US 20110067038A1

(19) United States

(12) Patent Application Publication

Troccoli et al.

(10) Pub. No.: US 2011/0067038 A1

(43) Pub. Date: Mar. 17, 2011

(54) CO-PROCESSING TECHNIQUES ON HETEROGENEOUS GPUS HAVING DIFFERENT DEVICE DRIVER INTERFACES

(75) Inventors: **Alejandro Troccoli**, Santa Clara, CA (US); **Franck Diard**, Mountain View, CA (US)

(73) Assignee: **NVIDIA CORPORATION**, Santa Clara, CA (US)

(21) Appl. No.: **12/649,864**

(22) Filed: **Dec. 30, 2009**

Related U.S. Application Data

(60) Provisional application No. 61/243,155, filed on Sep. 16, 2009, provisional application No. 61/243,164, filed on Sep. 17, 2009.

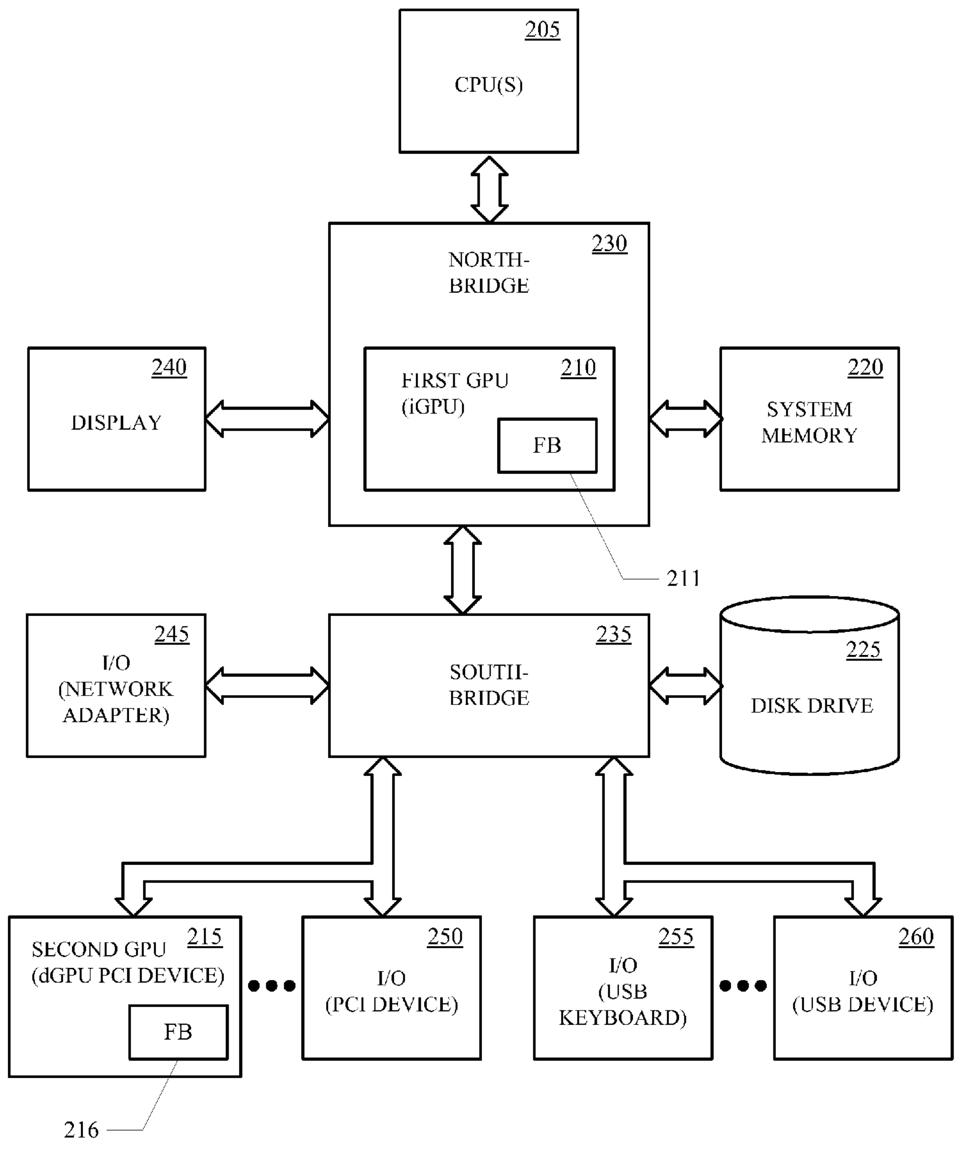
Publication Classification

(51) **Int. Cl.**
G06F 9/44 (2006.01)

(52) **U.S. Cl.** **719/327**

(57) **ABSTRACT**

The graphics co-processing technique includes loading a shim layer library. The shim layer library loads and initializes a device driver interface of a first class on the primary adapter and a device driver interface of a second class on an unattached adapter. The shim layer also translates calls between the first device driver interface of the first class on the primary adapter and the second device driver interface of the second class on the unattached adapter.



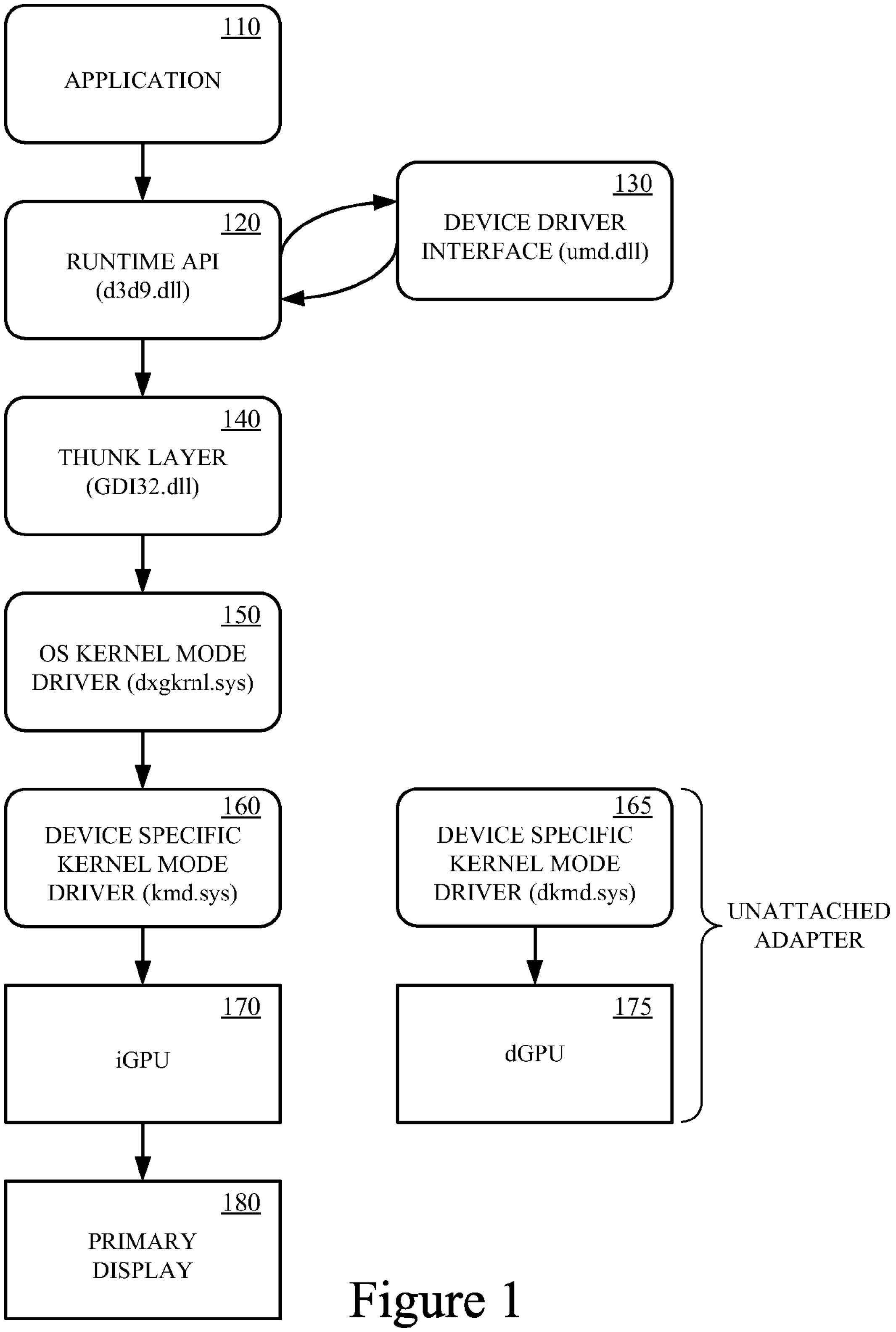

110
APPLICATION
120
RUNTIME API (d3d9.dll)
130
DEVICE DRIVER INTERFACE (umd.dll)
140
THUNK LAYER (GDI32.dll)
150
OS KERNEL MODE DRIVER (dxgkrnl.sys)
160
DEVICE SPECIFIC KERNEL MODE DRIVER (kmd.sys)
165
DEVICE SPECIFIC KERNEL MODE DRIVER (dkmd.sys)
UNATTACHED ADAPTER
170
iGPU
175
dGPU
180
PRIMARY DISPLAY

