

UNITED STATES PATENT AND TRADEMARK OFFICE

BEFORE THE PATENT TRIAL AND APPEAL BOARD

REALTEK SEMICONDUCTOR CORP.,
Petitioner,

v.

ATI TECHNOLOGIES ULC
Patent Owner.

Case No. IPR2023-00687
U.S. Patent No. 8,854,381

**DECLARATION OF JOHN HART REGARDING
U.S. PATENT NO. 8,854,381**

TABLE OF CONTENTS

	<u>Page</u>
I. INTRODUCTION	1
II. QUALIFICATIONS AND EXPERIENCE.....	2
III. MATERIALS CONSIDERED	4
IV. LEGAL STANDARDS	5
A. Claim Construction.....	6
B. Obviousness	6
V. CLAIM CONSTRUCTION	8
A. “task”	8
B. “engine”	11
C. “configured to”	14
D. “each of the plurality of tasks”	15
VI. RELEVANT TECHNICAL BACKGROUND AT THE TIME OF THE ’381 PATENT	15
A. Basic Graphics Processing.....	15
B. Multi-Threading Versus Simultaneous Multi-Tasking	16
C. Shader Programs	17
VII. ’381 PATENT OVERVIEW	19
A. Parallel Engines	21
B. Engines Versus Shader Programs.....	22
C. Simultaneous Multi-Tasking	24
VIII. DESCRIPTION OF ASSERTED ART	25
A. Wilt (U.S. Patent No. 8,151,095)	25
B. Jiao (U.S. Patent No. 8,319,774)	28
C. Stuttard (International Publication No. WO 00/62182)	32
IX. EVALUATION OF PETITIONER’S PROPOSED GROUNDS	34
A. Ground 1: Wilt and the Knowledge of a POSITA Do Not Render Claims 15-20 of the ’381 Patent Obvious.....	34
1. Wilt Fails To Disclose or Render Obvious Claim 15.....	34

2.	Wilt Fails To Disclose or Render Obvious Claim 19.....	45
3.	Wilt Fails To Disclose or Render Obvious Claim 20.....	47
B.	Ground 2: Jiao and the Knowledge of A POSITA Do Not Render Claims 15-20 of the '381 Patent Obvious.....	50
1.	Jiao Fails To Disclose or Render Obvious Claim 15	50
2.	Jiao Fails To Disclose or Render Obvious Claim 19	61
3.	Jiao Fails To Disclose or Render Obvious Claim 20	64
C.	Ground 3: Stuttard and the Knowledge of A POSITA Do Not Render Claims 15-20 of the '381 Patent Obvious	65
1.	Stuttard Fails To Disclose or Render Obvious Claim 15	65
2.	Stuttard Fails To Disclose or Render Obvious Claim 19	84
3.	Stuttard Fails To Disclose or Render Obvious Claim 20	86
X.	CONCLUSION.....	88

I, John Hart, being of legal age, hereby declare, affirm, and state the following:

I. INTRODUCTION

1. I have been retained by ATI Technologies UCL (“ATI” or “Patent Owner”), as an independent expert in this proceeding before the Patent Trial and Appeal Board (“PTAB” or “Board”). I understand that the Board has instituted an Inter Partes Review (“IPR”) proceeding of U.S. Patent No. 8,854,381 (the “’381 patent”) (Ex. 1001) in IPR2023-00687 brought forth by Petitioner Realtek.

2. I am providing this testimony for consideration by the Patent Trial and Appeal Board in this proceeding. Many of my opinions are also responsive to the Declaration of Massoud Pedram (“Pedram Decl.”) (Ex. 1003), which I understand was submitted in support of Realtek’s Petition for *Inter Partes* Review of the ’381 patent. I have personal knowledge of the facts and opinions set forth in this declaration and all of the opinions and conclusions found in this declaration are my own.

3. I am being compensated at my standard hourly rate for my services. I have no personal or financial stake or interest in the outcome of the present proceedings, and my compensation is not dependent in any way upon the outcome of this proceeding.

II. QUALIFICATIONS AND EXPERIENCE

4. I have been a professor of computer science, focusing on computer graphics, since 1992. I am currently a tenured Professor in the Department of Computer Science at the University of Illinois at Urbana-Champaign, a department consistently ranked in the top-5 by US News and World Report. I am also Director of the Office for Professional Education of the University of Illinois at Urbana-Champaign.

5. I received a Bachelor's of Science degree in Computer Science from Aurora University in 1987, a Master's of Science in Electrical Engineering and Computer Science from the University of Illinois at Chicago in 1989, and a Ph.D. in Electrical Engineering and Computer Science from the University of Illinois at Chicago in 1991.

6. I worked as an intern in 1998 at the IBM TJ Watson Research Center in Hawthorne, New York, and in 1999 at AT&T Pixel Machines in Holmdel, New Jersey. I worked as a postdoc in 1991-1992 at the Electronic Visualization Laboratory at the University of Illinois at Chicago with funding from the National Center for Supercomputing Applications at the University of Illinois at Urbana-Champaign. I worked as a visiting researcher in the summer of 2007 for the Graphics Research Group of Adobe Systems in Seattle, Washington.

7. From 1997-2000, I designed, simulated and patented graphics hardware for Silicon Reality and the Evans & Sutherland Computer Corp. I have also consulted for visual effects companies (e.g., Kleiser-Walczak, Blue Sky | VIFX), defense contractors (e.g., SAIC, Pratt & Whitney) and medical imaging companies (e.g., Intrinsic Medical Imaging).

8. My research in computer graphics has been supported by Adobe, DARPA, Intel, Nokia, NVIDIA, NSF and Microsoft. I have published over 75 reviewed papers on computer graphics, including papers in the most rigorous and prestigious venues, including the SIGGRAPH Annual Conference Proceedings, the ACM Transactions on Graphics, and the IEEE Transactions on Visualization and Computer Graphics. From 2002-2008, I was the longest-serving Editor-in-Chief of ACM Transactions on Graphics, the top journal in the field of computer graphics.

9. I was an Executive Producer for the ACM SIGGRAPH documentary “The Story of Computer Graphics” in 1999. I served from 1996-2000 on the ACM SIGGRAPH Executive Committee. I served as the founding computer graphics area editor of the ACM Books series, and oversaw the publication of “The VR Book: Human Centered Design for Virtual Reality.”

10. I co-authored the book “Real-Time Shading” in 2002 on the implementation of procedural shaders on graphics processors became fully user programmable GPU’s. I contributed two chapters to the book “Modeling and

Texturing: A Procedural Approach” in 2002, including a chapter on procedural texturing using rasterization and texturing operations.

11. I am being compensated for my time working on this matter at my usual rate of \$575 per hour. My compensation does not depend on the outcome of this litigation.

12. I believe I meet the criteria for a POSITA as laid out in Investigation No. 337-TA-1318, Ex. 1008 (ITC Markman Order), 9, and any other appropriate definition of a POSITA for this patent.

13. Additional details about my employment history, fields of expertise, and publications are further described in my curriculum vitae, which is attached as Exhibit A to this report.

III. MATERIALS CONSIDERED

14. In forming my opinion, I have relied on the following documents:

Exhibit	Description
Ex. 1001	U.S. Patent 8,854,381 to Lefebvre et al.
Ex. 1002	Prosecution History of U.S. Patent 8,854,381
Ex. 1003	Declaration of Massoud Pedram, Ph.D.
Ex. 1004	U.S. Patent No. 8,151,095 to Wilt (“Wilt”)
Ex. 1005	U.S. Patent No. 8,319,774 to Jiao <i>et al.</i> (“Jiao”)

Exhibit	Description
Ex. 1006	PCT Pub. No. WO00/62182A2 to Stuttard <i>et al.</i> (“Stuttard”)
Ex. 1007	Curriculum Vitae of Massoud Pedram, Ph.D.
Ex. 1008	<i>Certain Graphics Systems, Components Thereof, and Digital Televisions Containing The Same</i> , Inv. No. 337-TA-1318 (Order No. 31, Oct. 18, 2022)
Ex. 2002	Pat Hanrahan and Jim Lawson. 1990 “A Language for Shading and Lighting Calculations.” Proc. SIGGRAPH ’90. pp. 289-298
Ex. 2003	<i>Certain Graphics Systems, Components Thereof, and Digital Televisions Containing The Same</i> , Inv. No. 337-TA-1318, Final Initial Determination (July 7, 2023)
Ex. 2004	<i>Certain Graphics Systems, Components Thereof, and Digital Televisions Containing The Same</i> , Inv. No. 337-TA-1318, Respondents Post-Hearing Brief (April 10, 2023) (Redacted)
Ex. 2005	<i>Certain Graphics Systems, Components Thereof, and Digital Televisions Containing The Same</i> , Inv. No. 337-TA-1318, Deposition Transcript of Dr. Pedram (January 4, 2023)
Ex. 2006	Declaration Transcript of Massoud Pedram, January 5 and 8, 2024

IV. LEGAL STANDARDS

15. In preparing and expressing my opinions and considering the subject matter of the ’381 patent, I am relying on certain basic legal principles that counsel have explained to me. These principles are discussed below.

A. Claim Construction

16. I understand that the first step in determining the validity of a claim is for the claim to be properly construed. I have been further advised that, in an *inter partes* review proceeding, the claims of a patent are typically given their ordinary and customary meaning as would be understood by one of ordinary skill in the art in the context of the claims and the patent disclosure (specification).

17. I further understand that an inventor can provide special definitions for specific claim term(s) within the patent. However, any special definition for a claim term must be set forth in the specification with reasonable clarity and precision.

18. I also understand that a statement made by an inventor during examination of a patent application which can limit the scope of protection provided by the resulting patent. For example, if an inventor disclaims a certain embodiment during prosecution, then that embodiment is not considered within the scope of the claims of the patent.

B. Obviousness

19. I have been informed that a claim may be invalid as obvious if the subject matter described by the claim as a whole would have been obvious to a POSITA at the time the claimed invention was made. I understand that the standard for obviousness in an *inter partes* review proceeding is by a preponderance of the evidence.

20. I have also been informed that a determination of obviousness involves an analysis of the scope and content of the prior art, the similarities between the claimed invention and the prior art, and the level of ordinary skill in the art. I have been informed and understand that a prior art reference should be viewed as a whole.

21. I have been informed that in considering whether an invention for a claimed combination would have been obvious, I may assess whether there are apparent reasons to combine known elements in the prior art in the manner claimed in view of interrelated teachings of multiple prior art references, the effects of demands known to the design community or present in the market place, and/or the background knowledge possessed by a POSITA. I also understand that other principles may be relied on in evaluating whether a claimed invention would have been obvious.

22. I have been informed that, in making a determination as to whether or not the claimed invention would have been obvious to a POSITA, one may consider certain objective indicators of non-obviousness if they are present, such as: commercial success of product(s) practicing the claimed invention; long-felt but unsolved need; teaching away; unexpected results; copying; and praise by others in the field. I understand that for such objective evidence to be relevant to the non-obviousness of a claim, however, there must be a causal relationship (called a “nexus”) between the claim and the evidence. I also understand that this nexus must

be based on a novel element of the claim rather than something available in the prior art.

23. I have also been informed and understand that when considering the obviousness of a patent claim, one should consider whether a reason or motivation existed for combining the elements of the references in the manner claimed, and that the prior art must create a reasonable expectation of success in producing the claimed subject matter.

V. CLAIM CONSTRUCTION

24. I understand certain terms of the '381 patent have been previously construed by the ALJ in the ITC Investigation. *See* Ex. 1008, 36-41. Dr. Pedram applied the ALJ's constructions for the terms "shader core," "task," and "load." Ex. 1003 (Pedram Decl.), ¶¶40, 41, 44. I have applied the constructions for "shader core" and "load" as well. As explained below, I have applied a similar construction for "task."

A. "task"

25. Dr. Pedram adopted the ITC construction of this task: "a unit of processing work or portion of an application to be executed." Ex. 1003, ¶41. An additional clarification that both Dr. Pedram and I substantively agree with: a thread is not a task. Ex. 1003, ¶102. The reason is simple: as the '381 Patent states, and as

a POSITA would know, a task corresponds more than one thread, which is why Dr. Pedram has repeatedly stated threads are not tasks.

26. The '381 Patent “is directed to a processing unit, such as a graphics-processing unit (GPU), [which] is a complex integrated circuit that is adapted to perform data-parallel computing tasks, such as graphics-processing tasks.” Ex. 1001, 1:18-25. A POSITA would have understood a GPU to be able to execute a graphics processing task, such as rendering a 3-D scene consisting of triangles and textures into a photorealistic image depicting that scene from a particular viewpoint. A POSITA would have understood that a task, in order to be advantageously executed on the GPU, would need to be decomposed into many threads. A POSITA would have understood that the GPU was capable of executing multiple threads in parallel. A POSITA would have understood that a graphics task consisted of a plurality of vertex shader threads and a plurality of pixel shader threads (and optionally a plurality of geometry shader threads). A POSITA would have understood that a non-graphics task (*e.g.*, such as a “physics task” from a video-game application relating to non-graphics physical motion simulation) consisted of a plurality of what were known as “compute” shader threads.

27. The disclosure of reception of and execution of a thread does not disclose reception and execution of a task. As the '381 Patent states, and as a POSITA would know, a task comprises more than one thread. The '381 Patent

describes the relationship between task and thread as follows: “To perform a single graphics-processing task, a GPU may execute *a plurality of different threads* (sequence of instructions). Each thread may comprise a shader program, such as a geometry shader, a pixel shader, a vertex shader, or the like.” Ex. 1001, 1:49-57 (emphasis added). Taking Dr. Pedram’s example, a graphics task may be drawing a triangle of given shape and size with white boundaries filled with a solid blue color in the right upper corner of a 1920 x 1080 LCD Display. Ex. 1003, ¶102. This parent task is ultimately decomposed into a plurality of threads (*e.g.*, vertex and covered pixel generation) and shader programs (*e.g.*, vertex processing, lighting, line segment processing, pixel processing, color and texture mapping). *Id.* The ’381 Patent further explains that tasks are stored by an operating-system scheduler in a command buffer prior to being passed to the GPU. *See* Ex. 1001, 2:8-19. Thus, the claimed task originates from outside of the GPU. Petitioner’s adopted construction also recognizes this by the definition “a portion of an application.” Pet., 41.

28. Although Dr. Pedram implicitly argues that a single thread can be a task, he also explicitly states that a thread is not a task. Ex. 1003, ¶102. A POSITA would not know how to trace a thread back to its parent task—even if one thread could be a task — without substantial technical details, none of which are present in the asserted references. Some technical details would be information regarding the microarchitecture, the hardware, the system, what compiler is used, programming

language, etc. And even so, it would be a difficult endeavor, one that even a person with ordinary skill in the art might not be able to achieve. In some instances, even with all of this information, it is impossible to determine which parent task a thread was decomposed from.

29. Tasks are not the same construct as threads, even if a task was decomposed into one thread. Threads, and in reality, the instructions of a thread, are what is ultimately executed by a GPU, whereas a task as claimed by the '381 Patent, which originates from outside of the GPU, must be transformed, *e.g.*, decomposed into threads that are executable by the GPU. The threads, when executed by the GPU, achieve the goal of the parent task, an example of which is Dr. Pedram's "draw a triangle" described above.

Therefore, "task" should be construed as "a unit of processing work decomposable into two or more related threads."

B. "engine"

30. The '381 invention of a GPU with multiple engines, such that each engine manages its own buffer and can manage its own task independently of the other engines, would have allowed the GPU to schedule and allocate the GPU to process all of the tasks more efficiently. A POSITA would have recognized the term "engine" from game development, where a "game engine" would include a "graphics engine," an "AI engine," a "physics engine," and perhaps others. A

POSITA would have understood these component software “engines” of a software “game engine” would each have been managed by a corresponding hardware “engine” in the novel GPU design disclosed by ’381.

