AMD CrossFire guide for Direct3D11 applications

Monitors supporting high resolutions such as 4K resolution (3840x2160) or 5K resolution (5120x2880) have become very popular in PC gaming. However, depending on the underlying game engine implementation, 4K and 5K resolutions typically require 4-5 times more pixel processing compared to the more common 1080p resolution. To play games at 4K or 5K with multisampling antialiasing and high refresh rates, an important amount of GPU processing power is needed. Fortunately, using multiple GPUs can dramatically increase performance and improve the gaming experience at high resolutions. The Radeon R9 295X2 and Radeon R9 Fury X2 represent the fastest AMD cards and rely on a dual GPU solution. Beside dual GPU cards it is also possible to connect up to 4 AMD GPUs to a single motherboard and use them in different Multi-GPU modes.

To achieve the highest rendering performance on multi-GPU systems, care must be taken during the development of a game engine. This document gives recommendations to help developers achieve the highest performance on multi-GPU systems (MGPU systems).

Table of Contents

Using Radeon Settings to setup MGPU

An application profile is needed in order to use CrossFire. A profile is a set of CrossFire options and other performance and image quality settings. There are 2 types of profiles:

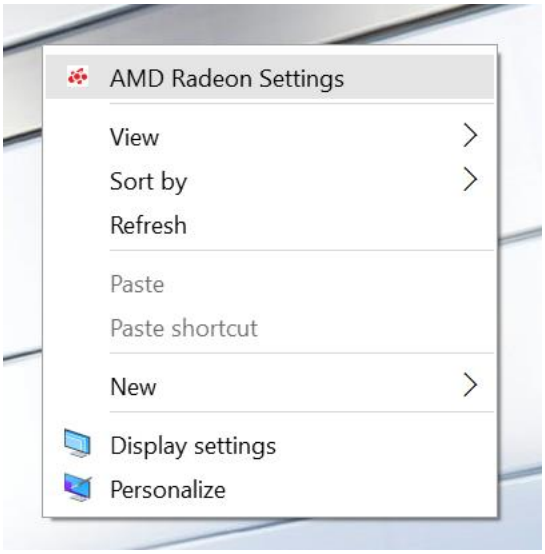
- User profile: a user profile is created using Radeon Settings and allows enabling and configuring CrossFire options.
- Driver profile: a driver profile is a set of CrossFire options and other optimizations that are built in the driver. A driver profile exists for many popular applications. The driver uses the application name to detect if a profile exists for that name.

Starting from version 16.15 of AMD Radeon Software Crimson edition, CrossFire is enabled by default for applications with a driver profile. For other applications, a user profile must be created to enable CrossFire. The user profile can be configured to override the driver profile.

It is important to note that CrossFire does not operate in windowed applications. The CrossFire profile created in Radeon Settings will only have when the application is running in full screen mode.

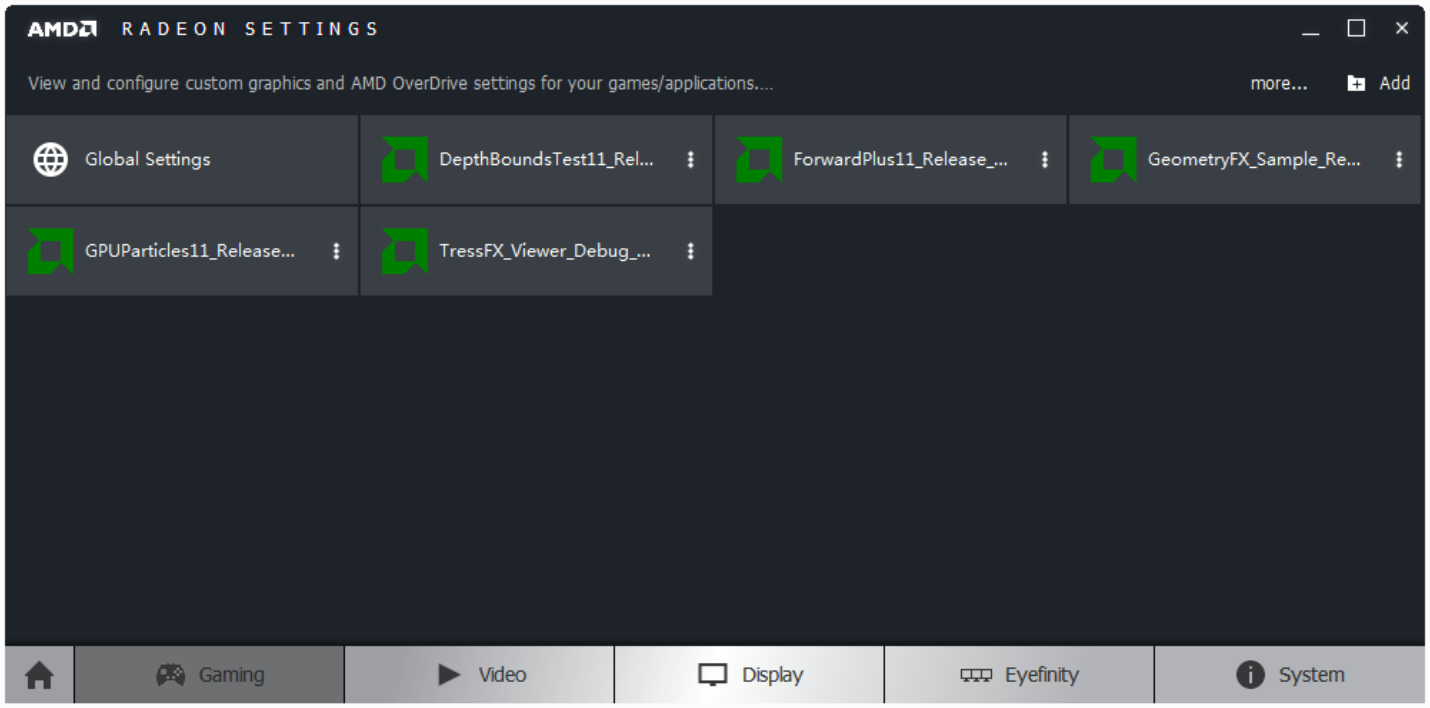
The following steps show how to change CrossFire settings:

Step1: Right-Click on the Desktop and select AMD Radeon Settings.

