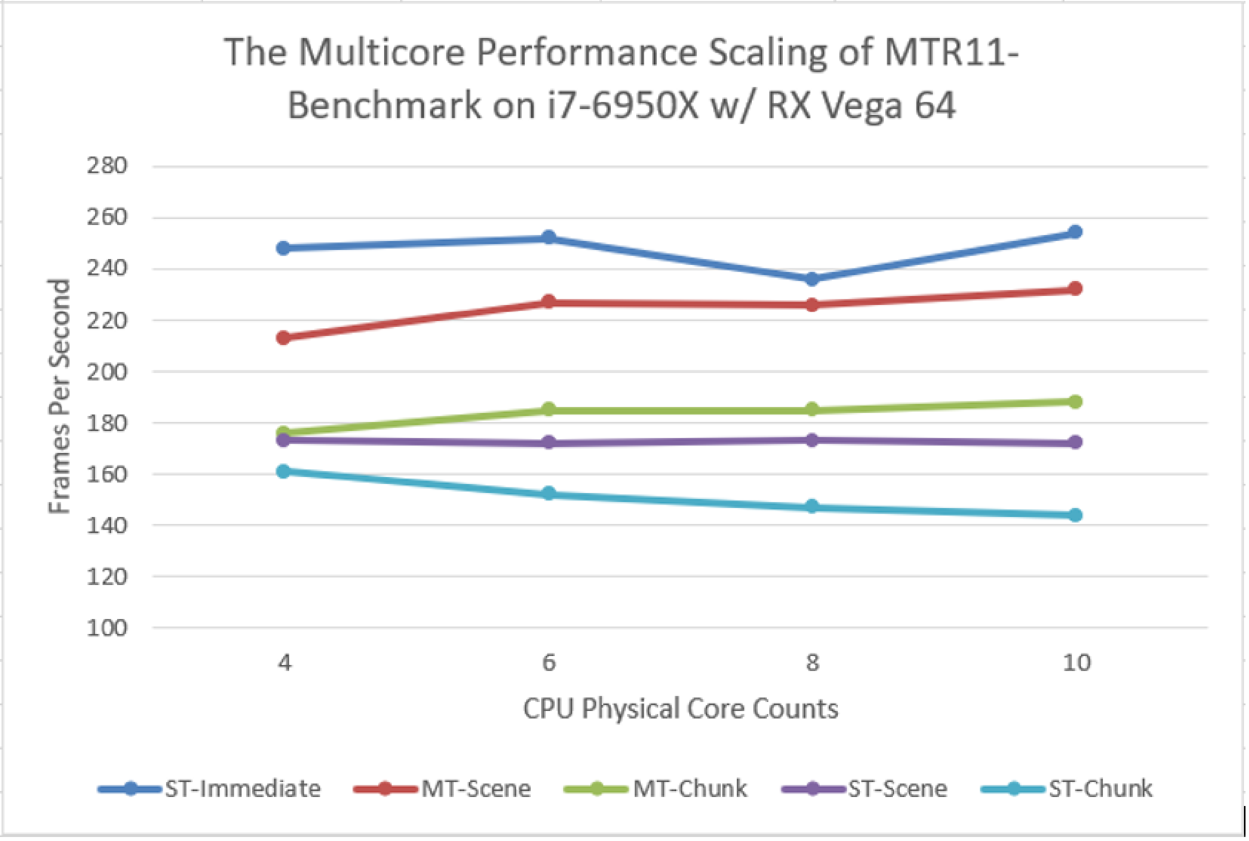
Secondly, comparing between two multithread modes, “MT-Scene” and “MT-Chunk”, Figure 6 shows that: on all test platforms, the scene-based multithreading (“MT-Scene”) generally has better performance than the chunk-based multithreading (“MT-Chunk”), except for on Nvidia graphics where the chunk-based has better performance in some core counts (Figure 6.a). Note that Nvidia graphics supports hard acceleration for DirectX 11 multithreading, and the scene-based has less command-list submissions than the chunk-based (Table 2).

Finally, comparing all single-thread modes, “ST-Immediate”, “ST-Scene” and “ST-Chunk”, to evaluate the cost of calling D3D11 APIs in different contexts and strategies without multithread overhead interference. Figure 6 shows that: on all test platforms, calling D3D APIs in immediate context (“ST-Immediate”) has much better performance than calling them in deferred context (“ST-Scene” and “ST-Chunk”). Furthermore, on all CPU core counts, calling D3D APIs in deferred context has better performance in the scene-based mode (“ST-Scene”) than in the chunk-based mode (“ST-Chunk”) which degrades performance with increasing CPU core counts. The rendering performance looks like negatively correlated with the number of command-list submissions.



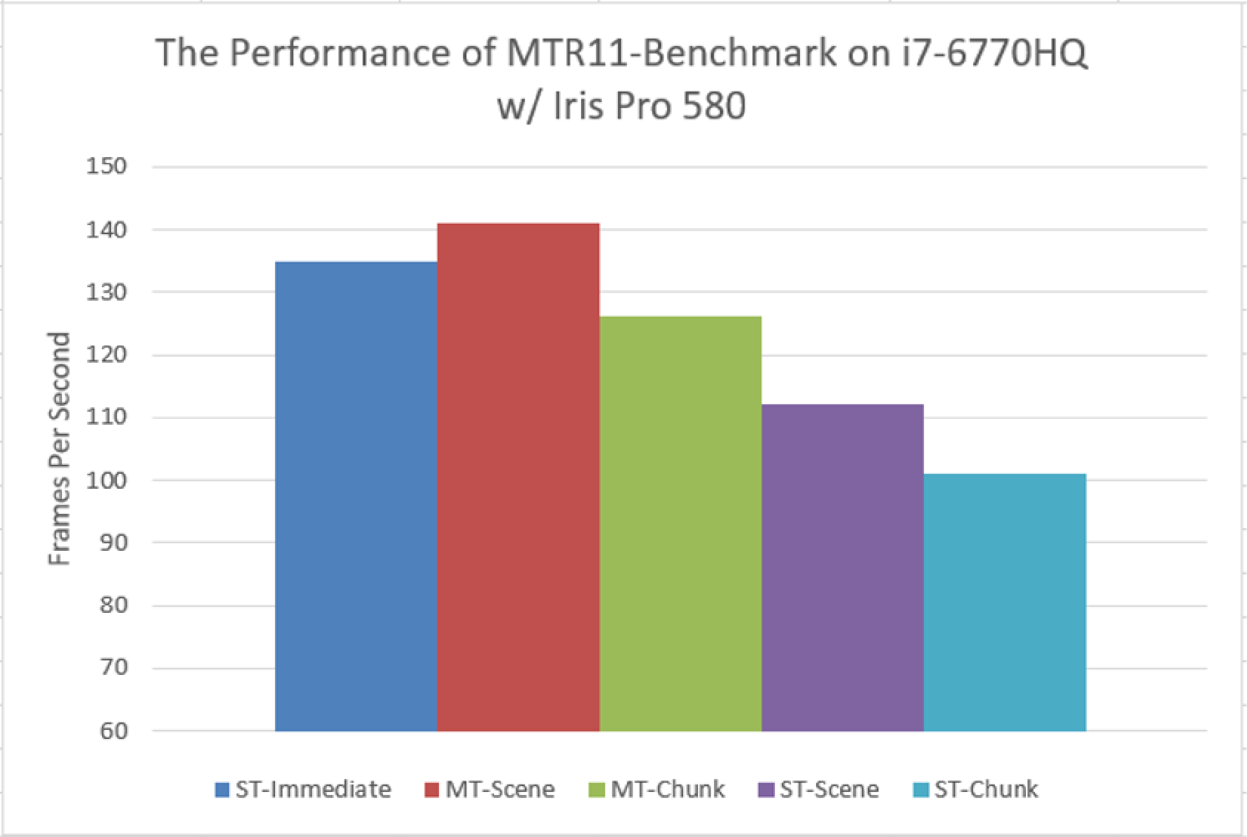
Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products. Performance results are based on testing by Intel engineer as of Jan 15th, 2019 and may not reflect all publicly available security updates. Configuration: Intel® Core™ i7-6950X CPU @ 3.00GHz, 32GB RAM, Windows® 10 Enterprise 64-bit (10.0, Build 17134).

