Advanced Micro Devices Inc.

LiquidVR SDK SimpleMGPU Sample

Sample Technical Guide

11-23-2015

Disclaimer

The information contained herein is for informational purposes only, and is subject to change without notice. While every precaution has been taken in the preparation of this document, it may contain technical inaccuracies, omissions and typographical errors, and AMD is under no obligation to update or otherwise correct this information.

Advanced Micro Devices, Inc. makes no representations or warranties with respect to the accuracy or completeness of the contents of this document, and assumes no liability of any kind, including the implied warranties of noninfringement, merchantability or fitness for particular purposes, with respect to the operation or use of AMD hardware, software or other products described herein. No license, including implied or arising by estoppel, to any intellectual property rights is granted by this document. Terms and limitations applicable to the purchase or use of AMD's products are as set forth in a signed agreement between the parties or in AMD's Standard Terms and Conditions of Sale.

AMD, the AMD Arrow logo, ATI Radeon™, CrossFireX™, LiquidVR™ and combinations thereof are trademarks of Advanced Micro Devices, Inc. Other product names used in this publication are for identification purposes only and may be trademarks of their respective companies.

Windows™, Visual Studio and DirectX are trademark of Microsoft Corp.

Copyright Notice

Overview

This sample demonstrates the performance gain from using two CrossFire™-connected GPUs to render 3D scenes for the left and the right eye in parallel on separate GPUs.

The sample continuously renders a stereoscopic image of a rotating cube with a parallax to create the perception of depth using different methods of rendering. To simulate higher GPU loads which would reflect the load in typical VR scenarios without having to create a complex 3D scene, the cube is rendered multiple times in a loop.

The sample provides a basic user interface to toggle mutli-GPU rendering on and off, as well as to switch between two different rendering models supported by LiquidVR™, which, in different circumstances may provide performance gains.

Pre-requisites

To run the sample, the following hardware and software requirements need to be fulfilled:

- A PC with a motherboard supporting AMD CrossFireX™ and two CrossFireX™-compatible Radeon graphics cards.
- Microsoft Windows 7, Windows 8.1 or Windows 10 with the Radeon Crimson graphics drivers.
- CrossFire™ enabled under AMD Radeon Settings→Preferences→Radeon Additional Settings→Gaming→CrossFire.
- To compile the sample, Microsoft Visual Studio 2013 is required.

Sample Code Overview

The sample's source code is arranged in three files:

1. SimpleMGPU.cpp – contains the main() function and most of the sample's logic
2. D3DHelper.cpp – contains lower-level rendering code
3. LvrLogger.cpp – contains code to log messages to the terminal window and measure frame rates, as well as some general purpose helper functions

Initialization

All of the initialization, such as loading the LiquidVR DLL and creation of the required interfaces is performed in the Init() function located in the SimpleMGPU.cpp file.

Load the LiquidVR DLL and obtain a pointer to the ALVRFactory interface used to obtain other LiquidVR interfaces:

```
ALVR_RESULT res = ALVR_OK;
g_hLiquidVRDLL = LoadLibraryW(ALVR_DLL_NAME);
CHECK_RETURN(g_hLiquidVRDLL != NULL, ALVR_FAIL, L"DLL " << ALVR_DLL_NAME << L" is not found");
ALVRInit_Fn pInit = (ALVRInit_Fn)GetProcAddress(g_hLiquidVRDLL, ALVR_INIT_FUNCTION_NAME);
res = pInit(ALVR_FULL_VERSION, (void**)&g_pFactory);
CHECK_ALVR_ERROR_RETURN(res, ALVR_INIT_FUNCTION_NAME << L"failed");
```

Create and enable GPU Affinity:

```
res = g_pFactory->CreateGpuAffinity(&m_pLvrAffinity);
CHECK_ALVR_ERROR_RETURN(res, L"CreateGpuAffinity() failed");
res = m_pLvrAffinity->EnableGpuAffinity(ALVR_GPU_AFFINITY_FLAGS_NONE);
CHECK_ALVR_ERROR_RETURN(res, L"EnableGpuAffinity() failed");
```