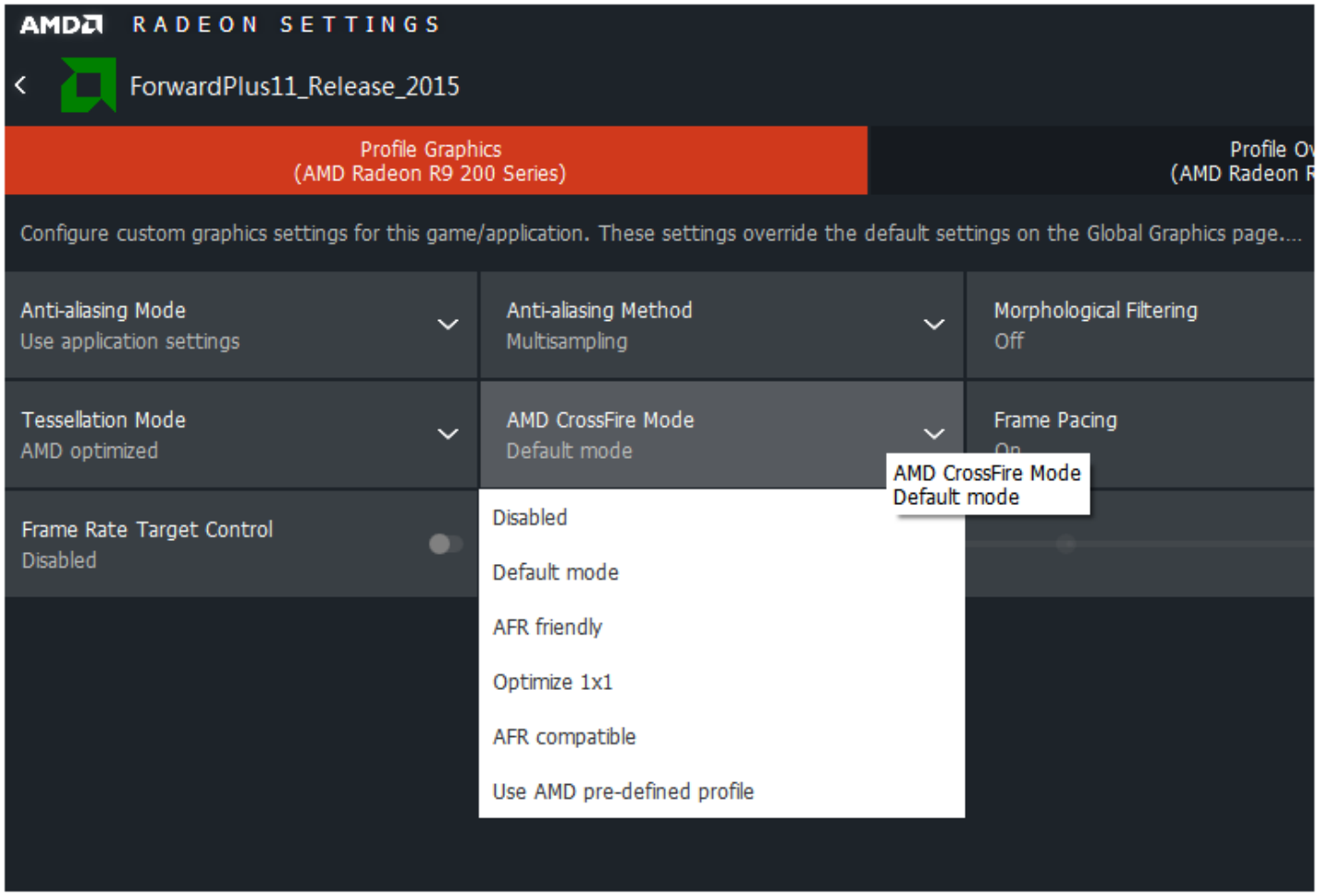


Step2: Click on the Gaming tab at the top of Radeon Settings.

Step3: Locate the title to modify and click on its icon. If the title is not in the list click the “Add" button located in the top-right corner and add the executable.



Step4: Locate the AMD CrossFire Mode box, and select one of the available options in the drop-down menu.



The available settings are:

- *Disabled*: The application runs in single GPU mode.
- *Default mode*: If the application has a driver profile it will be used. If a driver profile does not exist the application will run in single GPU mode.
- *AFR friendly*: The application will run in MGPU mode but resource tracking is disabled. The next chapter will describe resource tracking and what an AFR friendly application is.
- *Optimize 1x1*: The application runs in MGPU mode and resources will be tracked by the driver and will be rendered on each GPU if the render target corresponding to resource has a resolution of 1x1. There will no transfer of 1x1 render targets. This mode is useful if the overhead of the small transfers outweighs re-rendering resources on each frame.
- *AFR compatible*: The application runs in MGPU mode and resource tracking is enabled. This mode will be detailed in the next chapter.
- *Use AMD pre-defined profile*: This mode allows using one of the existing driver profiles and applying it to the current application.

An alternative to using Radeon Settings is to rename the D3D11 executable to one of the following:

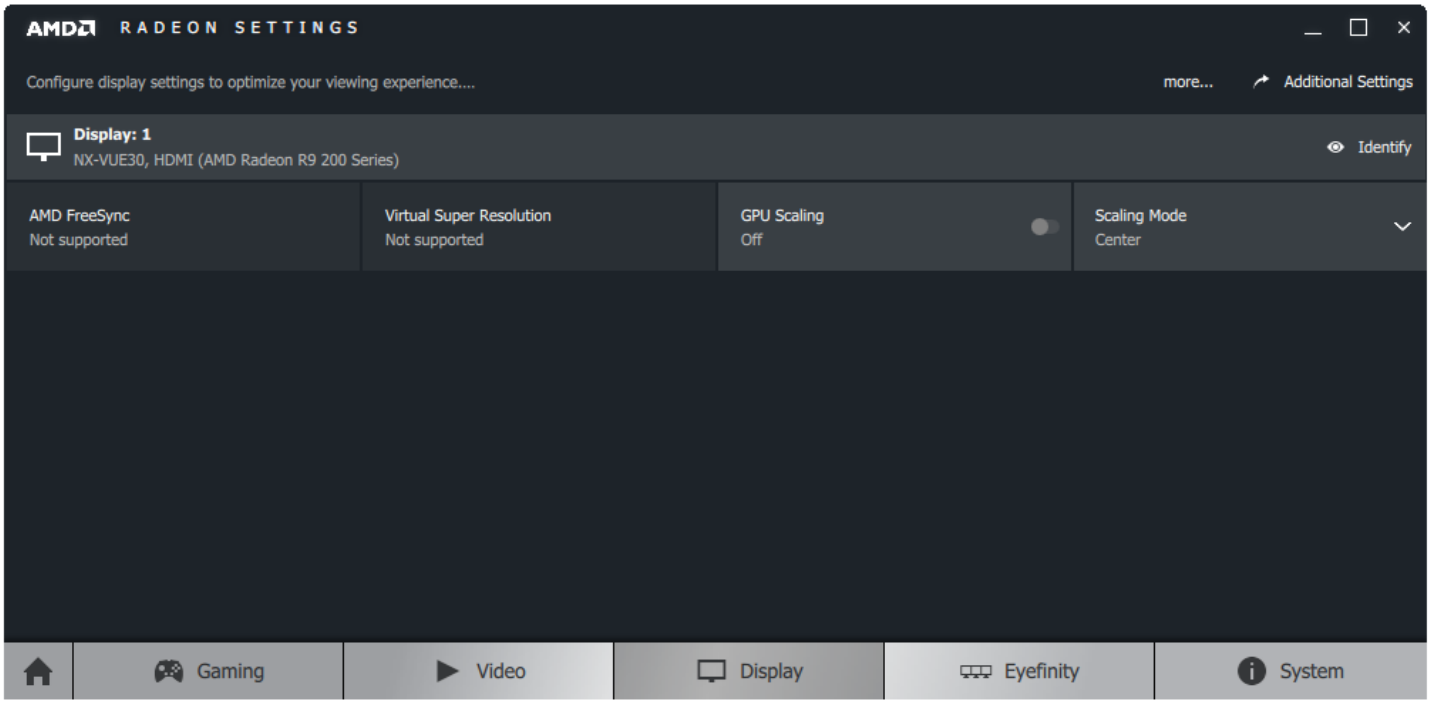
- *ForceSingleGPU.exe*: equivalent to Radeon Settings *Disabled*
- *AFR-FriendlyD3D.exe*: equivalent to Radeon Settings *AFR friendly*
- *CompatAFR-1x1.exe*: equivalent to Radeon Settings *Optimize 1x1*

Additional details about the different CrossFire modes can be found in Table 1.

