Optical Character Recognition on Graphics Hardware

Adrian Ilie
Department of Computer Science
University of North Carolina at Chapel Hill

Abstract.

The study of Artificial Neural Networks (ANNs) is a branch of Artificial Intelligence research that tries to emulate cognitive processes using computers. An example of such a process is optical character recognition (OCR). Using well-established models and algorithms, OCR can be reduced to simple matrix operations, which can be performed using today's computers.

However, these algorithms and models usually have high computational and/or memory requirements. This has led to several implementations of matrix operations that take advantage of specific characteristic of various hardware configurations in order to speed up the algorithms.

This paper presents a new algorithm that takes advantage of the features present in today's graphics cards in order to perform quickly and cheaply the computations required by OCR. We believe that our implementation is a viable alternative to classical hardware and software implementations because it comes at a lower price and provides higher overall efficiency by making use of the graphics card, a component that is usually idle during OCR.

1. Introduction.

OCR is the process by which characters are extracted from scanned images. The main benefits of this transformation are the possibility of searching for particular pieces of information and the reduced memory space text occupies while conveying essentially the same information.

One of the prevalent methods for OCR is to use artificial neural networks (ANNs). Traditionally, due to their parallel nature, ANNs have been implemented either using specialized parallel hardware or completely in software. Recent approaches employ general-purpose hardware that exhibits some degree of parallelism, combined with fine-tuned algorithms that take advantage of the special features present in the hardware. This paper presents a new approach that uses a different type of hardware, a graphics processing unit (GPU).

The paper organization is as follows: We first provide a short background on ANNs and describe some of the most widely-used architectures and algorithms in Section 2. In Section 3 we present some of the design considerations when using ANNs for OCR and describe how the algorithm can be reduced to matrix operations. Section 4 provides some basic graphics concepts necessary for understanding our implementation. Sections 5 and 6 describe the implementation of our algorithm: a fast matrix multiplication algorithm on GPUs combined with a GPU implementation of operations that are less efficient if implemented with matrices. Section 7 analyzes the suitability of our approach for ANNs and OCR. The paper is concluded in Section 8.

2. Artificial Neural Networks.

The anatomy of the human brain has been known for more than a century [1]. However, the way information is processed is still largely unknown, despite the progress in neurophysiology. This section describes ANNs, the way computers are used to emulate some of the known features of human brains.

2.1. The Formal Neuron Model.

Of the known features of the human brain, ANNs only implement two: parallel processing and *the formal neuron*, a very simple neuron model. The implementation is at a very basic level compared to the complexity of the anatomical system it tries to emulate. In spite of their limitations, relatively small ANNs (1000 neurons with 100 connections per neuron, compared to 10^{11} neurons with 10000 connections per neuron in a human brain) exhibit useful properties found in real brains: learning from examples, generalization, associative memory, and tolerance to failures of neurons and connections.

The formal neuron model used in most ANNs was proposed by McCulloch and Pitts [2]:

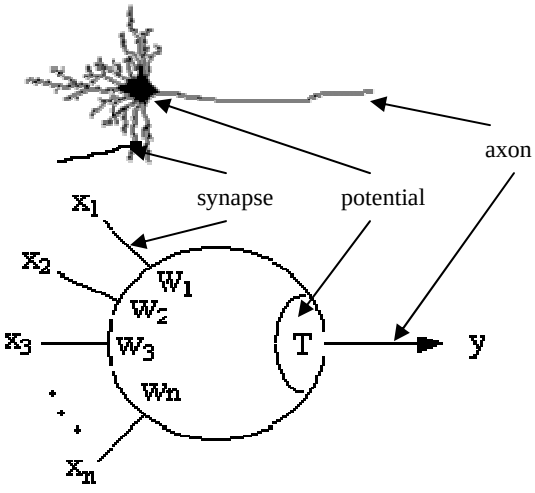

Figure 1: Real neuron vs. formal neuron.

In the brain, neurons are linked together by synapses through which they receive information. A *synapse* consists of one of a neuron's input fibers, its *dendrites* and another neuron's output fiber, or *axon*. Chemical transmitters received through synapses cause a neuron's membrane potential to change. If the potential exceeds a certain threshold, the neuron is activated and sends a nervous influx along its own axon. A synapse is characterized by its *strength*. A *learning process*, as described in [3], consists of modifications of a synapse's strength when the two neurons that it connects are activated simultaneously.

To emulate this behavior, the formal neuron model has binary inputs and outputs. It computes its potential as the sum of its inputs x_i weighted by the synaptic coefficients w_i. If the potential exceeds a threshold T, the neuron output is +1; otherwise it is -1. Without loss of generality, the threshold T is usually modeled as a bias of the potential, using an extra input that is always equal to 1 and its corresponding weight that is equal to $-T$. This model is powerful enough that networks of such neurons can emulate any finite Boolean function, provided the synaptic coefficients are set properly.

