SLI Best Practices

Last updated on 02/15/2011

Abstract

This document describes techniques that can be used to perform application-side detection of SLI-configured systems, as well as to ensure maximum performance scaling in such configurations. The accompanying code sample introduces NVAPI and demonstrates different methods of handling texture render targets and stream out buffers in Direct3D.

Table of Contents

2701 San Tomas Expressway | Santa Clara, CA 95050 | T 408.486.2000 | F 408.486.2200 | www.nvidia.com

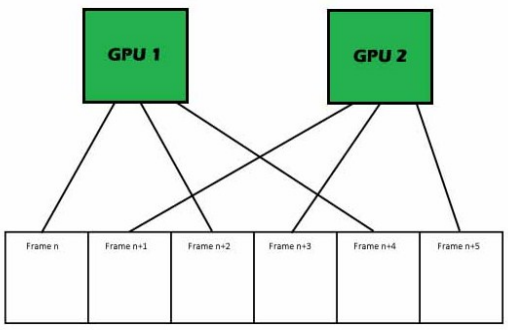
Introduction to SLI

Scalable Link Interface (SLI) is a multi-GPU configuration that offers increased rendering performance by dividing the workload across multiple GPUs. To take advantage of SLI the system must use an SLI-certified motherboard. Such motherboards have multiple PCI-Express x16 slots and are specifically engineered for SLI configurations. To create a multi-GPU SLI configuration NVIDIA GPUs must be attached to at least two of these slots, and then these GPUs must be linked using external SLI bridge connectors. Once the hardware is configured for SLI, and the driver is properly installed for all the GPUs, SLI rendering must be enabled in the NVIDIA control panel. At this point the driver can treat both GPUs as one logical device and divide rendering workload automatically depending on the selected mode. There are five SLI rendering modes available: Alternate Frame Rendering (AFR), Split Frame Rendering (SFR), Boost Performance Hybrid SLI, SLIAA and Compatibility mode.

Alternate Frame Rendering (AFR)

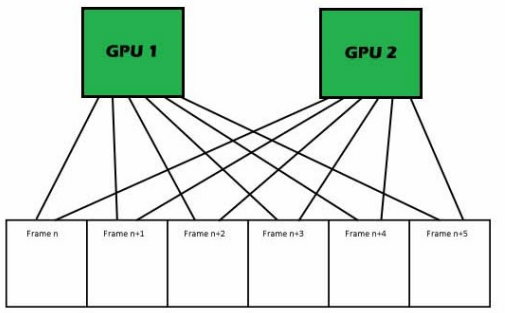
The driver divides workload by alternating GPUs every frame. For example, on a system with two SLI-enabled GPUs, frame 1 would be rendered by GPU 1, frame 2 would be rendered by GPU 2, frame 3 would be rendered by GPU 1, and so on. This is typically the preferred SLI rendering mode as it divides workload evenly between GPUs and requires little inter-GPU communication.

Users can optionally forcefully enable AFR mode for an individual application using the NVIDIA driver control panel. However, this approach may not lead to any scaling due to a variety of pitfalls that are covered in the section on AFR Performance.



Split Frame Rendering (SFR)

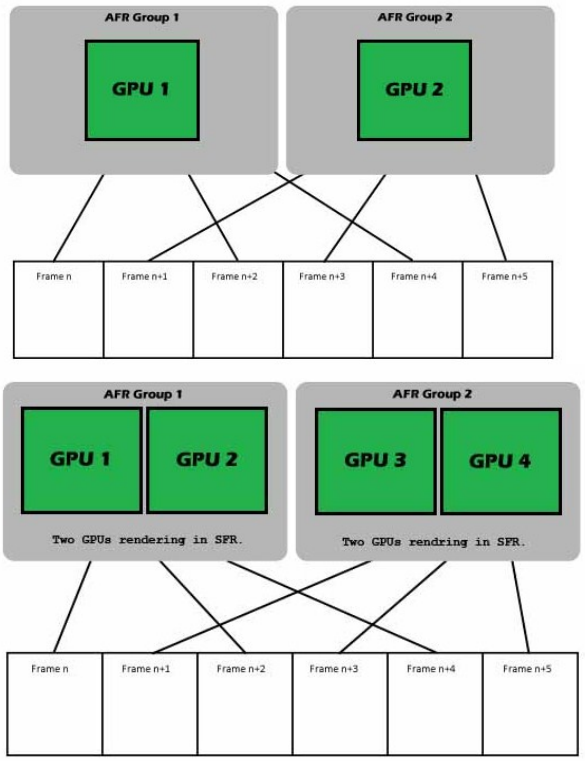
The driver will split the scene workload into multiple regions and assign these regions to different GPUs. For example, on a system with two SLI-enabled GPUs, a render target may be divided vertically, with GPU 1 rendering the left region and GPU 2 rendering the right region. Rendering is also dynamically load balanced, so the 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 of the work is duplicated and communications overhead is higher.