Create and wrap a D3D11 device (refer to the *D3DHelper* class for more details. Please note that the *D3DHelper* class is shared among several samples):

```
res = g_D3DHelper.CreateD3D11Device();
CHECK_ALVR_ERROR_RETURN(res, L"CreateD3D11Device() failed");
CComPtr<ID3D11Device>                pd3dDeviceWrapped;
CComPtr<ID3D11DeviceContext>         pd3dDeviceContextWrapped;
res = m_pLvrAffinity->WrapDeviceD3D11(g_D3DHelper.m_pd3dDevice, &pd3dDeviceWrapped,
&pd3dDeviceContextWrapped, &g_pLvrDeviceContext);
CHECK_ALVR_ERROR_RETURN(res, L"WrapDeviceD3D11() failed");
g_D3DHelper.m_pd3dDevice = pd3dDeviceWrapped;
g_D3DHelper.m_pd3dDeviceContext = pd3dDeviceContextWrapped;
```

From this point on use only the “wrapped” D3D device and context for any D3D rendering calls that require the *ID3D11Device* and the *ID3D11DeviceContext* interfaces. Please note the last two lines in the code above, which replace pointers to the original *ID3D11Device* and the *ID3D11DeviceContext* interfaces in the *g_D3DHelper* object with their “wrapped” counterparts to ensure that the original interfaces can no longer be used. The *WrapDeviceD3D11* method also returns a pointer to the *ALVRMultiGpuDeviceContext* interface. Wrapping allows LiquidVR to intercept certain D3D11 calls and handle them in a specific way needed for proper functioning of LiquidVR.

Set GPU affinity to send commands to both GPUs simultaneously:

```
res = g_pLvrDeviceContext->SetGpuRenderAffinity(GPUMASK_BOTH);
CHECK_ALVR_ERROR_RETURN(res, L"SetGpuRenderAffinity(GPUMASK_BOTH) failed");
```

Create GPU synchronization objects:

```
res = g_pLvrDevice->CreateGpuSemaphore(&g_pRenderComplete0);
CHECK_ALVR_ERROR_RETURN(res, L"CreateGpuSemaphore() failed");
res = g_pLvrDevice->CreateGpuSemaphore(&g_pRenderComplete1);
CHECK_ALVR_ERROR_RETURN(res, L"CreateGpuSemaphore() failed");
res = g_pLvrDevice->CreateGpuSemaphore(&g_pTransferComplete0);
CHECK_ALVR_ERROR_RETURN(res, L"CreateGpuSemaphore() failed");
res = g_pLvrDevice->CreateGpuSemaphore(&g_pTransferComplete1);
CHECK_ALVR_ERROR_RETURN(res, L"CreateGpuSemaphore() failed");
res = g_pLvrDevice->CreateFence(&g_pFenceD3D11);
CHECK_ALVR_ERROR_RETURN(res, L"CreateFence() failed");
```

A GPU semaphore is a synchronization object allowing to synchronize operations performed on different GPUs by letting one GPU to wait on a semaphore that is being triggered on another GPU, as well as synchronize different queues of the same GPU.

A GPU fence is another synchronization object allowing to synchronize operations between a GPU and the CPU by letting the CPU wait on a fence that is being triggered by a GPU.

For more information about GPU semaphores and fences please refer to the LiquidVR API documentation.

Create a swap chain using the *D3DHelper::CreateSwapChain* method:

```
res = g_D3DHelper.CreateSwapChain(g_hWindow, g_iBackbufferCount);
CHECK_ALVR_ERROR_RETURN(res, L"CreateSwapChain() failed");
```

Create a 3D scene using the *D3DHelper::Create3DScene* method:

```
res = g_D3DHelper.Create3DScene(width, height, true);
CHECK_ALVR_ERROR_RETURN(res, L"Create3DScene() failed");
```

Mark the buffer containing vertex shader parameters used to render the views for the left and the right eyes as “instanced”:

```
g_pLvrDeviceContext->MarkResourceAsInstanced(g_D3DHelper.m_pConstantBuffer);
```

Marking a resource as “instanced” creates an individual copy of this resource on each GPU since they need to be different to produce the left and the right eye’s views of a stereoscopic scene.

