

Ex. A11 – Invalidity of U.S. Patent No. 8,648,867 – GPU Gems

Prior Art Reference: GPU Gems 2 (“GPU Gems”)

GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation (“GPU Gems”) was published in March 2005 and was publicly available no later than July 2005, and therefore qualifies as prior art to U.S. Patent No. 8,648,867 (the “’867 patent”) under at least pre-AIA 35 U.S.C. §§ 102(a) and (b). GPU Gems anticipates and/or renders obvious claims 16–19 of the ’867 patent (the “Asserted Claims”).

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

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

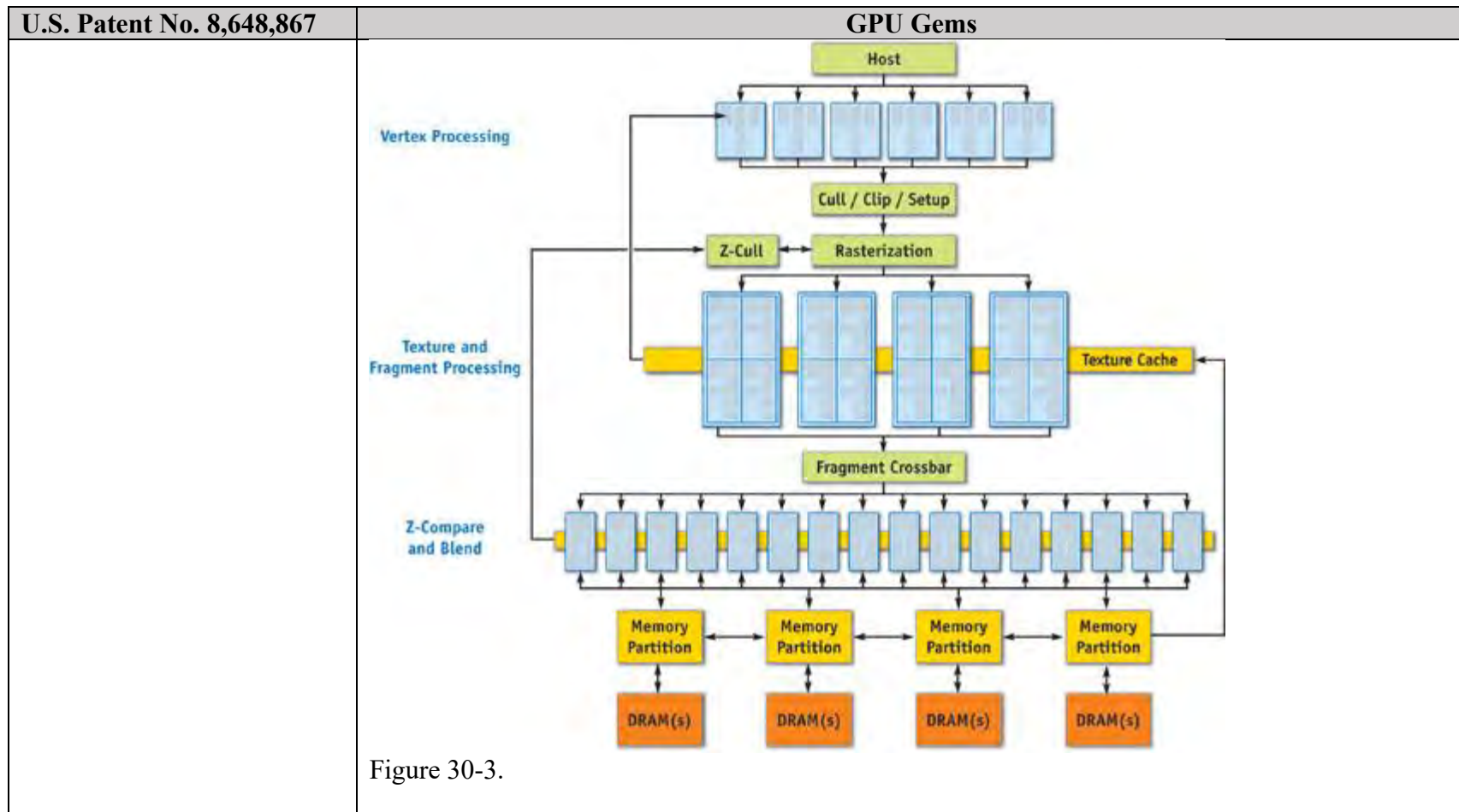
To the extent Plaintiff argues that any element below is not disclosed by GPU Gems, a person of ordinary skill in the art would have found it obvious to combine the teachings of GPU Gems with the background knowledge of a person of ordinary skill in the art and/or the additional references, and exemplary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

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

Ex. A11 – Invalidity of U.S. Patent No. 8,648,867 – GPU Gems

U.S. Patent No. 8,648,867	GPU Gems
<p>16<pre>[pre]: A method for performing a numerical simulation on input data in a computer system including a central processing unit and an accelerator, the method comprising:</pre></p>	<p>To the extent the preamble is limiting, GPU Gems discloses “[a] method for performing a numerical simulation on input data in a computer system including a central processing unit and an accelerator.” <i>See e.g.:</i></p> <p><i>As a non-limiting example, GPU Gems discloses a computer system with a central processing unit (e.g., a CPU) and an accelerator (e.g., a Graphics Processing Unit (GPU)) which performs numerical simulations (e.g., a reaction-diffusion simulation).</i></p> <div data-bbox="615 524 1608 1156" data-label="Diagram"> <pre> graph TD CPU[CPU] <--> 6.4 GB/s NB[North Bridge] NB <--> 6.4 GB/s or More SD[System DRAM] NB <--> Up to 8 GB/s GPU[GPU] GPU <--> Up to 35 GB/s GDRAM[Graphics DRAM] GPU --> Display[To Display] NB <--> SB[South Bridge] SB <--> OP[Other Peripherals] </pre> </div> <p>Figure 30-2.</p>

Ex. A11 – Invalidity of U.S. Patent No. 8,648,867 – GPU Gems



Ex. A11 – Invalidity of U.S. Patent No. 8,648,867 – GPU Gems

U.S. Patent No. 8,648,867	GPU Gems
	<div data-bbox="611 233 1604 566">  </div> <p data-bbox="611 573 768 605">Figure 31-3.</p> <p data-bbox="611 651 1098 678">Listing 31-1. The Reaction-Diffusion Kernel Program</p> <pre data-bbox="611 699 1535 1365"> float4 rd(float2 coords : DOMAIN, uniform stream concentration, uniform float2 DuDv, uniform float F, uniform float k) : RANGE { float2 center = concentration.value2(coords).xy; float2 diffusion = concentration.value2(coords + half2(1, 0)); diffusion += concentration.value2(coords + half2(-1, 0)); diffusion += concentration.value2(coords + half2(0, 1)); diffusion += concentration.value2(coords + half2(0, -1)); // Average and scale by diffusion coeffs diffusion *= 0.25f * DuDv; float2 reaction = center.xx * center.yy * center.yy; reaction.x = -1; reaction.x += (1 - DuDv.x) * center.x + F * (1 - center.x); reaction.y += (-F - k + (1 - DuDv.y)) * center.y; // Now add the diffusion to the reaction to get the result. return float4(diffusion + reaction, 0, 0); } </pre> <p data-bbox="611 1386 774 1419">Listing 31-1.</p>

Ex. A11 – Invalidity of U.S. Patent No. 8,648,867 – GPU Gems

U.S. Patent No. 8,648,867	GPU Gems
	<p>Listing 31-2. C++ Code to Set Up and Run a Reaction-Diffusion Simulation on the GPU <i>See the source code on the accompanying CD for more details.</i></p> <hr/> <pre> PUGBuffer *rdBuffer; PUGProgram *rdProgram; void init_rd(){ // Call this once. pugInit(); // Start up the GPU framework // Create a "double-buffered" PUGBuffer. Two buffers allow the // simulation to alternate using one as input, one as output PUGBuffer *rdBuffer = pugAllocateBuffer(width, height, PUG_READWRITE, 4, true); PUGProgram *rdProgram = pugLoadProgram("rd.cg", "rd"); pugBindFloat(rdProgram, "DuDv", du, dv); // bind parameters pugBindFloat(rdProgram, "F", F); pugBindFloat(rdProgram, "k", k); </pre> <hr/> <p>Listing 31-2 (continued). C++ Code to Set Up and Run a Reaction-Diffusion Simulation on the GPU</p> <pre> // Initialize the state of the simulation with values in array pugInitBuffer(rdBuffer, array); } void update_rd(){ // Call this every iteration pugBindStream(rdProgram, "concentration", rdBuffer, currentSource); PUGRect range(0, width, 0, height); pugRunProgram(rdProgram, rdBuffer, range, currentTarget); std::swap(currentSource, currentTarget); } </pre> <hr/> <p>Listing 31-2.</p> <p>The CPU in a modern computer system communicates with the GPU through a graphics connector such as a PCI Express or AGP slot on the motherboard. Because the graphics connector is responsible for transferring all command, texture, and vertex data from the CPU to the GPU, the bus technology has evolved alongside GPUs over the past few years. The original AGP slot ran at</p>

Ex. A11 – Invalidity of U.S. Patent No. 8,648,867 – GPU Gems

U.S. Patent No. 8,648,867	GPU Gems
	<p>66 MHz and was 32 bits wide, giving a transfer rate of 264 MB/sec. AGP 2x, 4x, and 8x followed, each doubling the available bandwidth, until finally the PCI Express standard was introduced in 2004, with a maximum theoretical bandwidth of 4 GB/sec simultaneously available to and from the GPU. (Your mileage may vary; currently available motherboard chipsets fall somewhat below this limit—around 3.2 GB/sec or less.) 471–72.</p> <p>The next two subsections go into detail about the architecture of the GeForce 6 Series GPUs. Section 30.2.1 describes the architecture in terms of its graphics capabilities. Section 30.2.2 describes the architecture with respect to the general computational capabilities that it provides. See Figure 30-2 for an illustration of the system architecture. 473.</p> <p>Figure 30-3 illustrates the major blocks in the GeForce 6 Series architecture. In this section, we take a trip through the graphics pipeline, starting with input arriving from the CPU and finishing with pixels being drawn to the frame buffer. 473.</p> <p>Recently, graphics processors have emerged as a powerful computational platform. A variety of encouraging results, mostly from researchers using GPUs to accelerate scientific computing and visualization applications, have shown that significant speedups can be achieved by applying GPUs to data-parallel computational problems. However, attaining these speedups requires knowledge of GPU programming and architecture.</p> <p>The preceding chapters have described the architecture of modern GPUs and the trends that govern their performance and design. Continuing from the concepts introduced in those chapters, in this chapter we present intuitive mappings of standard computational concepts onto the special-purpose features of GPUs. After presenting the basics, we introduce a simple GPU programming framework and demonstrate the use of the framework in a short sample program. 493.</p> <p>31.4 From Analogies to Implementation</p>

Ex. A11 – Invalidity of U.S. Patent No. 8,648,867 – GPU Gems

U.S. Patent No. 8,648,867	GPU Gems
	<p>By now you have a high-level understanding of how most GPGPU programs operate on current GPUs. The analogies from the previous section provide us a general implementation plan. Now, so that you can start putting the GPU to work, we dig into the nitty-gritty implementation details that you will need in practice.</p> <p>31.4.1 Putting It All Together: A Basic GPGPU Framework</p> <p>To make this easy, we've put together a very basic framework in C++ that you can use to write GPGPU programs. You'll find the framework on the CD included with this book. The framework, called pug, incorporates all of the analogies from the previous section in order to abstract GPGPU programming in a manner more accessible to experienced CPU programmers.</p> <p>Initializing and Finalizing a GPGPU Application</p> <p>The first step in a GPGPU application is to initialize the GPU. This is very easy in the framework: just call pugInit(). When the application is finished with the GPU, it can clean up the memory used by the graphics API by calling pugCleanup().</p> <p>Specifying Kernels</p> <p>Kernels in our framework are written in the Cg language. Although Cg is designed for graphics, it is based on the C language and therefore is very easy to learn. For more information, we recommend The Cg Tutorial (Fernando and Kilgard 2003) and the documentation included with the Cg distribution (NVIDIA 2004). There are very few graphics-specific keywords in Cg. We provide a Cg file, pug.cg, that you can include in your kernel files. This file defines a structure called Stream that abstracts the most graphics-centric concept: texture fetching. Stream is a wrapper for the texture sampler Cg type. Calling one of the value() functions of the Stream structure gives the value of a stream element.</p> <p>To load and initialize a kernel program from a file, use the pugLoadProgram() function. The function returns a pointer to a PUGProgram structure, which you will need to store and pass to the other framework functions to bind constants and streams to the kernel and to run it.</p> <p>Stream Management</p>

Ex. A11 – Invalidity of U.S. Patent No. 8,648,867 – GPU Gems

U.S. Patent No. 8,648,867	GPU Gems
	<p>Arrays of data in the framework are called buffers. The <code>pugAllocateBuffer()</code> function creates a new buffer. Buffers can be read-only, write-only, or read-write, and this can be specified using the mode parameter of this function. This function returns a pointer to a <code>PUGBuffer</code> structure, which can be passed to a number of functions in the framework, including the following two.</p> <p>To load initial data into a buffer, pass the data to the <code>pugInitBuffer()</code> function.</p> <p>To bind an input stream for a kernel to a buffer, call <code>pugBindStream()</code>, which takes as arguments a pointer to the <code>PUGProgram</code> to bind to, the <code>PUGBuffer</code> pointer, and a string containing the name of the <code>Stream</code> parameter in the Cg kernel program to which this <code>PUGBuffer</code> should be bound.</p> <p>Specifying Computational Domain and Range To specify the domain and range of a computation, define a <code>PUGRect</code> structure with the coordinates of the corners of the rectangle that should be used as either the domain or range. Multiple domains can be specified. These will generate additional texture coordinates that the kernel program can use as needed.</p> <p>To bind a domain, call <code>pugBindDomain()</code>, passing it the <code>PUGProgram</code>, a string parameter name for this domain defined in the kernel Cg program, and the <code>PUGRect</code> for the domain.</p> <p>There can be only one range for a kernel, due to the inability of fragment programs to scatter. To specify the range, simply pass a <code>PUGRect</code> to the range parameter of the <code>pugRunProgram()</code> function.</p> <p>Specifying Constant Parameters Constant one-, two-, three-, or four-component floating-point parameters can be specified using <code>pugBindFloat()</code>, which takes as arguments a pointer to the <code>PUGProgram</code>, a string parameter name, and up to four floating-point values.</p> <p>Invoking a Kernel Once the preceding steps have been done, executing the computation is simple: just call the function <code>pugRunProgram()</code>. Pass it a pointer to the <code>PUGProgram</code>, the output <code>PUGBuffer</code> (which</p>

Ex. A11 – Invalidity of U.S. Patent No. 8,648,867 – GPU Gems

U.S. Patent No. 8,648,867	GPU Gems
	<p>must be writable), and an optional PUGRect to specify the range. The entire buffer will be written if the range is not specified.</p> <p>Note that a stream cannot be bound to a buffer if that buffer is currently being used for the output of a kernel. The framework will automatically release all streams bound to any buffer that is specified as the output buffer in <code>pugRunProgram()</code>.</p> <p>Getting Data Back to the CPU To bring results on the GPU back to the CPU, call the <code>pugGetBufferData()</code> function, passing it a pointer to the <code>PUGBuffer</code>. This function returns a pointer to a C array of type <code>float</code>. Reading data back from the GPU can cause the GPU pipeline to flush, so use this function sparingly to avoid hurting application performance.</p> <p>Parallel Reductions The framework provides support for three types of simple parallel reduction. Buffers can be reduced along rows (to a single column vector), along columns (to a single row vector), or both (to a single value). The framework functions for these operations are <code>pugReduce1D()</code> and <code>pugReduce2D()</code>. 503–05.</p> <p>31.5 A Simple Example As a basic but nontrivial GPGPU example, we use a simulation of a phenomenon known as chemical reaction-diffusion. Reaction-diffusion is a model of how the concentrations of two or more reactants in a solution evolve in space and time as they undergo the processes of chemical reaction and diffusion. The reaction-diffusion model we use is called the Grey-Scott model (Pearson 1993) and involves just two chemical reactants. This phenomenological model does not represent a particular real chemical reaction; it serves as a simple model for studying the general reaction-diffusion phenomenon. Figure 31-3 shows the results of many iterations of the Grey-Scott model.</p> <p>Grey-Scott consists of two simple partial differential equations that govern the evolution of the concentrations of two chemical reactants, <code>U</code> and <code>V</code>:</p>

Ex. A11 – Invalidity of U.S. Patent No. 8,648,867 – GPU Gems

U.S. Patent No. 8,648,867	GPU Gems
	<div data-bbox="604 261 1335 553"> $\frac{\partial U}{\partial t} = \boxed{D_u \nabla^2 U} - \boxed{UV^2 + F(1 - U)},$ $\frac{\partial V}{\partial t} = \boxed{D_v \nabla^2 V} + \boxed{UV^2 - (F + k)V}.$ <p style="text-align: center;">Diffusion Reaction</p> </div> <p data-bbox="604 586 1797 773">Here k and F are constants; Du and Dv are the diffusion rates of the reactants; and ∇^2 is the Laplacian operator, which represents diffusion. The Laplacian operator appears commonly in physics, most notably in the form of diffusion equations, such as the heat equation. A finite difference form of the Laplacian operator applied to a scalar field p on a two-dimensional Cartesian grid (with indices i, j, and cell size x) is</p> $\nabla^2 p = \frac{p_{i+1,j} + p_{i-1,j} + p_{i,j+1} + p_{i,j-1} - 4p_{i,j}}{(\Delta x)^2}$ <p data-bbox="604 935 1850 1154">The implementation of the Grey-Scott model is fairly simple. There is a single data stream: the U and V chemical concentrations are stored in two channels of a single texture that represents a discrete spatial grid. This stream serves as input to a simple kernel, which implements the preceding equations in discrete form. The kernel is shown in Listing 31-1. C code that uses the framework to implement the simulation is shown in Listing 31-2. 505–08.</p> <p data-bbox="604 1195 995 1227">34.1 Flow-Control Challenges</p> <p data-bbox="604 1227 1871 1373">Let's start by discussing the most obvious form of flow control on GPUs. All current high-level shading languages for GPUs support traditional C-style explicit flow-control constructs, such as if-then-else, for, and while. The underlying implementations of these, however, are much different from their implementations on CPUs.</p>

