

Ex. C9 – Invalidity of U.S. Patent No. RE49,461 – Wilt

Prior Art Reference: U.S. Patent Application Publication No. 2003/0140179 (“Wilt”)

Wilt was published on July 24, 2003 and therefore qualifies as prior art to U.S. Patent No. RE49,461 (the “’461 patent”) under, at least, pre-AIA 35 U.S.C. §§ 102(a), (b), and/or (e). Wilt anticipates and/or renders obvious claims 21–31 of the ’461 patent (the “Asserted Claims”).

The chart below provides representative examples of where each element of each claim is found within Wilt. Citations are meant to be exemplary, not exhaustive, and NVIDIA reserves the right to identify and discuss additional portions of the reference in support of its contentions and/or to rebut arguments made by Plaintiff. Where NVIDIA states that Wilt “discloses” a limitation, that disclosure may be express, implicit, and/or inherent. Citations to figures, drawings, tables, and the like include reference to any accompanying or related text. All internal cross-references are meant to incorporate the cross-referenced material as if fully set forth therein.

It is NVIDIA’s position that Plaintiff’s Infringement Contentions have not established that any accused product or service infringes any valid claim. Thus, NVIDIA’s statements below should not be treated as an admission, implication, or suggestion that NVIDIA agrees with Plaintiff regarding either the scope, construction, or interpretation of any of the Asserted Claims or the infringement theories advanced by Plaintiff in its Infringement Contentions, including whether any Asserted Claim satisfies 35 U.S.C. §§ 101 or 112. The exemplary disclosures in these invalidity contentions are informed by the apparent interpretation of the asserted claims by Plaintiff. These statements are not intended to suggest that NVIDIA agrees with Plaintiff’s application of any claim term, suggest a proposed construction at this stage of the case, or suggest that construction is needed.

To the extent Plaintiff argues that any element below is not disclosed by Wilt, a person of ordinary skill in the art would have found it obvious to combine the teachings of Wilt with the background knowledge of a person of ordinary skill in the art and/or the additional references, and exemplary teachings, set forth in NVIDIA’s Invalidity Contentions, including all exhibits and appendices thereto. Additional exemplary combinations and reasons for obviousness are described in the cover pleading, and are incorporated by reference herein.

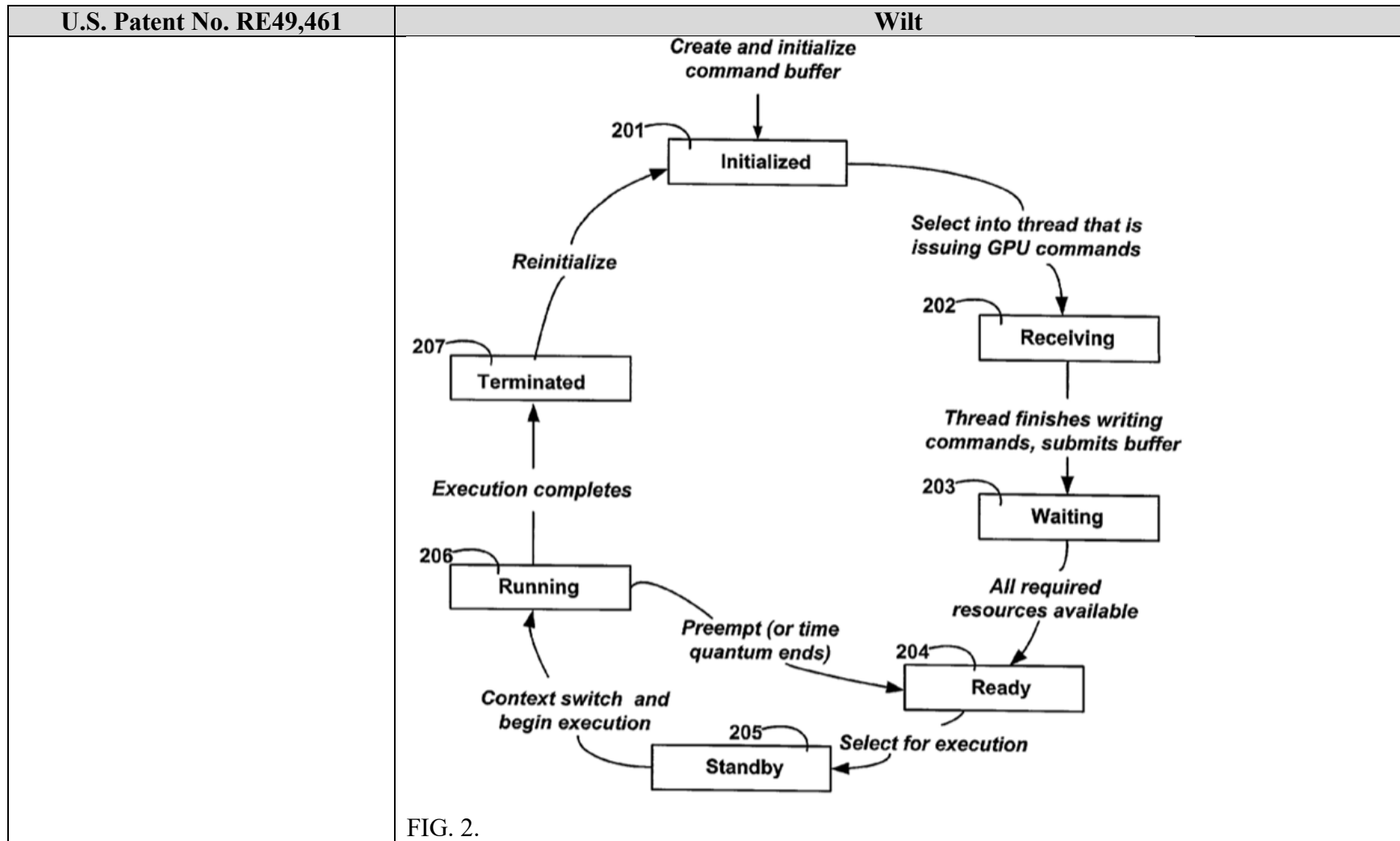
Plaintiff has yet to identify any limitation of the Asserted Claims that it contends is not anticipated and/or rendered obvious by Wilt. NVIDIA therefore expressly reserves the right to respond to any such contention, including by identifying additional obviousness combinations, if Plaintiff makes any such contention. NVIDIA reserves the right to rely on additional citations or sources of evidence that also may be applicable, or that may become applicable in light of claim construction, changes in Plaintiff’s infringement contentions, and/or information obtained during discovery as the case progresses.

Ex. C9 – Invalidity of U.S. Patent No. RE49,461 – Wilt

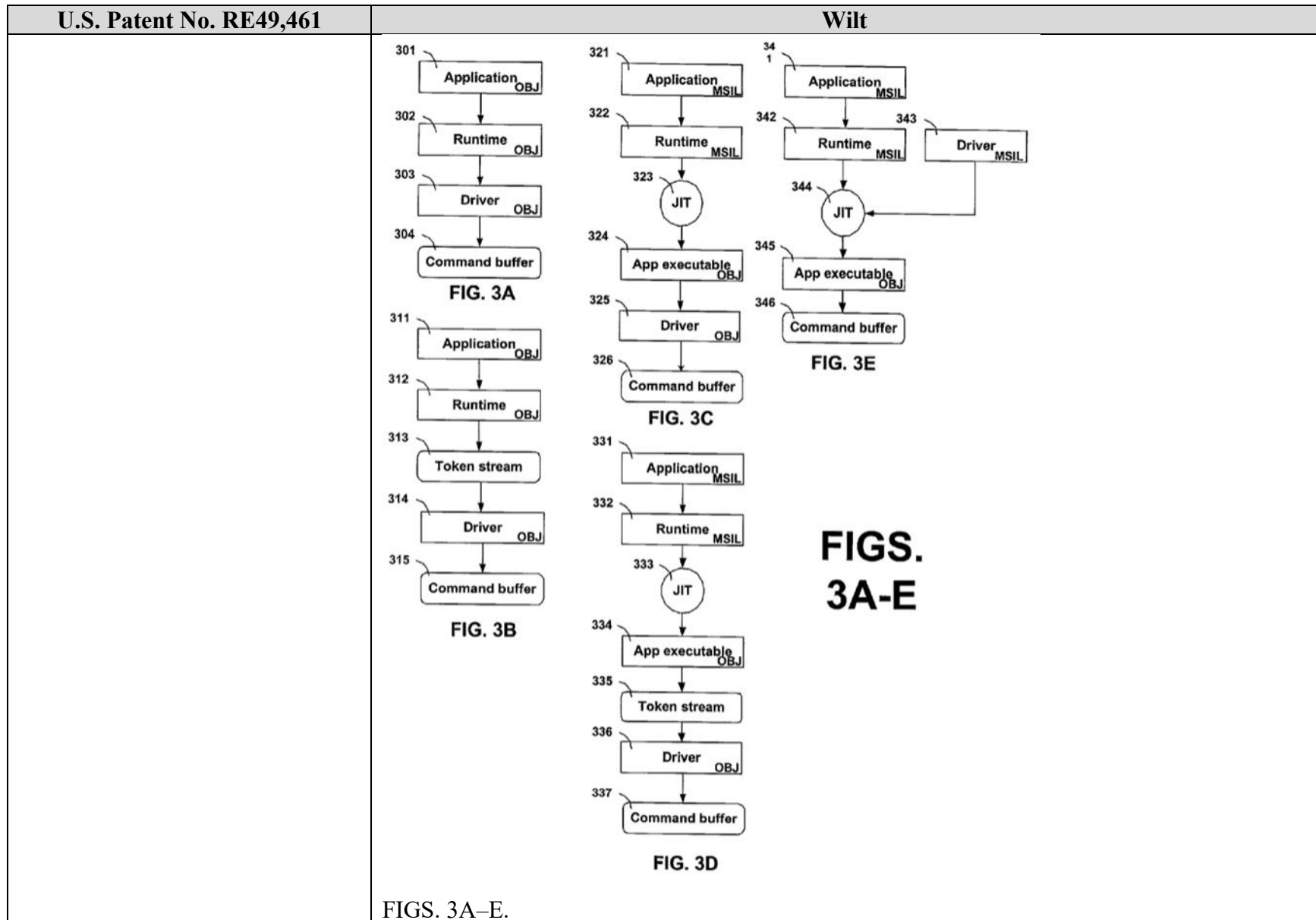
U.S. Patent No. RE49,461	Wilt
21[pre] A method of executing computations representing an artificial neural network on a computer system comprising at least one central processing unit (CPU), a processing unit, a first memory partition, and a second memory partition, the method comprising:	NVIDIA takes no position in these Invalidity Contentions on whether the entirety of the preamble of this claim is a limitation. To the extent it is a limitation, Wilt discloses, expressly and/or inherently a method of executing computations representing an artificial neural network on a computer system comprising at least one central processing unit (CPU), a processing unit, a first memory partition, and a second memory partition. <i>See e.g.:</i> FIG. 1B

FIG. 1B.

Ex. C9 – Invalidity of U.S. Patent No. RE49,461 – Wilt



Ex. C9 – Invalidity of U.S. Patent No. RE49,461 – Wilt



Ex. C9 – Invalidity of U.S. Patent No. RE49,461 – Wilt

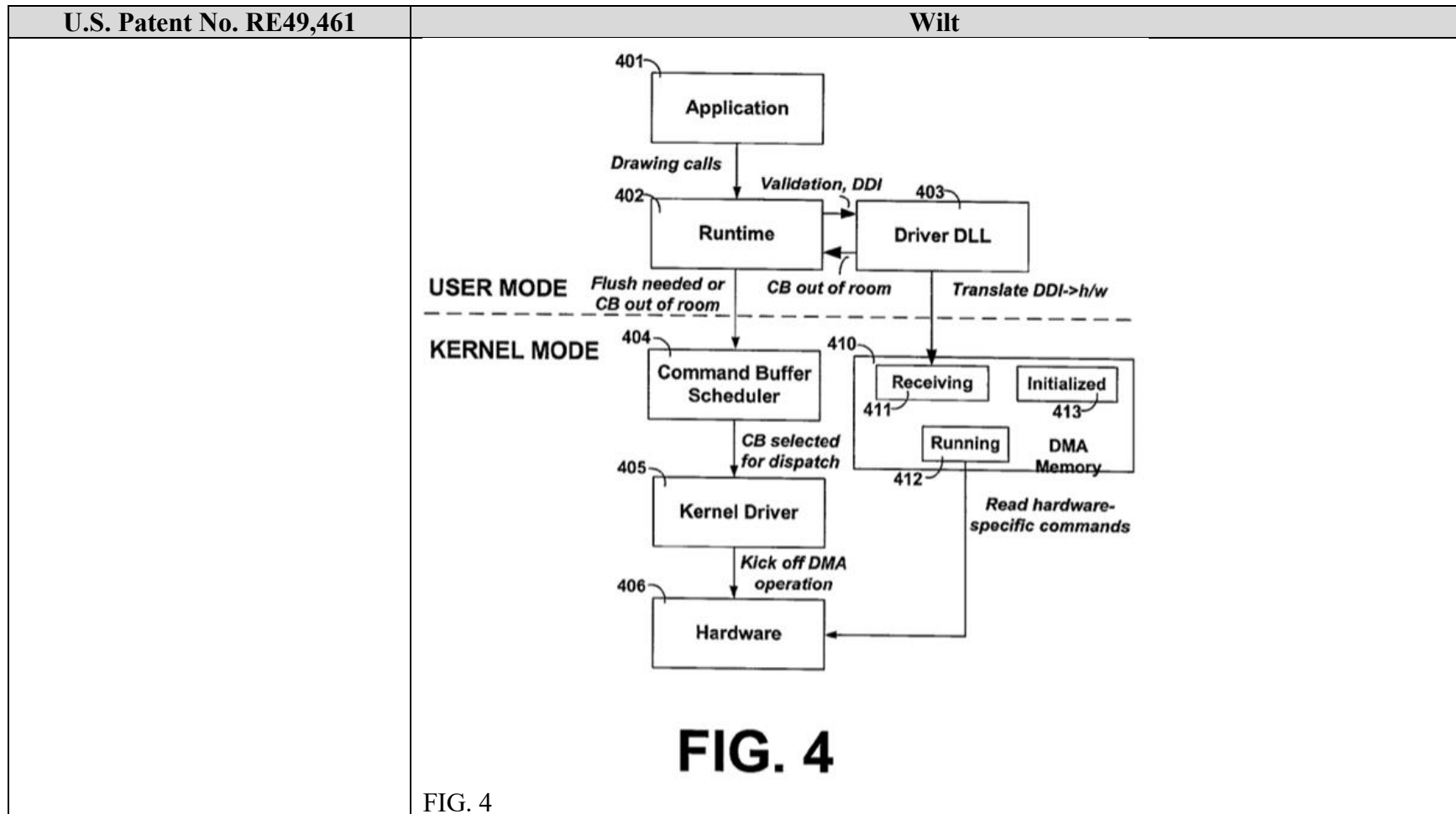


FIG. 4

Ex. C9 – Invalidity of U.S. Patent No. RE49,461 – Wilt

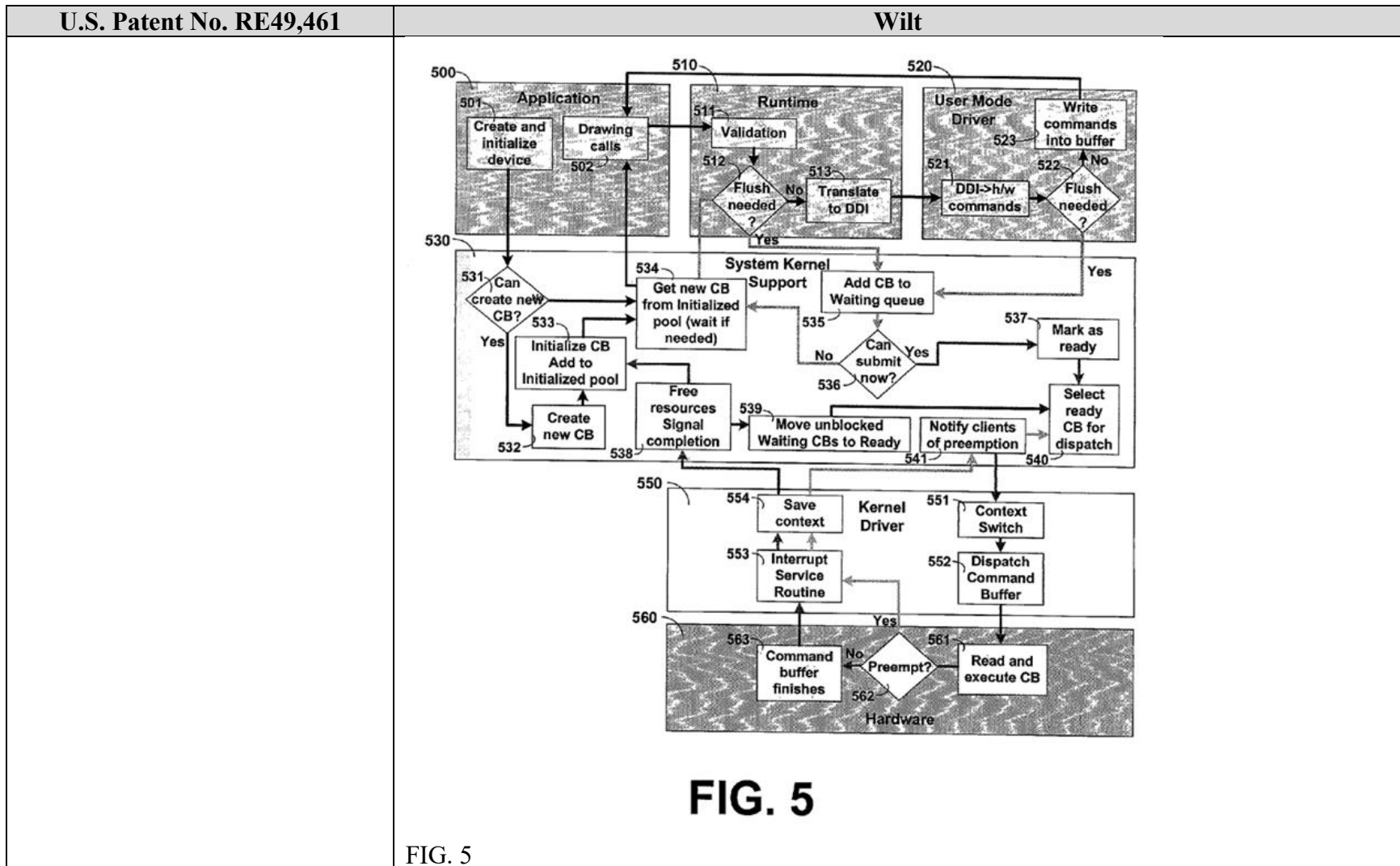
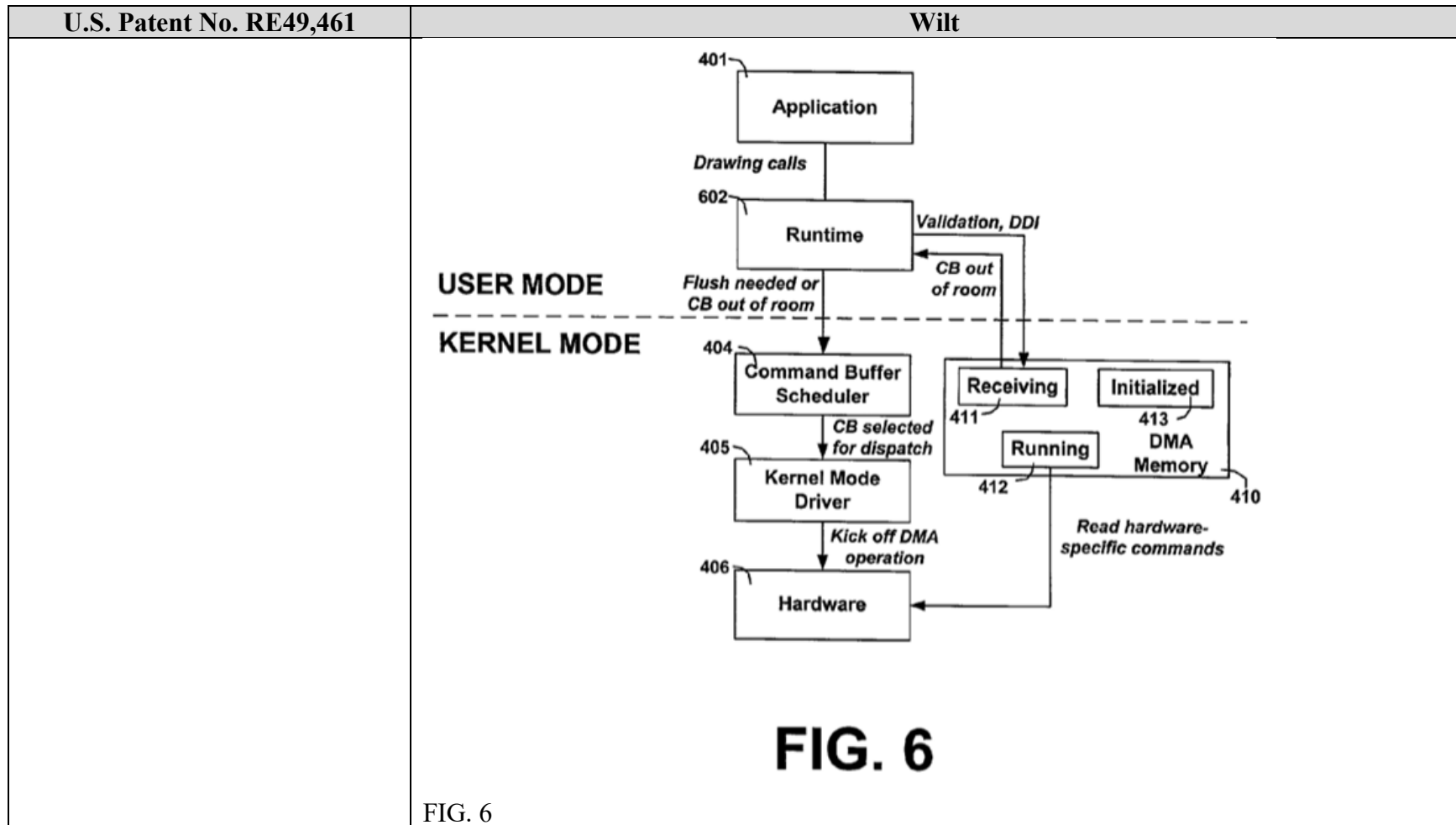
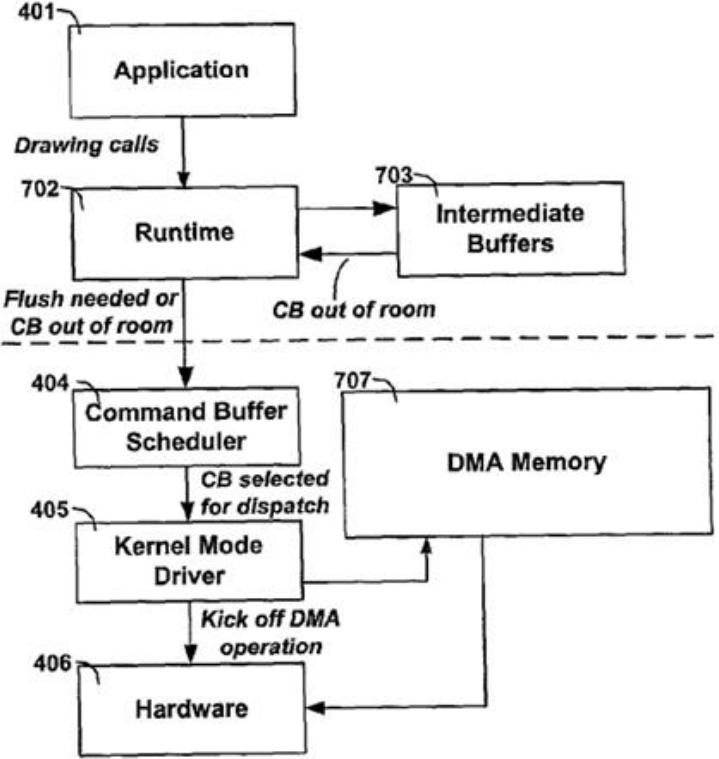


FIG. 5

Ex. C9 – Invalidity of U.S. Patent No. RE49,461 – Wilt



U.S. Patent No. RE49,461	Wilt
	 FIG. 7 FIG. 7. Systems and methods are provided for managing the computational resources of coprocessor(s), such as graphics processor(s), in a computing system. The systems and methods illustrate management of computational resources of coprocessors to facilitate efficient execution of multiple applications in a multitasking environment. By enabling

Ex. C9 – Invalidity of U.S. Patent No. RE49,461 – Wilt

U.S. Patent No. RE49,461	Wilt
	multiple threads of execution to compose command buffers in parallel, submitting those command buffers for scheduling and dispatch by the operating system, and fielding interrupts that notify of completion of command buffers, the system enables multiple applications to efficiently share the computational resources available in the system. Abstract The present invention relates to systems and methods for managing computational resources of coprocessors in a computing system. [0002] Beginning with the first time-sharing system in the mid-1960s, operating systems (OSs) have implemented numerous methods of allowing multiple applications to share computational resources of a computer without knowledge of one another. By allocating small ‘time slices’ to each application, and interrupting when a ‘time slice’ has expired, a computer can present each application with the illusion that it is running alone on the computer. For example, two applications could be running on a system with 1 millisecond time slices. In such a case, each application would run somewhat less than half as fast (due to the overhead needed to swap between the two) than if they were running on the computer alone, each getting about 500 time slices per second. Longer time slices involve less overhead, but also result in a coarser granularity of execution, making the system less suitable for timing-sensitive applications. An enormous amount of work has gone into developing various abstractions such as virtual memory, processes, and threads that interact to provide applications with software models that enable the computational resources of the central processing unit (CPU) to be shared. However, these abstractions have not yet been augmented so that they can apply to the management of computational resources in graphics processing units (GPUs) as well as host microprocessors. In this regard, in the last few years, graphics processors have become significantly more functional. The number of transistors in PC graphics chips has grown far faster than Moore’s Law would suggest, i.e., the number of transistors in graphics chips has grown from about 200,000 in 1995 to about 60,000,000 transistors in 2001. The computational power of these chips has also increased with the number of transistors; that is, not only can graphics chips process more data, but they can also apply more sophisticated computations to the data as well.

Ex. C9 – Invalidity of U.S. Patent No. RE49,461 – Wilt

U.S. Patent No. RE49,461	Wilt
	As a result, today, the graphics chip(s) in a computer system can be considered a computational resource that complements the computational resources of the host microprocessor(s). The software model presented by a graphics chip is somewhat different than the software model presented by the host microprocessor. Both models involve context, a set of data that describes exactly what the processor is doing. The contexts may contain data registers, which contain intermediate results of whatever operation is currently being performed, or control registers, which change the processor's behavior when it performs certain operations. On a 32-bit INTEL® processor, for example, the EAX data register is used as an accumulator, to perform multiplications, to hold function return values, and so on. The floating point control word (FPCW) is a control register that controls how floating point instructions round inexact results (single, double, or extended precision, toward positive or negative infinity, toward zero, or toward the nearest; and so on). As a general rule, however, graphics processors have a great deal more state in control registers than general-purpose microprocessors. Graphics processors' high performance stems from their pipelined, flexible, yet fixed function architecture. A great deal of control register state is needed to set up the operations performed by the graphics processor. For example, a set of control registers may include (a) the base address(es) and dimensions of one or more texture maps currently serving as input, (b) the texture addressing and filtering modes, the blending operation to perform between texture values and interpolated color values, (c) the tests to apply to the alpha and Z values of the fragment to decide whether to incorporate it into the color buffer and (d) the alpha blend operation to use when incorporating the color fragment into the color buffer at the final rendering stage. While graphics processors contain numerous scratch registers such as iterators that control their processing, generally it is not necessary to save those registers during context switches because context switches are not permitted on a granularity that requires them to be saved. In any case, even if such registers must be saved during a context switch, generally they are not directly available to software applications. The opacity of volatile register state to client software is merely one distinction between the software model presented by graphics processors, as compared to the software model presented by general purpose microprocessors. To date, attempts to manage the computational resources of coprocessors, such as graphics processors, have been ad hoc at best. Historically, there has not been much demand for careful management of these computational resources because only one application has been active at

Ex. C9 – Invalidity of U.S. Patent No. RE49,461 – Wilt

U.S. Patent No. RE49,461	Wilt
	a time. In the context of the commercial workstation applications that 3D acceleration hardware initially was designed to accelerate, such as 3D modeling and animation, end users typically would operate one application at a time. Even if more than one application were active at a given time, the end user would perform a significant amount of work on each application before switching to another and the granularity of switching between applications was on the order of seconds or much longer. Game applications, the second set of applications to substantially benefit from graphics hardware acceleration, also are typically run one at a time. In fact, the DIRECTX® application programming interfaces (APIs) in WINDOWS® specifically enable game applications to gain exclusive access to the hardware resources in the computer system and particularly the graphics chip. As graphics chips become more functional, it is reasonable to expect the number of active applications that demand significant computational resources from them to increase, and for the granularity of switching between these applications to become finer. In some areas, this trend is already evident. For example, video decoding acceleration such as hardware-accelerated motion compensation (“mocomp”) and integer discrete cosine transform (“IDCT”) has been added to most graphics chips in the 2001 timeframe. Since it is possible to launch a video playback application and run other applications at the same time, playing back video and running any other application that demands computational resources from the graphics processor will require careful management of those resources, to ensure that the video playback and other application(s) both deliver a high quality end user experience. Other potential sources of increased demand for graphics processors' computational resources include the composition of multiple applications' output, and improved utilization of hardware acceleration by 2D graphics APIs such as GDI (graphical developer interface) or GDI+. In short, the need for efficient and effective management of computational resources of graphics processor(s) in a computing system will only be increasing, along with the increasing power, flexibility and speed of the graphic processors themselves and along with increasing number of applications making simultaneous use of the computational resources of the graphics processors. [0003]–[0009] In view of the foregoing, the present invention provides systems and methods for managing the computational resources of coprocessor(s), such as graphics processor(s), in a computing

Ex. C9 – Invalidity of U.S. Patent No. RE49,461 – Wilt

U.S. Patent No. RE49,461	Wilt
	system. The systems and methods illustrate management of computational resources of coprocessors to facilitate efficient execution of multiple applications in a multitasking environment. By enabling multiple threads of execution to compose command buffers in parallel, submitting those command buffers for scheduling and dispatch by the operating system, and fielding interrupts that notify of completion of command buffers, the system enables multiple applications to efficiently share the computational resources available in the system. In various embodiments, the present invention provides methods for controlling the computational resources of coprocessor(s) in a host computing system having a host processor, wherein the methods include controlling one or more coprocessors of the computing system with command buffers submitted to the coprocessor(s) by the host processor, transmitting data back to the host computing system in response to commands in the command buffer(s) and scheduling the transmission of the command buffer(s) by a managing object included in the host computing system, wherein the computational resources of the coprocessor(s) are simultaneously available to multiple applications instantiated on the host computing system. In related and other embodiments, the invention provides a computing device and computer readable media having stored thereon a plurality of computer-executable modules for controlling the computational resources of coprocessor(s) in a host computing system having a host processor, the computing device and computer executable modules each including a managing object for controlling the coprocessor(s) of the computing system with command buffers submitted to the coprocessor(s) by the host processor and for scheduling the transmission of the command buffers. The computing device and computer executable modules also each include means for transmitting data back to the host computing system in response to commands in the command buffer(s), whereby the computational resources of the coprocessor(s) are simultaneously available to multiple applications instantiated on the host computing system. [0010]–[0012] The system and methods for managing the computational resources of graphics processor(s) are further described with reference to the accompanying drawings in which: FIG. 1A is a block diagram representing an exemplary network environment having a variety of computing devices in which the present invention may be implemented;

Ex. C9 – Invalidity of U.S. Patent No. RE49,461 – Wilt

U.S. Patent No. RE49,461	Wilt
	FIG. 1B is a block diagram representing an exemplary nonlimiting computing device in which the present invention may be implemented; FIG. 2 illustrates an exemplary life cycle of a command buffer as it is used by applications to send commands to a coprocessor; FIGS. 3A through 3C illustrate exemplary implementations of a driver component that translates runtime-level commands to hardware-specific commands; FIG. 4 illustrates an exemplary user mode driver DLL implementation of the invention; FIG. 5 illustrates an exemplary sequence of events that occur when an application makes application programming interface calls to perform graphics operations in exemplary detail; FIG. 6 illustrates an exemplary hardware consumable canonical command buffer format in accordance with an embodiment of the invention; and FIG. 7 illustrates an exemplary intermediate canonical command buffer format in accordance with an embodiment of the invention. [0014]–[0022] As described in the background, there are many present and potential sources of increased demand for graphics processors' computational resources. When more than one of these sources demands access to a graphics processor at approximately the same time, the graphics processor may not be free to serve both sources in a way that meets the demand satisfactorily. Thus, the present invention provides various ways to manage the scheduling and dispatch of graphics processing requests by multiple applications in order to ensure the computational resources of coprocessor units, such as graphics processing units, are efficiently distributed to the requesting applications. Accordingly, the present invention provides methods and systems for controlling the computation resources of coprocessor(s) in a host computing system, wherein the coprocessor(s) are controlled with command buffers submitted by the host processor and wherein the scheduling of the transmission of the command buffers to the coprocessor(s) is performed by an object, such as the operating system, of the host computing system, making the computational resources of the coprocessor(s) simultaneously available to multiple applications. [0024]