Figure 1

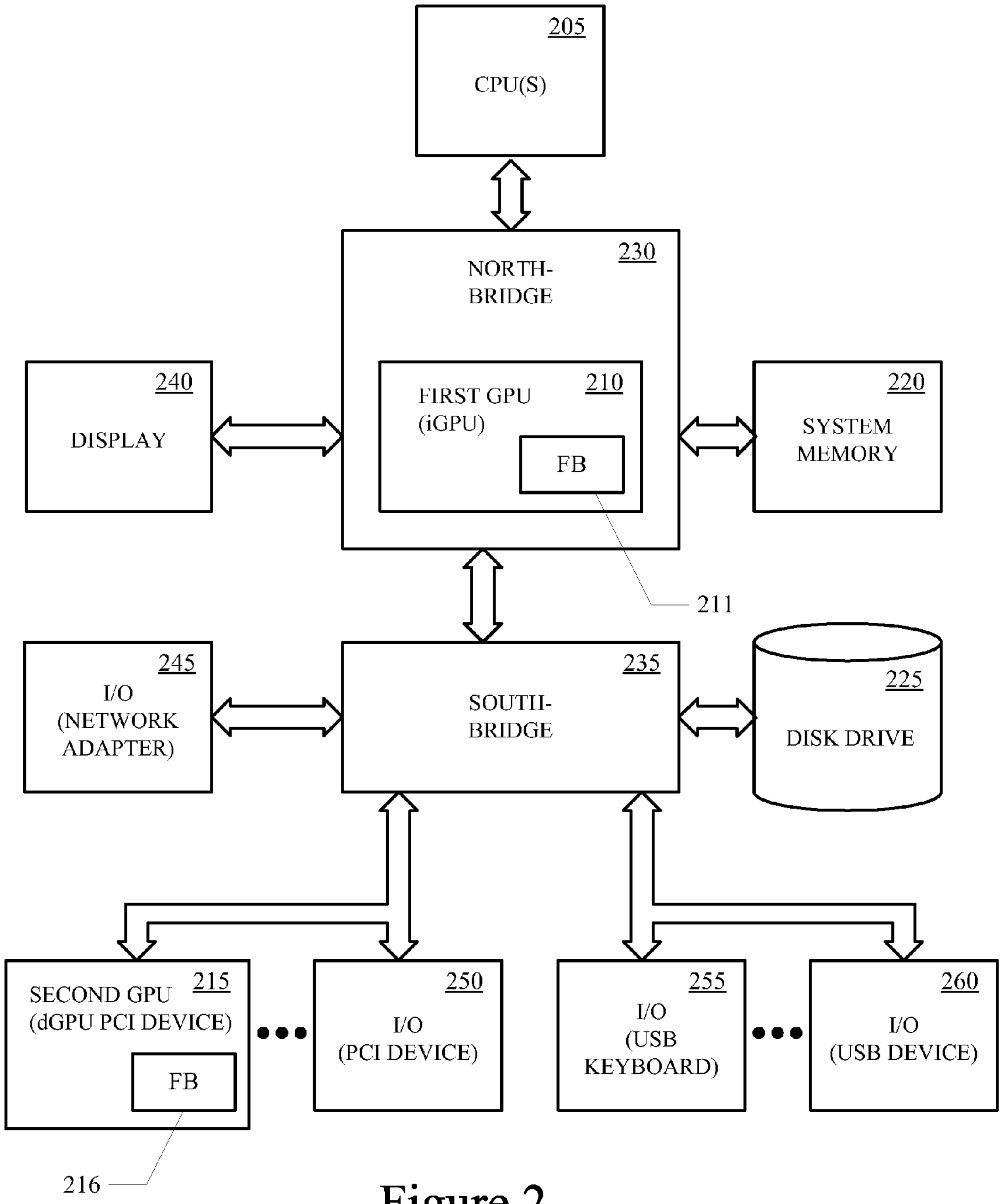

205
CPU(S)
230
NORTH-BRIDGE
210
FIRST GPU (iGPU)
FB
211
240
DISPLAY
220
SYSTEM MEMORY
245
I/O (NETWORK ADAPTER)
235
SOUTH-BRIDGE
225
DISK DRIVE
215
SECOND GPU (dGPU PCI DEVICE)
FB
216
250
I/O (PCI DEVICE)
255
I/O (USB KEYBOARD)
260
I/O (USB DEVICE)

Figure 2

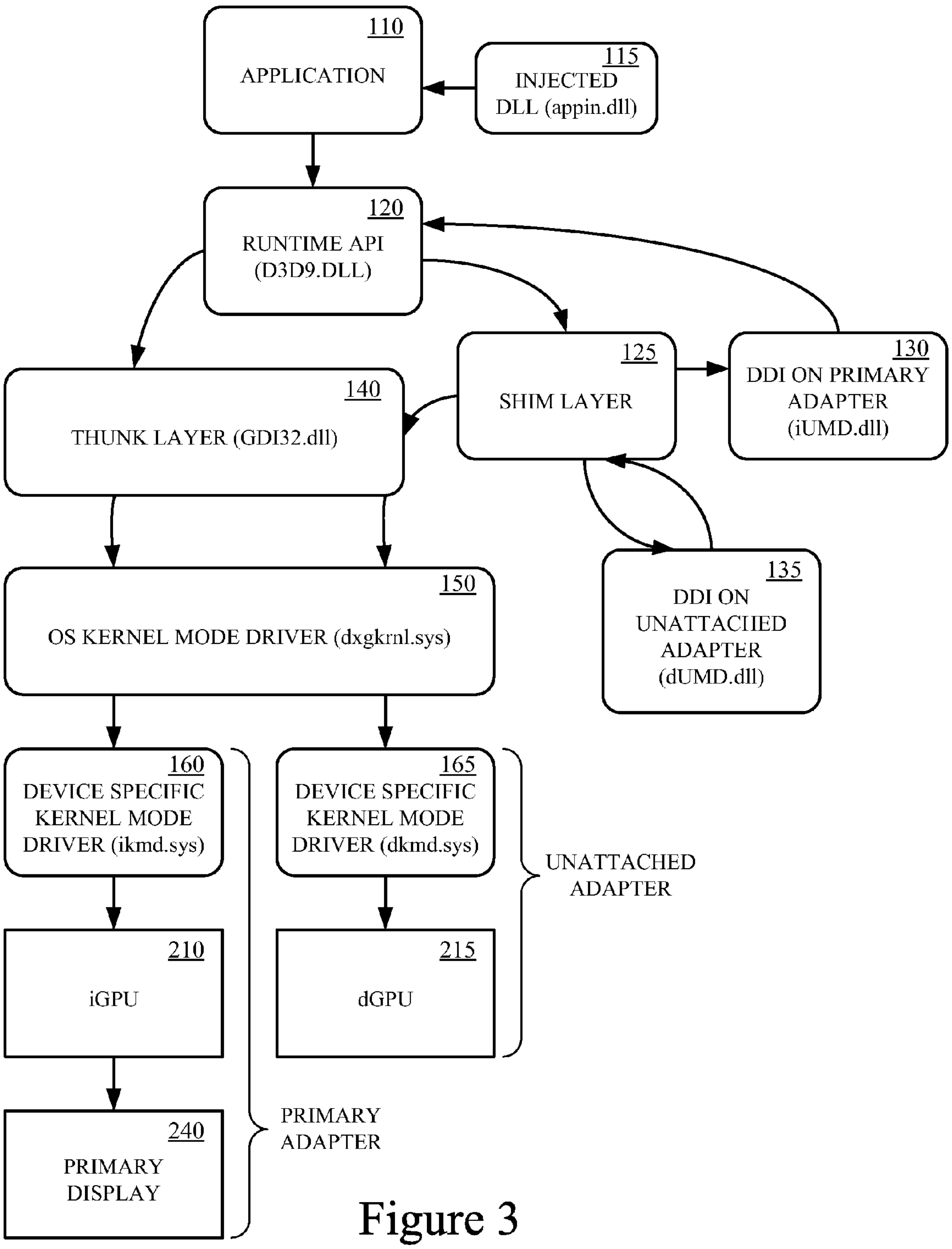

110
APPLICATION
115
INJECTED DLL (appin.dll)
120
RUNTIME API (D3D9.DLL)
125
SHIM LAYER
130
DDI ON PRIMARY ADAPTER (iUMD.dll)
135
DDI ON UNATTACHED ADAPTER (dUMD.dll)
140
THUNK LAYER (GDI32.dll)
150
OS KERNEL MODE DRIVER (dxgkrnl.sys)
160
DEVICE SPECIFIC KERNEL MODE DRIVER (ikmd.sys)
165
DEVICE SPECIFIC KERNEL MODE DRIVER (dkmd.sys)
UNATTACHED ADAPTER
210
iGPU
215
dGPU
240
PRIMARY DISPLAY
PRIMARY ADAPTER

