

Ex. A8 – Invalidity of U.S. Patent No. 8,648,867 – Nickolls

Prior Art Reference: U.S. Patent No. 7,861,060 (“Nickolls”)

Nickolls was filed on December 15, 2005, and therefore qualifies as prior art to U.S. Patent No. 8,648,867 (the “’867 patent”) under at least pre-AIA 35 U.S.C. § 102(e). Nickolls anticipates and/or renders obvious claims 16–19 of the ’867 patent (the “Asserted Claims”).

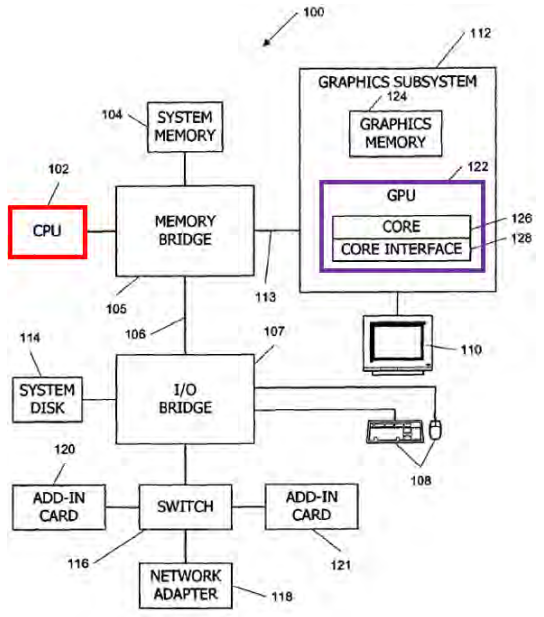
The chart below provides representative examples of where each element of each claim is found within Nickolls. Citations are meant to be exemplary, not exhaustive, and NVIDIA reserves the right to identify and discuss additional portions of the reference in support of its contentions and/or to rebut arguments made by Plaintiff. Where NVIDIA states that Nickolls “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 Nickolls, a person of ordinary skill in the art would have found it obvious to combine the teachings of Nickolls 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 Nickolls. 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. A8 – Invalidity of U.S. Patent No. 8,648,867 – Nickolls

U.S. Patent No. 8,648,867	Nickolls
16<pre>]: A method for performing a numerical simulation on input data in a computer system including a central processing unit and an accelerator, the method comprising:</pre>	To the extent the preamble is limiting, Nicholls discloses “[a] method for performing a numerical simulation on input data in a computer system including a central processing unit and an accelerator.” <i>See e.g.</i>: <i>As a non-limiting example, Nickolls discloses a computer system with a central processing unit (e.g., CPU 102) and an accelerator (e.g., GPU 122):</i>  FIG. 1 Fig. 1 (annotated). FIG.1 is a block diagram of a computer system 100 according to an embodiment of the present invention. Computer system 100 includes a central processing unit (CPU) 102 and a system memory 104 communicating via a bus path that includes a memory bridge 105. Memory bridge 105 is connected via a bus path 106 to an I/O (input/output) bridge 107. I/O bridge 107 receives user input from one or more user input devices 108 (e.g., keyboard, mouse) and forwards the input

Ex. A8 – Invalidity of U.S. Patent No. 8,648,867 – Nickolls

U.S. Patent No. 8,648,867	Nickolls
	to CPU 102 via bus 106 and memory bridge 105. Visual output is provided on a pixel based display device 110 (e.g., a conventional CRT or LCD based monitor) operating under control of a graphics subsystem 112 coupled to memory bridge 105 via a bus 113. A system disk 114 is also connected to I/O bridge 107. A switch 116 provides connections between I/O bridge 107 and other components such as a network adapter 118 and various add-in cards 120, 121. Other components (not explicitly shown), including USB or other port connections, CD drives, DVD drives, and the like, may also be connected to I/O bridge 107. Bus connections among the various components may be implemented using bus protocols such as PCI (Peripheral Component Interconnect), PCI Express (PCI-E), AGP (Accelerated Graphics Port), Hyper-Transport, or any other bus protocol(s), and connections between different devices may use different protocols as is known in the art. 6:9–34. Graphics processing subsystem 112 includes a graphics processing unit (GPU) 122 and a graphics memory 124, which may be implemented, e.g., using one or more integrated circuit devices such as programmable processors, application specific integrated circuits (ASICs), and memory devices. GPU 122 may be configured to perform various tasks related to generating pixel data from graphics data supplied by CPU 102 and/or system memory 104 via memory bridge 105 and bus 113, interacting with graphics memory 124 to store and update pixel data, and the like. For example, GPU 122 may generate pixel data from 2-D or 3-D scene data provided by various programs executing on CPU 102. GPU 122 has at least one processing core 126 for generating pixel data and a core interface 128 that controls operation of core 126. Core 126 advantageously includes multiple parallel processing engines that can be used to execute various shader programs, including vertex shader programs, geometry shader programs, and/or pixel shader programs, in the course of generating images from scene data. Core 126 can also be leveraged to perform general-purpose computations as described below. GPU 122 may also include other components, not explicitly shown, such as a memory interface that can store pixel data received via memory bridge 105 to graphics memory 124 with or without

Ex. A8 – Invalidity of U.S. Patent No. 8,648,867 – Nickolls

U.S. Patent No. 8,648,867	Nickolls
	further processing, a scan out module configured to deliver pixel data from graphics memory 124 to display device 110, and so on. 6:35–61. <i>Nickolls further discloses a method for using the system with a central processing unit and an accelerator to perform numerical simulations on input data:</i> The present invention relates in general to parallel data processing, and in particular to parallel data processing methods using arrays of threads that are capable of sharing data, including intermediate results, with other threads in a thread-specific manner. 1:21–26. Embodiments of the present invention provide data processing systems and methods that use cooperative thread arrays (CTAs) to perform computations. As used herein, a “cooperative thread array,” or “CTA,” is a group of multiple threads that concurrently execute the same program on an input data set to produce an output data set. Each thread in a CTA has a unique identifier (thread ID) assigned at thread launch time that controls various aspects of the thread’s processing behavior. For instance, a thread ID may be used to determine which portion of the input data set a thread is to process, to identify one or more threads with which a given thread is to share an intermediate result, and/or to determine which portion of the output data set the thread is to compute or write. 2:21–23. Embodiments of the present invention provide data processing systems and methods that use cooperative thread arrays (CTAs) to perform computations. As used herein, a "cooperative thread array," or "CTA," is a group of multiple threads that concurrently execute the same program on an input data set to produce an output data set. Each thread in a CTA has a unique identifier (thread ID) assigned at thread launch time that controls various aspects of the thread's processing behavior. For instance, a thread ID may be used to determine which portion of the input data set a thread is to process, to identify one or more other threads with which a given thread is to share an intermediate result, and/or to determine which portion of the output data set the thread is to compute or write.

Ex. A8 – Invalidity of U.S. Patent No. 8,648,867 – Nickolls

U.S. Patent No. 8,648,867	Nickolls
	5:31–44. According to one aspect of the present invention, a computer implemented method for processing data includes defining, in a processor, a thread array having multiple threads, each thread configured to execute a same program on an input data set. In some embodiments, the program represents a unit of a data parallel decomposition of a data processing algorithm. The processor assigns each of the threads in the thread array a unique thread identifier (which can be, e.g., a one-dimensional or multidimensional identifier). The processor executes the threads in the thread array concurrently with each other; during execution, each thread uses the unique thread identifier to determine at least one processing behavior. Any processor, including but not limited to a graphics processor, may be used to implement the method. Parallel and multithreaded processors can execute multiple threads of a CTA in parallel for enhanced performance. 2:65–3:13. CTAs are advantageously employed to perform computations that lend themselves to a data parallel decomposition, i.e., application of the same processing algorithm to different portions of an input data set in order to effect a transformation of the input data set to an output data set. The processing algorithm is specified in a "CTA program," and each thread in a CTA executes the same CTA program on a different subset of an input data set. A CTA program can implement algorithms using a wide range of mathematical and logical operations, and the program can include conditional or branching execution paths and direct and/or indirect memory access. 7:66–8:9. <i>Nickolls further provides examples of the kinds of numerical computations (including numerical simulations) that its system performs:</i> The Fast Fourier Transform (FFT) algorithm is a well-known example of a data-parallel decomposition. As is generally known, discrete Fourier transforms can be computed as a dot products of an N-component sample vector with each of N basis vectors, requiring $O(N^2)$ multiplications and additions. To reduce the computational burden to $O(N \log N)$, the FFT algorithm employs a data-parallel decomposition: the well-known "butterfly" calculations.

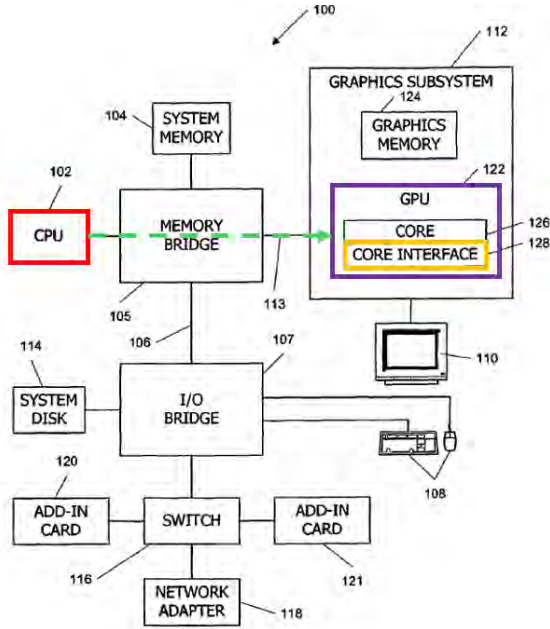
Ex. A8 – Invalidity of U.S. Patent No. 8,648,867 – Nickolls

U.S. Patent No. 8,648,867	Nickolls
	In accordance with an embodiment of the present invention, FFT butterfly calculations are advantageously implemented in a CTA. Each thread starts by reading its own input value, then reads an input value of another thread (determined based on thread IDs) and performs two multiplications and two additions to arrive at the next intermediate value. The next intermediate value is written back to the global register file so that it can be shared with another thread, and so on. At each FFT stage, the CTA threads synchronize and wait for other threads' intermediate values to be written before reading those intermediate values. Matrix algebra provides many more examples of data parallel decompositions. In matrix multiplication, for instance, each element of a product matrix is determined by multiplying each element of a row of one matrix by a corresponding element of a column of the other matrix and summing the products. CTAs can be used for this process, as well as for more complex operations such as linear equation solving (which generally requires inverting a matrix). In this application, multidimensional thread IDs advantageously support either row-major or column-major addressing in a natural fashion, which facilitates operations that require a matrix transpose. Convolution filters can also be implemented using a data-parallel decomposition. As is known in the art, convolution filters are often employed in various types of signal processing to reduce noise and other artifacts. In the case of image processing, for instance, a two-dimensional kernel may be defined, and a value for a particular pixel is determined by convolving the kernel with the pixel and its neighbors. CTAs can be used to convolve multiple pixels with the same kernel in parallel, and multiple CTAs can be used to apply the filter kernel to different regions of the image in parallel. More generally, a CTA can implement a convolution filter in an arbitrary number of dimensions, without regard to the particular application for which convolution filtering is being employed. Separable filters, a special case of convolution filters, can also be implemented using a data parallel decomposition. A two-dimensional (N×N) separable filter is a convolution filter in which the filter can be expressed in terms of a "row" vector and a "column" vector. The row vector is applied to each row 55 as a convolution filter with a kernel that is one row high and N columns wide, and the column vector is applied to the row-filter results as a convolution filter with a kernel that is one column wide and N rows high. A CTA for a separable filter can be configured with N

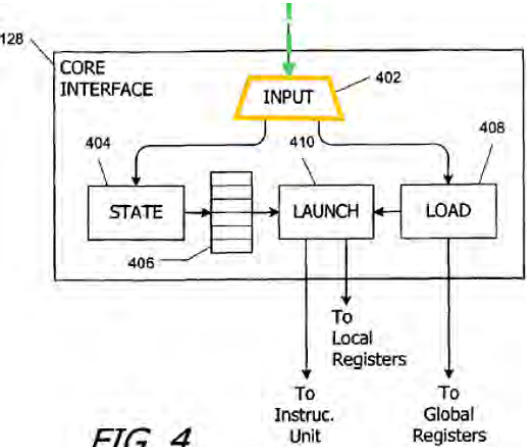
Ex. A8 – Invalidity of U.S. Patent No. 8,648,867 – Nickolls

U.S. Patent No. 8,648,867	Nickolls
	threads. Each thread applies the row vector to one of N rows, writing its row result to shared memory. After each CTA thread has written its row result to shared memory, it waits for the other CTA threads to write their row results. Barrier synchronization, as described above, or other synchronization techniques may be used to ensure that a row result is available in the shared memory before any thread attempts to read that row result. Thereafter, each thread uses the row results from shared memory to apply the column vector to one of N columns. Embodiments of the present invention allow an application programmer to define a data parallel decomposition suitable for a particular general-purpose computing application and to control the extent and manner of sharing of data between various threads. The programmer also defines dimensions of one or more CTAs that are to execute the algorithm and supplies the input data. The manner in which the CTA is executed is advantageously transparent to the programmer. In particular, the programmer is not required to know details of how the processing hardware executes a CTA, such as whether there is SIMD parallelism, the number of threads in a SIMD group, and so on. CTAs can be executed on parallel processors, multi-threaded processors, vector processors, or any other processor capable of exploiting the explicit parallelism made available by the concurrent threads of the CTA and the parallelism made available by concurrent execution of multiple CTAs. CTAs can also be executed on sequential processors such as conventional CPUs by exploiting software techniques such as thread scheduling, although performance in such systems might not be as high as in systems that can leverage CTA parallelism. In some embodiments, a CTA provides a flexible, general-purpose computational capacity in a GPU that may be used for computations in any field, including but not limited to bioinformatics, seismic signal processing, modeling and simulation, matrix algebra, physics, chemistry, image processing, supercomputing, and so on. 29:6–30:31. To the extent Plaintiff contends that the preamble is not expressly or inherently disclosed, or is not obvious in view of Nicholls and the knowledge of a POSITA, it would have been obvious to combine the teachings of Nicholls with any of the prior art references in NVIDIA’s Invalidity