Ex. C9 – Invalidity of U.S. Patent No. RE49,461 – Wilt

U.S. Patent No. RE49,461	Wilt
	An accelerated graphics port (AGP) is a high speed bus designed to facilitate the rapid transmission of data from a host microprocessor to a graphics peripheral. A command buffer is a stream of hardware-specific drawing commands ready for consumption by graphics hardware, e.g., see the definition for token stream below. A context is the hardware state utilized to execute commands by a processor. Herein, a thread context refers to the hardware state of a host microprocessor and a hardware context refers to the state of graphics hardware. A context is loaded before a stream of instructions (on the host) or commands (on a graphics chip) can be executed. A central processing unit (CPU) includes a host microprocessor. A display list is a series of commands for the graphics hardware that has been recorded so the commands can be readily ‘played back,’ purportedly more efficiently than if the original series of commands were executed again. Display lists often are opaque to the application, i.e., they cannot be edited, since they have been translated into a hardware-specific form. Direct memory access (DMA) is a hardware feature that enables a peripheral to read or write directly from host-accessible memory. A graphics processor is a hardware peripheral designed to accelerate drawing commands for graphics, such as bit block transfers (BLTs) and 3D graphics primitive drawing processes such as triangle rasterization. In this regard, a graphics processor may be contrasted with the more general and clumsy resources of the host microprocessor. A host microprocessor is a processor that executes the operating system and/or applications being run in the operating system. The host microprocessor(s) in a system contrast with graphics processors or other computational resources in the system. An interrupt is a hardware feature that enables a peripheral to divert a host microprocessor from its current process and transfer control to a special location in memory. The operating system uses this mechanism to call driver code known as an interrupt service routine (ISR) to respond to the interrupt. An interrupt service routine (ISR) is a function, typically in a device driver, that is called by the operating system when the hardware controlled by the driver signals an interrupt, e.g., see also DPC. Just-in-time (JIT) compiling is a compilation process that introduces an extra step in the translation from source code to object code. In this regard, the source code is translated into

Ex. C9 – Invalidity of U.S. Patent No. RE49,461 – Wilt

U.S. Patent No. RE49,461	Wilt
	readily compilable, hardware-independent intermediate language that is transmitted to a client computer, where it is compiled “just in time” into object code that can run on the client. A process is a logical division of labor in an operating system. In WINDOWS®, a process comprises a virtual address space, an executable program, one or more threads of execution, some portion of the user's resource quotas, and the system resources that the operating system has allocated to the process's threads. A resource is a synchronization primitive that can grant non-exclusive access to multiple threads, or exclusive access to a single thread. A synchronization primitive is an object that can be used to synchronize multiple threads' access to shared resources, such as critical sections, mutexes, semaphores or events. A thread is an executable entity that comprises a program counter, a user-mode stack, a kernel-mode stack and a set of register values. A token stream is a stream of hardware-independent tokens that describe a series of drawing operations. A token stream can be translated by a hardware-specific software component, such as a driver, into a command buffer or other series of hardware commands. [0027]–[0030], [0032], [0034], [0036]–[0040], [0042]–[0043], [0044]–[0046] Direct Memory Access (DMA) is a common method for moving data within computer systems, and has found application in the control of graphics processors as well as disk drives, network adapters, and external buses such as USB and IEEE 1394. DMA generally refers to the ability of a peripheral, such as a digital camera or disk controller, to read from or write to memory without directly involving the host processor. The host sets up the DMA transfer in advance, page locking the memory region where a digital camera, for example, is to write the image data from a picture it is about to take, and specifying that memory location to the camera. Page locking causes the virtual memory manager to forego writing the memory locations in question out to disk. Once this is done, the camera can take the picture and write the image data into memory, then notify the host that the DMA transfer has been completed. In the context of graphics processor control, a series of commands for the graphics processor can be written into memory, and then transferred to the graphics processor via a DMA operation. DMA has many advantages over the other methods of submitting graphics processor commands. First, it facilitates parallel execution by the host and graphics processors. Since these ‘other operations’ to be performed by the CPU often involve generating more commands

Ex. C9 – Invalidity of U.S. Patent No. RE49,461 – Wilt

U.S. Patent No. RE49,461	Wilt
	for the graphics processor, it is common to maintain multiple DMA buffers and ‘ping-pong’ between them, writing to one while the graphics processor reads and executes another. Second, DMA enables multiple applications to write command buffers into memory, and then submit the contents of the command buffers to the graphics processor when they are ready for execution. The graphics processor can only execute one command buffer at a time, but DMA facilitates the simultaneous construction of many command buffers because different threads can write into distinct memory regions that contain command buffers. Third, DMA frequently results in more efficient usage of the bus resources in the system. In other words, the bandwidth achievable through DMA is often higher than the bandwidth achievable by direct participation of the host microprocessor. The Accelerated Graphics Port (AGP) is a specialized bus designed specifically to serve as a high bandwidth conduit to transfer data from system memory to a graphics processor. For the purposes of the present description, AGP may be considered a special case of DMA. Besides the mechanism for delivering information to and receiving information from the hardware, another important distinction to make in our background discussion centers on notification of the host microprocessor that the graphics processor has completed some task, or that some event has occurred that is visible to the graphics processor. Non-limiting examples of tasks include finishing execution of a command buffer or the completion of a specific command. Non-limiting examples of events include vertical retrace of the display device, the completion of a field of digitized video from a video port, or the retracing of a specific scanline in the display device. An alternative method for notifying the host microprocessor of a completed task or an external event is known as an interrupt. An interrupt is a hardware mechanism that diverts the host microprocessor from whatever it is doing and transfers control to a special location in memory. The operating system uses this mechanism to call driver code known as an interrupt service routine to respond to the interrupt. Interrupts have several advantages over polling based mechanisms. For one thing, the cost of setting up and responding to an interrupt is fixed, while the cost of a polling based mechanism increases approximately linearly with the number of polls performed. As a result, interrupt based schemes generally consume less bus bandwidth as well as fewer CPU cycles than polling based schemes. Interrupts can also result in a more timely notification if the interrupt

Ex. C9 – Invalidity of U.S. Patent No. RE49,461 – Wilt

U.S. Patent No. RE49,461	Wilt
	latency is smaller than the time between when the event occurs and when the next poll is performed. [0085]–[0088], [0090]–[0091]. Given these assumptions, we can broaden our focus from graphics processors and contemplate a system with two processors: a host CPU, and a coprocessor that is controlled by sending command buffers from the CPU, that can send data back to the CPU, and that coordinates control by notifying the CPU of various events when they occur. The examples in this document frequently assume that the coprocessor is a graphics processor (as when describing possible synchronization problems and how a system can be designed to solve those problems efficiently), but these examples are for illustrative purposes only and do not limit the scope of this invention to graphics processors. In line with the above assumptions regarding computational resources in a graphics processing system, a non-limiting system is described below that makes efficient, shared use of the computational resources in a coprocessor. FIG. 2 depicts the life cycle of a command buffer as it is used by applications to send commands to the coprocessor, such as a graphics processor. When a command buffer is created and initialized, the buffer enters the initialized state 201. The command buffer is then available to be selected and written into by a thread that wishes to issue commands to the graphics processor. When a thread selects the command buffer, the memory in the command buffer is made available to that thread (perhaps by mapping the memory in the command buffer into that thread's address space) and the command buffer enters the Receiving state 202. As one of ordinary skill in the art can appreciate, the details of this mapping may vary. For example, the command buffer memory could be made visible to user mode, kernel mode, or both. In addition, there are numerous options as to the nature of the executable code that is writing into the Receiving command buffer. In this regard, FIGS. 3A through 3E depict several of these options, i.e., several different architectures for writing into the command buffer. For purposes of describing FIGS. 3A through 3E, an application is a process, or a thread in a process, that implements some specific graphical application participating in the system. The runtime is middleware provided in the operating system that the application uses. Among other things, the runtime may validate the application's usage of an application programming

Ex. C9 – Invalidity of U.S. Patent No. RE49,461 – Wilt

U.S. Patent No. RE49,461	Wilt
	interface (API), translate simple-to-use, high-level API calls into more complicated, lower-level constructs that are more readily translatable to hardware commands, and so on. The driver is a hardware-specific component that translates runtime-level commands to hardware-specific commands. The driver components depicted in FIGS. 3A through 3E are the portions of the driver that translate runtime-level commands to hardware-specific commands. Some portions of the driver that perform tasks other than this translation, such as interrupt handling and command buffer dispatch, would probably reside in kernel mode and are not contemplated in FIGS. 3A through 3E. Most of the object code depicted in FIGS. 3A through 3E may run in user mode, although some portions, as pointed out below, can run in kernel mode. The command buffer is the piece of memory that includes commands that can be consumed directly by the hardware via some DMA mechanism. Several of the figures also show a token stream, which is a hardware-independent stream of runtime-level primitives that can be translated into hardware-specific commands by the driver. Lastly, some of the figures show a JIT (Just-In-Time) compilation component that can translate a hardware-independent intermediate language into object code for execution by the host microprocessor. The boxes with OBJ in the corner include object code that can be directly executed by the microprocessor. Boxes with MSIL in the corner include intermediate code that is compiled by the JIT in order to execute on the microprocessor. If the application and runtime are both available on the client in intermediate form, the JIT can translate them into a unified piece of object code, shown as app executables 324, 334 and 345 FIGS. 3C to 3E, respectively, that implements the application. FIG. 3A shows the application 301, runtime 302 and driver 303 each performing their respective roles, calling into lower-level object code until the driver 303 writes hardware-specific commands into the command buffer 304. The application 301 could be an executable process, or a thread in a process, while the runtime 302 and driver 303 could each reside in DLLs. As depicted in FIG. 3A, all of the object code may reside in user mode (although some or all of the runtime 302 and/or driver 303 may reside in kernel mode). If the runtime 302 determines that the command buffer 304 must be submitted before execution can continue, or the driver 303 determines that the command buffer 304 is out of space, a transition to kernel mode may be requested to submit the command buffer 304 and retrieve another one in order to continue.

Ex. C9 – Invalidity of U.S. Patent No. RE49,461 – Wilt

U.S. Patent No. RE49,461	Wilt
	FIG. 3B shows a variant in which the runtime 312 emits a hardware-independent stream of commands into a buffer 313, which is then parsed by the driver 314 and written into a command buffer 315. Application 311 operates to call lower-level object code in connection with runtime 312. FIG. 3B approximates the current architecture of the DIRECTX® software component, in which the runtime translates application activity into DrawPrimitives2 (“DP2”) tokens. When the DP2 token stream is submitted, a kernel transition occurs and the driver 314 translates the DP2 token stream into hardware-specific commands in kernel mode. FIG. 3B does not make any assumptions about whether the driver 314 is in user mode or kernel mode, and in this regard, driver component 314 could reside in user mode as well as kernel mode. FIG. 3C shows a variant in which the application 321 and runtime 322 are synthesized into a unified piece of object code by a JIT compiler 323. The resulting piece of object code 324 then interacts with the driver 325 to write hardware-specific commands into the command buffer 326. FIG. 3D shows the JIT variant of the situation described for FIG. 3B, in which a hardware-independent token stream 335 is emitted by the software 334 above the driver 336. The driver 336 then translates the token stream to hardware-specific commands for the command buffer 337. As with FIG. 3C, application 331 and runtime 332 are synthesized into a unified piece of object code 334 by JIT compiler 333. FIG. 3E shows a system in which the application 341, runtime 342, and part of the driver 343 are all available on the client in intermediate language form MSIL. The JIT compiler 344 synthesizes these components into a unified piece of object code 345 specifically optimized to run on that particular client computer, taking into account not only the host microprocessor but also the coprocessor hardware configuration. Referring back to the command buffer life cycle depicted in FIG. 2, regardless of the method by which commands are written into the command buffer, the thread writing into the receiving command buffer eventually submits to the command buffer either because the command buffer has been filled, or because the drawing commands that have been written into the buffer are executed forthwith. At that time, the command buffer enters the waiting state 203, and the contents of the command buffer may not be submitted to the hardware until all of the resources utilized to execute the command buffer are available. As an example of a circumstance that may force a wait, if an application draws into a surface, the drawing must be completed before that surface may be selected as a texture.

Ex. C9 – Invalidity of U.S. Patent No. RE49,461 – Wilt

U.S. Patent No. RE49,461	Wilt
	Once the waiting criteria have been fulfilled, the command buffer enters the ready state 204, where the scheduler considers it for submission to the coprocessor. The scheduler has a great deal of flexibility in deciding which command buffer should be submitted to the hardware. The scheduler may take into account the priority, if any, of the command buffer, the priority of the requesting thread, the real-time requirements of the system, and so on. Once the command buffer has been selected as the next to run, the command buffer enters the standby state 205. On some systems, only one command buffer per coprocessor may be in the standby state 205 in the system. When the command buffer in the standby state 205 can be submitted to the hardware, the system switches the coprocessor context to the context required by the command buffer, and then submits the command buffer to the hardware. This operation entails some involvement from the driver, for example to initiate a DMA operation using the base address and size of the command buffer. The command buffer remains in the running state 206 until some event preempts execution of the command buffer or the entire contents of the command buffer have been read and executed. If the entire command buffer has been executed, the command buffer enters the terminated state 207, where it becomes available for the system to reinitialize and make available to another thread in the initialized state 201. If the command buffer is preempted, it is assumed that the hardware has the capability to save the hardware context in such a way that execution of the command buffer can be resumed where the preemption occurred. The command buffer can then be recycled into the ready state 204 and be re-submitted to the hardware to resume from where it stopped. The description below regarding preemptible command buffer execution describes this in more detail. [0095]–[0110] With respect to queueing of command buffers, due to the high performance of graphics hardware, it is critical to minimize the latency between command buffer completion and the submission of another command buffer. Otherwise, the graphics hardware wastes valuable time waiting for another command buffer to be submitted. One possible solution to this problem is to queue up a context switch and command buffer dispatch before the current command buffer has finished executing. Then, provided the hardware and driver support this capability, the hardware can immediately begin execution of another command buffer when

Ex. C9 – Invalidity of U.S. Patent No. RE49,461 – Wilt

U.S. Patent No. RE49,461	Wilt
	the current command buffer has completed. The system then has more time to prepare another command buffer for execution: the graphics hardware will only go idle if it completes the new command buffer before another one has been queued up. With respect to preemptible command buffer execution, the system described here provides for hardware support for preemption. A variety of events could potentially cause preemption of a command buffer during execution, ranging from an external event such as VSYNC to hardware enforcement of a time quantum. To implement this feature, the hardware and driver would have to enable the hardware context to be saved, such that execution of the preempted command buffer can be resumed where it left off; and the operating system would have to be notified so it could put the command buffer back in the Ready state mentioned in FIG. 2. One reason to implement such preemption is to guarantee that a display frame is composed immediately after VSYNC (to make sure it will be ready when the next VSYNC occurs). Hardware enforcement of a time quantum would cause a command buffer to be preempted after executing for some set amount of time, such as 2 milliseconds. If the command buffer contains very expensive commands, such as drawing large, texture-mapped, alpha blended triangles, the hardware may have to preempt its execution to enable other command buffers to be considered for submission. [0126]–[0127] To the extent Plaintiff contends that this claim element is not expressly or inherently disclosed, or is not obvious in view of Wilt and the knowledge of a POSITA, it would have been obvious to combine the teachings of Wilt with any of the prior art references in NVIDIA’s Invalidity Contentions and any supplements thereto and the relevant section of charts for other prior art for the ’461 patent in a manner that would result in the subject matter of this limitation given, at the very least, the person's understanding of the state of the art, the problems addressed and solved in the prior art, and the teachings of Wilt. Specific examples of such references are cited and explained in these invalidity contentions, including the cover pleading and charts C1–C15, and Appendix C.
21[a]: executing, by the at least one CPU, a user interaction stream, the user interaction stream controlling transfer of inputs to the	Wilt discloses “executing, by the at least one CPU, a user interaction stream, the user interaction stream controlling transfer of inputs to the artificial neural network to the first memory partition and the second memory partition.” See <i>e.g.</i> :

Ex. C9 – Invalidity of U.S. Patent No. RE49,461 – Wilt

U.S. Patent No. RE49,461	Wilt
	storage devices, or any other medium which can be used to store the desired information and which can accessed by computer 110. Communication media typically embodies computer readable instructions, data structures, program modules or other data in a modulated data signal such as a carrier wave or other transport mechanism and includes any information delivery media. The term “modulated data signal” means a signal that has one or more of its characteristics set or changed in such a manner as to encode information in the signal. By way of example, and not limitation, communication media includes wired media such as a wired network or direct-wired connection, and wireless media such as acoustic, RF, infrared and other wireless media. Combinations of any of the above should also be included within the scope of computer readable media. The system memory 130 includes computer storage media in the form of volatile and/or nonvolatile memory such as read only memory (ROM) 131 and random access memory (RAM) 132. A basic input/output system 133 (BIOS), containing the basic routines that help to transfer information between elements within computer 110, such as during start-up, is typically stored in ROM 131. RAM 132 typically contains data and/or program modules that are immediately accessible to and/or presently being operated on by processing unit 120. By way of example, and not limitation, FIG. 1B illustrates operating system 134, application programs 135, other program modules 136, and program data 137. The computer 110 may also include other removable/non-removable, volatile/nonvolatile computer storage media. By way of example only, FIG. 1B illustrates a hard disk drive 141 that reads from or writes to non-removable, nonvolatile magnetic media, a magnetic disk drive 151 that reads from or writes to a removable, nonvolatile magnetic disk 152, and an optical disk drive 155 that reads from or writes to a removable, nonvolatile optical disk 156, such as a CD ROM or other optical media. Other removable/non-removable, volatile/nonvolatile computer storage media that can be used in the exemplary operating environment include, but are not limited to, magnetic tape cassettes, flash memory cards, digital versatile disks, digital video tape, solid state RAM, solid state ROM, and the like. The hard disk drive 141 is typically connected to the system bus 121 through an non-removable memory interface such as interface 140, and magnetic disk drive 151 and optical disk drive 155 are typically connected to the system bus 121 by a removable memory interface, such as interface 150. The drives and their associated computer storage media discussed above and illustrated in FIG. 1B provide storage of computer readable instructions, data structures, program modules and

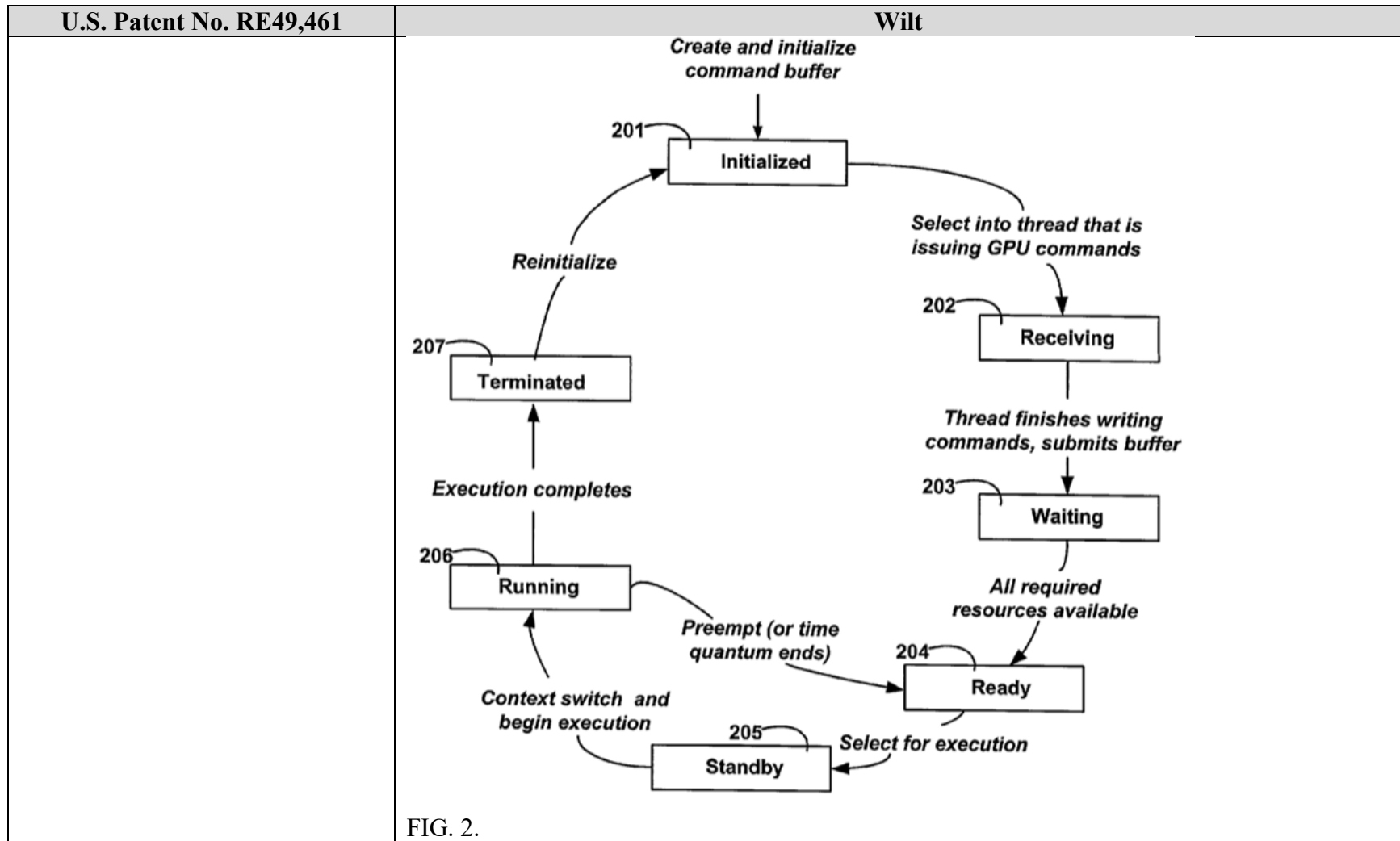
Ex. C9 – Invalidity of U.S. Patent No. RE49,461 – Wilt

U.S. Patent No. RE49,461	Wilt
	other data for the computer 110. In FIG. 1B, for example, hard disk drive 141 is illustrated as storing operating system 144, application programs 145, other program modules 146, and program data 147. Note that these components can either be the same as or different from operating system 134, application programs 135, other program modules 136, and program data 137. Operating system 144, application programs 145, other program modules 146, and program data 147 are given different numbers here to illustrate that, at a minimum, they are different copies. A user may enter commands and information into the computer 110 through input devices such as a keyboard 162 and pointing device 161, commonly referred to as a mouse, trackball or touch pad. Other input devices (not shown) may include a microphone, joystick, game pad, satellite dish, scanner, or the like. These and other input devices are often connected to the processing unit 120 through a user input interface 160 that is coupled to the system bus 121, but may be connected by other interface and bus structures, such as a parallel port, game port or a universal serial bus (USB). A graphics interface 182, such as Northbridge, may also be connected to the system bus 121. Northbridge is a chipset that communicates with the CPU, or host processing unit 120, and assumes responsibility for AGP communications. One or more graphics processing units (GPUs) 184 may communicate with graphics interface 182. In this regard, GPUs 184 generally include on-chip memory storage, such as register storage and GPUs 184 communicate with a video memory 186. The invention may be described more generally in connection with any coprocessor, of which GPUs 184 are but one example. A monitor 191 or other type of display device is also connected to the system bus 121 via an interface, such as a video interface 190, which may in turn communicate with video memory 186. In addition to monitor 191, computers may also include other peripheral output devices such as speakers 197 and printer 196, which may be connected through an output peripheral interface 195. [0068]–[0071] To the extent Plaintiff contends that this claim element is not expressly or inherently disclosed, or is not obvious in view of Wilt and the knowledge of a POSITA, it would have been obvious to combine the teachings of Wilt with any of the prior art references in NVIDIA’s Invalidity Contentions and any supplements thereto and the relevant section of charts for other prior art for the ’461 patent in a manner that would result in the subject matter of this limitation given, at the very least, the person's understanding of the state of the art, the problems addressed and

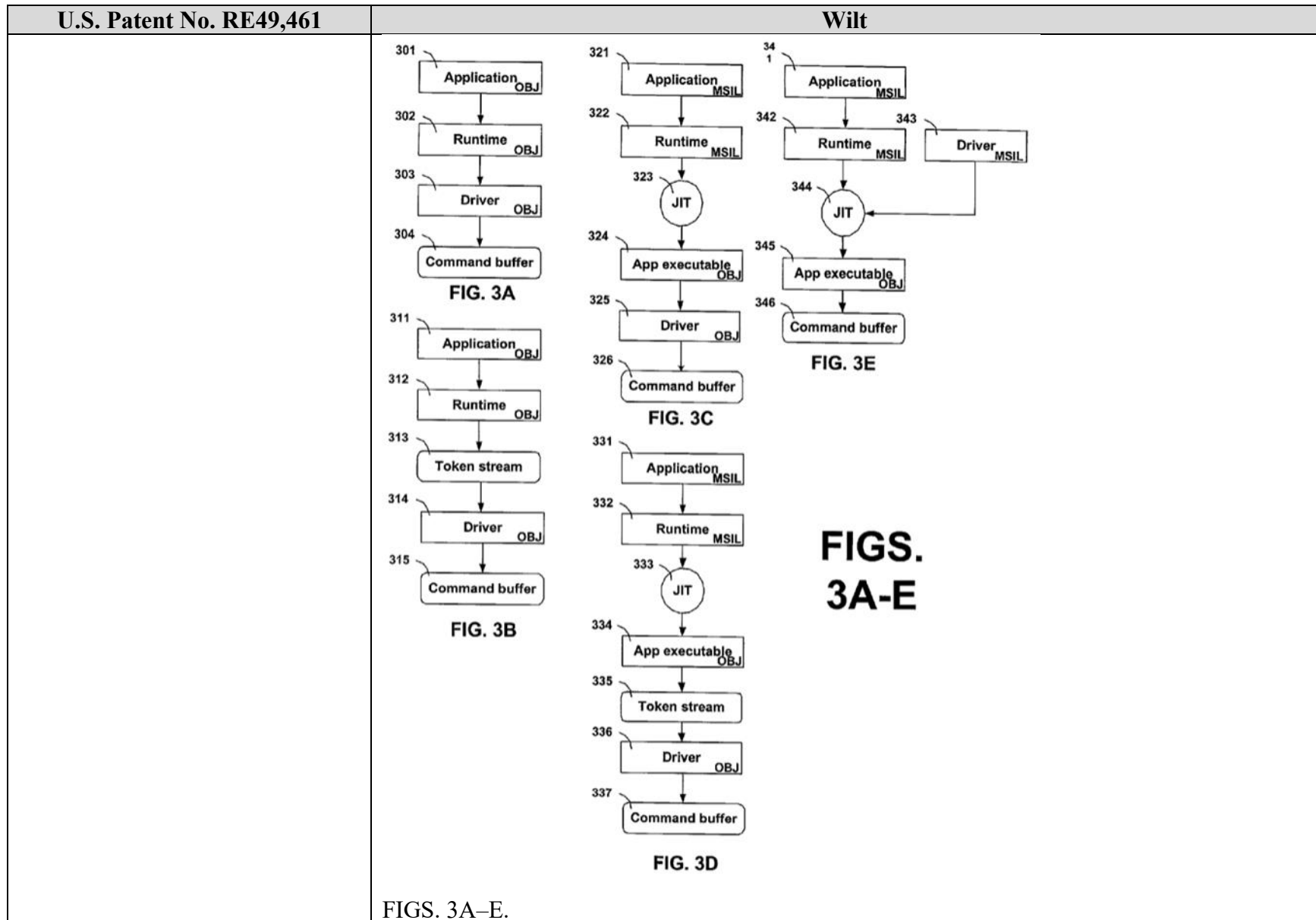
Ex. C9 – Invalidity of U.S. Patent No. RE49,461 – Wilt

U.S. Patent No. RE49,461	Wilt
	solved in the prior art, and the teachings of Wilt. Specific examples of such references are cited and explained in these invalidity contentions, including the cover pleading and charts C1–C15, and Appendix C.
21[b]: executing, by the processing unit, a computational stream, the computational stream controlling data exchange between the user interaction stream and the computational stream during execution of the computations representing the artificial neural network;	Wilt discloses executing, by the processing unit, a computational stream, the computational stream controlling data exchange between the user interaction stream and the computational stream during execution of the computations representing the artificial neural network. <i>See e.g.:</i>