2701 San Tomas Expressway | Santa Clara, CA 95050 | T 408.486.2000 | F 408.486.2200 | www.nvidia.com

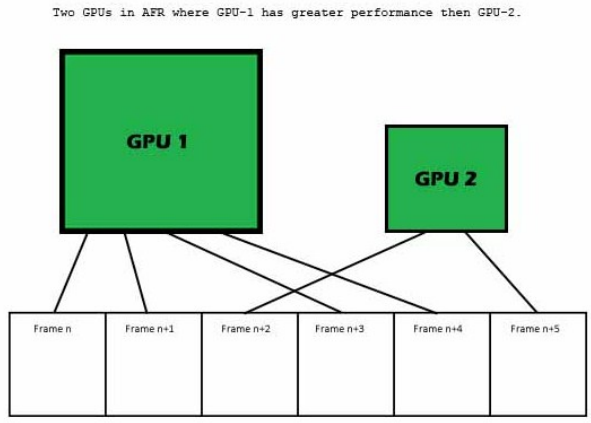
AFR of SFR

The driver may decide to use a hybrid AFR of SFR approach. In this mode the driver creates groups of multiple GPUs that share the work for a given frame in SFR mode and then uses these groups (AFR Group) in Alternate Frame Rendering (AFR) mode. AFR groups can consist of any number of GPUs. While running standard AFR mode we refer to the individual GPUs as AFR groups even though the group only consists of 1 GPU. The figure on the right shows a diagram of AFR Groups for a configuration of 4 GPUs where the driver separates the GPUs into 2 AFR Groups of 2 GPUs each, resulting in the workload of every other frame being handled by 2 GPUs.



Boost Performance Hybrid SLI Rendering

The driver behaves much like it does in AFR mode. However there will most likely be a large performance difference between GPU 1 and GPU 2. Thus, the driver will separate the rendering workload based on the performance capabilities of the two or more GPUs. For example if GPU 1 is about double the performance of GPU 2, the driver is likely to draw multiple frames on GPU 1 for a single frame on GPU 2.



2701 San Tomas Expressway | Santa Clara, CA 95050 | T 408.486.2000 | F 408.486.2200 | www.nvidia.com

SLIAA

SLIAA increases antialiasing performance by splitting the rendering workload for each frame across multiple NVIDIA GPUs. In other words, the visual quality of each rendered frame is increased by the use of more samples in for antialiasing, while the performance level is maintained. The mode relies on combining the final rendered frame generated on multiple GPUs at different sampling locations into a single one. SLIAA can be enabled via the NVIDIA Control Panel by selecting any of the SLIAA modes under the standard 'Antialiasing Settings'. The supported modes when using two GPUs are SLI8x and SLI16x. When using four GPUs one additional mode is available: SLI32x.

Compatibility mode

In this mode only GPU1 is used by the graphics API device (or context) and any other GPU in the system may be idle, used on a separate device (for either a graphics or compute API), or used by other applications. This offers no graphics performance scaling but ensures compatibility. This is the default setting for all applications that don't have an SLI profile (more information in the SLI profile section).

SLI Profiles

Depending on the application and the SLI configuration one or more of the SLI modes listed above may be more appropriate, while some may be undesirable, and some may only be a good choice with appropriate application-specific settings.