Ex. A8 – Invalidity of U.S. Patent No. 8,648,867 – Nickolls

U.S. Patent No. 8,648,867	Nickolls
	Contentions and any supplements thereto and the relevant section of charts for other prior art for the '867 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 Nicholls. Specific examples of such references are cited and explained in these invalidity contentions, including the cover pleading and charts A1–A13, and Appendix A.
16[a]: receiving, by an accelerator, first input data from the central processing unit;	Nicholls discloses “receiving, by an accelerator, first input data from the central processing unit.” <i>See e.g.:</i> <i>As a non-limiting example, Nickolls discloses an accelerator (e.g., GPU 122) receiving “first input data,” such as a data word, state information, or other input data to use in computations, from a central processing unit (e.g., CPU 102):</i>  FIG. 1

Ex. A8 – Invalidity of U.S. Patent No. 8,648,867 – Nickolls

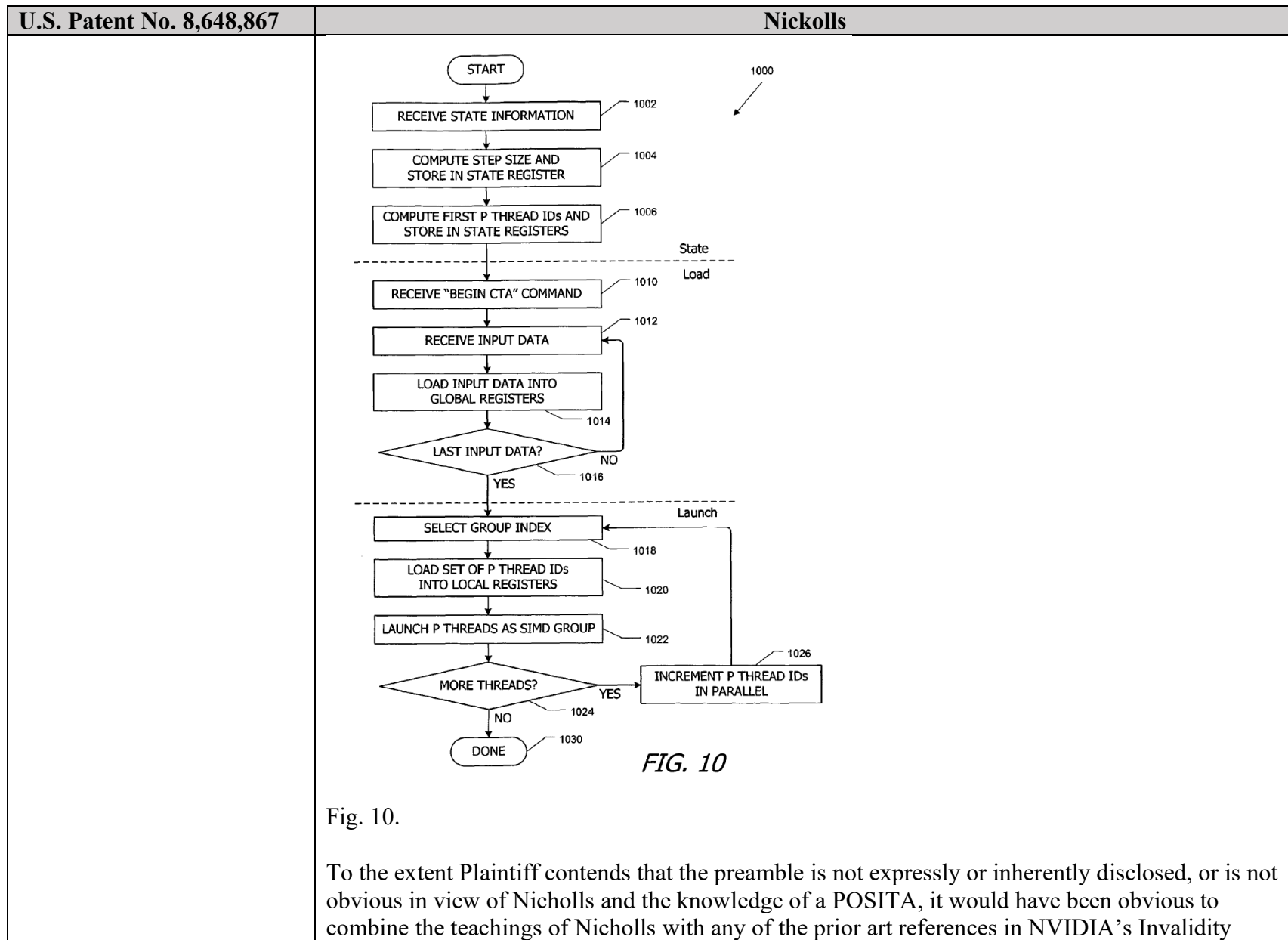
U.S. Patent No. 8,648,867	Nickolls
	Fig. 1 (annotated).  FIG. 4 Fig. 4 (annotated). Graphics processing subsystem 112 includes a graphics processing unit (GPU) 122 and a graphics memory 124, which may be implemented, e.g., using one or more integrated circuit devices such as programmable processors, application specific integrated circuits (ASICs), and memory devices. GPU 122 may be configured to perform various tasks related to generating pixel data from graphics data supplied by CPU 102 and/or system memory 104 via memory bridge 105 and bus 113, interacting with graphics memory 124 to store and update pixel data, and the like. For example, GPU 122 may generate pixel data from 2-D or 3-D scene data provided by various programs executing on CPU 102. 6:35–46. CPU 102 operates as the master processor of system 100, controlling and coordinating operations of other system components. In particular, CPU 102 issues commands that control the operation of GPU 122. In some embodiments, CPU 102 writes a stream of commands for GPU 122 to a command buffer, which may be in system memory 104, graphics memory 124, or another storage

Ex. A8 – Invalidity of U.S. Patent No. 8,648,867 – Nickolls

U.S. Patent No. 8,648,867	Nickolls
	location accessible to both CPU 102 and GPU 122. GPU 122 reads the command stream from the command buffer and executes commands asynchronously with operation of CPU 102. 6:62–7:4. Input unit 402 receives all incoming signals, including state information, data and commands, and directs the signals to state module 404 or load module 406. In one embodiment, input unit 402 receives a data word (e.g., 32 bits) along with control signals indicating whether the data word corresponds to state information or data to be processed. Based on the control signals, input unit 402 determines where to direct the data word, and all or part of the control signal may be forwarded together with the data word. For instance, if the data word corresponds to state information, the control signals may be used by state module 404 to determine where to store the data word, as described below. 14:29–31. Referring again to FIG. 4, load module 408 receives input parameters for a CTA and loads the input parameters into (shared) global register file 306 (FIG. 3) of core 126. These parameters may include, e.g., an (x,y) position of the CTA within a grid of CTAs or other CTA identifier and/or other information specific to the CTA. In some embodiments, the input parameters may also include some or all of an input data set to be processed by the CTA. 16:63–17:3. Core interface 128 remains in the state phase and continues 40 to update the state registers as new state information is received, until such time as core interface 128 receives a "begin CTA" command (step 1010) indicating that input data is to follow. At that point, core interface 128 enters the load phase. During the load phase, core interface 128 receives input data (step 1012) and loads the input data into (shared) global register file 306 of core 126 (step 1014). At step 1016, core interface 128 determines whether the last input data has been received. If not, core interface 128 returns to step 1012 to receive more input data. 19:40–50.

Ex. A8 – Invalidity of U.S. Patent No. 8,648,867 – Nickolls

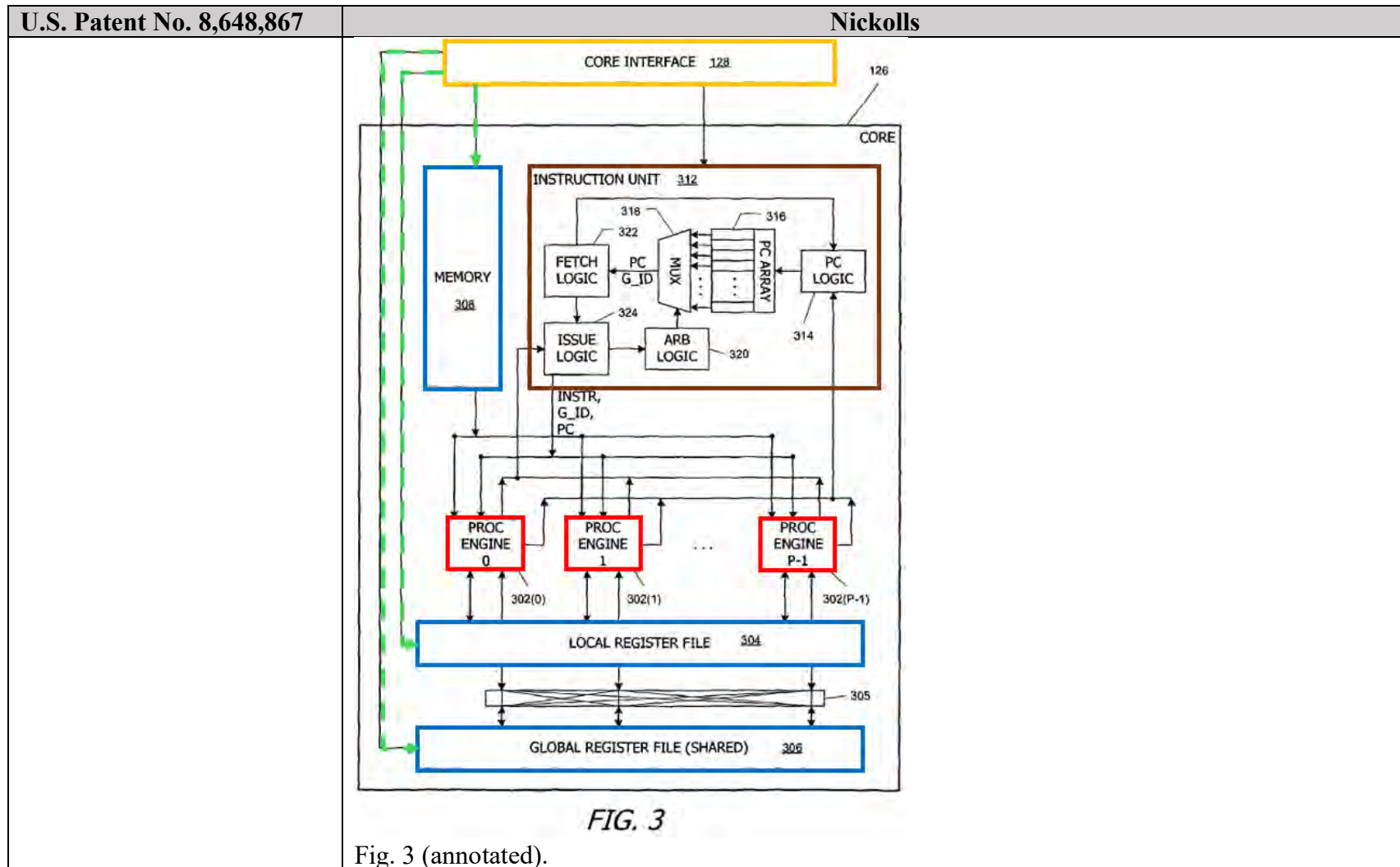
Ex. A8 – Invalidity of U.S. Patent No. 8,648,867 – Nickolls



Ex. A8 – Invalidity of U.S. Patent No. 8,648,867 – Nickolls

U.S. Patent No. 8,648,867	Nickolls
	Contentions and any supplements thereto and the relevant section of charts for other prior art for the '867 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 Nicholls. Specific examples of such references are cited and explained in these invalidity contentions, including the cover pleading and charts A1–A13, and Appendix A.
16[b]: transferring, by an accelerator controller, the first input data into a first partition, referenced by first pointer, of an accelerator memory before a first computational cycle of the numerical simulation;	Nicholls discloses “transferring, by an accelerator controller, the first input data into a first partition, referenced by first pointer, of an accelerator memory before a first computational cycle of the numerical simulation.” <i>See e.g.:</i> <i>As a non-limiting example, Nickolls discloses an accelerator controller (e.g., core interface 128, which controls operation of core 126) and accelerator memory (e.g., memory 308, local register file 304, and/or global register file 306). Nickolls further discloses transferring the first input data (see 16[a]) into a “first partition” of the accelerator memory before a first computational cycle of the numerical simulation (e.g., by allocating and storing input data in accelerator memory). Nickolls also discloses referencing partitions by pointers (e.g., using indirect memory access, including calculating an address (which can be stored in a pointer) at which data can be accessed).</i>

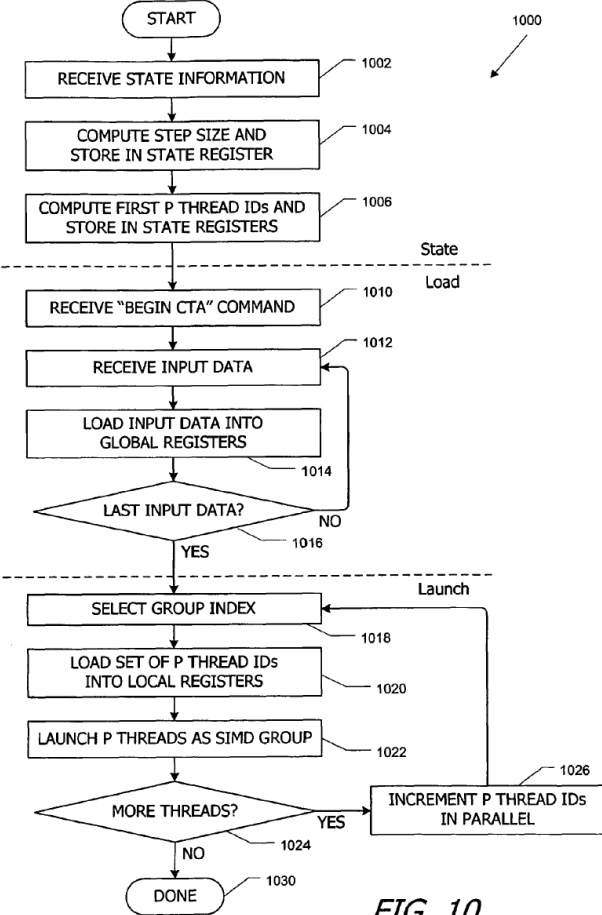
Ex. A8 – Invalidity of U.S. Patent No. 8,648,867 – Nickolls



Ex. A8 – Invalidity of U.S. Patent No. 8,648,867 – Nickolls

U.S. Patent No. 8,648,867	Nickolls
	FIG. 4 Fig. 4 (annotated).

