Multi-Adapter Support in DirectX* 12

Introduction

This sample shows how to implement an explicit multi-adapter application using DirectX 12. Intel's integrated GPU (iGPU) and a discrete NVIDIA GPU (dGPU) are used to share the workload of ray-tracing a scene. The parallel use of both GPUs allows for an increase in performance and for more complex workloads.

This sample uses multiple adapters to render a simple ray-traced scene using a pixel shader. Both adapters render a portion of the scene in parallel.

Explicit Multi-Adapter Overview

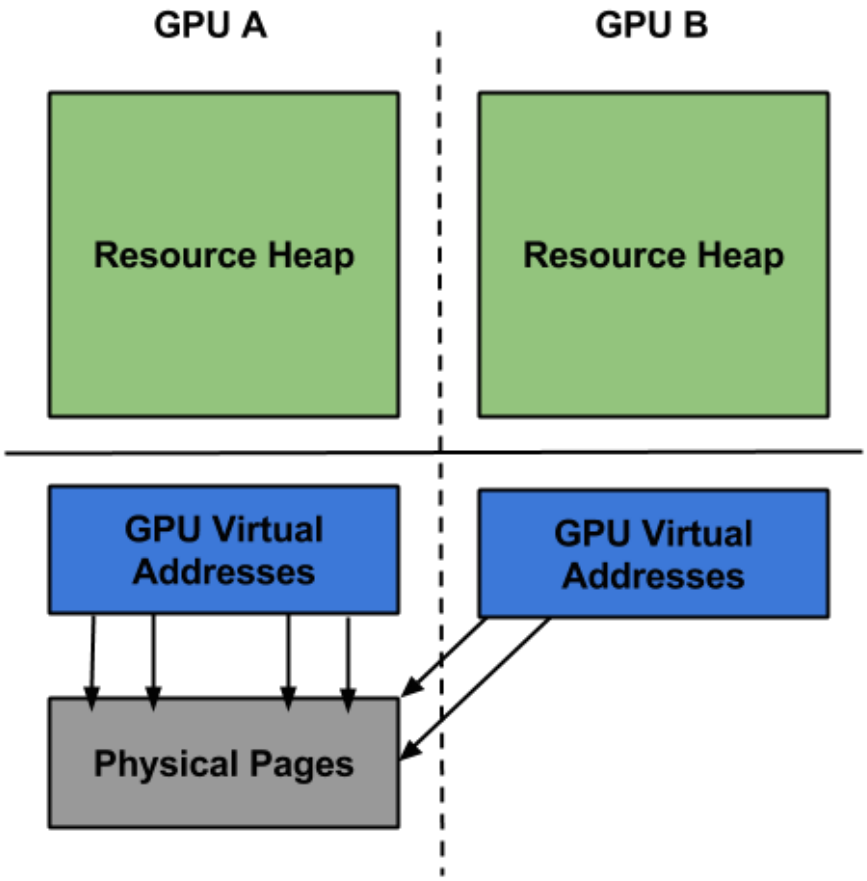
Support for explicit multi-adapter is a new feature in DirectX 12. This feature allows for the parallel use of multiple GPUs regardless of manufacturer and type (for example, integrated or discrete). The ability to separate work across multiple GPUs is provided by the presence of independent resource management and parallel queues for each GPU at the API level.

DirectX 12 introduces two main API features that help enable multi-adapter applications:

- **Cross-adapter memory** that is visible to both adapters.

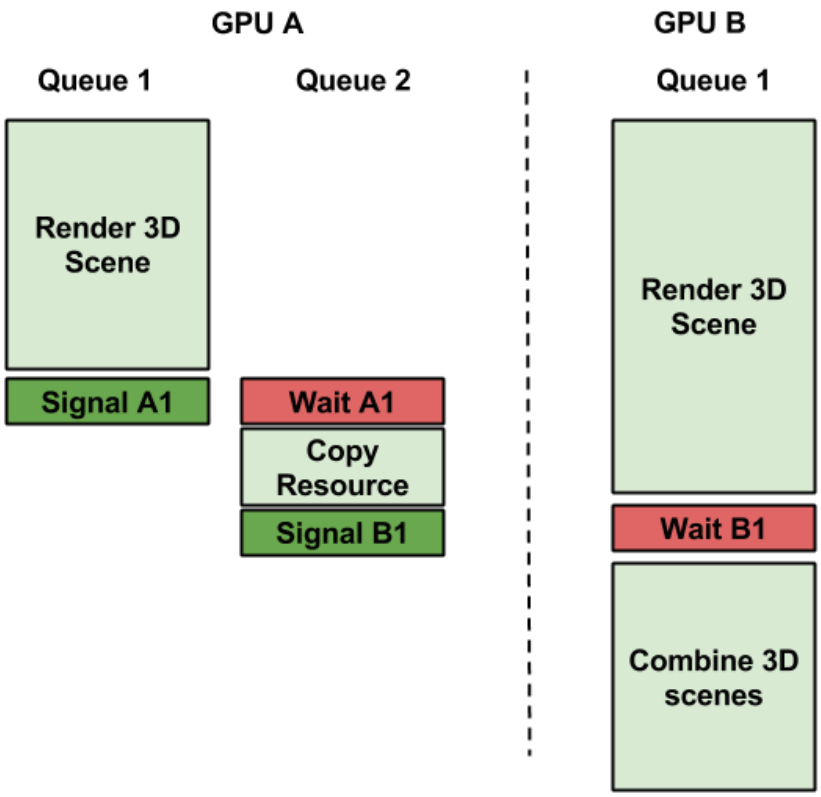
  DirectX 12 introduces cross-adapter-specific resource and heap flags:
  - `D3D12_RESOURCE_FLAG_ALLOW_CROSS_ADAPTER`
  - `D3D12_HEAP_FLAG_SHARED_CROSS_ADAPTER`

