

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

Prior Art Reference: GPU Gems 2: Programming Techniques for High-Performance Graphics and General Purpose Computation (“GPU Gems”)

GPU Gems was published in March 2005, and was publicly available no later than July 2005. GPU Gems therefore qualifies as prior art to U.S. Patent No. RE49,461 (the “’461 patent”) under, at least, (pre-AIA) 35 U.S.C. § 102(a) and/or (b). GPU Gems anticipates and/or renders obvious claims 21–31 of the ’461 patent (the “Asserted Claims”).

The chart below provides representative examples of where each element of each claim is found within 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.

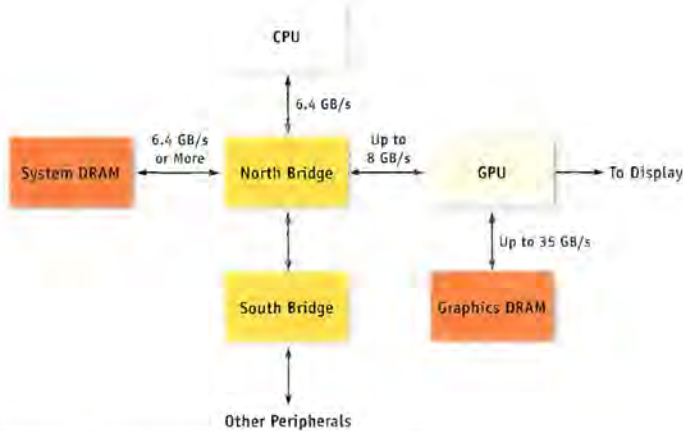
To the extent Plaintiff argues that any element below is not disclosed by GPU Gems, a person of ordinary skill in the art would have found it obvious to combine the teachings of GPU Gems with the background knowledge of a person of ordinary skill in the art and/or the additional references, and exemplary 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. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
21[pre]: A method of executing computations representing an artificial neural network on a computer system comprising at least one central processing unit (CPU), a processing unit, a first memory partition, and a second memory partition, the method comprising:	NVIDIA takes no position in these Invalidity Contentions on whether the entirety of the preamble of this claim is a limitation. To the extent it is a limitation, GPU Gems discloses, expressly and/or inherently a method of executing computations representing an artificial neural network on a computer system comprising at least one central processing unit (CPU), a processing unit, a first memory partition, and a second memory partition. <i>See e.g.:</i> <i>As a non-limiting example, GPU Gems discloses a method of executing computations representing an artificial neural network on a computer system;</i> Chapter 40—titled “Computer Vision on the GPU”—explores “methods of efficiently mapping common mathematical operations of computer vision onto modern computer graphics architecture.”

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
	Table 30-1 reiterates some of the points made in the preceding chapter: there is a vast amount of bandwidth available internally on the GPU. Algorithms that run on the GPU can therefore take advantage of this bandwidth to achieve dramatic performance improvements. 30.2 Overall System Architecture 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.  Figure 30-2. The Overall System Architecture of a PC 30.2.1 Functional Block Diagram for Graphics Operations 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. 30.2 Overall System Architecture 473

473.

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
	Figure 30-3. A Block Diagram of the GeForce 6 Series Architecture First, commands, textures, and vertex data are received from the host CPU through shared buffers in system memory or local frame-buffer memory. A command stream is written by the CPU, which initializes and modifies state, sends rendering commands, and references the texture and vertex data. Commands are parsed, and a vertex fetch unit is used to read the vertices referenced by the rendering commands. The commands, vertices, and state changes flow downstream, where they are used by subsequent pipeline stages. The vertex processors (sometimes called “vertex shaders”), shown in Figure 30-4, allow for a program to be applied to each vertex in the object, performing transformations, skinning, and any other per-vertex operation the user specifies. For the first time, a 474 Chapter 30 The GeForce 6 Series GPU Architecture

474.

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
649.	Chapter 40 Computer Vision on the GPU <i>James Fung</i> <i>University of Toronto</i>  Computer vision tasks are computationally intensive and repetitive, and they often exceed the real-time capabilities of the CPU, leaving little time for higher-level tasks. However, many computer vision operations map efficiently onto the modern GPU, whose programmability allows a wide variety of computer vision algorithms to be implemented. This chapter presents methods of efficiently mapping common mathematical operations of computer vision onto modern computer graphics architecture. 40.1 Introduction 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. 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. 40.1 Introduction 649

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
651.	40.3 Application Examples This chapter presents examples that implement common computer vision techniques on the GPU. The common thread between each of these algorithms is that they are all used in our system to create a fully interactive, real-time <i>computer-mediated reality</i>, in which the environment of a user appears to be augmented, diminished, or otherwise altered. Some examples are shown in Figure 40-1.  Figure 40-1. Using Computer Vision to Mediate Reality <i>Computer vision techniques can analyze video to augment, diminish, or otherwise alter a user's perception of the environment. On the left, detected ceiling planes become terminal windows. On the right, a terminal window is mapped to the plane of the floppy disk.</i> 40.3.1 Using Sequences of Fragment Programs for Computer Vision In this section, we demonstrate how a sequence of fragment programs can work together to perform more-complex computer vision operations. Correcting Radial Distortion Many camera lenses cause some sort of radial distortion (also known as <i>barrel distortion</i>) of the image, most commonly in wide-field-of-view or low-cost lenses. There are several ways of correcting for radial distortion on the graphics hardware; we present an example of how a fragment program can be used to do so. 40.3 Application Examples 651

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
	40.4 Parallel Computer Vision Processing When multiple computer vision algorithms are needed for an application, parallel GPU processing can be achieved by placing multiple graphics cards in the PCI bus slots of a motherboard. Figure 40-7 shows a machine built to explore the benefits of parallel computations by using multiple graphics cards. Running each computer vision program on a different GPU allows the cards to process in parallel; conventional shared memory can be used to allow the programs to interact. Although creating multi-GPU systems using PCI cards will soon no longer be possible due to the obsolescence of the PCI bus, the advent of PCI Express has made new multi-GPU approaches possible. One such approach is NVIDIA's SLI, which makes the presence of multiple GPUs transparent to the developer by automatically distributing work between them.  Figure 40-7. Multiple GPUs Used for a Computer Vision Application Left: A computer vision application (Fung and Mann 2002) using graphics hardware to display six video streams in a hexagonal form. Right: A computer vision machine that runs computer vision tasks in parallel using six PCI graphics cards and one AGP graphics card. Each PCI card is a GeForce FX 5200 GPU, resulting in an inexpensive, powerful, and easily constructible parallel architecture well suited for pattern recognition and computer vision. 40.5 Conclusion This chapter has presented techniques for mapping computer vision algorithms onto modern GPUs. We hope that this chapter has provided an understanding of how to write computer vision algorithms for fast processing on the GPU. Although other 664 Chapter 40 Computer Vision on the GPU 664.

33.2 The GPU Memory Model

Graphics processors have their own memory hierarchy analogous to the one used by serial microprocessors, including main memory, caches, and registers. This memory hierarchy, however, is designed for accelerating graphics operations that fit into the streaming programming model rather than general, serial computation. Moreover, graphics APIs such as OpenGL and Direct3D further limit the use of this memory to graphics-specific primitives such as vertices, textures, and frame buffers. This section gives an overview of the current memory model on GPUs and how stream-based computation fits into it.

33.2.1 Memory Hierarchy

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.

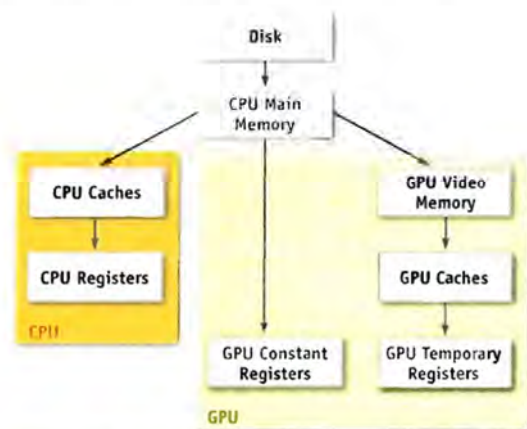


Figure 33-2. The CPU and GPU Memory Hierarchies

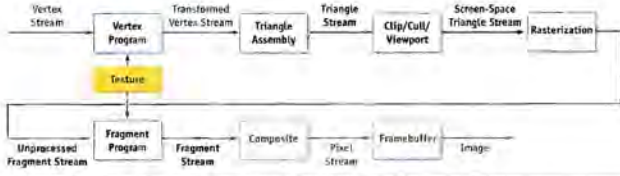
Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
	524. 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 ’461 patent in a manner that would result in the subject matter of this limitation given, at the very least, the person's understanding of the state of the art, the problems addressed and solved in the prior art, and the teachings of GPU Gems. Specific examples of such references are cited and explained in these invalidity contentions, including the cover pleading and charts C1–C15, and Appendix C.
21[a]: executing, by the at least one CPU, a user interaction stream, the user interaction stream controlling transfer of inputs to the artificial neural network to the first memory partition and the second memory partition;	GPU Gems discloses, expressly and/or inherently, executing, by the at least one CPU, a user interaction stream, the user interaction stream controlling transfer of inputs to the artificial neural network to the first memory partition and the second memory partition. <i>See e.g.:</i> <i>As a non-limiting example, GPU Gems discloses a CPU executing a “user interaction stream” (e.g., commands provided by a user through a user interface). GPU Gems also discloses that the user interaction stream controls the transfer of inputs to the numerical simulation to the first and second memory partitions (e.g., GPU memory such as shader registers or fragment source buffers, and/or main memory such as system RAM or disk).</i>

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
	progress. Consequently, GPU memory systems contain several levels of cache memory (making up a substantial fraction of the chip's transistors) to minimize this latency. However, caches are ineffective for many types of data, including graphics inputs and data that is accessed only once. For the graphics pipeline, maximizing throughput for all elements rather than minimizing latency for each element results in better utilization of the memory system and a higher-performance implementation overall. 29.3 Stream Computation In the previous section, we have seen that building high-performance processors today requires both efficient computation and efficient communication. Part of the reason that CPUs are poorly suited to many of these high-performance applications is their serial programming model, which does not expose the parallelism and communication patterns in the application. In this section, we describe the <i>stream</i> programming model, which structures programs in a way that allows high efficiency in computation and communication (Kapasi et al. 2003). This programming model is the basis for programming GPUs today. 29.3.1 The Stream Programming Model In the <i>stream</i> programming model, all data is represented as a <i>stream</i>, which we define as an ordered set of data of the same data type. That data type can be simple (a stream of integers or floating-point numbers) or complex (a stream of points or triangles or transformation matrices). While a stream can be any length, we will see that operations on streams are most efficient if streams are long (hundreds or more elements in a stream). Allowed operations on streams include copying them, deriving substreams from them, indexing into them with a separate index stream, and performing computation on them with <i>kernels</i>. A kernel operates on entire streams, taking one or more streams as inputs and producing one or more streams as outputs. The defining characteristic of a kernel is that it operates on entire <i>streams</i> of elements as opposed to individual elements. The most typical use of a kernel is to evaluate a function on each element of an input stream (a "map" operation); for example, a transformation kernel may project each element of a stream of points into a different coordinate system. Other desirable kernel operations include expansions (in which more than one output element is produced for each input element), reductions (in which more than one element is combined into a single output element), or filters (in which a subset of input elements are output).
464.	464 Chapter 29 Streaming Architectures and Technology Trends

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
	Kernel outputs are functions only of their kernel inputs, and within a kernel, computations on one stream element are never dependent on computations on another element. These restrictions have two major advantages. First, the data required for kernel execution is completely known when the kernel is written (or compiled). Kernels can thus be highly efficient when their input elements and their intermediate computed data are stored locally or are carefully controlled global references. Second, requiring independence of computation on separate stream elements within a single kernel allows mapping what appears to be a serial kernel calculation onto data-parallel hardware. In the stream programming model, applications are constructed by chaining multiple kernels together. For instance, implementing the graphics pipeline in the stream programming model involves writing a vertex program kernel, a triangle assembly kernel, a clipping kernel, and so on, and then connecting the output from one kernel into the input of the next kernel. Figure 29-3 shows how the entire graphics pipeline maps onto the stream model. This model makes the communication between kernels explicit, taking advantage of the data locality between kernels inherent in the graphics pipeline. The graphics pipeline is a good match for the stream model for several reasons. The graphics pipeline is traditionally structured as stages of computation connected by data flow between the stages. This structure is analogous to the stream and kernel abstractions of the stream programming model. Data flow between stages in the graphics pipeline is highly localized, with data produced by a stage immediately consumed by the next stage; in the stream programming model, streams passed between kernels exhibit similar behavior. And the computation involved in each stage of the pipeline is typically uniform across different primitives, allowing these stages to be easily mapped to kernels.  Figure 29-3. Mapping the Graphics Pipeline to the Stream Model The stream formulation of the graphics pipeline expresses all data as streams (indicated by arrows) and all computation as kernels (indicated by blue boxes). Both user-programmable and nonprogrammable stages in the graphics pipeline can be expressed as kernels. 29.3 Stream Computation 465

465.

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
466.	Efficient Computation The stream model enables efficient computation in several ways. Most important, streams expose parallelism in the application. Because kernels operate on entire streams, stream elements can be processed in parallel using data-parallel hardware. Long streams with many elements allow this data-level parallelism to be highly efficient. Within the processing of a single element, we can exploit instruction-level parallelism. And because applications are constructed from multiple kernels, multiple kernels can be deeply pipelined and processed in parallel, using task-level parallelism. Dividing the application of interest into kernels allows a hardware implementation to specialize hardware for one or more kernels' execution. Special-purpose hardware, with its superior efficiency over programmable hardware, can thus be used appropriately in this programming model. Finally, allowing only simple control flow in kernel execution (such as the data-parallel evaluation of a function on each input element) permits hardware implementations to devote most of their transistors to datapath hardware rather than control hardware. Efficient Communication Efficient communication is also one of the primary goals of the stream programming model. First, off-chip (global) communication is more efficient when entire streams, rather than individual elements, are transferred to or from memory, because the fixed cost of initiating a transfer can be amortized over an entire stream rather than a single element. Next, structuring applications as chains of kernels allows the intermediate results between kernels to be kept on-chip and not transferred to and from memory. Efficient kernels attempt to keep their inputs and their intermediate computed data local within kernel execution units; therefore, data references within kernel execution do not go off-chip or across a chip to a data cache, as would typically happen in a CPU. And finally, deep pipelining of execution allows hardware implementations to continue to do useful work while waiting for data to return from global memories. This high degree of latency tolerance allows hardware implementations to optimize for throughput rather than latency. 29.3.2 Building a Stream Processor The stream programming model structures programs in a way that both exposes parallelism and permits efficient communication. Expressing programs in the stream model is only half the solution, however. High-performance graphics hardware must effec-
	466 Chapter 29 Streaming Architectures and Technology Trends

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
	tively exploit the high arithmetic performance and the efficient computation exposed by the stream model. How do we structure a hardware implementation of a GPU to ensure the highest overall performance? The first step to building a high-performance GPU is to map kernels in the graphics pipeline to independent functional units on a single chip. Each kernel is thus implemented on a separate area of the chip in an organization known as <i>task parallel</i>, which permits not only task-level parallelism (because all kernels can be run simultaneously) but also hardware specialization of each functional unit to the given kernel. The task-parallel organization also allows efficient communication between kernels: because the functional units implementing neighboring kernels in the graphics pipeline are adjacent on the chip, they can communicate effectively without requiring global memory access. Within each stage of the graphics pipeline that maps to a processing unit on the chip, GPUs exploit the independence of each stream element by processing multiple data elements in parallel. The combination of task-level and data-level parallelism allows GPUs to profitably use dozens of functional units simultaneously. Inputs to the graphics pipeline must be processed by each kernel in sequence. Consequently, it may take thousands of cycles to complete the processing of a single element. If a high-latency memory reference is required in processing any given element, the processing unit can simply work on other elements while the data is being fetched. The deep pipelines of modern GPUs, then, effectively tolerate high-latency operations. For many years, the kernels that make up the graphics pipeline were implemented in graphics hardware as fixed-function units that offered little to no user programmability. In 2000, for the first time, GPUs allowed users the opportunity to program individual kernels in the graphics pipeline. Today's GPUs feature high-performance data-parallel processors that implement two kernels in the graphics pipeline: a <i>vertex program</i> that allows users to run a program on each vertex that passes through the pipeline, and a <i>fragment program</i> that allows users to run a program on each fragment. Both of these stages permit single-precision floating-point computation. Although these additions were primarily intended to provide users with more flexible shading and lighting calculations, their ability to sustain high computation rates on user-specified programs with sufficient precision to address general-purpose computing problems has effectively made them <i>programmable stream processors</i>—that is, processors that are attractive for a much wider variety of applications than simply the graphics pipeline. 29.3 Stream Computation 467

467.

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
	Figure 30-3. A Block Diagram of the GeForce 6 Series Architecture First, commands, textures, and vertex data are received from the host CPU through shared buffers in system memory or local frame-buffer memory. A command stream is written by the CPU, which initializes and modifies state, sends rendering commands, and references the texture and vertex data. Commands are parsed, and a vertex fetch unit is used to read the vertices referenced by the rendering commands. The commands, vertices, and state changes flow downstream, where they are used by subsequent pipeline stages. The vertex processors (sometimes called “vertex shaders”), shown in Figure 30-4, allow for a program to be applied to each vertex in the object, performing transformations, skinning, and any other per-vertex operation the user specifies. For the first time, a 474 Chapter 30 The GeForce 6 Series GPU Architecture

474.

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
	all improve. For example, the yearly capability increase has led to an enormous degree of integration on a single die. Fifteen years ago, designers were only just beginning to integrate floating-point arithmetic units onto a processor die; today, such a unit occupies less than a square millimeter of die area, and hundreds can be placed onto the same die. However, the most important consequences of these technology trends are the <i>differences</i> between them. When one metric changes at a different rate than another, it requires rethinking the assumptions behind processor and system design. We can identify three major issues that will help drive GPU architectures of the future: compute versus communicate, latency versus bandwidth, and power. Compute vs. Communicate As both clock speeds and chip sizes increase, the amount of time it takes for a signal to travel across an entire chip, measured in clock cycles, is also increasing. On today's fastest processors, sending a signal from one side of a chip to another typically requires multiple clock cycles, and this amount of time increases with each new process generation. We can characterize this trend as an increase in the cost of communication when compared to the cost of computation. As a consequence, designers of the future will increasingly use computation via cheap transistors to replace the need for expensive communication. Another likely impact will be an increase in the amount of computation available per word of memory bandwidth. As an example, let us compare NVIDIA's last three flagship GPUs (2002's GeForce FX 5800, 2003's GeForce FX 5950, and 2004's GeForce 6800) in measured peak programmable floating-point performance against peak off-chip bandwidth to memory. The GeForce FX 5800 could sustain 2 floating-point operations for every word of off-chip bandwidth, while the GeForce FX 5950 could sustain 2.66 operations, and the GeForce 6800 could sustain nearly 6. We expect this trend to continue in future chip generations. Figure 29-1 shows historical data for observed floating-point operations per second and available memory bandwidth for a series of GPU architectures. Latency vs. Bandwidth The gap between the trends of bandwidth and latency will also be an important driver of future architectures. Because latency will continue to improve more slowly than bandwidth (Patterson 2004), designers must implement solutions that can tolerate larger and larger amounts of latency by continuing to do useful work while waiting for data to return from operations that take a long time. 29.1 Technology Trends 459 459.

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

The fragment processor uses the texture unit to fetch data from memory, optionally filtering the data before returning it to the fragment processor. The texture unit supports many source data formats (see Section 30.3.3, "Supported Data Storage Formats"). Data can be filtered using bilinear, trilinear, or anisotropic filtering. All data is returned to the fragment processor in fp32 or fp16 format. A texture can be viewed as a 2D or 3D array of data that can be read by the texture unit at arbitrary locations and filtered to reconstruct a continuous function. The GeForce 6 Series supports filtering of fp16 textures in hardware.

The fragment processor has two fp32 shader units per pipeline, and fragments are routed through both shader units and the branch processor before recirculating through the entire pipeline to execute the next series of instructions. This rerouting happens once for each core clock cycle. Furthermore, the first fp32 shader can be used for perspective correction of texture coordinates when needed (by dividing by w), or for general-purpose multiply operations. In general, it is possible to perform eight or more math operations in the pixel shader during each clock cycle, or four math operations if a texture fetch occurs in the first shader unit.

On the final pass through the pixel shader pipeline, the fog unit can be used to blend fog in fixed-point precision with no performance penalty. Fog blending happens often in conventional graphics applications and uses the following function:

$$\text{out} = \text{FogColor} * \text{fogFraction} + \text{SrcColor} * (1 - \text{fogFraction})$$

This function can be made fast and small using fixed-precision math, but in general IEEE floating point, it requires two full multiply-adds to do effectively. Because fixed point is efficient and sufficient for fog, it exists in a separate small unit at the end of the shader. This is a good example of the trade-offs in providing flexible programmable hardware while still offering maximum performance for legacy applications.

Fragments leave the fragment-processing unit in the order that they are rasterized and are sent to the z-compare and blend units, which perform depth testing (z comparison and update), stencil operations, alpha blending, and the final color write to the target surface (an off-screen render target or the frame buffer).

The memory system is partitioned into up to four independent memory partitions, each with its own dynamic random-access memories (DRAMs). GPUs use standard DRAM modules rather than custom RAM technologies to take advantage of market economies and thereby reduce cost. Having smaller, independent memory partitions allows the memory subsystem to operate efficiently regardless of whether large or small blocks of data are transferred. All rendered surfaces are stored in the DRAMs, while textures and input data can be stored in the DRAMs or in system memory. The four

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
	477. documentation included with the Cg distribution (NVIDIA 2004). There are very few graphics-specific keywords in Cg. We provide a Cg file, <code>pug.cg</code>, that you can include in your kernel files. This file defines a structure called <code>Stream</code> that abstracts the most graphics-centric concept: texture fetching. <code>Stream</code> is a wrapper for the texture sampler Cg type. Calling one of the <code>value()</code> functions of the <code>Stream</code> structure gives the value of a stream element. 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. Stream Management Arrays of data in the framework are called <i>buffers</i>. 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 <code>mode</code> 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. To load initial data into a buffer, pass the data to the <code>pugInitBuffer()</code> function. 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. 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. 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. 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 <code>range</code> parameter of the <code>pugRunProgram()</code> function. 504 Chapter 31 Mapping Computational Concepts to GPUs

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
	504. 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. 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 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. 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>. 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. 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>. 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. 31.5 A Simple Example 505

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
	505.

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
	33.2 The GPU Memory Model Graphics processors have their own memory hierarchy analogous to the one used by serial microprocessors, including main memory, caches, and registers. This memory hierarchy, however, is designed for accelerating graphics operations that fit into the streaming programming model rather than general, serial computation. Moreover, graphics APIs such as OpenGL and Direct3D further limit the use of this memory to graphics-specific primitives such as vertices, textures, and frame buffers. This section gives an overview of the current memory model on GPUs and how stream-based computation fits into it. 33.2.1 Memory Hierarchy 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. <pre>graph TD Disk --> CPU_Main_Memory[CPU Main Memory] CPU_Main_Memory --> CPU_Caches[CPU Caches] CPU_Main_Memory --> GPU_Video_Memory[GPU Video Memory] CPU_Caches --> CPU_Registers[CPU Registers] GPU_Video_Memory --> GPU_Caches[GPU Caches] GPU_Caches --> GPU_Temporary_Registers[GPU Temporary Registers] CPU_Main_Memory --> GPU_Constant_Registers[GPU Constant Registers]</pre> Figure 33-2. The CPU and GPU Memory Hierarchies 524 Chapter 33 Implementing Efficient Parallel Data Structures on GPUs 524.

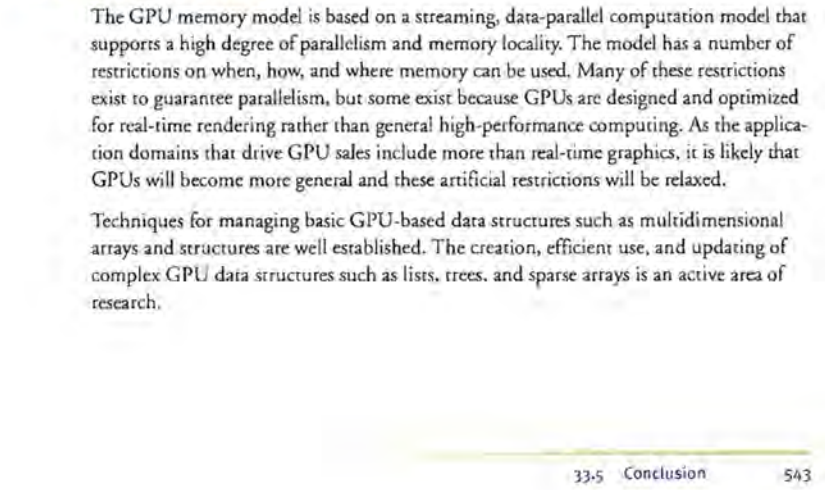
Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
	<i>As a non-limiting example, GPU Gems discloses user interaction streams.</i> 33.2.2 GPU Stream Types Unlike CPU memory, GPU memory has a number of usage restrictions and is accessible only via the abstractions of a graphics programming interface. Each of these abstractions can be thought of as a different stream type with its own set of access rules. The three types of streams visible to the GPU programmer are vertex streams, frame-buffer streams, and texture streams. A fourth stream type, fragment streams, is produced and consumed entirely within the GPU. Figure 33-3 shows the pipeline for a modern GPU, the three user-accessible streams, and the points in the pipeline where each can be used. Vertex Streams Vertex streams are specified as vertex buffers via the graphics API. These streams hold vertex positions and a variety of per-vertex attributes. These attributes have traditionally been used for texture coordinates, colors, normals, and so on, but they can be used for arbitrary input stream data for vertex programs. Vertex programs are not allowed to randomly index into their input vertices. Until recently, vertex streams could be updated only by transferring data from the CPU to the GPU. The GPU was not allowed to write to vertex streams. Recent API enhancements, however, have made it possible for the GPU to write to vertex streams. This is accomplished by either "copy-to-vertex-buffer" or "render-to-vertex-buffer." In the former technique, rendering results are copied from a frame buffer to a vertex buffer; in the latter technique, the rendering results are written directly to a vertex buffer. The recent addition of GPU-writable vertex streams enables GPUs, for the first time, to loop stream results from the end to the beginning of the pipeline. Figure 33-3. Streams in Modern GPUs. The GPU programmer can directly access vertices, frame buffers, and textures. Fragment streams are generated by the rasterizer and consumed by the fragment processor. They are the input streams to fragment programs but are created and consumed entirely within the GPU and so are not directly accessible to the programmer. 33.2 The GPU Memory Model 525 525.

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
	Fragment Streams Fragment streams are generated by the rasterizer and consumed by the fragment processor. They are the stream inputs to fragment programs, but they are not directly accessible to programmers because they are created and consumed entirely within the graphics processor. Fragment stream values include all of the interpolated outputs from the vertex processor: position, color, texture coordinates, and so on. As with per-vertex stream attributes, the per-fragment values that have traditionally been used for texture coordinates may now be used for any stream value required by the fragment program. Fragment streams cannot be randomly accessed by fragment programs. Permitting random access to the fragment stream would create a dependency between fragment stream elements, thus breaking the data-parallel guarantee of the programming model. If random access to fragment streams is needed by algorithms, the stream must first be saved to memory and converted to a texture stream. Frame-Buffer Streams Frame-buffer streams are written by the fragment processor. They have traditionally been used to hold pixels for display to the screen. Streaming GPU computation, however, uses frame buffers to hold the results of intermediate computation stages. In addition, modern GPUs are able to write to multiple frame-buffer surfaces (that is, multiple RGBA buffers) simultaneously. Current GPUs can write up to 16 floating-point scalar values per render pass (this value is expected to increase in future hardware). Frame-buffer streams cannot be randomly accessed by either fragment or vertex programs. They can, however, be directly read from or written to by the CPU via the graphics API. Lastly, recent API enhancements have begun to blur the distinction between frame buffers, vertex buffers, and textures by allowing a render pass to be directly written to any of these stream types. Texture Streams Textures are the only GPU memory that is randomly accessible by fragment programs and, for Vertex Shader 3.0 GPUs, vertex programs. If programmers need to randomly index into a vertex, fragment, or frame-buffer stream, they must first convert it to a texture. Textures can be read from and written to by either the CPU or the GPU. The GPU writes to textures either by rendering directly to them instead of to a frame buffer or by copying data from a frame buffer to texture memory. Textures are declared as 1D, 2D, or 3D streams and addressed with a 1D, 2D, or 3D address, respectively. A texture can also be declared as a cube map, which can be treated as an array of six 2D textures. 526 Chapter 33 Implementing Efficient Parallel Data Structures on GPUs 526.

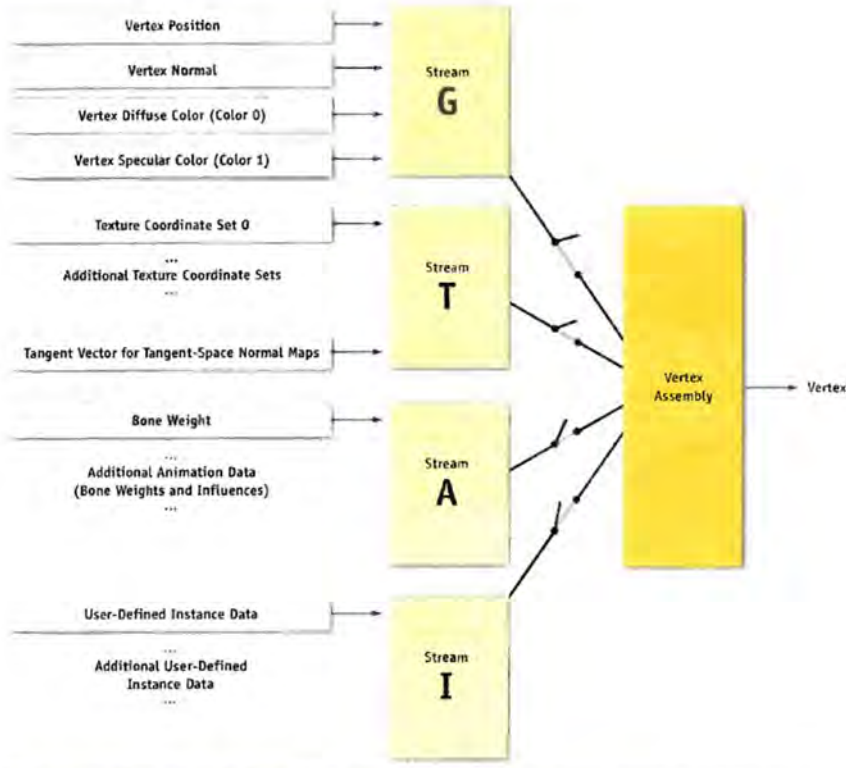
Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
	33.5 Conclusion The GPU memory model is based on a streaming, data-parallel computation model that supports a high degree of parallelism and memory locality. The model has a number of restrictions on when, how, and where memory can be used. Many of these restrictions exist to guarantee parallelism, but some exist because GPUs are designed and optimized for real-time rendering rather than general high-performance computing. As the application domains that drive GPU sales include more than real-time graphics, it is likely that GPUs will become more general and these artificial restrictions will be relaxed. Techniques for managing basic GPU-based data structures such as multidimensional arrays and structures are well established. The creation, efficient use, and updating of complex GPU data structures such as lists, trees, and sparse arrays is an active area of research.  543. 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 '461 patent in a manner that would result in the subject matter of this limitation given, at the very least, the person's understanding of the state of the art, the problems addressed and solved in the prior art, and the teachings of GPU Gems. Specific examples of such references are cited and explained in these invalidity contentions, including the cover pleading and charts C1–C15, and Appendix C.
21[b]: executing, by the processing unit, a computational stream, the computational stream controlling data exchange between	GPU Gems discloses, expressly and/or inherently, executing, by the processing unit, a computational stream, the computational stream controlling data exchange between the user interaction stream and the computational stream during execution of the computations representing the artificial neural network.

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
the user interaction stream and the computational stream during execution of the computations representing the artificial neural network;	<i>As a non-limiting example, GPU Gems discloses a processing unit (e.g., GPU) executing a computational stream (e.g., a sequence of computations) that controls data exchange between the user interaction stream (see 21[a]) and the computational stream.</i> <i>See Chapter 29 – Streaming Architecture and Technology Trends.</i> <i>See e.g.:</i>

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
	 Figure 5-2. Combining Currently Needed Streams 5.2.1 Multistreaming with DirectX 9.0 Microsoft DirectX 8.0 introduced the notion of a stream to bind data to input registers. A stream is a uniform array of component data. Each component consists of one or more elements that represent a single entity, such as position, normal, color, texture coordinate set, and so on. With streams, GPUs are able to perform a direct memory access (DMA) from multiple vertex buffers in parallel. The DirectX9 caps bit 78.

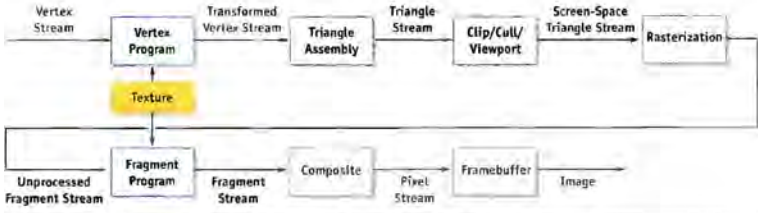
Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
	all improve. For example, the yearly capability increase has led to an enormous degree of integration on a single die. Fifteen years ago, designers were only just beginning to integrate floating-point arithmetic units onto a processor die; today, such a unit occupies less than a square millimeter of die area, and hundreds can be placed onto the same die. However, the most important consequences of these technology trends are the <i>differences</i> between them. When one metric changes at a different rate than another, it requires rethinking the assumptions behind processor and system design. We can identify three major issues that will help drive GPU architectures of the future: compute versus communicate, latency versus bandwidth, and power. Compute vs. Communicate As both clock speeds and chip sizes increase, the amount of time it takes for a signal to travel across an entire chip, measured in clock cycles, is also increasing. On today's fastest processors, sending a signal from one side of a chip to another typically requires multiple clock cycles, and this amount of time increases with each new process generation. We can characterize this trend as an increase in the cost of communication when compared to the cost of computation. As a consequence, designers of the future will increasingly use computation via cheap transistors to replace the need for expensive communication. Another likely impact will be an increase in the amount of computation available per word of memory bandwidth. As an example, let us compare NVIDIA's last three flagship GPUs (2002's GeForce FX 5800, 2003's GeForce FX 5950, and 2004's GeForce 6800) in measured peak programmable floating-point performance against peak off-chip bandwidth to memory. The GeForce FX 5800 could sustain 2 floating-point operations for every word of off-chip bandwidth, while the GeForce FX 5950 could sustain 2.66 operations, and the GeForce 6800 could sustain nearly 6. We expect this trend to continue in future chip generations. Figure 29-1 shows historical data for observed floating-point operations per second and available memory bandwidth for a series of GPU architectures. Latency vs. Bandwidth The gap between the trends of bandwidth and latency will also be an important driver of future architectures. Because latency will continue to improve more slowly than bandwidth (Patterson 2004), designers must implement solutions that can tolerate larger and larger amounts of latency by continuing to do useful work while waiting for data to return from operations that take a long time. 29.1 Technology Trends 459 459.

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
	progress. Consequently, CPU memory systems contain several levels of cache memory (making up a substantial fraction of the chip's transistors) to minimize this latency. However, caches are ineffective for many types of data, including graphics inputs and data that is accessed only once. For the graphics pipeline, maximizing throughput for all elements rather than minimizing latency for each element results in better utilization of the memory system and a higher-performance implementation overall. 29.3 Stream Computation In the previous section, we have seen that building high-performance processors today requires both efficient computation and efficient communication. Part of the reason that CPUs are poorly suited to many of these high-performance applications is their serial programming model, which does not expose the parallelism and communication patterns in the application. In this section, we describe the <i>stream</i> programming model, which structures programs in a way that allows high efficiency in computation and communication (Kapasi et al. 2003). This programming model is the basis for programming GPUs today. 29.3.1 The Stream Programming Model In the stream programming model, all data is represented as a <i>stream</i>, which we define as an ordered set of data of the same data type. That data type can be simple (a stream of integers or floating-point numbers) or complex (a stream of points or triangles or transformation matrices). While a stream can be any length, we will see that operations on streams are most efficient if streams are long (hundreds or more elements in a stream). Allowed operations on streams include copying them, deriving substreams from them, indexing into them with a separate index stream, and performing computation on them with <i>kernels</i>. A kernel operates on entire streams, taking one or more streams as inputs and producing one or more streams as outputs. The defining characteristic of a kernel is that it operates on entire <i>streams</i> of elements as opposed to individual elements. The most typical use of a kernel is to evaluate a function on each element of an input stream (a "map" operation); for example, a transformation kernel may project each element of a stream of points into a different coordinate system. Other desirable kernel operations include expansions (in which more than one output element is produced for each input element), reductions (in which more than one element is combined into a single output element), or filters (in which a subset of input elements are output). 464 Chapter 29 Streaming Architectures and Technology Trends 464.