Figure 3

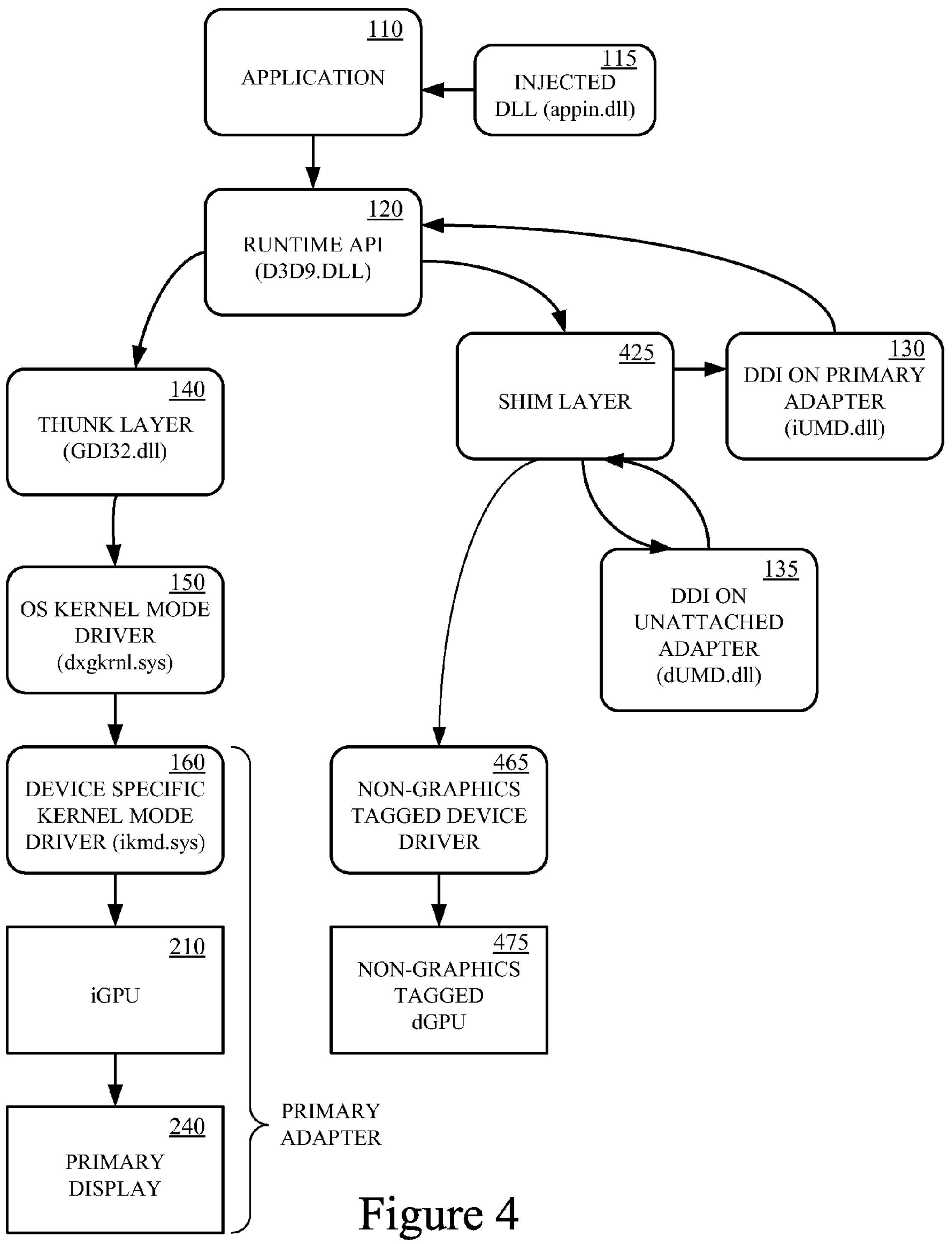

110
APPLICATION
115
INJECTED DLL (appin.dll)
120
RUNTIME API (D3D9.DLL)
140
THUNK LAYER (GDI32.dll)
425
SHIM LAYER
130
DDI ON PRIMARY ADAPTER (iUMD.dll)
150
OS KERNEL MODE DRIVER (dxgkrnl.sys)
135
DDI ON UNATTACHED ADAPTER (dUMD.dll)
160
DEVICE SPECIFIC KERNEL MODE DRIVER (ikmd.sys)
465
NON-GRAPHICS TAGGED DEVICE DRIVER
210
iGPU
475
NON-GRAPHICS TAGGED dGPU
240
PRIMARY DISPLAY
PRIMARY ADAPTER

Figure 4

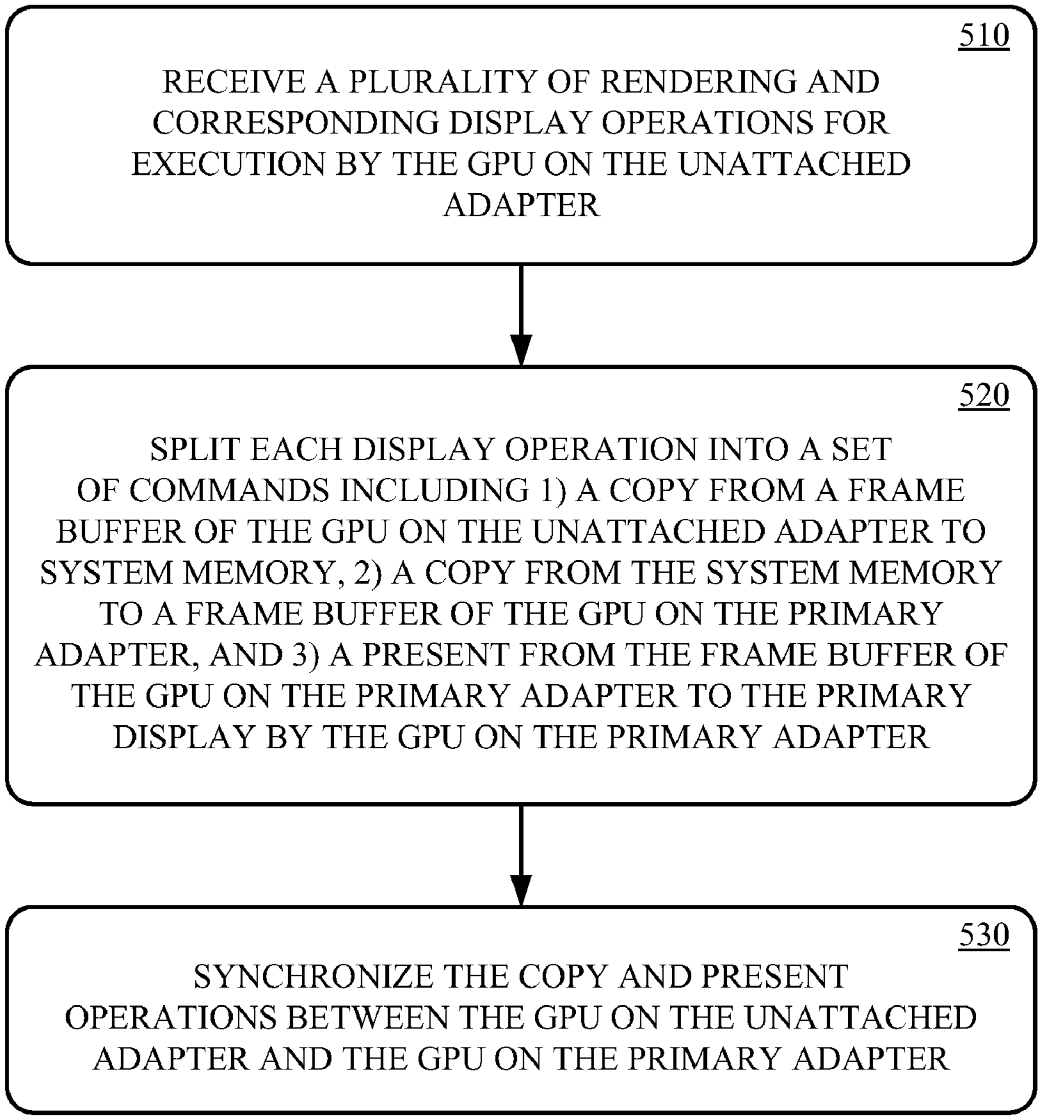

510
RECEIVE A PLURALITY OF RENDERING AND CORRESPONDING DISPLAY OPERATIONS FOR EXECUTION BY THE GPU ON THE UNATTACHED ADAPTER
520
SPLIT EACH DISPLAY OPERATION INTO A SET OF COMMANDS INCLUDING 1) A COPY FROM A FRAME BUFFER OF THE GPU ON THE UNATTACHED ADAPTER TO SYSTEM MEMORY, 2) A COPY FROM THE SYSTEM MEMORY TO A FRAME BUFFER OF THE GPU ON THE PRIMARY ADAPTER, AND 3) A PRESENT FROM THE FRAME BUFFER OF THE GPU ON THE PRIMARY ADAPTER TO THE PRIMARY DISPLAY BY THE GPU ON THE PRIMARY ADAPTER
530
SYNCHRONIZE THE COPY AND PRESENT OPERATIONS BETWEEN THE GPU ON THE UNATTACHED ADAPTER AND THE GPU ON THE PRIMARY ADAPTER

