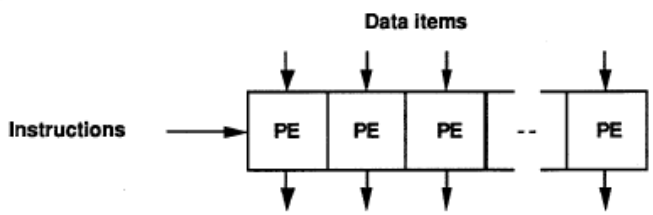
ADDITIONAL REFERENCES AND OBVIOUSNESS CHART FOR U.S. PATENT NO. RE48,438

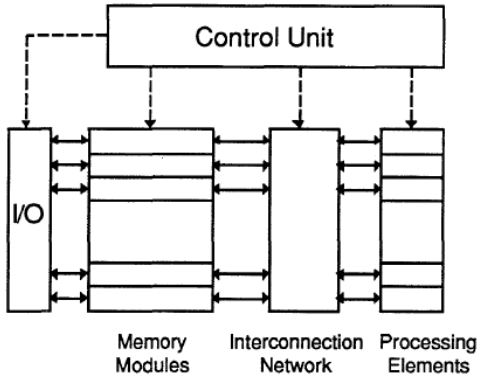
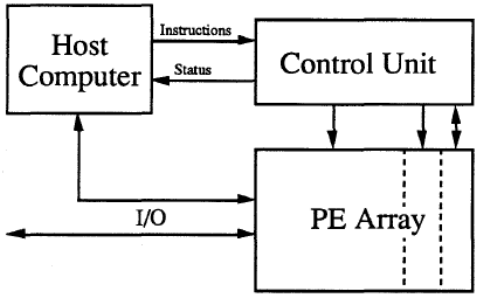
For the reasons explained in more detail in the cover pleading to NVIDIA’s Invalidity Contentions, to the extent a primary reference does not disclose one or more of the elements of the asserted claims, a person of ordinary skill in the art would be motivated to combine the primary reference with any other primary reference(s) that disclose that limitation or with the knowledge of one of ordinary skill in the art, including admitted prior art, and/or one or more of the references listed in the charts below or in the cover pleading to NVIDIA’s Invalidity Contentions. For clarity, obviousness grounds for each dependent claim incorporate obviousness grounds for the claim(s) from which it depends in addition to any obviousness grounds identified in the charts for the dependent claim. References to figures, drawings, and the like refer to the cited items as well as any accompanying or related text.

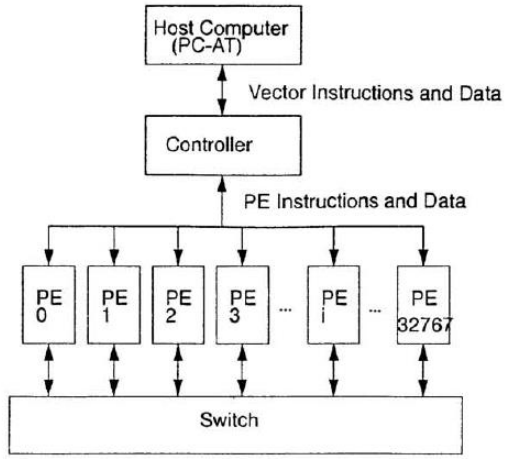
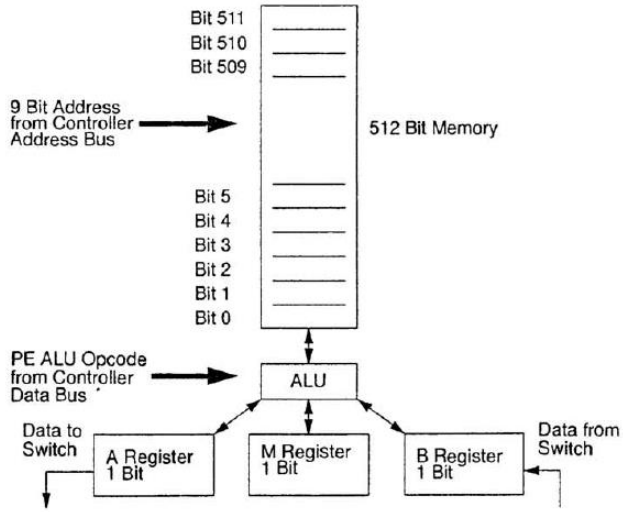
The prior art references included below are the following:

- Bertil Svensson, SIMD Processor Array Architectures in PARALLEL PROCESSING IN INDUSTRIAL REAL-TIME APPLICATIONS (1992) (“Svensson”)
- Christopher Davis, Graphics Processing Unit Computation of Neural Networks (2005) (“Davis Thesis”)
- Francisco Mesa-Martinez, The UCSC Kestrel High Performance SIMD Processor: Present and Future (2001) (“Mesa-Martinez”)
- GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation (2005) (“GPU Gems”)
- Japanese Unexamined Patent Appl. No. H04-237388A (“Tamura”)
- K. Jung, Neural network-based text location in color images (2001) (“Jung”)
- K. Oh, GPU implementation of neural networks (2004) (“Oh”)
- Mark Harris, Real-Time Cloud Simulation and Rendering (2003) (“Harris Dissertation”)
- Michael Glover, A Massively-Parallel SIMD Processor for Neural Network and Machine Vision Applications (1993) (“Glover”)
- The C Programming Language (1988)
- U.K. Patent Application No. GB2291553A (“Wilson”)
- U.S. Patent No. 7,139,003 (“Kirk”)
- Z. Luo, Artificial Neural Network Computation on Graphic Process Unit (IEEE 2005) (“ANN”)
- A Survey of General-Purpose Computation on Graphics Hardware, 2005 (“Owens”)
- OpenVIDIA: Parallel GPU Computer Vision, 2005 (“OpenVIDIA”)

Asserted Claims	Additional References and Disclosure
1[pre] “A computer system, comprising:”	*As a non-limiting example, Svensson discloses a computer system (e.g., host computer and attached processing array):*

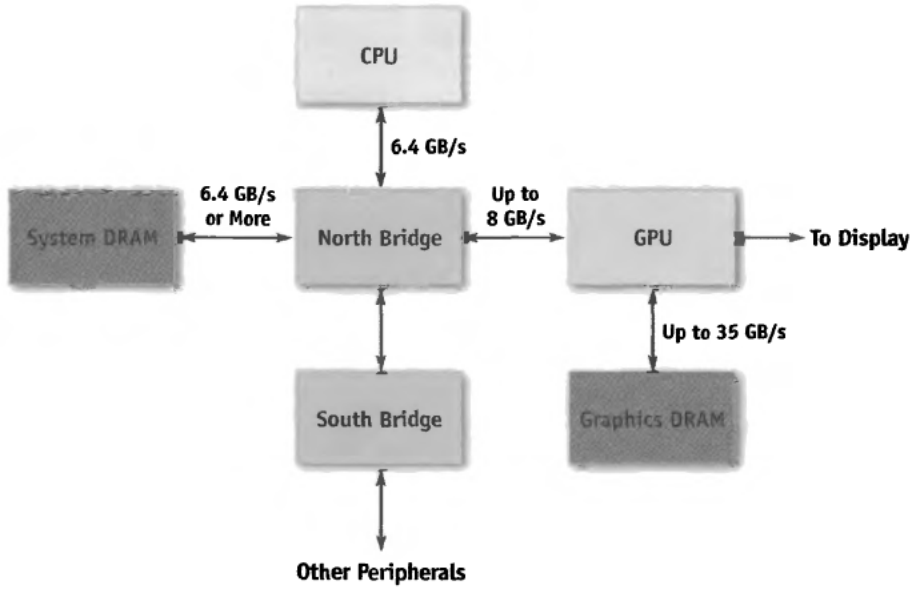
Asserted Claims	Additional References and Disclosure
	1. Principles of SIMD In Single Instruction stream, Multiple Data stream (SIMD) processors one instruction works on several data items simultaneously by using several Processing Elements (PEs), all of which carry out the same operation as illustrated in Figure SIMD-1.  Principle of SIMD processor. Figure SIMD-1. A SIMD Processor has a single Control Unit reading instructions pointed to by a single Program Counter, decoding them and sending control signals to the PEs. Data are to be supplied to, and derived from, the PEs by a memory with as many data paths as there are PEs. Figure SIMD-2a shows the resulting processing structure which is also known as a Processor Array. The Interconnection Network provides flexibility in choosing source and destination for data to and from the PEs, necessary in many algorithms. The I/O system plays the role of converting – typically at very high rates – input/output data between the format of the outside world and the internal format of the array. The design of the I/O system is highly application dependent. In order to provide overall control of the Processor Array, as well as to execute sequential operations, it is common to use a conventional Host Computer which then views the Processor Array as an attached processor as illustrated in figure SIMD-2b. 1.

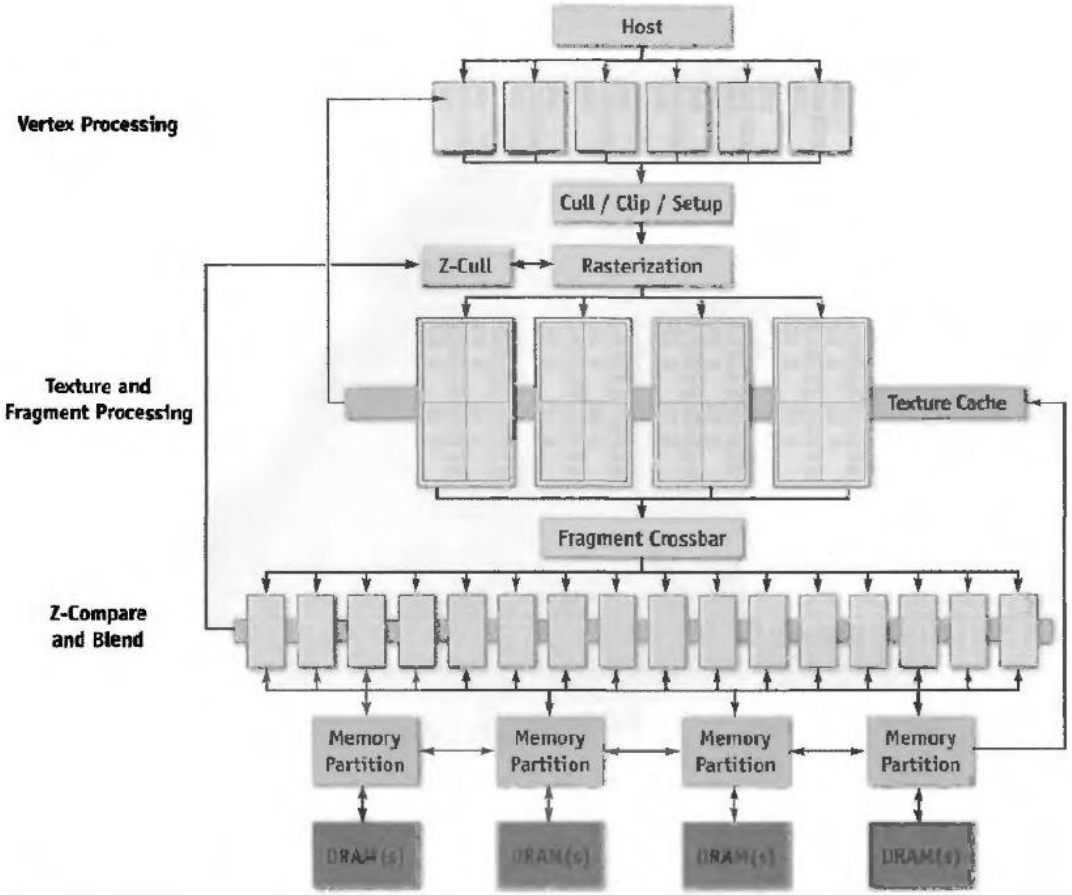
Asserted Claims	Additional References and Disclosure
	 Processor Array Figure SIMD-2a  Host Computer - Processor Array Relationship Figure SIMD-2b 2. *As a non-limiting example, Glover discloses a computer system (e.g., the MM32k SIMD computer):*

Asserted Claims	Additional References and Disclosure
	 Figure 1: A block diagram of the MM32k.  Figure 2: A block diagram of a single processing element (PE). 845.

Asserted Claims	Additional References and Disclosure
	As a non-limiting example, GPU Gems discloses a computer system (e.g., an overall computer system and/or a PC): The previous chapter described how GPU architecture has changed as a result of computational and communications trends in microprocessing. This chapter describes the architecture of the GeForce 6 Series GPUs from NVIDIA, which owe their formidable computational power to their ability to take advantage of these trends. Most notably, we focus on the GeForce 6800 (NVIDIA's flagship GPU at the time of writing, shown in Figure 30-1), which delivers hundreds of gigaflops of single-precision floating-point computation, as compared to approximately 12 gigaflops for current high-end CPUs. In this chapter—and throughout the book—references to GeForce 6 Series GPUs should be read to include the latest Quadro FX GPUs supporting Shader Model 3.0, which provide a superset of the functionality offered by the GeForce 6 Series. We start with a general overview of where the GPU fits into the overall computer system, and then we describe the architecture along with details of specific features and performance characteristics. **30.1 How the GPU Fits into the Overall Computer System** The CPU in a modern computer system communicates with the GPU through a graphics connector such as a PCI Express or AGP slot on the motherboard. Because the graphics connector is responsible for transferring all command, texture, and vertex data from the CPU to the GPU, the bus technology has evolved alongside GPUs over the past few years. The original AGP slot ran at 66 MHz and was 32 bits wide, giving a transfer rate of 264 MB/sec. AGP 2×, 4×, and 8× followed, each doubling the available

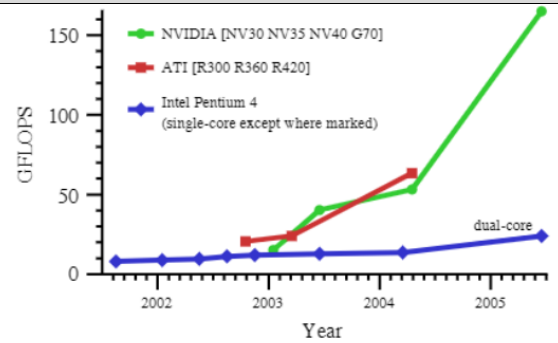
Asserted Claims	Additional References and Disclosure
	bandwidth, until finally the PCI Express standard was introduced in 2004, with a maximum theoretical bandwidth of 4 GB/sec simultaneously available to and from the GPU. (Your mileage may vary; currently available motherboard chipsets fall somewhat below this limit—around 3.2 GB/sec or less.) It is important to note the vast differences between the GPU's memory interface bandwidth and bandwidth in other parts of the system, as shown in Table 30-1. **Table 30-1.** Available Memory Bandwidth in Different Parts of the Computer System Component — Bandwidth GPU Memory Interface — 35 GB/sec PCI Express Bus (×16) — 8 GB/sec CPU Memory Interface (800 MHz Front-Side Bus) — 6.4 GB/sec 471–472.

Asserted Claims	Additional References and Disclosure
	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 473.

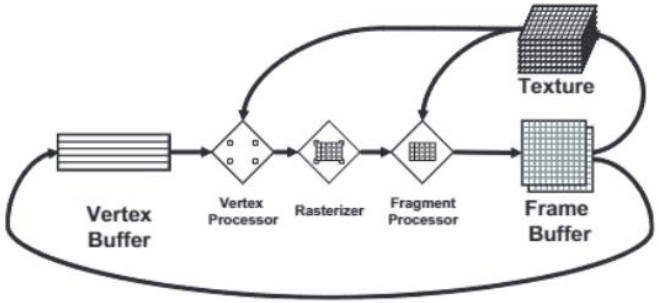
Asserted Claims	Additional References and Disclosure
	 **Figure 30-3. A Block Diagram of the GeForce 6 Series Architecture** 474. *As a non-limiting example, Owens discloses a computer system including a central processing unit (CPU) and an accelerator (GPU) that performs a numerical simulation (e.g., simulating artificial neural networks and performing computer vision and image processing tasks).*

Asserted Claims	Additional References and Disclosure
	Commodity computer graphics chips are probably today’s most powerful computational hardware for the dollar. These chips, known generically as Graphics Processing Units or GPUs, have gone from afterthought peripherals to modern, powerful, and programmable processors in their own right. Many researchers and developers have become interested in harnessing the power of commodity graphics hardware for general-purpose computing. Recent years have seen an explosion in interest in such research efforts, known collectively as GPGPU (for “General Purpose GPU”) computing. In this State of the Art Report we summarize the motivation and essential developments in the hardware and software behind GPGPU. We give an overview of the techniques and computational building blocks used to map general-purpose computation to graphics hardware and provide a survey of the various general-purpose computing applications to which GPUs have been applied.

Asserted Claims	Additional References and Disclosure
	1.1. Powerful and Inexpensive Recent graphics architectures provide tremendous memory bandwidth and computational horsepower. For example, the NVIDIA GeForce 6800 Ultra ($417 as of June 2005) can achieve a sustained 35.2 GB/sec of memory bandwidth; the ATI X800 XT ($447) can sustain over 63 GFLOPS (compare to 14.8 GFLOPS theoretical peak for a 3.7 GHz Intel Pentium4 SSE unit [Buc04]). GPUs are also on the cutting edge of processor technology; for example, the most recently announced GPU at this writing contains over 300 million transistors and is built on a 110-nanometer fabrication process. Not only is current graphics hardware fast, it is accelerating quickly. For example, the measured throughput of the GeForce 6800 is more than double that of the GeForce 5900, NVIDIA's previous flagship architecture. In general, the computational capabilities of GPUs, measured by the traditional metrics of graphics performance, have compounded at an average yearly rate of 1.7× (pixels/second) to 2.3× (vertices/second). This rate of growth outpaces the oft-quoted Moore's Law as applied to traditional microprocessors; compare to a yearly rate of roughly 1.4× for CPU perfor-

Asserted Claims	Additional References and Disclosure
	 **Figure 1:** *The programmable floating-point performance of GPUs (measured on the multiply-add instruction as 2 floating-point operations per MAD) has increased dramatically over the last four years when compared to CPUs. Figure courtesy Ian Buck, Stanford University.* mance [EWN05]. Put another way, graphics hardware performance is roughly doubling every six months (Figure 1).

Asserted Claims	Additional References and Disclosure
	Modern graphics architectures have become flexible as well as powerful. Once fixed-function pipelines capable of outputting only 8-bit-per-channel color values, modern GPUs include fully programmable processing units that support vectorized floating-point operations at full IEEE single precision. High level languages have emerged to support the new programmability of the vertex and pixel pipelines [BFH*04, MGAK03, MTP*04]. Furthermore, additional levels of programmability are emerging with every major generation of GPU (roughly every 18 months). Examples of major architectural changes in the current generation (as of this writing) GPUs include vertex texture access, full branching support in the vertex pipeline, and limited branching capability in the fragment pipeline. The next generation is expected to expand on these changes and add “geometry shaders”, or programmable primitive assembly, bringing flexibility to an entirely new stage in the pipeline. In short, the raw speed, increased precision, and rapidly expanding programmability of the hardware make it an attractive platform for general-purpose computation. Owens, 1–2.

Asserted Claims	Additional References and Disclosure
	 **Figure 2:** *The modern graphics hardware pipeline. The vertex and fragment processor stages are both programmable by the user.*

Asserted Claims	Additional References and Disclosure
	Each new generation of GPUs has increased the functionality and generality of these two programmable stages. 1999 marked the introduction of the first programmable stage, NVIDIA's register combiner operations that allowed a limited combination of texture and interpolated color values to compute a fragment color. In 2002, ATI's Radeon 9700 led the transition to floating-point computation in the fragment pipeline. The vital step for enabling general-purpose computation on GPUs was the introduction of fully programmable hardware and an assembly language for specifying programs to run on each vertex [LKM01] or fragment. This programmable shader hardware is explicitly designed to process multiple data-parallel primitives at the same time. As of 2005, the vertex shader and pixel shader standards are both in their third revision, and the OpenGL Architecture Review Board maintains extensions for both [Ope04, Ope03]. The instruction sets of each stage are limited compared to CPU instruction sets; they are primarily math operations, many of which are graphics-specific. The newest addition to the instruction sets of these stages has been limited control flow operations. Owens, 3–4.

	Because typical scenes have more fragments than vertices, in modern GPUs the programmable stage with the highest arithmetic rates is the fragment processor. A typical GPGPU program uses the fragment processor as the computation engine in the GPU. Such a program is structured as follows [Har05a]: 1. First, the programmer determines the data-parallel portions of his application. The application must be segmented into independent parallel sections. Each of these sections can be considered a *kernel* and is implemented as a fragment program. The input and output of each kernel program is one or more data arrays, which are stored (sometimes only transiently) in textures in GPU memory. In stream processing terms, the data in the textures comprise *streams*, and a kernel is invoked in parallel on each stream element. 2. To invoke a kernel, the range of the computation (or the size of the output stream) must be specified. The programmer does this by passing vertices to the GPU. A typical GPGPU invocation is a quadrilateral (quad) oriented parallel to the image plane, sized to cover a rectangular region of pixels matching the desired size of the output array. Note that GPUs excel at processing data in two-dimensional arrays, but are limited when processing one-dimensional arrays. 3. The rasterizer generates a fragment for every pixel location in the quad, producing thousands to millions of fragments. 4. Each of the generated fragments is then processed by the active kernel fragment program. Note that every fragment is processed by the same fragment program. The fragment program can read from arbitrary global memory locations (with texture reads) but can only write to memory locations corresponding to the location of the	

Asserted Claims	Additional References and Disclosure
	fragment in the frame buffer (as determined by the rasterizer). The domain of the computation is specified for each input texture (stream) by specifying texture coordinates at each of the input vertices, which are then interpolated at each generated fragment. Texture coordinates can be specified independently for each input texture, and can also be computed on the fly in the fragment program, allowing arbitrary memory addressing. 5. The output of the fragment program is a value (or vector of values) per fragment. This output may be the final result of the application, or it may be stored as a texture and then used in additional computations. Complex applications may require several or even dozens of passes (“multipass”) through the pipeline. While the complexity of a single pass through the pipeline may be limited (for example, by the number of instructions, by the number of outputs allowed per pass, or by the limited control complexity allowed in a single pass), using multiple passes allows the implementation of programs of arbitrary complexity. For example, Peercy et al. [POAU00] demonstrated that even the fixed-function pipeline, given enough passes, can implement arbitrary RenderMan shaders.

	2.4. GPU Program Flow Control Flow control is a fundamental concept in computation. Branching and looping are such basic concepts that it can be daunting to write software for a platform that supports them to only a limited extent. 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 section surveys some of the limitations of branching on current GPUs and describes a variety of techniques for iteration and decision-making in GPGPU programs. For more detail on GPU flow control, see Harris and Buck [HB05]. **2.4.1. Hardware Mechanisms for Flow Control** There are three basic implementations of data-parallel branching in use on current GPUs: predication, MIMD branching, and SIMD branching. Architectures that support only predication do not have true data-dependent branch instructions. Instead, the GPU 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 predication is that evaluating both sides of the branch can be costly, but not all current GPUs have true data-dependent branching support. The compiler for high-level shading languages like Cg or the OpenGL Shading Language automatically generates predicated assembly language instructions if the target GPU supports only predication for flow control. In Multiple Instruction Multiple Data (MIMD) architectures that support branching, different processors can follow different paths through the program. In Single Instruction	

Asserted Claims	Additional References and Disclosure
	Multiple Data (SIMD) architectures, all active processors must execute the same instructions at the same time. The only MIMD processors in a current GPU are the vertex processors of the NVIDIA GeForce 6 and NV40 Quadro GPUs. All current GPU fragment processors are SIMD. In SIMD, when evaluation of the branch condition is identical on all active processors, only the taken side of the branch must be evaluated, but if one or more of the processors evaluates the branch condition differently, then both sides must be evaluated and the results predicated. As a result, divergence in the branching of simultaneously processed fragments can lead to reduced performance. **2.4.2. Moving Branching Up The Pipeline** Because explicit branching can hamper performance on GPUs, it is useful to have multiple techniques to reduce the cost of branching. A useful strategy is to move flow-control decisions up the pipeline to an earlier stage where they can be more efficiently evaluated.

	2.4.2.1. Static Branch Resolution On the GPU, as on the CPU, avoiding branching inside inner loops is beneficial. For example, when evaluating a partial differential equation (PDE) on a discrete spatial grid, an efficient 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 *static branch resolution* results in loops that contain efficient code without branches. (In stream processing terminology, this technique is typically referred to as the division of a stream into *substreams*.) On the GPU, 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 one-pixel edge of the output buffer. The boundary program is applied to fragments of lines drawn over the edge pixels. Static branch resolution is further discussed by Goodnight et al. [GWL*03], Harris and James [HJ03], and Lefohn et al. [LKHW03]. **2.4.2.2. Pre-computation** In the example above, 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 are known to change, and store the results for use over many subsequent iterations. This can result in a large performance boost. This technique is used to pre-compute an obstacle offset array in the Navier-Stokes fluid simulation example in the NVIDIA SDK [Har05b]. **2.4.2.3. Z-Cull** Precomputed branch results can be taken a step further by using 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	

Asserted Claims	Additional References and Disclosure
	Owens, 4–5.

Asserted Claims	Additional References and Disclosure
	3.1. High-level Shading Languages Most high-level GPU programming languages today share one thing in common: they are designed around the idea that GPUs generate pictures. As such, the high-level programming languages are often referred to as shading languages. That is, they are a high-level language that compiles into a vertex shader and a fragment shader to produce the image described by the program. Cg [MGAK03], HLSL [Mic05a], and the OpenGL Shading Language [KBR04] all abstract the capabilities of the underlying GPU and allow the programmer to write GPU programs in a more familiar C-like programming language. They do not stray far from their origins as languages designed to shade polygons. All retain graphics-specific constructs: vertices, fragments, textures, etc. Cg and HLSL provide abstractions that are very close to the hardware, with instruction sets that expand as the underlying hardware capabilities expand. The OpenGL Shading Language was designed looking a bit further out, with many language features (e.g. integers) that do not directly map to hardware available today. Sh is a shading language implemented on top of C++ [MTP*04]. Sh provides a shader algebra for manipulating and defining procedurally parameterized shaders. Sh manages buffers and textures, and handles shader partitioning into multiple passes. Finally, Ashli [BP03] works at a level one step above that of Cg, HLSL, or the OpenGL Shading Language. Ashli reads as input shaders written in HLSL, the OpenGL Shading Language, or a subset of RenderMan. Ashli then automatically compiles and partitions the input shaders to run on a programmable GPU.

Asserted Claims	Additional References and Disclosure
	3.2. GPGPU Languages and Libraries More often than not, the graphics-centric nature of shading languages makes GPGPU programming more difficult than it needs to be. As a simple example, initiating a GPGPU computation usually involves drawing a primitive. Looking up data from memory is done by issuing a texture fetch. The GPGPU program may conceptually have nothing to do with drawing geometric primitives and fetching textures, yet the shading languages described in the previous section force the GPGPU application writer to think in terms of geometric primitives, fragments, and textures. Instead, GPGPU algorithms are often best described as memory and math operations, concepts much more familiar to CPU programmers. The programming systems below attempt to provide GPGPU functionality while hiding the GPU-specific details from the programmer. The Brook programming language extends ANSI C with concepts from stream programming [BFH*04]. Brook can use the GPU as a compilation target. Brook streams are con-

Asserted Claims	Additional References and Disclosure
	ceptually similar to arrays, except all elements can be operated on in parallel. Kernels are the functions that operate on streams. Brook automatically maps kernels and streams into fragment programs and texture memory. Scout is a GPU programming language designed for scientific visualization [MIA*04]. Scout allows runtime mapping of mathematical operations over data sets for visualization. Finally, the Glift template library provides a generic template library for a wide range of GPU data structures [LKS*05]. It is designed to be a stand-alone GPU data structure library that helps simplify data structure design and separate GPU algorithms from data structures. The library integrates with a C++, Cg, and OpenGL GPU development environment. Owens, 6–7.

Asserted Claims	Additional References and Disclosure
	The wide deployment of GPUs in the last several years has resulted in an increase in experimental research with graphics hardware. The earliest work on desktop graphics processors used non-programmable (“fixed-function”) GPUs. Lengyel et al. used rasterization hardware for robot motion planning [LRDG90]. Hoff et al. described the use of z-buffer techniques for the computation of Voronoi diagrams [HCK*99] and extended their method to proximity detection [HZLM01]. Bohn et al. used fixed-function graphics hardware in the computation of artificial neural networks [Boh98]. Convolution and wavelet transforms with the fixed-function pipeline were realized by Hopf and Ertl [HE99a, HE99b]. **5.2. Physically-Based Simulation** **5.3. Signal and Image Processing** **5.3.2. Other Signal and Image Processing Applications** **Computer Vision** Fung et al. use graphics hardware to accelerate image projection and compositing operations [FTM02] in a camera-based head-tracking system [FM04]; their implementation has been released as the open-source OpenVIDIA computer vision library [Ope05], whose website also features a good bibliography of papers for GPU-based computer/machine vision applications. Owens, 16–17.

Asserted Claims	Additional References and Disclosure
	As a non-limiting example, OpenVIDIA discloses a computer system including a central processing unit and an accelerator (GPU) that performs a numerical simulation (e.g., computer vision and image processing). **ABSTRACT** Graphics and vision are approximate inverses of each other: ordinarily Graphics Processing Units (GPUs) are used to convert “numbers into pictures” (i,e. computer graphics). In this paper, we propose using GPUs in approximately the reverse way: to assist in “converting pictures into numbers” (i.e. computer vision). The OpenVIDIA project uses single or multiple graphics cards to accelerate image analysis and computer vision. It is a library and API aimed at providing a graphics hardware accelerated processing framework for image processing and computer vision. OpenVIDIA explores the creation of a parallel computer architecture consisting of multiple Graphics Processing Units (GPUs) built entirely from commodity hardware. OpenVIDIA uses multiple Graphics Processing Units in parallel to operate as a general-purpose parallel computer architecture. It provides a simple API which implements some common computer vision algorithms. Many components can be used immediately and because the project is Open Source, the code is intended to serve as templates and examples for how similar algorithms are mapped onto graphics hardware. Implemented are image processing techniques (Canny edge detection, filtering), image feature handling (identifying and matching features) and image registration, to name a few.

Asserted Claims	Additional References and Disclosure
	1. INTRODUCTION OpenVIDIA is a programming framework that uses single or multiple graphics cards to operate as a general-purpose parallel computer architecture for fast computer vision and image processing. Realtime image processing and computer vision algorithms can be computationally intensive, exceeding the capabilities of the CPU. OpenVIDIA utilizes the GPU that is present on modern graphics hardware to expand the computational resources that are available to these algorithms. Furthermore, the design of the GPU architecture provides higher performance for many vision algorithms than the CPU. OpenVIDIA, 1.

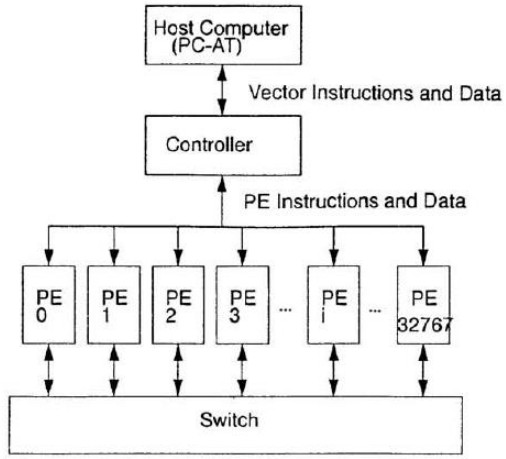
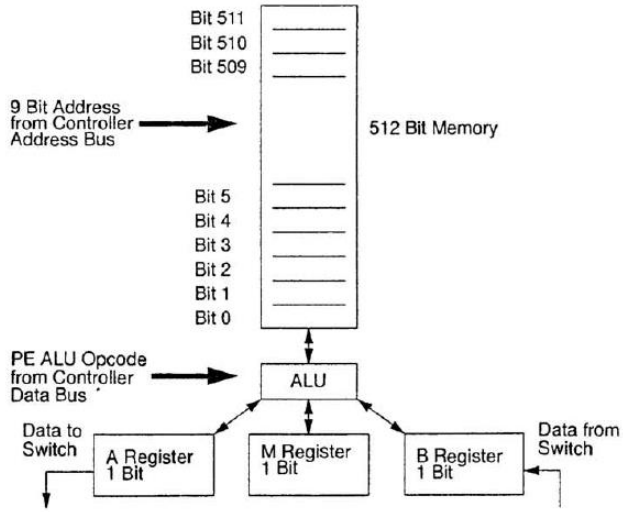
	The GPU has been applied to other calculations beyond graphics. A survey of applications is given by Owens et al [7][1]. This paper will discuss the OpenVIDIA design and API and show some techniques OpenVIDIA uses for GPU computer vision. **2. OPENVIDIA** **2.1 Design and Intent** OpenVIDIA is designed as a library that is called from within user written OpenGL programs. OpenVIDIA provides an API that implements the OpenGL calls that are needed for vision processing. Essentially, this API is implementing a set of mappings of computer vision algorithms onto graphics hardware. For instance, the OpenVIDIA API provides an interface for creating and applying image filters that use fragment shading hardware on the GPU, resulting in hardware accelerated image filtering operations. This paper will discuss both the API of OpenVIDIA as well as the methods it uses to perform some common computer vision tasks on graphics hardware. OpenVIDIA is built with C/C++ and OpenGL. Additionally, it interfaces with libraries common to computer vision tasks. For example, it interfaces with an IEEE 1394 camera interface library, allowing video to be brought into an OpenGL vision processing framework in a simple fashion. The API provides some function calls to run common vision processing algorithms. These can be used “as is” when they suit the application. Alternately, as the project is Open Source, the implementations can serve as template examples, which can then be altered to create similar but more specific functions as needed by users. Included in the OpenVIDIA package are image filtering operations (many vision algorithms include sequences of image filtering operations), feature detection and tracking, image registration, stereo vision, and Hough transformations to name a few.	

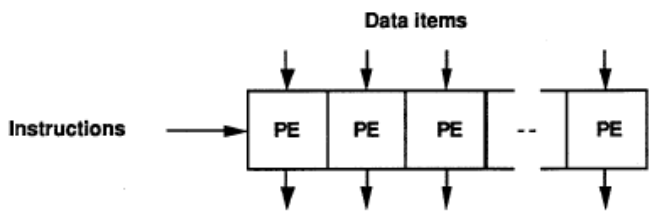
Asserted Claims	Additional References and Disclosure
	3.1 Image Processing and Filtering Many computer vision operations can be considered sequences of filtering operations. Fragment programs can be considered short programs run on each pixel of an image that implement the desired filter. The architecture of the GPU allows many pixels to be processed in a parallel fashion, providing significant hardware acceleration. OpenVIDIA, 2. **3.5 Locating and Tracking Features** OpenVIDIA provides an algorithm that performs image feature detection on the GPU. This algorithm also calculates unique appearance based feature vectors computed on the GPU. These vectors can then be re-calculated for each video image, and matched to track features in video. Figure 3 shows how to use the OpenVIDIA API calls which create the library tracker object, and match features in sequential frames of video. Feature detection in OpenVIDIA also provides an example of how to use the GPU for local histogramming and to create multiple outputs for single points. Feature detection in OpenVIDIA begins using a Harris corner detector, implemented using the filtering pipeline technique discussed in section 3.1. The result of the feature detection stage is essentially a binary image, with feature points flagged. This is then read back to the CPU which places the locations into a 1-D array which is sent back to the GPU as a 1-D texture. The unique feature vectors are then calculated.

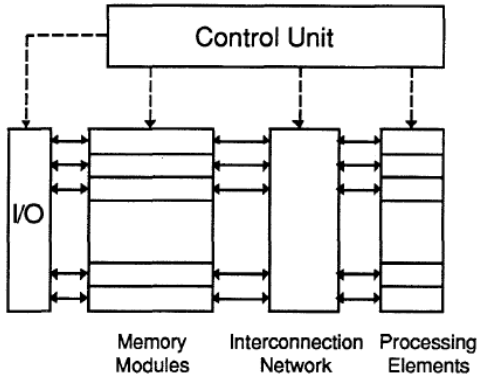
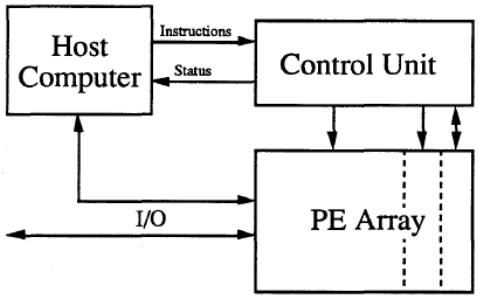
Asserted Claims	Additional References and Disclosure
	OpenVIDIA, 3. [7] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krger, A. E. Lefohn, and T. J. Purcell. A survey of general-purpose computation on graphics hardware. In *Eurographics 2005, State of the Art Reports*, pages 21–51, Aug. 2005. OpenVIDIA, 4.
1[a] “a central processing unit to receive input data;”	*As a non-limiting example, ANN discloses receiving input data (e.g., video data from a camera):* Three layer MLP neural network is selected to recognize the ball. The input layer consists of seven nodes for seven characters. The hidden layer consists of three nodes, and the output layer consists of just one node. Back propagation method was chosen to train the network. The train set takes as follow. 16 locations are selected from each of 10 robot cars, 4 locations are selected from center of ball, and one location is selected from background field. Totally, 165 locations are selected as the train set. For each selected location, seven characters are calculated and take as an element of the train set. The trained MLP is used to recognize and trace the ball in real time. 624.