(6.a) Test result on Platform A



Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products. Performance results are based on testing by Intel engineer as of Jan 15th, 2019 and may not reflect all publicly available security updates. Configuration: Intel® Core™ i7-6950X CPU @ 3.00GHz, 32GB RAM, Windows® 10 Enterprise 64-bit (10.0, Build 17134).

(6.b) Test result on Platform B



Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products. Performance results are based on testing by Intel engineer as of Jan 15th, 2019 and may not reflect all publicly available security updates. Configuration: Intel Core i7-6770HQ CPU @ 2.60GHz, 32GB RAM, Windows® 10 Enterprise 64-bit (10.0, Build 17134).

(6.c) Test result on Platform C

Figure 6. The performance of MTR11-Benchmark on different CPU and GPU configurations

Based on the test results above, a few performance characteristics of DirectX 11 multithreading can be inferred:

1. DirectX 11 multithreading involves bigger overhead. Calling D3D APIs in deferred context and submitting command-lists yield more overhead than calling APIs in immediate context. And the more command-lists submitted, the more overhead generated.

2. DirectX 11 multithreading doesn’t necessarily improve performance, especially when purely calling D3D APIs. It is because of not only the significant overhead generated by deferred contexts, but also that not all of the extra overhead can be amortized by multithreading, for example, serialized command-list submission. Thus, it’s possible that multithreaded deferred rendering may result in a longer total rendering time than single-threaded immediate rendering.

3. The hardware acceleration significantly benefits to DirectX 11 multithreading performance. The drive load accounts for a notable portion of the overall rendering load (Figure 12). Hardware acceleration essentially parallelizes a part of the driver load, so shortens the rendering time.

2.2 DirectX 12 Experiments

“D3D12Multithreading” (Figure 5) is a sample of DirectX 12 multithreaded rendering. Compared to its DirectX 11 counterpart “MultithreadedRendering11”, it simplifies the rendering logic so much that the rendering procedure almost only calls DirectX 12 APIs. And it just implements two modes: single-thread and multithread. To objectively evaluate the performance and overhead of calling DirectX 12 APIs in multiple threads, the sample was optimized and extended with the following functions:

1. The number of worker threads, which was fixed to four in the original sample, can be scaled according to the active CPU core count. This modification helps us evaluate multicore performance scaling for DirectX 12 multithreading.

2. Adding two duplicated shadow passes, having the number of draw calls per frame increase from 2k to 4k as many as the “MTR11_Benchmark”. This modification not only offers more CPU load to test multicore scaling, but also enables us to compare multithreading performance between DirectX 12 and DirectX 11.

3. Minimizing the synchronization between the main thread and worker threads, by having the main thread wait for all command-lists to complete before submitting them in a batch. This modification helps us isolate the factors affecting DirectX 12 multithreading performance.

4. Implementing a single-threaded mode with only one command-list. The original single-thread mode is actually the serialized version of the multithread mode with many command-lists, which results in the single-thread mode has much worse performance than the multithread mode. However, the actual single-thread mode may just need one or a small amount of command-lists.

5. Implementing 5 rendering modes (Table 3) for evaluating the performance of different DirectX 12 multithreading methods.

Mode	**Description**
ST-One	Single-threaded. Main thread only uses one command-list to render all passes.
MT-Pass	Multithreaded. Each pass is assigned a command-list and a worker thread to render it. The main thread finally submits all command-lists in a batch. (command-list submission # per frame = pass #)

ST-Pass	Single-threaded. The serialized version of MT-Pass. Main thread sequentially executes all tasks of worker threads of MT-Pass.
MT-Chunk	Multithreaded. The meshes drawn in each pass are evenly divided into N (Core# -1) chunks. Each chunk is assigned a command-list and a worker thread to render it. The main thread finally submits the command-lists of all passes in a batch. (command-list submission # per frame = (core#-1) * “pass #”)
ST-Chunk	Single-threaded. The serialized version of MT-Chunk. Main thread sequentially executes all tasks of worker threads of MT-Chunk.

Table 3. The rendering modes of MTR12_Benchmark

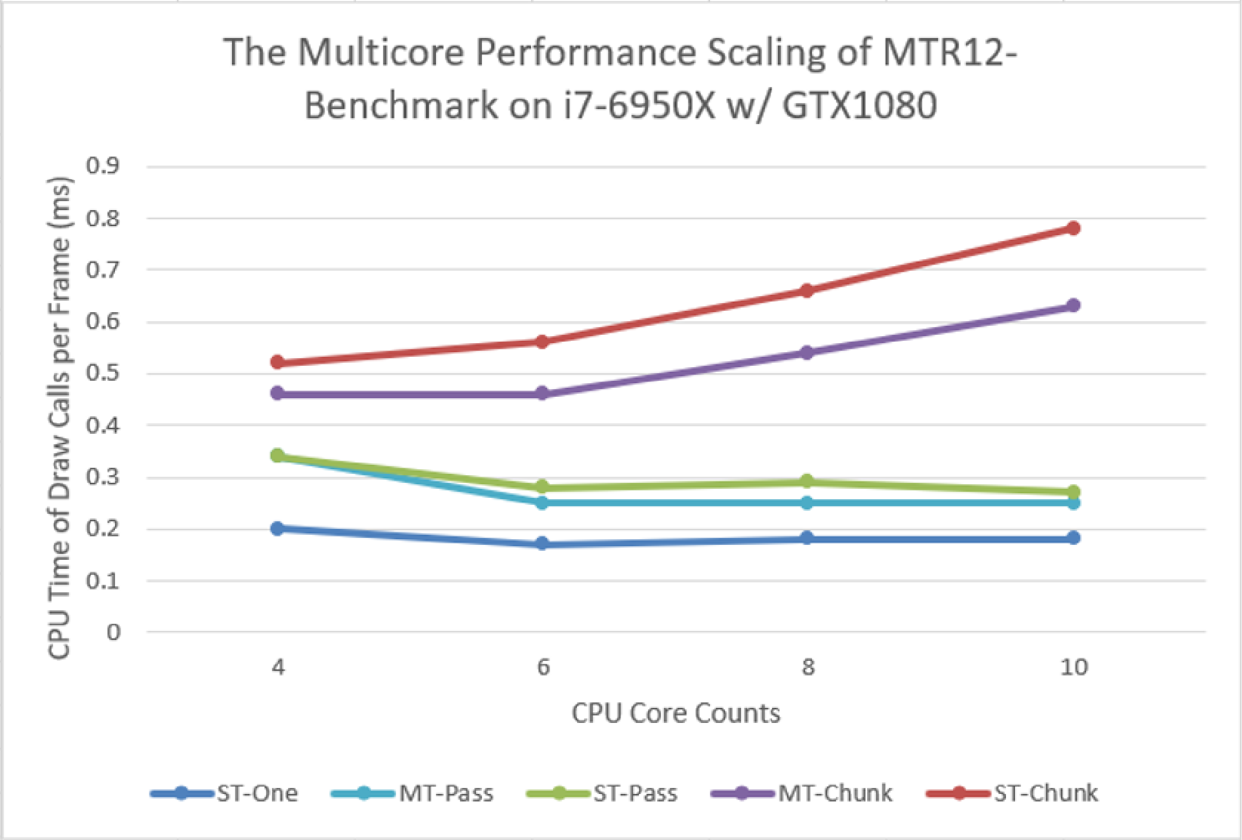
The modified “D3D12Multithreading” sample is referred to as “MTR12_Benchmark”. Running MTR12_Benchmark on different CPU and GPU configurations, the average CPU time of calling D3D APIs per frame are shown in Figure 7, resulting in a couple of interesting observations.

