Technical Report

SLI Best Practices

DEVELOPMENT

Abstract

This paper describes techniques that can be used to perform application-side detection of SLI-configured systems, as well as ensure maximum performance scaling under SLI. The accompanying sample introduces NVAPI and demonstrates different methods of handling texture render targets and Direct3D queries.

Peter Young/Bryan Dudash

sdkfeedback@nvidia.com

NVIDIA Corporation
2701 San Tomas Expressway
Santa Clara, CA 95050

October 2, 2007

Table of Contents

Introduction to SLI

Scalable Link Interface (SLI) is a multi-GPU configuration that offers increased rendering performance by dividing workload across multiple GPUs. SLI-certified motherboards are PCI-Express motherboards with multiple x16 lanes. Each lane accepts a PCI-Express GPU, and GPUs are linked via an external bridge connector. Once SLI rendering has been enabled in the control panel, the driver will treat both GPUs as one logical device and divide rendering workload automatically. There are three SLI rendering modes available: Alternate Frame Rendering (AFR), Split Frame Rendering (SFR), and compatibility mode.

In Alternate Frame Rendering, the driver divides workload by alternating GPUs every frame. For example, on a system with two SLI-enabled GPUs, frame 0 would be rendered by GPU 0, frame 1 would be rendered by GPU1, frame 2 would be rendered by GPU0, and so on. This is typically the preferred SLI rendering mode as it divides workload evenly between GPUs and requires little inter-GPU communication, allowing for up to a 1.9x performance increase.

In Split Frame Rendering, the driver will clip the scene into multiple regions and designate rendering workload for these regions to different GPUs. For example, on a system with two SLI-enabled GPUs, the screen may be divided vertically, with GPU0 rendering the top region and GPU1 rendering the bottom region. Rendering is also dynamically load balanced, so the scene division will change whenever the driver determines that one GPU is working more than another. This SLI rendering mode is typically not as desirable as AFR mode, since some rendering work is duplicated and communications overhead is higher.

In compatibility mode, only GPU0 is active and all other GPUs are idle. This offers no performance benefit but ensures compatibility.

In all SLI-rendering modes, local memory is duplicated across all GPUs. This means that on an SLI system with two 256MB video cards, there is still only 256MB of video memory available to applications. Additionally, any change to local memory on one GPU (for example, dynamic texture updates) will often require a data broadcast to other GPUs. This can introduce a performance penalty depending on the size and characteristics of the data.

The rest of this document will describe how to best prepare your application for running AFR mode and achieving maximum performance benefit from it.

Using NVAPI

NVAPI is a lightweight library that you can link to, which allows applications to query a number of important details regarding the user's system configuration. Generally speaking, for SLI you would use NVAPI for querying the number of SLI-configured GPUs on the system.

To use NVAPI, include nvapi.lin in the linked static library list. Then include the header nvapi.h:

```
#include "nvapi.h"
```

Then you initialize the API interface through a provided function.

```
NvAPI_Status   status;
status = NvAPI_Initialize();

if (status != NVAPI_OK)
{
        NvAPI_ShortString     string;
        NvAPI_GetErrorMessage(status, string);
        printf("NVAPI Error: %s¥n", string);
        return false;
}
```

After that, you should verify that there is a compatible NVIDIA driver installed.

```
NV_DISPLAY_DRIVER_VERSION    version = {0};
version.version = NV_DISPLAY_DRIVER_VERSION_VER;

status = NvAPI_GetDisplayDriverVersion(NVAPI_DEFAULT_HANDLE, &version);

if (status != NVAPI_OK)
{
        NvAPI_ShortString    string;
        NvAPI_GetErrorMessage(status, string);
        printf("NVAPI Error: %s¥n", string);
        return false;
}
```

Then you can use provided API functions to query the # of logical and physical GPUs:

```
// enumerate logical gpus
status = NvAPI_EnumLogicalGPUs(logicalGPUs, &logicalGPUCount);
if (status != NVAPI_OK)
{
        NvAPI_ShortString    string;
        NvAPI_GetErrorMessage(status, string);
        printf("NVAPI Error: %s¥n", string);
        return false;
}

// enumerate physical gpus
status = NvAPI_EnumPhysicalGPUs(physicalGPUs, &physicalGPUCount);
if (status != NVAPI_OK)
{
        NvAPI_ShortString    string;
        NvAPI_GetErrorMessage(status, string);
        printf("NVAPI Error: %s¥n", string);
        return false;
}
```

And that's it! If there are less logical GPUs than physical ones you are now ready to take advantage of the performance benefits of SLI.

Getting Maximum Performance Benefit from SLI

SLI performance is inversely proportional to how much data is shared between GPUs. In the optimal case, no data is shared between GPUs, which eliminates synchronization overhead and allows for maximum parallelism. In most cases (especially in AFR mode) this can be achieved without any extra work. However, there are some cases where applications can unintentionally introduce SLI bottlenecks. The rest of this document will detail these situations and cover various solutions.