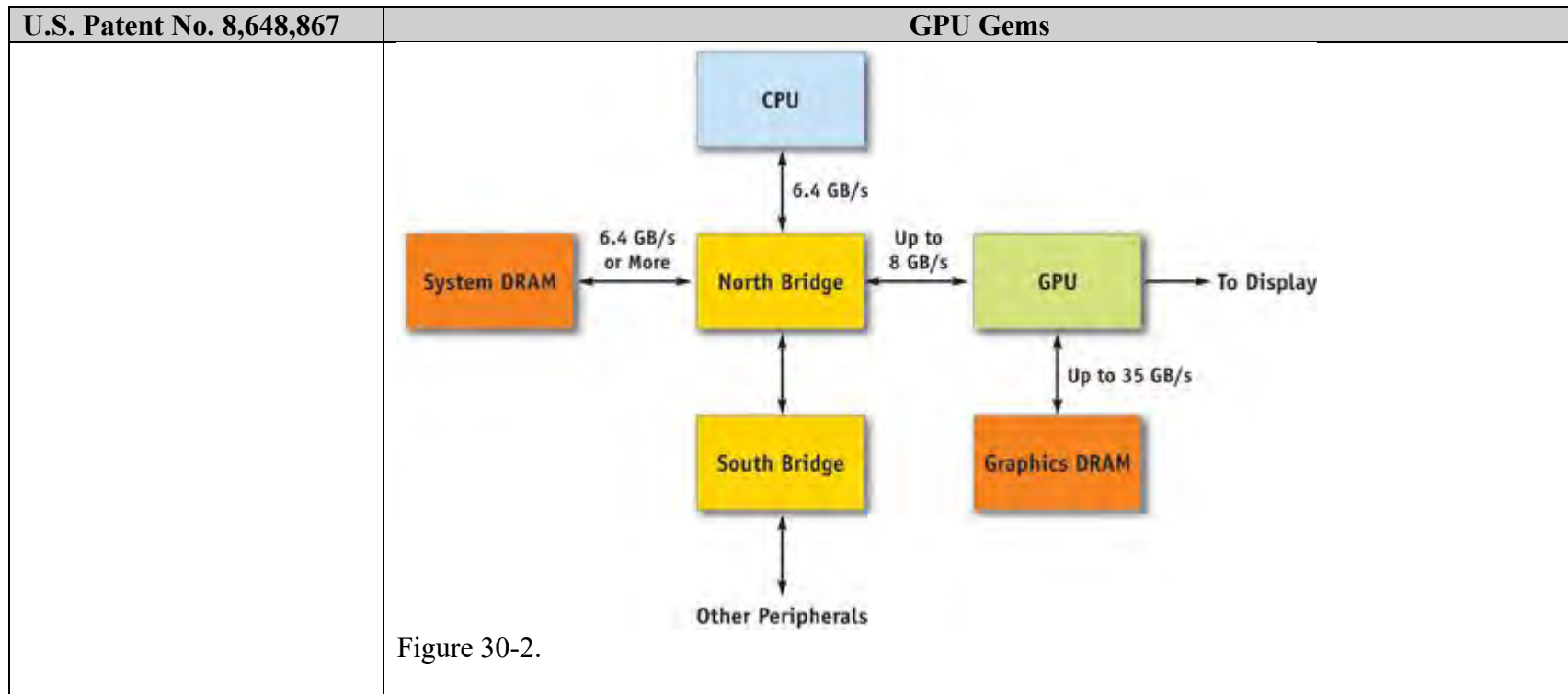
Ex. A11 – Invalidity of U.S. Patent No. 8,648,867 – GPU Gems

U.S. Patent No. 8,648,867	GPU Gems
	<p>For example, consider the following code:</p> <pre>if (a) b = f(); else b = g();</pre> <p>CPUs can easily branch based on the Boolean a and evaluate either the f() or g() functions. The performance characteristics of this branch are relatively easily understood: CPUs generally have long instruction pipelines, so it is important that the CPU be able to accurately predict whether a particular branch will be taken. If this prediction is done successfully, branching generally incurs a small penalty. If the branch is not correctly predicted, the CPU may stall for a number of cycles as the pipeline is flushed and must be refilled from the correct target address. As long as the functions f() and g() have a reasonable number of instructions, these costs aren't too high.</p> <p>The latest GPUs, such as the NVIDIA GeForce 6 Series, have similar branch instructions, though their performance characteristics are slightly different. Older GPUs do not have native branching of this form, so other strategies are necessary to emulate these operations.</p> <p>The two most common control mechanisms in parallel architectures are single instruction, multiple data (SIMD) and multiple instruction, multiple data (MIMD). All processors in a SIMD-parallel architecture execute the same instruction at the same time; in a MIMD-parallel architecture, different processors may simultaneously execute different instructions. There are three current methods used by GPUs to implement branching: MIMD branching, SIMD branching, and condition codes.</p> <p>MIMD branching is the ideal case, in which different processors can take different data-dependent branches without penalty, much like a CPU. The NVIDIA GeForce 6 Series supports MIMD branching in its vertex processors.</p> <p>SIMD branching allows branching and looping inside a program, but because all processors must execute identical instructions, divergent branching can result in reduced performance. For example, imagine a fragment program that decides the output value of each fragment by branching</p>

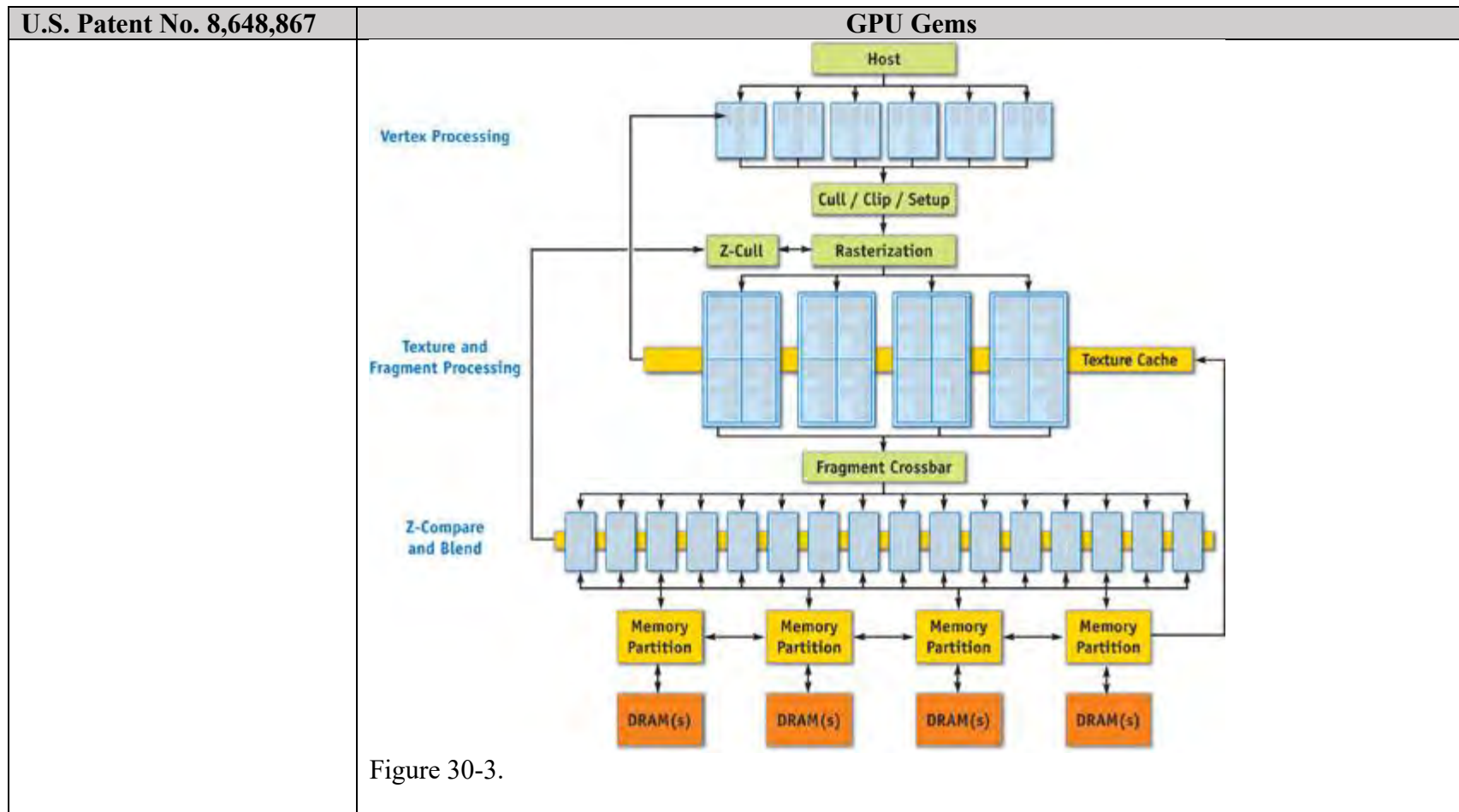
Ex. A11 – Invalidity of U.S. Patent No. 8,648,867 – GPU Gems

U.S. Patent No. 8,648,867	GPU Gems
	<p>using conditions based on random input numbers. The fragments will randomly take different branches, which can cause processors that are running threads for fragments that do not take the branch to stall until other processors are finished running threads for fragments that do take the branch. The end result is that many fragments will take as long as both branches together, plus the overhead of the branch instruction. SIMD branching is very useful in cases where the branch conditions are fairly "spatially" coherent, but lots of incoherent branching can be expensive. NVIDIA GeForce FX GPUs support SIMD branching in their vertex processors, and NVIDIA GeForce 6 Series GPUs support it in their fragment processors. 547–49.</p> <p><i>See also</i>, 457–70 (giving background on developments in computer hardware motivating development of GPU accelerators); 471–91 (discussing a computer system using the GeForce 6 Series GPU); 595–785 (providing examples of numerical simulations performed with a GPU accelerator).</p> <p>To the extent Plaintiff contends that the preamble is not expressly or inherently disclosed, or is not obvious in view of GPU Gems and the knowledge of a POSITA, it would have been obvious to combine the teachings of GPU Gems with any of the prior art references in NVIDIA’s Invalidity Contentions and any supplements thereto and the relevant section of charts for other prior art for the ’867 patent in a manner that would result in the subject matter of this limitation given, at the very least, the person's understanding of the state of the art, the problems addressed and solved in the prior art, and the teachings of GPU Gems. Specific examples of such references are cited and explained in these invalidity contentions, including the cover pleading and charts A1–A13, and Appendix A.</p>
16[a]: receiving, by an accelerator, first input data from the central processing unit;	<p>GPU Gems discloses “receiving, by an accelerator, first input data from the central processing unit.” <i>See e.g.</i>:</p> <p><i>As a non-limiting example, GPU Gems discloses an accelerator (e.g., a GPU) receiving “first input data” (e.g., providing initial values of U and V for a reaction-diffusion equation).</i></p>

Ex. A11 – Invalidity of U.S. Patent No. 8,648,867 – GPU Gems



Ex. A11 – Invalidity of U.S. Patent No. 8,648,867 – GPU Gems



Ex. A11 – Invalidity of U.S. Patent No. 8,648,867 – GPU Gems

U.S. Patent No. 8,648,867	GPU Gems
	<div data-bbox="611 233 1604 566">  </div> <p data-bbox="611 573 768 605">Figure 31-3.</p> <p data-bbox="611 688 1098 714">Listing 31-1. The Reaction-Diffusion Kernel Program</p> <pre data-bbox="611 735 1535 1403"> float4 rd(float2 coords : DOMAIN, uniform stream concentration, uniform float2 DuDv, uniform float F, uniform float k) : RANGE { float2 center = concentration.value2(coords).xy; float2 diffusion = concentration.value2(coords + half2(1, 0)); diffusion += concentration.value2(coords + half2(-1, 0)); diffusion += concentration.value2(coords + half2(0, 1)); diffusion += concentration.value2(coords + half2(0, -1)); // Average and scale by diffusion coeffs diffusion *= 0.25f * DuDv; float2 reaction = center.xx * center.yy * center.yy; reaction.x = -1; reaction.x += (1 - DuDv.x) * center.x + F * (1 - center.x); reaction.y += (-F - k + (1 - DuDv.y)) * center.y; // Now add the diffusion to the reaction to get the result. return float4(diffusion + reaction, 0, 0); } </pre>

Ex. A11 – Invalidity of U.S. Patent No. 8,648,867 – GPU Gems

U.S. Patent No. 8,648,867	GPU Gems
	<p>Listing 31-1.</p> <p>Listing 31-2. C++ Code to Set Up and Run a Reaction-Diffusion Simulation on the GPU <i>See the source code on the accompanying CD for more details.</i></p> <hr/> <pre> PUGBuffer *rdBuffer; PUGProgram *rdProgram; void init_rd(){ // Call this once. pugInit(); // Start up the GPU framework // Create a "double-buffered" PUGBuffer. Two buffers allow the // simulation to alternate using one as input, one as output PUGBuffer *rdBuffer = pugAllocateBuffer(width, height, PUG_READWRITE, 4, true); PUGProgram *rdProgram = pugLoadProgram("rd.cg", "rd"); pugBindFloat(rdProgram, "DuDv", du, dv); // bind parameters pugBindFloat(rdProgram, "F", F); pugBindFloat(rdProgram, "k", k); </pre> <hr/> <p>Listing 31-2 (continued). C++ Code to Set Up and Run a Reaction-Diffusion Simulation on the GPU</p> <pre> // Initialize the state of the simulation with values in array pugInitBuffer(rdBuffer, array); } void update_rd(){ // Call this every iteration pugBindStream(rdProgram, "concentration", rdBuffer, currentSource); PUGRect range(0, width, 0, height); pugRunProgram(rdProgram, rdBuffer, range, currentTarget); std::swap(currentSource, currentTarget); } </pre> <hr/> <p>Listing 31-2.</p> <p>Render-to-Texture</p> <p>When an image is generated by the GPU, it can be written to frame-buffer memory that can be displayed, or it can be written to texture memory. This render-to-texture functionality is essential</p>

Ex. A11 – Invalidity of U.S. Patent No. 8,648,867 – GPU Gems

U.S. Patent No. 8,648,867	GPU Gems
	<p>for GPGPU, because it is the only current mechanism with which to implement direct feedback of GPU output to input without going back to the host processor. (Indirect feedback is also available via copy-to-texture, which requires a copy from one location in the GPU's memory to another.) We can think of render-to-texture as a write-only memory interface.</p> <p>You may be wondering why we don't consider the texture unit and render-to-texture together as a read-write memory interface. The reason is that the fragment processor can read memory as many times as it wants inside a kernel, but it can write data only at the end of the kernel program (this is stream out). Thus, memory reads and writes are fundamentally separate on GPUs, so it helps to think about them that way.</p> <p>499.</p> <p>31.4.1 Putting It All Together: A Basic GPGPU Framework</p> <p>To make this easy, we've put together a very basic framework in C++ that you can use to write GPGPU programs. You'll find the framework on the CD included with this book. The framework, called pug, incorporates all of the analogies from the previous section in order to abstract GPGPU programming in a manner more accessible to experienced CPU programmers.</p> <p>Initializing and Finalizing a GPGPU Application</p> <p>The first step in a GPGPU application is to initialize the GPU. This is very easy in the framework: just call pugInit(). When the application is finished with the GPU, it can clean up the memory used by the graphics API by calling pugCleanup().</p> <p>Specifying Kernels</p> <p>Kernels in our framework are written in the Cg language. Although Cg is designed for graphics, it is based on the C language and therefore is very easy to learn. For more information, we recommend The Cg Tutorial (Fernando and Kilgard 2003) and the documentation included with the Cg distribution (NVIDIA 2004). There are very few graphics-specific keywords in Cg. We provide a Cg file, pug.cg, that you can include in your kernel files. This file defines a structure called Stream that abstracts the most graphics-centric concept: texture fetching. Stream is a wrapper for the texture sampler Cg type. Calling one of the value() functions of the Stream structure gives the value of a stream element.</p>

Ex. A11 – Invalidity of U.S. Patent No. 8,648,867 – GPU Gems

U.S. Patent No. 8,648,867	GPU Gems
	<p>To load and initialize a kernel program from a file, use the <code>pugLoadProgram()</code> function. The function returns a pointer to a <code>PUGProgram</code> structure, which you will need to store and pass to the other framework functions to bind constants and streams to the kernel and to run it.</p> <p>Stream Management Arrays of data in the framework are called buffers. The <code>pugAllocateBuffer()</code> function creates a new buffer. Buffers can be read-only, write-only, or read-write, and this can be specified using the mode parameter of this function. This function returns a pointer to a <code>PUGBuffer</code> structure, which can be passed to a number of functions in the framework, including the following two.</p> <p>To load initial data into a buffer, pass the data to the <code>pugInitBuffer()</code> function.</p> <p>To bind an input stream for a kernel to a buffer, call <code>pugBindStream()</code>, which takes as arguments a pointer to the <code>PUGProgram</code> to bind to, the <code>PUGBuffer</code> pointer, and a string containing the name of the Stream parameter in the Cg kernel program to which this <code>PUGBuffer</code> should be bound.</p> <p>Specifying Computational Domain and Range To specify the domain and range of a computation, define a <code>PUGRect</code> structure with the coordinates of the corners of the rectangle that should be used as either the domain or range. Multiple domains can be specified. These will generate additional texture coordinates that the kernel program can use as needed.</p> <p>To bind a domain, call <code>pugBindDomain()</code>, passing it the <code>PUGProgram</code>, a string parameter name for this domain defined in the kernel Cg program, and the <code>PUGRect</code> for the domain.</p> <p>There can be only one range for a kernel, due to the inability of fragment programs to scatter. To specify the range, simply pass a <code>PUGRect</code> to the range parameter of the <code>pugRunProgram()</code> function.</p> <p>Specifying Constant Parameters</p>

