CUDA와 OpenMP를 이용한 빠르고 효율적인 신경망 구현

(Fast and Efficient Implementation of Neural Networks using CUDA and OpenMP)

박 안 진 † 장 홍 훈 †† 정 기 철 †††
(Anjin Park) (Honghoon Jang) (Keechul Jung)

요 약 컴퓨터 비전이나 패턴 인식 분야에서 이용되고 있는 많은 알고리즘들이 최근 빠른 수행시간을 위해 GPU에서 구현되고 있지만, GPU를 이용하여 알고리즘을 구현할 경우 크게 두 가지 문제점을 고려해야 한다. 첫째, 컴퓨터 그래픽스 분야의 지식이 필요한 쉐이딩(shading) 언어를 알아야 한다. 둘째, GPU를 효율적으로 활용하기 위해 CPU와 GPU간의 데이터 교환을 최소화해야 한다. 이를 위해 CPU는 GPU에서 처리할 수 있는 최대 용량의 데이터를 생성하여 GPU에 전송해야 하기 때문에 CPU에서 많은 처리시간을 소모하며, 이로 인해 CPU와 GPU 사이에 많은 오버헤드가 발생한다. 본 논문에서는 그래픽 하드웨어와 멀티코어(multi-core) CPU를 이용한 빠르고 효율적인 신경망 구현 방법을 제안한다. 기존 GPU의 첫 번째 문제점을 해결하기 위해 제안된 방법은 복잡한 쉐이딩 언어 대신 그래픽스적인 기본지식 없이도 GPU를 이용하여 응용프로그램 개발이 가능한 CUDA를 이용하였다. 두 번째 문제점을 해결하기 위해 멀티코어 CPU에서 공유 메모리 환경의 병렬화를 수행할 수 있는 OpenMP를 이용하였으며, 이는 CPU의 처리시간을 줄여 CPU와 GPU 환경에서 오버헤드를 최소화할 수 있다. 실험에서 제안된 CUDA와 OpenMP기반의 구현 방법을 신경망을 이용한 문자영역 검출 알고리즘에 적용하였으며, CPU에서의 수행시간과 비교하여 약 15배, GPU만을 이용한 수행시간과 비교하여 약 4배정도 빠른 수행시간을 보였다.

키워드 : CUDA, OpenMP, 신경망, 문자추출

Abstract Many algorithms for computer vision and pattern recognition have recently been implemented on GPU (graphic processing unit) for faster computational times. However, the implementation has two problems. First, the programmer should master the fundamentals of the graphics shading languages that require the prior knowledge on computer graphics. Second, in a job that needs much cooperation between CPU and GPU, which is usual in image processing and pattern recognition contrary to the graphic area, CPU should generate raw feature data for GPU processing as much as possible to effectively utilize GPU performance. This paper proposes more quick and efficient implementation of neural networks on both GPU and multi-core CPU. We use CUDA (compute unified device architecture) that can be easily programmed due to its simple C language-like style instead of GPU to solve the first problem. Moreover, *OpenMP* (Open Multi-Processing) is used to concurrently process multiple data with single instruction on multi-core CPU, which results in effectively utilizing the memories of GPU. In the experiments, we implemented neural networks-based text extraction system using the proposed architecture, and the computational times showed about 15 times faster than implementation on only GPU without *OpenMP*.

Key words : CUDA, OpenMP, Neural Network, Text Extraction

· 본 연구는 지식경제부 및 정보통신연구진흥원의 대학 IT 연구센터 지원사업의 연구결과로 수행되었음(IITA-2008-(C1090-0803-0006))
· 이 논문은 2008 한국컴퓨터종합학술대회에서 'CUDA와 OpenMP를 이용한 신경망 구현'의 제목으로 발표된 논문을 확장한 것임

† 학생회원 : 숭실대학교 미디어학과
anjin@ssu.ac.kr
†† 학생회원 : 숭실대학교 미디어학부
rollco@empal.com
††† 종신회원 : 숭실대학교 미디어학부 교수
kcjung@ssu.ac.kr

논문접수 : 2008년 8월 27일
심사완료 : 2009년 2월 23일

정보과학회논문지: 소프트웨어 및 응용 제36권 제4호(2009.4)