One of the most common modes is AFR. By default, when AFR mode is forcefully enabled for a given application using the NVIDA control panel, the driver has to allow all inter-GPU synchronization and communication required to handle inter-frame dependencies and guarantee the correctness of the results. This typically will lead to no SLI performance scaling. The NVIDIA driver supports application-specific SLI profiles that select the best mode for SLI performance scaling, and allow the driver to use heuristics to avoid certain forms of inter-GPU communication or CPU-GPU synchronization. By sending your application to NVIDIA we can create a profile for your application, which will obviate the need for some of the common changes suggested in this document to handle SLI configurations. In some cases, however, driver profiles may not be the most optimal solution, and application changes may be recommended. Once we have created a profile for an application, the profile is added to our next driver release, making it available to the end users as soon as they install the updated driver.

In the absence of an SLI profile, SLIAA is a good alternative to take advantage of multiple GPUs in SLI configurations, even if the application is CPU bound, since SLIAA doesn't require buffering more frames than in a single GPU configuration, or taking care of any major synchronization across GPUs. SLIAA also does not require any SLI-specific work from the application developer. However, when the application uses D3D10 and later it does require that the application supports regular MSAA. When the application uses D3D9, SLIAA can be used in any application that already has the ability to use the driver override for MSAA, which is available in some applications that don't support MSAA directly.

2701 San Tomas Expressway | Santa Clara, CA 95050 | T 408.486.2000 | F 408.486.2200 | www.nvidia.com

Achieving Peak Performance in AFR Mode with Direct3D

The performance of an application running in SLI AFR mode is inversely proportional to the amount data shared between GPUs, as well as the timing of data synchronization events. In the optimal case, no data is shared between GPUs, which eliminates synchronization overhead and allows for maximum parallelism. In many cases this can be achieved without any extra work. However, there are also many cases where applications may prevent SLI AFR scaling due to a variety of pitfalls.

In general terms, there are three common types of pitfalls: CPU boundedness, CPU-GPU synchronization and inter-frame dependencies (which introduce inter-GPU synchronization and communication). Of these pitfalls, CPU boundedness is the one that may be most difficult to solve (ignoring the option of substituting the CPU with a higher performing one) and its solution is largely application dependent. The other two pitfalls however can be addressed either using an SLI profile or following some of the advice in the following sections.

A note on GPU Memory in SLI

In all SLI-rendering modes all the graphics API resources (such as buffers or textures) that would normally be expected to be placed in GPU memory are automatically replicated in the memory of all the GPUs in the SLI configuration. This means that on an SLI system with two 512MB video cards, there is still only 512MB of onboard video memory available to the application.

Any data update performed from the CPU on a resource placed in GPU memory (for example, dynamic texture updates) will usually require the update to be broadcast other GPUs. This can introduce a performance penalty depending on the size and characteristics of the data. Other performance considerations are covered in the section on SLI performance.

Testing the AFR Scaling Potential of your Application

Before spending any time trying to resolve the mentioned pitfalls, developers of Direct3D applications can take advantage of a feature in the NVIDIA driver that allows them to check the maximum possible scaling in a given SLI configuration. When running on a system with multiple GPUs configured in SLI mode, simply running your application executable renamed as "AFR-FriendlyD3D.exe" will make the driver skip any form of inter-GPU synchronizations, as well as common forms of CPU-GPU synchronization. This will lead to the maximum expected scaling in that system, but may introduce rendering artifacts (since the driver is no longer performing all the operations required to guarantee correctness in AFR mode). Lack of scaling in this case typically indicates the application is CPU bound.

In some cases, this approach changes the behavior of the application such that it may not reflect its real AFR scaling potential. In those cases it is possible that either adding an SLI profile, or following the advice in the following sections, or a combination of both, may still result in good performance scaling.

2701 San Tomas Expressway | Santa Clara, CA 95050 | T 408.486.2000 | F 408.486.2200 | www.nvidia.com

Avoiding Common Causes of CPU-GPU Synchronization

By default, Direct3D allows buffering up to three frames' worth of work before Direct3D calls will start stalling. Historically, this was done to grant a performance benefit on single GPU systems by allowing overlap of CPU and GPU work. SLI configurations require this buffering of frames to achieve performance scaling in AFR mode.

The drawback of this buffering is that it introduces additional latency between the user's input and its results being visible on screen (often referred to as "input lag"). The overall goal of SLI is to achieve the greatest performance benefit while minimizing perceived input lag.