Ex. A8 – Invalidity of U.S. Patent No. 8,648,867 – Nickolls

U.S. Patent No. 8,648,867	Nickolls
	 <pre> graph TD START([START]) --> 1002[RECEIVE STATE INFORMATION] 1002 --> 1004[COMPUTE STEP SIZE AND STORE IN STATE REGISTER] 1004 --> 1006[COMPUTE FIRST P THREAD IDs AND STORE IN STATE REGISTERS] 1006 --> 1010[RECEIVE "BEGIN CTA" COMMAND] 1010 --> 1012[RECEIVE INPUT DATA] 1012 --> 1014[LOAD INPUT DATA INTO GLOBAL REGISTERS] 1014 --> 1016{LAST INPUT DATA?} 1016 -- NO --> 1012 1016 -- YES --> 1018[SELECT GROUP INDEX] 1018 --> 1020[LOAD SET OF P THREAD IDs INTO LOCAL REGISTERS] 1020 --> 1022[LAUNCH P THREADS AS SIMD GROUP] 1022 --> 1024{MORE THREADS?} 1024 -- YES --> 1026[INCREMENT P THREAD IDs IN PARALLEL] 1026 -- Launch --> 1018 1024 -- NO --> 1030([DONE]) </pre> FIG. 10 Fig. 10. According to another aspect of the present invention, a method for processing an input data set using a processor (which might be, e.g., a graphics processor) configured to execute a plurality of threads concurrently includes communicating to the processor information defining a thread array having a plurality of threads, each thread being identified by a unique thread identifier (which can

Ex. A8 – Invalidity of U.S. Patent No. 8,648,867 – Nickolls

U.S. Patent No. 8,648,867	Nickolls
	be, e.g., a one-dimensional or multidimensional identifier). Also communicated to the processor is a program to be executed by each thread of the thread array on at least a portion of an input data set stored in a shared memory to produce at least a portion of an output data set; during program execution, the unique thread identifier is used by each thread to determine at least one processing behavior. The input data set is communicated to the processor, and the processor stores the input data set in a shared memory communicably coupled to the processing engines. An instruction to begin concurrently executing all of the threads of the thread array to process the input data set in the shared memory to produce the output data set is communicated to the processor. In some embodiments, each thread processes a different portion of the input data set, and in some embodiments, each thread produces a different portion of the output data set. 4:29–31. Embodiments of the present invention provide data processing systems and methods that use cooperative thread arrays (CTAs) to perform computations. As used herein, a "cooperative thread array," or "CTA," is a group of multiple threads that concurrently execute the same program on an input data set to produce an output data set. Each thread in a CTA has a unique identifier (thread ID) assigned at thread launch time that controls various aspects of the thread's processing behavior. For instance, a thread ID may be used to determine which portion of the input data set a thread is to process, to identify one or more other threads with which a given thread is to share an intermediate result, and/or to determine which portion of the output data set the thread is to compute or write. CTAs can be executed in a variety of processing architectures, including any sequential single processors, parallel processors, multithreaded processors, and any architecture that can support multiple concurrent threads and that provides at least some shared memory, interconnection network, or other technology that allows threads to communicate with each other. In some embodiments, CTAs are executed using a processing core of a graphics processor that has multiple parallel processing engines, each capable of supporting multiple concurrent threads. The threads are advantageously executed in SIMD (single instruction, multiple data) groups, with one thread of the group being associated with each processing engine. A single instruction unit issues an instruction to an entire SIMD group in parallel, and each processing engine executes the instruction in the context of its thread of the current SIMD group; instructions for different SIMD

Ex. A8 – Invalidity of U.S. Patent No. 8,648,867 – Nickolls

U.S. Patent No. 8,648,867	Nickolls
	groups can be issued in any order. By executing each instruction in the appropriate context, each processing engine executes one thread in each of multiple concurrent SIMD groups. 5:33–64. GPU 122 has at least one processing core 126 for generating pixel data and a core interface 128 that controls operation of core 126. Core 126 advantageously includes multiple parallel processing engines that can be used to execute various shader programs, including vertex shader programs, geometry shader programs, and/or pixel shader programs, in the course of generating images from scene data. Core 126 can also be leveraged to perform general-purpose computations as described below. 6:47–55. CTAs are advantageously employed to perform computations that lend themselves to a data parallel decomposition, i.e., application of the same processing algorithm to different portions of an input data set in order to effect a transformation of the input data set to an output data set. The processing algorithm is specified in a "CTA program," and each thread in 5 a CTA executes the same CTA program on a different subset of an input data set. A CTA program can implement algorithms using a wide range of mathematical and logical operations, and the program can include conditional or branching execution paths and direct and/or indirect memory access. Threads in a CTA can share intermediate results with other threads in the same CTA using a shared memory that is accessible to all of the threads, an interconnection network, or other technologies for inter-thread communication, including other technologies known in the art. In some embodiments, the CTA program includes an instruction to compute an address in shared memory to which particular data is to be written, with the address being a function of thread ID. Each thread computes the function using its own thread ID and writes to the corresponding location. The function is advantageously defined such that different threads write to different locations; as long as the function is deterministic, the location written to by any thread is well-defined. The CTA program can also include an instruction to compute an address in shared memory from which data is to be read, with the address being a function of thread ID. By defining suitable functions, data can be written to a given location by one thread and read from that location

Ex. A8 – Invalidity of U.S. Patent No. 8,648,867 – Nickolls

U.S. Patent No. 8,648,867	Nickolls
	by a different thread in a predictable manner. Consequently, any desired pattern of data sharing among threads can be supported, and any thread in a CTA can share data with any other thread in the same CTA. 7:66–8:30. Each processing engine 302 is allocated space in a local register file 304 for storing its local input data, intermediate results, and the like. In one embodiment, local register file 304 is divided into P lanes, each having some number of entries (where each entry might be, e.g., a 32-bit word). One lane is allocated to each processing unit, and corresponding entries in different lanes can be populated with data for corresponding thread types to facilitate SIMD execution of multiple threads in parallel as described below. The number of entries in local register file 304 is advantageously large enough to support multiple concurrent threads per processing engine 302; and in some embodiments, the number of entries allocated to a thread is dynamically configurable. Each processing engine 302 also has access, via a crossbar switch 305, to a (shared) global register file 306 that is shared among all of the processing engines 302 in core 126. Global register file 306 may be as large as desired, and in some embodiments, any processing engine 302 can read to or write from any location in global register file 306. In addition to global register file 306, some embodiments also provide an on-chip shared memory 308, which may be implemented, e.g., as a conventional RAM. On-chip memory 308 is advantageously used to store data that is expected to be used in multiple threads, such as coefficients of attribute equations, which are usable in pixel shader programs. In some embodiments, processing engines 302 may also have access to additional off-chip shared memory (not shown), which might be located, e.g., within graphics memory 124 and/or system memory 104 of FIG. 1. 10:57–11:18. Instruction unit 312 advantageously manages instruction fetch and issue for each SIMD group so as to ensure that threads in a group that have diverged eventually resynchronize. In one embodiment, instruction unit 312 includes program counter (PC) logic 314, a program counter register array 316, a multiplexer 318, arbitration logic 320, fetch logic 322, and issue logic 324. Program counter register array 316 stores G program counter values (one per SIMD group), which are updated independently of each other by PC logic 314. PC logic 314 updates the PC values

Ex. A8 – Invalidity of U.S. Patent No. 8,648,867 – Nickolls

U.S. Patent No. 8,648,867	Nickolls
	based on information received from processing engines 302 and/or fetch logic 322. PC logic 314 is advantageously configured to track divergence among threads in a SIMD group and to select instructions in a way that ultimately results in the threads re-synchronizing. 12:19–33. For optimal performance, all threads within a SIMD group are advantageously launched on the same clock cycle so that they begin in a synchronized state. In one embodiment, core interface 128 advantageously loads SIMD groups into core 15 126, then instructs core 126 to launch the group. "Loading" a thread, as used herein, includes supplying instruction unit 312 and processing engines 302 with various input parameters required to execute the program. In some instances, the input parameters may include the input data to be processed by the 20 program; in other instances, the input data is stored in global register file 306 or other shared memory (e.g., graphics memory 124 or system memory 104 of FIG. 1) prior to loading of any threads, and the input parameters may include a reference to a location where the input data is stored. For example, in the case of CTA processing, the input data set may be loaded into graphics memory 124 or system memory 104 before core interface is instructed to begin CTA processing. Core interface 128 loads the starting PC value for the CTA program into a slot in PC array 316 that is not currently 30 in use; this slot corresponds to the group index GID assigned to the new SIMD group that will process P of the CTA threads. Core interface 128 allocates sufficient space in the local register file for each processing engine 302 to execute one CTA thread, then loads input parameters into shared memory (e.g., 35 global register file 306). Core interface 128 loads a unique thread ID into a thread ID register for each thread or into a predetermined register in the allocated portion of local register file 304 for each processing engine 302. In one embodiment, thread IDs for P threads are loaded in parallel, as 40 described below. Once the input parameters and thread IDs for all threads in the SIMD group have been loaded, core interface 128 launches the group by signaling instruction unit 312 to begin fetching and issuing instructions corresponding to the group index GID of the new group. It will be appreciated that the processing core described herein is illustrative and that variations and modifications are possible. Any number of processing units may be included. In some embodiments, each processing unit has its own local register file, and the allocation of local register file entries per thread can be fixed or configurable as desired.

Ex. A8 – Invalidity of U.S. Patent No. 8,648,867 – Nickolls

U.S. Patent No. 8,648,867	Nickolls
	In some embodiments, core 126 is operated at a higher clock rate than core interface 128, allowing the core to process more data in a given amount of time. For instance, core 126 can be operated at a clock rate that is twice the clock rate 55 of core interface 128. If core 126 includes P processing engines 302 producing data at twice the core interface clock rate, then core 126 can produce 2*P data values per core interface clock cycle. Provided there is sufficient space in local register file 304, from the perspective of core interface 60 128, the situation is effectively identical to a core with 2*P processing units. Thus, P-way SIMD parallelism could be produced either by including P processing units in core 126 and operating core 126 at the same clock rate as core interface 128 or by including P/2 processing units in core 126 and 65 operating core 126 at twice the clock rate of core interface 128. Other timing variations are also possible. 13:12–67. FIG. 4 is a block diagram of core interface 128 according to an embodiment of the present invention. Core interface 128 includes an input unit 402, a state module 404, state registers 406, a load module 408, and a launch module 410. As described above, core interface 128 (FIG. 1) controls operation of core 126. In particular, core interface 128 loads and launches threads of a CTA in SIMD groups until all threads have been launched. In one embodiment, core interface 128 generates thread IDs for each thread in a SIMD group and writes the thread ID into a suitable location in local register file 304, then launches the SIMD group. These aspects of core interface 128 will now be described. Input unit 402 receives all incoming signals, including state information, data and commands, and directs the signals to state module 404 or load module 406. In one embodiment, input unit 402 receives a data word (e.g., 32 bits) along with control signals indicating whether the data word corresponds to state information or data to be processed. Based on the control signals, input unit 402 determines where to direct the data word, and all or part of the control signal may be forwarded together with the data word. For instance, if the data word corresponds to state information, the control signals may be used by state module 404 to determine where to store the data word, as described below.