2.2. Network Architectures.

In ANNs, the topology of the networks is just as important as the neurons themselves. There are many possible architectures, but in this paper we only describe networks such as the one shown in Figure 2, called *feed-forward networks.*

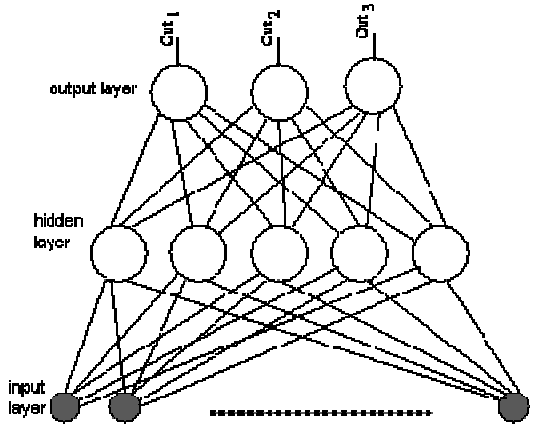

Figure 2: A basic ANN architecture.

The neurons are organized into *layers*. The *input layer* is formed of neurons that perform no computation, only distributing the inputs to the neurons in the next layers instead. The *output layer* is formed of neurons whose output is considered the network output. The neurons that are neither input nor output units are placed in several layers called *hidden layers*. Their role is ultimately to memorize features of the inputs through the learning process.

The goal of the learning process is to modify the weights of a given architecture's synapses such that the resulting network implements a given Boolean function. If the weights are modified iteratively, the learning process is called *training*. If training is done to minimize a cost function that measures the error between the actual and desired outputs, the process is called *supervised training.*

2.3. The Backpropagation Algorithm.

One example of supervised learning is *the backpropagation algorithm* [4], which uses gradient descent to minimize the cost function. The network architecture is similar to the one in Figure 2: one input layer at the bottom, one output layer at the top, and several hidden layers in between. Connections are only allowed from lower layers to higher layers and may skip intermediate layers.

In its classic form, the algorithm consists of two steps: computing the outputs of all the neurons during the forward propagation pass, followed by computing the errors and modifying the synaptic weights in the backpropagation pass.

The total input, x_j, to neuron j is the weighted sum of the outputs, y_i, of neurons connected to it and multiplied by the synaptic weights, w_{ji}, of each connection:

$$x_j = \sum_i y_i w_{ji} \qquad (1)$$

The output of each neuron is computed as an increasing differentiable non-linear function of its input. An example of such a function is the sigmoid:

$$y_j = 1 \Big/ \left(1 + e^{-x_j}\right) \qquad (2)$$

Input vectors are presented to the input layer. The states of neurons on the other layers (the hidden layers and the output layer) are determined by applying equations (1) and (2) layer by layer using the weights of the connections from lower layers.

The error is computed as a distance function between the outputs, y_{jc}, and the desired outputs, d_{jc}:

$$E = \tfrac{1}{2} \sum_c \sum_j \left(y_{jc} - d_{jc}\right)^2 \qquad (3)$$

This function is minimized by gradient descent and its partial derivatives with respect to each synaptic weight are used to correct the weights. The process is repeated for a training set composed of inputs and their corresponding outputs. It is beyond the scope of this paper to describe the algorithm in detail. The interested reader is referred to [4]. The details of our implementation are presented in Section 6.

Once a network has been trained, it can be used to recognize sets of inputs based on their similarity with the internal representations constructed in the training process.

The most obvious drawback of this method is that the error function may have local minima, and finding the global minimum is not guaranteed. However, experience shows that this situation can be avoided by using alternative ways to correct the weights. Also, the method requires careful design of the network architecture for each task and choosing the output functions; otherwise the convergence speed may vary by orders of magnitude. Although the model is not an anatomically accurate model of the learning process in brains, the results obtained in applications show that internal representations can be constructed by gradient descent in weight space.

3. Optical Character Recognition Using Artificial Neural Networks.

One of the tasks the backpropagation algorithm performs very well in is optical character recognition (OCR). Some details of implementing OCR using ANNs are described in this section.

3.1. Design Considerations for Network Architectures Used in OCR.

The OCR process consists of the following steps:

- scan the images containing the text;
- segment the scanned images into tiles containing just one character;
- process each tile to enhance its contrast;
- scale each processed tile to a predetermined size;
- apply the recognition to each scaled tile.

The size to which each tile is scaled is determined by the size of the input layer. For example, an input layer with 64 neurons may correspond to a scaled image tile of 8x8 *pixels* (picture elements), and each pixel's color can be converted to a binary input using a threshold value.

The size of the output layer is determined by the number of symbols that need to be recognized and their encoding. For example, if a network has to recognize the 10 digits present in decimal numbers and the digits are binary encoded, then 4 binary digits are sufficient for encoding, and the output layer has 4 neurons. Another possible encoding is to have 10 neurons on the output layer, with only one being activated for each of the 10 different digits.