1. 서 론

GPU(graphic processing unit)는 하드웨어 구조의 특성상 동일한 연산의 반복 수행에서 CPU보다 더욱 빠른 수행속도를 보인다. 이런 이유 때문에 반복되는 연산이 많은 컴퓨터 비전이나 패턴 인식 등의 분야에서 알고리즘의 빠른 수행을 위해 GPU를 많이 활용하고 있다. 특히 최근 GPU의 속도, 가격, 프로그래밍 가능성 등의 다양한 면에서 경쟁력을 가짐으로써 컴퓨터 비전이나 패턴 인식 분야 등에서 GPU를 이용한 알고리즘 구현이 더욱더 활발히 이루어지고 있다[1-5]. Oh와 Jung[1]은 계층간의 반복적인 내적 연산을 행렬 곱으로 표현할 수 있는 신경망 연산을 GPU에서 구현하였으며, 정점 쉐이더(vertex shader)와 픽셀 쉐이더(pixel shader)를 이용하여 행렬 곱을 수행하였다. Kenneth Moreland와 Eward Angel[2]은 GPU를 이용하여 FFT(fast Fourier transform)를 구현하였으며, Julien Mairal[3]는 GPU를 이용하여 스테레오 매칭을 구현하였다. Gey와 Gool[4]은 GPU를 이용하여 3차원 복원(3D reconstruction)을 구현하였으며, GPU에서 대략적인 복원을 빠르게 수행하고, CPU에서 대략적으로 복원된 영역에서 정교한 복원을 수행하였다. Yang과 Welch[5]는 레지스터 결합(register combiner)과 혼합(blending) 기술을 이용하여 GPU 내에서 컴퓨터 비전 분야의 기본적인 연산인 영상 분할과 모폴로지 연산을 구현하였다.

위에서 제안된 방법들은 GPU를 이용하여 구현함으로써 CPU에서 보다 빠른 수행시간을 보였지만, GPU를 이용하여 일반적인 프로그램을 구현할 때 고려해야 할 두 가지 문제점에 대해 언급하지 않았다.

첫째, GPU에서 알고리즘을 수행하기 위해선 HLSL(DirectX)[6], Cg(nVIDIA)[7], GLSL(OpenGL 2.0)[8] 등의 쉐이딩(shading) 언어를 이용하여 작성해야 하며, 쉐이딩 언어로 알고리즘을 구현하기 위해선 기본적인 그래픽스 개념과 GPU의 구조를 이해해야만 한다. 보다 쉽게 GPU를 이용하여 알고리즘을 구현할 수 있게 Brook[9]과 같은 언어들이 발표되었지만, 기존의 쉐이딩 언어보다 느린 수행시간을 보이기 때문에 GPU를 사용하는 목적이 반감될 수 밖에 없어 일반적으로 잘 사용하지 않는다.

둘째, CPU와 GPU가 서로 다른 메모리를 사용하기 때문에 용량이 많은 데이터를 처리하기 위해선 서로간의 잦은 데이터 교환이 필요하며, 이는 GPU의 효율적인 사용을 저해한다. CPU와 GPU 간의 데이터 교환을 줄이기 위해선 GPU에서 처리할 수 있는 최대한의 크기로 서로간의 데이터 교환이 이루어져야 한다. 하지만 전송할 데이터의 크기가 커지면 커질수록 CPU에서의 연산이 많아지고, 이로 인해 CPU와 GPU 사이의 병목현상 때문에 GPU를 효율적으로 사용하지 못하는 결과를 초래한다.