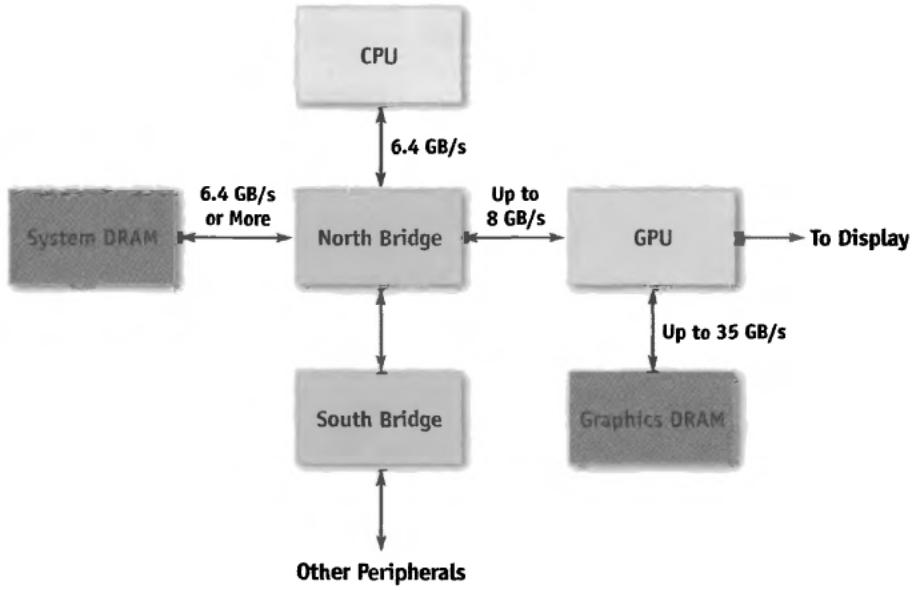
Asserted Claims	Additional References and Disclosure
	B. MLP Computation Time for CPU and GPU The application of MLP in this paper is to trace the ball in robot soccer in real time. The result is shown in table 1. 625. The implementation on graphic hardware introduce in this paper has other implicit benefit too. For the SOM computing, a multi-texture or 3-D texture can be used to store the map and make more general SOM computing without the restriction of the vector length of 4. For the MLP used in robot soccer, some graphic hardware have “video in” function, using this kind of graphic hardware; image information can be retrieved directly from camera and store on graphic memory, and don’t need to transfer data between CPU and GPU, which will speed the process. 626. *As a non-limiting example, Oh discloses receiving input data (e.g., video and image data):* In the case of using a neural network (NN) for image processing and pattern recognition, the main problem is the computational complexity in the testing stage, which accounts for most of the processing time. Moreover, NN-based image convolution has to exhaustively scan an input image in order to process an entire image [3]. Although an NN can be simulated using software, many potential NN applications require real-time processing, which means fully parallel specially designed hardware implementations, such as an FPGA-based realization of an NN. However, this is somewhat expensive and involves extra design overheads [4]. 1311. Recently, researchers have attempted text-based retrieval of image and video data using several image processing techniques [3]. As such, an automatic text detection algorithm for image data and video

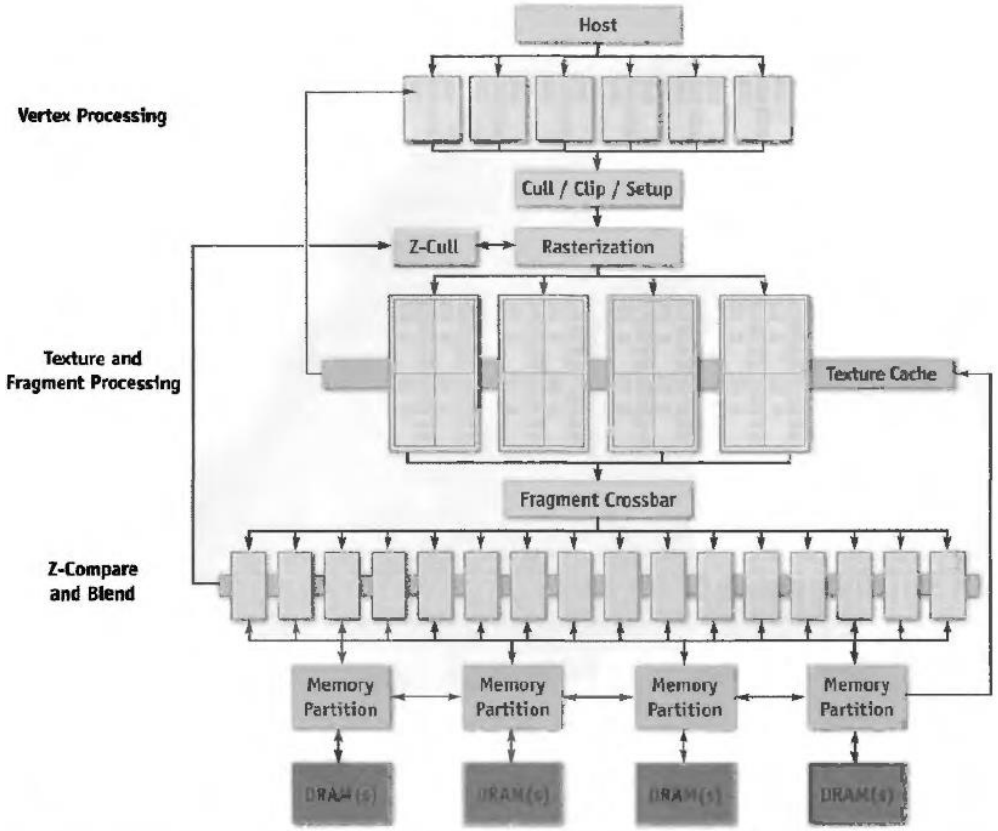
Asserted Claims	Additional References and Disclosure
	documents is important as a preprocessing stage for optical character recognition, and an NN-based text detection method has several advantages over other methods [3]. Therefore, this paper briefly describes such a text detection method, and readers are referred to the author’s previous publication for more details [3]. In the proposed method, an NN is used to classify the pixels of input images, whereby the feature extraction and pattern recognition stage are integrated in the neural network. The NN then examines local regions looking for text pixels that may be contained in a text region. Therefore, an M × M pixel region in the image is received as the input and a classified image is generated as the output. After the pattern passes the network, the value of the output node is compared with a threshold value and the class of each pixel determined, resulting in a classified image. GPU-based pipelining processing is used to reduce the processing time, and the GPU’s performance is maximized by accumulating a large number of input vectors 1 to create a two-dimensional texture. The input layer then receives the grey values for the pixels at predefined positions inside an M×M window over the input image. Experiments were conducted using an 11 × 11 input window size, with the number of nodes in each hidden layer set at 30. As a result, the processing time for pixel classification was significantly reduced using a GPU. Fig. 2(b) shows the pixel classification result for the left input image, where a black pixel denotes a text pixel. The classification using a GPU produced almost the same result as without a GPU. 1313. *As a non-limiting example, Glover discloses a central processing unit (e.g., host computer):*

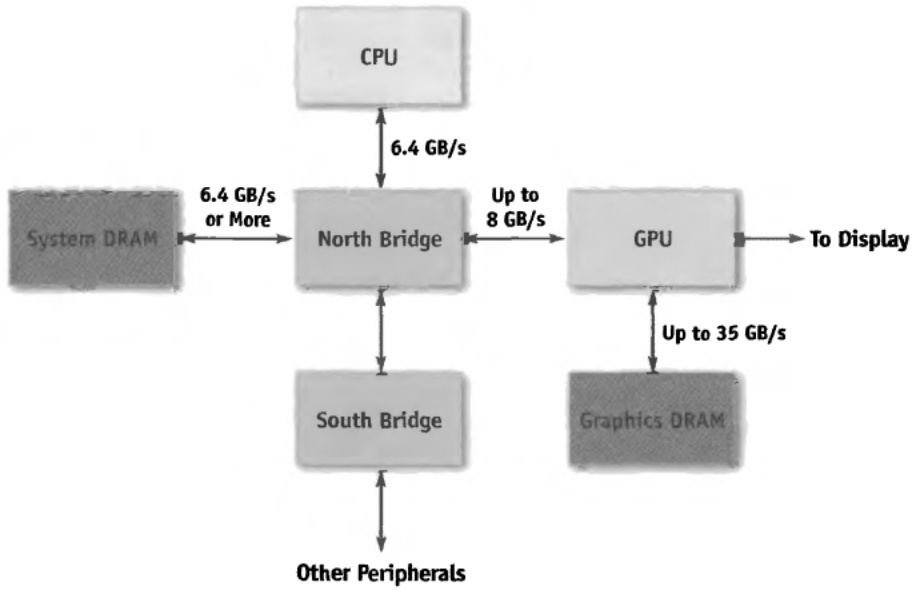
Asserted Claims	Additional References and Disclosure
	 Figure 1: A block diagram of the MM32k.  Figure 2: A block diagram of a single processing element (PE). 845.

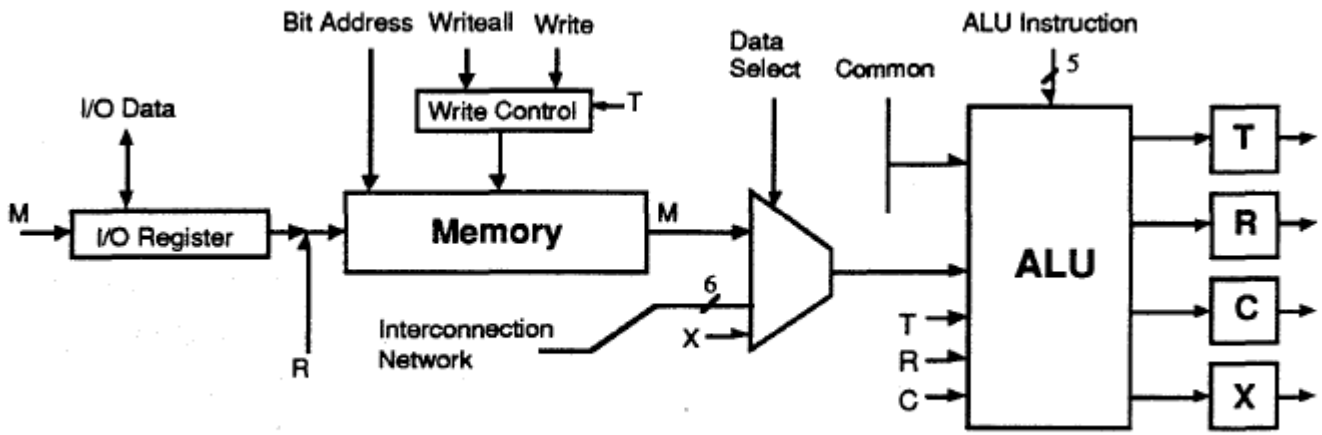
Asserted Claims	Additional References and Disclosure
	As a non-limiting example, Svensson discloses a central processing unit (e.g., host computer): ***1. Principles of SIMD*** In Single Instruction stream, Multiple Data stream (SIMD) processors one instruction works on several data items simultaneously by using several Processing Elements (PEs), all of which carry out the same operation as illustrated in Figure SIMD-1.  Principle of SIMD processor. Figure SIMD-1. A SIMD Processor has a single Control Unit reading instructions pointed to by a single Program Counter, decoding them and sending control signals to the PEs. Data are to be supplied to, and derived from, the PEs by a memory with as many data paths as there are PEs. Figure SIMD-2a shows the resulting processing structure which is also known as a Processor Array. The Interconnection Network provides flexibility in choosing source and destination for data to and from the PEs, necessary in many algorithms. The I/O system plays the role of converting – typically at very high rates – input/output data between the format of the outside world and the internal format of the array. The design of the I/O system is highly application dependent. In order to provide overall control of the Processor Array, as well as to execute sequential operations, it is common to use a conventional Host Computer which then views the Processor Array as an attached processor as illustrated in figure SIMD-2b.

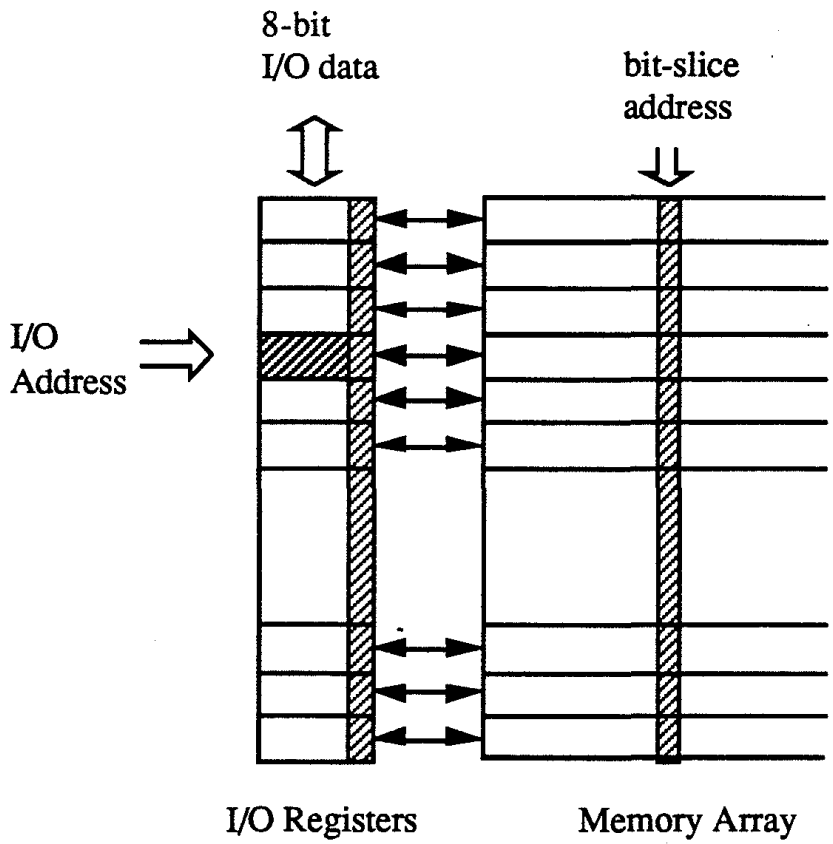
Asserted Claims	Additional References and Disclosure
	1.  Processor Array Figure SIMD-2a  Host Computer - Processor Array Relationship Figure SIMD-2b 2. *As a non-limiting example, GPU Gems discloses a central processing unit (e.g., a CPU):*

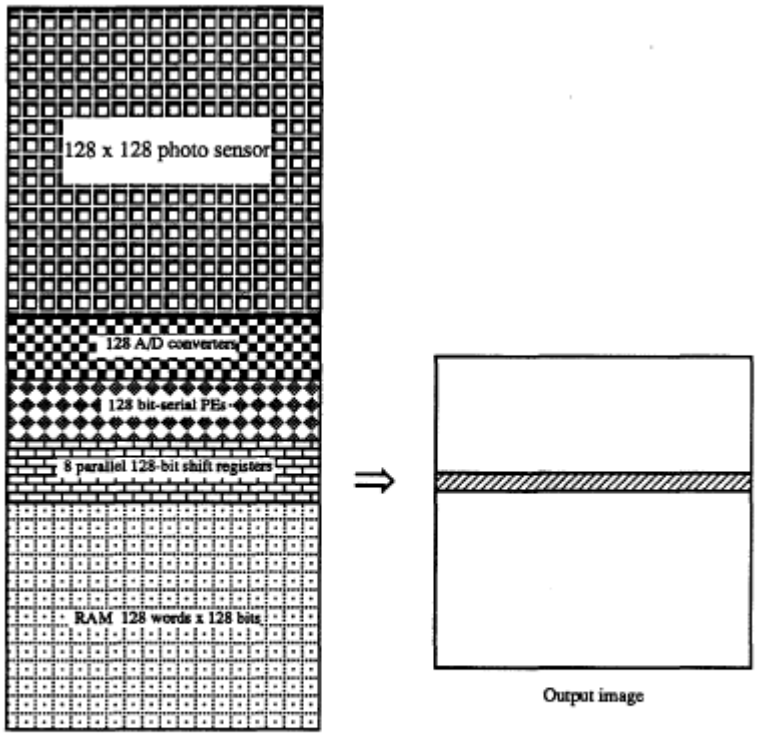
Asserted Claims	Additional References and Disclosure
	30.2 Overall System Architecture The next two subsections go into detail about the architecture of the GeForce 6 Series GPUs. Section 30.2.1 describes the architecture in terms of its graphics capabilities. Section 30.2.2 describes the architecture 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 473.

Asserted Claims	Additional References and Disclosure
	 **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. 474.

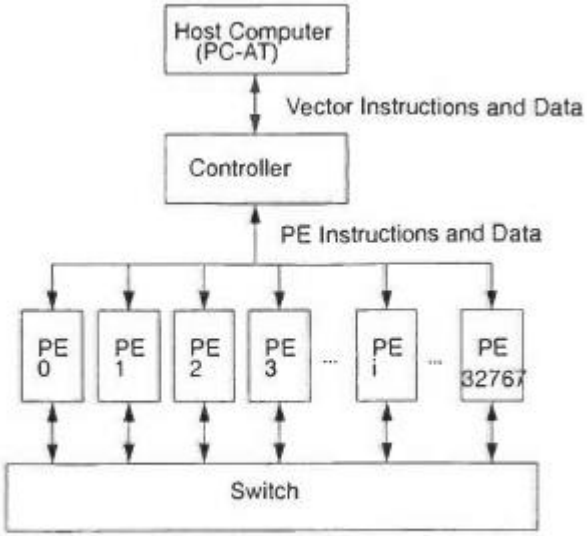
Asserted Claims	Additional References and Disclosure
1[b] “main memory, operably coupled to the central processing unit via a bus, to store the input data received by the central processing unit;”	*As a non-limiting example, GPU Gems discloses main memory (e.g., System DRAM), operably coupled to the central processing unit:* **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 473.
1[c] “an accelerator, operably coupled to the central processing unit	*As a non-limiting example, Svensson discloses an accelerator (e.g., a SIMD processor and/or SIMD array) with the required configuration.*

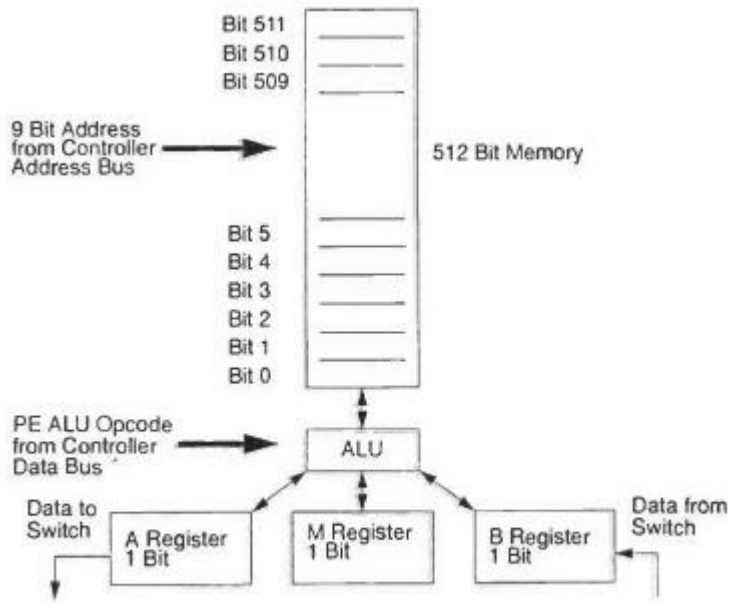
Asserted Claims	Additional References and Disclosure
and the main memory via the bus, to receive at least a portion of the input data from the main memory, the accelerator comprising:”	 Svensson Fig. SIMD-10a.

Asserted Claims	Additional References and Disclosure
	 Input/Output structure of LUCAS Figure SIMD-15 Svensson Fig. SIMD-15.

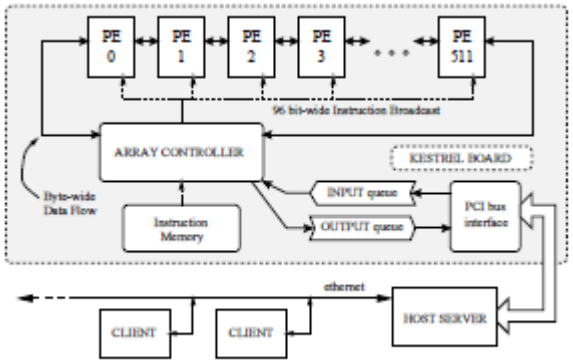
Asserted Claims	Additional References and Disclosure
	 PASIC overview Figure SIMD-16 Svensson Fig. SIMD-16. We illustrate the transfer between the different formats on the I/O system of LUCAS. See figure SIMD-15 and figure SIMD-l0a. A set of 8-bit I/O Registers (shift registers) is connected to the Memory Array, one register per memory word. The I/O Registers can be read or written from the Front-End Processor or dedicated I/O Processor in the conventional word-at-a-time format A data input process can be divided into two phases: one to fill the I/O Registers from the Front-End or I/O Processor, one to shift the contents out

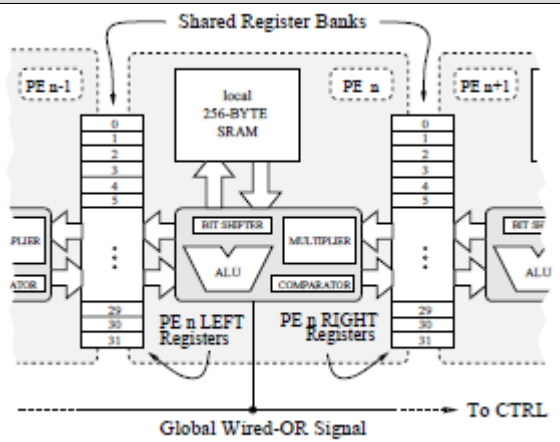
Asserted Claims	Additional References and Disclosure
	of the I/O Registers and into any field of the Memory Array. The first phase needs one write cycle of the Front-End or I/O Processor to transfer 8 bits, the second phase requires 8 Memory Array write cycles, in each of which an entire bit-slice is transferred. A variation of this input/output scheme that is used in some designs does not allow for addressing of the I/O registers. Instead, data is shifted in and out of the register set. Such I/O register systems may be used also for communication between the PEs. PASIC (Chen et al. 1990), see figure SIMD-16, is an example of a design with such a bit-parallel shift register along the row of PEs. This can be used for image output as well as for inter-PE communication. However, for image input there is a direct, word-parallel, bit-serial interface between the photosensor array (128 by 128 pixels in present version) and the linear array of PEs. Thus, PASIC is an example of a design which has special facilities to interface between the data format of the application and the processing data format, resulting in a significantly higher input speed. PASIC is primarily intended for low-level vision processing where high speed image I/O is required. Svensson, at 18–19. *As a non-limiting example, Glover discloses an accelerator (e.g., processing elements (PE)) with the required configuration.*

Asserted Claims	Additional References and Disclosure
	 Figure 1: A block diagram of the MM32k. Glover Fig. 1.

Asserted Claims	Additional References and Disclosure
	 Figure 2: A block diagram of a single processing element (PE). Glover Fig. 2.

Asserted Claims	Additional References and Disclosure
	2 SIMD ARCHITECTURE The SIMD PE array contains 32768 one bit processors, each with 512 bits of memory and a connection to the interconnection network. The PE array design is unique in that 2048 PEs, including their PE memory, are realized on a single chip. The total PE array memory is 2 megabytes and has a peak memory bandwidth is 25 gigabytes per second. The PE array can add 8 bit integers at 2.5 gigaoperations per second. It also dissipates less than 10 watts of power and is shown in Figure 1. Each PE has three one bit registers, a 512 bit memory, and a one bit ALU. It performs bit serial arithmetic and can therefore vary the number of bits of precision to fit the problem at hand, saving SIMD instruction cycles and SIMD memory. There are 17 instructions in the PE instruction set, all of which execute at a 6.25 MIPS rate. The PE instruction set is functionally complete in that it can perform boolean NOT and OR functions and can therefore perform any operation, including arithmetic and conditional operations. A single PE is shown in Figure 2. The interconnection network allows data to be sent from one PE to another. It is implemented by a 64*64 full crossbar switch with 512 PEs connected to each port of the switch. It allows data to be sent from one PE to another PE, an arbitrary distance away, in constant time. The peak switch bandwidth is 280 megabytes per second. The switch also allows the PE array to perform data reduction operations, such as taking the sum or maximum over data elements distributed across all PEs. Glover at 844. *As a non-limiting example, Mesa-Martinez discloses an accelerator (e.g., UCSC Kestrel SIMD computer) with the required configuration.*

Asserted Claims	Additional References and Disclosure
	 Fig. 2. UCSC Kestrel SIMD computer. Mesa-Martinez Fig. 2.  Fig. 3. Kestrel board architecture. Mesa-Martinez Fig. 3.

Asserted Claims	Additional References and Disclosure
	 Fig. 4. Kestrel processing element overview. Mesa-Martinez Fig. 4. Kestrel Architecture The first generation UCSC Kestrel processor is implemented as a systolic array [8] of 512 processing elements (PEs). The system’s configuration as a linear array is ideal for such target applications as sequence analysis, at the same time that it provides a simpler and more flexible programming model than more complex multiple-instruction, multiple data stream (MIMD) machines would when applied to the same domain of problems [7]. For the applications that we were envisioning, programmability was more important than large local memories or reconfigurable interconnection fabrics [4]. We believe our SIMD approach to be a more cost effective solution than systems using fully reconfigurable general purpose multiprocessors [5], [6]. Each processing element in the array is 8-bits wide and contains its own local memory and datapath. The Kestrel board also contains several SDRAM banks that implement the instruction memory (IM) as well as the input/output queues (IQm, OQm) [3]. Instruction broadcasting and sequencing is performed by an on board controller, that is also in charge of memory management (Figure 3). The whole system is self contained and implemented as a single full length PCI card (Figure 2).

Asserted Claims	Additional References and Disclosure
	Like most SIMD processors Kestrel broadcasts each instruction to each PE in the array. Due to the relatively large number of PEs that Kestrel has, and the fact that these PEs are distributed among several chips, instruction broadcasting can easily become more taxing than the actual time to execute the instruction. In order to solve this bottleneck Kestrel implements a board-level instruction issue and broadcast [4]. In order to handle inter-PE communication each PE shares a file of registers with its left and right neighbors. These register banks are known as Systolic Shared Registers (SSRs) [9]. Although this topology might seem limiting at first, it allows for computation and communication to occur concurrently. When a result is stored in the SSR after the instruction has been completed it is immediately visible to the next PE, thus communication takes place automatically (Figure 4). All addresses to the SSRs are issued globally in order to prevent adjacent PEs from writing to the same bank at the same time. This communication method is both elegant and efficient since it greatly simplifies the design of the PE and allowing most of its area to be dedicated for processing and memory. Mesa-Martinez at 2. Controller The controller is implemented using an FPGA. This device serves as the instruction sequencer for the system. The controller also handles the board’s I/O by interacting with the onboard PCI interface chip. The program instructions received from the host computer are stored in a separate instruction memory (IM). The Kestrel controller is programmed by using 54 bits of the 96-bit instruction word. The remaining bits are used as the instruction to be broadcast to the array. Each cycle, the controller must decide whether or not data is read from the input queue, whether or not data is written to the output queue, and whether or not the immediate field in the instruction should be replaced with data from the controller. A small amount of local memory and logic is used by the controller in order to enable loop counting and the recirculation of data among the PEs. When a critical condition is reached by the system the controller will generate the corresponding interrupt to inform the host about the board’s condition. Processing Element

Asserted Claims	Additional References and Disclosure
	The heart of the Kestrel architecture is its custom processing element (PE). Each Kestrel PE has the following elements in its datapath: 8-bit ALU and comparator, multiplier, bit shifter, and result selector (Figure 4). All the elements within the datapath are designed to complete one operation per cycle. The inclusion of a multiplier and some extra functionality in the ALU was made in order to expand the scope of possible applications for the architecture. Mesa-Martinez at 3.
1[c][i] “at least one graphics processing unit to perform a sequence of computations on the at least a portion of the input data so as to generate output data, the sequence of computations representing an artificial neural network, intermediate computations in the sequence of computations representing respective layers of the artificial neural network and yielding intermediate results; and”	*See also* 1[c]. *As a non-limiting example, ANN discloses a graphics processing unit (e.g., NVIDIA’s GF6000 GPU) to perform a sequence of a computations on at least a portion of input data so as to generate output data (e.g. results, of computations in each layer of a neural network), the sequence of computations representing an artificial neural network (e.g., a three layer MLP neural network), intermediate computations in the sequence of computations representing respective layers of the artificial neural network and yielding intermediate results (e.g., results of computations in layers other than the output layer):*

Asserted Claims	Additional References and Disclosure	
	Artificial Neural Network (ANN) is widely used in pattern recognition related area. In some case, the computational load is very heavy, in other case, real time process is required. So there is a need to apply a parallel algorithm on it, and usually the computation for ANN is inherently parallel. In this paper, graphic hardware is used to speed up the computation of ANN. In recent years, graphic processing unit (GPU) grows faster than CPU. Graphic hardware venders provide programmability on GPU. In this paper, application of commodity available GPU for two kinds of ANN models was explored. One is the self-organizing maps (SOM); the other is multi layer perceptron (MLP). The computation result shows that ANN computing on GPU is much faster than on standard CPU when the neural network is large. And some design rules for improve the efficiency on GPU are given. **Keywords: Graphic Process Unit; ANN; SOM; MLP** I. INTRODUCTION Artificial neural network is widely used in classification and pattern recognition. In this paper, two kinds of ANN, self-organizing maps (SOM) [1] and multi layer perceptron (MLP), are implemented on graphic hardware for speed up the computation. MLP is a very simple neural network, the process is linear with one input layer, several hidden layer, and an output layer. It is usually trained by back propagation (BP) algorithm. SOM consists of one layer of n-dimensional units (neurons). It is fully connected with the network input. Additionally, there exist lateral connections through which a topological structure is imposed. For the standard model, the topology is a regular two-dimensional map instantiated by connections between each unit and its direct neighbors. 622.	

Asserted Claims	Additional References and Disclosure
	In this paper, two kinds of ANN computation on GPU are given. In section 2, SOM computation model and implementation on graphic hardware is discussed. In section 3, MLP computation model and implementation on GPU is discussed. In section 4, the computation result and comparison are given for both CPU and GPU. In section 5, some of the design details and lessons we learned during implementation are given. In section 6, conclusion and some future works are given. *Id.* *B.Computational Model of MLP* Three layer MLP neural network is selected to recognize the ball. The input layer consists of seven nodes for seven characters. The hidden layer consists of three nodes, and the output layer consists of just one node. Back propagation method was chosen to train the network. The train set takes as follow. 16 locations are selected from each of 10 robot cars, 4 locations are selected from center of ball, and one location is selected from background field. Totally, 165 locations are selected as the train set. For each selected location, seven characters are calculated and take as an element of the train set. The trained MLP is used to recognize and trace the ball in real time. 624.

Asserted Claims	Additional References and Disclosure
	Current GPU has a limited instruction length and a limited number of temporary variables for calculation at each location. So multipass is needed for complex problem. For MLP discussed in this section, three passes are performed. In the first pass, average values of three color and luminance are calculated. In the second pass, the standard deviations of the three colors are calculated. In the third pass, classification result is got from MLP calculation on the characteristic. 624. In this calculation, we use a new Nvidia’s GF6000 serious graphic hardware. The reason is that the ATI’s GPU supports very few numbers of operations in each pass; so more passes are need for MLP calculation. Old Nvidia’s GPU does not support fully float texture, which is crucial for the precision of result. 625.

Asserted Claims	Additional References and Disclosure
	Second, if possible, do one’s best to decrease the calculation passes. For computation scheme described in section 2.2, the main bottleneck is at finding minimum procedure. The test shows that the bottleneck comes from multi-pass used in finding minimum procedure. Table 2 shows result for different scheme of finding the minimum. To decrease the pass, we make two changes, the first is to combine the value calculation with one pass of find minimum, and the second is to combine two pass of find minimum computation into one pass. Instead of calculate 4 units, 16 units are calculated. GPUA represent calculation on ATI graphic card and GPUN represent the calculation on NVIDIA graphic card. The performance increase is clear, especially for Nvidia’s graphic card. TABLE II: COMPARISON BETWEEN MORE AND FEW PASS KFM size | 128*128 | 256*256 | 512*512 GPUA more pass /ms | 366 | 400 | 533 GPUA few pass /ms | 211 | 244 | 511 GPUN more pass /ms | 190 | 640 | 2889 GPUN few pass /ms | 104 | 256 | 900 Third, do best to decrease the data exchange between CPU and GPU. Usually OpenGL’s PBuffer are used to save the intermediate result in a texture on GPU and reuse it as an input data. Harris[3] had created a class called “Render to Texture” to easy the use of PBuffer. We had used this class in our program. *Id.*

Asserted Claims	Additional References and Disclosure
	In this paper, implementation for two ANN models on graphic hardware is given. Inherent parallelism of commodity graphic hardware is used to accelerate the computation of ANN. The result shows that GPU is capable for some of ANN calculation, the graphic hardware make it possible for an increasing performance/cost ratio on the area of large size ANN computation. Compared to Bohn’s initial computation on SGI workstation, our implementation has two benefits. One is our calculation is more precise, for we had use the float point computing. The other is that we only use a commodity available graphic card, which is much more easily available than SGI workstation so can be widely used. The implementation on graphic hardware introduce in this paper has other implicit benefit too. For the SOM computing, a multi-texture or 3-D texture can be used to store the map and make more general SOM computing without the restriction of the vector length of 4. For the MLP used in robot soccer, some graphic hardware have “video in” function, using this kind of graphic hardware; image information can be retrieved directly from camera and store on graphic memory, and don’t need to transfer data between CPU and GPU, which will speed the process. We can also do other general ANN computation on GPU, because GPU provide almost all arithmetic operation, logic operation and some important mathematic function. And for the application of neural network on images, it is more naive to make such computing on a graphic hardware. 626.

Asserted Claims	Additional References and Disclosure
	As a non-limiting example, Oh discloses a graphics processing unit (e.g., GPU) to perform a sequence of a computations on at least a portion of input data so as to generate output data (e.g. results of computations in each layer of a neural network), the sequence of computations representing an artificial neural network, intermediate computations in the sequence of computations representing respective layers of the artificial neural network and yielding intermediate results (e.g., results of computations in layers other than the output layer): An artificial neural network, usually referred to as ‘neural network’, is based on the concept of the workings of the human brain. There are many different types of NN, with the more popular being a multilayer perceptron, learning vector quantization, radial basis function, Hop/eld, and Kohonen. The current study focuses on using a GPU to implement a multilayer perceptron, which is usually fully connected between adjacent layers. The input layer receives the input features of a given application. Although the network structure can vary as regards the number of layers, number of nodes in each layer, and input mask size, each layer performs the same inner-product operation between the given input vectors and the weight vectors, followed by a non-linear function. Moreover, many inner-product operations can be replaced with a matrix multiplication, which is more appropriate for GPU implementation. 1311–12. If there is more than one layer in an NN, the above procedure is repeated for each layer. The result of the previous layer is saved in the form of a render target texture, which is then used as an input for the next layer. Note that, even though an NN may have multiple layers, the GPU can perform all the operations after texture creation. 1312. As described above, the inner-product operation for each layer of an NN can be replaced with a matrix multiplication based on accumulating the input vectors and weight vectors. As such, the computation-per-layer can be written as follows:

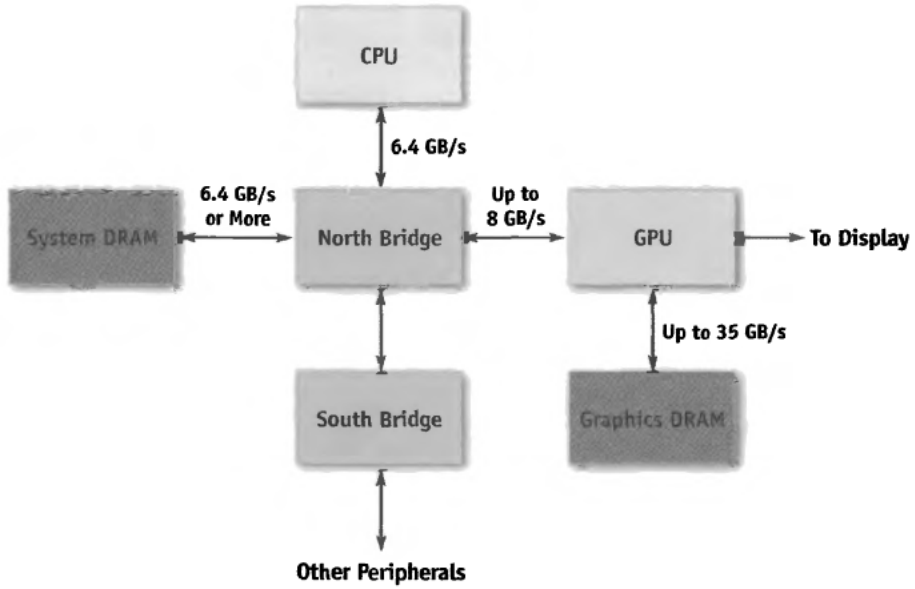
Asserted Claims	Additional References and Disclosure
	$$W = \begin{bmatrix} w_{11} & w_{12} & w_{13} & \dots & w_{1N} \\ w_{21} & w_{22} & w_{23} & \dots & w_{2N} \\ \dots & \dots & \dots & \dots & \dots \\ w_{M1} & w_{M2} & w_{M3} & \dots & w_{MN} \end{bmatrix} = \begin{bmatrix} W_1 \\ W_2 \\ \dots \\ W_M \end{bmatrix},$$ $$X = \begin{bmatrix} x_{11} & x_{12} & x_{13} & \dots & x_{1L} \\ x_{21} & x_{22} & x_{23} & \dots & x_{2L} \\ \dots & \dots & \dots & \dots & \dots \\ x_{N1} & x_{N2} & x_{N3} & \dots & x_{NL} \end{bmatrix}$$ $$= \begin{bmatrix} X_1 & X_2 & X_3 & \cdots & X_L \end{bmatrix},$$ $$B = \begin{bmatrix} b_1 & b_1 & b_1 & \dots & b_1 \\ b_2 & b_2 & b_2 & \dots & b_2 \\ \dots & \dots & \dots & \dots & \dots \\ b_M & b_M & b_M & \dots & b_M \end{bmatrix},$$

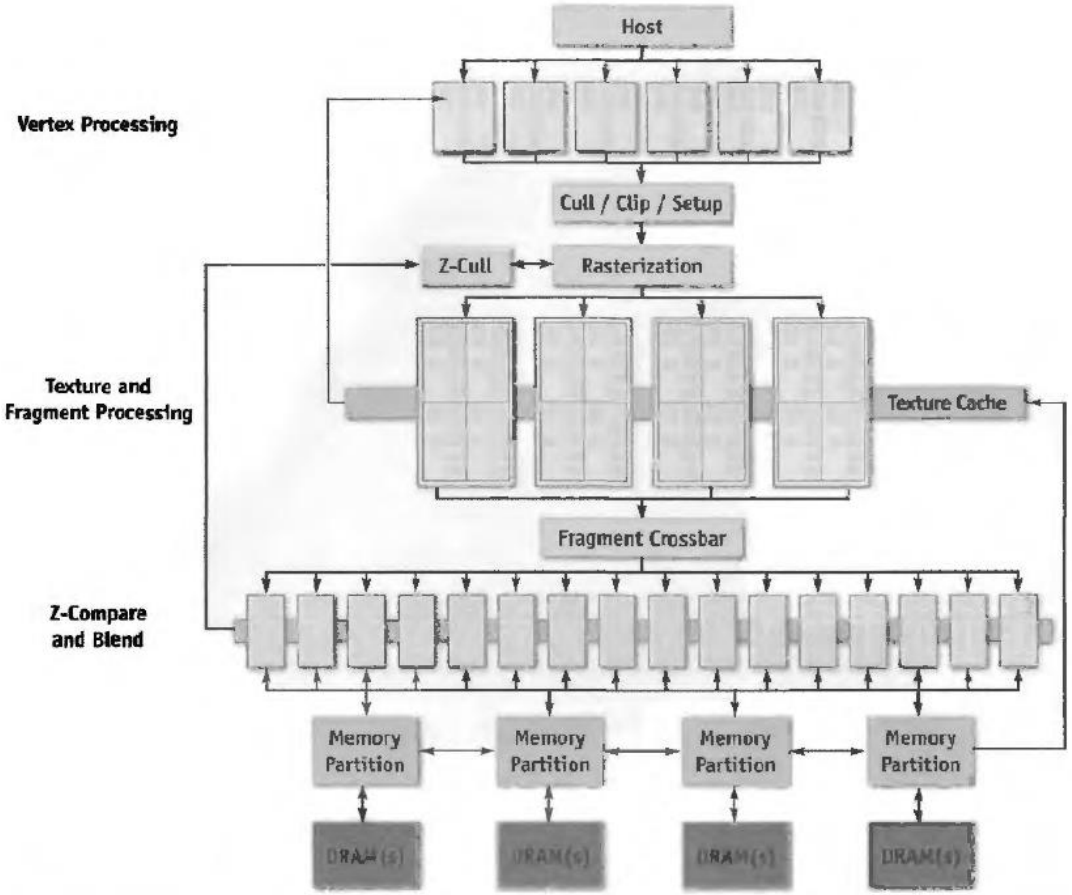
Asserted Claims	Additional References and Disclosure
	$M = W \times X + B$ $= \begin{bmatrix} W_1 \cdot X_1 & W_1 \cdot X_2 & W_1 \cdot X_3 & \dots & W_1 \cdot X_N \\ W_2 \cdot X_1 & W_2 \cdot X_2 & W_2 \cdot X_3 & \dots & W_2 \cdot X_N \\ \dots & \dots & \dots & \dots & \dots \\ W_M \cdot X_1 & W_M \cdot X_2 & W_M \cdot X_3 & \dots & W_M \cdot X_N \end{bmatrix}$ $+ \begin{bmatrix} b_1 & b_1 & b_1 & \dots & b_1 \\ b_2 & b_2 & b_2 & \dots & b_2 \\ \dots & \dots & \dots & \dots & \dots \\ b_M & b_M & b_M & \dots & b_M \end{bmatrix}$ $= \begin{bmatrix} m_{11} & m_{12} & m_{13} & \dots & m_{1L} \\ m_{21} & m_{22} & m_{23} & \dots & m_{2L} \\ m_{31} & m_{32} & m_{33} & \dots & m_{3L} \\ \dots & \dots & \dots & \dots & \dots \\ m_{M1} & m_{M2} & m_{M3} & \dots & m_{ML} \end{bmatrix}, \quad (2)$ $R = \text{sigmoid}(M)$ $= \begin{bmatrix} 1+e^{-m_{11}} & 1+e^{-m_{12}} & 1+e^{-m_{13}} & \dots & 1+e^{-m_{1L}} \\ 1+e^{-m_{21}} & 1+e^{-m_{22}} & 1+e^{-m_{23}} & \dots & 1+e^{-m_{2L}} \\ \dots & \dots & \dots & \dots & \dots \\ 1+e^{-m_{M1}} & 1+e^{-m_{M2}} & 1+e^{-m_{M3}} & \dots & 1+e^{-m_{ML}} \end{bmatrix}, \quad (3)$ where w_{ij} denotes the weight at the connection between the *i*th node of the output layer and the *j*th node of the input layer, M is the number of nodes in the output layer, and N is the number of nodes in the input layer. In addition, x_{ij} is the *i*th feature value of the *j*th input vector and bi is the bias term for the *i*th output node from L input vectors. The final result R_{ij} is the output of the *i*th output node for the *j*th input vector. 1312.

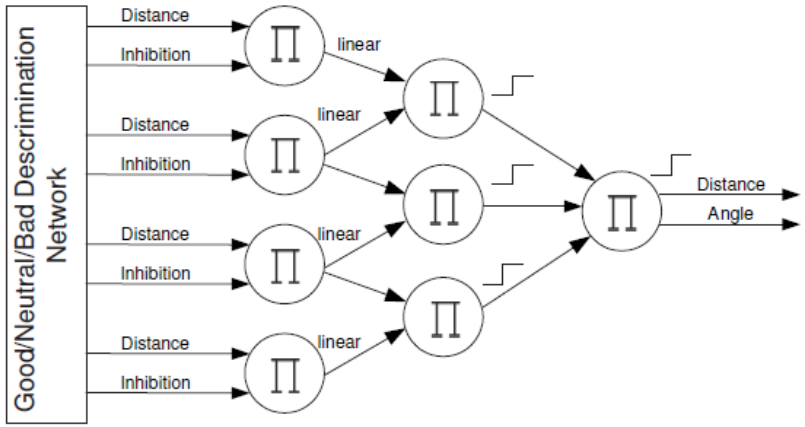
Asserted Claims	Additional References and Disclosure
	The above computation comprises of a matrix multiplication followed by a bias factor addition and sigmoid operation. The matrix multiplication is explained first. The method proposed by Moravanszky [1] is used to implement the matrix multiplication. The two matrices are converted into textures, denoted by texture W and texture X , then the matrix multiplication is performed by rendering. A rectangle is rendered to cover the whole screen. The vertex shader outputs the position and texture coordinates for each vertex of the rectangle, where each vertex has two texture coordinates: one for the row of texture W and the other for the column of texture X . For example, the upper left vertex will have the texture coordinates of the first row of texture W and the first column of texture X , while the upper right vertex will have the texture coordinates of the first row of texture W and the last column of texture X , and so on. As a result of the vertex shader, every pixel (i; j) has texture coordinates corresponding to the *i*th row of W and the *j*th column of X . The pixel shader then performs the inner-product between the row of W and the column of X specified by the texture coordinates. Fig. 1 shows an example of matrix multiplication using a GPU. The number of rendering passes required for matrix multiplication depends on the capability of the GPU, including the number of pixel shader operations and number of texture load operations. 1312–13. Therefore, this paper briefly describes such a text detection method, and readers are referred to the author’s previous publication for more details [3]. In the proposed method, an NN is used to classify the pixels of input images, whereby the feature extraction and pattern recognition stage are integrated in the neural network. The NN then examines local regions looking for text pixels that may be contained in a text region. Therefore, an M × M pixel region in the image is received as the input and a classified image is generated as the output. After the pattern passes the network, the value of the output node is compared with a threshold value and the class of each pixel determined, resulting in a classified image. GPU-based pipelining processing is used to reduce the processing time, and the GPU’s performance is maximized by accumulating a large number of input vectors 1 to create a two-dimensional texture. The input layer then receives the grey values for the pixels at predefined positions inside an M×M window over the input image. Experiments were conducted using an 11 × 11 input window size, with the number of nodes in each hidden layer set at 30. As a result, the processing time for pixel classification was significantly reduced using a GPU. Fig. 2(b) shows the pixel classification result for the left input image, where a black pixel denotes a text pixel. The classification using a GPU produced almost the same result as without a GPU. 1313.