Ex. A11 – Invalidity of U.S. Patent No. 8,648,867 – GPU Gems

U.S. Patent No. 8,648,867	GPU Gems
	<p>Constant one-, two-, three-, or four-component floating-point parameters can be specified using <code>pugBindFloat()</code>, which takes as arguments a pointer to the <code>PUGProgram</code>, a string parameter name, and up to four floating-point values.</p> <p>Invoking a Kernel</p> <p>Once the preceding steps have been done, executing the computation is simple: just call the function <code>pugRunProgram()</code>. Pass it a pointer to the <code>PUGProgram</code>, the output <code>PUGBuffer</code> (which must be writable), and an optional <code>PUGRect</code> to specify the range. The entire buffer will be written if the range is not specified.</p> <p>Note that a stream cannot be bound to a buffer if that buffer is currently being used for the output of a kernel. The framework will automatically release all streams bound to any buffer that is specified as the output buffer in <code>pugRunProgram()</code>.</p> <p>Getting Data Back to the CPU</p> <p>To bring results on the GPU back to the CPU, call the <code>pugGetBufferData()</code> function, passing it a pointer to the <code>PUGBuffer</code>. This function returns a pointer to a C array of type <code>float</code>. Reading data back from the GPU can cause the GPU pipeline to flush, so use this function sparingly to avoid hurting application performance.</p> <p>Parallel Reductions</p> <p>The framework provides support for three types of simple parallel reduction. Buffers can be reduced along rows (to a single column vector), along columns (to a single row vector), or both (to a single value). The framework functions for these operations are <code>pugReduce1D()</code> and <code>pugReduce2D()</code>.</p> <p>503–505.</p> <p>31.5 A Simple Example</p> <p>As a basic but nontrivial GPGPU example, we use a simulation of a phenomenon known as chemical reaction-diffusion. Reaction-diffusion is a model of how the concentrations of two or more reactants in a solution evolve in space and time as they undergo the processes of chemical</p>

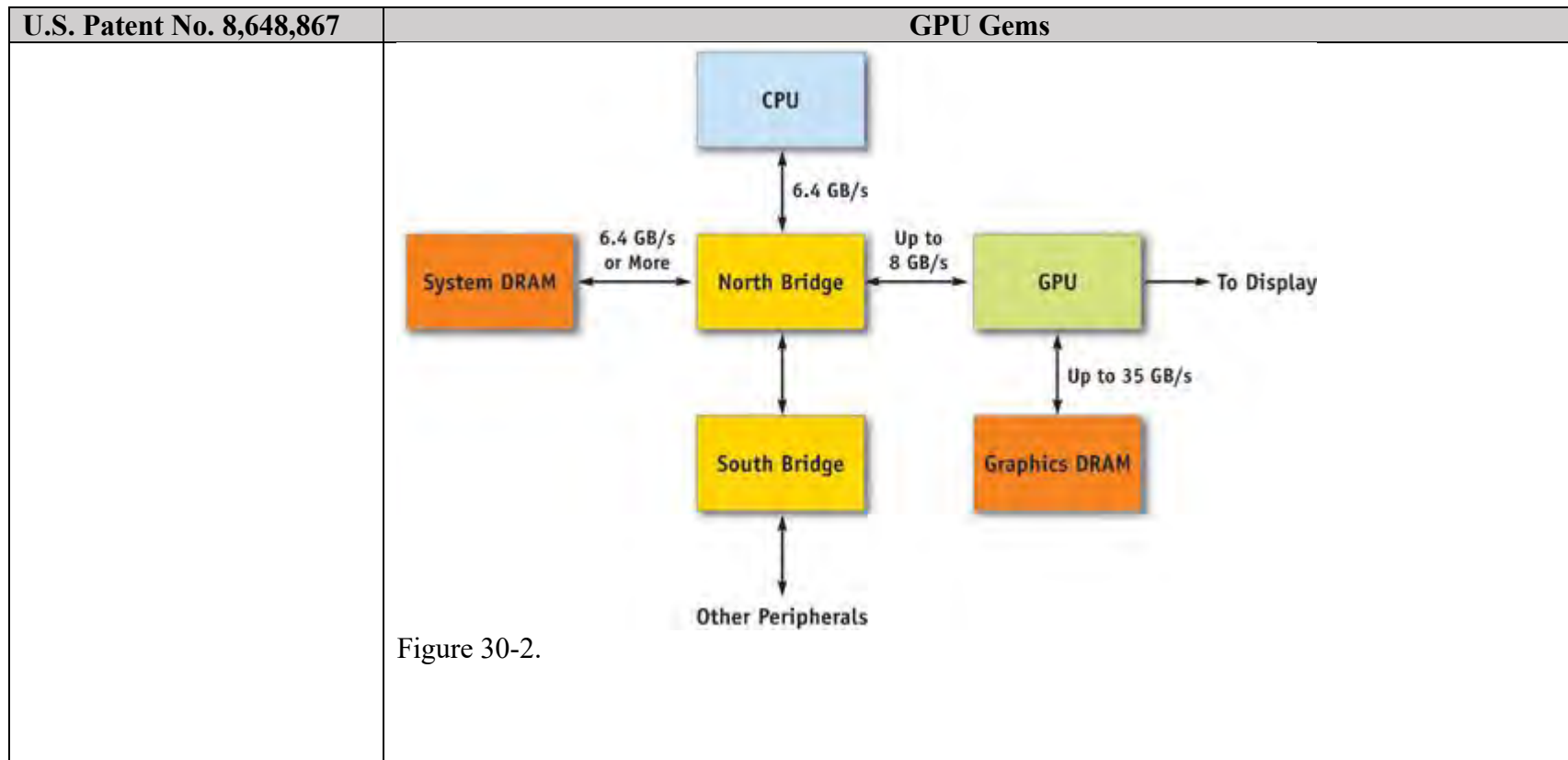
Ex. A11 – Invalidity of U.S. Patent No. 8,648,867 – GPU Gems

U.S. Patent No. 8,648,867	GPU Gems
	<p>reaction and diffusion. The reaction-diffusion model we use is called the Grey-Scott model (Pearson 1993) and involves just two chemical reactants. This phenomenological model does not represent a particular real chemical reaction; it serves as a simple model for studying the general reaction-diffusion phenomenon. Figure 31-3 shows the results of many iterations of the Grey-Scott model.</p> <p>Grey-Scott consists of two simple partial differential equations that govern the evolution of the concentrations of two chemical reactants, U and V:</p> $\begin{aligned}\frac{\partial U}{\partial t} &= \boxed{D_u \nabla^2 U} - \boxed{UV^2 + F(1 - U)}, \\ \frac{\partial V}{\partial t} &= \boxed{D_v \nabla^2 V} + \boxed{UV^2 - (F + k)V}.\end{aligned}$ <p style="text-align: center;">Diffusion Reaction</p> <p>Here k and F are constants; D_u and D_v are the diffusion rates of the reactants; and ∇^2 is the Laplacian operator, which represents diffusion. The Laplacian operator appears commonly in physics, most notably in the form of diffusion equations, such as the heat equation. A finite difference form of the Laplacian operator applied to a scalar field p on a two-dimensional Cartesian grid (with indices i, j, and cell size x) is</p> $\nabla^2 p = \frac{p_{i+1,j} + p_{i-1,j} + p_{i,j+1} + p_{i,j-1} - 4p_{i,j}}{(\Delta x)^2}$ <p>The implementation of the Grey-Scott model is fairly simple. There is a single data stream: the U and V chemical concentrations are stored in two channels of a single texture that represents a discrete spatial grid. This stream serves as input to a simple kernel, which implements the preceding equations in discrete form. The kernel is shown in Listing 31-1. C code that uses the framework to implement the simulation is shown in Listing 31-2.</p>

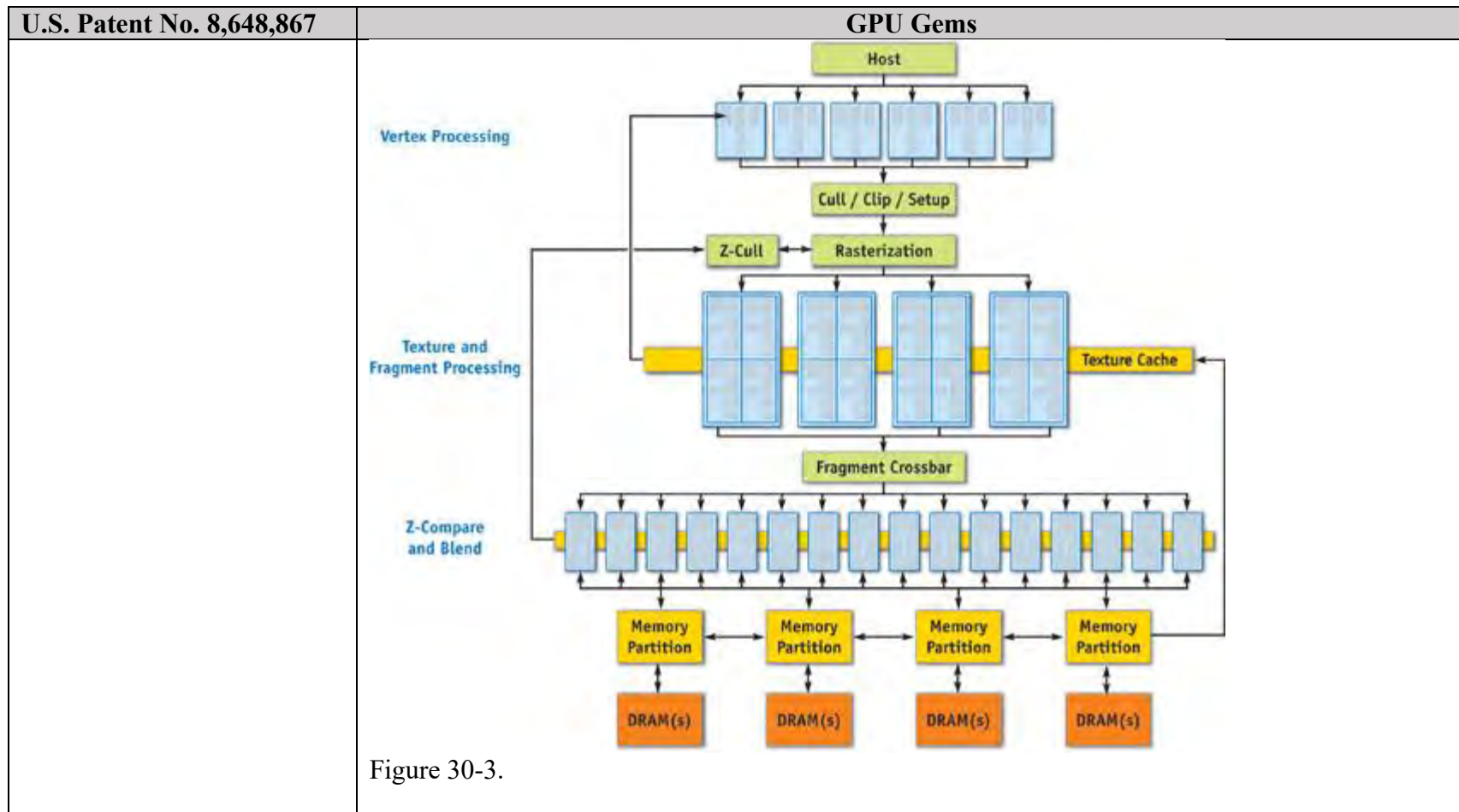
Ex. A11 – Invalidity of U.S. Patent No. 8,648,867 – GPU Gems

U.S. Patent No. 8,648,867	GPU Gems
	<p>505–08.</p> <p><i>See also</i>, 471–91 (discussing a computer system using the GeForce 6 Series GPU); 520–45 (discussing accelerator memory structures and access).</p> <p>To the extent Plaintiff contends that this claim element is not expressly or inherently disclosed, or is not obvious in view of GPU Gems and the knowledge of a POSITA, it would have been obvious to combine the teachings of GPU Gems with any of the prior art references in NVIDIA’s Invalidity Contentions and any supplements thereto and the relevant section of charts for other prior art for the ’867 patent in a manner that would result in the subject matter of this limitation given, at the very least, the person’s understanding of the state of the art, the problems addressed and solved in the prior art, and the teachings of GPU Gems. Specific examples of such references are cited and explained in these invalidity contentions, including the cover pleading and charts A1–A13, and Appendix A.</p>
<p>16[b]: transferring, by an accelerator controller, the first input data into a first partition, referenced by first pointer, of an accelerator memory before a first computational cycle of the numerical simulation;</p>	<p>GPU Gems discloses “transferring, by an accelerator controller, the first input data into a first partition, referenced by first pointer, of an accelerator memory before a first computational cycle of the numerical simulation.” <i>See e.g.</i>:</p> <p><i>As a non-limiting example, GPU Gems discloses an accelerator controller (e.g., vertex processors and/or a rasterizer of a GPU) and accelerator memory (e.g., on-chip memory, GPU caches, GPU Video Memory, GPU Constant Registers, GPU Temporary Registers, and/or a buffer). GPU Gems further discloses transferring the first input data (e.g., data within the variable array, see also 16[a]) into a “first partition” of the accelerator memory (e.g., using the function <code>pugInitBuffer()</code>) referenced by a first pointer (e.g., the pointer variable <code>rdBuffer</code>).</i></p>

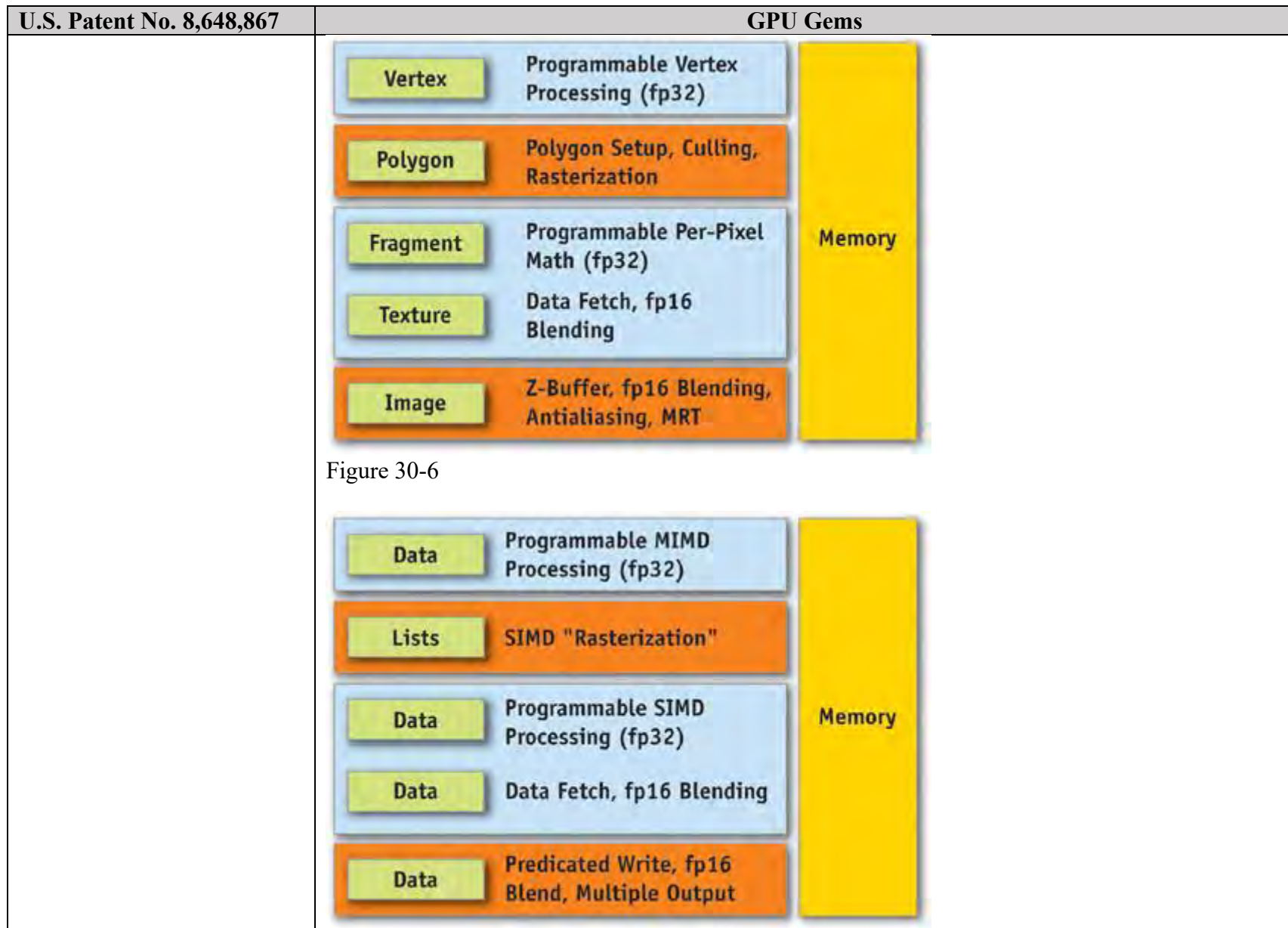
Ex. A11 – Invalidity of U.S. Patent No. 8,648,867 – GPU Gems



Ex. A11 – Invalidity of U.S. Patent No. 8,648,867 – GPU Gems



Ex. A11 – Invalidity of U.S. Patent No. 8,648,867 – GPU Gems



Ex. A11 – Invalidity of U.S. Patent No. 8,648,867 – GPU Gems

U.S. Patent No. 8,648,867	GPU Gems
	<p data-bbox="611 237 768 264">Figure 30-7.</p>  <p data-bbox="611 651 768 678">Figure 31-3.</p>