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
	Kernel outputs are functions only of their kernel inputs, and within a kernel, computations on one stream element are never dependent on computations on another element. These restrictions have two major advantages. First, the data required for kernel execution is completely known when the kernel is written (or compiled). Kernels can thus be highly efficient when their input elements and their intermediate computed data are stored locally or are carefully controlled global references. Second, requiring independence of computation on separate stream elements within a single kernel allows mapping what appears to be a serial kernel calculation onto data-parallel hardware. In the stream programming model, applications are constructed by chaining multiple kernels together. For instance, implementing the graphics pipeline in the stream programming model involves writing a vertex program kernel, a triangle assembly kernel, a clipping kernel, and so on, and then connecting the output from one kernel into the input of the next kernel. Figure 29-3 shows how the entire graphics pipeline maps onto the stream model. This model makes the communication between kernels explicit, taking advantage of the data locality between kernels inherent in the graphics pipeline. The graphics pipeline is a good match for the stream model for several reasons. The graphics pipeline is traditionally structured as stages of computation connected by data flow between the stages. This structure is analogous to the stream and kernel abstractions of the stream programming model. Data flow between stages in the graphics pipeline is highly localized, with data produced by a stage immediately consumed by the next stage; in the stream programming model, streams passed between kernels exhibit similar behavior. And the computation involved in each stage of the pipeline is typically uniform across different primitives, allowing these stages to be easily mapped to kernels.  Figure 29-3. Mapping the Graphics Pipeline to the Stream Model The stream formulation of the graphics pipeline expresses all data as streams (indicated by arrows) and all computation as kernels (indicated by blue boxes). Both user-programmable and nonprogrammable stages in the graphics pipeline can be expressed as kernels. 29.3 Stream Computation 465 465.

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
	Efficient Computation The stream model enables efficient computation in several ways. Most important, streams expose parallelism in the application. Because kernels operate on entire streams, stream elements can be processed in parallel using data-parallel hardware. Long streams with many elements allow this data-level parallelism to be highly efficient. Within the processing of a single element, we can exploit instruction-level parallelism. And because applications are constructed from multiple kernels, multiple kernels can be deeply pipelined and processed in parallel, using task-level parallelism. Dividing the application of interest into kernels allows a hardware implementation to specialize hardware for one or more kernels' execution. Special-purpose hardware, with its superior efficiency over programmable hardware, can thus be used appropriately in this programming model. Finally, allowing only simple control flow in kernel execution (such as the data-parallel evaluation of a function on each input element) permits hardware implementations to devote most of their transistors to datapath hardware rather than control hardware. Efficient Communication Efficient communication is also one of the primary goals of the stream programming model. First, off-chip (global) communication is more efficient when entire streams, rather than individual elements, are transferred to or from memory, because the fixed cost of initiating a transfer can be amortized over an entire stream rather than a single element. Next, structuring applications as chains of kernels allows the intermediate results between kernels to be kept on-chip and not transferred to and from memory. Efficient kernels attempt to keep their inputs and their intermediate computed data local within kernel execution units; therefore, data references within kernel execution do not go off-chip or across a chip to a data cache, as would typically happen in a CPU. And finally, deep pipelining of execution allows hardware implementations to continue to do useful work while waiting for data to return from global memories. This high degree of latency tolerance allows hardware implementations to optimize for throughput rather than latency. 29.3.2 Building a Stream Processor The stream programming model structures programs in a way that both exposes parallelism and permits efficient communication. Expressing programs in the stream model is only half the solution, however. High-performance graphics hardware must effec- 466 Chapter 29 Streaming Architectures and Technology Trends 466.

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
	tively exploit the high arithmetic performance and the efficient computation exposed by the stream model. How do we structure a hardware implementation of a GPU to ensure the highest overall performance? The first step to building a high-performance GPU is to map kernels in the graphics pipeline to independent functional units on a single chip. Each kernel is thus implemented on a separate area of the chip in an organization known as <i>task parallel</i>, which permits not only task-level parallelism (because all kernels can be run simultaneously) but also hardware specialization of each functional unit to the given kernel. The task-parallel organization also allows efficient communication between kernels: because the functional units implementing neighboring kernels in the graphics pipeline are adjacent on the chip, they can communicate effectively without requiring global memory access. Within each stage of the graphics pipeline that maps to a processing unit on the chip, GPUs exploit the independence of each stream element by processing multiple data elements in parallel. The combination of task-level and data-level parallelism allows GPUs to profitably use dozens of functional units simultaneously. Inputs to the graphics pipeline must be processed by each kernel in sequence. Consequently, it may take thousands of cycles to complete the processing of a single element. If a high-latency memory reference is required in processing any given element, the processing unit can simply work on other elements while the data is being fetched. The deep pipelines of modern GPUs, then, effectively tolerate high-latency operations. For many years, the kernels that make up the graphics pipeline were implemented in graphics hardware as fixed-function units that offered little to no user programmability. In 2000, for the first time, GPUs allowed users the opportunity to program individual kernels in the graphics pipeline. Today's GPUs feature high-performance data-parallel processors that implement two kernels in the graphics pipeline: a <i>vertex program</i> that allows users to run a program on each vertex that passes through the pipeline, and a <i>fragment program</i> that allows users to run a program on each fragment. Both of these stages permit single-precision floating-point computation. Although these additions were primarily intended to provide users with more flexible shading and lighting calculations, their ability to sustain high computation rates on user-specified programs with sufficient precision to address general-purpose computing problems has effectively made them <i>programmable stream processors</i>—that is, processors that are attractive for a much wider variety of applications than simply the graphics pipeline. 29.3 Stream Computation 467 467.

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
	29.4 The Future and Challenges The migration of GPUs into programmable stream processors reflects the culmination of several historical trends. The first trend is the ability to concentrate large amounts of computation on a single processor die. Equally important has been the ability and talent of GPU designers in effectively using these computation resources. The economies of scale that are associated with building tens of millions of processors per year have allowed the cost of a GPU to fall enough to make a GPU a standard part of today's desktop computer. And the addition of reasonably high-precision programmability to the pipeline has completed the transition from a hard-wired, special-purpose processor to a powerful programmable processor that can address a wide variety of tasks. What, then, can we expect from future GPU development? 29.4.1 Challenge: Technology Trends Each new generation of hardware will present a challenge to GPU vendors: How can they effectively use additional hardware resources to increase performance and functionality? New transistors will be devoted to increased performance, in large part through greater amounts of parallelism, and to new functionality in the pipeline. We will also see these architectures evolve with changes in technology. As we described in Section 29.1.2, future architectures will increasingly use transistors to replace the need for communication. We can expect more aggressive caching techniques that not only alleviate off-chip communication but also mitigate the need for some on-chip communication. We will also see computation increasingly replace communication when appropriate. For example, the use of texture memory as a lookup table may be replaced by calculating the values in that lookup table dynamically. And instead of sending data to a distant on-chip computation resource and then sending the result back, we may simply replicate the resource and compute our result locally. In the trade-off between <i>communicate</i> and <i>recompute/cache</i>, we will increasingly choose the latter. The increasing cost of communication will also influence the microarchitecture of future chips. Designers must now explicitly plan for the time required to send data across a chip; even local communication times are becoming significant in a timing budget. 29.4.2 Challenge: Power Management Ideas for how to use future GPU transistors must be tempered by the realities of their costs. Power management has become a critical piece of today's GPU designs as each 468 Chapter 29 Streaming Architectures and Technology Trends 468.

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
	generation of hardware has increased its power demand. The future may hold more aggressive dynamic power management targeted at individual stages; increasing amounts of custom or power-aware design for power-hungry parts of the GPU; and more sophisticated cooling management for high-end GPUs. Technology trends indicate that the power demand will only continue to rise with future chip generations, so continued work in this area will remain an important challenge. 29.4.3 Challenge: Supporting More Programmability and Functionality While the current generation of graphics hardware features substantially more programmability than previous generations, the general programmability of GPUs is still far from ideal. One step toward addressing this trend is to improve the functionality and flexibility within the two current programmable units (vertex and fragment). It is likely that we will see their instruction sets converge and add functionality, and that their control flow capabilities will become more general as well. We may even see programmable hardware shared between these two stages in an effort to better utilize these resources. GPU architects will have to be mindful, however, that such improvements not affect the GPU's performance on its core tasks. Another option will be expanding programmability to different units. Geometric primitives particularly benefit from programmability, so we may soon see programmable processing on surfaces, triangles, and pixels. As GPU vendors support more general pipelines and more complex and varied shader computation, many researchers have used the GPU to address tasks outside the bounds of the graphics pipeline. This book contains examples of many of these efforts; the general-purpose computation on GPUs (GPGPU) community has successfully addressed problems in visual simulation, image processing, numerical methods, and databases with graphics hardware. We can expect that these efforts will grow in the future as GPUs continue to increase in performance and functionality. Historically, we have seen GPUs subsume functionality previously belonging to the CPU. Early consumer-level graphics hardware could not perform geometry processing on the graphics processor; it was only five years ago that the entire graphics pipeline could be fabricated on a single chip. Though since that time the primary increase in GPU functionality has been directed toward programmability within the graphics pipeline, we should not expect that GPU vendors have halted their efforts to identify more functions to integrate onto the GPU. In particular, today's games often require large amounts of computation in physics and artificial intelligence computations. Such computation may be attractive for future GPUs. 29.4 The Future and Challenges 469 469.

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
	Though modern technologies allow us a large number of transistors on each die, our resources are not infinite. Our use of transistors can be broadly divided into three categories: <i>control</i>, the hardware used to direct the computation; <i>datapath</i>, the hardware used to perform the computation; and <i>storage</i>, the hardware used to store data. If our goal is to maximize performance, we must make hardware and software decisions that allow us to maximize the transistors in the datapath that are devoted to performing computation. Within the datapath, we can allow simultaneous operations at the same time. This technique is termed <i>parallelism</i>. We can envision several ways to exploit parallelism and permit simultaneous execution. Complex tasks such as graphics processing are typically composed of several sequential tasks. When running these applications, we may be able to run several of these tasks on different data at the same time (<i>task parallelism</i>). Within a stage, if we are running a task on several data elements, we may be able to exploit <i>data parallelism</i> in evaluating them at the same time. And within the complex evaluation of a single data element, we may be able to evaluate several simple operations at the same time (<i>instruction parallelism</i>). To effectively use hundreds of computation units, we may take advantage of any or all of these types of parallelism. Within each of the tasks, we could make each of our arithmetic units fully programmable and thus able to perform any task. However, we can gain more efficiency from our transistors by taking advantage of <i>specialization</i>. If a specific arithmetic unit performs only one kind of computation, that unit can be specialized to that computation with a considerable gain in efficiency. For example, triangle rasterization, in which a screen-space triangle is transformed into the fragments that cover that triangle, realizes an enormous efficiency benefit when the rasterization task is implemented in special-purpose hardware instead of programmable hardware. 29.2.2 Methods for Efficient Communication As we saw in Section 29.1.2, off-chip bandwidth is growing more slowly than on-chip arithmetic capability, so high-performance processors must minimize off-chip communication. The easiest way to reduce this cost is to eliminate it: modern processors attempt to keep as much required data communication on-chip as possible, requiring off-chip communication only to fetch or store truly global data. Another common way to mitigate the increasing cost of communication is through caching: a copy of recently used data memory is stored on-chip, removing the need for a fetch from off-chip if that data is needed again. Such data caching will be more com- 462 Chapter 29 Streaming Architectures and Technology Trends 462.

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
	Z-Cull NVIDIA GPUs since GeForce3 have technology, called <i>z-cull</i>, that allows hidden surface removal at speeds much faster than conventional rendering. The GeForce 6 Series z-cull unit is the third generation of this technology, which has increased efficiency for a wider range of cases. Also, in cases where stencil is not being updated, early stencil reject can be employed to remove rendering early when stencil test (based on equals comparison) fails. Occlusion Query Occlusion query is the ability to collect statistics on how many fragments passed or failed the depth test and to report the result back to the host CPU. Occlusion query can be used either while rendering objects or with color and z-write masks turned off, returning depth test status for the objects that would have been rendered, without modifying the contents of the frame buffer. This feature has been available since the GeForce3 was introduced. Texturing Like previous GPUs, GeForce 6 Series GPUs support bilinear, trilinear, and anisotropic filtering on 2D and cube-map textures of various formats. Three-dimensional textures support bilinear, trilinear, and quad-linear filtering, with and without mipmapping. Here are the new texturing features on GeForce 6 Series GPUs: • Support for all texture types (2D, cube map, 3D) with fp16×2, fp16×4, fp32×1, fp32×2, and fp32×4 formats • Support for all filtering modes on fp16×2 and fp16×4 texture formats • Extended support for non-power-of-two textures to match support for power-of-two textures, specifically: – Mipmapping – Wrapping and clamping – Cube map and 3D textures Shadow Buffer Support NVIDIA GPUs support shadow buffering directly. The application first renders the scene from the light source into a separate z-buffer. Then during the lighting phase, it fetches the shadow buffer as a projective texture and performs z-compare of the shadow buffer data against a value corresponding to the distance from the light. If the 482 Chapter 30 The GeForce 6 Series GPU Architecture 482.

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
	31.2 An Inventory of GPU Computational Resources 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. 31.2.1 Programmable Parallel Processors GPUs have two types of programmable processors: <i>vertex processors</i> and <i>fragment processors</i>. 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 <i>vertex program</i> (sometimes called a <i>vertex shader</i>) 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 <i>fragments</i> 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 <i>fragment program</i> (sometimes called a <i>pixel shader</i>) to each fragment in the stream to compute the final color of each pixel. Vertex Processors 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. Because of this, vertex processors have hardware to process four-component vectors. This allows them to produce transformed vertex positions in fewer cycles. 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, 31.2 An Inventory of GPU Computational Resources 497 497.

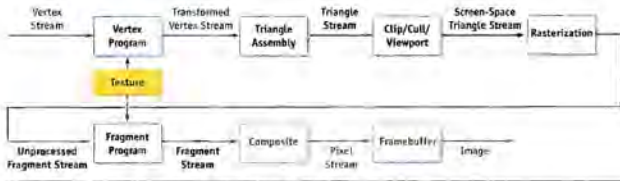
Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
	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 ’461 patent in a manner that would result in the subject matter of this limitation given, at the very least, the person's understanding of the state of the art, the problems addressed and solved in the prior art, and the teachings of GPU Gems. Specific examples of such references are cited and explained in these invalidity contentions, including the cover pleading and charts C1–C15, and Appendix C.
21[c]: shifting control of a data exchange between the user interaction stream and the computational stream to the computational stream in response to starting execution of the computations representing the artificial neural network;	GPU Gems discloses, expressly and/or inherently, shifting control of a data exchange between the user interaction stream and the computational stream to the computational stream in response to starting execution of the computations representing the artificial neural network. <i>See e.g.:</i> <i>As a non-limiting example, GPU Gems discloses shifting control of a data exchange between the user interaction stream and the computational stream (e.g., by switching contexts) in response to starting to perform computations.</i>

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
	progress. Consequently, GPU memory systems contain several levels of cache memory (making up a substantial fraction of the chip's transistors) to minimize this latency. However, caches are ineffective for many types of data, including graphics inputs and data that is accessed only once. For the graphics pipeline, maximizing throughput for all elements rather than minimizing latency for each element results in better utilization of the memory system and a higher-performance implementation overall. 29.3 Stream Computation In the previous section, we have seen that building high-performance processors today requires both efficient computation and efficient communication. Part of the reason that CPUs are poorly suited to many of these high-performance applications is their serial programming model, which does not expose the parallelism and communication patterns in the application. In this section, we describe the <i>stream</i> programming model, which structures programs in a way that allows high efficiency in computation and communication (Kapasi et al. 2003). This programming model is the basis for programming GPUs today. 29.3.1 The Stream Programming Model In the stream programming model, all data is represented as a <i>stream</i>, which we define as an ordered set of data of the same data type. That data type can be simple (a stream of integers or floating-point numbers) or complex (a stream of points or triangles or transformation matrices). While a stream can be any length, we will see that operations on streams are most efficient if streams are long (hundreds or more elements in a stream). Allowed operations on streams include copying them, deriving substreams from them, indexing into them with a separate index stream, and performing computation on them with <i>kernels</i>. A kernel operates on entire streams, taking one or more streams as inputs and producing one or more streams as outputs. The defining characteristic of a kernel is that it operates on entire <i>streams</i> of elements as opposed to individual elements. The most typical use of a kernel is to evaluate a function on each element of an input stream (a "map" operation); for example, a transformation kernel may project each element of a stream of points into a different coordinate system. Other desirable kernel operations include expansions (in which more than one output element is produced for each input element), reductions (in which more than one element is combined into a single output element), or filters (in which a subset of input elements are output).
464.	464 Chapter 29 Streaming Architectures and Technology Trends

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
	Kernel outputs are functions only of their kernel inputs, and within a kernel, computations on one stream element are never dependent on computations on another element. These restrictions have two major advantages. First, the data required for kernel execution is completely known when the kernel is written (or compiled). Kernels can thus be highly efficient when their input elements and their intermediate computed data are stored locally or are carefully controlled global references. Second, requiring independence of computation on separate stream elements within a single kernel allows mapping what appears to be a serial kernel calculation onto data-parallel hardware. In the stream programming model, applications are constructed by chaining multiple kernels together. For instance, implementing the graphics pipeline in the stream programming model involves writing a vertex program kernel, a triangle assembly kernel, a clipping kernel, and so on, and then connecting the output from one kernel into the input of the next kernel. Figure 29-3 shows how the entire graphics pipeline maps onto the stream model. This model makes the communication between kernels explicit, taking advantage of the data locality between kernels inherent in the graphics pipeline. The graphics pipeline is a good match for the stream model for several reasons. The graphics pipeline is traditionally structured as stages of computation connected by data flow between the stages. This structure is analogous to the stream and kernel abstractions of the stream programming model. Data flow between stages in the graphics pipeline is highly localized, with data produced by a stage immediately consumed by the next stage; in the stream programming model, streams passed between kernels exhibit similar behavior. And the computation involved in each stage of the pipeline is typically uniform across different primitives, allowing these stages to be easily mapped to kernels.  Figure 29-3. Mapping the Graphics Pipeline to the Stream Model The stream formulation of the graphics pipeline expresses all data as streams (indicated by arrows) and all computation as kernels (indicated by blue boxes). Both user-programmable and nonprogrammable stages in the graphics pipeline can be expressed as kernels. 29.3 Stream Computation 465

465.

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
	Efficient Computation The stream model enables efficient computation in several ways. Most important, streams expose parallelism in the application. Because kernels operate on entire streams, stream elements can be processed in parallel using data-parallel hardware. Long streams with many elements allow this data-level parallelism to be highly efficient. Within the processing of a single element, we can exploit instruction-level parallelism. And because applications are constructed from multiple kernels, multiple kernels can be deeply pipelined and processed in parallel, using task-level parallelism. Dividing the application of interest into kernels allows a hardware implementation to specialize hardware for one or more kernels' execution. Special-purpose hardware, with its superior efficiency over programmable hardware, can thus be used appropriately in this programming model. Finally, allowing only simple control flow in kernel execution (such as the data-parallel evaluation of a function on each input element) permits hardware implementations to devote most of their transistors to datapath hardware rather than control hardware. Efficient Communication Efficient communication is also one of the primary goals of the stream programming model. First, off-chip (global) communication is more efficient when entire streams, rather than individual elements, are transferred to or from memory, because the fixed cost of initiating a transfer can be amortized over an entire stream rather than a single element. Next, structuring applications as chains of kernels allows the intermediate results between kernels to be kept on-chip and not transferred to and from memory. Efficient kernels attempt to keep their inputs and their intermediate computed data local within kernel execution units; therefore, data references within kernel execution do not go off-chip or across a chip to a data cache, as would typically happen in a CPU. And finally, deep pipelining of execution allows hardware implementations to continue to do useful work while waiting for data to return from global memories. This high degree of latency tolerance allows hardware implementations to optimize for throughput rather than latency. 29.3.2 Building a Stream Processor The stream programming model structures programs in a way that both exposes parallelism and permits efficient communication. Expressing programs in the stream model is only half the solution, however. High-performance graphics hardware must effec- 466 Chapter 29 Streaming Architectures and Technology Trends 466.

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
	tively exploit the high arithmetic performance and the efficient computation exposed by the stream model. How do we structure a hardware implementation of a GPU to ensure the highest overall performance? The first step to building a high-performance GPU is to map kernels in the graphics pipeline to independent functional units on a single chip. Each kernel is thus implemented on a separate area of the chip in an organization known as <i>task parallel</i>, which permits not only task-level parallelism (because all kernels can be run simultaneously) but also hardware specialization of each functional unit to the given kernel. The task-parallel organization also allows efficient communication between kernels: because the functional units implementing neighboring kernels in the graphics pipeline are adjacent on the chip, they can communicate effectively without requiring global memory access. Within each stage of the graphics pipeline that maps to a processing unit on the chip, GPUs exploit the independence of each stream element by processing multiple data elements in parallel. The combination of task-level and data-level parallelism allows GPUs to profitably use dozens of functional units simultaneously. Inputs to the graphics pipeline must be processed by each kernel in sequence. Consequently, it may take thousands of cycles to complete the processing of a single element. If a high-latency memory reference is required in processing any given element, the processing unit can simply work on other elements while the data is being fetched. The deep pipelines of modern GPUs, then, effectively tolerate high-latency operations. For many years, the kernels that make up the graphics pipeline were implemented in graphics hardware as fixed-function units that offered little to no user programmability. In 2000, for the first time, GPUs allowed users the opportunity to program individual kernels in the graphics pipeline. Today's GPUs feature high-performance data-parallel processors that implement two kernels in the graphics pipeline: a <i>vertex program</i> that allows users to run a program on each vertex that passes through the pipeline, and a <i>fragment program</i> that allows users to run a program on each fragment. Both of these stages permit single-precision floating-point computation. Although these additions were primarily intended to provide users with more flexible shading and lighting calculations, their ability to sustain high computation rates on user-specified programs with sufficient precision to address general-purpose computing problems has effectively made them <i>programmable stream processors</i>—that is, processors that are attractive for a much wider variety of applications than simply the graphics pipeline. 29.3 Stream Computation 467

467.

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
469.	generation of hardware has increased its power demand. The future may hold more aggressive dynamic power management targeted at individual stages; increasing amounts of custom or power-aware design for power-hungry parts of the GPU; and more sophisticated cooling management for high-end GPUs. Technology trends indicate that the power demand will only continue to rise with future chip generations, so continued work in this area will remain an important challenge. 29.4.3 Challenge: Supporting More Programmability and Functionality While the current generation of graphics hardware features substantially more programmability than previous generations, the general programmability of GPUs is still far from ideal. One step toward addressing this trend is to improve the functionality and flexibility within the two current programmable units (vertex and fragment). It is likely that we will see their instruction sets converge and add functionality, and that their control flow capabilities will become more general as well. We may even see programmable hardware shared between these two stages in an effort to better utilize these resources. GPU architects will have to be mindful, however, that such improvements not affect the GPU's performance on its core tasks. Another option will be expanding programmability to different units. Geometric primitives particularly benefit from programmability, so we may soon see programmable processing on surfaces, triangles, and pixels. As GPU vendors support more general pipelines and more complex and varied shader computation, many researchers have used the GPU to address tasks outside the bounds of the graphics pipeline. This book contains examples of many of these efforts; the general-purpose computation on GPUs (GPGPU) community has successfully addressed problems in visual simulation, image processing, numerical methods, and databases with graphics hardware. We can expect that these efforts will grow in the future as GPUs continue to increase in performance and functionality. Historically, we have seen GPUs subsume functionality previously belonging to the CPU. Early consumer-level graphics hardware could not perform geometry processing on the graphics processor; it was only five years ago that the entire graphics pipeline could be fabricated on a single chip. Though since that time the primary increase in GPU functionality has been directed toward programmability within the graphics pipeline, we should not expect that GPU vendors have halted their efforts to identify more functions to integrate onto the GPU. In particular, today's games often require large amounts of computation in physics and artificial intelligence computations. Such computation may be attractive for future GPUs. 29.4 The Future and Challenges 469

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
	31.2 An Inventory of GPU Computational Resources 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. 31.2.1 Programmable Parallel Processors GPUs have two types of programmable processors: <i>vertex processors</i> and <i>fragment processors</i>. 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 <i>vertex program</i> (sometimes called a <i>vertex shader</i>) 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 <i>fragments</i> 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 <i>fragment program</i> (sometimes called a <i>pixel shader</i>) to each fragment in the stream to compute the final color of each pixel. Vertex Processors 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. Because of this, vertex processors have hardware to process four-component vectors. This allows them to produce transformed vertex positions in fewer cycles. 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, 31.2 An Inventory of GPU Computational Resources 497 497.

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
	Figure 31-2. Max Reduction Performed with Multiple Passes 31.4 From Analogies to Implementation 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. 31.4.1 Putting It All Together: A Basic GPGPU Framework 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 <i>pug</i>, incorporates all of the analogies from the previous section in order to abstract GPGPU programming in a manner more accessible to experienced CPU programmers. Initializing and Finalizing a GPGPU Application The first step in a GPGPU application is to initialize the GPU. This is very easy in the framework: just call <code>pugInit()</code>. When the application is finished with the GPU, it can clean up the memory used by the graphics API by calling <code>pugCleanup()</code>. Specifying Kernels 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 <i>The Cg Tutorial</i> (Fernando and Kilgard 2003) and the 31.4 From Analogies to Implementation 503

503.

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
	documentation included with the Cg distribution (NVIDIA 2004). There are very few graphics-specific keywords in Cg. We provide a Cg file, <code>pug.cg</code>, that you can include in your kernel files. This file defines a structure called <code>Stream</code> that abstracts the most graphics-centric concept: texture fetching. <code>Stream</code> is a wrapper for the texture sampler Cg type. Calling one of the <code>value()</code> functions of the <code>Stream</code> structure gives the value of a stream element. 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. Stream Management Arrays of data in the framework are called <i>buffers</i>. 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 <code>mode</code> 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. To load initial data into a buffer, pass the data to the <code>pugInitBuffer()</code> function. 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. 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. 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. 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 <code>range</code> parameter of the <code>pugRunProgram()</code> function. 504. 504 Chapter 31 Mapping Computational Concepts to GPUs

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
	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. 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 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. 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>. 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. 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>. 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. 505. 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

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
	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 '461 patent in a manner that would result in the subject matter of this limitation given, at the very least, the person's understanding of the state of the art, the problems addressed and solved in the prior art, and the teachings of GPU Gems. Specific examples of such references are cited and explained in these invalidity contentions, including the cover pleading and charts C1–C15, and Appendix C.
21[d]: shifting control of the data exchange between the user interaction stream and the computational stream to the user interaction stream in response to completion or interruption of the computations representing the artificial neural network;	GPU Gems discloses, expressly and/or inherently, shifting control of the data exchange between the user interaction stream and the computational stream to the user interaction stream in response to completion or interruption of the computations representing the artificial neural network. <i>See e.g.:</i> <i>As a non-limiting example, GPU Gems discloses shifting control of a data exchange between the user interaction stream and the computational stream (e.g., by switching contexts) in response to completion or interruption of the computations.</i>

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
469.	generation of hardware has increased its power demand. The future may hold more aggressive dynamic power management targeted at individual stages; increasing amounts of custom or power-aware design for power-hungry parts of the GPU; and more sophisticated cooling management for high-end GPUs. Technology trends indicate that the power demand will only continue to rise with future chip generations, so continued work in this area will remain an important challenge. 29.4.3 Challenge: Supporting More Programmability and Functionality While the current generation of graphics hardware features substantially more programmability than previous generations, the general programmability of GPUs is still far from ideal. One step toward addressing this trend is to improve the functionality and flexibility within the two current programmable units (vertex and fragment). It is likely that we will see their instruction sets converge and add functionality, and that their control flow capabilities will become more general as well. We may even see programmable hardware shared between these two stages in an effort to better utilize these resources. GPU architects will have to be mindful, however, that such improvements not affect the GPU's performance on its core tasks. Another option will be expanding programmability to different units. Geometric primitives particularly benefit from programmability, so we may soon see programmable processing on surfaces, triangles, and pixels. As GPU vendors support more general pipelines and more complex and varied shader computation, many researchers have used the GPU to address tasks outside the bounds of the graphics pipeline. This book contains examples of many of these efforts; the general-purpose computation on GPUs (GPGPU) community has successfully addressed problems in visual simulation, image processing, numerical methods, and databases with graphics hardware. We can expect that these efforts will grow in the future as GPUs continue to increase in performance and functionality. Historically, we have seen GPUs subsume functionality previously belonging to the CPU. Early consumer-level graphics hardware could not perform geometry processing on the graphics processor; it was only five years ago that the entire graphics pipeline could be fabricated on a single chip. Though since that time the primary increase in GPU functionality has been directed toward programmability within the graphics pipeline, we should not expect that GPU vendors have halted their efforts to identify more functions to integrate onto the GPU. In particular, today's games often require large amounts of computation in physics and artificial intelligence computations. Such computation may be attractive for future GPUs. 29.4 The Future and Challenges 469

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
	31.2 An Inventory of GPU Computational Resources 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. 31.2.1 Programmable Parallel Processors GPUs have two types of programmable processors: <i>vertex processors</i> and <i>fragment processors</i>. 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 <i>vertex program</i> (sometimes called a <i>vertex shader</i>) 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 <i>fragments</i> 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 <i>fragment program</i> (sometimes called a <i>pixel shader</i>) to each fragment in the stream to compute the final color of each pixel. Vertex Processors 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. Because of this, vertex processors have hardware to process four-component vectors. This allows them to produce transformed vertex positions in fewer cycles. 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, 31.2 An Inventory of GPU Computational Resources 497

497.

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
	Figure 31-2. Max Reduction Performed with Multiple Passes 31.4 From Analogies to Implementation 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. 31.4.1 Putting It All Together: A Basic GPGPU Framework 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 <i>pug</i>, incorporates all of the analogies from the previous section in order to abstract GPGPU programming in a manner more accessible to experienced CPU programmers. Initializing and Finalizing a GPGPU Application The first step in a GPGPU application is to initialize the GPU. This is very easy in the framework: just call <code>pugInit()</code>. When the application is finished with the GPU, it can clean up the memory used by the graphics API by calling <code>pugCleanup()</code>. Specifying Kernels 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 <i>The Cg Tutorial</i> (Fernando and Kilgard 2003) and the 31.4 From Analogies to Implementation 503

503.

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
	documentation included with the Cg distribution (NVIDIA 2004). There are very few graphics-specific keywords in Cg. We provide a Cg file, <code>pug.cg</code>, that you can include in your kernel files. This file defines a structure called <code>Stream</code> that abstracts the most graphics-centric concept: texture fetching. <code>Stream</code> is a wrapper for the texture sampler Cg type. Calling one of the <code>value()</code> functions of the <code>Stream</code> structure gives the value of a stream element. 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. Stream Management Arrays of data in the framework are called <i>buffers</i>. 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 <code>mode</code> 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. To load initial data into a buffer, pass the data to the <code>pugInitBuffer()</code> function. 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. 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. 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. 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 <code>range</code> parameter of the <code>pugRunProgram()</code> function. 504 Chapter 31 Mapping Computational Concepts to GPUs 504.

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
	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. 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 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. 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>. 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. 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>. 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. 31.5 A Simple Example 505 505.

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
649.	Chapter 40 Computer Vision on the GPU <i>James Fung</i> University of Toronto Computer vision tasks are computationally intensive and repetitive, and they often exceed the real-time capabilities of the CPU, leaving little time for higher-level tasks. However, many computer vision operations map efficiently onto the modern GPU, whose programmability allows a wide variety of computer vision algorithms to be implemented. This chapter presents methods of efficiently mapping common mathematical operations of computer vision onto modern computer graphics architecture. 40.1 Introduction 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. 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. 40.1 Introduction 649

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
	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. 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. 40.2 Implementation Framework Many computer vision operations can be considered sequences of filtering operations, with each sequential filtering stage acting upon the output of the previous filtering stage. On the GPU, these filtering operations are carried out by fragment programs. To apply these fragment programs to input images, the input images are initialized as textures and then mapped onto quadrilaterals. By displaying these quadrilaterals in appropriately sized windows, we can ensure that there is a one-to-one correspondence of image pixels to output fragments. When the textured quadrilateral is displayed, the fragment program then runs, operating identically on each pixel of the image. The fragment program is analogous to the body of a common for-loop program statement, but each iteration of the body can be thought of as executing in parallel. The resulting output is not the original image, but rather the filtered image. Modern GPUs allow input and results in full 32-bit IEEE floating-point precision, providing sufficient accuracy for many algorithms. A complete computer vision algorithm can be created by implementing sequences of these filtering operations. After the texture has been filtered by the fragment program, the resulting image is placed into texture memory, either by using render-to-texture extensions or by copying the frame buffer into texture memory. In this way, the output image becomes the input texture to the next fragment program. This creates a pipeline that runs the entire computer vision algorithm. However, often a complete vision algorithm will require operations beyond filtering. For example, summations are common operations; we present examples of how they can be implemented and used. Furthermore, more-generalized calculations, such as feature tracking, can also be mapped effectively onto graphics hardware. 650 Chapter 40 Computer Vision on the GPU 650.

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
651.	These methods are implemented in the OpenVIDIA GPU computer vision library, which can be used to create real-time computer vision programs (http://openvidia.sourceforge.net/). 40.3 Application Examples This chapter presents examples that implement common computer vision techniques on the GPU. The common thread between each of these algorithms is that they are all used in our system to create a fully interactive, real-time <i>computer-mediated reality</i>, in which the environment of a user appears to be augmented, diminished, or otherwise altered. Some examples are shown in Figure 40-1.  Figure 40-1. Using Computer Vision to Mediate Reality <i>Computer vision techniques can analyze video to augment, diminish, or otherwise alter a user's perception of the environment. On the left, detected ceiling planes become terminal windows. On the right, a terminal window is mapped to the plane of the floppy disk.</i> 40.3.1 Using Sequences of Fragment Programs for Computer Vision In this section, we demonstrate how a sequence of fragment programs can work together to perform more-complex computer vision operations. Correcting Radial Distortion Many camera lenses cause some sort of radial distortion (also known as <i>barrel distortion</i>) of the image, most commonly in wide-field-of-view or low-cost lenses. There are several ways of correcting for radial distortion on the graphics hardware; we present an example of how a fragment program can be used to do so. 40.3 Application Examples 651

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
	We begin by assuming that the displacement of a pixel due to radial distortion is Δx, Δy and is commonly modeled by: $\Delta x = x(k_1 r^2 + k_2 r^4 + \dots),$ $\Delta y = y(k_1 r^2 + k_2 r^4 + \dots),$ where x, y are the image coordinates, r is the distance of the pixel from the principal point of the camera, and the coefficients $k_{1,2,3, \dots}$ are camera parameters determined by some known calibration method. Given a particular camera, the required calibration parameters can be found using the tools provided at Bouquet 2004. The fragment program shown in Listing 40-1 applies this correction by using the current texture coordinates to calculate an offset. The program then adds this offset to the current pixel's coordinates and uses the result to look up the texture value at the corrected coordinate. This texture value is then output, and the image's radial distortion is thereby corrected. Listing 40-1. Radial Undistortion <pre> float4 CorrectBarrelDistortion(float2 iptexCoord : TEXCOORD0, float4 wpos : WPOS, uniform samplerRECT FPEL : COLOR) { // known constants for the particular lens and sensor float f = 368.26488; // focal length float ox = 147.54834; // principal point, x axis float oy = 126.01673; // principal point, y axis float k1 = 0.4162; // constant for radial distortion correction float k2 = 0.40348; float2 xy = (iptexCoord - float2(ox, oy))/float2(f,f); r = sqrt(dot(xy, xy)); float r2 = r * r; float r4 = r2 * r2; float coeff = (k1 * r2 + k2 * r4); float2 correctedCoord = xy + float2(xy.x * coeff, xy.y * coeff); float4 texel = FPEL.Sample(correctedCoord); return texel; } </pre> 652 Chapter 40 Computer Vision on the GPU

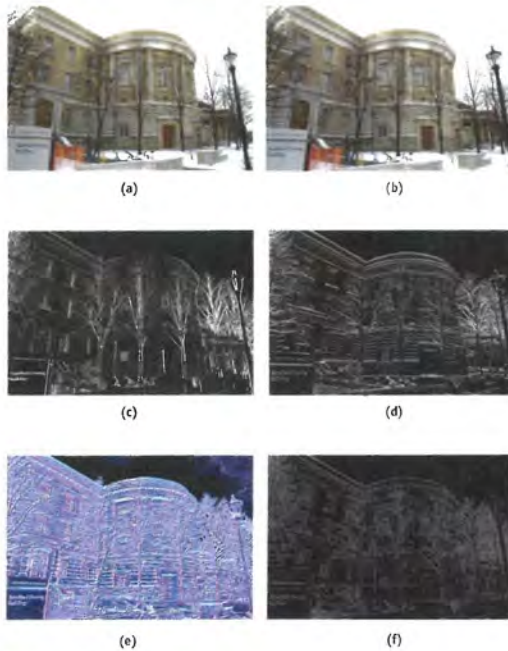
652.