  The cross-adapter resources exist in memory on the primary adapter and can be referenced from another adapter with minimal cost.



- **Parallel queues and cross-adapter synchronization** that allows for parallel execution of commands. A special flag is used when creating a synchronization fence: `D3D12_FENCE_FLAG_SHARED_CROSS_ADAPTER`.

  A cross-adapter fence allows a queue on one adapter to be signaled by a queue on the other.



The above diagram shows the use of three queues to facilitate copying into cross-adapter resources. This is the technique used in this sample and showcases the following steps:

1. Queue 1 on GPU A and Queue 1 on GPU B render portions of a 3D scene in parallel.
2. When rendering is complete, Queue 1 signals, allowing Queue 2 to begin copying.
3. Queue 2 copies the rendered scene into a cross-adapter resource and signals.
4. Queue 1 on GPU B waits for Queue 2 on GPU A to signal and combines both rendered scenes into the final output.

Cross-Adapter Implementation Steps

Incorporating a secondary adapter into a DirectX 12 application involves the following steps:

1. Create cross-adapter resources on the primary GPU as well as a handle to these resources on the secondary GPU.

```
// Describe cross-adapter shared resources on primaryDevice adapt
er

D3D12_RESOURCE_DESC crossAdapterDesc = mRenderTargets[0]->GetDesc
();
crossAdapterDesc.Flags = D3D12_RESOURCE_FLAG_ALLOW_CROSS_ADAPTER;
crossAdapterDesc.Layout = D3D12_TEXTURE_LAYOUT_ROW_MAJOR;

// Create a shader resource and shared handle
for (int i = 0; i < NumRenderTargets; i++)
{
    mPrimaryDevice->CreateCommittedResource(
        &CD3DX12_HEAP_PROPERTIES(D3D12_HEAP_TYPE_DEFAULT),
        D3D12_HEAP_FLAG_SHARED | D3D12_HEAP_FLAG_SHARED_CROSS_ADA
PTER,
        &crossAdapterDesc,
        D3D12_RESOURCE_STATE_PIXEL_SHADER_RESOURCE,
        nullptr,
        IID_PPV_ARGS(&shaderResources[i]));

    HANDLE heapHandle = nullptr;
    mPrimaryDevice->CreateSharedHandle(
        mShaderResources[i].Get(),
        nullptr,
        GENERIC_ALL,
        nullptr,
        &heapHandle);

    // Open shared handle on secondaryDevice device
    mSecondaryDevice->OpenSharedHandle(heapHandle, IID_PPV_ARGS(&
shaderResourceViews[i]));

    CloseHandle(heapHandle);
}

// Create a shader resource view (SRV) for each of the cross adap
ter resources
CD3DX12_CPU_DESCRIPTOR_HANDLE secondarySRVHandle(mSecondaryCbvSrv
UavHeap->GetCPUDescriptorHandleForHeapStart());
for (int i = 0; i < NumRenderTargets; i++)
{
    mSecondaryDevice->CreateShaderResourceView(shaderResourceView
s[i].Get(), nullptr, secondarySRVHandle);
    secondarySRVHandle.Offset(mSecondaryCbvSrvUavDescriptorSize);
}
```