Ex. A8 – Invalidity of U.S. Patent No. 8,648,867 – Nickolls

U.S. Patent No. 8,648,867	Nickolls
	State module 404 receives state information from input unit 402 and loads the state information into state registers 406. “State information,” as used herein, includes any information (other than input data) relevant to defining a CTA. For example, in one embodiment, state information includes the size of the input data set, the amount of local register file space required for each thread, and a starting program counter (e.g., memory address) for a program to be executed by each thread. State information advantageously also includes size information for the CTA; for example, referring to FIG. 2A, array dimensions D0, D1 and D2 may be provided. In some embodiments, the total number (7) of threads is also provided; in other embodiments, T can be computed from the array dimensions (e.g., $T=D0*D1*D2$ for the embodiment of FIG. 2A). When T is provided, T is advantageously less than or equal to $D0*D1*D2$, in order to ensure that each thread is assigned a unique thread ID. 14:17–57. Referring again to FIG. 4, load module 408 receives input parameters for a CTA and loads the input parameters into (shared) global register file 306 (FIG. 3) of core 126. These parameters may include, e.g., an (x,y) position of the CTA within a grid of CTAs or other CTA identifier and/or other information specific to the CTA. In some embodiments, the input parameters may also include some or all of an input data set to be processed by the CTA. In one embodiment, the first received input parameter is accompanied by a “begin CTA” control signal indicating that what follows are input parameters for a CTA that is to be executed using the currently defined state. The control signal may also indicate the size of the input parameter set, where in global register file 306 the parameters are to be stored, and other information, e.g., a control signal identifying the last of the input parameters so that core interface 128 immediately recognizes when the CTA is ready to be launched. Alternatively, state information defining the size of the input parameter set may be stored in state registers 406, and load module 408 may use this information to determine when loading of input parameters is complete. For instance, if state registers 406 store information about the number of input parameters, load module 408 might count received input parameters until the expected number had been received. Once all the input parameters have been loaded into global 20 register file 306, load module 408 issues a "GO" control signal to launch module 410. Launch module 410 is configured to respond

Ex. A8 – Invalidity of U.S. Patent No. 8,648,867 – Nickolls

U.S. Patent No. 8,648,867	Nickolls
	to the GO control signal by launching all of the threads for one CTA based on the current definition of CTA size stored in state registers 406. Launch module 410 supplies each thread with a unique thread ID, which is written to one of the local registers 304 (FIG. 3) allocated to that thread or to a per-thread register dedicated to storing a thread ID. Once all the threads in a SIMD group have been supplied with thread IDs, launch module 410 advantageously launches that SIMD group before beginning to supply thread IDs for the next SIMD group. Launch module 410 is advantageously designed to generate and assign thread IDs rapidly so as to minimize the delay between launching successive SIMD groups. In one embodiment, the delay can be as little as one clock cycle between successive launches. 16:63–17:36. To launch the first group of threads of the CTA, the P initial thread IDs are supplied to selection unit 808. In response to a control signal from counter logic 804 indicating that the first thread group is being launched, selection unit 808 selects the initial thread IDs for writing to registers in local register file 304. For all subsequent thread groups in the CTA, selection unit 808 selects the P new thread IDs computed by adders 806. (It should be noted that the thread IDs for a SIMD group need not be correlated with the group index GID assigned to the group for purposes of controlling execution.) 17:63–18:3. During the load phase, core interface 128 receives input data (step 1012) and loads the input data into (shared) global register file 306 of core 126 (step 1014). At step 1016, core interface 128 determines whether the last input data has been received. If not, core interface 128 returns to step 1012 to receive more input data. 19:44–50. The CTA program may also include instructions to read from and/or write to the shared global register file 306, on-chip shared memory 308, and/or other memory such as graphics memory 124 or system memory 104 of FIG. 1. For instance, the input data set to be processed by the CTA may be stored in graphics memory 124 or system memory 104. Intermediate results may be written to global register file 306 and/or other memory such as graphics memory 124 or system memory 104

Ex. A8 – Invalidity of U.S. Patent No. 8,648,867 – Nickolls

U.S. Patent No. 8,648,867	Nickolls
	of FIG. 1, where they can be shared with other threads. Final results (output data) may be written to graphics memory 124 or system memory 104. Instructions to access shared memory or shared global register file space advantageously identify locations to be read and/or written as a function of the thread ID. Where such an instruction is encountered, the thread reads its thread ID and computes the target address from the thread ID using the function specified in the CTA program. For instance, referring to FIG. 2C, a thread with ID (il, i0) might be instructed to read the input data corresponding to the pixel to its right, with the address being specified as InputData[il, i0+1]. In some embodiments, a conditional branch or conditional instruction in the CTA program might be provided to handle cases where the computed address is not within the valid range (e.g., for a thread processing a pixel at the right edge of the screen, an address such as InputData [il, i0+1] would not be a valid pixel address). 20:35–60. In some embodiments of the present invention, a barrier synchronization technique is advantageously used to support fast synchronization of any number of CTA threads. More specifically, barrier instructions are inserted into the CTA program at points (referred to herein as "barrier points") where synchronization is desired. A thread executes a barrier instruction to indicate that it has arrived at a barrier point and waits at that point until all other participating threads have also arrived at that point, thus synchronizing the participating threads before resuming execution. Arrival of each thread (or group of threads) at a barrier point is detected, and this information is used to synchronize two or more threads (or groups of threads). In one embodiment, execution of barrier instructions, i.e., arrival of threads (or SIMD groups) at barrier points, is detected by issue logic 324 of instruction unit 312 of FIG. 3, which can suspend the issue of instructions to any threads that are waiting at a barrier point while continuing to issue instructions to threads that are not at a barrier point. Eventually, all relevant threads reach the barrier point, and execution of the waiting threads resumes. FIG. 11A is a block diagram of barrier synchronization logic 1100 according to an embodiment of the present invention. In some embodiments, barrier synchronization logic 1100 is implemented in issue logic 324 of instruction unit 312 and synchronizes SIMD groups rather than individual threads.

Ex. A8 – Invalidity of U.S. Patent No. 8,648,867 – Nickolls

U.S. Patent No. 8,648,867	Nickolls
	As shown in FIG. 11A, instruction unit 312 also includes selection logic 1110 that selects a next instruction to issue. Selection logic 1110 may be of generally conventional design, and a detailed description is omitted as not being critical to understanding the present invention. Barrier detection circuit 1112 receives each selected instruction (INST), along with the group identifier (GID) of the SIMD group for which the instruction is being issued. If the selected instruction is a barrier instruction, barrier detection circuit 1112 directs the instruction to barrier synchronization logic 1100; otherwise, barrier detection circuit 1112 forwards the instruction to the next issue stage for eventual delivery to processing engines 302 of FIG. 3. Barrier synchronization logic 1100 includes a counter 1104, a target register 1105, a comparison circuit 1106, and 65 wait/go registers 1108. Counter 1104 tracks the number of threads that have arrived at a particular barrier point. Target register 1105 stores a target value, which corresponds to the number of SIMD groups that are expected to synchronize at the barrier point. In one embodiment, the target value is supplied as an immediate operand with the barrier instruction and is loaded into target register 1105 by barrier detection circuit 1112 when the first barrier instruction is received. Once loaded, the target value advantageously remains stored in target register 1105 until target register 1105 is reset. 21:25–22:8. <i>Additionally and alternatively, as a non-limiting example, Nickolls discloses that “instruction unit 312” is an accelerator controller, together with core interface 128.</i> Instruction unit 312 is configured such that, for any given processing cycle, the same instruction (INSTR) is issued to all P processing engines 302. Thus, at the level of a single clock cycle, core 126 implements P-way SIMD execution. Each processing engine 302 is also multithreaded, supporting up to G concurrent threads, and instructions for different ones of the G threads can be issued in any order relative to each other. Accordingly, core 126 in this embodiment can have up to P*G threads in flight concurrently. For instance, if P=16 and G=24, then core 126 can support up to 384 concurrent threads. In some embodiments, P*G determines an upper limit on the number of threads that can be included in a CTA; it is to be understood that some CTAs may include fewer than this number of threads.

Ex. A8 – Invalidity of U.S. Patent No. 8,648,867 – Nickolls

U.S. Patent No. 8,648,867	Nickolls
	11:26–40. Instruction unit 312 advantageously manages instruction fetch and issue for each SIMD group so as to ensure that threads in a group that have diverged eventually resynchronize. In one embodiment, instruction unit 312 includes program counter (PC) logic 314, a program counter register array 316, a multiplexer 318, arbitration logic 320, fetch logic 322, and issue logic 324. Program counter register array 316 stores G program counter values (one per SIMD group), which are updated independently of each other by PC logic 314. PC logic 314 updates the PC values based on information received from processing engines 302 and/or fetch logic 322. PC logic 314 is advantageously configured to track divergence among threads in a SIMD group and to select instructions in a way that ultimately results in the threads re-synchronizing. 12:19–33. For optimal performance, all threads within a SIMD group are advantageously launched on the same clock cycle so that they begin in a synchronized state. In one embodiment, core interface 128 advantageously loads SIMD groups into core 126, then instructs core 126 to launch the group. "Loading" a thread, as used herein, includes supplying instruction unit 312 and processing engines 302 with various input parameters required to execute the program. In some instances, the input parameters may include the input data to be processed by the 20 program; in other instances, the input data is stored in global register file 306 or other shared memory (e.g., graphics memory 124 or system memory 104 of FIG. 1) prior to loading of any threads, and the input parameters may include a reference to a location where the input data is stored. For 25 example, in the case of CTA processing, the input data set may be loaded into graphics memory 124 or system memory 104 before core interface is instructed to begin CTA processing. Core interface 128 loads the starting PC value for the CTA program into a slot in PC array 316 that is not currently in use; this slot corresponds to the group index GID assigned to the new SIMD group that will process P of the CTA threads. Core interface 128 allocates sufficient space in the local register file for each processing engine 302 to execute one CTA thread, then loads input parameters into shared memory (e.g., 35 global register file 306). Core interface 128 loads a unique thread ID into a thread ID register for each thread or into a predetermined register in the allocated portion of local register file 304 for each processing engine 302. In one embodiment, thread IDs for P threads are loaded in parallel, as 40 described below. Once the input parameters and thread IDs for all threads

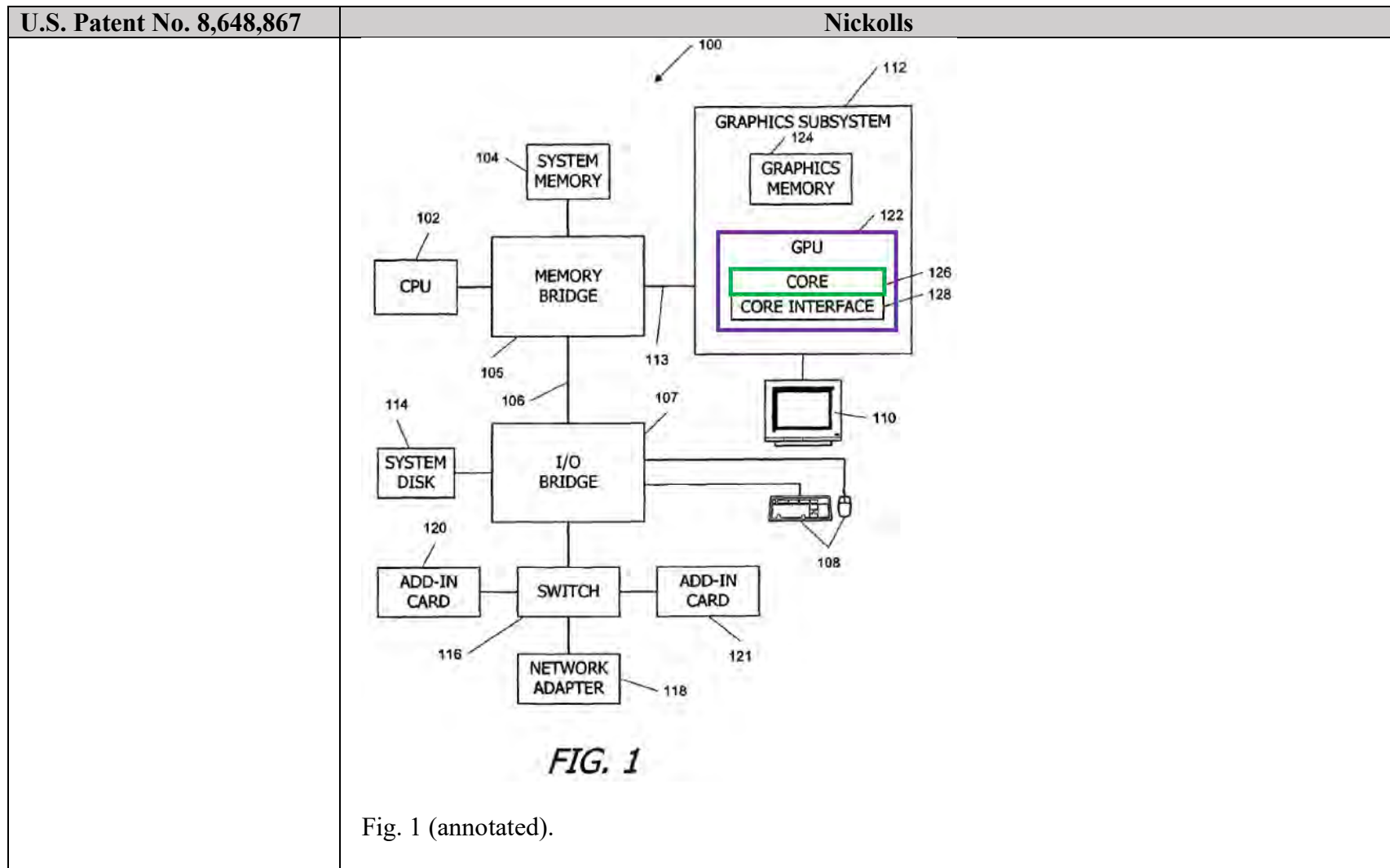
Ex. A8 – Invalidity of U.S. Patent No. 8,648,867 – Nickolls

U.S. Patent No. 8,648,867	Nickolls
	in the SIMD group have been loaded, core interface 128 launches the group by signaling instruction unit 312 to begin fetching and issuing instructions corresponding to the group index GID of the new group. 13:12–45. In some embodiments of the present invention, a barrier synchronization technique is advantageously used to support fast synchronization of any number of CTA threads. More specifically, barrier instructions are inserted into the CTA program at points (referred to herein as "barrier points") where synchronization is desired. A thread executes a barrier instruction to indicate that it has arrived at a barrier point and waits at that point until all other participating threads have also arrived at that point, thus synchronizing the participating threads before resuming execution. Arrival of each thread (or group of threads) at a barrier point is detected, and this information is used to synchronize two or more threads (or groups of threads). In one embodiment, execution of barrier instructions, i.e., arrival of threads (or SIMD groups) at barrier points, is detected by issue logic 324 of instruction unit 312 of FIG. 3, which can suspend the issue of instructions to any threads that are waiting at a barrier point while continuing to issue instructions to threads that are not at a barrier point. Eventually, all relevant threads reach the barrier point, and execution of the waiting threads resumes. FIG. 11A is a block diagram of barrier synchronization logic 1100 according to an embodiment of the present invention. In some embodiments, barrier synchronization logic 1100 is implemented in issue logic 324 of instruction unit 312 and synchronizes SIMD groups rather than individual threads. As shown in FIG. 11A, instruction unit 312 also includes selection logic 1110 that selects a next instruction to issue. Selection logic 1110 may be of generally conventional design, and a detailed description is omitted as not being critical to understanding the present invention. Barrier detection circuit 1112 receives each selected instruction (INST), along with the group identifier (GID) of the SIMD group for which the instruction is being issued. If the selected instruction is a barrier instruction, barrier detection circuit 1112 directs the instruction to barrier synchronization logic 1100; otherwise, barrier detection circuit 1112 forwards the instruction to the next issue stage for eventual delivery to processing engines 302 of FIG. 3.