Asserted Claims	Additional References and Disclosure
	As a non-limiting example, GPU Gems discloses a graphics processing unit (e.g., NVIDIA’s GeForce 6 Series GPUs) to perform a sequence of a computations on at least a portion of input data so as to generate output data: The previous chapter described how GPU architecture has changed as a result of computational and communications trends in microprocessing. This chapter describes the architecture of the GeForce 6 Series GPUs from NVIDIA, which owe their formidable computational power to their ability to take advantage of these trends. Most notably, we focus on the GeForce 6800 (NVIDIA’s flagship GPU at the time of writing, shown in Figure 30-1), which delivers hundreds of gigaflops of single-precision floating-point computation, as compared to approximately 12 gigaflops for current high-end CPUs. In this chapter—and throughout the book—references to GeForce 6 Series GPUs should be read to include the latest Quadro FX GPUs supporting Shader Model 3.0, which provide a superset of the functionality offered by the GeForce 6 Series. We start with a general overview of where the GPU fits into the overall computer system, and then we describe the architecture along with details of specific features and performance characteristics. **30.1 How the GPU Fits into the Overall Computer System** The CPU in a modern computer system communicates with the GPU through a graphics connector such as a PCI Express or AGP slot on the motherboard. Because the graphics connector is responsible for transferring all command, texture, and vertex data from the CPU to the GPU, the bus technology has evolved alongside GPUs over the past few years. The original AGP slot ran at 66 MHz and was 32 bits wide, giving a transfer rate of 264 MB/sec. AGP 2×, 4×, and 8× followed, each doubling the available

Asserted Claims	Additional References and Disclosure
	bandwidth, until finally the PCI Express standard was introduced in 2004, with a maximum theoretical bandwidth of 4 GB/sec simultaneously available to and from the GPU. (Your mileage may vary; currently available motherboard chipsets fall somewhat below this limit—around 3.2 GB/sec or less.) It is important to note the vast differences between the GPU's memory interface bandwidth and bandwidth in other parts of the system, as shown in Table 30-1. **Table 30-1.** Available Memory Bandwidth in Different Parts of the Computer System Component — Bandwidth GPU Memory Interface — 35 GB/sec PCI Express Bus (×16) — 8 GB/sec CPU Memory Interface (800 MHz Front-Side Bus) — 6.4 GB/sec 471–472.

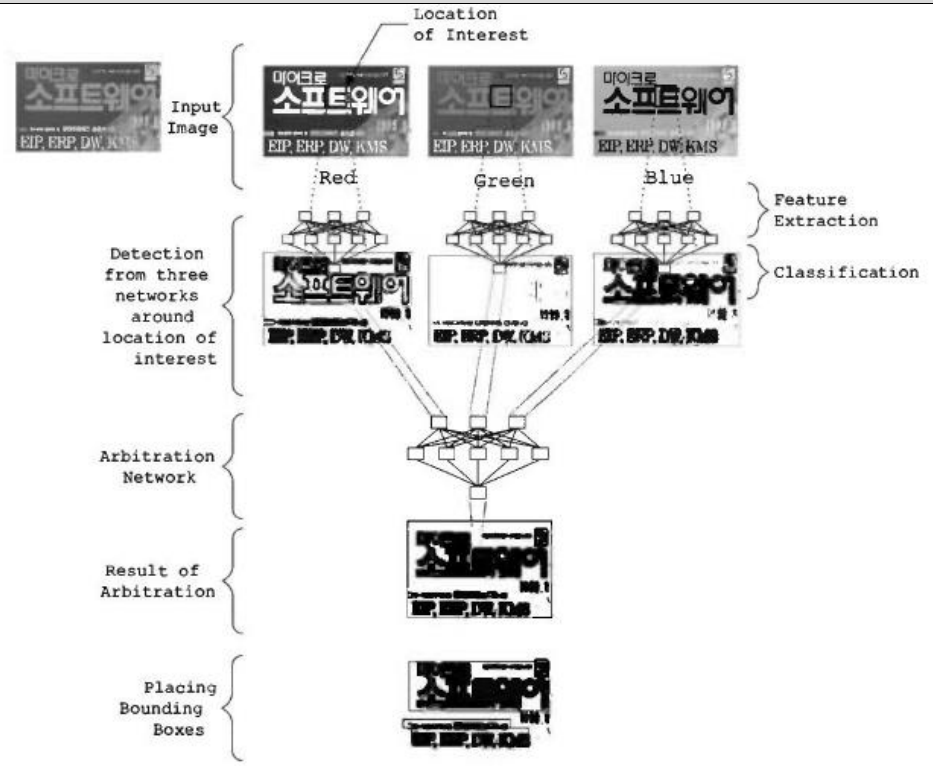
Asserted Claims	Additional References and Disclosure
	30.2 Overall System Architecture The next two subsections go into detail about the architecture of the GeForce 6 Series GPUs. Section 30.2.1 describes the architecture in terms of its graphics capabilities. Section 30.2.2 describes the architecture 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 473.

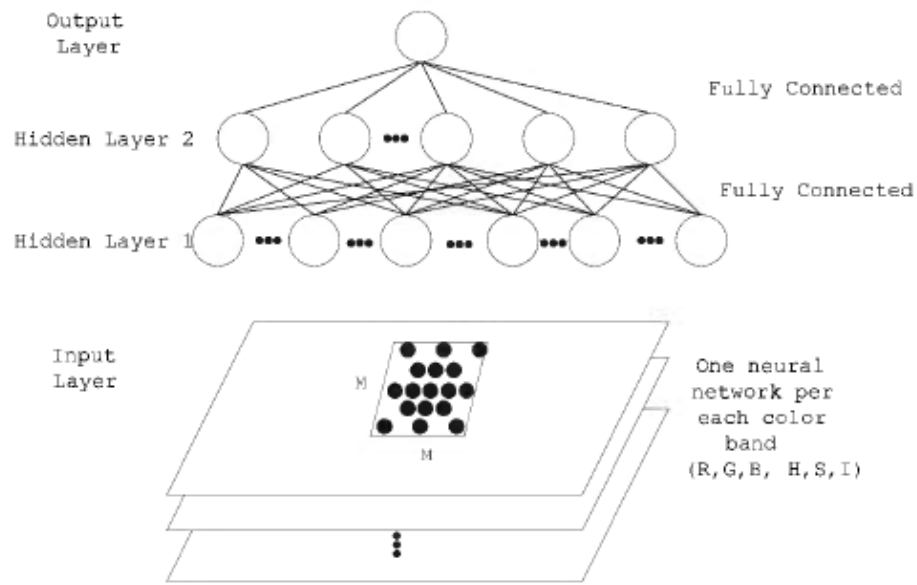
Asserted Claims	Additional References and Disclosure
	 **Figure 30-3. A Block Diagram of the GeForce 6 Series Architecture** 474. *As a non-limiting example, the Davis Thesis discloses simulating a neural network on a graphics processing unit, with the computations as claimed.*

Asserted Claims	Additional References and Disclosure
	Chapter 1. Introduction Introductions to both GPUS and ANNs provide a basic overview of the information needed to understand the application of ANNs on GPUs. A survey of current GPU and CPU programming languages with ANN supporting features is presented and briefly discussed. Finally, a comparison between several ANNs implemented on both the CPU and the GPU are presented, allowing analysis and generalization about the applicability of ANNs on GPUs.  Figure 2.3: Example of a artificial neural network designed as part of a simulator robot control system. This network utilizes “Pi” (multiplication) neurons instead of the “Sigma” (summation) neurons used in this thesis research.

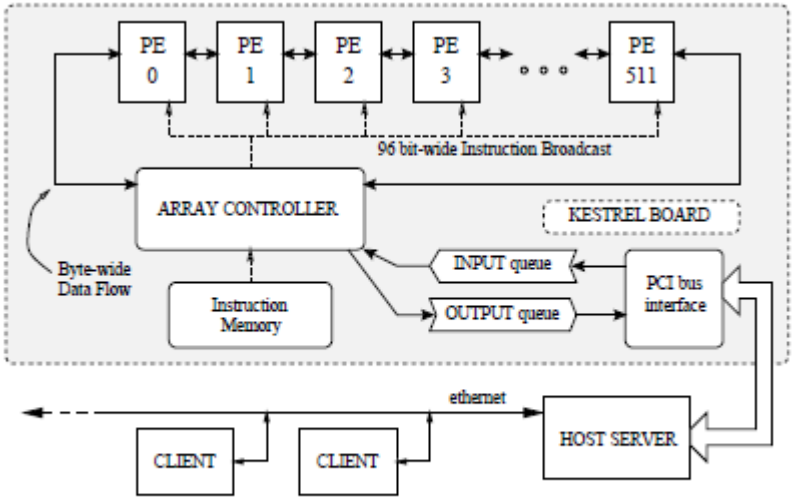
Asserted Claims	Additional References and Disclosure
	2.1.3 Artificial Neural Network Graphical Processing Unit Simulation Thomas Rolfes has published several versions of an article[16] on GPU simulation of the multi-layer perceptron (MLP). In these articles, he outlines an approach to the implementation of a MLP on the GPU through DirectX. In short, he uses matrix-matrix products to compute the activation levels for the neurons, then applies his non-linearity threshold function. Each layer in the network is a single matrix represented explicitly as a texture in DirectX, and inputs are aggregated into a matrix of compatible dimensionality. The matrix product is based on an approach presented in [14, 12, 20]. See Appendix G Kyoung-Su Oh and Keechul Jung have also implemented a multi-layer perceptron for the GPU to perform text detection[10]. Again, they use a method for matrix products outlined in [14, 12, 20].

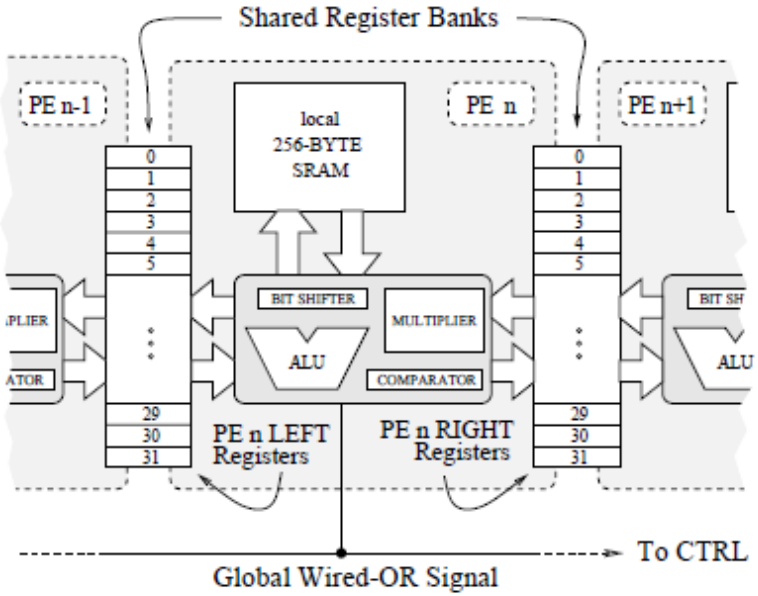
Asserted Claims	Additional References and Disclosure
	3.2.7 What makes an ANN amenable to GPU execution? About now hopefully you have an idea of what kinds of traits make a ANN ill or well suited to GPU simulation, but let’s summarize some of the characteristics here: • SIMD type operations – SIMD type operations are extremely well suited to programmable GPU execution. The basic form of the shader kernel and the hardware implementation are direct analogs of the SIMD architectures used on the CPU. The GPU has evolved to handle these kernel repeated convolutions in a parallel fashion to improve graphics performance and SIMD operations for computation can exploit this trait. • vector-matrix or matrix-matrix style linear algebra – Linear algebra that utilizes a matrix (or a collection of vectors) scales nicely for the GPU. For vector-matrix operations as you increase the size of your matrix, you linearly increase the number of operations per value (float). If you are able to utilize matrix-matrix operations this scaling becomes N^2 where N is the size of your matrix. This scaling factor, or algorithmic intensity, is of upmost importance for the GPU. The design of the GPU is such that it is meant to be heavily utilized with a high operation per byte ratio (more on this below). Davis Thesis at 3, 8, 24; *see also id.* at Appendices A–G. *As a non-limiting example, Jung discloses simulating a neural network for text location, with the computations as claimed.*

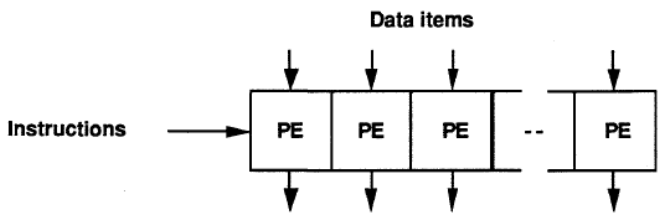
Asserted Claims	Additional References and Disclosure
	 Fig. 3. Architecture of discrimination network that can arbitrate among multiple text detection networks.

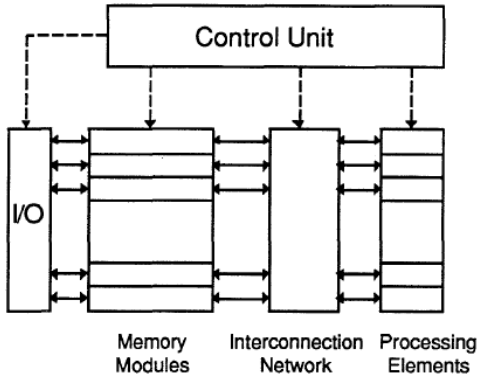
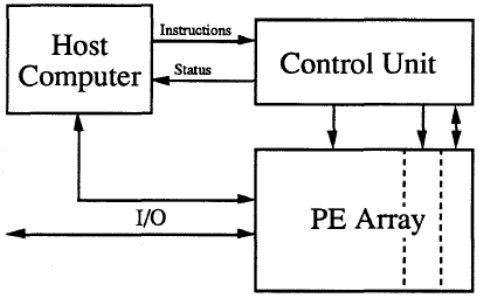
Asserted Claims	Additional References and Disclosure
	 Fig. 4. Three-layer feed-forward neural network (Jain and Karu, 1996). Jung at 1506–15.
1[c][ii] “accelerator memory, operably coupled to the at least one graphics processing unit, to store the results of the sequence of computations; and”	*See also* 1[c]. *As a non-limiting example, Mesa-Martinex discloses accelerator memory (e.g., Shared Register Banks), operably coupled to at least one graphics processing unit (e.g. processing elements (PEs)), to store the results of the sequence of computations:* The first generation UCSC Kestrel processor is implemented as a systolic array [8] of 512 processing elements (PEs). The system’s configuration as a linear array is ideal for such target applications as sequence analysis, at the same time that it provides a simpler and more flexible programming model than more complex multiple-instruction, multiple data stream (MIMD) machines would when applied to the same domain of problems [7]. For the applications that we were envisioning, programmability was more important than large local memories or reconfigurable interconnection fabrics [4]. We believe our SIMD

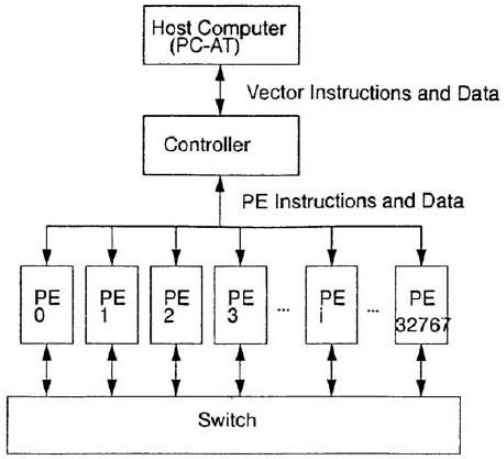
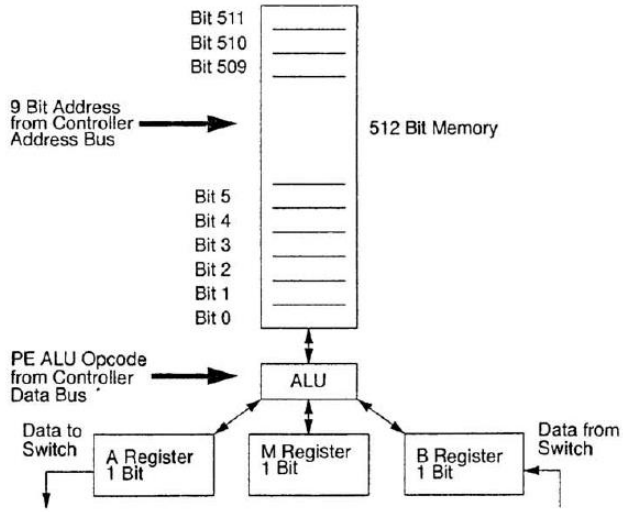
Asserted Claims	Additional References and Disclosure
	approach to be a more cost effective solution than systems using fully reconfigurable general purpose multiprocessors [5], [6]. Each processing element in the array is 8-bits wide and contains its own local memory and datapath. The Kestrel board also contains several SDRAM banks that implement the instruction memory (IM) as well as the input/output queues (IQm, OQm) [3]. Instruction broadcasting and sequencing is performed by an on board controller, that is also in charge of memory management (Figure 3). The whole system is self contained and implemented as a single full length PCI card (Figure 2). Like most SIMD processors Kestrel broadcasts each instruction to each PE in the array. Due to the relatively large number of PEs that Kestrel has, and the fact that these PEs are distributed among several chips, instruction broadcasting can easily become more taxing than the actual time to execute the instruction. In order to solve this bottleneck Kestrel implements a board-level instruction issue and broadcast [4]. In order to handle inter-PE communication each PE shares a file of registers with its left and right neighbors. These register banks are known as Systolic Shared Registers (SSRs) [9]. Although this topology might seem limiting at first, it allows for computation and communication to occur concurrently. When a result is stored in the SSR after the instruction has been completed it is immediately visible to the next PE, thus communication takes place automatically (Figure 4). All addresses to the SSRs are issued globally in order to prevent adjacent PEs from writing to the same bank at the same time. This communication method is both elegant and efficient since it greatly simplifies the design of the PE and allowing most of its area to be dedicated for processing and memory. A-9.

Asserted Claims	Additional References and Disclosure
	 Fig. 3. Kestrel board architecture. FIG. 3.

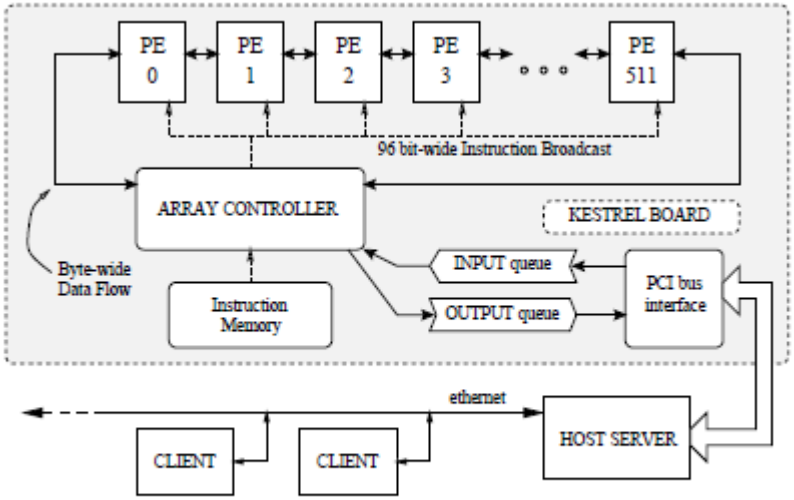
Asserted Claims	Additional References and Disclosure
	 Fig. 4. Kestrel processing element overview. FIG. 4.
1[d] “a controller, operably coupled to the at least one graphics processing unit and the accelerator memory,”	*As a non-limiting example, Svensson discloses a controller (e.g., control unit), operably coupled to the at least one graphics processing unit (e.g., host computer and/or processing elements):*

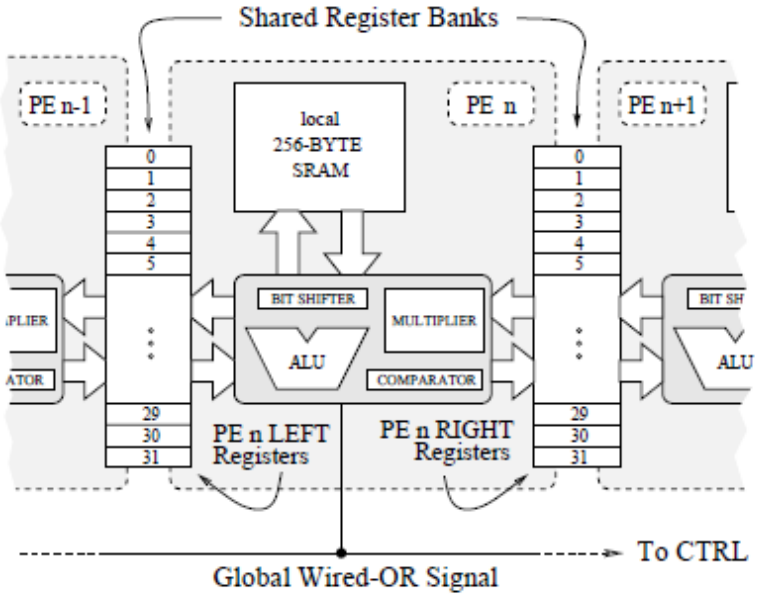
Asserted Claims	Additional References and Disclosure
	1. Principles of SIMD In Single Instruction stream, Multiple Data stream (SIMD) processors one instruction works on several data items simultaneously by using several Processing Elements (PEs), all of which carry out the same operation as illustrated in Figure SIMD-1.  Principle of SIMD processor. Figure SIMD-1. A SIMD Processor has a single Control Unit reading instructions pointed to by a single Program Counter, decoding them and sending control signals to the PEs. Data are to be supplied to, and derived from, the PEs by a memory with as many data paths as there are PEs. Figure SIMD-2a shows the resulting processing structure which is also known as a Processor Array. The Interconnection Network provides flexibility in choosing source and destination for data to and from the PEs, necessary in many algorithms. The I/O system plays the role of converting – typically at very high rates – input/output data between the format of the outside world and the internal format of the array. The design of the I/O system is highly application dependent. 1.

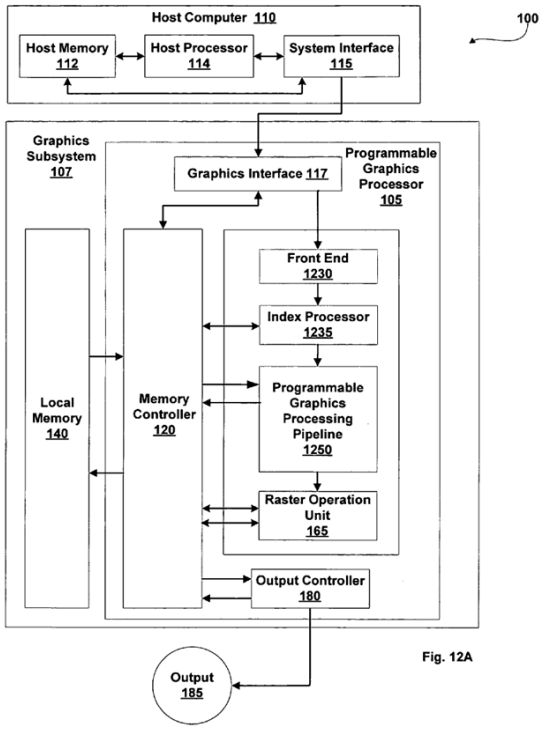
Asserted Claims	Additional References and Disclosure
	 Processor Array Figure SIMD-2a  Host Computer - Processor Array Relationship Figure SIMD-2b 2. *As a non-limiting example, Glover discloses a controller operably coupled to the at least one graphics processing unit (e.g., host computer and/or processing elements):*

Asserted Claims	Additional References and Disclosure
	 Figure 1: A block diagram of the MM32k.  Figure 2: A block diagram of a single processing element (PE). 845.

Asserted Claims	Additional References and Disclosure
	As a non-limiting example, Mesa-Martinez discloses a controller (e.g., array controller) operably coupled to the at least one graphics processing unit (e.g., processing elements (PEs)) and accelerator memory (e.g., Shared Register Banks): The first generation UCSC Kestrel processor is implemented as a systolic array [8] of 512 processing elements (PEs). The system’s configuration as a linear array is ideal for such target applications as sequence analysis, at the same time that it provides a simpler and more flexible programming model than more complex multiple-instruction, multiple data stream (MIMD) machines would when applied to the same domain of problems [7]. For the applications that we were envisioning, programmability was more important than large local memories or reconfigurable interconnection fabrics [4]. We believe our SIMD approach to be a more cost effective solution than systems using fully reconfigurable general purpose multiprocessors [5], [6]. Each processing element in the array is 8-bits wide and contains its own local memory and datapath. The Kestrel board also contains several SDRAM banks that implement the instruction memory (IM) as well as the input/output queues (IQm, OQm) [3]. Instruction broadcasting and sequencing is performed by an on board controller, that is also in charge of memory management (Figure 3). The whole system is self contained and implemented as a single full length PCI card (Figure 2). Like most SIMD processors Kestrel broadcasts each instruction to each PE in the array. Due to the relatively large number of PEs that Kestrel has, and the fact that these PEs are distributed among several chips, instruction broadcasting can easily become more taxing than the actual time to execute the instruction. In order to solve this bottleneck Kestrel implements a board-level instruction issue and broadcast [4]. In order to handle inter-PE communication each PE shares a file of registers with its left and right neighbors. These register banks are known as Systolic Shared Registers (SSRs) [9]. Although this topology might seem limiting at first, it allows for computation and communication to occur concurrently. When a result is stored in the SSR after the instruction has been completed it is immediately visible to the next PE, thus communication takes place automatically (Figure 4). All addresses to the SSRs are issued globally in order to prevent adjacent PEs from writing to the same bank at the same time. This communication method is both elegant and efficient since it greatly simplifies the design of the PE and allowing most of its area to be dedicated for processing and memory. A-9.

Asserted Claims	Additional References and Disclosure
	 Fig. 3. Kestrel board architecture. FIG. 3.

Asserted Claims	Additional References and Disclosure
	 Fig. 4. Kestrel processing element overview. FIG. 4. The controller is implemented using an FPGA. This device serves as the instruction sequencer for the system. The controller also handles the board’s I/O by interacting with the onboard PCI interface chip. The program instructions received from the host computer are stored in a separate instruction memory (IM). The Kestrel controller is programmed by using 54 bits of the 96-bit instruction word. The remaining bits are used as the instruction to be broadcast to the array. Each cycle, the controller must decide whether or not data is read from the input queue, whether or not data is written to the output queue, and whether or not the immediate field in the instruction should be replaced with data from the controller. A small amount of local memory and logic is used by the controller in order to enable loop counting and the recirculation of data among the PEs. When a critical condition is reached by the system the controller will generate the corresponding interrupt to inform the host about the board’s condition.

Asserted Claims	Additional References and Disclosure
	A-10. *As a non-limiting example, to the extent Plaintiff identifies a memory controller in the accused products as an “accelerator controller,” Kirk discloses a memory controller similar to those in the accused products (e.g., Memory Controller 120).*  Fig. 12A
1[d][i] “to initialize textures and shaders in the accelerator memory for performing the	*As a non-limiting example, Oh discloses initializing textures and shaders in the accelerator memory for performing a sequence of computations:* An artificial neural network, usually referred to as ‘neural network’, is based on the concept of the workings of the human brain. There are many different types of NN, with the more popular being a

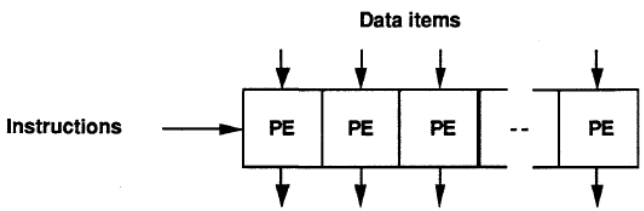
Asserted Claims	Additional References and Disclosure
sequence of computations,”	multilayer perceptron, learning vector quantization, radial basis function, Hop/eld, and Kohonen. The current study focuses on using a GPU to implement a multilayer perceptron, which is usually fully connected between adjacent layers. The input layer receives the input features of a given application. Although the network structure can vary as regards the number of layers, number of nodes in each layer, and input mask size, each layer performs the same inner-product operation between the given input vectors and the weight vectors, followed by a non-linear function. Moreover, many inner-product operations can be replaced with a matrix multiplication, which is more appropriate for GPU implementation. 1311–12. If there is more than one layer in an NN, the above procedure is repeated for each layer. The result of the previous layer is saved in the form of a render target texture, which is then used as an input for the next layer. Note that, even though an NN may have multiple layers, the GPU can perform all the operations after texture creation. 1312. As described above, the inner-product operation for each layer of an NN can be replaced with a matrix multiplication based on accumulating the input vectors and weight vectors. As such, the computation-per-layer can be written as follows:

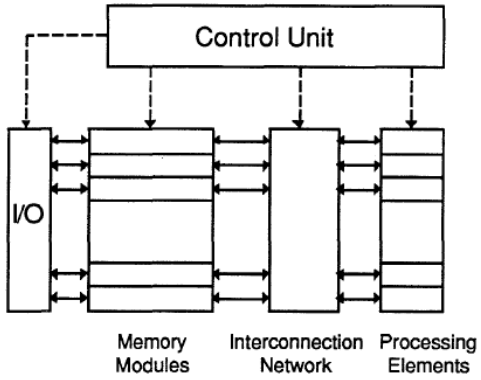
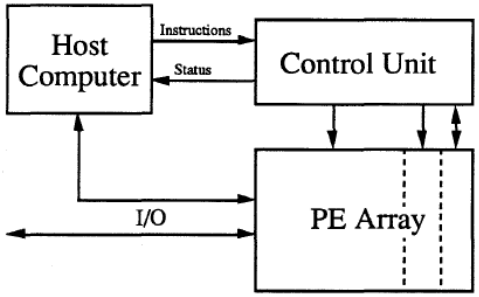
Asserted Claims	Additional References and Disclosure
	$$W = \begin{bmatrix} w_{11} & w_{12} & w_{13} & \dots & w_{1N} \\ w_{21} & w_{22} & w_{23} & \dots & w_{2N} \\ \dots & \dots & \dots & \dots & \dots \\ w_{M1} & w_{M2} & w_{M3} & \dots & w_{MN} \end{bmatrix} = \begin{bmatrix} W_1 \\ W_2 \\ \dots \\ W_M \end{bmatrix},$$ $$X = \begin{bmatrix} x_{11} & x_{12} & x_{13} & \dots & x_{1L} \\ x_{21} & x_{22} & x_{23} & \dots & x_{2L} \\ \dots & \dots & \dots & \dots & \dots \\ x_{N1} & x_{N2} & x_{N3} & \dots & x_{NL} \end{bmatrix}$$ $$= \begin{bmatrix} X_1 & X_2 & X_3 & \cdots & X_L \end{bmatrix},$$ $$B = \begin{bmatrix} b_1 & b_1 & b_1 & \dots & b_1 \\ b_2 & b_2 & b_2 & \dots & b_2 \\ \dots & \dots & \dots & \dots & \dots \\ b_M & b_M & b_M & \dots & b_M \end{bmatrix},$$

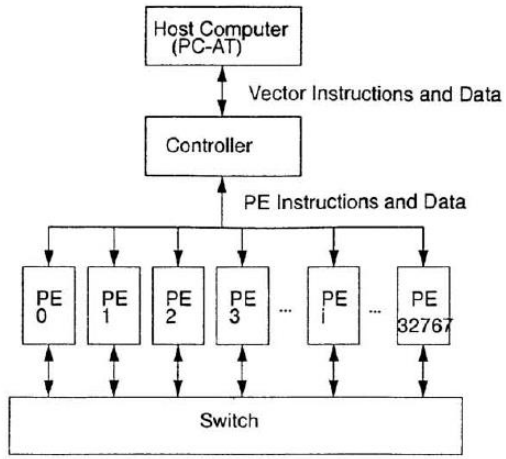
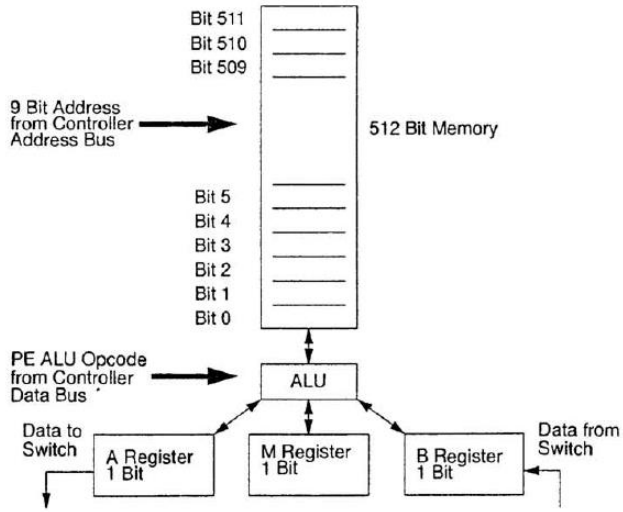
Asserted Claims	Additional References and Disclosure
	$M = W \times X + B$ $= \begin{bmatrix} W_1 \cdot X_1 & W_1 \cdot X_2 & W_1 \cdot X_3 & \dots & W_1 \cdot X_N \\ W_2 \cdot X_1 & W_2 \cdot X_2 & W_2 \cdot X_3 & \dots & W_2 \cdot X_N \\ \dots & \dots & \dots & \dots & \dots \\ W_M \cdot X_1 & W_M \cdot X_2 & W_M \cdot X_3 & \dots & W_M \cdot X_N \end{bmatrix}$ $+ \begin{bmatrix} b_1 & b_1 & b_1 & \dots & b_1 \\ b_2 & b_2 & b_2 & \dots & b_2 \\ \dots & \dots & \dots & \dots & \dots \\ b_M & b_M & b_M & \dots & b_M \end{bmatrix}$ $= \begin{bmatrix} m_{11} & m_{12} & m_{13} & \dots & m_{1L} \\ m_{21} & m_{22} & m_{23} & \dots & m_{2L} \\ m_{31} & m_{32} & m_{33} & \dots & m_{3L} \\ \dots & \dots & \dots & \dots & \dots \\ m_{M1} & m_{M2} & m_{M3} & \dots & m_{ML} \end{bmatrix}, \quad (2)$ $R = \text{sigmoid}(M)$ $= \begin{bmatrix} 1+e^{-m_{11}} & 1+e^{-m_{12}} & 1+e^{-m_{13}} & \dots & 1+e^{-m_{1L}} \\ 1+e^{-m_{21}} & 1+e^{-m_{22}} & 1+e^{-m_{23}} & \dots & 1+e^{-m_{2L}} \\ \dots & \dots & \dots & \dots & \dots \\ 1+e^{-m_{M1}} & 1+e^{-m_{M2}} & 1+e^{-m_{M3}} & \dots & 1+e^{-m_{ML}} \end{bmatrix}, \quad (3)$ where w_{ij} denotes the weight at the connection between the *i*th node of the output layer and the *j*th node of the input layer, M is the number of nodes in the output layer, and N is the number of nodes in the input layer. In addition, x_{ij} is the *i*th feature value of the *j*th input vector and bi is the bias term for the *i*th output node from L input vectors. The final result R_{ij} is the output of the *i*th output node for the *j*th input vector. 1312.

Asserted Claims	Additional References and Disclosure
	The above computation comprises of a matrix multiplication followed by a bias factor addition and sigmoid operation. The matrix multiplication is explained first. The method proposed by Moravanszky [1] is used to implement the matrix multiplication. The two matrices are converted into textures, denoted by texture W and texture X , then the matrix multiplication is performed by rendering. A rectangle is rendered to cover the whole screen. The vertex shader outputs the position and texture coordinates for each vertex of the rectangle, where each vertex has two texture coordinates: one for the row of texture W and the other for the column of texture X . For example, the upper left vertex will have the texture coordinates of the first row of texture W and the first column of texture X , while the upper right vertex will have the texture coordinates of the first row of texture W and the last column of texture X , and so on. As a result of the vertex shader, every pixel (i; j) has texture coordinates corresponding to the *i*th row of W and the *j*th column of X . The pixel shader then performs the inner-product between the row of W and the column of X specified by the texture coordinates. Fig. 1 shows an example of matrix multiplication using a GPU. The number of rendering passes required for matrix multiplication depends on the capability of the GPU, including the number of pixel shader operations and number of texture load operations. 1312–13. Therefore, this paper briefly describes such a text detection method, and readers are referred to the author’s previous publication for more details [3]. In the proposed method, an NN is used to classify the pixels of input images, whereby the feature extraction and pattern recognition stage are integrated in the neural network. The NN then examines local regions looking for text pixels that may be contained in a text region. Therefore, an M × M pixel region in the image is received as the input and a classified image is generated as the output. After the pattern passes the network, the value of the output node is compared with a threshold value and the class of each pixel determined, resulting in a classified image. GPU-based pipelining processing is used to reduce the processing time, and the GPU’s performance is maximized by accumulating a large number of input vectors 1 to create a two-dimensional texture. The input layer then receives the grey values for the pixels at predefined positions inside an M×M window over the input image. Experiments were conducted using an 11 × 11 input window size, with the number of nodes in each hidden layer set at 30. As a result, the processing time for pixel classification was significantly reduced using a GPU. Fig. 2(b) shows the pixel classification result for the left input image, where a black pixel denotes a text pixel. The classification using a GPU produced almost the same result as without a GPU. 1313.

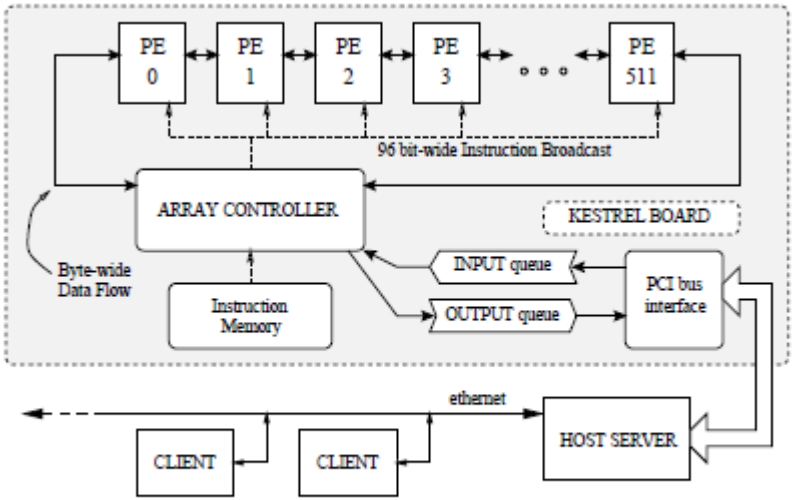
Asserted Claims	Additional References and Disclosure
	As a non-limiting example, ANN discloses initializing textures and shaders in the accelerator memory for performing a sequence of computations: Third, do best to decrease the data exchange between CPU and GPU. Usually OpenGL's PBuffer are used to save the intermediate result in a texture on GPU and reuse it as an input data. Harris[3] had created a class called "Render to Texture" to easy the use of PBuffer. We had used this class in our program. 625.
1[d][ii] "to control performance of the sequence of computations by the at least one graphics processing unit,"	*As a non-limiting example, Svensson discloses a controller (e.g., control unit) controlling performance of the sequence of computations (e.g., via instructions) by at least one graphics processing unit (e.g., host computer and/or processing elements):*

Asserted Claims	Additional References and Disclosure
	1. Principles of SIMD In Single Instruction stream, Multiple Data stream (SIMD) processors one instruction works on several data items simultaneously by using several Processing Elements (PEs), all of which carry out the same operation as illustrated in Figure SIMD-1.  Principle of SIMD processor. Figure SIMD-1. A SIMD Processor has a single Control Unit reading instructions pointed to by a single Program Counter, decoding them and sending control signals to the PEs. Data are to be supplied to, and derived from, the PEs by a memory with as many data paths as there are PEs. Figure SIMD-2a shows the resulting processing structure which is also known as a Processor Array. The Interconnection Network provides flexibility in choosing source and destination for data to and from the PEs, necessary in many algorithms. The I/O system plays the role of converting – typically at very high rates – input/output data between the format of the outside world and the internal format of the array. The design of the I/O system is highly application dependent. In order to provide overall control of the Processor Array, as well as to execute sequential operations, it is common to use a conventional Host Computer which then views the Processor Array as an attached processor as illustrated in figure SIMD-2b. 1.

Asserted Claims	Additional References and Disclosure
	 Processor Array Figure SIMD-2a  Host Computer - Processor Array Relationship Figure SIMD-2b 2. *As a non-limiting example, Glover discloses a controller controlling performance of the sequence of computations (e.g., via instructions) by at least one graphics processing unit (e.g., host computer and/or processing elements):*

Asserted Claims	Additional References and Disclosure
	 Figure 1: A block diagram of the MM32k.  Figure 2: A block diagram of a single processing element (PE).