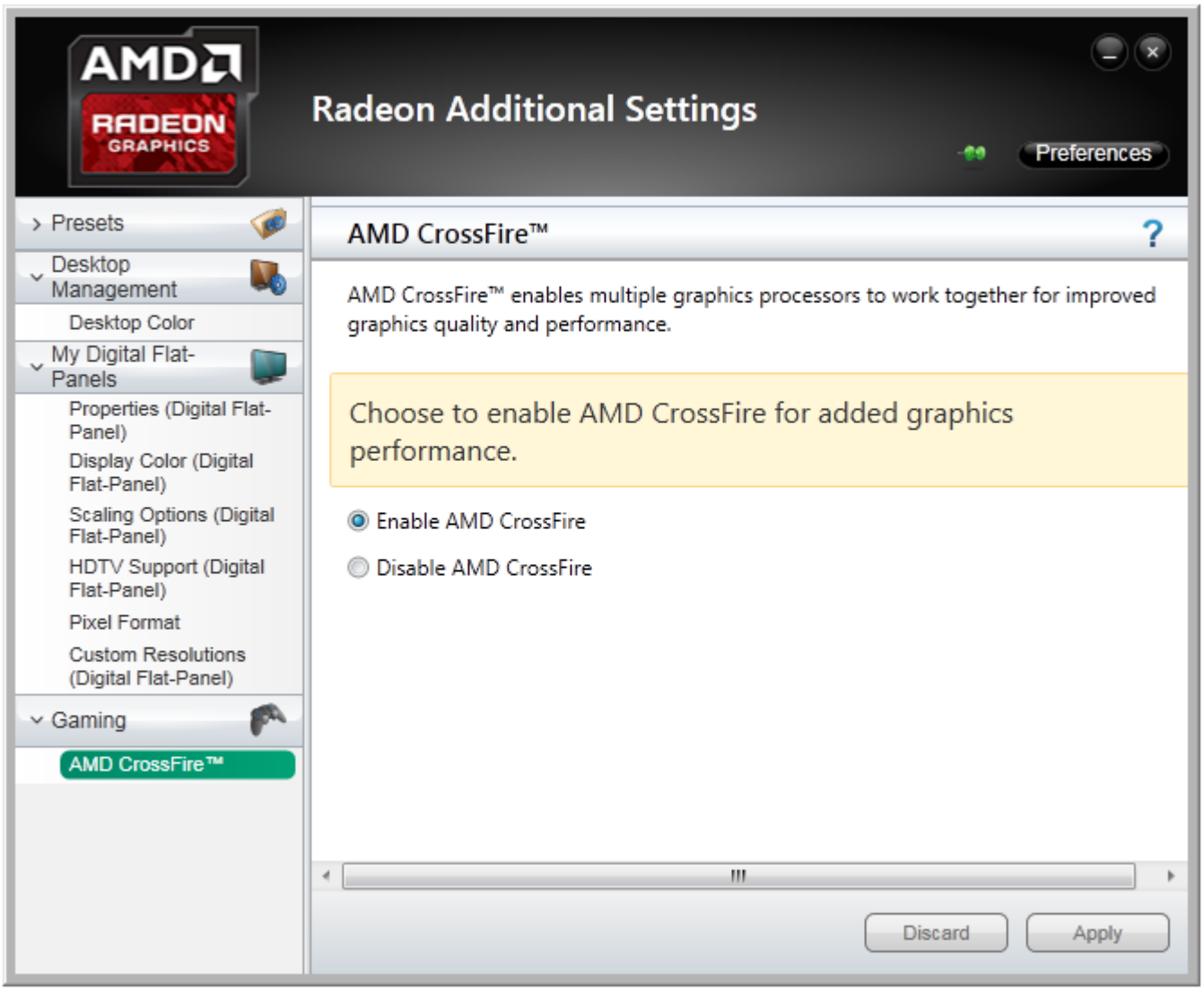
Turning off CrossFire

It is possible to completely disable CrossFire regardless of the presence of a profile for the application. The following steps describe how to disable and re-enable crossfire:

Step1: In Radeon Settings, click on the Display tab then click the “Additional Settings" button located in the top-right corner to bring the additional settings dialog.

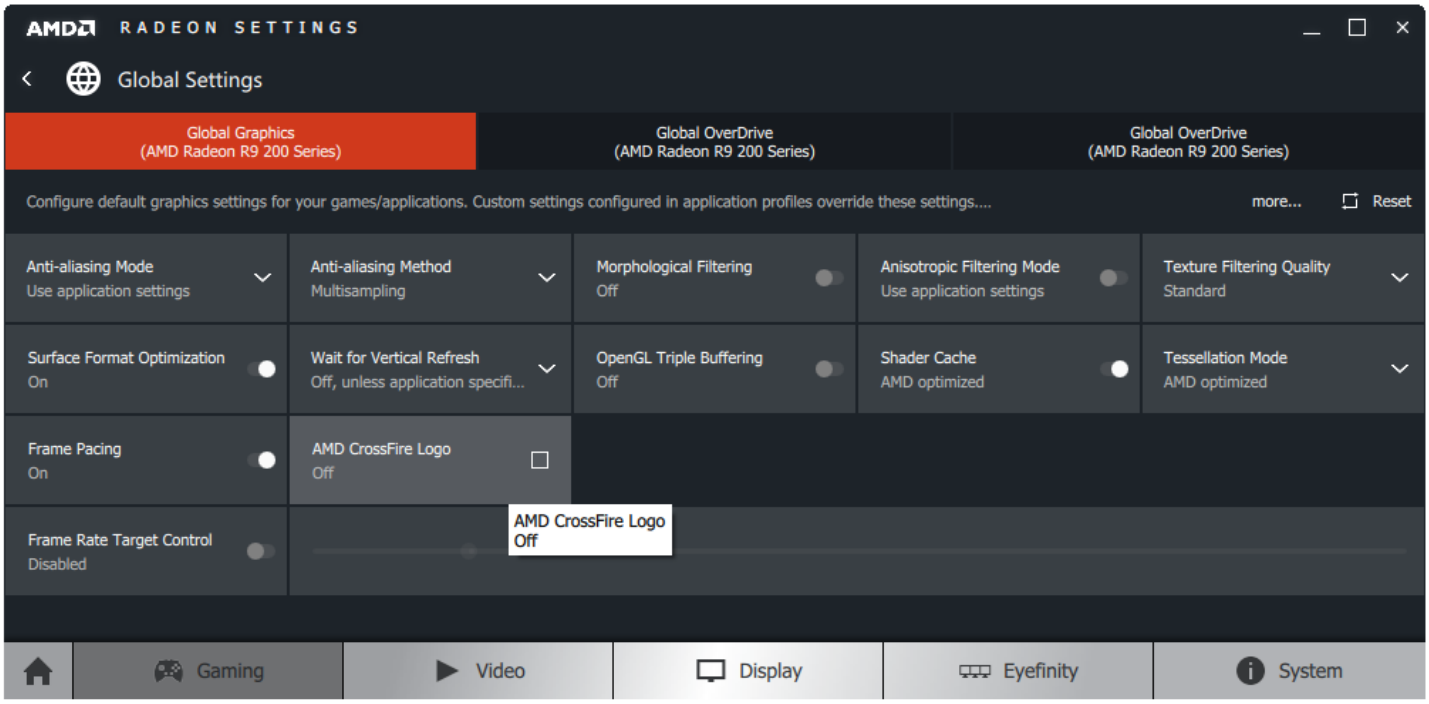


Step2: In the left menu select “AMD CrossFire” then choose “Disable AMD CrossFire” to turn off CrossFire or choose “Enable AMD CrossFire” to re-enable CrossFire through user or driver profiles.



Note that Crossfire cannot be disabled on Bi-GPU cards like the Radeon R9 295X2 or Radeon R9 Fury X2.

It is possible to display an AMD CrossFire logo within the game to check whether CrossFire is operating or not. The logo is only displayed when CrossFire is operating. The logo can be enabled in Radeon Settings: In the Gaming tab, select “Global Settings” and then check the box AMD CrossFire Logo as shown in the below image.



Implicit Multi-GPU with AMD CrossFire

This document focuses on D3D11 but note that concepts described in this chapter also apply to D3D9, D3D10 and OpenGL.

Alternate frame rendering (AFR)

AMD CrossFireX (or simply CrossFire) is a technology that takes advantage of multiple AMD GPUs in D3D11 applications. In CrossFire multiple GPUs appear to the programmer as a single device and the CrossFire driver employs a technique called *Alternate Frame Rendering* (AFR). In this mode, each GPU renders a separate frame. The driver queues up future frames of rendering commands without stalling the CPU then distributes the queued frames to the various GPUs. This is depicted in Figure 1.