31. The claims themselves require that *each* engine must be configured to independently (i) receive, from a scheduling module, at least one task, and to (ii) load state data. Ex. 1001, Claim 15.

32. Claim 15 requires “a plurality of engines configured to receive, from a scheduling module associated with a second processing unit, a plurality of tasks and to load state data associated with each of the plurality of tasks[.]” *Id.*, Claim 15. Thus, the claim itself compels that each of the plurality of engines must (i) receive at least one task. *Id.* Furthermore, Claim 15’s use of the antecedent “the” in the phrase “and to load state data associated with each of the plurality of tasks” refers to the task received by a given engine. *Id.* Thus, a POSITA reading claim 15 would understand that any given engine of the plurality of engines must perform both of the claimed functions of (i) receiving a task from a scheduling module and (ii) loading the state data associated with the task received by the engine. *See, e.g., id.*

33. This reality is reflected throughout the entirety of the specification, including, for example, Figure 4, which depicts each engine performing the required tasks—not a random assortment of functional blocks, some of which perform some of the claimed functions but not others, and others that perform the remaining

functions. *See also id.*, 9:34-36 (“OS scheduler 310 receives the tasks generated by application 302 and issues the tasks to different virtual engines of GPU 110. For example, OS scheduler 310 issues first task 308A to a first virtual engine 312A and issues second task 308B to a second virtual engine 312B.”); *see also id.*, 6:65-7:19, 9:47-49, 10:25-39, Figs. 2, 3A, 3B, 4.

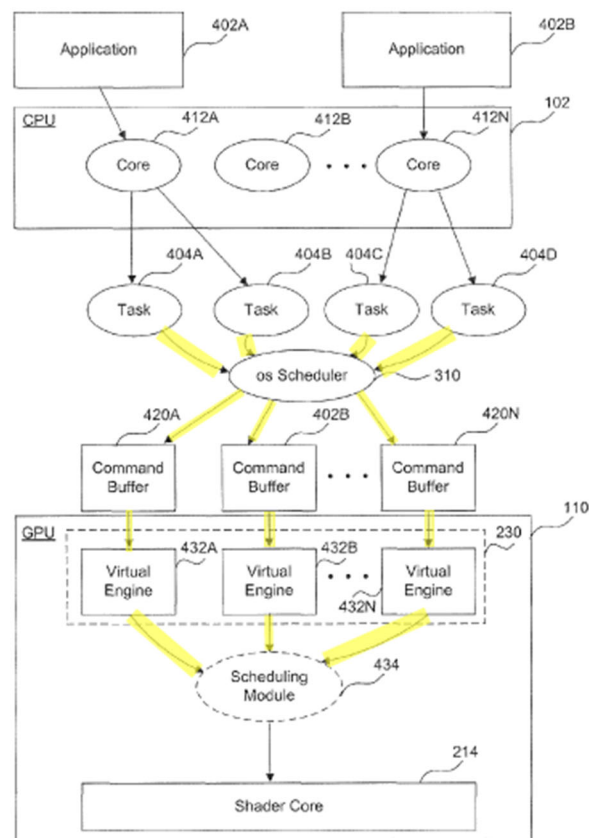


FIG. 4

Id., Fig. 4 (annotated, where yellow each path represents a task).

34. Furthermore, these two requirements are supported by the '381 Patent's disclosure and depiction of the engines, and their receipt of tasks in a parallel

configuration, which a POSITA would understand to mean that each engine loads the state data for whichever task it receives. *E.g., id.*, 4:48-51, 6:1-6. For example, the '381 Patent expressly provides that each engine must function independently:

Each virtual engine is configured to receive data-parallel processing tasks (e.g., graphics-processing tasks and general-compute tasks) and independently execute these tasks on the single shader core.

Id., 4:48-51.

35. Thus, “engine” should be construed as being required to independently:

(i) receive, from a scheduling module, at least one task, and (ii) load state data.

C. “configured to”

36. The term “configured to” claim language refers to the design of an apparatus to perform an action and not merely for the action to occur some of the time. In the context of claim 15, the claim language “a shader core configured to ...” means it is insufficient that the shader core is simply “capable of” executing a first task and a second task simultaneously. Rather, the shader core must be specifically programmed or designed to cause a first task and a second task to be executed simultaneously – and not just any two tasks, but a first task and a second task from *the same plurality of tasks* received by the plurality of engines. Further, because the claim requires the system to be “configured to” execute a graphic task while executing a general compute task, it is insufficient to point to “the result of chance or probability, which only requires capability.” “Configured to” refers to the

system’s design, not its use at any given point in time. The shader core must be specifically programmed or designed to cause a first task and a second task to be executed simultaneously—and not just any two tasks, but a first task and a second task from the same plurality of tasks received by the plurality of engines. If a first task and a second task from a plurality of tasks are occasionally executed at the same time, and it happens as a matter of probability or chance, this is nothing more than a shader core that is capable of executing a first task and a second task simultaneously.

D. “each of the plurality of tasks”

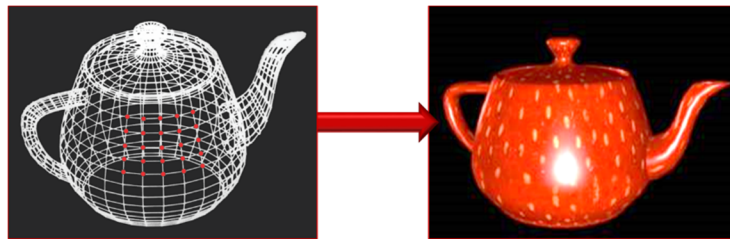
37. I understand that Petitioner “assert[ed] that each individual engine ‘must be configured to load state data for each of the pluralities of tasks that are received’” in the 1318 Investigation. Ex. 2003, 99-100; Ex. 2004 (Pet. ITC Post-Hearing Br.), 73-76. I agree that each engine must be configured to load state data, although I do not agree that each item of state data must correspond to a plurality of tasks, *i.e.*, more than one task.

VI. RELEVANT TECHNICAL BACKGROUND AT THE TIME OF THE ’381 PATENT

A. Basic Graphics Processing

38. Graphics processing involves manipulating and rendering 3D graphical objects onto a 2D display screen. Specifically, 3D objects (typically represented by a collection of “primitives” geometrical shapes such as triangles) are converted into a 2D image (typically comprising a collection of pixels) as shown below. The 2D

image represents a depiction of the 3D objects, rendered from the perspective of a given viewer, and shaded to convey the object's surface material illuminated from a given light source. V.A



39. The conversion of a 3D shape consisting of a mesh of triangles into a photorealistic 2D image consisting of an array of pixels happens in a number of processing stages. The “pixel processing” stage relevant here includes the computation of the colors of each of these pixels/fragments, as well as a process called “texture mapping.”

B. Single-Threading Versus Multi-Threading

40. An ordinary “single-threaded” graphics processing system executes a single thread, proceeding instruction by instruction, in their original order, on one piece of data a time. A multi-threaded processing system can improve upon single-threading by enabling the switch of the active thread being processed to another thread. A multi-threaded processing system does not enable the processing of multiple *tasks* (as claimed by the '381 Patent).

41. Such multi-threaded systems have substantial disadvantages, though. First, switching threads is time-consuming. Ex. 1001, 2:38-43, 46-56. Second, it

requires the system to have additional local memory to hold data for an instruction that is in the middle of execution at the time of switch, so that progress is not lost, and the system can later switch back and keep executing without starting from the beginning. *Id.*, 2:43-46.

C. Shader Programs

42. Shader programs, such as vertex shaders, geometry shaders, and pixel shaders, are pre-programmed software modules that receive data and operate on them. The concept of a “shader” program dates back to Hanrahan and Lawson’s “A Language for Shading and Lighting Calculations”, which showed how Pixar’s non-real-time film production rendering system (which would take minutes if not hours to produce each frame) could include short custom programs called “shaders” that would be run pixel-by-pixel to compute the appearance of a surface. *See generally* Ex. 2002 (Hanrahan). As the paper explains, these are “preprogrammed shaders” that are written in code, compiled once, loaded into the processor, and then run, and while running they process (or “evaluate”) data that they receive. Ex. 2002, Sections 2 (“The shading language allows procedures, called shaders, to be written that compute the various terms in the above equation.”), 4 (describing the “implementation” of shader programs, starting with compilation of the code comprising the program, followed by loading the compiled program and the program evaluating data as it runs), Table 1 (showing only one compilation step).

43. When the graphics industry settled on a user programmable interface for the graphics rendering tasks performed by the GPU for video-game applications, it used this same “shader” concept that was developed for cinematic graphics rendering tasks. While the original “shaders” were used to compute color and other visual aspects of each pixel, on the GPU they became a more generalized term for a sequence of instructions that are designed to be run in parallel on a SIMD processor. For example, a single shader (sequence of instructions) might be executed simultaneously on 32 different sets of data values on a SIMD processor with 32 execution units, which would represent the execution of 32 threads in parallel.

44. A typical GPU graphics rendering task could consist of a vertex shader that was executed in parallel on all of the triangle vertices in a 3-D scene, then a geometry shader that was executed in parallel on all of the triangles (whose vertices resulted from the vertex shader), and a pixel shader that was executed in parallel on all of the pixels inside the triangles (resulting from the geometry shader).

45. A typical GPU non-graphics task, such as a physics task, would also consist of one or more shaders, known as “compute” shaders, even though these “shaders” were not designed to compute the color of a pixel. These compute shaders generalized the GPU from graphics tasks to other non-graphics tasks, such as physics and AI, and in doing so, kept the “shader” nomenclature originally used in its graphics tasks.

46. Notably, as explained further below, shader programs are not intelligent processing units that can receive, interpret, and execute “tasks.” They do not receive “tasks” at all.

VII. '381 PATENT OVERVIEW

47. The '381 patent sought to solve the problems with prior-art graphics processing units (GPUs), including the issues with simpler multi-threaded architectures described above. *See* Section VI.B.

48. Where other systems serially entered tasks into a shader core at a single entry point so that the shader core executed tasks one at a time, the '381 Patent enabled multi-tasking, the execution of two or more different streams of processing tasks at the same time. Ex. 1001, 4:30-5:45. This was achieved the novel introduction of a set of parallel engines (as shown, for example, as blocks 202A–E in Fig. 2 below) that provide multiple entry paths for tasks to reach the shader core. Ex. 1001, 4:30-5:45. Each parallel engine is configured to (i) receive, from a scheduling module (*e.g.*, an operating system (OS) scheduler), a plurality of tasks substantially in parallel with each other and (ii) load a set of state data associated with each of the plurality of tasks. *Id.*, Abstract, 3:3-7

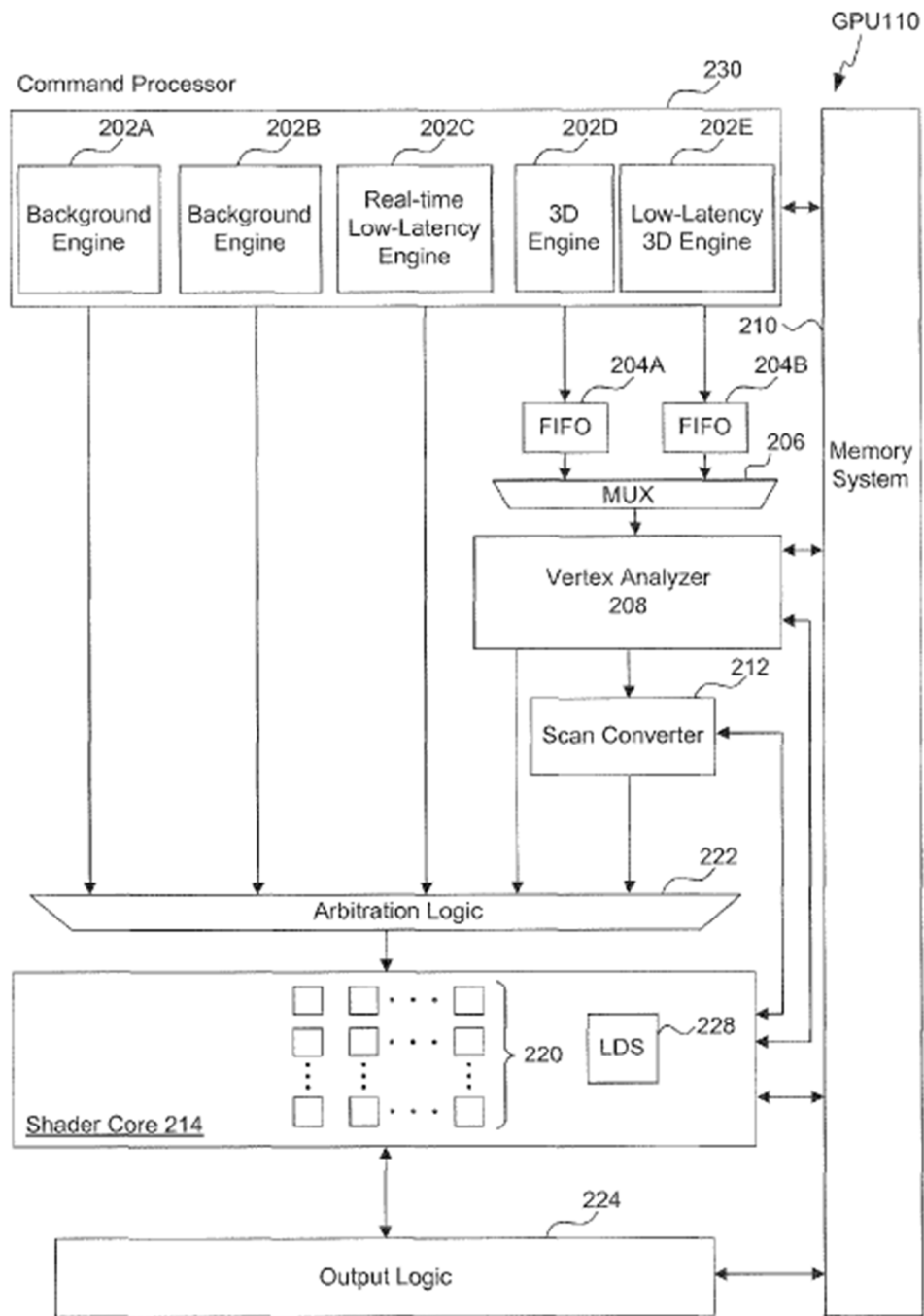


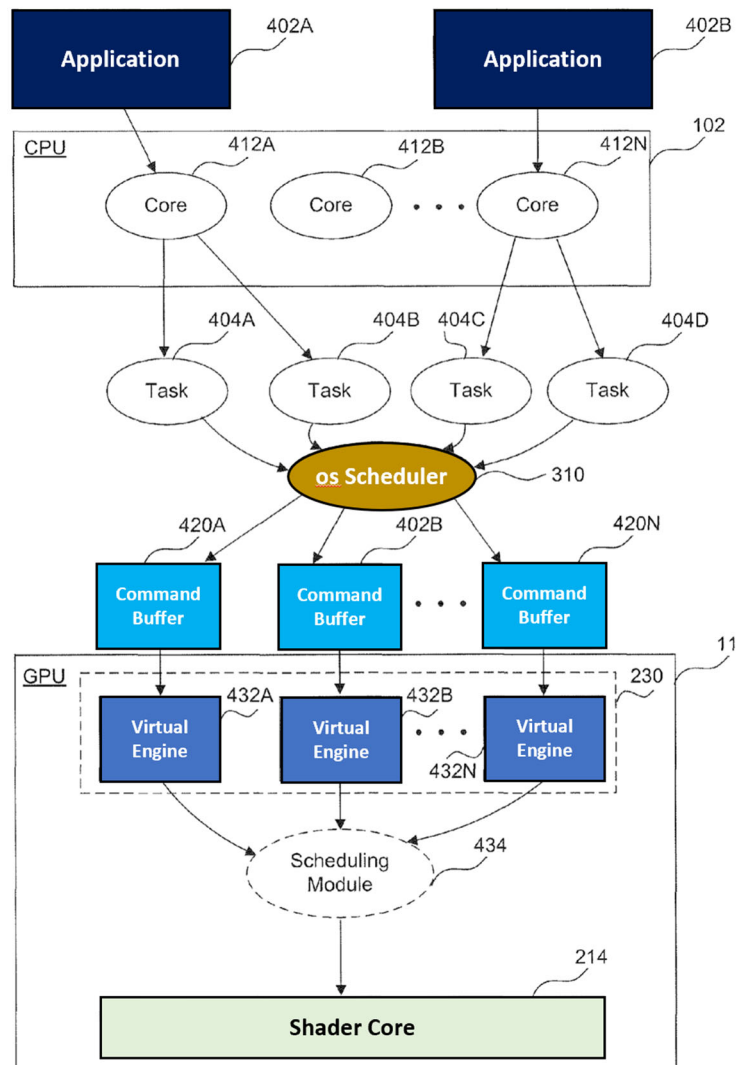
FIG. 2

Ex. 1001, Fig. 2.

