GPU를 이용한 신경망 구현

오 경 수[†] · 정 기 철[†]

요 약

본 논문은 일반적인 그래픽스 하드웨어를 이용하여 더욱 빠른 신경망을 구현하고, 구현된 시스템을 영상 처리 분야에 적용함으로써 효용성을 검증한다. GPU의 병렬성을 효율적으로 사용하기 위하여, 다수의 입력벡터와 연결가중치벡터를 모아서 많은 내적연산을 하나의 행렬곱 연산으로 대체하였고, 시그모이드와 바이어스 항 덧셈 연산도 GPU 상에서 픽셀세이더로 구현하였다. ATI RADEON 9800 XT 보드를 이용하여 구현된 신경망 시스템은 CPU를 사용한 기존의 시스템과 비교하여 정확도의 차이 없이 30배 정도의 속도 향상을 얻을 수 있었다.

Implementation of Neural Networks using GPU

Kyoung-su Oh[†] · Keechul Jung[†]

ABSTRACT

We present a new use of common graphics hardware to perform a faster artificial neural network. And we examine the use of GPU enhances the time performance of the image processing system using neural network. In the case of parallel computation of multiple input sets, the vector-matrix products become matrix-matrix multiplications. As a result, we can fully utilize the parallelism of GPU. Sigmoid operation and bias term addition are also implemented using pixel shader on GPU. Our preliminary result shows a performance enhancement of about thirty times faster using ATI RADEON 9800 XT board.

키워드 : 그래픽처리 장치(Graphics Processing Unit : GPU), 신경망(Neural Network : NN), 다층 퍼셉트론(Multi-layer Perceptron : MLP), 문자 검출(Text Detection), 영상처리(Image Processing)

1. 서 론

기존의 그래픽스 하드웨어 활용과 관련된 연구는 CPU에서 GPU(Graphics Processing Unit)로 전달되는 기하학적 자료의 양을 줄여서 GPU의 부담을 줄이는데 집중되었다[1]. 최근에는 컴퓨터 그래픽스에서만 사용되던 하드웨어가 속도, 프로그래밍 가능성, 가격 등의 면에서 경쟁력을 가져감에 따라, 계산기하학(computational geometry)이나 과학계산 등 주로 CPU를 사용해서 구현하던 분야에서도 많은 알고리즘을 GPU로 구현하고 있다[2-9].

행렬 연산은 여러 값들에 대해서 같은 연산들이 수행하는 경우(Single-Instruction Multiple-Data)가 많기 때문에 병렬 구조인 GPU를 사용하는 것이 적합하다. Larson과 McAllister는 멀티 텍스처를 이용해서 행렬 곱셈을 하는 방법을 제안하였고[2], Hall 등은 픽셀세이더를 이용하여 행렬 곱 연산을 보다 효율적으로 개선했으며[3], Moravanszky는 행렬을 텍스처로 표현하는 구체적인 자료구조와 구현 방법을 제시하였다[4]. Kruger와 Westermann은 행렬을 2차원 텍스처를 사용하는 대각선 벡터들로 표현하고 이를 이용해서 여러 행렬 연산을 수행하는 방법을 제안하였다[5].

또한 최근에는 영상 처리 분야에서도 몇몇 연구 결과가 발표되고 있다. Yang과 Welch는 상용 GPU를 이용하여 영상 분할과 모폴로지 연산을 수행하였다[6]. 이는 레지스터 결합(register combiner)과 혼합(blending) 기술을 이용하여 GPU 내에서 컴퓨터비젼 분야의 기본적인 연산인 영상 분할과 모폴로지 연산을 구현하였다.

본 논문에서는 일반적인 그래픽스 하드웨어를 이용하여 더욱 빠른 신경망을 구현하는 방법을 제안한다. 신경망의 학습 단계와 테스트 단계 중, 테스트 단계에서는 실시간 처리가 절실하다. 그러나 입력 데이터가 많아지는 경우에는 실시간 처리가 힘들다. 이러한 이유로 기존에는 신경망을 이용한 실시간 처리를 위해서는 전용 병렬 FPGA 등의 하드웨어를 이용하며 구현했는데, 이를 구현하기 위해서는 비용이나 설계 시의 추가 노력을 동반한다[9]. 비록 그래픽스 하드웨어가 신경망 구현을 위한 전용 하드웨어는 아니더라도 이미 일반인에게 많이 보급되어 있는 장비이며, 가격이 저렴하고, 구현할 때 오버헤드가 적어서 대용량 패턴인식이나 영상처리 등의 문제에 적용될 때 많은 장점이 있다.

※ 본 연구는 숭실대학교 교내 연구비 지원으로 이루어졌음.
† 종신회원 : 숭실대학교 미디어학부 교수
논문접수 : 2004년 1월 30일, 심사완료 : 2004년 7월 30일

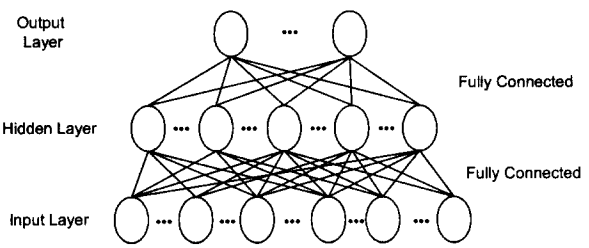
신경망의 기본적인 연산은 한 노드(node)에서의 연결가중치벡터(weight vector)와 입력벡터간의 내적(inner-product) 연산이다. 이러한 신경망의 기본 연산을 GPU 상에서 구현하기 위해서, 많은 입력벡터와 연결가중치벡터를 누적함으로써 다수의 내적 연산을 하나의 행렬곱 연산으로 대체해서 GPU의 병렬성의 효과를 극대화할 수 있다. 이와 같이 행렬곱과 활성함수(activation function) 등의 신경망 구현에 필수적인 연산을 GPU 상에서 정점세이더(vectex shader)와 픽셀세이더(pixel shader)를 이용하여 효과적으로 구현할 수 있다.