Ex. A11 – Invalidity of U.S. Patent No. 8,648,867 – GPU Gems

U.S. Patent No. 8,648,867	GPU Gems
	<p><u>Listing 31-1. The Reaction-Diffusion Kernel Program</u></p> <pre> float4 rd(float2 coords : DOMAIN, uniform stream concentration, uniform float2 DuDv, uniform float F, uniform float k) : RANGE { float2 center = concentration.value2(coords).xy; float2 diffusion = concentration.value2(coords + half2(1, 0)); diffusion += concentration.value2(coords + half2(-1, 0)); diffusion += concentration.value2(coords + half2(0, 1)); diffusion += concentration.value2(coords + half2(0, -1)); // Average and scale by diffusion coeffs diffusion *= 0.25f * DuDv; float2 reaction = center.xx * center.yy * center.yy; reaction.x = -1; reaction.x += (1 - DuDv.x) * center.x + F * (1 - center.x); reaction.y += (-F - k + (1 - DuDv.y)) * center.y; // Now add the diffusion to the reaction to get the result. return float4(diffusion + reaction, 0, 0); } </pre> <hr/> <p>Listing 31-1.</p>

Ex. A11 – Invalidity of U.S. Patent No. 8,648,867 – GPU Gems

U.S. Patent No. 8,648,867	GPU Gems
	<p>Listing 31-2. C++ Code to Set Up and Run a Reaction-Diffusion Simulation on the GPU <i>See the source code on the accompanying CD for more details.</i></p> <hr/> <pre> PUGBuffer *rdBuffer; PUGProgram *rdProgram; void init_rd(){ // Call this once. pugInit(); // Start up the GPU framework // Create a "double-buffered" PUGBuffer. Two buffers allow the // simulation to alternate using one as input, one as output PUGBuffer *rdBuffer = pugAllocateBuffer(width, height, PUG_READWRITE, 4, true); PUGProgram *rdProgram = pugLoadProgram("rd.cg", "rd"); pugBindFloat(rdProgram, "DuDv", du, dv); // bind parameters pugBindFloat(rdProgram, "F", F); pugBindFloat(rdProgram, "k", k); </pre> <hr/> <p>Listing 31-2 (continued). C++ Code to Set Up and Run a Reaction-Diffusion Simulation on the GPU</p> <hr/> <pre> // Initialize the state of the simulation with values in array pugInitBuffer(rdBuffer, array); } void update_rd(){ // Call this every iteration pugBindStream(rdProgram, "concentration", rdBuffer, currentSource); PUGRect range(0, width, 0, height); pugRunProgram(rdProgram, rdBuffer, range, currentTarget); std::swap(currentSource, currentTarget); } </pre> <hr/> <p>Listing 31-2 (annotated).</p>

Ex. A11 – Invalidity of U.S. Patent No. 8,648,867 – GPU Gems

U.S. Patent No. 8,648,867	GPU Gems
	<div data-bbox="606 233 1386 867" data-label="Diagram"> <pre> graph TD Disk[Disk] --> CPU_MM[CPU Main Memory] CPU_MM --> CPU_Caches[CPU Caches] CPU_MM --> GPU_Video_Memory[GPU Video Memory] CPU_MM --> GPU_Const_Regs[GPU Constant Registers] CPU_Caches --> CPU_Regs[CPU Registers] GPU_Video_Memory --> GPU_Caches[GPU Caches] GPU_Caches --> GPU_Temp_Regs[GPU Temporary Registers] subgraph CPU CPU_Caches CPU_Regs end subgraph GPU GPU_Video_Memory GPU_Caches GPU_Const_Regs GPU_Temp_Regs end </pre> </div> <p data-bbox="606 883 768 915">Figure 33-2.</p> <p data-bbox="606 956 1419 989">30.2.2 Functional Block Diagram for Non-Graphics Operations</p> <p data-bbox="606 997 1883 1094">As graphics hardware becomes more and more programmable, applications unrelated to the standard polygon pipeline (as described in the preceding section) are starting to present themselves as candidates for execution on GPUs.</p> <p data-bbox="606 1143 1883 1240">Figure 30-6 shows a simplified view of the GeForce 6 Series architecture, when used as a graphics pipeline. It contains a programmable vertex engine, a programmable fragment engine, a texture load/filter engine, and a depth-compare/blending data write engine.</p> <p data-bbox="606 1289 1883 1386">In this alternative view, a GPU can be seen as a large amount of programmable floating-point horsepower and memory bandwidth that can be exploited for compute-intensive applications completely unrelated to computer graphics.</p>

Ex. A11 – Invalidity of U.S. Patent No. 8,648,867 – GPU Gems

U.S. Patent No. 8,648,867	GPU Gems
	<p>Figure 30-7 shows another way to view the GeForce 6 Series architecture. When used for non-graphics applications, it can be viewed as two programmable blocks that run serially: the vertex processor and the fragment processor, both with support for fp32 operands and intermediate values. Both use the texture unit as a random-access data fetch unit and access data at a phenomenal 35 GB/sec (550 MHz DDR memory clock x 256 bits per clock cycle x 2 transfers per clock cycle). In addition, both the vertex and the fragment processor are highly computationally capable. (Performance details follow in Section 30.4.)</p> <p>The vertex processor operates on data, passing it directly to the fragment processor, or by using the rasterizer to expand the data into interpolated values. At this point, each triangle (or point) from the vertex processor has become one or more fragments.</p> <p>Before a fragment reaches the fragment processor, the z-cull unit compares the pixel's depth with the values that already exist in the depth buffer. If the pixel's depth is greater, the pixel will not be visible, and there is no point shading that fragment, so the fragment processor isn't even executed. (This optimization happens only if it's clear that the fragment processor isn't going to modify the fragment's depth.) Thinking in a general-purpose sense, this early culling feature makes it possible to quickly decide to skip work on specific fragments based on a scalar test. Chapter 34 of this book, "GPU Flow-Control Idioms," explains how to take advantage of this feature to efficiently predicate work for general-purpose computations.</p> <p>After the fragment processor runs on a potential pixel (still a "fragment" because it has not yet reached the frame buffer), the fragment must pass a number of tests in order to move farther down the pipeline. (There may also be more than one fragment that comes out of the fragment processor if multiple render targets [MRTs] are being used. Up to four MRTs can be used to write out large amounts of data—up to 16 scalar floating-point values at a time, for example—plus depth.)</p> <p>First, the scissor test rejects the fragment if it lies outside a specified subrectangle of the frame buffer. Although the popular graphics APIs define scissoring at this location in the pipeline, it is more efficient to perform the scissor test in the rasterizer. Scissoring in x and y actually happens in the rasterizer, before fragment processing, and z scissoring happens during z-cull. This avoids all fragment processor work on scissored (rejected) pixels. Scissoring is rarely useful for general-</p>

Ex. A11 – Invalidity of U.S. Patent No. 8,648,867 – GPU Gems

U.S. Patent No. 8,648,867	GPU Gems
	<p>purpose computation because general-purpose programmers typically draw rectangles to perform computations in the first place.</p> <p>Next, the fragment's depth is compared with the depth in the frame buffer. If the depth test passes, the fragment moves on in the pipeline. Optionally, the depth value in the frame buffer can be replaced at this stage.</p> <p>After this, the fragment can optionally test and modify what is known as the stencil buffer, which stores an integer value per pixel. The stencil buffer was originally intended to allow programmers to mask off certain pixels (for example, to restrict drawing to a cockpit's windshield), but it has found other uses as a way to count values by incrementing or decrementing the existing value. This feature is used for stencil shadow volumes, for example.</p> <p>If the fragment passes the depth and stencil tests, it can then optionally modify the contents of the frame buffer using the blend function. A blend function can be described as</p> $\text{out} = \text{src} * \text{srcOp} + \text{dst} * \text{dstOp}$ <p>where source is the fragment color flowing down the pipeline; dst is the color value in the frame buffer; and srcOp and dstOp can be specified to be constants, source color components, or destination color components. Full blend functionality is supported for all pixel formats up to fp16x4. However, fp32 frame buffers don't support blending—only updating the buffer is allowed.</p> <p>Finally, a feature called occlusion query makes it possible to quickly determine if any of the fragments that would be rendered in a particular computation would cause results to be written to the frame buffer. (Recall that fragments that do not pass the z-test don't have any effect on the values in the frame buffer.) Traditionally, the occlusion query test is used to allow graphics applications to avoid making draw calls for occluded objects, but it is useful for GPGPU applications as well. For instance, if the depth test is used to determine which outputs need to be updated in a sparse array, updating depth can be used to indicate when a given output has converged and no further work is needed. In this case, occlusion query can be used to tell when all</p>

Ex. A11 – Invalidity of U.S. Patent No. 8,648,867 – GPU Gems

U.S. Patent No. 8,648,867	GPU Gems
	<p>output calculations are done. See Chapter 34 of this book, "GPU Flow-Control Idioms," for further information about this idea. 478–480.</p> <p>31.5 A Simple Example As a basic but nontrivial GPGPU example, we use a simulation of a phenomenon known as chemical reaction-diffusion. Reaction-diffusion is a model of how the concentrations of two or more reactants in a solution evolve in space and time as they undergo the processes of chemical reaction and diffusion. The reaction-diffusion model we use is called the Grey-Scott model (Pearson 1993) and involves just two chemical reactants. This phenomenological model does not represent a particular real chemical reaction; it serves as a simple model for studying the general reaction-diffusion phenomenon. Figure 31-3 shows the results of many iterations of the Grey-Scott model.</p> <p>Grey-Scott consists of two simple partial differential equations that govern the evolution of the concentrations of two chemical reactants, U and V:</p> $\begin{aligned}\frac{\partial U}{\partial t} &= \boxed{D_u \nabla^2 U} - \boxed{UV^2 + F(1 - U)}, \\ \frac{\partial V}{\partial t} &= \boxed{D_v \nabla^2 V} + \boxed{UV^2 - (F + k)V}.\end{aligned}$ <p style="text-align: center;">Diffusion Reaction</p> <p>Here k and F are constants; D_u and D_v are the diffusion rates of the reactants; and ∇^2 is the Laplacian operator, which represents diffusion. The Laplacian operator appears commonly in physics, most notably in the form of diffusion equations, such as the heat equation. A finite difference form of the Laplacian operator applied to a scalar field p on a two-dimensional Cartesian grid (with indices i, j, and cell size x) is</p>

Ex. A11 – Invalidity of U.S. Patent No. 8,648,867 – GPU Gems

U.S. Patent No. 8,648,867	GPU Gems
	$\nabla^2 p = \frac{p_{i+1,j} + p_{i-1,j} + p_{i,j+1} + p_{i,j-1} - 4p_{i,j}}{(\Delta x)^2}$ <p>The implementation of the Grey-Scott model is fairly simple. There is a single data stream: the U and V chemical concentrations are stored in two channels of a single texture that represents a discrete spatial grid. This stream serves as input to a simple kernel, which implements the preceding equations in discrete form. The kernel is shown in Listing 31-1. C code that uses the framework to implement the simulation is shown in Listing 31-2.</p> <p>505–08.</p> <p>33.2.1 Memory Hierarchy</p> <p>Figure 33-2 shows the combined CPU and GPU memory hierarchy. The GPU's memory system creates a branch in a modern computer's memory hierarchy. The GPU, just like a CPU, has its own caches and registers to accelerate data access during computation. GPUs, however, also have their own main memory with its own address space—meaning that programmers must explicitly copy data into GPU memory before beginning program execution. This transfer has traditionally been a bottleneck for many applications, but the advent of the new PCI Express bus standard may make sharing memory between the CPU and GPU a more viable possibility in the near future.</p> <p>524.</p> <p>33.2.3 GPU Kernel Memory Access</p> <p>Vertex and fragment programs (kernels) are the workhorses of modern GPUs. Vertex programs operate on vertex stream elements and send output to the rasterizer. Fragment programs operate on fragment streams and write output to frame buffers. The capabilities of these programs are defined by the arithmetic operations they can perform and the memory they are permitted to access. The variety of available arithmetic operations permitted in GPU kernels is approaching those available on CPUs, yet there are numerous memory access restrictions. As described previously, many of these restrictions are in place to preserve the parallelism required for GPUs to maintain their speed advantage. Other restrictions, however, are artifacts of the evolving GPU architecture and will almost certainly be relaxed in future generations.</p>

Ex. A11 – Invalidity of U.S. Patent No. 8,648,867 – GPU Gems

U.S. Patent No. 8,648,867	GPU Gems
	<p>The following is a list of memory access rules for vertex and fragment kernels on GPUs that support Pixel Shader 3.0 and Vertex Shader 3.0 functionality (Microsoft 2004a, b):</p> <p>No CPU main memory access; no disk access. No GPU stack or heap. Random reads from global texture memory. Reads from constant registers.</p> <ul style="list-style-type: none"> - Vertex programs can use relative indexing of constant registers. <p>Reads/writes to temporary registers.</p> <ul style="list-style-type: none"> - Registers are local to the stream element being processed. - No relative indexing of registers. <p>Streaming reads from stream input registers.</p> <ul style="list-style-type: none"> - Vertex kernels read from vertex streams. - Fragment kernels read from fragment streams (rasterizer results). <p>Streaming writes (at end of kernel only).</p> <ul style="list-style-type: none"> - Write location is fixed by the position of the element in the stream. <p>Cannot write to computed address (that is, no scatter).</p> <ul style="list-style-type: none"> - Vertex kernels write to vertex output streams. <p>Can write up to 12 four-component floating-point values.</p> <ul style="list-style-type: none"> - Fragment kernels write to frame-buffer streams. <p>Can write up to 4 four-component floating-point values.</p> <p>An additional access pattern emerges from the preceding set of rules and the stream types described in Section 33.2.2: pointer streams (Purcell et al. 2002). Pointer streams arise out of the ability to use any input stream as the address for a texture read. Figure 33-4 shows that pointer streams are simply streams whose values are memory addresses. If the pointer stream is read from a texture, this capability is called dependent texturing.</p>

Ex. A11 – Invalidity of U.S. Patent No. 8,648,867 – GPU Gems

U.S. Patent No. 8,648,867	GPU Gems
	<p>527–28.</p> <p><i>See also</i>, 471–91 (discussing a computer system using the GeForce 6 Series GPU); 520–45 (discussing accelerator memory structures and access).</p> <p>To the extent Plaintiff contends that this claim element is not expressly or inherently disclosed, or is not obvious in view of GPU Gems and the knowledge of a POSITA, it would have been obvious to combine the teachings of GPU Gems with any of the prior art references in NVIDIA’s Invalidity Contentions and any supplements thereto and the relevant section of charts for other prior art for the ’867 patent in a manner that would result in the subject matter of this limitation given, at the very least, the person’s understanding of the state of the art, the problems addressed and solved in the prior art, and the teachings of GPU Gems. Specific examples of such references are cited and explained in these invalidity contentions, including the cover pleading and charts A1–A13, and Appendix A.</p>
<p>16[c]: performing, by at least one graphics processing unit during the first computational cycle, at least one calculation on the first portion of the input data as to generate first output data;</p>	<p>GPU Gems discloses “performing, by at least one graphics processing unit during the first computational cycle, at least one calculation on the first portion of the input data as to generate first output data.” <i>See e.g.</i>:</p> <p><i>As a non-limiting example, GPU Gems discloses at least one graphics processing unit (e.g., vertex processors and/or fragment processors) that performs at least one calculation (e.g., an iteration of the reaction-diffusion equation, a fragment program, and/or a kernel program see also 16[pre]) on a “first portion of the input data” (see 16[a]) during a first computational cycle to generate first output data (e.g., diffusion + reaction).</i></p>

Ex. A11 – Invalidity of U.S. Patent No. 8,648,867 – GPU Gems

U.S. Patent No. 8,648,867	GPU Gems
	<div data-bbox="611 233 1604 565">  </div> <p data-bbox="611 573 768 605">Figure 31-3.</p> <p data-bbox="611 686 1098 714">Listing 31-1. The Reaction-Diffusion Kernel Program</p> <pre data-bbox="611 735 1535 1404"> float4 rd(float2 coords : DOMAIN, uniform stream concentration, uniform float2 DuDv, uniform float F, uniform float k) : RANGE { float2 center = concentration.value2(coords).xy; float2 diffusion = concentration.value2(coords + half2(1, 0)); diffusion += concentration.value2(coords + half2(-1, 0)); diffusion += concentration.value2(coords + half2(0, 1)); diffusion += concentration.value2(coords + half2(0, -1)); // Average and scale by diffusion coeffs diffusion *= 0.25f * DuDv; float2 reaction = center.xx * center.yy * center.yy; reaction.x = -1; reaction.x += (1 - DuDv.x) * center.x + F * (1 - center.x); reaction.y += (-F - k + (1 - DuDv.y)) * center.y; // Now add the diffusion to the reaction to get the result. return float4(diffusion + reaction, 0, 0); } </pre>

Ex. A11 – Invalidity of U.S. Patent No. 8,648,867 – GPU Gems

U.S. Patent No. 8,648,867	GPU Gems
	<p>Listing 31-1.</p> <p>Listing 31-2. C++ Code to Set Up and Run a Reaction-Diffusion Simulation on the GPU <i>See the source code on the accompanying CD for more details.</i></p> <hr/> <pre> PUGBuffer *rdBuffer; PUGProgram *rdProgram; void init_rd(){ // Call this once. pugInit(); // Start up the GPU framework // Create a "double-buffered" PUGBuffer. Two buffers allow the // simulation to alternate using one as input, one as output PUGBuffer *rdBuffer = pugAllocateBuffer(width, height, PUG_READWRITE, 4, true); PUGProgram *rdProgram = pugLoadProgram("rd.cg", "rd"); pugBindFloat(rdProgram, "DuDv", du, dv); // bind parameters pugBindFloat(rdProgram, "F", F); pugBindFloat(rdProgram, "k", k); </pre> <hr/> <p>Listing 31-2 (continued). C++ Code to Set Up and Run a Reaction-Diffusion Simulation on the GPU</p> <pre> // Initialize the state of the simulation with values in array pugInitBuffer(rdBuffer, array); } void update_rd(){ // Call this every iteration pugBindStream(rdProgram, "concentration", rdBuffer, currentSource); PUGRect range(0, width, 0, height); pugRunProgram(rdProgram, rdBuffer, range, currentTarget); std::swap(currentSource, currentTarget); } </pre> <hr/> <p>Listing 31-2.</p> <p>31.4 From Analogies to Implementation</p> <p>By now you have a high-level understanding of how most GPGPU programs operate on current GPUs. The analogies from the previous section provide us a general implementation plan. Now, so</p>