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
	Listing 40-1 (continued). Radial Undistortion <pre> // add the calculated offsets to the current texture coordinates xy = ((xy + xy * coeff.xx) * f.xx) + float2(ox, oy); // look up the texture at the corrected coordinates // and output the color return texRECT(FPE1, xy); } </pre> A Canny Edge Detector Edge detection is a key algorithm used in many vision applications. Here we present an implementation of the commonly used "Canny" edge-detection algorithm that runs entirely on the GPU. See Figure 40-2 for an example. We implement the Canny edge detector as a series of fragment programs, each performing a step of the algorithm: • Step 1. Filter the image using a separable Gaussian edge filter (Jargstorff 2004). The filter footprint is typically around 15×15 elements wide, and separable. A separable filter is one whose 2D mask can be calculated by applying two 1D filters in the x and y directions. The separability saves us a large number of texture lookups, at the cost of an additional pass. • Step 2. Determine the magnitude, l, of the derivatives at each pixel computed in step 1, and quantize the direction d. The values of d and l are given by: $d = \arctan 2(d_y, d_x),$ $l = \sqrt{d_x^2 + d_y^2},$ where d is the direction of the gradient vector, quantized to one of the eight possible directions one may traverse a 3×3 region of pixels. The length l is the magnitude of the vector. • Step 3. Perform a nonmaximal suppression, as shown in Listing 40-2. The nonmaximal suppression uses the direction of the local gradient to determine which pixels are in the forward and backward direction of the gradient. This direction is used to calculate the x and y offset of the texture coordinates to look up the forward and backward neighbors. An edge is considered found only if the gradient is at a maximum as compared to its neighbors, ensuring a thin line along the detected edge. 40.3 Application Examples 653

653.

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
	 (a) (b) (c) (d) (e) (f) Figure 40-2. Canny Edge Detection Each image shows the result of a fragment program operating on the previous image. (a) Original input. (b) Radial undistorted image. (c) Horizontal derivative. (d) Vertical derivative. (e) Gradient quantization. (f) Canny edges. 654 Chapter 40 Computer Vision on the GPU 654.

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
655.	Listing 40-2. A Canny Edge Detector Using Nonmaximal Suppression <i>This program looks up the direction of the gradient at the current pixel and then retrieves the pixels in the forward and backward directions along the gradient. The program returns 1 if the current pixel value is greater than the two values along the gradient; otherwise, it suppresses (or zeroes) the output. A thresholding value allows the sensitivity of the edge detector to be varied in real time to suit the vision apparatus and environment.</i> <pre> float4 CannySearch(uniform float4 thresh, float2 fptexCoord : TEXCOORD0, uniform samplerRECT FPEL1 : COLOR) { // magdir holds { dx, dy, mag, diract } float4 magdir = texRECT(FPEL1, fptexCoord); float alpha = 0.5/sin(3.14159/8); // eight directions on grid float2 offset = round(alpha.xx * magdir.xy/magdir.zx); float4 fwdneighbour, backneighbour; fwdneighbour = texRECT(FPEL1, fptexCoord + offset); backneighbour = texRECT(FPEL1, fptexCoord - offset); float4 color0; if (fwdneighbour.z > magdir.z && backneighbour.z > magdir.z) color0 = float4(0.0, 0.0, 0.0, 0.0); // not an edge! else color0 = float4(1.0, 1.0, 1.0, 1.0); // is an edge! if (magdir.z < thresh.x) color0 = float4(0.0, 0.0, 0.0, 0.0); // thresholding return color0; } </pre> 40.3.2 Summation Operations Many common vision operations involve the summation of an image buffer. Some methods that require the summation are moment calculations and the solutions to linear systems of equations. Buck and Purcell (2004) discuss a general method of efficiently summing buffers by using local neighborhoods. On systems that do not support render-to-texture, however, the number of passes required by this technique can be limiting, due to the number of graphics context switches involved. We have found performing the computation in two 40.3 Application Examples 655

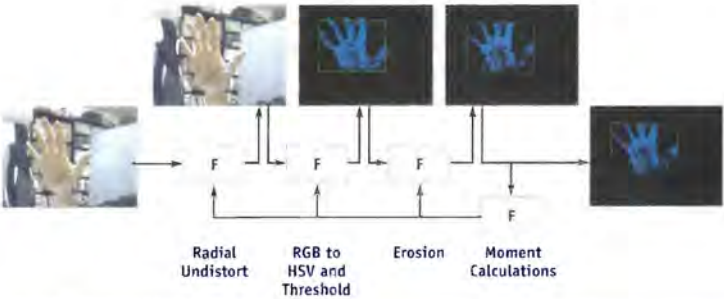
Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
	passes to be more efficient on such systems. In our approach, we create a fragment program that sums each row of pixels, resulting in a single column holding the sums of each row. Then, in a second pass, we draw a single pixel and execute a program that sums down the column. The result is a single pixel that has as its "color" the sum of the entire buffer. The Cg code for computing the sum in this way is given in this chapter's appendix on the book's CD. We use the summation operation within the context of two different, but commonly used, algorithms that require summation over a buffer: a hand-tracking algorithm and an algorithm for creating panoramas from images by determining how to stitch them together. Tracking Hands Tracking a user's hands or face is a useful vision tool for creating interactive systems, and it can be achieved on the GPU by using a combination of image filtering and moment calculations. A common way to track a user's hand is to use color segmentation to find areas that are skin-colored, and then continually track the location of those areas through the image. Additionally, it is useful to track the average color of the skin tone, because it changes from frame to frame, typically due to changes in lighting. It is common to carry out the segmentation in an HSV color space, because skin tone of all hues tends to cluster in HSV color space. The HSV color space describes colors in terms of their <i>hue</i> (the type of color, such as red, green, or blue); <i>saturation</i> (the vibrancy of that color); and <i>value</i> (the brightness of the color). Cameras typically produce an RGB image, so a fragment program can be used to perform a fast color space conversion as a filtering operation. Assuming that we have some sort of initial guess for the HSV color of the skin, we can also use a fragment program to threshold pixels. Given a particular pixel's HSV value, we can determine its difference from the mean HSV skin color and either output to 0 if the difference is too great or output the HSV value otherwise. For instance, the comparison might be easily done as: <pre> if (distance(hsv, meanhsv) < thresh) { colorO = hsv; } else colorO = float4(0.0, 0.0, 0.0, 0.0); </pre> 656 Chapter 40 Computer Vision on the GPU 656.

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
	Finally, we determine the centroid of the skin-colored region and update the mean skin value based on what is currently being seen. For both of these steps, we need to calculate sums over the image. To compute centroids, we can first compute the first three moments of the image: $M_{00} = \sum_{x=0}^{W-1} \sum_{y=0}^{H-1} E(x, y),$ $M_{10} = \sum_{x=0}^{W-1} \sum_{y=0}^{H-1} xE(x, y),$ $M_{01} = \sum_{x=0}^{W-1} \sum_{y=0}^{H-1} yE(x, y),$ where W and H are the sizes of the image in each axis, and $E(x, y)$ is the pixel value at location (x, y). The x coordinate of the centroid is then computed as M_{10}/M_{00}, and M_{01}/M_{00} is the centroid's y coordinate. These summations can be done by first running a fragment program that fills a buffer with: $E(x, y), \quad xE(x, y), \quad yE(x, y)$ and then using a GPU summation method to sum the entire buffer. The result will be a single pixel whose value can then be read back to the CPU to update the location of the centroid. In our case—tracking the centroid of a thresholded skin-tone blob—we might simply let the fragment program output 1.0 in skin-colored areas and 0 elsewhere. Listing 40-3 shows a Cg program that produces four output components that, after summation, will provide the sums needed to calculate the zeroth and first-order moments. Figure 40-3 depicts this process. Should we require more statistics like the preceding, we could either reduce the precision and output eight 16-bit floating-point results to a 128-bit floating-point texture or use the multiple render targets (MRTs) OpenGL extension. 40.3 Application Examples 657 657.

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
	Listing 40-3. Creating Output Statistics for Moment Calculations <pre> float4 ComputeMoments(float2 texCoord : TEXCOORD0, uniform samplerRECT FPE1: TEXUNIT0) : COLOR { float4 samp = texRECT(FPE1, fptexCoord); float4 colorO = float4(0, 0, 0, 0); if (samp.x != 0.0) colorO = float4(1.0, fptexCoord.x, fptexCoord.y, samp.y); } </pre>  Figure 40-3. Tracking a User's Hand 40.3.3 Systems of Equations for Creating Image Panoramas We also encounter this compute-and-sum type of calculation when we form systems of equations of the familiar form $Ax = b$. VideoOrbits VideoOrbits is an image-registration algorithm that calculates a coordinate transformation between pairs of images taken with a camera that is free to pan, tilt, rotate about its optical axis, and zoom. The technique solves the problem for two cases: (1) images taken from the same location of an arbitrary 3D scene or (2) images taken from arbitrary locations of a flat scene. Figure 40-4 shows an image panorama composed from several frames of video. Equivalently, Mann and Fung (2002) show how the same algorithm can be used to replace planes in the scene with computer-generated content, such as in Figure 40-1. 658 Chapter 40 Computer Vision on the GPU

658.

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
	Figure 40-4. A VideoOrbits Panorama <i>This panorama shows how the images have been projected to fit together properly.</i> VideoOrbits solves for a projection of an image in order to register it with another image. Consider a pair of images of the same scene, with the camera having moved between images. A given point in the scene appearing at coordinates $[x, y]$ in the first image now appears at coordinates $[x', y']$ in the second image. Under the constraints mentioned earlier, all the points in the image can be considered to move according to eight parameters, given by the following equation: $\begin{bmatrix} x' \\ y' \end{bmatrix} = \frac{\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \end{bmatrix}}{\begin{bmatrix} c_1 & c_2 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + 1} \quad (1)$ The eight scalar parameters describing the projection are denoted by: $\mathbf{p} = [a_{11}, a_{12}, b_1, a_{21}, a_{22}, b_2, c_1, c_2]$ and are calculated by VideoOrbits. Once the parameters $\mathbf{p}$ have been determined for each image, the images can then be transformed into the same coordinate system to 40.3 Application Examples 659 659.

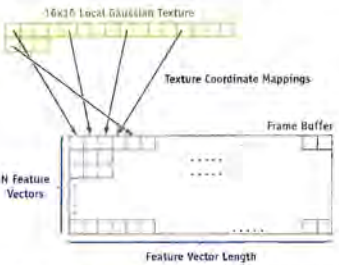
Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
	create the image panoramas of Figure 40-4. (For a more in-depth treatment of projective flow and the VideoOrbits algorithm, see Mann 1998.) VideoOrbits calculates the parameters p by solving a linear system of equations. The bulk of the computation is in the initialization of this system of equations, which is of the form $Ax = b$. Here, A is an 8×8 matrix whose values we must initialize, x is an 8×1 column vector that corresponds to p, and b is an 8×1 column vector we must also initialize. Solving this system is computationally expensive because each element of A and b is actually a summation of a series of multiplications and additions at each point in the image. Initializing each of the 64 entries of the matrix A involves many multiplications and additions over all the pixels. A is of the form: $A = \begin{bmatrix} \sum a_{11} & \sum a_{12} & \sum a_{13} & \dots & \sum a_{18} \\ \sum a_{21} & \sum a_{22} & \sum a_{23} & \dots & \sum a_{28} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \sum a_{81} & \sum a_{82} & \sum a_{83} & \dots & \sum a_{88} \end{bmatrix}_{8 \times 8} \quad (2)$ where each of the elements is a summation. (Note that the elements a_{11}, a_{13}, a_{21}, a_{23} are not those of Equation 1.) The arguments of these summations involve many arithmetic operations. For instance, element a_{21} is: $a_{21} = (x^2 E_x + xy E_y)(xy E_x + y^2 E_y). \quad (3)$ This operation must be carried out at every pixel and then summed to obtain each entry in matrix A. This compute-and-sum operation is similar to moment calculations discussed previously. Not surprisingly, we can calculate this on the GPU in a similar fashion, filling a buffer with the argument of the summation and then summing the buffer. Typically, to calculate all the matrix entries, multiple passes will be required. As each summation is completed, the results are copied to a texture that stores the results. After all the entries for the matrix A and vector b are computed, the values are read back to the CPU. This keeps the operations on the GPU, deferring the readback until all matrix entries have been calculated. This allows the GPU to operate with greater independence from the CPU. The system can then be solved on the CPU, with the majority of the per-pixel calculations already solved by the GPU. 660 Chapter 40 Computer Vision on the GPU 660.

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
661.	40.3.4 Feature Vector Computations Calculating <i>feature vectors</i> at detected feature points is a common operation in computer vision. A <i>feature</i> in an image is a local area around a point with some higher-than-average amount of "uniqueness." This makes the point easier to recognize in subsequent frames of video. The uniqueness of the point is characterized by computing a feature vector for each feature point. Feature vectors can be used to recognize the same point in different images and can be extended to more generalized object recognition techniques. Figure 40-5 shows the motion of the features between frames as the camera moves.  Figure 40-5. Detecting Distinctive Points as the Camera Pans <i>The GPU is used to detect and track distinctive points (features) in real time as the camera pans. The green lines connect the same point in the previous image to its new location in the current image.</i> Computing feature vectors on the GPU requires handling the image data in a different manner than the operations we have thus far presented. Because only a few points in the image are feature points, this calculation does not need to be done at every point in the image. Additionally, for a single feature point, a large feature vector (100 elements or more) is typically calculated. Thus, each single point produces a larger number of results, rather than the one-to-one or many-to-one mappings we have discussed so far. These types of computations can still be carried out on the GPU. We take as an example a feature vector that is composed of the filtered gradient magnitudes in a local region around a located feature point. Feature detection can be achieved using methods similar to the Canny edge detector that instead search for corners rather than lines. If the feature points are being detected using sequences of filtering, as is common, we can perform the filtering on the GPU 40.3 Application Examples 661

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
	and read back to the CPU a buffer that flags which pixels are feature points. The CPU can then quickly scan the buffer to locate each of the feature points, creating a list of image locations at which we will calculate feature vectors on the GPU. There are many approaches to generating feature vectors in order to track feature points. We show one approach as an example. For each detected feature point, we examine a 16×16 neighborhood of gradient magnitudes and directions (as computed in the Canny edge example). This neighborhood is divided up into 4×4-pixel regions, giving 16 different regions of 16 pixels. In each region, we create an 8-element histogram of gradient magnitudes. Each pixel's gradient magnitude is added to the bin of the histogram corresponding to its quantized gradient direction. For each region, the result is then an 8-element histogram. The feature vector is the concatenation of all 16 of these 8-element histograms, giving a feature vector length of $16 \times 8 = 128$ elements. Features in different images can be matched to each other by computing the Euclidean distance between pairs of feature vectors, with the minimum distance providing a best match. To produce the 128-element feature vector result, we draw rows of vertices, with one row for each feature, such that each drawn point in the row corresponds to an element of the feature vector for that feature. Associating texture values pointwise allows for the most flexibility in mapping texture coordinates for use in computing the feature vector. Figure 40-6 shows how the vertices are laid out, and the binding of the image texture coordinates. Each point has different texture coordinates mapped to it to allow a single fragment program to access the appropriate gradients of the local region used to calculate the histogram.  Figure 40-6. OpenGL Vertex Layout and Texture Bindings for Feature Vertices 662 Chapter 40 Computer Vision on the GPU 662.

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
	By associating the correct texture coordinates for each point vertex, the single fragment program shown in Listing 40-4 calculates the feature vectors. To produce 8 elements per region, we have packed the results into 16-bit half-precision floating-point values; alternatively, we could use MRTs to output multiple 32-bit values. After completion, we read back the frame buffer, containing one feature vector per row, to the CPU. <u>Listing 40-4. Fragment Program for Calculating Feature Vectors</u> <pre> Float4 FeatureVector() { float2 fptexCoord : TEXCOORD0, float2 tapsCoord : TEXCOORD1, float4 colorZero : COLOR0, float4 wpos : WPOS, uniform samplerRECT FPE1, uniform samplerRECT FPE2 : COLOR { const float4x4 lclGauss = (0.3536, 0.3536, 0.3536, 0.3536, 0.3536, 0.7071, 0.7071, 0.3536, 0.3536, 0.7071, 0.7071, 0.3536, 0.3536, 0.3536, 0.3536, 0.3536); const float eps = 0.001; float4 input; int xoffset = 1, yoffset = 1; float4 bins_0 = float4(0.0, 0.0, 0.0, 0.0); float4 bins_1 = float4(0.0, 0.0, 0.0, 0.0); for (xoffset = 0 ; xoffset < 4 ; xoffset++) { for (yoffset = 0 ; yoffset < 4 ; yoffset++) { input = texRECT(FPE1, fptexCoord + float2(xoffset, yoffset)); float dir = input.y; input *= lclGauss[xoffset,yoffset].xxxx * texRECT(FPE2, tapsCoord + float2(xoffset, yoffset)).xxxx; if (dir <= (1.0 + eps)/8.0) bins_0.x += input.x; /* . . . for each bin . . . */ else if (dir <= (7.0 + eps)/8.0) bins_1.x += input.x; else bins_1.w += input.x; } } return float4(pack_2half(bins_0.xy), pack_2half(bins_0.zw), pack_2half(bins_1.xy), pack_2half(bins_1.zw)) } </pre> 40.3 Application Examples 663 663.

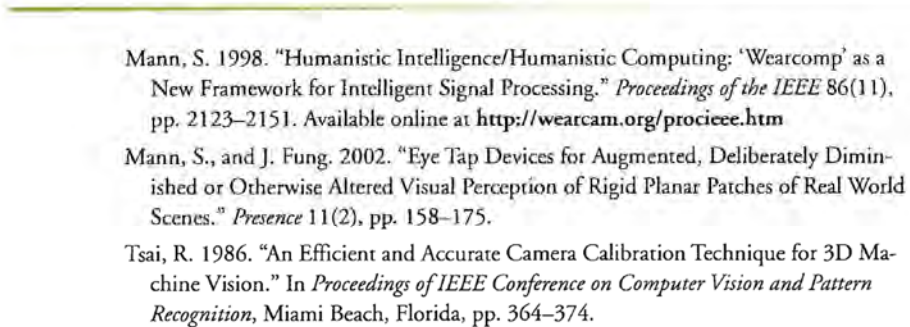
Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
	40.4 Parallel Computer Vision Processing When multiple computer vision algorithms are needed for an application, parallel GPU processing can be achieved by placing multiple graphics cards in the PCI bus slots of a motherboard. Figure 40-7 shows a machine built to explore the benefits of parallel computations by using multiple graphics cards. Running each computer vision program on a different GPU allows the cards to process in parallel; conventional shared memory can be used to allow the programs to interact. Although creating multi-GPU systems using PCI cards will soon no longer be possible due to the obsolescence of the PCI bus, the advent of PCI Express has made new multi-GPU approaches possible. One such approach is NVIDIA's SLI, which makes the presence of multiple GPUs transparent to the developer by automatically distributing work between them.  Figure 40-7. Multiple GPUs Used for a Computer Vision Application <i>Left: A computer vision application (Fung and Mann 2002) using graphics hardware to display six video streams in a hexagonal form. Right: A computer vision machine that runs computer vision tasks in parallel using six PCI graphics cards and one AGP graphics card. Each PCI card is a GeForce FX 5200 GPU, resulting in an inexpensive, powerful, and easily constructible parallel architecture well suited for pattern recognition and computer vision.</i> 40.5 Conclusion This chapter has presented techniques for mapping computer vision algorithms onto modern GPUs. We hope that this chapter has provided an understanding of how to write computer vision algorithms for fast processing on the GPU. Although other 664 Chapter 40 Computer Vision on the GPU 664.

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
	special-purpose hardware systems could be used to provide hardware acceleration of computer vision algorithms, the low cost and widespread availability of GPUs will make hardware-accelerated computer vision algorithms more accessible. 40.6 References The methods presented here are implemented in the OpenVIDIA project. OpenVIDIA is an open-source project that includes library functions and sample programs that run computer vision applications on the GPU. For more information, visit OpenVIDIA's Web site: http://openvidia.sourceforge.net. Bouguet, J.-Y. 2004. Camera Calibration Toolbox for Matlab. Available online at: http://www.vision.caltech.edu/bouguetj/calib_doc/index.html Bradski, G. 1998. "Computer Vision Face Tracking for Use in a Perceptual User Interface." <i>Intel Technology Journal</i>. Buck, L., and T. Purcell. 2004. "A Toolkit for Computation on GPUs." In <i>GPU Gems</i>, edited by Randima Fernando, pp. 621–636. Addison-Wesley. Canny, J. F. 1986. "A Computational Approach to Edge Detection." <i>IEEE PAMI</i> 8(6), pp. 679–698. Fung, J., and S. Mann. 2004a. "Computer Vision Signal Processing on Graphics Processing Units." In <i>Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP 2004)</i>, Montreal, Quebec, Canada, pp. 83–89. Fung, J., and S. Mann. 2004b. "Using Multiple Graphics Cards as a General Purpose Parallel Computer: Applications to Computer Vision." In <i>Proceedings of the 17th International Conference on Pattern Recognition (ICPR2004)</i>, Cambridge, United Kingdom, pp. 805–808. Fung, J., F. Tang, and S. Mann. 2002. "Mediated Reality Using Computer Graphics Hardware for Computer Vision." In <i>Proceedings of the International Symposium on Wearable Computing 2002 (ISWC2002)</i>, Seattle, Washington, pp. 83–89. Horn, B., and B. Schunk. 1981. "Determining Optical Flow." <i>Artificial Intelligence</i> 17, pp. 185–203. Jangstorff, E. 2004. "A Framework for Image Processing." In <i>GPU Gems</i>, edited by Randima Fernando, pp. 445–467. Addison-Wesley. Lowe, D. 2004. "Distinctive Image Features from Scale-Invariant Keypoints." <i>International Journal of Computer Vision</i> 60(2), pp. 91–110. 40.6 References 665 665.

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
	 666. 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 '461 patent in a manner that would result in the subject matter of this limitation given, at the very least, the person's understanding of the state of the art, the problems addressed and solved in the prior art, and the teachings of GPU Gems. Specific examples of such references are cited and explained in these invalidity contentions, including the cover pleading and charts C1–C15, and Appendix C.
21[e]: queueing a user command received by the user interaction stream during execution of the computations representing the artificial neural network; and	GPU Gems discloses, expressly and/or inherently, queueing a user command received by the user interaction stream during execution of the computations representing the artificial neural network. <i>See e.g.</i>:

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
	<i>As a non-limiting example, GPU Gems discloses queueing a user command (e.g., scheduling a task associated with a user interaction) received by the user interaction stream during execution of the computations representing the artificial neural network.</i>

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

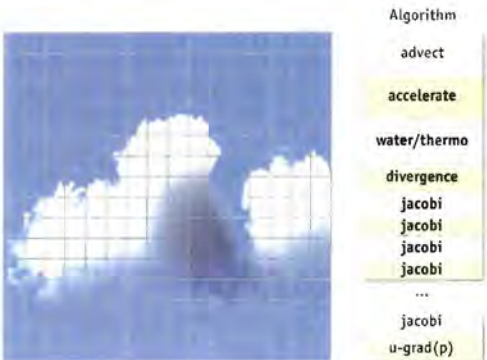
U.S. Patent No. RE49,461	GPU Gems
483.	distance passes the test, it's in light; if not, it's in shadow. NVIDIA GPUs have dedicated transistors to perform four z-compare per pixel (on four neighboring z-values) per clock, and to perform bilinear filtering of the pass/fail data. This more advanced variation of percentage-closer filtering saves many shader instructions compared to GPUs that don't have direct shadow buffer support. High-Dynamic-Range Blending Using fp16 Surfaces, Texture Filtering, and Blending GeForce 6 Series GPUs allow for fp16×4 (four components, each represented by a 16-bit float) filtered textures in the pixel shaders; they also allow performing all alpha-blending operations on fp16×4 filtered surfaces. This permits intermediate rendered buffers at a much higher precision and range, enabling high-dynamic-range rendering, motion blur, and many other effects. In addition, it is possible to specify a separate blending function for color and alpha values. (The lowest-end member of the GeForce 6 Series family, the GeForce 6200 TC, does not support floating-point blending or floating-point texture filtering because of its lower memory bandwidth, as well as to save area on the chip.) 30.3.2 Shader Model 3.0 Programming Model Along with the fixed-function features listed previously, the capabilities of the vertex and the fragment processors have been enhanced in GeForce 6 Series GPUs. With Shader Model 3.0, the programming models for vertex and fragment processors are converging; both support fp32 precision, texture lookups, and the same instruction set. Specifically, here are the new features that have been added. Vertex Processor • Increased instruction count. The total instruction count is now 512 static instructions and 65,536 dynamic instructions. The static instruction count represents the number of instructions in a program as it is compiled. The dynamic instruction count represents the number of instructions actually executed. In practice, the dynamic count can be much higher than the static count due to looping and subroutine calls. • More temporary registers. Up to 32 four-wide temporary registers can be used in a vertex program. • Support for instancing. This enhancement was described earlier. 30.3 GPU Features 483

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems												
	Table 30-2. Overhead Incurred When Executing Flow-Control Operations in Fragment Programs <table> <tr> <th>Instruction</th><th>Cost (Cycles)</th></tr> <tr> <td>If / endif</td><td>4</td></tr> <tr> <td>If / else / endif</td><td>6</td></tr> <tr> <td>Call</td><td>2</td></tr> <tr> <td>Ret</td><td>2</td></tr> <tr> <td>Loop / endloop</td><td>4</td></tr> </table> Furthermore, branching in the fragment processor is affected by the level of divergence of the branches. Because the fragment processor operates on hundreds of pixels per instruction, if a branch is taken by some fragments and not others, all fragments execute both branches, but only writing to the registers on the branches each fragment is supposed to take. For low-frequency and mid-frequency branch changes, this effect is hidden, although it can become a limiter as the branch frequency increases. 30.3.3 Supported Data Storage Formats Table 30-3 summarizes the data formats supported by the graphics pipeline. 30.4 Performance The GeForce 6800 Ultra is the flagship product of the GeForce 6 Series family at the time of writing. Its performance is summarized as follows: • 425 MHz internal graphics clock • 550 MHz memory clock • 600 million vertices/second • 6.4 billion texels/second • 12.8 billion pixels/second, rendering z/stencil-only (useful for shadow volumes and shadow buffers) • 6 four-wide fp32 vector MADs per clock cycle in the vertex shader, plus one scalar multi-function operation (a complex math operation, such as a sine or reciprocal square root) • 16 four-wide fp32 vector MADs per clock cycle in the fragment processor, plus 16 four-wide fp32 multiplies per clock cycle • 64 pixels per clock cycle early z-cull (reject rate) As you can see, there's plenty of programmable floating-point horsepower in the vertex and fragment processors that can be exploited for computationally demanding problems. 488 Chapter 30 The GeForce 6 Series GPU Architecture	Instruction	Cost (Cycles)	If / endif	4	If / else / endif	6	Call	2	Ret	2	Loop / endloop	4
Instruction	Cost (Cycles)												
If / endif	4												
If / else / endif	6												
Call	2												
Ret	2												
Loop / endloop	4												

488.

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
	 Figure 31-1. Simulation on a Grid <i>This cloud simulation executes on the GPU. The data streams used in the simulation are represented on the grid (shown with exaggerated coarseness) and stored in textures. The algorithm is shown at right; each step in the algorithm is a kernel, implemented using a fragment program on the GPU.</i> 31.1.3 Stream Communication: Gather vs. Scatter 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 two main types of communication: <i>gather</i> and <i>scatter</i>. 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. 496 Chapter 31 Mapping Computational Concepts to GPUs

496.

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
	31.2 An Inventory of GPU Computational Resources 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. 31.2.1 Programmable Parallel Processors GPUs have two types of programmable processors: <i>vertex processors</i> and <i>fragment processors</i>. 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 <i>vertex program</i> (sometimes called a <i>vertex shader</i>) 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 <i>fragments</i> 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 <i>fragment program</i> (sometimes called a <i>pixel shader</i>) to each fragment in the stream to compute the final color of each pixel. Vertex Processors 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. Because of this, vertex processors have hardware to process four-component vectors. This allows them to produce transformed vertex positions in fewer cycles. 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, 31.2 An Inventory of GPU Computational Resources 497

497.

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
554.	34.3 Data-Dependent Looping with Occlusion Queries Another GPU feature designed to avoid drawing what is not visible is the hardware <i>occlusion query</i>. This feature enables you to query the number of pixels updated by a rendering call. Such queries are pipelined, which means that they provide a way to get a limited amount of data (an integer count) back from the GPU without stalling the pipeline, as occurs when we read back actual pixels. Because in GPGPU we are almost always drawing quads with known pixel coverage, we can use occlusion query with the <code>discard</code> keyword discussed in Section 34.2.3 to get a count of fragments updated and discarded. This allows us to implement global decisions controlled by the CPU based on GPU processing. Suppose we have a computation that needs to proceed iteratively on elements of a stream until all elements satisfy a termination criterion. To implement this on the GPU, we write a fragment program that implements the computation and one that tests the termination criterion. The latter program <code>discards</code> fragments that satisfy the termination criterion. We then write a loop on the CPU like the following pseudocode, which will execute until all stream elements satisfy the termination criteria. <pre> int numberNotTerminated = streamSize; while (numberNotTerminated > 0) { gpuRunProgram(computationProgram); gpuBeginQuery(myQuery); gpuRunProgram(terminationProgram); gpuEndQuery(myQuery); numberNotTerminated = gpuGetQueryResults(myQuery); } </pre> This technique can also be used for subdivision algorithms, such as the adaptive radiosity solution in Chapter 39 of this book, “Global Illumination Using Progressive Refinement Radiosity.”

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
	Chapter 35 GPU Program Optimization <i>Cliff Woolley</i> <i>University of Virginia</i> As GPU programmability has become more pervasive and GPU performance has become almost irresistibly appealing, increasing numbers of programmers have begun to recast applications of all sorts to make use of GPUs. But an interesting trend has appeared along the way: it seems that many programmers make the same performance mistakes in their GPU programs regardless of how much experience they have programming CPUs. The goal of this chapter is to help CPU programmers who are new to GPU programming avoid some of these common mistakes so that they gain the benefits of GPU performance without all the headaches of the GPU programmer's learning curve. 35.1 Data-Parallel Computing One of the biggest hurdles you'll face when first programming a GPU is learning how to get the most out of a data-parallel computing environment. This parallelism exists at several levels on the GPU, as described in Chapter 29 of this book, "Streaming Architectures and Technology Trends." First, parallel execution on multiple data elements is a key design feature of modern GPUs. Vertex and fragment processors operate on four-vectors, performing four-component instructions such as additions, multiplications, multiply-accumulates, or dot products in a single cycle. They can schedule more than one of these instructions per cycle per pipeline. This provides ample opportunities for the extraction of instruction-level parallelism within a GPU program. For example, a 35.1 Data-Parallel Computing 557 557.

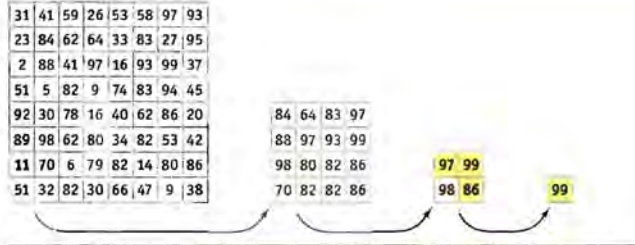
Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
	29.4 The Future and Challenges The migration of GPUs into programmable stream processors reflects the culmination of several historical trends. The first trend is the ability to concentrate large amounts of computation on a single processor die. Equally important has been the ability and talent of GPU designers in effectively using these computation resources. The economies of scale that are associated with building tens of millions of processors per year have allowed the cost of a GPU to fall enough to make a GPU a standard part of today's desktop computer. And the addition of reasonably high-precision programmability to the pipeline has completed the transition from a hard-wired, special-purpose processor to a powerful programmable processor that can address a wide variety of tasks. What, then, can we expect from future GPU development? 29.4.1 Challenge: Technology Trends Each new generation of hardware will present a challenge to GPU vendors: How can they effectively use additional hardware resources to increase performance and functionality? New transistors will be devoted to increased performance, in large part through greater amounts of parallelism, and to new functionality in the pipeline. We will also see these architectures evolve with changes in technology. As we described in Section 29.1.2, future architectures will increasingly use transistors to replace the need for communication. We can expect more aggressive caching techniques that not only alleviate off-chip communication but also mitigate the need for some on-chip communication. We will also see computation increasingly replace communication when appropriate. For example, the use of texture memory as a lookup table may be replaced by calculating the values in that lookup table dynamically. And instead of sending data to a distant on-chip computation resource and then sending the result back, we may simply replicate the resource and compute our result locally. In the trade-off between <i>communicate</i> and <i>recompute/cache</i>, we will increasingly choose the latter. The increasing cost of communication will also influence the microarchitecture of future chips. Designers must now explicitly plan for the time required to send data across a chip; even local communication times are becoming significant in a timing budget. 29.4.2 Challenge: Power Management Ideas for how to use future GPU transistors must be tempered by the realities of their costs. Power management has become a critical piece of today's GPU designs as each 468 Chapter 29 Streaming Architectures and Technology Trends 468.

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

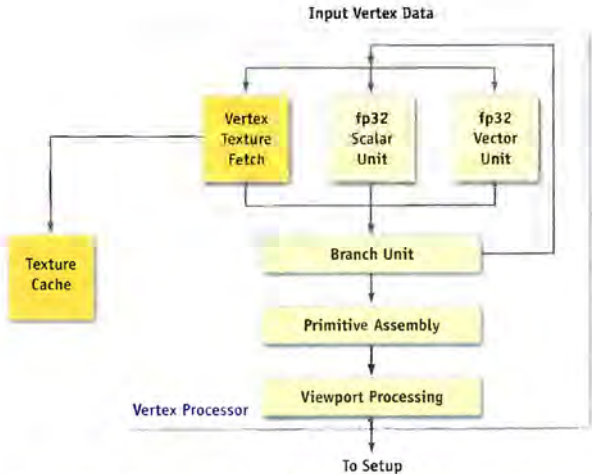
U.S. Patent No. RE49,461	GPU Gems
469.	generation of hardware has increased its power demand. The future may hold more aggressive dynamic power management targeted at individual stages; increasing amounts of custom or power-aware design for power-hungry parts of the GPU; and more sophisticated cooling management for high-end GPUs. Technology trends indicate that the power demand will only continue to rise with future chip generations, so continued work in this area will remain an important challenge. 29.4.3 Challenge: Supporting More Programmability and Functionality While the current generation of graphics hardware features substantially more programmability than previous generations, the general programmability of GPUs is still far from ideal. One step toward addressing this trend is to improve the functionality and flexibility within the two current programmable units (vertex and fragment). It is likely that we will see their instruction sets converge and add functionality, and that their control flow capabilities will become more general as well. We may even see programmable hardware shared between these two stages in an effort to better utilize these resources. GPU architects will have to be mindful, however, that such improvements not affect the GPU's performance on its core tasks. Another option will be expanding programmability to different units. Geometric primitives particularly benefit from programmability, so we may soon see programmable processing on surfaces, triangles, and pixels. As GPU vendors support more general pipelines and more complex and varied shader computation, many researchers have used the GPU to address tasks outside the bounds of the graphics pipeline. This book contains examples of many of these efforts; the general-purpose computation on GPUs (GPGPU) community has successfully addressed problems in visual simulation, image processing, numerical methods, and databases with graphics hardware. We can expect that these efforts will grow in the future as GPUs continue to increase in performance and functionality. Historically, we have seen GPUs subsume functionality previously belonging to the CPU. Early consumer-level graphics hardware could not perform geometry processing on the graphics processor; it was only five years ago that the entire graphics pipeline could be fabricated on a single chip. Though since that time the primary increase in GPU functionality has been directed toward programmability within the graphics pipeline, we should not expect that GPU vendors have halted their efforts to identify more functions to integrate onto the GPU. In particular, today's games often require large amounts of computation in physics and artificial intelligence computations. Such computation may be attractive for future GPUs. 29.4 The Future and Challenges 469

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
	 Figure 31-2. Max Reduction Performed with Multiple Passes 31.4 From Analogies to Implementation 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. 31.4.1 Putting It All Together: A Basic GPGPU Framework 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 <i>pug</i>, incorporates all of the analogies from the previous section in order to abstract GPGPU programming in a manner more accessible to experienced CPU programmers. Initializing and Finalizing a GPGPU Application The first step in a GPGPU application is to initialize the GPU. This is very easy in the framework: just call <code>pugInit()</code>. When the application is finished with the GPU, it can clean up the memory used by the graphics API by calling <code>pugCleanup()</code>. Specifying Kernels 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 <i>The Cg Tutorial</i> (Fernando and Kilgard 2003) and the 31.4 From Analogies to Implementation 503

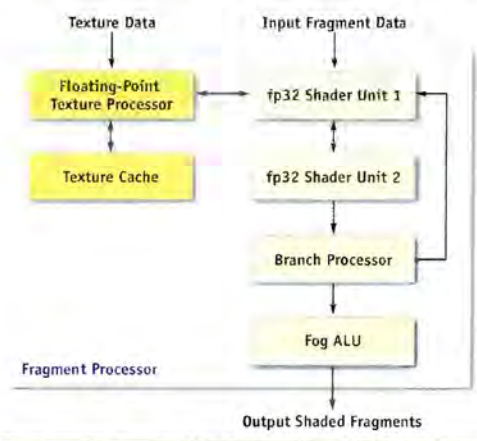
503.