Ex. C9 – Invalidity of U.S. Patent No. RE49,461 – Wilt



Ex. C9 – Invalidity of U.S. Patent No. RE49,461 – Wilt



Ex. C9 – Invalidity of U.S. Patent No. RE49,461 – Wilt

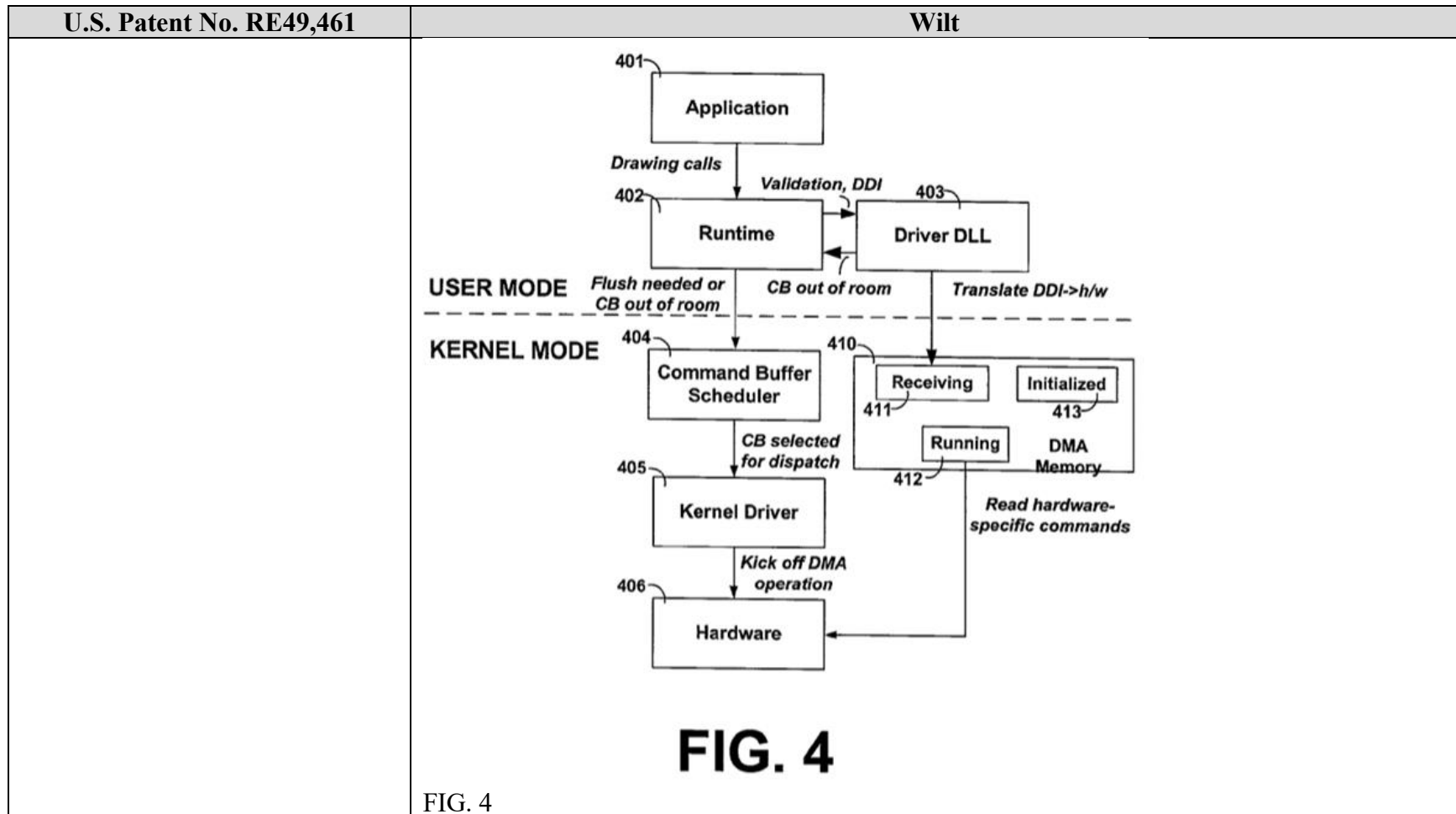


FIG. 4

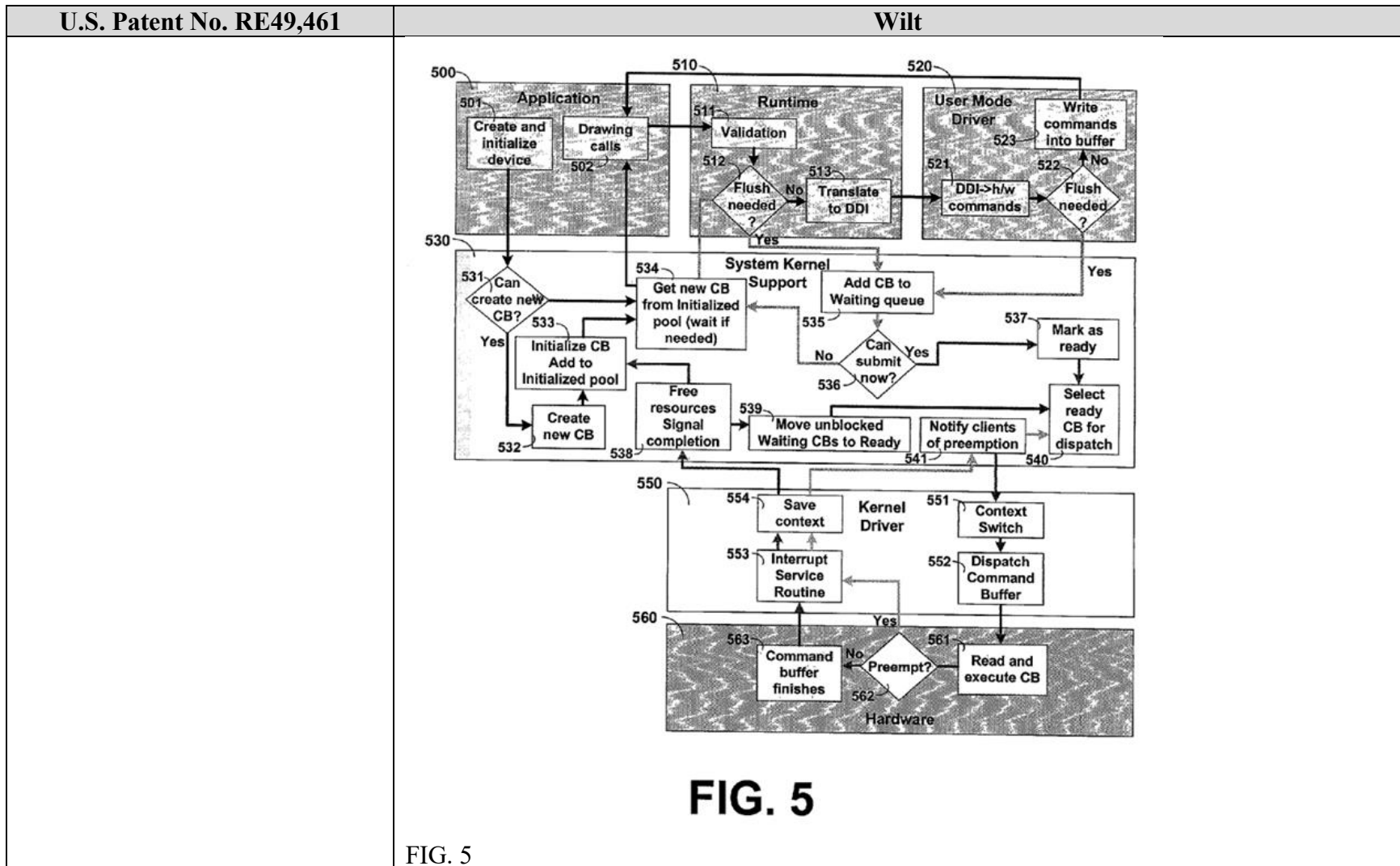


FIG. 5

Ex. C9 – Invalidity of U.S. Patent No. RE49,461 – Wilt

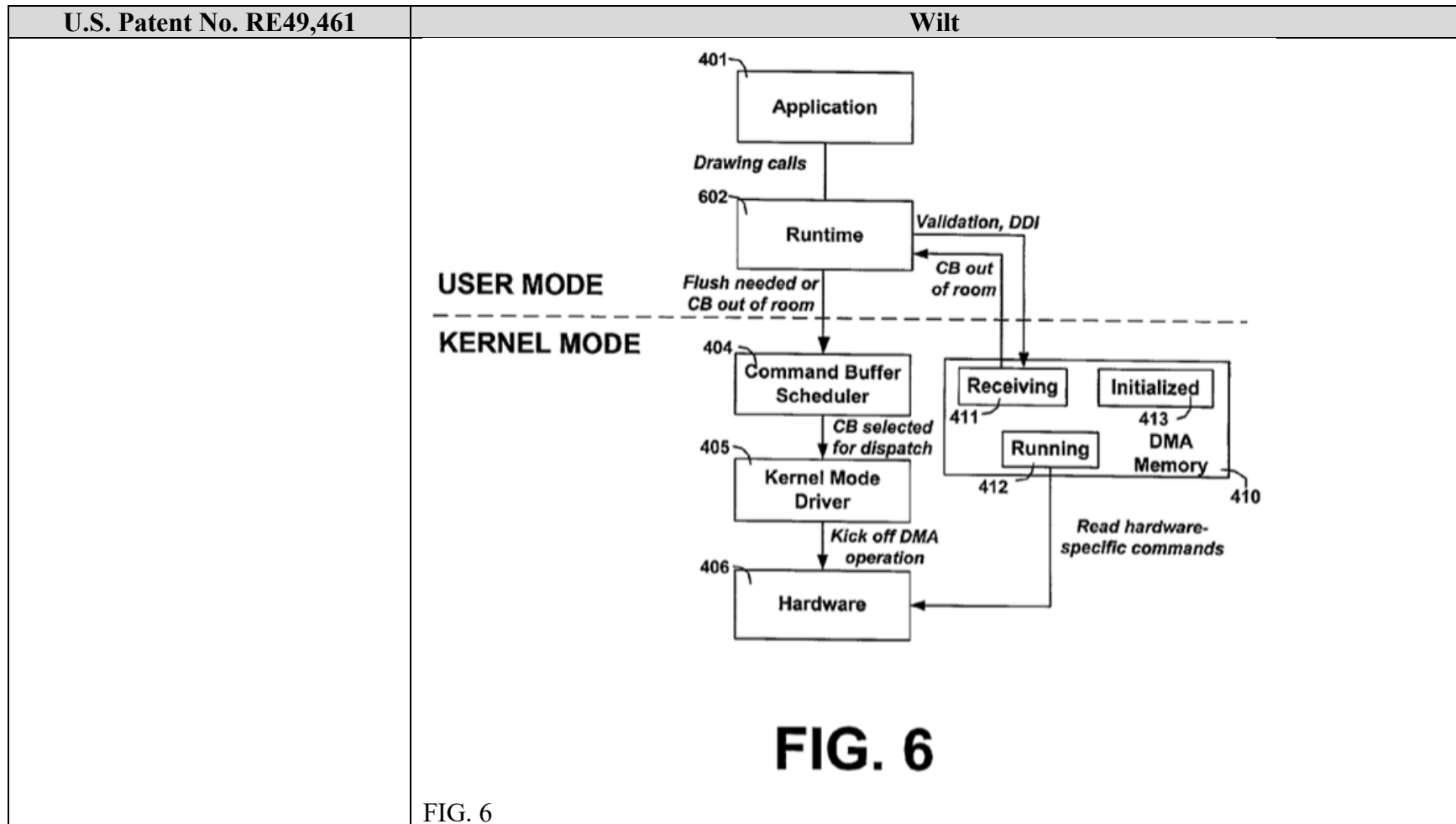
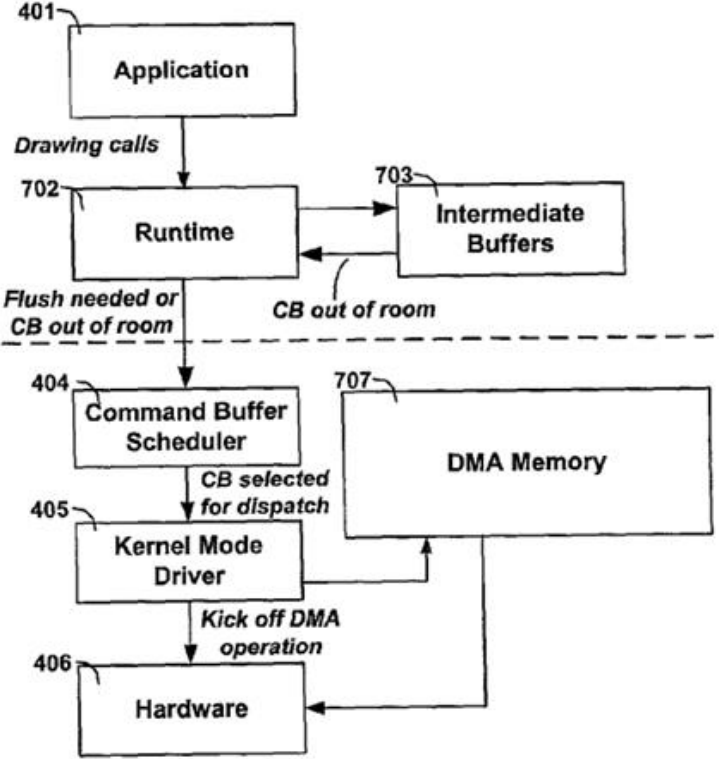


FIG. 6

U.S. Patent No. RE49,461	Wilt
	 FIG. 7 FIG. 7. A command buffer is a stream of hardware-specific drawing commands ready for consumption by graphics hardware, e.g., see the definition for token stream below.

Ex. C9 – Invalidity of U.S. Patent No. RE49,461 – Wilt

U.S. Patent No. RE49,461	Wilt
	A token stream is a stream of hardware-independent tokens that describe a series of drawing operations. A token stream can be translated by a hardware-specific software component, such as a driver, into a command buffer or other series of hardware commands. [0027]–[0046] Direct Memory Access (DMA) is a common method for moving data within computer systems, and has found application in the control of graphics processors as well as disk drives, network adapters, and external buses such as USB and IEEE 1394. DMA generally refers to the ability of a peripheral, such as a digital camera or disk controller, to read from or write to memory without directly involving the host processor. The host sets up the DMA transfer in advance, page locking the memory region where a digital camera, for example, is to write the image data from a picture it is about to take, and specifying that memory location to the camera. Page locking causes the virtual memory manager to forego writing the memory locations in question out to disk. Once this is done, the camera can take the picture and write the image data into memory, then notify the host that the DMA transfer has been completed. In the context of graphics processor control, a series of commands for the graphics processor can be written into memory, and then transferred to the graphics processor via a DMA operation. DMA has many advantages over the other methods of submitting graphics processor commands. First, it facilitates parallel execution by the host and graphics processors. Since these ‘other operations’ to be performed by the CPU often involve generating more commands for the graphics processor, it is common to maintain multiple DMA buffers and ‘ping-pong’ between them, writing to one while the graphics processor reads and executes another. Second, DMA enables multiple applications to write command buffers into memory, and then submit the contents of the command buffers to the graphics processor when they are ready for execution. The graphics processor can only execute one command buffer at a time, but DMA facilitates the simultaneous construction of many command buffers because different threads can write into distinct memory regions that contain command buffers. Third, DMA frequently results in more efficient usage of the bus resources in the system. In other words, the bandwidth achievable through DMA is often higher than the bandwidth achievable by direct participation of the host microprocessor.

Ex. C9 – Invalidity of U.S. Patent No. RE49,461 – Wilt

U.S. Patent No. RE49,461	Wilt
	The Accelerated Graphics Port (AGP) is a specialized bus designed specifically to serve as a high bandwidth conduit to transfer data from system memory to a graphics processor. For the purposes of the present description, AGP may be considered a special case of DMA. Besides the mechanism for delivering information to and receiving information from the hardware, another important distinction to make in our background discussion centers on notification of the host microprocessor that the graphics processor has completed some task, or that some event has occurred that is visible to the graphics processor. Non-limiting examples of tasks include finishing execution of a command buffer or the completion of a specific command. Non-limiting examples of events include vertical retrace of the display device, the completion of a field of digitized video from a video port, or the retracing of a specific scanline in the display device. An alternative method for notifying the host microprocessor of a completed task or an external event is known as an interrupt. An interrupt is a hardware mechanism that diverts the host microprocessor from whatever it is doing and transfers control to a special location in memory. The operating system uses this mechanism to call driver code known as an interrupt service routine to respond to the interrupt. Interrupts have several advantages over polling based mechanisms. For one thing, the cost of setting up and responding to an interrupt is fixed, while the cost of a polling based mechanism increases approximately linearly with the number of polls performed. As a result, interrupt based schemes generally consume less bus bandwidth as well as fewer CPU cycles than polling based schemes. Interrupts can also result in a more timely notification if the interrupt latency is smaller than the time between when the event occurs and when the next poll is performed. [0085]–[0088], [0090]–[0091]. Given these assumptions, we can broaden our focus from graphics processors and contemplate a system with two processors: a host CPU, and a coprocessor that is controlled by sending command buffers from the CPU, that can send data back to the CPU, and that coordinates control by notifying the CPU of various events when they occur. The examples in this document frequently assume that the coprocessor is a graphics processor (as when describing possible synchronization problems and how a system can be designed to solve those problems

Ex. C9 – Invalidity of U.S. Patent No. RE49,461 – Wilt

U.S. Patent No. RE49,461	Wilt
	efficiently), but these examples are for illustrative purposes only and do not limit the scope of this invention to graphics processors. In line with the above assumptions regarding computational resources in a graphics processing system, a non-limiting system is described below that makes efficient, shared use of the computational resources in a coprocessor. FIG. 2 depicts the life cycle of a command buffer as it is used by applications to send commands to the coprocessor, such as a graphics processor. When a command buffer is created and initialized, the buffer enters the initialized state 201. The command buffer is then available to be selected and written into by a thread that wishes to issue commands to the graphics processor. When a thread selects the command buffer, the memory in the command buffer is made available to that thread (perhaps by mapping the memory in the command buffer into that thread's address space) and the command buffer enters the Receiving state 202. As one of ordinary skill in the art can appreciate, the details of this mapping may vary. For example, the command buffer memory could be made visible to user mode, kernel mode, or both. In addition, there are numerous options as to the nature of the executable code that is writing into the Receiving command buffer. In this regard, FIGS. 3A through 3E depict several of these options, i.e., several different architectures for writing into the command buffer. For purposes of describing FIGS. 3A through 3E, an application is a process, or a thread in a process, that implements some specific graphical application participating in the system. The runtime is middleware provided in the operating system that the application uses. Among other things, the runtime may validate the application's usage of an application programming interface (API), translate simple-to-use, high-level API calls into more complicated, lower-level constructs that are more readily translatable to hardware commands, and so on. The driver is a hardware-specific component that translates runtime-level commands to hardware-specific commands. The driver components depicted in FIGS. 3A through 3E are the portions of the driver that translate runtime-level commands to hardware-specific commands. Some portions of the driver that perform tasks other than this translation, such as interrupt handling and command buffer dispatch, would probably reside in kernel mode and are not contemplated in FIGS. 3A through 3E. Most of the object code depicted in FIGS. 3A through 3E may run in user mode, although some portions, as pointed out below, can run in kernel mode. The command buffer is the piece of memory that includes commands that can be consumed directly by the hardware via some DMA mechanism. Several of the figures also

Ex. C9 – Invalidity of U.S. Patent No. RE49,461 – Wilt

U.S. Patent No. RE49,461	Wilt
	show a token stream, which is a hardware-independent stream of runtime-level primitives that can be translated into hardware-specific commands by the driver. Lastly, some of the figures show a JIT (Just-In-Time) compilation component that can translate a hardware-independent intermediate language into object code for execution by the host microprocessor. The boxes with OBJ in the corner include object code that can be directly executed by the microprocessor. Boxes with MSIL in the corner include intermediate code that is compiled by the JIT in order to execute on the microprocessor. If the application and runtime are both available on the client in intermediate form, the JIT can translate them into a unified piece of object code, shown as app executables 324, 334 and 345 FIGS. 3C to 3E, respectively, that implements the application. FIG. 3A shows the application 301, runtime 302 and driver 303 each performing their respective roles, calling into lower-level object code until the driver 303 writes hardware-specific commands into the command buffer 304. The application 301 could be an executable process, or a thread in a process, while the runtime 302 and driver 303 could each reside in DLLs. As depicted in FIG. 3A, all of the object code may reside in user mode (although some or all of the runtime 302 and/or driver 303 may reside in kernel mode). If the runtime 302 determines that the command buffer 304 must be submitted before execution can continue, or the driver 303 determines that the command buffer 304 is out of space, a transition to kernel mode may be requested to submit the command buffer 304 and retrieve another one in order to continue. FIG. 3B shows a variant in which the runtime 312 emits a hardware-independent stream of commands into a buffer 313, which is then parsed by the driver 314 and written into a command buffer 315. Application 311 operates to call lower-level object code in connection with runtime 312. FIG. 3B approximates the current architecture of the DIRECTX® software component, in which the runtime translates application activity into DrawPrimitives2 (“DP2”) tokens. When the DP2 token stream is submitted, a kernel transition occurs and the driver 314 translates the DP2 token stream into hardware-specific commands in kernel mode. FIG. 3B does not make any assumptions about whether the driver 314 is in user mode or kernel mode, and in this regard, driver component 314 could reside in user mode as well as kernel mode. FIG. 3C shows a variant in which the application 321 and runtime 322 are synthesized into a unified piece of object code by a JIT compiler 323. The resulting piece of object code 324 then

Ex. C9 – Invalidity of U.S. Patent No. RE49,461 – Wilt

U.S. Patent No. RE49,461	Wilt
	interacts with the driver 325 to write hardware-specific commands into the command buffer 326. FIG. 3D shows the JIT variant of the situation described for FIG. 3B, in which a hardware-independent token stream 335 is emitted by the software 334 above the driver 336. The driver 336 then translates the token stream to hardware-specific commands for the command buffer 337. As with FIG. 3C, application 331 and runtime 332 are synthesized into a unified piece of object code 334 by JIT compiler 333. FIG. 3E shows a system in which the application 341, runtime 342, and part of the driver 343 are all available on the client in intermediate language form MSIL. The JIT compiler 344 synthesizes these components into a unified piece of object code 345 specifically optimized to run on that particular client computer, taking into account not only the host microprocessor but also the coprocessor hardware configuration. Referring back to the command buffer life cycle depicted in FIG. 2, regardless of the method by which commands are written into the command buffer, the thread writing into the receiving command buffer eventually submits to the command buffer either because the command buffer has been filled, or because the drawing commands that have been written into the buffer are executed forthwith. At that time, the command buffer enters the waiting state 203, and the contents of the command buffer may not be submitted to the hardware until all of the resources utilized to execute the command buffer are available. As an example of a circumstance that may force a wait, if an application draws into a surface, the drawing must be completed before that surface may be selected as a texture. Once the waiting criteria have been fulfilled, the command buffer enters the ready state 204, where the scheduler considers it for submission to the coprocessor. The scheduler has a great deal of flexibility in deciding which command buffer should be submitted to the hardware. The scheduler may take into account the priority, if any, of the command buffer, the priority of the requesting thread, the real-time requirements of the system, and so on. Once the command buffer has been selected as the next to run, the command buffer enters the standby state 205. On some systems, only one command buffer per coprocessor may be in the standby state 205 in the system. When the command buffer in the standby state 205 can be submitted to the hardware, the system switches the coprocessor context to the context required by the command buffer, and then submits the command buffer to the hardware. This operation entails some involvement

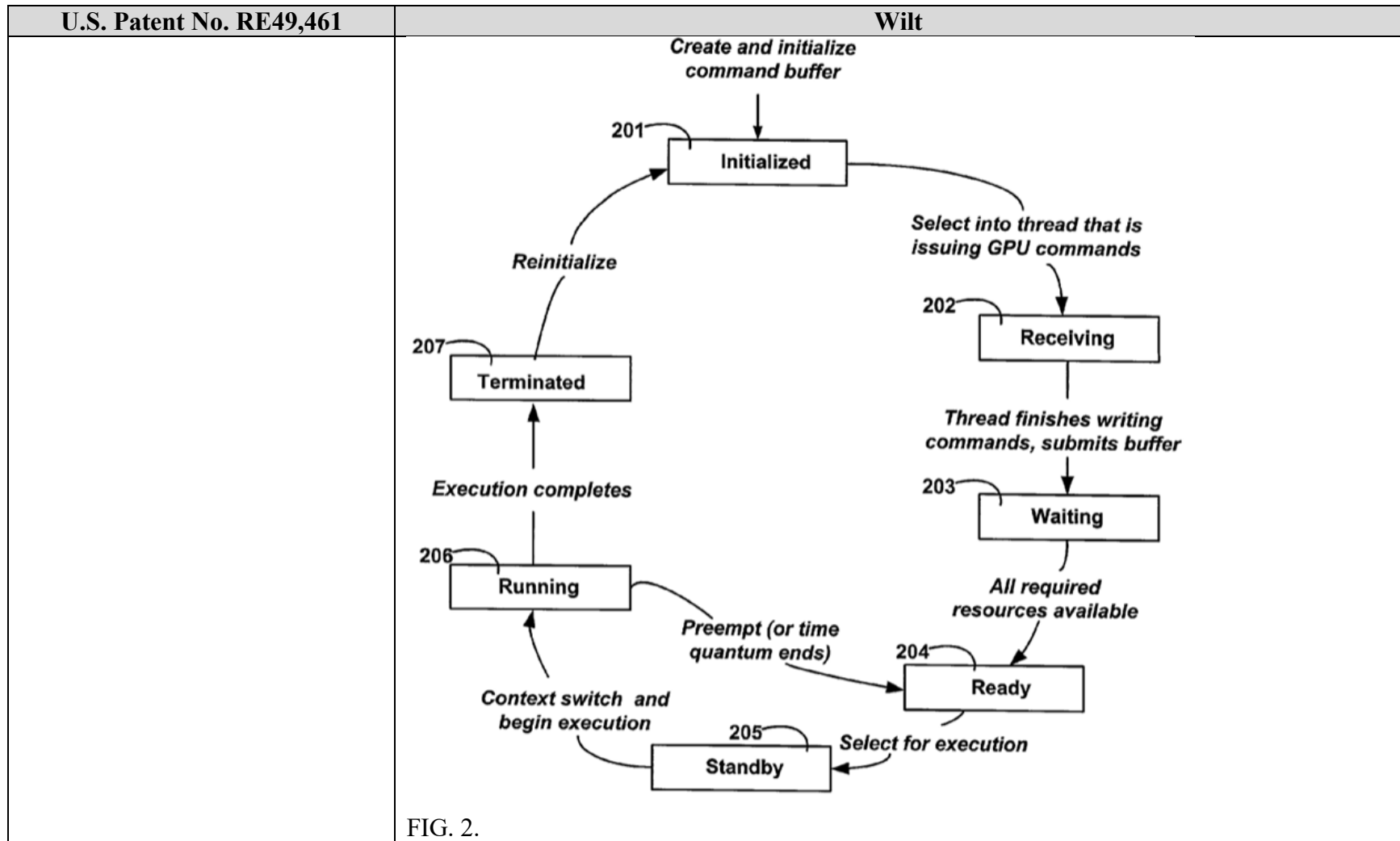
Ex. C9 – Invalidity of U.S. Patent No. RE49,461 – Wilt

U.S. Patent No. RE49,461	Wilt
	from the driver, for example to initiate a DMA operation using the base address and size of the command buffer. The command buffer remains in the running state 206 until some event preempts execution of the command buffer or the entire contents of the command buffer have been read and executed. If the entire command buffer has been executed, the command buffer enters the terminated state 207, where it becomes available for the system to reinitialize and make available to another thread in the initialized state 201. If the command buffer is preempted, it is assumed that the hardware has the capability to save the hardware context in such a way that execution of the command buffer can be resumed where the preemption occurred. The command buffer can then be recycled into the ready state 204 and be re-submitted to the hardware to resume from where it stopped. The description below regarding preemptible command buffer execution describes this in more detail. [0095]–[0110] With respect to queueing of command buffers, due to the high performance of graphics hardware, it is critical to minimize the latency between command buffer completion and the submission of another command buffer. Otherwise, the graphics hardware wastes valuable time waiting for another command buffer to be submitted. One possible solution to this problem is to queue up a context switch and command buffer dispatch before the current command buffer has finished executing. Then, provided the hardware and driver support this capability, the hardware can immediately begin execution of another command buffer when the current command buffer has completed. The system then has more time to prepare another command buffer for execution: the graphics hardware will only go idle if it completes the new command buffer before another one has been queued up. With respect to preemptible command buffer execution, the system described here provides for hardware support for preemption. A variety of events could potentially cause preemption of a command buffer during execution, ranging from an external event such as VSYNC to hardware enforcement of a time quantum. To implement this feature, the hardware and driver would have to enable the hardware context to be saved, such that execution of the preempted command buffer can be resumed where it left off; and the operating system would have to be notified so it could put the command buffer back in the Ready state mentioned in FIG. 2. One reason to implement such preemption is to guarantee that a display frame is composed

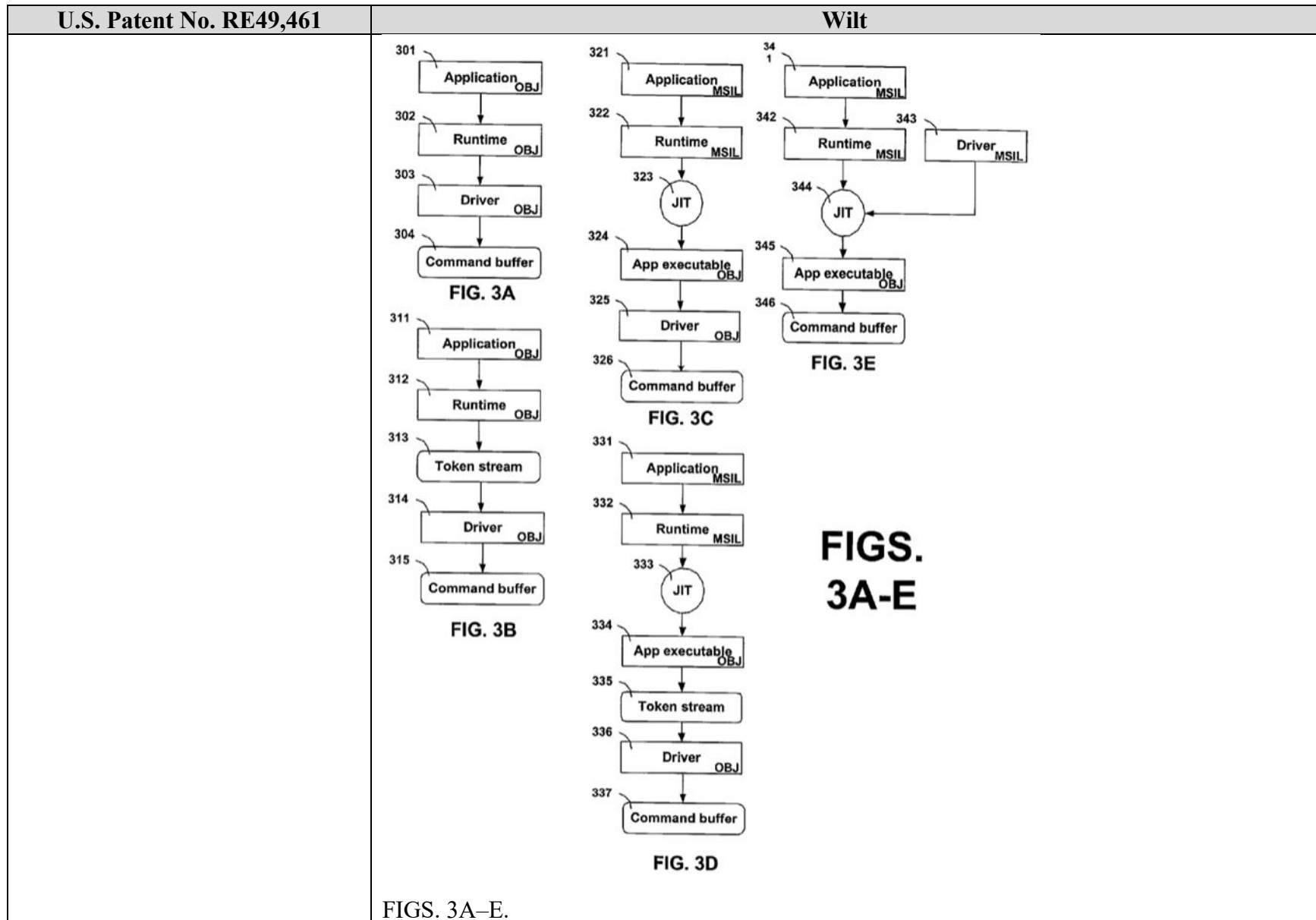
Ex. C9 – Invalidity of U.S. Patent No. RE49,461 – Wilt

U.S. Patent No. RE49,461	Wilt
	immediately after VSYNC (to make sure it will be ready when the next VSYNC occurs). Hardware enforcement of a time quantum would cause a command buffer to be preempted after executing for some set amount of time, such as 2 milliseconds. If the command buffer contains very expensive commands, such as drawing large, texture-mapped, alpha blended triangles, the hardware may have to preempt its execution to enable other command buffers to be considered for submission. [0126]–[0127] To the extent Plaintiff contends that this claim element is not expressly or inherently disclosed, or is not obvious in view of Wilt and the knowledge of a POSITA, it would have been obvious to combine the teachings of Wilt with any of the prior art references in NVIDIA’s Invalidity Contentions and any supplements thereto and the relevant section of charts for other prior art for the ’461 patent in a manner that would result in the subject matter of this limitation given, at the very least, the person's understanding of the state of the art, the problems addressed and solved in the prior art, and the teachings of Wilt. Specific examples of such references are cited and explained in these invalidity contentions, including the cover pleading and charts C1–C15, and Appendix C.
21[c]: shifting control of a data exchange between the user interaction stream and the computational stream to the computational stream in response to starting execution of the computations representing the artificial neural network;	Wilt discloses shifting control of a data exchange between the user interaction stream and the computational stream to the computational stream in response to starting execution of the computations representing the artificial neural network. <i>See e.g.:</i>