Ex. A11 – Invalidity of U.S. Patent No. 8,648,867 – GPU Gems

U.S. Patent No. 8,648,867	GPU Gems
	<p>that you can start putting the GPU to work, we dig into the nitty-gritty implementation details that you will need in practice.</p> <p>31.4.1 Putting It All Together: A Basic GPGPU Framework</p> <p>To make this easy, we've put together a very basic framework in C++ that you can use to write GPGPU programs. You'll find the framework on the CD included with this book. The framework, called pug, incorporates all of the analogies from the previous section in order to abstract GPGPU programming in a manner more accessible to experienced CPU programmers.</p> <p>Initializing and Finalizing a GPGPU Application</p> <p>The first step in a GPGPU application is to initialize the GPU. This is very easy in the framework: just call pugInit(). When the application is finished with the GPU, it can clean up the memory used by the graphics API by calling pugCleanup().</p> <p>Specifying Kernels</p> <p>Kernels in our framework are written in the Cg language. Although Cg is designed for graphics, it is based on the C language and therefore is very easy to learn. For more information, we recommend The Cg Tutorial (Fernando and Kilgard 2003) and the documentation included with the Cg distribution (NVIDIA 2004). There are very few graphics-specific keywords in Cg. We provide a Cg file, pug.cg, that you can include in your kernel files. This file defines a structure called Stream that abstracts the most graphics-centric concept: texture fetching. Stream is a wrapper for the texture sampler Cg type. Calling one of the value() functions of the Stream structure gives the value of a stream element.</p> <p>To load and initialize a kernel program from a file, use the pugLoadProgram() function. The function returns a pointer to a PUGProgram structure, which you will need to store and pass to the other framework functions to bind constants and streams to the kernel and to run it.</p> <p>Stream Management</p> <p>Arrays of data in the framework are called buffers. The pugAllocateBuffer() function creates a new buffer. Buffers can be read-only, write-only, or read-write, and this can be specified using the</p>

Ex. A11 – Invalidity of U.S. Patent No. 8,648,867 – GPU Gems

U.S. Patent No. 8,648,867	GPU Gems
	<p>mode parameter of this function. This function returns a pointer to a PUGBuffer structure, which can be passed to a number of functions in the framework, including the following two.</p> <p>To load initial data into a buffer, pass the data to the pugInitBuffer() function.</p> <p>To bind an input stream for a kernel to a buffer, call pugBindStream(), which takes as arguments a pointer to the PUGProgram to bind to, the PUGBuffer pointer, and a string containing the name of the Stream parameter in the Cg kernel program to which this PUGBuffer should be bound.</p> <p>Specifying Computational Domain and Range</p> <p>To specify the domain and range of a computation, define a PUGRect structure with the coordinates of the corners of the rectangle that should be used as either the domain or range. Multiple domains can be specified. These will generate additional texture coordinates that the kernel program can use as needed.</p> <p>To bind a domain, call pugBindDomain(), passing it the PUGProgram, a string parameter name for this domain defined in the kernel Cg program, and the PUGRect for the domain.</p> <p>There can be only one range for a kernel, due to the inability of fragment programs to scatter. To specify the range, simply pass a PUGRect to the range parameter of the pugRunProgram() function. 503–04; <i>see also</i> 497–502 .</p> <p>31.5 A Simple Example</p> <p>As a basic but nontrivial GPGPU example, we use a simulation of a phenomenon known as chemical reaction-diffusion. Reaction-diffusion is a model of how the concentrations of two or more reactants in a solution evolve in space and time as they undergo the processes of chemical reaction and diffusion. The reaction-diffusion model we use is called the Grey-Scott model (Pearson 1993) and involves just two chemical reactants. This phenomenological model does not represent a particular real chemical reaction; it serves as a simple model for studying the general reaction-diffusion phenomenon. Figure 31-3 shows the results of many iterations of the Grey-Scott model.</p>

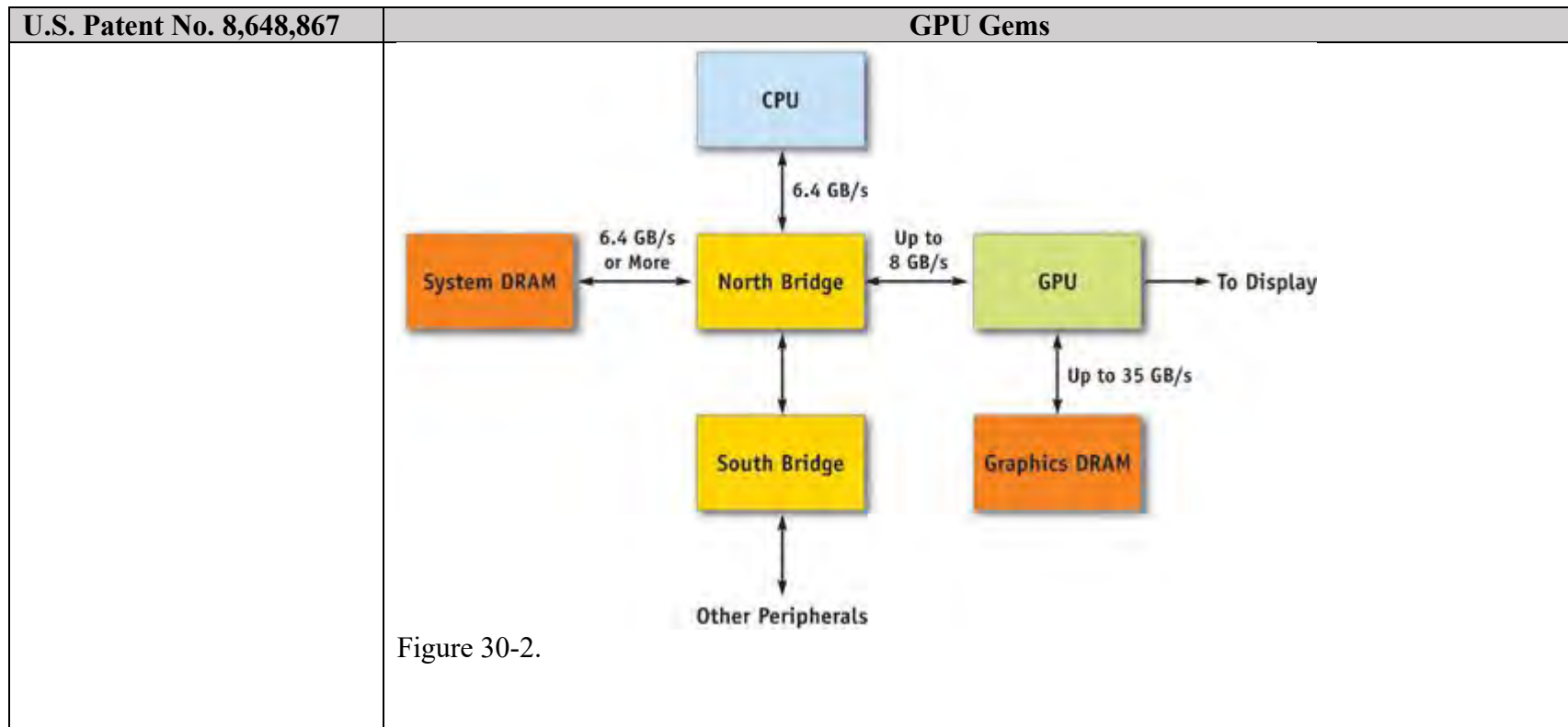
Ex. A11 – Invalidity of U.S. Patent No. 8,648,867 – GPU Gems

U.S. Patent No. 8,648,867	GPU Gems
	<p>Grey-Scott consists of two simple partial differential equations that govern the evolution of the concentrations of two chemical reactants, U and V:</p> $\frac{\partial U}{\partial t} = D_u \nabla^2 U - UV^2 + F(1 - U),$ $\frac{\partial V}{\partial t} = D_v \nabla^2 V + UV^2 - (F + k)V.$ <div style="display: flex; justify-content: center; gap: 50px; margin-top: -10px;"> Diffusion Reaction </div> <p>Here k and F are constants; Du and Dv are the diffusion rates of the reactants; and ∇^2 is the Laplacian operator, which represents diffusion. The Laplacian operator appears commonly in physics, most notably in the form of diffusion equations, such as the heat equation. A finite difference form of the Laplacian operator applied to a scalar field p on a two-dimensional Cartesian grid (with indices i, j, and cell size x) is</p> $\nabla^2 p = \frac{p_{i+1,j} + p_{i-1,j} + p_{i,j+1} + p_{i,j-1} - 4p_{i,j}}{(\Delta x)^2}$ <p>The implementation of the Grey-Scott model is fairly simple. There is a single data stream: the U and V chemical concentrations are stored in two channels of a single texture that represents a discrete spatial grid. This stream serves as input to a simple kernel, which implements the preceding equations in discrete form. The kernel is shown in Listing 31-1. C code that uses the framework to implement the simulation is shown in Listing 31-2.</p> <p>505–08.</p> <p><i>See also</i> 457–70 (giving historical background on GPU accelerators as a computational platform built for parallelism); 520–45 (discussing a stream model for GPU processing); 595–785 (providing additional examples of numerical simulations performed with a GPU accelerator).</p>

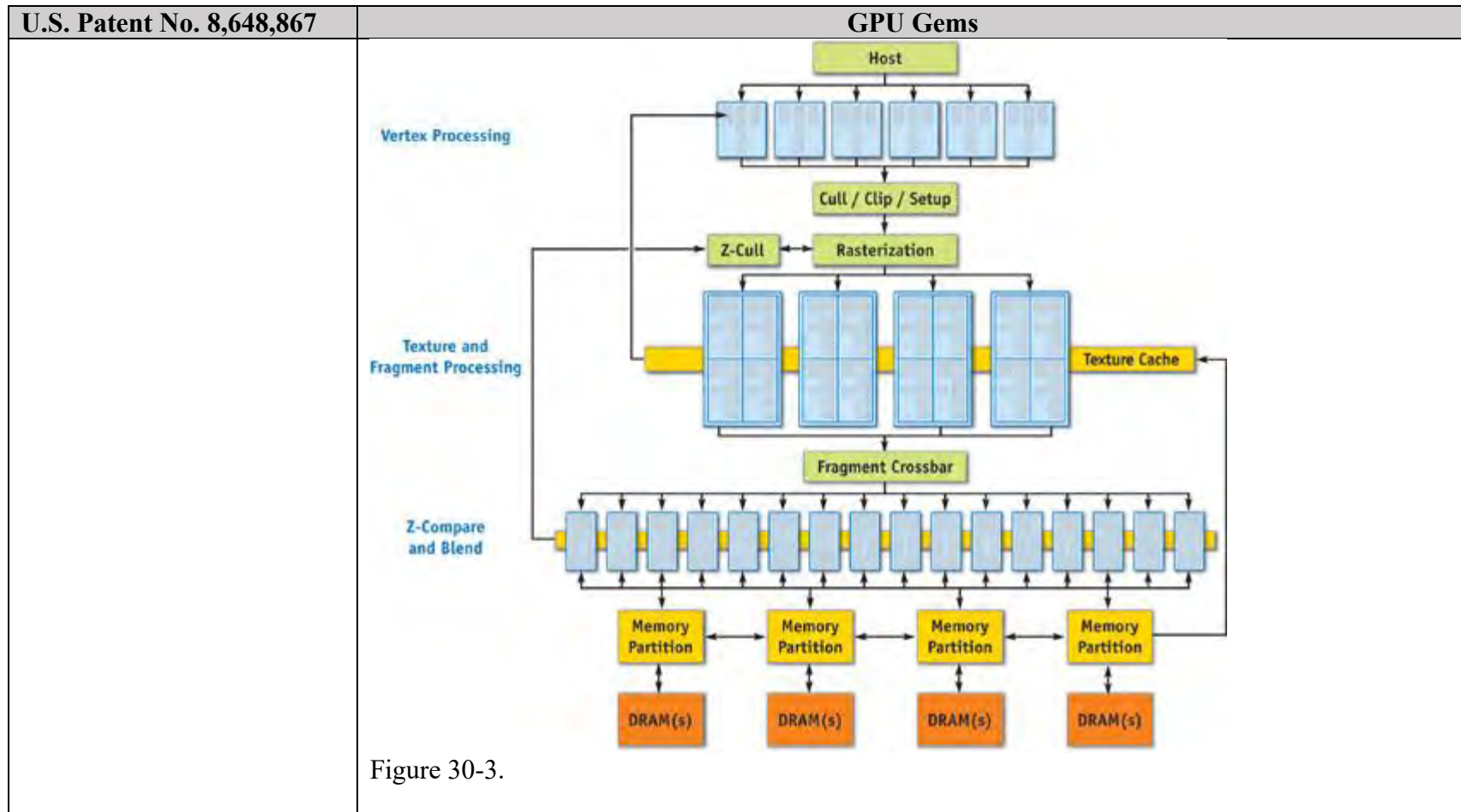
Ex. A11 – Invalidity of U.S. Patent No. 8,648,867 – GPU Gems

U.S. Patent No. 8,648,867	GPU Gems
	<p>To the extent Plaintiff contends that this claim element is not expressly or inherently disclosed, or is not obvious in view of GPU Gems and the knowledge of a POSITA, it would have been obvious to combine the teachings of GPU Gems with any of the prior art references in NVIDIA’s Invalidity Contentions and any supplements thereto and the relevant section of charts for other prior art for the ’867 patent in a manner that would result in the subject matter of this limitation given, at the very least, the person's understanding of the state of the art, the problems addressed and solved in the prior art, and the teachings of GPU Gems. Specific examples of such references are cited and explained in these invalidity contentions, including the cover pleading and charts A1–A13, and Appendix A.</p>
<p>16[d]: storing, by the accelerator controller, the first output data into a second partition, referenced by a second pointer, of the accelerator memory; and</p>	<p>GPU Gems discloses “storing, by the accelerator controller, the first output data into a second partition, referenced by a second pointer, of the accelerator memory.” <i>See e.g.:</i></p> <p><i>See 16[b].</i></p> <p><i>As a non-limiting example, GPU Gems discloses storage at the direction and/or under the control of the accelerator controller (e.g., vertex processors of a GPU, see 16[b]) of first output data into a “second partition” of the accelerator memory (e.g., by allocating and storing input data in accelerator memory) referenced by a “second pointer” (e.g., currentTarget).</i></p>

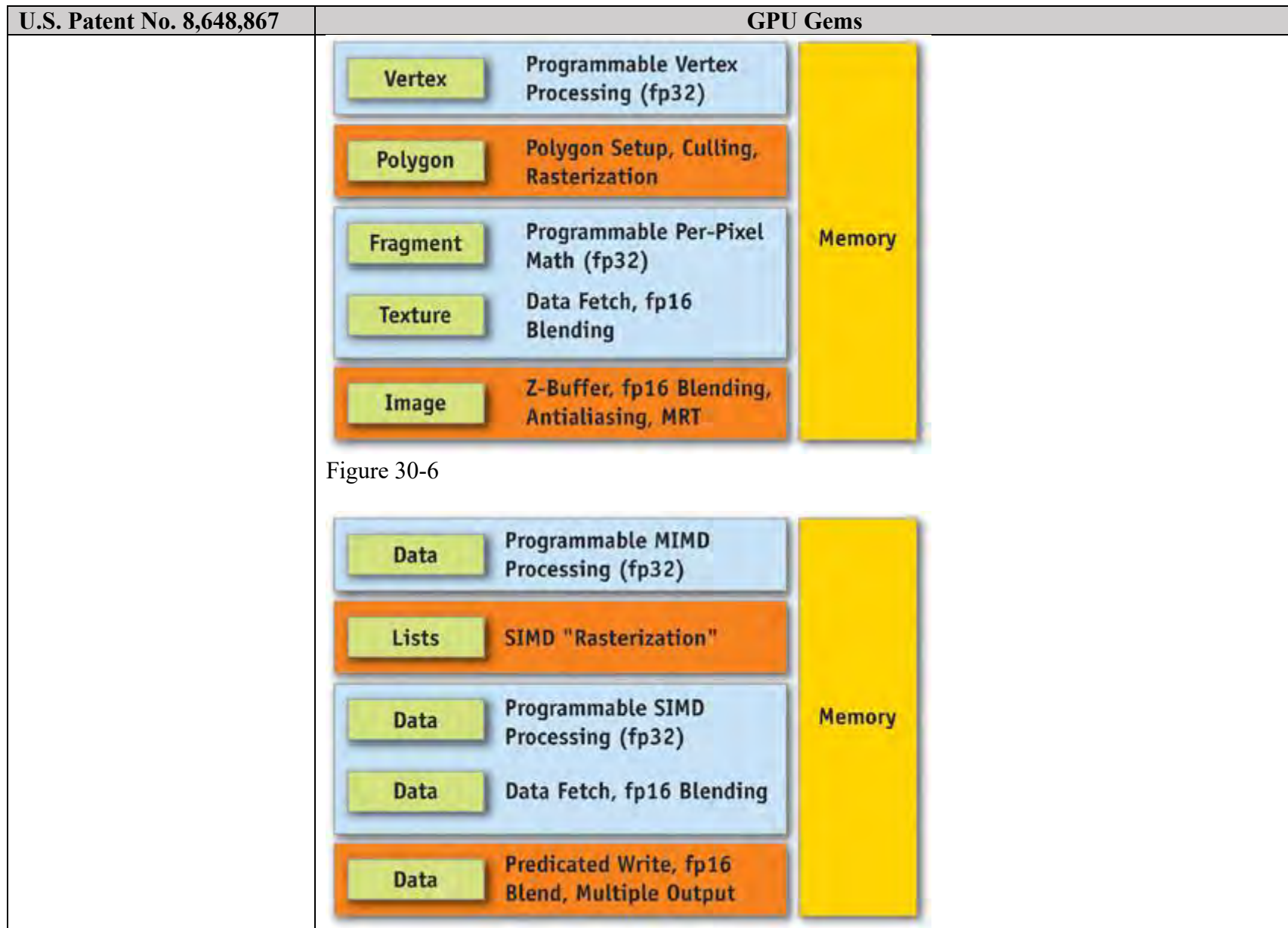
Ex. A11 – Invalidity of U.S. Patent No. 8,648,867 – GPU Gems