Asserted Claims	Additional References and Disclosure
	845. *As a non-limiting example, Mesa-Martinez discloses a controller (e.g., array controller) controlling performance of the sequence of computations (e.g., via instructions) by at least one graphics processing unit (e.g., processing elements (PEs)):* The first generation UCSC Kestrel processor is implemented as a systolic array [8] of 512 processing elements (PEs). The system’s configuration as a linear array is ideal for such target applications as sequence analysis, at the same time that it provides a simpler and more flexible programming model than more complex multiple-instruction, multiple data stream (MIMD) machines would when applied to the same domain of problems [7]. For the applications that we were envisioning, programmability was more important than large local memories or reconfigurable interconnection fabrics [4]. We believe our SIMD approach to be a more cost effective solution than systems using fully reconfigurable general purpose multiprocessors [5], [6]. Each processing element in the array is 8-bits wide and contains its own local memory and datapath. The Kestrel board also contains several SDRAM banks that implement the instruction memory (IM) as well as the input/output queues (IQm, OQm) [3]. Instruction broadcasting and sequencing is performed by an on board controller, that is also in charge of memory management (Figure 3). The whole system is self contained and implemented as a single full length PCI card (Figure 2). Like most SIMD processors Kestrel broadcasts each instruction to each PE in the array. Due to the relatively large number of PEs that Kestrel has, and the fact that these PEs are distributed among several chips, instruction broadcasting can easily become more taxing than the actual time to execute the instruction. In order to solve this bottleneck Kestrel implements a board-level instruction issue and broadcast [4]. In order to handle inter-PE communication each PE shares a file of registers with its left and right neighbors. These register banks are known as Systolic Shared Registers (SSRs) [9]. Although this topology might seem limiting at first, it allows for computation and communication to occur concurrently. When a result is stored in the SSR after the instruction has been completed it is immediately visible to the next PE, thus communication takes place automatically (Figure 4). All addresses to the SSRs are issued globally in order to prevent adjacent PEs from writing to the same bank at the same time. This communication method is both elegant and efficient since it greatly simplifies the design of the PE and allowing most of its area to be dedicated for processing and memory.

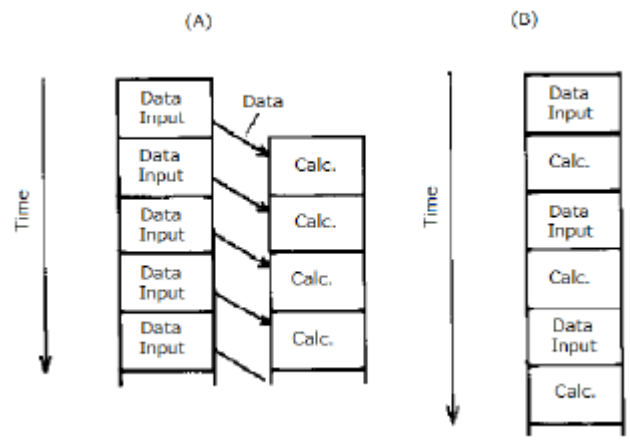
Asserted Claims	Additional References and Disclosure
	A-9.  Fig. 3. Kestrel board architecture. FIG. 3. The controller is implemented using an FPGA. This device serves as the instruction sequencer for the system. The controller also handles the board’s I/O by interacting with the onboard PCI interface chip. The program instructions received from the host computer are stored in a separate instruction memory (IM). The Kestrel controller is programmed by using 54 bits of the 96-bit instruction word. The remaining bits are used as the instruction to be broadcast to the array. Each cycle, the controller must decide whether or not data is read from the input queue, whether or not data is written to the output queue, and whether or not the immediate field in the instruction should be replaced with data from the controller. A small amount of local memory and logic is used by the controller in order to enable loop counting and the recirculation of data among the PEs. When a critical condition is reached by the system the controller will generate the corresponding interrupt to inform the host about the board’s condition.

Asserted Claims	Additional References and Disclosure
	A-10.
1[d][iii] “to transfer the at least a portion of the input data into the accelerator memory during performance of the intermediate computations in the sequence of computations by the at least one graphics processing unit, and”	*See* 1[c], 1[c][ii]. *As a non-limiting example, ANN discloses transferring at least a portion of input data (e.g., real-time video input data) into accelerator memory during performance of the intermediate computations in the sequence of computations by at least one graphics processing unit. In ANN’s multi-layer neural network, it would have been obvious to transfer subsequent frames of real-time input data into accelerator memory while performing a computation on a preceding frame:*

Asserted Claims	Additional References and Disclosure	
	Artificial Neural Network (ANN) is widely used in pattern recognition related area. In some case, the computational load is very heavy, in other case, real time process is required. So there is a need to apply a parallel algorithm on it, and usually the computation for ANN is inherently parallel. In this paper, graphic hardware is used to speed up the computation of ANN. In recent years, graphic processing unit (GPU) grows faster than CPU. Graphic hardware venders provide programmability on GPU. In this paper, application of commodity available GPU for two kinds of ANN models was explored. One is the self-organizing maps (SOM); the other is multi layer perceptron (MLP). The computation result shows that ANN computing on GPU is much faster than on standard CPU when the neural network is large. And some design rules for improve the efficiency on GPU are given. **Keywords: Graphic Process Unit; ANN; SOM; MLP** I. INTRODUCTION Artificial neural network is widely used in classification and pattern recognition. In this paper, two kinds of ANN, self-organizing maps (SOM) [1] and multi layer perceptron (MLP), are implemented on graphic hardware for speed up the computation. MLP is a very simple neural network, the process is linear with one input layer, several hidden layer, and an output layer. It is usually trained by back propagation (BP) algorithm. SOM consists of one layer of n-dimensional units (neurons). It is fully connected with the network input. Additionally, there exist lateral connections through which a topological structure is imposed. For the standard model, the topology is a regular two-dimensional map instantiated by connections between each unit and its direct neighbors. 622.	

Asserted Claims	Additional References and Disclosure
	Three layer MLP neural network is selected to recognize the ball. The input layer consists of seven nodes for seven characters. The hidden layer consists of three nodes, and the output layer consists of just one node. Back propagation method was chosen to train the network. The train set takes as follow. 16 locations are selected from each of 10 robot cars, 4 locations are selected from center of ball, and one location is selected from background field. Totally, 165 locations are selected as the train set. For each selected location, seven characters are calculated and take as an element of the train set. The trained MLP is used to recognize and trace the ball in real time. 624. *B. MLP Computation Time for CPU and GPU* The application of MLP in this paper is to trace the ball in robot soccer in real time. The result is shown in table 1. 625.

Asserted Claims	Additional References and Disclosure
	The result shows that GPU based MLP computation is about 200 times faster than that of CPU. And the result also shows that GPU computation is fast enough for the locating of the ball in real time. TABLE I: MLP COMPUTATION ON CPU AND GPU. CPU /ms: 11328 GPUN /ms: 46 *Id.* The implementation on graphic hardware introduce in this paper has other implicit benefit too. For the SOM computing, a multi-texture or 3-D texture can be used to store the map and make more general SOM computing without the restriction of the vector length of 4. For the MLP used in robot soccer, some graphic hardware have “video in” function, using this kind of graphic hardware; image information can be retrieved directly from camera and store on graphic memory, and don’t need to transfer data between CPU and GPU, which will speed the process. 626. *As a non-limiting example, Tamura discloses transferring at least a portion of input data into accelerator memory during performance of the intermediate computations in the sequence of computations by at least one graphics processing unit (e.g., inputting data and writing it to memory while other input data is processed through the layers of an artificial neural network):* In order to solve this problem, the present invention is a neuroprocessor comprising a neural network arithmetic processing unit and pairs of input signal memories and teacher signal memories, wherein input signal data and teacher signal data to be used in next calculations are inputted and written to the other of the pair of input signal memories and the other of the pair of teacher signal memories while learning

Asserted Claims	Additional References and Disclosure
	calculations are being performed using input signal data in one of the pair of input signal memories and teacher signal data in one of the pair of teacher signal memories. [0029]. In this configuration, the data input operation is carried out in parallel with calculations, so time for data input is apparently not necessary. In other words, the total processing time is reduced by the amount of time that was previously required for data input. Therefore, processing can be accelerated. [0030]. FIG. 3 (A) is a diagram showing processing flow. Data input and calculations are performed in parallel, and the calculations are performed using data inputted in the previous step. As can be seen in comparison with the prior art method in FIG. 3 (B), one processing iteration is completed in two steps in the prior art method in FIG. 3 (B), whereas in the method of the present invention, one processing iteration is apparently completed in one step. [0035].  FIG. 3.

Asserted Claims	Additional References and Disclosure
	As a non-limiting example, Oh discloses transferring at least a portion of input data into accelerator memory during performance of the intermediate computations in the sequence of computations by at least one graphics processing unit (e.g., while processing video input data, which may require real-time processing in which input data is processed through the layers of an artificial neural network): In the case of using a neural network (NN) for image processing and pattern recognition, the main problem is the computational complexity in the testing stage, which accounts for most of the processing time. Moreover, NN-based image convolution has to exhaustively scan an input image in order to process an entire image [3]. Although an NN can be simulated using software, many potential NN applications require real-time processing, which means fully parallel specially designed hardware implementations, such as an FPGA-based realization of an NN. However, this is somewhat expensive and involves extra design overheads [4]. 1311. Recently, researchers have attempted text-based retrieval of image and video data using several image processing techniques [3]. As such, an automatic text detection algorithm for image data and video documents is important as a preprocessing stage for optical character recognition, and an NN-based text detection method has several advantages over other methods [3]. Therefore, this paper briefly describes such a text detection method, and readers are referred to the author’s previous publication for more details [3]. In the proposed method, an NN is used to classify the pixels of input images, whereby the feature extraction and pattern recognition stage are integrated in the neural network. The NN then examines local regions looking for text pixels that may be contained in a text region. Therefore, an M × M pixel region in the image is received as the input and a classified image is generated as the output. After the pattern passes the network, the value of the output node is compared with a threshold value and the class of each pixel determined, resulting in a classified image. GPU-based pipelining processing is used to reduce the processing time, and the GPU’s performance is maximized by accumulating a large number of input vectors 1 to create a two-dimensional texture. The input layer then receives the grey values for the pixels at predefined positions inside an M×M window over the input image. Experiments were conducted using an 11 × 11 input window size, with the number of nodes in each hidden layer set at 30. As a result, the processing time for pixel classification was significantly reduced using a GPU. Fig. 2(b) shows the pixel classification result for the left input image, where a black pixel denotes a text pixel. The classification using a GPU produced almost the same result as without a GPU.

Asserted Claims	Additional References and Disclosure
	1313.
1[d][iv] “to transfer at least a portion of the output data from the accelerator memory to the main memory during performance of the intermediate computations in the sequence of computations by the at least one graphics processing unit.”	*As a non-limiting example, ANN discloses transferring at least a portion of output data from accelerator memory to main memory during performance of the intermediate computations in the sequence of computations by at least one graphics processing unit. It would have been obvious to transfer output data in accelerator memory from one pass through the artificial neural network to main memory while intermediate computations occur on other input data:*

Asserted Claims	Additional References and Disclosure	
	Artificial Neural Network (ANN) is widely used in pattern recognition related area. In some case, the computational load is very heavy, in other case, real time process is required. So there is a need to apply a parallel algorithm on it, and usually the computation for ANN is inherently parallel. In this paper, graphic hardware is used to speed up the computation of ANN. In recent years, graphic processing unit (GPU) grows faster than CPU. Graphic hardware venders provide programmability on GPU. In this paper, application of commodity available GPU for two kinds of ANN models was explored. One is the self-organizing maps (SOM); the other is multi layer perceptron (MLP). The computation result shows that ANN computing on GPU is much faster than on standard CPU when the neural network is large. And some design rules for improve the efficiency on GPU are given. **Keywords: Graphic Process Unit; ANN; SOM; MLP** I. INTRODUCTION Artificial neural network is widely used in classification and pattern recognition. In this paper, two kinds of ANN, self-organizing maps (SOM) [1] and multi layer perceptron (MLP), are implemented on graphic hardware for speed up the computation. MLP is a very simple neural network, the process is linear with one input layer, several hidden layer, and an output layer. It is usually trained by back propagation (BP) algorithm. SOM consists of one layer of n-dimensional units (neurons). It is fully connected with the network input. Additionally, there exist lateral connections through which a topological structure is imposed. For the standard model, the topology is a regular two-dimensional map instantiated by connections between each unit and its direct neighbors. 622.	

Asserted Claims	Additional References and Disclosure
	Three layer MLP neural network is selected to recognize the ball. The input layer consists of seven nodes for seven characters. The hidden layer consists of three nodes, and the output layer consists of just one node. Back propagation method was chosen to train the network. The train set takes as follow. 16 locations are selected from each of 10 robot cars, 4 locations are selected from center of ball, and one location is selected from background field. Totally, 165 locations are selected as the train set. For each selected location, seven characters are calculated and take as an element of the train set. The trained MLP is used to recognize and trace the ball in real time. 624. *B. MLP Computation Time for CPU and GPU* The application of MLP in this paper is to trace the ball in robot soccer in real time. The result is shown in table 1. 625.

Asserted Claims	Additional References and Disclosure
	The result shows that GPU based MLP computation is about 200 times faster than that of CPU. And the result also shows that GPU computation is fast enough for the locating of the ball in real time. TABLE I: MLP COMPUTATION ON CPU AND GPU. CPU /ms: 11328 GPUN /ms: 46 *Id.* The implementation on graphic hardware introduce in this paper has other implicit benefit too. For the SOM computing, a multi-texture or 3-D texture can be used to store the map and make more general SOM computing without the restriction of the vector length of 4. For the MLP used in robot soccer, some graphic hardware have “video in” function, using this kind of graphic hardware; image information can be retrieved directly from camera and store on graphic memory, and don’t need to transfer data between CPU and GPU, which will speed the process. 626. *As a non-limiting example, Oh discloses transferring at least a portion of output data from accelerator memory to main memory during performance of the intermediate computations in the sequence of computations by at least one graphics processing unit (e.g., while processing video input data, which may require real-time processing in which input data is processed while output data is transferred):* In the case of using a neural network (NN) for image processing and pattern recognition, the main problem is the computational complexity in the testing stage, which accounts for most of the processing time. Moreover, NN-based image convolution has to exhaustively scan an input image in order to process an entire image [3]. Although an NN can be simulated using software, many potential NN applications require

Asserted Claims	Additional References and Disclosure
	real-time processing, which means fully parallel specially designed hardware implementations, such as an FPGA-based realization of an NN. However, this is somewhat expensive and involves extra design overheads [4]. 1311. Recently, researchers have attempted text-based retrieval of image and video data using several image processing techniques [3]. As such, an automatic text detection algorithm for image data and video documents is important as a preprocessing stage for optical character recognition, and an NN-based text detection method has several advantages over other methods [3]. Therefore, this paper briefly describes such a text detection method, and readers are referred to the author’s previous publication for more details [3]. In the proposed method, an NN is used to classify the pixels of input images, whereby the feature extraction and pattern recognition stage are integrated in the neural network. The NN then examines local regions looking for text pixels that may be contained in a text region. Therefore, an M × M pixel region in the image is received as the input and a classified image is generated as the output. After the pattern passes the network, the value of the output node is compared with a threshold value and the class of each pixel determined, resulting in a classified image. GPU-based pipelining processing is used to reduce the processing time, and the GPU’s performance is maximized by accumulating a large number of input vectors 1 to create a two-dimensional texture. The input layer then receives the grey values for the pixels at predefined positions inside an M×M window over the input image. Experiments were conducted using an 11 × 11 input window size, with the number of nodes in each hidden layer set at 30. As a result, the processing time for pixel classification was significantly reduced using a GPU. Fig. 2(b) shows the pixel classification result for the left input image, where a black pixel denotes a text pixel. The classification using a GPU produced almost the same result as without a GPU. 1313.
2 “The computer system of claim 1, wherein the central processing unit is configured to receive the input data in response to a user interaction.”	*See* claim 1. *As a non-limiting example, Oh discloses receiving input data in response to a user interaction (e.g., uploading data from an external device, such as a camera):* In the case of using a neural network (NN) for image processing and pattern recognition, the main problem is the computational complexity in the testing stage, which accounts for most of the processing time. Moreover, NN-based image convolution has to exhaustively scan an input image in order to process an

Asserted Claims	Additional References and Disclosure
	entire image [3]. Although an NN can be simulated using software, many potential NN applications require real-time processing, which means fully parallel specially designed hardware implementations, such as an FPGA-based realization of an NN. However, this is somewhat expensive and involves extra design overheads [4]. 1311. Recently, researchers have attempted text-based retrieval of image and video data using several image processing techniques [3]. As such, an automatic text detection algorithm for image data and video documents is important as a preprocessing stage for optical character recognition, and an NN-based text detection method has several advantages over other methods [3]. Therefore, this paper briefly describes such a text detection method, and readers are referred to the author’s previous publication for more details [3]. In the proposed method, an NN is used to classify the pixels of input images, whereby the feature extraction and pattern recognition stage are integrated in the neural network. The NN then examines local regions looking for text pixels that may be contained in a text region. Therefore, an M × M pixel region in the image is received as the input and a classified image is generated as the output. After the pattern passes the network, the value of the output node is compared with a threshold value and the class of each pixel determined, resulting in a classified image. GPU-based pipelining processing is used to reduce the processing time, and the GPU’s performance is maximized by accumulating a large number of input vectors 1 to create a two-dimensional texture. The input layer then receives the grey values for the pixels at predefined positions inside an M×M window over the input image. Experiments were conducted using an 11 × 11 input window size, with the number of nodes in each hidden layer set at 30. As a result, the processing time for pixel classification was significantly reduced using a GPU. Fig. 2(b) shows the pixel classification result for the left input image, where a black pixel denotes a text pixel. The classification using a GPU produced almost the same result as without a GPU. 1313. The bias term addition and sigmoid operation can be performed in one rendering pass. The bias texture and texture that contains the result of the matrix multiplication, texture W ×X , are set as the active texture. The vertex shader then outputs a full-screen rectangle as before. Each vertex’s texture coordinate for the texture W × X correspond to its position. For example, the upper left vertex has the texture coordinate (0; 0), while the texture coordinate for the upper right vertex is (1; 0). As the bias term is identical for one row, the bias

Asserted Claims	Additional References and Disclosure
	term matrix is one-dimensional and the bias texture coordinates for each vertex correspond to its vertical position. The pixel shader adds two textures and performs a sigmoid operation. If there is more than one layer in an NN, the above procedure is repeated for each layer. The result of the previous layer is saved in the form of a render target texture, which is then used as an input for the next layer. Note that, even though an NN may have multiple layers, the GPU can perform all the operations after texture creation. 1313.
3[a] “The computer system of claim 1, wherein: the central processing unit is configured to receive the input data at a first rate; and”	*See* claim 1. *As a non-limiting example, ANN discloses receiving input data (e.g., video input data) at a first rate:*

Asserted Claims	Additional References and Disclosure	
	Artificial Neural Network (ANN) is widely used in pattern recognition related area. In some case, the computational load is very heavy, in other case, real time process is required. So there is a need to apply a parallel algorithm on it, and usually the computation for ANN is inherently parallel. In this paper, graphic hardware is used to speed up the computation of ANN. In recent years, graphic processing unit (GPU) grows faster than CPU. Graphic hardware venders provide programmability on GPU. In this paper, application of commodity available GPU for two kinds of ANN models was explored. One is the self-organizing maps (SOM); the other is multi layer perceptron (MLP). The computation result shows that ANN computing on GPU is much faster than on standard CPU when the neural network is large. And some design rules for improve the efficiency on GPU are given. **Keywords: Graphic Process Unit; ANN; SOM; MLP** I. INTRODUCTION Artificial neural network is widely used in classification and pattern recognition. In this paper, two kinds of ANN, self-organizing maps (SOM) [1] and multi layer perceptron (MLP), are implemented on graphic hardware for speed up the computation. MLP is a very simple neural network, the process is linear with one input layer, several hidden layer, and an output layer. It is usually trained by back propagation (BP) algorithm. SOM consists of one layer of n-dimensional units (neurons). It is fully connected with the network input. Additionally, there exist lateral connections through which a topological structure is imposed. For the standard model, the topology is a regular two-dimensional map instantiated by connections between each unit and its direct neighbors. 622.	

Asserted Claims	Additional References and Disclosure
	Three layer MLP neural network is selected to recognize the ball. The input layer consists of seven nodes for seven characters. The hidden layer consists of three nodes, and the output layer consists of just one node. Back propagation method was chosen to train the network. The train set takes as follow. 16 locations are selected from each of 10 robot cars, 4 locations are selected from center of ball, and one location is selected from background field. Totally, 165 locations are selected as the train set. For each selected location, seven characters are calculated and take as an element of the train set. The trained MLP is used to recognize and trace the ball in real time. 624. *B. MLP Computation Time for CPU and GPU* The application of MLP in this paper is to trace the ball in robot soccer in real time. The result is shown in table 1. 625.

Asserted Claims	Additional References and Disclosure
	The result shows that GPU based MLP computation is about 200 times faster than that of CPU. And the result also shows that GPU computation is fast enough for the locating of the ball in real time. TABLE I: MLP COMPUTATION ON CPU AND GPU. CPU /ms: 11328 GPUN /ms: 46 *Id.* The implementation on graphic hardware introduce in this paper has other implicit benefit too. For the SOM computing, a multi-texture or 3-D texture can be used to store the map and make more general SOM computing without the restriction of the vector length of 4. For the MLP used in robot soccer, some graphic hardware have “video in” function, using this kind of graphic hardware; image information can be retrieved directly from camera and store on graphic memory, and don’t need to transfer data between CPU and GPU, which will speed the process. 626. *As a non-limiting example, Oh discloses receiving input data (e.g., video data, which can be input at different rates):* In the case of using a neural network (NN) for image processing and pattern recognition, the main problem is the computational complexity in the testing stage, which accounts for most of the processing time. Moreover, NN-based image convolution has to exhaustively scan an input image in order to process an entire image [3]. Although an NN can be simulated using software, many potential NN applications require real-time processing, which means fully parallel specially designed hardware implementations, such as an

Asserted Claims	Additional References and Disclosure
	FPGA-based realization of an NN. However, this is somewhat expensive and involves extra design overheads [4]. 1311. Recently, researchers have attempted text-based retrieval of image and video data using several image processing techniques [3]. As such, an automatic text detection algorithm for image data and video documents is important as a preprocessing stage for optical character recognition, and an NN-based text detection method has several advantages over other methods [3]. Therefore, this paper briefly describes such a text detection method, and readers are referred to the author’s previous publication for more details [3]. In the proposed method, an NN is used to classify the pixels of input images, whereby the feature extraction and pattern recognition stage are integrated in the neural network. The NN then examines local regions looking for text pixels that may be contained in a text region. Therefore, an M × M pixel region in the image is received as the input and a classified image is generated as the output. After the pattern passes the network, the value of the output node is compared with a threshold value and the class of each pixel determined, resulting in a classified image. GPU-based pipelining processing is used to reduce the processing time, and the GPU’s performance is maximized by accumulating a large number of input vectors 1 to create a two-dimensional texture. The input layer then receives the grey values for the pixels at predefined positions inside an M×M window over the input image. Experiments were conducted using an 11 × 11 input window size, with the number of nodes in each hidden layer set at 30. As a result, the processing time for pixel classification was significantly reduced using a GPU. Fig. 2(b) shows the pixel classification result for the left input image, where a black pixel denotes a text pixel. The classification using a GPU produced almost the same result as without a GPU. 1313.
3[b] “the at least one graphics processing unit is configured to perform the sequence of computations at a second rate different than the first rate.”	*As a non-limiting example, ANN discloses at least one graphics processing unit performing the sequence of computations a second rate different than the first rate (e.g., different rates depending on the device and the number of passes):*

Asserted Claims	Additional References and Disclosure
	The result shows that GPU based MLP computation is about 200 times faster than that of CPU. And the result also shows that GPU computation is fast enough for the locating of the ball in real time. TABLE I: MLP COMPUTATION ON CPU AND GPU. CPU /ms: 11328 GPUN /ms: 46 V. SOME IMPLEMENTATION DETAILS AND LESSONS To increase the efficiency, some basic rule should obey. The main rules are given below. First, create the GPU hardware program only once and enable it when it is needed. The reason is that when a new program is created, it will be compiled by Cg, which is time consuming. Second, if possible, do one’s best to decrease the calculation passes. For computation scheme described in section 2.2, the main bottleneck is at finding minimum procedure. The test shows that the bottleneck comes from multi-pass used in finding minimum procedure. Table 2 shows result for different scheme of finding the minimum. To decrease the pass, we make two changes, the first is to combine the value calculation with one pass of find minimum, and the second is to combine two pass of find minimum computation into one pass. Instead of calculate 4 units, 16 units are calculated. GPUA represent calculation on ATI graphic card and GPUN represent the calculation on NVIDIA graphic card. The performance increase is clear, especially for Nvidia’s graphic card. TABLE II: COMPARISON BETWEEN MORE AND FEW PASS KFM size: 128*128, 256*256, 512*512 GPUA more pass /ms: 366, 400, 533 GPUA few pass /ms: 211, 244, 511 GPUN more pass /ms: 190, 640, 2889 GPUN few pass /ms: 104, 256, 900 625.

Asserted Claims	Additional References and Disclosure
	As a non-limiting example, Oh discloses at least one graphics processing unit performing the sequence of computations a second rate different than the first rate (e.g., different rates depending on the pixel region selected for processing): Recently, researchers have attempted text-based retrieval of image and video data using several image processing techniques [3]. As such, an automatic text detection algorithm for image data and video documents is important as a preprocessing stage for optical character recognition, and an NN-based text detection method has several advantages over other methods [3]. Therefore, this paper briefly describes such a text detection method, and readers are referred to the author’s previous publication for more details [3]. In the proposed method, an NN is used to classify the pixels of input images, whereby the feature extraction and pattern recognition stage are integrated in the neural network. The NN then examines local regions looking for text pixels that may be contained in a text region. Therefore, an M × M pixel region in the image is received as the input and a classified image is generated as the output. After the pattern passes the network, the value of the output node is compared with a threshold value and the class of each pixel determined, resulting in a classified image. GPU-based pipelining processing is used to reduce the processing time, and the GPU’s performance is maximized by accumulating a large number of input vectors 1 to create a two-dimensional texture. The input layer then receives the grey values for the pixels at predefined positions inside an M×M window over the input image. Experiments were conducted using an 11 × 11 input window size, with the number of nodes in each hidden layer set at 30. As a result, the processing time for pixel classification was significantly reduced using a GPU. Fig. 2(b) shows the pixel classification result for the left input image, where a black pixel denotes a text pixel. The classification using a GPU produced almost the same result as without a GPU. 1313.
4 “The computer system of claim 1, wherein the main memory is configured to store a copy of the output data stored in the accelerator memory.”	*See* 1[d][iv], claim 1.

Asserted Claims	Additional References and Disclosure
5 “The computer system of claim 1, wherein an output of at least one computation in the sequence of computations represents an output of at least one neuron in an artificial neural network.”	*See* 1[c][i], claim 1. *As a non-limiting example, ANN discloses that an output of at least one computation in the sequence of computations represents an output of at least one neuron (e.g., nodes) in an artificial neural network:*

Asserted Claims	Additional References and Disclosure	
	Artificial Neural Network (ANN) is widely used in pattern recognition related area. In some case, the computational load is very heavy, in other case, real time process is required. So there is a need to apply a parallel algorithm on it, and usually the computation for ANN is inherently parallel. In this paper, graphic hardware is used to speed up the computation of ANN. In recent years, graphic processing unit (GPU) grows faster than CPU. Graphic hardware venders provide programmability on GPU. In this paper, application of commodity available GPU for two kinds of ANN models was explored. One is the self-organizing maps (SOM); the other is multi layer perceptron (MLP). The computation result shows that ANN computing on GPU is much faster than on standard CPU when the neural network is large. And some design rules for improve the efficiency on GPU are given. **Keywords: Graphic Process Unit; ANN; SOM; MLP** I. INTRODUCTION Artificial neural network is widely used in classification and pattern recognition. In this paper, two kinds of ANN, self-organizing maps (SOM) [1] and multi layer perceptron (MLP), are implemented on graphic hardware for speed up the computation. MLP is a very simple neural network, the process is linear with one input layer, several hidden layer, and an output layer. It is usually trained by back propagation (BP) algorithm. SOM consists of one layer of n-dimensional units (neurons). It is fully connected with the network input. Additionally, there exist lateral connections through which a topological structure is imposed. For the standard model, the topology is a regular two-dimensional map instantiated by connections between each unit and its direct neighbors. 622.	

Asserted Claims	Additional References and Disclosure
	In this paper, two kinds of ANN computation on GPU are given. In section 2, SOM computation model and implementation on graphic hardware is discussed. In section 3, MLP computation model and implementation on GPU is discussed. In section 4, the computation result and comparison are given for both CPU and GPU. In section 5, some of the design details and lessons we learned during implementation are given. In section 6, conclusion and some future works are given. *Id.* *B.Computational Model of MLP* Three layer MLP neural network is selected to recognize the ball. The input layer consists of seven nodes for seven characters. The hidden layer consists of three nodes, and the output layer consists of just one node. Back propagation method was chosen to train the network. The train set takes as follow. 16 locations are selected from each of 10 robot cars, 4 locations are selected from center of ball, and one location is selected from background field. Totally, 165 locations are selected as the train set. For each selected location, seven characters are calculated and take as an element of the train set. The trained MLP is used to recognize and trace the ball in real time. 624.

Asserted Claims	Additional References and Disclosure
	Current GPU has a limited instruction length and a limited number of temporary variables for calculation at each location. So multipass is needed for complex problem. For MLP discussed in this section, three passes are performed. In the first pass, average values of three color and luminance are calculated. In the second pass, the standard deviations of the three colors are calculated. In the third pass, classification result is got from MLP calculation on the characteristic. 624. In this calculation, we use a new Nvidia’s GF6000 serious graphic hardware. The reason is that the ATI’s GPU supports very few numbers of operations in each pass; so more passes are need for MLP calculation. Old Nvidia’s GPU does not support fully float texture, which is crucial for the precision of result. 625.

Asserted Claims	Additional References and Disclosure
	Second, if possible, do one’s best to decrease the calculation passes. For computation scheme described in section 2.2, the main bottleneck is at finding minimum procedure. The test shows that the bottleneck comes from multi-pass used in finding minimum procedure. Table 2 shows result for different scheme of finding the minimum. To decrease the pass, we make two changes, the first is to combine the value calculation with one pass of find minimum, and the second is to combine two pass of find minimum computation into one pass. Instead of calculate 4 units, 16 units are calculated. GPUA represent calculation on ATI graphic card and GPUN represent the calculation on NVIDIA graphic card. The performance increase is clear, especially for Nvidia’s graphic card. TABLE II: COMPARISON BETWEEN MORE AND FEW PASS KFM size | 128*128 | 256*256 | 512*512 GPUA more pass /ms | 366 | 400 | 533 GPUA few pass /ms | 211 | 244 | 511 GPUN more pass /ms | 190 | 640 | 2889 GPUN few pass /ms | 104 | 256 | 900 Third, do best to decrease the data exchange between CPU and GPU. Usually OpenGL’s PBuffer are used to save the intermediate result in a texture on GPU and reuse it as an input data. Harris[3] had created a class called “Render to Texture” to easy the use of PBuffer. We had used this class in our program. *Id.*

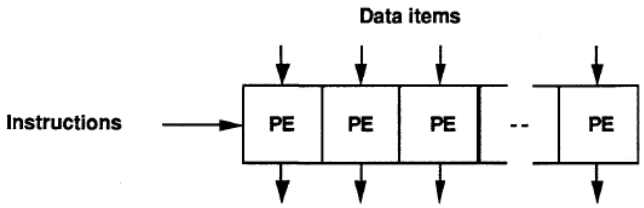
Asserted Claims	Additional References and Disclosure
	In this paper, implementation for two ANN models on graphic hardware is given. Inherent parallelism of commodity graphic hardware is used to accelerate the computation of ANN. The result shows that GPU is capable for some of ANN calculation, the graphic hardware make it possible for an increasing performance/cost ratio on the area of large size ANN computation. Compared to Bohn’s initial computation on SGI workstation, our implementation has two benefits. One is our calculation is more precise, for we had use the float point computing. The other is that we only use a commodity available graphic card, which is much more easily available than SGI workstation so can be widely used. The implementation on graphic hardware introduce in this paper has other implicit benefit too. For the SOM computing, a multi-texture or 3-D texture can be used to store the map and make more general SOM computing without the restriction of the vector length of 4. For the MLP used in robot soccer, some graphic hardware have “video in” function, using this kind of graphic hardware; image information can be retrieved directly from camera and store on graphic memory, and don’t need to transfer data between CPU and GPU, which will speed the process. We can also do other general ANN computation on GPU, because GPU provide almost all arithmetic operation, logic operation and some important mathematic function. And for the application of neural network on images, it is more naive to make such computing on a graphic hardware. 626.

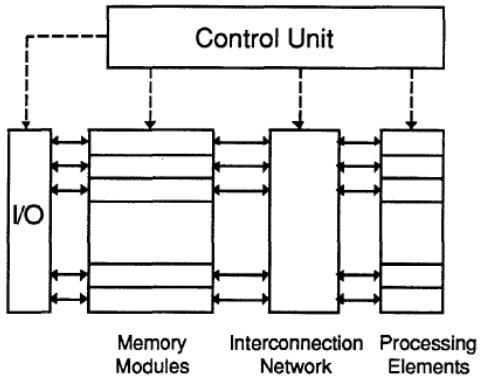
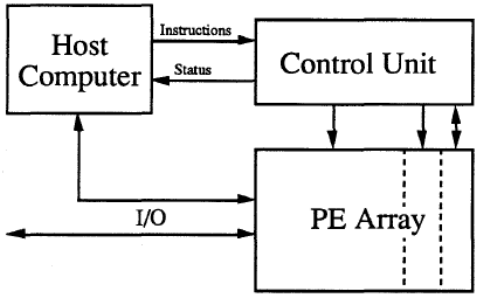
Asserted Claims	Additional References and Disclosure
	As a non-limiting example, Oh discloses that an output of at least one computation in the sequence of computations represents an output of at least one neuron (e.g., nodes) in an artificial neural network: An artificial neural network, usually referred to as ‘neural network’, is based on the concept of the workings of the human brain. There are many different types of NN, with the more popular being a multilayer perceptron, learning vector quantization, radial basis function, Hop/eld, and Kohonen. The current study focuses on using a GPU to implement a multilayer perceptron, which is usually fully connected between adjacent layers. The input layer receives the input features of a given application. Although the network structure can vary as regards the number of layers, number of nodes in each layer, and input mask size, each layer performs the same inner-product operation between the given input vectors and the weight vectors, followed by a non-linear function. Moreover, many inner-product operations can be replaced with a matrix multiplication, which is more appropriate for GPU implementation. 1311–12. If there is more than one layer in an NN, the above procedure is repeated for each layer. The result of the previous layer is saved in the form of a render target texture, which is then used as an input for the next layer. Note that, even though an NN may have multiple layers, the GPU can perform all the operations after texture creation. 1312. As described above, the inner-product operation for each layer of an NN can be replaced with a matrix multiplication based on accumulating the input vectors and weight vectors. As such, the computation-per-layer can be written as follows:

Asserted Claims	Additional References and Disclosure
	$$W = \begin{bmatrix} w_{11} & w_{12} & w_{13} & \dots & w_{1N} \\ w_{21} & w_{22} & w_{23} & \dots & w_{2N} \\ \dots & \dots & \dots & \dots & \dots \\ w_{M1} & w_{M2} & w_{M3} & \dots & w_{MN} \end{bmatrix} = \begin{bmatrix} W_1 \\ W_2 \\ \dots \\ W_M \end{bmatrix},$$ $$X = \begin{bmatrix} x_{11} & x_{12} & x_{13} & \dots & x_{1L} \\ x_{21} & x_{22} & x_{23} & \dots & x_{2L} \\ \dots & \dots & \dots & \dots & \dots \\ x_{N1} & x_{N2} & x_{N3} & \dots & x_{NL} \end{bmatrix}$$ $$= \begin{bmatrix} X_1 & X_2 & X_3 & \cdots & X_L \end{bmatrix},$$ $$B = \begin{bmatrix} b_1 & b_1 & b_1 & \dots & b_1 \\ b_2 & b_2 & b_2 & \dots & b_2 \\ \dots & \dots & \dots & \dots & \dots \\ b_M & b_M & b_M & \dots & b_M \end{bmatrix},$$

Asserted Claims	Additional References and Disclosure
	$M = W \times X + B$ $= \begin{bmatrix} W_1 \cdot X_1 & W_1 \cdot X_2 & W_1 \cdot X_3 & \dots & W_1 \cdot X_N \\ W_2 \cdot X_1 & W_2 \cdot X_2 & W_2 \cdot X_3 & \dots & W_2 \cdot X_N \\ \dots & \dots & \dots & \dots & \dots \\ W_M \cdot X_1 & W_M \cdot X_2 & W_M \cdot X_3 & \dots & W_M \cdot X_N \end{bmatrix}$ $+ \begin{bmatrix} b_1 & b_1 & b_1 & \dots & b_1 \\ b_2 & b_2 & b_2 & \dots & b_2 \\ \dots & \dots & \dots & \dots & \dots \\ b_M & b_M & b_M & \dots & b_M \end{bmatrix}$ $= \begin{bmatrix} m_{11} & m_{12} & m_{13} & \dots & m_{1L} \\ m_{21} & m_{22} & m_{23} & \dots & m_{2L} \\ m_{31} & m_{32} & m_{33} & \dots & m_{3L} \\ \dots & \dots & \dots & \dots & \dots \\ m_{M1} & m_{M2} & m_{M3} & \dots & m_{ML} \end{bmatrix}, \quad (2)$ $R = \text{sigmoid}(M)$ $= \begin{bmatrix} 1+e^{-m_{11}} & 1+e^{-m_{12}} & 1+e^{-m_{13}} & \dots & 1+e^{-m_{1L}} \\ 1+e^{-m_{21}} & 1+e^{-m_{22}} & 1+e^{-m_{23}} & \dots & 1+e^{-m_{2L}} \\ \dots & \dots & \dots & \dots & \dots \\ 1+e^{-m_{M1}} & 1+e^{-m_{M2}} & 1+e^{-m_{M3}} & \dots & 1+e^{-m_{ML}} \end{bmatrix}, \quad (3)$ where w_{ij} denotes the weight at the connection between the *i*th node of the output layer and the *j*th node of the input layer, M is the number of nodes in the output layer, and N is the number of nodes in the input layer. In addition, x_{ij} is the *i*th feature value of the *j*th input vector and bi is the bias term for the *i*th output node from L input vectors. The final result R_{ij} is the output of the *i*th output node for the *j*th input vector. 1312.

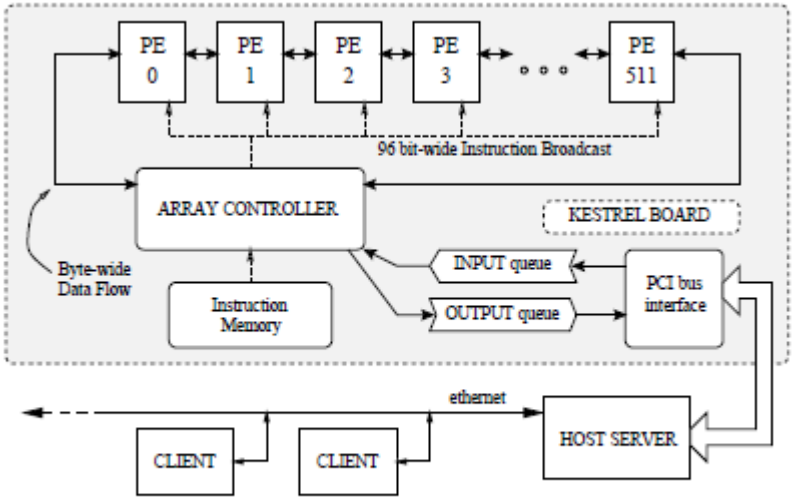
Asserted Claims	Additional References and Disclosure
	The above computation comprises of a matrix multiplication followed by a bias factor addition and sigmoid operation. The matrix multiplication is explained first. The method proposed by Moravanszky [1] is used to implement the matrix multiplication. The two matrices are converted into textures, denoted by texture W and texture X , then the matrix multiplication is performed by rendering. A rectangle is rendered to cover the whole screen. The vertex shader outputs the position and texture coordinates for each vertex of the rectangle, where each vertex has two texture coordinates: one for the row of texture W and the other for the column of texture X . For example, the upper left vertex will have the texture coordinates of the first row of texture W and the first column of texture X , while the upper right vertex will have the texture coordinates of the first row of texture W and the last column of texture X , and so on. As a result of the vertex shader, every pixel (i; j) has texture coordinates corresponding to the *i*th row of W and the *j*th column of X . The pixel shader then performs the inner-product between the row of W and the column of X specified by the texture coordinates. Fig. 1 shows an example of matrix multiplication using a GPU. The number of rendering passes required for matrix multiplication depends on the capability of the GPU, including the number of pixel shader operations and number of texture load operations. 1312–13. Therefore, this paper briefly describes such a text detection method, and readers are referred to the author’s previous publication for more details [3]. In the proposed method, an NN is used to classify the pixels of input images, whereby the feature extraction and pattern recognition stage are integrated in the neural network. The NN then examines local regions looking for text pixels that may be contained in a text region. Therefore, an M × M pixel region in the image is received as the input and a classified image is generated as the output. After the pattern passes the network, the value of the output node is compared with a threshold value and the class of each pixel determined, resulting in a classified image. GPU-based pipelining processing is used to reduce the processing time, and the GPU’s performance is maximized by accumulating a large number of input vectors 1 to create a two-dimensional texture. The input layer then receives the grey values for the pixels at predefined positions inside an M×M window over the input image. Experiments were conducted using an 11 × 11 input window size, with the number of nodes in each hidden layer set at 30. As a result, the processing time for pixel classification was significantly reduced using a GPU. Fig. 2(b) shows the pixel classification result for the left input image, where a black pixel denotes a text pixel. The classification using a GPU produced almost the same result as without a GPU. 1313.