영상처리나 패턴 인식은 신경망 응용 분야들 가운데, 입력 데이터의 양이 많은 분야 중 하나이다. 예를들어, 신경망을 이용한 컨벌루션(convolution)을 영상 전체 영역에 수행할 때는 상당한 수행 시간을 필요로 한다. 본 논문에서는 GPU를 이용하여 신경망을 구현함으로써 영상 처리 등의 분야와 같은 대용량 데이터의 신경망 처리 문제를 저가로, 그리고 적은 노력으로 해결할 수 있었다

본 논문의 구성은 다음과 같다. 2장에서는 GPU를 이용하여 구현할 신경망의 구조와 관련 연구에 대해 기술하고, 3장에서는 GPU 기반 신경망 구현을 위한 제안된 방법을 구체적으로 설명하고, 4장에서는 실험 결과를, 5장에서는 결론과 향후 연구 과제로 마무리 한다.

2. 관련 연구

인공 신경망은 간단한 기능을 갖는 다수의 처리기들이 인접한 처리기들과 정보를 주고 받으면서 동작하는 형태로 이루어지므로 병렬 하드웨어로 구현하기에 적합하다. 인공신경망의 대표적인 학습 알고리즘으로는 multilayer perceptron, learning vector quantization, radial basis function, hopfield, kohonen 등을 주로 많이 사용한다[10].



(그림 1) 전형적인 MLP의 구조

본 논문에서 구현할 신경망은 (그림 1)과 같은 다층 퍼셉트론(multilayer perceptron : MLP)이다. MLP는 1개 이상의 은닉층(hidden layer)을 가지며 다수의 출력 노드를 가질 수 있다. 일반적으로 MLP는 인접한 층(layer)의 노드들이 완전연결(fully-connected)되어 있고, 층의 개수나 각 층의 노드수 등에서 변화가 있을 수 있지만, 기본적으로 각 노드는 식 (1)과 같이 연결가중치벡터와 해당 노드의 입력벡터의 내적연산을 수행한 후, 식 (2)와 같은 활성함수 연산을 수행한다.

$$m_j = \sum w_{ji} x_i + b_j \tag{1}$$

$$r_j = \left(1 + e^{-m_j}\right)^{-1} \tag{2}$$

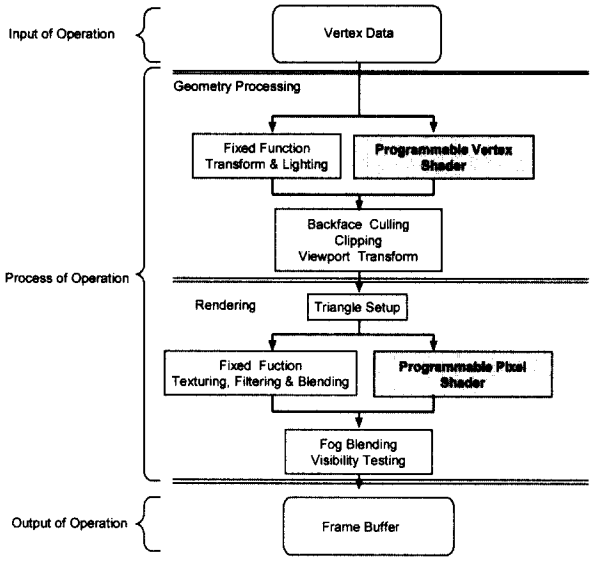
식 (1)과 식 (2)에서 첨자 j는 출력 노드를 의미하며, i는 j-번째 노드와 연결되어 있는 하위층의 노드이다. w_{ji}는 j-번째 노드와 i-번째 노드를 연결하는 연결가중치, x_i는 입력값, b_j는 j-번째 노드의 바이어스 항(bias term), r_j는 j-번째 노드의 최종 출력값을 의미한다. MLP의 첫 번째 은닉층의 노드들부터 이와 같은 연산을 수행하여 차례로 출력층까지 계산한다. MLP 이외의 다른 종류의 신경망들도 기본적으로 각 노드는 MLP의 노드들과 유사한 연산을 수행하기 때문에, 본 논문의 연구 결과는 쉽게 다른 신경망에 적용될 수 있다.

신경망의 학습 단계와 테스트 단계 중에서, 테스트 단계에서는 실시간 처리가 절실한데, 입력 데이터가 많아지는 경우에는 실시간 처리가 힘들다. 이러한 이유로 기존에는 신경망을 이용한 실시간 처리를 위해서, 다음과 같이 전용 병렬 FPGA 등의 하드웨어를 이용하며 구현하려는 시도가 있었다.

Antonio d'Acierno는 노드 연결(synapse) 병렬화, 뉴론 병렬화, 학습 데이터 병렬화, 이러한 3가지의 병렬화 방법을 기술하고, 그에 따라 SIMD, MIMD 시스템 상에서 신경망의 학습 알고리즘 중의 하나인 역전파 알고리즘을 병렬 알고리즘으로 구현하고 이를 분석하였다[18]. FPGA를 이용한 다른 연구 결과로써, Nazeih M. Botros와 M. Abdul-Aziz는 FPGA를 이용하여 3층 퍼셉트론의 테스트 단계를 구현하였다[19]. Economou 등은 인공신경망을 FPGA로 구현하고 이를 의료 전문가 시스템에 적용하였다[20]. 이와 같이 그동안 FPGA를 이용한 많은 연구가 진행되었으며, FPGA를 이용함으로써 시스템의 구조를 쉽게 변경(reconfigurable)할 수 있다는 장점이 있으나, 프로세서를 직접 디자인, 구현, 장착해야하는 어려움이 있다.