Rendering a 3D Scene

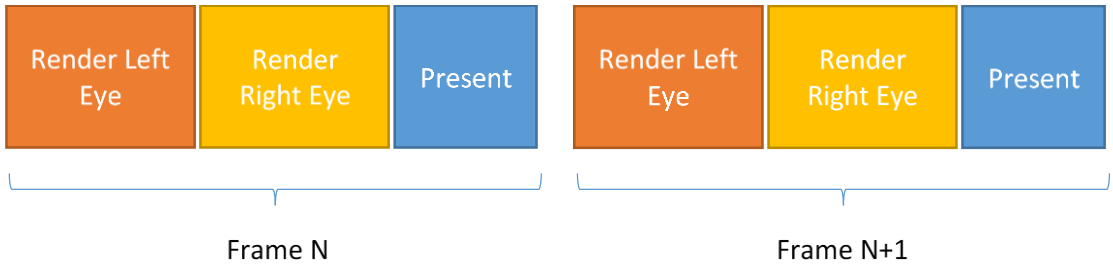
Rendering of the scene is performed from the main loop. The rendering process includes both eye-specific and non-specific operations.

When two or more GPUs are working in together in CrossFire mode, the display can only be connected to the primary graphics card, with all other cards working as slaves, performing rendering tasks, but unable to present. Hence, presentation must always happen on the primary GPU. Any images rendered on slave GPUs must be transferred to the primary GPU before they can be displayed.

There are three different ways of rendering a stereoscopic scene, each being demonstrated by the multi-GPU:

1. Rendering the image for both eyes on a single GPU (multi-GPU mode is OFF).

   Rendering is performed on GPU0 only for both eyes. This is equivalent to having only a single GPU in the system. Each eye’s image is rendered sequentially, once both eyes’ views are rendered, they are presented on the display.



   This is being accomplished with the following code:

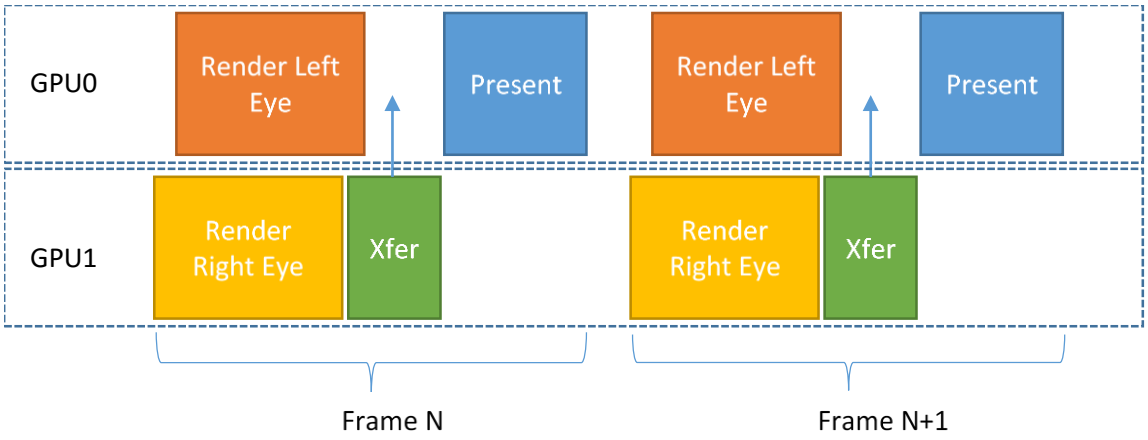
```
// submit right eye
g_D3DHelper.SetupViewportForEye(false);
g_D3DHelper.SetupMatrixForEye(false);
g_D3DHelper.RenderFrame(g_iDrawRepeat);
// submit left eye
g_D3DHelper.SetupViewportForEye(true);
g_D3DHelper.SetupMatrixForEye(true);
g_D3DHelper.RenderFrame(g_iDrawRepeat);
```

2. Rendering the image for each eye on a different GPU (multi-GPU mode is ON). Submissions to each GPU are performed explicitly and individually (Multiple Submission mode).

   Rendering for the left eye is performed on GPU0, while rendering for the right eye is performed on GPU1. The right eye’s image is transferred from GPU1 to GPU0 after it has been rendered and before presentation can occur. Presentation starts when both GPU0 is finished rendering the left eye’s image and GPU1 is finished rendering the right eye’s image and transferring it back to GPU0.