2. Create synchronization fences for the resources that are shared between both adapters.

```
// Create fence for cross adapter resources
mPrimaryDevice->CreateFence(mCurrentFenceValue,
    D3D12_FENCE_FLAG_SHARED | D3D12_FENCE_FLAG_SHARED_CROSS_ADAPT
ER,
    IID_PPV_ARGS(&primaryFence));

// Create a shared handle to the cross adapter fence
HANDLE fenceHandle = nullptr;
mPrimaryDevice->CreateSharedHandle(
    primaryFence.Get(),
    nullptr,
    GENERIC_ALL,
    nullptr,
    &fenceHandle));

// Open shared handle to fence on secondaryDevice GPU
mSecondaryDevice->OpenSharedHandle(fenceHandle, IID_PPV_ARGS(&sec
ondaryFence));
```

3. Render on primary GPU into an offscreen render target, and signal queue on completion.

```
// Render scene on primary device
mPrimaryCommandQueue->ExecuteCommandLists(1, primaryCommandList);
;

// Signal primary device command queue to indicate render is comp
lete
mPrimaryCommandQueue->Signal(mPrimaryFence.Get(), currentFenceVal
ue));
fenceValues[currentFrameIndex] = currentFenceValue;
mCurrentFenceValue++;
```

4. Copy resources from offscreen render target into cross-adapter resources, and signal queue on completion.

```
// Wait for primary device to finish rendering the frame
mCopyCommandQueue->Wait(mPrimaryFence.Get(), fenceValues[currentF
rameIndex]);

// Copy from off-screen render target to cross-adapter resource
mCopyCommandQueue->ExecuteCommandLists(1, crossAdapterResources->
mCopyCommandLists.Get());

// Signal secondary device to indicate copy is complete
mCopyCommandQueue->Signal(mPrimaryCrossAdapterFence.Get(), mCurre
ntCrossAdapterFenceValue));
mCrossAdapterFenceValues[mCurrentFrameIndex] = mCurrentCrossAdapt
erFenceValue;
mCurrentCrossAdapterFenceValue++;
```

5. Render on secondary GPU, using handle to cross-adapter resource to access resources as a texture.

```
// Wait for primary device to finish copying
mSecondaryCommandQueue->Wait(mSecondaryCrossAdapterFence.Get(), m
CrossAdapterFenceValues[mCurrentFrameIndex]));

// Render cross adapter resources and segmented texture overlay o
n secondary device
mSecondaryCommandQueue->ExecuteCommandLists(1, secondaryCommandLi
st);
```

6. Secondary GPU displays frame to screen.

```
mSwapChain->Present(0, 0);
MoveToNextFrame();
```

Note that the code provided above has been modified for simplification with all error checking removed. It is not expected to compile.

Performance and Results

Using multiple adapters to render a scene in parallel yields a significant increase in performance compared to relying on a single adapter to perform the entire rendering workload.

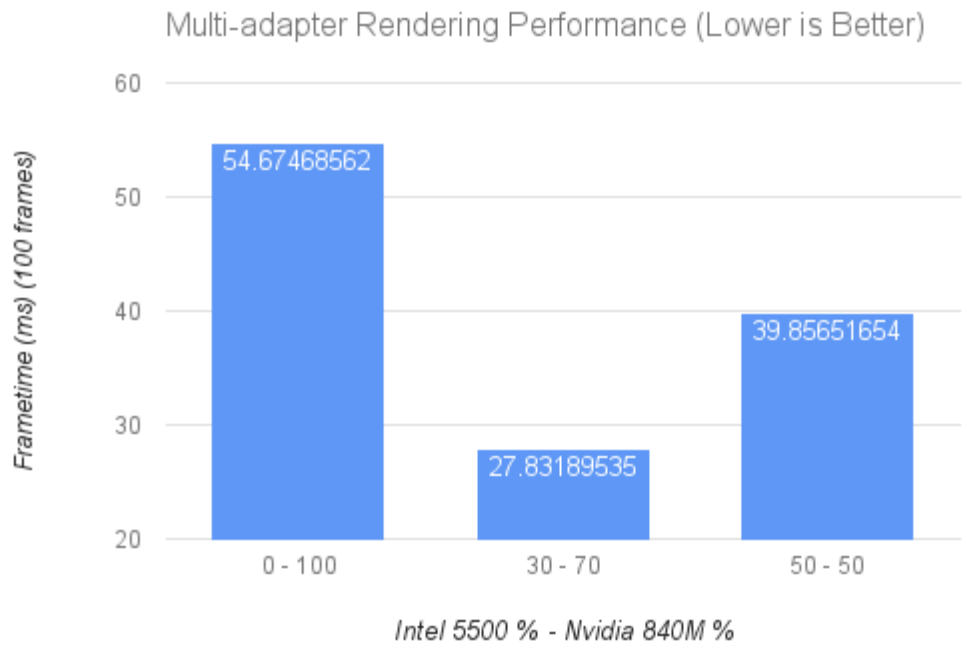
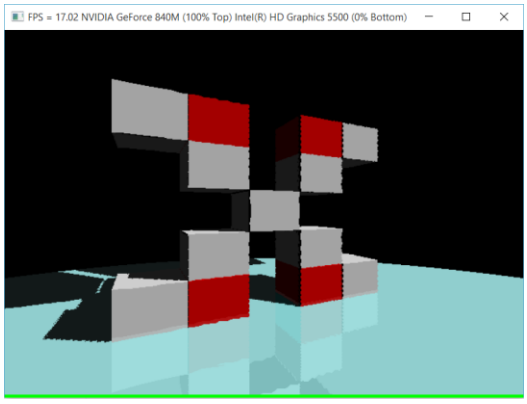
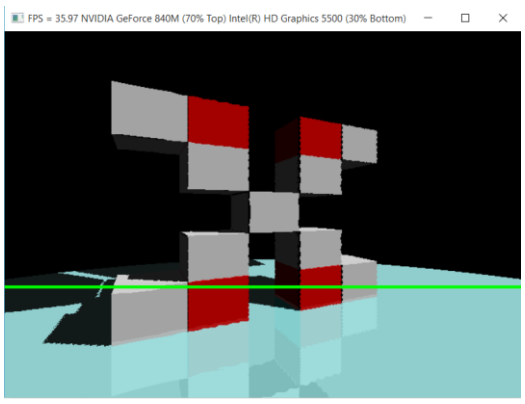