3. GPU를 이용한 MLP 구현

그래픽스 하드웨어는 여러 해 동안 그래픽스의 렌더링만을 위해 사용되었으나, 차츰 성능이 발전함에 따라 그래픽스 이외의 분야에 사용될 수 있을 정도의 복잡한 연산을 지원하게 발전하였다. 또한 프로그래밍 가능한 정점세이더와 픽셀세이더의 출현으로 더욱 융통성있는 일반적인 연산을 지원할 수 있게 되었다. GPU는 반복 연산이 많은 분야인 렌더링을 위해서 개발되었기 때문에, 반복된 연산을 많이 가지는 분야에서는 CPU보다 성능을 향상시킬 수 있다.



(그림 2) 렌더링 파이프라인

할 수 있다. 그리고 바이어스 항의 덧셈과 시그모이드 연산들은 픽셀세이더로 쉽게 구현가능하다. 이로써 신경망의 각 '층'의 연산은 식 (3)~식 (5)와 같이 기술할 수 있다.

여기서 w_{ij}는 출력층의 i-번째 노드와 입력층의 j-번째 노드사이의 연결가중치, M은 출력층 노드의 수, N은 입력층 노드의 수, x_{ij}는 j-번째 입력벡터의 i-번째 특징값, b_i는 전체 L 개의 입력벡터 중 i-번째 입력 노드를 위한 바이어스 항(term)을 의미한다. 최종 연산 결과인 R_{ij}는 j-번째 입력 벡터에 대한 i-번째 출력 노드의 값이다. 위의 연산은 행렬연산, 바이어스 항 덧셈연산, 활성함수인 시그모이드 연산의 순으로 수행된다.

행렬의 곱을 위해서 Moravanszky[4]에 의해 제안된 방법을 사용한다. (그림 3)은 GPU 내에서의 행렬 곱 연산의 개괄도이다. 두개의 행렬을 *texture W*와 *texture X*로 변환하고 렌더링 연산을 수행한다. 행렬 곱의 출력을 위해 전체 화면을 덮는 사각형을 렌더링한다. 정점세이더는 사각형의 각

$$W=\begin{bmatrix} W_{11} & W_{12} & W_{13} & \cdots & W_{1N} \\ W_{21} & W_{22} & W_{23} & \cdots & W_{2N} \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ W_{M1} & W_{M2} & W_{M3} & \cdots & W_{MN} \end{bmatrix}=\begin{bmatrix} W_1 \\ W_2 \\ \cdots \\ W_M \end{bmatrix},\quad X=\begin{bmatrix} x_{11} & x_{12} & x_{13} & \cdots & x_{1L} \\ x_{21} & x_{22} & x_{23} & \cdots & x_{2L} \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ x_{N1} & x_{N2} & x_{N3} & \cdots & x_{NL} \end{bmatrix}=[X_1 X_2 X_3 \cdots X_L],\quad B=\begin{bmatrix} b_1 & b_1 & b_1 & \cdots & b_1 \\ b_2 & b_2 & b_2 & \cdots & b_2 \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ b_M & b_M & b_M & \cdots & b_M \end{bmatrix} \tag{3}$$

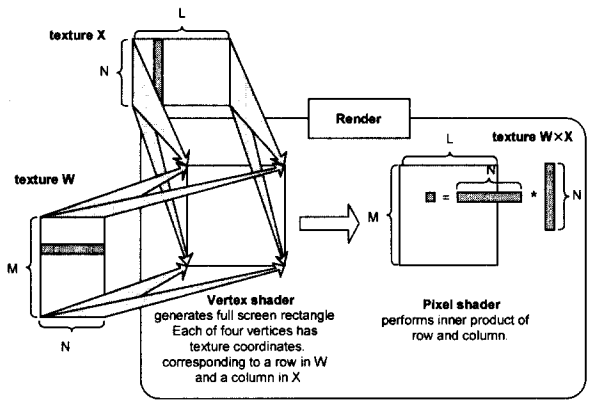
$$W=W\times X+B=\begin{bmatrix} W_1\cdot X_1 & W_1\cdot X_2 & W_1\cdot X_3 & \cdots & W_1\cdot X_N \\ W_2\cdot X_1 & W_2\cdot X_2 & W_2\cdot X_3 & \cdots & W_2\cdot X_N \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ W_M\cdot X_1 & W_M\cdot X_2 & W_M\cdot X_3 & \cdots & W_M\cdot X_N \end{bmatrix}+\begin{bmatrix} b_1 & b_1 & b_1 & \cdots & b_1 \\ b_2 & b_2 & b_2 & \cdots & b_2 \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ b_M & b_M & b_M & \cdots & b_M \end{bmatrix}=\begin{bmatrix} m_{11} & m_{12} & m_{13} & \cdots & m_{1N} \\ m_{21} & m_{22} & m_{23} & \cdots & m_{2N} \\ m_{31} & m_{32} & m_{33} & \cdots & m_{3N} \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ m_{M1} & m_{M2} & m_{M3} & \cdots & m_{MN} \end{bmatrix} \tag{4}$$

$$R=sigmoid(M)=\begin{bmatrix} (1+e^{-m_{11}})^{-1} & (1+e^{-m_{12}})^{-1} & (1+e^{-m_{13}})^{-1} & \cdots & (1+e^{-m_{1L}})^{-1} \\ (1+e^{-m_{21}})^{-1} & (1+e^{-m_{22}})^{-1} & (1+e^{-m_{23}})^{-1} & \cdots & (1+e^{-m_{2L}})^{-1} \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ (1+e^{-m_{M1}})^{-1} & (1+e^{-m_{M2}})^{-1} & (1+e^{-m_{M3}})^{-1} & \cdots & (1+e^{-m_{ML}})^{-1} \end{bmatrix} \tag{5}$$

GPU를 이용해서 일반적인 연산을 하는 과정은 (그림 2)와 같다. 연산의 입력값을 텍스처나 정점 값의 형태로 GPU에게 전달한다. 이후 여러 번의 렌더링 과정 동안 GPU는 정점세이더와 픽셀세이더를 수행하여 원하는 연산을 수행한다. 정점세이더는 모든 정점마다 수행되며 각 픽셀의 위치, 색상, 텍스처 좌표값을 계산하고, 픽셀세이더는 다각형 모델에 의해 덮히는 모든 픽셀에서 수행되며 각 픽셀의 색상을 출력한다.