49. This enabled the improved scheduling of tasks to the GPU to provide more tasks to the shader core. *Id.* Having more tasks available to the shader core means that the core is more likely to have a task available and ready for execution. *Id.* As the shader core grows to hundreds and thousands of individual processors, it becomes more challenging to produce enough tasks to keep them all running at peak efficiency. *Id.* Further, the capability of each of these engines to load state data associated with their respective tasks serves to reduce latency relative to conventional systems which relied on a single functional unit or the CPU to load and switch between different state data (*e.g.*, context data). *Id.*

A. Parallel Engines

50. The '381 patent distinguishes its parallel configuration of the engines from the conventional serial reception of tasks by the GPU as in the prior art. For example, the '381 patent explains that the engines function independently to receive tasks scheduled for execution by shared resources of the GPU. Ex. 1001, 4:48-51 (“Each virtual engine is configured to receive data-parallel processing tasks (*e.g.*, graphics-processing tasks and general-compute tasks) and independently execute these tasks on the single shader core.”); 6:1-6 (“GPU 110 includes a plurality of virtual engines that share resources of a single shader core. In this way, the plurality of virtual engines of GPU 110 can execute a plurality of tasks substantially in parallel.”). This is depicted, for example, in Figure 4:



Id., Fig. 4 (annotated).

B. Engines Versus Shader Programs

51. Further, the engines described in the '381 patent are differentiated from other elements, including shader programs. For example, as shown in Fig. 2 (reproduced above on page 17) and as disclosed at 7:5-19, the graphics-processing tasks are received by, *e.g.*, the 3D engine 202D and the low-latency 3D engine 202E, which connect to certain graphics resources of the GPU, such as the vertex analyzer

208 and scan converter 212. The '381 patent describes that these graphics components may invoke shader programs to be run on the shader core: “Vertex analyzer 208 identifies shader programs associated with a graphics-processing and/or general-compute task and schedules when each shader program can be launched in shader core 214 based on input and output data that will be available.” Ex. 1001, 7:42-46. “Processing elements 220 may implement a plurality of different shader programs (*e.g.*, geometry shader, vertex shader, pixel shader, tessellation shader, or the like).” *Id.*, 8:34-41. By this point in the processing chain, the engines 202 have already received the tasks, and as explained in the '381 patent and in Section VI.C above, the shader programs merely receive data and do not receive tasks: they are preprogrammed pieces of software that are invoked as part of carrying out a task, *i.e.*, they are constituent components of tasks. They bear no similarity to the “engines” recited in the '381 patent, nor do they perform any of the actions the engines are required to do in the challenged claims.

52. The '381 patent confirms in other places that “shader programs” are actually constituent components of tasks; they are not *engines*, which according to the claim language, receive tasks. The '381 patent teaches that in its terminology, “shader programs” are parts of “tasks”—they are not engines that *receive* tasks. More specifically, each thread (of a task) may execute a shader program. The '381 patent states that “[t]o perform a single graphics-processing task, a GPU may

execute a plurality of different threads (sequence of instructions). *Each thread may comprise a shader program, such as a geometry shader, a pixel shader, a vertex shader, or the like.* Ex. 1001, 1:47-52. The '381 patent further states, “[t]o efficiently execute the *various threads (e.g., shader programs)* of a single graphics-processing task, the GPU includes an array of processing elements, called a shader core.” *Id.*, 1:58-61.

C. Simultaneous Multi-Tasking

53. As distinguished from single-threading and multi-threading described above, the '381 patent introduced the ability to run tasks at the same time by partitioning the shader core (element 214 in Fig. 2 above) to run separate tasks in parallel. The '381 patent discloses that “[t]o enable shader core 214 to (substantially) simultaneously process tasks from the plurality of virtual engines of command processor 230, processing elements 220 of shader core 214 are partitioned in space and/or time.” Ex. 1001, 8:3-7. Further, the “spatial partitioning” of the shader core was implemented, in part, by organizing the shader core into multiple data-parallel subsets of processing elements. *Id.*, 8:8-10.

54. By sharing available shader resources, the '381 patent explains how the plurality of engines serve to improve utilization of the GPU relative to conventional approaches which included a single path to the GPU's execution resources (e.g., the shader core). For example, the '381 patent discloses that “[a]s illustrated in FIG. 3,

command processor 230 includes a plurality of virtual engines that share resources of GPU 110.” *Id.*, 6:67-7:2. Moreover, the ’381 patent discloses that “[s]haring the resources of the GPU to provide a plurality of virtual engines in accordance with embodiments of the present invention improves the use of the GPU resources, especially on large chips.” *Id.*, 5:29-32.

VIII. DESCRIPTION OF ASSERTED ART

A. Wilt (U.S. Patent No. 8,151,095)

55. Wilt (Ex. 1004) is directed to improvements in utilizing context data by processing units in a parallel processing subsystem. Specifically, Wilt explains that conventional systems may include multiple parallel processing units (PPUs) “that may be configured to beneficially provide a high volume of computational throughput that is impractical to achieve with a single processing unit.” Ex. 1004, 1:11-14.

56. Figure 1 shows an overview of the system, with the parallel processing subsystem at issue here highlighted:

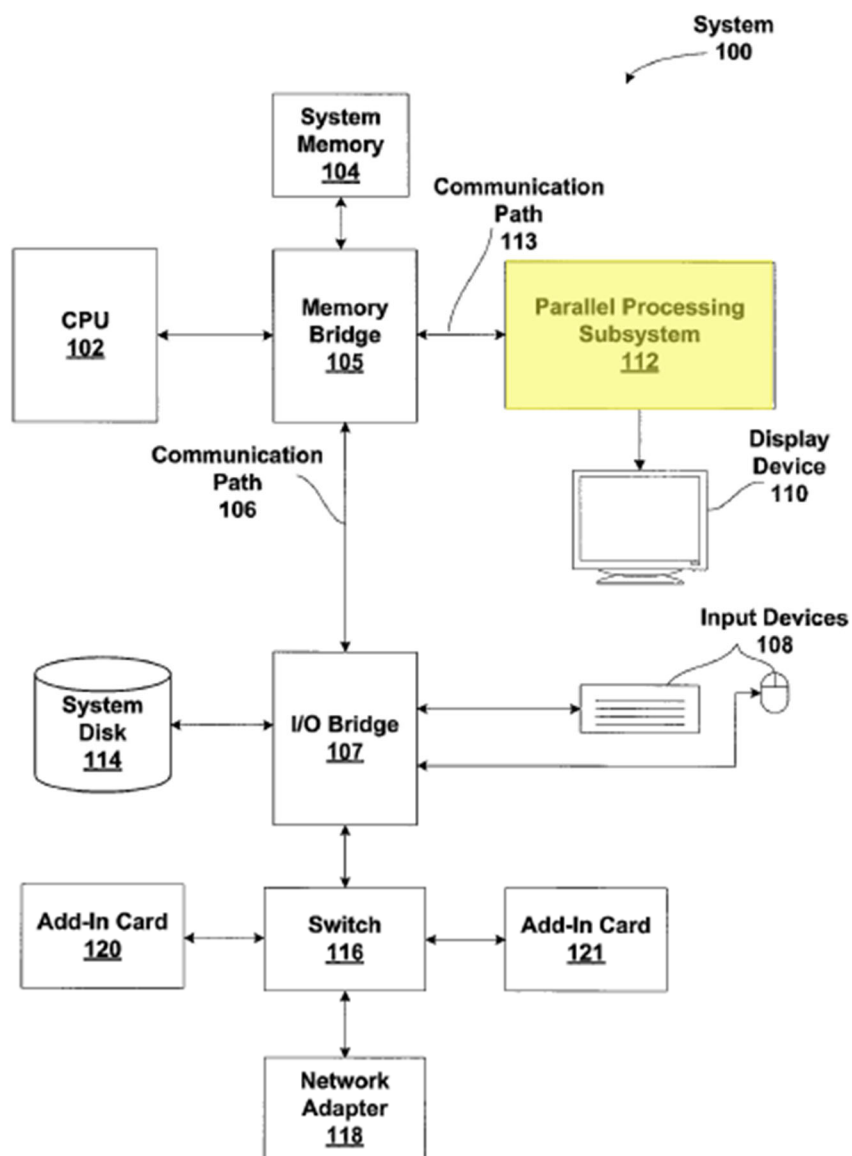


Figure 1

Id., Fig. 1 (annotated). Figure 2 provides more detail on the parallel processing subsystem, including the multiple parallel processing units 220:

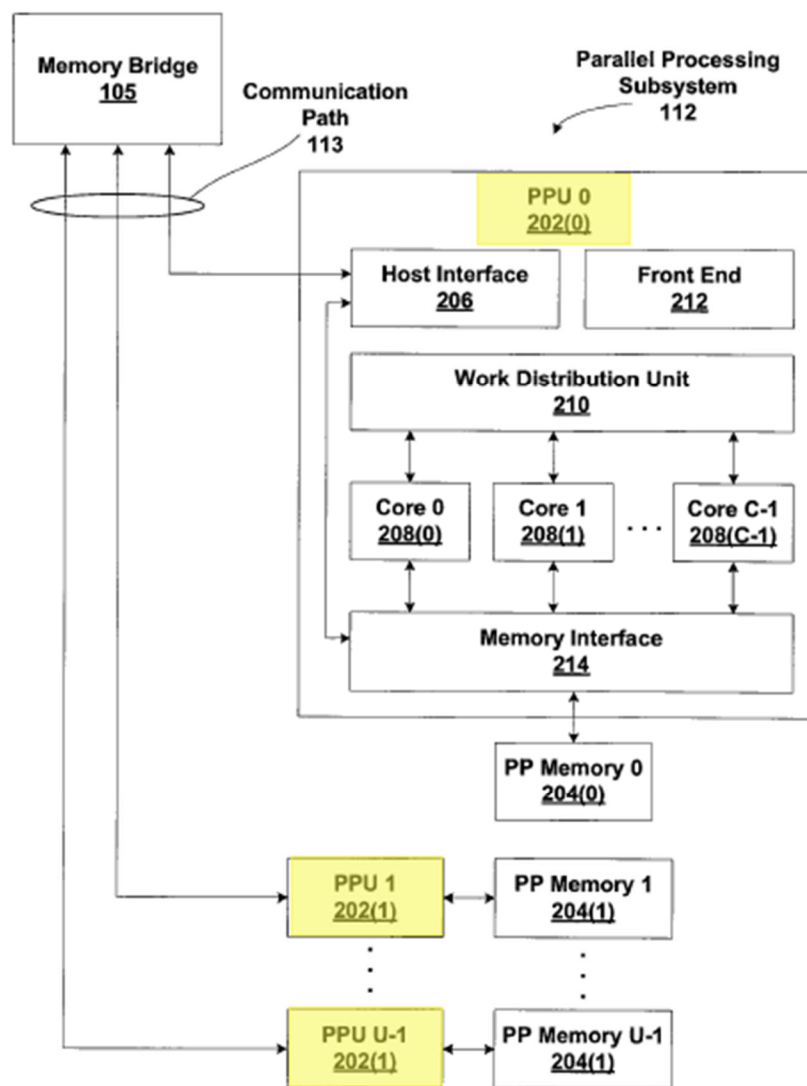


Figure 2

Id., Fig. 2 (annotated). Figure 3 is further detail on the PPU 202:

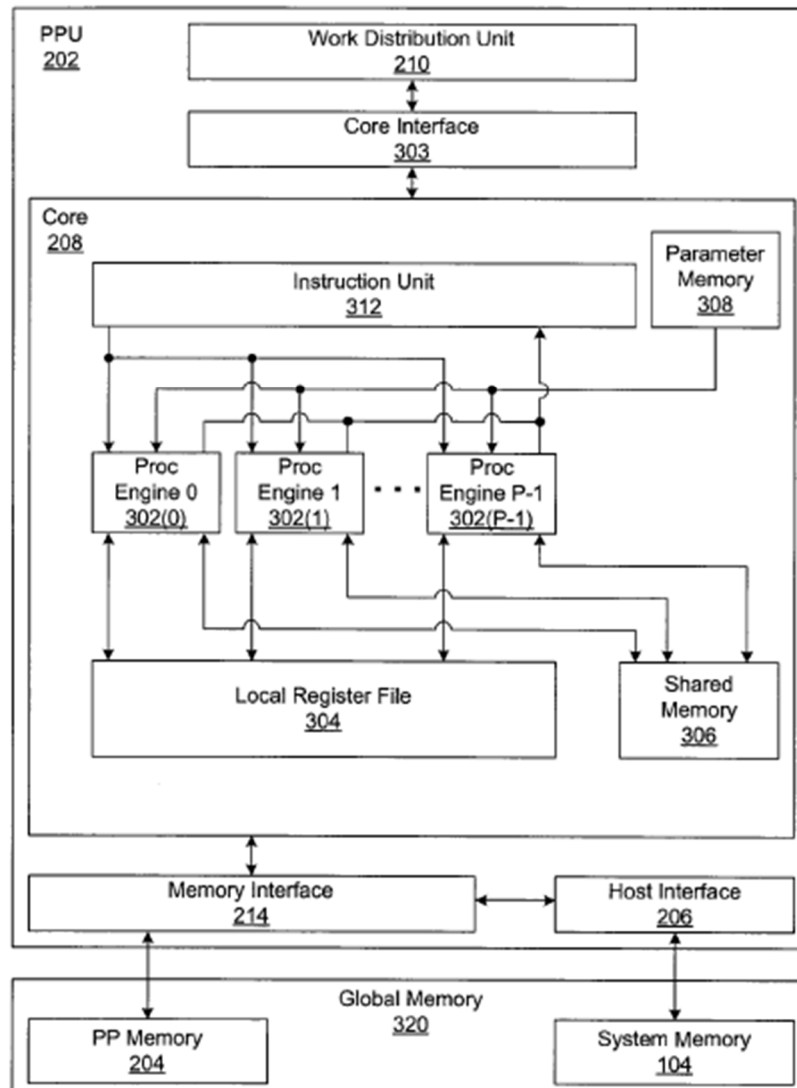


Figure 3

Id., Fig. 3.

B. Jiao (U.S. Patent No. 8,319,774)

57. Jiao (Ex. 1005) is directed to a graphics processing system that is intended to address challenges in updating constants in a constant buffer, and the associated latencies encountered during a constant flush operation and the loading of new constant data.