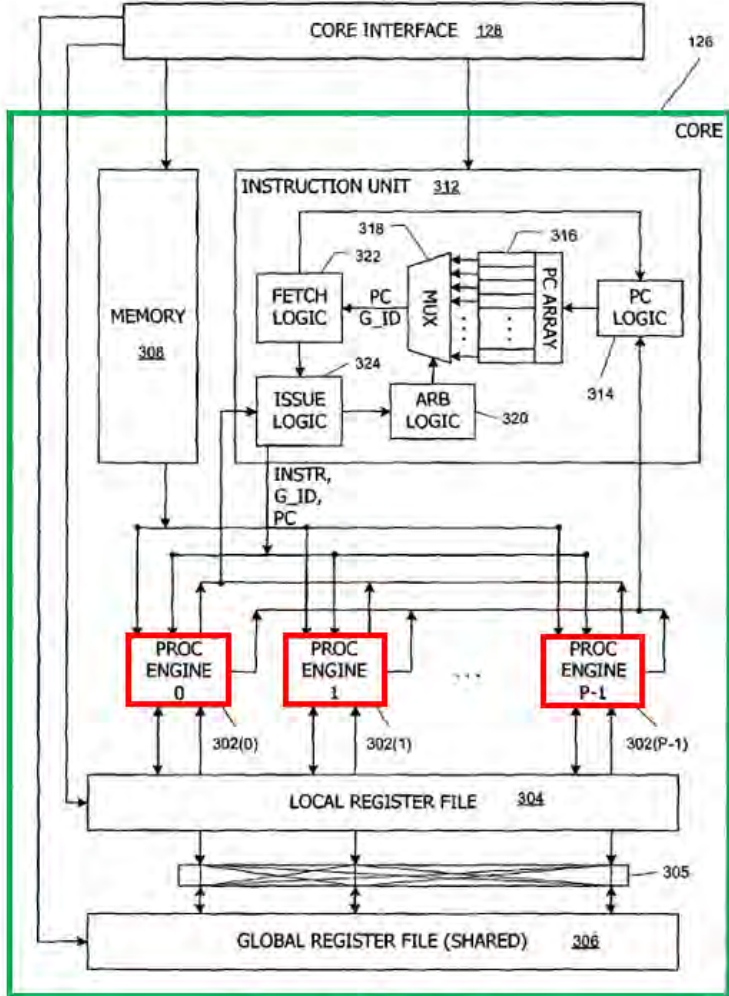
Ex. A8 – Invalidity of U.S. Patent No. 8,648,867 – Nickolls

U.S. Patent No. 8,648,867	Nickolls
	21:25–63. To the extent Plaintiff contends that the preamble is not expressly or inherently disclosed, or is not obvious in view of Nicholls and the knowledge of a POSITA, it would have been obvious to combine the teachings of Nicholls 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 ’867 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 Nicholls. Specific examples of such references are cited and explained in these invalidity contentions, including the cover pleading and charts A1–A13, and Appendix A.
16[c]: performing, by at least one graphics processing unit during the first computational cycle, at least one calculation on the first portion of the input data as to generate first output data;	Nicholls discloses “performing, by at least one graphics processing unit during the first computational cycle, at least one calculation on the first portion of the input data as to generate first output data.” <i>See e.g.:</i> <i>As a non-limiting example, Nickolls discloses at least one graphics processing unit (e.g., processing core 126 and/or processing engines within core 126) that performs at least one calculation (e.g., one or more steps of the “numerical simulation” described in 16[pre]) on a “first portion of the input data” (see 16[a]) during a first computational cycle to generate first output data (e.g., intermediate output data and/or final output data).</i>

Ex. A8 – Invalidity of U.S. Patent No. 8,648,867 – Nickolls



Ex. A8 – Invalidity of U.S. Patent No. 8,648,867 – Nickolls

U.S. Patent No. 8,648,867	Nickolls
	 FIG. 3 Fig. 3 (annotated).

Ex. A8 – Invalidity of U.S. Patent No. 8,648,867 – Nickolls

U.S. Patent No. 8,648,867	Nickolls
	In one embodiment, the input data set is loaded into a shared memory accessible to all of the plurality of threads (such as a global register file of the processor or other on-chip shared memory), and during the act of executing the plurality of threads, the thread identifier of a thread is used to determine a location in the shared memory of input data to be read. In some embodiments, the thread identifier may be used to determine a location in the shared memory or other memory to which output data, including intermediate and/or final result data, is to be written. 3:14–29. Graphics processing subsystem 112 includes a graphics processing unit (GPU) 122 and a graphics memory 124, which may be implemented, e.g., using one or more integrated circuit devices such as programmable processors, application specific integrated circuits (ASICs), and memory devices. GPU 122 may be configured to perform various tasks related to generating pixel data from graphics data supplied by CPU 102 and/or system memory 104 via memory bridge 105 and bus 113, interacting with graphics memory 124 to store and update pixel data, and the like. For example, GPU 122 may generate pixel data from 2-D or 3-D scene data provided by various programs executing on CPU 102. GPU 122 has at least one processing core 126 for generating pixel data and a core interface 128 that controls operation of core 126. Core 126 advantageously includes multiple parallel processing engines that can be used to execute various shader programs, including vertex shader programs, geometry shader programs, and/or pixel shader programs, in the course of generating images from scene data. Core 126 can also be leveraged to perform general-purpose computations as described below. 6:35–55. In accordance with an embodiment of the present invention, core 126 of GPU 122 is a multithreaded or parallel processing core that can be leveraged for general-purpose computations by executing cooperative thread arrays (CTAs). As used herein, a "CTA" is a group of multiple threads that concurrently execute the same program on an input data set to produce an output data set. Each thread in the CTA is assigned a unique thread identifier ("thread ID") that is accessible to the thread during its execution. The thread ID controls various aspects of the thread's processing

Ex. A8 – Invalidity of U.S. Patent No. 8,648,867 – Nickolls

U.S. Patent No. 8,648,867	Nickolls
	behavior. For instance, a thread ID may be used to determine which portion of the input data set a thread is to process, to identify one or more other threads with which a given thread is to share an intermediate result, and/or to determine which portion of an output data set a thread is to produce or write. CTAs are advantageously employed to perform computations that lend themselves to a data parallel decomposition, i.e., application of the same processing algorithm to different portions of an input data set in order to effect a transformation of the input data set to an output data set. The processing algorithm is specified in a "CTA program," and each thread in a CTA executes the same CTA program on a different subset of an input data set. A CTA program can implement algorithms using a wide range of mathematical and logical operations, and the program can include conditional or branching execution paths and direct and/or indirect memory access. 7:51–8:3. In one embodiment, GPU 122 implements a rendering pipeline that includes vertex and geometry processing, primitive setup and rasterization, attribute assembly, and pixel processing. The rendering pipeline supports various shader programs including vertex shaders, geometry shaders, and pixel shaders, examples of which are known in the art. Shader programs of arbitrary complexity are advantageously supported using a "unified shader" architecture in which one or more processing cores 126 support concurrent execution of a (preferably large) number of instances of vertex, geometry, and/or pixel shader programs in various combinations. In accordance with an embodiment of the present invention, processing core 126 is leveraged to execute CTAs for general-purpose computations. FIG. 3 is a block diagram of a processing core 126 according to an embodiment of the present invention. Core 126 is advantageously configured to execute a large number of threads in parallel. During a rendering operation, a thread might be an instance of a vertex shader program executing on the attributes of a single vertex or an instance of a pixel shader program executing on a given primitive and pixel. During general-purpose computing, a thread can be an instance of a CTA program executing on a portion of an input data set. In core 126, single-instruction, multiple-data (SIMD) instruction issue techniques are used to support parallel execution of a large number of threads by multiple processing engines without requiring multiple instruction units.

Ex. A8 – Invalidity of U.S. Patent No. 8,648,867 – Nickolls

U.S. Patent No. 8,648,867	Nickolls
	In one embodiment, core 126 includes an array of P (e.g., 8, 16, or any other number) parallel processing engines 302 configured to receive and execute SIMD instructions from a single instruction unit 312. Each parallel processing engine 302 advantageously includes an identical set of functional units such as arithmetic logic units, load/store units, and the like (not explicitly shown). The functional units may be pipelined, allowing a new instruction to be issued before a previous instruction has finished, as is known in the art. Any combination of functional units may be provided. In one embodiment, the functional units support a variety of operations including integer and floating point arithmetic (e.g., addition and multiplication), comparison operations, Boolean operations (AND, OR, XOR), bit-shifting, and computation of various algebraic functions (e.g., planar interpolation, trigonometric, exponential, and logarithmic functions, etc.); and the same functional-unit hardware can be leveraged to perform different operations. A particular implementation of processing engines 302 is not critical to the present invention, and a detailed description has been omitted. Each processing engine 302 is allocated space in a local register file 304 for storing its local input data, intermediate results, and the like. In one embodiment, local register file 304 is divided into P lanes, each having some number of entries (where each entry might be, e.g., a 32-bit word). One lane is allocated to each processing unit, and corresponding entries in different lanes can be populated with data for corresponding thread types to facilitate SIMD execution of multiple threads in parallel as described below. The number of entries in local register file 304 is advantageously large enough to support multiple concurrent threads per processing engine 302; and in some embodiments, the number of entries allocated to a thread is dynamically configurable. 10:9–11:2. Once all of the input data has been received, core interface 128 enters the launch phase. In the launch phase, core interface 128 selects a group index GID for a first SIMD group to be launched (step 1018); any group index GID that is not already in use in core 126 may be selected. At step 1020, core interface 128 loads the initial set of P thread IDs into the local register file 304 of core 126 in locations corresponding to the selected group index GID or into per-thread registers in core 126 dedicated to storing thread IDs. Core interface 126 then instructs core 126 to launch the P threads as a SIMD group (step 1022).

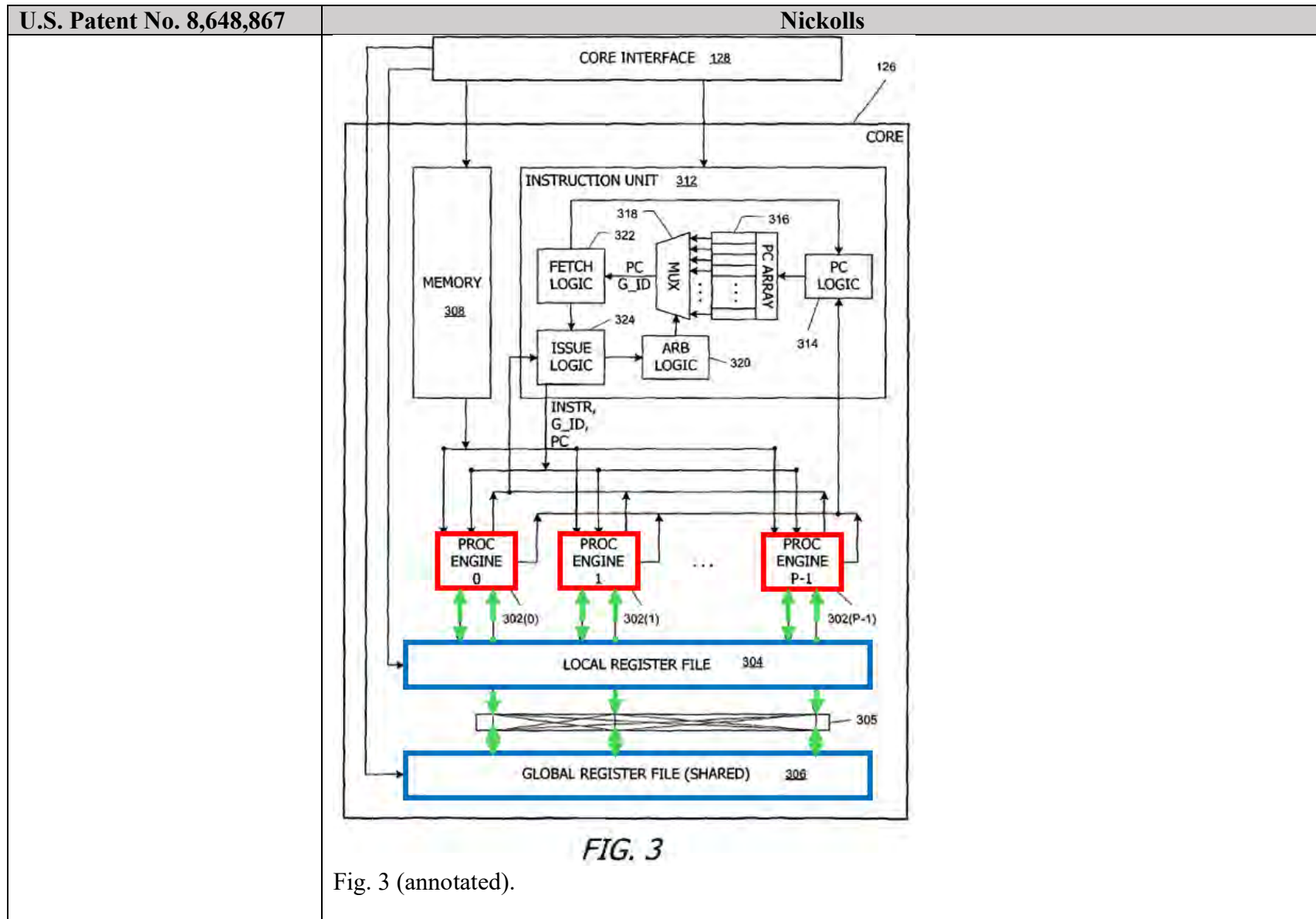
Ex. A8 – Invalidity of U.S. Patent No. 8,648,867 – Nickolls