이미 기술한 바와 같이, 신경망의 각 '노드'에서의 내적 연산은 입력벡터와 연결가중치벡터를 축적함으로써 행렬의 곱 연산으로 변환할 수 있다. 이렇게 분리되어 수행되는 노드들에서의 내적 연산들을 하나의 행렬곱으로 해석함으로써 GPU의 병렬구조를 효율적으로 활용하여 신경망을 구현

정점의 위치 외에 텍스처 좌표를 출력하는데, 각각의 정점은 두개의 텍스처 좌표를 가지고 있다. 그 중 하나는 *texture W*의 행(row) 좌표이고, 또 다른 하나는 *texture X*의 열(column) 좌표이다. 예를 들어, 왼쪽 상단의 좌표는 *texture W*의 첫 번째 행의 텍스처 좌표와 *texture X*의 첫 번째 열 텍스처 좌표를 가지며, 오른쪽 상단의 좌표는 *texture W*의 첫 번째 행의 텍스처 좌표와 *texture X*의 마지막 열의 텍스처 좌표를 가지는 식이다. 정점세이더의 결과로써, 각 픽셀(i, j)들은 W의 i-번째 행과 X의 j-번째 열에 해당하는 텍스처 좌표를 가진다. 픽셀세이더는 텍스처 좌표에 의한 *texture W*의 행과 *texture X*의 열 사이의 내적 연산을 수행한다. 행렬곱의 결과는 *texture W×X*에 저장된다.



(그림 3) GPU를 이용한 행렬 곱 연산의 개괄도

(알고리즘 1)은 행렬 곱셈을 위한 픽셀세이더의 의사코드이다. 이 의사코드에서 *i, j*는 현재 픽셀 위치를 나타내고, *X*와 *W*는 두 입력 행렬을 담고 있는 텍스처이다. *F*는 곱셈의 결과를 저장하는 텍스처를 나타낸다. 텍스처의 픽셀들에는 행렬의 세로로 연속인 네 개의 원소를 RGBA 각 성분에 저장한다. *texture W*로부터 네 개의 연속된 픽셀을 읽고 *texture X*로부터 한 픽셀을 읽어서 이들의 내적을 계산하는 것을 반복한다. *texture X*의 너비와 *texture W*의 높이가 *n*일 경우 이 과정을 *n*/4번 반복해서 네 개의 행과 하나의 열의 내적연산을 수행한다. 이 과정이 한번의 픽셀세이더 연산으로 수행하는 것이 불가능하면 여러 번의 렌더링을 거쳐서 수행한다. 행렬 곱을 위한 렌더링 단계의 횟수는 GPU의 성능에 따라 달라지며 이는 픽셀세이더 연산의 수, 텍스처 로드 연산의 수 등에 따라 달라진다.[1)]

```
/* texture X의 i행과 texture W의 j열을 내적 */
/* texel하나에는 행렬의 4개의 원소를 저장한다. 이들은 열이 같고 행이 연속이다. */
/* texture X의 네개의 원소(행렬의 4×4 부분에 해당)와 texture W의 하나의 원소(행렬의 1×4 부분에 해당)와의 곱셈을 한꺼번에 한다. */

for k = 1 … n/4 step by 4
    F(i, j).r = F(i, j).r + X(i, k×4).r×W(k, j).r + X(i, k×4+1).r × W(k, j).g +
                X(i, k×4+2).r×W(k, j).b + X(i, k×4+3).r×W(k, j).a ;
    F(i, j).r = F(i, j).g + X(i, k×4).g×W(k, j).r + X(i, k×4+1).g×W(k, j).g+
                X(i, k×4+2).g×W(k, j).b + X(i, k×4+3).g×W(k, j).a ;
    F(i, j).r = F(i, j).b + X(i, k×4).b×W(k, j).r + X(i, k×4+1).b×W(k, j).g+
                X(i, k×4+2).b×W(k, j).b + X(i, k×4+3).b×W(k, j).a ;
    F(i, j).r = F(i, j).a + X(i, k×4).a×W(k, j).r + X(i, k×4+1).a×W(k, j).g+
                X(i, k×4+2).a×W(k, j).b + X(i, k×4+3).a×W(k, j).a ;
End
```

(알고리즘 1) 행렬 곱을 위한 픽셀세이더 연산

행렬 곱 연산 후, 바이어스 항의 덧셈과 시그모이드 연산은 픽셀세이더를 이용하여 한번의 렌더링 연산으로 수행한다. 렌더링을 시작하기 전에 바이어스 텍스처와 이전 단계에서의 행렬곱의 결과로 생성된 *texture W×X*를 활성화한다. 정점세이더는 다시 한번 전체 스크린을 사각형 모델로 출력한다. *texture W×X*를 위한 각 정점의 텍스처 좌표는 각 위치에 해당한다. 예를 들어, 왼쪽 상단의 정점은 텍스처 좌표 (0, 0)을 가지며, 오른쪽 상단의 정점의 좌표는 (1, 0)이 되는 식이다. 한 행을 위한 바이어스 항의 값은 모두 동일하기 때문에, 바이어스 항을 위한 행렬은 1차원이다. 바이어스 텍스처를 위한 각 정점의 텍스처 좌표는 각각의 정점의 세로 위치이다. 픽셀세이더는 두 텍스처 값을 더하고 시그모이드 연산을 수행한다. (알고리즘 2)는 바이어스 항 덧셈과 시그모이드 연산을 한꺼번에 수행하기 위한 픽셀세이더의 의사코드이다. (알고리즘 1)에서와 같이 *i, j*는 현재 픽셀 위치를 나타내고, *F*는 이전 곱셈의 결과가 저장된 텍스처이고 *Bias*는 바이어스 텍스처를 나타낸다.