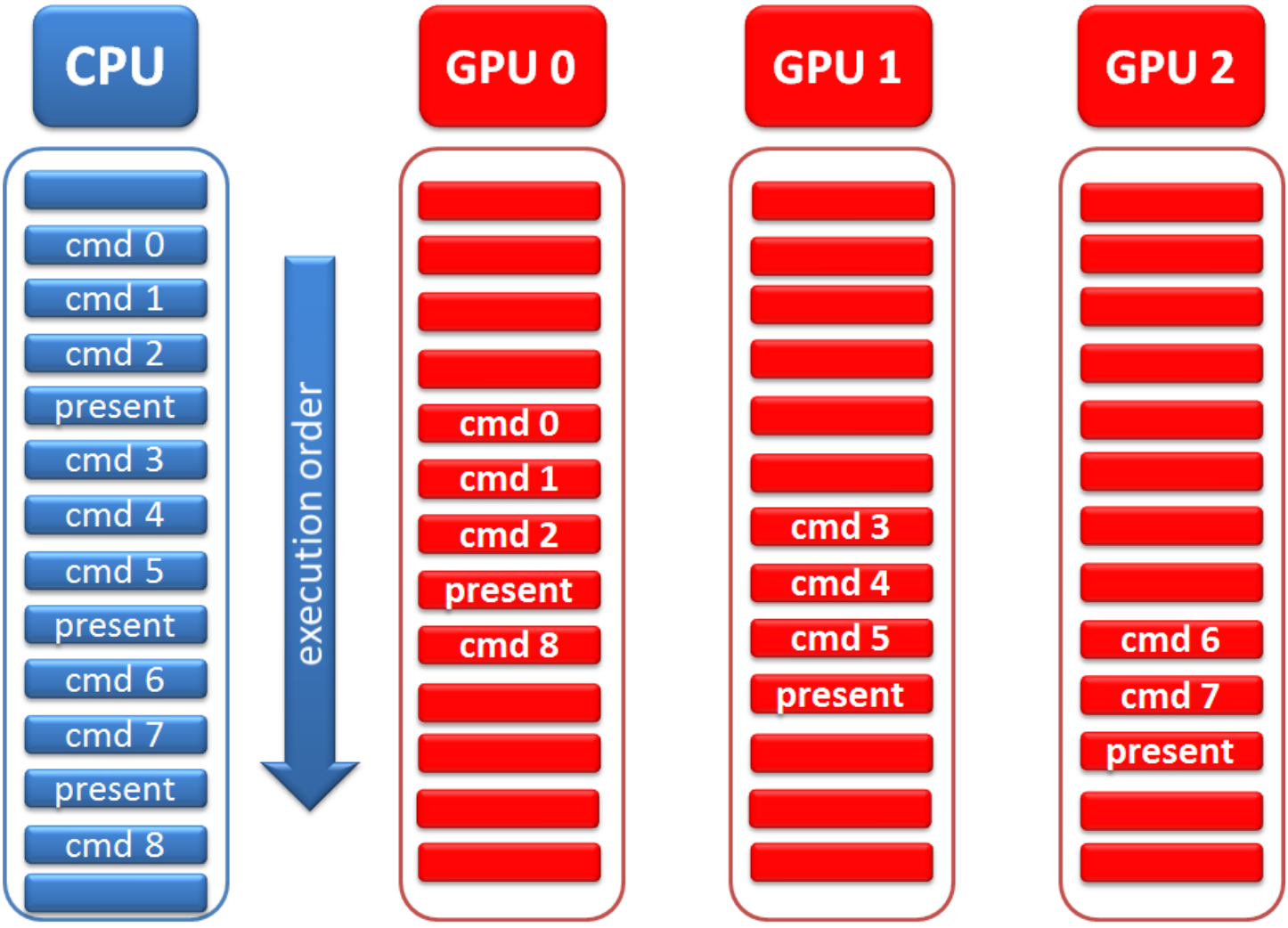


Figure 1

In the example of Figure 1 where the system has three GPUs, the application sequentially submits commands to the driver. The driver buffers the commands into command buffers and submits the command buffers of the first frame to GPU 0, the command buffers of the second frame to GPU 1, the command buffers of the third frame to GPU 2 then repeats the same process for the next sets of three frames. The 3 GPUs process the 3 frames in parallel. It should be noted that frames start to get processed by each GPU in order and that the GPUs are slightly out of sync with each other.

In both single GPU mode and MGPU mode, graphic commands are submitted to the driver in the same order. CPU command submission is generally much faster than their execution by the GPU. When multiple GPUs are present graphics commands can be consumed much faster than in a single GPU case. Because of this, CPU bound applications will get a limited performance gain from MGPU configurations. If the application does not submit enough commands, the GPUs will be starving. Synchronization commands can also starve the GPU. Finally, the API call SetMaximumFrameLatency can limit the number of queued frames and therefore the number of frames that can be processed in parallel.

It is important to address any CPU bottleneck that limits graphics performance. When using SetMaximumFrameLatency, it is recommended to set the number of frames that the system can queue to at least 1+N where N is the number of CrossFire GPUs. To test performance scaling, it is best to be as much GPU bound as possible by setting the video quality and the resolution at their highest.

Video memory updates

MGPU configurations do not have shared memory architecture, and for this reason each GPU has its own copy of all local video memory resources, such as textures, render targets, and geometry. If a resource is updated in a frame by GPU X and used in the next frame by GPU X+1, then the resource needs to be copied from GPU X to GPU X+1. This interdependency between frames requires synchronization between GPUs to ensure that the memory is always up to date on every GPU. If the driver believes that a GPU is trying to read from a stale copy of a writeable resource it will initiate a peer to peer (P2P) copy of the resource from the GPU it believes has the most up to date copy. In Figure 2 we can see that GPU 1 was about to use a stale copy of Resource A, and consequently the driver had to initiate a P2P copy from GPU 0 to GPU 1. A P2P copy may involve the CPU on some chipsets, and will always stall all GPUs, therefore serializing the whole process.

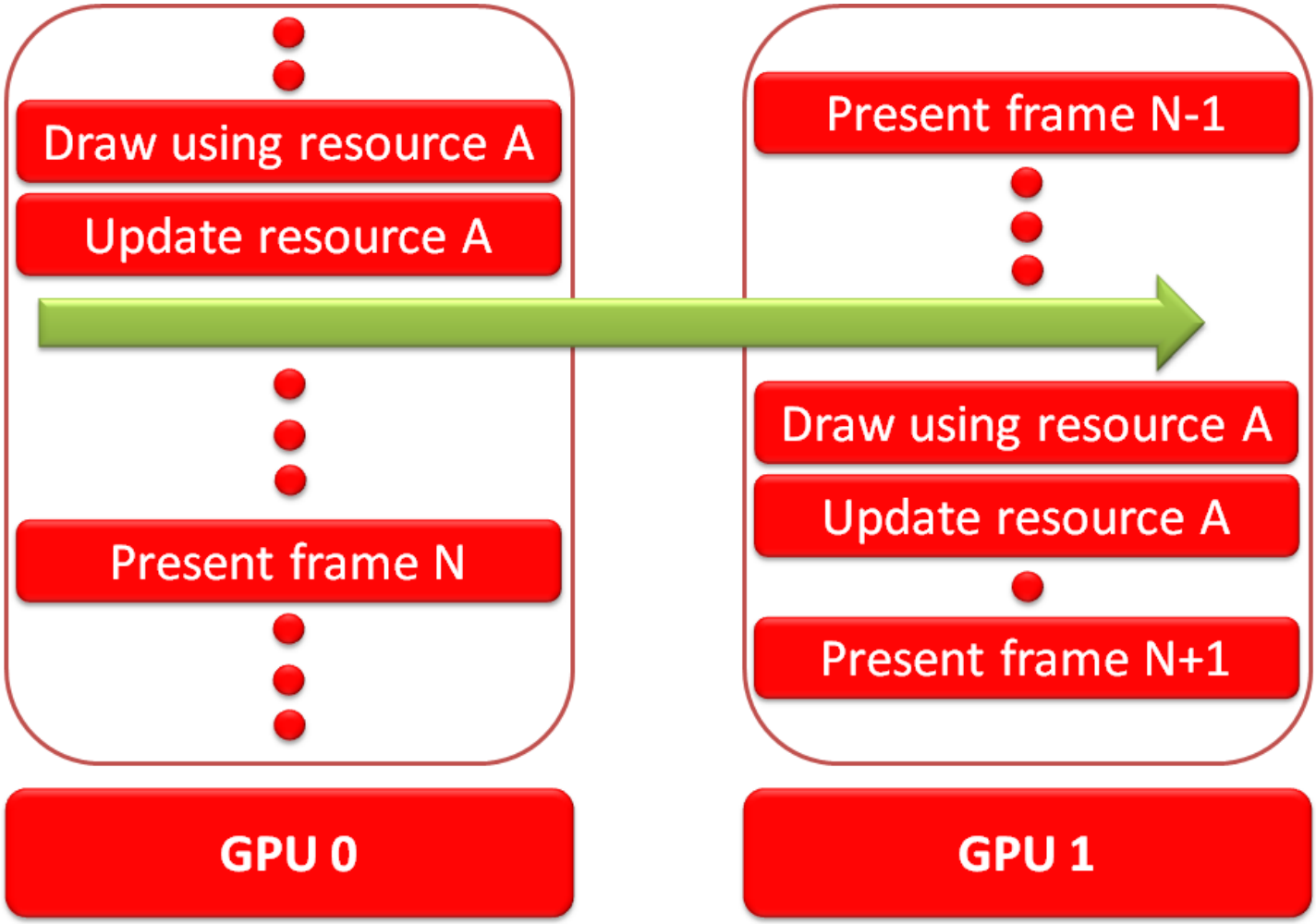


Figure 2

Excessive P2P copy of resources reduces performance and can lead to negative scaling. The next section describes how to avoid the expensive memory transfers. At a basic level developers must ensure that there are no dependencies between frames. When the application does not have inter-frame dependencies, the application is called *AFR-Friendly*.