Ex. C9 – Invalidity of U.S. Patent No. RE49,461 – Wilt



Ex. C9 – Invalidity of U.S. Patent No. RE49,461 – Wilt



Ex. C9 – Invalidity of U.S. Patent No. RE49,461 – Wilt

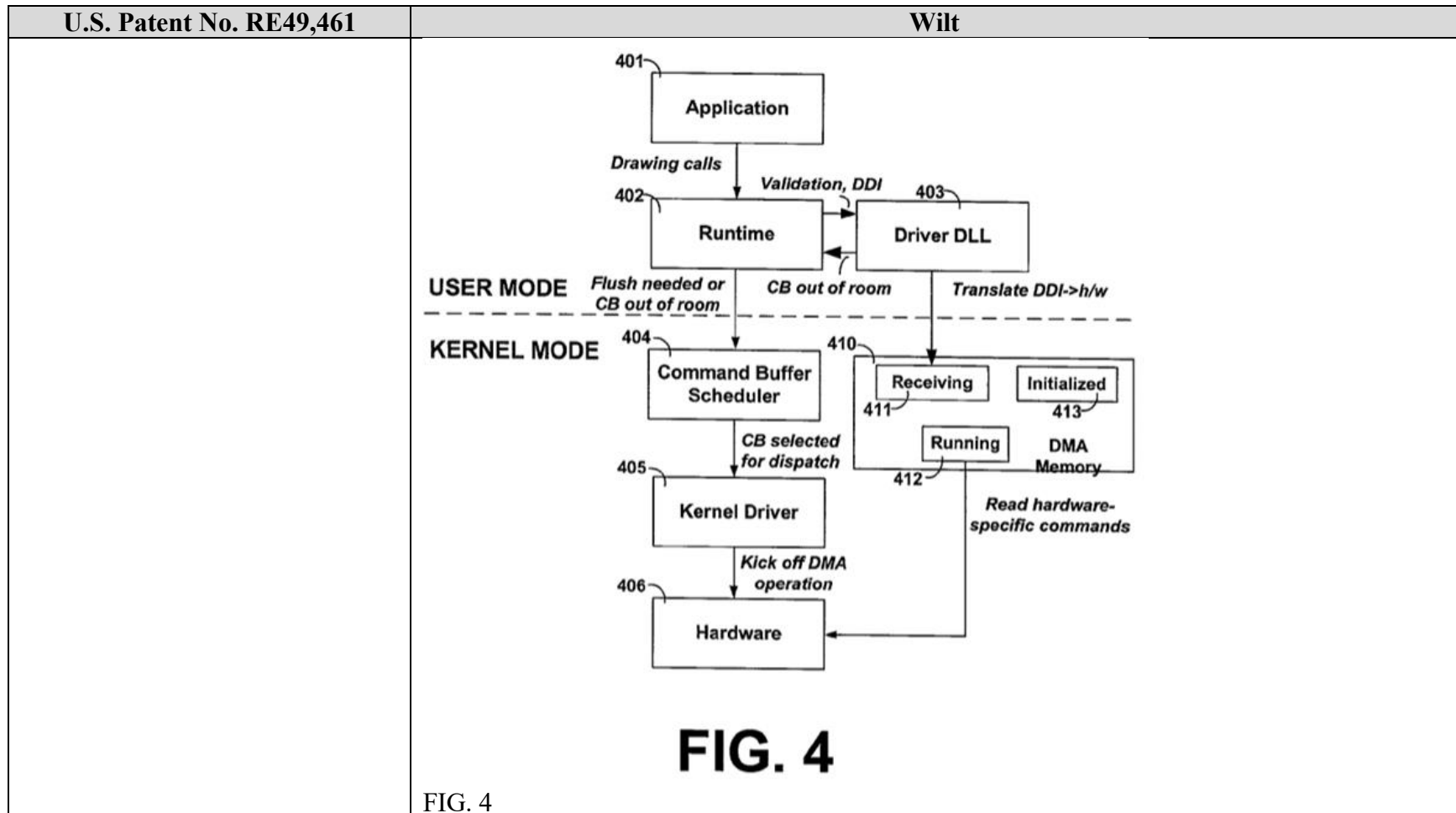


FIG. 4

Ex. C9 – Invalidity of U.S. Patent No. RE49,461 – Wilt

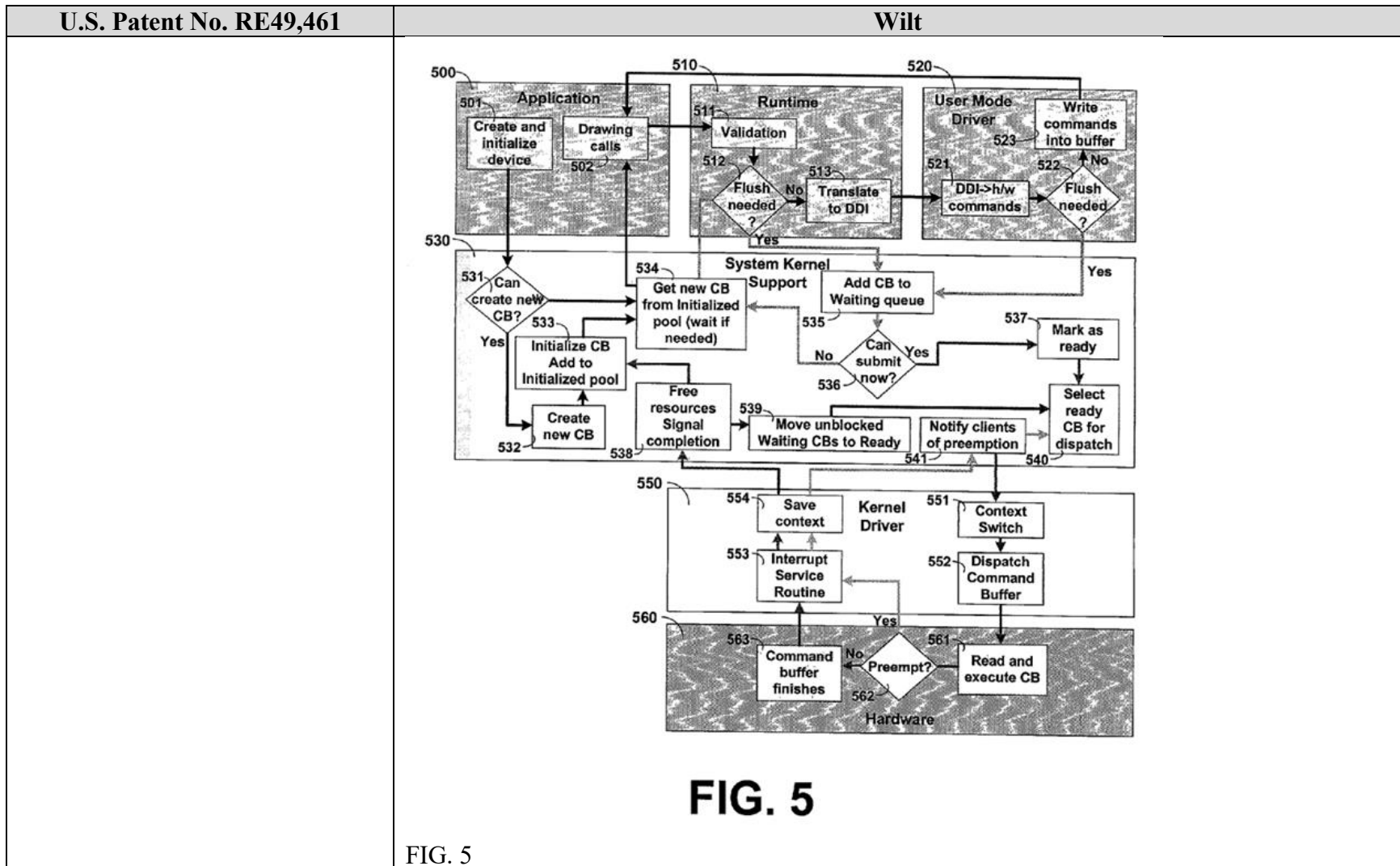


FIG. 5

FIG. 5

Ex. C9 – Invalidity of U.S. Patent No. RE49,461 – Wilt

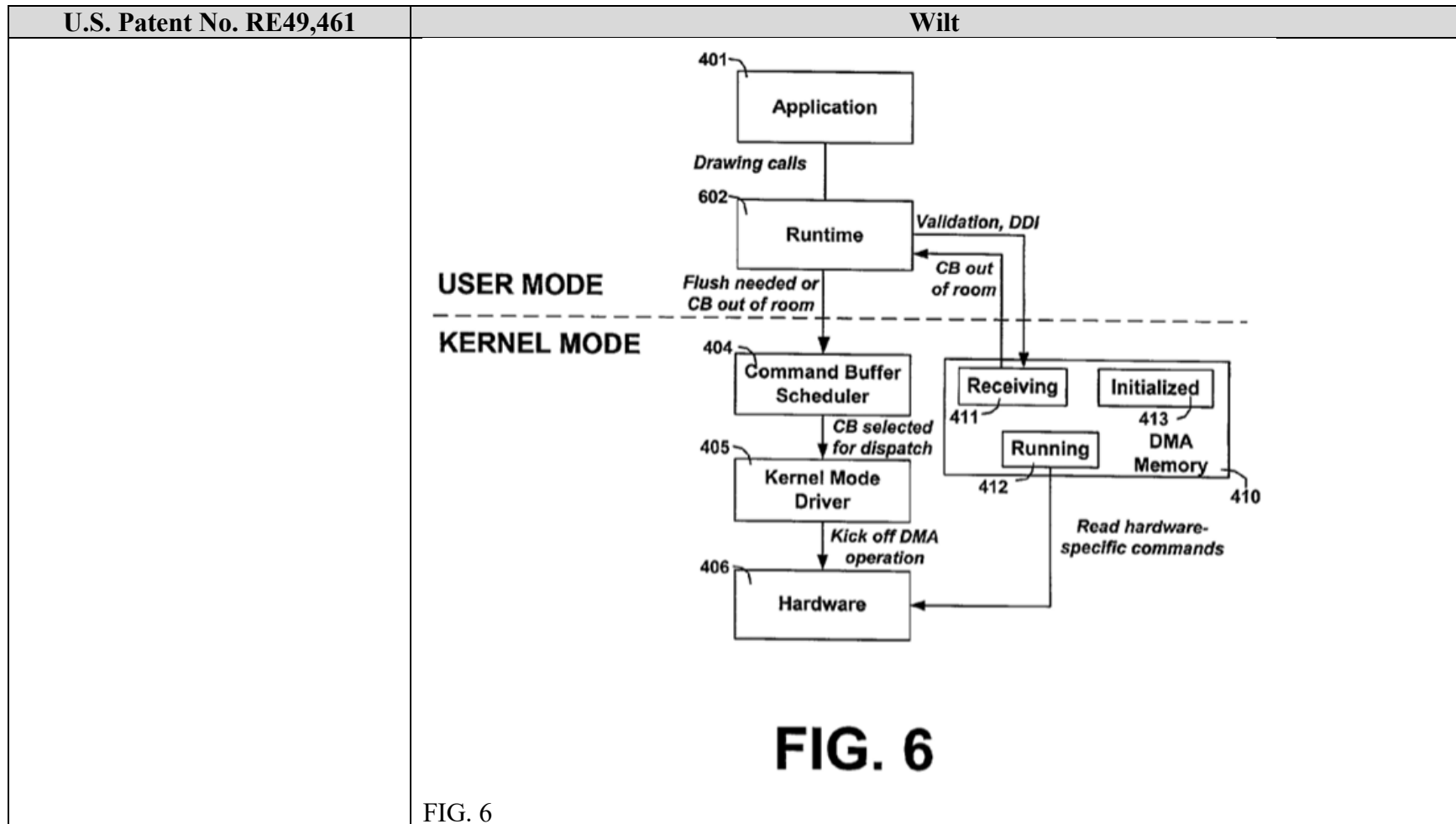
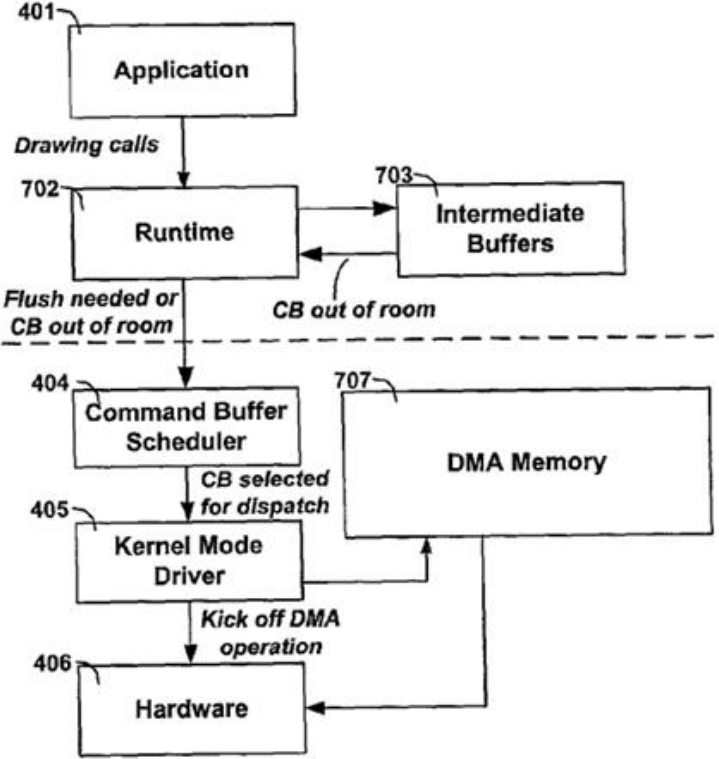


FIG. 6

U.S. Patent No. RE49,461	Wilt
	 FIG. 7 FIG. 7. A command buffer is a stream of hardware-specific drawing commands ready for consumption by graphics hardware, e.g., see the definition for token stream below.

Ex. C9 – Invalidity of U.S. Patent No. RE49,461 – Wilt

U.S. Patent No. RE49,461	Wilt
	A token stream is a stream of hardware-independent tokens that describe a series of drawing operations. A token stream can be translated by a hardware-specific software component, such as a driver, into a command buffer or other series of hardware commands. [0027]–[0046] Direct Memory Access (DMA) is a common method for moving data within computer systems, and has found application in the control of graphics processors as well as disk drives, network adapters, and external buses such as USB and IEEE 1394. DMA generally refers to the ability of a peripheral, such as a digital camera or disk controller, to read from or write to memory without directly involving the host processor. The host sets up the DMA transfer in advance, page locking the memory region where a digital camera, for example, is to write the image data from a picture it is about to take, and specifying that memory location to the camera. Page locking causes the virtual memory manager to forego writing the memory locations in question out to disk. Once this is done, the camera can take the picture and write the image data into memory, then notify the host that the DMA transfer has been completed. In the context of graphics processor control, a series of commands for the graphics processor can be written into memory, and then transferred to the graphics processor via a DMA operation. DMA has many advantages over the other methods of submitting graphics processor commands. First, it facilitates parallel execution by the host and graphics processors. Since these ‘other operations’ to be performed by the CPU often involve generating more commands for the graphics processor, it is common to maintain multiple DMA buffers and ‘ping-pong’ between them, writing to one while the graphics processor reads and executes another. Second, DMA enables multiple applications to write command buffers into memory, and then submit the contents of the command buffers to the graphics processor when they are ready for execution. The graphics processor can only execute one command buffer at a time, but DMA facilitates the simultaneous construction of many command buffers because different threads can write into distinct memory regions that contain command buffers. Third, DMA frequently results in more efficient usage of the bus resources in the system. In other words, the bandwidth achievable through DMA is often higher than the bandwidth achievable by direct participation of the host microprocessor.

Ex. C9 – Invalidity of U.S. Patent No. RE49,461 – Wilt

U.S. Patent No. RE49,461	Wilt
	The Accelerated Graphics Port (AGP) is a specialized bus designed specifically to serve as a high bandwidth conduit to transfer data from system memory to a graphics processor. For the purposes of the present description, AGP may be considered a special case of DMA. Besides the mechanism for delivering information to and receiving information from the hardware, another important distinction to make in our background discussion centers on notification of the host microprocessor that the graphics processor has completed some task, or that some event has occurred that is visible to the graphics processor. Non-limiting examples of tasks include finishing execution of a command buffer or the completion of a specific command. Non-limiting examples of events include vertical retrace of the display device, the completion of a field of digitized video from a video port, or the retracing of a specific scanline in the display device. An alternative method for notifying the host microprocessor of a completed task or an external event is known as an interrupt. An interrupt is a hardware mechanism that diverts the host microprocessor from whatever it is doing and transfers control to a special location in memory. The operating system uses this mechanism to call driver code known as an interrupt service routine to respond to the interrupt. Interrupts have several advantages over polling based mechanisms. For one thing, the cost of setting up and responding to an interrupt is fixed, while the cost of a polling based mechanism increases approximately linearly with the number of polls performed. As a result, interrupt based schemes generally consume less bus bandwidth as well as fewer CPU cycles than polling based schemes. Interrupts can also result in a more timely notification if the interrupt latency is smaller than the time between when the event occurs and when the next poll is performed. [0085]–[0088], [0090]–[0091]. Given these assumptions, we can broaden our focus from graphics processors and contemplate a system with two processors: a host CPU, and a coprocessor that is controlled by sending command buffers from the CPU, that can send data back to the CPU, and that coordinates control by notifying the CPU of various events when they occur. The examples in this document frequently assume that the coprocessor is a graphics processor (as when describing possible synchronization problems and how a system can be designed to solve those problems

Ex. C9 – Invalidity of U.S. Patent No. RE49,461 – Wilt

U.S. Patent No. RE49,461	Wilt
	efficiently), but these examples are for illustrative purposes only and do not limit the scope of this invention to graphics processors. In line with the above assumptions regarding computational resources in a graphics processing system, a non-limiting system is described below that makes efficient, shared use of the computational resources in a coprocessor. FIG. 2 depicts the life cycle of a command buffer as it is used by applications to send commands to the coprocessor, such as a graphics processor. When a command buffer is created and initialized, the buffer enters the initialized state 201. The command buffer is then available to be selected and written into by a thread that wishes to issue commands to the graphics processor. When a thread selects the command buffer, the memory in the command buffer is made available to that thread (perhaps by mapping the memory in the command buffer into that thread's address space) and the command buffer enters the Receiving state 202. As one of ordinary skill in the art can appreciate, the details of this mapping may vary. For example, the command buffer memory could be made visible to user mode, kernel mode, or both. In addition, there are numerous options as to the nature of the executable code that is writing into the Receiving command buffer. In this regard, FIGS. 3A through 3E depict several of these options, i.e., several different architectures for writing into the command buffer. For purposes of describing FIGS. 3A through 3E, an application is a process, or a thread in a process, that implements some specific graphical application participating in the system. The runtime is middleware provided in the operating system that the application uses. Among other things, the runtime may validate the application's usage of an application programming interface (API), translate simple-to-use, high-level API calls into more complicated, lower-level constructs that are more readily translatable to hardware commands, and so on. The driver is a hardware-specific component that translates runtime-level commands to hardware-specific commands. The driver components depicted in FIGS. 3A through 3E are the portions of the driver that translate runtime-level commands to hardware-specific commands. Some portions of the driver that perform tasks other than this translation, such as interrupt handling and command buffer dispatch, would probably reside in kernel mode and are not contemplated in FIGS. 3A through 3E. Most of the object code depicted in FIGS. 3A through 3E may run in user mode, although some portions, as pointed out below, can run in kernel mode. The command buffer is the piece of memory that includes commands that can be consumed directly by the hardware via some DMA mechanism. Several of the figures also

Ex. C9 – Invalidity of U.S. Patent No. RE49,461 – Wilt

U.S. Patent No. RE49,461	Wilt
	show a token stream, which is a hardware-independent stream of runtime-level primitives that can be translated into hardware-specific commands by the driver. Lastly, some of the figures show a JIT (Just-In-Time) compilation component that can translate a hardware-independent intermediate language into object code for execution by the host microprocessor. The boxes with OBJ in the corner include object code that can be directly executed by the microprocessor. Boxes with MSIL in the corner include intermediate code that is compiled by the JIT in order to execute on the microprocessor. If the application and runtime are both available on the client in intermediate form, the JIT can translate them into a unified piece of object code, shown as app executables 324, 334 and 345 FIGS. 3C to 3E, respectively, that implements the application. FIG. 3A shows the application 301, runtime 302 and driver 303 each performing their respective roles, calling into lower-level object code until the driver 303 writes hardware-specific commands into the command buffer 304. The application 301 could be an executable process, or a thread in a process, while the runtime 302 and driver 303 could each reside in DLLs. As depicted in FIG. 3A, all of the object code may reside in user mode (although some or all of the runtime 302 and/or driver 303 may reside in kernel mode). If the runtime 302 determines that the command buffer 304 must be submitted before execution can continue, or the driver 303 determines that the command buffer 304 is out of space, a transition to kernel mode may be requested to submit the command buffer 304 and retrieve another one in order to continue. FIG. 3B shows a variant in which the runtime 312 emits a hardware-independent stream of commands into a buffer 313, which is then parsed by the driver 314 and written into a command buffer 315. Application 311 operates to call lower-level object code in connection with runtime 312. FIG. 3B approximates the current architecture of the DIRECTX® software component, in which the runtime translates application activity into DrawPrimitives2 (“DP2”) tokens. When the DP2 token stream is submitted, a kernel transition occurs and the driver 314 translates the DP2 token stream into hardware-specific commands in kernel mode. FIG. 3B does not make any assumptions about whether the driver 314 is in user mode or kernel mode, and in this regard, driver component 314 could reside in user mode as well as kernel mode. FIG. 3C shows a variant in which the application 321 and runtime 322 are synthesized into a unified piece of object code by a JIT compiler 323. The resulting piece of object code 324 then

Ex. C9 – Invalidity of U.S. Patent No. RE49,461 – Wilt

U.S. Patent No. RE49,461	Wilt
	interacts with the driver 325 to write hardware-specific commands into the command buffer 326. FIG. 3D shows the JIT variant of the situation described for FIG. 3B, in which a hardware-independent token stream 335 is emitted by the software 334 above the driver 336. The driver 336 then translates the token stream to hardware-specific commands for the command buffer 337. As with FIG. 3C, application 331 and runtime 332 are synthesized into a unified piece of object code 334 by JIT compiler 333. FIG. 3E shows a system in which the application 341, runtime 342, and part of the driver 343 are all available on the client in intermediate language form MSIL. The JIT compiler 344 synthesizes these components into a unified piece of object code 345 specifically optimized to run on that particular client computer, taking into account not only the host microprocessor but also the coprocessor hardware configuration. Referring back to the command buffer life cycle depicted in FIG. 2, regardless of the method by which commands are written into the command buffer, the thread writing into the receiving command buffer eventually submits to the command buffer either because the command buffer has been filled, or because the drawing commands that have been written into the buffer are executed forthwith. At that time, the command buffer enters the waiting state 203, and the contents of the command buffer may not be submitted to the hardware until all of the resources utilized to execute the command buffer are available. As an example of a circumstance that may force a wait, if an application draws into a surface, the drawing must be completed before that surface may be selected as a texture. Once the waiting criteria have been fulfilled, the command buffer enters the ready state 204, where the scheduler considers it for submission to the coprocessor. The scheduler has a great deal of flexibility in deciding which command buffer should be submitted to the hardware. The scheduler may take into account the priority, if any, of the command buffer, the priority of the requesting thread, the real-time requirements of the system, and so on. Once the command buffer has been selected as the next to run, the command buffer enters the standby state 205. On some systems, only one command buffer per coprocessor may be in the standby state 205 in the system. When the command buffer in the standby state 205 can be submitted to the hardware, the system switches the coprocessor context to the context required by the command buffer, and then submits the command buffer to the hardware. This operation entails some involvement

Ex. C9 – Invalidity of U.S. Patent No. RE49,461 – Wilt

U.S. Patent No. RE49,461	Wilt
	from the driver, for example to initiate a DMA operation using the base address and size of the command buffer. The command buffer remains in the running state 206 until some event preempts execution of the command buffer or the entire contents of the command buffer have been read and executed. If the entire command buffer has been executed, the command buffer enters the terminated state 207, where it becomes available for the system to reinitialize and make available to another thread in the initialized state 201. If the command buffer is preempted, it is assumed that the hardware has the capability to save the hardware context in such a way that execution of the command buffer can be resumed where the preemption occurred. The command buffer can then be recycled into the ready state 204 and be re-submitted to the hardware to resume from where it stopped. The description below regarding preemptible command buffer execution describes this in more detail. [0095]–[0110] With respect to queueing of command buffers, due to the high performance of graphics hardware, it is critical to minimize the latency between command buffer completion and the submission of another command buffer. Otherwise, the graphics hardware wastes valuable time waiting for another command buffer to be submitted. One possible solution to this problem is to queue up a context switch and command buffer dispatch before the current command buffer has finished executing. Then, provided the hardware and driver support this capability, the hardware can immediately begin execution of another command buffer when the current command buffer has completed. The system then has more time to prepare another command buffer for execution: the graphics hardware will only go idle if it completes the new command buffer before another one has been queued up. With respect to preemptible command buffer execution, the system described here provides for hardware support for preemption. A variety of events could potentially cause preemption of a command buffer during execution, ranging from an external event such as VSYNC to hardware enforcement of a time quantum. To implement this feature, the hardware and driver would have to enable the hardware context to be saved, such that execution of the preempted command buffer can be resumed where it left off; and the operating system would have to be notified so it could put the command buffer back in the Ready state mentioned in FIG. 2. One reason to implement such preemption is to guarantee that a display frame is composed

Ex. C9 – Invalidity of U.S. Patent No. RE49,461 – Wilt

U.S. Patent No. RE49,461	Wilt
	immediately after VSYNC (to make sure it will be ready when the next VSYNC occurs). Hardware enforcement of a time quantum would cause a command buffer to be preempted after executing for some set amount of time, such as 2 milliseconds. If the command buffer contains very expensive commands, such as drawing large, texture-mapped, alpha blended triangles, the hardware may have to preempt its execution to enable other command buffers to be considered for submission. [0126]–[0127] To the extent Plaintiff contends that this claim element is not expressly or inherently disclosed, or is not obvious in view of Wilt and the knowledge of a POSITA, it would have been obvious to combine the teachings of Wilt with any of the prior art references in NVIDIA’s Invalidity Contentions and any supplements thereto and the relevant section of charts for other prior art for the ’461 patent in a manner that would result in the subject matter of this limitation given, at the very least, the person's understanding of the state of the art, the problems addressed and solved in the prior art, and the teachings of Wilt. Specific examples of such references are cited and explained in these invalidity contentions, including the cover pleading and charts C1–C15, and Appendix C.
21[d]: shifting control of the data exchange between the user interaction stream and the computational stream to the user interaction stream in response to completion or interruption of the computations representing the artificial neural network;	Wilt discloses shifting control of the data exchange between the user interaction stream and the computational stream to the user interaction stream in response to completion or interruption of the computations representing the artificial neural network. <i>See e.g.:</i> Direct Memory Access (DMA) is a common method for moving data within computer systems, and has found application in the control of graphics processors as well as disk drives, network adapters, and external buses such as USB and IEEE 1394. DMA generally refers to the ability of a peripheral, such as a digital camera or disk controller, to read from or write to memory without directly involving the host processor. The host sets up the DMA transfer in advance, page locking the memory region where a digital camera, for example, is to write the image data from a picture it is about to take, and specifying that memory location to the camera. Page locking causes the virtual memory manager to forego writing the memory locations in question out to disk. Once this is done, the camera can take the picture and write the image data into memory, then notify the host that the DMA transfer has been completed. In

Ex. C9 – Invalidity of U.S. Patent No. RE49,461 – Wilt

U.S. Patent No. RE49,461	Wilt
	the context of graphics processor control, a series of commands for the graphics processor can be written into memory, and then transferred to the graphics processor via a DMA operation. DMA has many advantages over the other methods of submitting graphics processor commands. First, it facilitates parallel execution by the host and graphics processors. Since these ‘other operations’ to be performed by the CPU often involve generating more commands for the graphics processor, it is common to maintain multiple DMA buffers and ‘ping-pong’ between them, writing to one while the graphics processor reads and executes another. Second, DMA enables multiple applications to write command buffers into memory, and then submit the contents of the command buffers to the graphics processor when they are ready for execution. The graphics processor can only execute one command buffer at a time, but DMA facilitates the simultaneous construction of many command buffers because different threads can write into distinct memory regions that contain command buffers. Third, DMA frequently results in more efficient usage of the bus resources in the system. In other words, the bandwidth achievable through DMA is often higher than the bandwidth achievable by direct participation of the host microprocessor. The Accelerated Graphics Port (AGP) is a specialized bus designed specifically to serve as a high bandwidth conduit to transfer data from system memory to a graphics processor. For the purposes of the present description, AGP may be considered a special case of DMA. Besides the mechanism for delivering information to and receiving information from the hardware, another important distinction to make in our background discussion centers on notification of the host microprocessor that the graphics processor has completed some task, or that some event has occurred that is visible to the graphics processor. Non-limiting examples of tasks include finishing execution of a command buffer or the completion of a specific command. Non-limiting examples of events include vertical retrace of the display device, the completion of a field of digitized video from a video port, or the retracing of a specific scanline in the display device. An alternative method for notifying the host microprocessor of a completed task or an external event is known as an interrupt. An interrupt is a hardware mechanism that diverts the host microprocessor from whatever it is doing and transfers control to a special location in memory. The operating system uses this mechanism to call driver code known as an interrupt service routine to respond to the interrupt.

Ex. C9 – Invalidity of U.S. Patent No. RE49,461 – Wilt

U.S. Patent No. RE49,461	Wilt
	Interrupts have several advantages over polling based mechanisms. For one thing, the cost of setting up and responding to an interrupt is fixed, while the cost of a polling based mechanism increases approximately linearly with the number of polls performed. As a result, interrupt based schemes generally consume less bus bandwidth as well as fewer CPU cycles than polling based schemes. Interrupts can also result in a more timely notification if the interrupt latency is smaller than the time between when the event occurs and when the next poll is performed. [0085]–[0088], [0090]–[0091]. Given these assumptions, we can broaden our focus from graphics processors and contemplate a system with two processors: a host CPU, and a coprocessor that is controlled by sending command buffers from the CPU, that can send data back to the CPU, and that coordinates control by notifying the CPU of various events when they occur. The examples in this document frequently assume that the coprocessor is a graphics processor (as when describing possible synchronization problems and how a system can be designed to solve those problems efficiently), but these examples are for illustrative purposes only and do not limit the scope of this invention to graphics processors. In line with the above assumptions regarding computational resources in a graphics processing system, a non-limiting system is described below that makes efficient, shared use of the computational resources in a coprocessor. FIG. 2 depicts the life cycle of a command buffer as it is used by applications to send commands to the coprocessor, such as a graphics processor. When a command buffer is created and initialized, the buffer enters the initialized state 201. The command buffer is then available to be selected and written into by a thread that wishes to issue commands to the graphics processor. When a thread selects the command buffer, the memory in the command buffer is made available to that thread (perhaps by mapping the memory in the command buffer into that thread's address space) and the command buffer enters the Receiving state 202. As one of ordinary skill in the art can appreciate, the details of this mapping may vary. For example, the command buffer memory could be made visible to user mode, kernel mode, or both. In addition, there are numerous options as to the nature of the executable code that is writing into the Receiving command buffer. In this regard, FIGS. 3A through 3E depict several of these options, i.e., several different architectures for writing into the command buffer.

Ex. C9 – Invalidity of U.S. Patent No. RE49,461 – Wilt

U.S. Patent No. RE49,461	Wilt
	For purposes of describing FIGS. 3A through 3E, an application is a process, or a thread in a process, that implements some specific graphical application participating in the system. The runtime is middleware provided in the operating system that the application uses. Among other things, the runtime may validate the application's usage of an application programming interface (API), translate simple-to-use, high-level API calls into more complicated, lower-level constructs that are more readily translatable to hardware commands, and so on. The driver is a hardware-specific component that translates runtime-level commands to hardware-specific commands. The driver components depicted in FIGS. 3A through 3E are the portions of the driver that translate runtime-level commands to hardware-specific commands. Some portions of the driver that perform tasks other than this translation, such as interrupt handling and command buffer dispatch, would probably reside in kernel mode and are not contemplated in FIGS. 3A through 3E. Most of the object code depicted in FIGS. 3A through 3E may run in user mode, although some portions, as pointed out below, can run in kernel mode. The command buffer is the piece of memory that includes commands that can be consumed directly by the hardware via some DMA mechanism. Several of the figures also show a token stream, which is a hardware-independent stream of runtime-level primitives that can be translated into hardware-specific commands by the driver. Lastly, some of the figures show a JIT (Just-In-Time) compilation component that can translate a hardware-independent intermediate language into object code for execution by the host microprocessor. The boxes with OBJ in the corner include object code that can be directly executed by the microprocessor. Boxes with MSIL in the corner include intermediate code that is compiled by the JIT in order to execute on the microprocessor. If the application and runtime are both available on the client in intermediate form, the JIT can translate them into a unified piece of object code, shown as app executables 324, 334 and 345 FIGS. 3C to 3E, respectively, that implements the application. FIG. 3A shows the application 301, runtime 302 and driver 303 each performing their respective roles, calling into lower-level object code until the driver 303 writes hardware-specific commands into the command buffer 304. The application 301 could be an executable process, or a thread in a process, while the runtime 302 and driver 303 could each reside in DLLs. As depicted in FIG. 3A, all of the object code may reside in user mode (although some or all of the runtime 302 and/or driver 303 may reside in kernel mode). If the runtime 302 determines that the command buffer 304 must be submitted before execution can continue, or

Ex. C9 – Invalidity of U.S. Patent No. RE49,461 – Wilt

U.S. Patent No. RE49,461	Wilt
	the driver 303 determines that the command buffer 304 is out of space, a transition to kernel mode may be requested to submit the command buffer 304 and retrieve another one in order to continue. FIG. 3B shows a variant in which the runtime 312 emits a hardware-independent stream of commands into a buffer 313, which is then parsed by the driver 314 and written into a command buffer 315. Application 311 operates to call lower-level object code in connection with runtime 312. FIG. 3B approximates the current architecture of the DIRECTX® software component, in which the runtime translates application activity into DrawPrimitives2 (“DP2”) tokens. When the DP2 token stream is submitted, a kernel transition occurs and the driver 314 translates the DP2 token stream into hardware-specific commands in kernel mode. FIG. 3B does not make any assumptions about whether the driver 314 is in user mode or kernel mode, and in this regard, driver component 314 could reside in user mode as well as kernel mode. FIG. 3C shows a variant in which the application 321 and runtime 322 are synthesized into a unified piece of object code by a JIT compiler 323. The resulting piece of object code 324 then interacts with the driver 325 to write hardware-specific commands into the command buffer 326. FIG. 3D shows the JIT variant of the situation described for FIG. 3B, in which a hardware-independent token stream 335 is emitted by the software 334 above the driver 336. The driver 336 then translates the token stream to hardware-specific commands for the command buffer 337. As with FIG. 3C, application 331 and runtime 332 are synthesized into a unified piece of object code 334 by JIT compiler 333. FIG. 3E shows a system in which the application 341, runtime 342, and part of the driver 343 are all available on the client in intermediate language form MSIL. The JIT compiler 344 synthesizes these components into a unified piece of object code 345 specifically optimized to run on that particular client computer, taking into account not only the host microprocessor but also the coprocessor hardware configuration. Referring back to the command buffer life cycle depicted in FIG. 2, regardless of the method by which commands are written into the command buffer, the thread writing into the receiving command buffer eventually submits to the command buffer either because the command buffer has been filled, or because the drawing commands that have been written into the buffer are executed forthwith. At that time, the command buffer enters the waiting state 203, and the contents of the command buffer may not be submitted to the hardware until all of the resources

Ex. C9 – Invalidity of U.S. Patent No. RE49,461 – Wilt

U.S. Patent No. RE49,461	Wilt
	utilized to execute the command buffer are available. As an example of a circumstance that may force a wait, if an application draws into a surface, the drawing must be completed before that surface may be selected as a texture. Once the waiting criteria have been fulfilled, the command buffer enters the ready state 204, where the scheduler considers it for submission to the coprocessor. The scheduler has a great deal of flexibility in deciding which command buffer should be submitted to the hardware. The scheduler may take into account the priority, if any, of the command buffer, the priority of the requesting thread, the real-time requirements of the system, and so on. Once the command buffer has been selected as the next to run, the command buffer enters the standby state 205. On some systems, only one command buffer per coprocessor may be in the standby state 205 in the system. When the command buffer in the standby state 205 can be submitted to the hardware, the system switches the coprocessor context to the context required by the command buffer, and then submits the command buffer to the hardware. This operation entails some involvement from the driver, for example to initiate a DMA operation using the base address and size of the command buffer. The command buffer remains in the running state 206 until some event preempts execution of the command buffer or the entire contents of the command buffer have been read and executed. If the entire command buffer has been executed, the command buffer enters the terminated state 207, where it becomes available for the system to reinitialize and make available to another thread in the initialized state 201. If the command buffer is preempted, it is assumed that the hardware has the capability to save the hardware context in such a way that execution of the command buffer can be resumed where the preemption occurred. The command buffer can then be recycled into the ready state 204 and be re-submitted to the hardware to resume from where it stopped. The description below regarding preemptible command buffer execution describes this in more detail. [0095]–[0110] With respect to context switching, the system described above can accommodate hardware-accelerated, or at least driver-facilitated, context switching, in which hardware contexts are maintained by the driver and context switching may be supported natively in the hardware. Contexts may be cached in local video memory or other high speed memory, for example. In

Ex. C9 – Invalidity of U.S. Patent No. RE49,461 – Wilt

U.S. Patent No. RE49,461	Wilt
	this case, the operating system requests that the driver load a given context into the hardware. The context switch and command buffer dispatch may be specified in a single DDI (Device Driver Interface) call for performance reasons. With respect to interrupt handling, interrupts may be used for efficient management of coprocessors' computational resources because they facilitate the host microprocessor's performance of useful work while the coprocessor hardware is running. The presently described embodiments detail some of the applications for interrupts in the system and the features that interrupts would enable in each case. In all cases, interrupts present an opportunity for the operating system to notify clients of the event that caused the interrupt, whether by writing memory or signaling a thread synchronization object such as an event. For interrupts on command buffer completion, when a command buffer is finished executing, the system is notified so it can select another command buffer to submit to the hardware. Whether the coprocessor is idle from when it signals the interrupt until another command buffer is submitted, or if the coprocessor begins executing a previously-queued command buffer (as described below), the system must be notified so it can select another command buffer for submission (or eventual submission) to the hardware. Since the interrupt signals that the command buffer has been executed, the system can release all of the resources that were needed for the command buffer and unblocks pending command buffers that need those resources, and can [0111], [0114]–[0115] The command buffer scheduler 404 (“scheduler”) and kernel driver 405 work together in kernel mode to dispatch command buffers to the hardware 406 (the scheduler 404 decides which command buffer should be dispatched, while the kernel driver 405 instructs the hardware 406 to dispatch a command buffer at the request of the scheduler). This system contemplates that the bulk of the driver logic would reside in the user mode driver DLL 403, not the kernel driver 405. While the User Mode Driver can contain large amounts of code that maps DDI-level calls into hardware-specific commands (which operation can be complicated and error-prone, especially when compiling a vertex and/or shader program), the Kernel Driver is comparatively small and simple, maximizing system robustness. FIG. 5 clarifies the sequence of events that occur when an application is making API calls to perform graphics operations. The block diagram components of FIG. 5 are shading coded: user

Ex. C9 – Invalidity of U.S. Patent No. RE49,461 – Wilt

U.S. Patent No. RE49,461	Wilt
	mode components are of medium shade, kernel mode components are shaded lightly, and the hardware is shaded darkly. The command buffers are not specifically depicted in FIG. 5 as a hardware component; as per FIG. 4, the user mode driver 520 writes hardware-specific commands into the device's current command buffer, the command buffer scheduler (part of the system kernel support 530) dispatches the command buffer to the hardware 560 via the kernel mode driver 550, and finished command buffers are recycled for use by an application in the system. It is noted that multiple applications 500 can potentially share the pool of available command buffers, with the system kernel support 530 arbitrating sharing of that resource. When the application 500 initially creates a drawing context 501, the system kernel support 530 checks to see whether a new command buffer can be created 531. If so, the new command buffer is created 532 and initialized 533, and the thread obtains an initialized command buffer 534 before the Application 500 can perform drawing calls 502. If a command buffer could not be created in step 531, the application 500 must wait until an initialized command buffer becomes available 534. Once the application 500 has obtained a command buffer, the application 500, runtime 510 and user mode driver 520 enter the typical interaction between the three components that cause hardware-specific commands to be written into the command buffer. The drawing calls 502 from the application 500 are validated 511 by the runtime 510; a check 512 then determines whether a flush of the current command buffer is needed. If not, the drawing command is translated to a simpler, canonical DDI call 513 and passed to the user mode driver 520. The driver translates the DDI call into hardware specific commands and attempts to write them into the command buffer. If the check 522 for flush determines that there is no room in the command buffer, the command buffer must be submitted to the system kernel support 530 and a new command buffer obtained from same before the command can be written and execution can continue. If either the runtime 510 or the user mode driver 520 determines that a flush is needed, per step 535 the command buffer is added to the waiting queue. At that time, the system kernel can check 536 whether the command buffer can be submitted forthwith (typically because no command buffer is running). If not, the command buffer is left in the waiting queue and a new command buffer must be obtained 534. It is noted that this functional block, which waits until a suitable initialized command buffer is available and then allocates it to the device, is identical to the operation needed by the application 500 before it can begin drawing.