$$F(i,j).r = \frac{1}{1+e^{-(F(i,j).r+Bias(i).r)}}$$

$$F(i,j).g = \frac{1}{1+e^{-(F(i,j).g+Bias(i).g)}}$$

$$F(i,j).b = \frac{1}{1+e^{-(F(i,j).b+Bias(i).b)}}$$

$$F(i,j).a = \frac{1}{1+e^{-(F(i,j).a+Bias(i).a)}}$$

(알고리즘 2) 바이어스 항 덧셈과 시그모이드 함수를 위한 픽셀세이더 연산

신경망에 둘 이상의 은닉층이 존재한다면, 위의 연산 과정을 각 층 마다 반복 수행한다. 이전 층의 결과는 렌더링도 가능하고 텍스처로도 사용할 수 있는 렌더 타겟 텍스처(render target texture) 형태로 저장되며 이는 다음 층의 입력으로 사용된다. 신경망이 다수개의 은닉층을 지니더라도 텍스처 생성 후에는 위의 모든 과정이 CPU의 개입 없이 GPU만으로 수행할 수 있다.

4. 실험 결과

본 장에서는 제안한 GPU를 이용한 신경망을 '영상 내의 문자 추출' 분야에 적용하고 유용성을 검증한다.

영상 내의 문자 추출은 입력 영상 내의 문자 영역을 자동으로 추출하는 것을 의미한다. 영상에 내재된 문자를 추출하고 인식하는 연구는 인덱싱에 유용한 고급 정보를 자동으로 얻을 수 있다는 장점으로 인하여, 멀티미디어 시스템, 전자 도서관, 비디오 인덱싱, 문서 구조 분석, 우편 영상 내의 주소 영역 추출, 자동차 번호판 추출 등 다양한 관련 분야에서 진행되고 있으나, 양질의 문서의 자동 문자 인식(OCR)과는 달리 상당히 어려운 문제로 인식되고 있다[11-13].

기존의 문자 추출 방법은 크게 연결 성분 방법(connect-

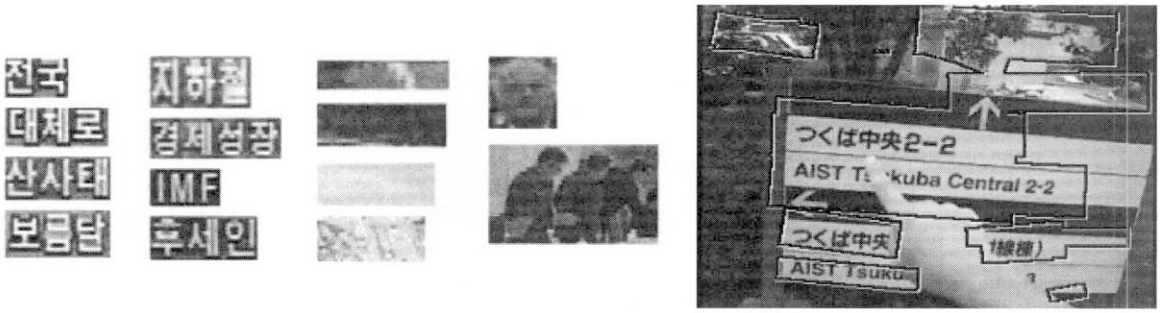
1) 본 실험에서 사용된 ATI RADEON 9800 XT는 PIXEL SHADER v2.0을 지원한다. PIXEL SHADER v2.0에서는 최대 32번의 텍스처 로드 연산을 허용한다. 즉, (알고리즘 1) 루프 내의 연산을 (= ⌈(32 − 1)/5⌉)번을 한번에 처리할 수 있다.

ed component method : CCM)과 질감(texture) 기반 방법으로 나눌 수 있다. 연결 성분 방법은 색상, 질감 등의 정보를 이용하여 입력 영상을 분할한 후, 각 연결 성분들을 특정 조건들을 이용하여 문자 영역과 비-문자 영역으로 나누는 방법이다. CCM과 달리 질감 기반 방법은 문자 영역의 질감 성질을 이용하기 위해 Gabor filter, wavelet, spatial variance 등의 질감 분석기(texture analyzer)를 사용하는 방법이다[14].

본 연구에서는 입력 영상 내의 문자의 유무를 판별하기 위한 질감 분석기로써 신경망을 사용하였다[15-17]. 신경망을 사용함으로써 '질감 분석기의 구성'이라는 문제가 '신경망의 학습' 문제로 변환된다. (그림 4)는 신경망의 학습에 사용되는 학습 샘플의 일부이며, 신경망의 학습 단계에서는 문자 클래스와 비-문자 클래스의 명확한 경계 구분을 위해 부트스트랩(bootstrap) 방법을 사용한다. 이는 초기에 정해진 비-문자 클래스 데이터를 이용해서 신경망을 학습 시킨 후에, 부분적으로(partially) 학습된 신경망을 테스트 영상에 테스트하여 오인식 된 데이터로 신경망을 재 학습시키는 방법이다. (그림 4)는 이의 예를 보이는 것으로써, (a)는 문자 클래스의 학습 데이터 예, (b)는 비-문자 클래스의 학습 데이터 예, (c)는 문자 추출 결과의 예이다. 비-문자 부분을 문자 영역이라고 오-추출한 부분의 샘플 데이터를 이용하여 신경망을 재 학습하게 된다.