Ex. A11 – Invalidity of U.S. Patent No. 8,648,867 – GPU Gems



Ex. A11 – Invalidity of U.S. Patent No. 8,648,867 – GPU Gems



Ex. A11 – Invalidity of U.S. Patent No. 8,648,867 – GPU Gems

U.S. Patent No. 8,648,867	GPU Gems
	<p data-bbox="611 235 768 267">Figure 30-7.</p>  <p data-bbox="611 685 768 717">Figure 31-3.</p>

Ex. A11 – Invalidity of U.S. Patent No. 8,648,867 – GPU Gems

U.S. Patent No. 8,648,867	GPU Gems
	<p>Listing 31-1. The Reaction-Diffusion Kernel Program</p> <hr/> <pre> float4 rd(float2 coords : DOMAIN, uniform stream concentration, uniform float2 DuDv, uniform float F, uniform float k) : RANGE { float2 center = concentration.value2(coords).xy; float2 diffusion = concentration.value2(coords + half2(1, 0)); diffusion += concentration.value2(coords + half2(-1, 0)); diffusion += concentration.value2(coords + half2(0, 1)); diffusion += concentration.value2(coords + half2(0, -1)); // Average and scale by diffusion coeffs diffusion *= 0.25f * DuDv; float2 reaction = center.xx * center.yy * center.yy; reaction.x = -1; reaction.x += (1 - DuDv.x) * center.x + F * (1 - center.x); reaction.y += (-F - k + (1 - DuDv.y)) * center.y; // Now add the diffusion to the reaction to get the result. return float4(diffusion + reaction, 0, 0); } </pre> <hr/> <p>Listing 31-1.</p>

Ex. A11 – Invalidity of U.S. Patent No. 8,648,867 – GPU Gems

U.S. Patent No. 8,648,867	GPU Gems
	<p>Listing 31-2. C++ Code to Set Up and Run a Reaction-Diffusion Simulation on the GPU <i>See the source code on the accompanying CD for more details.</i></p> <hr/> <pre> PUGBuffer *rdBuffer; PUGProgram *rdProgram; void init_rd(){ // Call this once. pugInit(); // Start up the GPU framework // Create a "double-buffered" PUGBuffer. Two buffers allow the // simulation to alternate using one as input, one as output PUGBuffer *rdBuffer = pugAllocateBuffer(width, height, PUG_READWRITE, 4, true); PUGProgram *rdProgram = pugLoadProgram("rd.cg", "rd"); pugBindFloat(rdProgram, "DuDv", du, dv); // bind parameters pugBindFloat(rdProgram, "F", F); pugBindFloat(rdProgram, "k", k); </pre> <hr/> <p>Listing 31-2 (continued). C++ Code to Set Up and Run a Reaction-Diffusion Simulation on the GPU</p> <hr/> <pre> // Initialize the state of the simulation with values in array pugInitBuffer(rdBuffer, array); } void update_rd(){ // Call this every iteration pugBindStream(rdProgram, "concentration", rdBuffer, currentSource); PUGRect range(0, width, 0, height); pugRunProgram(rdProgram, rdBuffer, range, currentTarget); std::swap(currentSource, currentTarget); } </pre> <hr/> <p>Listing 31-2 (annotated).</p> <p>30.2.2 Functional Block Diagram for Non-Graphics Operations</p> <p>As graphics hardware becomes more and more programmable, applications unrelated to the standard polygon pipeline (as described in the preceding section) are starting to present themselves as candidates for execution on GPUs.</p>

Ex. A11 – Invalidity of U.S. Patent No. 8,648,867 – GPU Gems

U.S. Patent No. 8,648,867	GPU Gems
	<p>Figure 30-6 shows a simplified view of the GeForce 6 Series architecture, when used as a graphics pipeline. It contains a programmable vertex engine, a programmable fragment engine, a texture load/filter engine, and a depth-compare/blending data write engine.</p> <p>In this alternative view, a GPU can be seen as a large amount of programmable floating-point horsepower and memory bandwidth that can be exploited for compute-intensive applications completely unrelated to computer graphics.</p> <p>Figure 30-7 shows another way to view the GeForce 6 Series architecture. When used for non-graphics applications, it can be viewed as two programmable blocks that run serially: the vertex processor and the fragment processor, both with support for fp32 operands and intermediate values. Both use the texture unit as a random-access data fetch unit and access data at a phenomenal 35 GB/sec (550 MHz DDR memory clock x 256 bits per clock cycle x 2 transfers per clock cycle). In addition, both the vertex and the fragment processor are highly computationally capable. (Performance details follow in Section 30.4.)</p> <p>The vertex processor operates on data, passing it directly to the fragment processor, or by using the rasterizer to expand the data into interpolated values. At this point, each triangle (or point) from the vertex processor has become one or more fragments.</p> <p>Before a fragment reaches the fragment processor, the z-cull unit compares the pixel's depth with the values that already exist in the depth buffer. If the pixel's depth is greater, the pixel will not be visible, and there is no point shading that fragment, so the fragment processor isn't even executed. (This optimization happens only if it's clear that the fragment processor isn't going to modify the fragment's depth.) Thinking in a general-purpose sense, this early culling feature makes it possible to quickly decide to skip work on specific fragments based on a scalar test. Chapter 34 of this book, "GPU Flow-Control Idioms," explains how to take advantage of this feature to efficiently predicate work for general-purpose computations.</p> <p>After the fragment processor runs on a potential pixel (still a "fragment" because it has not yet reached the frame buffer), the fragment must pass a number of tests in order to move farther down</p>

Ex. A11 – Invalidity of U.S. Patent No. 8,648,867 – GPU Gems

U.S. Patent No. 8,648,867	GPU Gems
	<p>the pipeline. (There may also be more than one fragment that comes out of the fragment processor if multiple render targets [MRTs] are being used. Up to four MRTs can be used to write out large amounts of data—up to 16 scalar floating-point values at a time, for example—plus depth.)</p> <p>First, the scissor test rejects the fragment if it lies outside a specified subrectangle of the frame buffer. Although the popular graphics APIs define scissoring at this location in the pipeline, it is more efficient to perform the scissor test in the rasterizer. Scissoring in x and y actually happens in the rasterizer, before fragment processing, and z scissoring happens during z-cull. This avoids all fragment processor work on scissored (rejected) pixels. Scissoring is rarely useful for general-purpose computation because general-purpose programmers typically draw rectangles to perform computations in the first place.</p> <p>Next, the fragment's depth is compared with the depth in the frame buffer. If the depth test passes, the fragment moves on in the pipeline. Optionally, the depth value in the frame buffer can be replaced at this stage.</p> <p>After this, the fragment can optionally test and modify what is known as the stencil buffer, which stores an integer value per pixel. The stencil buffer was originally intended to allow programmers to mask off certain pixels (for example, to restrict drawing to a cockpit's windshield), but it has found other uses as a way to count values by incrementing or decrementing the existing value. This feature is used for stencil shadow volumes, for example.</p> <p>If the fragment passes the depth and stencil tests, it can then optionally modify the contents of the frame buffer using the blend function. A blend function can be described as</p> $\text{out} = \text{src} * \text{srcOp} + \text{dst} * \text{dstOp}$ <p>where source is the fragment color flowing down the pipeline; dst is the color value in the frame buffer; and srcOp and dstOp can be specified to be constants, source color components, or destination color components. Full blend functionality is supported for all pixel formats up to fp16x4. However, fp32 frame buffers don't support blending—only updating the buffer is allowed.</p>

Ex. A11 – Invalidity of U.S. Patent No. 8,648,867 – GPU Gems

U.S. Patent No. 8,648,867	GPU Gems
	<p>Finally, a feature called occlusion query makes it possible to quickly determine if any of the fragments that would be rendered in a particular computation would cause results to be written to the frame buffer. (Recall that fragments that do not pass the z-test don't have any effect on the values in the frame buffer.) Traditionally, the occlusion query test is used to allow graphics applications to avoid making draw calls for occluded objects, but it is useful for GPGPU applications as well. For instance, if the depth test is used to determine which outputs need to be updated in a sparse array, updating depth can be used to indicate when a given output has converged and no further work is needed. In this case, occlusion query can be used to tell when all output calculations are done. See Chapter 34 of this book, "GPU Flow-Control Idioms," for further information about this idea.</p> <p>478–480.</p> <p>31.5 A Simple Example</p> <p>As a basic but nontrivial GPGPU example, we use a simulation of a phenomenon known as chemical reaction-diffusion. Reaction-diffusion is a model of how the concentrations of two or more reactants in a solution evolve in space and time as they undergo the processes of chemical reaction and diffusion. The reaction-diffusion model we use is called the Grey-Scott model (Pearson 1993) and involves just two chemical reactants. This phenomenological model does not represent a particular real chemical reaction; it serves as a simple model for studying the general reaction-diffusion phenomenon. Figure 31-3 shows the results of many iterations of the Grey-Scott model.</p> <p>Grey-Scott consists of two simple partial differential equations that govern the evolution of the concentrations of two chemical reactants, U and V:</p> $\begin{aligned}\frac{\partial U}{\partial t} &= \boxed{D_u \nabla^2 U} - \boxed{UV^2 + F(1 - U)}, \\ \frac{\partial V}{\partial t} &= \boxed{D_v \nabla^2 V} + \boxed{UV^2 - (F + k)V}.\end{aligned}$ <p style="text-align: center;">Diffusion Reaction</p>

Ex. A11 – Invalidity of U.S. Patent No. 8,648,867 – GPU Gems

U.S. Patent No. 8,648,867	GPU Gems
	<p>Here k and F are constants; Du and Dv are the diffusion rates of the reactants; and ∇^2 is the Laplacian operator, which represents diffusion. The Laplacian operator appears commonly in physics, most notably in the form of diffusion equations, such as the heat equation. A finite difference form of the Laplacian operator applied to a scalar field p on a two-dimensional Cartesian grid (with indices i, j, and cell size x) is</p> $\nabla^2 p = \frac{p_{i+1,j} + p_{i-1,j} + p_{i,j+1} + p_{i,j-1} - 4p_{i,j}}{(\Delta x)^2}$ <p>The implementation of the Grey-Scott model is fairly simple. There is a single data stream: the U and V chemical concentrations are stored in two channels of a single texture that represents a discrete spatial grid. This stream serves as input to a simple kernel, which implements the preceding equations in discrete form. The kernel is shown in Listing 31-1. C code that uses the framework to implement the simulation is shown in Listing 31-2.</p> <p>505–08.</p> <p>To the extent Plaintiff contends that this claim element is not expressly or inherently disclosed, or is not obvious in view of GPU Gems and the knowledge of a POSITA, it would have been obvious to combine the teachings of GPU Gems with any of the prior art references in NVIDIA’s Invalidity Contentions and any supplements thereto and the relevant section of charts for other prior art for the ’867 patent in a manner that would result in the subject matter of this limitation given, at the very least, the person's understanding of the state of the art, the problems addressed and solved in the prior art, and the teachings of GPU Gems. Specific examples of such references are cited and explained in these invalidity contentions, including the cover pleading and charts A1–A13, and Appendix A.</p>
16[e]: swapping the first pointer with the second pointer at the end of the first computational cycle, such that the first output data becomes an input for a	<p>GPU Gems discloses “swapping the first pointer with the second pointer at the end of the first computational cycle, such that the first output data becomes an input for a second computational cycle of the numerical simulation.” <i>See e.g.:</i></p> <p><i>As a non-limiting example, GPU Gems discloses swapping the first pointer with the second pointer (e.g., using <code>std::swap(currentSource, currentTarget)</code>) at the end of the first</i></p>

Ex. A11 – Invalidity of U.S. Patent No. 8,648,867 – GPU Gems

U.S. Patent No. 8,648,867	GPU Gems
second computational cycle of the numerical simulation.	<p data-bbox="604 233 1766 310"><i>computation cycle (e.g., after an iteration of <code>update_rd()</code>, solving a reaction-diffusion equation).</i></p> <div data-bbox="611 344 1604 680"></div> <p data-bbox="604 685 768 719">Figure 31-3.</p>

Ex. A11 – Invalidity of U.S. Patent No. 8,648,867 – GPU Gems

U.S. Patent No. 8,648,867	GPU Gems
	<p>Listing 31-1. The Reaction-Diffusion Kernel Program</p> <hr/> <pre> float4 rd(float2 coords : DOMAIN, uniform stream concentration, uniform float2 DuDv, uniform float F, uniform float k) : RANGE { float2 center = concentration.value2(coords).xy; float2 diffusion = concentration.value2(coords + half2(1, 0)); diffusion += concentration.value2(coords + half2(-1, 0)); diffusion += concentration.value2(coords + half2(0, 1)); diffusion += concentration.value2(coords + half2(0, -1)); // Average and scale by diffusion coeffs diffusion *= 0.25f * DuDv; float2 reaction = center.xx * center.yy * center.yy; reaction.x = -1; reaction.x += (1 - DuDv.x) * center.x + F * (1 - center.x); reaction.y += (-F - k + (1 - DuDv.y)) * center.y; // Now add the diffusion to the reaction to get the result. return float4(diffusion + reaction, 0, 0); } </pre> <hr/> <p>Listing 31-1.</p>

Ex. A11 – Invalidity of U.S. Patent No. 8,648,867 – GPU Gems

U.S. Patent No. 8,648,867	GPU Gems
	<p>Listing 31-2. C++ Code to Set Up and Run a Reaction-Diffusion Simulation on the GPU <i>See the source code on the accompanying CD for more details.</i></p> <hr/> <pre> PUGBuffer *rdBuffer; PUGProgram *rdProgram; void init_rd(){ // Call this once. pugInit(); // Start up the GPU framework // Create a "double-buffered" PUGBuffer. Two buffers allow the // simulation to alternate using one as input, one as output PUGBuffer *rdBuffer = pugAllocateBuffer(width, height, PUG_READWRITE, 4, true); PUGProgram *rdProgram = pugLoadProgram("rd.cg", "rd"); pugBindFloat(rdProgram, "DuDv", du, dv); // bind parameters pugBindFloat(rdProgram, "F", F); pugBindFloat(rdProgram, "k", k); </pre> <hr/> <p>Listing 31-2 (continued). C++ Code to Set Up and Run a Reaction-Diffusion Simulation on the GPU</p> <hr/> <pre> // Initialize the state of the simulation with values in array pugInitBuffer(rdBuffer, array); } void update_rd(){ // Call this every iteration pugBindStream(rdProgram, "concentration", rdBuffer, currentSource); PUGRect range(0, width, 0, height); pugRunProgram(rdProgram, rdBuffer, range, currentTarget); std::swap(currentSource, currentTarget); } </pre> <hr/> <p>Listing 31-2 (annotated).</p>

Ex. A11 – Invalidity of U.S. Patent No. 8,648,867 – GPU Gems

U.S. Patent No. 8,648,867	GPU Gems
	<p>Listing 44-1. The Conjugate Gradient Solver</p> <hr/> <pre> void clCGSolver::solveInit() { Matrix->matrixVectorOp(CL_SUB, X, B, R); // R = A * x - b R->multiply(-1); // R = -R R->clone(P); // P = R R->reduceAdd(R, Rho); // rho = sum(R * R); } void clCGSolver::solveIteration() { Matrix->matrixVectorOp(CL_NULL, P, NULL, Q); // Q = Ap; P->reduceAdd(Q, Temp); // temp = sum(P * Q); Rho->div(Temp, Alpha); // alpha = rho/temp; X->addVector(P, X, 1, Alpha); // X = X + alpha * P R->subtractVector(Q, R, 1, Alpha); // R = R - alpha * Q R->reduceAdd(R, NewRho); // newrho = sum(R * R); NewRho->divZ(Rho, Beta); // beta = newrho/rho R->addVector(P, P, 1, Beta); // P = R + beta * P; clFloat *temp; temp=NewRho; NewRho=Rho; Rho=temp; // swap rho and newrho pointers } void clCGSolver::solve(int maxI) { solveInit(); for (int i = 0; i < maxI; i++) solveIteration(); } int clCGSolver::solve(float rhoTresh, int maxI) { solveInit(); Rho->clone(NewRho); for (int i = 0; i < maxI && NewRho.getData() > rhoTresh; i++) solveIteration(); return i; } </pre> <hr/> <p>Listing 44-1.</p> <p>2.6.3 Memory-Free Terrain Synthesis</p>

Ex. A11 – Invalidity of U.S. Patent No. 8,648,867 – GPU Gems

U.S. Patent No. 8,648,867	GPU Gems
	<p>Fractal terrain synthesis is so fast on the GPU that we can consider regenerating the clipmap levels during every frame, thereby saving video memory. We allocate two textures, T 1 and T 2, and toggle between them as follows. Let T 1 initially contain the coarse geometry used to seed the synthesis process. First, we use T 1 as a source texture in the update pixel shader to upsample and synthesize the next-finer level into the destination texture T 2. Second, we use T 1 as a vertex texture to render its clipmap ring. Then, we swap the roles of the two textures and iterate the process until all levels are synthesized and rendered. Initial experiments are extremely promising, with frame rates of about 59 frames/sec using L = 9 levels.</p> <p>44.</p> <p>31.5 A Simple Example</p> <p>As a basic but nontrivial GPGPU example, we use a simulation of a phenomenon known as chemical reaction-diffusion. Reaction-diffusion is a model of how the concentrations of two or more reactants in a solution evolve in space and time as they undergo the processes of chemical reaction and diffusion. The reaction-diffusion model we use is called the Grey-Scott model (Pearson 1993) and involves just two chemical reactants. This phenomenological model does not represent a particular real chemical reaction; it serves as a simple model for studying the general reaction-diffusion phenomenon. Figure 31-3 shows the results of many iterations of the Grey-Scott model.</p> <p>Grey-Scott consists of two simple partial differential equations that govern the evolution of the concentrations of two chemical reactants, U and V:</p> $\begin{aligned}\frac{\partial U}{\partial t} &= \boxed{D_u \nabla^2 U} - \boxed{UV^2 + F(1 - U)}, \\ \frac{\partial V}{\partial t} &= \boxed{D_v \nabla^2 V} + \boxed{UV^2 - (F + k)V}.\end{aligned}$ <p style="text-align: center;">Diffusion Reaction</p>