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
	GPU—the GeForce 6 Series—allows vertex programs to fetch texture data. All operations are done in 32-bit floating-point (fp32) precision per component. The GeForce 6 Series architecture supports scalable vertex-processing horsepower, allowing the same architecture to service multiple price/performance points. In other words, high-end models may have six vertex units, while low-end models may have two. Because vertex processors can perform texture accesses, the vertex engines are connected to the texture cache, which is shared with the fragment processors. In addition, there is a vertex cache that stores vertex data both before and after the vertex processor, reducing fetch and computation requirements. This means that if a vertex index occurs twice in a draw call (for example, in a triangle strip), the entire vertex program doesn't have to be rerun for the second instance of the vertex—the cached result is used instead. Vertices are then grouped into primitives, which are points, lines, or triangles. The Cull/Clip/Setup blocks perform per-primitive operations, removing primitives that aren't visible at all, clipping primitives that intersect the view frustum, and performing edge and plane equation setup on the data in preparation for rasterization.  Figure 30-4. The GeForce 6 Series Vertex Processor 30.2 Overall System Architecture 475

475.

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

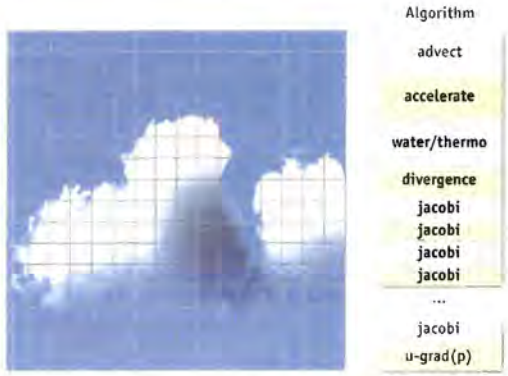
U.S. Patent No. RE49,461	GPU Gems
	The rasterization block calculates which pixels (or samples, if multisampling is enabled) are covered by each primitive, and it uses the z-cull block to quickly discard pixels (or samples) that are occluded by objects with a nearer depth value. Think of a fragment as a "candidate pixel": that is, it will pass through the fragment processor and several tests, and if it gets through all of them, it will end up carrying depth and color information to a pixel on the frame buffer (or render target). Figure 30-5 illustrates the fragment processor (sometimes called a "pixel shader") and texel pipeline. The texture and fragment-processing units operate in concert to apply a shader program to each fragment independently. The GeForce 6 Series architecture supports a scalable amount of fragment-processing horsepower. Another popular way to say this is that GPUs in the GeForce 6 Series can have a varying number of <i>fragment pipelines</i> (or "pixel pipelines"). Similar to the vertex processor, texture data is cached on-chip to reduce bandwidth requirements and improve performance. The texture and fragment-processing unit operates on squares of four pixels (called <i>quads</i>) at a time, allowing for direct computation of derivatives for calculating texture level of detail. Furthermore, the fragment processor works on groups of hundreds of pixels at a time in single-instruction, multiple-data (SIMD) fashion (with each fragment processor engine working on one fragment concurrently), hiding the latency of texture fetch from the computational performance of the fragment processor.  Figure 30-5. The GeForce 6 Series Fragment Processor and Texel Pipeline 476 Chapter 30 The GeForce 6 Series GPU Architecture

476.

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
	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 ’461 patent in a manner that would result in the subject matter of this limitation given, at the very least, the person's understanding of the state of the art, the problems addressed and solved in the prior art, and the teachings of GPU Gems. Specific examples of such references are cited and explained in these invalidity contentions, including the cover pleading and charts C1–C15, and Appendix C.
21[f]: executing the user command during execution of the computations representing the artificial neural network at times determined by the computational stream.	GPU Gems discloses, expressly and/or inherently, executing the user command during execution of the computations representing the artificial neural network at times determined by the computational stream. <i>See e.g.:</i> <i>As a non-limiting example, GPU Gems discloses executing a user command during execution of computations at times based determined by the computational stream (e.g., whether computations in the computational stream are being executed by the GPU).</i>

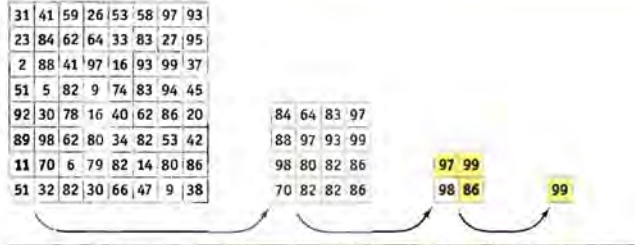
Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
	 Figure 31-1. Simulation on a Grid <i>This cloud simulation executes on the GPU. The data streams used in the simulation are represented on the grid (shown with exaggerated coarseness) and stored in textures. The algorithm is shown at right; each step in the algorithm is a kernel, implemented using a fragment program on the GPU.</i> 31.1.3 Stream Communication: Gather vs. Scatter 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 two main types of communication: <i>gather</i> and <i>scatter</i>. 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. 496 Chapter 31 Mapping Computational Concepts to GPUs 496.

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
497.	31.2 An Inventory of GPU Computational Resources 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. 31.2.1 Programmable Parallel Processors GPUs have two types of programmable processors: <i>vertex processors</i> and <i>fragment processors</i>. 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 <i>vertex program</i> (sometimes called a <i>vertex shader</i>) 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 <i>fragments</i> 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 <i>fragment program</i> (sometimes called a <i>pixel shader</i>) to each fragment in the stream to compute the final color of each pixel. Vertex Processors 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. Because of this, vertex processors have hardware to process four-component vectors. This allows them to produce transformed vertex positions in fewer cycles. 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, 31.2 An Inventory of GPU Computational Resources 497

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
	 Figure 31-2. Max Reduction Performed with Multiple Passes 31.4 From Analogies to Implementation 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. 31.4.1 Putting It All Together: A Basic GPGPU Framework 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 <i>pug</i>, incorporates all of the analogies from the previous section in order to abstract GPGPU programming in a manner more accessible to experienced CPU programmers. Initializing and Finalizing a GPGPU Application The first step in a GPGPU application is to initialize the GPU. This is very easy in the framework: just call <code>pugInit()</code>. When the application is finished with the GPU, it can clean up the memory used by the graphics API by calling <code>pugCleanup()</code>. Specifying Kernels 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 <i>The Cg Tutorial</i> (Fernando and Kilgard 2003) and the 31.4 From Analogies to Implementation 503

503.

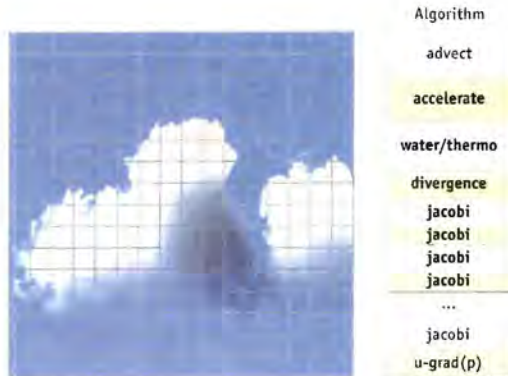
Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
	Chapter 35 GPU Program Optimization <i>Cliff Woolley</i> University of Virginia As GPU programmability has become more pervasive and GPU performance has become almost irresistibly appealing, increasing numbers of programmers have begun to recast applications of all sorts to make use of GPUs. But an interesting trend has appeared along the way: it seems that many programmers make the same performance mistakes in their GPU programs regardless of how much experience they have programming CPUs. The goal of this chapter is to help CPU programmers who are new to GPU programming avoid some of these common mistakes so that they gain the benefits of GPU performance without all the headaches of the GPU programmer's learning curve. 35.1 Data-Parallel Computing One of the biggest hurdles you'll face when first programming a GPU is learning how to get the most out of a data-parallel computing environment. This parallelism exists at several levels on the GPU, as described in Chapter 29 of this book, "Streaming Architectures and Technology Trends." First, parallel execution on multiple data elements is a key design feature of modern GPUs. Vertex and fragment processors operate on four-vectors, performing four-component instructions such as additions, multiplications, multiply-accumulates, or dot products in a single cycle. They can schedule more than one of these instructions per cycle per pipeline. This provides ample opportunities for the extraction of instruction-level parallelism within a GPU program. For example, a 35.1 Data-Parallel Computing 557 557. 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

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
	NVIDIA's Invalidity Contentions and any supplements thereto and the relevant section of charts for other prior art for the '461 patent in a manner that would result in the subject matter of this limitation given, at the very least, the person's understanding of the state of the art, the problems addressed and solved in the prior art, and the teachings of GPU Gems. Specific examples of such references are cited and explained in these invalidity contentions, including the cover pleading and charts C1–C15, and Appendix C.
22. The method of claim 21, wherein the user interaction stream controls the data exchange between the user interaction stream and the computational stream outside of execution of the computations representing the artificial neural network.	GPU Gems discloses, expressly and/or inherently, the method of claim 21, wherein the user interaction stream controls the data exchange between the user interaction stream and the computational stream outside of execution of the computations representing the artificial neural network. <i>See e.g.:</i> <i>See claim 21.</i> <i>As a non-limiting example, GPU Gems discloses that the user interaction stream controls the data exchange between the user interaction stream and the computational stream when computations are not being executed (e.g., by allowing user input and commands to set or edit various parameters of the computations).</i>

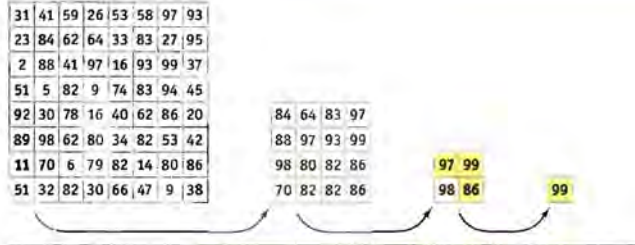
Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
	 Figure 31-1. Simulation on a Grid This cloud simulation executes on the GPU. The data streams used in the simulation are represented on the grid (shown with exaggerated coarseness) and stored in textures. The algorithm is shown at right; each step in the algorithm is a kernel, implemented using a fragment program on the GPU. 31.1.3 Stream Communication: Gather vs. Scatter 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 two main types of communication: <i>gather</i> and <i>scatter</i>. 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. 496 Chapter 31 Mapping Computational Concepts to GPUs 496.

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
497.	31.2 An Inventory of GPU Computational Resources 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. 31.2.1 Programmable Parallel Processors GPUs have two types of programmable processors: <i>vertex processors</i> and <i>fragment processors</i>. 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 <i>vertex program</i> (sometimes called a <i>vertex shader</i>) 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 <i>fragments</i> 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 <i>fragment program</i> (sometimes called a <i>pixel shader</i>) to each fragment in the stream to compute the final color of each pixel. Vertex Processors 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. Because of this, vertex processors have hardware to process four-component vectors. This allows them to produce transformed vertex positions in fewer cycles. 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, 31.2 An Inventory of GPU Computational Resources 497

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
	 Figure 31-2. Max Reduction Performed with Multiple Passes 31.4 From Analogies to Implementation 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. 31.4.1 Putting It All Together: A Basic GPGPU Framework 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 <i>pug</i>, incorporates all of the analogies from the previous section in order to abstract GPGPU programming in a manner more accessible to experienced CPU programmers. Initializing and Finalizing a GPGPU Application The first step in a GPGPU application is to initialize the GPU. This is very easy in the framework: just call <code>pugInit()</code>. When the application is finished with the GPU, it can clean up the memory used by the graphics API by calling <code>pugCleanup()</code>. Specifying Kernels 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 <i>The Cg Tutorial</i> (Fernando and Kilgard 2003) and the 31.4 From Analogies to Implementation 503
503.	

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
	documentation included with the Cg distribution (NVIDIA 2004). There are very few graphics-specific keywords in Cg. We provide a Cg file, <code>pug.cg</code>, that you can include in your kernel files. This file defines a structure called <code>Stream</code> that abstracts the most graphics-centric concept: texture fetching. <code>Stream</code> is a wrapper for the texture sampler Cg type. Calling one of the <code>value()</code> functions of the <code>Stream</code> structure gives the value of a stream element. 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. Stream Management Arrays of data in the framework are called <i>buffers</i>. 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 <code>mode</code> 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. To load initial data into a buffer, pass the data to the <code>pugInitBuffer()</code> function. 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. 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. 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. 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 <code>range</code> parameter of the <code>pugRunProgram()</code> function. 504 Chapter 31 Mapping Computational Concepts to GPUs 504.

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
	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. 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 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. 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>. 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. 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>. 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. 31.5 A Simple Example 505 505.

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
	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. 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. 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. 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. 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 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. Condition codes (predication) are used in older architectures to emulate true branching. If-then statements compiled to these architectures must evaluate both taken and not 548 Chapter 34 GPU Flow-Control Idioms 548.

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
649.	Chapter 40 Computer Vision on the GPU <i>James Fung</i> <i>University of Toronto</i> Computer vision tasks are computationally intensive and repetitive, and they often exceed the real-time capabilities of the CPU, leaving little time for higher-level tasks. However, many computer vision operations map efficiently onto the modern GPU, whose programmability allows a wide variety of computer vision algorithms to be implemented. This chapter presents methods of efficiently mapping common mathematical operations of computer vision onto modern computer graphics architecture. 40.1 Introduction 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. 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. 40.1 Introduction 649

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
	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. 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. 40.2 Implementation Framework Many computer vision operations can be considered sequences of filtering operations, with each sequential filtering stage acting upon the output of the previous filtering stage. On the GPU, these filtering operations are carried out by fragment programs. To apply these fragment programs to input images, the input images are initialized as textures and then mapped onto quadrilaterals. By displaying these quadrilaterals in appropriately sized windows, we can ensure that there is a one-to-one correspondence of image pixels to output fragments. When the textured quadrilateral is displayed, the fragment program then runs, operating identically on each pixel of the image. The fragment program is analogous to the body of a common for-loop program statement, but each iteration of the body can be thought of as executing in parallel. The resulting output is not the original image, but rather the filtered image. Modern GPUs allow input and results in full 32-bit IEEE floating-point precision, providing sufficient accuracy for many algorithms. A complete computer vision algorithm can be created by implementing sequences of these filtering operations. After the texture has been filtered by the fragment program, the resulting image is placed into texture memory, either by using render-to-texture extensions or by copying the frame buffer into texture memory. In this way, the output image becomes the input texture to the next fragment program. This creates a pipeline that runs the entire computer vision algorithm. However, often a complete vision algorithm will require operations beyond filtering. For example, summations are common operations; we present examples of how they can be implemented and used. Furthermore, more-generalized calculations, such as feature tracking, can also be mapped effectively onto graphics hardware. 650 Chapter 40 Computer Vision on the GPU 650.

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
651.	These methods are implemented in the OpenVIDIA GPU computer vision library, which can be used to create real-time computer vision programs (http://openvidia.sourceforge.net/). 40.3 Application Examples This chapter presents examples that implement common computer vision techniques on the GPU. The common thread between each of these algorithms is that they are all used in our system to create a fully interactive, real-time <i>computer-mediated reality</i>, in which the environment of a user appears to be augmented, diminished, or otherwise altered. Some examples are shown in Figure 40-1.  Figure 40-1. Using Computer Vision to Mediate Reality <i>Computer vision techniques can analyze video to augment, diminish, or otherwise alter a user's perception of the environment. On the left, detected ceiling planes become terminal windows. On the right, a terminal window is mapped to the plane of the floppy disk.</i> 40.3.1 Using Sequences of Fragment Programs for Computer Vision In this section, we demonstrate how a sequence of fragment programs can work together to perform more-complex computer vision operations. Correcting Radial Distortion Many camera lenses cause some sort of radial distortion (also known as <i>barrel distortion</i>) of the image, most commonly in wide-field-of-view or low-cost lenses. There are several ways of correcting for radial distortion on the graphics hardware; we present an example of how a fragment program can be used to do so. 40.3 Application Examples 651

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
	We begin by assuming that the displacement of a pixel due to radial distortion is Δx, Δy and is commonly modeled by: $\Delta x = x(k_1 r^2 + k_2 r^4 + \dots),$ $\Delta y = y(k_1 r^2 + k_2 r^4 + \dots),$ where x, y are the image coordinates, r is the distance of the pixel from the principal point of the camera, and the coefficients $k_{1,2,3, \dots}$ are camera parameters determined by some known calibration method. Given a particular camera, the required calibration parameters can be found using the tools provided at Bouquet 2004. The fragment program shown in Listing 40-1 applies this correction by using the current texture coordinates to calculate an offset. The program then adds this offset to the current pixel's coordinates and uses the result to look up the texture value at the corrected coordinate. This texture value is then output, and the image's radial distortion is thereby corrected. Listing 40-1. Radial Undistortion <pre> float4 CorrectBarrelDistortion(float2 iptexCoord : TEXCOORD0, float4 wpos : WPOS, uniform samplerRECT FPEL) : COLOR { // known constants for the particular lens and sensor float f = 368.26488; // focal length float ox = 147.54834; // principal point, x axis float oy = 126.01673; // principal point, y axis float k1 = 0.4162; // constant for radial distortion correction float k2 = 0.40348; float2 xy = (iptexCoord - float2(ox, oy))/float2(f,f); r = sqrt(dot(xy, xy)); float r2 = r * r; float r4 = r2 * r2; float coeff = (k1 * r2 + k2 * r4); </pre> 652 Chapter 40 Computer Vision on the GPU

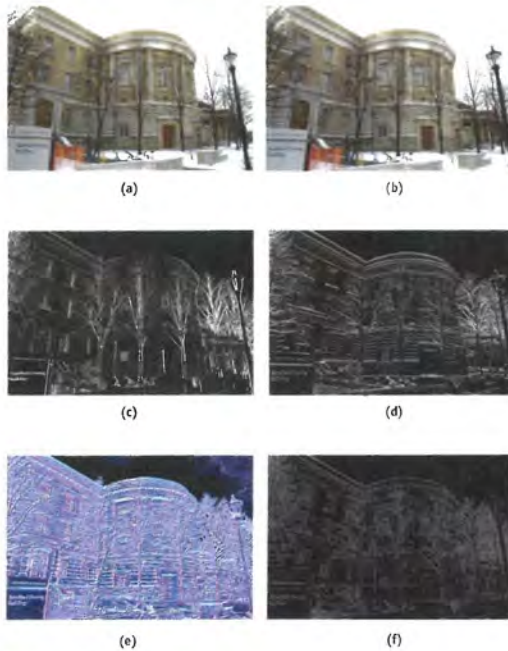
652.

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
	Listing 40-1 (continued). Radial Undistortion <pre> // add the calculated offsets to the current texture coordinates xy = (xy + xy * coeff.xx) * f.xx + float2(ox, oy); // look up the texture at the corrected coordinates // and output the color return texRECT(FPEL, xy); } </pre> A Canny Edge Detector Edge detection is a key algorithm used in many vision applications. Here we present an implementation of the commonly used "Canny" edge-detection algorithm that runs entirely on the GPU. See Figure 40-2 for an example. We implement the Canny edge detector as a series of fragment programs, each performing a step of the algorithm: • Step 1. Filter the image using a separable Gaussian edge filter (Jargstorff 2004). The filter footprint is typically around 15×15 elements wide, and separable. A separable filter is one whose 2D mask can be calculated by applying two 1D filters in the x and y directions. The separability saves us a large number of texture lookups, at the cost of an additional pass. • Step 2. Determine the magnitude, l, of the derivatives at each pixel computed in step 1, and quantize the direction d. The values of d and l are given by: $d = \arctan 2(d_y, d_x),$ $l = \sqrt{d_x^2 + d_y^2},$ where d is the direction of the gradient vector, quantized to one of the eight possible directions one may traverse a 3×3 region of pixels. The length l is the magnitude of the vector. • Step 3. Perform a nonmaximal suppression, as shown in Listing 40-2. The nonmaximal suppression uses the direction of the local gradient to determine which pixels are in the forward and backward direction of the gradient. This direction is used to calculate the x and y offset of the texture coordinates to look up the forward and backward neighbors. An edge is considered found only if the gradient is at a maximum as compared to its neighbors, ensuring a thin line along the detected edge. 40.3 Application Examples 653

653.

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
	 (a) (b) (c) (d) (e) (f) Figure 40-2. Canny Edge Detection Each image shows the result of a fragment program operating on the previous image. (a) Original input. (b) Radial undistorted image. (c) Horizontal derivative. (d) Vertical derivative. (e) Gradient quantization. (f) Canny edges. 654 Chapter 40 Computer Vision on the GPU 654.

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
655.	Listing 40-2. A Canny Edge Detector Using Nonmaximal Suppression <i>This program looks up the direction of the gradient at the current pixel and then retrieves the pixels in the forward and backward directions along the gradient. The program returns 1 if the current pixel value is greater than the two values along the gradient; otherwise, it suppresses (or zeroes) the output. A thresholding value allows the sensitivity of the edge detector to be varied in real time to suit the vision apparatus and environment.</i> <pre> float4 CannySearch(uniform float4 thresh, float2 fptexCoord : TEXCOORD0, uniform samplerRECT FPPE1 : COLOR) { // magdir holds { dx, dy, mag, dirct } float4 magdir = texRECT(FPPE1, fptexCoord); float alpha = 0.5/sin(3.14159/8); // eight directions on grid float2 offset = round(alpha.xx * magdir.xy/magdir.zx); float4 fwdneighbour, backneighbour; fwdneighbour = texRECT(FPPE1, fptexCoord + offset); backneighbour = texRECT(FPPE1, fptexCoord - offset); float4 color0; if (fwdneighbour.z > magdir.z && backneighbour.z > magdir.z) color0 = float4(0.0, 0.0, 0.0, 0.0); // not an edge! else color0 = float4(1.0, 1.0, 1.0, 1.0); // is an edge! if (magdir.z < thresh.x) color0 = float4(0.0, 0.0, 0.0, 0.0); // thresholding return color0; } </pre> 40.3.2 Summation Operations Many common vision operations involve the summation of an image buffer. Some methods that require the summation are moment calculations and the solutions to linear systems of equations. Buck and Purcell (2004) discuss a general method of efficiently summing buffers by using local neighborhoods. On systems that do not support render-to-texture, however, the number of passes required by this technique can be limiting, due to the number of graphics context switches involved. We have found performing the computation in two 40.3 Application Examples 655

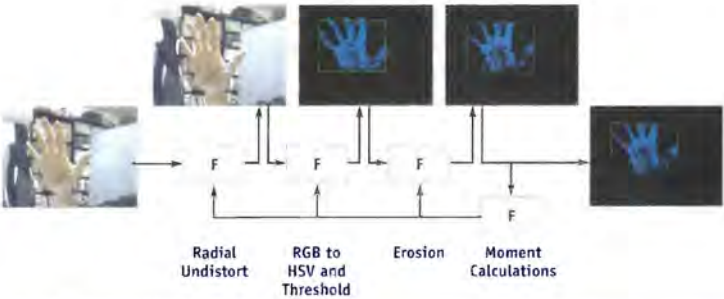
Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
	passes to be more efficient on such systems. In our approach, we create a fragment program that sums each row of pixels, resulting in a single column holding the sums of each row. Then, in a second pass, we draw a single pixel and execute a program that sums down the column. The result is a single pixel that has as its "color" the sum of the entire buffer. The Cg code for computing the sum in this way is given in this chapter's appendix on the book's CD. We use the summation operation within the context of two different, but commonly used, algorithms that require summation over a buffer: a hand-tracking algorithm and an algorithm for creating panoramas from images by determining how to stitch them together. Tracking Hands Tracking a user's hands or face is a useful vision tool for creating interactive systems, and it can be achieved on the GPU by using a combination of image filtering and moment calculations. A common way to track a user's hand is to use color segmentation to find areas that are skin-colored, and then continually track the location of those areas through the image. Additionally, it is useful to track the average color of the skin tone, because it changes from frame to frame, typically due to changes in lighting. It is common to carry out the segmentation in an HSV color space, because skin tone of all hues tends to cluster in HSV color space. The HSV color space describes colors in terms of their <i>hue</i> (the type of color, such as red, green, or blue); <i>saturation</i> (the vibrancy of that color); and <i>value</i> (the brightness of the color). Cameras typically produce an RGB image, so a fragment program can be used to perform a fast color space conversion as a filtering operation. Assuming that we have some sort of initial guess for the HSV color of the skin, we can also use a fragment program to threshold pixels. Given a particular pixel's HSV value, we can determine its difference from the mean HSV skin color and either output to 0 if the difference is too great or output the HSV value otherwise. For instance, the comparison might be easily done as: <pre> if (distance(hsv, meanhsv) < thresh) { color0 = hsv; } else color0 = float4(0.0, 0.0, 0.0, 0.0); </pre> 656 Chapter 40 Computer Vision on the GPU 656.

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
	Finally, we determine the centroid of the skin-colored region and update the mean skin value based on what is currently being seen. For both of these steps, we need to calculate sums over the image. To compute centroids, we can first compute the first three moments of the image: $M_{00} = \sum_{x=0}^{W-1} \sum_{y=0}^{H-1} E(x, y),$ $M_{10} = \sum_{x=0}^{W-1} \sum_{y=0}^{H-1} xE(x, y),$ $M_{01} = \sum_{x=0}^{W-1} \sum_{y=0}^{H-1} yE(x, y),$ where W and H are the sizes of the image in each axis, and $E(x, y)$ is the pixel value at location (x, y). The x coordinate of the centroid is then computed as M_{10}/M_{00}, and M_{01}/M_{00} is the centroid's y coordinate. These summations can be done by first running a fragment program that fills a buffer with: $E(x, y), \quad xE(x, y), \quad yE(x, y)$ and then using a GPU summation method to sum the entire buffer. The result will be a single pixel whose value can then be read back to the CPU to update the location of the centroid. In our case—tracking the centroid of a thresholded skin-tone blob—we might simply let the fragment program output 1.0 in skin-colored areas and 0 elsewhere. Listing 40-3 shows a Cg program that produces four output components that, after summation, will provide the sums needed to calculate the zeroth and first-order moments. Figure 40-3 depicts this process. Should we require more statistics like the preceding, we could either reduce the precision and output eight 16-bit floating-point results to a 128-bit floating-point texture or use the multiple render targets (MRTs) OpenGL extension. 40.3 Application Examples 657 657.

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
	Listing 40-3. Creating Output Statistics for Moment Calculations <pre> float4 ComputeMoments(float2 texCoord : TEXCOORD0, uniform samplerRECT FPE1: TEXUNIT0) : COLOR { float4 samp = texRECT(FPE1, fptexCoord); float4 colorO = float4(0, 0, 0, 0); if (samp.x != 0.0) colorO = float4(1.0, fptexCoord.x, fptexCoord.y, samp.y); } </pre>  Figure 40-3. Tracking a User's Hand 40.3.3 Systems of Equations for Creating Image Panoramas We also encounter this compute-and-sum type of calculation when we form systems of equations of the familiar form $Ax = b$. VideoOrbits VideoOrbits is an image-registration algorithm that calculates a coordinate transformation between pairs of images taken with a camera that is free to pan, tilt, rotate about its optical axis, and zoom. The technique solves the problem for two cases: (1) images taken from the same location of an arbitrary 3D scene or (2) images taken from arbitrary locations of a flat scene. Figure 40-4 shows an image panorama composed from several frames of video. Equivalently, Mann and Fung (2002) show how the same algorithm can be used to replace planes in the scene with computer-generated content, such as in Figure 40-1. 658 Chapter 40 Computer Vision on the GPU

658.

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
	Figure 40-4. A VideoOrbits Panorama <i>This panorama shows how the images have been projected to fit together properly.</i> VideoOrbits solves for a projection of an image in order to register it with another image. Consider a pair of images of the same scene, with the camera having moved between images. A given point in the scene appearing at coordinates $[x, y]$ in the first image now appears at coordinates $[x', y']$ in the second image. Under the constraints mentioned earlier, all the points in the image can be considered to move according to eight parameters, given by the following equation: $\begin{bmatrix} x' \\ y' \end{bmatrix} = \frac{\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \end{bmatrix}}{\begin{bmatrix} c_1 & c_2 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + 1} \quad (1)$ The eight scalar parameters describing the projection are denoted by: $\mathbf{p} = [a_{11}, a_{12}, b_1, a_{21}, a_{22}, b_2, c_1, c_2]$ and are calculated by VideoOrbits. Once the parameters $\mathbf{p}$ have been determined for each image, the images can then be transformed into the same coordinate system to 40.3 Application Examples 659 659.

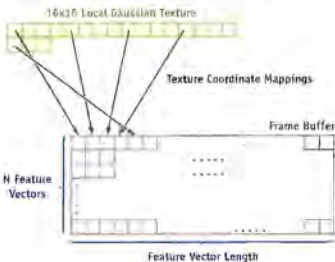
Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
660.	create the image panoramas of Figure 40-4. (For a more in-depth treatment of projective flow and the VideoOrbits algorithm, see Mann 1998.) VideoOrbits calculates the parameters p by solving a linear system of equations. The bulk of the computation is in the initialization of this system of equations, which is of the form $Ax = b$. Here, A is an 8×8 matrix whose values we must initialize, x is an 8×1 column vector that corresponds to p, and b is an 8×1 column vector we must also initialize. Solving this system is computationally expensive because each element of A and b is actually a summation of a series of multiplications and additions at each point in the image. Initializing each of the 64 entries of the matrix A involves many multiplications and additions over all the pixels. A is of the form: $A = \begin{bmatrix} \sum a_{11} & \sum a_{12} & \sum a_{13} & \dots & \sum a_{18} \\ \sum a_{21} & \sum a_{22} & \sum a_{23} & & \sum a_{28} \\ \vdots & & & & \vdots \\ \sum a_{81} & \sum a_{82} & \sum a_{83} & & \sum a_{88} \end{bmatrix}_{8 \times 8} \quad (2)$ where each of the elements is a summation. (Note that the elements a_{11}, a_{13}, a_{21}, a_{23} are not those of Equation 1.) The arguments of these summations involve many arithmetic operations. For instance, element a_{21} is: $a_{21} = (x^2 E_x + xy E_y)(xy E_x + y^2 E_y). \quad (3)$ This operation must be carried out at every pixel and then summed to obtain each entry in matrix A. This compute-and-sum operation is similar to moment calculations discussed previously. Not surprisingly, we can calculate this on the GPU in a similar fashion, filling a buffer with the argument of the summation and then summing the buffer. Typically, to calculate all the matrix entries, multiple passes will be required. As each summation is completed, the results are copied to a texture that stores the results. After all the entries for the matrix A and vector b are computed, the values are read back to the CPU. This keeps the operations on the GPU, deferring the readback until all matrix entries have been calculated. This allows the GPU to operate with greater independence from the CPU. The system can then be solved on the CPU, with the majority of the per-pixel calculations already solved by the GPU. 660 Chapter 40 Computer Vision on the GPU

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
661.	40.3.4 Feature Vector Computations Calculating <i>feature vectors</i> at detected feature points is a common operation in computer vision. A <i>feature</i> in an image is a local area around a point with some higher-than-average amount of "uniqueness." This makes the point easier to recognize in subsequent frames of video. The uniqueness of the point is characterized by computing a feature vector for each feature point. Feature vectors can be used to recognize the same point in different images and can be extended to more generalized object recognition techniques. Figure 40-5 shows the motion of the features between frames as the camera moves.  Figure 40-5. Detecting Distinctive Points as the Camera Pans <i>The GPU is used to detect and track distinctive points (features) in real time as the camera pans. The green lines connect the same point in the previous image to its new location in the current image.</i> Computing feature vectors on the GPU requires handling the image data in a different manner than the operations we have thus far presented. Because only a few points in the image are feature points, this calculation does not need to be done at every point in the image. Additionally, for a single feature point, a large feature vector (100 elements or more) is typically calculated. Thus, each single point produces a larger number of results, rather than the one-to-one or many-to-one mappings we have discussed so far. These types of computations can still be carried out on the GPU. We take as an example a feature vector that is composed of the filtered gradient magnitudes in a local region around a located feature point. Feature detection can be achieved using methods similar to the Canny edge detector that instead search for corners rather than lines. If the feature points are being detected using sequences of filtering, as is common, we can perform the filtering on the GPU 40.3 Application Examples 661

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
	and read back to the CPU a buffer that flags which pixels are feature points. The CPU can then quickly scan the buffer to locate each of the feature points, creating a list of image locations at which we will calculate feature vectors on the GPU. There are many approaches to generating feature vectors in order to track feature points. We show one approach as an example. For each detected feature point, we examine a 16×16 neighborhood of gradient magnitudes and directions (as computed in the Canny edge example). This neighborhood is divided up into 4×4-pixel regions, giving 16 different regions of 16 pixels. In each region, we create an 8-element histogram of gradient magnitudes. Each pixel's gradient magnitude is added to the bin of the histogram corresponding to its quantized gradient direction. For each region, the result is then an 8-element histogram. The feature vector is the concatenation of all 16 of these 8-element histograms, giving a feature vector length of $16 \times 8 = 128$ elements. Features in different images can be matched to each other by computing the Euclidean distance between pairs of feature vectors, with the minimum distance providing a best match. To produce the 128-element feature vector result, we draw rows of vertices, with one row for each feature, such that each drawn point in the row corresponds to an element of the feature vector for that feature. Associating texture values pointwise allows for the most flexibility in mapping texture coordinates for use in computing the feature vector. Figure 40-6 shows how the vertices are laid out, and the binding of the image texture coordinates. Each point has different texture coordinates mapped to it to allow a single fragment program to access the appropriate gradients of the local region used to calculate the histogram.  Figure 40-6. OpenGL Vertex Layout and Texture Bindings for Feature Vertices 662 Chapter 40 Computer Vision on the GPU

662.

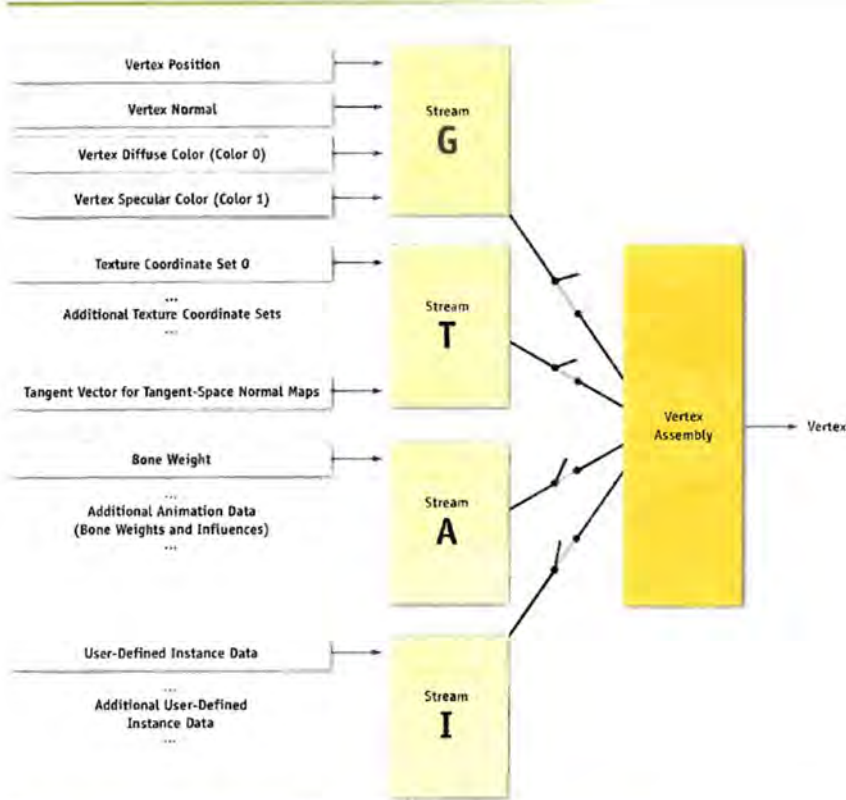
Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
	By associating the correct texture coordinates for each point vertex, the single fragment program shown in Listing 40-4 calculates the feature vectors. To produce 8 elements per region, we have packed the results into 16-bit half-precision floating-point values; alternatively, we could use MRTs to output multiple 32-bit values. After completion, we read back the frame buffer, containing one feature vector per row, to the CPU. Listing 40-4. Fragment Program for Calculating Feature Vectors <pre> Float4 FeatureVector() { float2 fptexCoord : TEXCOORD0, float2 tapsCoord : TEXCOORD1, float4 colorZero : COLOR0, float4 wpos : WPOS, uniform samplerRECT FPE1, uniform samplerRECT FPE2 : COLOR { const float4x4 lclGauss = (0.3536, 0.3536, 0.3536, 0.3536, 0.3536, 0.7071, 0.7071, 0.3536, 0.3536, 0.7071, 0.7071, 0.3536, 0.3536, 0.3536, 0.3536, 0.3536); const float eps = 0.001; float4 input; int xoffset = 1, yoffset = 1; float4 bins_0 = float4(0.0, 0.0, 0.0, 0.0); float4 bins_1 = float4(0.0, 0.0, 0.0, 0.0); for (xoffset = 0 ; xoffset < 4 ; xoffset++) { for (yoffset = 0 ; yoffset < 4 ; yoffset++) { input = texRECT(FPE1, fptexCoord + float2(xoffset, yoffset)); float dir = input.y; input *= lclGauss[xoffset,yoffset].xxxx * texRECT(FPE2, tapsCoord + float2(xoffset, yoffset)).xxxx; if (dir <= (1.0 + eps)/8.0) bins_0.x += input.x; /* . . . for each bin . . . */ else if (dir <= (7.0 + eps)/8.0) bins_1.x += input.x; else bins_1.w += input.x; } } return float4(pack_2half(bins_0.xy), pack_2half(bins_0.zw), pack_2half(bins_1.xy), pack_2half(bins_1.zw)) } </pre> 40.3 Application Examples 663 663.