Although there are several ANN architectures that can be applied to OCR, their design starts with the same steps that determine the number of neurons on the input and output layers. The main differences are in the design of the hidden layers. There are no universal rules for designing the hidden layers, just rules of thumb derived from experiments. It is beyond the scope of this paper to describe the ANN architectures used for OCR in more detail. The interested reader is referred to [5].

A possible network architecture for OCR is as follows: an input layer with one input for each pixel of the input image, two hidden layers for detection and recognition of horizontal and vertical features respectively, and an output layer. Since the input layer is organized as a rectangular grid, it is common practice that the hidden layers are also rectangular.

To "encourage" recognition of local features in characters, neurons in an inferior layer distribute their output only to some part of the neurons in the layer above, forming "pyramids" with overlapping bases, as shown in 2D in the Figure 3 below. The neurons on the output layer are an exception, being connected to all the neurons on the hidden layer immediately below. This structure is "suggested" in the training phase by setting the starting weights accordingly: larger values inside the pyramids than outside them.

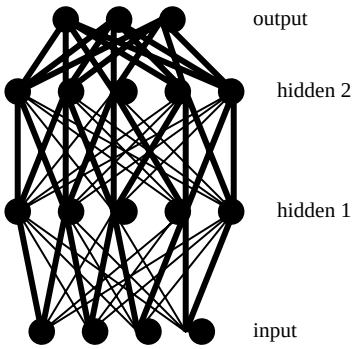

Figure 3: Pyramidal structure of a feed-forward ANN. The thicker connections are stronger.

Comparisons between the performances of different network architectures in OCR [5,6] have shown that more complex architectures can outperform classical networks in accuracy while also requiring less memory capacity. Due to their complexity, such networks have been traditionally implemented in specialized parallel hardware consisting of many interconnected programmable units. While in this paper we concentrate on simple architectures, our method is applicable to more complex architectures as well.

3.2. Software Implementations.

Neural networks have been traditionally implemented in software using matrices. For example, the weights of the connections between two consecutive layers of sizes N_1 and N_2 can be mapped to a N_1xN_2 matrix.

Some of the operations involved in the backpropagation algorithm also map easily to matrix operations. Computing the input to a neuron as in equation (1) is in fact a dot product, the basic operation in matrix multiplication. Computing the input to a layer of neurons maps to a matrix product. Also, function calls can easily be converted to table lookups, which in turn are implemented as vector indexing. Other steps in the gradient descent

minimization similarly map to matrix operations and dot products, although the mapping is not straightforward. Other training algorithms have similar implementations using matrix operations. Most implementations available today make extensive use of matrix libraries.

The existence of this mapping between ANNs and matrices brings about the conjecture that a matrix-based implementation would greatly benefit from speedups in matrix operations, especially matrix multiplications. Speeding up matrix operations would be a direct application of the well-known principle from computer architecture: making the common case fast [12]. This is the reason why, in Section 5, we describe a matrix multiplication algorithm implemented on graphics hardware that can replace other matrix libraries used traditionally in implementing ANN algorithms.

Even though generalized matrix multiplication (GMM) is the most commonly used operation in implementing ANNs, the algorithms involve other operations that cannot be easily mapped to GMMs. The traditional approach is to take advantage of the inherent parallelism in matrix libraries and find a mapping from these operations to matrix operations, even if the mapping is counterintuitive or counterproductive. The other available option is to implement these operations separately, without using matrices. We think the latter approach is better for an implementation on graphics hardware. Section 6 describes our vision of such an implementation.

4. Some Helpful Graphics Concepts.

Before we can present the details of the algorithms in this paper, we must provide a brief and high-level overview of some of the fundamental concepts in graphics and their role in our method. For an extensive description of these concepts, the interested reader is directed to [14,15]. In this section we concentrate on presenting the concepts and their implementation using *OpenGL*, a standard graphics API (Application Programming Interface).

4.1. The Structure of a GPU.

In this paper, we use the term GPU to refer to the graphics cards commonly present in personal computers. Examples of GPUs from different manufacturers include nVidia's GeForce, ATI's Radeon and Matrox's Parhelia. This subsection presents the components of a GPU that we refer to in the rest of the paper.

The most important component of a graphics card is the *graphics processor*, a highly-specialized processor that executes graphics operations on data sent from the CPU (the central processing unit, also known as the microprocessor) before sending it to the display.

The other important component of a graphics card is the *graphics memory*. The components we are interested in are the frame buffer and the texture memory. The *frame buffer* is a buffer that usually contains the final image sent to the screen. It is the memory buffer objects are "drawn" into, a process called *rendering*. The contents of the frame buffer can also be *read back* for various purposes. We use the read back facility to retrieve the results. The *texture memory* is a part of graphics memory that holds images called *textures*.