Some applications try to minimize the amount of frame buffering to reduce input latency. A common approach to do this is the use of Direct3D Event queries: the application inserts an Event query at the end of every frame and then, one frame later for example, stalls by checking the state of this query on the CPU until the Event has been reached by the GPU. This allows the application to explicitly control how many frames are buffered, but also prevents AFR scaling. This approach falls under the category of CPU-GPU synchronization pitfalls mentioned above. Similarly, some applications often cause undesired CPU-GPU synchronization when using Occlusion Queries.

Event queries, Occlusion queries, and any other type of asynchronous Direct3D query may all lead to poor SLI performance scaling if not used carefully. Applications using queries should avoid stalling the CPU too early after the query is issued (such stalls usually show up as loops that call GetData on a query object until it returns a valid result). Doing so can lead to idle GPU times if enough GPU work is not scheduled to keep the GPU busy between the point where the query completes and the next buffer of GPU work is submitted to the GPU.

When using Direct3D queries, a common practice to achieve good performance scaling in AFR mode is to avoid the use of stalling GetData loops on Queries issued less than N frames before (where N is the number of AFR Groups in the configuration). There are different ways to achieve this, but typically it involves tracking somehow how many frames earlier a Query was issued and guaranteeing that a stalling GetData loop will only be allowed after N frames have gone by. This technique offers a good compromise, allowing input lag minimization on single-GPU systems, and performance scaling on SLI-enabled systems in AFR mode. The only potential drawback to be aware of is that when applied to some algorithms using Occlusion Queries it may introduce some popping artifacts because of the higher latency to obtain the visibility information.

One additional detail worth noting is that while input latency is the same on SLI AFR configurations as it is on single GPU configurations (each frame will take as long to be complete), inter-frame latency is reduced due to parallelism, so the application appears more responsive. For example, if a typical frame takes 30 ms to render, in a 2-GPU AFR configuration with perfect scaling the latency between those frames is only 15 ms. Thus, increasing the number of frames buffered in SLI does not increase input lag.

2701 San Tomas Expressway | Santa Clara, CA 95050 | T 408.486.2000 | F 408.486.2200 | www.nvidia.com

Avoiding Common Causes of Inter-frame Dependencies

Render to Texture

Rendering to textures other then the SwapChain's back buffer (assuming double buffering) is a common practice and is useful in many rendering techniques. However, there is a difference between a texture render target and the frame buffer back buffer. When a SwapChain is created a discard flag can be set (D3DSWAPEFFECT_DISCARD in D3D9 or DXGI_SWAP_EFFECT_DISCARD in D3D10 and later). This flag tells the driver that the data written to the back buffer does not need to be preserved across frames, so once Present is called the driver can start a new frame on a separate AFR Group without having to transfer the back buffer data from the GPU(s) in one AFR Group to the others.

Render targets (RTs), on the other hand, don't have these creation flags. As a result, the driver must preserve their contents across frames because it doesn't know if the application will need the data written to that render target in future frames. Some rendering techniques rely on the data being preserved, but many don't. The driver's only option, however, is to copy the RT contents to all the other GPUs to allow them access to the latest state of the RT. This typically leads to large data transfers between GPUs, which aren't optimal, as well as synchronization overhead. For example: GPU2 will stall waiting for GPU1 to update a shared RT. Depending on the timing and size of the data dependencies, as well as on the available PCI Express bandwidth, the amount of SLI performance scaling will vary.

In the absence of an SLI profile that prevents the inter-GPU transfer of render targets, there is a way in which the application can to avoid this performance penalty: clearing the render target each frame before using it by calling the appropriate Clear() function. This applies to all render targets, whether used as color or depth-stencil buffers. If the application clears render targets in this fashion, the driver will assume that data does not need to be preserved and forego the need to copy the render targets between GPUs.

It is important to keep in mind that Clear calls should continue being used where important for performance either on single GPU or in SLI configurations, independently of the existence of a SLI profile for your application. Clears are important for good performance on depth buffers and multisampled surfaces (either color or depth), and should always be preferred over full-screen quads that fill the surface with a constant value. At the same time, while clearing such surfaces once per frame before they are used is a good practice, Clears should be used judiciously, such that redundant Clear calls are avoided.