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
	40.4 Parallel Computer Vision Processing When multiple computer vision algorithms are needed for an application, parallel GPU processing can be achieved by placing multiple graphics cards in the PCI bus slots of a motherboard. Figure 40-7 shows a machine built to explore the benefits of parallel computations by using multiple graphics cards. Running each computer vision program on a different GPU allows the cards to process in parallel; conventional shared memory can be used to allow the programs to interact. Although creating multi-GPU systems using PCI cards will soon no longer be possible due to the obsolescence of the PCI bus, the advent of PCI Express has made new multi-GPU approaches possible. One such approach is NVIDIA's SLI, which makes the presence of multiple GPUs transparent to the developer by automatically distributing work between them.  Figure 40-7. Multiple GPUs Used for a Computer Vision Application <i>Left: A computer vision application (Fung and Mann 2002) using graphics hardware to display six video streams in a hexagonal form. Right: A computer vision machine that runs computer vision tasks in parallel using six PCI graphics cards and one AGP graphics card. Each PCI card is a GeForce FX 5200 GPU, resulting in an inexpensive, powerful, and easily constructible parallel architecture well suited for pattern recognition and computer vision.</i> 40.5 Conclusion This chapter has presented techniques for mapping computer vision algorithms onto modern GPUs. We hope that this chapter has provided an understanding of how to write computer vision algorithms for fast processing on the GPU. Although other 664 Chapter 40 Computer Vision on the GPU 664.

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

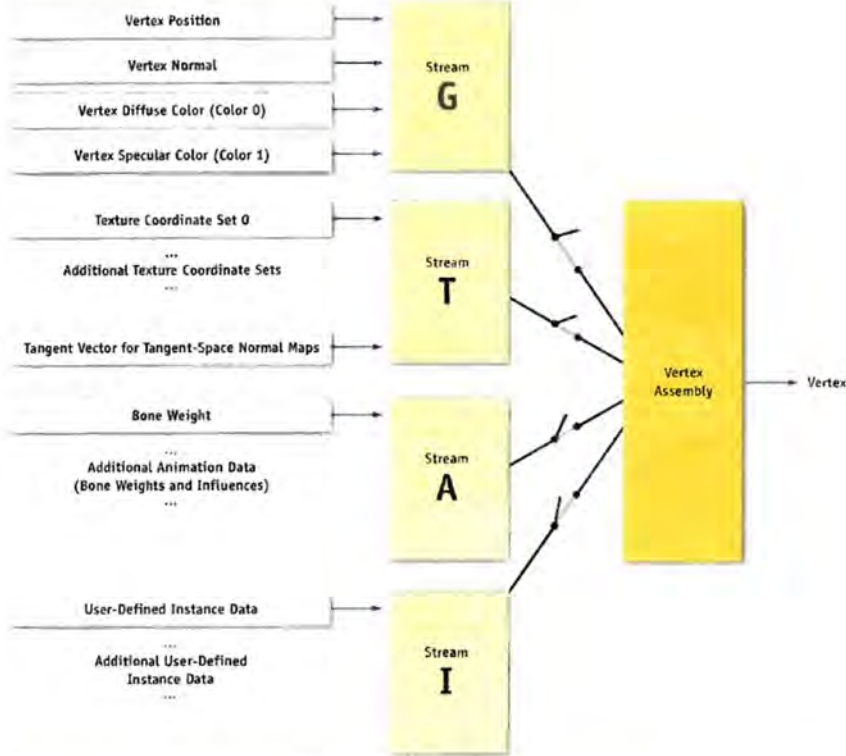
U.S. Patent No. RE49,461	GPU Gems
	 Figure 5-2. Combining Currently Needed Streams 5.2.1 Multistreaming with DirectX 9.0 Microsoft DirectX 8.0 introduced the notion of a stream to bind data to input registers. A stream is a uniform array of component data. Each component consists of one or more elements that represent a single entity, such as position, normal, color, texture coordinate set, and so on. With streams, GPUs are able to perform a direct memory access (DMA) from multiple vertex buffers in parallel. The DirectX9 caps bit

78.

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

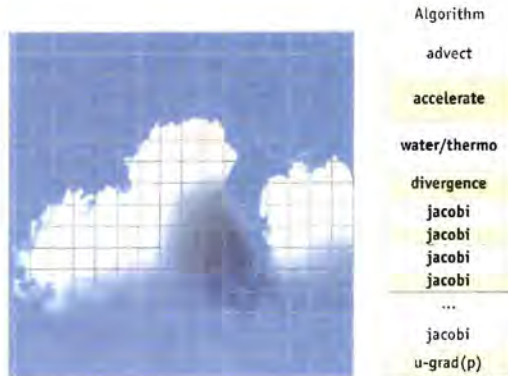
U.S. Patent No. RE49,461	GPU Gems
	To the extent Plaintiff contends that this claim element is not expressly or inherently disclosed, or is not obvious in view of GPU Gems and the knowledge of a POSITA, it would have been obvious to combine the teachings of GPU Gems with any of the prior art references in NVIDIA’s Invalidity Contentions and any supplements thereto and the relevant section of charts for other prior art for the ’461 patent in a manner that would result in the subject matter of this limitation given, at the very least, the person's understanding of the state of the art, the problems addressed and solved in the prior art, and the teachings of GPU Gems. Specific examples of such references are cited and explained in these invalidity contentions, including the cover pleading and charts C1–C15, and Appendix C.
23. The method of claim 21, wherein executing the user interaction stream comprises: controlling setting and editing of computational elements of the computations representing the artificial neural network.	GPU Gems discloses, expressly and/or inherently, the method of claim 21, wherein executing the user interaction stream comprises controlling setting and editing of computational elements of the computations representing the artificial neural network. <i>See claim 21.</i> <i>See e.g.:</i> <i>As a non-limiting example, GPU Gems discloses user interactions that control the setting and editing of computational elements of the computations.</i>

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
	 Figure 5-2. Combining Currently Needed Streams 5.2.1 Multistreaming with DirectX 9.0 Microsoft DirectX 8.0 introduced the notion of a stream to bind data to input registers. A stream is a uniform array of component data. Each component consists of one or more elements that represent a single entity, such as position, normal, color, texture coordinate set, and so on. With streams, GPUs are able to perform a direct memory access (DMA) from multiple vertex buffers in parallel. The DirectX9 caps bit

78.

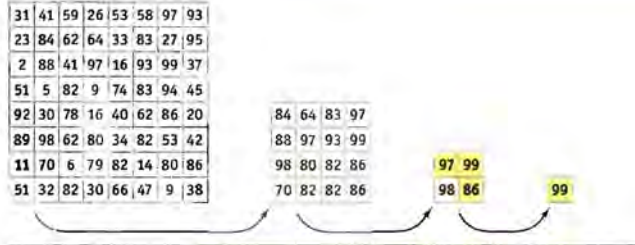
Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
	 Figure 31-1. Simulation on a Grid <i>This cloud simulation executes on the GPU. The data streams used in the simulation are represented on the grid (shown with exaggerated coarseness) and stored in textures. The algorithm is shown at right; each step in the algorithm is a kernel, implemented using a fragment program on the GPU.</i> 31.1.3 Stream Communication: Gather vs. Scatter 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 two main types of communication: <i>gather</i> and <i>scatter</i>. 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. 496 Chapter 31 Mapping Computational Concepts to GPUs 496.

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
497.	31.2 An Inventory of GPU Computational Resources 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. 31.2.1 Programmable Parallel Processors GPUs have two types of programmable processors: <i>vertex processors</i> and <i>fragment processors</i>. 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 <i>vertex program</i> (sometimes called a <i>vertex shader</i>) 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 <i>fragments</i> 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 <i>fragment program</i> (sometimes called a <i>pixel shader</i>) to each fragment in the stream to compute the final color of each pixel. Vertex Processors 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. Because of this, vertex processors have hardware to process four-component vectors. This allows them to produce transformed vertex positions in fewer cycles. 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, 31.2 An Inventory of GPU Computational Resources 497

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
	 Figure 31-2. Max Reduction Performed with Multiple Passes 31.4 From Analogies to Implementation 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. 31.4.1 Putting It All Together: A Basic GPGPU Framework 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 <i>pug</i>, incorporates all of the analogies from the previous section in order to abstract GPGPU programming in a manner more accessible to experienced CPU programmers. Initializing and Finalizing a GPGPU Application The first step in a GPGPU application is to initialize the GPU. This is very easy in the framework: just call <code>pugInit()</code>. When the application is finished with the GPU, it can clean up the memory used by the graphics API by calling <code>pugCleanup()</code>. Specifying Kernels 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 <i>The Cg Tutorial</i> (Fernando and Kilgard 2003) and the 31.4 From Analogies to Implementation 503

503.

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
	documentation included with the Cg distribution (NVIDIA 2004). There are very few graphics-specific keywords in Cg. We provide a Cg file, <code>pug.cg</code>, that you can include in your kernel files. This file defines a structure called <code>Stream</code> that abstracts the most graphics-centric concept: texture fetching. <code>Stream</code> is a wrapper for the texture sampler Cg type. Calling one of the <code>value()</code> functions of the <code>Stream</code> structure gives the value of a stream element. 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. Stream Management Arrays of data in the framework are called <i>buffers</i>. 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 <code>mode</code> 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. To load initial data into a buffer, pass the data to the <code>pugInitBuffer()</code> function. 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. 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. 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. 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 <code>range</code> parameter of the <code>pugRunProgram()</code> function. 504 Chapter 31 Mapping Computational Concepts to GPUs 504.

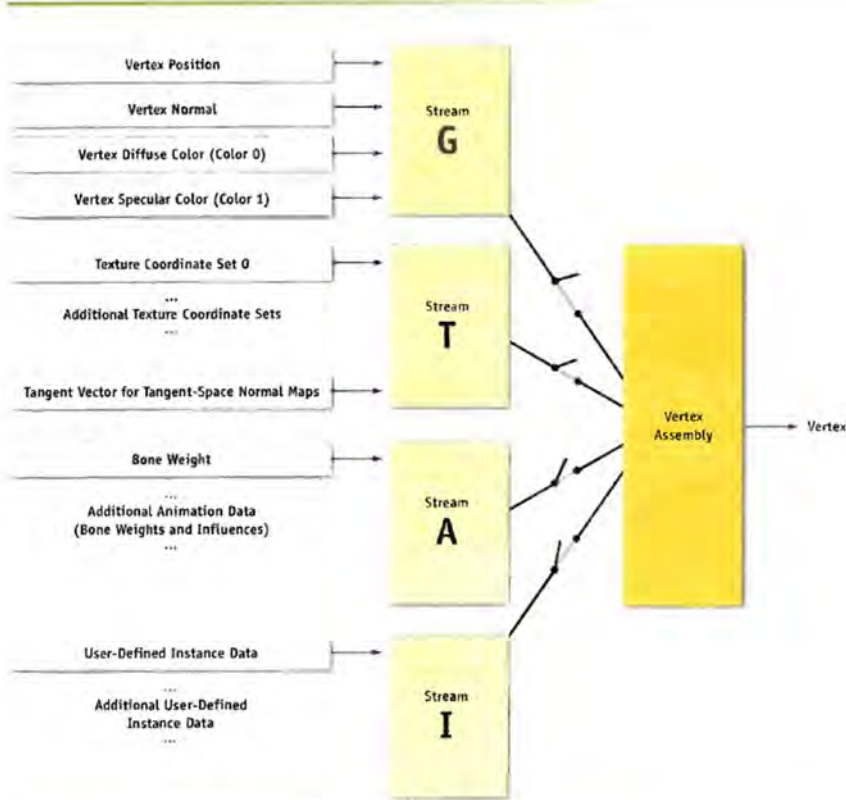
Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
	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. 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 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. 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>. 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. 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>. 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. 31.5 A Simple Example 505 505.

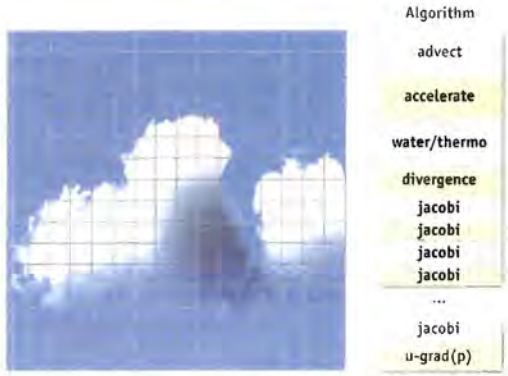
Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
	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 ’461 patent in a manner that would result in the subject matter of this limitation given, at the very least, the person's understanding of the state of the art, the problems addressed and solved in the prior art, and the teachings of GPU Gems. Specific examples of such references are cited and explained in these invalidity contentions, including the cover pleading and charts C1–C15, and Appendix C.
24. The method of claim 21, wherein executing the user interaction stream comprises: controlling setting and editing of parameters of the computations representing the artificial neural network.	GPU Gems discloses, expressly and/or inherently, the method of claim 21, wherein executing the user interaction stream comprises controlling setting and editing of parameters of the computations representing the artificial neural network. <i>See claims 21, 23.</i> <i>See e.g.:</i> <i>As a non-limiting example, GPU Gems discloses user interactions that control the setting and editing of parameters of the computations.</i>

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
	 Figure 5-2. Combining Currently Needed Streams 5.2.1 Multistreaming with DirectX 9.0 Microsoft DirectX 8.0 introduced the notion of a stream to bind data to input registers. A stream is a uniform array of component data. Each component consists of one or more elements that represent a single entity, such as position, normal, color, texture coordinate set, and so on. With streams, GPUs are able to perform a direct memory access (DMA) from multiple vertex buffers in parallel. The DirectX9 caps bit 78.

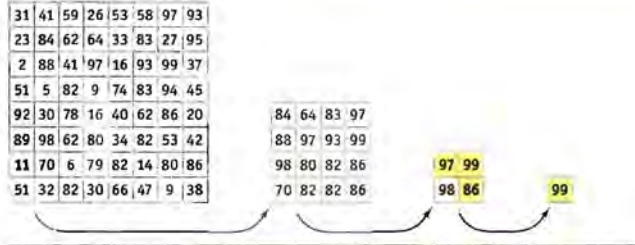
Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
	 Figure 31-1. Simulation on a Grid <i>This cloud simulation executes on the GPU. The data streams used in the simulation are represented on the grid (shown with exaggerated coarseness) and stored in textures. The algorithm is shown at right; each step in the algorithm is a kernel, implemented using a fragment program on the GPU.</i> 31.1.3 Stream Communication: Gather vs. Scatter 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 two main types of communication: <i>gather</i> and <i>scatter</i>. 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. 496 Chapter 31 Mapping Computational Concepts to GPUs 496.

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
497.	31.2 An Inventory of GPU Computational Resources 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. 31.2.1 Programmable Parallel Processors GPUs have two types of programmable processors: <i>vertex processors</i> and <i>fragment processors</i>. 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 <i>vertex program</i> (sometimes called a <i>vertex shader</i>) 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 <i>fragments</i> 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 <i>fragment program</i> (sometimes called a <i>pixel shader</i>) to each fragment in the stream to compute the final color of each pixel. Vertex Processors 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. Because of this, vertex processors have hardware to process four-component vectors. This allows them to produce transformed vertex positions in fewer cycles. 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, 31.2 An Inventory of GPU Computational Resources 497

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
	 Figure 31-2. Max Reduction Performed with Multiple Passes 31.4 From Analogies to Implementation 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. 31.4.1 Putting It All Together: A Basic GPGPU Framework 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 <i>pug</i>, incorporates all of the analogies from the previous section in order to abstract GPGPU programming in a manner more accessible to experienced CPU programmers. Initializing and Finalizing a GPGPU Application The first step in a GPGPU application is to initialize the GPU. This is very easy in the framework: just call <code>pugInit()</code>. When the application is finished with the GPU, it can clean up the memory used by the graphics API by calling <code>pugCleanup()</code>. Specifying Kernels 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 <i>The Cg Tutorial</i> (Fernando and Kilgard 2003) and the 31.4 From Analogies to Implementation 503

503.

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
	documentation included with the Cg distribution (NVIDIA 2004). There are very few graphics-specific keywords in Cg. We provide a Cg file, <code>pug.cg</code>, that you can include in your kernel files. This file defines a structure called <code>Stream</code> that abstracts the most graphics-centric concept: texture fetching. <code>Stream</code> is a wrapper for the texture sampler Cg type. Calling one of the <code>value()</code> functions of the <code>Stream</code> structure gives the value of a stream element. 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. Stream Management Arrays of data in the framework are called <i>buffers</i>. 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 <code>mode</code> 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. To load initial data into a buffer, pass the data to the <code>pugInitBuffer()</code> function. 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. 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. 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. 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 <code>range</code> parameter of the <code>pugRunProgram()</code> function. 504 Chapter 31 Mapping Computational Concepts to GPUs 504.

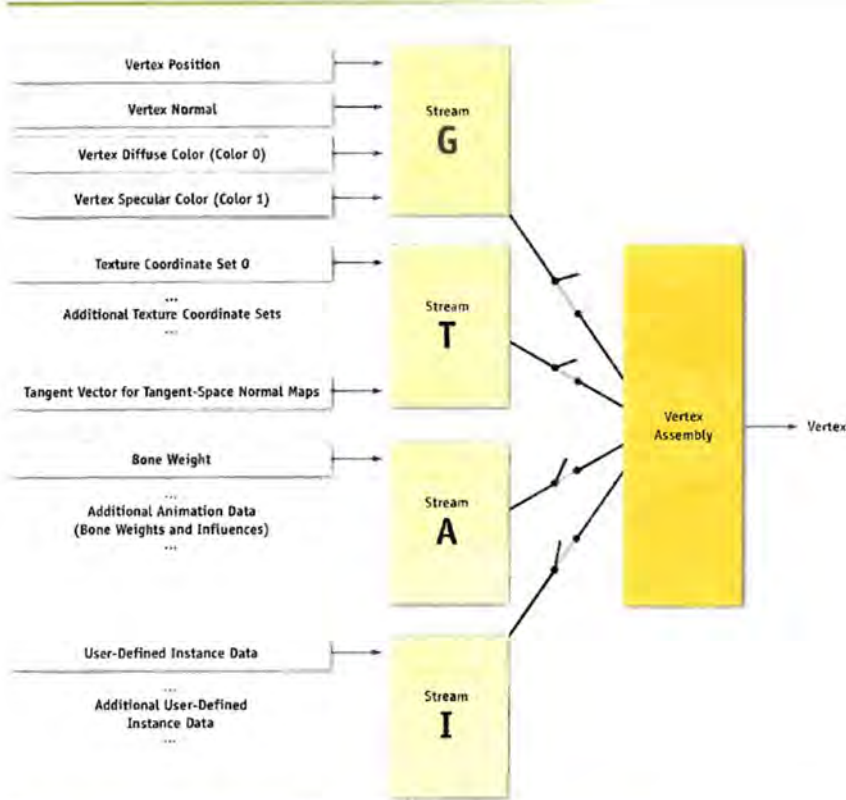
Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
	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. 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 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. 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>. 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. 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>. 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. 31.5 A Simple Example 505 505.

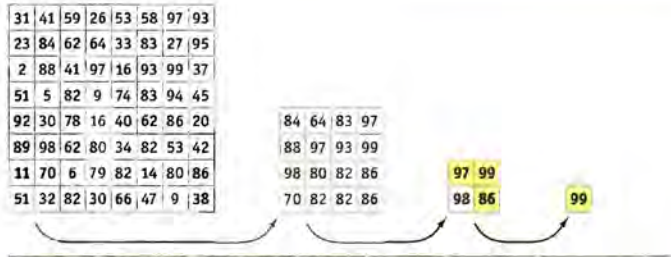
Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
	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 ’461 patent in a manner that would result in the subject matter of this limitation given, at the very least, the person's understanding of the state of the art, the problems addressed and solved in the prior art, and the teachings of GPU Gems. Specific examples of such references are cited and explained in these invalidity contentions, including the cover pleading and charts C1–C15, and Appendix C.
25. The method of claim 21, wherein executing the user interaction stream comprises: controlling setting and editing of parameters of the inputs to the artificial neural network.	GPU Gems discloses, expressly and/or inherently, the method of claim 21, wherein executing the user interaction stream comprises controlling setting and editing of parameters of the inputs to the artificial neural network. <i>See claim 21; Chapter 34.</i> <i>See e.g.:</i> <i>As a non-limiting example, GPU Gems discloses user interactions that control the setting and editing of parameters of the inputs to the computations.</i>

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
	 Figure 5-2. Combining Currently Needed Streams 5.2.1 Multistreaming with DirectX 9.0 Microsoft DirectX 8.0 introduced the notion of a stream to bind data to input registers. A stream is a uniform array of component data. Each component consists of one or more elements that represent a single entity, such as position, normal, color, texture coordinate set, and so on. With streams, GPUs are able to perform a direct memory access (DMA) from multiple vertex buffers in parallel. The DirectX9D caps bit 78.

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
	 Figure 31-2. Max Reduction Performed with Multiple Passes 31.4 From Analogies to Implementation 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. 31.4.1 Putting It All Together: A Basic GPGPU Framework 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 <i>pug</i>, incorporates all of the analogies from the previous section in order to abstract GPGPU programming in a manner more accessible to experienced CPU programmers. Initializing and Finalizing a GPGPU Application The first step in a GPGPU application is to initialize the GPU. This is very easy in the framework: just call <code>pugInit()</code>. When the application is finished with the GPU, it can clean up the memory used by the graphics API by calling <code>pugCleanup()</code>. Specifying Kernels 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 <i>The Cg Tutorial</i> (Fernando and Kilgard 2003) and the 31.4 From Analogies to Implementation 503
503.	

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
	documentation included with the Cg distribution (NVIDIA 2004). There are very few graphics-specific keywords in Cg. We provide a Cg file, <code>pug.cg</code>, that you can include in your kernel files. This file defines a structure called <code>Stream</code> that abstracts the most graphics-centric concept: texture fetching. <code>Stream</code> is a wrapper for the texture sampler Cg type. Calling one of the <code>value()</code> functions of the <code>Stream</code> structure gives the value of a stream element. 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. Stream Management Arrays of data in the framework are called <i>buffers</i>. 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 <code>mode</code> 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. To load initial data into a buffer, pass the data to the <code>pugInitBuffer()</code> function. 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. 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. 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. 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 <code>range</code> parameter of the <code>pugRunProgram()</code> function. 504 Chapter 31 Mapping Computational Concepts to GPUs 504.

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
	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. 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 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. 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>. 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. 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>. 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. 31.5 A Simple Example 505 505.

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
	Chapter 34 GPU Flow-Control Idioms <i>Mark Harris</i> <i>NVIDIA Corporation</i> <i>Ian Buck</i> <i>Stanford University</i> Flow control is one of the most fundamental concepts taught to beginning programmers. Branching and looping are such basic concepts that it can be daunting to write software for a platform that supports them in a limited manner. The latest GPUs support vertex and fragment program branching in multiple forms, but their highly parallel nature requires care in how they are used. This chapter surveys some of the limitations of branching on current GPUs and describes a variety of techniques for iteration and decision making in GPGPU programs. 34.1 Flow-Control Challenges 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. For example, consider the following code: <pre>if (a) b = f(); else b = g();</pre> 34.1 Flow-Control Challenges 547 547.

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
	CPUs can easily branch based on the Boolean <code>a</code> and evaluate either the <code>f()</code> or <code>g()</code> 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 <code>f()</code> and <code>g()</code> have a reasonable number of instructions, these costs aren't too high. 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. 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. 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. 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 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. Condition codes (predication) are used in older architectures to emulate true branching. If-then statements compiled to these architectures must evaluate both taken and not 548 Chapter 34 GPU Flow-Control Idioms 548.

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
	taken branch instructions on all fragments. The branch condition is evaluated and a condition code is set. The instructions in each part of the branch must check the value of the condition code before writing their results to registers. As a result, only instructions in taken branches write their output. Thus, in these architectures all branches cost as much as both parts of the branch, plus the cost of evaluating the branch condition. Branching should be used sparingly on such architectures. NVIDIA GeForce FX Series GPUs use condition-code branch emulation in their fragment processors. 34.2 Basic Flow-Control Strategies 34.2.1 Predication The simplest approach to implementing branching on the GPU is predication, as discussed earlier. With predication, the GPU effectively evaluates both sides of the branch and then discards one of the results, based on the value of the Boolean branch condition. The disadvantage of this approach is that evaluating both sides of the branch can be costly if $f()$ and $g()$ are large functions. While predication may be effective for small branches, alternative strategies are necessary for more complex branching. 34.2.2 Moving Branching up the Pipeline Because explicit branching can be tricky on GPUs, it's handy to have a number of techniques in your repertoire. A useful strategy is to move flow-control decisions up the pipeline to an earlier stage, where they can be more efficiently evaluated. Static Branch Resolution When performing computations on streams or arrays of data on the CPU, most programmers know that they should strive to avoid branching inside the inner loops of the computation. Doing so can cause the pipeline to stall due to incorrect branch prediction. For example, consider evaluating a partial differential equation (PDE) on a discrete spatial grid. Correct evaluation of the PDE on a finite domain requires boundary conditions. A naive CPU implementation might iterate over the entire grid, deciding at each cell if it is a boundary cell and applying the appropriate computation based on the result of the decision. A better implementation divides the processing into multiple loops: one over the interior of the grid, excluding boundary cells, and one or more over the boundary edges. This <i>static branch resolution</i> results in loops that contain efficient code without branches. 34.2 Basic Flow-Control Strategies 549 549.

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
	The same optimization is even more important on most GPUs. In this case, the computation is divided into two fragment programs: one for interior cells and one for boundary cells. The interior program is applied to the fragments of a quad drawn over all but the outer single-cell-wide edge of the output buffer. The boundary program is applied to fragments of lines drawn over the edge pixels. Precomputation In the preceding example, the result of a branch was constant over a large domain of input (or range of output) values. Similarly, sometimes the result of a branch is constant for a period of time or a number of iterations of a computation. In this case, we can evaluate the branches only when the results change, and store the results for use over many subsequent iterations. This can result in a large performance boost. The "gpgpu_fluid" example in the NVIDIA SDK uses this technique to avoid branching when computing boundary conditions at the edges of arbitrary obstacles in the flow field. In this case, fluid cells with no neighboring obstacles can be processed normally, but cells with neighboring obstacles require more work. These cells must check their neighbors to figure out in which direction the obstacle lies, and they use this direction to look up more data to be used in the computation. In the example, the obstacles change only when the user "paints" them. Therefore, we can precompute the offset direction and store it in an offset texture to be reused until the user changes the obstacles again. 34.2.3 Z-Cull We can take precomputed branch results a step further and use another GPU feature to entirely skip unnecessary work. Modern GPUs have a number of features designed to avoid shading pixels that will not be seen. One of these is z-cull. Z-cull is a technique for comparing the depth (z) of incoming fragments with the depth of the corresponding fragments in the z-buffer such that if the incoming fragments fail the depth test, they are discarded before their pixel colors are calculated in the fragment processor. Thus, only fragments that pass the depth test are processed, work is saved, and the application runs faster. We can use this technique for general-purpose computation, too. In the example of obstacles in fluid flow given earlier, there are some cells that are completely "landlocked": all of their neighbors are obstacle cells. We don't need to do <i>any</i> computation on these cells. To skip these cells, we do a bit of setup work whenever the user paints new obstacles. We run a fragment program that examines the neighbors of each fragment. The program draws only fragments that are landlocked and discards all others using the 559 Chapter 34 GPU Flow-Control Idioms 550.

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
	discard keyword. (The discard keyword is available in Cg and GLSL. The HLSL equivalent is <code>cull()</code>). The Cg code for this program is shown in Listing 34-1. The pseudocode that follows in Listing 34-2 demonstrates how z-cull is set up and then used to skip processing of landlocked cells. What happens in the code in these two listings is that the preprocessing pass sets up a "mask" in the z-buffer where landlocked cells have a depth of 0.0 and all other cells have a depth of 1.0. Therefore, when we draw a quad at $z = 0.5$, the landlocked cells will be "blocked" by the values of 0.0 in the z-buffer and will be automatically culled by the GPU. If the obstacles are fairly large, then we will save a lot of work by not processing these cells. Listing 34-1. Cg Code to Set Z-Depth Values for Z-Culling in Subsequent Passes <pre> half obstacles(half2 coords : WPOS, uniform samplerRECT obstacleGrid) : COLOR { // get neighboring boundary values (on or off) half4 bounds; bounds.x = texRECT(obstacleGrid, coords - half2(1, 0)).x; bounds.y = texRECT(obstacleGrid, coords + half2(1, 0)).x; bounds.z = texRECT(obstacleGrid, coords - half2(0, 1)).x; bounds.w = texRECT(obstacleGrid, coords + half2(0, 1)).x; bounds.x = dot(bounds, (1).xxxx); // add them up // discard cells that are not landlocked if (bounds.x < 4) discard; return 0; } </pre> Listing 34-2. Application Pseudocode <i>We set z values in the first pass. In the second pass, we execute a fragment program at pixels where the z test passes.</i> <pre> // Application Code--Preprocess pass ClearZBuffer(1.0); Enable(DEPTH_TEST); DepthFunc(LESS); BindFragmentProgram("obstacles"); DrawQuad(z=0.0); </pre> 34.2 Basic Flow-Control Strategies 551

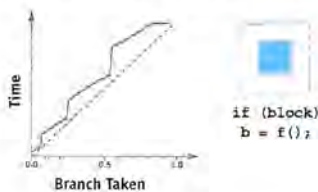
551.

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
	Listing 34-2 (continued). Application Pseudocode <pre>// Application code--Passes in which landlocked cells are to be skipped Enable(DEPTH_TEST); Disable(DEPTH_WRITE); // want to read depth, but not modify it DepthFunc(LESS); // bind normal fragment program for each pass DrawQuad(Z=0.5);</pre> One caveat with this technique is that z-culling is often performed by the GPU at a coarser resolution than a per-fragment basis. The GPU will skip shading operations only if all the fragments in a small contiguous region of the frame buffer fail the depth test. The exact size of this region varies from GPU to GPU, but in general, z-cull will provide a performance improvement only if your branches have some locality. To illustrate this, we can compare the performance of z-cull with a random Boolean condition versus a Boolean condition with high spatial locality. For the random Boolean case, we fill a texture with random values. For the Boolean with high spatial locality, we simply set a rectangular region to a constant Boolean value. As you can see from Figure 34-1, z-cull is most effective if there is plenty of locality in the branch. The good news is that this locality is naturally present if the probability of taking the branch is either very low (that is, if very few fragments pass the depth test) or very high. If locality is not present, z-cull is not much better than predication. Note, however, that z-cull is very different from branching inside a fragment program. Z-cull prevents the fragment program from ever executing. Therefore, z-cull is a powerful method for skipping lots of unnecessary work based on a static, preevaluated condi- Figure 34-1. The Costs of Different Types of Branches When Using Z-Cull <i>The time of executing a program versus the probability of taking a branch with a random or a block Boolean condition. The diagonal line indicates the ideal execution time, assuming the Boolean always prevents $f()$ from executing.</i> 552 Chapter 34 GPU Flow-Control Idioms

552.

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
	tion. To do the same with fragment program branching would require the program to be executed, the condition to be evaluated, and the program to exit early. All of this requires more processor cycles than skipping the fragment program altogether. 34.2.4 Branching Instructions The first GPUs to support fragment branching instructions are the NVIDIA GeForce 6 Series. These instructions are available with both the Microsoft DirectX Pixel Shader 3.0 instruction set and the OpenGL <code>NV_fragment_program2</code> extension. As more GPUs support branch instructions, using predication for branching should no longer be necessary. Note, however, that the locality issues that affect early z-cull also apply to these branch instructions. GPUs execute fragment programs on groups of fragments in parallel, where each group can execute only one branch. Where this is not possible, the GPU will effectively fall back to predication. Figure 34-2 shows the time versus probability test implemented using branch instructions. As you can see, the same spatial locality caveat applies to branching efficiency. However, as with z-cull, if the branch has either a very low or a very high probability, branching instructions are quite effective.  Figure 34-2. The Costs of Different Types of Branches with Pixel Shader 3.0 The time to execute a program using the branch instructions with a high-locality Boolean branch condition. 34.2.5 Choosing a Branching Mechanism Choosing an effective branching mechanism for your program depends primarily on the amount of code inside a branch and the amount of state present. For short branches—two to four operations—predication is preferred, because the overhead for either z-cull or the branching instructions can negate any benefits. For branches that are embedded in larger programs, consider using the branching instructions rather than z-cull. Even 34.2 Basic Flow-Control Strategies 553

553.

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
	though the branching instructions are more sensitive to locality issues, using z-cull requires you to save all program state, execute the branch in a separate pass, and restore the state of the program to continue. For large programs, these saves and restores can make z-culling inefficient. However, if we can effectively isolate the branch component of our program, z-cull can provide the best performance. 34.3 Data-Dependent Looping with Occlusion Queries Another GPU feature designed to avoid drawing what is not visible is the hardware <i>occlusion query</i>. This feature enables you to query the number of pixels updated by a rendering call. Such queries are pipelined, which means that they provide a way to get a limited amount of data (an integer count) back from the GPU without stalling the pipeline, as occurs when we read back actual pixels. Because in GPGPU we are almost always drawing quads with known pixel coverage, we can use occlusion query with the <code>discard</code> keyword discussed in Section 34.2.3 to get a count of fragments updated and discarded. This allows us to implement global decisions controlled by the CPU based on GPU processing. Suppose we have a computation that needs to proceed iteratively on elements of a stream until all elements satisfy a termination criterion. To implement this on the GPU, we write a fragment program that implements the computation and one that tests the termination criterion. The latter program <code>discards</code> fragments that satisfy the termination criterion. We then write a loop on the CPU like the following pseudocode, which will execute until all stream elements satisfy the termination criteria. <pre> int numberNotTerminated = streamSize; while (numberNotTerminated > 0) { gpuRunProgram (computationProgram); gpuBeginQuery (myQuery); gpuRunProgram (terminationProgram); gpuEndQuery (myQuery); numberNotTerminated = gpuGetQueryResults (myQuery); } </pre> This technique can also be used for subdivision algorithms, such as the adaptive radiosity solution in Chapter 39 of this book, “Global Illumination Using Progressive Refinement Radiosity.” 554 Chapter 34 GPU Flow-Control Idioms

554.

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
	34.4 Conclusion With their support for branching at the fragment processor level, the latest generation of GPUs makes branching much easier to use in GPGPU programs. Higher-level language constructs such as <code>if</code>, <code>for</code>, and <code>while</code> compile directly to GPU assembly instructions, freeing the developer from having to use more complex strategies such as z-cull and occlusion queries, as is necessary on earlier GPUs. Nevertheless, there is still a penalty for incoherent branching on GPUs. Employing techniques based on precomputation and moving computation higher up the pipeline—either to the vertex processing unit or to the CPU, as described in Section 34.2.2—will continue to be a useful GPGPU strategy. 34.4 Conclusion 555 555.

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
	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 ’461 patent in a manner that would result in the subject matter of this limitation given, at the very least, the person's understanding of the state of the art, the problems addressed and solved in the prior art, and the teachings of GPU Gems. Specific examples of such references are cited and explained in these invalidity contentions, including the cover pleading and charts C1–C15, and Appendix C.
26. The method of claim 21, wherein executing the user interaction stream comprises: specifying an output to be saved to disk and/or displayed on a screen.	GPU Gems discloses, expressly and/or inherently, the method of claim 21, wherein executing the user interaction stream comprises specifying an output to be saved to disk and/or displayed on a screen. <i>See claim 21; Chapter 40.</i> <i>See e.g.:</i> <i>As a non-limiting example, GPU Gems discloses user interactions that specify an output to be saved to disk and/or displayed on a screen.</i>

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
	Chapter 12 Tile-Based Texture Mapping <i>Li-Yi Wei</i> <i>NVIDIA Corporation</i> Many graphics applications such as games and simulations frequently use large textures for walls, floors, or terrain. There are several issues with large textures. First, they can consume significant storage, in either disk space, system memory, or graphics memory. Second, they can consume significant bandwidth, during either initial texture loading or rendering. Third, most graphics cards impose an upper limit on individual texture sizes. For example, the GeForce 6800 has an upper resolution limit of $4,096 \times 4,096$ for RGBA8 textures, and many other GPUs have stricter limitations. One possible solution to address these problems is texture compression. However, existing GPU texture-compression techniques such as DXT have a limited compression ratio. In addition, texture compression cannot handle the problem of texture size limit. An alternative approach is texture tiling, as shown in Figure 12-1. For example, if we have a large wall or floor consisting of repeating tiles, then obviously we don't have to store all the tiles. Instead, we could have just one tile and repeat it on the wall. For more complex patterns, we can dice up the wall or floor into smaller polygons and apply different texture tiles or texture coordinate transformations for each polygon. The advantage of this approach is that we can achieve infinite compression ratio in theory, because we can produce an arbitrarily large output from a few tiles. The disadvantage, however, lies in the complication of application code and data. Introduction 186 189.