   Considering that GPU1 needs to perform the transfer of the rendered right eye’s image to GPU0, it is useful to submit the rendering job for the right eye to GPU1 first, so that the transfer of the rendered

image could occur at least partially while GPU0 is still rendering the left eye's image. This can be achieved by manually scheduling submission of jobs to GPU1 before submitting jobs to GPU0 using the Multiple Submission mode, which is implemented by explicitly setting the GPU affinity to *GPUMASK_RIGHT* for the right eye and *GPUMASK_LEFT* for the left eye:



This can be accomplished with the following code:

```
// submit right eye first
SetGpuAffinity(GPUMASK_RIGHT);
g_D3DHelper.SetupViewportForEye(false);
g_D3DHelper.SetupMatrixForEye(false);
g_D3DHelper.RenderFrame(g_iDrawRepeat);
// submit left eye
SetGpuAffinity(GPUMASK_LEFT);
g_D3DHelper.SetupViewportForEye(true);
g_D3DHelper.SetupMatrixForEye(true);
g_D3DHelper.RenderFrame(g_iDrawRepeat);
```

The multiple submission mode provides for the most optimal job scheduling between the two GPUs. This approach, however, has an increased overhead caused by the extra call to *RenderFrame()*, which sets up the rendering process on a GPU. Depending on the complexity of the scene and the number of parameters that need to be passed to each GPU may or may not be significant.

3. Rendering the image for each eye on a different GPU (multi-GPU mode is ON). Submissions to both GPUs are performed together in a single call (Single Submission mode).

Likewise, rendering for the left eye is performed on GPU0, while rendering for the right eye is performed on GPU1. The right eye's image is transferred from GPU1 to GPU0 after it has been rendered and before presentation can occur. Presentation starts when both GPU0 is finished rendering the left eye's image and GPU1 is finished rendering the right eye's image and transferring it back to GPU0.

The following code implements this model:

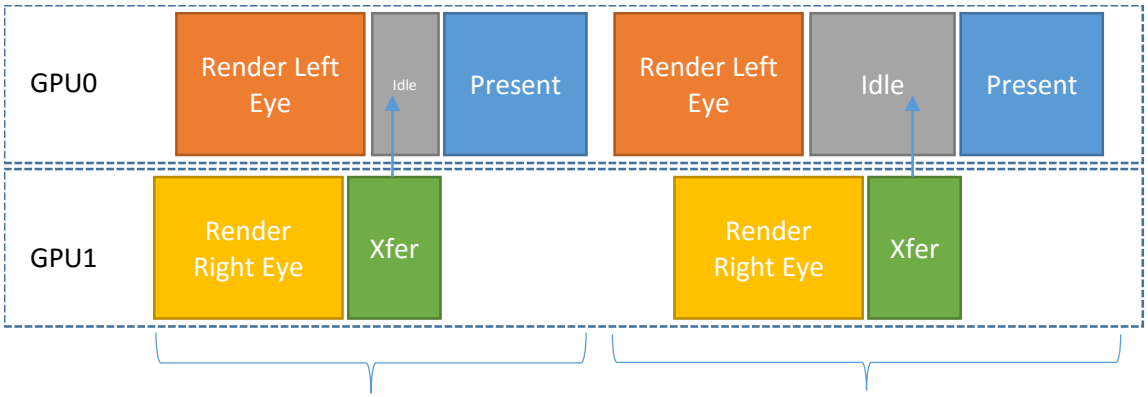
```
// setup right eye
SetGpuAffinity(GPUMASK_RIGHT);
g_D3DHelper.SetupViewportForEye(false);
g_D3DHelper.SetupMatrixForEye(false);
// setup left eye
SetGpuAffinity(GPUMASK_LEFT);
g_D3DHelper.SetupViewportForEye(true);
g_D3DHelper.SetupMatrixForEye(true);
// draw cube in both GPUs
SetGpuAffinity(GPUMASK_BOTH);
```

```
g_D3DHelper.RenderFrame(g_iDrawRepeat);
```