Compatible AFR

By default the driver always checks if a resource has become stale and performs a P2P transfer whenever it is necessary. This mode is called *Compatible AFR* mode. In this mode the performance is highly affected by the amount of P2P transfers. The driver uses two heuristics to determine if a resource is stale. These two heuristics are defined as follows:

- **Heuristic 1**: A resource that is updated before it gets used within a frame is not stale.
- **Heuristic 2**: Let N be the number of GPUs in the system. A resource that is updated for N frames in a row before it gets used is not stale.

Let us consider examples of how a game may break these two heuristics:

In temporal filtering techniques each frame relies on resources rendered in the previous frames and this breaks **Heuristic 1**. In section **Explicit Multi-GPU control** we will describe the CrossFire API that among other features allows disabling transfer for a resource. With that API an AFR-Friendly approximation of temporal filtering can be achieved by disabling the transfer of resources involved in the filtering and making sure the filtering uses previous resources computed using the same GPU: Frame N uses resources computed in Frame N-M where M is the number of GPUs.

Now let us look at how a game might use **Heuristic 2**. Consider an example where a shadow map gets updated every 60 frames but used every frame. Let us say the system is equipped with 3 GPUs. Now let us follow the sequence of events: GPU 0 updates the shadow map then uses it. Meanwhile, GPU 1 needs to use the shadow map but it has a stale copy. Because of that, the shadow map is copied from GPU 0 to GPU 1 before it gets used in GPU 1. GPU 2 also needs a copy of the shadow map from GPU 1. To avoid the transfer the rendering can be changed to match **Heuristic 2** requirements. When the shadow map needs to be updated (60 frames have passed) the shadow map rendering is repeated 3 times (as many times as there are GPUs). This way each GPU has an up to date shadow map according to **Heuristic 2** and the transfer is therefore avoided.

Note on subresource or region updates

The heuristic described above determines whether a resource is stale or not. When a subresource is updated, e.g. a texture array layer or a mipmap level, the whole resource is considered updated. Similarly, if a region within a resource (or a region within a subresource) is updated the whole resource is considered updated. Care must be taken when updating a subresource or a region of a resource.

Consider the case from Figure 3 where different frames update different regions within a same resource. 2 GPUs are used in compatible AFR mode. In frame N, GPU 0 updates Region1 then reads from both Region1 and Region2. In frame N+1, GPU 1 updates Region2 then reads from both Region1 and Region2. According to **Heuristic 2** the resource is not stale and therefore the resource is not copied from GPU 0 to GPU 1. In Frame N+1 both Region1 and Region2 should have been updated before reading from the resource but this is not the case for Region1. Similarly, in Frame N+2 GPU 2 expects both Region1 and Region2 to be up to date but it is not the case of Region2.

The default behavior of the compatible AFR mode will likely lead to flickering and corruption in the example of Figure 3. This however can be solved by applying all updates in the same frame and possibly repeating the updates in the next frames as many times as there are GPUs to avoid breaking the 2 previously described heuristics. In section **Explicit Multi-GPU control** we will describe how to use the CrossFire API to notify the driver that only a sub-resource or region within a resource needs to be transferred.

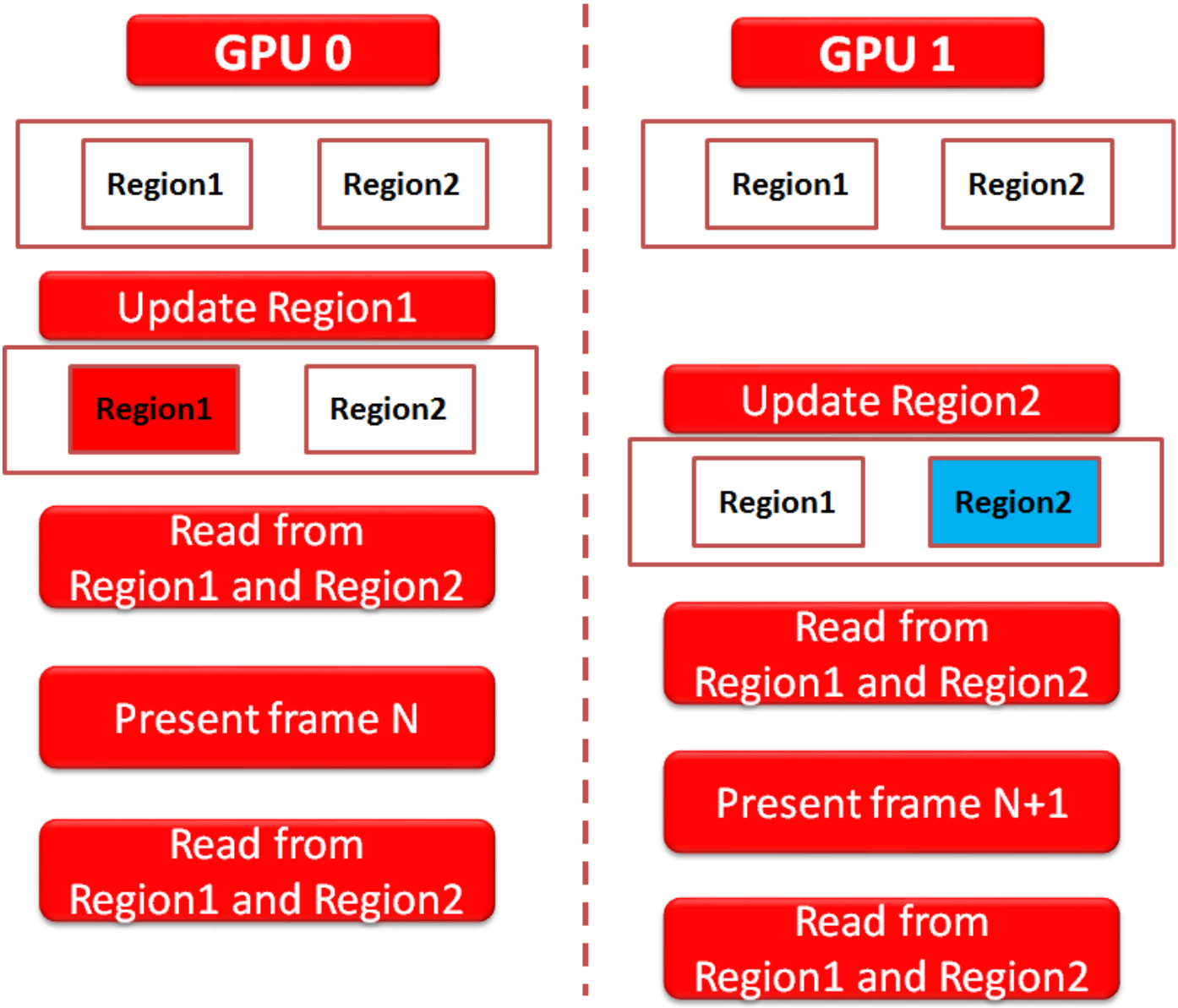


Figure 3

Mapping resources

Accessing a resource that has been mapped can cause the GPU to stall until the resource is unlocked by the driver. In a MGPU configuration, all GPUs are stalled until the resource is unlocked. Additionally, a broadcast takes place to update all GPUs. Any excess in resource mapping can strongly reduce performance.

As in single GPU configuration, it is advised to use MAP_DISCARD whenever it is applicable. The maximum recommended size for MAP_DISCARD dynamic buffers is 16MB. By default, the driver uses 32MB for a resource and its renamed versions and there can be up to 32 renames.

Note on when the driver initiates a transfer

The driver checks whether a resource is stale at the first use of the resource within a frame. If the resource is stale the driver inserts an asynchronous copy command at the end of the previous frame and a synchronization command that precedes the first use of the resource to ensure the copy finishes before the actual use of the resource. It is therefore recommended to have the first reference to the resource as late in the frame as possible since the copy will not start until the previous frame has completed and the copy will have to finish before the use of the resource.