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
	33.2 The GPU Memory Model Graphics processors have their own memory hierarchy analogous to the one used by serial microprocessors, including main memory, caches, and registers. This memory hierarchy, however, is designed for accelerating graphics operations that fit into the streaming programming model rather than general, serial computation. Moreover, graphics APIs such as OpenGL and Direct3D further limit the use of this memory to graphics-specific primitives such as vertices, textures, and frame buffers. This section gives an overview of the current memory model on GPUs and how stream-based computation fits into it. 33.2.1 Memory Hierarchy 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. Figure 33-2. The CPU and GPU Memory Hierarchies 524 Chapter 33 Implementing Efficient Parallel Data Structures on GPUs 524.

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
649.	Chapter 40 Computer Vision on the GPU <i>James Fung</i> <i>University of Toronto</i> Computer vision tasks are computationally intensive and repetitive, and they often exceed the real-time capabilities of the CPU, leaving little time for higher-level tasks. However, many computer vision operations map efficiently onto the modern GPU, whose programmability allows a wide variety of computer vision algorithms to be implemented. This chapter presents methods of efficiently mapping common mathematical operations of computer vision onto modern computer graphics architecture. 40.1 Introduction 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. 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. 40.1 Introduction 649

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
	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. 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. 40.2 Implementation Framework Many computer vision operations can be considered sequences of filtering operations, with each sequential filtering stage acting upon the output of the previous filtering stage. On the GPU, these filtering operations are carried out by fragment programs. To apply these fragment programs to input images, the input images are initialized as textures and then mapped onto quadrilaterals. By displaying these quadrilaterals in appropriately sized windows, we can ensure that there is a one-to-one correspondence of image pixels to output fragments. When the textured quadrilateral is displayed, the fragment program then runs, operating identically on each pixel of the image. The fragment program is analogous to the body of a common for-loop program statement, but each iteration of the body can be thought of as executing in parallel. The resulting output is not the original image, but rather the filtered image. Modern GPUs allow input and results in full 32-bit IEEE floating-point precision, providing sufficient accuracy for many algorithms. A complete computer vision algorithm can be created by implementing sequences of these filtering operations. After the texture has been filtered by the fragment program, the resulting image is placed into texture memory, either by using render-to-texture extensions or by copying the frame buffer into texture memory. In this way, the output image becomes the input texture to the next fragment program. This creates a pipeline that runs the entire computer vision algorithm. However, often a complete vision algorithm will require operations beyond filtering. For example, summations are common operations; we present examples of how they can be implemented and used. Furthermore, more-generalized calculations, such as feature tracking, can also be mapped effectively onto graphics hardware. 650 Chapter 40 Computer Vision on the GPU 650.

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
651.	These methods are implemented in the OpenVIDIA GPU computer vision library, which can be used to create real-time computer vision programs (http://openvidia.sourceforge.net/). 40.3 Application Examples This chapter presents examples that implement common computer vision techniques on the GPU. The common thread between each of these algorithms is that they are all used in our system to create a fully interactive, real-time <i>computer-mediated reality</i>, in which the environment of a user appears to be augmented, diminished, or otherwise altered. Some examples are shown in Figure 40-1.  Figure 40-1. Using Computer Vision to Mediate Reality <i>Computer vision techniques can analyze video to augment, diminish, or otherwise alter a user's perception of the environment. On the left, detected ceiling planes become terminal windows. On the right, a terminal window is mapped to the plane of the floppy disk.</i> 40.3.1 Using Sequences of Fragment Programs for Computer Vision In this section, we demonstrate how a sequence of fragment programs can work together to perform more-complex computer vision operations. Correcting Radial Distortion Many camera lenses cause some sort of radial distortion (also known as <i>barrel distortion</i>) of the image, most commonly in wide-field-of-view or low-cost lenses. There are several ways of correcting for radial distortion on the graphics hardware; we present an example of how a fragment program can be used to do so. 40.3 Application Examples 651

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

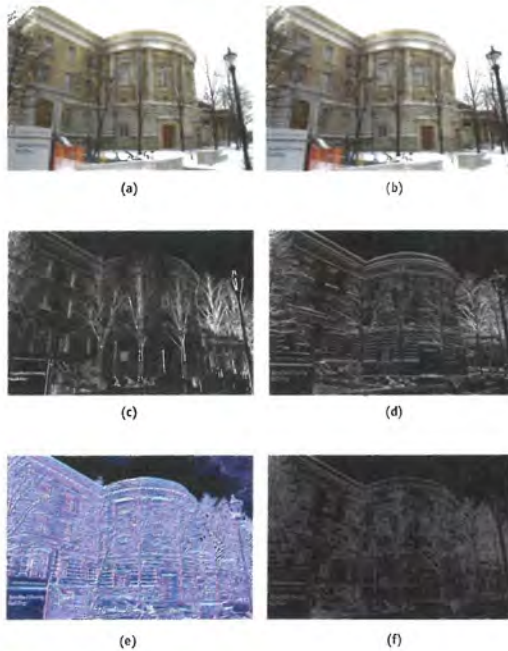
U.S. Patent No. RE49,461	GPU Gems
	We begin by assuming that the displacement of a pixel due to radial distortion is Δx, Δy and is commonly modeled by: $\Delta x = x(k_1 r^2 + k_2 r^4 + \dots),$ $\Delta y = y(k_1 r^2 + k_2 r^4 + \dots),$ where x, y are the image coordinates, r is the distance of the pixel from the principal point of the camera, and the coefficients $k_1, k_2, \dots$ are camera parameters determined by some known calibration method. Given a particular camera, the required calibration parameters can be found using the tools provided at Bouquet 2004. The fragment program shown in Listing 40-1 applies this correction by using the current texture coordinates to calculate an offset. The program then adds this offset to the current pixel's coordinates and uses the result to look up the texture value at the corrected coordinate. This texture value is then output, and the image's radial distortion is thereby corrected. Listing 40-1. Radial Undistortion <pre> float4 CorrectBarrelDistortion(float2 iptexCoord : TEXCOORD0, float4 wpos : WPOS, uniform samplerRECT FPEL : COLOR) { // known constants for the particular lens and sensor float f = 368.26488; // focal length float ox = 147.54834; // principal point, x axis float oy = 126.01673; // principal point, y axis float k1 = 0.4162; // constant for radial distortion correction float k2 = 0.40348; float2 xy = (iptexCoord - float2(ox, oy))/float2(f,f); r = sqrt(dot(xy, xy)); float r2 = r * r; float r4 = r2 * r2; float coeff = (k1 * r2 + k2 * r4); </pre> 652 Chapter 40 Computer Vision on the GPU

652.

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
653.	Listing 40-1 (continued). Radial Undistortion <pre> // add the calculated offsets to the current texture coordinates xy = ((xy + xy * coeff.xx) * f.xx) + float2(ox, oy); // look up the texture at the corrected coordinates // and output the color return texRECT(FPE1, xy); } </pre> A Canny Edge Detector Edge detection is a key algorithm used in many vision applications. Here we present an implementation of the commonly used "Canny" edge-detection algorithm that runs entirely on the GPU. See Figure 40-2 for an example. We implement the Canny edge detector as a series of fragment programs, each performing a step of the algorithm: • Step 1. Filter the image using a separable Gaussian edge filter (Jargstorff 2004). The filter footprint is typically around 15×15 elements wide, and separable. A separable filter is one whose 2D mask can be calculated by applying two 1D filters in the x and y directions. The separability saves us a large number of texture lookups, at the cost of an additional pass. • Step 2. Determine the magnitude, l, of the derivatives at each pixel computed in step 1, and quantize the direction d. The values of d and l are given by: $d = \arctan 2(d_y, d_x),$ $l = \sqrt{d_x^2 + d_y^2},$ where d is the direction of the gradient vector, quantized to one of the eight possible directions one may traverse a 3×3 region of pixels. The length l is the magnitude of the vector. • Step 3. Perform a nonmaximal suppression, as shown in Listing 40-2. The nonmaximal suppression uses the direction of the local gradient to determine which pixels are in the forward and backward direction of the gradient. This direction is used to calculate the x and y offset of the texture coordinates to look up the forward and backward neighbors. An edge is considered found only if the gradient is at a maximum as compared to its neighbors, ensuring a thin line along the detected edge. 40.3 Application Examples 653

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
	 (a) (b) (c) (d) (e) (f) Figure 40-2. Canny Edge Detection Each image shows the result of a fragment program operating on the previous image. (a) Original input. (b) Radial undistorted image. (c) Horizontal derivative. (d) Vertical derivative. (e) Gradient quantization. (f) Canny edges. 654 Chapter 40 Computer Vision on the GPU 654.

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
655.	Listing 40-2. A Canny Edge Detector Using Nonmaximal Suppression <i>This program looks up the direction of the gradient at the current pixel and then retrieves the pixels in the forward and backward directions along the gradient. The program returns 1 if the current pixel value is greater than the two values along the gradient; otherwise, it suppresses (or zeroes) the output. A thresholding value allows the sensitivity of the edge detector to be varied in real time to suit the vision apparatus and environment.</i> <pre> float4 CannySearch(uniform float4 thresh, float2 fptexCoord : TEXCOORD0, uniform samplerRECT FPPE1 : COLOR) { // magdir holds { dx, dy, mag, dirct } float4 magdir = texRECT(FPPE1, fptexCoord); float alpha = 0.5/sin(3.14159/8); // eight directions on grid float2 offset = round(alpha.xx * magdir.xy/magdir.zx); float4 fwdneighbour, backneighbour; fwdneighbour = texRECT(FPPE1, fptexCoord + offset); backneighbour = texRECT(FPPE1, fptexCoord - offset); float4 color0; if (fwdneighbour.z > magdir.z && backneighbour.z > magdir.z) color0 = float4(0.0, 0.0, 0.0, 0.0); // not an edge! else color0 = float4(1.0, 1.0, 1.0, 1.0); // is an edge! if (magdir.z < thresh.x) color0 = float4(0.0, 0.0, 0.0, 0.0); // thresholding return color0; } </pre> 40.3.2 Summation Operations Many common vision operations involve the summation of an image buffer. Some methods that require the summation are moment calculations and the solutions to linear systems of equations. Buck and Purcell (2004) discuss a general method of efficiently summing buffers by using local neighborhoods. On systems that do not support render-to-texture, however, the number of passes required by this technique can be limiting, due to the number of graphics context switches involved. We have found performing the computation in two 40.3 Application Examples 655

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
	passes to be more efficient on such systems. In our approach, we create a fragment program that sums each row of pixels, resulting in a single column holding the sums of each row. Then, in a second pass, we draw a single pixel and execute a program that sums down the column. The result is a single pixel that has as its "color" the sum of the entire buffer. The Cg code for computing the sum in this way is given in this chapter's appendix on the book's CD. We use the summation operation within the context of two different, but commonly used, algorithms that require summation over a buffer: a hand-tracking algorithm and an algorithm for creating panoramas from images by determining how to stitch them together. Tracking Hands Tracking a user's hands or face is a useful vision tool for creating interactive systems, and it can be achieved on the GPU by using a combination of image filtering and moment calculations. A common way to track a user's hand is to use color segmentation to find areas that are skin-colored, and then continually track the location of those areas through the image. Additionally, it is useful to track the average color of the skin tone, because it changes from frame to frame, typically due to changes in lighting. It is common to carry out the segmentation in an HSV color space, because skin tone of all hues tends to cluster in HSV color space. The HSV color space describes colors in terms of their <i>hue</i> (the type of color, such as red, green, or blue); <i>saturation</i> (the vibrancy of that color); and <i>value</i> (the brightness of the color). Cameras typically produce an RGB image, so a fragment program can be used to perform a fast color space conversion as a filtering operation. Assuming that we have some sort of initial guess for the HSV color of the skin, we can also use a fragment program to threshold pixels. Given a particular pixel's HSV value, we can determine its difference from the mean HSV skin color and either output to 0 if the difference is too great or output the HSV value otherwise. For instance, the comparison might be easily done as: <pre> if (distance(hsv, meanhsv) < thresh) { colorO = hsv; } else colorO = float4(0.0, 0.0, 0.0, 0.0); </pre> 656 Chapter 40 Computer Vision on the GPU 656.

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
	Finally, we determine the centroid of the skin-colored region and update the mean skin value based on what is currently being seen. For both of these steps, we need to calculate sums over the image. To compute centroids, we can first compute the first three moments of the image: $M_{00} = \sum_{x=0}^{W-1} \sum_{y=0}^{H-1} E(x, y),$ $M_{10} = \sum_{x=0}^{W-1} \sum_{y=0}^{H-1} xE(x, y),$ $M_{01} = \sum_{x=0}^{W-1} \sum_{y=0}^{H-1} yE(x, y),$ where W and H are the sizes of the image in each axis, and $E(x, y)$ is the pixel value at location (x, y). The x coordinate of the centroid is then computed as M_{10}/M_{00}, and M_{01}/M_{00} is the centroid's y coordinate. These summations can be done by first running a fragment program that fills a buffer with: $E(x, y), \quad xE(x, y), \quad yE(x, y)$ and then using a GPU summation method to sum the entire buffer. The result will be a single pixel whose value can then be read back to the CPU to update the location of the centroid. In our case—tracking the centroid of a thresholded skin-tone blob—we might simply let the fragment program output 1.0 in skin-colored areas and 0 elsewhere. Listing 40-3 shows a Cg program that produces four output components that, after summation, will provide the sums needed to calculate the zeroth and first-order moments. Figure 40-3 depicts this process. Should we require more statistics like the preceding, we could either reduce the precision and output eight 16-bit floating-point results to a 128-bit floating-point texture or use the multiple render targets (MRTs) OpenGL extension. 657.

Listing 40-3. Creating Output Statistics for Moment Calculations

```
float4 ComputeMoments(
    float2 texCoord : TEXCOORD0,
    uniform samplerRECT FPE1: TEXUNIT0 ) : COLOR
{
    float4 samp = texRECT(FPE1, fptexCoord);
    float4 colorO = float4(0, 0, 0, 0);
    if ( samp.x != 0.0 )
        colorO = float4( 1.0, fptexCoord.x, fptexCoord.y, samp.y);
}
```

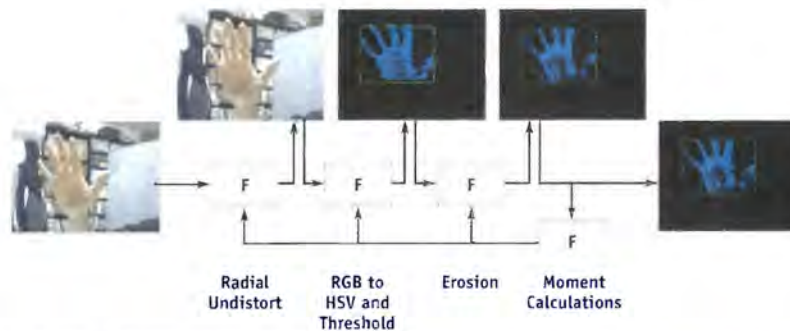


Figure 40-3. Tracking a User's Hand

40.3.3 Systems of Equations for Creating Image Panoramas

We also encounter this compute-and-sum type of calculation when we form systems of equations of the familiar form $Ax = b$.

VideoOrbits

VideoOrbits is an image-registration algorithm that calculates a coordinate transformation between pairs of images taken with a camera that is free to pan, tilt, rotate about its optical axis, and zoom. The technique solves the problem for two cases: (1) images taken from the same location of an arbitrary 3D scene or (2) images taken from arbitrary locations of a flat scene. Figure 40-4 shows an image panorama composed from several frames of video. Equivalently, Mann and Fung (2002) show how the same algorithm can be used to replace planes in the scene with computer-generated content, such as in Figure 40-1.

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
	658.  Figure 40-4. A VideoOrbits Panorama <i>This panorama shows how the images have been projected to fit together properly.</i> VideoOrbits solves for a projection of an image in order to register it with another image. Consider a pair of images of the same scene, with the camera having moved between images. A given point in the scene appearing at coordinates $[x, y]$ in the first image now appears at coordinates $[x', y']$ in the second image. Under the constraints mentioned earlier, all the points in the image can be considered to move according to eight parameters, given by the following equation: $\begin{bmatrix} x' \\ y' \end{bmatrix} = \frac{\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \end{bmatrix}}{\begin{bmatrix} c_1 & c_2 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + 1} \quad (1)$ The eight scalar parameters describing the projection are denoted by: $p = [a_{11}, a_{12}, b_1, a_{21}, a_{22}, b_2, c_1, c_2]$ and are calculated by VideoOrbits. Once the parameters p have been determined for each image, the images can then be transformed into the same coordinate system to 40.3 Application Examples 659 659.

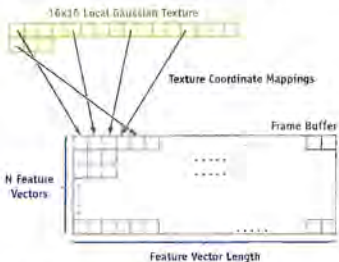
Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
	create the image panoramas of Figure 40-4. (For a more in-depth treatment of projective flow and the VideoOrbits algorithm, see Mann 1998.) VideoOrbits calculates the parameters p by solving a linear system of equations. The bulk of the computation is in the initialization of this system of equations, which is of the form $Ax = b$. Here, A is an 8×8 matrix whose values we must initialize, x is an 8×1 column vector that corresponds to p, and b is an 8×1 column vector we must also initialize. Solving this system is computationally expensive because each element of A and b is actually a summation of a series of multiplications and additions at each point in the image. Initializing each of the 64 entries of the matrix A involves many multiplications and additions over all the pixels. A is of the form: $A = \begin{bmatrix} \sum a_{11} & \sum a_{12} & \sum a_{13} & \dots & \sum a_{18} \\ \sum a_{21} & \sum a_{22} & \sum a_{23} & \dots & \sum a_{28} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \sum a_{81} & \sum a_{82} & \sum a_{83} & \dots & \sum a_{88} \end{bmatrix}_{8 \times 8} \quad (2)$ where each of the elements is a summation. (Note that the elements a_{11}, a_{12}, a_{21}, a_{22} are not those of Equation 1.) The arguments of these summations involve many arithmetic operations. For instance, element a_{21} is: $a_{21} = (x^2 E_x + xy E_y)(xy E_x + y^2 E_y) \quad (3)$ This operation must be carried out at every pixel and then summed to obtain each entry in matrix A. This compute-and-sum operation is similar to moment calculations discussed previously. Not surprisingly, we can calculate this on the GPU in a similar fashion, filling a buffer with the argument of the summation and then summing the buffer. Typically, to calculate all the matrix entries, multiple passes will be required. As each summation is completed, the results are copied to a texture that stores the results. After all the entries for the matrix A and vector b are computed, the values are read back to the CPU. This keeps the operations on the GPU, deferring the readback until all matrix entries have been calculated. This allows the GPU to operate with greater independence from the CPU. The system can then be solved on the CPU, with the majority of the per-pixel calculations already solved by the GPU. 660 Chapter 40 Computer Vision on the GPU 660.

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
661.	40.3.4 Feature Vector Computations Calculating <i>feature vectors</i> at detected feature points is a common operation in computer vision. A <i>feature</i> in an image is a local area around a point with some higher-than-average amount of "uniqueness." This makes the point easier to recognize in subsequent frames of video. The uniqueness of the point is characterized by computing a feature vector for each feature point. Feature vectors can be used to recognize the same point in different images and can be extended to more generalized object recognition techniques. Figure 40-5 shows the motion of the features between frames as the camera moves.  Figure 40-5. Detecting Distinctive Points as the Camera Pans <i>The GPU is used to detect and track distinctive points (features) in real time as the camera pans. The green lines connect the same point in the previous image to its new location in the current image.</i> Computing feature vectors on the GPU requires handling the image data in a different manner than the operations we have thus far presented. Because only a few points in the image are feature points, this calculation does not need to be done at every point in the image. Additionally, for a single feature point, a large feature vector (100 elements or more) is typically calculated. Thus, each single point produces a larger number of results, rather than the one-to-one or many-to-one mappings we have discussed so far. These types of computations can still be carried out on the GPU. We take as an example a feature vector that is composed of the filtered gradient magnitudes in a local region around a located feature point. Feature detection can be achieved using methods similar to the Canny edge detector that instead search for corners rather than lines. If the feature points are being detected using sequences of filtering, as is common, we can perform the filtering on the GPU 40.3 Application Examples 661

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
	and read back to the CPU a buffer that flags which pixels are feature points. The CPU can then quickly scan the buffer to locate each of the feature points, creating a list of image locations at which we will calculate feature vectors on the GPU. There are many approaches to generating feature vectors in order to track feature points. We show one approach as an example. For each detected feature point, we examine a 16×16 neighborhood of gradient magnitudes and directions (as computed in the Canny edge example). This neighborhood is divided up into 4×4-pixel regions, giving 16 different regions of 16 pixels. In each region, we create an 8-element histogram of gradient magnitudes. Each pixel's gradient magnitude is added to the bin of the histogram corresponding to its quantized gradient direction. For each region, the result is then an 8-element histogram. The feature vector is the concatenation of all 16 of these 8-element histograms, giving a feature vector length of $16 \times 8 = 128$ elements. Features in different images can be matched to each other by computing the Euclidean distance between pairs of feature vectors, with the minimum distance providing a best match. To produce the 128-element feature vector result, we draw rows of vertices, with one row for each feature, such that each drawn point in the row corresponds to an element of the feature vector for that feature. Associating texture values pointwise allows for the most flexibility in mapping texture coordinates for use in computing the feature vector. Figure 40-6 shows how the vertices are laid out, and the binding of the image texture coordinates. Each point has different texture coordinates mapped to it to allow a single fragment program to access the appropriate gradients of the local region used to calculate the histogram.  Figure 40-6. OpenGL Vertex Layout and Texture Bindings for Feature Vertices 662 Chapter 40 Computer Vision on the GPU 662.

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
	By associating the correct texture coordinates for each point vertex, the single fragment program shown in Listing 40-4 calculates the feature vectors. To produce 8 elements per region, we have packed the results into 16-bit half-precision floating-point values; alternatively, we could use MRTs to output multiple 32-bit values. After completion, we read back the frame buffer, containing one feature vector per row, to the CPU. <u>Listing 40-4. Fragment Program for Calculating Feature Vectors</u> <pre> Float4 FeatureVector() { float2 fptexCoord : TEXCOORD0, float2 tapsCoord : TEXCOORD1, float4 colorZero : COLOR0, float4 wpos : WPOS, uniform samplerRECT FPE1, uniform samplerRECT FPE2 : COLOR { const float4x4 lclGauss = (0.3536, 0.3536, 0.3536, 0.3536, 0.3536, 0.7071, 0.7071, 0.3536, 0.3536, 0.7071, 0.7071, 0.3536, 0.3536, 0.3536, 0.3536, 0.3536); const float eps = 0.001; float4 input; int xoffset = 1, yoffset = 1; float4 bins_0 = float4(0.0, 0.0, 0.0, 0.0); float4 bins_1 = float4(0.0, 0.0, 0.0, 0.0); for (xoffset = 0 ; xoffset < 4 ; xoffset++) { for (yoffset = 0 ; yoffset < 4 ; yoffset++) { input = texRECT(FPE1, fptexCoord + float2(xoffset, yoffset)); float dir = input.y; input *= lclGauss[xoffset,yoffset].xxxx * texRECT(FPE2, tapsCoord + float2(xoffset, yoffset)).xxxx; if (dir <= (1.0 + eps)/8.0) bins_0.x += input.x; /* . . . for each bin . . . */ else if (dir <= (7.0 + eps)/8.0) bins_1.x += input.x; else bins_1.w += input.x; } } return float4(pack_2half(bins_0.xy), pack_2half(bins_0.zw), pack_2half(bins_1.xy), pack_2half(bins_1.zw)) } </pre> 40.3 Application Examples 663 663.

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
	40.4 Parallel Computer Vision Processing When multiple computer vision algorithms are needed for an application, parallel GPU processing can be achieved by placing multiple graphics cards in the PCI bus slots of a motherboard. Figure 40-7 shows a machine built to explore the benefits of parallel computations by using multiple graphics cards. Running each computer vision program on a different GPU allows the cards to process in parallel; conventional shared memory can be used to allow the programs to interact. Although creating multi-GPU systems using PCI cards will soon no longer be possible due to the obsolescence of the PCI bus, the advent of PCI Express has made new multi-GPU approaches possible. One such approach is NVIDIA's SLI, which makes the presence of multiple GPUs transparent to the developer by automatically distributing work between them.  Figure 40-7. Multiple GPUs Used for a Computer Vision Application <i>Left: A computer vision application (Fung and Mann 2002) using graphics hardware to display six video streams in a hexagonal form. Right: A computer vision machine that runs computer vision tasks in parallel using six PCI graphics cards and one AGP graphics card. Each PCI card is a GeForce FX 5200 GPU, resulting in an inexpensive, powerful, and easily constructible parallel architecture well suited for pattern recognition and computer vision.</i> 40.5 Conclusion This chapter has presented techniques for mapping computer vision algorithms onto modern GPUs. We hope that this chapter has provided an understanding of how to write computer vision algorithms for fast processing on the GPU. Although other 664 Chapter 40 Computer Vision on the GPU 664.

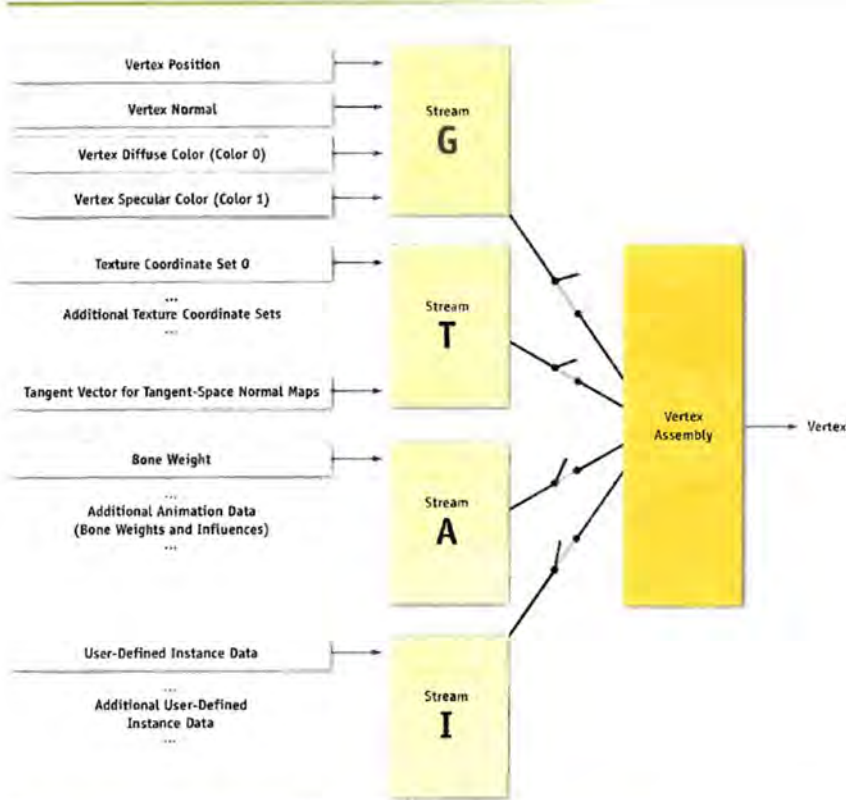
Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
	special-purpose hardware systems could be used to provide hardware acceleration of computer vision algorithms, the low cost and widespread availability of GPUs will make hardware-accelerated computer vision algorithms more accessible. 40.6 References The methods presented here are implemented in the OpenVIDIA project. OpenVIDIA is an open-source project that includes library functions and sample programs that run computer vision applications on the GPU. For more information, visit OpenVidia's Web site: http://openvidia.sourceforge.net. Bouguet, J.-Y. 2004. Camera Calibration Toolbox for Matlab. Available online at http://www.vision.caltech.edu/bouguetj/calib_doc/index.html Bradski, G. 1998. "Computer Vision Face Tracking for Use in a Perceptual User Interface." <i>Intel Technology Journal</i>. Buck, L., and T. Purcell. 2004. "A Toolkit for Computation on GPUs." In <i>GPU Gems</i>, edited by Randima Fernando, pp. 621–636. Addison-Wesley. Canny, J. F. 1986. "A Computational Approach to Edge Detection." <i>IEEE PAMI</i> 8(6), pp. 679–698. Fung, J., and S. Mann. 2004a. "Computer Vision Signal Processing on Graphics Processing Units." In <i>Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP 2004)</i>, Montreal, Quebec, Canada, pp. 83–89. Fung, J., and S. Mann. 2004b. "Using Multiple Graphics Cards as a General Purpose Parallel Computer: Applications to Computer Vision." In <i>Proceedings of the 17th International Conference on Pattern Recognition (ICPR2004)</i>, Cambridge, United Kingdom, pp. 805–808. Fung, J., F. Tang, and S. Mann. 2002. "Mediated Reality Using Computer Graphics Hardware for Computer Vision." In <i>Proceedings of the International Symposium on Wearable Computing 2002 (ISWC2002)</i>, Seattle, Washington, pp. 83–89. Horn, B., and B. Schunk. 1981. "Determining Optical Flow." <i>Artificial Intelligence</i> 17, pp. 185–203. Jargstorff, F. 2004. "A Framework for Image Processing." In <i>GPU Gems</i>, edited by Randima Fernando, pp. 445–467. Addison-Wesley. Lowe, D. 2004. "Distinctive Image Features from Scale-Invariant Keypoints." <i>International Journal of Computer Vision</i> 60(2), pp. 91–110. 40.6 References 665 665.

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
	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 ’461 patent in a manner that would result in the subject matter of this limitation given, at the very least, the person's understanding of the state of the art, the problems addressed and solved in the prior art, and the teachings of GPU Gems. Specific examples of such references are cited and explained in these invalidity contentions, including the cover pleading and charts C1–C15, and Appendix C.
27. The method of claim 21, wherein executing the user interaction stream comprises: parsing elements to be used in the computations representing the artificial neural network.	GPU Gems discloses, expressly and/or inherently, the method of claim 21, wherein executing the user interaction stream comprises parsing elements to be used in the computations representing the artificial neural network. <i>See</i> claim 21; PART IV: GENERAL-PURPOSE COMPUTATION on GPUS: A PRIMER, Chapters 29-36. <i>See e.g.:</i> <i>As a non-limiting example, GPU Gems discloses user interactions that result in parsing elements to be used in the computations.</i>

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

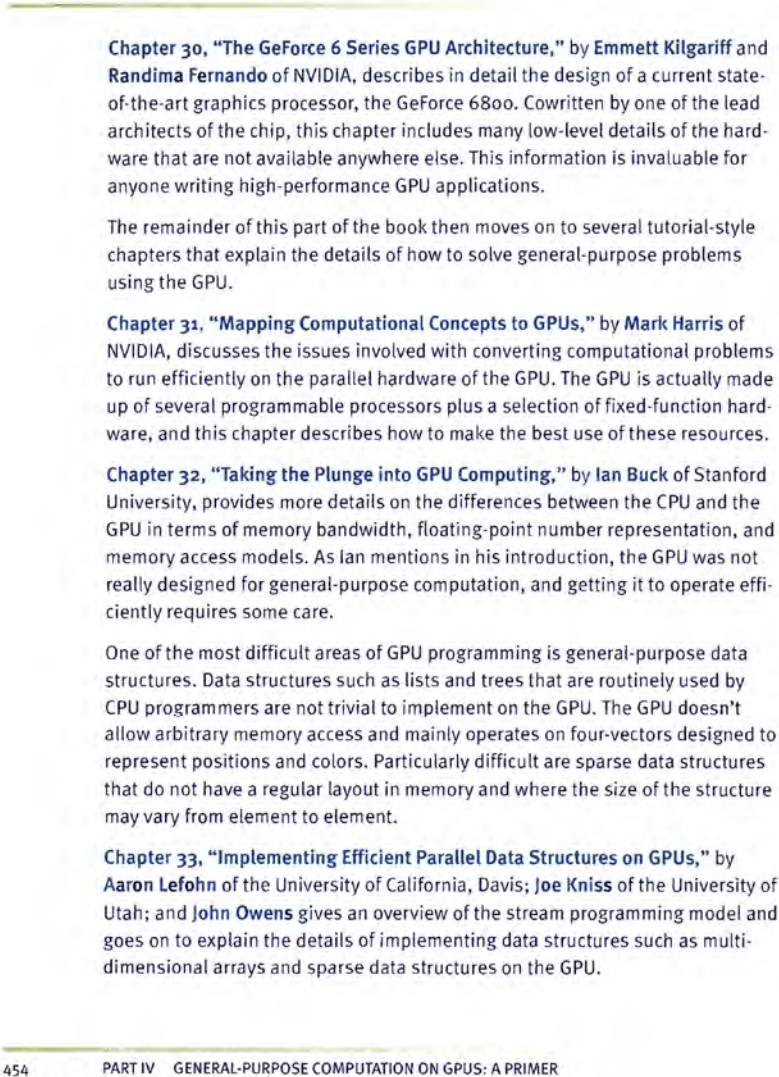
U.S. Patent No. RE49,461	GPU Gems
	 Figure 5-2. Combining Currently Needed Streams 5.2.1 Multistreaming with DirectX 9.0 Microsoft DirectX 8.0 introduced the notion of a stream to bind data to input registers. A stream is a uniform array of component data. Each component consists of one or more elements that represent a single entity, such as position, normal, color, texture coordinate set, and so on. With streams, GPUs are able to perform a direct memory access (DMA) from multiple vertex buffers in parallel. The DirectX9D caps bit

78.

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
	This part of the book aims to provide a gentle introduction to the world of general-purpose computation on graphics processing units, or “GPGPU,” as it has come to be known. The text is intended to be understandable to programmers with no graphics experience, as well as to those who have been programming graphics for years but have little knowledge of parallel computing for other applications. Since the publication of <i>GPU Gems</i>, GPGPU has grown from something of a curiosity to a well-respected active new area of graphics and systems research. Why would you want to go to the trouble of converting your computational problems to run on the GPU? There are two reasons: price and performance. Economics and the rise of video games as mass-market entertainment have driven down prices to the point where you can now buy a graphics processor capable of several hundred billion floating-point operations per second for just a few hundred dollars. The GPU is not well suited to all types of problems, but there are many examples of applications that have achieved significant speedups from using graphics hardware. The applications that achieve the best performance are typically those with high “arithmetic intensity”; that is, those with a large ratio of mathematical operations to memory accesses. These applications range all the way from audio processing and physics simulation to bioinformatics and computational finance. Anybody with any exposure to modern computing cannot fail to notice the rapid pace of technological change in our industry. The first chapter in this part, Chapter 29, “Streaming Architectures and Technology Trends,” by John Owens of the University of California, Davis, sets the stage for the chapters to come by describing the trends in semiconductor design and manufacturing that are driving the evolution of both the CPU and the GPU. One of the important factors driving these changes is the memory “gap”—the fact that computation speeds are increasing at a much faster rate than memory access speeds. This chapter also introduces the “streaming” computational model, which is a reasonably close match to the characteristics of modern GPU hardware. By using this style of programming, application programmers can take advantage of the GPU’s massive computation and memory bandwidth resources, and the resulting programs can achieve large performance gains over equivalent CPU implementations. PART IV GENERAL-PURPOSE COMPUTATION ON GPUS: A PRIMER 453 453.

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

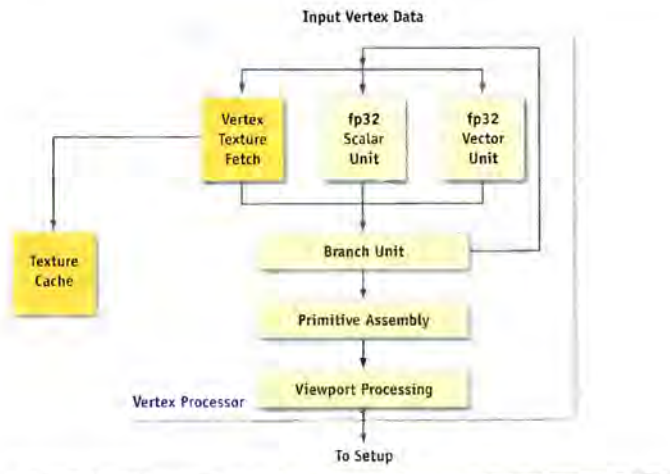
U.S. Patent No. RE49,461	GPU Gems
	 Chapter 30, "The GeForce 6 Series GPU Architecture," by Emmett Kilgariff and Randima Fernando of NVIDIA, describes in detail the design of a current state-of-the-art graphics processor, the GeForce 6800. Cowritten by one of the lead architects of the chip, this chapter includes many low-level details of the hardware that are not available anywhere else. This information is invaluable for anyone writing high-performance GPU applications. The remainder of this part of the book then moves on to several tutorial-style chapters that explain the details of how to solve general-purpose problems using the GPU. Chapter 31, "Mapping Computational Concepts to GPUs," by Mark Harris of NVIDIA, discusses the issues involved with converting computational problems to run efficiently on the parallel hardware of the GPU. The GPU is actually made up of several programmable processors plus a selection of fixed-function hardware, and this chapter describes how to make the best use of these resources. Chapter 32, "Taking the Plunge into GPU Computing," by Ian Buck of Stanford University, provides more details on the differences between the CPU and the GPU in terms of memory bandwidth, floating-point number representation, and memory access models. As Ian mentions in his introduction, the GPU was not really designed for general-purpose computation, and getting it to operate efficiently requires some care. One of the most difficult areas of GPU programming is general-purpose data structures. Data structures such as lists and trees that are routinely used by CPU programmers are not trivial to implement on the GPU. The GPU doesn't allow arbitrary memory access and mainly operates on four-vectors designed to represent positions and colors. Particularly difficult are sparse data structures that do not have a regular layout in memory and where the size of the structure may vary from element to element. Chapter 33, "Implementing Efficient Parallel Data Structures on GPUs," by Aaron Lefohn of the University of California, Davis; Joe Kniss of the University of Utah; and John Owens gives an overview of the stream programming model and goes on to explain the details of implementing data structures such as multi-dimensional arrays and sparse data structures on the GPU. 454 PART IV GENERAL-PURPOSE COMPUTATION ON GPUS: A PRIMER

454.