Ex. C9 – Invalidity of U.S. Patent No. RE49,461 – Wilt

U.S. Patent No. RE49,461	Wilt
	The arrows are also shading coded. The darkest arrows trace the “typical” thread execution of the application 500 as it initializes and performs drawing commands and the runtime 510 and user mode driver 520 translate those drawing commands into hardware-specific commands and write them into the command buffer. If the runtime 510 or user mode driver 520 determine that a flush is needed (i.e., the command buffer must be submitted), the thread of execution designated by green arrows is followed: the command buffer is added to the waiting queue and submitted if possible, then a new command buffer is obtained so execution can continue. The medium shaded arrows trace the “typical” path taken by a command buffer after it is added to the waiting queue in step 535. If the command buffer can be dispatched immediately (the check 536), the command buffer is marked as ready 537 and selected for dispatch 540. Otherwise, the waiting command buffer must be marked Ready in the normal course of events, when the current running command buffer finishes execution. When a ready command buffer is selected for dispatch 540, the system kernel support 530 has the kernel driver 550 context switch the hardware to the appropriate context 551 and dispatch the command buffer to the hardware 552. The hardware then reads and executes the command buffer 561, until it is preempted or the command buffer finishes. If the command buffer completes normally 563, the hardware signals an interrupt and the interrupt service routine 553 executes. The ISR may wish to save the hardware context 554 at this time, although the driver may wish to defer this operation to the context switch 551, in case the hardware should be asked to execute two command buffers in a row that operate on the same context. After this step 554, the kernel system support 530 can free the resources needed by that command buffer 538, as well as signal any notification mechanisms such as events to let interested clients know that the command buffer is completed. After step 538, the kernel system support has two distinct tasks: it must reinitialize the newly available command buffer and add it to the initialized pool 533, and it must unblock any waiting command buffers and move them into the ready queue 539. After step 539, another command buffer can be selected for dispatch 540. The case of preemption is handled slightly differently than normal command buffer completion, as delineated by the orange arrows. Since the preempted command buffer did not finish, the system must not notify clients of its completion (step 538) or unblock dependent command buffers (step 539). Instead, the driver saves the context 554 of the partially executed command buffer such that it can be restarted where the preemption occurred, notifies any

Ex. C9 – Invalidity of U.S. Patent No. RE49,461 – Wilt

U.S. Patent No. RE49,461	Wilt
	clients 541 that need to know about the preemption, and selects the next Ready command buffer to dispatch 540. Examples of occurrences that might cause preemption include external events such as VSYNC or the arrival of the display refresh to a particular location such as a scanline, or expiration of a time quantum. In addition, for management of time quanta, the operating system may use a mechanism to preempt the graphics processor by calling the Kernel Driver 550. This would equate to expiration of a time quantum as arbitrated by the hardware, except that the complexity and responsibility for deciding when to preempt the graphics processor then rests with the System Kernel Support 530 rather than the Hardware 560. It is presumed that the system has previously set up a response to such an event, such as executing a series of Blt commands when the refresh passes a certain scanline (for a tear-free update). [0146]–[0153] To the extent Plaintiff contends that this claim element is not expressly or inherently disclosed, or is not obvious in view of Wilt and the knowledge of a POSITA, it would have been obvious to combine the teachings of Wilt with any of the prior art references in NVIDIA’s Invalidity Contentions and any supplements thereto and the relevant section of charts for other prior art for the ’461 patent in a manner that would result in the subject matter of this limitation given, at the very least, the person's understanding of the state of the art, the problems addressed and solved in the prior art, and the teachings of Wilt. Specific examples of such references are cited and explained in these invalidity contentions, including the cover pleading and charts C1–C15, and Appendix C.
21[e]: queueing a user command received by the user interaction stream during execution of the computations representing the artificial neural network; and	Wilt discloses queueing a user command received by the user interaction stream during execution of the computations representing the artificial neural network. <i>See e.g.:</i> Direct Memory Access (DMA) is a common method for moving data within computer systems, and has found application in the control of graphics processors as well as disk drives, network adapters, and external buses such as USB and IEEE 1394. DMA generally refers to the ability of a peripheral, such as a digital camera or disk controller, to read from or write to memory without directly involving the host processor. The host sets up the DMA transfer in advance, page locking the memory region where a digital camera, for example, is to write the image data from a picture it is about to take, and specifying that memory location to the

Ex. C9 – Invalidity of U.S. Patent No. RE49,461 – Wilt

U.S. Patent No. RE49,461	Wilt
	camera. Page locking causes the virtual memory manager to forego writing the memory locations in question out to disk. Once this is done, the camera can take the picture and write the image data into memory, then notify the host that the DMA transfer has been completed. In the context of graphics processor control, a series of commands for the graphics processor can be written into memory, and then transferred to the graphics processor via a DMA operation. DMA has many advantages over the other methods of submitting graphics processor commands. First, it facilitates parallel execution by the host and graphics processors. Since these ‘other operations’ to be performed by the CPU often involve generating more commands for the graphics processor, it is common to maintain multiple DMA buffers and ‘ping-pong’ between them, writing to one while the graphics processor reads and executes another. Second, DMA enables multiple applications to write command buffers into memory, and then submit the contents of the command buffers to the graphics processor when they are ready for execution. The graphics processor can only execute one command buffer at a time, but DMA facilitates the simultaneous construction of many command buffers because different threads can write into distinct memory regions that contain command buffers. Third, DMA frequently results in more efficient usage of the bus resources in the system. In other words, the bandwidth achievable through DMA is often higher than the bandwidth achievable by direct participation of the host microprocessor. The Accelerated Graphics Port (AGP) is a specialized bus designed specifically to serve as a high bandwidth conduit to transfer data from system memory to a graphics processor. For the purposes of the present description, AGP may be considered a special case of DMA. Besides the mechanism for delivering information to and receiving information from the hardware, another important distinction to make in our background discussion centers on notification of the host microprocessor that the graphics processor has completed some task, or that some event has occurred that is visible to the graphics processor. Non-limiting examples of tasks include finishing execution of a command buffer or the completion of a specific command. Non-limiting examples of events include vertical retrace of the display device, the completion of a field of digitized video from a video port, or the retracing of a specific scanline in the display device. An alternative method for notifying the host microprocessor of a completed task or an external event is known as an interrupt. An interrupt is a hardware mechanism that diverts the host microprocessor from whatever it is doing and transfers control to a special location in memory.

Ex. C9 – Invalidity of U.S. Patent No. RE49,461 – Wilt

U.S. Patent No. RE49,461	Wilt
	The operating system uses this mechanism to call driver code known as an interrupt service routine to respond to the interrupt. Interrupts have several advantages over polling based mechanisms. For one thing, the cost of setting up and responding to an interrupt is fixed, while the cost of a polling based mechanism increases approximately linearly with the number of polls performed. As a result, interrupt based schemes generally consume less bus bandwidth as well as fewer CPU cycles than polling based schemes. Interrupts can also result in a more timely notification if the interrupt latency is smaller than the time between when the event occurs and when the next poll is performed. [0085]–[0088], [0090]–[0091]. Given these assumptions, we can broaden our focus from graphics processors and contemplate a system with two processors: a host CPU, and a coprocessor that is controlled by sending command buffers from the CPU, that can send data back to the CPU, and that coordinates control by notifying the CPU of various events when they occur. The examples in this document frequently assume that the coprocessor is a graphics processor (as when describing possible synchronization problems and how a system can be designed to solve those problems efficiently), but these examples are for illustrative purposes only and do not limit the scope of this invention to graphics processors. In line with the above assumptions regarding computational resources in a graphics processing system, a non-limiting system is described below that makes efficient, shared use of the computational resources in a coprocessor. FIG. 2 depicts the life cycle of a command buffer as it is used by applications to send commands to the coprocessor, such as a graphics processor. When a command buffer is created and initialized, the buffer enters the initialized state 201. The command buffer is then available to be selected and written into by a thread that wishes to issue commands to the graphics processor. When a thread selects the command buffer, the memory in the command buffer is made available to that thread (perhaps by mapping the memory in the command buffer into that thread's address space) and the command buffer enters the Receiving state 202. As one of ordinary skill in the art can appreciate, the details of this mapping may vary. For example, the command buffer memory could be made visible to user mode, kernel mode, or both. In addition, there are numerous options as to the nature of the executable code that is

Ex. C9 – Invalidity of U.S. Patent No. RE49,461 – Wilt

U.S. Patent No. RE49,461	Wilt
	writing into the Receiving command buffer. In this regard, FIGS. 3A through 3E depict several of these options, i.e., several different architectures for writing into the command buffer. For purposes of describing FIGS. 3A through 3E, an application is a process, or a thread in a process, that implements some specific graphical application participating in the system. The runtime is middleware provided in the operating system that the application uses. Among other things, the runtime may validate the application's usage of an application programming interface (API), translate simple-to-use, high-level API calls into more complicated, lower-level constructs that are more readily translatable to hardware commands, and so on. The driver is a hardware-specific component that translates runtime-level commands to hardware-specific commands. The driver components depicted in FIGS. 3A through 3E are the portions of the driver that translate runtime-level commands to hardware-specific commands. Some portions of the driver that perform tasks other than this translation, such as interrupt handling and command buffer dispatch, would probably reside in kernel mode and are not contemplated in FIGS. 3A through 3E. Most of the object code depicted in FIGS. 3A through 3E may run in user mode, although some portions, as pointed out below, can run in kernel mode. The command buffer is the piece of memory that includes commands that can be consumed directly by the hardware via some DMA mechanism. Several of the figures also show a token stream, which is a hardware-independent stream of runtime-level primitives that can be translated into hardware-specific commands by the driver. Lastly, some of the figures show a JIT (Just-In-Time) compilation component that can translate a hardware-independent intermediate language into object code for execution by the host microprocessor. The boxes with OBJ in the corner include object code that can be directly executed by the microprocessor. Boxes with MSIL in the corner include intermediate code that is compiled by the JIT in order to execute on the microprocessor. If the application and runtime are both available on the client in intermediate form, the JIT can translate them into a unified piece of object code, shown as app executables 324, 334 and 345 FIGS. 3C to 3E, respectively, that implements the application. FIG. 3A shows the application 301, runtime 302 and driver 303 each performing their respective roles, calling into lower-level object code until the driver 303 writes hardware-specific commands into the command buffer 304. The application 301 could be an executable process, or a thread in a process, while the runtime 302 and driver 303 could each reside in DLLs. As depicted in FIG. 3A, all of the object code may reside in user mode (although some

Ex. C9 – Invalidity of U.S. Patent No. RE49,461 – Wilt

U.S. Patent No. RE49,461	Wilt
	or all of the runtime 302 and/or driver 303 may reside in kernel mode). If the runtime 302 determines that the command buffer 304 must be submitted before execution can continue, or the driver 303 determines that the command buffer 304 is out of space, a transition to kernel mode may be requested to submit the command buffer 304 and retrieve another one in order to continue. FIG. 3B shows a variant in which the runtime 312 emits a hardware-independent stream of commands into a buffer 313, which is then parsed by the driver 314 and written into a command buffer 315. Application 311 operates to call lower-level object code in connection with runtime 312. FIG. 3B approximates the current architecture of the DIRECTX® software component, in which the runtime translates application activity into DrawPrimitives2 (“DP2”) tokens. When the DP2 token stream is submitted, a kernel transition occurs and the driver 314 translates the DP2 token stream into hardware-specific commands in kernel mode. FIG. 3B does not make any assumptions about whether the driver 314 is in user mode or kernel mode, and in this regard, driver component 314 could reside in user mode as well as kernel mode. FIG. 3C shows a variant in which the application 321 and runtime 322 are synthesized into a unified piece of object code by a JIT compiler 323. The resulting piece of object code 324 then interacts with the driver 325 to write hardware-specific commands into the command buffer 326. FIG. 3D shows the JIT variant of the situation described for FIG. 3B, in which a hardware-independent token stream 335 is emitted by the software 334 above the driver 336. The driver 336 then translates the token stream to hardware-specific commands for the command buffer 337. As with FIG. 3C, application 331 and runtime 332 are synthesized into a unified piece of object code 334 by JIT compiler 333. FIG. 3E shows a system in which the application 341, runtime 342, and part of the driver 343 are all available on the client in intermediate language form MSIL. The JIT compiler 344 synthesizes these components into a unified piece of object code 345 specifically optimized to run on that particular client computer, taking into account not only the host microprocessor but also the coprocessor hardware configuration. Referring back to the command buffer life cycle depicted in FIG. 2, regardless of the method by which commands are written into the command buffer, the thread writing into the receiving command buffer eventually submits to the command buffer either because the command buffer has been filled, or because the drawing commands that have been written into the buffer

Ex. C9 – Invalidity of U.S. Patent No. RE49,461 – Wilt

U.S. Patent No. RE49,461	Wilt
	are executed forthwith. At that time, the command buffer enters the waiting state 203, and the contents of the command buffer may not be submitted to the hardware until all of the resources utilized to execute the command buffer are available. As an example of a circumstance that may force a wait, if an application draws into a surface, the drawing must be completed before that surface may be selected as a texture. Once the waiting criteria have been fulfilled, the command buffer enters the ready state 204, where the scheduler considers it for submission to the coprocessor. The scheduler has a great deal of flexibility in deciding which command buffer should be submitted to the hardware. The scheduler may take into account the priority, if any, of the command buffer, the priority of the requesting thread, the real-time requirements of the system, and so on. Once the command buffer has been selected as the next to run, the command buffer enters the standby state 205. On some systems, only one command buffer per coprocessor may be in the standby state 205 in the system. When the command buffer in the standby state 205 can be submitted to the hardware, the system switches the coprocessor context to the context required by the command buffer, and then submits the command buffer to the hardware. This operation entails some involvement from the driver, for example to initiate a DMA operation using the base address and size of the command buffer. The command buffer remains in the running state 206 until some event preempts execution of the command buffer or the entire contents of the command buffer have been read and executed. If the entire command buffer has been executed, the command buffer enters the terminated state 207, where it becomes available for the system to reinitialize and make available to another thread in the initialized state 201. If the command buffer is preempted, it is assumed that the hardware has the capability to save the hardware context in such a way that execution of the command buffer can be resumed where the preemption occurred. The command buffer can then be recycled into the ready state 204 and be re-submitted to the hardware to resume from where it stopped. The description below regarding preemptible command buffer execution describes this in more detail. [0095]–[0110] The command buffer scheduler 404 (“scheduler”) and kernel driver 405 work together in kernel mode to dispatch command buffers to the hardware 406 (the scheduler 404 decides

Ex. C9 – Invalidity of U.S. Patent No. RE49,461 – Wilt

U.S. Patent No. RE49,461	Wilt
	which command buffer should be dispatched, while the kernel driver 405 instructs the hardware 406 to dispatch a command buffer at the request of the scheduler). This system contemplates that the bulk of the driver logic would reside in the user mode driver DLL 403, not the kernel driver 405. While the User Mode Driver can contain large amounts of code that maps DDI-level calls into hardware-specific commands (which operation can be complicated and error-prone, especially when compiling a vertex and/or shader program), the Kernel Driver is comparatively small and simple, maximizing system robustness. FIG. 5 clarifies the sequence of events that occur when an application is making API calls to perform graphics operations. The block diagram components of FIG. 5 are shading coded: user mode components are of medium shade, kernel mode components are shaded lightly, and the hardware is shaded darkly. The command buffers are not specifically depicted in FIG. 5 as a hardware component; as per FIG. 4, the user mode driver 520 writes hardware-specific commands into the device's current command buffer, the command buffer scheduler (part of the system kernel support 530) dispatches the command buffer to the hardware 560 via the kernel mode driver 550, and finished command buffers are recycled for use by an application in the system. It is noted that multiple applications 500 can potentially share the pool of available command buffers, with the system kernel support 530 arbitrating sharing of that resource. When the application 500 initially creates a drawing context 501, the system kernel support 530 checks to see whether a new command buffer can be created 531. If so, the new command buffer is created 532 and initialized 533, and the thread obtains an initialized command buffer 534 before the Application 500 can perform drawing calls 502. If a command buffer could not be created in step 531, the application 500 must wait until an initialized command buffer becomes available 534. Once the application 500 has obtained a command buffer, the application 500, runtime 510 and user mode driver 520 enter the typical interaction between the three components that cause hardware-specific commands to be written into the command buffer. The drawing calls 502 from the application 500 are validated 511 by the runtime 510; a check 512 then determines whether a flush of the current command buffer is needed. If not, the drawing command is translated to a simpler, canonical DDI call 513 and passed to the user mode driver 520. The driver translates the DDI call into hardware specific commands and attempts to write them into the command buffer. If the check 522 for flush determines that there is no room in the command buffer, the command buffer must be submitted to the system

Ex. C9 – Invalidity of U.S. Patent No. RE49,461 – Wilt

U.S. Patent No. RE49,461	Wilt
	kernel support 530 and a new command buffer obtained from same before the command can be written and execution can continue. If either the runtime 510 or the user mode driver 520 determines that a flush is needed, per step 535 the command buffer is added to the waiting queue. At that time, the system kernel can check 536 whether the command buffer can be submitted forthwith (typically because no command buffer is running). If not, the command buffer is left in the waiting queue and a new command buffer must be obtained 534. It is noted that this functional block, which waits until a suitable initialized command buffer is available and then allocates it to the device, is identical to the operation needed by the application 500 before it can begin drawing. The arrows are also shading coded. The darkest arrows trace the “typical” thread execution of the application 500 as it initializes and performs drawing commands and the runtime 510 and user mode driver 520 translate those drawing commands into hardware-specific commands and write them into the command buffer. If the runtime 510 or user mode driver 520 determine that a flush is needed (i.e., the command buffer must be submitted), the thread of execution designated by green arrows is followed: the command buffer is added to the waiting queue and submitted if possible, then a new command buffer is obtained so execution can continue. The medium shaded arrows trace the “typical” path taken by a command buffer after it is added to the waiting queue in step 535. If the command buffer can be dispatched immediately (the check 536), the command buffer is marked as ready 537 and selected for dispatch 540. Otherwise, the waiting command buffer must be marked Ready in the normal course of events, when the current running command buffer finishes execution. When a ready command buffer is selected for dispatch 540, the system kernel support 530 has the kernel driver 550 context switch the hardware to the appropriate context 551 and dispatch the command buffer to the hardware 552. The hardware then reads and executes the command buffer 561, until it is preempted or the command buffer finishes. If the command buffer completes normally 563, the hardware signals an interrupt and the interrupt service routine 553 executes. The ISR may wish to save the hardware context 554 at this time, although the driver may wish to defer this operation to the context switch 551, in case the hardware should be asked to execute two command buffers in a row that operate on the same context. After this step 554, the kernel system support 530 can free the resources needed by that command buffer 538, as well as signal any notification mechanisms such as events to let interested clients know that the command buffer is completed. After step 538, the kernel system support has two

Ex. C9 – Invalidity of U.S. Patent No. RE49,461 – Wilt

U.S. Patent No. RE49,461	Wilt
	distinct tasks: it must reinitialize the newly available command buffer and add it to the initialized pool 533, and it must unblock any waiting command buffers and move them into the ready queue 539. After step 539, another command buffer can be selected for dispatch 540. The case of preemption is handled slightly differently than normal command buffer completion, as delineated by the orange arrows. Since the preempted command buffer did not finish, the system must not notify clients of its completion (step 538) or unblock dependent command buffers (step 539). Instead, the driver saves the context 554 of the partially executed command buffer such that it can be restarted where the preemption occurred, notifies any clients 541 that need to know about the preemption, and selects the next Ready command buffer to dispatch 540. Examples of occurrences that might cause preemption include external events such as VSYNC or the arrival of the display refresh to a particular location such as a scanline, or expiration of a time quantum. In addition, for management of time quanta, the operating system may use a mechanism to preempt the graphics processor by calling the Kernel Driver 550. This would equate to expiration of a time quantum as arbitrated by the hardware, except that the complexity and responsibility for deciding when to preempt the graphics processor then rests with the System Kernel Support 530 rather than the Hardware 560. It is presumed that the system has previously set up a response to such an event, such as executing a series of Blt commands when the refresh passes a certain scanline (for a tear-free update). Note that many of the ideas presented above can be applied to the system described in connection with FIGS. 4 and 5. For example, per the above, the system kernel support 540 could potentially take over the context switching task 551 by having the kernel driver 550 dispatch a context switch command buffer. For another example, lightweight contexts could be supported in this system by differentiating between context types and, potentially, types of command buffers, as described in further detail above. For another example, step 540 (“Select ready command buffer for dispatch”) could be modified slightly to take advantage of the ideas presented above, in which the hardware begins executing a previously-selected command buffer as soon as the running command buffer finishes. For yet another example, an efficient way for the runtime 510 and user mode Driver 520 to detect command buffer overflow 522 would be for the system kernel support 530 to allocate an unwriteable guard page at the end of each command buffer, and use structured exception handling to detect when a write to the guard page has been attempted. Finally, the ideas of making the system robust in the face of

Ex. C9 – Invalidity of U.S. Patent No. RE49,461 – Wilt

U.S. Patent No. RE49,461	Wilt
	invalid command buffers could be readily incorporated into this system, either by signaling an exception when an invalid command caused the hardware to generate an interrupt, or by resetting the hardware via a special kernel driver 550 entry point if it became unresponsive for too long. [0146]–[0154] As described in the background, there are many present and potential sources of increased demand for graphics processors' computational resources. When more than one of these sources demands access to a graphics processor at approximately the same time, the graphics processor may not be free to serve both sources in a way that meets the demand satisfactorily. Thus, the present invention provides various ways to manage the scheduling and dispatch of graphics processing requests by multiple applications in order to ensure the computational resources of coprocessor units, such as graphics processing units, are efficiently distributed to the requesting applications. Accordingly, the present invention provides methods and systems for controlling the computation resources of coprocessor(s) in a host computing system, wherein the coprocessor(s) are controlled with command buffers submitted by the host processor and wherein the scheduling of the transmission of the command buffers to the coprocessor(s) is performed by an object, such as the operating system, of the host computing system, making the computational resources of the coprocessor(s) simultaneously available to multiple applications. [0024] With respect to queueing of command buffers, due to the high performance of graphics hardware, it is critical to minimize the latency between command buffer completion and the submission of another command buffer. Otherwise, the graphics hardware wastes valuable time waiting for another command buffer to be submitted. One possible solution to this problem is to queue up a context switch and command buffer dispatch before the current command buffer has finished executing. Then, provided the hardware and driver support this capability, the hardware can immediately begin execution of another command buffer when the current command buffer has completed. The system then has more time to prepare another command buffer for execution: the graphics hardware will only go idle if it completes the new command buffer before another one has been queued up.

Ex. C9 – Invalidity of U.S. Patent No. RE49,461 – Wilt

U.S. Patent No. RE49,461	Wilt
	With respect to preemptible command buffer execution, the system described here provides for hardware support for preemption. A variety of events could potentially cause preemption of a command buffer during execution, ranging from an external event such as VSYNC to hardware enforcement of a time quantum. To implement this feature, the hardware and driver would have to enable the hardware context to be saved, such that execution of the preempted command buffer can be resumed where it left off; and the operating system would have to be notified so it could put the command buffer back in the Ready state mentioned in FIG. 2. One reason to implement such preemption is to guarantee that a display frame is composed immediately after VSYNC (to make sure it will be ready when the next VSYNC occurs). Hardware enforcement of a time quantum would cause a command buffer to be preempted after executing for some set amount of time, such as 2 milliseconds. If the command buffer contains very expensive commands, such as drawing large, texture-mapped, alpha blended triangles, the hardware may have to preempt its execution to enable other command buffers to be considered for submission. [0126]–[0127] Direct Memory Access (DMA) is a common method for moving data within computer systems, and has found application in the control of graphics processors as well as disk drives, network adapters, and external buses such as USB and IEEE 1394. DMA generally refers to the ability of a peripheral, such as a digital camera or disk controller, to read from or write to memory without directly involving the host processor. The host sets up the DMA transfer in advance, page locking the memory region where a digital camera, for example, is to write the image data from a picture it is about to take, and specifying that memory location to the camera. Page locking causes the virtual memory manager to forego writing the memory locations in question out to disk. Once this is done, the camera can take the picture and write the image data into memory, then notify the host that the DMA transfer has been completed. In the context of graphics processor control, a series of commands for the graphics processor can be written into memory, and then transferred to the graphics processor via a DMA operation. DMA has many advantages over the other methods of submitting graphics processor commands. First, it facilitates parallel execution by the host and graphics processors. Since these ‘other operations’ to be performed by the CPU often involve generating more commands for the graphics processor, it is common to maintain multiple DMA buffers and ‘ping-pong’

Ex. C9 – Invalidity of U.S. Patent No. RE49,461 – Wilt

U.S. Patent No. RE49,461	Wilt
	between them, writing to one while the graphics processor reads and executes another. Second, DMA enables multiple applications to write command buffers into memory, and then submit the contents of the command buffers to the graphics processor when they are ready for execution. The graphics processor can only execute one command buffer at a time, but DMA facilitates the simultaneous construction of many command buffers because different threads can write into distinct memory regions that contain command buffers. Third, DMA frequently results in more efficient usage of the bus resources in the system. In other words, the bandwidth achievable through DMA is often higher than the bandwidth achievable by direct participation of the host microprocessor. The Accelerated Graphics Port (AGP) is a specialized bus designed specifically to serve as a high bandwidth conduit to transfer data from system memory to a graphics processor. For the purposes of the present description, AGP may be considered a special case of DMA. [0085]–[0086] A first assumption made in accordance with the present invention is to use predominantly DMA-based input. This first assumption is that the majority of commands to the graphics processor should be submitted via a DMA mechanism. As described above, DMA-based command buffer submission facilitates the eventual consumption by the graphics processor of multiple DMA buffers generated by multiple applications running on the host microprocessor(s). Lower-frequency commands, such as initiating the execution of a DMA command stream, can be sent to the hardware via a method such as MMIO. But high frequency commands for the graphics chip, such as loading image data, commands to draw features such as lines and triangles, and commands to change hardware state, for the most part should be sent via DMA. [0093] To the extent Plaintiff contends that this claim element is not expressly or inherently disclosed, or is not obvious in view of Wilt and the knowledge of a POSITA, it would have been obvious to combine the teachings of Wilt with any of the prior art references in NVIDIA’s Invalidity Contentions and any supplements thereto and the relevant section of charts for other prior art for the ’461 patent in a manner that would

Ex. C9 – Invalidity of U.S. Patent No. RE49,461 – Wilt

U.S. Patent No. RE49,461	Wilt
	result in the subject matter of this limitation given, at the very least, the person's understanding of the state of the art, the problems addressed and solved in the prior art, and the teachings of Wilt. Specific examples of such references are cited and explained in these invalidity contentions, including the cover pleading and charts C1–C15, and Appendix C.
21[f]: executing the user command during execution of the computations representing the artificial neural network at times determined by the computational stream.	Wilt discloses executing the user command during execution of the computations representing the artificial neural network at times determined by the computational stream. <i>See e.g.</i>: Direct Memory Access (DMA) is a common method for moving data within computer systems, and has found application in the control of graphics processors as well as disk drives, network adapters, and external buses such as USB and IEEE 1394. DMA generally refers to the ability of a peripheral, such as a digital camera or disk controller, to read from or write to memory without directly involving the host processor. The host sets up the DMA transfer in advance, page locking the memory region where a digital camera, for example, is to write the image data from a picture it is about to take, and specifying that memory location to the camera. Page locking causes the virtual memory manager to forego writing the memory locations in question out to disk. Once this is done, the camera can take the picture and write the image data into memory, then notify the host that the DMA transfer has been completed. In the context of graphics processor control, a series of commands for the graphics processor can be written into memory, and then transferred to the graphics processor via a DMA operation. DMA has many advantages over the other methods of submitting graphics processor commands. First, it facilitates parallel execution by the host and graphics processors. Since these ‘other operations’ to be performed by the CPU often involve generating more commands for the graphics processor, it is common to maintain multiple DMA buffers and ‘ping-pong’ between them, writing to one while the graphics processor reads and executes another. Second, DMA enables multiple applications to write command buffers into memory, and then submit the contents of the command buffers to the graphics processor when they are ready for execution. The graphics processor can only execute one command buffer at a time, but DMA facilitates the simultaneous construction of many command buffers because different threads can write into distinct memory regions that contain command buffers. Third, DMA frequently results in more efficient usage of the bus resources in the system. In other words, the

Ex. C9 – Invalidity of U.S. Patent No. RE49,461 – Wilt

U.S. Patent No. RE49,461	Wilt
	bandwidth achievable through DMA is often higher than the bandwidth achievable by direct participation of the host microprocessor. The Accelerated Graphics Port (AGP) is a specialized bus designed specifically to serve as a high bandwidth conduit to transfer data from system memory to a graphics processor. For the purposes of the present description, AGP may be considered a special case of DMA. [0085]–[0087] Referring back to the command buffer life cycle depicted in FIG. 2, regardless of the method by which commands are written into the command buffer, the thread writing into the receiving command buffer eventually submits to the command buffer either because the command buffer has been filled, or because the drawing commands that have been written into the buffer are executed forthwith. At that time, the command buffer enters the waiting state 203, and the contents of the command buffer may not be submitted to the hardware until all of the resources utilized to execute the command buffer are available. As an example of a circumstance that may force a wait, if an application draws into a surface, the drawing must be completed before that surface may be selected as a texture. Once the waiting criteria have been fulfilled, the command buffer enters the ready state 204, where the scheduler considers it for submission to the coprocessor. The scheduler has a great deal of flexibility in deciding which command buffer should be submitted to the hardware. The scheduler may take into account the priority, if any, of the command buffer, the priority of the requesting thread, the real-time requirements of the system, and so on. Once the command buffer has been selected as the next to run, the command buffer enters the standby state 205. On some systems, only one command buffer per coprocessor may be in the standby state 205 in the system. When the command buffer in the standby state 205 can be submitted to the hardware, the system switches the coprocessor context to the context required by the command buffer, and then submits the command buffer to the hardware. This operation entails some involvement from the driver, for example to initiate a DMA operation using the base address and size of the command buffer. The command buffer remains in the running state 206 until some event preempts execution of the command buffer or the entire contents of the command buffer have been read and executed.