본 논문에서는 그래픽 하드웨어 GPU와 멀티코어 CPU를 이용한 빠르고 효율적인 신경망 구현 방법을 제안한다. GPU의 첫 번째 문제점인 기존의 복잡한 쉐이딩 언어 대신 C 언어 형식으로 구현이 쉽고 일반 수학적인 연산을 더욱 효율적으로 사용할 수 있는 CUDA를 이용하였으며, 신경망에서 반복적으로 이용되는 행렬의 곱 연산을 CUDA의 메모리 환경에 적합하고 병렬로 효과적으로 처리할 수 있게 구현하였다. 두 번째 문제점인 대용량의 데이터 수집을 위해 소모되는 CPU에서의 처리시간을 최소화하기 위해 공유 메모리 환경에서 프로그램을 병렬화할 수 있는 OpenMP를 이용하여 구현하였다. 제안된 멀티코어 CPU와 CUDA를 이용한 구현 방법을 평가하기 위해 본 논문에서 신경망을 구현하였으며, 그림 1은 전체 흐름도를 보여준다. 영상 데이터를 입력 받으면 GetConfigMatrix() 함수를 통해 특징값을 추출하고 추출된 데이터를 행렬의 형태로 만든다. 만들어진 행렬은 GPU로 전송되며 GPU는 신경망 과정을 거쳐 결과를 CPU로 재전송한다. 특징값 추출할 때 처리 시간을 최소화하기 위해 멀티코어 CPU를 이용하여 병렬로 처리하며 추출된 특징값을 GPU로 전달한다. CUDA는 공유 메모리와 쓰레드(thread)를 이용하여 행렬의 곱 연산을 병렬로 수행하며, 결론적으로 제안된 방법은 GPU와 멀티코어 CPU에서 병렬로 수행할 수 있게 신경망을 설계하였다. 실험에서 제안된 신경망 구현 방법의 정확도와 수행시간을 비교하기 위해 신경망을 이용한 문자영역 검출을 수행하였으며, CPU에서의 수행시간과 비교하여 약 15배, GPU만을 이용한 수행시간[1]과 비교하여도 약 4배 정도 빠른 수행시간을 보였다.

본 논문의 구성은 다음과 같다. 제2장에서 CUDA와 OpenMP에 대해 기술하고, 제3장에서 CUDA 기반의 신경망 구현을 위한 제안된 방법을 구체적으로 설명한다. 제4장에서는 실험 및 결과에 대해 언급하고, 제5장에서 결론 및 향후 연구 방향을 기술한다.

2. CUDA와 OpenMP

본 장에서는 효율적인 신경망 구현을 위한 CUDA와 OpenMP의 사용을 위해, CUDA와 OpenMP의 정의와 효율성에 대해 기술한다.

2.1 CUDA

CUDA(compute unified device architecture)는 C 언어를 기반으로 만들어진 nVIDIA에서 개발한 GPU 프로그래밍 언어이다[10]. 기존에 GPU를 이용하여 수학적인 연산을 하기 위해선 그래픽 함수를 이용한 쉐이딩

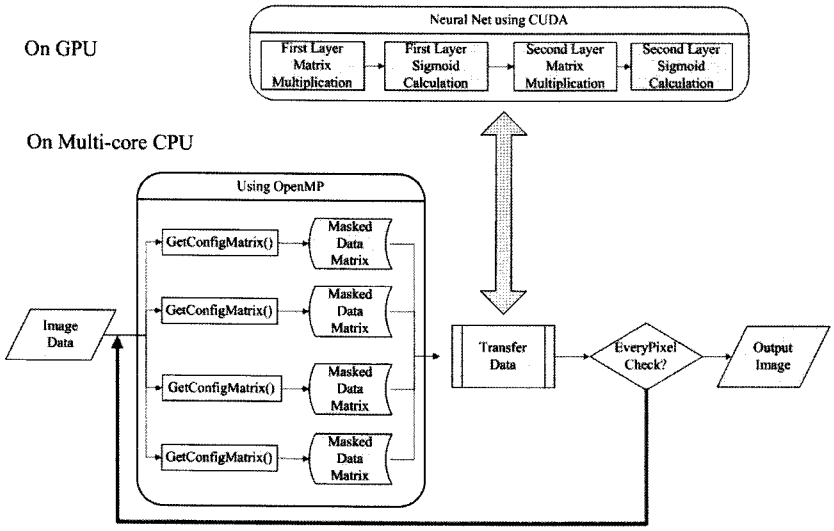


그림 1 CUDA와 OpenMP를 이용한 신경망 구현의 전체 흐름도

프로그래밍을 통하여 구현할 수 있었다. 그러나 쉐이딩 프로그래밍은 그래픽에 대한 사전지식을 많이 필요로 하기 때문에 사용하기도 어렵고 접근 또한 쉽지 않다. 하지만 CUDA는 C언어를 기반으로 하기 때문에 C를 알고 있는 프로그래머라면 복잡한 그래픽에 대한 사전 지식 없이 쉽게 GPU를 이용한 일반연산을 구현할 수 있다. 이는 일반적인 수학 연산자를 데이터를 병렬로 처리할 수 있는 GPU 상에서 쉽고 효과적으로 구현할 수 있다는 뜻이다. 또한 기존에 쉐이딩 언어를 사용함으로써 발생할 수 있는 오버헤드가 제거됨으로써 더욱 효율적인 연산수행을 가능하게 한다.