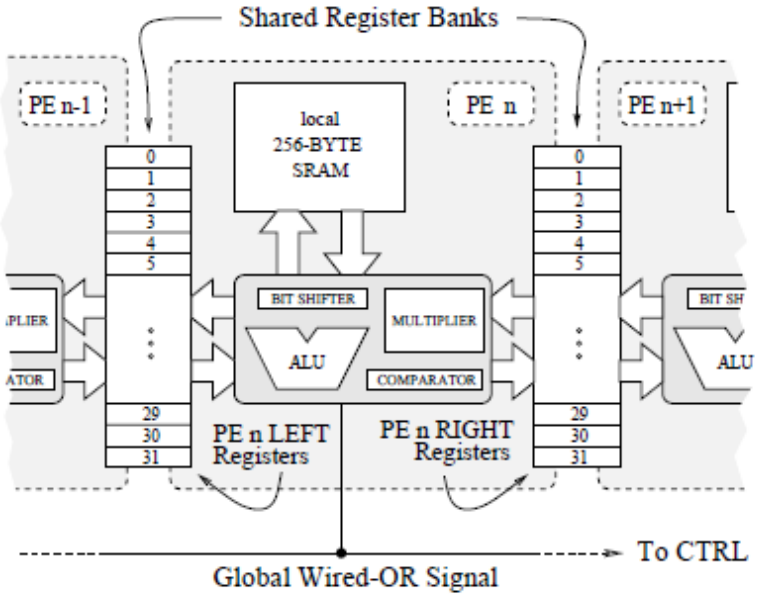
Asserted Claims	Additional References and Disclosure
6[a] “The computer system of claim 1, wherein accelerator memory comprises: a first memory bank to store parameters common to all of the computations in the sequence of computations; and”	*As a non-limiting example, Svensson discloses a first memory bank (e.g., one or more memory modules) to store parameters common to all of the computations in the sequence of computations:* ***1. Principles of SIMD*** In Single Instruction stream, Multiple Data stream (SIMD) processors one instruction works on several data items simultaneously by using several Processing Elements (PEs), all of which carry out the same operation as illustrated in Figure SIMD-1.  Principle of SIMD processor. Figure SIMD-1. A SIMD Processor has a single Control Unit reading instructions pointed to by a single Program Counter, decoding them and sending control signals to the PEs. Data are to be supplied to, and derived from, the PEs by a memory with as many data paths as there are PEs. Figure SIMD-2a shows the resulting processing structure which is also known as a Processor Array. The Interconnection Network provides flexibility in choosing source and destination for data to and from the PEs, necessary in many algorithms. The I/O system plays the role of converting – typically at very high rates – input/output data between the format of the outside world and the internal format of the array. The design of the I/O system is highly application dependent. In order to provide overall control of the Processor Array, as well as to execute sequential operations, it is common to use a conventional Host Computer which then views the Processor Array as an attached processor as illustrated in figure SIMD-2b. 1.

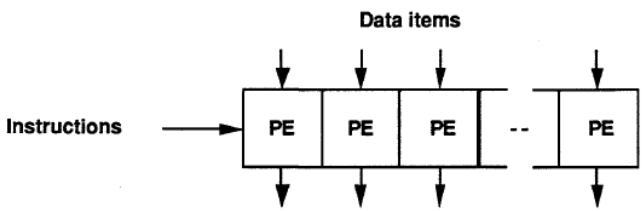
Asserted Claims	Additional References and Disclosure
	 Processor Array Figure SIMD-2a  Host Computer - Processor Array Relationship Figure SIMD-2b 2. *As a non-limiting example, Mesa-Martinez discloses a first memory bank (e.g., one or more local memories and/or Shared Register Banks):*

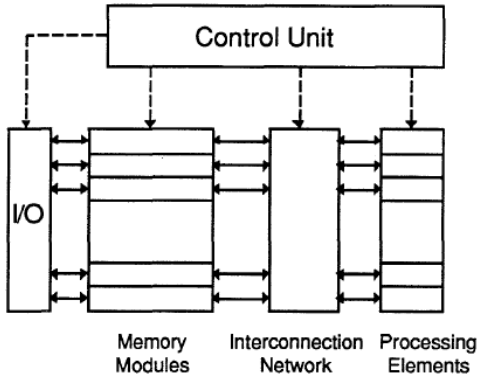
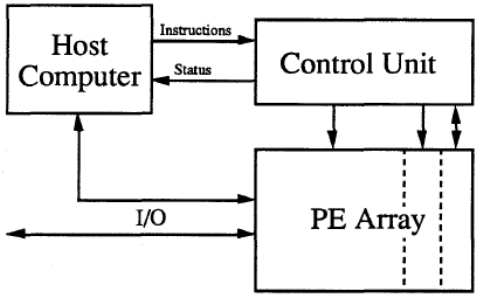
Asserted Claims	Additional References and Disclosure
	Kestrel Architecture The first generation UCSC Kestrel processor is implemented as a systolic array [8] of 512 processing elements (PEs). The system’s configuration as a linear array is ideal for such target applications as sequence analysis, at the same time that it provides a simpler and more flexible programming model than more complex multiple-instruction, multiple data stream (MIMD) machines would when applied to the same domain of problems [7]. For the applications that we were envisioning, programmability was more important than large local memories or reconfigurable interconnection fabrics [4]. We believe our SIMD approach to be a more cost effective solution than systems using fully reconfigurable general purpose multiprocessors [5], [6]. Each processing element in the array is 8-bits wide and contains its own local memory and datapath. The Kestrel board also contains several SDRAM banks that implement the instruction memory (IM) as well as the input/output queues (IQm, OQm) [3]. Instruction broadcasting and sequencing is performed by an on board controller, that is also in charge of memory management (Figure 3). The whole system is self contained and implemented as a single full length PCI card (Figure 2). Like most SIMD processors Kestrel broadcasts each instruction to each PE in the array. Due to the relatively large number of PEs that Kestrel has, and the fact that these PEs are distributed among several chips, instruction broadcasting can easily become more taxing than the actual time to execute the instruction. In order to solve this bottleneck Kestrel implements a board-level instruction issue and broadcast [4]. In order to handle inter-PE communication each PE shares a file of registers with its left and right neighbors. These register banks are known as Systolic Shared Registers (SSRs) [9]. Although this topology might seem limiting at first, it allows for computation and communication to occur concurrently. When a result is stored in the SSR after the instruction has been completed it is immediately visible to the next PE, thus communication takes place automatically (Figure 4). All addresses to the SSRs are issued globally in order to prevent adjacent PEs from writing to the same bank at the same time. This communication method is both elegant and efficient since it greatly simplifies the design of the PE and allowing most of its area to be dedicated for processing and memory. Mesa-Martinez at 2.

Asserted Claims	Additional References and Disclosure
	The first generation UCSC Kestrel processor is implemented as a systolic array [8] of 512 processing elements (PEs). The system’s configuration as a linear array is ideal for such target applications as sequence analysis, at the same time that it provides a simpler and more flexible programming model than more complex multiple-instruction, multiple data stream (MIMD) machines would when applied to the same domain of problems [7]. For the applications that we were envisioning, programmability was more important than large local memories or reconfigurable interconnection fabrics [4]. We believe our SIMD approach to be a more cost effective solution than systems using fully reconfigurable general purpose multiprocessors [5], [6]. Each processing element in the array is 8-bits wide and contains its own local memory and datapath. The Kestrel board also contains several SDRAM banks that implement the instruction memory (IM) as well as the input/output queues (IQm, OQm) [3]. Instruction broadcasting and sequencing is performed by an on board controller, that is also in charge of memory management (Figure 3). The whole system is self contained and implemented as a single full length PCI card (Figure 2). Like most SIMD processors Kestrel broadcasts each instruction to each PE in the array. Due to the relatively large number of PEs that Kestrel has, and the fact that these PEs are distributed among several chips, instruction broadcasting can easily become more taxing than the actual time to execute the instruction. In order to solve this bottleneck Kestrel implements a board-level instruction issue and broadcast [4]. In order to handle inter-PE communication each PE shares a file of registers with its left and right neighbors. These register banks are known as Systolic Shared Registers (SSRs) [9]. Although this topology might seem limiting at first, it allows for computation and communication to occur concurrently. When a result is stored in the SSR after the instruction has been completed it is immediately visible to the next PE, thus communication takes place automatically (Figure 4). All addresses to the SSRs are issued globally in order to prevent adjacent PEs from writing to the same bank at the same time. This communication method is both elegant and efficient since it greatly simplifies the design of the PE and allowing most of its area to be dedicated for processing and memory. A-9.

Asserted Claims	Additional References and Disclosure
	 Fig. 3. Kestrel board architecture. FIG. 3.

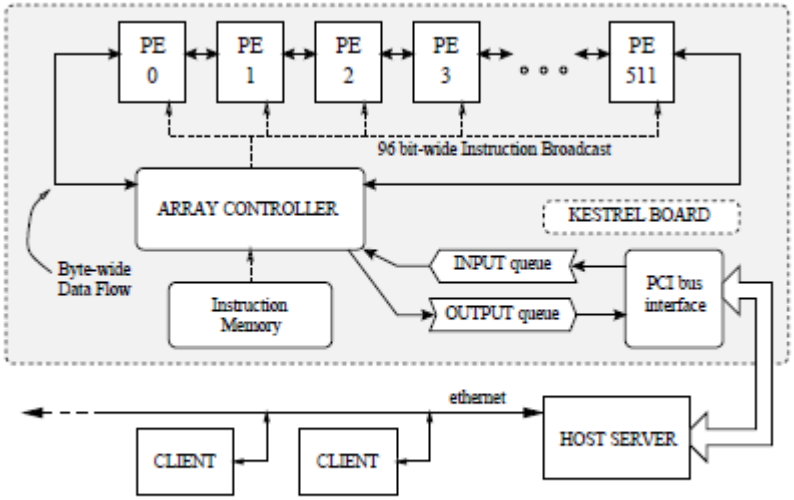
Asserted Claims	Additional References and Disclosure
	 Fig. 4. Kestrel processing element overview. FIG. 4.
6[b] “a second memory bank to store data specific to at least one computation in the sequence of computations.”	*As a non-limiting example, Svensson discloses a second memory bank (e.g., one or more memory modules) to store data specific to at least one computation in the sequence of computations:*

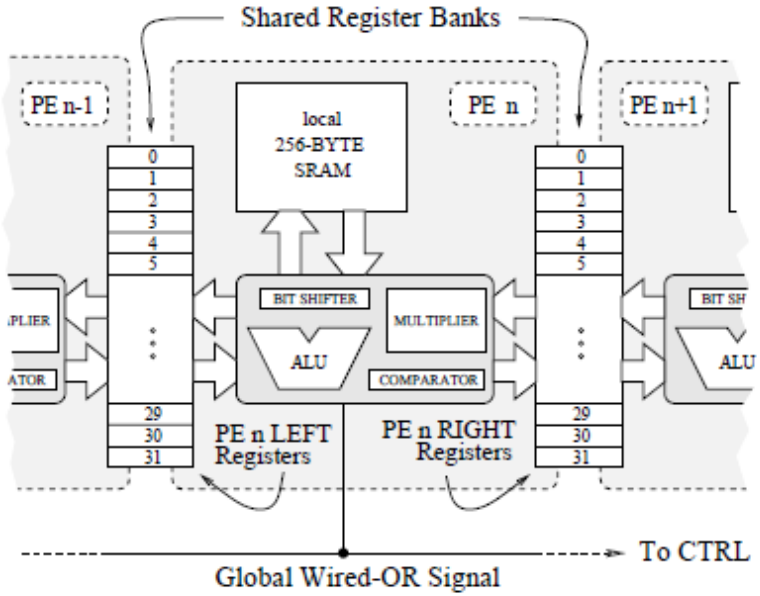
Asserted Claims	Additional References and Disclosure
	1. Principles of SIMD In Single Instruction stream, Multiple Data stream (SIMD) processors one instruction works on several data items simultaneously by using several Processing Elements (PEs), all of which carry out the same operation as illustrated in Figure SIMD-1.  Principle of SIMD processor. Figure SIMD-1. A SIMD Processor has a single Control Unit reading instructions pointed to by a single Program Counter, decoding them and sending control signals to the PEs. Data are to be supplied to, and derived from, the PEs by a memory with as many data paths as there are PEs. Figure SIMD-2a shows the resulting processing structure which is also known as a Processor Array. The Interconnection Network provides flexibility in choosing source and destination for data to and from the PEs, necessary in many algorithms. The I/O system plays the role of converting – typically at very high rates – input/output data between the format of the outside world and the internal format of the array. The design of the I/O system is highly application dependent. In order to provide overall control of the Processor Array, as well as to execute sequential operations, it is common to use a conventional Host Computer which then views the Processor Array as an attached processor as illustrated in figure SIMD-2b. 1.

Asserted Claims	Additional References and Disclosure
	 Processor Array Figure SIMD-2a  Host Computer - Processor Array Relationship Figure SIMD-2b 2. *As a non-limiting example, Mesa-Martinez discloses a second memory bank (e.g., one or more local memories and/or Shared Register Banks):*

Asserted Claims	Additional References and Disclosure
	Kestrel Architecture The first generation UCSC Kestrel processor is implemented as a systolic array [8] of 512 processing elements (PEs). The system’s configuration as a linear array is ideal for such target applications as sequence analysis, at the same time that it provides a simpler and more flexible programming model than more complex multiple-instruction, multiple data stream (MIMD) machines would when applied to the same domain of problems [7]. For the applications that we were envisioning, programmability was more important than large local memories or reconfigurable interconnection fabrics [4]. We believe our SIMD approach to be a more cost effective solution than systems using fully reconfigurable general purpose multiprocessors [5], [6]. Each processing element in the array is 8-bits wide and contains its own local memory and datapath. The Kestrel board also contains several SDRAM banks that implement the instruction memory (IM) as well as the input/output queues (IQm, OQm) [3]. Instruction broadcasting and sequencing is performed by an on board controller, that is also in charge of memory management (Figure 3). The whole system is self contained and implemented as a single full length PCI card (Figure 2). Like most SIMD processors Kestrel broadcasts each instruction to each PE in the array. Due to the relatively large number of PEs that Kestrel has, and the fact that these PEs are distributed among several chips, instruction broadcasting can easily become more taxing than the actual time to execute the instruction. In order to solve this bottleneck Kestrel implements a board-level instruction issue and broadcast [4]. In order to handle inter-PE communication each PE shares a file of registers with its left and right neighbors. These register banks are known as Systolic Shared Registers (SSRs) [9]. Although this topology might seem limiting at first, it allows for computation and communication to occur concurrently. When a result is stored in the SSR after the instruction has been completed it is immediately visible to the next PE, thus communication takes place automatically (Figure 4). All addresses to the SSRs are issued globally in order to prevent adjacent PEs from writing to the same bank at the same time. This communication method is both elegant and efficient since it greatly simplifies the design of the PE and allowing most of its area to be dedicated for processing and memory. Mesa-Martinez at 2.

Asserted Claims	Additional References and Disclosure
	The first generation UCSC Kestrel processor is implemented as a systolic array [8] of 512 processing elements (PEs). The system’s configuration as a linear array is ideal for such target applications as sequence analysis, at the same time that it provides a simpler and more flexible programming model than more complex multiple-instruction, multiple data stream (MIMD) machines would when applied to the same domain of problems [7]. For the applications that we were envisioning, programmability was more important than large local memories or reconfigurable interconnection fabrics [4]. We believe our SIMD approach to be a more cost effective solution than systems using fully reconfigurable general purpose multiprocessors [5], [6]. Each processing element in the array is 8-bits wide and contains its own local memory and datapath. The Kestrel board also contains several SDRAM banks that implement the instruction memory (IM) as well as the input/output queues (IQm, OQm) [3]. Instruction broadcasting and sequencing is performed by an on board controller, that is also in charge of memory management (Figure 3). The whole system is self contained and implemented as a single full length PCI card (Figure 2). Like most SIMD processors Kestrel broadcasts each instruction to each PE in the array. Due to the relatively large number of PEs that Kestrel has, and the fact that these PEs are distributed among several chips, instruction broadcasting can easily become more taxing than the actual time to execute the instruction. In order to solve this bottleneck Kestrel implements a board-level instruction issue and broadcast [4]. In order to handle inter-PE communication each PE shares a file of registers with its left and right neighbors. These register banks are known as Systolic Shared Registers (SSRs) [9]. Although this topology might seem limiting at first, it allows for computation and communication to occur concurrently. When a result is stored in the SSR after the instruction has been completed it is immediately visible to the next PE, thus communication takes place automatically (Figure 4). All addresses to the SSRs are issued globally in order to prevent adjacent PEs from writing to the same bank at the same time. This communication method is both elegant and efficient since it greatly simplifies the design of the PE and allowing most of its area to be dedicated for processing and memory. A-9.

Asserted Claims	Additional References and Disclosure
	 Fig. 3. Kestrel board architecture. FIG. 3.

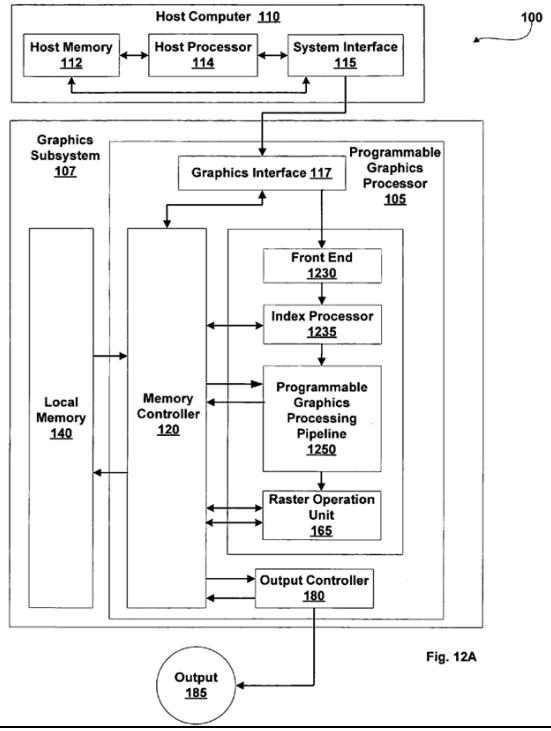
Asserted Claims	Additional References and Disclosure
	 Fig. 4. Kestrel processing element overview. FIG. 4.
7 “The computer system of claim 1, wherein the controller is configured to transfer the output data from the accelerator memory to the main memory without transferring any of the intermediate results from the accelerator memory to the main memory so as	*See* 1[d][iv], claim 1. *As a non-limiting example, ANN discloses transferring output data from accelerator memory to main memory without transferring any intermediate results from accelerator memory to main memory (e.g., saving intermediate results in textures on GPU) so as to reduce data transfer via the bus (e.g., decrease data exchange between CPU and GPU):* Third, do best to decrease the data exchange between CPU and GPU. Usually OpenGL’s PBuffer are used to save the intermediate result in a texture on GPU and reuse it as an input data. Harris[3] had created a class called “Render to Texture” to easy the use of PBuffer. We had used this class in our program.

Asserted Claims	Additional References and Disclosure
to reduce data transfer via the bus.”	625. *As a non-limiting example, Oh discloses transferring output data from accelerator memory to main memory without transferring any intermediate results from accelerator memory to main memory (e.g., by performing all computations on the GPU without transferring data to the CPU):* Graphics hardware has only been used for rendering within the last few decades, however, its extended capabilities in supporting complex operations have also become useful in non-graphics applications. In particular, the advent of a programmable vertex shader and pixel shader enables flexible functions for general computation. Since GPUs are designed for high-performance rendering where repeated operations are common, they are more effective in utilizing parallelism and more pipelined than general purpose CPUs. Therefore, in areas where repeated operations are common, a GPU can produce a better performance than a CPU. The mechanism of general computation using a GPU is as follows. The input is transferred to the GPU as textures or vertex values. The computation is then performed by the vertex shader and pixel shader during a number of rendering passes. The vertex shader performs a routine for every vertex that involves computing its position, color, and texture coordinates, while the pixel shader is performed for every pixel covered by polygons and outputs the color of the pixel. As described above, the inner-product operation for each layer of an NN can be replaced with a matrix multiplication based on accumulating the input vectors and weight vectors. As such, the computation-per-layer can be written as follows:

Asserted Claims	Additional References and Disclosure
	$$W = \begin{bmatrix} w_{11} & w_{12} & w_{13} & \dots & w_{1N} \\ w_{21} & w_{22} & w_{23} & \dots & w_{2N} \\ \dots & \dots & \dots & \dots & \dots \\ w_{M1} & w_{M2} & w_{M3} & \dots & w_{MN} \end{bmatrix} = \begin{bmatrix} W_1 \\ W_2 \\ \dots \\ W_M \end{bmatrix},$$ $$X = \begin{bmatrix} x_{11} & x_{12} & x_{13} & \dots & x_{1L} \\ x_{21} & x_{22} & x_{23} & \dots & x_{2L} \\ \dots & \dots & \dots & \dots & \dots \\ x_{N1} & x_{N2} & x_{N3} & \dots & x_{NL} \end{bmatrix}$$ $$= \begin{bmatrix} X_1 & X_2 & X_3 & \cdots & X_L \end{bmatrix},$$ $$B = \begin{bmatrix} b_1 & b_1 & b_1 & \dots & b_1 \\ b_2 & b_2 & b_2 & \dots & b_2 \\ \dots & \dots & \dots & \dots & \dots \\ b_M & b_M & b_M & \dots & b_M \end{bmatrix},$$

Asserted Claims	Additional References and Disclosure
	$M = W \times X + B$ $= \begin{bmatrix} W_1 \cdot X_1 & W_1 \cdot X_2 & W_1 \cdot X_3 & \dots & W_1 \cdot X_N \\ W_2 \cdot X_1 & W_2 \cdot X_2 & W_2 \cdot X_3 & \dots & W_2 \cdot X_N \\ \dots & \dots & \dots & \dots & \dots \\ W_M \cdot X_1 & W_M \cdot X_2 & W_M \cdot X_3 & \dots & W_M \cdot X_N \end{bmatrix}$ $+ \begin{bmatrix} b_1 & b_1 & b_1 & \dots & b_1 \\ b_2 & b_2 & b_2 & \dots & b_2 \\ \dots & \dots & \dots & \dots & \dots \\ b_M & b_M & b_M & \dots & b_M \end{bmatrix}$ $= \begin{bmatrix} m_{11} & m_{12} & m_{13} & \dots & m_{1L} \\ m_{21} & m_{22} & m_{23} & \dots & m_{2L} \\ m_{31} & m_{32} & m_{33} & \dots & m_{3L} \\ \dots & \dots & \dots & \dots & \dots \\ m_{M1} & m_{M2} & m_{M3} & \dots & m_{ML} \end{bmatrix}, \quad (2)$ $R = \text{sigmoid}(M)$ $= \begin{bmatrix} 1+e^{-m_{11}} & 1+e^{-m_{12}} & 1+e^{-m_{13}} & \dots & 1+e^{-m_{1L}} \\ 1+e^{-m_{21}} & 1+e^{-m_{22}} & 1+e^{-m_{23}} & \dots & 1+e^{-m_{2L}} \\ \dots & \dots & \dots & \dots & \dots \\ 1+e^{-m_{M1}} & 1+e^{-m_{M2}} & 1+e^{-m_{M3}} & \dots & 1+e^{-m_{ML}} \end{bmatrix}, \quad (3)$ where w_{ij} denotes the weight at the connection between the *i*th node of the output layer and the *j*th node of the input layer, M is the number of nodes in the output layer, and N is the number of nodes in the input layer. In addition, x_{ij} is the *i*th feature value of the *j*th input vector and bi is the bias term for the *i*th output node from L input vectors. The final result R_{ij} is the output of the *i*th output node for the *j*th input vector. 1312.

Asserted Claims	Additional References and Disclosure
	The above computation comprises of a matrix multiplication followed by a bias factor addition and sigmoid operation. The matrix multiplication is explained first. The method proposed by Moravanszky [1] is used to implement the matrix multiplication. The two matrices are converted into textures, denoted by texture W and texture X , then the matrix multiplication is performed by rendering. A rectangle is rendered to cover the whole screen. The vertex shader outputs the position and texture coordinates for each vertex of the rectangle, where each vertex has two texture coordinates: one for the row of texture W and the other for the column of texture X . For example, the upper left vertex will have the texture coordinates of the first row of texture W and the first column of texture X , while the upper right vertex will have the texture coordinates of the first row of texture W and the last column of texture X , and so on. As a result of the vertex shader, every pixel (i; j) has texture coordinates corresponding to the *i*th row of W and the *j*th column of X . The pixel shader then performs the inner-product between the row of W and the column of X specified by the texture coordinates. Fig. 1 shows an example of matrix multiplication using a GPU. The number of rendering passes required for matrix multiplication depends on the capability of the GPU, including the number of pixel shader operations and number of texture load operations. 1312–13. Therefore, this paper briefly describes such a text detection method, and readers are referred to the author’s previous publication for more details [3]. In the proposed method, an NN is used to classify the pixels of input images, whereby the feature extraction and pattern recognition stage are integrated in the neural network. The NN then examines local regions looking for text pixels that may be contained in a text region. Therefore, an M × M pixel region in the image is received as the input and a classified image is generated as the output. After the pattern passes the network, the value of the output node is compared with a threshold value and the class of each pixel determined, resulting in a classified image. GPU-based pipelining processing is used to reduce the processing time, and the GPU’s performance is maximized by accumulating a large number of input vectors 1 to create a two-dimensional texture. The input layer then receives the grey values for the pixels at predefined positions inside an M×M window over the input image. Experiments were conducted using an 11 × 11 input window size, with the number of nodes in each hidden layer set at 30. As a result, the processing time for pixel classification was significantly reduced using a GPU. Fig. 2(b) shows the pixel classification result for the left input image, where a black pixel denotes a text pixel. The classification using a GPU produced almost the same result as without a GPU. 1313.

Asserted Claims	Additional References and Disclosure
	As a non-limiting example, to the extent Plaintiff identifies a memory controller in the accused products as an “accelerator controller,” Kirk discloses a memory controller similar to those in the accused products (e.g., Memory Controller 120).  Fig. 12A
8 “The computer system of claim 1, wherein the controller is configured to transfer at least a portion of the output data from the accelerator memory to the main memory after	*See* 1[d][iv], claim 1.

Asserted Claims	Additional References and Disclosure
the at least one graphics processing unit has begun to perform another sequence of computations.”	
9 “The computer system of claim 8, wherein the controller is configured to initiate transfer of the at least a portion of the input data and to transfer the at least a portion of the output data in parallel with performance of at least one computation in the other sequence of computations by the at least one graphics processing unit.”	*See* 1[d][iii], 1[d][iv]; claim 8. *As a non-limiting example, ANN discloses transferring at least a portion of output data in parallel with performance of at least one computation in the other sequence of computations by at least one graphics processing unit (e.g., input and output data are transferred while computations are performed while the neural network is processing real-time data):*

Asserted Claims	Additional References and Disclosure	
	Artificial Neural Network (ANN) is widely used in pattern recognition related area. In some case, the computational load is very heavy, in other case, real time process is required. So there is a need to apply a parallel algorithm on it, and usually the computation for ANN is inherently parallel. In this paper, graphic hardware is used to speed up the computation of ANN. In recent years, graphic processing unit (GPU) grows faster than CPU. Graphic hardware venders provide programmability on GPU. In this paper, application of commodity available GPU for two kinds of ANN models was explored. One is the self-organizing maps (SOM); the other is multi layer perceptron (MLP). The computation result shows that ANN computing on GPU is much faster than on standard CPU when the neural network is large. And some design rules for improve the efficiency on GPU are given. **Keywords: Graphic Process Unit; ANN; SOM; MLP** I. INTRODUCTION Artificial neural network is widely used in classification and pattern recognition. In this paper, two kinds of ANN, self-organizing maps (SOM) [1] and multi layer perceptron (MLP), are implemented on graphic hardware for speed up the computation. MLP is a very simple neural network, the process is linear with one input layer, several hidden layer, and an output layer. It is usually trained by back propagation (BP) algorithm. SOM consists of one layer of n-dimensional units (neurons). It is fully connected with the network input. Additionally, there exist lateral connections through which a topological structure is imposed. For the standard model, the topology is a regular two-dimensional map instantiated by connections between each unit and its direct neighbors. 622.	

Asserted Claims	Additional References and Disclosure
	Three layer MLP neural network is selected to recognize the ball. The input layer consists of seven nodes for seven characters. The hidden layer consists of three nodes, and the output layer consists of just one node. Back propagation method was chosen to train the network. The train set takes as follow. 16 locations are selected from each of 10 robot cars, 4 locations are selected from center of ball, and one location is selected from background field. Totally, 165 locations are selected as the train set. For each selected location, seven characters are calculated and take as an element of the train set. The trained MLP is used to recognize and trace the ball in real time. 624. *B. MLP Computation Time for CPU and GPU* The application of MLP in this paper is to trace the ball in robot soccer in real time. The result is shown in table 1. 625.

Asserted Claims	Additional References and Disclosure
	The result shows that GPU based MLP computation is about 200 times faster than that of CPU. And the result also shows that GPU computation is fast enough for the locating of the ball in real time. TABLE I: MLP COMPUTATION ON CPU AND GPU. CPU /ms: 11328 GPUN /ms: 46 *Id.* The implementation on graphic hardware introduce in this paper has other implicit benefit too. For the SOM computing, a multi-texture or 3-D texture can be used to store the map and make more general SOM computing without the restriction of the vector length of 4. For the MLP used in robot soccer, some graphic hardware have “video in” function, using this kind of graphic hardware; image information can be retrieved directly from camera and store on graphic memory, and don’t need to transfer data between CPU and GPU, which will speed the process. 626. *As a non-limiting example, Oh discloses transferring at least a portion of output data in parallel with performance of at least one computation in the other sequence of computations by at least one graphics processing unit (e.g., input and output data are transferred while the neural network is processing video input data, which may be real-time data):* In the case of using a neural network (NN) for image processing and pattern recognition, the main problem is the computational complexity in the testing stage, which accounts for most of the processing time. Moreover, NN-based image convolution has to exhaustively scan an input image in order to process an

Asserted Claims	Additional References and Disclosure
	entire image [3]. Although an NN can be simulated using software, many potential NN applications require real-time processing, which means fully parallel specially designed hardware implementations, such as an FPGA-based realization of an NN. However, this is somewhat expensive and involves extra design overheads [4]. 1311. Recently, researchers have attempted text-based retrieval of image and video data using several image processing techniques [3]. As such, an automatic text detection algorithm for image data and video documents is important as a preprocessing stage for optical character recognition, and an NN-based text detection method has several advantages over other methods [3]. Therefore, this paper briefly describes such a text detection method, and readers are referred to the author’s previous publication for more details [3]. In the proposed method, an NN is used to classify the pixels of input images, whereby the feature extraction and pattern recognition stage are integrated in the neural network. The NN then examines local regions looking for text pixels that may be contained in a text region. Therefore, an M × M pixel region in the image is received as the input and a classified image is generated as the output. After the pattern passes the network, the value of the output node is compared with a threshold value and the class of each pixel determined, resulting in a classified image. GPU-based pipelining processing is used to reduce the processing time, and the GPU’s performance is maximized by accumulating a large number of input vectors 1 to create a two-dimensional texture. The input layer then receives the grey values for the pixels at predefined positions inside an M×M window over the input image. Experiments were conducted using an 11 × 11 input window size, with the number of nodes in each hidden layer set at 30. As a result, the processing time for pixel classification was significantly reduced using a GPU. Fig. 2(b) shows the pixel classification result for the left input image, where a black pixel denotes a text pixel. The classification using a GPU produced almost the same result as without a GPU. 1313.
10 “The computer system of claim 1, wherein the controller is configured to control execution of the sequence of computations by the at	*See* 1[d][ii], claim 1.

Asserted Claims	Additional References and Disclosure
least one graphics processing unit.”	
12[pre] “A method of performing a sequence of computations representing an artificial neural network on a computer system comprising a central processing unit (CPU), a main memory operably coupled to the central processing unit via a bus, an accelerator operably coupled to the CPU and the main memory via the bus, the accelerator comprising a graphics processing unit (GPU) and an accelerator memory, the method comprising:”	*See* 1[a]–1[c][ii].
12[a] “(A) performing, by the GPU, the sequence of computations on a first portion of input data so as to generate a first portion of output data, the first portion of the output data representing	*See* 1[c][i].

Asserted Claims	Additional References and Disclosure
an output of a neuron in a first layer of the artificial neural network, intermediate computations in the sequence of computations yielding intermediate results, wherein performing the sequence of computations on the first portion of the input data comprises “	
12[a][i] “(i) assigning an output variable to a first texture and a second texture, the output variable being included in a first computational element of a plurality of computational elements, the plurality of computational elements representing the sequence of computations and “	*See* claim 1. *As a non-limiting example, ANN discloses assigning an output variable to a first texture and a second texture (e.g., using Render to Texture functionality), the output variable being included in a first computational element of a plurality of computational elements, plurality of computational elements representing the sequence of computations:* Third, do best to decrease the data exchange between CPU and GPU. Usually OpenGL’s PBuffer are used to save the intermediate result in a texture on GPU and reuse it as an input data. Harris[3] had created a class called “Render to Texture” to easy the use of PBuffer. We had used this class in our program. 625. *As a non-limiting example, Oh discloses assigning an output variable to a first texture and a second texture (e.g., a render target texture), the output variable being included in a first computational element of a plurality of computational elements, plurality of computational elements representing the sequence of computations:*

Asserted Claims	Additional References and Disclosure
	As described above, the inner-product operation for each layer of an NN can be replaced with a matrix multiplication based on accumulating the input vectors and weight vectors. As such, the computation-per-layer can be written as follows: $$W = \begin{bmatrix} w_{11} & w_{12} & w_{13} & \dots & w_{1N} \\ w_{21} & w_{22} & w_{23} & \dots & w_{2N} \\ \dots & \dots & \dots & \dots & \dots \\ w_{M1} & w_{M2} & w_{M3} & \dots & w_{MN} \end{bmatrix} = \begin{bmatrix} W_1 \\ W_2 \\ \dots \\ W_M \end{bmatrix},$$ $$X = \begin{bmatrix} x_{11} & x_{12} & x_{13} & \dots & x_{1L} \\ x_{21} & x_{22} & x_{23} & \dots & x_{2L} \\ \dots & \dots & \dots & \dots & \dots \\ x_{N1} & x_{N2} & x_{N3} & \dots & x_{NL} \end{bmatrix}$$ $$= \begin{bmatrix} X_1 & X_2 & X_3 & \cdots & X_L \end{bmatrix},$$ $$B = \begin{bmatrix} b_1 & b_1 & b_1 & \dots & b_1 \\ b_2 & b_2 & b_2 & \dots & b_2 \\ \dots & \dots & \dots & \dots & \dots \\ b_M & b_M & b_M & \dots & b_M \end{bmatrix},$$

Asserted Claims	Additional References and Disclosure
	$M = W \times X + B$ $= \begin{bmatrix} W_1 \cdot X_1 & W_1 \cdot X_2 & W_1 \cdot X_3 & \dots & W_1 \cdot X_N \\ W_2 \cdot X_1 & W_2 \cdot X_2 & W_2 \cdot X_3 & \dots & W_2 \cdot X_N \\ \dots & \dots & \dots & \dots & \dots \\ W_M \cdot X_1 & W_M \cdot X_2 & W_M \cdot X_3 & \dots & W_M \cdot X_N \end{bmatrix}$ $+ \begin{bmatrix} b_1 & b_1 & b_1 & \dots & b_1 \\ b_2 & b_2 & b_2 & \dots & b_2 \\ \dots & \dots & \dots & \dots & \dots \\ b_M & b_M & b_M & \dots & b_M \end{bmatrix}$ $= \begin{bmatrix} m_{11} & m_{12} & m_{13} & \dots & m_{1L} \\ m_{21} & m_{22} & m_{23} & \dots & m_{2L} \\ m_{31} & m_{32} & m_{33} & \dots & m_{3L} \\ \dots & \dots & \dots & \dots & \dots \\ m_{M1} & m_{M2} & m_{M3} & \dots & m_{ML} \end{bmatrix}, \quad (2)$ $R = \text{sigmoid}(M)$ $= \begin{bmatrix} 1+e^{-m_{11}} & 1+e^{-m_{12}} & 1+e^{-m_{13}} & \dots & 1+e^{-m_{1L}} \\ 1+e^{-m_{21}} & 1+e^{-m_{22}} & 1+e^{-m_{23}} & \dots & 1+e^{-m_{2L}} \\ \dots & \dots & \dots & \dots & \dots \\ 1+e^{-m_{M1}} & 1+e^{-m_{M2}} & 1+e^{-m_{M3}} & \dots & 1+e^{-m_{ML}} \end{bmatrix}, \quad (3)$ where w_{ij} denotes the weight at the connection between the *i*th node of the output layer and the *j*th node of the input layer, M is the number of nodes in the output layer, and N is the number of nodes in the input layer. In addition, x_{ij} is the *i*th feature value of the *j*th input vector and bi is the bias term for the *i*th output node from L input vectors. The final result R_{ij} is the output of the *i*th output node for the *j*th input vector. 1312.

Asserted Claims	Additional References and Disclosure
	The above computation comprises of a matrix multiplication followed by a bias factor addition and sigmoid operation. The matrix multiplication is explained first. The method proposed by Moravanszky [1] is used to implement the matrix multiplication. The two matrices are converted into textures, denoted by texture W and texture X , then the matrix multiplication is performed by rendering. A rectangle is rendered to cover the whole screen. The vertex shader outputs the position and texture coordinates for each vertex of the rectangle, where each vertex has two texture coordinates: one for the row of texture W and the other for the column of texture X . For example, the upper left vertex will have the texture coordinates of the first row of texture W and the first column of texture X , while the upper right vertex will have the texture coordinates of the first row of texture W and the last column of texture X , and so on. As a result of the vertex shader, every pixel (i; j) has texture coordinates corresponding to the *i*th row of W and the *j*th column of X . The pixel shader then performs the inner-product between the row of W and the column of X specified by the texture coordinates. Fig. 1 shows an example of matrix multiplication using a GPU. The number of rendering passes required for matrix multiplication depends on the capability of the GPU, including the number of pixel shader operations and number of texture load operations. 1312–13. The bias term addition and sigmoid operation can be performed in one rendering pass. The bias texture and texture that contains the result of the matrix multiplication, texture W ×X , are set as the active texture. The vertex shader then outputs a full-screen rectangle as before. Each vertex’s texture coordinate for the texture W × X correspond to its position. For example, the upper left vertex has the texture coordinate (0; 0), while the texture coordinate for the upper right vertex is (1; 0). As the bias term is identical for one row, the bias term matrix is one-dimensional and the bias texture coordinates for each vertex correspond to its vertical position. The pixel shader adds two textures and performs a sigmoid operation. If there is more than one layer in an NN, the above procedure is repeated for each layer. The result of the previous layer is saved in the form of a render target texture, which is then used as an input for the next layer. Note that, even though an NN may have multiple layers, the GPU can perform all the operations after texture creation. 1313. *As a non-limiting example, GPU Gems discloses assigning an output variable to a first texture and a second texture (e.g., via render-to-texture), the output variable being included in a first computational*

Asserted Claims	Additional References and Disclosure
	element of a plurality of computational elements, plurality of computational elements representing the sequence of computations: **Render-to-Texture** When an image is generated by the GPU, it can be written to frame-buffer memory that can be displayed, or it can be written to texture memory. This *render-to-texture* functionality is essential for GPGPU, because it is the only current mechanism with which to implement direct feedback of GPU output to input without going back to the host processor. (Indirect feedback is also available via *copy-to-texture*, which requires a copy from one location in the GPU's memory to another.) We can think of render-to-texture as a write-only memory interface. You may be wondering why we don't consider the texture unit and render-to-texture together as a read-write memory interface. The reason is that the fragment processor can read memory as many times as it wants inside a kernel, but it can write data only at the *end* of the kernel program (this is *stream out*). Thus, memory reads and writes are fundamentally separate on GPUs, so it helps to think about them that way. 499.
12[a][ii] “(ii) accumulating a first value for the output variable in the first texture during a first time step;”	*See* 12[a][i]. *As a non-limiting example, ANN discloses accumulating a first value for the output variable in the first texture during a first time step (e.g., using Render to Texture functionality):* Third, do best to decrease the data exchange between CPU and GPU. Usually OpenGL's PBuffer are used to save the intermediate result in a texture on GPU and reuse it as an input data. Harris[3] had created a class called “Render to Texture” to easy the use of PBuffer. We had used this class in our program. 625.