Figure 5

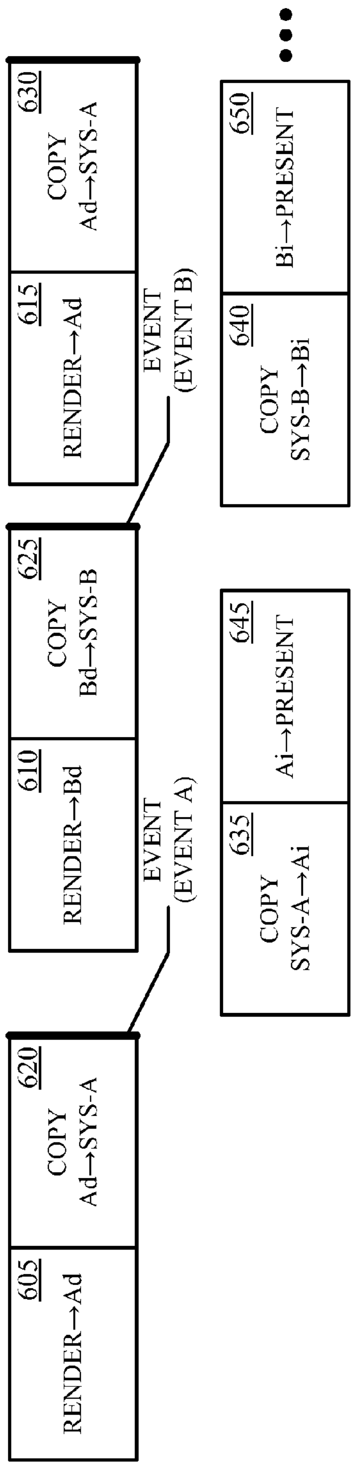


Figure 6

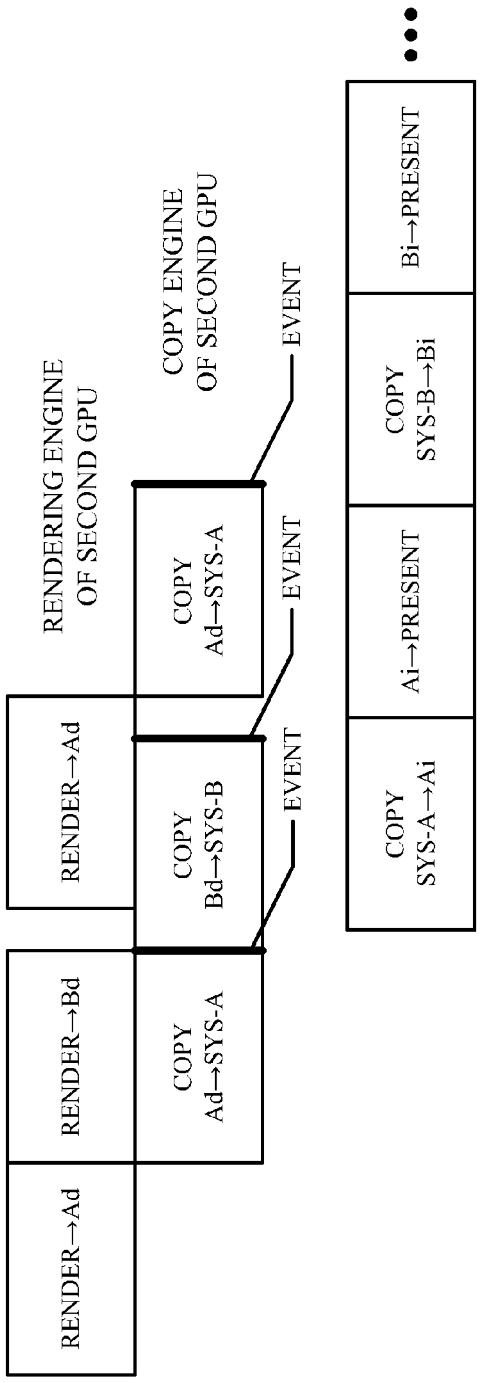


Figure 7

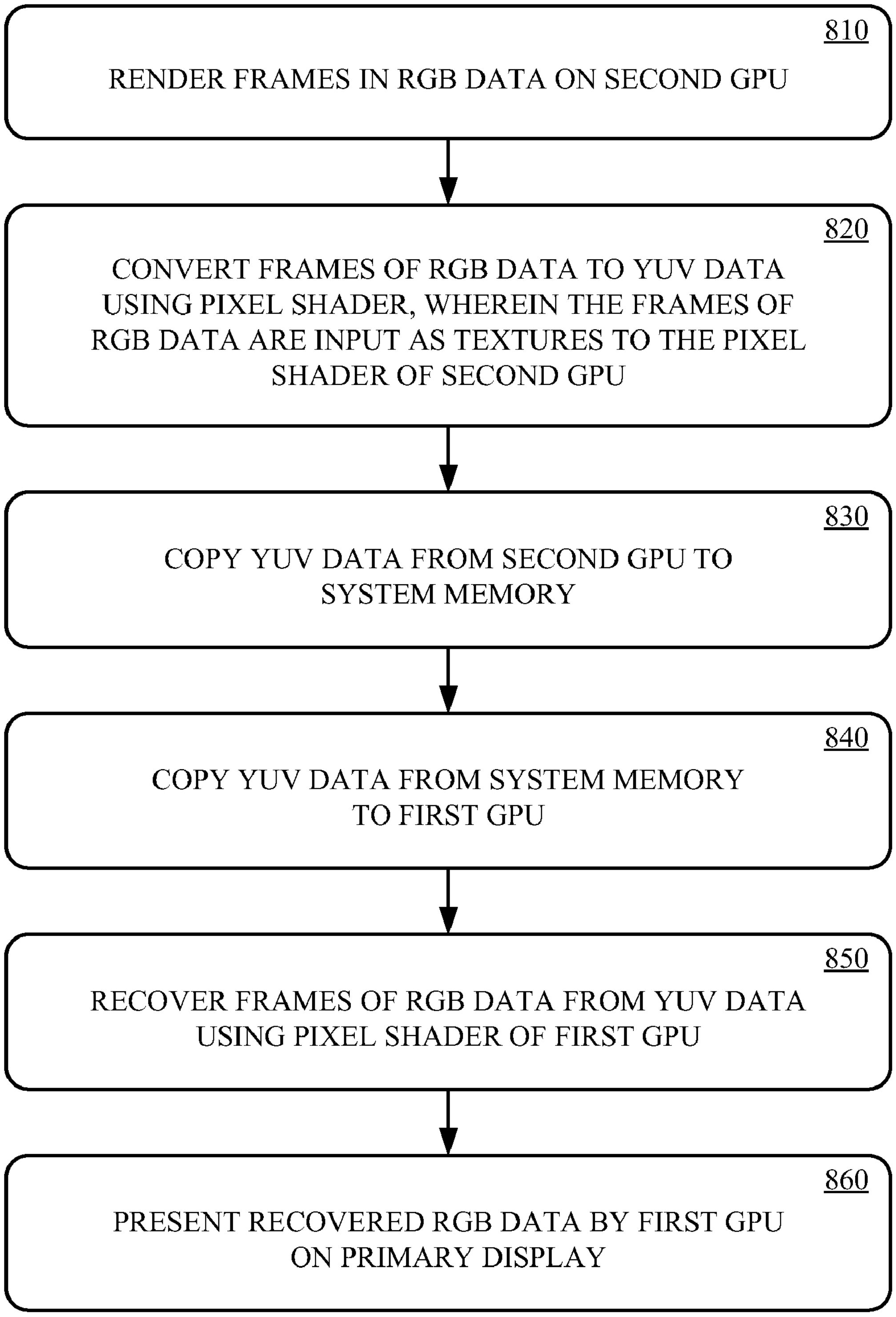

810
RENDER FRAMES IN RGB DATA ON SECOND GPU
820
CONVERT FRAMES OF RGB DATA TO YUV DATA USING PIXEL SHADER, WHEREIN THE FRAMES OF RGB DATA ARE INPUT AS TEXTURES TO THE PIXEL SHADER OF SECOND GPU
830
COPY YUV DATA FROM SECOND GPU TO SYSTEM MEMORY
840
COPY YUV DATA FROM SYSTEM MEMORY TO FIRST GPU
850
RECOVER FRAMES OF RGB DATA FROM YUV DATA USING PIXEL SHADER OF FIRST GPU
860
PRESENT RECOVERED RGB DATA BY FIRST GPU ON PRIMARY DISPLAY