Direct3D Render to Texture

Render to texture (RTT) in Direct3D is an extremely useful technique that offers a number of advantages; for example, RTT allows the application to employ high dynamic range rendering by rendering to floating point surfaces (formats that are normally unavailable to standard swap chain surfaces). However, there is one significant difference between swap chain surfaces and textures declared as render targets (RTs): The former can be declared as a discardable surface (through the flag D3DSWAPEFFECT_DISCARD), while the latter has no equivalent creation flag. Without this discard hint, the driver must assume that the data integrity of texture RTs must be preserved across frames. Because each GPU in an SLI configuration maintains its own copy of all local resources, any change to a texture RT needs to be broadcasted to all other GPUs on the system. This typically results in a large data copy, resulting in bus traffic and synchronization overhead.

The best way to avoid this performance penalty is to clear color for texture RTs each frame by calling Clear() on the surface that corresponds to the texture. By clearing RTs in this fashion, the driver can assume that data need not be preserved and forego the need to broadcast changes to other GPUs.

One notable exception to this rule is when applications require the results of the previous frame's rendering for the current frame. One common example of this is when the previous frame's rendering is used to approximate scene luminance for tone mapping. In this situation, clearing RTs is obviously not an option. One alternative technique is to allocate a separate RT for each GPU on the system, and only perform render to texture operations on the surface that corresponds to the active GPU for the current frame. For example, if your application uses one texture RT and happens to be running on an SLI system with two GPUs, allocate two RTs instead; on even frames, perform all RTT operations on renderTarget0, and on odd frames perform all RTT operations on renderTarget1. Since no GPU will ever need to access RT data being used by another GPU, there is no data dependency between frames and hence no data copy is necessary. This technique also has the advantage of being slightly faster than the Clear() method, as no clear operation is necessary. However, this comes at the cost of higher memory consumption due to the need for additional RTs.

In essence, the basic strategy is the same as it ever was: Only Clear() surface color when necessary. The only difference is that on SLI systems, you can essentially use Clear() to indirectly mark RTs as discardable and gain performance.

To summarize:

1. If an SLI-configured system is detected, Clear() RT color each frame.
2. If SLI is detected and Clear() is not an option (RT data must be preserved across frames), or if additional performance is desired and the higher memory requirements are not prohibitive, allocate separate RTs for each GPU and cycle through them round-robin style.
3. If SLI is not detected, only clear RT color when necessary.
4. In all rendering modes, only clear framebuffer color when necessary.
5. In all rendering modes, always clear framebuffer Z/depth.

OpenGL Render to Texture

Strategies for OpenGL render to texture performance in SLI generally follow the same principles as those of Direct3D; your goal should be to minimize data broadcasts and inter-frame dependencies wherever possible. That said, here is a list of OpenGL-specific tips:

- Use frame buffer objects (FBO) instead of pbuffers wherever possible. FBOs are faster and more flexible than pbuffers in almost all cases and with pbuffers there is no way to prevent pbuffer data broadcasts across GPUs in an SLI system.
- Avoid using surface-modifying functions such as glTexImage(), glCopyTexImage(), etc.
- When creating your OpenGL context, request a pixel format with the PFD_SWAP_EXCHANGE flag set. This will allow the driver to forego broadcasting the contents of the backbuffer to all GPUs.

Buffering Frames in Direct3D

By default, Direct3D buffers up to three frames' worth of render data. Historically, this was done to grant a performance benefit on single GPU systems. However, because only one frame can be physically displayed at time, multichip configurations such as SLI by definition require this buffering of frames to achieve any parallelism in an alternate frame rendering environment.

The drawback to this is that it introduces latency between issuing a command and seeing its results onscreen (often referred to as "input lag"). The overall goal is to achieve the greatest performance benefit while minimizing perceived input lag.

One of the worst ways to eliminate input lag is to force a flush at the end of every frame. Doing so stalls the CPU, flushes all buffers, then stalls the GPU. In this situation you are not only eliminating any performance benefit from SLI, but you are also incurring a performance penalty on single GPU systems since you are reducing parallelism between the CPU and GPU. This technique should be avoided wherever possible.

A less objectionable solution is to use event queries to control the number of buffered frames. This allows the application to explicitly specify how many frames are buffered. Ideally, you want to buffer at least one frame per GPU on the system. This technique offers a good compromise of minimizing input lag, while also achieving better performance on both SLI-enabled systems and single-GPU systems.

One additional detail worth noting is that while frame throughput is the same on SLI systems as it is on non-SLI systems, frame latency is reduced due to parallelism. For example, if a typical frame takes 30ms to render, the effective latency of those frames is only ~15ms (assuming an SLI system with two GPUs). Thus, increasing the number of frames buffered in SLI does not linearly increase input lag as one might expect. Actual results will depend on how well your application is scaling in SLI.

Conclusion

For more information on SLI performance programming, see the SLI chapter in the NVIDIA GPU Programming Guide, available for download at:

http://developer.nvidia.com/object/gpu_programming_guide.html

NVIDIA Corporation
2701 San Tomas Expressway
Santa Clara, CA 95050
www.nvidia.com