58. Figure 1 of Jiao shows the GPU in the context of the larger architecture:

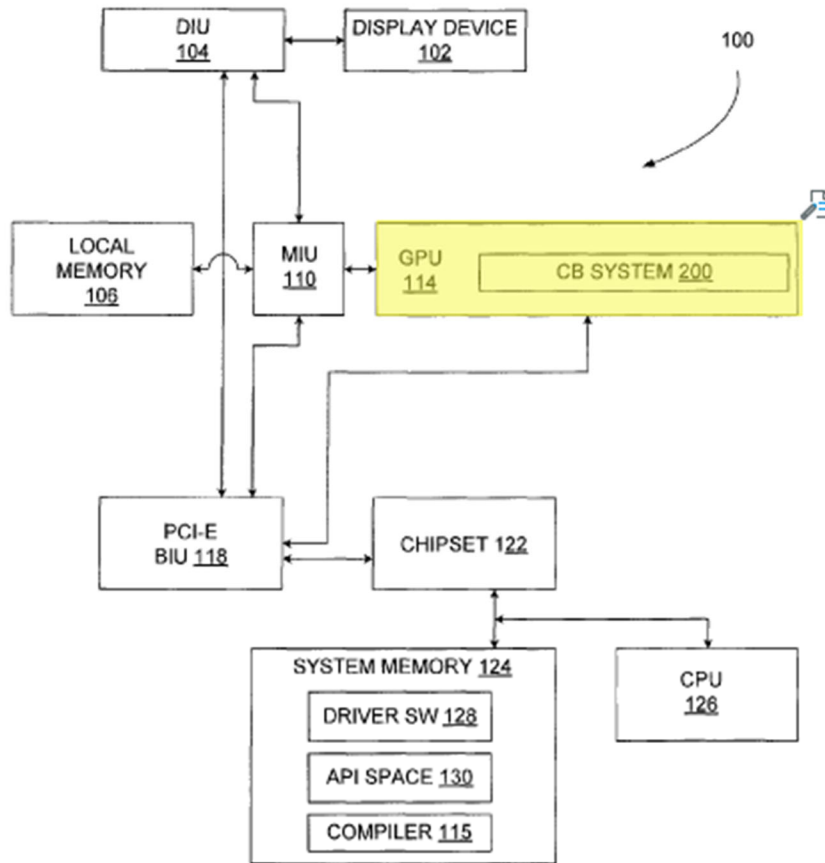


FIG. 1

Ex. 1005, Fig. 1 (annotated).

59. Figure 2 of Jiao is a high level diagram of the graphics processor 202 within GPU 114:

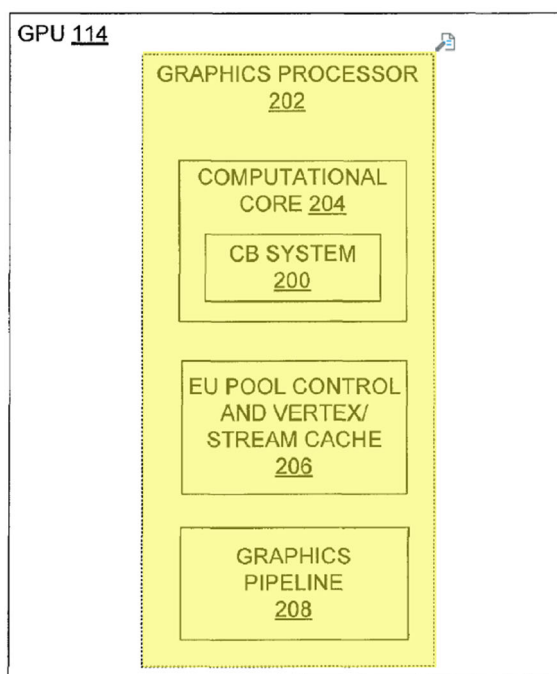


FIG. 2

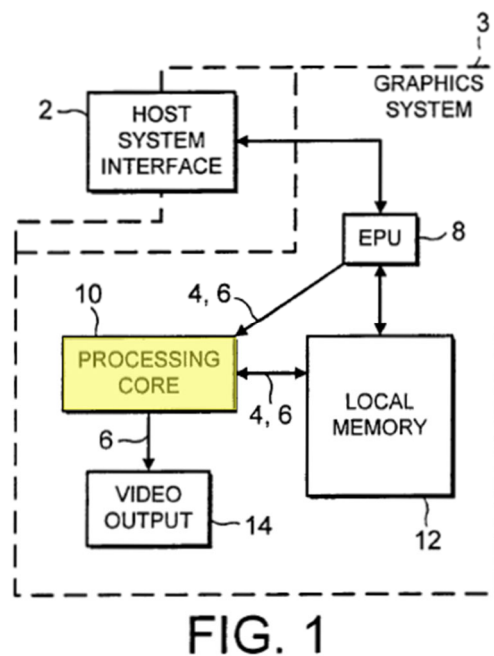
Id., Fig. 2 (annotated).

60. Figure 4 gives more detail of graphics processor 202. The key components are the Execution Units (EUs) 420 and EU Pool 414, the shader programs 304, 306, and 310, the EU Pool control 206, and the command stream processor 406.

C. Stuttard (International Publication No. WO 00/62182)

61. Stuttard (Ex. 1006) discloses an allegedly improved “single-instruction, multiple data” (“SIMD”) architecture that divides the processing elements into processing blocks. Ex. 1006, 2:1-24.

62. Figure 1 gives an overview of the architecture of Stuttard:



Ex. 1006, Fig. 1. The graphics system interfaces with a host system (not shown) via the host system interface 2. The graphics system 3 includes an embedded processing unit (EPU) 8 “for controlling the overall function of the graphics processor and for interfacing with the host system.” *Id.*, 4:26-30. Graphics processing takes place in the processing core 10. *Id.*, 7:25-27.

63. The processing core is shown in detail in Figure 3:

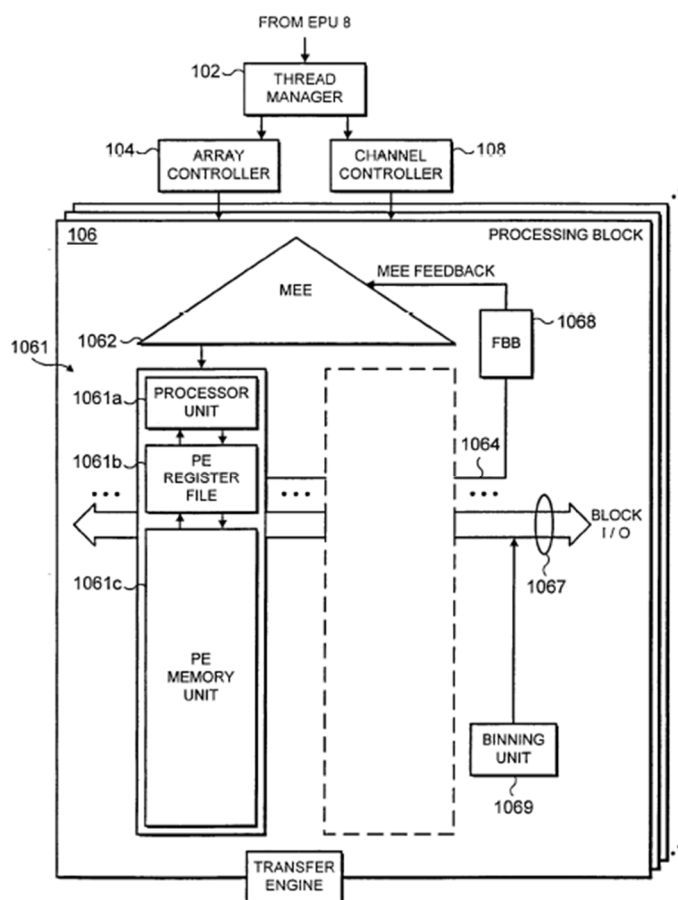


FIG. 3

Ex. 1006, Fig. 3. “The thread manager 102 is connected to receive control signals from the EPU 8. The control signals inform the thread manager as to when instructions are to be fetched and where the instructions are to be found. The thread manager 102 is connected to provide these instructions to the array controller 104 and to the channel controller 108. The array and channel controllers 104 and 108 are connected to transfer control signals to the processing blocks 106 dependent upon the received instructions.” *Id.*, 8:17-27. The processing core includes processing

blocks 106, which comprise an array 1061 of processor elements (PEs), a mathematical expression evaluator (MEE), and a binning unit. *Id.*, 8:15-9:3.

64. The entirety of Stuttard can only operate on one instruction at a time, and one “thread,” or “instruction stream” over a period of time, as explained more fully below with respect to Ground 3. *Id.*, 4:17-23, 4:25-29, 10:20-24, 11:33-35; Ex. 1003, ¶126 (referring to “thread scheduler 1025, which uses the status information to determine which *thread* should be active *at any one time*.”), 62 (“Stuttard’s thread scheduler recalculates which *thread* should be active”). All processing blocks 106 (Fig. 3) and all thread processors 1026 (Fig. 4) receive one instruction at a time out of the single instruction stream (thread) active over a given time period, and operate on different data, hence “single instruction, multiple data” (“SIMD”).

IX. EVALUATION OF PETITIONER’S PROPOSED GROUNDS

A. Ground 1: Wilt and the Knowledge of a POSITA Do Not Render Claims 15-20 of the ’381 Patent Obvious

65. In my opinion, claims 15-20 are not obvious in view of Wilt and the knowledge of a POSITA.

1. Wilt Fails To Disclose or Render Obvious All Challenged Claims

66. Claim 15 reads:

A computing system comprising:

a memory;

a first processing unit; and

a bus coupled to the memory and the first processing unit, wherein the first processing unit comprises:

a plurality of engines configured to receive, from a scheduling module associated with a second processing unit, a plurality of tasks and to load state data associated with each of the plurality of tasks; and

a shader core configured to receive the plurality of tasks from at least one of the plurality of engines and to execute a first task from the plurality of tasks while executing a second task from the plurality of tasks based on respective state data associated with each of the first and second tasks.

Ex. 1001, Claim 15. Claims 16-20 are dependent on Claim 15. *Id.*, Claims 16-20.

67. With respect to claimed plurality of engines and the claimed shader core, Dr. Pedram identifies two theories. The first, which I will refer to as the “Single PPU Theory,” Dr. Pedram opines that a single PPU is the claimed shader core (“Each parallel processing unit (PPU) 202 is the claimed ‘shader core...’”), and that “the work distribution unit 210, the front-end unit 212, and the core interface unit 303, all of which are within each PPU,” *i.e.*, these three items within one PPU, are the claimed plurality of engines. Ex. 1003, ¶¶71, 76.

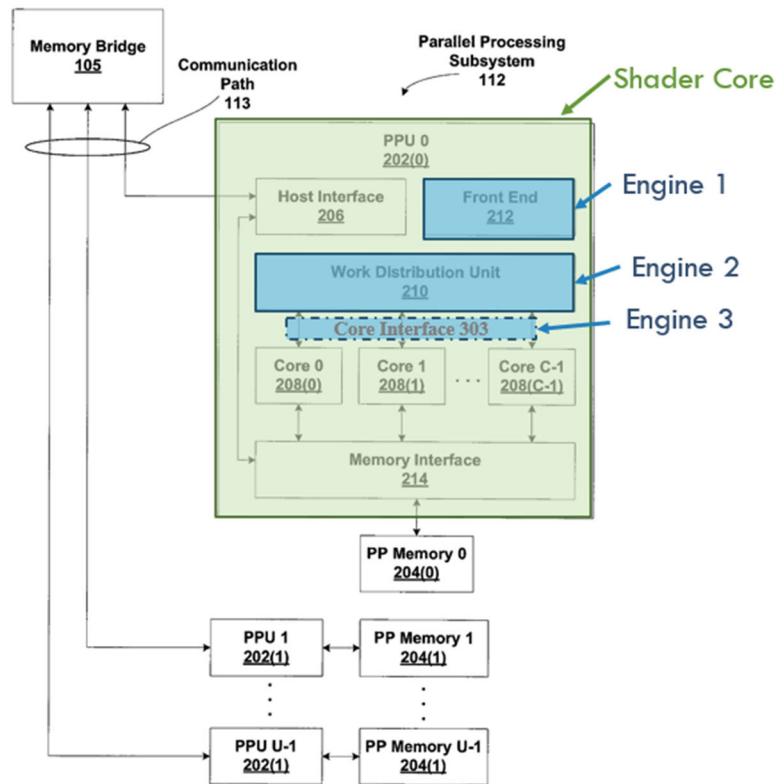
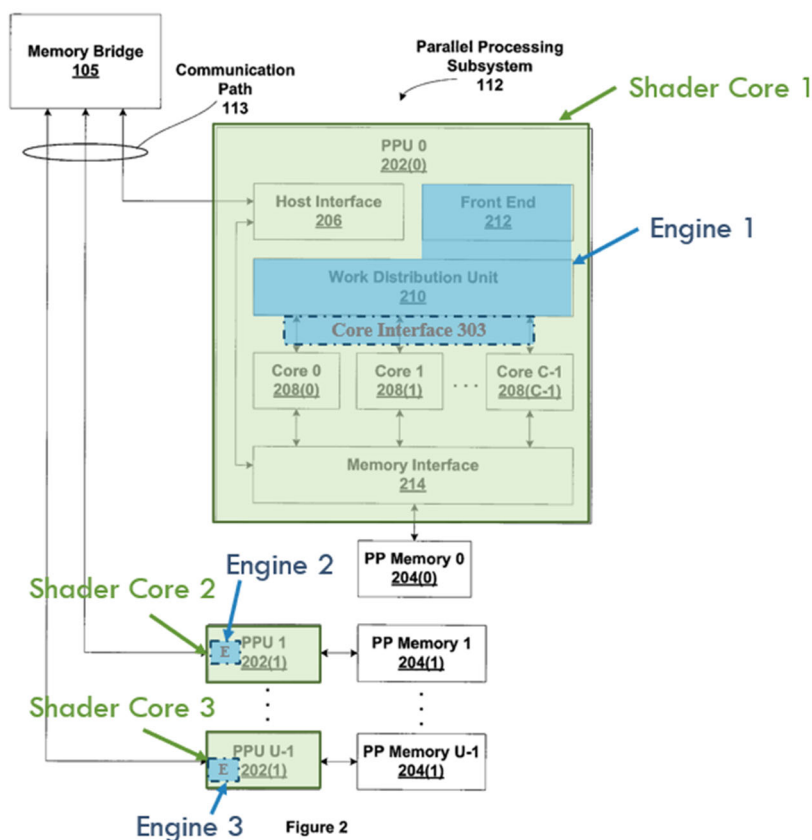


Figure 2

Ex. 1004, Fig. 2 (annotated with Dr. Pedram’s Single PPU theory).

68. Dr. Pedram’s second theory, which I’ll refer to as the “Multi-PPU Theory,” is that each of the alleged engines withing a single PPU in the Single PPU Theory together are one engine, and all PPUs together are the shader core. Ex. 1003, ¶¶73, 77.



Ex. 1004, Fig. 2 (annotated with Dr. Pedram's Multi-PPU theory).

a. Wilt Does Not Disclose Receiving and Simultaneously Executing Tasks

69. Dr. Pedram opines that the work distribution unit 210, the front-end unit 212, and the core interface unit 303, individually or collectively, receive and process tasks from software programs running on the host CPU. Ex. 1003, ¶¶71, 73. However, under both Single PPU and Multi-PPU theories, a close reading of Wilt shows that neither the alleged engines nor the alleged shader core receives the claimed tasks. Further, Dr. Pedram has not shown that the shader core is configured to execute a first task while executing a second task.

70. Wilt's "tasks" that are received and executed by cores 208 of the PPUs are not the tasks claimed by the '381 Patent. *See generally* Ex. 1004, 4:1-23. A POSITA would understand Wilt's cores 208 do not actually receive and execute tasks as claimed by the '381 Patent because the highest order item that Wilt's system receives are commands or command streams (*i.e.*, threads), where the '381 Patent defines a thread as a sequence of instructions, *i.e.*, a sequence of commands. Ex. 1001, 1:49-51. For example, Wilt discloses that CPU 102, the "master processor of system 100, . . . issues ***commands that control the operation of PPUs 202.***" Ex. 1004, 4:61-5:4. In addition to the above disclosure, Wilt also provides that "CPU 102 writes a ***stream of commands*** for each PPU 202." *Id.*, 4:61-5:4. Wilt also provides that "a driver program executing on CPU 102 of FIG. 1 writes ***commands*** defining the CTA to a pushbuffer (not explicitly shown) in memory (e.g., system memory 104), from which the ***commands are read by a PPU 202.***" *Id.*, 10:21-25 (emphasis added).

