CUDA DEVELOPER GUIDE FOR NVIDIA OPTIMUS PLATFORMS

DG-06715-001_v6.5 | August 2014

Reference Guide

TABLE OF CONTENTS

Chapter 1. Introduction to Optimus.....1
Chapter 2. CUDA Applications and Optimus.....2
Chapter 3. Querying for a CUDA Device.....3
3.1. Applications without Graphics Interoperability.....3
3.2. Applications with Graphics Interoperability.....5
3.3. CUDA Support with DirecX Interoperability.....6
3.4. CUDA Support with OpenGL Interoperability.....7
3.5. Control Panel Settings and Driver Updates.....7

Chapter 1. INTRODUCTION TO OPTIMUS

NVIDIA® Optimus™ is a revolutionary technology that delivers great battery life and great performance, in a way that simply works. It automatically and instantaneously uses the best tool for the job – the high performance NVIDIA GPU for GPU-Compute applications, video, and 3D games; and low power integrated graphics for applications like Office, Web surfing, or email.

The result is long lasting battery life without sacrificing great graphics performance, delivering an experience that is fully automatic and behind the scenes.

When the GPU can provide an increase in performance, functionality, or quality over the *IGP* for an application, the NVIDIA driver will enable the GPU. When the user launches an application, the NVIDIA driver will recognize whether the application being run can benefit from using the GPU. If the application can benefit from running on the GPU, the GPU is powered up from an idle state and is given all rendering calls.

Using NVIDIA's Optimus technology, when the discrete GPU is handling all the rendering duties, the final image output to the display is still handled by the Intel integrated graphics processor (IGP). In effect, the IGP is only being used as a simple display controller, resulting in a seamless, flicker-free experience with no need to reboot.

When the user closes all applications that benefit from the GPU, the discrete GPU is powered off and the Intel IGP handles both rendering and display calls to conserve power and provide the highest possible battery life.

The beauty of Optimus is that it leverages standard industry protocols and APIs to work. From relying on standard Microsoft APIs when communicating with the Intel IGP driver, to utilizing the PCI-Express bus to transfer the GPU's output to the Intel IGP, there are no proprietary hoops to jump through NVIDIA.

This document provides guidance to CUDA developers and explains how NVIDIA CUDA APIs can be used to query for GPU capabilities in Optimus systems. It is strongly recommended to follow these guidelines to ensure CUDA applications are compatible with all notebooks featuring Optimus.

Chapter 2.
CUDA APPLICATIONS AND OPTIMUS

Optimus systems all have an Intel *IGP* and an NVIDIA GPU. Display heads may be electrically connected to the IGP or the GPU. When a display is connected to a GPU head, all rendering and compute on that display happens on the NVIDIA GPU just like it would on a typical discrete system. When the display is connected to an IGP head, the NVIDIA driver decides if an application on that display should be rendered on the GPU or the IGP. If the driver decides to run the application on the NVIDIA GPU, the final rendered frames are copied to the IGP's display pipeline for scanout. Please consult the Optimus white paper for more details on this behavior: http://www.nvidia.com/object/optimus_technology.html.

CUDA developers should understand this scheme because it affects how applications should query for GPU capabilities. For example, a CUDA application launched on the *LVDS* panel of an Optimus notebook (which is an IGP-connected display), would see that the primary display device is the Intel's graphic adapter – a chip not capable of running CUDA. In this case, it is important for the application to detect the existence of a second device in the system – the NVIDIA GPU – and then create a CUDA context on this CUDA-capable device even when it is not the display device.

For applications that require use of Direct3D/CUDA or OpenGL/CUDA interop, there are restrictions that developers need to be aware of when creating a *Direct3D* or *OpenGL* context that will interoperate with CUDA. CUDA Support with DirecX Interoperability and CUDA Support with OpenGL Interoperability in this guide discuss this topic in more detail.

Chapter 3. QUERYING FOR A CUDA DEVICE

Depending on the application, there are different ways to query for a CUDA device.

3.1. Applications without Graphics Interoperability

For CUDA applications, finding the best CUDA capable device is done through the CUDA API. The CUDA API functions **`cudaGetDeviceProperties`** (CUDA runtime API), and **`cuDeviceComputeCapability`** (CUDA Driver API) are used. Refer to the CUDA Sample **`deviceQuery`** or **`deviceQueryDrv`** for more details.