First, comparing the multithread modes “MT-Pass” and “MT-Chunk” with the single-thread mode “ST-One”, Figure 7 shows that: on most test platforms, the single-threaded rendering with one command-list (“ST-One”) generally has better performance than the multithreaded rendering with multiple command-lists (“MT-Pass” and “MT-Chunk”), except that on AMD graphics “MT-Pass” is a little faster than “ST-One” (Figure 7.b).

Then, comparing between two multithread modes, “MT-Pass” and “MT-Chunk”, Figure 7 shows that: the pass-based multithreading (“MT-Pass”) always has better performance than the chunk-based multithreading (“MT-Chunk”) which degrades performance with increasing CPU core counts. Note that the pass-based multithreading has fixed numbers of threads and command-list submissions which however vary with the number of CPU cores in the chunk-based multithreading (Table 3).

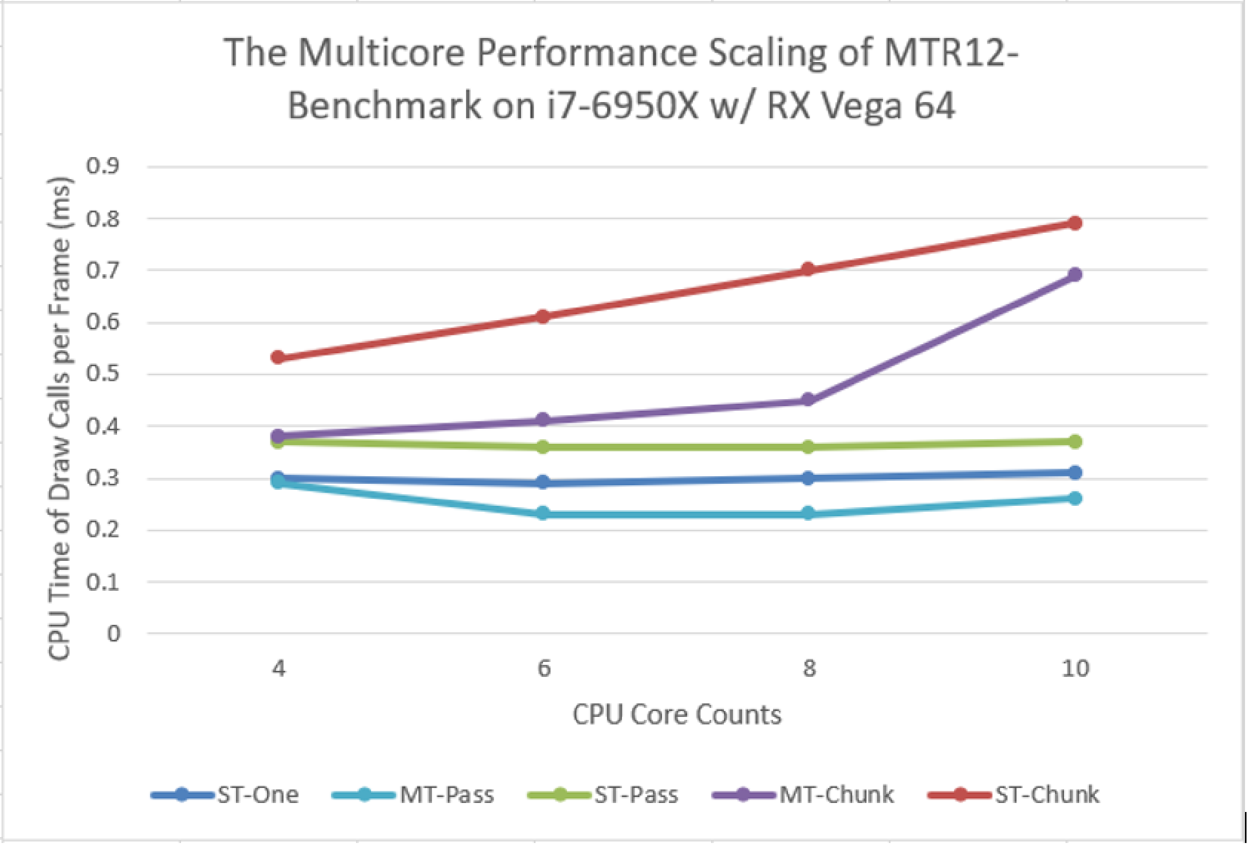
Also, comparing all single-thread modes (“ST-One”, “ST-Pass”, “ST-Chunk”), to evaluate the cost of calling D3D12 APIs with different strategies without multithreading overhead interference. Figure 7 shows that: on all test platforms, calling D3D APIs with one command-list per frame (“ST-One”) has the best performance. Calling D3D APIs with the mode submitting a small number of command-lists per frame (“ST-Pass”) has the poor performance. And Calling D3D APIs with the mode submitting more command-lists per frame (“ST-Chunk”) has the worst performance which will further degrades with increasing CPU core counts.

Lastly, comparing the test results of MTR11_Benchmark (Figure 6) and MTR12_Benchmark (Figure 7). It’s obvious that rendering with D3D12 APIs has much better performance than rendering with D3D 11 APIs even though there are some implementation differences between both workloads.