2701 San Tomas Expressway | Santa Clara, CA 95050 | T 408.486.2000 | F 408.486.2200 | www.nvidia.com

D3D9 Example:

```
pd3d9Device->SetRenderTarget(0, gppSurface);
if (bClearRT)
{
  pd3d9Device->Clear(0L, NULL, D3DCLEAR_TARGET,0x00000000, 1.0f, 0L);
}
```

D3D10 Example:

```
pDevice->OMSetRenderTargets(NumRenderTargets, ppRenderTargetViews,
                            pDepthStencilView);
if (bClearRT)
{
  for (int i = 0; i < NumRenderTargets; i++)
  {
    FLOAT clearColor[4] = { 0.0f };
    pd3d10Device->ClearRenderTargetView(pRenderTargetViews[i],
                                        clearColor);
    pd3d10Device->ClearDepthStencilView(pDepthStencilView);
  }
}
```

Stream Out Buffers in D3D10 and Later

Much like render targets, buffers used as the destination of stream output operations in D3D10 and later have the same problem. The driver doesn't know if the data contained in the buffer will be used in future frames, therefore causing the same copy and synchronization performance penalties that non-cleared render targets have. There is no API mechanism to clear a Buffer, and therefore the only way currently available to avoid a performance penalty in AFR mode when using stream output is to get an SLI profile for your application added to the driver.

Unordered Access Views in D3D11

D3D11 introduced the concept of Unordered Access Views that can be bound to the pipeline for random read/write accesses. Just like render targets and buffers used as the destination of stream output operations, any potential modification performed on UAVs will introduce inter-frame dependencies that, in the absence of a frame profile indicating otherwise, the driver will handle by introducing appropriate inter-GPU synchronization operations. Two D3D11 calls (ID3D11DeviceContext::ClearUnorderedAccessViewUint and ID3D11DeviceContext::ClearUnorderedAccessViewFloat) can be used to clear these buffers before they are used each frame, thus removing the inter-frame dependency and avoiding the overhead of synchronization between GPUs in different AFR groups.

2701 San Tomas Expressway | Santa Clara, CA 95050 | T 408.486.2000 | F 408.486.2200 | www.nvidia.com

Using NVAPI for SLI Detection

The advice in the previous sections involves some changes to your application that may only be required when running in SLI configurations. In addition some of the changes may require knowing how many AFR Groups exist in the currently running configuration. This section explains how you can use NVAPI in your application to detect whether it's running in an SLI AFR configuration and how many AFR Groups exist in that configuration.

NVAPI is a lightweight library included as part of the end user's driver installation, which allows applications to query a number of important details regarding the user's system configuration. Generally speaking, there are at least two tasks you would use NVAPI for when developing for SLI: Querying the number of SLI-configured GPUs on the system, and querying the current AFR index you're rendering to per-frame.

To use NVAPI, first you need to initialize it:

```
#pragma comment (lib, "nvapi.lib")
// File containing nvapi interface
#include "nvapi.h"

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;
}
```

It's a good practice to check the driver ID string to ensure NVAPI is working properly with the system configuration:

```
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;
}
```


2701 San Tomas Expressway | Santa Clara, CA 95050 | T 408.486.2000 | F 408.486.2200 | www.nvidia.com

Once the NVAPI library is initialized and the driver is verified you can use **NvAPI_D3D_GetCurrentSLIState** to query the current SLI state.

The following is the reference documentation for this call from nvapi.h:

```
typedef struct
{
   NvU32 version;              // Structure version (must be set to NV_GET_CURRENT_SLI_STATE_VER)
   NvU32 maxNumAFRGroups;      // [OUT] The maximum possible value of numAFRGroups
   NvU32 numAFRGroups;         // [OUT] The number of AFR groups enabled in the system
   NvU32 currentAFRIndex;      // [OUT] The AFR group index for the frame currently being rendered
   NvU32 nextFrameAFRIndex;    // [OUT] What the AFR group index will be for the next frame
                               //       (i.e. after calling Present)
   NvU32 previousFrameAFRIndex;// [OUT] The AFR group index that was used for the previous frame
                               //       (~0 if this is the first frame)
   NvU32 bIsCurAFRGroupNew;    // [OUT] boolean: Is this frame the first time running on the current
                               //       AFR group
} NV_GET_CURRENT_SLI_STATE;

#define NV_GET_CURRENT_SLI_STATE_VER  MAKE_NVAPI_VERSION(NV_GET_CURRENT_SLI_STATE,1)
```