CUDA도 GPU를 이용하는 언어이기 때문에 하드웨어의 구조적인 원인에서 오는 문제점을 가지고 있다. 즉 CPU와 GPU간에 메모리를 공유하여 사용할 수 없으며, GPU에서 연산을 하기 위해선 주기억장치로부터 데이터를 전송 받아 연산을 수행해야 한다. 데이터 전송에 따른 지연을 줄이고 프로그램의 효율을 높이기 위해선 최소한의 데이터 교환이 일어나야 하며, 이를 위해 CPU는 GPU에서 처리할 수 있는 최대 용량의 데이터를 생성해야 한다. 하지만 이 경우 CPU에서 많은 처리 시간이 소모되며, 이로 인해 CPU와 GPU사이에 병목현상이 발생한다. CPU에서 GPU에서 처리할 최대의 데이터를 만들 때 발생하는 처리시간을 최소화하기 위해 본 논문에서는 OpenMP를 이용한 CPU의 병렬처리를 이용하였다.

2.2 OpenMP

OpenMP는 C/C++와 Fortran에서 공유메모리 방식의 병렬화를 구현할 수 있는 API(application program interface)로 현재 버전 2.5까지 발표되었으며, 최근 멀티코어 CPU가 주류를 이루면서 OpenMP가 주목 받고 있다. OpenMP는 OpenMP를 인식하는 컴파일러가 멀티 쓰레드를 이용해서 병렬처리가 가능한 실행파일을 생성하도록 지시하는 방식으로 동작되며, 컴파일러 지시자를 통하여 병렬화될 코드 블록을 어떻게 처리할지 알려준다.

OpenMP의 병렬화 실행 모델은 fork-join 모델을 이용한다. 초기 마스터(master) 쓰레드 하나로 시작하여 병렬화될 코드 블록을 만나면 추가 슬레이브(slave) 쓰레드를 생성하여 코드를 수행하고 병렬화 구역이 끝나면 마스터 쓰레드를 남기고 추가된 슬레이브 쓰레드를 종료한다. 그림 2는 OpenMP의 fork-join 모델이 동작하는 방식을 도식화한 그림이며, 마스트 쓰레드가 여러 개의 슬레이브 쓰레드를 분기(fork)하고, 각 슬레이브 쓰레드는 병렬로 처리됨을 보여준다. 컴파일러 지시자

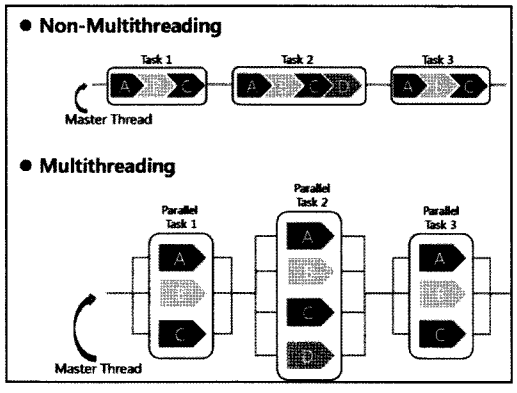


그림 2 OpenMP의 구조

외에도 OpenMP에서는 쓰레드와 관련된 몇 가지 런타임 루틴(runtime routine)을 제공한다. 런타임 루틴을 통해서 현재 사용되는 쓰레드의 정보, 병렬화에 이용할 쓰레드의 개수 설정, 최대 사용가능 쓰레드 개수 등을 알 수 있으며, 이를 이용하여 자신의 환경에 가장 적합한 병렬화 처리가 가능해진다.

OpenMP는 대부분의 컴파일러에서 지원하고 있기 때문에 개발 환경에 상관없이 OpenMP를 대부분의 컴파일러에서 이용이 가능하다. OpenMP에 대한 더 많은 정보는 OpenMP 웹사이트[11]에서 얻을 수 있다.