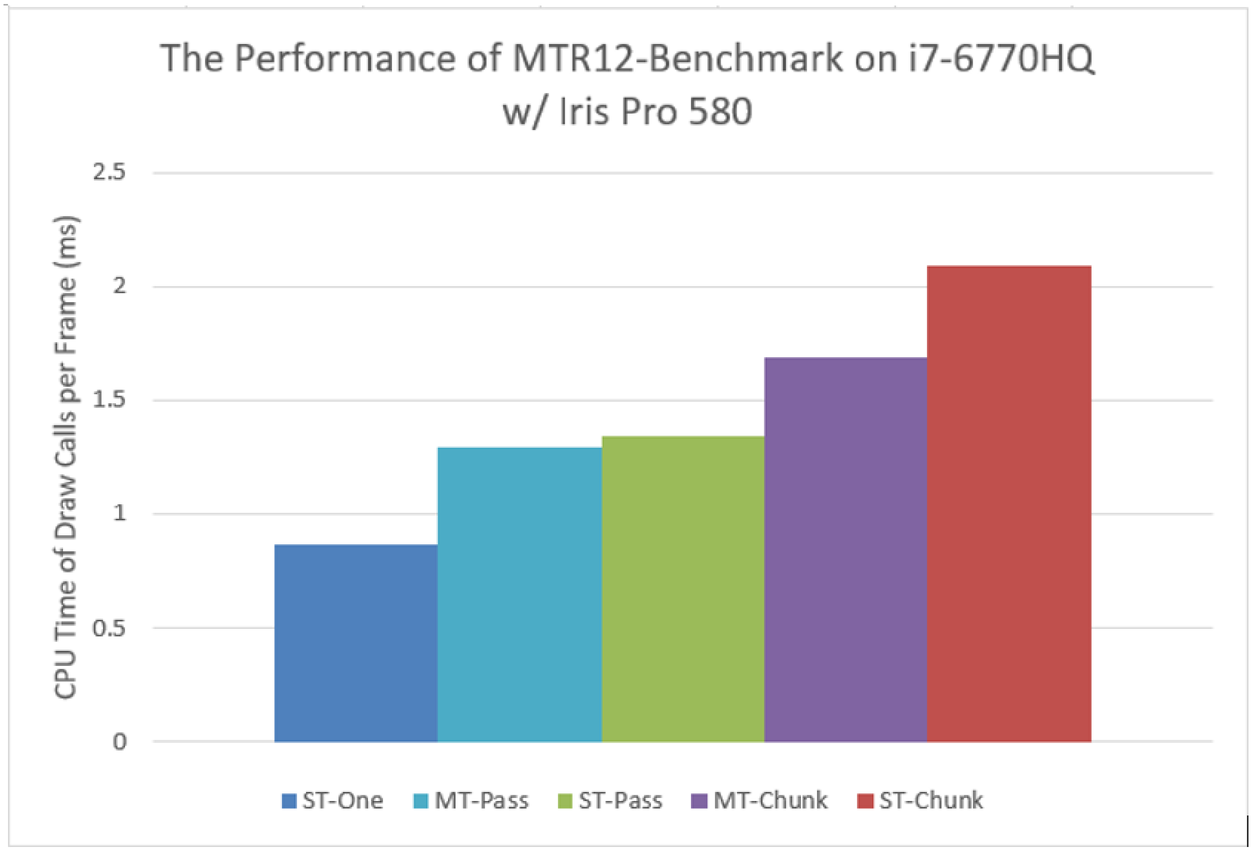
Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products. Performance results are based on testing by Intel engineer as of Jan 15th, 2019 and may not reflect all publicly available security updates. Configuration: Intel® Core™ i7-6950X CPU @ 3.00GHz, 32GB RAM, Windows® 10 Enterprise 64-bit (10.0, Build 17134).

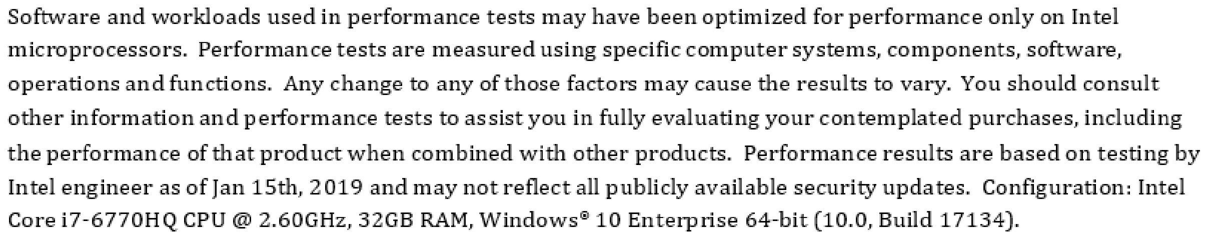
(7.a) Test result on Platform A



Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products. Performance results are based on testing by Intel engineer as of Jan 15th, 2019 and may not reflect all publicly available security updates. Configuration: Intel® Core™ i7-6950X CPU @ 3.00GHz, 32GB RAM, Windows® 10 Enterprise 64-bit (10.0, Build 17134).

(7.b) Test result on Platform B




Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products. Performance results are based on testing by Intel engineer as of Jan 15th, 2019 and may not reflect all publicly available security updates. Configuration: Intel Core i7-6770HQ CPU @ 2.60GHz, 32GB RAM, Windows® 10 Enterprise 64-bit (10.0, Build 17134).

(7.c) Test result on Platform C

Figure 7. The performance of MTR12-Benchmark on different CPU and GPU configurations

Based on the test results above, the following performance characteristics of DirectX 12 multithreading can be concluded:

1. Submitting D3D12 command-list involves much more overhead than calling other D3D12 APIs. The more command-lists submitted, the more overhead generated.

2. DirectX 12 multithreading doesn’t necessarily improve performance, especially when purely calling D3D APIs. As opposed to the single-threaded rendering submitting command-lists one or few times per frame, multithreaded rendering usually requires submitting more times. These submissions may have to be done sequentially and involve so much overhead that the performance gains of building command-lists in parallel are offset.

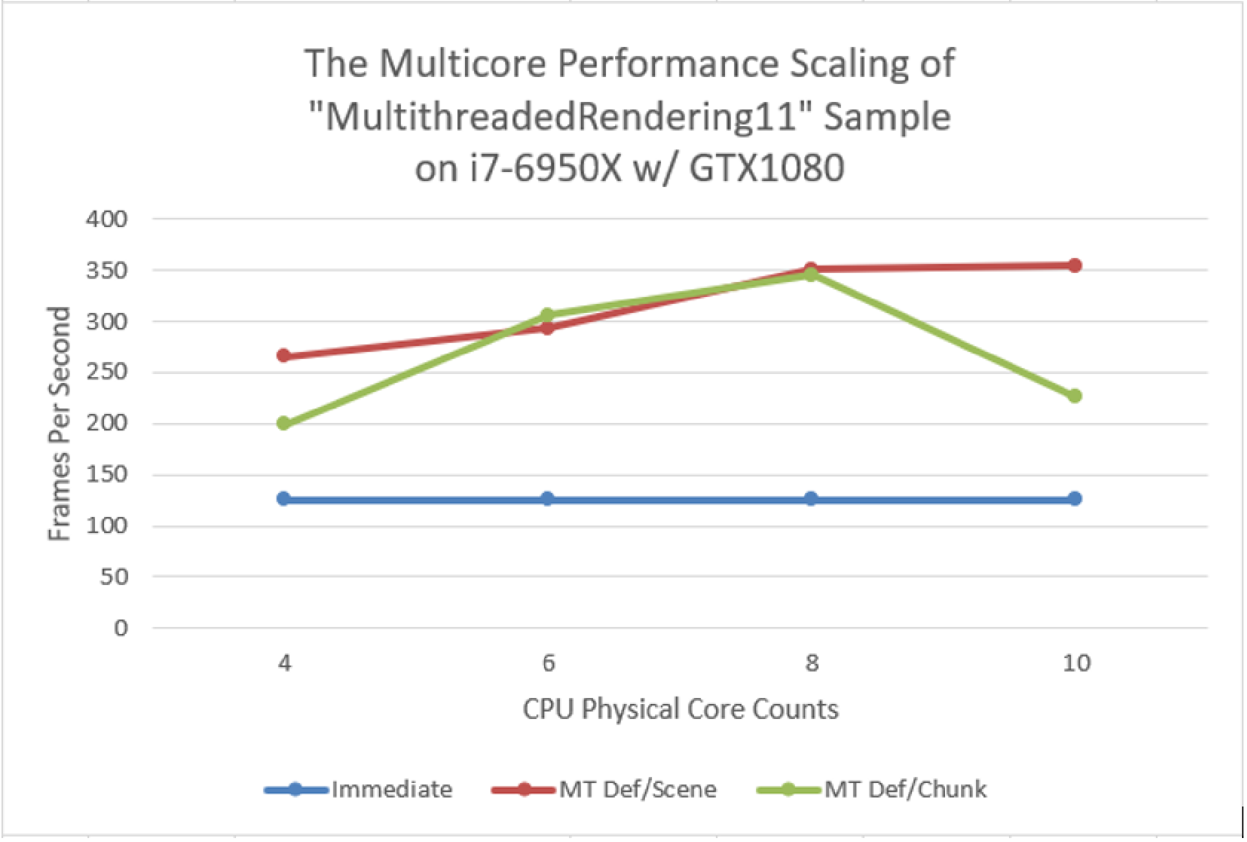
3. DirectX 12 benefits multithreaded rendering more than DirectX 11. Although DirectX 12 multithreading does involve some overhead, it has much less overhead than DirectX 11 multithreading, which will relieve the bottleneck and improve multithreading performance.