The next two code samples illustrate the best method of choosing a CUDA capable device with the best performance.

```
// CUDA Runtime API Version
inline int cutGetMaxGflopsDeviceId()
{
    int current_device   = 0, sm_per_multiproc = 0;
    int max_compute_perf = 0, max_perf_device  = 0;
    int device_count     = 0, best_SM_arch     = 0;
    int arch_cores_sm[3] = { 1, 8, 32, 192 };
    cudaDeviceProp deviceProp;

    cudaGetDeviceCount( &device_count );

    // Find the best major SM Architecture GPU device
    while ( current_device < device_count ) {
    cudaGetDeviceProperties( &deviceProp, current_device );
        if (deviceProp.major > 0 && deviceProp.major < 9999) {
            best_SM_arch = max(best_SM_arch, deviceProp.major);
        }
        current_device++;
    }

    // Find the best CUDA capable GPU device
    current_device = 0;
    while( current_device < device_count ) {
        cudaGetDeviceProperties( &deviceProp, current_device );
        if (deviceProp.major == 9999 && deviceProp.minor == 9999) {
            sm_per_multiproc = 1;
        } else if (deviceProp.major <= 3) {
            sm_per_multiproc = arch_cores_sm[deviceProp.major];
        } else { // Device has SM major > 3
            sm_per_multiproc = arch_cores_sm[3];
        }

        int compute_perf = deviceProp.multiProcessorCount *
                           sm_per_multiproc * deviceProp.clockRate;

        if( compute_perf > max_compute_perf ) {
            // If we find GPU of SM major > 3, search only these
            if ( best_SM_arch > 3 ) {
                // If device==best_SM_arch, choose this, or else pass
                if (deviceProp.major == best_SM_arch) {
                    max_compute_perf  = compute_perf;
                    max_perf_device   = current_device;
                }
            } else {
                max_compute_perf  = compute_perf;
                max_perf_device   = current_device;
            }
        }
        ++current_device;
    }

    cudaGetDeviceProperties(&deviceProp, max_compute_perf_device);
    printf("\nDevice %d: \"%s\"\n", max__perf_device,
                                    deviceProp.name);
    printf("Compute Capability   : %d.%d\n",
    deviceProp.major, deviceProp.minor);
    return max_perf_device;
}

// CUDA Driver API Version
inline int cutilDrvGetMaxGflopsDeviceId()
{
    CUdevice current_device = 0, max_perf_device  = 0;
    int device_count        = 0, sm_per_multiproc = 0;
    int max_compute_perf    = 0, best_SM_arch     = 0;
    int major = 0, minor    = 0, multiProcessorCount, clockRate;
    int arch_cores_sm[3] = { 1, 8, 32, 192 };

    cuInit(0);
    cuDeviceGetCount(&device_count);
    // Find the best major SM Architecture GPU device
    while ( current_device < device_count ) {
```

3.2. Applications with Graphics Interoperability

For CUDA applications that use the CUDA interop capability with *Direct3D* or *OpenGL*, developers should be aware of the restrictions and requirements to ensure compatibility with the Optimus platform. For CUDA applications that meet these descriptions:

1. Application requires CUDA interop capability with either Direct3D or OpenGL.
2. Application is not directly linked against **`cuda.lib`** or **`cudart.lib`** or **`LoadLibrary`** to dynamically load the **`nvcuda.dll`** or **`cudart*.dll`** and uses **`GetProcAddress`** to retrieve function addresses from **`nvcuda.dll`** or **`cudart*.dll`**.

A Direct3D or OpenGL context has to be created before the CUDA context. The Direct3D or OpenGL context needs to pass this into the CUDA. See the sample calls below in red below. Your application will need to create the graphics context first. The sample code below does not illustrate this.