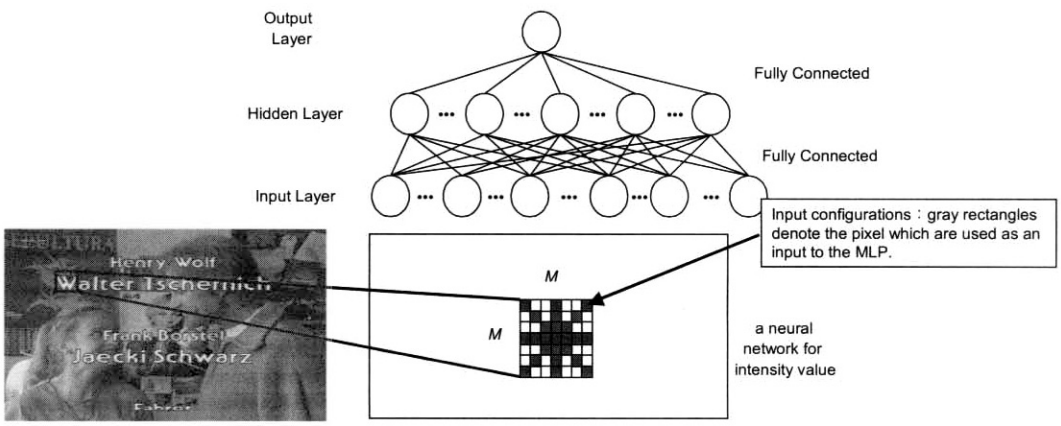
본 논문에서 사용하는 MLP의 학습 구조는 1개의 은닉층에 30개의 은닉층 노드, 1개의 출력 노드로 구성되며, 인접 층의 노드들은 모두 연결되어있고, 입력층은 (그림 5)에 표시되어 있는 바와 같이, 입력 영상에서 M×M 크기의 입력 창 내의 회색으로 표시된 화소들의 밝기값(intensity)을 사용하였다. 입력윈도우 내의 모든 픽셀을 사용하는 대신에 이러한 입력 구조를 사용하는 이유는, 이러한 구조가 질감 분석 분야에서 성능과 속도 향상에 좋은 것으로 알려져 있기때문이다[15]. MLP를 이용하여 입력 영상을 처리한 결과는 [0..1] 사이의 값을 지닌다.

본 연구에서는 신경망을 사용함으로써 다양한 환경에 적응성을 지니는 문자 추출기를 자동으로 생성할 수 있었다. 그러나 소프트웨어로 구현한 신경망은 앞에서 기술한 바와 같이, 테스트 단계에서의 많은 계산량으로 인한 느린 수행속도의 문제점이 있다. 특히 질감 기반의 컨벌루션(convolution) 필터링 기법은 문자 추출을 위해서 전체 입력 영상을 모두 스캔(scan) 해야 한다. 특정 필터로 전 영상을 컨벌루션하는 것은 상당한 수행시간을 필요로 하는 작업인데, 본 연구는 GPU를 이용한 신경망을 사용함으로써 속도 향상을 얻는다. GPU의 성능을 최대한 높이기 위해서 공간적으로 연속된 화



(a) (b) (c)

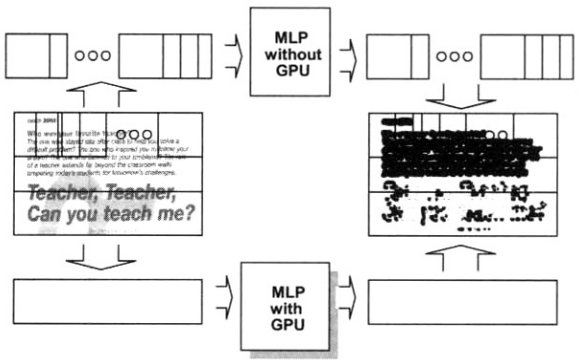
(그림 4) 신경망의 학습 데이터 (a) 문자 데이터, (b) 비-문자 데이터, (c) 문자 추출 예



(그림 5) 질감 분석기 MLP의 구조

소들에 대한 신경망의 입력벡터를 누적함으로써 신경망의 입력벡터 값들을 2차원 텍스처로 구성한다. 이렇게 함으로써 신경망의 연산을 GPU의 렌더링 연산으로 구현하기에 적합하게 구성하였고, 결과적으로 GPU의 병렬성을 이용함으로써 신경망의 수행 속도를 현저하게 향상시킬 수 있었다.

(그림 6)은 GPU 기반 신경망을 이용한 문자 영역 추출 방법을 나타내었다. 왼쪽의 입력영상에 해당하는 문자 추출 결과가 오른쪽의 영상이며, 검은 화소가 문자 영역을 의미한다. GPU를 이용함으로써 여러 번의 문자추출 작업을 한 번으로 병합하여 수행할 수 있다.



(그림 6) GPU를 이용한 신경망 처리

캡춰된 비디오 영상, 스캔 영상 등 다양한 실험 데이터를 이용하여 제안된 방법을 테스트하였다. 본 논문에서 실험한 데이터는 참고 논문 16에서 사용한 데이터를 사용함으로써, 이전의 연구와 비교 분석하였다. 본 실험에서는 펜티엄Ⅳ, 메모리 512M Bytes, ATI RADEON 9800 XT가 장착된 컴퓨터를 사용하였다. (그림 7)은 1152×1546 크기의 컬러 스캔 문서에 대한 실험 결과 예이다. 컬러영상을 그레이영상으로 변환한 후, 신경망을 이용하여 문자 영역을 검출하였다. 최종 결과인 (그림 7)(b)는 CPU만을 이용한 결과와 거의 일치하였으며 약 30배 정도 속도가 향상되었다.