3. Key Factors Affecting Multithreaded Rendering Performance

The “MultithreadedRendering11” (MTR11) sample provides a good experimental basis to disclose the key factors affecting multithreaded rendering performance. Besides of DirectX multithreading API overhead discussed in section 2, thread synchronization overhead and parallelizable application-layer load also play important roles in the performance of multithreaded rendering.

3.1 Thread Synchronization Overhead

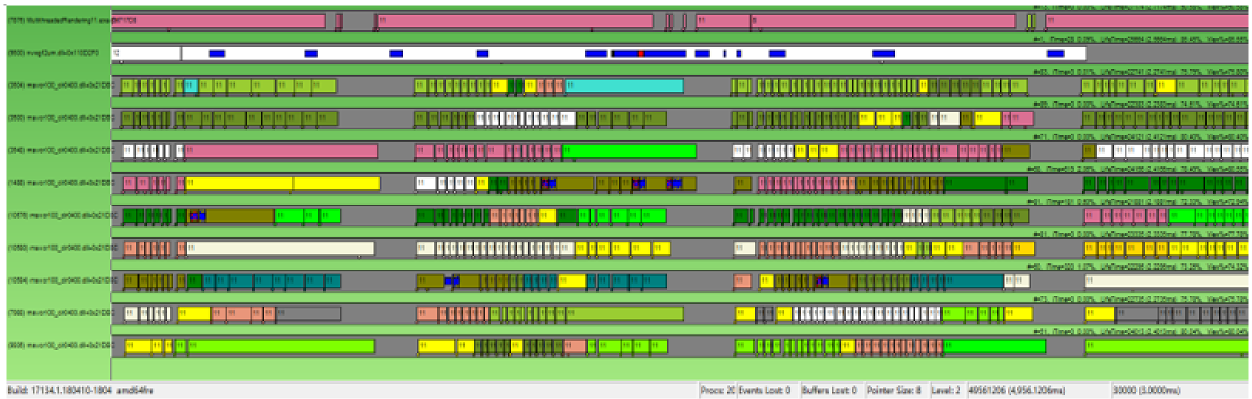
The result of testing the original “MultithreadedRendering11” sample (Figure 4) on Platform A, shows that both multithread modes, “MT Def/Scene” and “MT Def/Chunk”, have much better performance than the single-thread mode “Immediate” (Figure 8). However, when CPU core count increases to 10, the frame rate of “MT Def/Chunk” mode drops significantly. This anomaly should be investigated.



Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products. Performance results are based on testing by Intel engineer as of Jan 15th, 2019 and may not reflect all publicly available security updates. Configuration: Intel® Core™ i7-6950X CPU @ 3.00GHz, 32GB RAM, Windows® 10 Enterprise 64-bit (10.0, Build 17134).

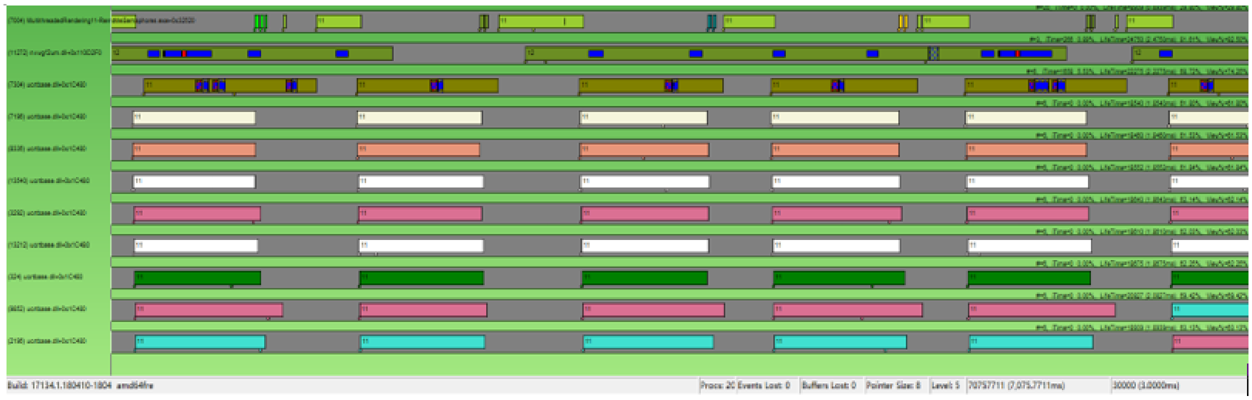
Figure 8. The performance comparison between multithreaded rendering and single-threaded rendering

Analyzing the thread activities of “MT Def/Chunk” mode on 10-core CPU by GPUView* [9], finds that worker thread running is interrupted frequently (Figure 9.a), which means there are a lot of thread state changes during rendering. The sample’s source code indicates that the main thread continuously produces more than 4 thousand work items to render each scene, meanwhile worker threads consume them in parallel by synchronize with main thread on each work item. It means there are considerable overhead of thread synchronization in rendering. When the CPU cores increase, the worker threads increase correspondingly, thus the probability and time of each worker thread waiting for a work item increase too. It results in the thread synchronization overhead expands further so that the performance dramatically dropped on 10-core CPU.


Build: 17134.1.180410-1804 amd64fre
Procs: 20 Events Lost: 0 Buffers Lost: 0 Pointer Size: 8 Level: 2 40561206 (4,056.1206ms) 30000 (3.0000ms)

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products. Performance results are based on testing by Intel engineer as of Jan 15th, 2019 and may not reflect all publicly available security updates. Configuration: Intel® Core™ i7-6950X CPU @ 3.00GHz, 32GB RAM, Windows® 10 Enterprise 64-bit (10.0, Build 17134).