Ex. C9 – Invalidity of U.S. Patent No. RE49,461 – Wilt

U.S. Patent No. RE49,461	Wilt
	If the entire command buffer has been executed, the command buffer enters the terminated state 207, where it becomes available for the system to reinitialize and make available to another thread in the initialized state 201. If the command buffer is preempted, it is assumed that the hardware has the capability to save the hardware context in such a way that execution of the command buffer can be resumed where the preemption occurred. The command buffer can then be recycled into the ready state 204 and be re-submitted to the hardware to resume from where it stopped. The description below regarding preemptible command buffer execution describes this in more detail. [0106]–[0110] With respect to queueing of command buffers, due to the high performance of graphics hardware, it is critical to minimize the latency between command buffer completion and the submission of another command buffer. Otherwise, the graphics hardware wastes valuable time waiting for another command buffer to be submitted. One possible solution to this problem is to queue up a context switch and command buffer dispatch before the current command buffer has finished executing. Then, provided the hardware and driver support this capability, the hardware can immediately begin execution of another command buffer when the current command buffer has completed. The system then has more time to prepare another command buffer for execution: the graphics hardware will only go idle if it completes the new command buffer before another one has been queued up. With respect to preemptible command buffer execution, the system described here provides for hardware support for preemption. A variety of events could potentially cause preemption of a command buffer during execution, ranging from an external event such as VSYNC to hardware enforcement of a time quantum. To implement this feature, the hardware and driver would have to enable the hardware context to be saved, such that execution of the preempted command buffer can be resumed where it left off; and the operating system would have to be notified so it could put the command buffer back in the Ready state mentioned in FIG. 2. One reason to implement such preemption is to guarantee that a display frame is composed immediately after VSYNC (to make sure it will be ready when the next VSYNC occurs). Hardware enforcement of a time quantum would cause a command buffer to be preempted after executing for some set amount of time, such as 2 milliseconds. If the command buffer contains very expensive commands, such as drawing large, texture-mapped, alpha blended

Ex. C9 – Invalidity of U.S. Patent No. RE49,461 – Wilt

U.S. Patent No. RE49,461	Wilt
	triangles, the hardware may have to preempt its execution to enable other command buffers to be considered for submission. [0126]–[0127] The case of preemption is handled slightly differently than normal command buffer completion, as delineated by the orange arrows. Since the preempted command buffer did not finish, the system must not notify clients of its completion (step 538) or unblock dependent command buffers (step 539). Instead, the driver saves the context 554 of the partially executed command buffer such that it can be restarted where the preemption occurred, notifies any clients 541 that need to know about the preemption, and selects the next Ready command buffer to dispatch 540. Examples of occurrences that might cause preemption include external events such as VSYNC or the arrival of the display refresh to a particular location such as a scanline, or expiration of a time quantum. In addition, for management of time quanta, the operating system may use a mechanism to preempt the graphics processor by calling the Kernel Driver 550. This would equate to expiration of a time quantum as arbitrated by the hardware, except that the complexity and responsibility for deciding when to preempt the graphics processor then rests with the System Kernel Support 530 rather than the Hardware 560. It is presumed that the system has previously set up a response to such an event, such as executing a series of Blt commands when the refresh passes a certain scanline (for a tear-free update). [0152]–[0153] To the extent Plaintiff contends that this claim element is not expressly or inherently disclosed, or is not obvious in view of Wilt and the knowledge of a POSITA, it would have been obvious to combine the teachings of Wilt with any of the prior art references in NVIDIA’s Invalidity Contentions and any supplements thereto and the relevant section of charts for other prior art for the ’461 patent in a manner that would result in the subject matter of this limitation given, at the very least, the person’s understanding of the state of the art, the problems addressed and solved in the prior art, and the teachings of Wilt. Specific examples of such references are cited and explained in these invalidity contentions, including the cover pleading and charts C1–C15, and Appendix C.

Ex. C9 – Invalidity of U.S. Patent No. RE49,461 – Wilt

U.S. Patent No. RE49,461	Wilt
22. The method of claim 21, wherein the user interaction stream controls the data exchange between the user interaction stream and the computational stream outside of execution of the computations representing the artificial neural network.	Wilt discloses the method of claim 21, wherein the user interaction stream controls the data exchange between the user interaction stream and the computational stream outside of execution of the computations representing the artificial neural network. See claim 21. Given these assumptions, we can broaden our focus from graphics processors and contemplate a system with two processors: a host CPU, and a coprocessor that is controlled by sending command buffers from the CPU, that can send data back to the CPU, and that coordinates control by notifying the CPU of various events when they occur. The examples in this document frequently assume that the coprocessor is a graphics processor (as when describing possible synchronization problems and how a system can be designed to solve those problems efficiently), but these examples are for illustrative purposes only and do not limit the scope of this invention to graphics processors. In line with the above assumptions regarding computational resources in a graphics processing system, a non-limiting system is described below that makes efficient, shared use of the computational resources in a coprocessor. FIG. 2 depicts the life cycle of a command buffer as it is used by applications to send commands to the coprocessor, such as a graphics processor. When a command buffer is created and initialized, the buffer enters the initialized state 201. The command buffer is then available to be selected and written into by a thread that wishes to issue commands to the graphics processor. When a thread selects the command buffer, the memory in the command buffer is made available to that thread (perhaps by mapping the memory in the command buffer into that thread's address space) and the command buffer enters the Receiving state 202. As one of ordinary skill in the art can appreciate, the details of this mapping may vary. For example, the command buffer memory could be made visible to user mode, kernel mode, or both. In addition, there are numerous options as to the nature of the executable code that is writing into the Receiving command buffer. In this regard, FIGS. 3A through 3E depict several of these options, i.e., several different architectures for writing into the command buffer. For purposes of describing FIGS. 3A through 3E, an application is a process, or a thread in a process, that implements some specific graphical application participating in the system. The runtime is middleware provided in the operating system that the application uses. Among other

Ex. C9 – Invalidity of U.S. Patent No. RE49,461 – Wilt

U.S. Patent No. RE49,461	Wilt
	things, the runtime may validate the application's usage of an application programming interface (API), translate simple-to-use, high-level API calls into more complicated, lower-level constructs that are more readily translatable to hardware commands, and so on. The driver is a hardware-specific component that translates runtime-level commands to hardware-specific commands. The driver components depicted in FIGS. 3A through 3E are the portions of the driver that translate runtime-level commands to hardware-specific commands. Some portions of the driver that perform tasks other than this translation, such as interrupt handling and command buffer dispatch, would probably reside in kernel mode and are not contemplated in FIGS. 3A through 3E. Most of the object code depicted in FIGS. 3A through 3E may run in user mode, although some portions, as pointed out below, can run in kernel mode. The command buffer is the piece of memory that includes commands that can be consumed directly by the hardware via some DMA mechanism. Several of the figures also show a token stream, which is a hardware-independent stream of runtime-level primitives that can be translated into hardware-specific commands by the driver. Lastly, some of the figures show a JIT (Just-In-Time) compilation component that can translate a hardware-independent intermediate language into object code for execution by the host microprocessor. The boxes with OBJ in the corner include object code that can be directly executed by the microprocessor. Boxes with MSIL in the corner include intermediate code that is compiled by the JIT in order to execute on the microprocessor. If the application and runtime are both available on the client in intermediate form, the JIT can translate them into a unified piece of object code, shown as app executables 324, 334 and 345 FIGS. 3C to 3E, respectively, that implements the application. FIG. 3A shows the application 301, runtime 302 and driver 303 each performing their respective roles, calling into lower-level object code until the driver 303 writes hardware-specific commands into the command buffer 304. The application 301 could be an executable process, or a thread in a process, while the runtime 302 and driver 303 could each reside in DLLs. As depicted in FIG. 3A, all of the object code may reside in user mode (although some or all of the runtime 302 and/or driver 303 may reside in kernel mode). If the runtime 302 determines that the command buffer 304 must be submitted before execution can continue, or the driver 303 determines that the command buffer 304 is out of space, a transition to kernel mode may be requested to submit the command buffer 304 and retrieve another one in order to continue.

Ex. C9 – Invalidity of U.S. Patent No. RE49,461 – Wilt

U.S. Patent No. RE49,461	Wilt
	FIG. 3B shows a variant in which the runtime 312 emits a hardware-independent stream of commands into a buffer 313, which is then parsed by the driver 314 and written into a command buffer 315. Application 311 operates to call lower-level object code in connection with runtime 312. FIG. 3B approximates the current architecture of the DIRECTX® software component, in which the runtime translates application activity into DrawPrimitives2 (“DP2”) tokens. When the DP2 token stream is submitted, a kernel transition occurs and the driver 314 translates the DP2 token stream into hardware-specific commands in kernel mode. FIG. 3B does not make any assumptions about whether the driver 314 is in user mode or kernel mode, and in this regard, driver component 314 could reside in user mode as well as kernel mode. FIG. 3C shows a variant in which the application 321 and runtime 322 are synthesized into a unified piece of object code by a JIT compiler 323. The resulting piece of object code 324 then interacts with the driver 325 to write hardware-specific commands into the command buffer 326. FIG. 3D shows the JIT variant of the situation described for FIG. 3B, in which a hardware-independent token stream 335 is emitted by the software 334 above the driver 336. The driver 336 then translates the token stream to hardware-specific commands for the command buffer 337. As with FIG. 3C, application 331 and runtime 332 are synthesized into a unified piece of object code 334 by JIT compiler 333. FIG. 3E shows a system in which the application 341, runtime 342, and part of the driver 343 are all available on the client in intermediate language form MSIL. The JIT compiler 344 synthesizes these components into a unified piece of object code 345 specifically optimized to run on that particular client computer, taking into account not only the host microprocessor but also the coprocessor hardware configuration. Referring back to the command buffer life cycle depicted in FIG. 2, regardless of the method by which commands are written into the command buffer, the thread writing into the receiving command buffer eventually submits to the command buffer either because the command buffer has been filled, or because the drawing commands that have been written into the buffer are executed forthwith. At that time, the command buffer enters the waiting state 203, and the contents of the command buffer may not be submitted to the hardware until all of the resources utilized to execute the command buffer are available. As an example of a circumstance that may force a wait, if an application draws into a surface, the drawing must be completed before that surface may be selected as a texture.

Ex. C9 – Invalidity of U.S. Patent No. RE49,461 – Wilt

U.S. Patent No. RE49,461	Wilt
	Once the waiting criteria have been fulfilled, the command buffer enters the ready state 204, where the scheduler considers it for submission to the coprocessor. The scheduler has a great deal of flexibility in deciding which command buffer should be submitted to the hardware. The scheduler may take into account the priority, if any, of the command buffer, the priority of the requesting thread, the real-time requirements of the system, and so on. Once the command buffer has been selected as the next to run, the command buffer enters the standby state 205. On some systems, only one command buffer per coprocessor may be in the standby state 205 in the system. When the command buffer in the standby state 205 can be submitted to the hardware, the system switches the coprocessor context to the context required by the command buffer, and then submits the command buffer to the hardware. This operation entails some involvement from the driver, for example to initiate a DMA operation using the base address and size of the command buffer. The command buffer remains in the running state 206 until some event preempts execution of the command buffer or the entire contents of the command buffer have been read and executed. If the entire command buffer has been executed, the command buffer enters the terminated state 207, where it becomes available for the system to reinitialize and make available to another thread in the initialized state 201. If the command buffer is preempted, it is assumed that the hardware has the capability to save the hardware context in such a way that execution of the command buffer can be resumed where the preemption occurred. The command buffer can then be recycled into the ready state 204 and be re-submitted to the hardware to resume from where it stopped. The description below regarding preemptible command buffer execution describes this in more detail. With respect to context switching, the system described above can accommodate hardware-accelerated, or at least driver-facilitated, context switching, in which hardware contexts are maintained by the driver and context switching may be supported natively in the hardware. Contexts may be cached in local video memory or other high speed memory, for example. In this case, the operating system requests that the driver load a given context into the hardware. The context switch and command buffer dispatch may be specified in a single DDI (Device Driver Interface) call for performance reasons. Alternatively, the operating system may maintain a canonical notion of the hardware context and construct a command buffer that sets the hardware to the state corresponding to a given

Ex. C9 – Invalidity of U.S. Patent No. RE49,461 – Wilt

U.S. Patent No. RE49,461	Wilt
	context. In this case, loading the context during the transition from the standby state 205 to the running state 206 entails dispatching such a constructed command buffer to the hardware. Alternatively, the head of a command buffer could be reserved for an optional context switch. If the context switch is deemed unnecessary by the operating system, for example, because the hardware already contains the appropriate context, the system may specify the command just beyond the context switch block as the base address of the command buffer. Whether a separate command buffer, head of a command buffer, or other mechanism is employed by the operating system to load context into the hardware before beginning execution, driver complexity is reduced since the driver would only know that it was dispatching command buffers to the hardware, without knowing whether they were loading hardware context or causing drawing commands to be executed. Driver complexity is reduced, however, at a potential performance cost since native hardware support for context switching using fast memory may have performance advantages over command buffer based context switching. Such a system would be more amenable to innovations such as the addition of lightweight context switching on existing drivers, however. With respect to lightweight contexts, the hardware context includes a great deal of state that may not be needed for certain classes of applications. For example, if the coprocessor is a graphics processor, the full multitexturing, vertex shading, and pixel shading state of a modern graphics chip may not be needed for simple 2D graphics applications. To accommodate such applications, the system could define one or more lightweight context types that would be more efficient to load into the hardware before beginning execution of a command buffer. In addition, the system could potentially remember which portions of the heavyweight context state was unaffected by the loading of a lightweight context, and optimize subsequent context switches to avoid re-loading state that was already present in the hardware. Such optimizations could be performed either by a system that required formal support of context switching at the driver level, or that loaded hardware context manually using a command buffer or other mechanism. With respect to interrupt handling, interrupts may be used for efficient management of coprocessors' computational resources because they facilitate the host microprocessor's performance of useful work while the coprocessor hardware is running. The presently described embodiments detail some of the applications for interrupts in the system and the features that interrupts would enable in each case. In all cases, interrupts present an

Ex. C9 – Invalidity of U.S. Patent No. RE49,461 – Wilt

U.S. Patent No. RE49,461	Wilt
	opportunity for the operating system to notify clients of the event that caused the interrupt, whether by writing memory or signaling a thread synchronization object such as an event. For interrupts on command buffer completion, when a command buffer is finished executing, the system is notified so it can select another command buffer to submit to the hardware. Whether the coprocessor is idle from when it signals the interrupt until another command buffer is submitted, or if the coprocessor begins executing a previously-queued command buffer (as described below), the system must be notified so it can select another command buffer for submission (or eventual submission) to the hardware. Since the interrupt signals that the command buffer has been executed, the system can release all of the resources that were needed for the command buffer and unblocks pending command buffers that need those resources, and can also recycle the command buffer for use by another application. [0095]–[0115] The command buffer scheduler 404 (“scheduler”) and kernel driver 405 work together in kernel mode to dispatch command buffers to the hardware 406 (the scheduler 404 decides which command buffer should be dispatched, while the kernel driver 405 instructs the hardware 406 to dispatch a command buffer at the request of the scheduler). This system contemplates that the bulk of the driver logic would reside in the user mode driver DLL 403, not the kernel driver 405. While the User Mode Driver can contain large amounts of code that maps DDI-level calls into hardware-specific commands (which operation can be complicated and error-prone, especially when compiling a vertex and/or shader program), the Kernel Driver is comparatively small and simple, maximizing system robustness. FIG. 5 clarifies the sequence of events that occur when an application is making API calls to perform graphics operations. The block diagram components of FIG. 5 are shading coded: user mode components are of medium shade, kernel mode components are shaded lightly, and the hardware is shaded darkly. The command buffers are not specifically depicted in FIG. 5 as a hardware component; as per FIG. 4, the user mode driver 520 writes hardware-specific commands into the device's current command buffer, the command buffer scheduler (part of the system kernel support 530) dispatches the command buffer to the hardware 560 via the kernel mode driver 550, and finished command buffers are recycled for use by an application in the system. It is noted that multiple applications 500 can potentially share the pool of

Ex. C9 – Invalidity of U.S. Patent No. RE49,461 – Wilt

U.S. Patent No. RE49,461	Wilt
	available command buffers, with the system kernel support 530 arbitrating sharing of that resource. When the application 500 initially creates a drawing context 501, the system kernel support 530 checks to see whether a new command buffer can be created 531. If so, the new command buffer is created 532 and initialized 533, and the thread obtains an initialized command buffer 534 before the Application 500 can perform drawing calls 502. If a command buffer could not be created in step 531, the application 500 must wait until an initialized command buffer becomes available 534. Once the application 500 has obtained a command buffer, the application 500, runtime 510 and user mode driver 520 enter the typical interaction between the three components that cause hardware-specific commands to be written into the command buffer. The drawing calls 502 from the application 500 are validated 511 by the runtime 510; a check 512 then determines whether a flush of the current command buffer is needed. If not, the drawing command is translated to a simpler, canonical DDI call 513 and passed to the user mode driver 520. The driver translates the DDI call into hardware specific commands and attempts to write them into the command buffer. If the check 522 for flush determines that there is no room in the command buffer, the command buffer must be submitted to the system kernel support 530 and a new command buffer obtained from same before the command can be written and execution can continue. If either the runtime 510 or the user mode driver 520 determines that a flush is needed, per step 535 the command buffer is added to the waiting queue. At that time, the system kernel can check 536 whether the command buffer can be submitted forthwith (typically because no command buffer is running). If not, the command buffer is left in the waiting queue and a new command buffer must be obtained 534. It is noted that this functional block, which waits until a suitable initialized command buffer is available and then allocates it to the device, is identical to the operation needed by the application 500 before it can begin drawing. The arrows are also shading coded. The darkest arrows trace the “typical” thread execution of the application 500 as it initializes and performs drawing commands and the runtime 510 and user mode driver 520 translate those drawing commands into hardware-specific commands and write them into the command buffer. If the runtime 510 or user mode driver 520 determine that a flush is needed (i.e., the command buffer must be submitted), the thread of execution designated by green arrows is followed: the command buffer is added to the waiting queue and submitted if possible, then a new command buffer is obtained so execution can continue.

Ex. C9 – Invalidity of U.S. Patent No. RE49,461 – Wilt

U.S. Patent No. RE49,461	Wilt
	The medium shaded arrows trace the “typical” path taken by a command buffer after it is added to the waiting queue in step 535. If the command buffer can be dispatched immediately (the check 536), the command buffer is marked as ready 537 and selected for dispatch 540. Otherwise, the waiting command buffer must be marked Ready in the normal course of events, when the current running command buffer finishes execution. When a ready command buffer is selected for dispatch 540, the system kernel support 530 has the kernel driver 550 context switch the hardware to the appropriate context 551 and dispatch the command buffer to the hardware 552. The hardware then reads and executes the command buffer 561, until it is preempted or the command buffer finishes. If the command buffer completes normally 563, the hardware signals an interrupt and the interrupt service routine 553 executes. The ISR may wish to save the hardware context 554 at this time, although the driver may wish to defer this operation to the context switch 551, in case the hardware should be asked to execute two command buffers in a row that operate on the same context. After this step 554, the kernel system support 530 can free the resources needed by that command buffer 538, as well as signal any notification mechanisms such as events to let interested clients know that the command buffer is completed. After step 538, the kernel system support has two distinct tasks: it must reinitialize the newly available command buffer and add it to the initialized pool 533, and it must unblock any waiting command buffers and move them into the ready queue 539. After step 539, another command buffer can be selected for dispatch 540. The case of preemption is handled slightly differently than normal command buffer completion, as delineated by the orange arrows. Since the preempted command buffer did not finish, the system must not notify clients of its completion (step 538) or unblock dependent command buffers (step 539). Instead, the driver saves the context 554 of the partially executed command buffer such that it can be restarted where the preemption occurred, notifies any clients 541 that need to know about the preemption, and selects the next Ready command buffer to dispatch 540. Examples of occurrences that might cause preemption include external events such as VSYNC or the arrival of the display refresh to a particular location such as a scanline, or expiration of a time quantum. In addition, for management of time quanta, the operating system may use a mechanism to preempt the graphics processor by calling the Kernel Driver 550. This would equate to expiration of a time quantum as arbitrated by the hardware, except that the complexity and responsibility for deciding when to preempt the graphics processor then rests

Ex. C9 – Invalidity of U.S. Patent No. RE49,461 – Wilt

U.S. Patent No. RE49,461	Wilt
	with the System Kernel Support 530 rather than the Hardware 560. It is presumed that the system has previously set up a response to such an event, such as executing a series of Blt commands when the refresh passes a certain scanline (for a tear-free update). [0146]–[0153] Direct Memory Access (DMA) is a common method for moving data within computer systems, and has found application in the control of graphics processors as well as disk drives, network adapters, and external buses such as USB and IEEE 1394. DMA generally refers to the ability of a peripheral, such as a digital camera or disk controller, to read from or write to memory without directly involving the host processor. The host sets up the DMA transfer in advance, page locking the memory region where a digital camera, for example, is to write the image data from a picture it is about to take, and specifying that memory location to the camera. Page locking causes the virtual memory manager to forego writing the memory locations in question out to disk. Once this is done, the camera can take the picture and write the image data into memory, then notify the host that the DMA transfer has been completed. In the context of graphics processor control, a series of commands for the graphics processor can be written into memory, and then transferred to the graphics processor via a DMA operation. DMA has many advantages over the other methods of submitting graphics processor commands. First, it facilitates parallel execution by the host and graphics processors. Since these ‘other operations’ to be performed by the CPU often involve generating more commands for the graphics processor, it is common to maintain multiple DMA buffers and ‘ping-pong’ between them, writing to one while the graphics processor reads and executes another. Second, DMA enables multiple applications to write command buffers into memory, and then submit the contents of the command buffers to the graphics processor when they are ready for execution. The graphics processor can only execute one command buffer at a time, but DMA facilitates the simultaneous construction of many command buffers because different threads can write into distinct memory regions that contain command buffers. Third, DMA frequently results in more efficient usage of the bus resources in the system. In other words, the bandwidth achievable through DMA is often higher than the bandwidth achievable by direct participation of the host microprocessor.

Ex. C9 – Invalidity of U.S. Patent No. RE49,461 – Wilt

U.S. Patent No. RE49,461	Wilt
	The Accelerated Graphics Port (AGP) is a specialized bus designed specifically to serve as a high bandwidth conduit to transfer data from system memory to a graphics processor. For the purposes of the present description, AGP may be considered a special case of DMA. [0085]–[0087] To the extent Plaintiff contends that this claim element is not expressly or inherently disclosed, or is not obvious in view of Wilt and the knowledge of a POSITA, it would have been obvious to combine the teachings of Wilt with any of the prior art references in NVIDIA’s Invalidity Contentions and any supplements thereto and the relevant section of charts for other prior art for the ’461 patent in a manner that would result in the subject matter of this limitation given, at the very least, the person's understanding of the state of the art, the problems addressed and solved in the prior art, and the teachings of Wilt. Specific examples of such references are cited and explained in these invalidity contentions, including the cover pleading and charts C1–C15, and Appendix C.
23. The method of claim 21, wherein executing the user interaction stream comprises: controlling setting and editing of computational elements of the computations representing the artificial neural network.	Wilt discloses the method of claim 21, wherein executing the user interaction stream comprises controlling setting and editing of computational elements of the computations representing the artificial neural network. See claim 21.

U.S. Patent No. RE49,461	Wilt
	Computing Environment 100 FIG. 1B FIG. 1B. Computer 110 typically includes a variety of computer readable media. Computer readable media can be any available media that can be accessed by computer 110 and includes both volatile and nonvolatile media, removable and non-removable media. By way of example, and not limitation, computer readable media may comprise computer storage media and communication media. Computer storage media includes both volatile and nonvolatile, removable and non-removable media implemented in any method or technology for storage of information such as computer readable instructions, data structures, program modules or other data. Computer storage media includes, but is not limited to, RAM, ROM, EEPROM, flash memory or other memory technology, CDROM, digital versatile disks (DVD) or other optical disk storage, magnetic cassettes, magnetic tape, magnetic disk storage or other magnetic storage devices, or any other medium which can be used to store the desired information and

Ex. C9 – Invalidity of U.S. Patent No. RE49,461 – Wilt

U.S. Patent No. RE49,461	Wilt
	which can accessed by computer 110. Communication media typically embodies computer readable instructions, data structures, program modules or other data in a modulated data signal such as a carrier wave or other transport mechanism and includes any information delivery media. The term “modulated data signal” means a signal that has one or more of its characteristics set or changed in such a manner as to encode information in the signal. By way of example, and not limitation, communication media includes wired media such as a wired network or direct-wired connection, and wireless media such as acoustic, RF, infrared and other wireless media. Combinations of any of the above should also be included within the scope of computer readable media. The system memory 130 includes computer storage media in the form of volatile and/or nonvolatile memory such as read only memory (ROM) 131 and random access memory (RAM) 132. A basic input/output system 133 (BIOS), containing the basic routines that help to transfer information between elements within computer 110, such as during start-up, is typically stored in ROM 131. RAM 132 typically contains data and/or program modules that are immediately accessible to and/or presently being operated on by processing unit 120. By way of example, and not limitation, FIG. 1B illustrates operating system 134, application programs 135, other program modules 136, and program data 137. The computer 110 may also include other removable/non-removable, volatile/nonvolatile computer storage media. By way of example only, FIG. 1B illustrates a hard disk drive 141 that reads from or writes to non-removable, nonvolatile magnetic media, a magnetic disk drive 151 that reads from or writes to a removable, nonvolatile magnetic disk 152, and an optical disk drive 155 that reads from or writes to a removable, nonvolatile optical disk 156, such as a CD ROM or other optical media. Other removable/non-removable, volatile/nonvolatile computer storage media that can be used in the exemplary operating environment include, but are not limited to, magnetic tape cassettes, flash memory cards, digital versatile disks, digital video tape, solid state RAM, solid state ROM, and the like. The hard disk drive 141 is typically connected to the system bus 121 through an non-removable memory interface such as interface 140, and magnetic disk drive 151 and optical disk drive 155 are typically connected to the system bus 121 by a removable memory interface, such as interface 150. The drives and their associated computer storage media discussed above and illustrated in FIG. 1B provide storage of computer readable instructions, data structures, program modules and other data for the computer 110. In FIG. 1B, for example, hard disk drive 141 is illustrated as

Ex. C9 – Invalidity of U.S. Patent No. RE49,461 – Wilt

U.S. Patent No. RE49,461	Wilt
	storing operating system 144, application programs 145, other program modules 146, and program data 147. Note that these components can either be the same as or different from operating system 134, application programs 135, other program modules 136, and program data 137. Operating system 144, application programs 145, other program modules 146, and program data 147 are given different numbers here to illustrate that, at a minimum, they are different copies. A user may enter commands and information into the computer 110 through input devices such as a keyboard 162 and pointing device 161, commonly referred to as a mouse, trackball or touch pad. Other input devices (not shown) may include a microphone, joystick, game pad, satellite dish, scanner, or the like. These and other input devices are often connected to the processing unit 120 through a user input interface 160 that is coupled to the system bus 121, but may be connected by other interface and bus structures, such as a parallel port, game port or a universal serial bus (USB). A graphics interface 182, such as Northbridge, may also be connected to the system bus 121. Northbridge is a chipset that communicates with the CPU, or host processing unit 120, and assumes responsibility for AGP communications. One or more graphics processing units (GPUs) 184 may communicate with graphics interface 182. In this regard, GPUs 184 generally include on-chip memory storage, such as register storage and GPUs 184 communicate with a video memory 186. The invention may be described more generally in connection with any coprocessor, of which GPUs 184 are but one example. A monitor 191 or other type of display device is also connected to the system bus 121 via an interface, such as a video interface 190, which may in turn communicate with video memory 186. In addition to monitor 191, computers may also include other peripheral output devices such as speakers 197 and printer 196, which may be connected through an output peripheral interface 195. [0068]–[0071] To the extent Plaintiff contends that this claim element is not expressly or inherently disclosed, or is not obvious in view of Wilt and the knowledge of a POSITA, it would have been obvious to combine the teachings of Wilt with any of the prior art references in NVIDIA’s Invalidity Contentions and any supplements thereto and the relevant section of charts for other prior art for the ’461 patent in a manner that would result in the subject matter of this limitation given, at the very least, the person's understanding of the state of the art, the problems addressed and solved in the prior art, and the teachings of Wilt. Specific examples of such references are cited and

Ex. C9 – Invalidity of U.S. Patent No. RE49,461 – Wilt

U.S. Patent No. RE49,461	Wilt
	explained in these invalidity contentions, including the cover pleading and charts C1–C15, and Appendix C.
24. The method of claim 21, wherein executing the user interaction stream comprises: controlling setting and editing of parameters of the computations representing the artificial neural network.	Wilt discloses the method of claim 21, wherein executing the user interaction stream comprises controlling setting and editing of parameters of the computations representing the artificial neural network. See claims 21, 23. FIG. 1B FIG. 1B.

Ex. C9 – Invalidity of U.S. Patent No. RE49,461 – Wilt

U.S. Patent No. RE49,461	Wilt
	Computer 110 typically includes a variety of computer readable media. Computer readable media can be any available media that can be accessed by computer 110 and includes both volatile and nonvolatile media, removable and non-removable media. By way of example, and not limitation, computer readable media may comprise computer storage media and communication media. Computer storage media includes both volatile and nonvolatile, removable and non-removable media implemented in any method or technology for storage of information such as computer readable instructions, data structures, program modules or other data. Computer storage media includes, but is not limited to, RAM, ROM, EEPROM, flash memory or other memory technology, CDROM, digital versatile disks (DVD) or other optical disk storage, magnetic cassettes, magnetic tape, magnetic disk storage or other magnetic storage devices, or any other medium which can be used to store the desired information and which can be accessed by computer 110. Communication media typically embodies computer readable instructions, data structures, program modules or other data in a modulated data signal such as a carrier wave or other transport mechanism and includes any information delivery media. The term “modulated data signal” means a signal that has one or more of its characteristics set or changed in such a manner as to encode information in the signal. By way of example, and not limitation, communication media includes wired media such as a wired network or direct-wired connection, and wireless media such as acoustic, RF, infrared and other wireless media. Combinations of any of the above should also be included within the scope of computer readable media. The system memory 130 includes computer storage media in the form of volatile and/or nonvolatile memory such as read only memory (ROM) 131 and random access memory (RAM) 132. A basic input/output system 133 (BIOS), containing the basic routines that help to transfer information between elements within computer 110, such as during start-up, is typically stored in ROM 131. RAM 132 typically contains data and/or program modules that are immediately accessible to and/or presently being operated on by processing unit 120. By way of example, and not limitation, FIG. 1B illustrates operating system 134, application programs 135, other program modules 136, and program data 137. The computer 110 may also include other removable/non-removable, volatile/nonvolatile computer storage media. By way of example only, FIG. 1B illustrates a hard disk drive 141 that reads from or writes to non-removable, nonvolatile magnetic media, a magnetic disk drive 151 that reads from or writes to a removable, nonvolatile magnetic disk 152, and an optical

Ex. C9 – Invalidity of U.S. Patent No. RE49,461 – Wilt

U.S. Patent No. RE49,461	Wilt
	disk drive 155 that reads from or writes to a removable, nonvolatile optical disk 156, such as a CD ROM or other optical media. Other removable/non-removable, volatile/nonvolatile computer storage media that can be used in the exemplary operating environment include, but are not limited to, magnetic tape cassettes, flash memory cards, digital versatile disks, digital video tape, solid state RAM, solid state ROM, and the like. The hard disk drive 141 is typically connected to the system bus 121 through an non-removable memory interface such as interface 140, and magnetic disk drive 151 and optical disk drive 155 are typically connected to the system bus 121 by a removable memory interface, such as interface 150. The drives and their associated computer storage media discussed above and illustrated in FIG. 1B provide storage of computer readable instructions, data structures, program modules and other data for the computer 110. In FIG. 1B, for example, hard disk drive 141 is illustrated as storing operating system 144, application programs 145, other program modules 146, and program data 147. Note that these components can either be the same as or different from operating system 134, application programs 135, other program modules 136, and program data 137. Operating system 144, application programs 145, other program modules 146, and program data 147 are given different numbers here to illustrate that, at a minimum, they are different copies. A user may enter commands and information into the computer 110 through input devices such as a keyboard 162 and pointing device 161, commonly referred to as a mouse, trackball or touch pad. Other input devices (not shown) may include a microphone, joystick, game pad, satellite dish, scanner, or the like. These and other input devices are often connected to the processing unit 120 through a user input interface 160 that is coupled to the system bus 121, but may be connected by other interface and bus structures, such as a parallel port, game port or a universal serial bus (USB). A graphics interface 182, such as Northbridge, may also be connected to the system bus 121. Northbridge is a chipset that communicates with the CPU, or host processing unit 120, and assumes responsibility for AGP communications. One or more graphics processing units (GPUs) 184 may communicate with graphics interface 182. In this regard, GPUs 184 generally include on-chip memory storage, such as register storage and GPUs 184 communicate with a video memory 186. The invention may be described more generally in connection with any coprocessor, of which GPUs 184 are but one example. A monitor 191 or other type of display device is also connected to the system bus 121 via an interface, such as a video interface 190, which may in turn communicate with video memory 186. In addition to monitor 191, computers may also include other peripheral output

Ex. C9 – Invalidity of U.S. Patent No. RE49,461 – Wilt

U.S. Patent No. RE49,461	Wilt
	devices such as speakers 197 and printer 196, which may be connected through an output peripheral interface 195. [0068]–[0071] To the extent Plaintiff contends that this claim element is not expressly or inherently disclosed, or is not obvious in view of Wilt and the knowledge of a POSITA, it would have been obvious to combine the teachings of Wilt with any of the prior art references in NVIDIA’s Invalidity Contentions and any supplements thereto and the relevant section of charts for other prior art for the ’461 patent in a manner that would result in the subject matter of this limitation given, at the very least, the person's understanding of the state of the art, the problems addressed and solved in the prior art, and the teachings of Wilt. Specific examples of such references are cited and explained in these invalidity contentions, including the cover pleading and charts C1–C15, and Appendix C.
25. The method of claim 21, wherein executing the user interaction stream comprises: controlling setting and editing of parameters of the inputs to the artificial neural network.	Wilt discloses the method of claim 21, wherein executing the user interaction stream comprises controlling setting and editing of parameters of the inputs to the artificial neural network. See claim 21.