U.S. Patent No. 8,648,867	Nickolls
	19:51–61. As described above, core 126 advantageously implements a multithreaded architecture, with SIMD groups of threads being executed in parallel and multiple SIMD groups executing concurrently. Maximum efficiency is obtained when the threads in a SIMD group do not diverge; however, in practice, threads are allowed to diverge without restriction. For instance, a program being executed by all threads in a SIMD group may include a conditional branch instruction, and the branch may be taken by some threads and not taken by others; as described above, core 126 can be configured to handle such instances. Threads may also diverge in other ways, e.g., due to conditional execution of various instructions that might be executed by some threads but not others within a SIMD group. Thus, CTA programs of arbitrary complexity can be implemented. The CTA program may also include instructions to read from and/or write to the shared global register file 306, onchip shared memory 308, and/or other memory such as graphics memory 124 or system memory 104 of FIG. 1. For instance, the input data set to be processed by the CTA may be stored in graphics memory 124 or system memory 104. Intermediate results may be written to global register file 306 and/or other memory such as graphics memory 124 or system memory 104 of FIG. 1, where they can be shared with other threads. Final results (output data) may be written to graphics memory 124 or system memory 104. 20:20–45. In some embodiments using a GPU, CTA threads and rendering threads can coexist in the same processor, e.g., in different cores or in the same core. Further, in systems with multiple GPUs, one GPU may be used for rendering images while another is used for general-purpose computations including CTAs. Alternatively, each GPU may be assigned a different portion of a general-purpose computation; allowing multiple GPUs to execute CTAs in parallel further increases the number of CTAs that can be executed in parallel. 30:52–60. To the extent Plaintiff contends that the preamble is not expressly or inherently disclosed, or is not obvious in view of Nicholls and the knowledge of a POSITA, it would have been obvious to combine the teachings of Nicholls with any of the prior art references in NVIDIA’s Invalidity

Ex. A8 – Invalidity of U.S. Patent No. 8,648,867 – Nickolls

U.S. Patent No. 8,648,867	Nickolls
	Contentions and any supplements thereto and the relevant section of charts for other prior art for the '867 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 Nicholls. Specific examples of such references are cited and explained in these invalidity contentions, including the cover pleading and charts A1–A13, and Appendix A.
16[d]: storing, by the accelerator controller, the first output data into a second partition, referenced by a second pointer, of the accelerator memory; and	Nicholls discloses “storing, by the accelerator controller, the first output data into a second partition, referenced by a second pointer, of the accelerator memory.” <i>See e.g.:</i> <i>See 16[b].</i> <i>As a non-limiting example, Nickolls discloses storage at the direction and/or under the control of the accelerator controller (e.g., core interface 128) of first output data into a “second partition” of the accelerator memory (e.g., by allocating and storing output data in accelerator memory). Nickolls further discloses referencing partitions by pointers (e.g., using indirect memory access, including calculating an address (which can be stored in a pointer) at which data can be accessed).</i>

Ex. A8 – Invalidity of U.S. Patent No. 8,648,867 – Nickolls



Ex. A8 – Invalidity of U.S. Patent No. 8,648,867 – Nickolls

U.S. Patent No. 8,648,867	Nickolls
	In one embodiment, the processing core includes a local register file having a number of entries allocated to each of the plurality of threads, and the thread identifier for each thread is stored in one of the local register file entries allocated to that thread. 3:54–62. Thread IDs are advantageously assigned as the threads of a CTA are launched, and the interface logic for the processing core is advantageously configured to automatically assign thread IDs to threads as the threads are being launched. Automatic assignment of thread IDs eliminates the need for an application programmer to explicitly configure each thread. Instead, the programmer can simply define the number of threads in the CTA and the operations each thread is to do; the particular manner in which the CTA is executed is advantageously made transparent to the programmer. 5:65–6:7. Threads in a CTA can share intermediate results with other threads in the same CTA using a shared memory that is accessible to all of the threads, an interconnection network, or other technologies for inter-thread communication, including other technologies known in the art. In some embodiments, the CTA program includes an instruction to compute an address in shared memory to which particular data is to be written, with the address being a function of thread ID. Each thread computes the function using its own thread ID and writes to the corresponding location. The function is advantageously defined such that different threads write to different locations; as long as the function is deterministic, the location written to by any thread is well-defined. The CTA program can also include an instruction to compute an address in shared memory from which data is to be read, with the address being a function of thread ID. By defining suitable functions, data can be written to a given location by one thread and read from that location by a different thread in a predictable manner. Consequently, any desired pattern of data sharing among threads can be supported, and any thread in a CTA can share data with any other thread in the same CTA. 8:10–30. In one embodiment, core 126 includes an array of P (e.g., 8, 16, or any other number) parallel processing engines 302 configured to receive and execute SIMD instructions from a single

Ex. A8 – Invalidity of U.S. Patent No. 8,648,867 – Nickolls

U.S. Patent No. 8,648,867	Nickolls
	instruction unit 312. Each parallel processing engine 302 advantageously includes an identical set of functional units such as arithmetic logic units, load/store units, and the like (not explicitly shown). The functional units may be pipelined, allowing a new instruction to be issued before a previous instruction has finished, as is known in the art. Any combination of functional units may be provided. In one embodiment, the functional units support a variety of operations including integer and floating point arithmetic (e.g., addition and multiplication), comparison operations, Boolean operations (AND, OR, XOR), bit-shifting, and computation of various algebraic functions (e.g., planar interpolation, trigonometric, exponential, and logarithmic functions, etc.); and the same functional-unit hardware can be leveraged to perform different operations. A particular implementation of processing engines 302 is not critical to the present invention, and a detailed description has been omitted. Each processing engine 302 is allocated space in a local register file 304 for storing its local input data, intermediate results, and the like. In one embodiment, local register file 304 is divided into P lanes, each having some number of entries (where each entry might be, e.g., a 32-bit word). One lane is allocated to each processing unit, and corresponding entries in different lanes can be populated with data for corresponding thread types to facilitate SIMD execution of multiple threads in parallel as described below. The number of entries in local register file 304 is advantageously large enough to support multiple concurrent threads per processing engine 302; and in some embodiments, the number of entries allocated to a thread is dynamically configurable. Each processing engine 302 also has access, via a crossbar switch 305, to a (shared) global register file 306 that is shared among all of the processing engines 302 in core 126. Global register file 306 may be as large as desired, and in some embodiments, any processing engine 302 can read to or write from any location in global register file 306. In addition to global register file 306, some embodiments also provide an on-chip shared memory 308, which may be implemented, e.g., as a conventional RAM. On-chip memory 308 is advantageously used to store data that is expected to be used in multiple threads, such as coefficients of attribute equations, which are usable in pixel shader programs. In some embodiments, processing engines 302 may also have access to additional off-chip shared memory (not shown), which might be located, e.g., within graphics memory 124 and/or system memory 104 of FIG. 1.

Ex. A8 – Invalidity of U.S. Patent No. 8,648,867 – Nickolls

U.S. Patent No. 8,648,867	Nickolls
	In one embodiment, each processing engine 302 is multithreaded and can execute up to some number G (e.g., 24) of threads concurrently. Processing engines 302 are advantageously designed to switch rapidly from any one active thread to any other active thread. For instance, processing engine 302 can maintain current state information associated with each thread in a different portion of its allocated lane in local register file 306, facilitating fast switching. Instruction unit 312 is configured such that, for any given processing cycle, the same instruction (INSTR) is issued to all P processing engines 302. Thus, at the level of a single clock cycle, core 126 implements P-way SIMD execution. Each processing engine 302 is also multithreaded, supporting up to G concurrent threads, and instructions for different ones of the G threads can be issued in any order relative to each other. Accordingly, core 126 in this embodiment can have up to P*G threads in flight concurrently. For instance, if P=16 and G=24, then core 126 can support up to 384 concurrent threads. In some embodiments, P*G determines an upper limit on the number of threads that can be included in a CTA; it is to be understood that some CTAs may include fewer than this number of threads. 10:37–11:40. Arbitration logic 320 and multiplexer 318 determine the order in which instructions are fetched. More specifically, on each clock cycle, arbitration logic 320 selects one of the G possible group indices GID as the SIMD group for which a next instruction should be fetched and supplies a corresponding control signal to multiplexer 318, which selects the corresponding PC. Arbitration logic 320 may include conventional logic for prioritizing and selecting among concurrent threads (e.g., using round-robin, least-recently serviced, or the like), and selection may be based in part on feedback information from fetch logic 322 or issue logic 324 as to how many instructions have been fetched but not yet issued for each SIMD group. 12:43–55. For optimal performance, all threads within a SIMD group are advantageously launched on the same clock cycle so that they begin in a synchronized state. In one embodiment, core interface 128 advantageously loads SIMD groups into core 126, then instructs core 126 to launch the group. "Loading" a thread, as used herein, includes supplying instruction unit 312 and processing engines 302 with various input parameters required to execute the program. In some instances, the input

Ex. A8 – Invalidity of U.S. Patent No. 8,648,867 – Nickolls

U.S. Patent No. 8,648,867	Nickolls
	parameters may include the input data to be processed by the program; in other instances, the input data is stored in global register file 306 or other shared memory (e.g., graphics memory 124 or system memory 104 of FIG. 1) prior to loading of any threads, and the input parameters may include a reference to a location where the input data is stored. For example, in the case of CTA processing, the input data set may be loaded into graphics memory 124 or system memory 104 before core interface is instructed to begin CTA processing. Core interface 128 loads the starting PC value for the CTA program into a slot in PC array 316 that is not currently 30 in use; this slot corresponds to the group index GID assigned to the new SIMD group that will process P of the CTA threads. Core interface 128 allocates sufficient space in the local register file for each processing engine 302 to execute one CTA thread, then loads input parameters into shared memory (e.g., global register file 306). Core interface 128 loads a unique thread ID into a thread ID register for each thread or into a predetermined register in the allocated portion of local register file 304 for each processing engine 302. In one embodiment, thread IDs for P threads are loaded in parallel, as described below. Once the input parameters and thread IDs for all threads in the SIMD group have been loaded, core interface 128 launches the group by signaling instruction unit 312 to begin fetching and issuing instructions corresponding to the group index GID of the new group. 13:12–45. State module 404 receives state information from input unit 402 and loads the state information into state registers 406. “State information,” as used herein, includes any information (other than input data) relevant to defining a CTA. For example, in one embodiment, state information includes the size of the input data set, the amount of local register file space required for each thread, and a starting program counter (e.g., memory address) for a program to be executed by each thread. State information advantageously also includes size information for the CTA; for example, referring to FIG. 2A, array dimensions D0, D1 and D2 may be provided. In some embodiments, the total number (7) of threads is also provided; in other embodiments, T can be computed from the array dimensions (e.g., $T=D0*D1*D2$ for the embodiment of FIG. 2A). When T is provided, T is advantageously less than or equal to $D0*D1*D2$, in order to ensure that each thread is assigned a unique thread ID. 14:41–57.