In Figure 4 GPU 0 updates 2 resources A and B without using them before the present call. Because of **Heuristic 1** a transfer from of Resources A and B is initiated from GPU 0 to GPU 1 at present call. GPU 1 tries to read from resource A but it has to wait for the transfer of A to finish before using the resource. GPU 1 does not need to wait to use resource B because the transfer of B has already finished.

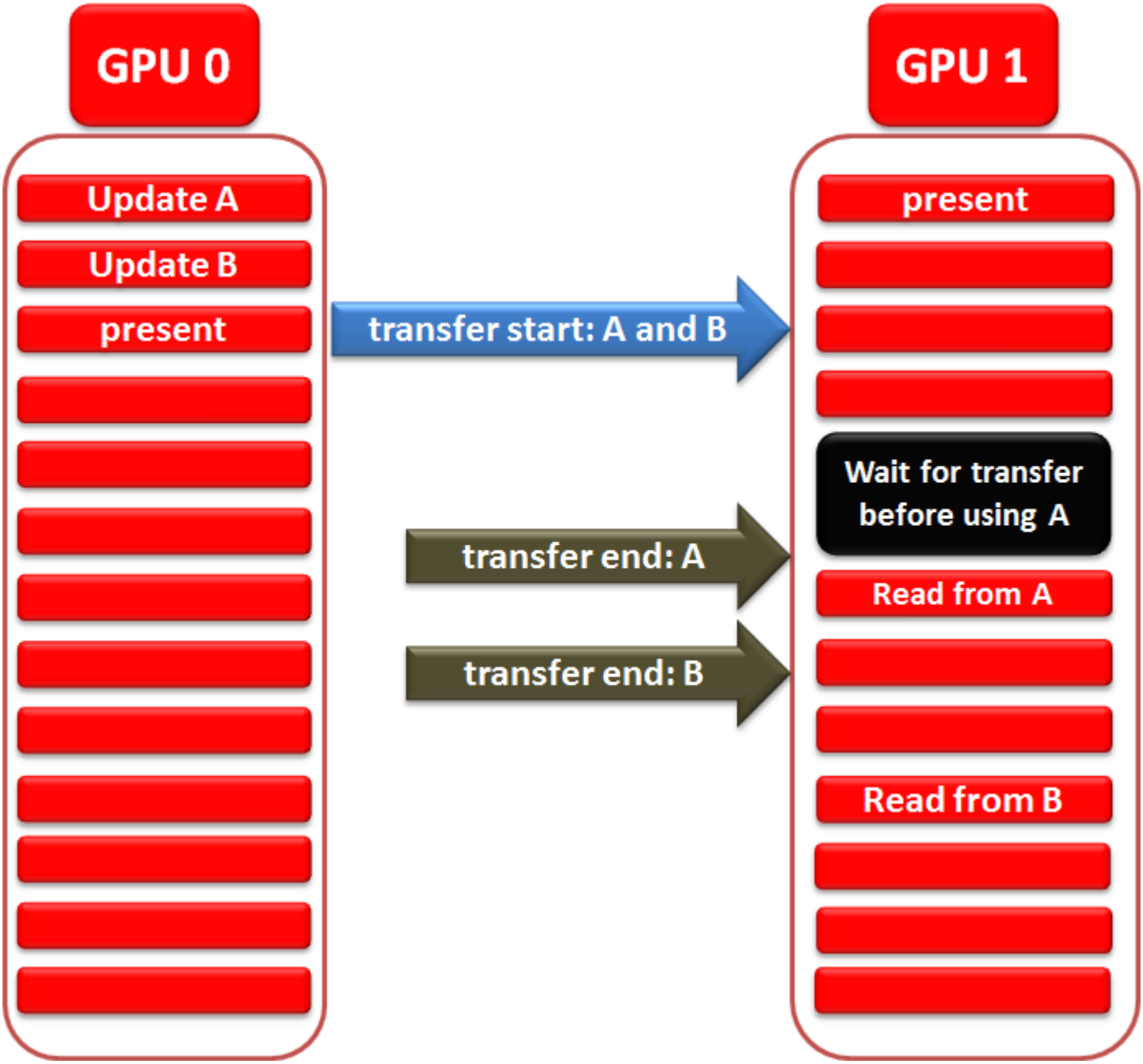


Figure 4

Queries

The use of queries is another key area where it is possible to introduce potentially AFR unfriendly behavior. If a query has the effect to limit the number of queued frames, this can totally destroy the chance to gain parallelism from AFR. If the application indefinitely waits for the result of a query, it will stall the CPU, which in turn starves the GPU command

buffers of work. Also, if the application changes drawing parameters based on the result of a query, this can lead to graphical artifacts. To avoid these problems:

- Avoid using queries wherever possible
- Make sure that a matched pair of BeginQuery() / EndQuery() calls occur within the same frame
- Avoid waiting on query results: obtain the result of a query at least N frames after it was issued (N is the number of GPUs)
- For queries issued every frame, create additional query objects for each GPU, and cycle through them
- For occlusion queries, consider a CPU based approach
- Never limit the number of queued frames through the use of queries or by locking the backbuffer

Application driver profile

For many applications the driver can have an internal profile where it performs a reduced set of checking, avoids unnecessary copies and applies useful optimizations like increasing the renaming limit for CPU updates. If the driver recognizes the name of an executable, its behavior will be fully guided by the profile for that application. Developers are encouraged to talk to their developer relations contacts about how to minimize dependencies between frames and possibly make a profile for their game.

Explicit Multi-GPU control using AMD APIs

The previous chapter presented the AFR-Friendly and the Compatible-AFR modes. The AFR-Friendly mode offers optimal performance but requires that all resources are used in an AFR-Friendly manner and this is not always possible. Unlike the AFR-Friendly mode the Compatible-AFR mode has the following disadvantages:

- The driver adds an overhead by checking every used resource to determine whether it is stale or not
- Memory transfers affect performance and require synchronizations that starve the GPUs from work.
- The rendering of frame needs to finish before starting a memory transfer to the next GPU and this is not optimal
- For certain resources a transfer can be skipped without affecting the rendering but the driver cannot automatically detect such cases

A driver profile can be added to minimize disadvantages of Compatible-AFR mode. A driver profile is not always optimal because the logics of the game engine are often unknown to the driver developer. The game developer can make better choices if he is given control over what resources require to be transferred, how they are transferred and when they are transferred.

AMD provides CrossFire API extensions to D3D11. The API gives the developer controls to minimize the limitations of the Compatible-AFR mode. The API provides the following advantages:

- Resources used in an AFR-Friendly manner can be flagged by the API and the driver will not check them. Driver overhead will be minimized and there will not be any transfers for these resources
- The API allows starting a transfer as soon as there are no more updates to a resource. Synchronizations are also guided by the API, therefore offering the best transfer performance
- The API lets the developer decide how to transfer the memory: peer to peer or a broadcast
- The driver profile becomes simplified and sometimes not even needed as memory optimizations are handled directly by the developer through the API

The AMD GPU Services Library

The AMD GPU Services Library (AGS) is part of AMD GPUOpen initiative. The following link can be used download AGS, its documentation and code samples. AGS provides an API to query information about the installed adapters. In particular, it gives information about the CrossFire configuration and also gives access to the CrossFire API extensions.