Asserted Claims	Additional References and Disclosure
	As a non-limiting example, Oh discloses accumulating a first value for the output variable in the first texture during a first time step (e.g., saving an output in a texture after a first computation): As described above, the inner-product operation for each layer of an NN can be replaced with a matrix multiplication based on accumulating the input vectors and weight vectors. As such, the computation-per-layer can be written as follows: $$W = \begin{bmatrix} w_{11} & w_{12} & w_{13} & \dots & w_{1N} \\ w_{21} & w_{22} & w_{23} & \dots & w_{2N} \\ \dots & \dots & \dots & \dots & \dots \\ w_{M1} & w_{M2} & w_{M3} & \dots & w_{MN} \end{bmatrix} = \begin{bmatrix} W_1 \\ W_2 \\ \dots \\ W_M \end{bmatrix},$$ $$X = \begin{bmatrix} x_{11} & x_{12} & x_{13} & \dots & x_{1L} \\ x_{21} & x_{22} & x_{23} & \dots & x_{2L} \\ \dots & \dots & \dots & \dots & \dots \\ x_{N1} & x_{N2} & x_{N3} & \dots & x_{NL} \end{bmatrix}$$ $$= \begin{bmatrix} X_1 & X_2 & X_3 & \cdots & X_L \end{bmatrix},$$ $$B = \begin{bmatrix} b_1 & b_1 & b_1 & \dots & b_1 \\ b_2 & b_2 & b_2 & \dots & b_2 \\ \dots & \dots & \dots & \dots & \dots \\ b_M & b_M & b_M & \dots & b_M \end{bmatrix},$$

Asserted Claims	Additional References and Disclosure
	$M = W \times X + B$ $= \begin{bmatrix} W_1 \cdot X_1 & W_1 \cdot X_2 & W_1 \cdot X_3 & \dots & W_1 \cdot X_N \\ W_2 \cdot X_1 & W_2 \cdot X_2 & W_2 \cdot X_3 & \dots & W_2 \cdot X_N \\ \dots & \dots & \dots & \dots & \dots \\ W_M \cdot X_1 & W_M \cdot X_2 & W_M \cdot X_3 & \dots & W_M \cdot X_N \end{bmatrix}$ $+ \begin{bmatrix} b_1 & b_1 & b_1 & \dots & b_1 \\ b_2 & b_2 & b_2 & \dots & b_2 \\ \dots & \dots & \dots & \dots & \dots \\ b_M & b_M & b_M & \dots & b_M \end{bmatrix}$ $= \begin{bmatrix} m_{11} & m_{12} & m_{13} & \dots & m_{1L} \\ m_{21} & m_{22} & m_{23} & \dots & m_{2L} \\ m_{31} & m_{32} & m_{33} & \dots & m_{3L} \\ \dots & \dots & \dots & \dots & \dots \\ m_{M1} & m_{M2} & m_{M3} & \dots & m_{ML} \end{bmatrix}, \quad (2)$ $R = \text{sigmoid}(M)$ $= \begin{bmatrix} 1+e^{-m_{11}} & 1+e^{-m_{12}} & 1+e^{-m_{13}} & \dots & 1+e^{-m_{1L}} \\ 1+e^{-m_{21}} & 1+e^{-m_{22}} & 1+e^{-m_{23}} & \dots & 1+e^{-m_{2L}} \\ \dots & \dots & \dots & \dots & \dots \\ 1+e^{-m_{M1}} & 1+e^{-m_{M2}} & 1+e^{-m_{M3}} & \dots & 1+e^{-m_{ML}} \end{bmatrix}, \quad (3)$ where w_{ij} denotes the weight at the connection between the *i*th node of the output layer and the *j*th node of the input layer, M is the number of nodes in the output layer, and N is the number of nodes in the input layer. In addition, x_{ij} is the *i*th feature value of the *j*th input vector and bi is the bias term for the *i*th output node from L input vectors. The final result R_{ij} is the output of the *i*th output node for the *j*th input vector. 1312.

Asserted Claims	Additional References and Disclosure
	The above computation comprises of a matrix multiplication followed by a bias factor addition and sigmoid operation. The matrix multiplication is explained first. The method proposed by Moravanszky [1] is used to implement the matrix multiplication. The two matrices are converted into textures, denoted by texture W and texture X , then the matrix multiplication is performed by rendering. A rectangle is rendered to cover the whole screen. The vertex shader outputs the position and texture coordinates for each vertex of the rectangle, where each vertex has two texture coordinates: one for the row of texture W and the other for the column of texture X . For example, the upper left vertex will have the texture coordinates of the first row of texture W and the first column of texture X , while the upper right vertex will have the texture coordinates of the first row of texture W and the last column of texture X , and so on. As a result of the vertex shader, every pixel (i; j) has texture coordinates corresponding to the *i*th row of W and the *j*th column of X . The pixel shader then performs the inner-product between the row of W and the column of X specified by the texture coordinates. Fig. 1 shows an example of matrix multiplication using a GPU. The number of rendering passes required for matrix multiplication depends on the capability of the GPU, including the number of pixel shader operations and number of texture load operations. 1312–13. The bias term addition and sigmoid operation can be performed in one rendering pass. The bias texture and texture that contains the result of the matrix multiplication, texture W ×X , are set as the active texture. The vertex shader then outputs a full-screen rectangle as before. Each vertex’s texture coordinate for the texture W × X correspond to its position. For example, the upper left vertex has the texture coordinate (0; 0), while the texture coordinate for the upper right vertex is (1; 0). As the bias term is identical for one row, the bias term matrix is one-dimensional and the bias texture coordinates for each vertex correspond to its vertical position. The pixel shader adds two textures and performs a sigmoid operation. If there is more than one layer in an NN, the above procedure is repeated for each layer. The result of the previous layer is saved in the form of a render target texture, which is then used as an input for the next layer. Note that, even though an NN may have multiple layers, the GPU can perform all the operations after texture creation. 1313. *As a non-limiting example, GPU Gems discloses accumulating a first value for the output variable in the first texture during a first time step (e.g., using render-to-texture to implement direct feedback of GPU output to input):*

Asserted Claims	Additional References and Disclosure
	Render-to-Texture When an image is generated by the GPU, it can be written to frame-buffer memory that can be displayed, or it can be written to texture memory. This *render-to-texture* functionality is essential for GPGPU, because it is the only current mechanism with which to implement direct feedback of GPU output to input without going back to the host processor. (Indirect feedback is also available via *copy-to-texture*, which requires a copy from one location in the GPU's memory to another.) We can think of render-to-texture as a write-only memory interface. You may be wondering why we don't consider the texture unit and render-to-texture together as a read-write memory interface. The reason is that the fragment processor can read memory as many times as it wants inside a kernel, but it can write data only at the *end* of the kernel program (this is *stream out*). Thus, memory reads and writes are fundamentally separate on GPUs, so it helps to think about them that way. 499.
12[b] “(B) in parallel with performing the sequence of computations by the GPU in (A), transferring a second portion of the input data from the main memory to the accelerator via the bus;”	*See* 1[d][iii]; claim 9. *As a non-limiting example, ANN discloses in parallel with performing the sequence of computations by the GPU, transferring a second portion of input data (e.g., real-time video input data) from main memory to the accelerator:*

	Artificial Neural Network (ANN) is widely used in pattern recognition related area. In some case, the computational load is very heavy, in other case, real time process is required. So there is a need to apply a parallel algorithm on it, and usually the computation for ANN is inherently parallel. In this paper, graphic hardware is used to speed up the computation of ANN. In recent years, graphic processing unit (GPU) grows faster than CPU. Graphic hardware venders provide programmability on GPU. In this paper, application of commodity available GPU for two kinds of ANN models was explored. One is the self-organizing maps (SOM); the other is multi layer perceptron (MLP). The computation result shows that ANN computing on GPU is much faster than on standard CPU when the neural network is large. And some design rules for improve the efficiency on GPU are given. **Keywords: Graphic Process Unit; ANN; SOM; MLP** I. INTRODUCTION Artificial neural network is widely used in classification and pattern recognition. In this paper, two kinds of ANN, self-organizing maps (SOM) [1] and multi layer perceptron (MLP), are implemented on graphic hardware for speed up the computation. MLP is a very simple neural network, the process is linear with one input layer, several hidden layer, and an output layer. It is usually trained by back propagation (BP) algorithm. SOM consists of one layer of n-dimensional units (neurons). It is fully connected with the network input. Additionally, there exist lateral connections through which a topological structure is imposed. For the standard model, the topology is a regular two-dimensional map instantiated by connections between each unit and its direct neighbors.	

Asserted Claims	Additional References and Disclosure
	622. Three layer MLP neural network is selected to recognize the ball. The input layer consists of seven nodes for seven characters. The hidden layer consists of three nodes, and the output layer consists of just one node. Back propagation method was chosen to train the network. The train set takes as follow. 16 locations are selected from each of 10 robot cars, 4 locations are selected from center of ball, and one location is selected from background field. Totally, 165 locations are selected as the train set. For each selected location, seven characters are calculated and take as an element of the train set. The trained MLP is used to recognize and trace the ball in real time. 624. *B. MLP Computation Time for CPU and GPU* The application of MLP in this paper is to trace the ball in robot soccer in real time. The result is shown in table 1. 625.

Asserted Claims	Additional References and Disclosure
	The result shows that GPU based MLP computation is about 200 times faster than that of CPU. And the result also shows that GPU computation is fast enough for the locating of the ball in real time. TABLE I: MLP COMPUTATION ON CPU AND GPU. CPU /ms: 11328 GPUN /ms: 46 *Id.* The implementation on graphic hardware introduce in this paper has other implicit benefit too. For the SOM computing, a multi-texture or 3-D texture can be used to store the map and make more general SOM computing without the restriction of the vector length of 4. For the MLP used in robot soccer, some graphic hardware have “video in” function, using this kind of graphic hardware; image information can be retrieved directly from camera and store on graphic memory, and don’t need to transfer data between CPU and GPU, which will speed the process. 626. *As a non-limiting example, Oh discloses in parallel with performing the sequence of computations by the GPU, transferring a second portion of input data (e.g., real-time video input data) from main memory to the accelerator:* In the case of using a neural network (NN) for image processing and pattern recognition, the main problem is the computational complexity in the testing stage, which accounts for most of the processing time. Moreover, NN-based image convolution has to exhaustively scan an input image in order to process an entire image [3]. Although an NN can be simulated using software, many potential NN applications require real-time processing, which means fully parallel specially designed hardware implementations, such as an

Asserted Claims	Additional References and Disclosure
	FPGA-based realization of an NN. However, this is somewhat expensive and involves extra design overheads [4]. 1311. The mechanism of general computation using a GPU is as follows. The input is transferred to the GPU as textures or vertex values. The computation is then performed by the vertex shader and pixel shader during a number of rendering passes. The vertex shader performs a routine for every vertex that involves computing its position, color, and texture coordinates, while the pixel shader is performed for every pixel covered by polygons and outputs the color of the pixel. 1312. Recently, researchers have attempted text-based retrieval of image and video data using several image processing techniques [3]. As such, an automatic text detection algorithm for image data and video documents is important as a preprocessing stage for optical character recognition, and an NN-based text detection method has several advantages over other methods [3]. Therefore, this paper briefly describes such a text detection method, and readers are referred to the author’s previous publication for more details [3]. In the proposed method, an NN is used to classify the pixels of input images, whereby the feature extraction and pattern recognition stage are integrated in the neural network. The NN then examines local regions looking for text pixels that may be contained in a text region. Therefore, an M × M pixel region in the image is received as the input and a classified image is generated as the output. After the pattern passes the network, the value of the output node is compared with a threshold value and the class of each pixel determined, resulting in a classified image. GPU-based pipelining processing is used to reduce the processing time, and the GPU’s performance is maximized by accumulating a large number of input vectors 1 to create a two-dimensional texture. The input layer then receives the grey values for the pixels at predefined positions inside an M×M window over the input image. Experiments were conducted using an 11 × 11 input window size, with the number of nodes in each hidden layer set at 30. As a result, the processing time for pixel classification was significantly reduced using a GPU. Fig. 2(b) shows the pixel classification result for the left input image, where a black pixel denotes a text pixel. The classification using a GPU produced almost the same result as without a GPU. 1313.
12[c] “(C) in parallel with performing the	*See* 1[d][iv]; claim 8.

Asserted Claims	Additional References and Disclosure
sequence of computations by the GPU in (A), transferring a second portion of the output data from the accelerator memory to the main memory via the bus, the second portion of the output data representing an output of a neuron in a second layer in the artificial neural network; and”	*As a non-limiting example, ANN discloses in parallel with performing the sequence of computations by the GPU, transferring a second portion of output data from accelerator memory to main memory. It would have been obvious to transfer output data in accelerator memory from one pass through the artificial neural network to main memory while intermediate computations occur on other input data:*

Asserted Claims	Additional References and Disclosure	
	Artificial Neural Network (ANN) is widely used in pattern recognition related area. In some case, the computational load is very heavy, in other case, real time process is required. So there is a need to apply a parallel algorithm on it, and usually the computation for ANN is inherently parallel. In this paper, graphic hardware is used to speed up the computation of ANN. In recent years, graphic processing unit (GPU) grows faster than CPU. Graphic hardware venders provide programmability on GPU. In this paper, application of commodity available GPU for two kinds of ANN models was explored. One is the self-organizing maps (SOM); the other is multi layer perceptron (MLP). The computation result shows that ANN computing on GPU is much faster than on standard CPU when the neural network is large. And some design rules for improve the efficiency on GPU are given. **Keywords: Graphic Process Unit; ANN; SOM; MLP** I. INTRODUCTION Artificial neural network is widely used in classification and pattern recognition. In this paper, two kinds of ANN, self-organizing maps (SOM) [1] and multi layer perceptron (MLP), are implemented on graphic hardware for speed up the computation. MLP is a very simple neural network, the process is linear with one input layer, several hidden layer, and an output layer. It is usually trained by back propagation (BP) algorithm. SOM consists of one layer of n-dimensional units (neurons). It is fully connected with the network input. Additionally, there exist lateral connections through which a topological structure is imposed. For the standard model, the topology is a regular two-dimensional map instantiated by connections between each unit and its direct neighbors. 622.	

Asserted Claims	Additional References and Disclosure
	Three layer MLP neural network is selected to recognize the ball. The input layer consists of seven nodes for seven characters. The hidden layer consists of three nodes, and the output layer consists of just one node. Back propagation method was chosen to train the network. The train set takes as follow. 16 locations are selected from each of 10 robot cars, 4 locations are selected from center of ball, and one location is selected from background field. Totally, 165 locations are selected as the train set. For each selected location, seven characters are calculated and take as an element of the train set. The trained MLP is used to recognize and trace the ball in real time. 624. *B. MLP Computation Time for CPU and GPU* The application of MLP in this paper is to trace the ball in robot soccer in real time. The result is shown in table 1. 625.

Asserted Claims	Additional References and Disclosure
	The result shows that GPU based MLP computation is about 200 times faster than that of CPU. And the result also shows that GPU computation is fast enough for the locating of the ball in real time. TABLE I: MLP COMPUTATION ON CPU AND GPU. CPU /ms: 11328 GPUN /ms: 46 *Id.* The implementation on graphic hardware introduce in this paper has other implicit benefit too. For the SOM computing, a multi-texture or 3-D texture can be used to store the map and make more general SOM computing without the restriction of the vector length of 4. For the MLP used in robot soccer, some graphic hardware have “video in” function, using this kind of graphic hardware; image information can be retrieved directly from camera and store on graphic memory, and don’t need to transfer data between CPU and GPU, which will speed the process. 626.
12[d] “(D) performing, by the GPU, the sequence of computations on the second portion of the input data, wherein performing the sequence of computations on the	*See* 12[a], 12[b].

Asserted Claims	Additional References and Disclosure
second portion of the input data comprises”	
12[d][i] “(i) accumulating a second value for the output variable in the second texture during a second time step and”	*See* 12[a][i], 12[a][ii]. *As a non-limiting example, ANN discloses accumulating a second value for the output variable in the second texture during a second time step (e.g., using Render to Texture functionality):* Third, do best to decrease the data exchange between CPU and GPU. Usually OpenGL’s PBuffer are used to save the intermediate result in a texture on GPU and reuse it as an input data. Harris[3] had created a class called “Render to Texture” to easy the use of PBuffer. We had used this class in our program. 625. *As a non-limiting example, Oh discloses accumulating a second value for the output variable in the second texture during a second time step (e.g., by saving an output value in a texture):* As described above, the inner-product operation for each layer of an NN can be replaced with a matrix multiplication based on accumulating the input vectors and weight vectors. As such, the computation-per-layer can be written as follows:

Asserted Claims	Additional References and Disclosure
	$$W = \begin{bmatrix} w_{11} & w_{12} & w_{13} & \dots & w_{1N} \\ w_{21} & w_{22} & w_{23} & \dots & w_{2N} \\ \dots & \dots & \dots & \dots & \dots \\ w_{M1} & w_{M2} & w_{M3} & \dots & w_{MN} \end{bmatrix} = \begin{bmatrix} W_1 \\ W_2 \\ \dots \\ W_M \end{bmatrix},$$ $$X = \begin{bmatrix} x_{11} & x_{12} & x_{13} & \dots & x_{1L} \\ x_{21} & x_{22} & x_{23} & \dots & x_{2L} \\ \dots & \dots & \dots & \dots & \dots \\ x_{N1} & x_{N2} & x_{N3} & \dots & x_{NL} \end{bmatrix}$$ $$= \begin{bmatrix} X_1 & X_2 & X_3 & \cdots & X_L \end{bmatrix},$$ $$B = \begin{bmatrix} b_1 & b_1 & b_1 & \dots & b_1 \\ b_2 & b_2 & b_2 & \dots & b_2 \\ \dots & \dots & \dots & \dots & \dots \\ b_M & b_M & b_M & \dots & b_M \end{bmatrix},$$

Asserted Claims	Additional References and Disclosure
	$M = W \times X + B$ $= \begin{bmatrix} W_1 \cdot X_1 & W_1 \cdot X_2 & W_1 \cdot X_3 & \dots & W_1 \cdot X_N \\ W_2 \cdot X_1 & W_2 \cdot X_2 & W_2 \cdot X_3 & \dots & W_2 \cdot X_N \\ \dots & \dots & \dots & \dots & \dots \\ W_M \cdot X_1 & W_M \cdot X_2 & W_M \cdot X_3 & \dots & W_M \cdot X_N \end{bmatrix}$ $+ \begin{bmatrix} b_1 & b_1 & b_1 & \dots & b_1 \\ b_2 & b_2 & b_2 & \dots & b_2 \\ \dots & \dots & \dots & \dots & \dots \\ b_M & b_M & b_M & \dots & b_M \end{bmatrix}$ $= \begin{bmatrix} m_{11} & m_{12} & m_{13} & \dots & m_{1L} \\ m_{21} & m_{22} & m_{23} & \dots & m_{2L} \\ m_{31} & m_{32} & m_{33} & \dots & m_{3L} \\ \dots & \dots & \dots & \dots & \dots \\ m_{M1} & m_{M2} & m_{M3} & \dots & m_{ML} \end{bmatrix}, \quad (2)$ $R = \text{sigmoid}(M)$ $= \begin{bmatrix} 1+e^{-m_{11}} & 1+e^{-m_{12}} & 1+e^{-m_{13}} & \dots & 1+e^{-m_{1L}} \\ 1+e^{-m_{21}} & 1+e^{-m_{22}} & 1+e^{-m_{23}} & \dots & 1+e^{-m_{2L}} \\ \dots & \dots & \dots & \dots & \dots \\ 1+e^{-m_{M1}} & 1+e^{-m_{M2}} & 1+e^{-m_{M3}} & \dots & 1+e^{-m_{ML}} \end{bmatrix}, \quad (3)$ where w_{ij} denotes the weight at the connection between the *i*th node of the output layer and the *j*th node of the input layer, M is the number of nodes in the output layer, and N is the number of nodes in the input layer. In addition, x_{ij} is the *i*th feature value of the *j*th input vector and bi is the bias term for the *i*th output node from L input vectors. The final result R_{ij} is the output of the *i*th output node for the *j*th input vector. 1312.

Asserted Claims	Additional References and Disclosure
	The above computation comprises of a matrix multiplication followed by a bias factor addition and sigmoid operation. The matrix multiplication is explained first. The method proposed by Moravanszky [1] is used to implement the matrix multiplication. The two matrices are converted into textures, denoted by texture W and texture X , then the matrix multiplication is performed by rendering. A rectangle is rendered to cover the whole screen. The vertex shader outputs the position and texture coordinates for each vertex of the rectangle, where each vertex has two texture coordinates: one for the row of texture W and the other for the column of texture X . For example, the upper left vertex will have the texture coordinates of the first row of texture W and the first column of texture X , while the upper right vertex will have the texture coordinates of the first row of texture W and the last column of texture X , and so on. As a result of the vertex shader, every pixel (i; j) has texture coordinates corresponding to the *i*th row of W and the *j*th column of X . The pixel shader then performs the inner-product between the row of W and the column of X specified by the texture coordinates. Fig. 1 shows an example of matrix multiplication using a GPU. The number of rendering passes required for matrix multiplication depends on the capability of the GPU, including the number of pixel shader operations and number of texture load operations. 1312–13. The bias term addition and sigmoid operation can be performed in one rendering pass. The bias texture and texture that contains the result of the matrix multiplication, texture W ×X , are set as the active texture. The vertex shader then outputs a full-screen rectangle as before. Each vertex’s texture coordinate for the texture W × X correspond to its position. For example, the upper left vertex has the texture coordinate (0; 0), while the texture coordinate for the upper right vertex is (1; 0). As the bias term is identical for one row, the bias term matrix is one-dimensional and the bias texture coordinates for each vertex correspond to its vertical position. The pixel shader adds two textures and performs a sigmoid operation. If there is more than one layer in an NN, the above procedure is repeated for each layer. The result of the previous layer is saved in the form of a render target texture, which is then used as an input for the next layer. Note that, even though an NN may have multiple layers, the GPU can perform all the operations after texture creation. 1313.
12[d][ii] “(ii) making the first value of the output variable in the first texture accessible	*See* 12[d][i], 1[c][ii].

Asserted Claims	Additional References and Disclosure
to other computational elements in the plurality of computational elements during the second time step.”	*As a non-limiting example, ANN discloses making the first value of the output variable in the first texture accessible to other computational elements during the second time step (e.g., by saving the intermediate result in a texture on GPU and reusing it as input data):* Third, do best to decrease the data exchange between CPU and GPU. Usually OpenGL’s PBuffer are used to save the intermediate result in a texture on GPU and reuse it as an input data. Harris[3] had created a class called “Render to Texture” to easy the use of PBuffer. We had used this class in our program. 625. *As a non-limiting example, ANN discloses making the first value of the output variable in the first texture accessible to other computational elements during the second time step (e.g., by saving a result in a texture and using it as input for the next layer of the neural network):* As described above, the inner-product operation for each layer of an NN can be replaced with a matrix multiplication based on accumulating the input vectors and weight vectors. As such, the computation-per-layer can be written as follows:

Asserted Claims	Additional References and Disclosure
	$$W = \begin{bmatrix} w_{11} & w_{12} & w_{13} & \dots & w_{1N} \\ w_{21} & w_{22} & w_{23} & \dots & w_{2N} \\ \dots & \dots & \dots & \dots & \dots \\ w_{M1} & w_{M2} & w_{M3} & \dots & w_{MN} \end{bmatrix} = \begin{bmatrix} W_1 \\ W_2 \\ \dots \\ W_M \end{bmatrix},$$ $$X = \begin{bmatrix} x_{11} & x_{12} & x_{13} & \dots & x_{1L} \\ x_{21} & x_{22} & x_{23} & \dots & x_{2L} \\ \dots & \dots & \dots & \dots & \dots \\ x_{N1} & x_{N2} & x_{N3} & \dots & x_{NL} \end{bmatrix}$$ $$= \begin{bmatrix} X_1 & X_2 & X_3 & \cdots & X_L \end{bmatrix},$$ $$B = \begin{bmatrix} b_1 & b_1 & b_1 & \dots & b_1 \\ b_2 & b_2 & b_2 & \dots & b_2 \\ \dots & \dots & \dots & \dots & \dots \\ b_M & b_M & b_M & \dots & b_M \end{bmatrix},$$

Asserted Claims	Additional References and Disclosure
	$M = W \times X + B$ $= \begin{bmatrix} W_1 \cdot X_1 & W_1 \cdot X_2 & W_1 \cdot X_3 & \dots & W_1 \cdot X_N \\ W_2 \cdot X_1 & W_2 \cdot X_2 & W_2 \cdot X_3 & \dots & W_2 \cdot X_N \\ \dots & \dots & \dots & \dots & \dots \\ W_M \cdot X_1 & W_M \cdot X_2 & W_M \cdot X_3 & \dots & W_M \cdot X_N \end{bmatrix}$ $+ \begin{bmatrix} b_1 & b_1 & b_1 & \dots & b_1 \\ b_2 & b_2 & b_2 & \dots & b_2 \\ \dots & \dots & \dots & \dots & \dots \\ b_M & b_M & b_M & \dots & b_M \end{bmatrix}$ $= \begin{bmatrix} m_{11} & m_{12} & m_{13} & \dots & m_{1L} \\ m_{21} & m_{22} & m_{23} & \dots & m_{2L} \\ m_{31} & m_{32} & m_{33} & \dots & m_{3L} \\ \dots & \dots & \dots & \dots & \dots \\ m_{M1} & m_{M2} & m_{M3} & \dots & m_{ML} \end{bmatrix}, \quad (2)$ $R = \text{sigmoid}(M)$ $= \begin{bmatrix} 1+e^{-m_{11}} & 1+e^{-m_{12}} & 1+e^{-m_{13}} & \dots & 1+e^{-m_{1L}} \\ 1+e^{-m_{21}} & 1+e^{-m_{22}} & 1+e^{-m_{23}} & \dots & 1+e^{-m_{2L}} \\ \dots & \dots & \dots & \dots & \dots \\ 1+e^{-m_{M1}} & 1+e^{-m_{M2}} & 1+e^{-m_{M3}} & \dots & 1+e^{-m_{ML}} \end{bmatrix}, \quad (3)$ where w_{ij} denotes the weight at the connection between the *i*th node of the output layer and the *j*th node of the input layer, M is the number of nodes in the output layer, and N is the number of nodes in the input layer. In addition, x_{ij} is the *i*th feature value of the *j*th input vector and bi is the bias term for the *i*th output node from L input vectors. The final result R_{ij} is the output of the *i*th output node for the *j*th input vector. 1312.

Asserted Claims	Additional References and Disclosure
	The above computation comprises of a matrix multiplication followed by a bias factor addition and sigmoid operation. The matrix multiplication is explained first. The method proposed by Moravanszky [1] is used to implement the matrix multiplication. The two matrices are converted into textures, denoted by texture W and texture X , then the matrix multiplication is performed by rendering. A rectangle is rendered to cover the whole screen. The vertex shader outputs the position and texture coordinates for each vertex of the rectangle, where each vertex has two texture coordinates: one for the row of texture W and the other for the column of texture X . For example, the upper left vertex will have the texture coordinates of the first row of texture W and the first column of texture X , while the upper right vertex will have the texture coordinates of the first row of texture W and the last column of texture X , and so on. As a result of the vertex shader, every pixel (i; j) has texture coordinates corresponding to the *i*th row of W and the *j*th column of X . The pixel shader then performs the inner-product between the row of W and the column of X specified by the texture coordinates. Fig. 1 shows an example of matrix multiplication using a GPU. The number of rendering passes required for matrix multiplication depends on the capability of the GPU, including the number of pixel shader operations and number of texture load operations. 1312–13. The bias term addition and sigmoid operation can be performed in one rendering pass. The bias texture and texture that contains the result of the matrix multiplication, texture W ×X , are set as the active texture. The vertex shader then outputs a full-screen rectangle as before. Each vertex’s texture coordinate for the texture W × X correspond to its position. For example, the upper left vertex has the texture coordinate (0; 0), while the texture coordinate for the upper right vertex is (1; 0). As the bias term is identical for one row, the bias term matrix is one-dimensional and the bias texture coordinates for each vertex correspond to its vertical position. The pixel shader adds two textures and performs a sigmoid operation. If there is more than one layer in an NN, the above procedure is repeated for each layer. The result of the previous layer is saved in the form of a render target texture, which is then used as an input for the next layer. Note that, even though an NN may have multiple layers, the GPU can perform all the operations after texture creation. 1313.
13 “The method of claim 12, further comprising: storing the input data in the main	*See* 1[b]; claims 2, 12.

Asserted Claims	Additional References and Disclosure
memory in response to a user interaction.”	
14[a] “The method of claim 12, further comprising: receiving the input data at a first rate; and”	*See* 3[a], claim 12.
14[b] “wherein (A) comprises performing the sequence of computations at a second rate different than the first rate.”	*See* 3[b].
16 “The method of claim 12, wherein (C) comprises: transferring the second portion of the output data from the accelerator memory to the main memory without transferring any of the intermediate results of the plurality of sequential computations from the accelerator memory to the main memory so as to reduce data transfer via the bus.”	*See* claims 7, 12.
17 “The method of claim 12, wherein (C) comprises: transferring	*See* 1[d][iv]; claims 8, 12.

Asserted Claims	Additional References and Disclosure
the second portion of the output data from the accelerator memory to the main memory after the GPU has begun to perform another sequence of computations.”	
18 “The method of claim 17, wherein (C) further comprises: initiating transfer of the second portion of the output data in parallel with performance of at least one computation in the other sequence of computations.”	*See* claims 9, 17.
20[a] “The method of claim 12, further comprising: storing parameters common to all of the computations in the sequence of computations in a first memory bank in the accelerator memory; and”	*See* 6[a], claim 12.
20[b] “storing data specific to at least one computation in the sequence of	*See* 6[b].

Asserted Claims	Additional References and Disclosure
computations in a second memory bank in the accelerator memory.”	
21[pre] “A method of performing a sequence of computations representing an artificial neural network, the method comprising:”	*See* 1[c][i], 12[pre].
21[a] “receiving, at a central processing unit (CPU), first input data acquired from an external system in real time;”	*See* 1[a].
21[b] “initializing, by a controller operably coupled to a graphics processing unit (GPU), textures and shaders in a memory operably coupled to the GPU;”	*See* 1[c][ii], 1[d], 1[d][i].
21[c] “transferring the first input data received by the CPU to the memory operably coupled to the GPU;”	*See* 1[b], 1[c], 1[d][iii].
21[d] “performing, by the graphics processing unit (GPU), a first	*See* 1[c][i], 12[a].

Asserted Claims	Additional References and Disclosure
computation in the sequence of computations on the first input data based on the textures and shaders to generate first output data, computations in the sequence of computations representing respective layers of neurons in the artificial neural network, an output of the first computation in the sequence of computations representing an output of a first neuron in a first layer in the artificial neural network;”	
21[e] “storing, in the memory operably coupled to the GPU, the first input data and the first output data; and”	*See* 1[c][ii], 1[d][iii], 12[a].
21[f] “transferring second input data acquired from the external system in real time into the memory operably coupled to the	*See* 1[d][iii], 1[d][iv], 12[b], 12[c].

Asserted Claims	Additional References and Disclosure
GPU after the GPU starts the first computation and before the GPU starts a second computation of the sequence of computations, an output of the second computation in the sequence of computations representing an output of a second neuron in a second layer in the artificial neural network.”	
22 “The method of claim 21, wherein transferring the second input data comprises transferring the second input data via a bus operably coupled to the CPU.”	*See* 1[c], 12[b], claim 21.
23 “The method of claim 21, further comprising: transferring the first output data from the memory to another memory during the second computation in	*See* 1[d][iv], claim 21.

Asserted Claims	Additional References and Disclosure
the sequence of computations.”	
24[a] “The method of claim 23, further comprising: storing intermediate results of the sequence of computations in the memory, and”	*See* 12[a]; claims 16, 23. *As a non-limiting example, Oh discloses storing intermediate results of the sequence of computations in memory (e.g., by saving results in textures):* Graphics hardware has only been used for rendering within the last few decades, however, its extended capabilities in supporting complex operations have also become useful in non-graphics applications. In particular, the advent of a programmable vertex shader and pixel shader enables flexible functions for general computation. Since GPUs are designed for high-performance rendering where repeated operations are common, they are more effective in utilizing parallelism and more pipelined than general purpose CPUs. Therefore, in areas where repeated operations are common, a GPU can produce a better performance than a CPU. The mechanism of general computation using a GPU is as follows. The input is transferred to the GPU as textures or vertex values. The computation is then performed by the vertex shader and pixel shader during a number of rendering passes. The vertex shader performs a routine for every vertex that involves computing its position, color, and texture coordinates, while the pixel shader is performed for every pixel covered by polygons and outputs the color of the pixel. As described above, the inner-product operation for each layer of an NN can be replaced with a matrix multiplication based on accumulating the input vectors and weight vectors. As such, the computation-per-layer can be written as follows:

Asserted Claims	Additional References and Disclosure
	$$W = \begin{bmatrix} w_{11} & w_{12} & w_{13} & \dots & w_{1N} \\ w_{21} & w_{22} & w_{23} & \dots & w_{2N} \\ \dots & \dots & \dots & \dots & \dots \\ w_{M1} & w_{M2} & w_{M3} & \dots & w_{MN} \end{bmatrix} = \begin{bmatrix} W_1 \\ W_2 \\ \dots \\ W_M \end{bmatrix},$$ $$X = \begin{bmatrix} x_{11} & x_{12} & x_{13} & \dots & x_{1L} \\ x_{21} & x_{22} & x_{23} & \dots & x_{2L} \\ \dots & \dots & \dots & \dots & \dots \\ x_{N1} & x_{N2} & x_{N3} & \dots & x_{NL} \end{bmatrix}$$ $$= \begin{bmatrix} X_1 & X_2 & X_3 & \cdots & X_L \end{bmatrix},$$ $$B = \begin{bmatrix} b_1 & b_1 & b_1 & \dots & b_1 \\ b_2 & b_2 & b_2 & \dots & b_2 \\ \dots & \dots & \dots & \dots & \dots \\ b_M & b_M & b_M & \dots & b_M \end{bmatrix},$$

Asserted Claims	Additional References and Disclosure
	$M = W \times X + B$ $= \begin{bmatrix} W_1 \cdot X_1 & W_1 \cdot X_2 & W_1 \cdot X_3 & \dots & W_1 \cdot X_N \\ W_2 \cdot X_1 & W_2 \cdot X_2 & W_2 \cdot X_3 & \dots & W_2 \cdot X_N \\ \dots & \dots & \dots & \dots & \dots \\ W_M \cdot X_1 & W_M \cdot X_2 & W_M \cdot X_3 & \dots & W_M \cdot X_N \end{bmatrix}$ $+ \begin{bmatrix} b_1 & b_1 & b_1 & \dots & b_1 \\ b_2 & b_2 & b_2 & \dots & b_2 \\ \dots & \dots & \dots & \dots & \dots \\ b_M & b_M & b_M & \dots & b_M \end{bmatrix}$ $= \begin{bmatrix} m_{11} & m_{12} & m_{13} & \dots & m_{1L} \\ m_{21} & m_{22} & m_{23} & \dots & m_{2L} \\ m_{31} & m_{32} & m_{33} & \dots & m_{3L} \\ \dots & \dots & \dots & \dots & \dots \\ m_{M1} & m_{M2} & m_{M3} & \dots & m_{ML} \end{bmatrix}, \quad (2)$ $R = \text{sigmoid}(M)$ $= \begin{bmatrix} 1+e^{-m_{11}} & 1+e^{-m_{12}} & 1+e^{-m_{13}} & \dots & 1+e^{-m_{1L}} \\ 1+e^{-m_{21}} & 1+e^{-m_{22}} & 1+e^{-m_{23}} & \dots & 1+e^{-m_{2L}} \\ \dots & \dots & \dots & \dots & \dots \\ 1+e^{-m_{M1}} & 1+e^{-m_{M2}} & 1+e^{-m_{M3}} & \dots & 1+e^{-m_{ML}} \end{bmatrix}, \quad (3)$ where w_{ij} denotes the weight at the connection between the *i*th node of the output layer and the *j*th node of the input layer, M is the number of nodes in the output layer, and N is the number of nodes in the input layer. In addition, x_{ij} is the *i*th feature value of the *j*th input vector and bi is the bias term for the *i*th output node from L input vectors. The final result R_{ij} is the output of the *i*th output node for the *j*th input vector. 1312.

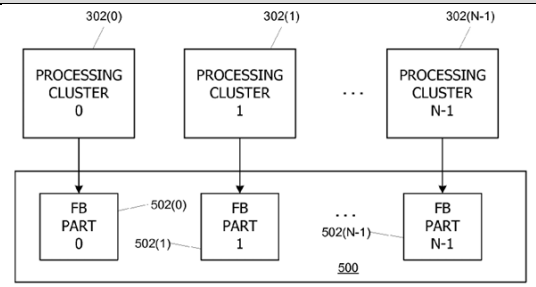
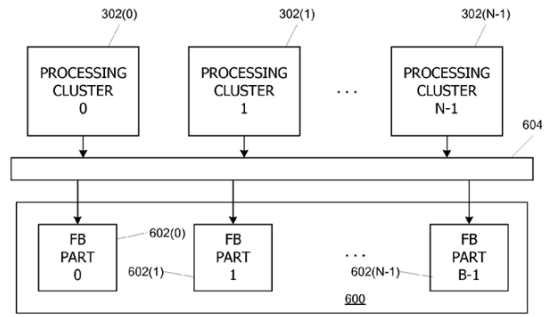
Asserted Claims	Additional References and Disclosure
	The above computation comprises of a matrix multiplication followed by a bias factor addition and sigmoid operation. The matrix multiplication is explained first. The method proposed by Moravanszky [1] is used to implement the matrix multiplication. The two matrices are converted into textures, denoted by texture W and texture X , then the matrix multiplication is performed by rendering. A rectangle is rendered to cover the whole screen. The vertex shader outputs the position and texture coordinates for each vertex of the rectangle, where each vertex has two texture coordinates: one for the row of texture W and the other for the column of texture X . For example, the upper left vertex will have the texture coordinates of the first row of texture W and the first column of texture X , while the upper right vertex will have the texture coordinates of the first row of texture W and the last column of texture X , and so on. As a result of the vertex shader, every pixel (i; j) has texture coordinates corresponding to the *i*th row of W and the *j*th column of X . The pixel shader then performs the inner-product between the row of W and the column of X specified by the texture coordinates. Fig. 1 shows an example of matrix multiplication using a GPU. The number of rendering passes required for matrix multiplication depends on the capability of the GPU, including the number of pixel shader operations and number of texture load operations. 1312–13. Therefore, this paper briefly describes such a text detection method, and readers are referred to the author’s previous publication for more details [3]. In the proposed method, an NN is used to classify the pixels of input images, whereby the feature extraction and pattern recognition stage are integrated in the neural network. The NN then examines local regions looking for text pixels that may be contained in a text region. Therefore, an M × M pixel region in the image is received as the input and a classified image is generated as the output. After the pattern passes the network, the value of the output node is compared with a threshold value and the class of each pixel determined, resulting in a classified image. GPU-based pipelining processing is used to reduce the processing time, and the GPU’s performance is maximized by accumulating a large number of input vectors 1 to create a two-dimensional texture. The input layer then receives the grey values for the pixels at predefined positions inside an M×M window over the input image. Experiments were conducted using an 11 × 11 input window size, with the number of nodes in each hidden layer set at 30. As a result, the processing time for pixel classification was significantly reduced using a GPU. Fig. 2(b) shows the pixel classification result for the left input image, where a black pixel denotes a text pixel. The classification using a GPU produced almost the same result as without a GPU. 1313.

Asserted Claims	Additional References and Disclosure
24[b] “wherein transferring the first output data from the memory to the other memory occurs without transferring the intermediate results of the sequence of computations.”	*See* claims 7, 16. *As a non-limiting example, ANN discloses transferring first output data from memory to another memory without transferring intermediate results (e.g., by saving intermediate results in a texture on GPU):* Third, do best to decrease the data exchange between CPU and GPU. Usually OpenGL’s PBuffer are used to save the intermediate result in a texture on GPU and reuse it as an input data. Harris[3] had created a class called “Render to Texture” to easy the use of PBuffer. We had used this class in our program. 625.
25 “The method of claim 23, wherein transferring the second input data and transferring the first output data occurs in parallel.”	*See* 12[b], 12[c]; claims 9, 23. *As a nonlimiting example, ANN discloses transferring second input data and first output data in parallel (e.g., while processing data in real-time):*

Asserted Claims	Additional References and Disclosure	
	Artificial Neural Network (ANN) is widely used in pattern recognition related area. In some case, the computational load is very heavy, in other case, real time process is required. So there is a need to apply a parallel algorithm on it, and usually the computation for ANN is inherently parallel. In this paper, graphic hardware is used to speed up the computation of ANN. In recent years, graphic processing unit (GPU) grows faster than CPU. Graphic hardware venders provide programmability on GPU. In this paper, application of commodity available GPU for two kinds of ANN models was explored. One is the self-organizing maps (SOM); the other is multi layer perceptron (MLP). The computation result shows that ANN computing on GPU is much faster than on standard CPU when the neural network is large. And some design rules for improve the efficiency on GPU are given. **Keywords: Graphic Process Unit; ANN; SOM; MLP** I. INTRODUCTION Artificial neural network is widely used in classification and pattern recognition. In this paper, two kinds of ANN, self-organizing maps (SOM) [1] and multi layer perceptron (MLP), are implemented on graphic hardware for speed up the computation. MLP is a very simple neural network, the process is linear with one input layer, several hidden layer, and an output layer. It is usually trained by back propagation (BP) algorithm. SOM consists of one layer of n-dimensional units (neurons). It is fully connected with the network input. Additionally, there exist lateral connections through which a topological structure is imposed. For the standard model, the topology is a regular two-dimensional map instantiated by connections between each unit and its direct neighbors. 622.	

Asserted Claims	Additional References and Disclosure
	Three layer MLP neural network is selected to recognize the ball. The input layer consists of seven nodes for seven characters. The hidden layer consists of three nodes, and the output layer consists of just one node. Back propagation method was chosen to train the network. The train set takes as follow. 16 locations are selected from each of 10 robot cars, 4 locations are selected from center of ball, and one location is selected from background field. Totally, 165 locations are selected as the train set. For each selected location, seven characters are calculated and take as an element of the train set. The trained MLP is used to recognize and trace the ball in real time. 624. *B. MLP Computation Time for CPU and GPU* The application of MLP in this paper is to trace the ball in robot soccer in real time. The result is shown in table 1. 625.