CUDA로 보낼 대용량의 데이터 생성시 OpenMP를 이용한 병렬화 방식은 동일한 시간에 더 많은 데이터 생성을 가능하게 한다. 이로 인해 CUDA로 전송할 대용량의 데이터를 생성할 때 발생하는 오버헤드를 줄일 수 있으며, CUDA만을 사용하였을 때 보다 약 1/4로 수행시간을 줄어든 것을 실험을 통해 확인할 수 있다.

3. 신경망 구현

신경망이라는 불리는 인공 신경망(artificial neural network)은 인간 두뇌의 동작 방식을 따라 구현하였으며, 다양한 모양의 신경망이 이용되고 있지만, 대표적으로 다층 퍼셉트론(multilayer perceptron: MLP)을 주로 많이 이용한다.

MLP는 1개 이상의 은닉층과 다수의 출력 노드를 가질 수 있다. 일반적으로 MLP는 인접한 층의 노드들이 완전 연결(fully connected)되어 있고, 층의 개수나 각 층의 노드 수 등에서 변화가 있을 수 있지만, 기본적으로 각 노드는 식 (1)과 같이 연결 가중치 벡터와 해당 노드의 입력 벡터의 내적 연산(inner product)을 수행한 후, 식 (2)와 같은 활성화 함수(activate function)를 수행한다.

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

$$r_j = (1 + e^{-p_j})^{-1} \tag{2}$$

식 (1)과 (2)에서 첨자 j는 현재 계산되는 노드의 인덱스를 의미하며, i는 j 번째 노드와 연결되어 있는 하위층 노드의 인덱스를 의미한다. 식 (1)의 w_{ji}는 j 번째 노드와 i 번째 노드 사이의 연결 가중치, x_i는 현재 노드의 입력값, b_j는 바이어스 항을 의미하며, 식 (2)의 r_j는 j 번째 노드의 최종 출력값을 의미한다. 식 (1)과 (2)의 연산은 MLP의 첫 번째 은닉층의 노드들부터 계산되며 차례로 출력층까지 수행한다.

신경망의 각 노드에서의 내적 연산은 입력 벡터와 가중치 벡터를 하나의 행렬에 축적함으로써 행렬의 곱 연산으로 변환할 수 있으며, 다음과 같이 표현할 수 있다.

$$W = \begin{bmatrix} w_{10} & w_{11} & \cdots & w_{1N} \\ w_{20} & w_{21} & \cdots & w_{2N} \\ \vdots & \vdots & \ddots & \vdots \\ w_{M0} & w_{M1} & \cdots & w_{MN} \end{bmatrix} = \begin{bmatrix} W_1 \\ W_2 \\ \vdots \\ W_M \end{bmatrix}, \tag{3}$$

$$X = \begin{bmatrix} 1 & 1 & \cdots & 1 \\ x_{11} & x_{12} & \cdots & x_{1L} \\ \vdots & \vdots & \ddots & \vdots \\ x_{N1} & x_{N2} & \cdots & x_{NL} \end{bmatrix} = [x_1 \quad x_2 \quad \cdots \quad x_L], \tag{4}$$

$$P = W \times X = \begin{bmatrix} w_1 \cdot x_1 & w_1 \cdot x_2 & \cdots & w_1 \cdot x_L \\ w_2 \cdot x_1 & w_2 \cdot x_2 & \cdots & w_2 \cdot x_L \\ \vdots & \vdots & \ddots & \vdots \\ w_M \cdot x_1 & w_M \cdot x_2 & \cdots & w_M \cdot x_L \end{bmatrix} = \begin{bmatrix} p_{11} & p_{12} & \cdots & p_{1L} \\ p_{21} & p_{22} & \cdots & p_{2L} \\ \vdots & \vdots & \ddots & \vdots \\ p_{M1} & p_{M2} & \cdots & p_{ML} \end{bmatrix}, \tag{5}$$

$$R = sigmoid(P) = \begin{bmatrix} (1+e^{-p_{11}})^{-1} & (1+e^{-p_{12}})^{-1} & \cdots & (1+e^{-p_{1L}})^{-1} \\ (1+e^{-p_{21}})^{-1} & (1+e^{-p_{22}})^{-1} & \cdots & (1+e^{-p_{2L}})^{-1} \\ \vdots & \vdots & \ddots & \vdots \\ (1+e^{-p_{M1}})^{-1} & (1+e^{-p_{M2}})^{-1} & \cdots & (1+e^{-p_{ML}})^{-1} \end{bmatrix}, \tag{6}$$