Figure 1. Frame time of 100 frames in milliseconds versus work split between integrated and discrete cards.

In the sample ray-traced scene, a decrease of approximately 26 milliseconds was observed when both an NVIDIA GeForce* 840M and Intel® HD Graphics 5500 were used to share the rendering load.

By parallelizing the workload, it is possible to reduce the frame time required by nearly 50 percent compared to using a single adapter.





Note that the workload shown in this sample is easily parallelizable and may not immediately translate to real-world gaming applications.

Appendix: Sample Architecture Overview

This sample is architected as follows:

- `WinMain.cpp`
  - Entry point to the application
  - Creates `DXDevice` objects and instantiates renderer
- `DXDevice.cpp`
  - Encapsulates `ID3D12Device` object alongside related objects
  - Contains command queue, allocator, render targets, fences, and descriptor heaps
- `DXRenderer.cpp`
  - Base renderer class
  - Implements shared functionality (for example, creating vertex buffer or updating texture)
- `DXMultiAdapterRenderer.cpp`
  - Perform all core, implementation-specific rendering functionality (that is, set up pipeline, load assets, and populate command lists)
- `DXCrossAdapterResources.cpp`
  - Abstracts creation and updating of multi-adapter resources
  - Handles copying of resources and fencing between both GPUs

`DXMultiAdapterRenderer.cpp` consists of the following functions:

```
public:
    DXMultiAdapterRenderer(std::vector<DXDevice*> devices, MS::ComPtr
<IDXGIFactory4> dxgiFactory, UINT width, UINT height, HWND hwnd);
    virtual void OnUpdate() override;
    float GetSharePercentage();
    void IncrementSharePercentage();
    void DecrementSharePercentage();
protected:
    virtual void CreateRootSignatures() override;
    virtual void LoadPipeline() override;
    virtual void LoadAssets() override;
    virtual void CreateCommandLists() override;
    virtual void PopulateCommandLists() override;
    virtual void ExecuteCommandLists() override;
    virtual void MoveToNextFrame() override;
```

This class implements all the core rendering functionality. The `LoadPipeline()` and `LoadAssets()` functions are responsible for creating all necessary root signatures, compiling shaders, and creating pipeline state objects as well as specifying and creating all textures, constant buffers, and vertex buffers and their associated views. All necessary command lists are created at this time as well.

For each frame, `PopulateCommandLists()` and `ExecuteCommandLists()` are called.

In order to separate the traditional DirectX 12 rendering functionality from that which is necessary for using multiple-adapters, all of the necessary cross-adapter functionality is encapsulated in the `DXCrossAdapterResources` class, which contains the following functions:

```
public:
    DXCrossAdapterResources(DXDevice* primaryDevice, DXDevice* second
aryDevice);
    void CreateResources();
    void CreateCommandList();
    void PopulateCommandList(int currentFrameIndex);
    void SetupFences();
```

The `CreateResources()`, `CreateCommandList()`, and `SetupFences()` functions are all called upon initialization to create the cross-adapter resources and initialize the synchronization objects.

Every frame, the `PopulateCommandList()` function is called to populate the copy command list.

The `DXCrossAdapterResources` class contains a separate command allocator, command queue, and command list that are used for copying resources from a render target on the primary adapter into the cross-adapter resources.