Unlike the multiple submission mode, the single submission mode starts rendering on both GPUs with a single call to *RenderFrame()*. This reduces the overhead of submitting jobs to GPU0 and GPU1 separately, however such technique does not allow to control which GPU would start rendering first. Therefore, two scenarios are possible:

1. Rendering starts on GPU1 first. This is a more optimal scenario closely resembling the task schedule achieved in the multiple submission mode. GPU0 cannot start presenting the frame until the transfer of the right eye's image from GPU1 to GPU0 us completed, but the idle time on GPU0 before presenting the complete image is minimized.
2. Rendering starts on GPU0 first. Assuming that rendering of the left and the right eye's views takes approximately the same amount of time, which is likely to be the case in most real life scenarios, the presentation operation would have to wait longer for the right eye's image being transferred from GPU1, resulting in longer idle time on GPU0 and, as the result, longer overall frame time.

Both scenarios are random and equally probable and the application has no control over which scenario would play out.



Scenario 1 – right eye rendering starts first Scenario 2 – left eye rendering starts first

The choice between these two techniques depends on where the bottleneck is. The multiple submission mode is clearly advantageous when GPU loads are heavy and GPU performance becomes the bottleneck. The single submission mode might prove to be beneficial with lighter GPU, but heavier CPU loads by reducing the amount of work needed to be done to set up the rendering process. The logic of the code implementing the single submission mode also resembles the code implementing the single GPU scenario, which may be of benefit when writing an application that needs to work on both single- and multi-GPU setups.

Transferring Data from GPU1 to GPU0

After GPU1 completes rendering of the right eye's image, it needs to be transferred to GPU0 to be presented on a display connected to GPU0, which is always the case when CrossFire is enabled. It is therefore necessary for GPU0 to wait until the transfer from GPU1 is completed before it can start presenting the frame. Not doing so would result in visible artifacts in the right eye's field of view.

LiquidVR supports two methods of synchronization between GPUs – automatic and manual. While the SimpleMGPU sample does not offer any user interface to switch between the two methods for simplicity's

sake, it provides the code that demonstrates the use of both methods. You can switch between automatic and manual synchronization by setting the *g_eTransferSyncMode* variable in SimpleMGPU.cpp to *TRANSFER_SYNC_AUTOMATIC* or *TRANSFER_SYNC_MANUAL* respectively. The respective calls to the LiquidVR API implementing transfer synchronization between GPUs are located in the *TransferEye()* function.

Automatic Synchronization

Automatic synchronization is implemented with a single call to the *ALVRMultiGpuDeviceContext::TransferResource()* method. This would transfer the specified rectangle using GPU1's 3D engine from GPU1 to GPU0 ensuring that all 3D operations on both GPUs have completed.

The following code performs the transfer:

```
case TRANSFER_SYNC_AUTOMATIC:
        // transfer from right to left engine
        res = g_pLvrDeviceContext->TransferResource(pResource, pResource, 1, 0, 0, 0, &rect, &rect);
        break;
```

This approach works in all cases, however, depending on how the scene is drawn, it may incur some unnecessary overhead since it makes no assumptions about the scene's drawing algorithm.

In many cases some assumptions about how a scene is drawn can be made, which allows for additional optimizations which can reduce GPU idle time and improve frame rendering time. In such cases manual synchronization is recommended.

Manual Synchronization

The drawing process in the sample includes the following steps:

1. Clearing of the entire frame, encompassing both eyes' field of view
2. Rendering the left eye's view on GPU0
3. Rendering the right eye's view on GPU1
4. Transferring the right eye's view from GPU1 to the combined view for both eyes on GPU0
5. Presentation of the combined view on GPU0

Two synchronization points are necessary:

1. Step 4 on GPU1 must commence only after steps 1 and 2 have completed on GPU0
2. Step 5 on GPU0 must commence only after Step 4 has been completed by GPU1