Detecting the presence of AMD CrossFire API

The AGS function agsDriverExtensions_Init initializes D3D11 AMD extensions. It returns a 32 bits mask that specifies what extensions are available in the installed AMD D3D11 driver. When the CrossFire API extension is available the flag AGS_EXTENSION_CROSSFIRE_API is returned by the function. The programmer has to make sure that the CrossFire extension is available before using it. agsDriverExtensions_Init has the following prototype:

```
// initialization of AGS
// context: AGS context created with agsInit
// device: a valid D3D11 device
// supported_extension: a pointer to a 32 bits mask of driver supported extensions
// AGS_SUCCESS is returned if the function succeeds; an error code is returned otherwise
AGSReturnCode agsDriverExtensions_Init(AGSContext* context, ID3D11Device* device, unsigned int*
supported_extension);
```

Detecting the number of GPUs in the system

The function agsGetCrossfireGPUCount returns the number of GPUs present in the system. The queried number of GPUs allows configuring parts of the engine to operate in a way that doesn't break the heuristics described in section **Alternate frame rendering** (AFR). agsGetCrossfireGPUCount has the following prototype:

```
// detecting the number of CrossFire GPUs
// context: AGS context created with agsInit
// gpu_count: a pointer to a int that holds the number of CrossFire GPUs
// AGS_SUCCESS is returned if the function succeeds; an error code is returned otherwise
AGSReturnCode agsGetCrossfireGPUCount(AGSContext* context, int* gpu_count);
```

The following function is an example showing how to initialize the CrossFire API and how to query the number of GPUs in the CrossFire chain:

```
void init_ags(ID3D11Device* d3d11_device, AGSContext*& ags_context, int& gpu_count, bool& use_cfx_api)
{
    assert(ags_context == nullptr);
    use_cfx_api = true;
    gpu_count = 0;

    AGSReturnCode ags_code = AGS_SUCCESS;
    AGSConfiguration ags_config;
    AGSGPUInfo ags_info;

    ags_config.crossfireMode = AGS_CROSSFIRE_MODE_EXPLICIT_AFR;
    ags_code = agsInit(&ags_context, &ags_config, &ags_info);
    handle_error(agsInit, ags_code);

    ags_code = agsGetCrossfireGPUCount(ags_context, &gpu_count);
    handle_error(agsGetCrossfireGPUCount, ags_code);

    if(gpu_count < 2)
    {
        use_cfx_api = false;
    }

    unsigned int extension_mask = 0;
    ags_code = agsDriverExtensions_Init(ags_context, d3d11_device, &extension_mask);
    handle_error(agsDriverExtensions_Init, ags_code);

    if((extension_mask & AGS_EXTENSION_CROSSFIRE_API) == 0)
    {
        use_cfx_api = false;
    }
}
```

CrossFire API transfer modes

The CrossFire API lets programmers explicitly mark the resources that need to be transferred from one GPU to the next one. It also allows specifying different transfer modes. To mark a resource with one of the AFR control modes available in the CrossFire API, the resource creation functions of ID3D11Device interface need to be replaced with one of the CrossFire API resource creation functions. The following table lists ID3D11Device resource creation functions and their CrossFire API counterpart.

ID3D11Device resource creation function	CrossFire API resource creation function
CreateBuffer	agsDriverExtensions_CreateBuffer
CreateTexture1D	agsDriverExtensions_CreateTexture1D
CreateTexture2D	agsDriverExtensions_CreateTexture2D
CreateTexture3D	agsDriverExtensions_CreateTexture3D

It is possible to use both ID3D11Device creation functions and the CrossFire API resource creation functions in the same program. Resources created using ID3D11Device functions will be tracked by the driver as described in section Implicit Multi-GPU with AMD CrossFire. The CrossFire API resource creation functions are similar to ID3D11Device functions except that the CrossFire API functions take 2 additional parameters: the AGS context and an AFR control mode. There are 5 AFR control modes:

AGS_AFR_TRANSFER_DISABLE: this mode disables driver tracking of a resource. Disabling tracking can greatly improve performance. This mode is used in 2 cases:

- A resource is only used in an AFR-friendly context
- It is guaranteed that not transferring the resource will not lead to erroneous rendering

AGS_AFR_TRANSFER_1STEP_P2P: enables a direct transfer to the next GPU. The scheduling of the transfer is controlled by the application.

AGS_AFR_TRANSFER_2STEP_NO_BROADCAST: enables a transfer to the next GPU but uses an intermediate system memory resource. The scheduling of the transfer is controlled by the application.

AGS_AFR_TRANSFER_2STEP_WITH_BROADCAST: enables broadcasting a resource to all GPUs. An intermediate system memory resource is used. The scheduling of the transfer is controlled by the application.

AGS_AFR_TRANSFER_DEFAULT: in this mode the resource creation is exactly the same as using ID3D11Device functions for resource creation. It should be noted that the behavior in this mode depends on the CrossFire profile. Table 1 describes how the profile affects this mode. The table particularly describes how different CrossFire options behave when the Crossfire control is implicit and explicit.

Radeon Settings CrossFire mode	Implicit control: The CrossFire API is not queried by the application	Explicit control: The CrossFire API is successfully queried by the application
Disabled	The application runs in single GPU mode	The CrossFire API cannot be queried. The application runs in single GPU mode
Default mode	If the application has a driver profile it will be used. If a driver profile does not exist the application will run in single GPU mode	If the application has a driver profile it will be used. If a driver profile does not exist, the CrossFire API cannot be queried and the application will run in single GPU mode
AFR friendly	The application runs in MGPU mode but resource tracking is disabled	Resource transfers are only applied when the application notifies the driver to do so. Resources created with the flag AFR_TRANSFER_DEFAULT or through the normal D3D11 API will not be tracked by the driver
Optimize 1x1	The application runs in MGPU mode and resources will be tracked by the driver and will be rendered on each GPU if their resolution is 1x1	Resource transfers are applied when the application notifies the driver to do so. Resources created with the flag AFR_TRANSFER_DEFAULT or through the normal D3D11 API will be tracked by the driver. Resources with dimensions of 1x1 will be rendered on each GPU
AFR Compatible	The application runs in MGPU mode and resource tracking is enabled	Resource transfers are applied when the application notifies the driver to do so. Resources created with the flag AFR_TRANSFER_DEFAULT or through the normal D3D11 API will be tracked by the driver
Use AMD pre-defined profile	The mode allows using one of the existing driver profiles and applying it to the current application	The mode allows using one of the existing driver profiles and applying it to the current application

Table 1

CrossFire API transfer notification

For AGS_AFR_TRANSFER_DEFAULT and AGS_AFR_TRANSFER_DISABLE marking the resource is the only needed operation. For the other transfer modes, 3 notification functions must be used to notify the driver when it is safe to start a transfer and wait for the transfer to complete. The 3 notification functions are:

```
// notify the driver that the application has begun reading or writing the resource
// context: AGS context
// resource: a pointer to the resource
// AGS_SUCCESS is returned if the function succeeds; an error code is returned otherwise
AGSReturnCode agsDriverExtensions_NotifyResourceBeginAllAccess(AGSContext* context, ID3D11Resource*
resource);

// notify the driver that the application has finished writing the resource
// context: AGS context
// resource: a pointer to the resource
// transfer_regions: an array of transfer regions. nullptr specifies the whole resource
// subresource_array: an array of subresource indices. nullptr specifies all subresources
// num_subresource: the number of subresources in subresource_array OR number of transfer_regions. 0 to
specifies all subresources and one transfer region (ignored if transfer_regions == nullptr)
// AGS_SUCCESS is returned if the function succeeds; an error code is returned otherwise
AGSReturnCode agsDriverExtensions_NotifyResourceEndWrites(AGSContext* context, ID3D11Resource*
resource, const D3D11_RECT* transfer_regions, const unsigned int* subresource_array, unsigned int
num_subresource);

// notify the driver that the application has finished reading or writing the resource
// context: AGS context
// resource: a pointer to the resource
// AGS_SUCCESS is returned if the function succeeds; an error code is returned otherwise
AGSReturnCode agsDriverExtensions_NotifyResourceEndAllAccess( AGSContext* context, ID3D11Resource*
resource);
```

It is not recommended to use the above functions with AGS_AFR_TRANSFER_DEFAULT or AGS_AFR_TRANSFER_DISABLE as they incur an overhead despite no memory transfer.

NotifyResourceBeginAllAccess notifies the driver that the program starts accessing a resource. The access operations can be a read or a write operation. NotifyResourceEndWrites notifies the driver that all write operations have finished. When the driver is notified that write operations have finished for a resource it can start the transfer if the target GPU or GPUs are not currently using their copy of the resource. NotifyResourceEndAllAccess notifies the driver that the resource is not being used anymore and that it is safe to use the resource as the target of a transfer.

For every NotifyResourceBeginAllAccess on a resource there has to be a NotifyResourceEndAllAccess for the same resource. A missing NotifyResourceBeginAllAccess or NotifyResourceEndAllAccess can lead to a deadlock.

NotifyResourceEndWrites can only be called between NotifyResourceBeginAllAccess and NotifyResourceEndAllAccess. Not doing so can lead to a deadlock.