71. These commands issued by CPU do not ever become the claimed tasks as they make their way through Wilt's system, *i.e.*, they do not ever become consolidated. Instead, a POSITA would understand that whatever work is initiated by the CPU, that work is decomposed into smaller and smaller components until it finally reaches the shader core, which executes threads and instructions.

72. These commands are what is actually received by all three alleged engines (and the collective single engine under the Multi-PPU theory), and in turn, by the alleged shader core. Ex. 1004, 10:21-25; *see also id.*, 3:60-64 (“For example, commands related to processing tasks may be directed to a front end unit 212 while commands related to memory operations (e.g., reading from or writing to PP memory 204) may be directed to a memory interface 214.”). Wilt also provides that work distribution unit 210 “receives commands **defining** processing tasks from a front end unit 210.” *Id.*, 4:7-15. Additionally, “[i]n response to commands, core interface 303 loads the state parameters into core 208.” *Id.*, 10:32-34, 8:15-19 (“[C]ore interface 303 receives . . . commands defining how the data is to be processed.”).

73. Wilt’s commands are, at best, a subcomponent of a task as claimed by the '381 Patent. Wilt describes commands as “defining how the data is to be processed (e.g., what **program** is to be executed).” *Id.*, 8:17-19 (emphasis added). Meanwhile, the '381 Patent provides that “[t]o perform a single graphics-processing task, a GPU may execute a plurality of different threads (sequence of instructions),” wherein “[e]ach thread may comprise **a shader program**.” Ex. 1001, 1:49-52 (emphasis added). Therefore, in the '381 patent terminology, those command streams are threads. As a result, as also explained above in Section V.A, a claimed task comprises several threads, *e.g.*, several shader programs, which perform a

specific function. Therefore, a POSITA would understand that Wilt's commands are at best equivalent to a single shader program, or thread. Without more, Wilt does not disclose receiving tasks because Dr. Pedram has not sufficiently tied Wilt's commands and/or command streams to different tasks, nor could he. The receipt and execution of different threads is insufficient to disclose receipt and execution of different tasks, as the threads could have come from the same parent task.

74. Dr. Pedram has also failed to explain how its Single PPU or Multiple-PPU theory executes a first task while executing a second task. A POSITA would understand that having the capability to execute multiple threads at the same time does not mean that the system is configured to execute multiple tasks at the same time. *See* Section V.A, *supra*.

75. With respect to Dr. Pedram's single PPU theory, Wilt explains that "PPU 202 includes a core 208 (or multiple cores 208) configured to execute a large number of threads in parallel, where the term 'thread' refers to an instance of a particular program executing on a particular set of input data." Ex. 1005, 5:61-65. Wilt further explains that the core of a single PPU may execute a thread, a thread group, or cooperative thread array." *Id.*, 6:16-27. This does not satisfy the claims' requirement of executing two tasks because Dr. Pedram has not sufficiently tied threads to tasks. *See* Section V.A, *supra*. Nothing in Wilt discloses that the threads executed by a single PPU correspond to multiple tasks. Nor can it be assumed that

any two (or more) threads will correspond to two different tasks. A POSITA would understand that it is very difficult to determine whether the threads are decomposed from more than one task, or even determine which task a thread was decomposed from. Absent additional disclosure explaining which tasks certain threads correspond to, it cannot be assumed that any two (or more) threads will correspond to two different tasks. Therefore, processing multiple threads simultaneously is not sufficient to disclose the requirement to “execute a first task from the plurality of tasks while executing a second task.”

76. Moreover, even if pointing to a thread were sufficient, Dr. Pedram has not shown that the system is “configured to” execute threads that were decomposed from two separate parent tasks. Any sort of “happenstance” or “probability” based argument, which is not supported in the first instance, would be insufficient to meet the requirement of “configured to.” *See* Section V.C, *supra*.

b. The Multi-PPU Theory Does Not Disclose Or Render Obvious The Claims

77. In the Multi-PPU Theory, Dr. Pedram opines that within one PPU the combination of that PPU’s front end, work distribution unit, and core interface is a single “engine,” Ex. 1003, ¶¶73, 77, but no POSITA would take such a view. A POSITA would not understand the three items together to be one single item, *i.e.*, an engine. No POSITA would group them together as one of the claimed “engines.”

Therefore, this theory, which Dr. Pedram's multi-PPU theory hinges on, fails to disclose or render obvious claim 15, and subsequently all of the dependent claims.

78. Similarly, under the Multi-PPU, a POSITA would not understand the set of cores 208 within and across all PPUs to constitute a "shader core." Cores 208 across different PPUs are not related to each other in any way, and such a box-drawing crosses PPU boundaries, which has no logical basis. Nor do the PPUs communicate with each other; they act completely independently. Additionally, each alleged engine only transmits to the cores within its own PPU. No engine can send threads to cores 208 of other PPUs, demonstrating that it makes no sense to consider all cores across all PPUs as one entity, let alone one shader core, as such shader core resource sharing is one of the defining features of the shader core and its relationship with the engines. Ex. 1001, 7:64-8:43; Ex. 1005, Figs. 2-3. Thus, the theory fails for at least two-high level reasons: it makes no sense from a structural perspective to draw a box crossing PPU boundaries to encircle sub-elements of different PPUs, and it makes no sense from a operational perspective as the cores 208 across different PPUs do not receive data in the manner they do in the '381 Patent claims.

**c. Wilt's PPU(s) Do Not Receive Tasks From The
Alleged Engines Because The Alleged Engines Are
Inside The PPU(s)**

79. Under both the Single PPU Theory and the Multi-PPU Theory, in which either a single PPU or the combination of all PPUs are the alleged shader core, the alleged shader core encompasses the alleged engines, contrary to the claims. That is explicitly Dr. Pedram's opinion: "[A] person of ordinary skill in the art would have understood the work distribution unit 210, the front-end unit 212, and the core interface unit 303, *all of which are within each PPU*, to be 'plurality of engines' as recited in claim 15." Ex. 1003, ¶71.

80. Claim 15, by contrast, requires the shader core to receive tasks *from* engines, and a plain and ordinary reading of this limitation means the shader core cannot logically receive something from within itself and so the engines must be outside the shader core. Ex. 1001, Claim 15. A POSITA would understand such an interpretation renders redundant the separate "receive" steps, which require that both the engines receive tasks and the shader core receive tasks. The specification, of course, confirms that the engines and the shader core are separate:

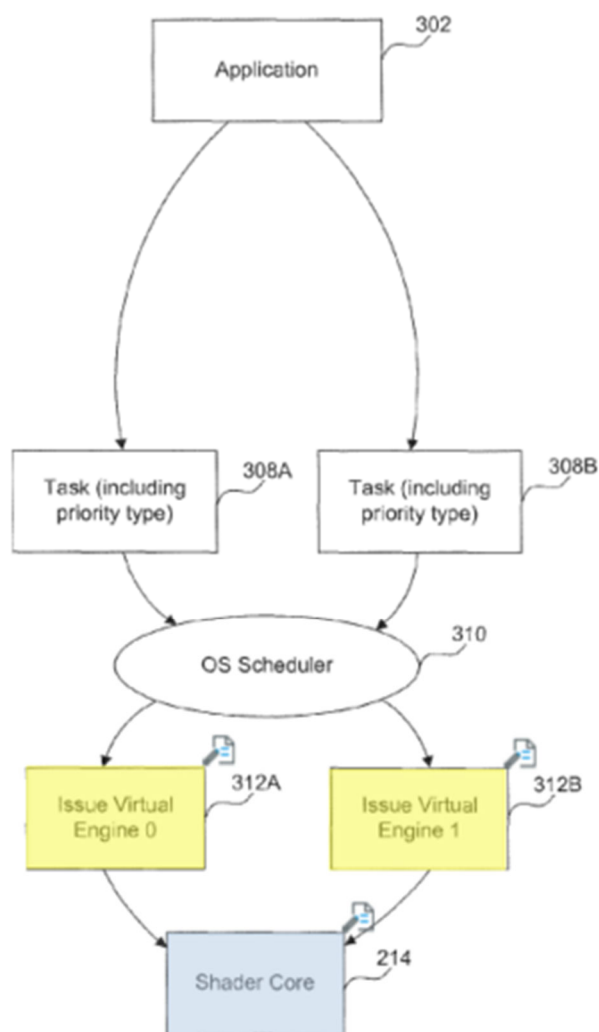


FIG. 3A

Id., Fig. 3A (annotated).

81. To the extent not clear by the claim language itself, I understand that prosecution disclaimer applies as well. Specifically, the applicant argued that because internal components of the alleged shader core in a prior art reference were

identified by the examiner as the claimed “engines,” the read failed because the alleged shader core cannot “receive the plurality of tasks from at least one of the plurality of engines” as required by the claims, because the “internal components of shader core 130 and shader core 130 are one and the same.” Ex. 1002 (’381 File History), 164-165. As a result, the examiner concluded the prior art did not disclose “the shader core configured to receive the plurality of tasks from at least one of the plurality of engines.” *Id.*, 158. Therefore, to allow the shader core to encompass the engines would be contrary to the applicant’s clear and unmistakable disavowal in the prosecution history.

2. Wilt Fails To Disclose or Render Obvious Claim 19

82. Claim 19 claims: The computing system of claim 15, wherein the first task comprises a graphics-processing task and the second task comprises a general-compute task. Ex. 1001, Claim 19. As a result of being dependent on Claim 15, this claim requires the shader core to execute a graphics-processing task *while* executing a general-compute task. *Id.*, Claim 15.

a. Dr. Pedram Does Not Opine That The Wilt Single PPU Theory Invalidates Claim 19

83. Dr. Pedram opines that Wilt discloses “PPUs for graphics processing and PPU for general-purpose processing.” Ex. 1003, ¶88. He does not opine that one PPU performs both types of processing. *Id.* Therefore, I understand that Dr.

Pedram's theory where a single PPU is the claimed shader core (the Single PPU theory) does not disclose or render obvious this claim.

84. Additionally, it is my opinion that Wilt does not disclose or render obvious a single PPU which executes a graphics-processing task while executing a general-compute task. Wilt discloses entire PPUs as being used for general-purpose computations. Ex. 1004, 4:55-58. Specifically, Wilt states: "In some embodiments, PP subsystem 112 may include one or more PPUs 202 that operate as graphics processors and one or more other PPUs 202 that are used for general-purpose computations." *Id.* Wilt does not disclose any embodiments where part of a PPU can be used for graphics processing and the other part used for general-purpose computations.

b. The Multiple PPU Theory Does Not Disclose Claim 19

85. Dr. Pedram's second theory, the Multi-PPU theory, is that each of the alleged engines (front end, work distribution, and core interface) within a single PPU together are one engine, and all PPUs together are the shader core. Ex. 1003, ¶77. As explained more in connection with claim 15, a POSITA would not consider the identified items to be a single engine. *See* Section IX.A.1.b, *supra*.

86. The Multi-PPU Theory does not disclose or render obvious claim 19. Wilt discloses only that its system is capable of executing "general-compute threads," but not that they are executed *while* executing graphics-processing threads.

Ex. 1004, 4:46-60. Specifically, Wilt states that some PPUs can be used as graphics processors and some can be used for general-purpose computations. Ex. 1004, 4:46-60; *see also id.*, 3:24-31. But this merely provides that Wilt is capable of executing general-purpose computations. In order for the alleged shader core execute a graphics-processing task while executing a general-compute task, Wilt's alleged shader core must receive both types of tasks from the alleged engines, which in turn, must receive both types of tasks from the scheduling module. But Dr. Pedram has not opined such, nor does Wilt disclose such. Ex. 1003, ¶88. Furthermore, Dr. Pedram has not shown that Wilt's system is actually configured to execute a graphics-processing task while executing general-compute task; occurring by happenstance is not enough to show the system is "configured" to do so. *See* Sections IX.A.1.a, *supra*.

87. Therefore, in my opinion, Wilt does not disclose or render obvious claim 19.

3. Wilt Fails To Disclose or Render Obvious Claim 20

88. Claim 20 claims: The computing system of claim 15, wherein the shader core is configured to receive the plurality of tasks based on a priority associated with each of the plurality of tasks. Ex. 1001, Claim 20. This claim

requires the shader core receive tasks based on the priority associated with an individual task. *Id.*

89. Dr. Pedram opines that Wilt discloses context swapping using a PPU context stack. Ex. 1003, ¶89 (citing Ex. 1004, Abstract, 12:51-63). However, Wilt does not describe its process as “swapping” contexts. Nor does Wilt’s disclosure of PPU contexts come nowhere near disclosing task priority at all, much less that the alleged shader core is configured to receive the plurality of tasks based on a priority associated with each of the plurality of tasks. Wilt describes its PPU context as associated with a *CPU thread*, not a task that is executing by the alleged shader core. Ex. 1004, 12:53-56 (“In step 624, the PPU driver associates a popped PPU context, if any, with the CPU thread.”); *see also id.*, 12:64-65 (“In sum, ***a technique for enabling CPU threads*** to use one or more existing PPU contexts is disclosed.”) (emphasis added). The CPU is not Dr. Pedram’s alleged shader core, and thus Wilt’s disclosure of PPU contexts (which applies only to the CPU) does not apply to the claimed shader core for this other reason. Per Figs. 1-2 of Wilt, CPU 102 is a separate component from Parallel Processing Subsystem 112, which Dr. Pedram’s alleged shader cores (under both Single PPU and Multi-PPU theories) are part of. *Id.*, Figs. 1-2.

90. Dr. Pedram opines that context swapping is “necessitated by tasks of different priority, such that a task with a higher priority, aided by pushing context to

and popping context from a stack, can pause the execution of a task with a lower priority.” Ex. 1003, ¶89. But that does not mean the opposite is true, that context swapping *requires* having tasks of different priorities. Context swapping is a technique used when switching the processing of one thread to another thread. Context swapping preserves the state of the thread being paused so that when the thread resumes execution, it can pick up where it left off when it stopped. Therefore, context swapping has nothing to do with task priority, not to mention mandating it.

91. Further, there are several reasons a system would context swap that are unrelated to task priority, or even thread priority. One example is if a thread needs various I/O resources in a system that aren’t ready, the thread can be put on hold while a new thread begins executing. Context swapping helps storing the thread’s current state so that when it is ready to execute, it can do so where it was stopped instead of starting at the very beginning. Another example is if a thread process is interrupted, the status of the thread process is using context swapping. After the interrupt is resolved, the thread process switches from a wait to a ready state so as to resume its execution later at the very same point at which the interrupt occurred.

92. Furthermore, Dr. Pedram has not shown that Wilt’s system is actually configured to receive tasks based on a priority associated with each task; occurring by happenstance is not enough to show the system is “configured” to do so. *See* Sections IX.A.1.a, *supra*.

93. Therefore, in my opinion, Wilt does not disclose or render obvious claim 20.

B. Ground 2: Jiao and the Knowledge of A POSITA Do Not Render Claims 15-20 of the '381 Patent Obvious

94. In my opinion, claims 15-20 are not obvious in view of Jiao and the knowledge of a POSITA.

1. Jiao Fails To Disclose or Render Obvious All Challenged Claims

95. Claim 15 reads:

A computing system comprising:

a memory;

a first processing unit; and

a bus coupled to the memory and the first processing unit, wherein the first processing unit comprises:

a plurality of engines configured to receive, from a scheduling module associated with a second processing unit, a plurality of tasks and to load state data associated with each of the plurality of tasks; and

a shader core configured to receive the plurality of tasks from at least one of the plurality of engines and to execute a first task from the plurality of tasks while executing a second task from the plurality of tasks based on respective state data associated with each of the first and second tasks.