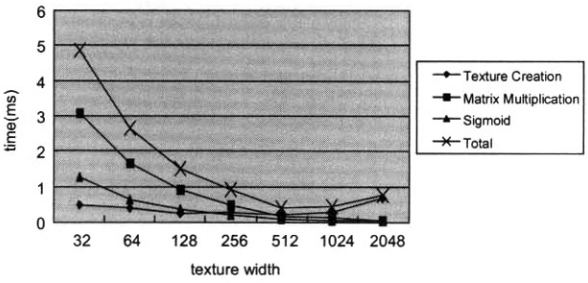
(그림 7) 실험 결과

<표 1>에서 GPU 기반 신경망과 CPU만을 사용한 신경망의 최소 수행시간을 비교하였다. GPU로 구현하였을 경우 약 30배 정도의 속도 향상을 얻을 수 있었다. GPU를 이용하였을 경우, 전체 수행시간의 1/2 정도를 텍스처 생성(Texture Creation) 단계에서 사용하였는데, 이는 GPU를 이용하면서 생기는 오버헤드이다.

〈표 1〉 각 계산 단위 별 수행 시간

	Texture Creation	Matrix Multiplication	Sigmoid	Total
GPU	0.18	0.14	0.06	0.38
CPU	11.743			

(그림 8)은 제안된 방법에서 텍스처의 너비를 바꿔가면서 (그림 7)의 입력 영상에 적용했을 때의 시간측정 결과이다. 여기서 텍스처의 너비는 3장에서 설명한 *texture X*의 너비이다. *texture X*는 GPU가 한번에 처리하는 입력의 개수를 결정하게 되는데 *texture X*의 너비가 작으면 입력들을 작게 나누어서 여러 번 처리하게 되고, 크면 입력들을 모아서 한꺼번에 처리하게 된다. 전체 계산 시간은 텍스처의 너비가 512일 경우 가장 작았다. 텍스처의 넓이가 512보다 작을 경우에는 행렬 곱과 시그모이드에 걸리는 시간이 많고 텍스처가 512보다 클 경우에는 텍스처 생성에 걸리는 시간이 상대적으로 많다. 행렬 곱과 시그모이드에 걸리는 시간은 텍스처의 넓이가 커질수록 작아지는데 텍스처의 크기가 커질수록 렌더링의 횟수가 작아지므로 렌더링을 시작하는데 걸리는 일정한 준비시간이 작아지기 때문이다.

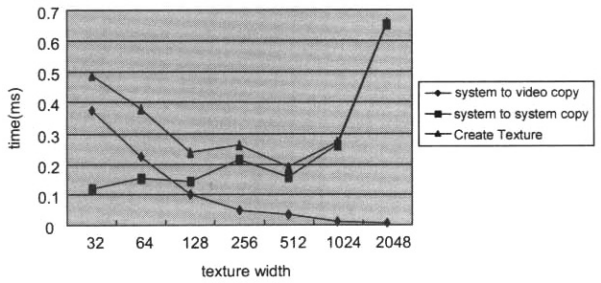


(그림 8) 텍스처 너비 변화에 따른 수행시간

텍스처 생성은 두 가지 연산으로 이루어져 있다. 행렬의 내용을 담고 있는 배열로부터 시스템 메모리에 할당된 텍스처를 생성하는 연산과, 시스템 메모리의 텍스처로부터 비디오 메모리의 텍스처로의 복사연산이 그것이다. 시스템 메모리 텍스처를 생성할 때 배열과 텍스처에서 행렬을 저장하는 형식이 서로 다르기 때문에 원소들을 재배열 한다.

(그림 9)는 텍스처 너비에 따른 텍스처 생성 시간의 변화를 분석한 그래프이다. 배열로부터 시스템 메모리의 텍스처로 복사하는데 걸리는 시간은 텍스처 너비가 512보다 작을 때는 별다는 패턴을 보이지 않고 512보다 커지면 급격히 증가한다. 실험에 사용한 텍스처는 한 픽셀당 16바이트(sizeof

(float)×4)를 사용하는데 텍스처의 너비가 512일 경우 텍스처에 한 줄에 필요한 메모리가 8킬로 바이트가 된다. 이는 Pentium Ⅳ의 L1 캐쉬(cache)의 크기와 일치한다. 그러므로 텍스처의 너비가 512이상일 경우 system to system copy 시간이 증가하는 것은 텍스처의 넓이가 캐시의 크기보다 크기 때문이라고 판단된다. 시스템 메모리에서 비디오메모리로의 복사연산에 걸리는 시간은 텍스처 넓이가 커짐에 따라 감소를 보인다. 이는 앞에서 렌더링연산이 적어짐에 따라 행렬 곱과 시그모이드 연산이 빨라지는 것과 같이 복사연산의 개수가 작아져서 복사를 위한 준비시간이 작아지기 때문이다.



(그림 9) 텍스처 너비 변화에 따른 텍스처 생성 시간

GPU에 의해서 행해지는 연산인 행렬 곱과 시그모이드 연산, 시스템 메모리에서 비디오 메모리로의 복사는 모두 텍스처 너비가 커질수록 빨라졌고, CPU만 관여하는 시스템 메모리간의 복사연산은 텍스처의 너비가 512보다 작을 때는 변화가 거의 없다가 그 이상이 될 경우에는 커졌다. GPU 관련 연산은 텍스처 너비를 크게 하는 것이 유리하고 CPU 관련 연산은 텍스처 너비가 작은 것이 유리하므로 GPU나 CPU 관련 연산의 복잡도와 시스템 사양에 따라 적절한 텍스처 너비를 선택해야 한다.

5. 결 론

본 논문에서는 일반적인 그래픽스 하드웨어를 이용하여 신경망을 구현하였으며 이를 영상처리 분야에 적용하여 실효성을 검증하였다. GPU를 이용한 신경망 구현은 가격이 저렴하고, 소프트웨어적으로 구현할 때 오버헤드가 적은 등 많은 잇점이 있다. 많은 입력벡터와 연결가중치벡터를 누적함으로써 다수의 내적 연산을 하나의 행렬곱 연산으로 대체해서 GPU의 병렬성의 효과를 극대화할 수 있었다. 또한 여러 입력벡터들에 대한 바이어스항과의 합과 시그모이드 연산도 한꺼번에 처리하여 GPU의 병렬 구조를 활용하였다.