Refer to the CUDA Sample **`simpleD3D9`** and **`simpleD3D9Texture`** for details.

```
// CUDA/Direct3D9 interop
// You will need to create the D3D9 context first
IDirect3DDevice9 * g_pD3D9Device; // Initialize D3D9 rendering device
// After creation, bind your D3D9 context to CUDA
cudaD3D9SetDirect3DDevice(g_pD3D9Device);
```

Refer to the CUDA Sample **`simpleD3D10`** and **`simpleD3D10Texture`** for details.

```
// CUDA/Direct3D10 interop
// You will need to create a D3D10 context first
ID3D10Device * g_pD3D10Device; // Initialize D3D10 rendering device
// After creation, bind your D3D10 context to CUDA
cudaD3D10SetDirect3DDevice(g_pD3D10Device);
```

Refer to the CUDA Sample **`simpleD3D11Texture`** for details.

```
// CUDA/Direct3D11 interop
// You will need to first create the D3D11 context first
ID3D11Device * g_pD3D11Device; // Initialize D3D11 rendering device
// After creation, bind your D3D11 context to CUDA
cudaD3D11SetDirect3DDevice(g_pD3D11Device);
```

Refer to the CUDA Samples **`simpleGL`** and **`postProcessGL`** for details.

```
// For CUDA/OpenGL interop
// You will need to create the OpenGL Context first
// After creation, bind your D3D11 context to CUDA
cudaGLSetGLDevice(deviceID);
```

On an Optimus platform, this type of CUDA application will not work properly. If a Direct3D or OpenGL graphics context is created before any CUDA API calls are used or initialized, the Graphics context may be created on the Intel *IGP*. The problem here is that the Intel IGP does not allow Graphics interoperability with CUDA running on the NVIDIA GPU. If the Graphics context is created on the NVIDIA GPU, then everything will work.

The solution is to create an application profile in the NVIDIA Control Panel. With an application profile, the Direct3D or OpenGL context will always be created on the NVIDIA GPU when the application gets launched. These application profiles can be

created manually through the NVIDIA control Panel (see Control Panel Settings and Driver UpdatesSection 5 for details). Contact NVIDIA support to have this application profile added to the drivers, so future NVIDIA driver releases and updates will include it.

3.3. CUDA Support with DirecX Interoperability

The following steps with code samples illustrate what is necessary to initialize your CUDA application to interoperate with *Direct3D9*.

An application profile may also be needed.

1. Create a Direct3D9 Context:

```
// Create the D3D object
if ((g_pD3D = Direct3DCreate9(D3D_SDK_VERSION))==NULL )
    return E_FAIL;
```

2. Find the CUDA Device that is also a Direct3D device

```
// Find the first CUDA capable device, may also want to check
// number of cores with cudaGetDeviceProperties for the best
// CUDA capable GPU (see previous function for details)
for(g_iAdapter = 0;
    g_iAdapter < g_pD3D->GetAdapterCount();
    g_iAdapter++)
{
    D3DCAPS9 caps;
    if (FAILED(g_pD3D->GetDeviceCaps(g_iAdapter,
        D3DDEVTYPE_HAL, &caps)))
        // Adapter doesn't support Direct3D
        continue;

        D3DADAPTER_IDENTIFIER9 ident;
        int device;
        g_pD3D->GetAdapterIdentifier(g_iAdapter,
                                     0, &ident);
        cudaD3D9GetDevice(&device, ident.DeviceName);
        if (cudaSuccess == cudaGetLastError() )
            break;
}
```

3. Create the Direct3D device

```
// Create the D3DDevice
if (FAILED( g_pD3D->CreateDevice( g_iAdapter, D3DDEVTYPE_HAL,
                                  hWnd, D3DCREATE_HARDWARE_VERTEXPROCESSING,
                                  &g_d3dpp, &g_pD3DDevice ) ) )
    return E_FAIL;
```

4. Bind the CUDA Context to the Direct3D device:

```
// Now we need to bind a CUDA context to the DX9 device
cudaD3D9SetDirect3DDevice(g_pD3DDevice);
```

3.4. CUDA Support with OpenGL Interoperability

The following steps with code samples illustrate what is needed to initialize your CUDA application to interoperate with *OpenGL*.

An application profile may also be needed.