Graphics memory can usually be addressed as elements in a 2D or 3D coordinate system. For example, pixels in the frame buffer and *texels* (texture elements) in a 2D texture are organized into a 2D coordinate system upon which all the library calls are based. Each pixel has 4 components: three colors (red, green and blue) and alpha. While in graphics applications the alpha component is used for transparency effects, we choose to treat all the 4 components consistently as a vector of 4 elements.

4.2. The OpenGL Library.

One of the most widely-used graphics APIs is OpenGL. In this subsection, we define the most important concepts our algorithms use and present some of the corresponding OpenGL calls that implement their functionality.

In OpenGL, 3D objects are formed of surfaces represented by polygons. A 3D polygon received from the CPU is transformed into a 2D polygon suitable for display, using a mathematical computation called *projection*. The hardware draws 2D polygons on the screen by converting them to groups of pixels, a process called *rasterization*.

A polygon description includes information about its vertices: their position, and sometimes their color and texture coordinates. The *texture coordinates* of a vertex are the position of the corresponding texel in a texture. A texture is usually a 2D image that can alter the appearance of polygons.

The process of altering the appearance of polygons using textures is called *texture mapping*. It consists of computing the color of a pixel by linearly interpolating the coordinates of its corresponding texel. The interpolation is based upon the pixel's position with respect to the polygon's vertices and their texture coordinates. Reading the color information from a texture using texture coordinates is called *texture lookup*. If more than one texture is used to compute the final color for a polygon, the process is called *multi-texturing*. There are several modes to compute a final color from multiple textures. We are interested in multiplication, set by the library call `glTexEnvf (GL_TEXTURE_ENV,GL_TEXTURE_ENV_MODE, GL_MODULATE)`.

The basic operating mode of OpenGL is copying data from a source to a destination. Sources can be constant colors, rectangular regions of textures, etc. Instead of just overwriting the destination with the source, OpenGL provides the *blending* facility, which allows for color blending operations such as weighed summation and multiplication. The library call that controls blending is `glBlendFunc`, which takes two arguments: the two terms with which the source and the destination are multiplied with before being summed and stored at the destination. The default blending mode is the simple summation of colors, `glBlendFunc(GL_ONE,GL_ONE)`, which results in the source and destination being multiplied by 1 and then added together.

Blending only allows for combinations of the 4 pixel components separately. To combine information from the 4 color channels, OpenGL uses the *color matrix*, a matrix that is multiplied with the 4 values of a pixel before writing to the destination. This makes possible operations such as exchanging components and weighted summations between them. The color matrix is set by the `PixelTransfer` library function. The actual transfer of the resulting color to the frame buffer is done either by drawing graphics primitives, or by copying pixel regions using `glDrawPixels`.

As mentioned in the previous subsection, we use the read back facility of the frame buffer to read back the final results of the computation to main memory. The corresponding library function is `glReadPixels`.

Having described both the theoretical concepts used in our algorithms and their implementation counterparts, we can now describe the algorithms themselves.

5. Matrix Multiplication on GPUs.

This section describes the GPU matrix multiplication algorithm from [8]. This approach can be used as a library for a classic implementation of ANNs, backpropagation and ultimately OCR.

5.1. Algorithm Description.

The authors of [8] describe their work as somewhat forward-looking, because the hardware required to fully implement the algorithm with high enough precision to make it useful in general-purpose applications is not yet available. However, available graphics hardware is specialized in a manner that makes it suitable for certain applications, giving faster results than general-purpose CPUs. In particular, matrix multiplication is traditionally memory-limited, and the availability of a large and fast texture memory makes graphics hardware that much more useful.

The algorithm shown in Figure 4 literally "draws" the multiplication process on the screen: *k* parallel rectangles (each *m* x *n*) are rendered using orthographic projection one behind the other, with elements of the matrices *A* and *B* texture-mapped onto each rectangle. Setting the multi-texturing mode to "modulate" with `glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE)` computes the terms of the dot products of the lines of *A* and columns of *B*. Setting the blending mode to "sum" with `glBlendFunc(GL_ONE,GL_ONE)` results in adding up the terms to form the final result, which is then simply read back to main memory with `glReadPixels`.

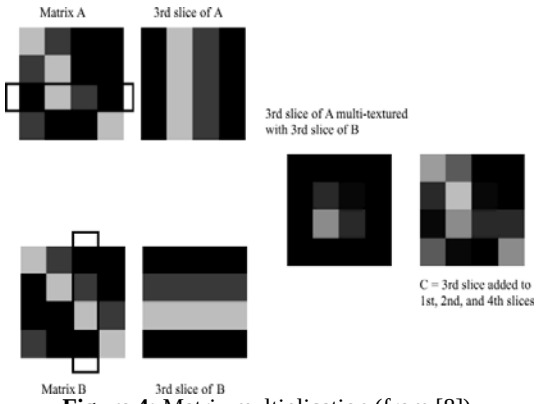

Figure 4: Matrix multiplication (from [8]). Numbers are mapped to different color intensities, shown as different shades of gray.