Figure 5 shows the importance of NotifyResourceBeginAllAccess and NotifyResourceEndAllAccess. In the figure, when GPU 1 notifies the driver that it will begin accessing the resource, the driver stalls GPU 1 until the transfer of the resource from GPU 0 to GPU 1 finishes. When GPU 1 notifies the driver that it has finished writing to the resource, the

driver does not start the transfer of the resource to GPU 0 because the resource is in in use. The driver starts the transfer when GPU 0 notifies that it has ended access to the resource.

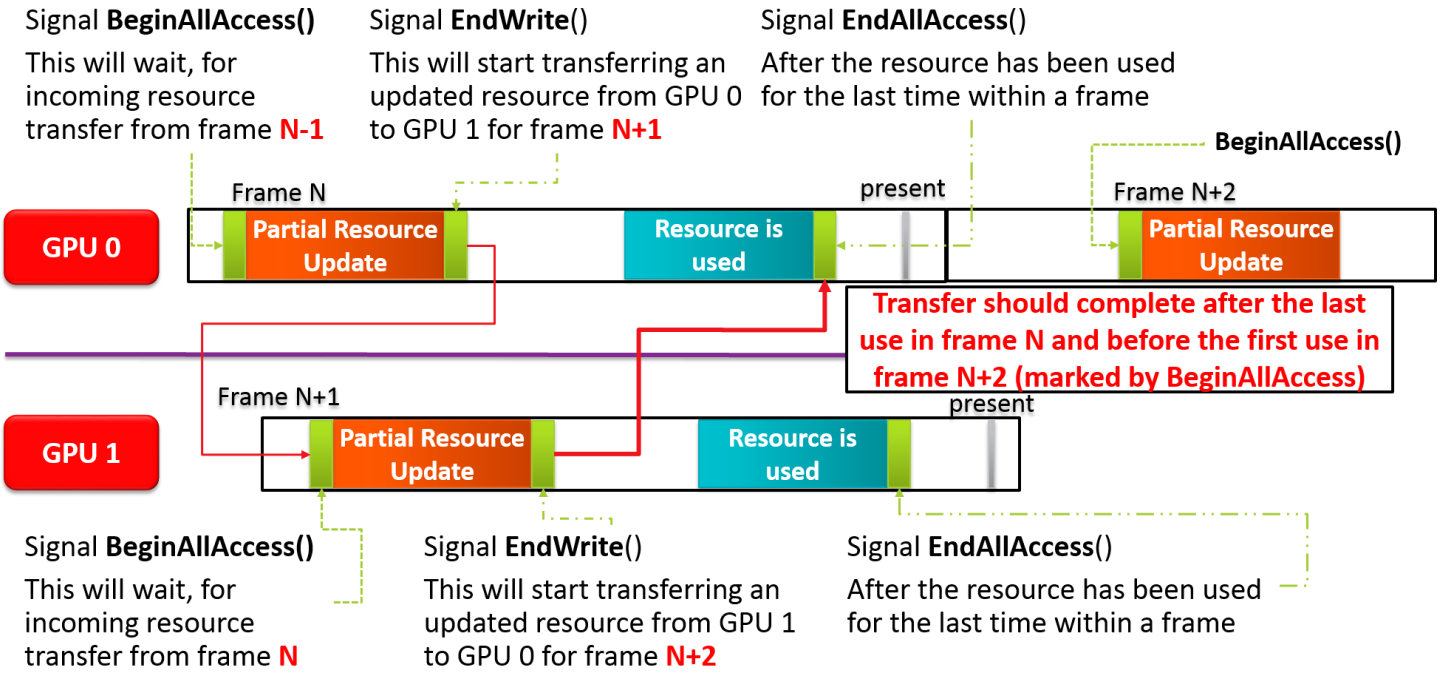


Figure 5

Reducing contention using 2-Steps-GPU-Transfer

As seen in Figure 5 NotifyResourceEndWrites cannot start the transfer if the resource is used by another GPU. Similarly, NotifyResourceBeginAllAccess blocks the execution if a transfer involving the resource is in progress. A possibility to improve the situation is to reorder the rendering passes to minimize waits. This is however not always possible. We describe a 2-Steps-GPU transfer technique that like reordering can also reduce waits. The technique requires creating an extra copy of the transferred resource so that one GPU can draw using the resource while the other GPU can write to the copy of the resource.

The technique has another advantage related to the placing of NotifyResourceEndAllAccess calls. It is often hard for a game engine to automatically find when it has done using a resource and call NotifyResourceEndAllAccess. A naïve solution would be to place NotifyResourceEndAllAccess calls at the end of the frame before presenting (it is guaranteed that the engine is done using the resource at presentation time). The technique we describe here simplifies determining when to call NotifyResourceEndAllAccess.

The 2-Steps-GPU transfer technique can reduce waits and can simplify the placement of NotifyResourceEndAllAccess but it requires duplicating the resource that needs to be transferred. Developers should only use it if it does actually improve the performance.

Let **O** denotes the **o**riginal resource and **T** the copy of the resource used for **t**ransferring. The technique works as follows:

- Disable transfers for **O**
- Enable transfers for **T**
- In each frame
    1. Do something that does not involve **O**
    2. NotifyResourceBeginAllAccess(**T**)
    3. Copy the content of **T** into **O** (**O** <- **T**)
    4. Update **O**
    5. Copy the content of **O** into **T** (**T** <- **O**)
    6. NotifyResourceEndWrites(**T**)
    7. NotifyResourceEndAllAccess(**T**)
    8. Draw using **O**
    9. Present

In the above sequence, the NotifyResourceBeginAllAccess in line 2 waits for transfers to **T** to finish before reading from it. In line 3 **T** is locally copied to **O** to ensure they have the same content. After updating **O** in line 4 its content is copied in **T** then a NotifyResourceEndWrites is called to transfer **T** to the other GPU (or GPUs). The NotifyResourceEndWrites is directly followed by NotifyResourceEndAllAccess as only **O** is used in drawing commands.

Code sample

The accompanying sample demonstrates the use of the CrossFire API. The sample uses AMD ShadowFX library to perform temporal updates of a shadow map for a point light source. There are 6 shadow maps corresponding to the 6 faces of a cube map centered on the point light source. The 6 shadow maps are stored in a 2D texture atlas or a 2D texture array. Each frame a single shadow map is updated. The shadow map is chosen as cube_face_id = frame_id % 6 where frame_id is a number that increments every frame and cube_face_id is the id of the cube face for which the shadow map is updated in the frame. After updating one shadow map, the scene is lit using all 6 shadow maps for shadowing.

The default AFR compatible mode causes shadow flickering because according to **Heuristic 1** the resource representing the 6 shadow maps is updated then used in the same frame. To resolve the flickering the sample uses the Crossfire API in the following modes:

- Transfer of the whole shadow map resource after updating it.
- Use of the 2-Steps-GPU-Transfer method described in the previous section to transfer shadow maps.
- Limiting the transfer to the region of the atlas or the subresource of the texture array that is updated.
- Update all shadow map cube faces every 6 frames

Annex

Disabling CrossFire programmatically

It is possible to disable CrossFire using a driver extension function. As with the AMD CrossFire API, this extended functionality is accessible via the AMD GPU Services Library (AGS). The accompanying sample provides an example of how to use the function to disable CrossFire. To disable CrossFire, set `crossfireMode to AGS_CROSSFIRE_MODE_DISABLE` in the `config` parameter of `asgInit`:

```
ags_config.crossfireMode = AGS_CROSSFIRE_MODE_DISABLE;
```

When disabling CrossFire programmatically, it is important to call `asgInit` before creating the D3D11 device. The config setting to disable CrossFire will not have any effect if `asgInit` is called after the device creation.

Dynamic Switchable Graphics

Dynamic switchable graphics aims to save power in configurations with a low power iGPU and a dGPU. It submits graphics workload to the iGPU for common desktop applications and only enables the dGPU when an application executes 3D rendering. The developer can ensure that the application always execute on the high performance dGPU by defining the following variable

```
extern "C" { _declspec(dllexport) DWORD AmdPowerXpressRequestHighPerformance = 0x00000001; }
```

The variable `AmdPowerXpressRequestHighPerformance` must be defined in the game process. A common mistake is to define the variable in a launcher process that starts the game as a separate process.