U.S. Patent No. RE49,461	Wilt
	Computing Environment 100 FIG. 1B FIG. 1B. Computer 110 typically includes a variety of computer readable media. Computer readable media can be any available media that can be accessed by computer 110 and includes both volatile and nonvolatile media, removable and non-removable media. By way of example, and not limitation, computer readable media may comprise computer storage media and communication media. Computer storage media includes both volatile and nonvolatile, removable and non-removable media implemented in any method or technology for storage of information such as computer readable instructions, data structures, program modules or other data. Computer storage media includes, but is not limited to, RAM, ROM, EEPROM, flash memory or other memory technology, CDROM, digital versatile disks (DVD) or other optical disk storage, magnetic cassettes, magnetic tape, magnetic disk storage or other magnetic

Ex. C9 – Invalidity of U.S. Patent No. RE49,461 – Wilt

U.S. Patent No. RE49,461	Wilt
	storage devices, or any other medium which can be used to store the desired information and which can accessed by computer 110. Communication media typically embodies computer readable instructions, data structures, program modules or other data in a modulated data signal such as a carrier wave or other transport mechanism and includes any information delivery media. The term “modulated data signal” means a signal that has one or more of its characteristics set or changed in such a manner as to encode information in the signal. By way of example, and not limitation, communication media includes wired media such as a wired network or direct-wired connection, and wireless media such as acoustic, RF, infrared and other wireless media. Combinations of any of the above should also be included within the scope of computer readable media. The system memory 130 includes computer storage media in the form of volatile and/or nonvolatile memory such as read only memory (ROM) 131 and random access memory (RAM) 132. A basic input/output system 133 (BIOS), containing the basic routines that help to transfer information between elements within computer 110, such as during start-up, is typically stored in ROM 131. RAM 132 typically contains data and/or program modules that are immediately accessible to and/or presently being operated on by processing unit 120. By way of example, and not limitation, FIG. 1B illustrates operating system 134, application programs 135, other program modules 136, and program data 137. The computer 110 may also include other removable/non-removable, volatile/nonvolatile computer storage media. By way of example only, FIG. 1B illustrates a hard disk drive 141 that reads from or writes to non-removable, nonvolatile magnetic media, a magnetic disk drive 151 that reads from or writes to a removable, nonvolatile magnetic disk 152, and an optical disk drive 155 that reads from or writes to a removable, nonvolatile optical disk 156, such as a CD ROM or other optical media. Other removable/non-removable, volatile/nonvolatile computer storage media that can be used in the exemplary operating environment include, but are not limited to, magnetic tape cassettes, flash memory cards, digital versatile disks, digital video tape, solid state RAM, solid state ROM, and the like. The hard disk drive 141 is typically connected to the system bus 121 through an non-removable memory interface such as interface 140, and magnetic disk drive 151 and optical disk drive 155 are typically connected to the system bus 121 by a removable memory interface, such as interface 150. The drives and their associated computer storage media discussed above and illustrated in FIG. 1B provide storage of computer readable instructions, data structures, program modules and

Ex. C9 – Invalidity of U.S. Patent No. RE49,461 – Wilt

U.S. Patent No. RE49,461	Wilt
	other data for the computer 110. In FIG. 1B, for example, hard disk drive 141 is illustrated as storing operating system 144, application programs 145, other program modules 146, and program data 147. Note that these components can either be the same as or different from operating system 134, application programs 135, other program modules 136, and program data 137. Operating system 144, application programs 145, other program modules 146, and program data 147 are given different numbers here to illustrate that, at a minimum, they are different copies. A user may enter commands and information into the computer 110 through input devices such as a keyboard 162 and pointing device 161, commonly referred to as a mouse, trackball or touch pad. Other input devices (not shown) may include a microphone, joystick, game pad, satellite dish, scanner, or the like. These and other input devices are often connected to the processing unit 120 through a user input interface 160 that is coupled to the system bus 121, but may be connected by other interface and bus structures, such as a parallel port, game port or a universal serial bus (USB). A graphics interface 182, such as Northbridge, may also be connected to the system bus 121. Northbridge is a chipset that communicates with the CPU, or host processing unit 120, and assumes responsibility for AGP communications. One or more graphics processing units (GPUs) 184 may communicate with graphics interface 182. In this regard, GPUs 184 generally include on-chip memory storage, such as register storage and GPUs 184 communicate with a video memory 186. The invention may be described more generally in connection with any coprocessor, of which GPUs 184 are but one example. A monitor 191 or other type of display device is also connected to the system bus 121 via an interface, such as a video interface 190, which may in turn communicate with video memory 186. In addition to monitor 191, computers may also include other peripheral output devices such as speakers 197 and printer 196, which may be connected through an output peripheral interface 195. [0068]–[0071] To the extent Plaintiff contends that this claim element is not expressly or inherently disclosed, or is not obvious in view of Wilt and the knowledge of a POSITA, it would have been obvious to combine the teachings of Wilt with any of the prior art references in NVIDIA’s Invalidity Contentions and any supplements thereto and the relevant section of charts for other prior art for the ’461 patent in a manner that would result in the subject matter of this limitation given, at the very least, the person's understanding of the state of the art, the problems addressed and

Ex. C9 – Invalidity of U.S. Patent No. RE49,461 – Wilt

U.S. Patent No. RE49,461	Wilt
	solved in the prior art, and the teachings of Wilt. Specific examples of such references are cited and explained in these invalidity contentions, including the cover pleading and charts C1–C15, and Appendix C.
26. The method of claim 21, wherein executing the user interaction stream comprises: specifying an output to be saved to disk and/or displayed on a screen.	Wilt discloses the method of claim 21, wherein executing the user interaction stream comprises specifying an output to be saved to disk and/or displayed on a screen. See claim 21.
	Computing Environment 100 The diagram illustrates a computing environment 100 enclosed in a dashed box. Inside, a system memory 130 contains ROM 131 (with BIOS 133) and RAM 132 (containing Operating System 134, Application Programs 135, Other Program Modules 136, and Program Data 137). A processing unit 120 is connected via a system bus 121 to a GPU 184, which is linked to video memory 186 and a graphics interface 182. The graphics interface connects to a video interface 190, which leads to an output peripheral interface 195. This interface controls a monitor 191, printer 196, and speakers 197. The system also includes non-removable non-volatile memory interface 140 and removable non-volatile memory interface 150, both connected to the system bus. A user input interface 160 receives signals from a mouse 161 and keyboard 162. A network interface 170 connects to a local area network 171 and a wide area network 173. The wide area network includes a modem 172 and a remote computer 180. Below the main system box, there's a detailed view of program storage media: operating system 144, application programs 145, other program mods. 146, and program data 147. These are shown as separate units or stored on various media types represented by icons (floppy disk 151, CD-ROM 152, DVD-ROM 155, hard drive 156). FIG. 1B
	FIG. 1B. Computer 110 typically includes a variety of computer readable media. Computer readable media can be any available media that can be accessed by computer 110 and includes both

Ex. C9 – Invalidity of U.S. Patent No. RE49,461 – Wilt

U.S. Patent No. RE49,461	Wilt
	volatile and nonvolatile media, removable and non-removable media. By way of example, and not limitation, computer readable media may comprise computer storage media and communication media. Computer storage media includes both volatile and nonvolatile, removable and non-removable media implemented in any method or technology for storage of information such as computer readable instructions, data structures, program modules or other data. Computer storage media includes, but is not limited to, RAM, ROM, EEPROM, flash memory or other memory technology, CDROM, digital versatile disks (DVD) or other optical disk storage, magnetic cassettes, magnetic tape, magnetic disk storage or other magnetic storage devices, or any other medium which can be used to store the desired information and which can be accessed by computer 110. Communication media typically embodies computer readable instructions, data structures, program modules or other data in a modulated data signal such as a carrier wave or other transport mechanism and includes any information delivery media. The term “modulated data signal” means a signal that has one or more of its characteristics set or changed in such a manner as to encode information in the signal. By way of example, and not limitation, communication media includes wired media such as a wired network or direct-wired connection, and wireless media such as acoustic, RF, infrared and other wireless media. Combinations of any of the above should also be included within the scope of computer readable media. The system memory 130 includes computer storage media in the form of volatile and/or nonvolatile memory such as read only memory (ROM) 131 and random access memory (RAM) 132. A basic input/output system 133 (BIOS), containing the basic routines that help to transfer information between elements within computer 110, such as during start-up, is typically stored in ROM 131. RAM 132 typically contains data and/or program modules that are immediately accessible to and/or presently being operated on by processing unit 120. By way of example, and not limitation, FIG. 1B illustrates operating system 134, application programs 135, other program modules 136, and program data 137. The computer 110 may also include other removable/non-removable, volatile/nonvolatile computer storage media. By way of example only, FIG. 1B illustrates a hard disk drive 141 that reads from or writes to non-removable, nonvolatile magnetic media, a magnetic disk drive 151 that reads from or writes to a removable, nonvolatile magnetic disk 152, and an optical disk drive 155 that reads from or writes to a removable, nonvolatile optical disk 156, such as a CD ROM or other optical media. Other removable/non-removable, volatile/nonvolatile

Ex. C9 – Invalidity of U.S. Patent No. RE49,461 – Wilt

U.S. Patent No. RE49,461	Wilt
	computer storage media that can be used in the exemplary operating environment include, but are not limited to, magnetic tape cassettes, flash memory cards, digital versatile disks, digital video tape, solid state RAM, solid state ROM, and the like. The hard disk drive 141 is typically connected to the system bus 121 through a non-removable memory interface such as interface 140, and magnetic disk drive 151 and optical disk drive 155 are typically connected to the system bus 121 by a removable memory interface, such as interface 150. The drives and their associated computer storage media discussed above and illustrated in FIG. 1B provide storage of computer readable instructions, data structures, program modules and other data for the computer 110. In FIG. 1B, for example, hard disk drive 141 is illustrated as storing operating system 144, application programs 145, other program modules 146, and program data 147. Note that these components can either be the same as or different from operating system 134, application programs 135, other program modules 136, and program data 137. Operating system 144, application programs 145, other program modules 146, and program data 147 are given different numbers here to illustrate that, at a minimum, they are different copies. A user may enter commands and information into the computer 110 through input devices such as a keyboard 162 and pointing device 161, commonly referred to as a mouse, trackball or touch pad. Other input devices (not shown) may include a microphone, joystick, game pad, satellite dish, scanner, or the like. These and other input devices are often connected to the processing unit 120 through a user input interface 160 that is coupled to the system bus 121, but may be connected by other interface and bus structures, such as a parallel port, game port or a universal serial bus (USB). A graphics interface 182, such as Northbridge, may also be connected to the system bus 121. Northbridge is a chipset that communicates with the CPU, or host processing unit 120, and assumes responsibility for AGP communications. One or more graphics processing units (GPUs) 184 may communicate with graphics interface 182. In this regard, GPUs 184 generally include on-chip memory storage, such as register storage and GPUs 184 communicate with a video memory 186. The invention may be described more generally in connection with any coprocessor, of which GPUs 184 are but one example. A monitor 191 or other type of display device is also connected to the system bus 121 via an interface, such as a video interface 190, which may in turn communicate with video memory 186. In addition to monitor 191, computers may also include other peripheral output devices such as speakers 197 and printer 196, which may be connected through an output peripheral interface 195.

Ex. C9 – Invalidity of U.S. Patent No. RE49,461 – Wilt

U.S. Patent No. RE49,461	Wilt
	[0068]–[0071] To the extent Plaintiff contends that this claim element is not expressly or inherently disclosed, or is not obvious in view of Wilt and the knowledge of a POSITA, it would have been obvious to combine the teachings of Wilt with any of the prior art references in NVIDIA’s Invalidity Contentions and any supplements thereto and the relevant section of charts for other prior art for the ’461 patent in a manner that would result in the subject matter of this limitation given, at the very least, the person’s understanding of the state of the art, the problems addressed and solved in the prior art, and the teachings of Wilt. Specific examples of such references are cited and explained in these invalidity contentions, including the cover pleading and charts C1–C15, and Appendix C.
27. The method of claim 21, wherein executing the user interaction stream comprises: parsing elements to be used in the computations representing the artificial neural network.	Wilt discloses the method of claim 21, wherein executing the user interaction stream comprises parsing elements to be used in the computations representing the artificial neural network. See claim 21. Given these assumptions, we can broaden our focus from graphics processors and contemplate a system with two processors: a host CPU, and a coprocessor that is controlled by sending command buffers from the CPU, that can send data back to the CPU, and that coordinates control by notifying the CPU of various events when they occur. The examples in this document frequently assume that the coprocessor is a graphics processor (as when describing possible synchronization problems and how a system can be designed to solve those problems efficiently), but these examples are for illustrative purposes only and do not limit the scope of this invention to graphics processors. In line with the above assumptions regarding computational resources in a graphics processing system, a non-limiting system is described below that makes efficient, shared use of the computational resources in a coprocessor. FIG. 2 depicts the life cycle of a command buffer as it is used by applications to send commands to the coprocessor, such as a graphics processor. When a command buffer is created and initialized, the buffer enters the initialized state 201. The command buffer is then available to be selected and written into by a thread that wishes to issue commands to the

Ex. C9 – Invalidity of U.S. Patent No. RE49,461 – Wilt

U.S. Patent No. RE49,461	Wilt
	graphics processor. When a thread selects the command buffer, the memory in the command buffer is made available to that thread (perhaps by mapping the memory in the command buffer into that thread's address space) and the command buffer enters the Receiving state 202. As one of ordinary skill in the art can appreciate, the details of this mapping may vary. For example, the command buffer memory could be made visible to user mode, kernel mode, or both. In addition, there are numerous options as to the nature of the executable code that is writing into the Receiving command buffer. In this regard, FIGS. 3A through 3E depict several of these options, i.e., several different architectures for writing into the command buffer. For purposes of describing FIGS. 3A through 3E, an application is a process, or a thread in a process, that implements some specific graphical application participating in the system. The runtime is middleware provided in the operating system that the application uses. Among other things, the runtime may validate the application's usage of an application programming interface (API), translate simple-to-use, high-level API calls into more complicated, lower-level constructs that are more readily translatable to hardware commands, and so on. The driver is a hardware-specific component that translates runtime-level commands to hardware-specific commands. The driver components depicted in FIGS. 3A through 3E are the portions of the driver that translate runtime-level commands to hardware-specific commands. Some portions of the driver that perform tasks other than this translation, such as interrupt handling and command buffer dispatch, would probably reside in kernel mode and are not contemplated in FIGS. 3A through 3E. Most of the object code depicted in FIGS. 3A through 3E may run in user mode, although some portions, as pointed out below, can run in kernel mode. The command buffer is the piece of memory that includes commands that can be consumed directly by the hardware via some DMA mechanism. Several of the figures also show a token stream, which is a hardware-independent stream of runtime-level primitives that can be translated into hardware-specific commands by the driver. Lastly, some of the figures show a JIT (Just-In-Time) compilation component that can translate a hardware-independent intermediate language into object code for execution by the host microprocessor. The boxes with OBJ in the corner include object code that can be directly executed by the microprocessor. Boxes with MSIL in the corner include intermediate code that is compiled by the JIT in order to execute on the microprocessor. If the application and runtime are both available on the client in intermediate form, the JIT can translate them into a

Ex. C9 – Invalidity of U.S. Patent No. RE49,461 – Wilt

U.S. Patent No. RE49,461	Wilt
	unified piece of object code, shown as app executables 324, 334 and 345 FIGS. 3C to 3E, respectively, that implements the application. FIG. 3A shows the application 301, runtime 302 and driver 303 each performing their respective roles, calling into lower-level object code until the driver 303 writes hardware-specific commands into the command buffer 304. The application 301 could be an executable process, or a thread in a process, while the runtime 302 and driver 303 could each reside in DLLs. As depicted in FIG. 3A, all of the object code may reside in user mode (although some or all of the runtime 302 and/or driver 303 may reside in kernel mode). If the runtime 302 determines that the command buffer 304 must be submitted before execution can continue, or the driver 303 determines that the command buffer 304 is out of space, a transition to kernel mode may be requested to submit the command buffer 304 and retrieve another one in order to continue. FIG. 3B shows a variant in which the runtime 312 emits a hardware-independent stream of commands into a buffer 313, which is then parsed by the driver 314 and written into a command buffer 315. Application 311 operates to call lower-level object code in connection with runtime 312. FIG. 3B approximates the current architecture of the DIRECTX® software component, in which the runtime translates application activity into DrawPrimitives2 (“DP2”) tokens. When the DP2 token stream is submitted, a kernel transition occurs and the driver 314 translates the DP2 token stream into hardware-specific commands in kernel mode. FIG. 3B does not make any assumptions about whether the driver 314 is in user mode or kernel mode, and in this regard, driver component 314 could reside in user mode as well as kernel mode. FIG. 3C shows a variant in which the application 321 and runtime 322 are synthesized into a unified piece of object code by a JIT compiler 323. The resulting piece of object code 324 then interacts with the driver 325 to write hardware-specific commands into the command buffer 326. FIG. 3D shows the JIT variant of the situation described for FIG. 3B, in which a hardware-independent token stream 335 is emitted by the software 334 above the driver 336. The driver 336 then translates the token stream to hardware-specific commands for the command buffer 337. As with FIG. 3C, application 331 and runtime 332 are synthesized into a unified piece of object code 334 by JIT compiler 333. FIG. 3E shows a system in which the application 341, runtime 342, and part of the driver 343 are all available on the client in intermediate language form MSIL. The JIT compiler 344

Ex. C9 – Invalidity of U.S. Patent No. RE49,461 – Wilt

U.S. Patent No. RE49,461	Wilt
	synthesizes these components into a unified piece of object code 345 specifically optimized to run on that particular client computer, taking into account not only the host microprocessor but also the coprocessor hardware configuration. Referring back to the command buffer life cycle depicted in FIG. 2, regardless of the method by which commands are written into the command buffer, the thread writing into the receiving command buffer eventually submits to the command buffer either because the command buffer has been filled, or because the drawing commands that have been written into the buffer are executed forthwith. At that time, the command buffer enters the waiting state 203, and the contents of the command buffer may not be submitted to the hardware until all of the resources utilized to execute the command buffer are available. As an example of a circumstance that may force a wait, if an application draws into a surface, the drawing must be completed before that surface may be selected as a texture. Once the waiting criteria have been fulfilled, the command buffer enters the ready state 204, where the scheduler considers it for submission to the coprocessor. The scheduler has a great deal of flexibility in deciding which command buffer should be submitted to the hardware. The scheduler may take into account the priority, if any, of the command buffer, the priority of the requesting thread, the real-time requirements of the system, and so on. Once the command buffer has been selected as the next to run, the command buffer enters the standby state 205. On some systems, only one command buffer per coprocessor may be in the standby state 205 in the system. When the command buffer in the standby state 205 can be submitted to the hardware, the system switches the coprocessor context to the context required by the command buffer, and then submits the command buffer to the hardware. This operation entails some involvement from the driver, for example to initiate a DMA operation using the base address and size of the command buffer. The command buffer remains in the running state 206 until some event preempts execution of the command buffer or the entire contents of the command buffer have been read and executed. If the entire command buffer has been executed, the command buffer enters the terminated state 207, where it becomes available for the system to reinitialize and make available to another thread in the initialized state 201. If the command buffer is preempted, it is assumed that the hardware has the capability to save the hardware context in such a way that execution of the command buffer can be resumed

Ex. C9 – Invalidity of U.S. Patent No. RE49,461 – Wilt

U.S. Patent No. RE49,461	Wilt
	where the preemption occurred. The command buffer can then be recycled into the ready state 204 and be re-submitted to the hardware to resume from where it stopped. The description below regarding preemptible command buffer execution describes this in more detail. [0095]–[0110] The command buffer scheduler 404 (“scheduler”) and kernel driver 405 work together in kernel mode to dispatch command buffers to the hardware 406 (the scheduler 404 decides which command buffer should be dispatched, while the kernel driver 405 instructs the hardware 406 to dispatch a command buffer at the request of the scheduler). This system contemplates that the bulk of the driver logic would reside in the user mode driver DLL 403, not the kernel driver 405. While the User Mode Driver can contain large amounts of code that maps DDI-level calls into hardware-specific commands (which operation can be complicated and error-prone, especially when compiling a vertex and/or shader program), the Kernel Driver is comparatively small and simple, maximizing system robustness. [0146] To the extent Plaintiff contends that this claim element is not expressly or inherently disclosed, or is not obvious in view of Wilt and the knowledge of a POSITA, it would have been obvious to combine the teachings of Wilt with any of the prior art references in NVIDIA’s Invalidity Contentions and any supplements thereto and the relevant section of charts for other prior art for the ’461 patent in a manner that would result in the subject matter of this limitation given, at the very least, the person’s understanding of the state of the art, the problems addressed and solved in the prior art, and the teachings of Wilt. Specific examples of such references are cited and explained in these invalidity contentions, including the cover pleading and charts C1–C15, and Appendix C.
28. The method of claim 27, wherein the processing unit comprises a graphics processing unit (GPU) and executing the user interaction stream further comprises: converting the elements into GPU programs.	Wilt discloses the method of claim 27, wherein the processing unit comprises a graphics processing unit (GPU) and executing the user interaction stream further comprises converting the elements into GPU programs. <i>See claim 27.</i>

Ex. C9 – Invalidity of U.S. Patent No. RE49,461 – Wilt

U.S. Patent No. RE49,461	Wilt
	Given these assumptions, we can broaden our focus from graphics processors and contemplate a system with two processors: a host CPU, and a coprocessor that is controlled by sending command buffers from the CPU, that can send data back to the CPU, and that coordinates control by notifying the CPU of various events when they occur. The examples in this document frequently assume that the coprocessor is a graphics processor (as when describing possible synchronization problems and how a system can be designed to solve those problems efficiently), but these examples are for illustrative purposes only and do not limit the scope of this invention to graphics processors. In line with the above assumptions regarding computational resources in a graphics processing system, a non-limiting system is described below that makes efficient, shared use of the computational resources in a coprocessor. FIG. 2 depicts the life cycle of a command buffer as it is used by applications to send commands to the coprocessor, such as a graphics processor. When a command buffer is created and initialized, the buffer enters the initialized state 201. The command buffer is then available to be selected and written into by a thread that wishes to issue commands to the graphics processor. When a thread selects the command buffer, the memory in the command buffer is made available to that thread (perhaps by mapping the memory in the command buffer into that thread's address space) and the command buffer enters the Receiving state 202. As one of ordinary skill in the art can appreciate, the details of this mapping may vary. For example, the command buffer memory could be made visible to user mode, kernel mode, or both. In addition, there are numerous options as to the nature of the executable code that is writing into the Receiving command buffer. In this regard, FIGS. 3A through 3E depict several of these options, i.e., several different architectures for writing into the command buffer. For purposes of describing FIGS. 3A through 3E, an application is a process, or a thread in a process, that implements some specific graphical application participating in the system. The runtime is middleware provided in the operating system that the application uses. Among other things, the runtime may validate the application's usage of an application programming interface (API), translate simple-to-use, high-level API calls into more complicated, lower-level constructs that are more readily translatable to hardware commands, and so on. The driver is a hardware-specific component that translates runtime-level commands to hardware-specific commands. The driver components depicted in FIGS. 3A through 3E are the portions of the driver that translate runtime-level commands to hardware-specific commands.

Ex. C9 – Invalidity of U.S. Patent No. RE49,461 – Wilt

U.S. Patent No. RE49,461	Wilt
	Some portions of the driver that perform tasks other than this translation, such as interrupt handling and command buffer dispatch, would probably reside in kernel mode and are not contemplated in FIGS. 3A through 3E. Most of the object code depicted in FIGS. 3A through 3E may run in user mode, although some portions, as pointed out below, can run in kernel mode. The command buffer is the piece of memory that includes commands that can be consumed directly by the hardware via some DMA mechanism. Several of the figures also show a token stream, which is a hardware-independent stream of runtime-level primitives that can be translated into hardware-specific commands by the driver. Lastly, some of the figures show a JIT (Just-In-Time) compilation component that can translate a hardware-independent intermediate language into object code for execution by the host microprocessor. The boxes with OBJ in the corner include object code that can be directly executed by the microprocessor. Boxes with MSIL in the corner include intermediate code that is compiled by the JIT in order to execute on the microprocessor. If the application and runtime are both available on the client in intermediate form, the JIT can translate them into a unified piece of object code, shown as app executables 324, 334 and 345 FIGS. 3C to 3E, respectively, that implements the application. FIG. 3A shows the application 301, runtime 302 and driver 303 each performing their respective roles, calling into lower-level object code until the driver 303 writes hardware-specific commands into the command buffer 304. The application 301 could be an executable process, or a thread in a process, while the runtime 302 and driver 303 could each reside in DLLs. As depicted in FIG. 3A, all of the object code may reside in user mode (although some or all of the runtime 302 and/or driver 303 may reside in kernel mode). If the runtime 302 determines that the command buffer 304 must be submitted before execution can continue, or the driver 303 determines that the command buffer 304 is out of space, a transition to kernel mode may be requested to submit the command buffer 304 and retrieve another one in order to continue. FIG. 3B shows a variant in which the runtime 312 emits a hardware-independent stream of commands into a buffer 313, which is then parsed by the driver 314 and written into a command buffer 315. Application 311 operates to call lower-level object code in connection with runtime 312. FIG. 3B approximates the current architecture of the DIRECTX® software component, in which the runtime translates application activity into DrawPrimitives2 (“DP2”) tokens. When the DP2 token stream is submitted, a kernel transition occurs and the driver 314

Ex. C9 – Invalidity of U.S. Patent No. RE49,461 – Wilt

U.S. Patent No. RE49,461	Wilt
	translates the DP2 token stream into hardware-specific commands in kernel mode. FIG. 3B does not make any assumptions about whether the driver 314 is in user mode or kernel mode, and in this regard, driver component 314 could reside in user mode as well as kernel mode. FIG. 3C shows a variant in which the application 321 and runtime 322 are synthesized into a unified piece of object code by a JIT compiler 323. The resulting piece of object code 324 then interacts with the driver 325 to write hardware-specific commands into the command buffer 326. FIG. 3D shows the JIT variant of the situation described for FIG. 3B, in which a hardware-independent token stream 335 is emitted by the software 334 above the driver 336. The driver 336 then translates the token stream to hardware-specific commands for the command buffer 337. As with FIG. 3C, application 331 and runtime 332 are synthesized into a unified piece of object code 334 by JIT compiler 333. FIG. 3E shows a system in which the application 341, runtime 342, and part of the driver 343 are all available on the client in intermediate language form MSIL. The JIT compiler 344 synthesizes these components into a unified piece of object code 345 specifically optimized to run on that particular client computer, taking into account not only the host microprocessor but also the coprocessor hardware configuration. Referring back to the command buffer life cycle depicted in FIG. 2, regardless of the method by which commands are written into the command buffer, the thread writing into the receiving command buffer eventually submits to the command buffer either because the command buffer has been filled, or because the drawing commands that have been written into the buffer are executed forthwith. At that time, the command buffer enters the waiting state 203, and the contents of the command buffer may not be submitted to the hardware until all of the resources utilized to execute the command buffer are available. As an example of a circumstance that may force a wait, if an application draws into a surface, the drawing must be completed before that surface may be selected as a texture. Once the waiting criteria have been fulfilled, the command buffer enters the ready state 204, where the scheduler considers it for submission to the coprocessor. The scheduler has a great deal of flexibility in deciding which command buffer should be submitted to the hardware. The scheduler may take into account the priority, if any, of the command buffer, the priority of the requesting thread, the real-time requirements of the system, and so on. Once the command buffer has been selected as the next to run, the command buffer enters the standby state 205.

Ex. C9 – Invalidity of U.S. Patent No. RE49,461 – Wilt

U.S. Patent No. RE49,461	Wilt
	On some systems, only one command buffer per coprocessor may be in the standby state 205 in the system. When the command buffer in the standby state 205 can be submitted to the hardware, the system switches the coprocessor context to the context required by the command buffer, and then submits the command buffer to the hardware. This operation entails some involvement from the driver, for example to initiate a DMA operation using the base address and size of the command buffer. The command buffer remains in the running state 206 until some event preempts execution of the command buffer or the entire contents of the command buffer have been read and executed. If the entire command buffer has been executed, the command buffer enters the terminated state 207, where it becomes available for the system to reinitialize and make available to another thread in the initialized state 201. If the command buffer is preempted, it is assumed that the hardware has the capability to save the hardware context in such a way that execution of the command buffer can be resumed where the preemption occurred. The command buffer can then be recycled into the ready state 204 and be re-submitted to the hardware to resume from where it stopped. The description below regarding preemptible command buffer execution describes this in more detail. [0095]–[0110] The command buffer scheduler 404 (“scheduler”) and kernel driver 405 work together in kernel mode to dispatch command buffers to the hardware 406 (the scheduler 404 decides which command buffer should be dispatched, while the kernel driver 405 instructs the hardware 406 to dispatch a command buffer at the request of the scheduler). This system contemplates that the bulk of the driver logic would reside in the user mode driver DLL 403, not the kernel driver 405. While the User Mode Driver can contain large amounts of code that maps DDI-level calls into hardware-specific commands (which operation can be complicated and error-prone, especially when compiling a vertex and/or shader program), the Kernel Driver is comparatively small and simple, maximizing system robustness. [0146]

Ex. C9 – Invalidity of U.S. Patent No. RE49,461 – Wilt

U.S. Patent No. RE49,461	Wilt
	To the extent Plaintiff contends that this claim element is not expressly or inherently disclosed, or is not obvious in view of Wilt and the knowledge of a POSITA, it would have been obvious to combine the teachings of Wilt with any of the prior art references in NVIDIA's Invalidity Contentions and any supplements thereto and the relevant section of charts for other prior art for the '461 patent in a manner that would result in the subject matter of this limitation given, at the very least, the person's understanding of the state of the art, the problems addressed and solved in the prior art, and the teachings of Wilt. Specific examples of such references are cited and explained in these invalidity contentions, including the cover pleading and charts C1–C15, and Appendix C.
29. The method of claim 28, wherein executing the user interaction stream comprises: compiling the GPU programs.	Wilt discloses the method of claim 28, wherein executing the user interaction stream comprises: compiling the GPU programs. See claim 28. Given these assumptions, we can broaden our focus from graphics processors and contemplate a system with two processors: a host CPU, and a coprocessor that is controlled by sending command buffers from the CPU, that can send data back to the CPU, and that coordinates control by notifying the CPU of various events when they occur. The examples in this document frequently assume that the coprocessor is a graphics processor (as when describing possible synchronization problems and how a system can be designed to solve those problems efficiently), but these examples are for illustrative purposes only and do not limit the scope of this invention to graphics processors. In line with the above assumptions regarding computational resources in a graphics processing system, a non-limiting system is described below that makes efficient, shared use of the computational resources in a coprocessor. FIG. 2 depicts the life cycle of a command buffer as it is used by applications to send commands to the coprocessor, such as a graphics processor. When a command buffer is created and initialized, the buffer enters the initialized state 201. The command buffer is then available to be selected and written into by a thread that wishes to issue commands to the graphics processor. When a thread selects the command buffer, the memory in the command buffer is made available to that thread (perhaps by mapping the memory in the command buffer into that thread's address space) and the command buffer enters the Receiving state 202.