(9.a) original implementation

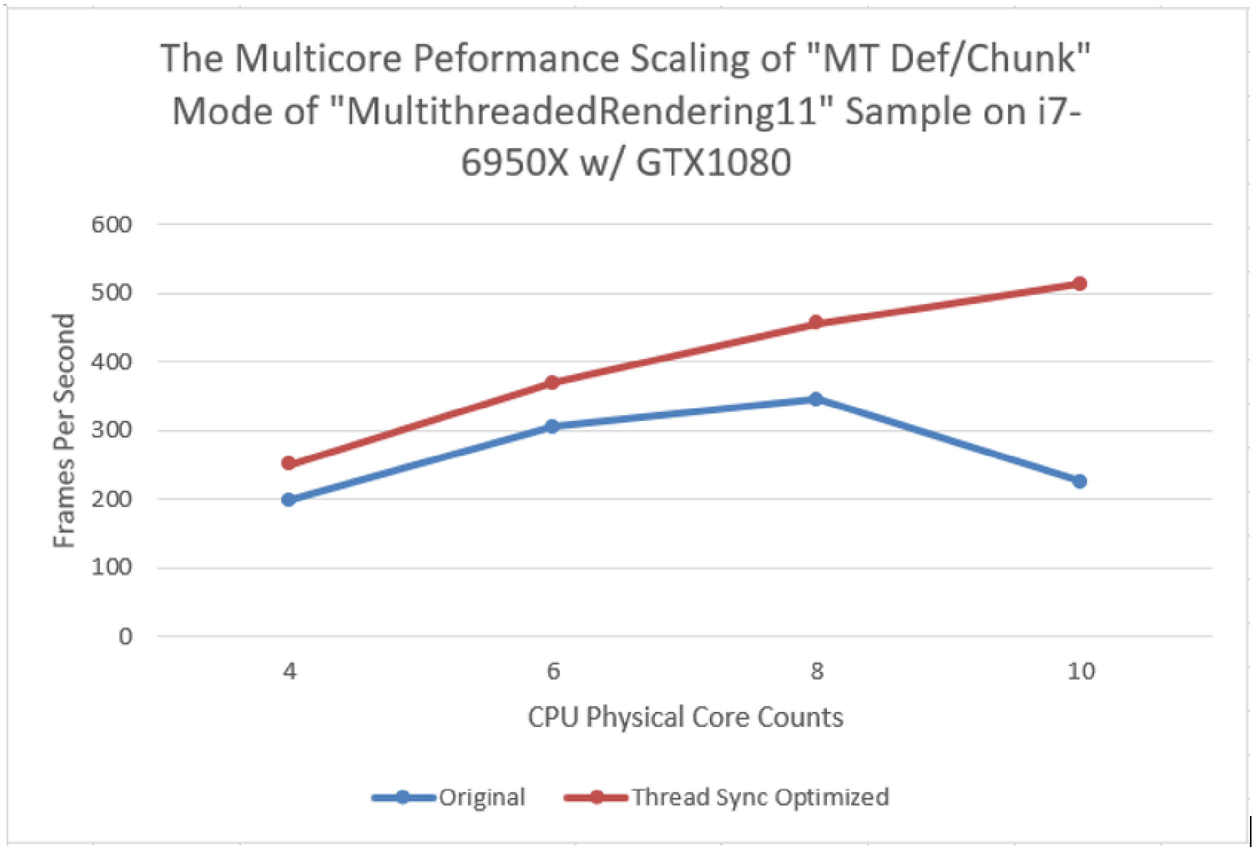

Build: 17134.1.180410-1804 amd64fre
Procs: 20 Events Lost: 0 Buffers Lost: 0 Pointer Size: 8 Level: 5 70757711 (7,075.7711ms) 30000 (3.0000ms)

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products. Performance results are based on testing by Intel engineer as of Jan 15th, 2019 and may not reflect all publicly available security updates. Configuration: Intel® Core™ i7-6950X CPU @ 3.00GHz, 32GB RAM, Windows® 10 Enterprise 64-bit (10.0, Build 17134).

(9.b) thread synchronization optimized implementation

Figure 9. Thread activities of the “MT Def/Chunk” mode of “MultithreadedRendering11” sample on 10 cores

To solve the problem, the source codes of the sample was modified to make the main thread produce all of the work items before notifying work threads to consume them in parallel, which greatly reduces the frequency and overhead of synchronization between the main thread and the worker thread. Figure 9.b shows the optimized sample improved the continuity of worker thread running, significantly reducing the rendering time of each scene. Figure 10 shows the performance of “MT Def/Chunk” mode is significantly improved after thread synchronization optimization. It not only solves the problem of frame rate dropping on 10 cores, but also improves the performance of this mode on all CPU core counts.



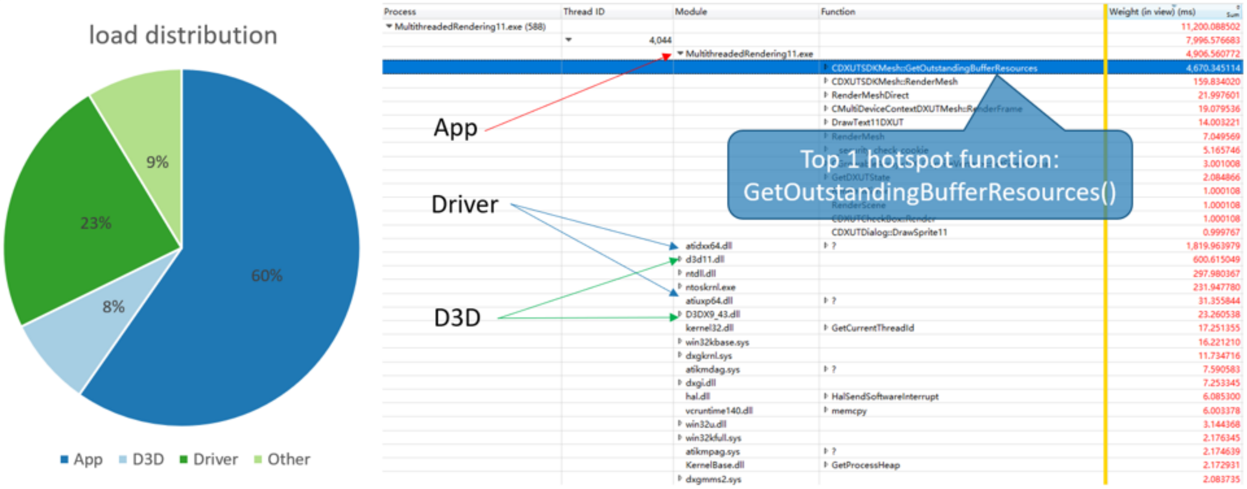
Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products. Performance results are based on testing by Intel engineer as of Jan 15th, 2019 and may not reflect all publicly available security updates. Configuration: Intel® Core™ i7-6950X CPU @ 3.00GHz, 32GB RAM, Windows® 10 Enterprise 64-bit (10.0, Build 17134).

Figure 10. The multicore performance scaling of the “MT Def/Chunk” mode of “MultithreadedRendering11” sample before and after thread synchronization optimization

This experiment demonstrates that thread synchronization overhead can significantly affect multithreaded performance. Inappropriate thread synchronization strategy, for example, using too small granularity of thread job, may impair the advantage of multithreaded rendering performance.

3.2 Parallelizable Application-layer Load

The “MultithreadedRendering11” sample shows that both multithread modes have much better performance than the single-thread mode (Figure 8). To understand the characteristics of the workload that may benefits from multithreaded rendering, the load distributed in the layers of application logic, D3D runtime and graphics driver was analyzed in the “Immediate” mode of the sample (Figure 4). According to the weights of CPU usage of the modules profiled by Windows* Performance Analyzer (WPA) [8] (Figure 11), the relative load of different layers can be calculated. And it shows that the application logic has the biggest load (60%), the driver load has 23%, and Direct3D runtime consumes 8% of the load.