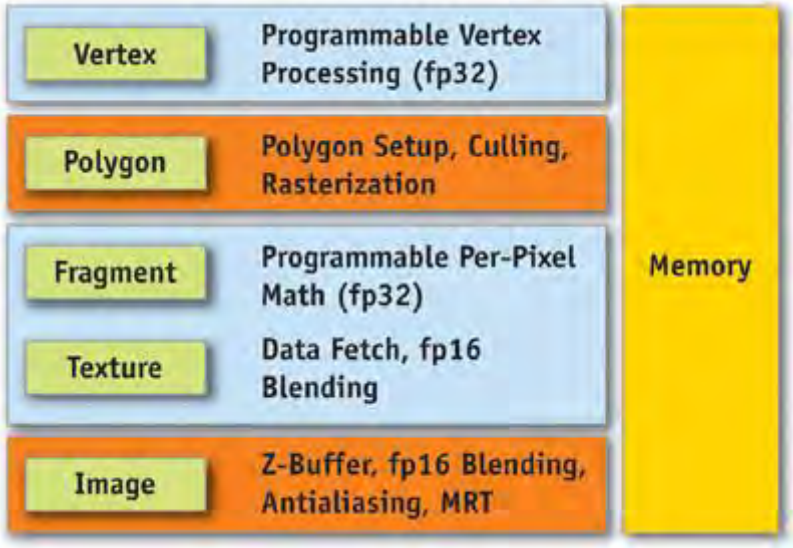
Ex. A11 – Invalidity of U.S. Patent No. 8,648,867 – GPU Gems

U.S. Patent No. 8,648,867	GPU Gems
	<p>Here k and F are constants; Du and Dv are the diffusion rates of the reactants; and ∇^2 is the Laplacian operator, which represents diffusion. The Laplacian operator appears commonly in physics, most notably in the form of diffusion equations, such as the heat equation. A finite difference form of the Laplacian operator applied to a scalar field p on a two-dimensional Cartesian grid (with indices i, j, and cell size x) is</p> $\nabla^2 p = \frac{p_{i+1,j} + p_{i-1,j} + p_{i,j+1} + p_{i,j-1} - 4p_{i,j}}{(\Delta x)^2}$ <p>The implementation of the Grey-Scott model is fairly simple. There is a single data stream: the U and V chemical concentrations are stored in two channels of a single texture that represents a discrete spatial grid. This stream serves as input to a simple kernel, which implements the preceding equations in discrete form. The kernel is shown in Listing 31-1. C code that uses the framework to implement the simulation is shown in Listing 31-2.</p> <p>505–08.</p> <p>44.3.5 Conjugate Gradient Solver</p> <p>We now possess an easy-to-use class framework that completely abstracts the underlying hardware implementation, so we can easily write more-complex algorithms such as the conjugate gradient linear system solver. Using our implementation, the complete GPU conjugate gradient class appears in Listing 44-1.</p> <p>713.</p> <p>46.4.1 Implementing Bitonic Merge Sort</p> <p>For the sample application of sorting particles according to camera distance, we implement a small fragment program that computes the sorting keys (distance) and indices (particle numbers) and packs two of them into one RGBA texture value. Because we must touch every item to sort anyway, we take the opportunity and also perform the very first sorting pass while writing the packed representation. The first pass simply has to produce alternating ascending/descending pairs of keys—an operation that is internal to each fragment (see Figure 46-3).</p>

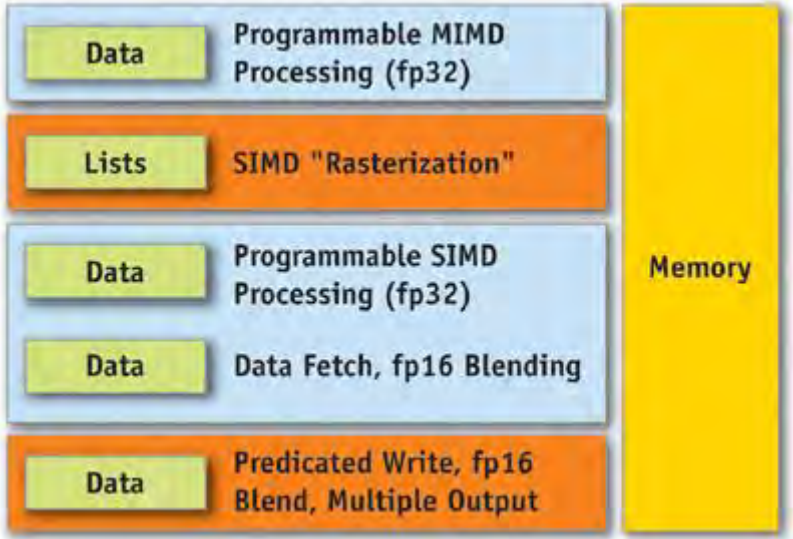
Ex. A11 – Invalidity of U.S. Patent No. 8,648,867 – GPU Gems

U.S. Patent No. 8,648,867	GPU Gems
	<p>All quads for one sorting pass are recorded in a display list. To sort the entire field, we just call the next list and swap the buffers until we're done. The display list records the vertices of the quads and two texture coordinates. The first texture coordinate holds all data that varies per quad (that is, the fragment position, the search direction, and the compare operation flag that is to be interpolated). The second texture coordinate holds data that is constant for the quad. Because we pack two consecutive items into one fragment, in the last pass of each stage, the fragment shader accesses the same items as the previous pass (compare Figure 46-4). We account for this by folding both passes into one, using a special shader for this case. Note that the fragment program does not need to perform a conversion from 1D indices to 2D texture coordinates, because this is done implicitly by the texture coordinates of the quads. Listing 46-2 shows the GLSL code for the special program.</p> <p>The program for passes greater than 1 along the rows skips the last compare operation in Listing 46-2 and returns the result variable. The program for passes along the columns differs just in building the adr variable: here x is 0 and y is computed according to the search direction.</p> <p>In summary, we need four programs to perform the sorting: The first one computes the key/index pairs and does the first sorting pass. The second performs row sorting for passes 0 and 1 of each stage simultaneously. The third does row sorting for all passes greater than 1 of each stage, and the fourth performs the column sort.</p> <p>When the sort is done, we have a permutation of the key/index pairs. The application can now reorder any data buffer according to the sorting criterion by simple indexed lookup.</p> <p>743.</p> <p>To the extent Plaintiff contends that this claim element is not expressly or inherently disclosed, or is not obvious in view of GPU Gems and the knowledge of a POSITA, it would have been obvious to combine the teachings of GPU Gems with any of the prior art references in NVIDIA's Invalidity Contentions and any supplements thereto and the relevant section of charts for other prior art for the '867 patent in a manner that would result in the subject matter of this limitation given, at the very least, the person's understanding of the state of the art, the problems addressed and solved in the prior art, and the teachings of GPU Gems. Specific examples of such references</p>

Ex. A11 – Invalidity of U.S. Patent No. 8,648,867 – GPU Gems

U.S. Patent No. 8,648,867	GPU Gems
	<p>are cited and explained in these invalidity contentions, including the cover pleading and charts A1–A13, and Appendix A.</p>
<p>17: The method as claimed in claim 16, further comprising: sending, by the accelerator controller, instructions for performing the at least one calculation to the at least one graphics processing unit.</p>	<p>GPU Gems discloses “sending, by the accelerator controller, instructions for performing the at least one calculation to the at least one graphics processing unit.” <i>See e.g.:</i></p> <p><i>As a non-limiting example, GPU Gems discloses an accelerator controller (e.g., vertex processors and/or a rasterizer) that sends instructions (e.g., kernel programs and/or fragment programs) to at least one graphics processing unit (e.g., a fragment processor).</i></p>  <p>Figure 30-6.</p>

Ex. A11 – Invalidity of U.S. Patent No. 8,648,867 – GPU Gems

U.S. Patent No. 8,648,867	GPU Gems
	 <p>Figure 30-7.</p> <p>30.2.2 Functional Block Diagram for Non-Graphics Operations</p> <p>As graphics hardware becomes more and more programmable, applications unrelated to the standard polygon pipeline (as described in the preceding section) are starting to present themselves as candidates for execution on GPUs.</p> <p>Figure 30-6 shows a simplified view of the GeForce 6 Series architecture, when used as a graphics pipeline. It contains a programmable vertex engine, a programmable fragment engine, a texture load/filter engine, and a depth-compare/blending data write engine.</p> <p>In this alternative view, a GPU can be seen as a large amount of programmable floating-point horsepower and memory bandwidth that can be exploited for compute-intensive applications completely unrelated to computer graphics.</p> <p>Figure 30-7 shows another way to view the GeForce 6 Series architecture. When used for non-graphics applications, it can be viewed as two programmable blocks that run serially: the vertex</p>

Ex. A11 – Invalidity of U.S. Patent No. 8,648,867 – GPU Gems

U.S. Patent No. 8,648,867	GPU Gems
	<p>processor and the fragment processor, both with support for fp32 operands and intermediate values. Both use the texture unit as a random-access data fetch unit and access data at a phenomenal 35 GB/sec (550 MHz DDR memory clock x 256 bits per clock cycle x 2 transfers per clock cycle). In addition, both the vertex and the fragment processor are highly computationally capable. (Performance details follow in Section 30.4.)</p> <p>The vertex processor operates on data, passing it directly to the fragment processor, or by using the rasterizer to expand the data into interpolated values. At this point, each triangle (or point) from the vertex processor has become one or more fragments.</p> <p>Before a fragment reaches the fragment processor, the z-cull unit compares the pixel's depth with the values that already exist in the depth buffer. If the pixel's depth is greater, the pixel will not be visible, and there is no point shading that fragment, so the fragment processor isn't even executed. (This optimization happens only if it's clear that the fragment processor isn't going to modify the fragment's depth.) Thinking in a general-purpose sense, this early culling feature makes it possible to quickly decide to skip work on specific fragments based on a scalar test. Chapter 34 of this book, "GPU Flow-Control Idioms," explains how to take advantage of this feature to efficiently predicate work for general-purpose computations.</p> <p>After the fragment processor runs on a potential pixel (still a "fragment" because it has not yet reached the frame buffer), the fragment must pass a number of tests in order to move farther down the pipeline. (There may also be more than one fragment that comes out of the fragment processor if multiple render targets [MRTs] are being used. Up to four MRTs can be used to write out large amounts of data—up to 16 scalar floating-point values at a time, for example—plus depth.)</p> <p>First, the scissor test rejects the fragment if it lies outside a specified subrectangle of the frame buffer. Although the popular graphics APIs define scissoring at this location in the pipeline, it is more efficient to perform the scissor test in the rasterizer. Scissoring in x and y actually happens in the rasterizer, before fragment processing, and z scissoring happens during z-cull. This avoids all fragment processor work on scissored (rejected) pixels. Scissoring is rarely useful for general-purpose computation because general-purpose programmers typically draw rectangles to perform computations in the first place.</p>

Ex. A11 – Invalidity of U.S. Patent No. 8,648,867 – GPU Gems

U.S. Patent No. 8,648,867	GPU Gems
	<p>Next, the fragment's depth is compared with the depth in the frame buffer. If the depth test passes, the fragment moves on in the pipeline. Optionally, the depth value in the frame buffer can be replaced at this stage.</p> <p>After this, the fragment can optionally test and modify what is known as the stencil buffer, which stores an integer value per pixel. The stencil buffer was originally intended to allow programmers to mask off certain pixels (for example, to restrict drawing to a cockpit's windshield), but it has found other uses as a way to count values by incrementing or decrementing the existing value. This feature is used for stencil shadow volumes, for example.</p> <p>If the fragment passes the depth and stencil tests, it can then optionally modify the contents of the frame buffer using the blend function. A blend function can be described as</p> $\text{out} = \text{src} * \text{srcOp} + \text{dst} * \text{dstOp}$ <p>where source is the fragment color flowing down the pipeline; dst is the color value in the frame buffer; and srcOp and dstOp can be specified to be constants, source color components, or destination color components. Full blend functionality is supported for all pixel formats up to fp16x4. However, fp32 frame buffers don't support blending—only updating the buffer is allowed.</p> <p>Finally, a feature called occlusion query makes it possible to quickly determine if any of the fragments that would be rendered in a particular computation would cause results to be written to the frame buffer. (Recall that fragments that do not pass the z-test don't have any effect on the values in the frame buffer.) Traditionally, the occlusion query test is used to allow graphics applications to avoid making draw calls for occluded objects, but it is useful for GPGPU applications as well. For instance, if the depth test is used to determine which outputs need to be updated in a sparse array, updating depth can be used to indicate when a given output has converged and no further work is needed. In this case, occlusion query can be used to tell when all output calculations are done. See Chapter 34 of this book, "GPU Flow-Control Idioms," for further information about this idea.</p> <p>478–480.</p>

Ex. A11 – Invalidity of U.S. Patent No. 8,648,867 – GPU Gems

U.S. Patent No. 8,648,867	GPU Gems
	<p>36.1 Filtering Through Compaction</p> <p>Assume we have a data set with arbitrary values and we wish to extract all positive values. On current GPUs, a fragment program that filters streams must produce null records for each output that should not be retained in the final output stream. Null records are represented with some value that the program will never generate when it does want to output a value. (In the examples that follow, we use floating-point infinity for this purpose. Floating-point infinity can be generated by computing 1.0/0.0, and it can be detected with the isinf() standard library function.) The following example uses the Brook programming language:</p> <pre> kernel void positive(float input<>, out float output<>, float inf<>) { if (input > 0) output = input; else output = inf; // inf 1./0 must be passed in as a stream on ATI GPUs } 574. </pre> <p>40.1 Introduction</p> <p>In some sense, computer graphics and computer vision are the opposites of one another. Computer graphics takes a numerical description of a scene and renders an image, whereas computer vision analyzes images to create numerical representations of the scene. Thus, carrying out computer vision tasks in graphics hardware uses the graphics hardware in an "inverse" fashion.</p> <p>The GPU provides a streaming, data-parallel arithmetic architecture. This type of architecture carries out a similar set of calculations on an array of image data. The single-instruction, multiple-data (SIMD) capability of the GPU makes it suitable for running computer vision tasks, which often involve similar calculations operating on an entire image.</p>

Ex. A11 – Invalidity of U.S. Patent No. 8,648,867 – GPU Gems

U.S. Patent No. 8,648,867	GPU Gems
	<p>Special-purpose computer vision hardware is rarely found in typical mass-produced personal computers. Instead, the CPU is usually used for computer vision tasks. Optimized computer vision libraries for the CPU often consume many of the CPU cycles to achieve real-time performance, leaving little time for other, nonvision tasks.</p> <p>GPUs, on the other hand, are found on most personal computers and often exceed the capabilities of the CPU. Thus, we can use the GPU to accelerate computer vision computation and free up the CPU for other tasks. Furthermore, multiple GPUs can be used on the same machine, creating an architecture capable of running multiple computer vision algorithms in parallel.</p> <p>649–50.</p> <p><i>See also</i> 471–91 (discussing a computer system using the GeForce 6 Series GPU); 520–45 (discussing a stream model for GPU processing).</p> <p>To the extent Plaintiff contends that this claim element is not expressly or inherently disclosed, or is not obvious in view of GPU Gems and the knowledge of a POSITA, it would have been obvious to combine the teachings of GPU Gems with any of the prior art references in NVIDIA’s Invalidity Contentions and any supplements thereto and the relevant section of charts for other prior art for the ’867 patent in a manner that would result in the subject matter of this limitation given, at the very least, the person’s understanding of the state of the art, the problems addressed and solved in the prior art, and the teachings of GPU Gems. Specific examples of such references are cited and explained in these invalidity contentions, including the cover pleading and charts A1–A13, and Appendix A.</p>
<p>18: The method as claimed in claim 16, further comprising: partitioning the accelerator memory into the first partition, the second partition, a third partition to store internal variables, and a fourth partition to store data</p>	<p>GPU Gems discloses “partitioning the accelerator memory into the first partition, the second partition, a third partition to store internal variables, and a fourth partition to store data used as input at a particular computation cycle of the numerical simulation.” <i>See e.g.</i>:</p> <p><i>See</i> 16[b], 16[d].</p> <p><i>As a non-limiting example, GPU Gems discloses that its accelerator memory (e.g., memory located in the GPU) is partitioned in a first partition to store input data and a second partition to</i></p>

Ex. A11 – Invalidity of U.S. Patent No. 8,648,867 – GPU Gems

U.S. Patent No. 8,648,867	GPU Gems
used as input at a particular computation cycle of the numerical simulation.	<p data-bbox="604 235 1871 381"><i>store output data. See 16[b]. GPU Gems also discloses storage of internal variables (e.g., the physical locations of each node, “coords”) in a “third partition” of the accelerator memory, and storage of data used as input at a particular computational cycle of the numerical simulation (e.g., diffusivity parameters du and dv) in a “fourth partition” of the accelerator memory.</i></p>  <p data-bbox="604 755 768 787">Figure 31-3.</p>

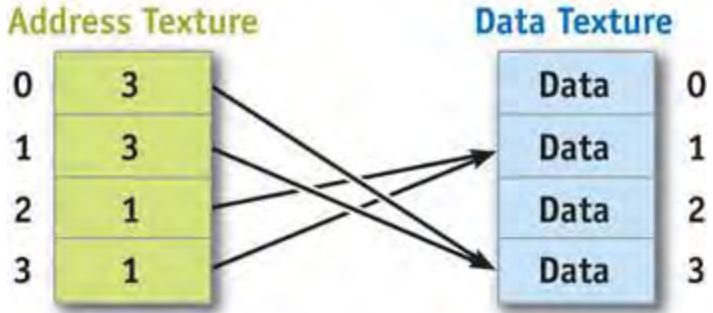
Ex. A11 – Invalidity of U.S. Patent No. 8,648,867 – GPU Gems