The algorithm has the following steps:

- *k* rectangles of size *m* x *n* are to be rendered using orthographic projection one behind the other;
- each line of *A* and column of *B* are converted into rectangular textures of sizes *1* x *m* and *n* x *1*, respectively (*k*+*k*=*2k* textures);
- each rectangle is textured with one texture from *A* and one texture from *B*, using "modulate" as the multi-texturing mode;
- the blending mode is set to "sum";
- all the *k* rectangles are rendered on the screen;
- the result is read back into main memory.

The computation shown in Figure 4 takes place on one of the 4 color channels available on a GPU. Grouping the elements of A, B and C into chunks of four elements speeds up the algorithm by fully utilizing the memory bus and the 4 color combiners of the GPU.

5.2. Algorithm Performance and Applications.

The memory system available to a GPU is 128 (nVidia) or even 256 (ATI and Matrox) bits wide and runs at double data rate, allowing 256 (512) bits to be read or written per GPU clock cycle. This results in a memory bandwidth of more than 8 GB/sec. The memory bus is split into 4 independent 32 (64) bit buses that can read or write data concurrently. The bandwidth is almost 4 times the memory bandwidth of a CPU (2.1 GB/sec for an AMD processor and 1.5 GB/sec for an Intel processor). Even though CPUs are almost 10 times faster than GPUs, in memory-limited applications such as matrix multiplication, it is

difficult to keep the CPU going at full speed. The performance of the algorithm is the same as the performance of a standard CPU-based matrix multiplication library (ATLAS [11]) in number of operations per second. However, it is important to note that the GPU operations have 8 bits of precision, while the CPU operations have 32 bits.

Unfortunately, most graphics architectures implement the accumulation used in the final blending stage by doing a frame-buffer read, while CPUs can keep results in registers. The authors of [8] show that the time taken by an elementary operation is almost equivalent to the time it would take for two texture reads (for the current elements of *A* and *B*), one frame-buffer read (for the previous value of the current element of *C*), and one frame-buffer write (for the result). This means that, just as in the CPU case, the application is bandwidth-limited. An embedded frame-buffer that avoids reads and writes (the conceptual equivalent of a cache) would dramatically improve performance.

The main drawback of the current implementation is its limited precision: most GPUs available today only have 8 bits of precision in the color buffers. Moreover, GPUs use saturation arithmetic, since it makes sense that adding intensity to an already saturated color (white) should not make it wrap around to black. The solution the authors of [8] propose is using *1/k* as the polygon color. Since polygon color gets multiplied into each product, this effectively shifts off the least significant *k* bits of the result. This gives a non-saturated but potentially erroneous result because the truncated bits might have introduced carries that should have been taken into account. Other approaches [9,10] are more general and rigorous, using range scaling to guarantee that no clamping occurs at any point and that the computation takes advantage of the full precision available in hardware. Also, the graphics hardware market is driven by the game industry's demand. Game developers have expressed interest in high-precision, even floating-point arithmetic for graphics hardware. The latest graphics adapters already feature full IEEE 754 floating-point arithmetic, which makes computational applications that much more feasible.

The application the authors of [8] mention for their matrix multiply implementation is computing path lengths in a graph by raising the graph's adjacency matrix to some large power. Using a divide and conquer approach to perform fewer matrix multiplies and performing all the operations in the graphics card's memory makes this implementation one of the fastest available.

Evaluating this implementation of GMM (generalized matrix multiplication) gave us more than the performance numbers. The authors of [8] present good ways to expand their research, but fail to mention other implications of the conclusion of their paper: graphics hardware has some limited programmability that can be exploited. One of these implications, mentioned in [13], is that programming the graphics hardware is very similar to programming in an assembly language. OpenGL commands are function calls that operate on graphics memory and a few internal stacks. While very limited, this capability can be used to perform other operations.

6. Backpropagation on GPUs.

This section presents some of the practical aspects of implementing OCR in graphics hardware using the backpropagation algorithm for feed-forward ANNs. The approach is similar to the one presented in [13] for Kohonen maps, but uses a different memory organization and implements a different algorithm using roughly the same fundamental operations. It also uses the matrix multiplication algorithm presented in the previous section.

6.1. Mapping ANNs to GPU Memory.

There are several ways the structure of a feed-forward ANN can be mapped onto pixels in the frame buffer. For this paper, we consider a particular case in which only connections between adjacent layers are allowed. This is a reasonable restriction for the OCR application, and more general cases are a straightforward generalization. The connections between two adjacent layers, L_{i-1} and L_i, of sizes N_{i-1} and N_i can be represented as a matrix of size N_{i-1} x N_i. As in [8], this matrix can then be represented as a rectangle of size (N_{i-1} / 4) x (N_i / 4) pixels by using all 4 color channels. The outputs can also straightforwardly be mapped to vectors of size N_i / 4 pixels for layer L_i. The inputs of layer L_i coincide with the outputs of layer L_{i-1}, and can be duplicated for convenience and ease of programming at the expense of memory. Summarizing, there is a (N_{i-1} / 4) x (N_i / 4) "window" on the screen for each hidden layer and for the output layer. In such a window, a row of pixels represents the synaptic weights of the connections to a neuron from neurons on the layer immediately below, and successive rows belong to neurons on the same layer.