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

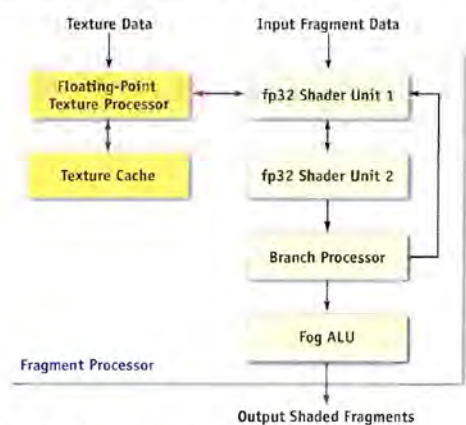
U.S. Patent No. RE49,461	GPU Gems
	Traditionally, GPUs have not been very good at executing code with branches. Because they are parallel machines, they achieve best performance when the the same operation can be applied to every data element. Chapter 34, “GPU Flow-Control Idioms,” by Mark Harris and Ian Buck, explains different ways in which flow-control structures such as loops and if statements can be efficiently implemented on the GPU. This includes using the depth-test and z-culling capabilities of modern GPUs, as well as the branching instructions available in the latest versions of the pixel shader hardware. Cliff Woolley of the University of Virginia has spent many hours writing GPGPU applications, and (like many of our other authors) he has published several papers based on his research. In Chapter 35, “GPU Program Optimization,” he passes on his experience on the best ways to optimize GPU code, and how to avoid the common mistakes made by novice GPU programmers. It is often said that premature optimization is the root of all evil, but it has to be done at some point. On the CPU, it is easy to write programs that have variable amounts of output per input data element. Unfortunately, this is much more difficult on a parallel machine like the GPU. Chapter 36, “Stream Reduction Operations for GPGPU Applications,” by Daniel Horn of Stanford University, illustrates several ways in which the GPU can be programmed to perform filtering operations that remove elements from a data stream in order to generate variable amounts of output. He demonstrates how this technique can be used to efficiently implement collision detection and subdivision surfaces. Only time will tell what the final division of labor between the CPU, the GPU, and other processors in the PC ecosystem will be. One thing is sure: the realities of semiconductor design and the memory gap mean that data-parallel programming is here to stay. By learning how to express your problems in this style today, you can ensure that your code will continue to execute at the maximum possible speed on all future hardware. <i>Simon Green, NVIDIA Corporation</i> PART IV GENERAL-PURPOSE COMPUTATION ON GPUS: A PRIMER 455 455.

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
	GPU—the GeForce 6 Series—allows vertex programs to fetch texture data. All operations are done in 32-bit floating-point (fp32) precision per component. The GeForce 6 Series architecture supports scalable vertex-processing horsepower, allowing the same architecture to service multiple price/performance points. In other words, high-end models may have six vertex units, while low-end models may have two. Because vertex processors can perform texture accesses, the vertex engines are connected to the texture cache, which is shared with the fragment processors. In addition, there is a vertex cache that stores vertex data both before and after the vertex processor, reducing fetch and computation requirements. This means that if a vertex index occurs twice in a draw call (for example, in a triangle strip), the entire vertex program doesn't have to be rerun for the second instance of the vertex—the cached result is used instead. Vertices are then grouped into primitives, which are points, lines, or triangles. The Cull/Clip/Setup blocks perform per-primitive operations, removing primitives that aren't visible at all, clipping primitives that intersect the view frustum, and performing edge and plane equation setup on the data in preparation for rasterization.  Figure 30-4. The GeForce 6 Series Vertex Processor 30.2 Overall System Architecture 475

475.

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
	The rasterization block calculates which pixels (or samples, if multisampling is enabled) are covered by each primitive, and it uses the z-cull block to quickly discard pixels (or samples) that are occluded by objects with a nearer depth value. Think of a fragment as a “candidate pixel”: that is, it will pass through the fragment processor and several tests, and if it gets through all of them, it will end up carrying depth and color information to a pixel on the frame buffer (or render target). Figure 30-5 illustrates the fragment processor (sometimes called a “pixel shader”) and texel pipeline. The texture and fragment-processing units operate in concert to apply a shader program to each fragment independently. The GeForce 6 Series architecture supports a scalable amount of fragment-processing horsepower. Another popular way to say this is that GPUs in the GeForce 6 Series can have a varying number of <i>fragment pipelines</i> (or “pixel pipelines”). Similar to the vertex processor, texture data is cached on-chip to reduce bandwidth requirements and improve performance. The texture and fragment-processing unit operates on squares of four pixels (called <i>quads</i>) at a time, allowing for direct computation of derivatives for calculating texture level of detail. Furthermore, the fragment processor works on groups of hundreds of pixels at a time in single-instruction, multiple-data (SIMD) fashion (with each fragment processor engine working on one fragment concurrently), hiding the latency of texture fetch from the computational performance of the fragment processor.  Figure 30-5. The GeForce 6 Series Fragment Processor and Texel Pipeline 476 Chapter 30 The GeForce 6 Series GPU Architecture 476.

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
	The fragment processor uses the texture unit to fetch data from memory, optionally filtering the data before returning it to the fragment processor. The texture unit supports many source data formats (see Section 30.3.3, "Supported Data Storage Formats"). Data can be filtered using bilinear, trilinear, or anisotropic filtering. All data is returned to the fragment processor in fp32 or fp16 format. A texture can be viewed as a 2D or 3D array of data that can be read by the texture unit at arbitrary locations and filtered to reconstruct a continuous function. The GeForce 6 Series supports filtering of fp16 textures in hardware. The fragment processor has two fp32 shader units per pipeline, and fragments are routed through both shader units and the branch processor before recirculating through the entire pipeline to execute the next series of instructions. This rerouting happens once for each core clock cycle. Furthermore, the first fp32 shader can be used for perspective correction of texture coordinates when needed (by dividing by w), or for general-purpose multiply operations. In general, it is possible to perform eight or more math operations in the pixel shader during each clock cycle, or four math operations if a texture fetch occurs in the first shader unit. On the final pass through the pixel shader pipeline, the fog unit can be used to blend fog in fixed-point precision with no performance penalty. Fog blending happens often in conventional graphics applications and uses the following function: <pre>out = FogColor * fogFraction + SrcColor * (1 - fogFraction)</pre> This function can be made fast and small using fixed-precision math, but in general IEEE floating point, it requires two full multiply-adds to do effectively. Because fixed point is efficient and sufficient for fog, it exists in a separate small unit at the end of the shader. This is a good example of the trade-offs in providing flexible programmable hardware while still offering maximum performance for legacy applications. Fragments leave the fragment-processing unit in the order that they are rasterized and are sent to the z-compare and blend units, which perform depth testing (z comparison and update), stencil operations, alpha blending, and the final color write to the target surface (an off-screen render target or the frame buffer). The memory system is partitioned into up to four independent memory partitions, each with its own dynamic random-access memories (DRAMs). GPUs use standard DRAM modules rather than custom RAM technologies to take advantage of market economics and thereby reduce cost. Having smaller, independent memory partitions allows the memory subsystem to operate efficiently regardless of whether large or small blocks of data are transferred. All rendered surfaces are stored in the DRAMs, while textures and input data can be stored in the DRAMs or in system memory. The four 30.2 Overall System Architecture 477 477.

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
498.	they are incapable of gather. The NVIDIA GeForce 6 Series GPUs have a new feature called <i>vertex texture fetch</i> (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. Fragment Processors 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. 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. Rasterizer As mentioned earlier, after the vertex processors transform vertices, each group of three vertices is used to compute a triangle (in the form of edge equations), and from this triangle a stream of fragments is generated. This work of generating fragments is done by the <i>rasterizer</i>. We can think of the rasterizer as an address interpolator. Later we show how memory addresses are represented as texture coordinates. The rasterizer interpolates these addresses and other per-vertex values based on the fragment position. Because it generates many data elements from only a few input elements, we can also think of the rasterizer as a data amplifier. These functions of the rasterizer are very specialized to rendering triangles and are not user-programmable. Texture Unit Fragment processors (and vertex processors on the latest GPUs) can access memory in the form of textures. We can think of the texture unit as a read-only memory interface. 498 Chapter 31 Mapping Computational Concepts to GPUs

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
	coordinates are passed as input to the fragment processor. In computer graphics, these coordinates are used as indices for texture fetches. For GPGPU, we can think of them as array indices, and we can use them to control the domain of the computation. The domain and range may be the same size, or the domain can be smaller than the range (data amplification/magnification), or the domain can be larger than the range (data minification). The rasterizer makes it easy to correctly sample the input stream at the correct intervals for each of these cases. 31.3.6 Vertex Coordinates = Computational Range As discussed before, fragments are generated from input geometry by the rasterizer, and these fragments become output pixels after fragment processing. Because the fragment processors are not directly capable of scatter, the input vertices and the vertex program determine which pixels are generated. Typically, we specify four vertices of a quad in output pixel coordinates and apply a vertex program that simply passes the vertices through untransformed. Thus, vertex coordinates directly control the output range of the computation. 31.3.7 Reductions Everything we've discussed up to this point has assumed purely parallel computation: each element is computed largely independently of the rest of the stream. However, there are times when we need to <i>reduce</i> a large vector of values to a smaller vector, or even to a single value. For example, we might need to compute the sum or the maximum of all values in an array. This sort of computation is called a <i>parallel reduction</i>. On GPUs, reductions can be performed by alternately rendering to and reading from a pair of buffers. On each pass, the size of the output (the computational range) is reduced by some fraction. To produce each element of the output, a fragment program reads two or more values and computes a new one using the reduction operator, such as addition or maximum. These passes continue until the output is a single-element buffer, at which point we have our reduced result. In general, this process takes $O(\log n)$ passes, where n is the number of elements to reduce. For example, for a 2D reduction, the fragment program might read four elements from four quadrants of the input buffer, such that the output size is halved in both dimensions at each step. Figure 31-2 demonstrates a max reduction on a 2D buffer. 502 Chapter 31 Mapping Computational Concepts to GPUs 502.

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
503.	Figure 31-2. Max Reduction Performed with Multiple Passes 31.4 From Analogies to Implementation 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. 31.4.1 Putting It All Together: A Basic GPGPU Framework 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 <i>pug</i>, incorporates all of the analogies from the previous section in order to abstract GPGPU programming in a manner more accessible to experienced CPU programmers. Initializing and Finalizing a GPGPU Application The first step in a GPGPU application is to initialize the GPU. This is very easy in the framework: just call <code>pugInit()</code>. When the application is finished with the GPU, it can clean up the memory used by the graphics API by calling <code>pugCleanup()</code>. Specifying Kernels 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 <i>The Cg Tutorial</i> (Fernando and Kilgard 2003) and the 31.4 From Analogies to Implementation 503

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
	Chapter 36 Stream Reduction Operations for GPGPU Applications <i>Daniel Horn</i> Stanford University Many GPGPU-based applications rely on the fragment processor, which operates across a large set of output memory locations, consuming a fixed number of input elements per location and operating a small program on those elements to produce a single output element in that location. Because the fragment program must write its results to a preordained memory location, it is not able to vary the amount of data that it outputs according to the input data it processes. (See Chapter 30 in this book, “The GeForce 6 Series GPU Architecture,” for more details on the capabilities and limits of the fragment processor.) Many algorithms are difficult to implement under these limitations—specifically, algorithms that reduce many data elements to few data elements. The reduction of data by a fixed factor has been carefully studied on GPUs (Buck et al. 2004); such operations require an amount of time linear in the size of the data to be reduced. However, nonuniform reductions—that is, reductions that filter out data based on its content on a per-element basis—have been less thoroughly studied, yet they are required for a number of interesting applications. We refer to such a technique of nonuniform reduction as a method of <i>data filtering</i>. This chapter presents a data filtering algorithm that makes it possible to write GPU programs that efficiently generate varying amounts of output data. Introduction 573 573. 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

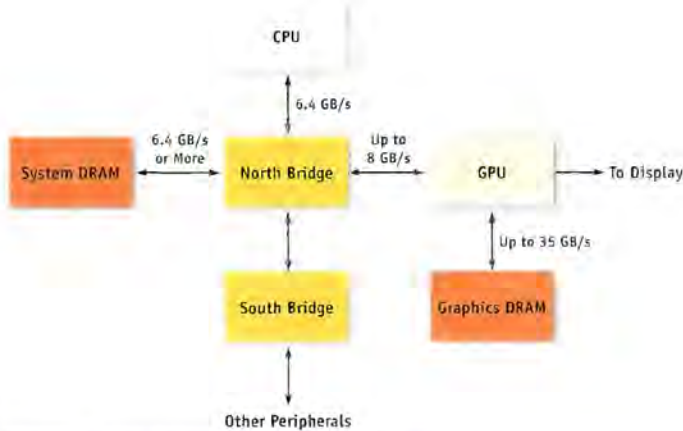
Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
	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 ’461 patent in a manner that would result in the subject matter of this limitation given, at the very least, the person's understanding of the state of the art, the problems addressed and solved in the prior art, and the teachings of GPU Gems. Specific examples of such references are cited and explained in these invalidity contentions, including the cover pleading and charts C1–C15, and Appendix C.
28. The method of claim 27, wherein the processing unit comprises a graphics processing unit (GPU) and executing the user interaction stream further comprises: converting the elements into GPU programs.	GPU Gems discloses, expressly and/or inherently, the method of claim 27, wherein the processing unit comprises a graphics processing unit (GPU) and executing the user interaction stream further comprises converting the elements into GPU programs. <i>See claim 27; Chapters 29-36.</i> <i>See e.g.:</i> <i>As a non-limiting example, GPU Gems discloses that the processing unit is a GPU. GPU Gems also discloses user interactions that result in converting the elements into GPU programs (e.g., vertex and shader programs).</i> Chapter 40 explains that 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.” Thus, GPUs can be used to “free up the CPU for other tasks.”

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
469.	29.4.3 Challenge: Supporting More Programmability and Functionality While the current generation of graphics hardware features substantially more programmability than previous generations, the general programmability of GPUs is still far from ideal. One step toward addressing this trend is to improve the functionality and flexibility within the two current programmable units (vertex and fragment). It is likely that we will see their instruction sets converge and add functionality, and that their control flow capabilities will become more general as well. We may even see programmable hardware shared between these two stages in an effort to better utilize these resources. GPU architects will have to be mindful, however, that such improvements not affect the GPU's performance on its core tasks. Another option will be expanding programmability to different units. Geometric primitives particularly benefit from programmability, so we may soon see programmable processing on surfaces, triangles, and pixels. As GPU vendors support more general pipelines and more complex and varied shader computation, many researchers have used the GPU to address tasks outside the bounds of the graphics pipeline. This book contains examples of many of these efforts; the general-purpose computation on GPUs (GPGPU) community has successfully addressed problems in visual simulation, image processing, numerical methods, and databases with graphics hardware. We can expect that these efforts will grow in the future as GPUs continue to increase in performance and functionality. Historically, we have seen GPUs subsume functionality previously belonging to the CPU. Early consumer-level graphics hardware could not perform geometry processing on the graphics processor; it was only five years ago that the entire graphics pipeline could be fabricated on a single chip. Though since that time the primary increase in GPU functionality has been directed toward programmability within the graphics pipeline, we should not expect that GPU vendors have halted their efforts to identify more functions to integrate onto the GPU. In particular, today's games often require large amounts of computation in physics and artificial intelligence computations. Such computation may be attractive for future GPUs. 29.4 The Future and Challenges 469

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
	Table 30-1 reiterates some of the points made in the preceding chapter: there is a vast amount of bandwidth available internally on the GPU. Algorithms that run on the GPU can therefore take advantage of this bandwidth to achieve dramatic performance improvements. 30.2 Overall System Architecture 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.  Figure 30-2. The Overall System Architecture of a PC 30.2.1 Functional Block Diagram for Graphics Operations 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. 30.2 Overall System Architecture 473

473.

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461

GPU Gems

```

for all particles p
  voxels[p.pos] += p.vel
  
```

Figure 32-4. Computing the Net Particle Velocity in a Voxel

Here the address we are using for the scatter is not fixed, because the particles are moving from voxel to voxel in each time step of our simulation. We cannot convert this scatter into a gather directly because we don't know where to fetch our scatter data until runtime.

We can still convert these scatter operations into gathers using a technique called *address sorting*. The algorithm is similar to the earlier example except instead of simply outputting the data to scatter, we also output the scatter address with the data. We can then sort the data address pairs based on the address. This places all of the data to be scattered in contiguous array locations. Finally, we loop over each element of the array into which we want to scatter, using a binary search to fetch the data to scatter. See Figure 32-5.

Scatter Data with Write Addresses								
Data	a	b	c	d	e	f	g	h
Address	5	2	1	5	2	3	6	0

Sorted Scatter Data								
Data	h	c	b	e	f	a	d	g
Address	0	1	2	2	3	5	5	6

array[i] = binarysearch(data, i)

Array	h	c	b+e	f	f	a+d	g
-------	---	---	-----	---	---	-----	---

Figure 32-5. Converting Scatters to Gathers with Address Sorting

32.3 Implementing Scatter

517

517.

517.

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

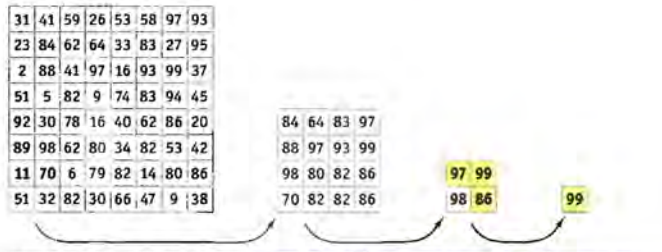
U.S. Patent No. RE49,461	GPU Gems
	Following are two sample snippets of code showing how to transform a serial program into a data-parallel stream program. The first sample shows a loop over an array (for example, the pixels in an image) for a serial processor. Note that the instructions in the loop body are specified for only a single data element at a time: <pre>for (i = 0; i < data.size(); i++) loopBody(data[i])</pre> The next sample shows the same section of code written in pseudocode for a streaming processor: <pre>inDataStream = specifyInputData() kernel = loopBody() outDataStream = apply(kernel, inDataStream)</pre> The first line specifies the data stream. In the case of our image example, the stream is all of the pixels in the image. The second line specifies the computational kernel, which is simply the body of the loop from the first sample. Lastly, the third line applies the kernel to all elements in the input stream and stores the results in an output stream. In the image example, this operation would process the entire image and produce a new, transformed image. Current GPU fragment processors have an additional programming restriction beyond the streaming model described earlier. Current GPU fragment processors are single-instruction, multiple-data (SIMD) parallel processors. This has traditionally meant that all stream elements (that is, fragments) must be processed by the same sequence of instructions. Recent GPUs (those supporting Pixel Shader 3.0 [Microsoft 2004a]) relax this strict SIMD model slightly by allowing variable-length loops and limited fragment-level branching. The hardware remains fundamentally SIMD, however, and thus branching must be spatially coherent between fragments for efficient execution (see Chapter 34 in this book, "GPU Flow-Control Idioms," for more information). Current vertex processors (Vertex Shader 3.0 [Microsoft 2004b]) are multiple-instruction, multiple-data (MIMD) machines, and can therefore execute kernel branches more efficiently than fragment processors. Although less flexible, the fragment processor's SIMD architecture is highly efficient and cost-effective. Because nearly all GPGPU computations are currently performed with the more powerful fragment processor, GPU-based data structures must fit into the fragment processor's streaming, SIMD programming model. Therefore, all data structures in this chapter are expressed as streams, and the computations on these data structures are in the form of SIMD, data-parallel kernels. 33-1 Programming with Streams 523

523.

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
497.	<p data-bbox="709 289 1251 321">31.2.1 Programmable Parallel Processors</p> <p data-bbox="806 331 1608 662">GPUs have two types of programmable processors: <i>vertex processors</i> and <i>fragment processors</i>. 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 <i>vertex program</i> (sometimes called a <i>vertex shader</i>) 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 <i>fragments</i> 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 <i>fragment program</i> (sometimes called a <i>pixel shader</i>) to each fragment in the stream to compute the final color of each pixel.</p> <p data-bbox="806 688 1008 714">Vertex Processors</p> <p data-bbox="806 721 1608 990">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. 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 data-bbox="806 1010 1608 1188">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,</p> <div data-bbox="1058 1318 1608 1344" data-cs="2" data-kind="parent"> <div>31.2 An Inventory of GPU Computational Resources</div> <div>497</div> </div>

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
	 Figure 31-2. Max Reduction Performed with Multiple Passes 31.4 From Analogies to Implementation 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. 31.4.1 Putting It All Together: A Basic GPGPU Framework 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 <i>pug</i>, incorporates all of the analogies from the previous section in order to abstract GPGPU programming in a manner more accessible to experienced CPU programmers. Initializing and Finalizing a GPGPU Application The first step in a GPGPU application is to initialize the GPU. This is very easy in the framework: just call <code>pugInit()</code>. When the application is finished with the GPU, it can clean up the memory used by the graphics API by calling <code>pugCleanup()</code>. Specifying Kernels 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 <i>The Cg Tutorial</i> (Fernando and Kilgard 2003) and the 31.4 From Analogies to Implementation 503
503.	

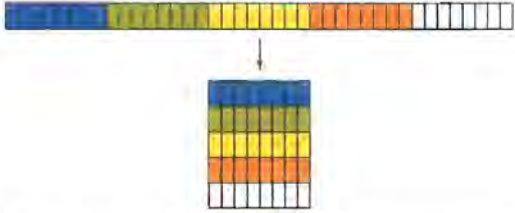
Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
507.	Listing 31-1. The Reaction-Diffusion Kernel Program <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> 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> <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> 31.5 A Simple Example 507

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
	Following are two sample snippets of code showing how to transform a serial program into a data-parallel stream program. The first sample shows a loop over an array (for example, the pixels in an image) for a serial processor. Note that the instructions in the loop body are specified for only a single data element at a time: <pre data-bbox="716 402 1035 443">for (i = 0; i < data.size(); i++) loopBody(data[i])</pre> The next sample shows the same section of code written in pseudocode for a streaming processor: <pre data-bbox="716 524 1125 581">inDataStream = specifyInputData() kernel = loopBody() outDataStream = apply(kernel, inDataStream)</pre> The first line specifies the data stream. In the case of our image example, the stream is all of the pixels in the image. The second line specifies the computational kernel, which is simply the body of the loop from the first sample. Lastly, the third line applies the kernel to all elements in the input stream and stores the results in an output stream. In the image example, this operation would process the entire image and produce a new, transformed image. Current GPU fragment processors have an additional programming restriction beyond the streaming model described earlier. Current GPU fragment processors are single-instruction, multiple-data (SIMD) parallel processors. This has traditionally meant that all stream elements (that is, fragments) must be processed by the same sequence of instructions. Recent GPUs (those supporting Pixel Shader 3.0 [Microsoft 2004a]) relax this strict SIMD model slightly by allowing variable-length loops and limited fragment-level branching. The hardware remains fundamentally SIMD, however, and thus branching must be spatially coherent between fragments for efficient execution (see Chapter 34 in this book, "GPU Flow-Control Idioms," for more information). Current vertex processors (Vertex Shader 3.0 [Microsoft 2004b]) are multiple-instruction, multiple-data (MIMD) machines, and can therefore execute kernel branches more efficiently than fragment processors. Although less flexible, the fragment processor's SIMD architecture is highly efficient and cost-effective. Because nearly all GPGPU computations are currently performed with the more powerful fragment processor, GPU-based data structures must fit into the fragment processor's streaming, SIMD programming model. Therefore, all data structures in this chapter are expressed as streams, and the computations on these data structures are in the form of SIMD, data-parallel kernels. 33-1 Programming with Streams 523 523.

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
	Given these restrictions, 2D textures with nearest-neighbor filtering are the substrate on which nearly all GPGPU-based data structures are built.² Note that until recently, the size of each texture dimension was required to be a power of two, but this restriction is thankfully no longer in place. All of the following examples use address translation to convert an N-D array address into a 2D texture address. Although it is important to minimize the cost of this address translation, remember that CPU-based multidimensional arrays must also perform address translation to look up values in the underlying 1D array. In fact, the GPU's texture-addressing hardware actually helps make these translations very efficient. Current GPUs do, however, suffer from one problem related to these address translations: the limitations of floating-point addressing. Chapter 32 of this book discusses important limitations and errors associated with using floating-point rather than integer addresses, and those problems apply to the techniques presented in this section. 1D Arrays One-dimensional arrays are represented by packing the data into a 2D texture, as shown in Figure 33-5. Current GPUs can therefore represent 1D arrays containing up to 16,777,216 ($4,096 \times 4,096$) elements.³ Each time this packed array is accessed from a fragment or vertex program, the 1D address must be converted to a 2D texture coordinate. Two versions of this address translation are shown in Cg syntax in Listings 33-1 and 33-2. Listing 33-1 (Buck et al. 2004) shows code for using <i>rectangle</i> textures, which use integer  Figure 33-5. 1D Arrays Packed into 2D Arrays Doing so avoids the size restrictions that current GPUs place on 1D textures. 2. If a data structure will remain static throughout an entire computation (for example, traditional texture uses), then 1D and 3D textures can and should be used. 3. Note that a 4096×4096 texture containing floating-point RGBA texels consumes 256 MB of RAM—as of January 2005, only the NVIDIA Quadro FX 4400 with unified frame buffer provides adequate contiguous memory. 33.3 GPU-Based Data Structures 529

529.

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
	addresses and do not support repeat-tiling address modes. Note that CONV_CONST is a constant based on the texture size and should be precomputed rather than recomputed for each stream element. Section 33.4 describes an elegant technique for computing values such as CONV_CONST with a feature of the Cg compiler called <i>program specialization</i>. With this optimization, Listing 33-1 compiles to three assembly instructions. Listing 33-1. 1D to 2D Address Translation for Integer-Addressed Textures <pre> float2 addrTranslation_1DtoRECT(float address1D, float2 texSize) { // Parameter explanation: // - "address1D" is a 1D array index into an array of size N // - "texSize" is size of RECT texture that stores the 1D array // Step 1) Convert 1D address from [0,N) to [0,texSize.y] float CONV_CONST = 1.0 / texSize.x; float normAddr1D = address1D * CONV_CONST; // Step 2) Convert the [0,texSize.y] 1D address to 2D. // Performing Step 1 first allows us to efficiently compute the // 2D address with "frac" and "floor" instead of modulus // and divide. Note that the "floor" of the y-component is // performed by the texture system. float2 address2D = float2(frac(normAddr1D) * texSize.x, normAddr1D); return address2D; } </pre> Listing 33-2 shows an address translation routine that can be used with traditional 2D textures, which use normalized [0,1] addressing. If CONV_CONST is precomputed, this address translation takes two fragment assembly instructions. It may also be possible to eliminate the <code>frac</code> instruction from Listing 33-2 by using the repeating-tiling addressing mode (such as <code>GL_REPEAT</code>). This reduces the conversion to a single assembly instruction. This optimization may be problematic on some hardware and texture configurations, however, so it should be carefully tested on your target architecture. 2D Arrays Two-dimensional arrays are represented simply as 2D textures. Their maximum size is limited by the GPU driver. That limit for current GPUs ranges from 2048×2048 to 4096×4096 depending on the display driver and the GPU. These limits can be queried via the graphics API. 530 Chapter 33 Implementing Efficient Parallel Data Structures on GPUs 530.

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
	Listing 33-2. 1D to 2D Address Translation for Normalized-Addressed Textures <pre> float2 addrTranslation_1Dto2D(float address1D, float2 texSize) { // Parameter explanation: // "address1D" is a 1D array index into an array of size N // - "texSize" is size of 2D texture that stores the 1D array // NOTE: Precompute CONV_CONST before running the kernel. float2 CONV_CONST = float2(1.0 / texSize.x, 1.0 / (texSize.x * texSize.y)); // Return a normalized 2D address (with values in [0,1]) float2 normAddr2D = address1D * CONV_CONST; float2 address2D = float2(frac(normAddr2D.x), normAddr2D.y); return address2D; } </pre> 3D Arrays Three-dimensional arrays may be stored in one of three ways: in a 3D texture, with each slice stored in a separate 2D texture, or packed into a single 2D texture (Harris et al. 2003, Lefohn et al. 2003, Goodnight et al. 2003). Each of these techniques has pros and cons that affect the best representation for an application. The simplest approach, using a 3D texture, has two distinct advantages. The first is that no address translation computation is required to access the memory. The second advantage is that the GPU's native trilinear filtering may be used to easily create high-quality volume renderings of the data. A disadvantage is that GPUs can update at most only four slices of the volume per render pass—thus requiring many passes to write to the entire array. The second solution is to store each slice of the 3D texture in a separate 2D texture, as shown in Figure 33-6. The advantages are that, like a native 3D representation, no address translation is required for data access, and yet each slice can be easily updated without requiring render-to-slice-of-3D-texture API support. The disadvantage is that the volume can no longer be truly randomly accessed, because each slice is a separate texture. The programmer must know which slice numbers will be accessed before the kernel executes, because fragment and vertex programs cannot dynamically compute which texture to access at runtime. 33-3 GPU-Based Data Structures 531

531.

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
649.	Chapter 40 Computer Vision on the GPU <i>James Fung</i> <i>University of Toronto</i> Computer vision tasks are computationally intensive and repetitive, and they often exceed the real-time capabilities of the CPU, leaving little time for higher-level tasks. However, many computer vision operations map efficiently onto the modern GPU, whose programmability allows a wide variety of computer vision algorithms to be implemented. This chapter presents methods of efficiently mapping common mathematical operations of computer vision onto modern computer graphics architecture. 40.1 Introduction 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. 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. 40.1 Introduction 649

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
	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. 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. 40.2 Implementation Framework Many computer vision operations can be considered sequences of filtering operations, with each sequential filtering stage acting upon the output of the previous filtering stage. On the GPU, these filtering operations are carried out by fragment programs. To apply these fragment programs to input images, the input images are initialized as textures and then mapped onto quadrilaterals. By displaying these quadrilaterals in appropriately sized windows, we can ensure that there is a one-to-one correspondence of image pixels to output fragments. When the textured quadrilateral is displayed, the fragment program then runs, operating identically on each pixel of the image. The fragment program is analogous to the body of a common for-loop program statement, but each iteration of the body can be thought of as executing in parallel. The resulting output is not the original image, but rather the filtered image. Modern GPUs allow input and results in full 32-bit IEEE floating-point precision, providing sufficient accuracy for many algorithms. A complete computer vision algorithm can be created by implementing sequences of these filtering operations. After the texture has been filtered by the fragment program, the resulting image is placed into texture memory, either by using render-to-texture extensions or by copying the frame buffer into texture memory. In this way, the output image becomes the input texture to the next fragment program. This creates a pipeline that runs the entire computer vision algorithm. However, often a complete vision algorithm will require operations beyond filtering. For example, summations are common operations; we present examples of how they can be implemented and used. Furthermore, more-generalized calculations, such as feature tracking, can also be mapped effectively onto graphics hardware. 650 Chapter 40 Computer Vision on the GPU 650.

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
	40.4 Parallel Computer Vision Processing When multiple computer vision algorithms are needed for an application, parallel GPU processing can be achieved by placing multiple graphics cards in the PCI bus slots of a motherboard. Figure 40-7 shows a machine built to explore the benefits of parallel computations by using multiple graphics cards. Running each computer vision program on a different GPU allows the cards to process in parallel; conventional shared memory can be used to allow the programs to interact. Although creating multi-GPU systems using PCI cards will soon no longer be possible due to the obsolescence of the PCI bus, the advent of PCI Express has made new multi-GPU approaches possible. One such approach is NVIDIA's SLI, which makes the presence of multiple GPUs transparent to the developer by automatically distributing work between them.  Figure 40-7. Multiple GPUs Used for a Computer Vision Application <i>Left: A computer vision application (Fung and Mann 2002) using graphics hardware to display six video streams in a hexagonal form. Right: A computer vision machine that runs computer vision tasks in parallel using six PCI graphics cards and one AGP graphics card. Each PCI card is a GeForce FX 5200 GPU, resulting in an inexpensive, powerful, and easily constructible parallel architecture well suited for pattern recognition and computer vision.</i> 40.5 Conclusion This chapter has presented techniques for mapping computer vision algorithms onto modern GPUs. We hope that this chapter has provided an understanding of how to write computer vision algorithms for fast processing on the GPU. Although other 664 Chapter 40 Computer Vision on the GPU 664. 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 '461 patent in a manner that would result in the subject matter of this limitation given, at the very least, the person's understanding of the state of the art, the problems addressed and solved in the prior art, and the teachings of GPU Gems. Specific

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
	examples of such references are cited and explained in these invalidity contentions, including the cover pleading and charts C1–C15, and Appendix C.
29. The method of claim 28, wherein executing the user interaction stream comprises: compiling the GPU programs.	GPU Gems discloses, expressly and/or inherently, the method of claim 28, wherein executing the user interaction stream comprises: compiling the GPU programs. <i>See claim 28; Chapter 33.</i> <i>See e.g.:</i> <i>As a non-limiting example, GPU Gems discloses user interactions that result in compiling the GPU programs (e.g., vertex and shader programs).</i>

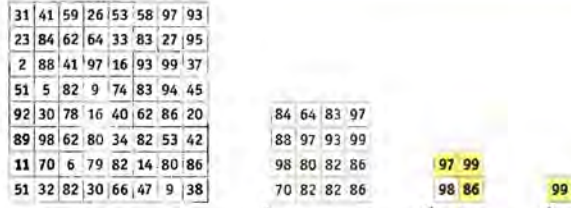
Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
469.	29.4.3 Challenge: Supporting More Programmability and Functionality While the current generation of graphics hardware features substantially more programmability than previous generations, the general programmability of GPUs is still far from ideal. One step toward addressing this trend is to improve the functionality and flexibility within the two current programmable units (vertex and fragment). It is likely that we will see their instruction sets converge and add functionality, and that their control flow capabilities will become more general as well. We may even see programmable hardware shared between these two stages in an effort to better utilize these resources. GPU architects will have to be mindful, however, that such improvements not affect the GPU's performance on its core tasks. Another option will be expanding programmability to different units. Geometric primitives particularly benefit from programmability, so we may soon see programmable processing on surfaces, triangles, and pixels. As GPU vendors support more general pipelines and more complex and varied shader computation, many researchers have used the GPU to address tasks outside the bounds of the graphics pipeline. This book contains examples of many of these efforts; the general-purpose computation on GPUs (GPGPU) community has successfully addressed problems in visual simulation, image processing, numerical methods, and databases with graphics hardware. We can expect that these efforts will grow in the future as GPUs continue to increase in performance and functionality. Historically, we have seen GPUs subsume functionality previously belonging to the CPU. Early consumer-level graphics hardware could not perform geometry processing on the graphics processor; it was only five years ago that the entire graphics pipeline could be fabricated on a single chip. Though since that time the primary increase in GPU functionality has been directed toward programmability within the graphics pipeline, we should not expect that GPU vendors have halted their efforts to identify more functions to integrate onto the GPU. In particular, today's games often require large amounts of computation in physics and artificial intelligence computations. Such computation may be attractive for future GPUs. 29.4 The Future and Challenges 469

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

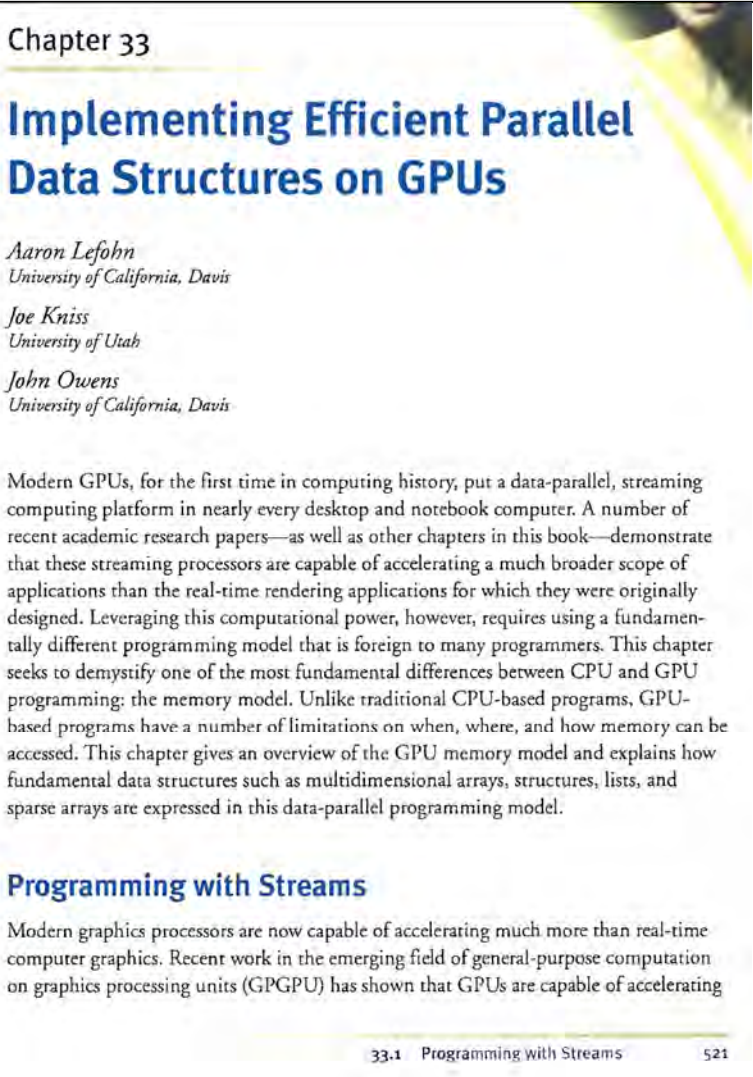
U.S. Patent No. RE49,461	GPU Gems
497.	31.2.1 Programmable Parallel Processors GPUs have two types of programmable processors: <i>vertex processors</i> and <i>fragment processors</i>. Vertex processors process streams of vertices (made up of positions, colors, normal vectors, and other attributes), which are the elements that compose polygonal geometric models. Computer graphics typically represents 3D objects with triangular meshes. The vertex processors apply a <i>vertex program</i> (sometimes called a <i>vertex shader</i>) 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 <i>fragments</i> 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 <i>fragment program</i> (sometimes called a <i>pixel shader</i>) to each fragment in the stream to compute the final color of each pixel. Vertex Processors 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. Because of this, vertex processors have hardware to process four-component vectors. This allows them to produce transformed vertex positions in fewer cycles. 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, 11.2 An Inventory of GPU Computational Resources 497

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

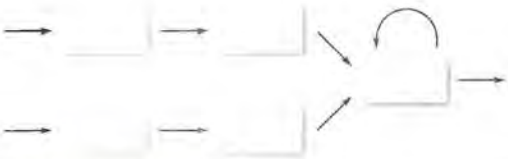
U.S. Patent No. RE49,461	GPU Gems
	 Figure 31-2. Max Reduction Performed with Multiple Passes 31.4 From Analogies to Implementation By now you have a high-level understanding of how most GPGPU programs operate on current GPUs. The analogies from the previous section provide us a general implementation 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. 31.4.1 Putting It All Together: A Basic GPGPU Framework 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 <i>pug</i>, incorporates all of the analogies from the previous section in order to abstract GPGPU programming in a manner more accessible to experienced CPU programmers. Initializing and Finalizing a GPGPU Application The first step in a GPGPU application is to initialize the GPU. This is very easy in the framework: just call <code>pugInit()</code>. When the application is finished with the GPU, it can clean up the memory used by the graphics API by calling <code>pugCleanup()</code>. Specifying Kernels 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 <i>The Cg Tutorial</i> (Fernando and Kilgard 2003) and the 31.4 From Analogies to Implementation 503

503.