Figure 8

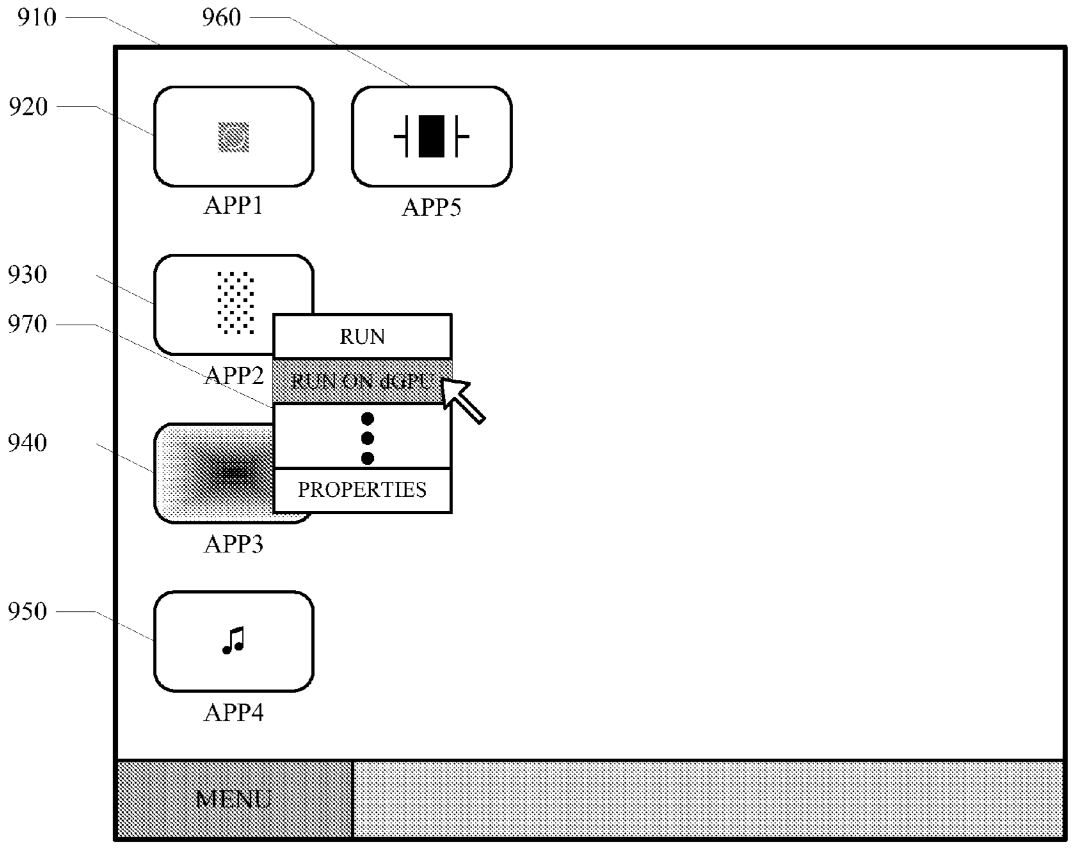

910
960
920
APP1
APP5
930
970
APP2
RUN
RUN ON dGPU
940
APP3
PROPERTIES
950
APP4
MENU

Figure 9

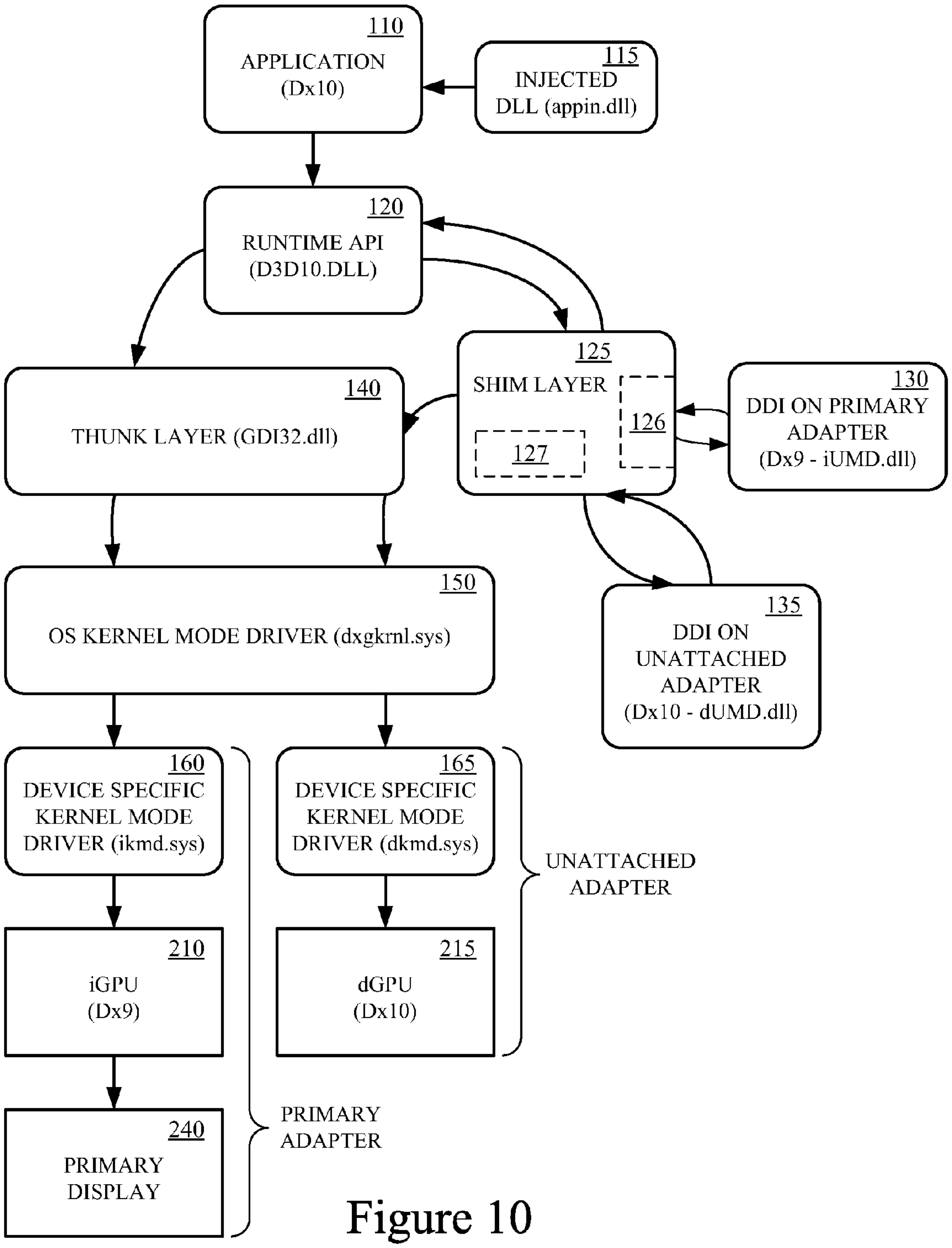

110
APPLICATION (Dx10)
115
INJECTED DLL (appin.dll)
120
RUNTIME API (D3D10.DLL)
125
SHIM LAYER
126
127
130
DDI ON PRIMARY ADAPTER (Dx9 - iUMD.dll)
140
THUNK LAYER (GDI32.dll)
135
DDI ON UNATTACHED ADAPTER (Dx10 - dUMD.dll)
150
OS KERNEL MODE DRIVER (dxgkrnl.sys)
160
DEVICE SPECIFIC KERNEL MODE DRIVER (ikmd.sys)
165
DEVICE SPECIFIC KERNEL MODE DRIVER (dkmd.sys)
UNATTACHED ADAPTER
210
iGPU (Dx9)
215
dGPU (Dx10)
240
PRIMARY DISPLAY
PRIMARY ADAPTER

Figure 10

CO-PROCESSING TECHNIQUES ON HETEROGENEOUS GPUS HAVING DIFFERENT DEVICE DRIVER INTERFACES

CROSS-REFERENCE TO RELATED APPLICATIONS

[0001] This claims the benefit of U.S. Provisional Patent Application No. 61/243,155 filed Sep. 16, 2009 and U.S. Provisional Patent Application No. 61/243,164 filed Sep. 17, 2009.

BACKGROUND OF THE INVENTION

[0002] Conventional computing systems may include a discrete graphics processing unit (dGPU) or an integral graphics processing unit (iGPU). The discrete GPU and integral GPU are heterogeneous because of their different designs. The integrated GPU generally has relatively poor processing performance compared to the discrete GPU. However, the integrated GPU generally consumes less power compared to the discrete GPU.

[0003] The conventional operating system does not readily support co-processing using such heterogeneous GPUs. Referring to FIG. **1**, a graphics processing technique according to the conventional art is shown. When an application **110** starts, it calls the user mode level runtime application programming interface (e.g., DirectX API d3d9.dll) **120** to determine what display adapters are available. In response, the runtime API **120** enumerates the adapters that are attached to the desktop (e.g., the primary display **180**). A display adapter **165**, **175**, even recognized and initialized by the operating system, will not be enumerated in the adapter list by the runtime API **120** if it is not attached to the desktop. The runtime API **120** loads the device driver interface (DDI) (e.g., user mode driver (umd.dll)) **130** for the GPU **170** attached to the primary display **180**. The runtime API **120** of the operating system will not load the DDI of the discrete GPU **175** because the discrete GPU **175** is not attached to the display adapter. The DDI **130** configures command buffers of the graphics processor **170** attached to the primary display **180**. The DDI **130** will then call back to the runtime API **120** when the command buffers have been configured.

[0004] Thereafter, the application **110** makes graphics request to the user mode level runtime API (e.g., DirectX API d3d9.dll) **120** of the operating system. The runtime **120** sends graphics requests to the DDI **130** which configures command buffers. The DDI calls to the operating system kernel mode driver (e.g., DirectX driver dxgkrnl.sys) **150**, through the runtime API **120**, to schedule the graphics request. The operating system kernel mode driver then calls to the device specific kernel mode driver (e.g., kmd.sys) **150** to set the command register of the GPU **170** attached to the primary display **180** to execute the graphics requests from the command buffers. The device specific kernel mode driver **160** controls the GPU **170** (e.g., integral GPU) attached to the primary display **180**.

[0005] Therefore, there is a need to enable co-processing on heterogeneous GPUs. For example, it may be desired to use a first GPU to perform graphics processing for a first class of applications and a second GPU for a second class of applications depending upon processing performance and power consumption parameters.

SUMMARY OF THE INVENTION

[0006] Embodiments of the present technology are directed toward graphics co-processing. The present technology may best be understood by referring to the following description and accompanying drawings that are used to illustrate embodiment of the present technology.

[0007] In one embodiment, a graphics co-processing method includes injecting an application initialization routine when an application starts. The injected application initialization routine includes an entry point that changes a search path for a device driver interface to a search path of a shim layer library. As a result, the loaded shim layer library initializes a device driver interface of a first class for a first graphics processing unit class on a primary adapter and a device driver interface of a second class for a second graphics processing unit on an unattached adapter. The shim translates calls between the first device driver interface of the first class and the second device driver interface of the second class.

[0008] In another embodiment, a graphics co-processing method includes loading a shim layer library, by a runtime application programming interface. The shim layer library loads and initializing a device driver interface on the primary adapter. The shim layer also loads and initializing a device driver interface on an unattached adapter. The shim layer also translates calls between the runtime application programming interface and commands of a first device driver interface class for the first device driver interface class. The shim layer may further convert a display format of a second device driver interface class for the device driver interface on an unattached adapter to a display format of the first device driver interface class.

BRIEF DESCRIPTION OF THE DRAWINGS

[0009] Embodiments of the present technology are illustrated by way of example and not by way of limitation, in the figures of the accompanying drawings and in which like reference numerals refer to similar elements and in which:

[0010] FIG. **1** shows a graphics processing technique according to the convention art.

[0011] FIG. **2** shows a graphics co-processing computing platform, in accordance with one embodiment of the present technology.

[0012] FIG. **3** shows a graphics co-processing technique, in accordance with one embodiment of the present technology.

[0013] FIG. **4** shows a graphics co-processing technique, in accordance with another embodiment of the present technology.

[0014] FIG. **5** shows a method of synchronizing copy and present operations on a first and second GPU, in accordance with one embodiment of the present technology.

[0015] FIG. **6** shows an exemplary set of render and display operations, in accordance with one embodiment of the present technology.

[0016] FIG. **7** shows an exemplary set of render and display operations, in accordance with another embodiment of the present technology.

[0017] FIG. **8** shows a method of compressing rendered data, in accordance with one embodiment of the present technology.

[0018] FIG. **9** shows an exemplary desktop **910** including an exemplary graphical user interface for selection of the GPU to run a given application, in accordance with one embodiment of the present technology.