U.S. Patent No. 8,648,867	GPU Gems
	<p>Listing 31-1. The Reaction-Diffusion Kernel Program</p> <hr/> <pre> float4 rd(float2 coords : DOMAIN, uniform stream concentration, uniform float2 DuDv, uniform float F, uniform float k) : RANGE { float2 center = concentration.value2(coords).xy; float2 diffusion = concentration.value2(coords + half2(1, 0)); diffusion += concentration.value2(coords + half2(-1, 0)); diffusion += concentration.value2(coords + half2(0, 1)); diffusion += concentration.value2(coords + half2(0, -1)); // Average and scale by diffusion coeffs diffusion *= 0.25f * DuDv; float2 reaction = center.xx * center.yy * center.yy; reaction.x = -1; reaction.x += (1 - DuDv.x) * center.x + F * (1 - center.x); reaction.y += (-F - k + (1 - DuDv.y)) * center.y; // Now add the diffusion to the reaction to get the result. return float4(diffusion + reaction, 0, 0); } </pre> <hr/> <p>Listing 31-1.</p>

Ex. A11 – Invalidity of U.S. Patent No. 8,648,867 – GPU Gems

U.S. Patent No. 8,648,867	GPU Gems
	<p>Listing 31-2. C++ Code to Set Up and Run a Reaction-Diffusion Simulation on the GPU <i>See the source code on the accompanying CD for more details.</i></p> <hr/> <pre> PUGBuffer *rdBuffer; PUGProgram *rdProgram; void init_rd(){ // Call this once. pugInit(); // Start up the GPU framework // Create a "double-buffered" PUGBuffer. Two buffers allow the // simulation to alternate using one as input, one as output PUGBuffer *rdBuffer = pugAllocateBuffer(width, height, PUG_READWRITE, 4, true); PUGProgram *rdProgram = pugLoadProgram("rd.cg", "rd"); pugBindFloat(rdProgram, "DuDv", du, dv); // bind parameters pugBindFloat(rdProgram, "F", F); pugBindFloat(rdProgram, "k", k); </pre> <hr/> <p>Listing 31-2 (continued). C++ Code to Set Up and Run a Reaction-Diffusion Simulation on the GPU</p> <hr/> <pre> // Initialize the state of the simulation with values in array pugInitBuffer(rdBuffer, array); } void update_rd(){ // Call this every iteration pugBindStream(rdProgram, "concentration", rdBuffer, currentSource); PUGRect range(0, width, 0, height); pugRunProgram(rdProgram, rdBuffer, range, currentTarget); std::swap(currentSource, currentTarget); } </pre> <hr/> <p>Listing 31-2.</p>

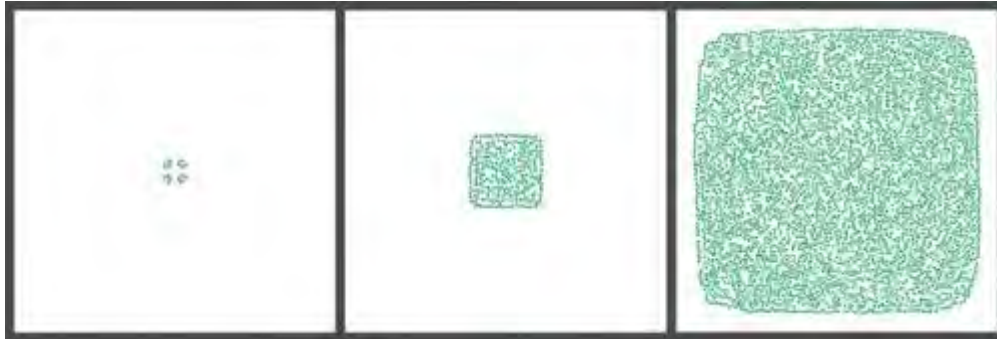
Ex. A11 – Invalidity of U.S. Patent No. 8,648,867 – GPU Gems

U.S. Patent No. 8,648,867	GPU Gems
	 <p>Figure 33-4.</p> <p>31.5 A Simple Example</p> <p>As a basic but nontrivial GPGPU example, we use a simulation of a phenomenon known as chemical reaction-diffusion. Reaction-diffusion is a model of how the concentrations of two or more reactants in a solution evolve in space and time as they undergo the processes of chemical reaction and diffusion. The reaction-diffusion model we use is called the Grey-Scott model (Pearson 1993) and involves just two chemical reactants. This phenomenological model does not represent a particular real chemical reaction; it serves as a simple model for studying the general reaction-diffusion phenomenon. Figure 31-3 shows the results of many iterations of the Grey-Scott model.</p> <p>Grey-Scott consists of two simple partial differential equations that govern the evolution of the concentrations of two chemical reactants, U and V:</p> $\frac{\partial U}{\partial t} = \boxed{D_u \nabla^2 U} - \boxed{UV^2 + F(1 - U)},$ $\frac{\partial V}{\partial t} = \boxed{D_v \nabla^2 V} + \boxed{UV^2 - (F + k)V}.$ <p style="text-align: center;">Diffusion Reaction</p>

Ex. A11 – Invalidity of U.S. Patent No. 8,648,867 – GPU Gems

U.S. Patent No. 8,648,867	GPU Gems
	<p>Here k and F are constants; Du and Dv are the diffusion rates of the reactants; and ∇^2 is the Laplacian operator, which represents diffusion. The Laplacian operator appears commonly in physics, most notably in the form of diffusion equations, such as the heat equation. A finite difference form of the Laplacian operator applied to a scalar field p on a two-dimensional Cartesian grid (with indices i, j, and cell size x) is</p> $\nabla^2 p = \frac{p_{i+1,j} + p_{i-1,j} + p_{i,j+1} + p_{i,j-1} - 4p_{i,j}}{(\Delta x)^2}$ <p>The implementation of the Grey-Scott model is fairly simple. There is a single data stream: the U and V chemical concentrations are stored in two channels of a single texture that represents a discrete spatial grid. This stream serves as input to a simple kernel, which implements the preceding equations in discrete form. The kernel is shown in Listing 31-1. C code that uses the framework to implement the simulation is shown in Listing 31-2.</p> <p>505–08.</p> <p>An additional access pattern emerges from the preceding set of rules and the stream types described in Section 33.2.2: pointer streams (Purcell et al. 2002). Pointer streams arise out of the ability to use any input stream as the address for a texture read. Figure 33-4 shows that pointer streams are simply streams whose values are memory addresses. If the pointer stream is read from a texture, this capability is called dependent texturing.</p> <p>527–28.</p> <p><i>See also</i> 520–45 (discussing accelerator memory structures and access).</p> <p>To the extent Plaintiff contends that this claim element is not expressly or inherently disclosed, or is not obvious in view of GPU Gems and the knowledge of a POSITA, it would have been obvious to combine the teachings of GPU Gems with any of the prior art references in NVIDIA’s Invalidity Contentions and any supplements thereto and the relevant section of charts for other prior art for the ’867 patent in a manner that would result in the subject matter of this limitation given, at the very least, the person's understanding of the state of the art, the problems addressed</p>

Ex. A11 – Invalidity of U.S. Patent No. 8,648,867 – GPU Gems

U.S. Patent No. 8,648,867	GPU Gems
	<p>and solved in the prior art, and the teachings of GPU Gems. Specific examples of such references are cited and explained in these invalidity contentions, including the cover pleading and charts A1–A13, and Appendix A.</p>
<p>19: The method of claim 16, further comprising: transferring, by the accelerator controller, the first output data to the main memory during the second computational cycle.</p>	<p>GPU Gems discloses “transferring, by the accelerator controller, the first output data to the main memory during the second computational cycle.” <i>See e.g.:</i></p> <p><i>As a non-limiting example, GPU Gems discloses transferring, by the accelerator controller (e.g., at the direction and/or under the control of the vertex processors) of first output data (e.g., results) to the main memory (e.g., the CPU).</i></p> <div data-bbox="606 597 1596 933">  </div> <p>Figure 31-3.</p>

Ex. A11 – Invalidity of U.S. Patent No. 8,648,867 – GPU Gems

U.S. Patent No. 8,648,867	GPU Gems
	<p>Listing 31-1. The Reaction-Diffusion Kernel Program</p> <hr/> <pre> float4 rd(float2 coords : DOMAIN, uniform stream concentration, uniform float2 DuDv, uniform float F, uniform float k) : RANGE { float2 center = concentration.value2(coords).xy; float2 diffusion = concentration.value2(coords + half2(1, 0)); diffusion += concentration.value2(coords + half2(-1, 0)); diffusion += concentration.value2(coords + half2(0, 1)); diffusion += concentration.value2(coords + half2(0, -1)); // Average and scale by diffusion coeffs diffusion *= 0.25f * DuDv; float2 reaction = center.xx * center.yy * center.yy; reaction.x = -1; reaction.x += (1 - DuDv.x) * center.x + F * (1 - center.x); reaction.y += (-F - k + (1 - DuDv.y)) * center.y; // Now add the diffusion to the reaction to get the result. return float4(diffusion + reaction, 0, 0); } </pre> <hr/> <p>Listing 31-1.</p>

Ex. A11 – Invalidity of U.S. Patent No. 8,648,867 – GPU Gems

U.S. Patent No. 8,648,867	GPU Gems
	<p>Listing 31-2. C++ Code to Set Up and Run a Reaction-Diffusion Simulation on the GPU <i>See the source code on the accompanying CD for more details.</i></p> <hr/> <pre> PUGBuffer *rdBuffer; PUGProgram *rdProgram; void init_rd(){ // Call this once. pugInit(); // Start up the GPU framework // Create a "double-buffered" PUGBuffer. Two buffers allow the // simulation to alternate using one as input, one as output PUGBuffer *rdBuffer = pugAllocateBuffer(width, height, PUG_READWRITE, 4, true); PUGProgram *rdProgram = pugLoadProgram("rd.cg", "rd"); pugBindFloat(rdProgram, "DuDv", du, dv); // bind parameters pugBindFloat(rdProgram, "F", F); pugBindFloat(rdProgram, "k", k); </pre> <hr/> <p>Listing 31-2 (continued). C++ Code to Set Up and Run a Reaction-Diffusion Simulation on the GPU</p> <hr/> <pre> // Initialize the state of the simulation with values in array pugInitBuffer(rdBuffer, array); } void update_rd(){ // Call this every iteration pugBindStream(rdProgram, "concentration", rdBuffer, currentSource); PUGRect range(0, width, 0, height); pugRunProgram(rdProgram, rdBuffer, range, currentTarget); std::swap(currentSource, currentTarget); } </pre> <hr/> <p>Listing 31-2.</p> <p>31.1.3 Stream Communication: Gather vs. Scatter</p> <p>High arithmetic intensity requires that communication between stream elements be minimized, but for many computations, communication is a necessary evil. In the cloud simulation, for example, some of the kernels must obtain information from cells other than the one currently being processed by the kernel. When discussing data communication on GPUs, it is helpful to consider</p>

Ex. A11 – Invalidity of U.S. Patent No. 8,648,867 – GPU Gems

U.S. Patent No. 8,648,867	GPU Gems
	<p>two main types of communication: gather and scatter. Gather occurs when the kernel processing a stream element requests information from other elements in the stream: it "gathers" information from other parts of memory. Scatter, on the other hand, occurs when the kernel processing a stream element distributes information to other stream elements: it "scatters" information to other parts of memory. In terms of traditional memory concepts, gather requires only random-access load capability, while scatter requires only random-access store capability. Later we show why gather is typically preferable to scatter.</p> <p>31.2 An Inventory of GPU Computational Resources</p> <p>To start mapping general computation onto the specialized hardware of a GPU, we should first survey the computational resources that GPUs provide. We start with the computational workhorses: the processors.</p> <p>31.2.1 Programmable Parallel Processors</p> <p>GPUs have two types of programmable processors: vertex processors and fragment processors. Vertex processors process streams of vertices (made up of positions, colors, normal vectors, and other attributes), which are the elements that compose polygonal geometric models. Computer graphics typically represents 3D objects with triangular meshes. The vertex processors apply a vertex program(sometimes called a vertex shader) to transform each vertex based on its position relative to the camera, and then each set of three vertices is used to compute a triangle, from which streams of fragments are generated. A fragment can be considered a "proto-pixel." It contains all information needed to generate a shaded pixel in the final image, including color, depth, and destination in the frame buffer. The fragment processors apply a fragment program (sometimes called a pixel shader) to each fragment in the stream to compute the final color of each pixel.</p> <p>Vertex Processors</p> <p>Modern GPUs have multiple vertex processors (the NVIDIA GeForce 6800 Ultra and the ATI Radeon X800 XT both have six). These processors are fully programmable and operate in either SIMD- or MIMD-parallel fashion on the input vertices (see Chapter 34 for more information on these terms). The basic primitives of 3D computer graphics are 3D vertices in projected space, represented by an (x, y, z, w) vector, and four-component colors stored as (red, green, blue, alpha) vectors (often abbreviated RGBA), where alpha typically represents an opacity percentage.</p>

Ex. A11 – Invalidity of U.S. Patent No. 8,648,867 – GPU Gems

U.S. Patent No. 8,648,867	GPU Gems
	<p>Because of this, vertex processors have hardware to process four-component vectors. This allows them to produce transformed vertex positions in fewer cycles.</p> <p>Vertex processors are capable of changing the position of input vertices. If you think about this, the position of these vertices ultimately affects where in the image pixels will be drawn. An image is nothing but an array of memory; thus, because vertex processors can control where in memory data will be written, they are thus capable of scatter. However, most current vertex processors cannot directly read information from vertex elements in the input stream other than the one currently being processed. Therefore, they are incapable of gather. The NVIDIA GeForce 6 Series GPUs have a new feature called vertex texture fetch (VTF). This means that GeForce 6 vertex processors are capable of random-access memory reads. So, we can store part or all of our input stream data in a vertex texture and use VTF to implement a gather operation.</p> <p>Fragment Processors</p> <p>Modern GPUs also have multiple fragment processors (the NVIDIA GeForce 6800 Ultra and the ATI X800 XT both have 16). Like vertex processors, these are fully programmable. Fragment processors operate in SIMD-parallel fashion on input elements, processing four-element vectors in parallel. Fragment processors have the ability to fetch data from textures, so they are capable of gather. However, the output address of a fragment is always determined before the fragment is processed: the processor cannot change the output location of a pixel. Fragment processors are thus not natively capable of scatter. However, see Section 31.3 for further discussion and techniques for working around this limitation.</p> <p>For GPGPU applications, the fragment processors are typically used more heavily than the vertex processors. There are two main reasons for this. First, there are more fragment processors than vertex processors on a typical programmable GPU. Second, the output of the fragment processors goes more or less directly into memory, which can be fed straight back in as a new stream of texture data. Vertex processor output, on the other hand, must pass through the rasterizer and fragment processors before reaching memory. This makes direct output from vertex processors less straightforward.</p> <p>496–98.</p>

Ex. A11 – Invalidity of U.S. Patent No. 8,648,867 – GPU Gems

U.S. Patent No. 8,648,867	GPU Gems
	<p>Getting Data Back to the CPU</p> <p>To bring results on the GPU back to the CPU, call the pugGetBufferData() function, passing it a pointer to the PUGBuffer. This function returns a pointer to a C array of type float. Reading data back from the GPU can cause the GPU pipeline to flush, so use this function sparingly to avoid hurting application performance.</p> <p>505.</p> <p>31.5 A Simple Example</p> <p>As a basic but nontrivial GPGPU example, we use a simulation of a phenomenon known as chemical reaction-diffusion. Reaction-diffusion is a model of how the concentrations of two or more reactants in a solution evolve in space and time as they undergo the processes of chemical reaction and diffusion. The reaction-diffusion model we use is called the Grey-Scott model (Pearson 1993) and involves just two chemical reactants. This phenomenological model does not represent a particular real chemical reaction; it serves as a simple model for studying the general reaction-diffusion phenomenon. Figure 31-3 shows the results of many iterations of the Grey-Scott model.</p> <p>Grey-Scott consists of two simple partial differential equations that govern the evolution of the concentrations of two chemical reactants, U and V:</p> $\begin{aligned}\frac{\partial U}{\partial t} &= D_u \nabla^2 U - UV^2 + F(1 - U), \\ \frac{\partial V}{\partial t} &= D_v \nabla^2 V + UV^2 - (F + k)V.\end{aligned}$ <p style="text-align: center;">Diffusion Reaction</p> <p>Here k and F are constants; Du and Dv are the diffusion rates of the reactants; and ∇^2 is the Laplacian operator, which represents diffusion. The Laplacian operator appears commonly in physics, most notably in the form of diffusion equations, such as the heat equation. A finite</p>

Ex. A11 – Invalidity of U.S. Patent No. 8,648,867 – GPU Gems

U.S. Patent No. 8,648,867	GPU Gems
	<p data-bbox="611 235 1751 305">difference form of the Laplacian operator applied to a scalar field p on a two-dimensional Cartesian grid (with indices i, j, and cell size x) is</p> $\nabla^2 p = \frac{p_{i+1,j} + p_{i-1,j} + p_{i,j+1} + p_{i,j-1} - 4p_{i,j}}{(\Delta x)^2}$ <p data-bbox="611 472 1850 686">The implementation of the Grey-Scott model is fairly simple. There is a single data stream: the U and V chemical concentrations are stored in two channels of a single texture that represents a discrete spatial grid. This stream serves as input to a simple kernel, which implements the preceding equations in discrete form. The kernel is shown in Listing 31-1. C code that uses the framework to implement the simulation is shown in Listing 31-2. 505–08.</p> <p data-bbox="611 727 1871 1044">To the extent Plaintiff contends that this claim element is not expressly or inherently disclosed, or is not obvious in view of GPU Gems and the knowledge of a POSITA, it would have been obvious to combine the teachings of GPU Gems with any of the prior art references in NVIDIA’s Invalidity Contentions and any supplements thereto and the relevant section of charts for other prior art for the ’867 patent in a manner that would result in the subject matter of this limitation given, at the very least, the person's understanding of the state of the art, the problems addressed and solved in the prior art, and the teachings of GPU Gems. Specific examples of such references are cited and explained in these invalidity contentions, including the cover pleading and charts A1–A13, and Appendix A.</p>