Packing 4 weight values into a pixel provides a rather counter-intuitive mapping from 4 synaptic weights to 1 pixel, but reduces memory constraints. Alternatively, the mapping can be 1 weight per pixel, allowing the remaining 3 memory locations to store intermediate results, or weights of connections from other layers below, relaxing the initial restriction that connections are only between adjacent layers.

Memory allocation also has to take into account the total size of video memory and its partitioning into buffers and texture memory; otherwise time-consuming pixel transfer operations are necessary to swap contents of various parts of video memory in and out of main memory.

6.2. Implementing Backpropagation Using OpenGL Library Calls.

Standard graphics operations have predictable effects on the screen, and we can program a sequence of operations that would have the desired effect on the ANN structure presented in the previous subsection. This subsection describes the implementation of the operations required by the backpropagation algorithm.

The operations needed by the backpropagation algorithm are: summation, multiplication, subtraction, dot product and function evaluation. Applying an operation is implemented by drawing primitives or copying pixel regions after setting the appropriate blending mode. Depending on the memory organization, setting the color matrix may also be required. In the following equations, *src* is the source, *dst* is the destination, and *ct* is a constant.

Summation is the default blending mode, implemented by `glBlendFunc(GL_ONE, GL_ONE)`:

$$dst = src * 1 + dst * 1 \quad (6)$$

Multiplication by a constant is implemented by calling `glBlendFunc(GL_ZERO, GL_SRC_COLOR)`:

$$dst = src * 0 + dst * src \quad (7)$$

Subtraction of two values is implemented by enabling blending with `glBlendFunc(GL_ZERO, GL_ONE_MINUS_SRC_COLOR)`:

$$dst = src * 0 + dst * (1 - src) \quad (8)$$

The dot product can be implemented either naïvely, with summations and products, or as in [8], using the summation blending mode combined with multi-texturing, then drawing primitives one by one, or copying pixel regions.

OpenGL does not provide standard operations for function evaluation. Instead, we store precomputed tables of values in the frame buffer, and retrieve them by copying and blending; or we store them into textures, and retrieve them with texture lookups.

The algorithm is literally "drawn" on screen, just as in [8] – processing is done entirely by the GPU in the frame buffer and the texture memory. Pixels are set with small random values for training or the values computed during training for pattern recognition. The library call for setting pixel values is `glDrawPixels`. The results are read back with `glReadPixels`.

Using the above operations, the forward phase of the backpropagation algorithm consists of the following steps:

- the net output of each neuron is computed using the dot product operation:

$$net_{pj} = \sum_i w_{ij} o_{pi} \quad (9)$$

where net p_j is the net output, w_{ij} are the weights of the connections to neurons on the lower layers and o_{pi} are the outputs of those neurons;

- the output of the neuron is then computed using the function evaluation operation:

$$o_{pj} = f_j(net_{pj}) \quad (10)$$

where o_{pj} is the final neuron output and f_j is the non-linear activation function;

- the process is repeated for all neurons and all layers in increasing order.

The forward phase is used both as a step in one iteration of the training algorithm and in the pattern recognition algorithm. Precomputing all the outputs before the dot products allows us to rewrite the dot products as a matrix product and use the approach presented in the previous section.

The implementation of the backward phase of the algorithm consists of the following steps:

- the error signal is computed for all the output units using the subtraction (8), function evaluation and multiplication (7) operations:

$$\delta_{pj} = (t_{pj} - o_{pj}) f'(net_{pj}) \quad (11)$$

where δ_{pj} is the error signal, t_{pj} is the threshold of the neuron, o_{pj} is the output, and $f'(net_{pj})$ is the derivative of the activation function;

- the weight change is computed using the multiplication operation (7), according to the generalized delta rule:

$$\Delta_p w_{ji} = \eta \delta_{pj} o_{pi} \quad (12)$$

where $\Delta_p w_{ji}$ is the weight change, and η is a proportionality constant;

- the weights of all the connections from neurons in lower layers to the current output neuron are updated using the summation operation (6):

$$w_{ji}' = w_{ji} + \Delta_p w_{ji} \quad (13)$$

where w_{ji} is the old weight, and w_{ji}' is the new weight;

- the process is similar for hidden neurons, but the error signal is computed recursively from the error signals of the neurons the current neuron is connected to, using the function evaluation, multiplication (7) and dot product operations:

$$\delta_{pj} = f'(net_{pj}) \sum_k \delta_{pk} w_{kj} \quad (14)$$

where δ_{pj} are the error signals of the neurons on the upper layers the current hidden neuron is connected to;

- the process is repeated for all neurons on all layers in decreasing order.