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
521.	 Chapter 33 Implementing Efficient Parallel Data Structures on GPUs <i>Aaron Lefohn</i> <i>University of California, Davis</i> <i>Joe Kniss</i> <i>University of Utah</i> <i>John Owens</i> <i>University of California, Davis</i> Modern GPUs, for the first time in computing history, put a data-parallel, streaming computing platform in nearly every desktop and notebook computer. A number of recent academic research papers—as well as other chapters in this book—demonstrate that these streaming processors are capable of accelerating a much broader scope of applications than the real-time rendering applications for which they were originally designed. Leveraging this computational power, however, requires using a fundamentally different programming model that is foreign to many programmers. This chapter seeks to demystify one of the most fundamental differences between CPU and GPU programming: the memory model. Unlike traditional CPU-based programs, GPU-based programs have a number of limitations on when, where, and how memory can be accessed. This chapter gives an overview of the GPU memory model and explains how fundamental data structures such as multidimensional arrays, structures, lists, and sparse arrays are expressed in this data-parallel programming model. 33.1 Programming with Streams Modern graphics processors are now capable of accelerating much more than real-time computer graphics. Recent work in the emerging field of general-purpose computation on graphics processing units (GPGPU) has shown that GPUs are capable of accelerating 33.1 Programming with Streams 521

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
	applications as varied as fluid dynamics, advanced image processing, photorealistic rendering, and even computational chemistry and biology (Buck et al. 2004, Harris et al. 2004). The key to using the GPU for purposes other than real-time rendering is to view it as a <i>streaming, data-parallel</i> computer (see Chapter 29 in this book, "Streaming Architectures and Technology Trends," as well as Dally et al. 2004). The way in which we structure computation and access memory in GPU programs is greatly influenced by this stream computation model. As such, we first give a brief overview of this model before discussing GPU-based data structures. Streaming processors such as GPUs are programmed in a fundamentally different way than serial processors like today's CPUs. Most programmers are familiar with a programming model in which they can write to any location in memory at any point in their program. When programming a streaming processor, in contrast, we access memory in a much more structured manner. In the stream model, programs are expressed as series of operations on data streams, as shown in Figure 33-1. The elements in a <i>stream</i> (that is, an ordered array of data) are processed by the instructions in a <i>kernel</i> (that is, a small program). A kernel operates on each element of a stream and writes the results to an output stream. The stream programming model restrictions allow GPUs to execute kernels in parallel and therefore process many data elements <i>simultaneously</i>. This <i>data parallelism</i> is made possible by ensuring that the computation on one stream element cannot affect the computation on another element in the same stream. Consequently, the only values that can be used in the computation of a kernel are the inputs to that kernel and global memory reads. In addition, GPUs require that the outputs of kernels be independent: kernels cannot perform random writes into global memory (in other words, they may write only to a single stream element position of the output stream). The data parallelism afforded by this model is fundamental to the speedup offered by GPUs over serial processors.  Figure 33-1. Stream Program as Data-Dependency Graph The nodes are kernels and the edges are data streams. Kernels process all stream elements in parallel and write their results to output streams. 522 Chapter 33 Implementing Efficient Parallel Data Structures on GPUs

522.

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
	Following are two sample snippets of code showing how to transform a serial program into a data-parallel stream program. The first sample shows a loop over an array (for example, the pixels in an image) for a serial processor. Note that the instructions in the loop body are specified for only a single data element at a time: <pre> for (i = 0; i < Data.size(); i++) loopBody(Data[i]) </pre> The next sample shows the same section of code written in pseudocode for a streaming processor: <pre> inDataStream = specifyInputData() kernel = loopBody() outDataStream = apply(kernel, inDataStream) </pre> The first line specifies the data stream. In the case of our image example, the stream is all of the pixels in the image. The second line specifies the computational kernel, which is simply the body of the loop from the first sample. Lastly, the third line applies the kernel to all elements in the input stream and stores the results in an output stream. In the image example, this operation would process the entire image and produce a new, transformed image. Current GPU fragment processors have an additional programming restriction beyond the streaming model described earlier. Current GPU fragment processors are single-instruction, multiple-data (SIMD) parallel processors. This has traditionally meant that all stream elements (that is, fragments) must be processed by the same sequence of instructions. Recent GPUs (those supporting Pixel Shader 3.0 [Microsoft 2004a]) relax this strict SIMD model slightly by allowing variable-length loops and limited fragment-level branching. The hardware remains fundamentally SIMD, however, and thus branching must be spatially coherent between fragments for efficient execution (see Chapter 34 in this book, "GPU Flow-Control Idioms," for more information). Current vertex processors (Vertex Shader 3.0 [Microsoft 2004b]) are multiple-instruction, multiple-data (MIMD) machines, and can therefore execute kernel branches more efficiently than fragment processors. Although less flexible, the fragment processor's SIMD architecture is highly efficient and cost-effective. Because nearly all GPGPU computations are currently performed with the more powerful fragment processor, GPU-based data structures must fit into the fragment processor's streaming, SIMD programming model. Therefore, all data structures in this chapter are expressed as streams, and the computations on these data structures are in the form of SIMD, data-parallel kernels. 33.1 Programming with Streams 523 523.

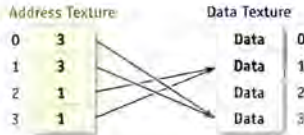
Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
525.	33.2.2 GPU Stream Types Unlike CPU memory, GPU memory has a number of usage restrictions and is accessible only via the abstractions of a graphics programming interface. Each of these abstractions can be thought of as a different stream type with its own set of access rules. The three types of streams visible to the GPU programmer are vertex streams, frame-buffer streams, and texture streams. A fourth stream type, fragment streams, is produced and consumed entirely within the GPU. Figure 33-3 shows the pipeline for a modern GPU, the three user-accessible streams, and the points in the pipeline where each can be used. Vertex Streams Vertex streams are specified as vertex buffers via the graphics API. These streams hold vertex positions and a variety of per-vertex attributes. These attributes have traditionally been used for texture coordinates, colors, normals, and so on, but they can be used for arbitrary input stream data for vertex programs. Vertex programs are not allowed to randomly index into their input vertices. Until recently, vertex streams could be updated only by transferring data from the CPU to the GPU. The GPU was not allowed to write to vertex streams. Recent API enhancements, however, have made it possible for the GPU to write to vertex streams. This is accomplished by either "copy-to-vertex-buffer" or "render-to-vertex-buffer." In the former technique, rendering results are copied from a frame buffer to a vertex buffer; in the latter technique, the rendering results are written directly to a vertex buffer. The recent addition of GPU-writable vertex streams enables GPUs, for the first time, to loop stream results from the end to the beginning of the pipeline. Figure 33-3. Streams in Modern GPUs <i>The GPU programmer can directly access vertices, frame buffers, and textures. Fragment streams are generated by the rasterizer and consumed by the fragment processor. They are the input streams to fragment programs but are created and consumed entirely within the GPU and so are not directly accessible to the programmer.</i> 33.2 The GPU Memory Model 525

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
526.	Fragment Streams Fragment streams are generated by the rasterizer and consumed by the fragment processor. They are the stream inputs to fragment programs, but they are not directly accessible to programmers because they are created and consumed entirely within the graphics processor. Fragment stream values include all of the interpolated outputs from the vertex processor: position, color, texture coordinates, and so on. As with per-vertex stream attributes, the per-fragment values that have traditionally been used for texture coordinates may now be used for any stream value required by the fragment program. Fragment streams cannot be randomly accessed by fragment programs. Permitting random access to the fragment stream would create a dependency between fragment stream elements, thus breaking the data-parallel guarantee of the programming model. If random access to fragment streams is needed by algorithms, the stream must first be saved to memory and converted to a texture stream. Frame-Buffer Streams Frame-buffer streams are written by the fragment processor. They have traditionally been used to hold pixels for display to the screen. Streaming GPU computation, however, uses frame buffers to hold the results of intermediate computation stages. In addition, modern GPUs are able to write to multiple frame-buffer surfaces (that is, multiple RGBA buffers) simultaneously. Current GPUs can write up to 16 floating-point scalar values per render pass (this value is expected to increase in future hardware). Frame-buffer streams cannot be randomly accessed by either fragment or vertex programs. They can, however, be directly read from or written to by the CPU via the graphics API. Lastly, recent API enhancements have begun to blur the distinction between frame buffers, vertex buffers, and textures by allowing a render pass to be directly written to any of these stream types. Texture Streams Textures are the only GPU memory that is randomly accessible by fragment programs and, for Vertex Shader 3.0 GPUs, vertex programs. If programmers need to randomly index into a vertex, fragment, or frame-buffer stream, they must first convert it to a texture. Textures can be read from and written to by either the CPU or the GPU. The GPU writes to textures either by rendering directly to them instead of to a frame buffer or by copying data from a frame buffer to texture memory. Textures are declared as 1D, 2D, or 3D streams and addressed with a 1D, 2D, or 3D address, respectively. A texture can also be declared as a cube map, which can be treated as an array of six 2D textures. 526 Chapter 33 Implementing Efficient Parallel Data Structures on GPUs

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
	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 <i>dependent texturing</i>.  Figure 33-4. Implementing Pointer Streams with Textures 33.3 GPU-Based Data Structures While the previous sections have described the GPU and its programming model with elegant abstractions, we now delve into the details of real-world data structures on current GPUs. The abstractions from Sections 33.1 and 33.2 continue to apply to the data structures herein, but the architectural restrictions of current GPUs make the real-world implementations slightly more complicated. We first describe implementations of basic structures: multidimensional arrays and structures. We then move on to more advanced structures in Sections 33.3.3 and 33.3.4: static and dynamic sparse structures. 33.3.1 Multidimensional Arrays Why include a section about multidimensional arrays if the GPU already provides 1D, 2D, and 3D textures? There are two reasons. First, current GPUs provide only 2D rasterization and 2D frame buffers. This means the only kind of texture that can be easily updated is a 2D texture, because it is a natural replacement for a 2D frame buffer.¹ Second, textures currently have restrictions on their size in each dimension. For example, current GPUs do not support 1D textures with more than 4,096 elements. If GPUs could write to an N-D frame buffer (where $N = 1, 2, 3, \dots$) with no size restrictions, this section would be trivial. 1. Upcoming extensions to OpenGL allow rendering to a slice of a 3D texture. 528 Chapter 33 Implementing Efficient Parallel Data Structures on GPUs 528.

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
	addresses and do not support repeat-tiling address modes. Note that CONV_CONST is a constant based on the texture size and should be precomputed rather than recomputed for each stream element. Section 33.4 describes an elegant technique for computing values such as CONV_CONST with a feature of the Cg compiler called <i>program specialization</i>. With this optimization, Listing 33-1 compiles to three assembly instructions. Listing 33-1. 1D to 2D Address Translation for Integer-Addressed Textures <pre> float2 addrTranslation_1DtoRECT(float address1D, float2 texSize) { // Parameter explanation: // - "address1D" is a 1D array index into an array of size N // - "texSize" is size of RECT texture that stores the 1D array // Step 1) Convert 1D address from [0,N] to [0,texSize.y] float CONV_CONST = 1.0 / texSize.x; float normAddr1D = address1D * CONV_CONST; // Step 2) Convert the [0,texSize.y] 1D address to 2D. // Performing Step 1 first allows us to efficiently compute the // 2D address with "frac" and "floor" instead of modulus // and divide. Note that the "floor" of the y-component is // performed by the texture system. float2 address2D = float2(frac(normAddr1D) * texSize.x, normAddr1D); return address2D; } </pre> Listing 33-2 shows an address translation routine that can be used with traditional 2D textures, which use normalized [0,1] addressing. If CONV_CONST is precomputed, this address translation takes two fragment assembly instructions. It may also be possible to eliminate the <code>frac</code> instruction from Listing 33-2 by using the repeating-tiling addressing mode (such as <code>GL_REPEAT</code>). This reduces the conversion to a single assembly instruction. This optimization may be problematic on some hardware and texture configurations, however, so it should be carefully tested on your target architecture. 2D Arrays Two-dimensional arrays are represented simply as 2D textures. Their maximum size is limited by the GPU driver. That limit for current GPUs ranges from 2048 × 2048 to 4096 × 4096 depending on the display driver and the GPU. These limits can be queried via the graphics API. 530 Chapter 33 Implementing Efficient Parallel Data Structures on GPUs

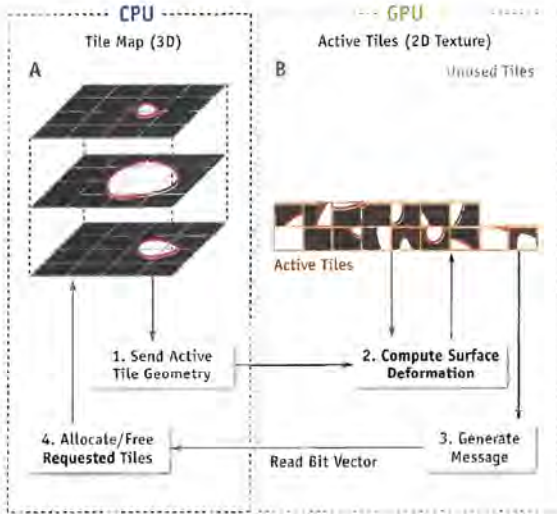
530.

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
	Figure 33-6. Storing a 3D Texture with Separate 2D Slices <i>Three-dimensional textures can be stored with each slice in a separate 2D texture. The advantage is that no address translation is required to access the data and 2D slices are easily updated in separate render passes. The disadvantages are that each slice must be updated in separate render passes and the slice number to be accessed must be known before a kernel may access the data.</i> The third option is to pack the 3D array into a single 2D texture, as shown in Figure 33-7. This solution requires an address translation for each data access, but the entire 3D array can be updated in a single render pass and no render-to-slice-of-3D-texture functionality is required. The ability to update the entire volume in a single render pass may have significant performance benefits, because the GPU's parallelism is better utilized when processing large streams. Additionally, unlike the 2D slice layout, the entire 3D array can be randomly accessed from within a kernel. A disadvantage of both the second and third schemes is that the GPU's native trilinear filtering cannot be used for high-quality volume rendering of the data. Fortunately, Figure 33-7. 3D Arrays Flattened into a Single 2D Texture <i>Three-dimensional arrays can be flattened into a single 2D texture (or pBuffer) so that the entire volume can be updated in a single render pass and the entire 3D array may be randomly accessed. The data may be laid out in slices (as shown here) or in a linear packing, as described in Listing 33-3.</i> 532 Chapter 33 Implementing Efficient Parallel Data Structures on GPUs

532.

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
	Another GPU-based dynamic sparse data structure is the sparse volume structure used for implicit surface deformation by Lefohn et al. (2003, 2004). An <i>implicit surface</i> defines a 3D surface as an <i>isosurface</i> (or level set) of a volume. A common example of 2D isosurfaces is the contour lines drawn on topographic maps. A contour line comprises points on the map that have the same elevation. Similarly, an implicit 3D surface is an isosurface of the scalar values stored in a volume's voxels. Representing surfaces in this way is very convenient from a mathematical standpoint, allowing the surface to freely distort and change topology. Efficient representations of implicit surfaces use a sparse data structure that stores only the voxels near the surface rather than the entire volume, as shown in Figure 33-9.  Figure 33-9. Dynamic Sparse Volume Data on the GPU The tile-based sparse data structure and algorithm used in Lefohn et al. 2003, 2004 to represent a deforming implicit surface. The implicit surface deformation computation is performed only on the sparse set of tiles that contain the surface. As the surface moves, new tiles must be allocated and others must be freed. The algorithm uses the GPU to compute the surface deformation and uses the CPU as a memory (tile) manager. The GPU requests tile allocations by sending a bit vector message to the CPU.

538

Chapter 33 Implementing Efficient Parallel Data Structures on GPUs

538.

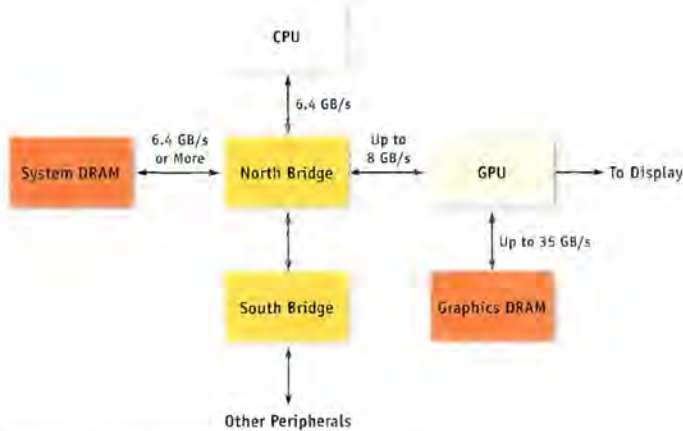
Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
	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 ’461 patent in a manner that would result in the subject matter of this limitation given, at the very least, the person's understanding of the state of the art, the problems addressed and solved in the prior art, and the teachings of GPU Gems. Specific examples of such references are cited and explained in these invalidity contentions, including the cover pleading and charts C1–C15, and Appendix C.
30. The method of claim 29, wherein executing the user interaction stream comprises: transferring the GPU programs to the second memory partition.	GPU Gems discloses, expressly and/or inherently, the method of claim 29, wherein executing the user interaction stream comprises transferring the GPU programs to the second memory partition. <i>See claim 29.</i> <i>See e.g.:</i> <i>As a non-limiting example, GPU Gems discloses user interactions that control transferring GPU programs (e.g., vertex and/or shader programs) to a second memory partition (e.g., a portion of GPU memory).</i>

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
469.	29.4.3 Challenge: Supporting More Programmability and Functionality While the current generation of graphics hardware features substantially more programmability than previous generations, the general programmability of GPUs is still far from ideal. One step toward addressing this trend is to improve the functionality and flexibility within the two current programmable units (vertex and fragment). It is likely that we will see their instruction sets converge and add functionality, and that their control flow capabilities will become more general as well. We may even see programmable hardware shared between these two stages in an effort to better utilize these resources. GPU architects will have to be mindful, however, that such improvements not affect the GPU's performance on its core tasks. Another option will be expanding programmability to different units. Geometric primitives particularly benefit from programmability, so we may soon see programmable processing on surfaces, triangles, and pixels. As GPU vendors support more general pipelines and more complex and varied shader computation, many researchers have used the GPU to address tasks outside the bounds of the graphics pipeline. This book contains examples of many of these efforts; the general-purpose computation on GPUs (GPGPU) community has successfully addressed problems in visual simulation, image processing, numerical methods, and databases with graphics hardware. We can expect that these efforts will grow in the future as GPUs continue to increase in performance and functionality. Historically, we have seen GPUs subsume functionality previously belonging to the CPU. Early consumer-level graphics hardware could not perform geometry processing on the graphics processor; it was only five years ago that the entire graphics pipeline could be fabricated on a single chip. Though since that time the primary increase in GPU functionality has been directed toward programmability within the graphics pipeline, we should not expect that GPU vendors have halted their efforts to identify more functions to integrate onto the GPU. In particular, today's games often require large amounts of computation in physics and artificial intelligence computations. Such computation may be attractive for future GPUs. 29.4 The Future and Challenges 469

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

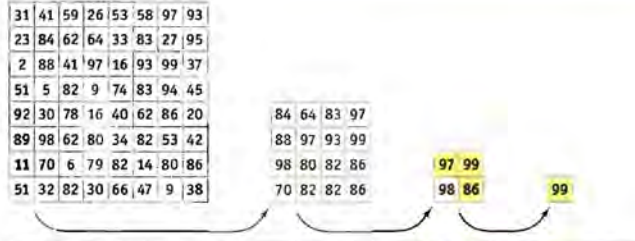
U.S. Patent No. RE49,461	GPU Gems
	Table 30-1 reiterates some of the points made in the preceding chapter: there is a vast amount of bandwidth available internally on the GPU. Algorithms that run on the GPU can therefore take advantage of this bandwidth to achieve dramatic performance improvements. 30.2 Overall System Architecture 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.  Figure 30-2. The Overall System Architecture of a PC 30.2.1 Functional Block Diagram for Graphics Operations 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. 30.2 Overall System Architecture 473

473.

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
	Figure 30-3. A Block Diagram of the GeForce 6 Series Architecture First, commands, textures, and vertex data are received from the host CPU through shared buffers in system memory or local frame-buffer memory. A command stream is written by the CPU, which initializes and modifies state, sends rendering commands, and references the texture and vertex data. Commands are parsed, and a vertex fetch unit is used to read the vertices referenced by the rendering commands. The commands, vertices, and state changes flow downstream, where they are used by subsequent pipeline stages. The vertex processors (sometimes called “vertex shaders”), shown in Figure 30-4, allow for a program to be applied to each vertex in the object, performing transformations, skinning, and any other per-vertex operation the user specifies. For the first time, a 474 Chapter 30 The GeForce 6 Series GPU Architecture 474.

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
	 Figure 31-2. Max Reduction Performed with Multiple Passes 31.4 From Analogies to Implementation 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. 31.4.1 Putting It All Together: A Basic GPGPU Framework 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 <i>pug</i>, incorporates all of the analogies from the previous section in order to abstract GPGPU programming in a manner more accessible to experienced CPU programmers. Initializing and Finalizing a GPGPU Application The first step in a GPGPU application is to initialize the GPU. This is very easy in the framework: just call <code>pugInit()</code>. When the application is finished with the GPU, it can clean up the memory used by the graphics API by calling <code>pugCleanup()</code>. Specifying Kernels 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 <i>The Cg Tutorial</i> (Fernando and Kilgard 2003) and the 31.4 From Analogies to Implementation 503
503.	

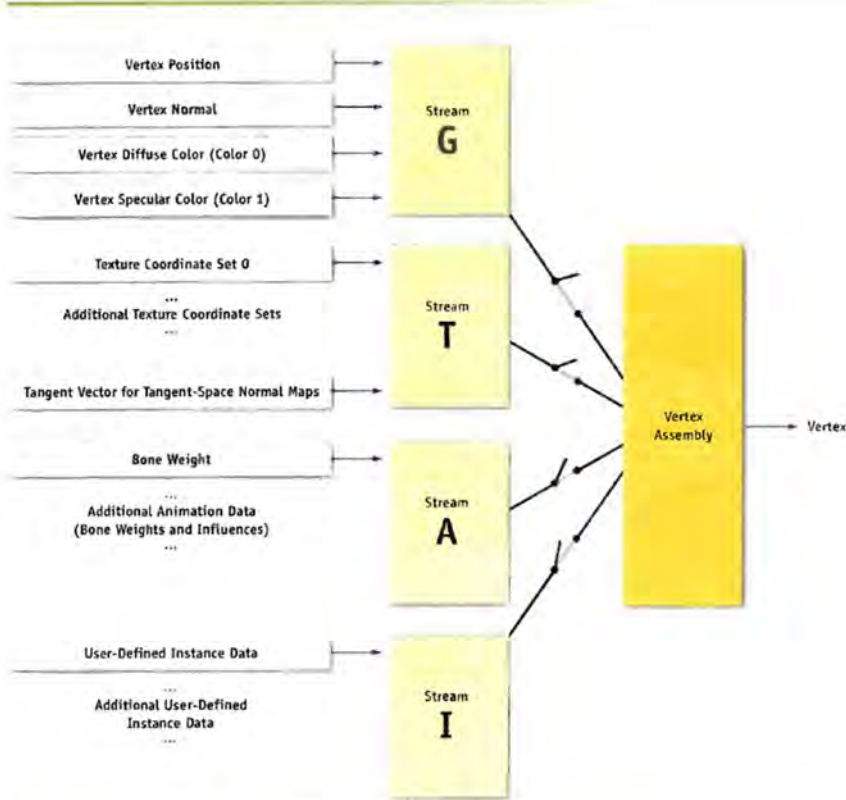
Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
497.	31.2.1 Programmable Parallel Processors GPUs have two types of programmable processors: <i>vertex processors</i> and <i>fragment processors</i>. 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 <i>vertex program</i> (sometimes called a <i>vertex shader</i>) 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 <i>fragments</i> 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 <i>fragment program</i> (sometimes called a <i>pixel shader</i>) to each fragment in the stream to compute the final color of each pixel. Vertex Processors 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. Because of this, vertex processors have hardware to process four-component vectors. This allows them to produce transformed vertex positions in fewer cycles. 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, 31.2 An Inventory of GPU Computational Resources 497

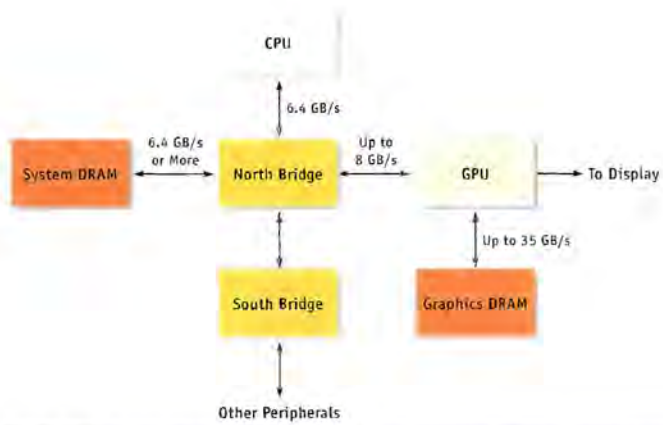
Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
	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 ’461 patent in a manner that would result in the subject matter of this limitation given, at the very least, the person's understanding of the state of the art, the problems addressed and solved in the prior art, and the teachings of GPU Gems. Specific examples of such references are cited and explained in these invalidity contentions, including the cover pleading and charts C1–C15, and Appendix C.
31. The method of claim 21, further comprising: executing, by the at least one CPU, a data output stream, the data output stream controlling transfer of outputs of the computations representing the artificial neural network to disk.	GPU Gems discloses, expressly and/or inherently, the method of claim 21, further comprising: executing, by the at least one CPU, a data output stream, the data output stream controlling transfer of outputs of the computations representing the artificial neural network to disk. <i>See claim 21.</i> <i>See e.g.:</i> <i>As a non-limiting example, GPU Gems discloses a data output stream controlling transfer of outputs of the computations to disk.</i>

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
	 Figure 5-2. Combining Currently Needed Streams 5.2.1 Multistreaming with DirectX 9.0 Microsoft DirectX 8.0 introduced the notion of a stream to bind data to input registers. A stream is a uniform array of component data. Each component consists of one or more elements that represent a single entity, such as position, normal, color, texture coordinate set, and so on. With streams, GPUs are able to perform a direct memory access (DMA) from multiple vertex buffers in parallel. The DirectX9 caps bit 78.

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
	Table 30-1 reiterates some of the points made in the preceding chapter: there is a vast amount of bandwidth available internally on the GPU. Algorithms that run on the GPU can therefore take advantage of this bandwidth to achieve dramatic performance improvements. 30.2 Overall System Architecture 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.  Figure 30-2. The Overall System Architecture of a PC 30.2.1 Functional Block Diagram for Graphics Operations 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. 30.2 Overall System Architecture 473

473.

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461

GPU Gems

Table 31-1. Floating-Point Formats Currently Supported by NVIDIA and ATI GPUs

Name	Sign	Exponent	Mantissa	Largest Values	Smallest Values	Whole Number Range ¹	Supports Specials (NaN, Inf, etc.)
NVIDIA 16-bit	15 (1)	14:10 (5)	9:0 (10)	$\pm 65,504$	$\pm 2^{-14} \approx 10^{-4}$ ($\pm 2^{-24}$ with denorms)	± 2048	Yes
ATI 16-bit	15 (1)	14:10 (5)	9:0 (10)	$\pm 131,008$	$\pm 2^{-14} \approx 10^{-4}$	± 2048	No
ATI 24-bit	23 (1)	22:16 (7)	15:0 (16)	$\pm \sim 2^{24} \approx 10^{16}$	$\pm \sim 2^{-62} \approx 10^{-19}$	$\pm 131,072$	No
NVIDIA 32-bit (IEEE 754)	31 (1)	30:23 (8)	22:0 (23)	$\pm \sim 2^{128} \approx 10^{38}$	$\pm \sim 2^{-126} \approx 10^{-38}$	$\pm 16,777,216$	Yes

1. This is the contiguous zero-centered range of exactly representable whole numbers.

31.3 CPU-GPU Analogies

Even for expert CPU programmers, getting started in GPU programming can be tricky without some knowledge of graphics programming. In this section, we try to aid your understanding by drawing some very simple analogies between traditional CPU computational concepts and their GPU counterparts. We start with the concept of streams and kernels.

31.3.1 Streams: GPU Textures = CPU Arrays

This one is easy. The fundamental array data structures on GPUs are textures and vertex arrays. As we observed before, fragment processors tend to be more useful for GPGPU than vertex processors. Therefore, anywhere we would use an array of data on the CPU, we can use a texture on the GPU.

31.3.2 Kernels: GPU Fragment Programs = CPU “Inner Loops”

The many parallel processors of a GPU are its computational workhorses—they perform the kernel computation on data streams. On the CPU, we would use a loop to iterate over the elements of a stream (stored in an array), processing them sequentially. In the CPU case, the instructions inside the loop are the kernel. On the GPU, we write similar instructions inside a fragment program, which are applied to all elements of the stream. The amount of parallelism in this computation depends on the number of processors on the GPU we use, but also on how well we exploit the instruction-level parallelism enabled by the four-vector structure of GPU arithmetic. Note that vertex programs can also be thought of as kernels operating on a stream of vertices.

500 Chapter 31 Mapping Computational Concepts to GPUs

500.

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
501.	31.3.3 Render-to-Texture = Feedback As mentioned before, most computations are broken into steps. Each step depends on the output of previous steps. In terms of streams, typically a kernel must process an entire stream before the next kernel can proceed, due to dependencies between stream elements. Also, in the case of physically based simulation, each time step of the simulation depends on the results of the previous time step. All of this feedback is trivial to implement on the CPU because of its unified memory model, in which memory can be read or written anywhere in a program. Things aren't so easy on the GPU, as we discussed before. To achieve feedback, we must use render-to-texture to write the results of a fragment program to memory so they can then be used as input to future programs. 31.3.4 Geometry Rasterization = Computation Invocation Now we have analogies for data representation, computation, and feedback. To run a program, though, we need to know how to invoke computation. Our kernels are fragment programs, so all we need to know is how to generate streams of fragments. This should be clear from the previous section; to invoke computation, we just draw geometry. The vertex processors will transform the geometry, and the rasterizer will determine which pixels in the output buffer it covers and generate a fragment for each one. In GPGPU, we are typically processing every element of a rectangular stream of fragments representing a grid. Therefore, the most common invocation in GPGPU programming is a single quadrilateral. 31.3.5 Texture Coordinates = Computational Domain Each kernel (fragment program) that executes on the GPU takes a number of streams as input and typically generates one stream of output. Newer GPUs that support multiple render targets can generate multiple output streams (currently limited to four RGBA streams). Any computation has an input domain and an output range. In many cases, the domain of a computation on the GPU may have different dimensions than the input streams. GPUs provide a simple way to deal with this, in the form of texture coordinates. These coordinates are stored at vertices, and the rasterizer linearly interpolates the coordinates at each vertex to generate a set of coordinates for each fragment. The interpolated 31.3 CPU-GPU Analogies 501

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
	documentation included with the Cg distribution (NVIDIA 2004). There are very few graphics-specific keywords in Cg. We provide a Cg file, <code>pug.cg</code>, that you can include in your kernel files. This file defines a structure called <code>Stream</code> that abstracts the most graphics-centric concept: texture fetching. <code>Stream</code> is a wrapper for the texture sampler Cg type. Calling one of the <code>value()</code> functions of the <code>Stream</code> structure gives the value of a stream element. 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. Stream Management Arrays of data in the framework are called <i>buffers</i>. 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 <code>mode</code> 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. To load initial data into a buffer, pass the data to the <code>pugInitBuffer()</code> function. 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. 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. 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. 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 <code>range</code> parameter of the <code>pugRunProgram()</code> function. 504 Chapter 31 Mapping Computational Concepts to GPUs 504.

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
	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. 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 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. 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>. 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. 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>. 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. 31.5 A Simple Example 505 505.

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
	33.2 The GPU Memory Model Graphics processors have their own memory hierarchy analogous to the one used by serial microprocessors, including main memory, caches, and registers. This memory hierarchy, however, is designed for accelerating graphics operations that fit into the streaming programming model rather than general, serial computation. Moreover, graphics APIs such as OpenGL and Direct3D further limit the use of this memory to graphics-specific primitives such as vertices, textures, and frame buffers. This section gives an overview of the current memory model on GPUs and how stream-based computation fits into it. 33.2.1 Memory Hierarchy 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. <pre>graph TD Disk --> CPU_Main_Memory[CPU Main Memory] CPU_Main_Memory --> CPU_Caches[CPU Caches] CPU_Main_Memory --> GPU_Video_Memory[GPU Video Memory] CPU_Main_Memory --> GPU_Constant_Registers[GPU Constant Registers] CPU_Caches --> CPU_Registers[CPU Registers] GPU_Video_Memory --> GPU_Caches[GPU Caches] GPU_Caches --> GPU_Temporary_Registers[GPU Temporary Registers] GPU_Constant_Registers --- GPU_Temporary_Registers subgraph CPU_Box [CPU] CPU_Caches CPU_Registers end subgraph GPU_Box [GPU] GPU_Constant_Registers GPU_Temporary_Registers end</pre> Figure 33-2. The CPU and GPU Memory Hierarchies 524 Chapter 33 Implementing Efficient Parallel Data Structures on GPUs 524.

Ex. C11 – Invalidity of U.S. Patent No. RE49,461 – GPU Gems

U.S. Patent No. RE49,461	GPU Gems
	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 ’461 patent in a manner that would result in the subject matter of this limitation given, at the very least, the person's understanding of the state of the art, the problems addressed and solved in the prior art, and the teachings of GPU Gems. Specific examples of such references are cited and explained in these invalidity contentions, including the cover pleading and charts C1–C15, and Appendix C.