Ex. 1001, Claim 15. Claims 16-20 are dependent on Claim 15. *Id.*, Claims 16-20.

**a. Jiao’s Shaders Do Not Disclose Or Render Obvious
The “Plurality Of Engines”**

**i. Jiao’s Shaders Do Not Receive Tasks, Or Even
Threads Or Instructions**

96. Dr. Pedram opines that Jiao’s shaders, vertex shader 304, geometry shader 306, and pixel shader 310, are the alleged engines. Ex. 1003, ¶¶97. But Dr. Pedram also admits that these shaders receive graphics *data* (e.g., graphics primitives, which are shapes) to produce pixel data. *Id.* Graphics data, including graphics primitives, are not tasks.

97. Dr. Pedram claims that Jiao’s shaders receive tasks from driver software 128. Ex. 1003, ¶¶99-100. While Jiao discloses that driver software 128 “communicates instruction sets or commands,” it never provides that those instruction sets or commands are provided to the alleged engines, Jiao’s shaders. Ex. 1005, 5:17-23. All Jiao discloses is that those instruction sets or commands are provided “registers in GPU 114.” *Id.* GPU 114 encompasses the entirety of Jiao’s system. *Id.*, Fig. 2. Dr. Pedram has not explained how these alleged tasks are received by the alleged engines or the alleged shader core once they reach the registers in GPU 114. Ex. 1003, ¶¶100, 104.

98. Jiao’s shaders only receive data. Jiao describes “primitives” and constants, called “vertex information” being passed to vertex shader 304: “Responsive to a command from the driver software 128, such as to draw a triangle,

vertex information is passed to vertex shader 304 in the computational core 204 to implement vertex transformations.” *Id.*, 7:57-60. This means that when driver software (not one of the alleged engines) receives a command, it responds to that command by sending vertex information (not tasks) to the vertex shader. *Id.* This does not state that the vertex shader receives commands.

99. Thereafter, the data is passed from vertex shader to geometry shader, and then to eventually to pixel shader. “[V]ertex shader 304 passes *data* to the geometry shader 306,” which “receives, as inputs, vertices for a full primitive. The geometry shader 306 receives, as inputs, vertices for a full primitive,” and “also receives constants.” *Id.*, 6:28-32 (emphasis added). “[G]eometry shader 306 then outputs information to a rasterizer 308 The pixel shader 310 is invoked for each pixel covered by the primitive,” and “also receives constants.” *Id.*, 6:34-44, 8:7-10 (“The Z-values are passed back to the pixel shader 310.”). Thus, in each instance, Jiao’s shaders are receiving *data*, not tasks.

100. Dr. Pedram’s theory seems to be that the shaders must be configured somehow, so they must receive tasks. But that is insufficient for several reasons. First, Jiao does not describe how the shaders receive their configuration, let alone that they receive them from a scheduling module. Further, there are numerous ways to configure a shader program, and not all of them require transmission of tasks to the shaders via a scheduling module.

101. Jiao further states that the tasks that are run by the shader core are *generated* in the first instance by the shaders. They are not received by the shaders, further confirming what the rest of the disclosure also states. For example, Column 6 of Jiao describes the shaders receiving data, and then follows that discussion with the statement “[h]aving described components that generate tasks for execution units in the computational core 204...” *Id.*, 6:49-52. The previous paragraphs were about the operation of the shaders, not any other element that generate tasks and send them to the shaders. *See also id.*, 9:35-38 (“As various shader tasks are generated by the vertex shader 304, geometry shader 306, and pixel shader 310, they are delivered to the respective execution units 420 to be carried out.”).

ii. Jiao’s Thread Slots Are Not Engines

102. Dr. Pedram’s alternative theory regarding thread slots as being included in the claimed plurality of engines also fails. Ex. 1003, ¶98. Jiao’s only disclosure regarding thread slots provides, in accord with their name, that “EU pool control unit 206 assigns *a thread* to one of the empty thread slots.” Ex. 1005, 9:18-21. Thus, each thread slot, at best, receives only one thread. Otherwise, EUP could assign threads to any thread slots, not merely the empty ones. Further, the storage of threads does not disclose the storage of multiple tasks because Dr. Pedram has not sufficiently shown that the threads were decomposed from multiple tasks.

103. Additionally, the thread slots are within the execution units 420 of EU pool 414, the alleged shader core. This means that the alleged shader core (EUP 414) cannot receive anything from the thread slots because the thread slots are internal components to the alleged shader core. As explained in Section IX.A.1.c, such a structure conflicts with the claim language.

iii. Jiao's Shaders Are Part Of The Shader Core

104. The alleged shader core is EU Pool 414. Ex. 1003, ¶110. But Jiao's shader programs are also within EUP 414, in contradiction to the claim language requiring they be separate. While the shaders appear to be outside the EUP 414 in Figure 4, Jiao clarifies that the shader programs are actually located inside EUP 414, despite the figure: "In general, an embodiment of the graphics processor 202 utilizes programmable vertex 304, geometry 306, and pixel shaders 310. *Rather than* implementing the functions or operations of these components as *separate fixed function units* with different designs and instruction sets, *the operations are instead executed by the pool 414* of execution units 420a, 420b . . . 420m with a unified instruction set."). Ex. 1005, 9:25-31. This disclaimed structure means that the alleged shader core cannot receive anything from the alleged engines. See Section IX.A.1.c, *supra*.

iv. Jiao's Shaders Are Shader Programs, Not Engines

105. Jiao's vertex, geometry, and pixel shaders are *shader programs* not engines nor even component parts of the claimed first processing unit. For example, Jiao states: "Various types of shader programs can be executed or mapped to the computational core 204, including the vertex shader 304, geometry shader 306, and pixel shader 310." Ex. 1005, 6:59-62. Consistent with the '381 Patent's disclosure, a POSITA would understand that shader programs are components of tasks, which are distinct from the claimed engines. *See* Ex. 1001, 1:51-66; 8:33-38 (a single graphics processing task typically comprises various shader programs). Jiao further distinguishes its shaders as being programmable, and describes them as having "operations" that are not "separate fixed function units with different designs and instruction sets." Ex. 1005, 9:26-31. This is evidence that the shaders are components of tasks, because tasks are decomposed into threads, which are composed of instructions. A POSITA would understand operations to be threads or instructions.

106. Further, Jiao's shader programs are *not* launched/issued like the '381 Patent's shader programs because Jiao discloses that the operations of the shader programs "are *executed* by the pool 414 of execution units." *Id.*; *see also id.*, 2:16-20 (describing shader programs as "running" on a computational core). The '381

Patent also describes shader programs as being executed by the shader core. Ex. 1001, 1:58-59 (“[t]o efficiently execute the various threads (e.g., shader programs) of a single graphics-processing task”), 8:64 (“when data is needed by any of these components to execute a shader program”).

107. Jiao’s shader programs are not engines because their inputs and outputs are connected to each other in a serial fashion. The geometry shader must finish executing, and provide its output as an input to the vertex shader before the vertex shader can do anything. Similarly, the vertex shader must finish executing and provide its input to the pixel shader before the pixel shader can do anything. Jiao describes them as being part of a “graphics pipeline,” where data is first passed to the vertex shader, then the geometry shader, and lastly the pixel shader. Ex. 1005, 6:10-44, Fig. 3. Therefore, they do not independently receive tasks from a scheduling module.

b. Jiao’s Alleged Scheduling Module Is Associated With The First Processing Unit

108. Dr. Pedram’s alleged engines (shader programs) and the alleged scheduling module (EU Pool control unit 206) are part of the same GPU 114. Ex. 1005, Fig. 2. This is contrary to the ID’s construction of the term “associated with,” which requires that the scheduling module *not be in a first processor*. ID, 9. Additionally, Dr. Pedram’s alleged first processing unit (computational core 204)

and second processing unit (command stream processor 406) are also within GPU 114. *Id.*, Figs. 2-4. However, EUP control unit 206 is not “associated with” the second processing unit because it is a separate component from command stream processor 406, and Jiao discloses only that command stream processor can send data to EUP control unit. *Id.*, 7:19-20 (“The command stream processor 406 provides triangle vertex indices to the EU pool control unit 206.”), Fig. 4. If this satisfied the claim requirement of being “associated with,” then EUP control unit would also be associated with the first processing unit given that EUP control unit also sends data to the shaders, which are components of the first processing unit. *Id.*, Fig. 4 (arrow notating sending “shader inputs”). However, the applicant distinguished prior art by arguing that the claimed scheduling module cannot be associated with the same processing unit that the claimed engines are associated with – namely, the claimed first processing unit. Ex. 1002, 182-183. The applicant explained that prior art did not disclose the claimed invention because “[t]hread scheduler 210 and the hardware units are not associated with different processing; units; rather, they are associated with the same processing unit—namely shader core 130.” *Id.*

c. Jiao’s Alleged Shader Core Does Not Receive “The Plurality Of Tasks”

109. The alleged tasks executed by the shader core are the ones generated in the first instance by the shaders. Ex. 1003, ¶110 (citing Ex. 1005, 9:35-38 (“As

various shader tasks *are generated by* the vertex shader 304, geometry shader 306, and pixel shader 310, they are delivered to the respective execution units 420 to be carried out.”)).

110. Based on the claim language, which refers to the tasks received by the shader core as “the” plurality of tasks, a POSITA would understand that the tasks received by the shader core must be the same tasks received earlier by the engines, even if the tasks have been decomposed into threads. Allowing for the tasks to be received in the first instance by the shader core (and not being received by the engines) is improper for the following reasons, even if the ’381 Patent specification describes some tasks as undergoing pre-processing through input logic.

111. As seen in Figure 2 of the ’381 Patent, not all tasks undergo pre-processing by the input logic, *e.g.*, the tasks from the background engines and the real-time low-latency engine go straight to the arbitration logic. Ex. 1001, Fig. 2.

112. The input logic does not “generate” the task, as the shaders in Jiao do, nor do they “process” them. Instead, “[i]nput logic arbitrates which tasks are issued to shader core 214.” Ex. 1001, 7:20-21. For example, components of input logic identifies shader programs and data relevant to the tasks: “[v]ertex analyzer 208 identifies shader programs associated with a graphics-processing and/or general-compute task and schedules when each shader program can be launched in shader core 214 based on input and output data that will be available” and “[s]can converter

212 traverses the primitives to determine pixels to be processed by shader core 214.” *Id.*, 7:42-46, 7:57-58. Meanwhile, arbitration logic merely provides tasks to the shader core. *Id.*, 7:32-33, 7:60-62. None of these components generate tasks, nor do they modify or process tasks.

113. It has not been shown, and it would not make sense, that the tasks generated by the shaders are merely modified versions of any alleged tasks received by the shaders (which do not exist in Jiao in the first case, as explained above). For example, even if Dr. Pedram were correct that the shaders received configuration instructions from a scheduling module, which it has not shown, once configured, the shaders receive primitives and, on the basis of those primitives that did not even exist when the shaders were configured, the shaders generate tasks. Those tasks are not and cannot be merely variations of the configuration instructions, certainly not in any way analogous to the examples of the specification of the '381 Patent.

114. Even if pointing to the generated shader tasks were sufficient, Dr. Pedram has not shown that the system is “configured to” execute shader tasks that were decomposed from two separate parent tasks. Any sort of “happenstance” or “probability” based argument, which is not supported in the first instance, would be insufficient to meet the requirement of “configured to.” *See* Section IX.A.1.a, *supra*.

d. Jiao Does Not Disclose Receiving and Simultaneously Executing Tasks

115. To the extent that Jiao's shader programs generate anything besides data, they generate threads at best, not tasks. Notably, as Dr. Pedram admits, Jiao refers to the tasks generated by the shaders as both tasks and threads. Ex. 1003, ¶103 (citing Ex. 1005, 8:65-67, 9:10-13). While Dr. Pedram opines the "tasks or threads" of Jiao are like tasks in the '381 Patent, that is incorrect; they are what the '381 Patent would call threads. For example, Jiao discloses that the "tasks" generated by the shaders are assigned to "available threads" within the various execution units 420, including in "thread slots." Ex. 1005, 9:39-54. "The EU pool control unit 206 particularly knows which threads have been assigned tasks and are occupied." *Id.* Thus, Jiao's reference to "task" does not comport with the claimed task, and instead to threads. Dr. Pedram has not sufficiently tied these threads to the claimed task.

116. Additionally, the word "task" as used in Jiao does not refer to what the '381 Patent refers to as "tasks," for the additional reason that in Jiao, multiple so-called "tasks" can be associated with a single context, indicating they are actually what the '381 Patent would call threads. The '381 Patent describes each thread is associated with *a* set of locally stored state data (such as texture handles, shader constants, etc.), which is called a context. Ex. 1001, 1:53-57. Meanwhile, Jiao describes *multiple* of its tasks as having a single context. Ex. 1005, 13:52-53 ("the

execution unit 420 is ready to receive tasks that use the new constant context (712)”), 14:29-31 (“the counter 904 in the constant reference table 910 keeps track of the number of outstanding tasks for a given shader constant context”); 14:34-36 (“When the counter 904 reaches zero, the EU pool control logic 605 is notified that all tasks for the context have completed and the EU pool 414 is flushed.”). Based on this, a POSITA would understand that what Jiao calls “tasks” are actually threads.

2. Jiao Fails To Disclose or Render Obvious Claim 19

117. Claim 19 claims: The computing system of claim 15, wherein the first task comprises a graphics-processing task and the second task comprises a general-compute task. Ex. 1001, Claim 19. As a result of being dependent on Claim 15, this claim requires the shader core to execute a graphics-processing task *while* executing a general-compute task. *Id.*, Claim 15.

118. Dr. Pedram relies on pure obviousness to opine that Jiao discloses claim 19. Ex. 1003, ¶117. Dr. Pedram has not pointed to a single disclosure in Jiao which provides that Jiao’s system can even execute general-compute tasks, much less executing them while executing a graphics-processing task. *See id.*

119. Jiao’s shaders do not send any general-compute tasks because its shaders are specifically tailored for graphics processing. Jiao’s shaders are specific to geometry, vertex, and shading. A POSITA would understand that any purported

tasks generated by Jiao's shaders are specific to 3D graphics processing. Nor would a POSITA be motivated to change Jiao's shaders to handle general-compute tasks given their specialized function.

120. I understand that the ALJ in the 1318 Investigation found that Dr. Pedram failed to prove Jiao rendered this claim obvious under the same arguments presented here because Dr. Pedram failed to show that the EUs of Jiao could function in the way disclosed by claim 19, or could be made to do so with modification. Ex. 2003, 161-162.

121. Dr. Pedram has not identified any general-compute tasks. Dr. Pedram cites Jiao's statement that "arithmetic logic units and other units corresponding to functionality pertaining to instruction decoding, operand fetch, branch computation, instruction execution, shift/logic, load/store, among other processing functionality or logic." Ex. 1003, ¶117 (citing Ex. 1005, 11:4-8). However, Dr. Pedram has not explained how this disclosure relates to the ability to execute general-compute tasks, or why this functionality would not be provided for the purpose of executing graphics tasks. Nor does Jiao provide any explanation. Additionally, these operations identified by Dr. Pedram are performed by the EUDP 514. Ex. 1005, 11:4-8 ("The EUDP 514 comprises arithmetic logic units and other logic"). The claim requires that the claimed shader core processes the general-compute task. Ex. 1001, Claim 19. Further, given that Jiao discloses this functionality in EUDP

514, there is no reason a POSITA would modify EUP 414 (the alleged shader core), or the EU 420s within, to also have this functionality.