[0019] FIG. **10** shows a graphics co-processing technique, in accordance with another embodiment of the present technology.

DETAILED DESCRIPTION OF THE INVENTION

[0020] Reference will now be made in detail to the embodiments of the present technology, examples of which are illustrated in the accompanying drawings. While the present technology will be described in conjunction with these embodiments, it will be understood that they are not intended to limit the invention to these embodiments. On the contrary, the invention is intended to cover alternatives, modifications and equivalents, which may be included within the scope of the invention as defined by the appended claims. Furthermore, in the following detailed description of the present technology, numerous specific details are set forth in order to provide a thorough understanding of the present technology. However, it is understood that the present technology may be practiced without these specific details. In other instances, well-known methods, procedures, components, and circuits have not been described in detail as not to unnecessarily obscure aspects of the present technology.

[0021] Embodiments of the present technology introduce a shim layer between the runtime API (e.g., DirectX) and the device driver interface (DDI) (e.g., user mode driver (UMD)) to separate the display commands from the rendering commands, allowing retargeting of rendering commands to an adapter other than the adapter the application is displaying on. In one implementation, the shim layer allows the DDI layer to redirect a runtime (e.g., Direct3D (D3D)) default adapter creation to an off-screen graphics processing unit (GPU), such as a discrete GPU, not attached to the desktop. The shim layer effectively layers the device driver interface, and therefore does not hook a system component.

[0022] Referring to FIG. **2**, a graphics co-processing computing platform, in accordance with one embodiment of the present technology is shown. The exemplary computing platform may include one or more central processing units (CPUs) **205**, a plurality of graphics processing units (GPUs) **210**, **215**, volatile and/or non-volatile memory (e.g., computer readable media) **220**, **225**, one or more chip sets **230**, **235**, and one or more peripheral devices **215**, **240-260** communicatively coupled by one or more busses. The GPUs include heterogeneous designs. In one implementation, a first GPU may be an integral graphics processing unit (iGPU) and a second GPU may be a discrete graphics processing unit (dGPU). The chipset **230**, **235** acts as a simple input/output hub for communicating data and instructions between the CPU **205**, the GPUs **210**, **215**, the computing device-readable media **220**, **225**, and peripheral devices **215**, **240-265**. In one implementation, the chipset includes a northbridge **230** and southbridge **235**. The northbridge **230** provides for communication between the CPU **205**, system memory **220** and the southbridge **235**. In one implementation, the northbridge **230** includes an integral GPU. The southbridge **235** provides for input/output functions. The peripheral devices **215**, **240-265** may include a display device **240**, a network adapter (e.g., Ethernet card) **245**, CD drive, DVD drive, a keyboard, a pointing device, a speaker, a printer, and/or the like. In one implementation, the second graphics processing unit is coupled as a discrete GPU peripheral device **215** by a bus such as a Peripheral Component Interconnect Express (PCIe) bus.

[0023] The computing device-readable media **220**, **225** may be characterized as primary memory and secondary memory. Generally, the secondary memory, such as a magnetic and/or optical storage, provides for non-volatile storage of computer-readable instructions and data for use by the computing device. For instance, the disk drive **225** may store the operating system (OS), applications and data. The primary memory, such as the system memory **220** and/or graphics memory, provides for volatile storage of computer-readable instructions and data for use by the computing device. For instance, the system memory **220** may temporarily store a portion of the operating system, a portion of one or more applications and associated data that are currently used by the CPU **205**, GPU **210** and the like. In addition, the GPUs **210**, **215** may include integral or discrete frame buffers **211**, **216**.

[0024] Referring to FIG. **3**, a graphics co-processing technique, in accordance with one embodiment of the present technology, is shown. When an application **110** starts, it calls the user mode level runtime application programming interface (e.g., DirectX API d3d9.dll) **120** to determine what display adapters are available. In addition, an application initialization routine is injected when the application starts. In one implementation, the application initialization routine is a short dynamic link library (e.g., appln.dll). The application initialization routine injected in the application includes some entry points, one of which includes a call (e.g., set_dll_searchpath( )) to change the search path for the display device driver interface. During initialization, the search path for the device driver interface (e.g., c:\windows\system32\ . . . \umd.dll) is changed to the search path of a shim layer library (e.g., c:\ . . . \coproc\ . . . \umd.dll). Therefore the runtime API **120** will search for the same DDI name but in a different path, which will result in the runtime API **120** loading the shim layer **125**.

[0025] The shim layer library **125** has the same entry points as a conventional display driver interface (DDI). The runtime API **120** passes one or more function pointers to the shim layer **125** when calling into the applicable entry point (e.g., OpenAdapter( ) in the shim layer **125**. The function pointers passed to the shim layer **125** are call backs into the runtime API **120**. The shim layer **125** stores the function pointers. The shim layer **125** loads and initializes the DDI on the primary adapter **130**. The DDI on the primary adapter **130** returns a data structure pointer to the shim layer **125** representing the attached adapter. The shim layer **125** also loads and initializes the device driver interface on the unattached adapter **135** by passing two function pointers which are call backs into local functions of the shim layer **125**. The DDI on the unattached adapter **135** also returns a data structure pointer to the shim layer **125** representing the unattached adapter. The data structure pointers returned by the DDI on the primary adapter **130** and unattached adapter **135** are stored by the shim layer **125**. The shim layer **125** returns to the runtime API **120** a pointer to a composite data structure that contains the two handles. Accordingly, the DDI on the unattached adapter **135** is able to initialize without talking back to the runtime API **120**.

[0026] In one implementation, the shim layer **125** is an independent library. The independent shim layer may be utilized when the primary GPU/display and the secondary GPU are provided by different vendors. In another implementation, the shim layer **125** may be integral to the display device interface on the unattached adapter. The shim layer integral to the display device driver may be utilized when the primary GPU/display and secondary GPU are from the same vendor.

[0027] The application initialization routine (e.g., appln.dll) injected in the application also includes other entry points, one of which includes an application identifier. In one implementation, the application identifier may be the name of the application. The shim layer **125** application makes a call to the injected application initialization routine (e.g., appln.dll) to determine the application identifier when a graphics command is received. The application identifier is compared with the applications in a white list (e.g., a text file). The white list indicates an affinity between one or more applications and the second graphics processing unit. In one implementation, the white list includes one or more applications that would perform better if executed on the second graphics processing unit.

[0028] If the application identifier is not on the white list, the shim layer **125** calls the device driver interface on the primary adapter **130**. The device driver interface on the primary adapter **130** sets the command buffers. The device driver interface on the primary adapter then calls, through the runtime **120** and a thunk layer **140**, to the operating system kernel mode driver (e.g., DirectX driver dxgkrnl.sys) **150**. The operating system kernel mode driver **160** in turn schedules the graphics command with the device specific kernel mode driver (e.g., kmd.sys) **160** for the GPU **210** attached to the primary display **240**. The GPU **210** attached to the primary display **240** is also referred to hereinafter as the first GPU. The device specific kernel mode driver **160** sets command register of the GPU **210** to execute the graphics command on the GPU **210** (e.g., integral GPU) attached to the primary display **240**.

[0029] If the application identifier is a match to one or more identifiers on the white list, the handle from the runtime API **120** is swapped by the shim layer **125** with functions local to the shim layer **125**. For a rendering command, the local function stored in the shim layer **125** will call into the DDI on the unattached adapter **135** to set command buffer. In response, the DDI on the unattached adapter **135** will call local functions in the shim layer **125** that route the call through the thunk layer **140** to the operating system kernel mode driver **150** to schedule the rendering command. The operating system kernel mode driver **150** calls the device specific kernel mode driver (e.g., dkmd.sys) **165** for the GPU on the unattached adapter **215** to set the command registers. The GPU on the unattached adapter **215** (e.g., discrete GPU) is also referred to hereinafter as the second GPU. Alternatively, the DDI on the unattached adapter **135** can call local functions in the thunk layer **140**. The thunk layer **140** routes the graphics request to the operating system kernel mode driver (e.g., DirectX driver dxgkrnl.sys) **150**. The operating system kernel mode driver **150** schedules the graphics command with the device specific kernel mode driver (e.g., dkmd.sys) **165** on the unattached adapter. The device specific kernel mode driver **165** controls the GPU on the unattached adapter **215**.

[0030] For a display related command (e.g., Present( ), the shim layer **125** splits the display related command received from the application **110** into a set of commands for execution by the GPU on the unattached adapter **215** and another set of commands for execution by the GPU on the primary adapter **210**. In one implementation, when the shim layer **125** receives a present call from the runtime **120**, the shim layer **125** calls to the DDI on the unattached adapter **135** to cause a copy the frame buffer **216** of the GPU on the unattached adapter **215** to a corresponding buffer in system memory **220**. The shim layer **125** will also call the DDI on the primary adapter **130** to cause a copy from the corresponding buffer in system memory **220** to the frame buffer **211** of the GPU on the attached adapter **210** and then a present by the GPU on the attached adapter **210**. The memory accesses between the frame buffers **211**, **216** and system memory **220** may be direct memory accesses (DMA). To synchronize the copy and presents on the GPUs **210**, **215**, a display thread is created, that is notified when the copy to system memory by the second GPU **215** is done. The display thread will then queue the copy from system memory **220** and the present call into the GPU on the attached adapter **210**.