Asserted Claims	Additional References and Disclosure
	The result shows that GPU based MLP computation is about 200 times faster than that of CPU. And the result also shows that GPU computation is fast enough for the locating of the ball in real time. TABLE I: MLP COMPUTATION ON CPU AND GPU. CPU /ms: 11328 GPUN /ms: 46 *Id.* The implementation on graphic hardware introduce in this paper has other implicit benefit too. For the SOM computing, a multi-texture or 3-D texture can be used to store the map and make more general SOM computing without the restriction of the vector length of 4. For the MLP used in robot soccer, some graphic hardware have “video in” function, using this kind of graphic hardware; image information can be retrieved directly from camera and store on graphic memory, and don’t need to transfer data between CPU and GPU, which will speed the process. 626.
26 “The method of claim 21, further comprising: storing, in a first memory partition of the memory, parameters common to all of the computations in the sequence of computations.”	*See* 6[a], claim 21. *As a non-limiting example, Danskin discloses storing data in one or more “partitions” (e.g., partitions 602).*

Asserted Claims	Additional References and Disclosure
	 *FIG. 5*  *FIG. 6* Frame buffer 600 is partitioned (physically or logically) into a number B of partitions 602, where B might be equal to or different from the number N of processing clusters 302. Processing clusters 302 are coupled to partitions 602 via a crossbar 604. Each cluster 302 can write pixel data to any one (or more) of the B partitions 602. Danskin at 2:45–56, 11:19–12:37, Figs. 5, 6.
27 “The method of claim 26, further comprising: storing, in a second memory partition of the	*See* claims 6, 26.

Asserted Claims	Additional References and Disclosure
memory, data specific to the first computation in the sequence of computations.”	
28 “The method of claim 27, further comprising: storing, in the second memory partition, external input data patterns, representations of internal variables, an input of the computation in the sequence of computations, and the output of the computation in the sequence of computations.”	*See* 1[c][ii], 1[d][iii], claim 27. *As a non-limiting example, Oh discloses storing external data input patterns (e.g., input from an external camera) and representations of internal variables (e.g., outputs of a hidden layer in a neural network which are used as input for the next layer):* In the case of using a neural network (NN) for image processing and pattern recognition, the main problem is the computational complexity in the testing stage, which accounts for most of the processing time. Moreover, NN-based image convolution has to exhaustively scan an input image in order to process an entire image [3]. Although an NN can be simulated using software, many potential NN applications require real-time processing, which means fully parallel specially designed hardware implementations, such as an FPGA-based realization of an NN. However, this is somewhat expensive and involves extra design overheads [4]. 1311. Recently, researchers have attempted text-based retrieval of image and video data using several image processing techniques [3]. As such, an automatic text detection algorithm for image data and video documents is important as a preprocessing stage for optical character recognition, and an NN-based text detection method has several advantages over other methods [3]. Therefore, this paper briefly describes such a text detection method, and readers are referred to the author’s previous publication for more details [3]. In the proposed method, an NN is used to classify the pixels of input images, whereby the feature extraction and pattern recognition stage are integrated in the neural network. The NN then examines local regions looking for text pixels that may be contained in a text region. Therefore, an M × M pixel region in the image is received as the input and a classified image is generated as the output. After the pattern passes the network, the value of the output node is compared with a threshold value and the class of each pixel determined, resulting in a classified image. GPU-based pipelining processing is used to reduce the processing time, and the GPU’s performance is maximized by accumulating a large number of input vectors 1 to create a two-dimensional texture. The input layer then

Asserted Claims	Additional References and Disclosure
	receives the grey values for the pixels at predefined positions inside an M×M window over the input image. Experiments were conducted using an 11 × 11 input window size, with the number of nodes in each hidden layer set at 30. As a result, the processing time for pixel classification was significantly reduced using a GPU. Fig. 2(b) shows the pixel classification result for the left input image, where a black pixel denotes a text pixel. The classification using a GPU produced almost the same result as without a GPU. 1313. The bias term addition and sigmoid operation can be performed in one rendering pass. The bias texture and texture that contains the result of the matrix multiplication, texture W ×X , are set as the active texture. The vertex shader then outputs a full-screen rectangle as before. Each vertex’s texture coordinate for the texture W × X correspond to its position. For example, the upper left vertex has the texture coordinate (0; 0), while the texture coordinate for the upper right vertex is (1; 0). As the bias term is identical for one row, the bias term matrix is one-dimensional and the bias texture coordinates for each vertex correspond to its vertical position. The pixel shader adds two textures and performs a sigmoid operation. If there is more than one layer in an NN, the above procedure is repeated for each layer. The result of the previous layer is saved in the form of a render target texture, which is then used as an input for the next layer. Note that, even though an NN may have multiple layers, the GPU can perform all the operations after texture creation. 1313.
29 “The method of claim 21, wherein storing the first output data comprises: accumulating, in the memory, outputs of computational elements executed by the GPU in performing the first computation in the sequence of computations.”	*See* 12[a][i], 12[a][ii], claim 21. *As a non-limiting example, Oh discloses accumulating, in the memory, outputs (e.g., results) of computational elements executed by the GPU in performing the first computation in the sequence of computations:* As described above, the inner-product operation for each layer of an NN can be replaced with a matrix multiplication based on accumulating the input vectors and weight vectors. As such, the computation-per-layer can be written as follows:

Asserted Claims	Additional References and Disclosure
	$$W = \begin{bmatrix} w_{11} & w_{12} & w_{13} & \dots & w_{1N} \\ w_{21} & w_{22} & w_{23} & \dots & w_{2N} \\ \dots & \dots & \dots & \dots & \dots \\ w_{M1} & w_{M2} & w_{M3} & \dots & w_{MN} \end{bmatrix} = \begin{bmatrix} W_1 \\ W_2 \\ \dots \\ W_M \end{bmatrix},$$ $$X = \begin{bmatrix} x_{11} & x_{12} & x_{13} & \dots & x_{1L} \\ x_{21} & x_{22} & x_{23} & \dots & x_{2L} \\ \dots & \dots & \dots & \dots & \dots \\ x_{N1} & x_{N2} & x_{N3} & \dots & x_{NL} \end{bmatrix}$$ $$= \begin{bmatrix} X_1 & X_2 & X_3 & \cdots & X_L \end{bmatrix},$$ $$B = \begin{bmatrix} b_1 & b_1 & b_1 & \dots & b_1 \\ b_2 & b_2 & b_2 & \dots & b_2 \\ \dots & \dots & \dots & \dots & \dots \\ b_M & b_M & b_M & \dots & b_M \end{bmatrix},$$

Asserted Claims	Additional References and Disclosure
	$M = W \times X + B$ $= \begin{bmatrix} W_1 \cdot X_1 & W_1 \cdot X_2 & W_1 \cdot X_3 & \dots & W_1 \cdot X_N \\ W_2 \cdot X_1 & W_2 \cdot X_2 & W_2 \cdot X_3 & \dots & W_2 \cdot X_N \\ \dots & \dots & \dots & \dots & \dots \\ W_M \cdot X_1 & W_M \cdot X_2 & W_M \cdot X_3 & \dots & W_M \cdot X_N \end{bmatrix}$ $+ \begin{bmatrix} b_1 & b_1 & b_1 & \dots & b_1 \\ b_2 & b_2 & b_2 & \dots & b_2 \\ \dots & \dots & \dots & \dots & \dots \\ b_M & b_M & b_M & \dots & b_M \end{bmatrix}$ $= \begin{bmatrix} m_{11} & m_{12} & m_{13} & \dots & m_{1L} \\ m_{21} & m_{22} & m_{23} & \dots & m_{2L} \\ m_{31} & m_{32} & m_{33} & \dots & m_{3L} \\ \dots & \dots & \dots & \dots & \dots \\ m_{M1} & m_{M2} & m_{M3} & \dots & m_{ML} \end{bmatrix}, \quad (2)$ $R = \text{sigmoid}(M)$ $= \begin{bmatrix} 1+e^{-m_{11}} & 1+e^{-m_{12}} & 1+e^{-m_{13}} & \dots & 1+e^{-m_{1L}} \\ 1+e^{-m_{21}} & 1+e^{-m_{22}} & 1+e^{-m_{23}} & \dots & 1+e^{-m_{2L}} \\ \dots & \dots & \dots & \dots & \dots \\ 1+e^{-m_{M1}} & 1+e^{-m_{M2}} & 1+e^{-m_{M3}} & \dots & 1+e^{-m_{ML}} \end{bmatrix}, \quad (3)$ where w_{ij} denotes the weight at the connection between the *i*th node of the output layer and the *j*th node of the input layer, M is the number of nodes in the output layer, and N is the number of nodes in the input layer. In addition, x_{ij} is the *i*th feature value of the *j*th input vector and bi is the bias term for the *i*th output node from L input vectors. The final result R_{ij} is the output of the *i*th output node for the *j*th input vector. 1312.

Asserted Claims	Additional References and Disclosure
	The above computation comprises of a matrix multiplication followed by a bias factor addition and sigmoid operation. The matrix multiplication is explained first. The method proposed by Moravanszky [1] is used to implement the matrix multiplication. The two matrices are converted into textures, denoted by texture W and texture X , then the matrix multiplication is performed by rendering. A rectangle is rendered to cover the whole screen. The vertex shader outputs the position and texture coordinates for each vertex of the rectangle, where each vertex has two texture coordinates: one for the row of texture W and the other for the column of texture X . For example, the upper left vertex will have the texture coordinates of the first row of texture W and the first column of texture X , while the upper right vertex will have the texture coordinates of the first row of texture W and the last column of texture X , and so on. As a result of the vertex shader, every pixel (i; j) has texture coordinates corresponding to the *i*th row of W and the *j*th column of X . The pixel shader then performs the inner-product between the row of W and the column of X specified by the texture coordinates. Fig. 1 shows an example of matrix multiplication using a GPU. The number of rendering passes required for matrix multiplication depends on the capability of the GPU, including the number of pixel shader operations and number of texture load operations. 1312–13. The bias term addition and sigmoid operation can be performed in one rendering pass. The bias texture and texture that contains the result of the matrix multiplication, texture W ×X , are set as the active texture. The vertex shader then outputs a full-screen rectangle as before. Each vertex’s texture coordinate for the texture W × X correspond to its position. For example, the upper left vertex has the texture coordinate (0; 0), while the texture coordinate for the upper right vertex is (1; 0). As the bias term is identical for one row, the bias term matrix is one-dimensional and the bias texture coordinates for each vertex correspond to its vertical position. The pixel shader adds two textures and performs a sigmoid operation. If there is more than one layer in an NN, the above procedure is repeated for each layer. The result of the previous layer is saved in the form of a render target texture, which is then used as an input for the next layer. Note that, even though an NN may have multiple layers, the GPU can perform all the operations after texture creation. 1313.
30[a] “The method of claim 21, further comprising: storing, in the memory, an output	*See* 12[a][i], 12[a][ii], claim 21. *As a non-limiting example, Oh discloses storing, in the memory, an output (e.g., result) of a previous computation in the sequence of computations:*

Asserted Claims	Additional References and Disclosure
of a previous computation in the sequence of computations; and”	As described above, the inner-product operation for each layer of an NN can be replaced with a matrix multiplication based on accumulating the input vectors and weight vectors. As such, the computation-per-layer can be written as follows: $$W = \begin{bmatrix} w_{11} & w_{12} & w_{13} & \dots & w_{1N} \\ w_{21} & w_{22} & w_{23} & \dots & w_{2N} \\ \dots & \dots & \dots & \dots & \dots \\ w_{M1} & w_{M2} & w_{M3} & \dots & w_{MN} \end{bmatrix} = \begin{bmatrix} W_1 \\ W_2 \\ \dots \\ W_M \end{bmatrix},$$ $$X = \begin{bmatrix} x_{11} & x_{12} & x_{13} & \dots & x_{1L} \\ x_{21} & x_{22} & x_{23} & \dots & x_{2L} \\ \dots & \dots & \dots & \dots & \dots \\ x_{N1} & x_{N2} & x_{N3} & \dots & x_{NL} \end{bmatrix}$$ $$= \begin{bmatrix} X_1 & X_2 & X_3 & \cdots & X_L \end{bmatrix},$$ $$B = \begin{bmatrix} b_1 & b_1 & b_1 & \dots & b_1 \\ b_2 & b_2 & b_2 & \dots & b_2 \\ \dots & \dots & \dots & \dots & \dots \\ b_M & b_M & b_M & \dots & b_M \end{bmatrix},$$

Asserted Claims	Additional References and Disclosure
	$M = W \times X + B$ $= \begin{bmatrix} W_1 \cdot X_1 & W_1 \cdot X_2 & W_1 \cdot X_3 & \dots & W_1 \cdot X_N \\ W_2 \cdot X_1 & W_2 \cdot X_2 & W_2 \cdot X_3 & \dots & W_2 \cdot X_N \\ \dots & \dots & \dots & \dots & \dots \\ W_M \cdot X_1 & W_M \cdot X_2 & W_M \cdot X_3 & \dots & W_M \cdot X_N \end{bmatrix}$ $+ \begin{bmatrix} b_1 & b_1 & b_1 & \dots & b_1 \\ b_2 & b_2 & b_2 & \dots & b_2 \\ \dots & \dots & \dots & \dots & \dots \\ b_M & b_M & b_M & \dots & b_M \end{bmatrix}$ $= \begin{bmatrix} m_{11} & m_{12} & m_{13} & \dots & m_{1L} \\ m_{21} & m_{22} & m_{23} & \dots & m_{2L} \\ m_{31} & m_{32} & m_{33} & \dots & m_{3L} \\ \dots & \dots & \dots & \dots & \dots \\ m_{M1} & m_{M2} & m_{M3} & \dots & m_{ML} \end{bmatrix}, \quad (2)$ $R = \text{sigmoid}(M)$ $= \begin{bmatrix} 1+e^{-m_{11}} & 1+e^{-m_{12}} & 1+e^{-m_{13}} & \dots & 1+e^{-m_{1L}} \\ 1+e^{-m_{21}} & 1+e^{-m_{22}} & 1+e^{-m_{23}} & \dots & 1+e^{-m_{2L}} \\ \dots & \dots & \dots & \dots & \dots \\ 1+e^{-m_{M1}} & 1+e^{-m_{M2}} & 1+e^{-m_{M3}} & \dots & 1+e^{-m_{ML}} \end{bmatrix}, \quad (3)$ where w_{ij} denotes the weight at the connection between the *i*th node of the output layer and the *j*th node of the input layer, M is the number of nodes in the output layer, and N is the number of nodes in the input layer. In addition, x_{ij} is the *i*th feature value of the *j*th input vector and bi is the bias term for the *i*th output node from L input vectors. The final result R_{ij} is the output of the *i*th output node for the *j*th input vector. 1312.

Asserted Claims	Additional References and Disclosure
	The above computation comprises of a matrix multiplication followed by a bias factor addition and sigmoid operation. The matrix multiplication is explained first. The method proposed by Moravanszky [1] is used to implement the matrix multiplication. The two matrices are converted into textures, denoted by texture W and texture X , then the matrix multiplication is performed by rendering. A rectangle is rendered to cover the whole screen. The vertex shader outputs the position and texture coordinates for each vertex of the rectangle, where each vertex has two texture coordinates: one for the row of texture W and the other for the column of texture X . For example, the upper left vertex will have the texture coordinates of the first row of texture W and the first column of texture X , while the upper right vertex will have the texture coordinates of the first row of texture W and the last column of texture X , and so on. As a result of the vertex shader, every pixel (i; j) has texture coordinates corresponding to the *i*th row of W and the *j*th column of X . The pixel shader then performs the inner-product between the row of W and the column of X specified by the texture coordinates. Fig. 1 shows an example of matrix multiplication using a GPU. The number of rendering passes required for matrix multiplication depends on the capability of the GPU, including the number of pixel shader operations and number of texture load operations. 1312–13. The bias term addition and sigmoid operation can be performed in one rendering pass. The bias texture and texture that contains the result of the matrix multiplication, texture W ×X , are set as the active texture. The vertex shader then outputs a full-screen rectangle as before. Each vertex’s texture coordinate for the texture W × X correspond to its position. For example, the upper left vertex has the texture coordinate (0; 0), while the texture coordinate for the upper right vertex is (1; 0). As the bias term is identical for one row, the bias term matrix is one-dimensional and the bias texture coordinates for each vertex correspond to its vertical position. The pixel shader adds two textures and performs a sigmoid operation. If there is more than one layer in an NN, the above procedure is repeated for each layer. The result of the previous layer is saved in the form of a render target texture, which is then used as an input for the next layer. Note that, even though an NN may have multiple layers, the GPU can perform all the operations after texture creation. 1313.
30[b] “accessing, by the GPU, the output of the previous computation during performance of	*See* 12[d][i], 12[d][ii].

Asserted Claims	Additional References and Disclosure
the computation in the sequence of computations.”	*As a non-limiting example, ANN discloses accessing the output of the previous computation during performance of the computation in the sequence of computations (e.g., by using the output of the previous computation as input for the next computation):*

Asserted Claims	Additional References and Disclosure	
	Artificial Neural Network (ANN) is widely used in pattern recognition related area. In some case, the computational load is very heavy, in other case, real time process is required. So there is a need to apply a parallel algorithm on it, and usually the computation for ANN is inherently parallel. In this paper, graphic hardware is used to speed up the computation of ANN. In recent years, graphic processing unit (GPU) grows faster than CPU. Graphic hardware venders provide programmability on GPU. In this paper, application of commodity available GPU for two kinds of ANN models was explored. One is the self-organizing maps (SOM); the other is multi layer perceptron (MLP). The computation result shows that ANN computing on GPU is much faster than on standard CPU when the neural network is large. And some design rules for improve the efficiency on GPU are given. **Keywords: Graphic Process Unit; ANN; SOM; MLP** I. INTRODUCTION Artificial neural network is widely used in classification and pattern recognition. In this paper, two kinds of ANN, self-organizing maps (SOM) [1] and multi layer perceptron (MLP), are implemented on graphic hardware for speed up the computation. MLP is a very simple neural network, the process is linear with one input layer, several hidden layer, and an output layer. It is usually trained by back propagation (BP) algorithm. SOM consists of one layer of n-dimensional units (neurons). It is fully connected with the network input. Additionally, there exist lateral connections through which a topological structure is imposed. For the standard model, the topology is a regular two-dimensional map instantiated by connections between each unit and its direct neighbors. 622.	

Asserted Claims	Additional References and Disclosure
	Third, do best to decrease the data exchange between CPU and GPU. Usually OpenGL’s PBuffer are used to save the intermediate result in a texture on GPU and reuse it as an input data. Harris[3] had created a class called “Render to Texture” to easy the use of PBuffer. We had used this class in our program. 625. *As a non-limiting example, Oh discloses accessing the output of the previous computation during performance of the computation in the sequence of computations (e.g., by using the output of the previous computation as input for the next computation):* As described above, the inner-product operation for each layer of an NN can be replaced with a matrix multiplication based on accumulating the input vectors and weight vectors. As such, the computation-per-layer can be written as follows:

Asserted Claims	Additional References and Disclosure
	$$W = \begin{bmatrix} w_{11} & w_{12} & w_{13} & \dots & w_{1N} \\ w_{21} & w_{22} & w_{23} & \dots & w_{2N} \\ \dots & \dots & \dots & \dots & \dots \\ w_{M1} & w_{M2} & w_{M3} & \dots & w_{MN} \end{bmatrix} = \begin{bmatrix} W_1 \\ W_2 \\ \dots \\ W_M \end{bmatrix},$$ $$X = \begin{bmatrix} x_{11} & x_{12} & x_{13} & \dots & x_{1L} \\ x_{21} & x_{22} & x_{23} & \dots & x_{2L} \\ \dots & \dots & \dots & \dots & \dots \\ x_{N1} & x_{N2} & x_{N3} & \dots & x_{NL} \end{bmatrix}$$ $$= \begin{bmatrix} X_1 & X_2 & X_3 & \cdots & X_L \end{bmatrix},$$ $$B = \begin{bmatrix} b_1 & b_1 & b_1 & \dots & b_1 \\ b_2 & b_2 & b_2 & \dots & b_2 \\ \dots & \dots & \dots & \dots & \dots \\ b_M & b_M & b_M & \dots & b_M \end{bmatrix},$$

Asserted Claims	Additional References and Disclosure
	$M = W \times X + B$ $= \begin{bmatrix} W_1 \cdot X_1 & W_1 \cdot X_2 & W_1 \cdot X_3 & \dots & W_1 \cdot X_N \\ W_2 \cdot X_1 & W_2 \cdot X_2 & W_2 \cdot X_3 & \dots & W_2 \cdot X_N \\ \dots & \dots & \dots & \dots & \dots \\ W_M \cdot X_1 & W_M \cdot X_2 & W_M \cdot X_3 & \dots & W_M \cdot X_N \end{bmatrix}$ $+ \begin{bmatrix} b_1 & b_1 & b_1 & \dots & b_1 \\ b_2 & b_2 & b_2 & \dots & b_2 \\ \dots & \dots & \dots & \dots & \dots \\ b_M & b_M & b_M & \dots & b_M \end{bmatrix}$ $= \begin{bmatrix} m_{11} & m_{12} & m_{13} & \dots & m_{1L} \\ m_{21} & m_{22} & m_{23} & \dots & m_{2L} \\ m_{31} & m_{32} & m_{33} & \dots & m_{3L} \\ \dots & \dots & \dots & \dots & \dots \\ m_{M1} & m_{M2} & m_{M3} & \dots & m_{ML} \end{bmatrix}, \quad (2)$ $R = \text{sigmoid}(M)$ $= \begin{bmatrix} 1+e^{-m_{11}} & 1+e^{-m_{12}} & 1+e^{-m_{13}} & \dots & 1+e^{-m_{1L}} \\ 1+e^{-m_{21}} & 1+e^{-m_{22}} & 1+e^{-m_{23}} & \dots & 1+e^{-m_{2L}} \\ \dots & \dots & \dots & \dots & \dots \\ 1+e^{-m_{M1}} & 1+e^{-m_{M2}} & 1+e^{-m_{M3}} & \dots & 1+e^{-m_{ML}} \end{bmatrix}, \quad (3)$ where w_{ij} denotes the weight at the connection between the *i*th node of the output layer and the *j*th node of the input layer, M is the number of nodes in the output layer, and N is the number of nodes in the input layer. In addition, x_{ij} is the *i*th feature value of the *j*th input vector and bi is the bias term for the *i*th output node from L input vectors. The final result R_{ij} is the output of the *i*th output node for the *j*th input vector. 1312.

Asserted Claims	Additional References and Disclosure
	The above computation comprises of a matrix multiplication followed by a bias factor addition and sigmoid operation. The matrix multiplication is explained first. The method proposed by Moravanszky [1] is used to implement the matrix multiplication. The two matrices are converted into textures, denoted by texture W and texture X , then the matrix multiplication is performed by rendering. A rectangle is rendered to cover the whole screen. The vertex shader outputs the position and texture coordinates for each vertex of the rectangle, where each vertex has two texture coordinates: one for the row of texture W and the other for the column of texture X . For example, the upper left vertex will have the texture coordinates of the first row of texture W and the first column of texture X , while the upper right vertex will have the texture coordinates of the first row of texture W and the last column of texture X , and so on. As a result of the vertex shader, every pixel (i; j) has texture coordinates corresponding to the *i*th row of W and the *j*th column of X . The pixel shader then performs the inner-product between the row of W and the column of X specified by the texture coordinates. Fig. 1 shows an example of matrix multiplication using a GPU. The number of rendering passes required for matrix multiplication depends on the capability of the GPU, including the number of pixel shader operations and number of texture load operations. 1312–13. The bias term addition and sigmoid operation can be performed in one rendering pass. The bias texture and texture that contains the result of the matrix multiplication, texture W ×X , are set as the active texture. The vertex shader then outputs a full-screen rectangle as before. Each vertex’s texture coordinate for the texture W × X correspond to its position. For example, the upper left vertex has the texture coordinate (0; 0), while the texture coordinate for the upper right vertex is (1; 0). As the bias term is identical for one row, the bias term matrix is one-dimensional and the bias texture coordinates for each vertex correspond to its vertical position. The pixel shader adds two textures and performs a sigmoid operation. If there is more than one layer in an NN, the above procedure is repeated for each layer. The result of the previous layer is saved in the form of a render target texture, which is then used as an input for the next layer. Note that, even though an NN may have multiple layers, the GPU can perform all the operations after texture creation. 1313.
31 “The method of claim 21, wherein performing the first computation comprises	*See* 1[c][i], 12[a][i], claim 21.

Asserted Claims	Additional References and Disclosure
executing a plurality of computational elements representing a layer of neurons in an artificial neural network.”	
32 “The method of claim 31, wherein all neurons in the layer of neurons are described by the same equation.”	*See* claim 31. *As a non-limiting example, ANN discloses all neurons (e.g., nodes) in the layer of neurons being described by the same equation (e.g., matrix multiplication and sigmoid functions):*

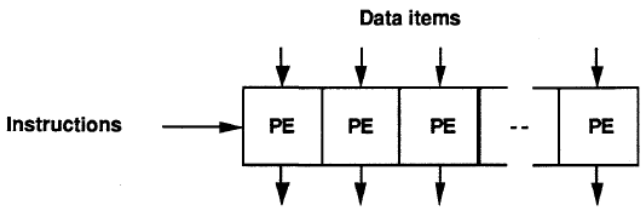
Asserted Claims	Additional References and Disclosure
	B.Computational Model of MLP Three layer MLP neural network is selected to recognize the ball. The input layer consists of seven nodes for seven characters. The hidden layer consists of three nodes, and the output layer consists of just one node. Back propagation method was chosen to train the network. The train set takes as follow. 16 locations are selected from each of 10 robot cars, 4 locations are selected from center of ball, and one location is selected from background field. Totally, 165 locations are selected as the train set. For each selected location, seven characters are calculated and take as an element of the train set. The trained MLP is used to recognize and trace the ball in real time. There are two main computation steps for MLP used as the classification machine. The first one is matrix multiplication: $net = w \bullet x + b$ [3] The second one is sigmoid function calculation: $\sigma(net) = \frac{1}{1+e^{-net}}$ [4] As for the determination of ball on robot soccer, the MLP calculation was applied on each point in the play ground. And MLP was used to distinguish between ball and non-ball position. 624.

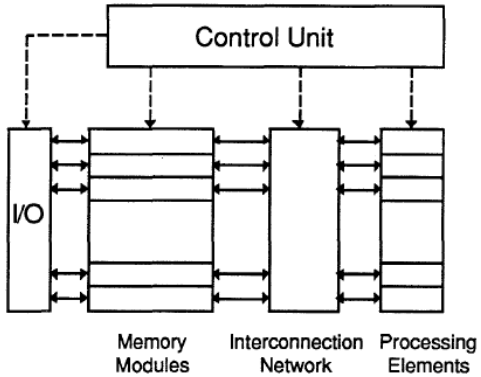
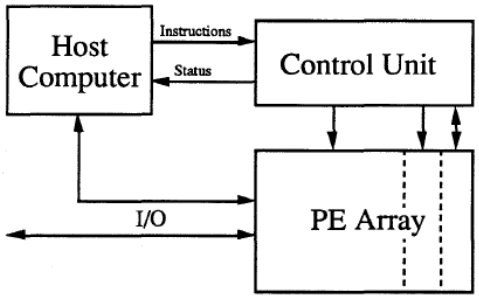
Asserted Claims	Additional References and Disclosure
	As a non-limiting example, ANN discloses all neurons (e.g., nodes) in the layer of neurons being described by the same equation (e.g., matrix multiplication and sigmoid functions): An artificial neural network, usually referred to as ‘neural network’, is based on the concept of the workings of the human brain. There are many different types of NN, with the more popular being a multilayer perceptron, learning vector quantization, radial basis function, Hop/eld, and Kohonen. The current study focuses on using a GPU to implement a multilayer perceptron, which is usually fully connected between adjacent layers. The input layer receives the input features of a given application. Although the network structure can vary as regards the number of layers, number of nodes in each layer, and input mask size, each layer performs the same inner-product operation between the given input vectors and the weight vectors, followed by a non-linear function. Moreover, many inner-product operations can be replaced with a matrix multiplication, which is more appropriate for GPU implementation. 1311–12. If there is more than one layer in an NN, the above procedure is repeated for each layer. The result of the previous layer is saved in the form of a render target texture, which is then used as an input for the next layer. Note that, even though an NN may have multiple layers, the GPU can perform all the operations after texture creation. 1312. As described above, the inner-product operation for each layer of an NN can be replaced with a matrix multiplication based on accumulating the input vectors and weight vectors. As such, the computation-per-layer can be written as follows:

Asserted Claims	Additional References and Disclosure
	$$W = \begin{bmatrix} w_{11} & w_{12} & w_{13} & \dots & w_{1N} \\ w_{21} & w_{22} & w_{23} & \dots & w_{2N} \\ \dots & \dots & \dots & \dots & \dots \\ w_{M1} & w_{M2} & w_{M3} & \dots & w_{MN} \end{bmatrix} = \begin{bmatrix} W_1 \\ W_2 \\ \dots \\ W_M \end{bmatrix},$$ $$X = \begin{bmatrix} x_{11} & x_{12} & x_{13} & \dots & x_{1L} \\ x_{21} & x_{22} & x_{23} & \dots & x_{2L} \\ \dots & \dots & \dots & \dots & \dots \\ x_{N1} & x_{N2} & x_{N3} & \dots & x_{NL} \end{bmatrix}$$ $$= \begin{bmatrix} X_1 & X_2 & X_3 & \cdots & X_L \end{bmatrix},$$ $$B = \begin{bmatrix} b_1 & b_1 & b_1 & \dots & b_1 \\ b_2 & b_2 & b_2 & \dots & b_2 \\ \dots & \dots & \dots & \dots & \dots \\ b_M & b_M & b_M & \dots & b_M \end{bmatrix},$$

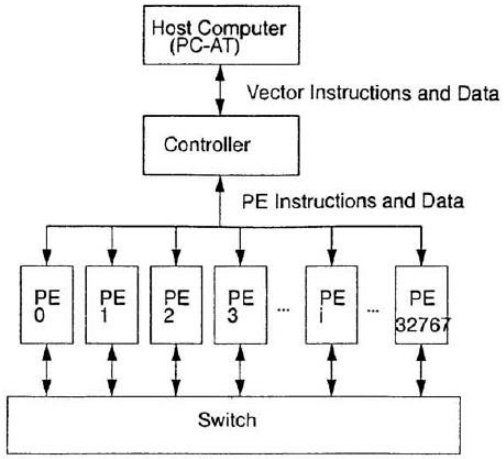
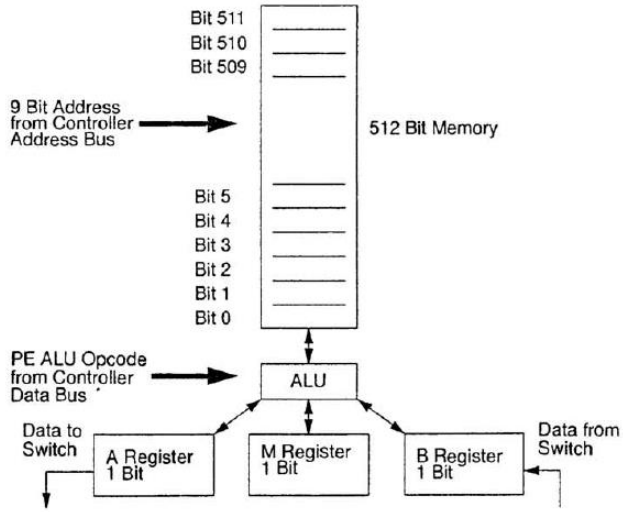
Asserted Claims	Additional References and Disclosure
	$M = W \times X + B$ $= \begin{bmatrix} W_1 \cdot X_1 & W_1 \cdot X_2 & W_1 \cdot X_3 & \dots & W_1 \cdot X_N \\ W_2 \cdot X_1 & W_2 \cdot X_2 & W_2 \cdot X_3 & \dots & W_2 \cdot X_N \\ \dots & \dots & \dots & \dots & \dots \\ W_M \cdot X_1 & W_M \cdot X_2 & W_M \cdot X_3 & \dots & W_M \cdot X_N \end{bmatrix}$ $+ \begin{bmatrix} b_1 & b_1 & b_1 & \dots & b_1 \\ b_2 & b_2 & b_2 & \dots & b_2 \\ \dots & \dots & \dots & \dots & \dots \\ b_M & b_M & b_M & \dots & b_M \end{bmatrix}$ $= \begin{bmatrix} m_{11} & m_{12} & m_{13} & \dots & m_{1L} \\ m_{21} & m_{22} & m_{23} & \dots & m_{2L} \\ m_{31} & m_{32} & m_{33} & \dots & m_{3L} \\ \dots & \dots & \dots & \dots & \dots \\ m_{M1} & m_{M2} & m_{M3} & \dots & m_{ML} \end{bmatrix}, \quad (2)$ $R = \text{sigmoid}(M)$ $= \begin{bmatrix} 1+e^{-m_{11}} & 1+e^{-m_{12}} & 1+e^{-m_{13}} & \dots & 1+e^{-m_{1L}} \\ 1+e^{-m_{21}} & 1+e^{-m_{22}} & 1+e^{-m_{23}} & \dots & 1+e^{-m_{2L}} \\ \dots & \dots & \dots & \dots & \dots \\ 1+e^{-m_{M1}} & 1+e^{-m_{M2}} & 1+e^{-m_{M3}} & \dots & 1+e^{-m_{ML}} \end{bmatrix}, \quad (3)$ where w_{ij} denotes the weight at the connection between the *i*th node of the output layer and the *j*th node of the input layer, M is the number of nodes in the output layer, and N is the number of nodes in the input layer. In addition, x_{ij} is the *i*th feature value of the *j*th input vector and bi is the bias term for the *i*th output node from L input vectors. The final result R_{ij} is the output of the *i*th output node for the *j*th input vector. 1312.

Asserted Claims	Additional References and Disclosure
	The above computation comprises of a matrix multiplication followed by a bias factor addition and sigmoid operation. The matrix multiplication is explained first. The method proposed by Moravanszky [1] is used to implement the matrix multiplication. The two matrices are converted into textures, denoted by texture W and texture X , then the matrix multiplication is performed by rendering. A rectangle is rendered to cover the whole screen. The vertex shader outputs the position and texture coordinates for each vertex of the rectangle, where each vertex has two texture coordinates: one for the row of texture W and the other for the column of texture X . For example, the upper left vertex will have the texture coordinates of the first row of texture W and the first column of texture X , while the upper right vertex will have the texture coordinates of the first row of texture W and the last column of texture X , and so on. As a result of the vertex shader, every pixel (i; j) has texture coordinates corresponding to the *i*th row of W and the *j*th column of X . The pixel shader then performs the inner-product between the row of W and the column of X specified by the texture coordinates. Fig. 1 shows an example of matrix multiplication using a GPU. The number of rendering passes required for matrix multiplication depends on the capability of the GPU, including the number of pixel shader operations and number of texture load operations. 1312–13. Therefore, this paper briefly describes such a text detection method, and readers are referred to the author’s previous publication for more details [3]. In the proposed method, an NN is used to classify the pixels of input images, whereby the feature extraction and pattern recognition stage are integrated in the neural network. The NN then examines local regions looking for text pixels that may be contained in a text region. Therefore, an M × M pixel region in the image is received as the input and a classified image is generated as the output. After the pattern passes the network, the value of the output node is compared with a threshold value and the class of each pixel determined, resulting in a classified image. GPU-based pipelining processing is used to reduce the processing time, and the GPU’s performance is maximized by accumulating a large number of input vectors 1 to create a two-dimensional texture. The input layer then receives the grey values for the pixels at predefined positions inside an M×M window over the input image. Experiments were conducted using an 11 × 11 input window size, with the number of nodes in each hidden layer set at 30. As a result, the processing time for pixel classification was significantly reduced using a GPU. Fig. 2(b) shows the pixel classification result for the left input image, where a black pixel denotes a text pixel. The classification using a GPU produced almost the same result as without a GPU. 1313.

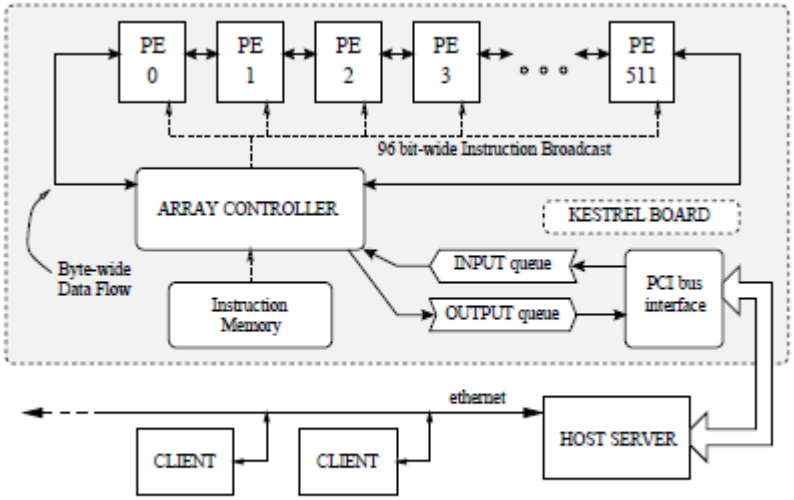
Asserted Claims	Additional References and Disclosure
	As a non-limiting example, Svensson discloses all neurons (e.g., processing elements) in the layer of neurons (e.g., a SIMD processor array) being described by the same equation (e.g., a single instruction working on several data items simultaneously by using several processing elements): ***1. Principles of SIMD*** In Single Instruction stream, Multiple Data stream (SIMD) processors one instruction works on several data items simultaneously by using several Processing Elements (PEs), all of which carry out the same operation as illustrated in Figure SIMD-1.  Principle of SIMD processor. Figure SIMD-1. A SIMD Processor has a single Control Unit reading instructions pointed to by a single Program Counter, decoding them and sending control signals to the PEs. Data are to be supplied to, and derived from, the PEs by a memory with as many data paths as there are PEs. Figure SIMD-2a shows the resulting processing structure which is also known as a Processor Array. The Interconnection Network provides flexibility in choosing source and destination for data to and from the PEs, necessary in many algorithms. The I/O system plays the role of converting – typically at very high rates – input/output data between the format of the outside world and the internal format of the array. The design of the I/O system is highly application dependent. In order to provide overall control of the Processor Array, as well as to execute sequential operations, it is common to use a conventional Host Computer which then views the Processor Array as an attached processor as illustrated in figure SIMD-2b.

Asserted Claims	Additional References and Disclosure
	1.  Processor Array Figure SIMD-2a  Host Computer - Processor Array Relationship Figure SIMD-2b 2.

Asserted Claims	Additional References and Disclosure
	As a non-limiting example, Glover discloses all neurons (e.g., nodes) in the layer of neurons being described by the same equation (e.g., a single instruction working on several data items simultaneously by using several processing elements):

Asserted Claims	Additional References and Disclosure
	 Figure 1: A block diagram of the MM32k.  Figure 2: A block diagram of a single processing element (PE). 845.

Asserted Claims	Additional References and Disclosure
	4 NEURAL NETWORK EXAMPLES A common operation found in neural network classifiers (Kohonen, ART, etc.) is the multi-dimensional nearest-neighbor match. If the network has a large number of nodes, this operation is particularly inefficient on single processor systems, which must compute the distance metric for each node sequentially. Using the MM32k, the distance metrics for all nodes (up to 32768 nodes) can be computed simultaneously, and the identification of the minimum distance can be made using an efficient tree compare included in the system microcode. 846. *As a non-limiting example, Mesa-Martinez discloses all neurons (e.g., processing elements(PEs)) in the layer of neurons (e.g., a SIMD processor array) being described by the same equation (e.g., a single instruction being broadcast to each PE in the array):* The first generation UCSC Kestrel processor is implemented as a systolic array [8] of 512 processing elements (PEs). The system’s configuration as a linear array is ideal for such target applications as sequence analysis, at the same time that it provides a simpler and more flexible programming model than more complex multiple-instruction, multiple data stream (MIMD) machines would when applied to the same domain of problems [7]. For the applications that we were envisioning, programmability was more important than large local memories or reconfigurable interconnection fabrics [4]. We believe our SIMD approach to be a more cost effective solution than systems using fully reconfigurable general purpose multiprocessors [5], [6]. Each processing element in the array is 8-bits wide and contains its own local memory and datapath. The Kestrel board also contains several SDRAM banks that implement the instruction memory (IM) as well as the input/output queues (IQm, OQm) [3]. Instruction broadcasting and sequencing is performed by an on board controller, that is also in charge of memory management (Figure 3). The whole system is self contained and implemented as a single full length PCI card (Figure 2). Like most SIMD processors Kestrel broadcasts each instruction to each PE in the array. Due to the relatively large number of PEs that Kestrel has, and the fact that these PEs are distributed among several chips, instruction broadcasting can easily become more taxing than the actual time to execute the

Asserted Claims	Additional References and Disclosure
	instruction. In order to solve this bottleneck Kestrel implements a board-level instruction issue and broadcast [4]. In order to handle inter-PE communication each PE shares a file of registers with its left and right neighbors. These register banks are known as Systolic Shared Registers (SSRs) [9]. Although this topology might seem limiting at first, it allows for computation and communication to occur concurrently. When a result is stored in the SSR after the instruction has been completed it is immediately visible to the next PE, thus communication takes place automatically (Figure 4). All addresses to the SSRs are issued globally in order to prevent adjacent PEs from writing to the same bank at the same time. This communication method is both elegant and efficient since it greatly simplifies the design of the PE and allowing most of its area to be dedicated for processing and memory. A-9.  Fig. 3. Kestrel board architecture. FIG. 3. The controller is implemented using an FPGA. This device serves as the instruction sequencer for the system. The controller also handles the board’s I/O by interacting with the onboard PCI interface chip. The

Asserted Claims	Additional References and Disclosure
	program instructions received from the host computer are stored in a separate instruction memory (IM). The Kestrel controller is programmed by using 54 bits of the 96-bit instruction word. The remaining bits are used as the instruction to be broadcast to the array. Each cycle, the controller must decide whether or not data is read from the input queue, whether or not data is written to the output queue, and whether or not the immediate field in the instruction should be replaced with data from the controller. A small amount of local memory and logic is used by the controller in order to enable loop counting and the recirculation of data among the PEs. When a critical condition is reached by the system the controller will generate the corresponding interrupt to inform the host about the board’s condition. A-10.
40[pre] “A system for executing an artificial neural network, the system comprising:”	*See* 21[pre].
40[a] “a central processing unit (CPU) to provide first input data;”	*See* 1[a].
40[b] “a memory, operably coupled to the CPU, to store the first input data in a first partition, referenced by a first pointer, before computing a first layer of neurons of the artificial neural network;”	*See* 1[c][ii]. *As a non-limiting example, GPU Gems discloses storing first input data in a first partition (e.g., a buffer), referenced by a first pointer, before computations begin (e.g., using the function pgInitBuffer()):*

Asserted Claims	Additional References and Disclosure
	Stream Management Arrays of data in the framework are called *buffers*. The `pugAllocateBuffer()` function creates a new buffer. Buffers can be read-only, write-only, or read-write, and this can be specified using the `mode` parameter of this function. This function returns a pointer to a `PUGBuffer` structure, which can be passed to a number of functions in the framework, including the following two. To load initial data into a buffer, pass the data to the `pugInitBuffer()` function. To bind an input stream for a kernel to a buffer, call `pugBindStream()`, which takes as arguments a pointer to the `PUGProgram` to bind to, the `PUGBuffer` pointer, and a string containing the name of the `Stream` parameter in the Cg kernel program to which this `PUGBuffer` should be bound. 504.