이와 같은 연구결과는 영상 처리, 패턴인식 등의 분야와 같은 대용량 데이터의 신경망 처리 문제를 저가로 그리고 적은 오버헤드로 해결하고자하는 분야에 활용될 수 있다.

향후 GPU를 이용한 신경망 학습과 CPU와 GPU의 병렬처리 극대화, 여러 컴퓨터의 CPU와 GPU를 사용한 신경망 구현등에 대한 연구를 할 예정이다. 또한 기존의 병렬 프로세서를 이용하여 구현한 신경망과의 성능 비교를 통해서 보다 객관적인 비교를 수행할 예정이다.

참 고 문 헌

[1] K. Oh, B. Shin and Y. G. Shin, "Mobility Culling-An Efficient Rendering Algorithm Using Temporal Coherence," The Journal of Visualization and Computer Animation, Vol.12, Issue 3, pp.159-166, 2001.

[2] E. S. Larsen and D. McAllister, "Fast Matrix Multiplies using Graphics Hardware," Proceedings of the 2001 ACM/IEEE Conference on Supercomputing, pp.55-55, 2001.

[3] J. D. Hall, N. A. Carr and J. C. Hart, "Cache and Bandwidth Aware Matrix Multiplication on the GPU," Technical Report UIUCDCS-R-2003-2328, University of Illinois Dept. of Computer Science, Mar., 2003.

[4] A. Moravanszky, 'Linear Algebra on the GPU,' in : W. F. Engel (Ed), Shader X2, Wordware Publishing, 2003.

[5] J. Kruger and R. Westermann, "Linear Algebra Operators for GPU Implementation of Numerical Algorithms," SIGGRAPH 2003, 2003.

[6] R. Yang and G. Welch. "Fast Image Segmentation and Smoothing Using Commodity Graphics Hardware," the Journal of Graphics Tools, Vol.7, No.4, pp.91-100, 2003.

[7] D. Manocha, "Interactive Geometric & Scientific Computations using Graphics Hardware," SIGGRAPH 2003 Tutorial Course #11, 2003.

[8] M. M. Trentacoste, "Implementing Performance Libraries on Graphics Hardware," Carnegie Mellon University Undergraduate Thesis, 2003.

[9] J. Zhu and P. Sutton, "FPGA Implementation of Neural Networks - a Survey of a Decade of Progress," Proceedings of 13th International Conference on Field Programmable Logic and Applications (FPL 2003), 2003.

[10] Haykin, Neural Networks, Prentice Hall, 1999.

[11] Biebelmann, E., Koppen, M. and Nickolay, B., "Practical Applications of Neural Networks in Texture Analysis," Neurocomputing, Vol.13, pp.261-279, 1996.

[12] H. Li, D. Doerman and O. Kia, "Automatic Text Detection and Tracking in Digital Video," IEEE Transactions on Image Processing, Vol.9, No.1, pp.147-156, 2000.

[13] Y. Zhong, H. Zhang and A. K. Jain, "Automatic Caption Localization in Compressed Video," IEEE Transactions on Pattern Analysis and Machine Intelligence, Vol.22, No.4, 2000.

[14] K. Jung, K. I. Kim and A. K. Jain, "Text Information Extraction in Images and Video : A Survey," International Journal of Pattern Recognition, Vol.37, Issue 5, pp.977-

997, May, 2004.

[15] K. Jung, "Neural network-based Text Location in Color Images," Pattern Recognition Letters, Vol.22, No.14, pp.1503-1515, 2001.

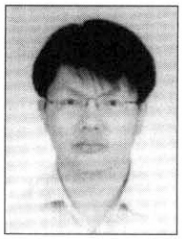
[16] K. Y. Jeong, K. Jung, E. Y. Kim and H. J. Kim, "Neural Network-based Text Location for News Video Indexing," Proceedings of International Conference of Image Processing, 1999.

[17] A. K. Jain and B. Yu, "Automatic Text Location in Images and Video Frames," Pattern Recognition, Vol.31, No.12, pp.2055-2076, 1998.

[18] Antonio d'Acierno, "Back-Propagation Learning Algorithm and Parallel Computers : The CLEPSYDRA Mapping Scheme," Neurocomputing, Vol.31, pp.67-85, 2000.

[19] Nazeih M. Botros and M. Abdul-Aziz, "Hardware Implementation of an Artificial Neural Network Using Field programmable Gate Array(FPGA's)" IEEE Transactions on Industrial Electronics, Vol.41, No.6, December, 1994.

[20] G. -P. K. Economou, E. P. Mariatos, N. M. Economopoulos, D. Lymberopoulos and C. E. Goutis, "FPGA Implementation of Artificial Neural Networks : An Application on Medical Expert Systems," 4th International Conference on Microelectronics for Neural Networks and Fuzzy Systems, Torino, Italy, pp.287-293, September, 1994.



오 경 수

e-mail : oks@ssu.ac.kr

1994년 서울대학교 계산통계학과(학사)

1996년 서울대학교 전산과학과(이학석사)

2001년 서울대학교 전기컴퓨터공학부 (공학박사)

2001년~2002년 조이멘트 연구원

2003년~현재 숭실대학교 정보과학대학 미디어학부 교수

관심분야 : 실시간 렌더링, 영상기반 렌더링, 컴퓨터 게임, GPU 프로그래밍

정 기 철

e-mail : kcjung@ssu.ac.kr

1996년 경북대학교 컴퓨터공학과 공학석사

2000년 경북대학교 컴퓨터공학과 공학박사

1999년 Machine Understanding Division, ElectroTechnical Laboratory, Japan, 방문연구원

2001년 PRIP Lab., Michigan State University, U.S. 박사후 연구원

2003년~현재 숭실대학교 정보과학대학 미디어학부 교수

관심분야 : Interactive Contents, HCI, 영상 처리, 패턴 인식, Augmented Reality, Mobile Vision System