122. Dr. Pedram also alleges that a POSITA would find it obvious that programmable execution units can do general-compute tasks, but doesn't explain why. Ex. 1003, ¶117. Jiao's disclosure on this provides that instead of implementing the functions/operations of the graphics shaders as separate fixed function units, the graphics operations can be executed by EUP 414 with a unified instruction set, and each EU 420 is capable of multi-threaded operations. Ex. 1005, 9:27-35. Nothing regarding this disclosure provides that the unified instruction set includes the capability of executing general-compute tasks, nor does multi-threaded operations have anything to do with the type of tasks that can be executed.

123. Furthermore, Dr. Pedram has not proven that Jiao's system is actually configured to execute a graphics-processing task while executing general-compute task; occurring by happenstance is not enough to show the system is "configured" to do so. *See* Sections V.C, IX.B.1.c, *supra*.

124. Therefore, in my opinion, Jiao does not disclose or render obvious claim 19.

3. Jiao Fails To Disclose or Render Obvious Claim 20

125. Claim 20 claims: The computing system of claim 15, wherein the shader core is configured to receive the plurality of tasks based on a priority associated with each of the plurality of tasks. Ex. 1001, Claim 20. This claim requires the shader core receive tasks based on the priority associated with an individual task. *Id.*

126. Dr. Pedram opines that Jiao's reference to rendering contexts containing "a plurality of constants with a priority level" discloses this claim. Ex. 1003, ¶118. Dr. Pedram also opines that Jiao uses the priority level of the constants or "constant blocks" to put the constant blocks onto "fast" or "slow" access paths, the fast path being for high-priority constant data, and the slow path for low priority. *Id.*

127. However, constants are not tasks; they are merely data. Jiao confirms this understanding. For example, Jiao provides an example of a constant as being part of a 4x4 matrix used to transform vertices. Ex. 1005, 1:62-66; *see also id.*, 1:56-58 ("When rendering an object or group of objects for a given rendering context, a shader utilizes a constant data pool (e.g., context-specific constants stored in a constant buffer)."), 10:22-25 ("The API space 130 comprises multiple constant blocks 502a through 502n (e.g., sixteen (16) shown as one exemplary embodiment), each block comprising in one embodiment 4000 (4K) constants.").

128. Rendering contexts are constants. Jiao explains that “the rendering context comprises a plurality of constants.” *Id.*, 2:28-29. Further, Jiao does not disclose that these priorities impact, in any way, the priority of tasks being received by the shader core. Jiao states only that “[t]he priorities assigned to the constant buffer slots *assist hardware in best utilizing any dedicated constant data access paths/mechanisms*, if present.” *Id.*, 12:15-17 (emphasis added). Thus, the constant block priorities are unrelated to task priority.

129. Furthermore, Dr. Pedram has not shown Jiao’s system is actually configured to receive tasks based on a priority associated with each task; occurring by happenstance is not enough to show the system is “configured” to do so. *See* Sections V.C and IX.B.1.c, *supra*.

130. Therefore, in my opinion, Jiao does not disclose or render obvious claim 20.

C. Ground 3: Stuttard and the Knowledge of A POSITA Do Not Render Claims 15-20 of the ’381 Patent Obvious

131. In my opinion, claims 15-20 are not obvious in view of Stuttard and the knowledge of a POSITA.

1. Stuttard Fails To Disclose or Render Obvious All Challenged Claims

132. Claim 15 reads:

A computing system comprising:

a memory;

a first processing unit; and

a bus coupled to the memory and the first processing unit, wherein the first processing unit comprises:

a plurality of engines configured to receive, from a scheduling module associated with a second processing unit, a plurality of tasks and to load state data associated with each of the plurality of tasks; and

a shader core configured to receive the plurality of tasks from at least one of the plurality of engines and to execute a first task from the plurality of tasks while executing a second task from the plurality of tasks based on respective state data associated with each of the first and second tasks.

Ex. 1001, Claim 15. Claims 16-20 are dependent on Claim 15. *Id.*, Claims 16-20.

a. Stuttard Does Not Disclose Receiving and Simultaneously Executing Tasks

133. Dr. Pedram opines that either the instruction fetch unit 1023 and/or the thread scheduler 1025 can be the claimed scheduling module. Ex. 1003, ¶125. He alternatively identifies EPU 8 as a claimed scheduling module. *Id.* He opines that the binning unit 1069, the channel controller 108, the array controller 104, and components of the thread manager 102 (including the cache memory unit 1024, the thread processors 1026, and the status block 1030) constitute the claimed “plurality of engines.” *Id.*

i. Stuttard Receives and Executes Threads, Not Tasks

134. In my opinion, Stuttard does not disclose the claimed “a plurality of tasks.” None of Dr. Pedram’s alleged scheduling modules send the claimed tasks, nor do Dr. Pedram’s alleged engines receive the claimed tasks. At best, the alleged engines and shader core in Stuttard’s system receive and process only threads, which Dr. Pedram has not sufficiently tied to tasks, and in fact acknowledges that threads are not tasks. Ex. 1003, ¶102 (“Threads are not tasks.”). Thus, the execution of multiple threads is insufficient to disclose or render obvious the claimed shader core, which requires executing a first task while executing a second task.

135. First, Stuttard does not disclose the use of tasks as claimed by the ’381 Patent. In fact, Stuttard makes no disclosure of tasks at all, except to state generally that threads “work[] co-operatively with other threads to perform a task or tasks.” Ex. 1006, 10:10-12. Stuttard also provides definitions for multithreading and multitasking, but never mentions that its system is capable of doing either. *Id.*, 10:12-15. All of Stuttard’s disclosures regarding how its components work relate to *threads*. See generally *id.*

136. Dr. Pedram’s specific theories regarding the instruction fetch unit and/or thread scheduler, as well as EPU 8, as the claimed scheduling modules do not send the claimed tasks to the alleged engines. Ex. 1003, ¶125. Stuttard explains that

“[i]nstructions for *a thread* are fetched from local memory or the EPU 8 by the fetch unit 1023.” Ex. 1006, 11:12-14, 23:12-16 (emphasis added). This is all Stuttard says about instruction fetch unit. This indicates that the instruction fetch unit fetches are instructions. And further, as implied by its nomenclature, a POSITA would understand that instruction fetch unit fetches instructions. Because the instruction fetch unit fetches, at best, instructions corresponding to one thread, and nothing in Stuttard suggests that the thread fetched corresponds to a task, it does not send entire tasks to the alleged engines.

137. Similarly, the thread scheduler schedules *threads*. *Id.* 12:15-17 (“The status information is used by the thread scheduler 1025 to determine which thread should be active at any one time.”), 23:18-22 (“At a given time, only one thread is executing, and the scheduling of the time multiplexing between threads is determined by the dynamic conditions of the program execution. This scheduling is performed by a thread scheduler in the thread manager 102”); 23:35-24:1 (“The thread scheduler, when running, recalculates which thread should be active”). EPU is also capable of calculating which thread should be active, and also sends control signals to thread manager. *Id.*, 8:17-19, 25:19-24. Meanwhile, Stuttard never discloses that the thread scheduler or EPU sends tasks, or that the scheduled threads correspond to more than one task.

138. The selection of which thread should be active does not disclose alleged engines receiving anything, not to mention tasks, from a scheduling module. But even thread selection did involve receiving something, what is being received by the alleged engines are not tasks, but threads. Stuttard explains very clearly how thread scheduler and/or EPU operates if a scheduling trigger occurs: “First, it stops the active *thread* from running, and waits a cycle for any semaphore decrements to propagate.” *Id.*, 25:1-3. Then, thread scheduler activates *a thread* based on parameters should as the thread’s priority, whether the thread is yielding, etc. *Id.*, 25:5-17. Thus, both thread scheduler and EPU, at best, are capable of only activating one thread for execution. A POSITA would have no way to know, based on Stuttard’s disclosures, whether that thread corresponded to a task, or not.

139. EPU’s ability to send control signals is even further removed from the requirement of receiving tasks from a scheduling module because control signals are not even threads or instructions. Instead, control signals are signals that set up communication channels and control the flow of data. “The control signals inform the thread manager as to when instructions are to be fetched and where the instructions are to be found.” *Id.*, 8:19-21. Thus, they are directions sent by EPU to thread manager, and are not tasks.

140. Third, Stuttard discloses that the alleged plurality of engines only receive threads, not tasks. Dr. Pedram alleges that binning unit 1069, the channel

controller 108, the array controller 104, and components of the thread manager 102 (including the cache memory unit 1024, the thread processors 1026, and the status block 1030) constitute the claimed “plurality of engines.” Ex. 1003, ¶125. But none of these components receive tasks. To the extent that they receive anything, they receive threads or data. For example, the binning unit sorts primitive data, which are not tasks. Ex. 1006, 21:15-21. The channel controller and the array controller receive only instructions, not tasks. *Id.*, 26:3-7. Similarly, the cache memory unit “stor[es] instructions fetched for each thread.” *Id.*, 23:5-6. Meanwhile, Stuttard provides that each thread has its own thread processor, meaning that each thread processor receives a thread, not tasks. *Id.*, 25:26-29. And since Dr. Pedram never ties these threads to tasks, the threads received by the alleged engines cannot disclose the claimed plurality of tasks.

141. Fourth, the alleged shader core (Stuttard’s processing blocks) does not receive the claimed tasks, but threads. Even Dr. Pedram’s allegations are directed to threads being received and processed. Ex. 1003, ¶129-130 (“Stuttard clearly teaches its 8 parallel **threads**”; “multiple **threads** can be simultaneously active”; thread manager 102 loads status information associated with **threads** and sends them to processing blocks 106”). Stuttard provides that the processing elements within the processing blocks are “controlled to carry out a series of instructions in an instruction stream,” which “are known as ‘threads’.” Ex. 1006, 10:7-10. Further,

Stuttard explains that thread scheduler determines which *thread* should be active and being processed. *Id.*, 23:34-25:24. And since Dr. Pedram never ties these threads to tasks, the threads being processed cannot disclose the claimed plurality of tasks.

ii. Stuttard Does Not Simultaneously Execute Different Threads

142. Even if Stuttard disclosed the receipt of tasks (which it does not), Dr. Pedram has not alleged that the alleged shader core can execute a first task while executing a second task, as required by the claims. Dr. Pedram alleges that Stuttard discloses the storage of multiple threads, which can be associated with different tasks, and that multiple threads can be simultaneously active at the same time. Ex. 1003, ¶129 (citing Ex. 1006, 11:19-25 (“At any particular moment in time, each thread may be assigned to any one of a number of tasks.”)). However, this disclosure, at best, relates only to the *storage of threads in memory*, and not the execution of the threads. Stuttard contains no disclosure that these threads will each be executed simultaneously by the processing blocks 106, the alleged shader core. Thus, even if there was evidence that the simultaneously stored different threads originated from different tasks – and there is not – there is no evidence that those stored threads are executed at the same time.

143. Dr. Pedram’s theory relies on the unstated and unsupported proposition that the multiple threads in the cache correspond to threads in the thread processor 1026 which then correspond to the multiple processing blocks 106.

144. In the entirety of Figure 4, the same instruction stream is supplied to all processing blocks, as made clear in the paragraph immediately after the quote Dr. Pedram points to—“this particular example” clearly refers to the only “example” referenced, Figure 4:

 In the example shown in Figure 4, the thread manager includes one thread processor 1026 for each thread. The thread processors 1026 control the issuance of core instructions from the thread manager so as to maintain processing of simultaneously active program threads, so that each the processing blocks 106 can be active for as much time as possible.  In this particular example the same instruction stream is supplied to all of the processing blocks in the system.

Ex. 1006, 11:27-35. There is no discussion stating that the processing blocks could each correspond to different threads in the cache 1024.

145. Stuttard’s Abstract states that the effect of dividing of the processing elements into a “plurality of processing blocks” is specifically to allow them to be “operable to process respective groups of *data items*”—not threads. *Id.*, Abstract.

146. The “Summary of the invention” states that “there is provided a data processing apparatus comprising an array of processing elements, which are operable to process respective data items in accordance with *a common received instruction*, wherein the processing elements are operably divided into a plurality of processing blocks having at least one processing element, the processing blocks being operable to process respective groups of *data items*.” *Id.*, 2:25-33.

147. Dr. Pedram has not identified any disclosure in Stuttard teaching that it can execute multiple threads at the same time, much less multiple *tasks* at the same time. Stuttard cannot because Stuttard’s system is “a data processing apparatus comprising *a SIMD* (single instruction multiple data) array of processing elements, wherein the processing elements are operably divided into a plurality of processing blocks.” *Id.*, 2:17-23 (emphasis added). A SIMD is only capable of processing a single thread at any given moment in time. Because a thread cannot correspond to more than one task, to the extent Stuttard is interpreted as disclosing executing complete tasks at all (which Patent Owner disagrees with) is only capable of processing a single task at one time.

148. Stuttard discloses that its single SIMD array system is capable only of processing the same instruction across the entire array. *Id.*, 2:25-34 (all processing blocks operate on a “common received instruction”), 11:33-35 (the same thread “is supplied to all of the processing blocks in the system”). Stuttard’s system includes

a thread manager, which controls the entire array of processing blocks. *Id.*, 8:16-24, 10:5-17, Fig. 3. When describing how the thread manager operates, Stuttard provides: “At a given time, only one thread is executing, and the scheduling of the time multiplexing between threads is determined by the dynamic conditions of the program execution.” *Id.*, 23:18-21. Stuttard further provides that the thread scheduler 1025, of which there is only one, “determine[s] which thread should be active at any one time.” *Id.*, 12:15-18. This is reiterated in a section describing the function of the thread scheduler in more detail, entitled “Determining which thread should be active.” *Id.*, 23:34. The section explains that the thread scheduler will recalculate which thread should be active given certain circumstances. *Id.*, 23:34-25:17. All of these disclosures provide that Stuttard only has a single active thread at one point in time.

149. The entire array of processing elements within Stuttard’s system is controlled by the thread manager. The main purpose of the thread manager is to control which thread is currently active, and how to switch from one thread to another. Stuttard explains that explains that the thread manager allows for faster switching in response to external conditions. *Id.*, 10:33-11:1 (“In addition, the system can have a faster response time to external events. Assigning particular threads to wait on different external events, so that when an event happens, it can be handled immediately.”), 25:5-9 (“If the previously active thread is yielding, the

scheduler activates the highest priority READY thread, or the lowest priority thread if no thread is ready (since this will cause another immediate scheduling trigger).”). Stuttard’s detailed disclosure regarding the thread manager is focused on how to switch from one thread to another thread. *See generally id.*, 23:1-27:32.

150. Dr. Pedram identifies a disclosure describing thread processors which “control the issuance of core instructions from the thread manager so as to maintain processing of simultaneously active program threads, so that each the processing blocks 106 can be active for as much time as possible.” *Id.*, 11:29-33. This is directed at how the thread processors within the thread manager operate, and not the processing blocks. Further, the thread processors in Fig. 4 are contained in block 102, block 102 appears in Fig. 3 as separate from the processing blocks, so the thread processors are not processing blocks. *Id.*, Figs. 3-4. Fig. 4 shows an output of thread manager 102 labeled 1032, which are core instructions. *Id.* “Core instructions 1032 issued by the thread manager 102 are sent to the array controller 104 and the channel controller 108 (figure 3).” *Id.*, 12:20-22. If the instruction is intended for the processing blocks, then it goes to the array controller to be converted into PE microcode for execution on the processing elements of the processing blocks, otherwise the instruction is an external I/O access and goes to channel controller for access to the memory or other I/O channel.

151. The purpose of the thread processors “are to allow high level flow control to be performed for a thread (such as looping and conditional branches), and to assemble instructions to be issued to the array controller 104 and channel controller 108.” *Id.*, 26:3-7. This means that thread processors do not execute tasks (or threads associated with different parent tasks). Instead, the thread processors prepare threads for execution to ensure that other threads are *ready to be executed* when the active thread stalls. As further evidence that this disclosure does not disclose that the processing blocks can execute multiple tasks or even threads at one time, the following sentence indicates that “the same instruction stream is provided to all of the processing blocks in the system.” *Id.*, 11:33-35.