Asserted Claims	Additional References and Disclosure

	Listing 31-2. C++ Code to Set Up and Run a Reaction-Diffusion Simulation on the GPU
	See the source code on the accompanying CD for more details.
	`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);`
	
	Listing 31-2 *(continued)*. C++ Code to Set Up and Run a Reaction-Diffusion Simulation on the GPU
	`  // Initialize the state of the simulation with values in array`
	`  pugInitBuffer(rdBuffer, array);`
	`}`
	
	`void update_rd(){ // Call this every iteration`
	`  pugBindStream(rdProgram, "concentration", rdBuffer, currentSource);`
	`  PUGRect range(0, width, 0, height);`
	`  pugRunProgram(rdProgram, rdBuffer, range, currentTarget);`
	
	`  std::swap(currentSource, currentTarget);`
	`}`
	
	507–8.

Asserted Claims	Additional References and Disclosure
40[c] “a processing unit, operably coupled to the memory, to perform, during computation of the first layer of neurons, at least one calculation on the first input data so as to generate first output data, the first output data representing an output of at least one neuron in the first layer of neurons; and”	*See* 1[c][i], 21[d].
40[d] “a controller, operably coupled to the processing unit and the memory, to:”	*See* 1[d].
40[d][i] “store the first output data in a second partition of the memory, the second partition referenced by a second pointer, and to swap the first pointer with the second pointer at the end of the computation of the first layer of neurons, such that the first output data becomes an input for a second layer of neurons	*See* 12[a][i], 21[d], 40[b]. *As a non-limiting example, ANN discloses the output data becoming input for a second layer of the artificial neural network:* Third, do best to decrease the data exchange between CPU and GPU. Usually OpenGL’s PBuffer are used to save the intermediate result in a texture on GPU and reuse it as an input data. Harris[3] had created a class called “Render to Texture” to easy the use of PBuffer. We had used this class in our program. 625.

Asserted Claims	Additional References and Disclosure
of the artificial neural network,”	*As a non-limiting example, Oh discloses the output data becoming input for a second layer of the artificial neural network:* As described above, the inner-product operation for each layer of an NN can be replaced with a matrix multiplication based on accumulating the input vectors and weight vectors. As such, the computation-per-layer can be written as follows: $W = \begin{bmatrix} w_{11} & w_{12} & w_{13} & \dots & w_{1N} \\ w_{21} & w_{22} & w_{23} & \dots & w_{2N} \\ \dots & \dots & \dots & \dots & \dots \\ w_{M1} & w_{M2} & w_{M3} & \dots & w_{MN} \end{bmatrix} = \begin{bmatrix} W_1 \\ W_2 \\ \dots \\ W_M \end{bmatrix},$ $X = \begin{bmatrix} x_{11} & x_{12} & x_{13} & \dots & x_{1L} \\ x_{21} & x_{22} & x_{23} & \dots & x_{2L} \\ \dots & \dots & \dots & \dots & \dots \\ x_{N1} & x_{N2} & x_{N3} & \dots & x_{NL} \end{bmatrix}$ $= \begin{bmatrix} X_1 & X_2 & X_3 & \cdots & X_L \end{bmatrix},$ $B = \begin{bmatrix} b_1 & b_1 & b_1 & \dots & b_1 \\ b_2 & b_2 & b_2 & \dots & b_2 \\ \dots & \dots & \dots & \dots & \dots \\ b_M & b_M & b_M & \dots & b_M \end{bmatrix},$

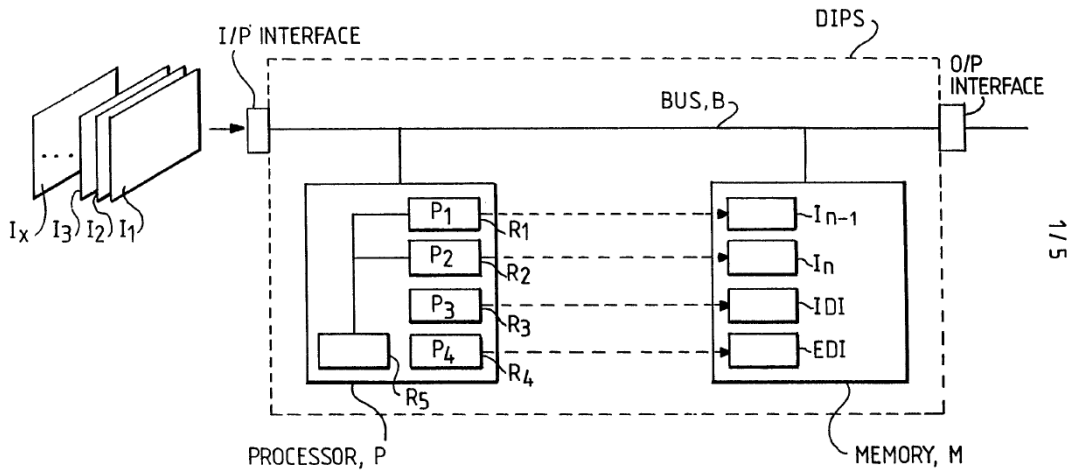
Asserted Claims	Additional References and Disclosure
	$M = W \times X + B$ $= \begin{bmatrix} W_1 \cdot X_1 & W_1 \cdot X_2 & W_1 \cdot X_3 & \dots & W_1 \cdot X_N \\ W_2 \cdot X_1 & W_2 \cdot X_2 & W_2 \cdot X_3 & \dots & W_2 \cdot X_N \\ \dots & \dots & \dots & \dots & \dots \\ W_M \cdot X_1 & W_M \cdot X_2 & W_M \cdot X_3 & \dots & W_M \cdot X_N \end{bmatrix}$ $+ \begin{bmatrix} b_1 & b_1 & b_1 & \dots & b_1 \\ b_2 & b_2 & b_2 & \dots & b_2 \\ \dots & \dots & \dots & \dots & \dots \\ b_M & b_M & b_M & \dots & b_M \end{bmatrix}$ $= \begin{bmatrix} m_{11} & m_{12} & m_{13} & \dots & m_{1L} \\ m_{21} & m_{22} & m_{23} & \dots & m_{2L} \\ m_{31} & m_{32} & m_{33} & \dots & m_{3L} \\ \dots & \dots & \dots & \dots & \dots \\ m_{M1} & m_{M2} & m_{M3} & \dots & m_{ML} \end{bmatrix}, \quad (2)$ $R = \text{sigmoid}(M)$ $= \begin{bmatrix} 1+e^{-m_{11}} & 1+e^{-m_{12}} & 1+e^{-m_{13}} & \dots & 1+e^{-m_{1L}} \\ 1+e^{-m_{21}} & 1+e^{-m_{22}} & 1+e^{-m_{23}} & \dots & 1+e^{-m_{2L}} \\ \dots & \dots & \dots & \dots & \dots \\ 1+e^{-m_{M1}} & 1+e^{-m_{M2}} & 1+e^{-m_{M3}} & \dots & 1+e^{-m_{ML}} \end{bmatrix}, \quad (3)$ where w_{ij} denotes the weight at the connection between the *i*th node of the output layer and the *j*th node of the input layer, M is the number of nodes in the output layer, and N is the number of nodes in the input layer. In addition, x_{ij} is the *i*th feature value of the *j*th input vector and bi is the bias term for the *i*th output node from L input vectors. The final result R_{ij} is the output of the *i*th output node for the *j*th input vector. 1312.

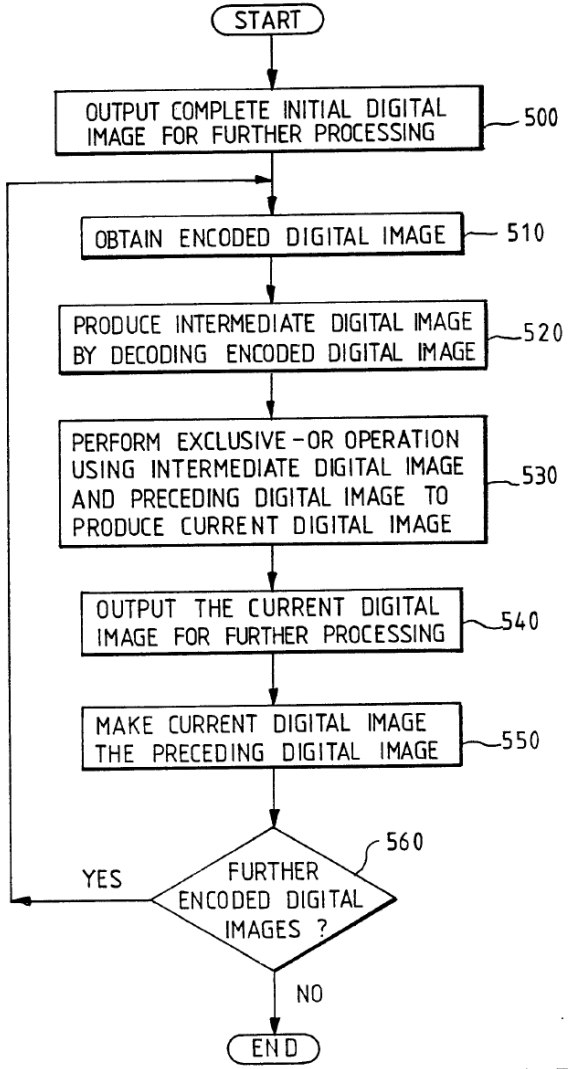
Asserted Claims	Additional References and Disclosure
	The above computation comprises of a matrix multiplication followed by a bias factor addition and sigmoid operation. The matrix multiplication is explained first. The method proposed by Moravanszky [1] is used to implement the matrix multiplication. The two matrices are converted into textures, denoted by texture W and texture X , then the matrix multiplication is performed by rendering. A rectangle is rendered to cover the whole screen. The vertex shader outputs the position and texture coordinates for each vertex of the rectangle, where each vertex has two texture coordinates: one for the row of texture W and the other for the column of texture X . For example, the upper left vertex will have the texture coordinates of the first row of texture W and the first column of texture X , while the upper right vertex will have the texture coordinates of the first row of texture W and the last column of texture X , and so on. As a result of the vertex shader, every pixel (i; j) has texture coordinates corresponding to the *i*th row of W and the *j*th column of X . The pixel shader then performs the inner-product between the row of W and the column of X specified by the texture coordinates. Fig. 1 shows an example of matrix multiplication using a GPU. The number of rendering passes required for matrix multiplication depends on the capability of the GPU, including the number of pixel shader operations and number of texture load operations. 1312–13. The bias term addition and sigmoid operation can be performed in one rendering pass. The bias texture and texture that contains the result of the matrix multiplication, texture W ×X , are set as the active texture. The vertex shader then outputs a full-screen rectangle as before. Each vertex’s texture coordinate for the texture W × X correspond to its position. For example, the upper left vertex has the texture coordinate (0; 0), while the texture coordinate for the upper right vertex is (1; 0). As the bias term is identical for one row, the bias term matrix is one-dimensional and the bias texture coordinates for each vertex correspond to its vertical position. The pixel shader adds two textures and performs a sigmoid operation. If there is more than one layer in an NN, the above procedure is repeated for each layer. The result of the previous layer is saved in the form of a render target texture, which is then used as an input for the next layer. Note that, even though an NN may have multiple layers, the GPU can perform all the operations after texture creation. 1313. *As a non-limiting example, Wilson discloses storing first output data in a second partition of the memory (e.g., a register), the second partition referenced by a second pointer, and swapping (e.g., exchanging) the first pointer with the second pointer at the end of the computation:*

Asserted Claims	Additional References and Disclosure
	Figure 1 illustrates schematically a digital image processing system DIPS for processing a sequence of digital images I_1 to I_x. The system comprises a memory M for storing current I_n and preceding I_{n-1} digital images, a processor P capable of accessing, via a bus B, the contents of the memory M and performing an exclusive-or operation between two given operands, and input (I/P) and output (O/P) interfaces for respectively receiving the sequence of digital images and outputting the compressed sequence of digital images. The processor comprises five registers R_1 to R_5. Registers R_1 to R_4 contain respective pointers P_1, P_2, P_3 and P_4. The pointers P_1, P_2, P_3 and P_4 are respectively used to determine where in the memory M the following are stored: the first data unit of the preceding digital image I_{n-1}, the first data unit of the current digital image I_n, the first data unit of the result of the exclusive-or operation between the current I_n and preceding I_{n-1} digital images or the result of decoding an encoded digital image (the intermediate digital image IDI), and the first data unit of the result of the encoded or decoded digital image EDI. Register R_5 is used as a temporary register to exchange the contents of the other registers R_1 to R_4. The system DIPS can operated in either image compression or image decompression modes. 5:9–28.

Asserted Claims	Additional References and Disclosure
	Having produced an encoded digital image EDI the contents P_1 and P_2 of registers R_1 and R_2 are exchanged such that the first digital image I_1 is discarded and the third digital image I_3 is loaded into memory M. Pointer P_1 is saved in register R_5. Pointer P_1 is set to equal pointer P_2 thereby discarding the first digital image I_1 and retaining the second digital image I_2. The second digital image I_2 thereby becomes the preceding digital image I_{n-1} relative to the third digital image I_3. The pointer P_2 is set to equal the contents of register R_5 and the third digital image I_3 is stored in memory M starting at the location determined by pointer P_2. Therefore, the first digital image I_1 is discarded or over-written by the third digital image I_3, the second digital image is pointed to by pointer P_1 and the third digital image is pointed to by pointer P_2. The system DIPS processes the second I_2 and **third I_3 digital images as per images I_1 and I_2. The encoded digital image EDI is then output via output interface O/P for further processing.** 6:25–7:2. Having produced the second digital image I_2 the contents P_1 and P_2 of respective registers R_1 and R_2 are exchanged such that the first digital image I_1 is discarded. Pointer P_1 is saved in register R_5. Pointer P_1 is set to equal pointer P_2 thereby discarding the first digital image I_1 and retaining the second digital image I_2. The second digital image I_2 thereby becomes the preceding decompressed digital image I_{n-1} relative to the third digital image I_3. The pointer P_2 is set to

Asserted Claims	Additional References and Disclosure
	equal the contents of register R_5. A further encoded digital image EDI is received via input interface I/P and stored in memory M starting at an address determined by pointer P_4. The encoded digital image EDI is decoded to produce an intermediate digital image IDI. The intermediate digital image IDI is stored in memory M starting at an address determined by pointer P_3. The processor performs an exclusive-or operation between the intermediate digital image IDI and the preceding decompressed digital image I_{n-1} to produce the current I_n or third I_3 digital image. The current I_n or third I_3 digital image is stored in memory M starting at an address indicated by pointer P_2. Therefore, the first digital image I_1 is discarded or over-written by the third digital image I_3, the second digital image is pointed to by pointer P_1 and the third digital image is pointed to by pointer P_2. The current I_{n-1} or third I_3 digital image is output via output interface O/P for further processing. 8:32–9:14. Referring to figure 5 there is shown a flow diagram representing the stages of image decompression according to the present invention. At step 500, because the initial digital image received has not been encoded and is complete it is output for further processing. At step 510 an encoded digital image is loaded into memory. The encoded digital image is decoded at step 520 to produce an intermediate digital image. An exclusive-or operation is performed between the intermediate digital image and the preceding decompressed digital image at step 530 to produce the current digital image. The current digital image is output for further processing at step 540. The pointers P_1 and P_2 are exchanged at step 550 thereby making the current digital image the preceding decompressed digital image. At step 560 a check is made to determine whether or not there are further encoded digital images to be processed. If so, processing of encoded digital images continues at step 500. If not, processing of encoded digital images terminates.

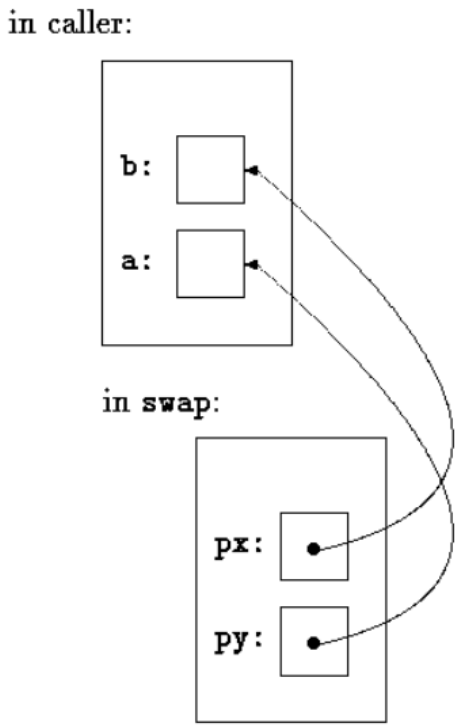
Asserted Claims	Additional References and Disclosure
	10:27–11:3.  FIG. 1. FIG. 1.

Asserted Claims	Additional References and Disclosure
	 FIG. 5.

Asserted Claims	Additional References and Disclosure
	As a non-limiting example, The C Programming Language discloses swapping pointers (e.g., passing pointers to the values to be changed):

Asserted Claims	Additional References and Disclosure

	5.2 Pointers and Function Arguments
	
	Since C passes arguments to functions by value, there is no direct way for the called function to alter a variable in the calling function. For instance, a sorting routine might exchange two out–of–order arguments with a function called swap. It is not enough to write
	
	`swap(a, b);`
	
	where the swap function is defined as
	
	`void swap(int x, int y)  /* WRONG */`
	`{`
	`  int temp;`
	
	`  temp = x;`
	`  x = y;`
	`  y = temp;`
	`}`
	
	Because of call by value, swap can't affect the arguments a and b in the routine that called it. The function above swaps *copies* of a and b.
	
	The way to obtain the desired effect is for the calling program to pass *pointers* to the values to be changed:
	
	`swap(`
	
	Since the operator & produces the address of a variable, &a is a pointer to a. In swap itself, the parameters are declared as pointers, and the operands are accessed indirectly through them.
	
	`void swap(int *px, int *py)  /* interchange *px and *py */`
	`{`
	`  int temp;`
	
	`  temp = *px;`
	`  *px = *py;`
	`  *py = temp;`
	`}`
	
	Pictorially:

Asserted Claims	Additional References and Disclosure
	 88–89. *As a non-limiting example, GPU Gems discloses swapping pointers:*

Asserted Claims	Additional References and Disclosure

	Listing 31-2. C++ Code to Set Up and Run a Reaction-Diffusion Simulation on the GPU
	See the source code on the accompanying CD for more details.
	
	`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);`
	
	Listing 31-2 *(continued)*. C++ Code to Set Up and Run a Reaction-Diffusion Simulation on the GPU
	
	`  // Initialize the state of the simulation with values in array`
	`  pugInitBuffer(rdBuffer, array);`
	`}`
	
	`void update_rd(){ // Call this every iteration`
	`  pugBindStream(rdProgram, "concentration", rdBuffer, currentSource);`
	`  PUGRect range(0, width, 0, height);`
	`  pugRunProgram(rdProgram, rdBuffer, range, currentTarget);`
	
	`  std::swap(currentSource, currentTarget);`
	`}`
	
	507–8.

Appendix B – ’438 Patent Obviousness References Invalidity Claim Chart

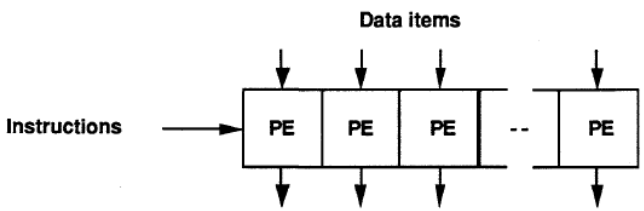
Asserted Claims	Additional References and Disclosure

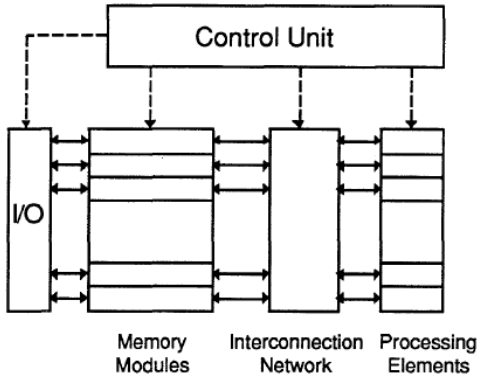
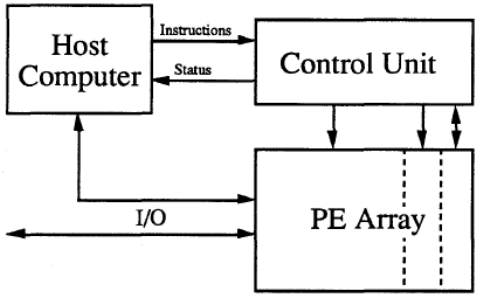
Asserted Claims	Additional References and Disclosure
	44.3.5 Conjugate Gradient Solver We now possess an easy-to-use class framework that completely abstracts the underlying hardware implementation, so we can easily write more-complex algorithms such as the conjugate gradient linear system solver. Using our implementation, the complete GPU conjugate gradient class appears in Listing 44-1. **Listing 44-1. The Conjugate Gradient Solver**

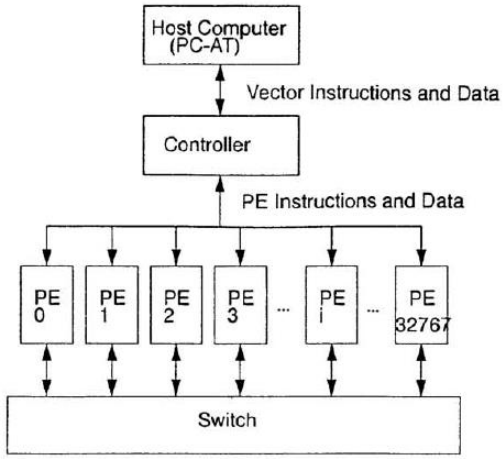
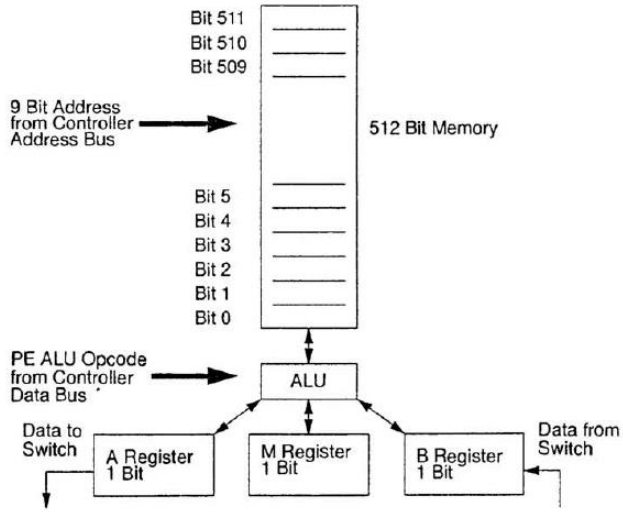
```
void clCGSolver::solveInit() {
  Matrix->matrixVectorOp(CL_SUB, X, B, R);  // R = A * x - b
  R->multiply(-1);  // R = -R
  R->clone(P);  // P =  R
  R->reduceAdd(R, Rho);  // rho = sum(R * R);
}
void clCGSolver::solveIteration() {
  Matrix->matrixVectorOp(CL_NULL, P, NULL,Q);  // Q = Ap;
  P->reduceAdd(Q, Temp);  // temp  = sum(P * Q);
  Rho->div(Temp, Alpha);  // alpha = rho/temp;
  X->addVector(P, X, 1, Alpha);  // X = X + alpha * P
  R->subtractVector(Q, R, 1, Alpha);  // R = R - alpha * Q
  R->reduceAdd(R, NewRho);  // newrho = sum(R * R);
  NewRho->divZ(Rho, Beta);  // beta = newrho/rho
  R->addVector(P, P, 1, Beta);  // P = R + beta * P;
  clFloat *temp; temp=NewRho;
  NewRho=Rho; Rho=temp;  // swap rho and newrho pointers
}
void clCGSolver::solve(int maxI) {
  solveInit();
  for (int i = 0; i < maxI; i++) solveIteration();
}

int clCGSolver::solve(float rhoTresh, int maxI) {
  solveInit(); Rho->clone(NewRho);
  for (int i = 0; i < maxI && NewRho.getData() > rhoTresh; i++)
    solveIteration();
  return i;
}
```

Asserted Claims	Additional References and Disclosure
	713. *As a non-limiting example, the Harris Dissertation discloses swapping “performing a swap” in simulating and rendering clouds.* I mentioned earlier that, in practice, each of the five steps in the algorithm updates a temporary grid and then performs a swap. RTT requires the use of two textures to implement feedback, because rendering to a texture while it is bound for reading is illegal (under the current specification (Poddar and Womack, 2001)). The swap in this case is merely a swap of texture IDs. The performance cost of RTT is therefore constant. CTT, on the other hand, requires only one texture. The frame buffer acts as a temporary grid, and a swap is performed by copying the data from the frame buffer to the texture. The performance cost of this copy is proportional to the texture size. Harris Dissertation at 61.
40[d][ii] “transfer the first output data to another memory during computation of the second layer of neurons, and”	*See* 1[d][iv], 12[c].
40[d][iii] “dictate an order of execution of instructions to the processing unit to perform the computation of the first layer of neurons.”	*See* 1[d][ii].
41 “The system of claim 40, wherein the processing unit	*See* 1[c][i]; claim 40.

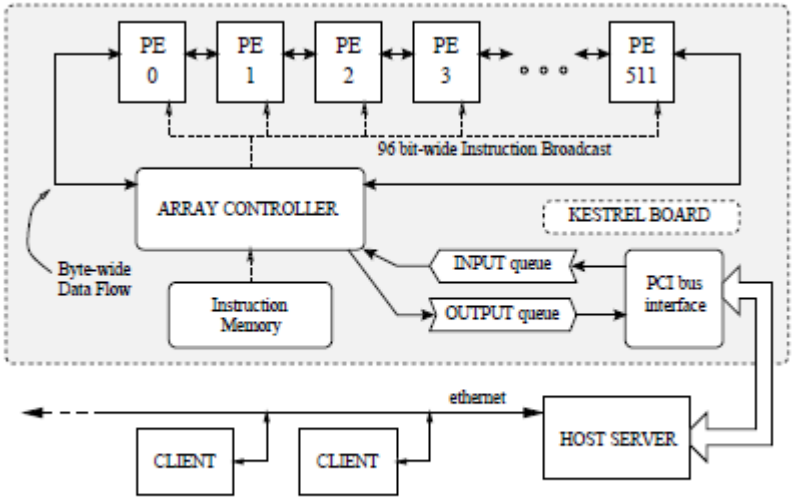
Asserted Claims	Additional References and Disclosure
comprises a graphics processing unit.”	
42 “The system of claim 40, wherein the controller is configured to send instructions for performing the at least one calculation to the processing unit.”	*See* 1[d][ii], claim 40. *As a non-limiting example, Svensson discloses a controller (e.g., control unit) sending instructions for performing at least one calculation to the processing unit (e.g., host computer and/or processing elements):*

Asserted Claims	Additional References and Disclosure
	1. Principles of SIMD In Single Instruction stream, Multiple Data stream (SIMD) processors one instruction works on several data items simultaneously by using several Processing Elements (PEs), all of which carry out the same operation as illustrated in Figure SIMD-1.  Principle of SIMD processor. Figure SIMD-1. A SIMD Processor has a single Control Unit reading instructions pointed to by a single Program Counter, decoding them and sending control signals to the PEs. Data are to be supplied to, and derived from, the PEs by a memory with as many data paths as there are PEs. Figure SIMD-2a shows the resulting processing structure which is also known as a Processor Array. The Interconnection Network provides flexibility in choosing source and destination for data to and from the PEs, necessary in many algorithms. The I/O system plays the role of converting – typically at very high rates – input/output data between the format of the outside world and the internal format of the array. The design of the I/O system is highly application dependent. In order to provide overall control of the Processor Array, as well as to execute sequential operations, it is common to use a conventional Host Computer which then views the Processor Array as an attached processor as illustrated in figure SIMD-2b. 1.

Asserted Claims	Additional References and Disclosure
	 Processor Array Figure SIMD-2a  Host Computer - Processor Array Relationship Figure SIMD-2b 2. *As a non-limiting example, Glover discloses a controller sending instructions for performing at least one calculation to the processing unit (e.g., host computer and/or processing elements):*

Asserted Claims	Additional References and Disclosure
	 Figure 1: A block diagram of the MM32k.  Figure 2: A block diagram of a single processing element (PE). 845.

Asserted Claims	Additional References and Disclosure
	As a non-limiting example, Mesa-Martinez discloses a controller (e.g., array controller) sending instructions for performing at least one calculation to the processing unit (e.g., processing elements (PEs)): The first generation UCSC Kestrel processor is implemented as a systolic array [8] of 512 processing elements (PEs). The system’s configuration as a linear array is ideal for such target applications as sequence analysis, at the same time that it provides a simpler and more flexible programming model than more complex multiple-instruction, multiple data stream (MIMD) machines would when applied to the same domain of problems [7]. For the applications that we were envisioning, programmability was more important than large local memories or reconfigurable interconnection fabrics [4]. We believe our SIMD approach to be a more cost effective solution than systems using fully reconfigurable general purpose multiprocessors [5], [6]. Each processing element in the array is 8-bits wide and contains its own local memory and datapath. The Kestrel board also contains several SDRAM banks that implement the instruction memory (IM) as well as the input/output queues (IQm, OQm) [3]. Instruction broadcasting and sequencing is performed by an on board controller, that is also in charge of memory management (Figure 3). The whole system is self contained and implemented as a single full length PCI card (Figure 2). Like most SIMD processors Kestrel broadcasts each instruction to each PE in the array. Due to the relatively large number of PEs that Kestrel has, and the fact that these PEs are distributed among several chips, instruction broadcasting can easily become more taxing than the actual time to execute the instruction. In order to solve this bottleneck Kestrel implements a board-level instruction issue and broadcast [4]. In order to handle inter-PE communication each PE shares a file of registers with its left and right neighbors. These register banks are known as Systolic Shared Registers (SSRs) [9]. Although this topology might seem limiting at first, it allows for computation and communication to occur concurrently. When a result is stored in the SSR after the instruction has been completed it is immediately visible to the next PE, thus communication takes place automatically (Figure 4). All addresses to the SSRs are issued globally in order to prevent adjacent PEs from writing to the same bank at the same time. This communication method is both elegant and efficient since it greatly simplifies the design of the PE and allowing most of its area to be dedicated for processing and memory. A-9.

Asserted Claims	Additional References and Disclosure
	 Fig. 3. Kestrel board architecture. FIG. 3. The controller is implemented using an FPGA. This device serves as the instruction sequencer for the system. The controller also handles the board’s I/O by interacting with the onboard PCI interface chip. The program instructions received from the host computer are stored in a separate instruction memory (IM). The Kestrel controller is programmed by using 54 bits of the 96-bit instruction word. The remaining bits are used as the instruction to be broadcast to the array. Each cycle, the controller must decide whether or not data is read from the input queue, whether or not data is written to the output queue, and whether or not the immediate field in the instruction should be replaced with data from the controller. A small amount of local memory and logic is used by the controller in order to enable loop counting and the recirculation of data among the PEs. When a critical condition is reached by the system the controller will generate the corresponding interrupt to inform the host about the board’s condition. A-10.

Asserted Claims	Additional References and Disclosure
43[a] “The system of claim 40, wherein the memory further comprises: a third partition to store internal variables; and”	*See* 6[a]; claims 28, 40. *As a non-limiting example, Oh discloses storing external data input patterns (e.g., input from an external camera):* In the case of using a neural network (NN) for image processing and pattern recognition, the main problem is the computational complexity in the testing stage, which accounts for most of the processing time. Moreover, NN-based image convolution has to exhaustively scan an input image in order to process an entire image [3]. Although an NN can be simulated using software, many potential NN applications require real-time processing, which means fully parallel specially designed hardware implementations, such as an FPGA-based realization of an NN. However, this is somewhat expensive and involves extra design overheads [4]. 1311. Recently, researchers have attempted text-based retrieval of image and video data using several image processing techniques [3]. As such, an automatic text detection algorithm for image data and video documents is important as a preprocessing stage for optical character recognition, and an NN-based text detection method has several advantages over other methods [3]. Therefore, this paper briefly describes such a text detection method, and readers are referred to the author’s previous publication for more details [3]. In the proposed method, an NN is used to classify the pixels of input images, whereby the feature extraction and pattern recognition stage are integrated in the neural network. The NN then examines local regions looking for text pixels that may be contained in a text region. Therefore, an M × M pixel region in the image is received as the input and a classified image is generated as the output. After the pattern passes the network, the value of the output node is compared with a threshold value and the class of each pixel determined, resulting in a classified image. GPU-based pipelining processing is used to reduce the processing time, and the GPU’s performance is maximized by accumulating a large number of input vectors 1 to create a two-dimensional texture. The input layer then receives the grey values for the pixels at predefined positions inside an M×M window over the input image. Experiments were conducted using an 11 × 11 input window size, with the number of nodes in each hidden layer set at 30. As a result, the processing time for pixel classification was significantly reduced using a GPU. Fig. 2(b) shows the pixel classification result for the left input image, where a black pixel denotes a text pixel. The classification using a GPU produced almost the same result as without a GPU.

Asserted Claims	Additional References and Disclosure
	1313.
43[b] “a fourth partition to store data used as input at a particular layer of neurons of the artificial neural network.”	*See* 6[b], 43[a], claims 28, 40. *As a non-limiting example, Oh discloses storing data used as input at a particular layer of neurons of the artificial neural network (e.g., outputs of a hidden layer in a neural network which are used as input for the next layer):* In the case of using a neural network (NN) for image processing and pattern recognition, the main problem is the computational complexity in the testing stage, which accounts for most of the processing time. Moreover, NN-based image convolution has to exhaustively scan an input image in order to process an entire image [3]. Although an NN can be simulated using software, many potential NN applications require real-time processing, which means fully parallel specially designed hardware implementations, such as an FPGA-based realization of an NN. However, this is somewhat expensive and involves extra design overheads [4]. 1311. Recently, researchers have attempted text-based retrieval of image and video data using several image processing techniques [3]. As such, an automatic text detection algorithm for image data and video documents is important as a preprocessing stage for optical character recognition, and an NN-based text detection method has several advantages over other methods [3]. Therefore, this paper briefly describes such a text detection method, and readers are referred to the author’s previous publication for more details [3]. In the proposed method, an NN is used to classify the pixels of input images, whereby the feature extraction and pattern recognition stage are integrated in the neural network. The NN then examines local regions looking for text pixels that may be contained in a text region. Therefore, an M × M pixel region in the image is received as the input and a classified image is generated as the output. After the pattern passes the network, the value of the output node is compared with a threshold value and the class of each pixel determined, resulting in a classified image. GPU-based pipelining processing is used to reduce the processing time, and the GPU’s performance is maximized by accumulating a large number of input vectors 1 to create a two-dimensional texture. The input layer then receives the grey values for the pixels at predefined positions inside an M×M window over the input image. Experiments were conducted using an 11 × 11 input window size, with the number of nodes in each hidden layer set at 30. As a result, the processing time for pixel classification was significantly reduced using a

Asserted Claims	Additional References and Disclosure
	GPU. Fig. 2(b) shows the pixel classification result for the left input image, where a black pixel denotes a text pixel. The classification using a GPU produced almost the same result as without a GPU. 1313. The bias term addition and sigmoid operation can be performed in one rendering pass. The bias texture and texture that contains the result of the matrix multiplication, texture W ×X , are set as the active texture. The vertex shader then outputs a full-screen rectangle as before. Each vertex’s texture coordinate for the texture W × X correspond to its position. For example, the upper left vertex has the texture coordinate (0; 0), while the texture coordinate for the upper right vertex is (1; 0). As the bias term is identical for one row, the bias term matrix is one-dimensional and the bias texture coordinates for each vertex correspond to its vertical position. The pixel shader adds two textures and performs a sigmoid operation. If there is more than one layer in an NN, the above procedure is repeated for each layer. The result of the previous layer is saved in the form of a render target texture, which is then used as an input for the next layer. Note that, even though an NN may have multiple layers, the GPU can perform all the operations after texture creation. 1313.
44[pre] “A computer system, comprising:”	*See* 1[pre].
44[a] “a central processing unit to receive input data acquired from an external system;”	*See* 1[a], 21[a].
44[b] “main memory, operably coupled to the central processing unit via a bus, to store the input data received by the central processing unit;”	*See* 1[b].
44[c] “an accelerator, operably coupled to the	*See* 1[c].

Asserted Claims	Additional References and Disclosure
central processing unit and the main memory via the bus, to receive at least a portion of the input data from the main memory, the accelerator comprising:”	
44[c][i] “at least one processing unit to perform a sequence of computations representing an artificial neural network on the at least a portion of the input data so as to generate output data, intermediate computations in the sequence of computations representing layers of the neural network and yielding intermediate results; and”	*See* 1[c][i].
44[c][ii] “accelerator memory, operably coupled to the at least one processing unit, to store the results of the sequence of computations; and”	*See* 1[c][ii].

Asserted Claims	Additional References and Disclosure
44[d] “a controller, operably coupled to the at least one processing unit and the accelerator memory,”	*See* 1[d].
44[d][i] “to control transfer of the at least a portion of the input data into the accelerator memory during performance of the intermediate computations in the sequence of computations by the at least one processing unit,”	*See* 1[d][iii].
44[d][ii] “to control transfer at least a portion of the output data from the accelerator memory to the main memory during performance of the intermediate computations in the sequence of computations by the at least one processing unit, and”	*See* 1[d][iv].
44[d][iii] “to control performance of the	*See* 1[d][ii].

Asserted Claims	Additional References and Disclosure
sequence of computations by the at least one processing unit.”	
45 “The computer system of claim 44, wherein the central processing unit is configured to receive the input data in response to a user interaction.”	*See* claims 2, 44.
46[a] “The computer system of claim 44, wherein: the central processing unit is configured to receive the input data at a first rate; and”	*See* 3[a].
46[b] “the at least one processing unit is configured to perform the sequence of computations at a second rate different than the first rate.”	*See* 3[b].
47 “The computer system of claim 44, wherein the main memory is configured to store a copy of the	*See* claims 4, 44.

Asserted Claims	Additional References and Disclosure
output data stored in the accelerator memory.”	
48 “The computer system of claim 44, wherein an output of at least one computation in the sequence of computations represents an output of at least one neuron in an artificial neural network.”	*See* claims 5, 44.
49[a] “The computer system of claim 44, wherein accelerator memory comprises: a first memory partition to store parameters common to all of the computations in the sequence of computations; and”	*See* 6[a]; claim 26, 44.
49[b] “a second memory partition to store data specific to at least one computation in the sequence of computations.”	*See* 6[b]; claim 27.
50 “The computer system of claim 44, wherein the controller is configured to transfer	*See* claims 7, 44.

Asserted Claims	Additional References and Disclosure
the output data from the accelerator memory to the main memory without transferring any of the intermediate results from the accelerator memory to the main memory so as to reduce data transfer via the bus.”	
51 “The computer system of claim 44, wherein the controller is configured to transfer at least a portion of the output data from the accelerator memory to the main memory after the at least one processing unit has begun to perform another sequence of computations.”	*See* claims 8, 44.
52 “The computer system of claim 51, wherein the controller is configured to initiate transfer of the at least a portion of the input data and to transfer the at least a portion of the output data in parallel	*See* claims 9, 51.

Asserted Claims	Additional References and Disclosure
with performance of at least one computation in the other sequence of computations by the at least one processing unit.”	
53 “The computer system of claim 44, wherein the controller is configured to control execution of the sequence of computations by the at least one processing unit.”	*See* claims 10, 44.
54 “The computer system of claim 44, further comprising: at least one of a video camera, a microphone, or a cell recording electrode, operably coupled to the central processing unit, to acquire the input data in real time.”	*See* 1[a], claim 44.