The backward phase is used as a step in one iteration of the training algorithm. While there are ways to rewrite it as a matrix operation, we believe this approach is more straightforward and faster.

Summarizing, the training algorithm consists of several iterations of the forward phase followed by the backward phase until the error drops below a

threshold; and the pattern recognition algorithm consists of applying the forward phase in a trained network to a new set of inputs. The forward phase is implemented as several matrix multiplies because recognition needs to be as fast as possible. The backward phase is implemented using other graphics operations in order to move all the computation to the GPU and avoid costly memory transfers between the GPU and the CPU.

7. Suitability of GPUs for OCR.

The previous two sections presented our way of implementing ANNs, backpropagation and OCR on graphics hardware. We believe that the combined approach of using the fast matrix operations presented in Section 5 when possible, and reverting to the other operations presented in Section 6 when needed, ensures optimal performance of the GPU-based algorithm. This section evaluates its suitability for OCR when competing with CPU-based implementations.

One such CPU-based approach for matrix multiplication is presented in [7]. It takes advantage of the hierarchy of caches of a Pentium III processor, and optimizes the code for each level of the hierarchy to minimize inter-level communication. At level L0, input data and intermediary results are kept in the registers for as long as possible to delay and ultimately hide the latency of memory reads and writes. At level L1, contiguous chunks of data are prefetched to ensure that the CPU is fully occupied most of the time, and that reads and writes to main memory and the L2 cache are minimal. At level L2, the same strategy is applied to ensure accesses to main memory are minimal. Also, an important role in the speed of the algorithm is played by the use of SSE (Streaming SIMD – "Single Instruction Multiple Data" Extensions) instructions that perform the same operation on 4 different sets of data in parallel. This optimization at every level brings about optimal overall performance: the implementation is 2.9 times faster than the publicly-available GMM library, ALTAS [11].

Our GPU approach is simpler, and relies heavily on the availability of texture memory. The texture memory effectively acts as a very fast (effectively 500-1000 MHz) and very large (128 MB) 4-way associative L1 cache. When performing matrix multiplications, our algorithm employs a brute-force method, using highly-optimized graphics operations that produce a result equivalent to the one of the multiply-accumulate used in matrix multiplications: multiplications of texture maps and accumulation in the frame buffer. Similarly to the CPU case, the parallelism comes from using all the 4 color channels simultaneously. The other operations needed in the training phase, even if implemented with graphics library calls, are also likely to be of comparable speed in a hypothetical CPU-based implementation.

The majority of operations in OCR are the pattern recognition operations, which map to a few matrix multiplications. The training phase can be considered as preprocessing, and its cost can be amortized over many uses of the trained network for recognition. It follows that the vast majority of operations in OCR are matrix multiplications. This leads us to the conclusion that at this time a CPU implementation is preferable. Not only is it faster, but also it is more flexible, and offers more precision. We now provide some arguments why we think a refined future GPU implementation will be competitive.

First, as seen in the previous sections, graphics hardware architectures are in fact a collection of primitive operations. Various computations can be implemented as the operations of the texture mapping unit and the frame buffer. Final results can be obtained in one or more rendering passes. This gives an arguably limited, but nonetheless important, applicability that has already been exploited and shown effective in computational applications. In conclusion, the advantage of CPUs is in fact negligible in terms of flexibility.

Moreover, languages are developed for programmable procedural shading systems as well as compilers that automatically generate instructions corresponding to rendering operations on graphics hardware. These efforts match closely the ongoing trend in introducing graphics hardware programmability that was witnessed in the last few years. Although still graphics-oriented, these tools will also increase the flexibility and usability of GPUs for non-graphics applications. The availability of these tools makes the CPUs and GPUs comparable in terms of usability and accessibility.

Being pushed by the game industry, the speed of graphics hardware doubles approximately every 6 months. This rate is much faster than the improvement in the clock rates of CPUs. Also, texture memory has been traditionally faster than main memory, and this trend is likely to be maintained. In the long run, if technological limitations occur, GPUs may follow in the CPUs' footsteps, having texture memory organized as a hierarchy of caches. In conclusion, because the components of a GPU (graphics processor and graphics memory) are tightly integrated and perform better together than the CPU-main memory ensemble, we think the speed of the GPU-based approach will soon surpass the speed of the CPU-based approach.

The above considerations are valid arguments that many computational applications can be implemented on graphics hardware. ANNs are a special category of applications that may have other potential benefits from being implemented on graphics hardware. Following the biological model, ANNs are formed of many interconnected neurons. The calculation inside each neuron is simple, but there are usually a large number of units. Therefore, a practical use of the

ANNs typically demands parallel supercomputers. Commodity graphics hardware can perform pixel-oriented operations very efficiently. Not only are the operations pipelined in dedicated hardware, but there are also usually up to 4 color channels and multiple pixel pipelines available that essentially provide parallel processing. The efforts for graphics hardware programmability are in a similar direction: each vertex in the incoming data and each pixel in the frame buffer can have a specified programmable behavior. Not only does this provide a better and more flexible way to implement current ANN algorithms, but also it offers the opportunity for new and more intuitive research models to be implemented quickly, easily, and cheaply.