Ex. C9 – Invalidity of U.S. Patent No. RE49,461 – Wilt

U.S. Patent No. RE49,461	Wilt
	As one of ordinary skill in the art can appreciate, the details of this mapping may vary. For example, the command buffer memory could be made visible to user mode, kernel mode, or both. In addition, there are numerous options as to the nature of the executable code that is writing into the Receiving command buffer. In this regard, FIGS. 3A through 3E depict several of these options, i.e., several different architectures for writing into the command buffer. For purposes of describing FIGS. 3A through 3E, an application is a process, or a thread in a process, that implements some specific graphical application participating in the system. The runtime is middleware provided in the operating system that the application uses. Among other things, the runtime may validate the application's usage of an application programming interface (API), translate simple-to-use, high-level API calls into more complicated, lower-level constructs that are more readily translatable to hardware commands, and so on. The driver is a hardware-specific component that translates runtime-level commands to hardware-specific commands. The driver components depicted in FIGS. 3A through 3E are the portions of the driver that translate runtime-level commands to hardware-specific commands. Some portions of the driver that perform tasks other than this translation, such as interrupt handling and command buffer dispatch, would probably reside in kernel mode and are not contemplated in FIGS. 3A through 3E. Most of the object code depicted in FIGS. 3A through 3E may run in user mode, although some portions, as pointed out below, can run in kernel mode. The command buffer is the piece of memory that includes commands that can be consumed directly by the hardware via some DMA mechanism. Several of the figures also show a token stream, which is a hardware-independent stream of runtime-level primitives that can be translated into hardware-specific commands by the driver. Lastly, some of the figures show a JIT (Just-In-Time) compilation component that can translate a hardware-independent intermediate language into object code for execution by the host microprocessor. The boxes with OBJ in the corner include object code that can be directly executed by the microprocessor. Boxes with MSIL in the corner include intermediate code that is compiled by the JIT in order to execute on the microprocessor. If the application and runtime are both available on the client in intermediate form, the JIT can translate them into a unified piece of object code, shown as app executables 324, 334 and 345 FIGS. 3C to 3E, respectively, that implements the application. FIG. 3A shows the application 301, runtime 302 and driver 303 each performing their respective roles, calling into lower-level object code until the driver 303 writes hardware-

Ex. C9 – Invalidity of U.S. Patent No. RE49,461 – Wilt

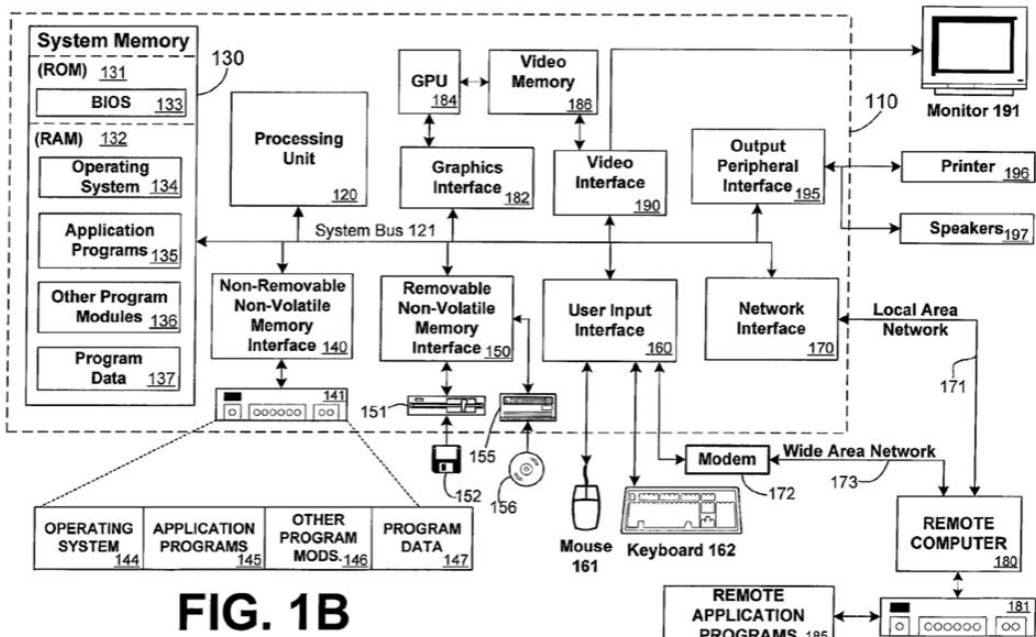
U.S. Patent No. RE49,461	Wilt
	specific commands into the command buffer 304. The application 301 could be an executable process, or a thread in a process, while the runtime 302 and driver 303 could each reside in DLLs. As depicted in FIG. 3A, all of the object code may reside in user mode (although some or all of the runtime 302 and/or driver 303 may reside in kernel mode). If the runtime 302 determines that the command buffer 304 must be submitted before execution can continue, or the driver 303 determines that the command buffer 304 is out of space, a transition to kernel mode may be requested to submit the command buffer 304 and retrieve another one in order to continue. FIG. 3B shows a variant in which the runtime 312 emits a hardware-independent stream of commands into a buffer 313, which is then parsed by the driver 314 and written into a command buffer 315. Application 311 operates to call lower-level object code in connection with runtime 312. FIG. 3B approximates the current architecture of the DIRECTX® software component, in which the runtime translates application activity into DrawPrimitives2 (“DP2”) tokens. When the DP2 token stream is submitted, a kernel transition occurs and the driver 314 translates the DP2 token stream into hardware-specific commands in kernel mode. FIG. 3B does not make any assumptions about whether the driver 314 is in user mode or kernel mode, and in this regard, driver component 314 could reside in user mode as well as kernel mode. FIG. 3C shows a variant in which the application 321 and runtime 322 are synthesized into a unified piece of object code by a JIT compiler 323. The resulting piece of object code 324 then interacts with the driver 325 to write hardware-specific commands into the command buffer 326. FIG. 3D shows the JIT variant of the situation described for FIG. 3B, in which a hardware-independent token stream 335 is emitted by the software 334 above the driver 336. The driver 336 then translates the token stream to hardware-specific commands for the command buffer 337. As with FIG. 3C, application 331 and runtime 332 are synthesized into a unified piece of object code 334 by JIT compiler 333. FIG. 3E shows a system in which the application 341, runtime 342, and part of the driver 343 are all available on the client in intermediate language form MSIL. The JIT compiler 344 synthesizes these components into a unified piece of object code 345 specifically optimized to run on that particular client computer, taking into account not only the host microprocessor but also the coprocessor hardware configuration.

Ex. C9 – Invalidity of U.S. Patent No. RE49,461 – Wilt

U.S. Patent No. RE49,461	Wilt
	Referring back to the command buffer life cycle depicted in FIG. 2, regardless of the method by which commands are written into the command buffer, the thread writing into the receiving command buffer eventually submits to the command buffer either because the command buffer has been filled, or because the drawing commands that have been written into the buffer are executed forthwith. At that time, the command buffer enters the waiting state 203, and the contents of the command buffer may not be submitted to the hardware until all of the resources utilized to execute the command buffer are available. As an example of a circumstance that may force a wait, if an application draws into a surface, the drawing must be completed before that surface may be selected as a texture. Once the waiting criteria have been fulfilled, the command buffer enters the ready state 204, where the scheduler considers it for submission to the coprocessor. The scheduler has a great deal of flexibility in deciding which command buffer should be submitted to the hardware. The scheduler may take into account the priority, if any, of the command buffer, the priority of the requesting thread, the real-time requirements of the system, and so on. Once the command buffer has been selected as the next to run, the command buffer enters the standby state 205. On some systems, only one command buffer per coprocessor may be in the standby state 205 in the system. When the command buffer in the standby state 205 can be submitted to the hardware, the system switches the coprocessor context to the context required by the command buffer, and then submits the command buffer to the hardware. This operation entails some involvement from the driver, for example to initiate a DMA operation using the base address and size of the command buffer. The command buffer remains in the running state 206 until some event preempts execution of the command buffer or the entire contents of the command buffer have been read and executed. If the entire command buffer has been executed, the command buffer enters the terminated state 207, where it becomes available for the system to reinitialize and make available to another thread in the initialized state 201. If the command buffer is preempted, it is assumed that the hardware has the capability to save the hardware context in such a way that execution of the command buffer can be resumed where the preemption occurred. The command buffer can then be recycled into the ready state 204 and be re-submitted to the hardware to resume from where it stopped. The description below regarding preemptible command buffer execution describes this in more detail.

Ex. C9 – Invalidity of U.S. Patent No. RE49,461 – Wilt

U.S. Patent No. RE49,461	Wilt
	[0095]–[0110] The command buffer scheduler 404 (“scheduler”) and kernel driver 405 work together in kernel mode to dispatch command buffers to the hardware 406 (the scheduler 404 decides which command buffer should be dispatched, while the kernel driver 405 instructs the hardware 406 to dispatch a command buffer at the request of the scheduler). This system contemplates that the bulk of the driver logic would reside in the user mode driver DLL 403, not the kernel driver 405. While the User Mode Driver can contain large amounts of code that maps DDI-level calls into hardware-specific commands (which operation can be complicated and error-prone, especially when compiling a vertex and/or shader program), the Kernel Driver is comparatively small and simple, maximizing system robustness. [0146] To the extent Plaintiff contends that this claim element is not expressly or inherently disclosed, or is not obvious in view of Wilt and the knowledge of a POSITA, it would have been obvious to combine the teachings of Wilt with any of the prior art references in NVIDIA’s Invalidity Contentions and any supplements thereto and the relevant section of charts for other prior art for the ’461 patent in a manner that would result in the subject matter of this limitation given, at the very least, the person's understanding of the state of the art, the problems addressed and solved in the prior art, and the teachings of Wilt. Specific examples of such references are cited and explained in these invalidity contentions, including the cover pleading and charts C1–C15, and Appendix C.
30. The method of claim 29, wherein executing the user interaction stream comprises: transferring the GPU programs to the second memory partition.	Wilt discloses the method of claim 29, wherein executing the user interaction stream comprises transferring the GPU programs to the second memory partition. See claim 29.

U.S. Patent No. RE49,461	Wilt
	Computing Environment 100  FIG. 1B FIG. 1B. Direct Memory Access (DMA) is a common method for moving data within computer systems, and has found application in the control of graphics processors as well as disk drives, network adapters, and external buses such as USB and IEEE 1394. DMA generally refers to the ability of a peripheral, such as a digital camera or disk controller, to read from or write to memory without directly involving the host processor. The host sets up the DMA transfer in advance, page locking the memory region where a digital camera, for example, is to write the image data from a picture it is about to take, and specifying that memory location to the camera. Page locking causes the virtual memory manager to forego writing the memory locations in question out to disk. Once this is done, the camera can take the picture and write the image data into memory, then notify the host that the DMA transfer has been completed. In

Ex. C9 – Invalidity of U.S. Patent No. RE49,461 – Wilt

U.S. Patent No. RE49,461	Wilt
	the context of graphics processor control, a series of commands for the graphics processor can be written into memory, and then transferred to the graphics processor via a DMA operation. DMA has many advantages over the other methods of submitting graphics processor commands. First, it facilitates parallel execution by the host and graphics processors. Since these ‘other operations’ to be performed by the CPU often involve generating more commands for the graphics processor, it is common to maintain multiple DMA buffers and ‘ping-pong’ between them, writing to one while the graphics processor reads and executes another. Second, DMA enables multiple applications to write command buffers into memory, and then submit the contents of the command buffers to the graphics processor when they are ready for execution. The graphics processor can only execute one command buffer at a time, but DMA facilitates the simultaneous construction of many command buffers because different threads can write into distinct memory regions that contain command buffers. Third, DMA frequently results in more efficient usage of the bus resources in the system. In other words, the bandwidth achievable through DMA is often higher than the bandwidth achievable by direct participation of the host microprocessor. The Accelerated Graphics Port (AGP) is a specialized bus designed specifically to serve as a high bandwidth conduit to transfer data from system memory to a graphics processor. For the purposes of the present description, AGP may be considered a special case of DMA. Besides the mechanism for delivering information to and receiving information from the hardware, another important distinction to make in our background discussion centers on notification of the host microprocessor that the graphics processor has completed some task, or that some event has occurred that is visible to the graphics processor. Non-limiting examples of tasks include finishing execution of a command buffer or the completion of a specific command. Non-limiting examples of events include vertical retrace of the display device, the completion of a field of digitized video from a video port, or the retracing of a specific scanline in the display device. An alternative method for notifying the host microprocessor of a completed task or an external event is known as an interrupt. An interrupt is a hardware mechanism that diverts the host microprocessor from whatever it is doing and transfers control to a special location in memory. The operating system uses this mechanism to call driver code known as an interrupt service routine to respond to the interrupt.

Ex. C9 – Invalidity of U.S. Patent No. RE49,461 – Wilt

U.S. Patent No. RE49,461	Wilt
	Interrupts have several advantages over polling based mechanisms. For one thing, the cost of setting up and responding to an interrupt is fixed, while the cost of a polling based mechanism increases approximately linearly with the number of polls performed. As a result, interrupt based schemes generally consume less bus bandwidth as well as fewer CPU cycles than polling based schemes. Interrupts can also result in a more timely notification if the interrupt latency is smaller than the time between when the event occurs and when the next poll is performed. [0085]–[0088], [0090]–[0091]. Given these assumptions, we can broaden our focus from graphics processors and contemplate a system with two processors: a host CPU, and a coprocessor that is controlled by sending command buffers from the CPU, that can send data back to the CPU, and that coordinates control by notifying the CPU of various events when they occur. The examples in this document frequently assume that the coprocessor is a graphics processor (as when describing possible synchronization problems and how a system can be designed to solve those problems efficiently), but these examples are for illustrative purposes only and do not limit the scope of this invention to graphics processors. In line with the above assumptions regarding computational resources in a graphics processing system, a non-limiting system is described below that makes efficient, shared use of the computational resources in a coprocessor. FIG. 2 depicts the life cycle of a command buffer as it is used by applications to send commands to the coprocessor, such as a graphics processor. When a command buffer is created and initialized, the buffer enters the initialized state 201. The command buffer is then available to be selected and written into by a thread that wishes to issue commands to the graphics processor. When a thread selects the command buffer, the memory in the command buffer is made available to that thread (perhaps by mapping the memory in the command buffer into that thread's address space) and the command buffer enters the Receiving state 202. As one of ordinary skill in the art can appreciate, the details of this mapping may vary. For example, the command buffer memory could be made visible to user mode, kernel mode, or both. In addition, there are numerous options as to the nature of the executable code that is writing into the Receiving command buffer. In this regard, FIGS. 3A through 3E depict several of these options, i.e., several different architectures for writing into the command buffer.

Ex. C9 – Invalidity of U.S. Patent No. RE49,461 – Wilt

U.S. Patent No. RE49,461	Wilt
	For purposes of describing FIGS. 3A through 3E, an application is a process, or a thread in a process, that implements some specific graphical application participating in the system. The runtime is middleware provided in the operating system that the application uses. Among other things, the runtime may validate the application's usage of an application programming interface (API), translate simple-to-use, high-level API calls into more complicated, lower-level constructs that are more readily translatable to hardware commands, and so on. The driver is a hardware-specific component that translates runtime-level commands to hardware-specific commands. The driver components depicted in FIGS. 3A through 3E are the portions of the driver that translate runtime-level commands to hardware-specific commands. Some portions of the driver that perform tasks other than this translation, such as interrupt handling and command buffer dispatch, would probably reside in kernel mode and are not contemplated in FIGS. 3A through 3E. Most of the object code depicted in FIGS. 3A through 3E may run in user mode, although some portions, as pointed out below, can run in kernel mode. The command buffer is the piece of memory that includes commands that can be consumed directly by the hardware via some DMA mechanism. Several of the figures also show a token stream, which is a hardware-independent stream of runtime-level primitives that can be translated into hardware-specific commands by the driver. Lastly, some of the figures show a JIT (Just-In-Time) compilation component that can translate a hardware-independent intermediate language into object code for execution by the host microprocessor. The boxes with OBJ in the corner include object code that can be directly executed by the microprocessor. Boxes with MSIL in the corner include intermediate code that is compiled by the JIT in order to execute on the microprocessor. If the application and runtime are both available on the client in intermediate form, the JIT can translate them into a unified piece of object code, shown as app executables 324, 334 and 345 FIGS. 3C to 3E, respectively, that implements the application. FIG. 3A shows the application 301, runtime 302 and driver 303 each performing their respective roles, calling into lower-level object code until the driver 303 writes hardware-specific commands into the command buffer 304. The application 301 could be an executable process, or a thread in a process, while the runtime 302 and driver 303 could each reside in DLLs. As depicted in FIG. 3A, all of the object code may reside in user mode (although some or all of the runtime 302 and/or driver 303 may reside in kernel mode). If the runtime 302 determines that the command buffer 304 must be submitted before execution can continue, or

Ex. C9 – Invalidity of U.S. Patent No. RE49,461 – Wilt

U.S. Patent No. RE49,461	Wilt
	the driver 303 determines that the command buffer 304 is out of space, a transition to kernel mode may be requested to submit the command buffer 304 and retrieve another one in order to continue. FIG. 3B shows a variant in which the runtime 312 emits a hardware-independent stream of commands into a buffer 313, which is then parsed by the driver 314 and written into a command buffer 315. Application 311 operates to call lower-level object code in connection with runtime 312. FIG. 3B approximates the current architecture of the DIRECTX® software component, in which the runtime translates application activity into DrawPrimitives2 (“DP2”) tokens. When the DP2 token stream is submitted, a kernel transition occurs and the driver 314 translates the DP2 token stream into hardware-specific commands in kernel mode. FIG. 3B does not make any assumptions about whether the driver 314 is in user mode or kernel mode, and in this regard, driver component 314 could reside in user mode as well as kernel mode. FIG. 3C shows a variant in which the application 321 and runtime 322 are synthesized into a unified piece of object code by a JIT compiler 323. The resulting piece of object code 324 then interacts with the driver 325 to write hardware-specific commands into the command buffer 326. FIG. 3D shows the JIT variant of the situation described for FIG. 3B, in which a hardware-independent token stream 335 is emitted by the software 334 above the driver 336. The driver 336 then translates the token stream to hardware-specific commands for the command buffer 337. As with FIG. 3C, application 331 and runtime 332 are synthesized into a unified piece of object code 334 by JIT compiler 333. FIG. 3E shows a system in which the application 341, runtime 342, and part of the driver 343 are all available on the client in intermediate language form MSIL. The JIT compiler 344 synthesizes these components into a unified piece of object code 345 specifically optimized to run on that particular client computer, taking into account not only the host microprocessor but also the coprocessor hardware configuration. Referring back to the command buffer life cycle depicted in FIG. 2, regardless of the method by which commands are written into the command buffer, the thread writing into the receiving command buffer eventually submits to the command buffer either because the command buffer has been filled, or because the drawing commands that have been written into the buffer are executed forthwith. At that time, the command buffer enters the waiting state 203, and the contents of the command buffer may not be submitted to the hardware until all of the resources

Ex. C9 – Invalidity of U.S. Patent No. RE49,461 – Wilt

U.S. Patent No. RE49,461	Wilt
	utilized to execute the command buffer are available. As an example of a circumstance that may force a wait, if an application draws into a surface, the drawing must be completed before that surface may be selected as a texture. Once the waiting criteria have been fulfilled, the command buffer enters the ready state 204, where the scheduler considers it for submission to the coprocessor. The scheduler has a great deal of flexibility in deciding which command buffer should be submitted to the hardware. The scheduler may take into account the priority, if any, of the command buffer, the priority of the requesting thread, the real-time requirements of the system, and so on. Once the command buffer has been selected as the next to run, the command buffer enters the standby state 205. On some systems, only one command buffer per coprocessor may be in the standby state 205 in the system. When the command buffer in the standby state 205 can be submitted to the hardware, the system switches the coprocessor context to the context required by the command buffer, and then submits the command buffer to the hardware. This operation entails some involvement from the driver, for example to initiate a DMA operation using the base address and size of the command buffer. The command buffer remains in the running state 206 until some event preempts execution of the command buffer or the entire contents of the command buffer have been read and executed. If the entire command buffer has been executed, the command buffer enters the terminated state 207, where it becomes available for the system to reinitialize and make available to another thread in the initialized state 201. If the command buffer is preempted, it is assumed that the hardware has the capability to save the hardware context in such a way that execution of the command buffer can be resumed where the preemption occurred. The command buffer can then be recycled into the ready state 204 and be re-submitted to the hardware to resume from where it stopped. The description below regarding preemptible command buffer execution describes this in more detail. [0095]–[0110]. To the extent Plaintiff contends that this claim element is not expressly or inherently disclosed, or is not obvious in view of Wilt and the knowledge of a POSITA, it would have been obvious to combine the teachings of Wilt with any of the prior art references in NVIDIA’s Invalidity Contentions and any supplements thereto and the relevant section of charts for other prior art

Ex. C9 – Invalidity of U.S. Patent No. RE49,461 – Wilt

U.S. Patent No. RE49,461	Wilt
	for the '461 patent in a manner that would result in the subject matter of this limitation given, at the very least, the person's understanding of the state of the art, the problems addressed and solved in the prior art, and the teachings of Wilt. Specific examples of such references are cited and explained in these invalidity contentions, including the cover pleading and charts C1–C15, and Appendix C.
31. The method of claim 21, further comprising: executing, by the at least one CPU, a data output stream, the data output stream controlling transfer of outputs of the computations representing the artificial neural network to disk.	Wilt discloses the method of claim 21, further comprising: executing, by the at least one CPU, a data output stream, the data output stream controlling transfer of outputs of the computations representing the artificial neural network to disk. See claim 21. Computing Environment 100 The diagram illustrates a computing environment with a central dashed box labeled 'Computing Environment 100'. Inside this box, a 'System Memory' block (130) contains 'ROM' (131) with 'BIOS' (133) and 'RAM' (132) containing 'Operating System' (134), 'Application Programs' (135), 'Other Program Modules' (136), and 'Program Data' (137). A 'Processing Unit' (120) is connected to the system memory via a 'System Bus' (121). The processing unit is also connected to a 'GPU' (184) and 'Video Memory' (188) via a 'Graphics Interface' (182) and 'Video Interface' (190). A 'Non-Removable Non-Volatile Memory Interface' (140) and a 'Removable Non-Volatile Memory Interface' (150) are connected to the system bus. A 'User Input Interface' (160) is connected to the processing unit. A 'Network Interface' (170) is connected to the system bus. Outside the dashed box, a 'Monitor' (191) is connected to the video interface. A 'Printer' (196) and 'Speakers' (197) are connected to the output peripheral interface (185). A 'Mouse' (161) and 'Keyboard' (162) are connected to the user input interface. A 'Modem' (172) is connected to the network interface. A 'Local Area Network' (171) and a 'Wide Area Network' (173) are connected to the network interface. A 'Remote Computer' (180) is connected to the wide area network. A 'Remote Application Programs' (181) block is connected to the remote computer. Below the diagram, the text 'FIG. 1B' is displayed. FIG. 1B

Ex. C9 – Invalidity of U.S. Patent No. RE49,461 – Wilt

U.S. Patent No. RE49,461	Wilt
	Direct Memory Access (DMA) is a common method for moving data within computer systems, and has found application in the control of graphics processors as well as disk drives, network adapters, and external buses such as USB and IEEE 1394. DMA generally refers to the ability of a peripheral, such as a digital camera or disk controller, to read from or write to memory without directly involving the host processor. The host sets up the DMA transfer in advance, page locking the memory region where a digital camera, for example, is to write the image data from a picture it is about to take, and specifying that memory location to the camera. Page locking causes the virtual memory manager to forego writing the memory locations in question out to disk. Once this is done, the camera can take the picture and write the image data into memory, then notify the host that the DMA transfer has been completed. In the context of graphics processor control, a series of commands for the graphics processor can be written into memory, and then transferred to the graphics processor via a DMA operation. DMA has many advantages over the other methods of submitting graphics processor commands. First, it facilitates parallel execution by the host and graphics processors. Since these ‘other operations’ to be performed by the CPU often involve generating more commands for the graphics processor, it is common to maintain multiple DMA buffers and ‘ping-pong’ between them, writing to one while the graphics processor reads and executes another. Second, DMA enables multiple applications to write command buffers into memory, and then submit the contents of the command buffers to the graphics processor when they are ready for execution. The graphics processor can only execute one command buffer at a time, but DMA facilitates the simultaneous construction of many command buffers because different threads can write into distinct memory regions that contain command buffers. Third, DMA frequently results in more efficient usage of the bus resources in the system. In other words, the bandwidth achievable through DMA is often higher than the bandwidth achievable by direct participation of the host microprocessor. The Accelerated Graphics Port (AGP) is a specialized bus designed specifically to serve as a high bandwidth conduit to transfer data from system memory to a graphics processor. For the purposes of the present description, AGP may be considered a special case of DMA. Besides the mechanism for delivering information to and receiving information from the hardware, another important distinction to make in our background discussion centers on notification of the host microprocessor that the graphics processor has completed some task, or that some event has occurred that is visible to the graphics processor. Non-limiting examples

Ex. C9 – Invalidity of U.S. Patent No. RE49,461 – Wilt

U.S. Patent No. RE49,461	Wilt
	of tasks include finishing execution of a command buffer or the completion of a specific command. Non-limiting examples of events include vertical retrace of the display device, the completion of a field of digitized video from a video port, or the retracing of a specific scanline in the display device. An alternative method for notifying the host microprocessor of a completed task or an external event is known as an interrupt. An interrupt is a hardware mechanism that diverts the host microprocessor from whatever it is doing and transfers control to a special location in memory. The operating system uses this mechanism to call driver code known as an interrupt service routine to respond to the interrupt. Interrupts have several advantages over polling based mechanisms. For one thing, the cost of setting up and responding to an interrupt is fixed, while the cost of a polling based mechanism increases approximately linearly with the number of polls performed. As a result, interrupt based schemes generally consume less bus bandwidth as well as fewer CPU cycles than polling based schemes. Interrupts can also result in a more timely notification if the interrupt latency is smaller than the time between when the event occurs and when the next poll is performed. [0085]–[0088], [0090]–[0091]. Given these assumptions, we can broaden our focus from graphics processors and contemplate a system with two processors: a host CPU, and a coprocessor that is controlled by sending command buffers from the CPU, that can send data back to the CPU, and that coordinates control by notifying the CPU of various events when they occur. The examples in this document frequently assume that the coprocessor is a graphics processor (as when describing possible synchronization problems and how a system can be designed to solve those problems efficiently), but these examples are for illustrative purposes only and do not limit the scope of this invention to graphics processors. In line with the above assumptions regarding computational resources in a graphics processing system, a non-limiting system is described below that makes efficient, shared use of the computational resources in a coprocessor. FIG. 2 depicts the life cycle of a command buffer as it is used by applications to send commands to the coprocessor, such as a graphics processor. When a command buffer is created and initialized, the buffer enters the initialized state 201. The command buffer is then

Ex. C9 – Invalidity of U.S. Patent No. RE49,461 – Wilt

U.S. Patent No. RE49,461	Wilt
	available to be selected and written into by a thread that wishes to issue commands to the graphics processor. When a thread selects the command buffer, the memory in the command buffer is made available to that thread (perhaps by mapping the memory in the command buffer into that thread's address space) and the command buffer enters the Receiving state 202. As one of ordinary skill in the art can appreciate, the details of this mapping may vary. For example, the command buffer memory could be made visible to user mode, kernel mode, or both. In addition, there are numerous options as to the nature of the executable code that is writing into the Receiving command buffer. In this regard, FIGS. 3A through 3E depict several of these options, i.e., several different architectures for writing into the command buffer. For purposes of describing FIGS. 3A through 3E, an application is a process, or a thread in a process, that implements some specific graphical application participating in the system. The runtime is middleware provided in the operating system that the application uses. Among other things, the runtime may validate the application's usage of an application programming interface (API), translate simple-to-use, high-level API calls into more complicated, lower-level constructs that are more readily translatable to hardware commands, and so on. The driver is a hardware-specific component that translates runtime-level commands to hardware-specific commands. The driver components depicted in FIGS. 3A through 3E are the portions of the driver that translate runtime-level commands to hardware-specific commands. Some portions of the driver that perform tasks other than this translation, such as interrupt handling and command buffer dispatch, would probably reside in kernel mode and are not contemplated in FIGS. 3A through 3E. Most of the object code depicted in FIGS. 3A through 3E may run in user mode, although some portions, as pointed out below, can run in kernel mode. The command buffer is the piece of memory that includes commands that can be consumed directly by the hardware via some DMA mechanism. Several of the figures also show a token stream, which is a hardware-independent stream of runtime-level primitives that can be translated into hardware-specific commands by the driver. Lastly, some of the figures show a JIT (Just-In-Time) compilation component that can translate a hardware-independent intermediate language into object code for execution by the host microprocessor. The boxes with OBJ in the corner include object code that can be directly executed by the microprocessor. Boxes with MSIL in the corner include intermediate code that is compiled by the JIT in order to execute on the microprocessor. If the application and runtime are both available on the client in intermediate form, the JIT can translate them into a

Ex. C9 – Invalidity of U.S. Patent No. RE49,461 – Wilt

U.S. Patent No. RE49,461	Wilt
	unified piece of object code, shown as app executables 324, 334 and 345 FIGS. 3C to 3E, respectively, that implements the application. FIG. 3A shows the application 301, runtime 302 and driver 303 each performing their respective roles, calling into lower-level object code until the driver 303 writes hardware-specific commands into the command buffer 304. The application 301 could be an executable process, or a thread in a process, while the runtime 302 and driver 303 could each reside in DLLs. As depicted in FIG. 3A, all of the object code may reside in user mode (although some or all of the runtime 302 and/or driver 303 may reside in kernel mode). If the runtime 302 determines that the command buffer 304 must be submitted before execution can continue, or the driver 303 determines that the command buffer 304 is out of space, a transition to kernel mode may be requested to submit the command buffer 304 and retrieve another one in order to continue. FIG. 3B shows a variant in which the runtime 312 emits a hardware-independent stream of commands into a buffer 313, which is then parsed by the driver 314 and written into a command buffer 315. Application 311 operates to call lower-level object code in connection with runtime 312. FIG. 3B approximates the current architecture of the DIRECTX® software component, in which the runtime translates application activity into DrawPrimitives2 (“DP2”) tokens. When the DP2 token stream is submitted, a kernel transition occurs and the driver 314 translates the DP2 token stream into hardware-specific commands in kernel mode. FIG. 3B does not make any assumptions about whether the driver 314 is in user mode or kernel mode, and in this regard, driver component 314 could reside in user mode as well as kernel mode. FIG. 3C shows a variant in which the application 321 and runtime 322 are synthesized into a unified piece of object code by a JIT compiler 323. The resulting piece of object code 324 then interacts with the driver 325 to write hardware-specific commands into the command buffer 326. FIG. 3D shows the JIT variant of the situation described for FIG. 3B, in which a hardware-independent token stream 335 is emitted by the software 334 above the driver 336. The driver 336 then translates the token stream to hardware-specific commands for the command buffer 337. As with FIG. 3C, application 331 and runtime 332 are synthesized into a unified piece of object code 334 by JIT compiler 333. FIG. 3E shows a system in which the application 341, runtime 342, and part of the driver 343 are all available on the client in intermediate language form MSIL. The JIT compiler 344

Ex. C9 – Invalidity of U.S. Patent No. RE49,461 – Wilt

U.S. Patent No. RE49,461	Wilt
	synthesizes these components into a unified piece of object code 345 specifically optimized to run on that particular client computer, taking into account not only the host microprocessor but also the coprocessor hardware configuration. Referring back to the command buffer life cycle depicted in FIG. 2, regardless of the method by which commands are written into the command buffer, the thread writing into the receiving command buffer eventually submits to the command buffer either because the command buffer has been filled, or because the drawing commands that have been written into the buffer are executed forthwith. At that time, the command buffer enters the waiting state 203, and the contents of the command buffer may not be submitted to the hardware until all of the resources utilized to execute the command buffer are available. As an example of a circumstance that may force a wait, if an application draws into a surface, the drawing must be completed before that surface may be selected as a texture. Once the waiting criteria have been fulfilled, the command buffer enters the ready state 204, where the scheduler considers it for submission to the coprocessor. The scheduler has a great deal of flexibility in deciding which command buffer should be submitted to the hardware. The scheduler may take into account the priority, if any, of the command buffer, the priority of the requesting thread, the real-time requirements of the system, and so on. Once the command buffer has been selected as the next to run, the command buffer enters the standby state 205. On some systems, only one command buffer per coprocessor may be in the standby state 205 in the system. When the command buffer in the standby state 205 can be submitted to the hardware, the system switches the coprocessor context to the context required by the command buffer, and then submits the command buffer to the hardware. This operation entails some involvement from the driver, for example to initiate a DMA operation using the base address and size of the command buffer. The command buffer remains in the running state 206 until some event preempts execution of the command buffer or the entire contents of the command buffer have been read and executed. If the entire command buffer has been executed, the command buffer enters the terminated state 207, where it becomes available for the system to reinitialize and make available to another thread in the initialized state 201. If the command buffer is preempted, it is assumed that the hardware has the capability to save the hardware context in such a way that execution of the command buffer can be resumed

Ex. C9 – Invalidity of U.S. Patent No. RE49,461 – Wilt

U.S. Patent No. RE49,461	Wilt
	where the preemption occurred. The command buffer can then be recycled into the ready state 204 and be re-submitted to the hardware to resume from where it stopped. The description below regarding preemptible command buffer execution describes this in more detail. [0095]–[0110]. To the extent Plaintiff contends that this claim element is not expressly or inherently disclosed, or is not obvious in view of Wilt and the knowledge of a POSITA, it would have been obvious to combine the teachings of Wilt with any of the prior art references in NVIDIA’s Invalidity Contentions and any supplements thereto and the relevant section of charts for other prior art for the ’461 patent in a manner that would result in the subject matter of this limitation given, at the very least, the person's understanding of the state of the art, the problems addressed and solved in the prior art, and the teachings of Wilt. Specific examples of such references are cited and explained in these invalidity contentions, including the cover pleading and charts C1–C15, and Appendix C.