Remember that an AFR Group can consist of either one GPU working on a given frame or multiple GPUs working on the same frame in SFR mode. In addition, note there are two different variables in the `NV_GET_CURRENT_SLI_STATE` struct that provide a number of AFR Groups: `maxNumAFRGroups` and `numAFRGroups`. This is because it is possible for the number of active AFR Groups to change dynamically while the application is running in some circumstances, such as when an SLI-enabled laptop is unplugged, switching to running on batteries. For this reason, we recommended that the application calls **NvAPI_D3D_GetCurrentSLIState** once per frame to detect potential changes to the number of active AFR Groups. The recommended usage pattern, per the example code below, is to check the value of `maxNumAFRGroups` once at application start time, and then check `numAFRGroups` and `bIsCurAFRGroupNew` at the beginning of every frame.

```
Int  gNumSLIGPUs = 0;
bool bSLIEnabled = false;

NV_GET_CURRENT_SLI_STATE sliState;
sliState.version = NV_GET_CURRENT_SLI_STATE_VER;

status = NvAPI_D3D_GetCurrentSLIState( pd3dDevice, &sliState);
if ( status != NVAPI_OK ) {
      /*
       * Error code here
       * Requesting a NVIDIA driver update
       *is the prefered action in this case
       */
}
else {
       gMaxNumAFRGroups = sliState.maxNumAFRGroups;
       gNumAFRGroups = sliState.numAFRGroups;
       bSLIEnabled = (gMaxNumAFRGroups > 1);
}
```


2701 San Tomas Expressway | Santa Clara, CA 95050 | T 408.486.2000 | F 408.486.2200 | www.nvidia.com

```
/*
 * While in Render Loop
 * One cycle iteration stands for one frame
 */
while(1)
{
      status = NvAPI_D3D_GetCurrentSLIState(pDevice, &sliState);
      if (status !== NVAPI_OK)
      {
            if (sliState.bIsCurAFRGroupNew)
            {
             /*
              * Rendering on a GPU AFR group which was recently
              * added to a list of active AFR groups or has never drawn
              * a frame yet.
              * Do any initilazion work here.
              * Example: sometimes it's usefuly to hold timers for
              * the individual AFR groups so this is were you would
              * intialize the timers.
              */
              GPUtimers[sliState.currentAFRIndex] =
                GPUtimers[sliState.previousFrameAFRIndex];

             /* Update our copy of numAFRGroups.
              * This may affect some of the logic, for example
              * to handle Asyncrhonous D3D queries
              */
              gNumAFRGroups = sliState.numAFRGroups;
            }


            /*
             * Do render updates based on AFR Index
             * Render Render Targets that are specific to only
             * this GPU.
             */
            Update_GPU(sliState.currentAFRIndex);

            GPUtimers[sliState.currentAFRIndex] = globalTimer;
      }
}
```


2701 San Tomas Expressway | Santa Clara, CA 95050 | T 408.486.2000 | F 408.486.2200 | www.nvidia.com

SLI Check List

1. Check the maximum potential scaling by running your application in an SLI configuration after renaming its executable to AFR-FriendlyD3D.exe. If necessary, resolve any CPU bottleneck before continuing.
2. If you are using Queries, make sure to allow buffering of at least N frames' worth of work before stalling to get their result (where N is the number of AFR groups in the configuration).
3. Use resource Clears were appropriate, depending on the availability of an SLI profile for your application.
4. For either 2 or 3, if your application requires an SLI-specific change, use the latest version of NVAPI from developer.nvidia.com to properly detect the current SLI configuration, including the number of AFR Groups.
5. Send NVIDIA a build of your application to get an SLI profile created for it. Alternatively, use the SLI-specific advice to eliminate inter-frame dependencies where possible (Clears).

Additional Resources

SLI Zone (http://www.slizone.com) provides a lot of NVIDIA SLI-specific information for end users.

2701 San Tomas Expressway | Santa Clara, CA 95050 | T 408.486.2000 | F 408.486.2200 | www.nvidia.com