1. Create an OpenGL Context and OpenGL window

```
// Create GL context
glutInit(&argc, argv);
glutInitDisplayMode(GLUT_RGBA | GLUT_ALPHA |
                    GLUT_DOUBLE | GLUT_DEPTH);
glutInitWindowSize(window_width, window_height);
glutCreateWindow("OpenGL Application");

// default initialization of the back buffer
glClearColor(0.5, 0.5, 0.5, 1.0);
```

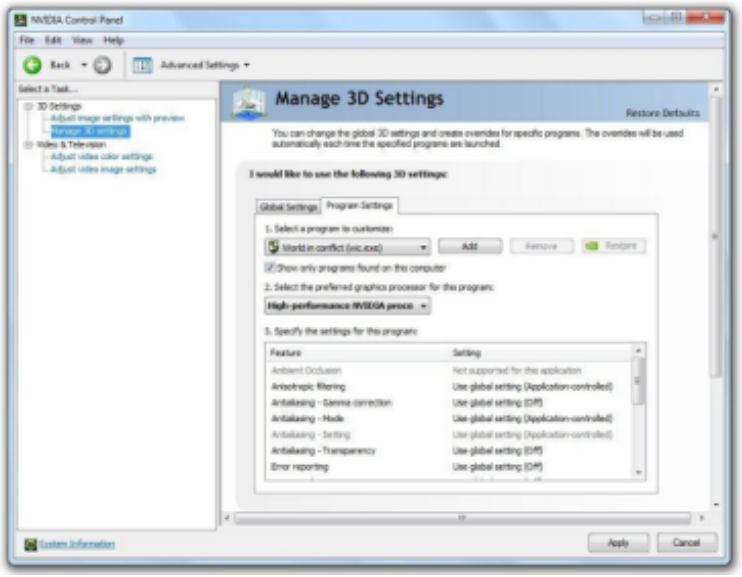
2. Create the CUDA Context and bind it to the OpenGL context

```
// Initialize CUDA context (ontop of the GL context)
int dev, deviceCount;
cudaGetDeviceCount(&deviceCount);
cudaDeviceProp deviceProp;
for (int i=0; i<deviceCount; i++) {
    cudaGetDeviceProperties(&deviceProp, dev));
}
cudaGLSetGLDevice(dev);
```

3.5. Control Panel Settings and Driver Updates

This section is for developers who create custom application-specific profiles. It describes how end users can be sure that their CUDA applications run on Optimus and how they can receive the latest updates on drivers and application profiles.

- Profile updates are frequently sent to end user systems, similar to how virus definitions work. Systems are automatically updated with new profiles in the background with no user intervention required. Contact NVIDIA developer support to create the appropriate driver application profiles. This will ensure that your CUDA application is compatible on Optimus and included in these automatic updates.
- End users can create their own application profiles in the NVIDIA Control panel and set when switch and not to switch to the NVIDIA GPU per application. The screenshot below shows you where to find these application profiles.



- NVIDIA regularly updates the graphics drivers through the NVIDIA Verde Driver Program. End users will be able to download drivers which include application-specific Optimus profiles for NVIDIA-powered notebooks from: http://www.nvidia.com/object/notebook_drivers.html

Notice

ALL NVIDIA DESIGN SPECIFICATIONS, REFERENCE BOARDS, FILES, DRAWINGS, DIAGNOSTICS, LISTS, AND OTHER DOCUMENTS (TOGETHER AND SEPARATELY, "MATERIALS") ARE BEING PROVIDED "AS IS." NVIDIA MAKES NO WARRANTIES, EXPRESSED, IMPLIED, STATUTORY, OR OTHERWISE WITH RESPECT TO THE MATERIALS, AND EXPRESSLY DISCLAIMS ALL IMPLIED WARRANTIES OF NONINFRINGEMENT, MERCHANTABILITY, AND FITNESS FOR A PARTICULAR PURPOSE.

Information furnished is believed to be accurate and reliable. However, NVIDIA Corporation assumes no responsibility for the consequences of use of such information or for any infringement of patents or other rights of third parties that may result from its use. No license is granted by implication of otherwise under any patent rights of NVIDIA Corporation. Specifications mentioned in this publication are subject to change without notice. This publication supersedes and replaces all other information previously supplied. NVIDIA Corporation products are not authorized as critical components in life support devices or systems without express written approval of NVIDIA Corporation.

Trademarks

NVIDIA and the NVIDIA logo are trademarks or registered trademarks of NVIDIA Corporation in the U.S. and other countries. Other company and product names may be trademarks of the respective companies with which they are associated.

Copyright

© 2014 NVIDIA Corporation. All rights reserved.