[0031] In another implementation, the operating system (e.g., Window7Starter) will not load a second graphics driver **165**. Referring now to FIG. **4**, a graphics co-processing technique, in accordance with another embodiment of the present technology, is shown. When the operation system will not load a second graphics driver, the second GPU **475** is tagged as a non-graphics device adapter that has its own driver **465**. Therefore the second GPU **475** and its device specific kernel mode driver **465** are not seen by the operating system as a graphics adapter. In one implementation, the second GPU **475** and its driver **465** are tagged as a memory controller. The shim layer **125** loads and configures the DDI **130** for the first GPU **210** on the primary adapter and the DDI **135** for the second GPU **475** If there is a specified affinity for executing rendering commands from the application **110** on the second GPU **475**, the shim layer **125** intercepts the rendering commands sent by the runtime API **120** to the DDI on the primary adapter **130**, calls the DDI on the unattached adapter to sets the commands buffers for the second GPU **475**, and routes them to the driver **465** for the second GPU **475**. The shim layer **125** also intercepts the callbacks from the driver **465** for the second GPU **475** to the runtime **120**. In another implementation, the shim layer **125** implements the DDI **135** for the second GPU **475**. Accordingly, the shim layer **125** splits graphics command and redirects them to the two DDIs **130**, **135**.

[0032] Accordingly, the embodiments described with reference to FIG. **3**, enables the application to run on a second GPU instead of a first GPU when the particular version of the operating system will allow the driver for the second GPU to be loaded but the runtime API will not allow a second device driver interface to be initialized. The embodiments described with reference to FIG. **4** enables an application to run on a second GPU, such as a discrete GPU, instead of a first GPU, such as an integrated GPU, when the particular version of the operation system (e.g., Win7Starter) will not allow the driver for the second GPU to be loaded. The DDI **135** for the second GPU **475** cannot talkback through the runtime **120** or the thunk layer **140** to a graphics adapter handled by an OS specific kernel mode driver.

[0033] Referring now to FIG. **5**, a method of synchronizing the copy and present operations on the first and second GPUs is shown. The method is illustrated in FIG. **6** with reference to an exemplary set of render and display operations, in accordance with one embodiment of the present technology. At **510**, the shim layer **125** receives a plurality of rendering **605-615** and display operations for execution by the GPU on the unattached adapter **215**. At **520**, the shim layer **125** splits each display operation into a set of commands including 1) a copy **620-630** from a frame buffer **216** of the GPU on the unattached adapter **215** to a corresponding buffer in system memory **220** having shared access with the GPU on the attached adapter **210**, 2) a copy **635**, **640** from the buffer in shared system memory **220** to a frame buffer of the GPU on the primary adapter **210**, and 3) a present **645**, **650** on the primary display **240** by the GPU on the primary adapter **210**. At **530**, the copy and present operations on the first and second GPUs **210**, **215** are synchronized.

[0034] The frame buffers **211**, **216** and shared system memory **220** may be double or ring buffered. In a double buffered implementation, the current rendering operations is stored in a given one of the double buffers **605** and the other one of the double buffers is blitted to a corresponding given one of the double buffers of the system memory. When the rendering operation is complete, the next rendering operation is stored in the other one of the double buffers and the content of the given one of the double buffers is blitted **620** to the corresponding other one of the double buffers of the system memory. The rendering and blitting alternate back and forth between the buffers of the frame buffer of the second GPU **215**. The blit to system memory is executed asynchronously. In another implementation, the frame buffer of the second GPU **215** is double buffered and the corresponding buffer in system memory **220** is a three buffer ring buffer.

[0035] After the corresponding one of the double buffers of the frame buffer **216** in the second GPU **215** is blitted **620** to the system memory **220**, the second GPU **210** generates an interrupt to the OS. In one implementation, the OS is programmed to signal an event to the shim layer **125** in response to the interrupt and the shim layer **125** is programmed to wait on the event before sending a copy command **635** and a present command **645** to the first GPU **210**. In a thread separate from the application thread, referred to hereinafter as the display thread, the shim layer waits for receipt of the event indicating that the copy from the frame buffer to system memory is done, referred to herein after as the copy event interrupt. A separate thread is used so that the rendering commands on the first and second GPUs **210**, **215** are not stalled in the application thread while waiting for the copy event interrupt. The display thread may also have a higher priority than the application thread.

[0036] A race condition may occur where the next rendering to a given one of the double buffers for the second GPU **215** begins before the previous copy from the given buffer is complete. In such case, a plurality of copy event interrupts may be utilized. In one implementation, a ring buffer and four events are utilized.

[0037] Upon receipt of the copy event interrupt, the display thread queues the blit from system memory **220** and the present call into the first GPU **210**. The first GPU **210** blits the given one of the system memory **220** buffers to a corresponding given one of the frame buffers of the first GPU **210**. When the blit operation is complete, the content of the given one of the frame buffers of the first GPU **210** is presented on the primary display **240**. When the next copy and present commands are received by the first GPU **210**, the corresponding other of the system memory **220** buffers is blitted into the other one of the frame buffer of the first GPU **210** and then the content is presented on the primary display **240**. The blit and present alternate back and forth between the double buffered frame buffer of the first GPU **210**. The copy event interrupt is used to delay programming, thereby effectively delaying the scheduling of the copy from system memory **220** to the frame buffer of the first GPU **210** and presenting on the primary display **240**.

[0038] In one implementation, a notification on the display side indicates that the frame has been present on the display **240** by the first GPU **210**. The OS is programmed to signal an event when the command buffer causing the first GPU **210** to present its frame buffer on the display is done executing. The notification maintains synchronization where an application runs with vertical blank (vblank) synchronization.

[0039] Referring now to FIG. **7**, an exemplary set of render and display operations, in accordance with another embodiment of the present technology, is shown. The rendering and copy operations executed on the second GPU **215** may be performed by different engines. Therefore, the rendering and copy operations may be performed substantially simultaneously in the second GPU **215**.

[0040] Generally, the second GPU **215** is coupled to the system memory **220** by a bus having a relatively high bandwidth. However, in some systems the bus coupling the second GPU **215** may not provide sufficient bandwidth for blitting the frame buffer **216** of the second GPU **215** to system memory **220**. For example, an application may be rendered at a resolution of 1280×1024 pixels. Therefore, approximately 5 MB/frame of RGB data is rendered. If the application renders at **100** frame/s, than the second GPU needs approximately 500 MB/s for blitting upstream to the system memory **220**. However, a Peripheral Component Interconnect Express (PCIe) 1× bus typically used to couple the second GPU **215** system memory **220** has a bandwidth of approximately 250 MB/s in each direction. Referring now to FIG. **8**, a method of compressing rendered data, in accordance with one embodiment of the present technology is shown. The second GPU **215** renders frames of RGB data, at **810**. At **820**, the frames of RGB data are converted using a pixel shader in the second GPU **215** to YUV sub-sample data. The RGB data is processed as texture data by the pixel shader in three passes to generate YUV sub-sample data. In one implementation, the U and V components are sub-sampled spatially, however, the Y is not sub-sampled. The RGB data may be converted to YUV data using the 4.2.0 color space conversion algorithm. At **830**, the YUV sub-sample data is blitted to the corresponding buffers in the system memory with an asynchronous copy engine of the second GPU. The YUV sub-sample data is blitted from the system memory to buffers of the first GPU, at **840**. The YUV data is blitted to corresponding texture buffers in the second GPU. The Y, U, and V sub-sample data are buffered in three corresponding buffers, and therefore the copy from frame buffer of the second GPU **215** to the system memory **220** and the copy from system memory **220** to the texture buffers of first GPU **210** are each implemented by sets of three copies. The YUV sub-sample data is converted using a pixel shader in the first GPU **210** to recreate the RGB frame data, at **850**. The device driver interface on the attached adapter is programmed to render a full screened aligned quad from the corresponding texture buffers holding the YUV data. At **860**, the recreated RGB frame data is then presented on the primary display **240** by the first GPU **210**. Accordingly, the shaders are utilized to provide YUV compression and decompression.

[0041] In one implementation, each buffer of Y, U and V samples is double buffered in the frame buffer of the second GPU **215** and the system memory **220**. In addition, the Y, U and V samples copied into the first GPU **210** are double buffered as textures. In another implementation, the Y, U and V sample buffers in the second GPU **215** and corresponding texture buffers in the first GPU **210** are each double buffered. The Y, U and V sample buffered in the system memory **220** may each be triple buffered.

[0042] In one implementation, the shim layer **125** tracks the bandwidth needed for blitting and the efficiency of transfers on the bus to enable the compression or not. In another implementation, the shim layer **125** enables the YUV compression or not based on the type of application. For example, the shim layer **125** may enable compression for game application but not for technical applications such as a Computer Aided Drawing (CAD) application.

[0043] In one embodiment the white list accessed by the shim layer **125** to determine if graphics requests should be executed on the first GPU **210** or the second GPU **215** is loaded and updated by the a vendor and/or system administrator. In another embodiment, a graphical user interface can be provided to allow the user to specific the use of the second GPU (e.g., discrete GPU) **215** for rendering a given application. The user may right click on the icon for the given application. In response to the user selection, a graphical user interface may be generated that allows the user to specify the second GPU for use when rendering image for the given application. In one implementation, the operating system is programmed to populate the graphical interface with a choice to run the given application on the GPU on the unattached adapter. A routine (e.g., dynamic linked library) registered to handle this context menu item will scan the shortcut link to the application, gather up the options and argument, and then call an application launcher that will spawn a process to launch the application as well as setting an environment variable that will be read by the shim layer **125**. In response, the shim layer **125** will run the graphics context for the given application on the second GPU **215**. Therefore, the user can override, update, or the like, the white list loaded on the computing device.

[0044] Referring now to FIG. **9**, an exemplary desktop **910** including an exemplary graphical user interface for selection of the GPU to run a given application on is shown. The desktop includes icons **920**-**950** for one or more applications. When the user right clicks on a given application, **930** a pull-down menu **970** is generated. The pull-down menu **970** is populated with an additional item of ‘run on dGPU’ or the like. The menu item for the second GPU **215** may provide for product branding by identifying the manufacturer and/or model of the second GPU. If the user selects the ‘run’ item or double left clicks on the icon, the graphics requests from the given application will run on the GPU on the primary adapter (e.g., the default iGPU) **210**. If the user selects the ‘run on dGPU’ item, the graphics requests from the given application will run on the GPU on the unattached adapter (e.g., dGPU) **215**.