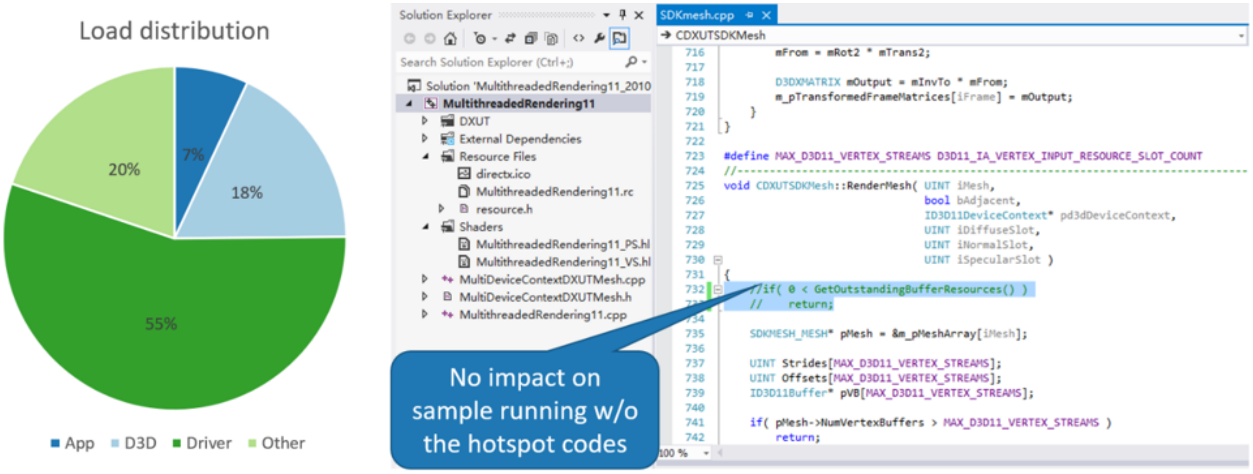


Process	Thread ID	Module	Function	Weight (in view) (ms) Sum
MultithreadedRendering11.exe (588)				11,200.088502
	4,044			7,996.576683
		MultithreadedRendering11.exe		4,906.560772
			CDXUTSDKMesh::GetOutstandingBufferResources	4,670.345114
			CDXUTSDKMesh::RenderMesh	159.834020
			RenderMeshDirect	21.997601
			CMultiDeviceContextDXUTMesh::RenderFrame	19.079536
			DrawText11DXUT	14.003221
			RenderMesh	7.049569
			__security_check_cookie	5.165746
			[illegible]	3.001008
			GetDXUTState	2.084866
			[illegible]	1.000108
			RenderScene	1.000108
			CDXUTCheckBox::Render	1.000108
			CDXUTDialog::DrawSprite11	0.999767
		atidxx64.dll	?	1,819.963979
		d3d11.dll		600.615049
		ntdll.dll		297.980367
		ntoskrnl.exe		231.947780
		atiuxp64.dll	?	31.355844
		D3DX9_43.dll		23.260538
		kernel32.dll	GetCurrentThreadId	17.251355
		win32kbase.sys		16.221210
		dxgkrnl.sys		11.734716
		atikmdag.sys	?	7.590583
		dxgi.dll		7.253345
		hal.dll	HalSendSoftwareInterrupt	6.085300
		vcruntime140.dll	memcpy	6.003378
		win32u.dll		3.144368
		win32kfull.sys		2.176345
		atikmpag.sys	?	2.174639
		KernelBase.dll	GetProcessHeap	2.172931
		dxgmms2.sys		2.083735

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products. Performance results are based on testing by Intel engineer as of Jan 15th, 2019 and may not reflect all publicly available security updates. Configuration: Intel® Core™ i7-6950X CPU @ 3.00GHz, 32GB RAM, Windows® 10 Enterprise 64-bit (10.0, Build 17134).

Figure 11. Load distribution of the “Immediate” mode of the “MultithreadedRendering11” sample

WPA also reveals the top 1 hotspot function (Figure 11). The source code indicates the function is used to validate the vertex buffers and index buffers of all meshes in the scene. Oddly, the function is called repeatedly as each mesh is drawn, which is unnecessary. Actually, this hotspot function has been optimized in the sample of the same name in the latest DirectX SDK sample package [5]. Commenting out this function call (Figure 12) does not affect the normal execution of the program, while the primary load transfers from application logic (7%) to the graphics driver (55%).

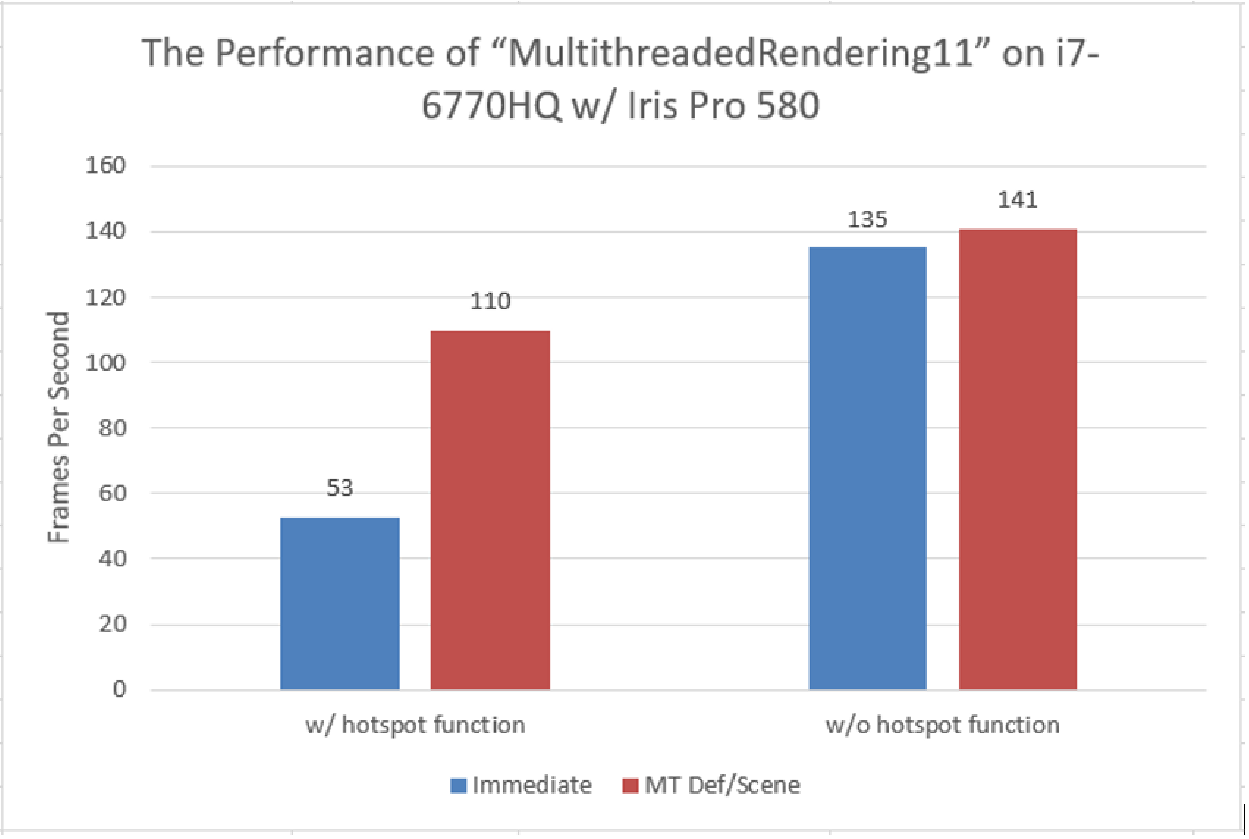


```
            mFrom = mRot2 * mTrans2;

            D3DXMATRIX mOutput = mInvTo * mFrom;
            m_pTransformedFrameMatrices[iFrame] = mOutput;
        }
}

#define MAX_D3D11_VERTEX_STREAMS D3D11_IA_VERTEX_INPUT_RESOURCE_SLOT_COUNT
//--------------------------------------------------------------------------------------
void CDXUTSDKMesh::RenderMesh( UINT iMesh,
                               bool bAdjacent,
                               ID3D11DeviceContext* pd3dDeviceContext,
                               UINT iDiffuseSlot,
                               UINT iNormalSlot,
                               UINT iSpecularSlot )
{
    //if( 0 < GetOutstandingBufferResources() )
    //    return;

    SDKMESH_MESH* pMesh = &m_pMeshArray[iMesh];

    UINT Strides[MAX_D3D11_VERTEX_STREAMS];
    UINT Offsets[MAX_D3D11_VERTEX_STREAMS];
    ID3D11Buffer* pVB[MAX_D3D11_VERTEX_STREAMS];

    if( pMesh->NumVertexBuffers > MAX_D3D11_VERTEX_STREAMS )
        return;
```