The precision problem is less relevant in ANNs and OCR, as the weights are always between 0 and 1. Usually, 8 bits of precision are enough for representing synaptic weights. The problem can be alleviated as shown in [9,10], and will soon be solved in hardware, since the manufacturers already started implementing full floating-point precision throughout the entire graphics pipeline.

Perhaps the most important argument in favor of using GPUs is the price. CPU prices are slightly higher than the price of a high performance graphics card. Moreover, computationally-intensive applications require large memories that are not included in the price of a CPU.

Besides the arguments provided above for general-purpose applications and ANN algorithms, there is one additional benefit to implementing OCR in graphics hardware: during the scanning process, the CPU is usually busy communicating with the scanner, while the GPU is mostly idle. Using it for OCR guarantees a better overall usage of system resources. Moreover, a GPU implementation does not even have to be of comparable speed to a CPU implementation: it only has to take an amount of time comparable to the time it takes the CPU to scan the image. The traditional sequential processing (first scan, then convert to text) becomes parallel. The CPU can send the scanned data to the GPU as soon as it receives it from the scanner, yielding a dramatically improved overall performance.

8. Conclusion.

In this paper we presented a new approach for implementing the ANN backpropagation algorithm. The approach combines a fast matrix multiplication algorithm with fast execution of other operations using graphics library calls. The advantages of using this approach include its price, its speed (already relatively fast and likely to improve in the near future), and its applicability in everyday situations where fast OCR is desired. Its main disadvantage is the limited precision: since we did not implement our approach, we have no sense on how the limited precision available in graphics hardware affects the OCR precision in correctly recognizing characters. Overall, considering the current trends in graphics hardware and CPUs, we think our approach is certainly worth further study.

Bibliography.

[1] S. Raymon y Cajal: “Histologie du systeme nerveux de l'homme et des vertebrates”. *Paris 1911.*

[2] W. S. McCulloch, W. Pitts: “A logical calculus of the ideas immanent in nervous activity”, *Bulletin of Mathematical Biophysics, 5, 1943, p. 115-133.*

[3] D. O. Hebb: “The Organization of Behavior”, *Wiley, New York, 1949.*

[4] D. E. Rumelhart, G. E. Hinton, R. J. Williams: “Learning internal representations by error propagation”, *Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Vol. 1, Bradford Books, Cambridge, MA, 1986 p. 318-362.*

[5] I. Guyon, I. Poujaud, L. Personnaz, G. Dreyfus, J. Denker, Y. Le Cun: “Comparing different neural network architectures for classifying hand-written digits”, *Inernational Joint Conference on Neural Networks, Vol. 2, Washington DC, 1989, p. 127-132.*

[6] I. Guyon: “Applications of neural networks to character recognition”, *International Journal of Pattern Recognition and Artificial Intelligence 5, 1991, p 353-382.*

[7] D. Aberdeen, J. Baxter: “Emmerald: a fast matrix-matrix multiply using Intel's SSE instructions”, *Concurrency and Computation: Practice and Experience 13, 2001, p. 103-119.*

[8] E. S. Larsen, D. McAllister: “Fast Matrix Multiplies using Graphics Hardware”, *The International Conference for High Performance Computing and Communications, 2001.*

[9] C. Trendall and A. J. Stewart: "General calculations using graphics hardware with applications to interactive caustics", *Rendering Techniques 2000: 11th Eurographics Workshop on Rendering, p. 287-298, June 2000.*

[10] Wei Li, Xiaoming Wei, A. Kaufman: “Implementing Lattice Boltzmann Computation on Graphics Hardware”, *The International Conference for High Performance Computing and Communications, 2001.*

[11] R. R. Whaley, J. J. Dongarra: “Automatically tuned linear algebra software”, *Technical Report, Computer Science Department, University of Tennessee, 1997.*

[12] J. L. Hennessy, D. A. Patterson: “Computer Architecture – A Quantitative Approach”, *Morgan Kaufmann Inc., San Francisco, CA, 1995.*

[13] C. A. Bohn: “Kohonen Feature Mapping through Graphics Hardware”, *3rd Int. Conf. on Computational Intelligence and Neurosciences, 1998.*

[14] J. D. Foley, A. van Dam, S. K. Feiner, J. F. Hughes: “Computer Graphics, Principles and Practice”, *Addison-Wesley, 2nd edition, 1996.*

[15] M. Woo, J. Neider, T. Davis, O. A. R. Board: “OpenGL Programming Guide”, *Addison-Wesley, 2nd edition, 1997.*