[0045] In another implementation, the second graphics processing unit may support a set of rendering application programming interfaces and the first graphics processing unit may support a limited subset of the same application programming interfaces. An application programming interface is implemented by a different runtime API **120** and a matching driver interface **130**. Referring now to FIG. **10**, a graphics co-processing technique, in accordance with another embodiment of the present technology, is shown. The runtime API **120** loads a shim layer **125** that will support all device driver interfaces. The shim layer **125** loads and configures the DDI **130** for the first GPU **210** using a device driver interface that this one supports on the primary adapter and the DDI **135** for the second GPU **215** of a second device driver interface that can talk with the runtime API **120**. For example, in one implementation, the second GPU **215** may be a DirectX10 class device and the first GPU **210** may be a DirectX9 class device that does not support DirectX10. The shim layer **125** appears to the DDI **130** for the first GPU **210** as a first application programming class runtime API (e.g., D3D9.dll), translates command between the two device driver interface classes and may also convert between display formats.

[0046] The shim layer **125** includes a translation layer **126** that translates calls between the runtime API **120** device driver interface and the device driver interface class. In one implementation, the shim layer **125** translates display commands between the DirectX10 runtime API **120** and the DirectX9 DDI on the primary adapter **130**. The shim layer, therefore, creates a Dx9 compatible context on the first GPU **210**, which is the recipient of frames rendered by the Dx10 class second GPU **215**. The shim layer **125** advantageously splits graphics commands into rendering and display commands, redirects the rendering commands to the DDI on the unattached adapter **135** and the display commands to the DDI on the primary adapter **130**. The shim layer also translates between the commands for the Dx9 DDI on the primary adapter **130**, the Dx10 DDI on the unattached adapter **135**, the Dx10 runtime API **120** and Dx10 thunk layer **140**, and provides for format conversion of necessary. The shim layer **125**, in one implementation, intercepts commands from the Dx10 runtime **120** and translates these into the DX9 DDI on the primary adapter (e.g., iUMD.dll). The commands may include: CreateResource, OpenResource, DestroyResource, DxgiPresent—which triggers the surface transfer mechanism that ends up with the surface displayed on the iGPU, DxgiRotateResourceIdentities, DxgiBlt—present blits are translated, and DxgiSetDisplayMode.

[0047] The Dx9 DDI **130** for the first GPU **210** cannot talkback directly through the runtime **120** to talk to a graphics adapter handled by an OS specific kernel mode driver because the runtime **120** expects the call to come from a Dx10 device. The shim layer **125** intercepts callbacks from the Dx9 DDI and exchanges device handles, before forwarding the callback to the Dx10 runtime API **120**, which expects the calls to come from a Dx10 device. Dx10 and Dx11 runtime APIs **120** use a layer for presentation called DXGI, which has its own present callback, not existing in the Dx9 callback interface. Therefore, when the display side DDI on the primary adapter calls the present callback, the shim layer translates it to a DXGI callback. For example:

[0048] PFND3DDDI_PRESENTCB->PFNDDXGIDDI_PRESENTCB

[0049] The shim layer **125** may also include a data structure **127** for converting display formats between the first graphics processing unit DDI and the second graphics processing unit DDI. For example, the shim layer **125** may include a lookup table to convert a 10 bit rendering format in Dx10 to an 8 bit format supported by the Dx9 class integrated GPU **210**. The rendered frame may be copied to a staging surface, a two-dimensional (2D) engine of the discrete GPU **215** utilizes the lookup table to convert the rendered frame to a Dx9 format. The Dx9 format frame is then copied to the frame buffer of the integrated GPU **210** and then presented on the primary display **240**. For example, the following format conversions may be performed:

DXGI_FORMAT_R16G16B16A16_FLOAT(render)->D3DDDIFMT_A8R8G8B8(display), DXGI_FORMAT_R10G10B10A2_UNORM(render)->D3DDDIFMT_A8R8G8B8(display).

In one implementation, the copying and conversion can happen as an atomic operation.

[0050] The foregoing descriptions of specific embodiments of the present technology have been presented for purposes of illustration and description. They are not intended to be exhaustive or to limit the invention to the precise forms disclosed, and obviously many modifications and variations are possible in light of the above teaching. The embodiments were chosen and described in order to best explain the principles of the present technology and its practical application, to thereby enable others skilled in the art to best utilize the present technology and various embodiments with various modifications as are suited to the particular use contemplated. It is intended that the scope of the invention be defined by the claims appended hereto and their equivalents.

What is claimed is:

1. One or more computing device readable media having computing device executable instructions which when executed perform a method comprising:

injecting an application initialization routine, when an application starts, that includes an entry point that changes a search path for a device driver interface to a search path of a shim layer library; and

loading the shim layer library, at the changed search path, that initializes a device driver interface of a first class for a first graphics processing unit class on a primary adapter and a device driver interface of a second class for a second graphics processing unit on an unattached adapter, wherein the shim layer library translates calls between the first device driver interface of the first class and the second device driver interface of the second class.

2. The one or more computing device readable media having computing device executable instructions which when executed perform the method of claim **1**, wherein the device driver interface on the primary adapter comprises a DirectX9 user mode driver dynamic linked library (UMD.dll) for the first graphics processing unit.

3. The one or more computing device readable media having computing device executable instructions which when executed perform the method of claim **1**, wherein the device driver interface on the unattached adapter comprises a DirectX10 or DirectX11 user mode driver dynamic linked library (UMD.dll) for the second graphics processing unit.

4. The one or more computing device readable media having computing device executable instructions which when executed perform the method of claim **1**, wherein the application initialization routine comprises a dynamic linked library.

5. The one or more computing device readable media having computing device executable instructions which when executed perform the method of claim **1**, wherein the runtime application programming interface comprises a DirectX10 or DirectX11 application programming interface (D3D10.dll or D3D11.dll).

6. The one or more computing device readable media having computing device executable instructions which when executed perform the method of claim **1**, wherein the first graphics processing unit comprises an integrated graphics processing unit.

7. The one or more computing device readable media having computing device executable instructions which when executed perform the method of claim **1**, wherein the second graphics processing unit comprises a discrete graphics processing unit.

8. The one or more computing device readable media having computing device executable instructions which when executed perform the method of claim **1**, wherein the first graphics processing unit and the second graphics processing unit are heterogeneous graphics processing units.

9. The one or more computing device readable media having computing device executable instructions which when executed perform the method of claim **1**, wherein the shim layer library converts a display format between the first device driver interface of the first class and the second device driver interface of the class.

10. The one or more computing device readable media having computing device executable instructions which when executed perform the method of claim **9**, wherein the shim layer uses a lookup table to convert the display format between the first device driver interface of the first class and the second device driver interface of the class.

11. One or more computing device readable media having computing device executable instructions which when executed perform a method comprising:

loading a shim layer library, by a runtime application programming interface;

loading and initializing a device driver interface on the primary adapter, by the shim layer library;

loading and initializing a device driver interface on an unattached adapter, by the shim layer library;

translating calls between the runtime application programming interface and commands of a first device driver interface class for the first device driver interface class, by the shim layer library; and

converting a display format of a second device driver interface class for the device driver interface on an unattached adapter to a display format of the first device driver interface class.

12. The one or more computing device readable media having computing device executable instructions which when executed perform the method of claim **11**, further comprising routing render commands from the runtime application to the second graphics processing unit running the same device driver interface class.

13. The one or more computing device readable media having computing device executable instructions which when executed perform the method of claim **11**, wherein converting the display format further comprises:

copying a frame by a second graphics processing unit of the second device driver interface class to a staging surface;

converting the rendered frame in the staging surface by a two-dimensional engine of the second graphics unit to a format of the first display driver model class; and

copying the frame in the format of the first device driver interface class to a frame buffer of a first graphics processing unit.

14. The one or more computing device readable media having computing device executable instructions which when executed perform the method of claim **13**, wherein converting the rendered frame in the format of the second device driver interface class to the format of the first device driver interface class comprises mapping the format of the second device driver interface class to the format of the first device driver interface class from a conversion lookup table.

15. The one or more computing device readable media having computing device executable instructions which when executed perform the method of claim **11**, wherein the first graphics processing unit is an integrated graphics processing unit and the second graphics processing unit is a discrete graphics processing unit having a different design than the integrated graphics processing unit.

16. A method comprising:

loading a shim layer library;

loading and initializing a device driver interface of a first class on the primary adapter, by the shim layer library;

loading and initializing a device driver interface of a second class on an unattached adapter, by the shim layer library; and

translating calls between the first device driver interface of the first class on the primary adapter and the second device driver interface of the second class on the unattached adapter, by the shim layer library.

17. The method according to claim **16**, wherein the translated calls comprise resource management and presentation functions.

18. The method according to claim **16**, wherein translating calls between the first device driver interface of the first class on the primary adapter and the second device driver interface of the second class on the unattached adapter comprises:

intercepting callbacks from the first device driver interface of the first class on the primary adapter to a runtime; and

exchanging the handles of the first class with corresponding handles of callbacks of the second class.

19. The method according to claim **16**, wherein the device driver interface on the primary adapter comprises a DirectX9 user mode driver dynamic linked library (UMD.dll) for a first graphics processing unit.

20. The method according to claim **19**, wherein the device driver interface on the unattached adapter comprises a DirectX10 or DirectX11 user mode driver dynamic linked library (UMD.dll) for a second graphics processing unit.

* * * * *