152. Dr. Pedram confuses threads and processing blocks. Stuttard does not disclose a system consisting of eight processing blocks and eight threads such that each of these eight threads is intended for a different processing block. Stuttard defines threads as “instruction streams for the processing blocks,” and that “the array of processing elements is controlled to carry out a series of instructions in an instruction stream.” *Id.*, 10:6-10. Stuttard then further defines threads. “There are several reasons for providing multiple threads in such a data processing architecture. The processing element array can be kept active, by processing another thread when the current active thread is halted. The threads can be assigned to any task as required. For example, by assigning a plurality of threads for handling data I/O

operations for transferring data to and from memory, these operations can be performed more efficiently, by overlapping I/O operations with processing operations. The latency of the memory I/O operations can effectively be masked from the system by the use of different threads.” *Id.*, 10:19-31. A POSITA would have understood that Stuttard was describing the *time-interleaved* processing of threads. “The processing element array can be kept active, by processing another thread when the current active thread is halted.” *Id.* Here thread is singular but applied to an entire processing element array, *i.e.*, all of the processing blocks. A POSITA would have understood Stuttard to state that all of the processing elements in the entire array (and thus all of the processing blocks) are processing the same instruction from the same thread. Furthermore, the entire processing element array (and thus all of the processing blocks) switches to processing a different thread when its current thread is halted, *e.g.*, due to the delay incurred by data I/O operations.

153. Dr. Pedram cites Stuttard’s Fig. 2(b) and Fig. 4 in support of his statement “since there are eight processing blocks in Stuttard system.” Ex. 1003, ¶62. Indeed Fig. 2(b) shows eight processing blocks, specifically labeled as “8 x FUZION BLOCKS.” Ex. 1006, Fig. 2(b). But Fig. 4 shows a thread manager with eight thread processors, labeled “THREAD PROCESSOR” and numbered 0-7. Stuttard describes these, stating “[t]he thread processors 1026 control the issuance of core instructions from the thread manager so as to maintain processing of

simultaneously active program threads, so that each the processing blocks 106 can be active for as much time as possible. In this particular example, the same instruction stream is supplied to all of the processing blocks in the system.” *Id.*, 11:29-35, Fig. 4. A POSITA would have understood from this disclosure that the purpose of a thread processor was not to control one thread per processing block, but was to manage the time-interleaved processing of up to eight threads, such that a single instruction of a single thread would be sent to all of the processing blocks (of which there were coincidentally eight as shown in other examples), and when that thread stalled, a single instruction from the next thread would be sent to all of the processing blocks. A POSITA would have further understood this to be the case by the next paragraph, which states that “the number of threads could exceed the number of thread processors” which clearly disagrees with Dr. Pedram’s statement that each processing block can execute one thread. *Id.*, 12:1-2.

154. There are several benefits to grouping the array of processing elements into processing blocks that is unrelated to the ability to process multiple threads at one time. One benefit is allowing the processing blocks to operate on different regions of the display, *i.e.*, different data groups. Stuttard’s system includes binning units whose purpose “is to sort primitive data by region, since the data is generally not provided by the host system in the correct order for region based processing.” *Id.*, 14:32-35. During this process, the binning units within each processing block

work collectively to retrieve primitive data from local memory, and sorts the primitives by region. *Id.*, 21:15-21. After sorting, the primitive data “is read back into the individual PEs,” with “all of the PEs in one processing block contain[ing] data concerning respective primitives occurring in a single region.” *Id.*, 22:4-10. “From this point, a given processing block operates on data associated with a single region of the display.” *Id.*

155. Dividing the array of processing elements into processing blocks also allows for redundancy. One concern with a large array of processing elements is that it “can result in a large die size for the manufacture of the device in a silicon device.” *Id.*, 9:22-28. Accordingly, it is preferable to have redundant processing elements such that if one die area is faulty, another can be used in its place. *Id.* Dividing the array of processing elements into processing blocks allows for such redundancy.

iii. Even If Stuttard Executes Multiple Threads Simultaneously, It Does Not Execute Different Tasks Simultaneously

156. Even if Stuttard’s system was capable of executing more than one thread at a time, absent additional disclosure explaining which parent task certain threads correspond to, it cannot be assumed that any two (or more) threads will correspond to two different parent tasks. Merely processing multiple threads simultaneously says nothing about whether the system can execute multiple tasks

simultaneously. Therefore, processing multiple threads simultaneously is not sufficient to disclose the requirement to “execute a first task from the plurality of tasks while executing a second task.”

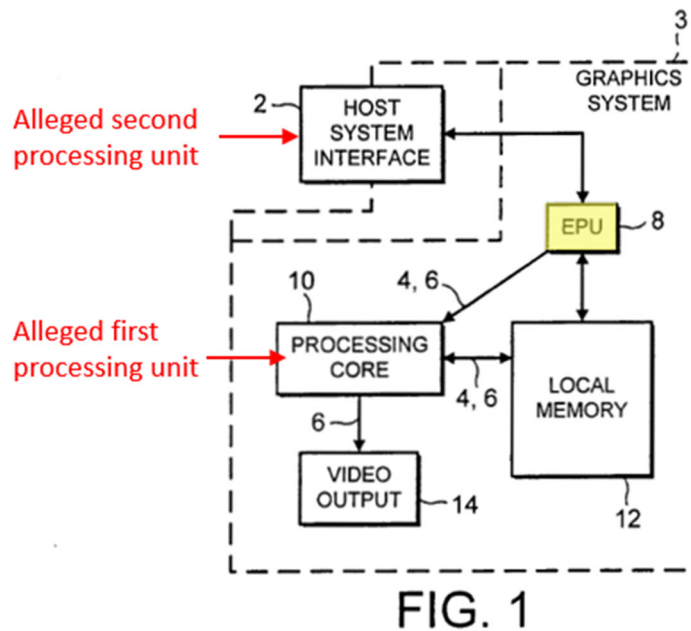
157. Pointing to the capability of multi-threading is also insufficient because Dr. Pedram has not shown that the system is “configured to” execute threads that were decomposed from two separate parent tasks. Any sort of “happenstance” or “probability” based argument, which is not supported in the first instance, would be insufficient to meet the requirement of “configured to.” *See* Section V.C, *supra*.

b. The Alleged Engines Are Within The Shader Core

158. Dr. Pedram argues that “binning unit 1069, the channel controller 108, the array controller 104, and components of the thread manager 102 (including the cache memory unit 1024, the thread processors 1026, and the status block 1030)” constitute the claimed plurality of engines. Ex. 1003, ¶125. But this theory is subject to prosecution history disclaimer, where the applicant distinguished the prior art on account of the prior art’s engines being encompassed by the prior art’s shader core. *See* Section IX.A.1.c, *supra*. Dr. Pedram argues that the plurality of processing blocks 106 constitutes the shader core. Ex. 1003, ¶¶129-130. However, as provided by Fig. 3, binning unit 1069 is part of Dr. Pedram’s alleged shader core. Ex. 1006, Fig. 3. Therefore, Dr. Pedram’s theory regarding “the plurality of engines” fails.

c. The Alleged Scheduling Module Is Not “Associated With” The Second Processing Unit

159. Dr. Pedram has two theories for the claimed “scheduling module:” (i) instruction fetch unit 1023 and/or thread scheduler 1025, and (ii) EPU 8. Ex. 1003, ¶125. I agree with the Board that instruction fetch unit 1023 and thread scheduler 1025 are units of thread manager 102 in processing core 10, which is the first processing unit and thus cannot be “associated with a second processing unit,” as required by the claims. ID, 34. However, EPU 8 is also not a scheduling module for the same reason – EPU is also not associated with a second processing unit.



Ex. 1006, Fig. 1 (annotated).

160. Dr. Pedram identifies the host system as the claimed second processing unit that EPU is associated with. Ex. 1003, ¶125. In order to be “associated with,”

the claimed scheduling unit must be incorporated within or a part of the claimed second processing unit. The claimed scheduling module cannot be associated with the first processing unit due to prosecution history disclaimer. The patentee successfully argued past a Jiao reference by arguing that the alleged scheduling module and the alleged engines “are not associated with different processing units; rather they are associated with the same processing unit—namely, shader core 130” specifically because both are “within shader core 130.” Ex. 1002, 183. Thus, the patentee equated “associated with” with “within.”

161. Stuttard provides that EPU 8 interfaces with both the host system and with processing core 10. *Id.*, Fig. 1; 8:17-19 (providing that the thread manager, which is part of processing core 10, is “connected to receive control signals from EPU 8”). If a mere interface is sufficient to disclose “associated with” then EPU 8 would be associated with the host system, but it would *also* be associated with processing core 10, which is Dr. Pedram’s alleged first processing unit. Ex. 1003, ¶54. However, the applicant distinguished prior art by arguing that the claimed scheduling module cannot be associated with the same processing unit that the claimed engines are associated with – namely, the claimed first processing unit. Ex. 1002, 182-183. The applicant explained that prior art did not disclose the claimed invention because “[t]hread scheduler 210 and the hardware units are not associated

with different processing; units; rather, they are associated with the same processing unit—namely shader core 130.” *Id.*

d. The Alleged Engines Are Not Engines

162. Petition identifies “binning unit 1069, the channel controller 108, the array controller 104, and components of the thread manager 102 (including the cache memory unit 1024, the thread processors 1026, and the status block 1030)” as the alleged engines that receive tasks from the scheduling module. Ex. 1003, ¶125. But as provided below, these alleged engines cannot be the claimed plurality of engines because they are not parallel, but serialized. Further, each of the engines do not receive tasks from a scheduling module or load state data.

163. Dr. Pedram’s alleged engines perform different functions at different stages in a serialized manner. Based on Stuttard’s figures, it is clear that several components, such as array controller, channel controller, and binning unit 1069 are downstream from thread manager. Ex. 1006, Fig. 3. Additionally, their operation is also serialized. Components of the thread manager receives threads first, before the array controller, the channel controller, or the binning unit. Thread manager “is connected to receive control signals from EPU 8,” which provide information on where and when instructions are fetched. *Id.*, 8:17-27. “Thread manager is connected to provide instructions to the array controller 104 and to the channel controller 108.” *Id.* “The array and channel controllers 104 and 108 are connected

to transfer control signals to the processing blocks 106[, which the binning unit is part of,] dependent upon the received instructions.” *Id.* Stuttard discloses only single serial graphics pipeline architecture which provides one pathway to providing work to the GPU’s execution resources. Thus, the alleged engines are not parallel and cannot be a plurality of engines.

164. Further, every alleged engine does not receive a task. For example, binning unit receives only “primitive data,” which are not tasks. *Id.*, 14:32-35, 21:15-21. And status block receives “status information,” which is also not a task. *Id.*, 12:11-14. Nor does every alleged engine load state data – Dr. Pedram does not opines that binning unit 1069, cache memory unit, or thread processors load state data. Ex. 1003, ¶56-58.

2. Stuttard Fails To Disclose or Render Obvious Claim 19

165. Claim 19 claims: The computing system of claim 15, wherein the first task comprises a graphics-processing task and the second task comprises a general-compute task. Ex. 1001, Claim 19. As a result of being dependent on Claim 15, this claim requires the shader core to execute a graphics-processing task *while* executing a general-compute task. *Id.*, Claim 15.

166. Dr. Pedram opines that Stuttard’s disclosure of “general system control” tasks are general-compute tasks or renders general-compute tasks obvious.

Ex. 1003, ¶135. However, a POSITA would understand that general system control tasks are not the same as general-compute tasks despite their nomenclature. Further, even if Stuttard disclosed general-compute tasks, which it does not, Dr. Pedram failed to explain how Stuttard's system can execute a graphics-processing task *while* executing a general-compute task.

167. The '381 Patent describes general-compute tasks as “mathematical algorithms, physics simulations, or the like.” Ex. 1001, 9:24-26. In contrast, Stuttard's disclosure of general system control tasks, and the specific tasks that it discloses as examples, are not general-compute tasks. First, Stuttard does not provide any explanation suggesting system control tasks are general-compute tasks or involve computation. The specific non-graphics tasks that Stuttard describes and Dr. Pedram cites to, are handling data input/output (“I/O”) operations, such as transferring data to and from memory. Ex. 1006, 10:23-31. These types of tasks are not general compute tasks because they are not computational in nature.

168. Furthermore, Dr. Pedram has not shown that Stuttard's system is actually configured to execute a graphics-processing task while executing general-compute task; occurring by happenstance is not enough to show the system is “configured” to do so. *See* Sections V.C and IX.C.1.a, *supra*.

169. Therefore, in my opinion, Stuttard does not disclose or render obvious claim 19.

3. Stuttard Fails To Disclose or Render Obvious Claim 20

170. Claim 20 claims: The computing system of claim 15, wherein the shader core is configured to receive the plurality of tasks based on a priority associated with each of the plurality of tasks. Ex. 1001, Claim 20. This claim requires the shader core receive tasks based on the priority associated with an individual task. *Id.*

171. Stuttard does not describe task-level priority, only thread-level priority at best. *E.g.*, Ex. 1006, 23:18-28, 23:35-24:6 (all discussing priority of threads only). Dr. Pedram opines that based on Stuttard's disclosure of thread priority, a POSITA would understand that a high priority task would give rise to high-priority threads and so switching threads based on thread priority discloses that "the shader core is configured to receive the plurality of tasks based on a priority associated with each of the plurality of tasks," as required by claim 20. Ex. 1003, ¶¶136-137.

172. A POSITA would not understand that in a high priority task would necessarily give rise to high priority threads. A POSITA would understand that the threads that were decomposed from a parent task can have different priorities, some high and some low, regardless of the parent task's priorities.

173. A POSITA would understand that thread priority may change depending on certain conditions that have nothing to do with task priority. For example, threads may have dependencies that impact their ability to be processed,

which may impact their priority level. A POSITA would also understand that the thread priority may change depending on certain conditions. For example, when a thread pauses, Stuttard describes changing which thread is executing depending on if that thread is “yielding” or “not yielding.” Ex. 1006, 24:22-26. This further indicates that the priority of a task does not relate to thread priority, and contrary to Dr. Pedram’s opinion that “all high-priority threads associated with a task are completed based on their assigned priority levels, the task has received a high priority processing by the computing system.” Ex. 1003, ¶137.

174. Most importantly, Dr. Pedram has not explained how the shader core receives tasks based on priority, which is the requirement of this claim. All of his opinion is directed to the *execution* of threads based on priority, as opposed to the *receipt* of those threads.

175. Finally, Dr. Pedram has not proven that Stuttard’s system is actually configured to receive tasks based on a priority associated with each task; occurring by happenstance is not enough to show the system is “configured” to do so. See Sections V.C and IX.C.1.a, *supra*.

176. Therefore, in my opinion, Stuttard does not disclose or render obvious claim 20.

X. CONCLUSION

177. In signing this declaration, I recognize that it will be filed as evidence in a contested case before the Patent Trial and Appeal Board of the United States Patent and Trademark Office. I also recognize that I may be subject to cross-examination in the case and that cross-examination will take place in the United States. If cross-examination is required of me, I will appear for cross-examination within the United States during the time allotted for cross-examination.

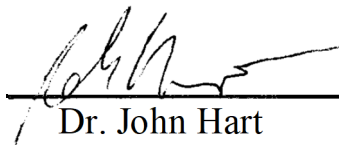
178. The statements and opinions in this declaration comprise and are based upon my review, analysis and understanding of the source materials that I have reviewed as discussed herein, based on the disclosure and statements made within such source materials.

179. I reserve the right to supplement my opinions in the future to respond to any arguments that the Petitioner raises and to take into account new information as it becomes available to me.

180. I declare under penalty of perjury under the laws of the United States of America that the foregoing is true and correct.

January 17, 2024

Respectfully Submitted,



Dr. John Hart