Manual synchronization is implemented with the following code:

```
case TRANSFER_SYNC_MANUAL:
{
    // flip-flop semaphores to avoid reuse before completion of previous frame
    CComPtr<ALVRGpuSemaphore> pRenderComplete = g_bTransferFlip ? g_pRenderComplete0 : g_pRenderComplete1;
    CComPtr<ALVRGpuSemaphore> pTransferComplete = g_bTransferFlip ? g_pTransferComplete0 :
                                                                    g_pTransferComplete1;

    g_bTransferFlip = !g_bTransferFlip;

    // right engine waits for the left engine for D3D render completion using semaphore
    res = g_pLvrDevice->QueueSemaphoreSignal(ALVR_GPU_ENGINE_3D, 0, pRenderComplete);   // GPUMASK_LEFT
    // has index 0
    res = g_pLvrDevice->QueueSemaphoreWait(ALVR_GPU_ENGINE_3D, 1, pRenderComplete);     // GPUMASK_RIGHT
    // has index 1
```

```
    // transfer from right to left engine - transfer happens in right engine
    g_pLvrDeviceContext->TransferResourceEx(pResource, pResource, 1, 0, 0, 0, &rect, &rect,
                                            ALVR_GPU_ENGINE_3D, false);

    // left engine waits for the right engine for transfer completion using semaphore
    res = g_pLvrDevice->QueueSemaphoreSignal(ALVR_GPU_ENGINE_3D, 1, pTransferComplete);
    res = g_pLvrDevice->QueueSemaphoreWait(ALVR_GPU_ENGINE_3D, 0, pTransferComplete);
}
break;
```

The two synchronization points described above are implemented with the help of two sets of GPU semaphores – one set for the first synchronization point and one for the second. All GPU synchronization objects such as semaphores and fences have a limitation, which does not allow for them to be reused until they have been signalled and waited on, therefore at least two objects are required in each set to ensure that they go through the full use cycle before they are reused (depending on the rendering speed more than two objects in each set might be necessary). They are alternated on each subsequent frame based on the value of the *g_bTransferFlip* variable.

Since both eyes' views share the same presentable texture on GPU0, the *pRenderComplete* semaphore ensures the transfer from GPU1 to GPU0 does not start until clearing of the texture on GPU0 is finished. An instruction to signal the *pRenderComplete* semaphore is queued to GPU0, while an instruction to wait for the *pRenderComplete* semaphore to be signalled is queued to GPU1. After that an instruction to commence a transfer from GPU1 to GPU0 is queued to GPU1.

Following the above, an instruction to signal the *pTransferComplete* semaphore is queued to GPU1, while an instruction to wait for the *pTransferComplete* semaphore to be signalled is queued to GPU0, ensuring that GPU0 would be stalled until the transfer of the right eye's view has been completed. After that execution of GPU0's command queue is resumed and the entire frame is presented to the display by calling the *g_D3DHelper.Present()* method, which, in turn, executes the *IDXGISwapChain::Present()* API call.

This manual approach allows to optimize the synchronization process based on the drawing algorithm implemented in your application. For example, when the entire frame is not being cleared and rendering of the right eye's view is completely independent of any GPU0 activities, the first synchronization point could be eliminated, which would reduce idle time on GPU1.

No Synchronization

Not synchronizing GPU0 and GPU1 activities will cause various drawing artifacts, such as image corruption, flicker, etc. Always synchronize GPU activities in a multi-GPU configuration to obtain properly rendered images.

Further Notes

Run the sample and try toggling the multi-GPU mode by pressing the 'm' key on the keyboard. Observe the change in rendering time in both configurations.

Try toggling between the single and the multiple submission modes by pressing the 's' key on the keyboard and observe the change in rendering time.

Try changing the synchronization mode between automatic and manual by setting the value of the *g_eTransferSyncMode* variable to *TRANSFER_SYNC_AUTOMATIC* and *TRANSFER_SYNC_MANUAL*. Rebuild and re-run the sample.

Try simulating higher and lower GPU loads by increasing and decreasing the value of the *g_iDrawRepeat* variable. Observe the change in rendering time by turning the multi-GPU mode on and off at different GPU load levels.

Try switching between different synchronization modes at different levels of GPU load and observe the effect of different synchronization techniques.

Feel free to use this sample as a basis for your own application which renders a different scene using different methods. Try optimizing the manual synchronization procedure based on your own rendering technique and compare performance of automatic and manually optimized synchronization.

Remember to always run the binaries built using the Release configuration when looking at performance numbers.