Ex. A8 – Invalidity of U.S. Patent No. 8,648,867 – Nickolls

U.S. Patent No. 8,648,867	Nickolls
	The CTA program may also include instructions to read from and/or write to the shared global register file 306, on-chip shared memory 308, and/or other memory such as graphics memory 124 or system memory 104 of FIG. 1. For instance, the input data set to be processed by the CTA may be stored in graphics memory 124 or system memory 104. Intermediate results may be written to global register file 306 and/or other memory such as graphics memory 124 or system memory 104 of FIG. 1, where they can be shared with other threads. Final results (output data) may be written to graphics memory 124 or system memory 104. Instructions to access shared memory or shared global register file space advantageously identify locations to be read and/or written as a function of the thread ID. Where such an instruction is encountered, the thread reads its thread ID and computes the target address from the thread ID using the function specified in the CTA program. For instance, referring to FIG. 2C, a thread with ID (il, i0) might be instructed to read the input data corresponding to the pixel to its right, with the address being specified as InputData[il, i0+1]. In some embodiments, a conditional branch or conditional instruction in the CTA program might be provided to handle cases where the computed address is not within the valid range (e.g., for a thread processing a pixel at the right edge of the screen, an address such as InputData [il, i0+1] would not be a valid pixel address). On completion of the CTA program, the final results (output data) produced by the threads are advantageously placed in memory for use by a subsequent CTA program or made accessible to CPU 102 (FIG. 1). For example, the final instructions in a CTA program might include an instruction to write some or all of the data generated by the thread to graphics memory 124. After execution of the CTA is finished, GPU 122 may transfer the data (e.g., using a conventional DMA operation) to system memory 104, making it available to application programs executing on CPU 102. Other data transfer mechanisms may also be used. Core 126 advantageously signals core interface 128 upon completion of a CTA, so that core interface 128 can initiate execution of a next CTA, reusing the resources that became free when the first CTA was completed. 20:35–21:8. To the extent Plaintiff contends that the preamble is not expressly or inherently disclosed, or is not obvious in view of Nicholls and the knowledge of a POSITA, it would have been obvious to

Ex. A8 – Invalidity of U.S. Patent No. 8,648,867 – Nickolls

U.S. Patent No. 8,648,867	Nickolls
	combine the teachings of Nicholls 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 ’867 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 Nicholls. Specific examples of such references are cited and explained in these invalidity contentions, including the cover pleading and charts A1–A13, and Appendix A.
16[e]: swapping the first pointer with the second pointer at the end of the first computational cycle, such that the first output data becomes an input for a second computational cycle of the numerical simulation.	Nicholls discloses “swapping the first pointer with the second pointer at the end of the first computational cycle, such that the first output data becomes an input for a second computational cycle of the numerical simulation.” <i>See e.g.:</i> <i>As a non-limiting example, Nickolls discloses use of intermediate or output data from one step (including from any related thread in a CTA) as input data in a subsequent step, thus allowing first output data to be an input for a second computational cycle of the numerical simulation. When using pointers, as disclosed in Nickolls, this use of an output as a subsequent input is most efficiently performed by swapping the first pointer to output data with the second pointer to input data.</i> The present invention relates in general to parallel data processing, and in particular to parallel data processing methods using arrays of threads that are capable of sharing data, including intermediate results, with other threads in a thread-specific manner. 1:22–26. Embodiments of the present invention provide data processing systems and methods that use cooperative thread arrays (CTAs) to perform computations. As used herein, a "cooperative thread array," or "CTA," is a group of multiple threads that concurrently execute the same program on an input data set to produce an output data set. Each thread in a CTA has a unique identifier (thread ID) assigned at thread launch time that controls various aspects of the thread's processing behavior. For instance, a thread ID may be used to determine which portion of the input data set a thread is to process, to identify one or more other threads with which a given thread is to share an intermediate result, and/or to determine which portion of the output data set the thread is to compute or write.

Ex. A8 – Invalidity of U.S. Patent No. 8,648,867 – Nickolls

U.S. Patent No. 8,648,867	Nickolls
	2:22–35. While different threads of a CTA can execute independently of each other, it is often useful to synchronize some or all of the threads at certain points during execution of the CTA program. For instance, if one thread produces an intermediate result that will be consumed by another thread, the two threads are advantageously synchronized at least to the extent that the producing thread is guaranteed to write the intermediate result to the designated location in shared memory before the consuming thread attempts to read it. 21:10–19. A barrier arrival instruction might be used in preference to a barrier arrive-and-wait instruction, e.g., where one thread produces an intermediate result to be consumed by one or more other threads but does not consume intermediate results from those threads. 25:8–11. In accordance with an embodiment of the present invention, FFT butterfly calculations are advantageously implemented in a CTA. Each thread starts by reading its own input value, then reads an input value of another thread (determined based on thread IDs) and performs two multiplications and two additions to arrive at the next intermediate value. The next intermediate value is written back to the global register file so that it can be shared with another thread, and so on. At each FFT stage, the CTA threads synchronize and wait for other threads' intermediate values to be written before reading those intermediate values. 29:16–24. To the extent Plaintiff contends that the preamble is not expressly or inherently disclosed, or is not obvious in view of Nicholls and the knowledge of a POSITA, it would have been obvious to combine the teachings of Nicholls 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 '867 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 Nicholls. Specific examples of such references are cited and

Ex. A8 – Invalidity of U.S. Patent No. 8,648,867 – Nickolls

U.S. Patent No. 8,648,867	Nickolls
	explained in these invalidity contentions, including the cover pleading and charts A1–A13, and Appendix A.
17: The method as claimed in claim 16, further comprising: sending, by the accelerator controller, instructions for performing the at least one calculation to the at least one graphics processing unit.	Nicholls discloses “sending, by the accelerator controller, instructions for performing the at least one calculation to the at least one graphics processing unit.” <i>See e.g.:</i> <i>As a non-limiting example, Nickolls discloses sending, by the accelerator controller (e.g., core interface 128) instructions (such as specific programs) for performing at least one calculation to at least one graphics processing unit (e.g., the core 126 and/or the processing engines within core 126). In the alternative, as a further non-limiting example, Nickolls discloses an instruction unit 312 which works together with the core interface 128, and so together are an accelerator controller, which fetches and sends instructions to perform calculations to the processing engines in the core.</i>

Ex. A8 – Invalidity of U.S. Patent No. 8,648,867 – Nickolls

U.S. Patent No. 8,648,867	Nickolls
	The diagram illustrates a processing core (126) with a CORE INTERFACE (128) at the top. Inside the core, there is a MEMORY (308) on the left. The INSTRUCTION UNIT (312) is a central component containing several sub-units: FETCH LOGIC (318), PC G_ID (322), MUX (316), PC LOGIC (314), ISSUE LOGIC (324), and ARB LOGIC (320). Below the instruction unit are multiple PROC ENGINES (0, 1, ..., P-1). These engines are connected to a LOCAL REGISTER FILE (304) and a GLOBAL REGISTER FILE (SHARED) (306) via a bus (305). Signals such as INSTR, G_ID, and PC are shown flowing between the instruction unit and the proc engines. The entire core is enclosed in a green box labeled CORE. FIG. 3 Fig. 3 (annotated). According to another aspect of the present invention, a processor includes a processing core and core interface logic coupled to the processing core. The processing core includes multiple processing engines configured to execute multiple threads concurrently and a shared memory (e.g., a global register file or other on-chip or off-chip memory) communicably coupled to the processing engines. The core interface logic includes a launch module. The launch module is configured to initiate concurrent execution by the processing core of threads making up a thread array, each thread of the thread array being configured to execute a same program on an input data

Ex. A8 – Invalidity of U.S. Patent No. 8,648,867 – Nickolls

U.S. Patent No. 8,648,867	Nickolls
	set. The launch module is also configured to assign a unique thread identifier (which can be, e.g., a one-dimensional or multidimensional identifier) to each of the threads prior to instructing the processing core to initiate execution of the thread. In operation, each thread of the thread array uses the unique thread identifier to determine at least one processing behavior, such as a subset of an input data set to process and/or a portion of an output data set to produce or write. 3:39–57. In one embodiment, core 126 includes an array of P (e.g., 8, 16, or any other number) parallel processing engines 302 configured to receive and execute SIMD instructions from a single instruction unit 312. Each parallel processing engine 302 advantageously includes an identical set of functional units such as arithmetic logic units, load/store units, and the like (not explicitly shown). The functional units may be pipelined, allowing a new instruction to be issued before a previous instruction has finished, as is known in the art. Any combination of functional units may be provided. In one embodiment, the functional units support a variety of operations including integer and floating point arithmetic (e.g., addition and multiplication), comparison operations, Boolean operations (AND, OR, XOR), bit-shifting, and computation of various algebraic functions (e.g., planar interpolation, trigonometric, exponential, and logarithmic functions, etc.); and the same functional-unit hardware can be leveraged to perform different operations. A particular implementation of processing engines 302 is not critical to the present invention, and a detailed description has been omitted. 10:37–56. Instruction unit 312 is configured such that, for any given processing cycle, the same instruction (INSTR) is issued to all P processing engines 302. Thus, at the level of a single clock cycle, core 126 implements P-way SIMD execution. Each processing engine 302 is also multithreaded, supporting up to G concurrent threads, and instructions for different ones of the G threads can be issued in any order relative to each other. Accordingly, core 126 in this embodiment can have up to P*G threads in flight concurrently. For instance, if P=16 and G=24, then core 126 can support up to 384 concurrent threads. In some embodiments, P*G determines an upper limit on the number of threads that can be included in a CTA; it is to be understood that some CTAs may include fewer than this number of threads.

Ex. A8 – Invalidity of U.S. Patent No. 8,648,867 – Nickolls

U.S. Patent No. 8,648,867	Nickolls
	Because instruction unit 312 issues the same instruction to all P processing engines 302 in parallel, core 126 is advantageously used to process threads in "SIMD groups." As used herein, a "SIMD group" refers to a group of up to P threads of execution of the same program on different input data, with one thread of the group being assigned to each processing engine 302. For example, a SIMD group might consist of any P threads of a CTA, each of which executes the same CTA program. A SIMD group can include fewer than P threads, in which case some of processing engines 302 will simply be idle during cycles when that SIMD group is being processed. Since a processing engine 302 can support up to G threads, it follows that up to G SIMD groups can be in flight in core 126 at any given time. 11:27–54. Instruction unit 312 advantageously manages instruction fetch and issue for each SIMD group so as to ensure that threads in a group that have diverged eventually resynchronize. In one embodiment, instruction unit 312 includes program counter (PC) logic 314, a program counter register array 316, a multiplexer 318, arbitration logic 320, fetch logic 322, and issue logic 324. Program counter register array 316 stores G program counter values (one per SIMD group), which are updated independently of each other by PC logic 314. PC logic 314 updates the PC values based on information received from processing engines 302 and/or fetch logic 322. PC logic 314 is advantageously configured to track divergence among threads in a SIMD group and to select instructions in a way that ultimately results in the threads re-synchronizing. 12:19–34. For optimal performance, all threads within a SIMD group are advantageously launched on the same clock cycle so that they begin in a synchronized state. In one embodiment, core interface 128 advantageously loads SIMD groups into core 126, then instructs core 126 to launch the group. "Loading" a thread, as used herein, includes supplying instruction unit 312 and processing engines 302 with various input parameters required to execute the program. In some instances, the input parameters may include the input data to be processed by the 20 program; in other instances, the input data is stored in global register file 306 or other shared memory (e.g., graphics memory 124 or system memory 104 of FIG. 1) prior to loading of any threads, and the input parameters may include a reference to a location where the input data is stored. For 25 example, in the case of CTA processing, the input data set may be loaded into graphics memory 124 or system memory

Ex. A8 – Invalidity of U.S. Patent No. 8,648,867 – Nickolls

U.S. Patent No. 8,648,867	Nickolls
	104 before core interface is instructed to begin CTA processing. Core interface 128 loads the starting PC value for the CTA program into a slot in PC array 316 that is not currently in use; this slot corresponds to the group index GID assigned to the new SIMD group that will process P of the CTA threads. Core interface 128 allocates sufficient space in the local register file for each processing engine 302 to execute one CTA thread, then loads input parameters into shared memory (e.g., 35 global register file 306). Core interface 128 loads a unique thread ID into a thread ID register for each thread or into a predetermined register in the allocated portion of local register file 304 for each processing engine 302. In one embodiment, thread IDs for P threads are loaded in parallel, as 40 described below. Once the input parameters and thread IDs for all threads in the SIMD group have been loaded, core interface 128 launches the group by signaling instruction unit 312 to begin fetching and issuing instructions corresponding to the group index GID of the new group. 13:12–45. State module 404 receives state information from input unit 402 and loads the state information into state registers 406. “State information,” as used herein, includes any information (other than input data) relevant to defining a CTA. For example, in one embodiment, state information includes the size of the input data set, the amount of local register file space required for each thread, and a starting program counter (e.g., memory address) for a program to be executed by each thread. State information advantageously also includes size information for the CTA; for example, referring to FIG. 2A, array dimensions D0, D1 and D2 may be provided. In some embodiments, the total number (7) of threads is also provided; in other embodiments, T can be computed from the array dimensions (e.g., $T=D0*D1*D2$ for the embodiment of FIG. 2A). When T is provided, T is advantageously less than or equal to $D0*D1*D2$, in order to ensure that each thread is assigned a unique thread ID. 14:41–57. Once all the input parameters have been loaded into global register file 306, load module 408 issues a “GO” control signal to launch module 410. Launch module 410 is configured to respond to the GO control signal by launching all of the threads for one CTA based on the current definition of CTA size stored in state registers 406. Launch module 410 supplies each thread with a unique thread ID, which is written to one of the local registers 304 (FIG. 3) allocated to that

Ex. A8 – Invalidity of U.S. Patent No. 8,648,867 – Nickolls

U.S. Patent No. 8,648,867	Nickolls
	thread or to a per-thread register dedicated to storing a thread ID. Once all the threads in a SIMD group have been supplied with thread IDs, launch module 410 advantageously launches that SIMD group before beginning to supply thread IDs for the next SIMD group. Launch module 410 is advantageously designed to generate and assign thread IDs rapidly so as to minimize the delay between launching successive SIMD groups. In one embodiment, the delay can be as little as one clock cycle between successive launches. 17:20–36. To the extent Plaintiff contends that the preamble is not expressly or inherently disclosed, or is not obvious in view of Nicholls and the knowledge of a POSITA, it would have been obvious to combine the teachings of Nicholls 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 ’867 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 Nicholls. Specific examples of such references are cited and explained in these invalidity contentions, including the cover pleading and charts A1–A13, and Appendix A.
18: The method as claimed in claim 16, further comprising: partitioning the accelerator memory into the first partition, the second partition, a third partition to store internal variables, and a fourth partition to store data used as input at a particular computation cycle of the numerical simulation.	Nicholls discloses “partitioning the accelerator memory into the first partition, the second partition, a third partition to store internal variables, and a fourth partition to store data used as input at a particular computation cycle of the numerical simulation.” <i>See e.g.</i>: <i>See</i> 16[b], 16[d]. <i>As a non-limiting example, Nickolls discloses that its accelerator memory (e.g., memory located in the GPU) is partitioned in a first partition to store input data and a second partition to store output data. See 16[b]. Nickolls also discloses storage of internal variables (e.g., input parameters specific to a particular thread or the total number of threads in a CTA) in a “third partition” of the accelerator memory (e.g., allocated space within state register 406). Nickolls further discloses storage of data used as input at a particular computational cycle of the numerical simulation (e.g., input data to be processed or used, such as coefficients of attribute equations) in a “fourth partition” of the accelerator memory (e.g., allocated space within the local register file for each processing engine).</i>

Ex. A8 – Invalidity of U.S. Patent No. 8,648,867 – Nickolls

U.S. Patent No. 8,648,867	Nickolls
	FIG. 4 Fig 4. Each processing engine 302 also has access, via a crossbar switch 305, to a (shared) global register file 306 that is shared among all of the processing engines 302 in core 126. Global register file 306 may be as large as desired, and in some embodiments, any processing engine 302 can read to or write from any location in global register file 306. In addition to global register file 306, some embodiments also provide an on-chip shared memory 308, which may be implemented, e.g., as a conventional RAM. On-chip memory 308 is advantageously used to store data that is expected to be used in multiple threads, such as coefficients of attribute equations, which are usable in pixel shader programs. In some embodiments, processing engines 302 may also have access to additional off-chip shared memory (not shown), which might be located, e.g., within graphics memory 124 and/or system memory 104 of FIG. 1. 11:3–18.