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products. Performance results are based on testing by Intel engineer as of Jan 15th, 2019 and may not reflect all publicly available security updates. Configuration: Intel® Core™ i7-6950X CPU @ 3.00GHz, 32GB RAM, Windows® 10 Enterprise 64-bit (10.0, Build 17134).

Figure 12. Load distribution of the “Immediate” mode of the “MultithreadedRendering11” sample after removing hotspot codes of application-layer

Retesting the sample that comments out the hotspot function shows the speedup of multithread mode (“MT Def/Scene”) against single-thread mode (“Immediate”) decreases dramatically (Figure 13). It means the significant advantage of multithreaded rendering comes primarily from calling the hotspot function of application-layer, other than D3D APIs, in multiple threads.



Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products. Performance results are based on testing by Intel engineer as of Jan 15th, 2019 and may not reflect all publicly available security updates. Configuration: Intel Core i7-6770HQ CPU @ 2.60GHz, 32GB RAM, Windows® 10 Enterprise 64-bit (10.0, Build 17134).

Figure 13. Application-layer load significantly affects the advantage of multithreaded rendering over single-threaded rendering

This experiment demonstrates that the amount of parallelizable application-layer load can significantly affect the performance advantage of multithreaded rendering. Without enough parallelizable application-layer load, multi-threaded rendering may not show much of the performance and multi-core scalability advantage over single-threaded rendering.

4. Tips and Tricks for Multithreaded Rendering

Based on the conclusions from above experiments, a couple of tips and tricks may be considered to take advantage of multithreaded rendering performance.

Before multithreading the renderer, it's recommended to use WPA or Intel® VTune™ [6] firstly to evaluate the proportion of the application-layer load of the renderer to the load of D3D runtime and graphics driver. If the application-layer load is parallelizable and has a large enough proportion, multithreading renderer is helpful to improve the performance. Otherwise, it may have little benefit or even hit the performance.

When designing or selecting the methods to multithread the renderer, developers should keep in mind that the thread synchronization and command-list submission may cause considerable overhead and thus hit performance. Therefore, it's required to carefully control the amount of thread synchronization and command-list submission per frame in the implementation of multithreaded rendering.

There are two basic methods of multithreaded rendering: the pass(scene)-based and the chunk-based, as illustrated in "MultithreadedRendering11" sample. The pass-based method divides the rendering task in the granularity of pass(scene). The number of thread jobs generated each frame is equal to the number of passes(scenes) per frame. Therefore, it has the limited multithreading overhead due to the small amount of thread synchronization and command-list submission per frame. But it may not have balanced thread load and cannot scale performance with CPU core counts. Generally, the pass-based method has moderate performance advantage.

The chunk-based method divides the rendering task of pass (scene) into smaller-grained jobs. The number of thread jobs generated each frame can be many times the number of CPU cores. Theoretically, this method can achieve the best performance due to good thread load balancing and multi-core performance scalability. However, actually it sometimes may not even as fast as the pass-based method, especially when the parallelizable application-layer load is less. It's because that this method may produce more thread synchronization and command-list submission per frame. And the more CPU cores, the more thread synchronization and command-list submission per frame. It may result in significant overhead which offsets the multithreading performance gain.

Both methods have certain limitations. The “ideal” method for multithreaded rendering should be able to make the most of multicore to improve performance in all cases, no matter how large the application load, and whether or not the graphics hardware acceleration for DirectX multithreading is supported.

A promising solution is to use the concept of "intermediate rendering command" to decouple the rendering logic and graphics (D3D) API calls. The "intermediate rendering command" is a wrapper of graphics APIs, similar to the “Rendering Hardware Interface”(RHI) command in Unreal Engine* 4 [10], used to cache the arguments for graphics APIs. The renderer front-end may execute the rendering logic to generate "intermediate rendering commands". And the renderer back-end may translate the "intermediate rendering commands" to graphics APIs and call them.

This solution allows rendering logic and graphics API calls to respectively use the most appropriate task decomposition strategy for each. Rendering logic may be divided into small-grained jobs to improve multicore utilization and load balancing. And graphics API calls may be divided into one or several jobs (command-lists), depending on the number of draw calls per frame, to limit the number of command-list submissions. To minimize the thread synchronization overhead, the efficient thread library with work-stealing task scheduler, for instance, Intel® Threading Building Blocks (Intel® TBB) [7], is recommended.

With minimizing the overhead involved in DirectX multithreading, when the application-layer load of the renderer is small, this solution may avoid the performance of multithreaded rendering worse than the performance of the single-threaded rendering across a wide range of graphics cards. When the parallelizable application-layer load is large, this solution may make the most of multicore to achieve much better performance than the single-threaded rendering.

5. Summary

This article disclosed the performance characteristics of DirectX multithreading with a series of experiments based on the modified and extended workload of official DirectX samples.

Through these experiments, it’s found that for either D3D11 or D3D12, the command-list submission has much more overhead than other draw calls, so that many command-list submissions may negate the performance gains of parallelizing draw calls. For D3D11, the hardware acceleration significantly benefits to multithreaded rendering performance. It’s also shown that the thread synchronization overhead and parallelizable application-layer load have the significant impact on the performance advantage of multithreaded rendering versus single-threaded rendering.

Based on these qualitative analyses, it's recommended to evaluate the amount of parallelizable application-layer load before multithreading the renderer, and carefully control the amount of thread synchronization and command-list submission in multithreaded implementation. Also, a multithreaded rendering method based on "intermediate rendering command" is proposed to maintain the multithreading performance advantage in various situations and make the most of multicore for the best performance.

CPU keeps on evolving to more cores. Based on the in-depth understanding of DirectX multithreading performance, multithreading the renderer is promising to relieve the CPU bottleneck on multiple cores, or to extract more performance headroom to render scenes with richer visual content.

References

[1] Introduction to Multithreading in Direct3D 11

[2] How To: Check for Driver Support

[3] DirectX SDK (June 2010)

[4] DirectX 12 Graphics samples

[5] DirectX SDK samples for Windows 8.x SDK or Windows 10 SDK

[6] Intel® VTune™ Amplifier

[7] Intel® Threading Building Blocks (Intel® TBB)

[8] Windows Performance Analyzer

[9] GPUView

[10] Render Hardware Interface (RHI)

About the Author

Sheng Guo is a senior application engineer on game enabling in Intel Corporation. He has been helping top game ISVs improve games by Intel platforms and technologies for over 10 years. His expertise includes performance profiling, optimization, cutting-edge feature programming for games. He has contributed quite a few technical articles and white papers to game communities, and published a couple of papers in academic conferences and journals.