Ex. A8 – Invalidity of U.S. Patent No. 8,648,867 – Nickolls

U.S. Patent No. 8,648,867	Nickolls
	For optimal performance, all threads within a SIMD group are advantageously launched on the same clock cycle so that they begin in a synchronized state. In one embodiment, core interface 128 advantageously loads SIMD groups into core 126, then instructs core 126 to launch the group. "Loading" a thread, as used herein, includes supplying instruction unit 312 and processing engines 302 with various input parameters required to execute the program. In some instances, the input parameters may include the input data to be processed by the 20 program; in other instances, the input data is stored in global register file 306 or other shared memory (e.g., graphics memory 124 or system memory 104 of FIG. 1) prior to loading of any threads, and the input parameters may include a reference to a location where the input data is stored. For 25 example, in the case of CTA processing, the input data set may be loaded into graphics memory 124 or system memory 104 before core interface is instructed to begin CTA processing. Core interface 128 loads the starting PC value for the CTA program into a slot in PC array 316 that is not currently in use; this slot corresponds to the group index GID assigned to the new SIMD group that will process P of the CTA threads. Core interface 128 allocates sufficient space in the local register file for each processing engine 302 to execute one CTA thread, then loads input parameters into shared memory (e.g., 35 global register file 306). Core interface 128 loads a unique thread ID into a thread ID register for each thread or into a predetermined register in the allocated portion of local register file 304 for each processing engine 302. In one embodiment, thread IDs for P threads are loaded in parallel, as 40 described below. Once the input parameters and thread IDs for all threads in the SIMD group have been loaded, core interface 128 launches the group by signaling instruction unit 312 to begin fetching and issuing instructions corresponding to the group index GID of the new group. 13:12–45. Referring again to FIG. 4, load module 408 receives input parameters for a CTA and loads the input parameters into (shared) global register file 306 (FIG. 3) of core 126. These parameters may include, e.g., an (x,y) position of the CTA within a grid of CTAs or other CTA identifier and/or other information specific to the CTA. In some embodiments, the input parameters may also include some or all of an input data set to be processed by the CTA. In one embodiment, the first received input parameter is accompanied by a "begin CTA" control signal indicating that what follows are input parameters for a CTA that is to be executed using the

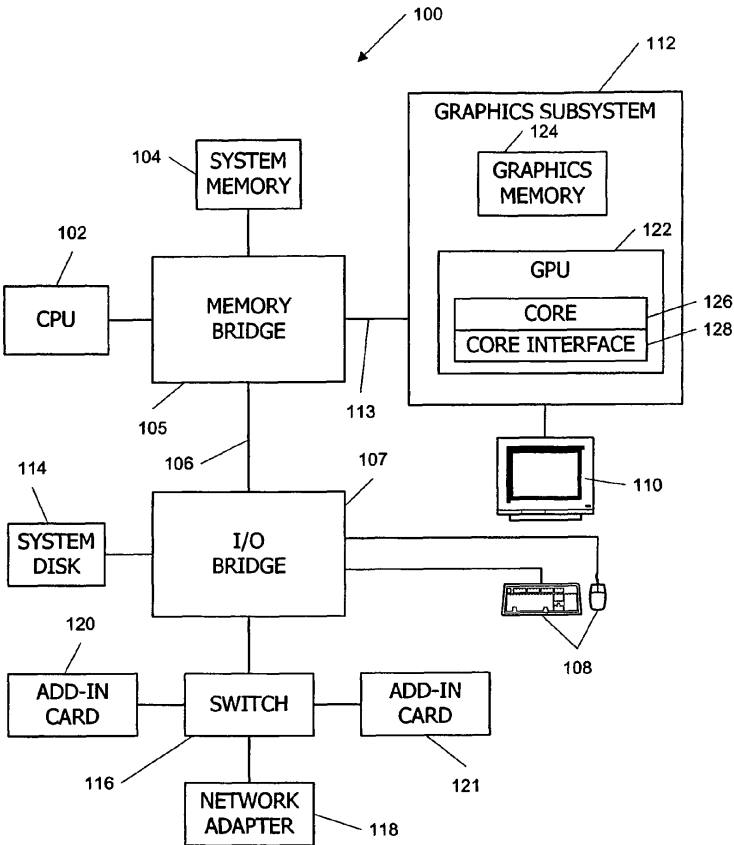
Ex. A8 – Invalidity of U.S. Patent No. 8,648,867 – Nickolls

U.S. Patent No. 8,648,867	Nickolls
	currently defined state. The control signal may also indicate the size of the input parameter set, where in global register file 306 the parameters are to be stored, and other information, e.g., a control signal identifying the last of 10 the input parameters so that core interface 128 immediately recognizes when the CTA is ready to be launched. Alternatively, state information defining the size of the input parameter set may be stored in state registers 406, and load module 408 may use this information to determine when loading of input parameters is complete. For instance, if state registers 406 store information about the number of input parameters, load module 408 might count received input parameters until the expected number had been received. 16:63–17:19. In some embodiments, the target value used to determine when synchronization is achieved may be specified as being equal to the total number of executing threads of the CTA, which can be dynamically determined by barrier synchronization logic 1100 or 1130. Although the total number of threads in a CTA can be an input parameter, as described above, in some instances, not all threads are necessarily executing at a given time; accordingly, a dynamic determination of the total is advantageous. Specifying “all executing threads” as the target value can be done, e.g., by using a predefined special value (e.g., zero) for the argument that specifies the target value or by providing a separate barrier instruction that signifies that the target value is “all executing threads.” (Such an instruction would not include a target value as an argument.) Where dynamic determination of the target number is used, barrier synchronization logic 1100 or 1130 advantageously recomputes the target number from time to time so that the target remains current. 24:39–56. To the extent Plaintiff contends that the preamble is not expressly or inherently disclosed, or is not obvious in view of Nicholls and the knowledge of a POSITA, it would have been obvious to combine the teachings of Nicholls 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 ’867 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 Nicholls. Specific examples of such references are cited and explained in these invalidity contentions, including the cover pleading and charts A1–A13, and Appendix A.

Ex. A8 – Invalidity of U.S. Patent No. 8,648,867 – Nickolls

U.S. Patent No. 8,648,867	Nickolls
19: The method of claim 16, further comprising: transferring, by the accelerator controller, the first output data to the main memory during the second computational cycle.	Nicholls discloses “transferring, by the accelerator controller, the first output data to the main memory during the second computational cycle.” <i>See e.g.:</i> <i>As a non-limiting example, Nickolls discloses transferring by the accelerator controller (e.g., at the direction and/or under the control of the core interface 128) of first output data to the main memory during a second computation cycle (e.g., from a Graphics Subsystem 112 through a Memory Bridge 105 to a System Memory 104, including using a conventional DMA process).</i>

Ex. A8 – Invalidity of U.S. Patent No. 8,648,867 – Nickolls

U.S. Patent No. 8,648,867	Nickolls
	 FIG. 1 Fig. 1. FIG.1 is a block diagram of a computer system 100 according to an embodiment of the present invention. Computer system 100 includes a central processing unit (CPU) 102 and a system memory 104 communicating via a bus path that includes a memory bridge 105. Memory bridge 105 is connected via a bus path 106 to an I/O (input/output) bridge 107. I/O bridge 107 receives

Ex. A8 – Invalidity of U.S. Patent No. 8,648,867 – Nickolls

U.S. Patent No. 8,648,867	Nickolls
	user input from one or more user input devices 108 (e.g., keyboard, mouse) and forwards the input to CPU 102 via bus 106 and memory bridge 105. Visual output is provided on a pixel based display device 110 (e.g., a conventional CRT or LCD based monitor) operating under control of a graphics subsystem 112 coupled to memory bridge 105 via a bus 113. A system disk 114 is also connected to I/O bridge 107. A switch 116 provides connections between I/O bridge 107 and other components such as a network adapter 118 and various add-in cards 120, 121. Other components (not explicitly shown), including USB or other port connections, CD drives, DVD drives, and the like, may also be connected to I/O bridge 107. Bus connections among the various components may be implemented using bus protocols such as PCI (Peripheral Component Interconnect), PCI Express (PCI-E), AGP (Accelerated Graphics Port), Hyper-Transport, or any other bus protocol(s), and connections between different devices may use different protocols as is known in the art. 6:9–34. In one embodiment, core 126 includes an array of P (e.g., 8, 16, or any other number) parallel processing engines 302 configured to receive and execute SIMD instructions from a single instruction unit 312. Each parallel processing engine 302 advantageously includes an identical set of functional units such as arithmetic logic units, load/store units, and the like (not explicitly shown). The functional units may be pipelined, allowing a new instruction to be issued before a previous instruction has finished, as is known in the art. Any combination of functional units may be provided. In one embodiment, the functional units support a variety of operations including integer and floating point arithmetic (e.g., addition and multiplication), comparison operations, Boolean operations (AND, OR, XOR), bit-shifting, and computation of various algebraic functions (e.g., planar interpolation, trigonometric, exponential, and logarithmic functions, etc.); and the same functional-unit hardware can be leveraged to perform different operations. A particular implementation of processing engines 302 is not critical to the present invention, and a detailed description has been omitted. 10:37–56. For optimal performance, all threads within a SIMD group are advantageously launched on the same clock cycle so that they begin in a synchronized state. In one embodiment, core interface 128 advantageously loads SIMD groups into core 126, then instructs core 126 to launch the group.

Ex. A8 – Invalidity of U.S. Patent No. 8,648,867 – Nickolls

U.S. Patent No. 8,648,867	Nickolls
	“Loading” a thread, as used herein, includes supplying instruction unit 312 and processing engines 302 with various input parameters required to execute the program. In some instances, the input parameters may include the input data to be processed by the program; in other instances, the input data is stored in global register file 306 or other shared memory (e.g., graphics memory 124 or system memory 104 of FIG. 1) prior to loading of any threads, and the input parameters may include a reference to a location where the input data is stored. For example, in the case of CTA processing, the input data set may be loaded into graphics memory 124 or system memory 104 before core interface is instructed to begin CTA processing. Core interface 128 loads the starting PC value for the CTA program into a slot in PC array 316 that is not currently in use; this slot corresponds to the group index GID assigned to the new SIMD group that will process P of the CTA threads. Core interface 128 allocates sufficient space in the local register file for each processing engine 302 to execute one CTA thread, then loads input parameters into shared memory (e.g., global register file 306). Core interface 128 loads a unique thread ID into a thread ID register for each thread or into a predetermined register in the allocated portion of local register file 304 for each processing engine 302. In one embodiment, thread IDs for P threads are loaded in parallel, as described below. Once the input parameters and thread IDs for all threads in the SIMD group have been loaded, core interface 128 launches the group by signaling instruction unit 312 to begin fetching and issuing instructions corresponding to the group index GID of the new group. 13:12–45. The CTA program may also include instructions to read from and/or write to the shared global register file 306, on-chip shared memory 308, and/or other memory such as graphics memory 124 or system memory 104 of FIG. 1. For instance, the input data set to be processed by the CTA may be stored in graphics memory 124 or system memory 104. Intermediate results may be written to global register file 306 and/or other memory such as graphics memory 124 or system memory 104 of FIG. 1, where they can be shared with other threads. Final results (output data) may be written to graphics memory 124 or system memory 104. 20:35–45. On completion of the CTA program, the final results (output data) produced by the threads are advantageously placed in memory for use by a subsequent CTA program or made accessible to CPU 102 (FIG. 1). For example, the final instructions in a CTA program might include an

Ex. A8 – Invalidity of U.S. Patent No. 8,648,867 – Nickolls

U.S. Patent No. 8,648,867	Nickolls
	instruction to write some or all of the data generated by the thread to graphics memory 124. After execution of the CTA is finished, GPU 122 may transfer the data (e.g., using a conventional DMA operation) to system memory 104, making it available to application programs executing on CPU 102. Other data transfer mechanisms may also be used. Core 126 advantageously signals core interface 128 upon completion of a CTA, so that core interface 128 can initiate execution of a next CTA, reusing the resources that became free when the first CTA was completed. 20:61–21:8. To the extent Plaintiff contends that the preamble is not expressly or inherently disclosed, or is not obvious in view of Nicholls and the knowledge of a POSITA, it would have been obvious to combine the teachings of Nicholls 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 ’867 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 Nicholls. Specific examples of such references are cited and explained in these invalidity contentions, including the cover pleading and charts A1–A13, and Appendix A.