여기서 M은 현재 계층의 노드 개수를 의미하고, N은 하위 계층의 노드 개수를 의미하며, x_{ji}는 j 번째 입력벡터의 i 번째 값을 나타낸다. p_{ji}는 j 번째 입력 벡터에 대한 i 번째 출력 노드의 결과이며, R은 한 계층의 연산 결과의 행렬을 의미한다.

그림 3은 CUDA를 이용한 행렬의 곱(식 (5))과 활성화 함수(식 (6))를 보여준다. CUDA에서 큰 특징 중 하나는 공유 메모리라는 새로운 영역이 추가된 것이며, 이를 이용하여 행렬의 곱을 계산함으로써 더 효율적인 연산이 가능하다. 기존 GPU를 이용할 경우 전역 메모리

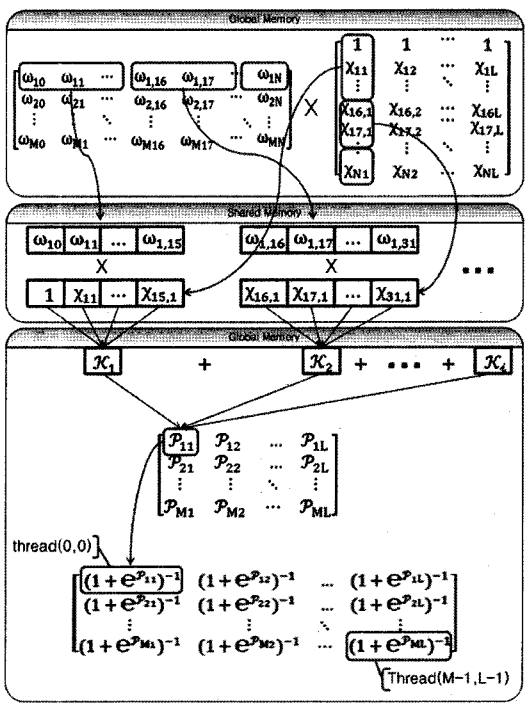


그림 3 CUDA를 이용한 신경망 연산

를 이용하여 데이터를 접근하였으며 대략 400~600 사이클의 메모리 지연이 발생하게 된다. 하지만 공유 메모리를 이용할 경우 메모리 지연이 4 사이클 밖에 발생하지 않기 때문에 훨씬 효율적인 연산이 가능하다. 활성화 함수는 행렬의 크기만큼 쓰레드를 생성하여 각각의 쓰레드에서 식 (6)의 각 요소(예를 들어 $\left(1+e^{-p_{ij}}\right)^{-1}$)를 독립적으로 수행함으로써 병렬처리를 수행한다.

4. 실험 및 결과

모든 실험은 다음과 같은 환경에서 수행하였다. OpenMP를 사용하기 위해 Intel Core 2 Quad Q6600 (2.4GHz)를 사용하였고, CUDA 사용을 위해 GPU는 GeForce 8800 GTX(768MB)를 사용하였다. 신경망의 실험을 위해 컴퓨터 비전 분야의 하나인 문자영역 검출 알고리즘을 제안된 구현방법에 적용하였다. 이번 장에서는 4.1에서 신경망을 이용한 문자영역 검출 알고리즘을 설명하고, 4.2에서 문자영역 검출의 수행 시간과 결과를 비교 분석한다.

4.1 신경망 기반의 문자영역 검출

최근 영상 처리 기술을 이용하여 비디오 데이터와 영상으로부터 문자영역을 검출하는 연구들이 다양하게 시도되고 있다. 영상이나 비디오 데이터로부터 문자영역을 검출하는 방법은 문자 인식을 위한 전처리 단계로써 중요한 부분을 차지하며, 신경망을 이용할 경우 다른 방법보다 많은 이점이 있다[12].

문자영역 검출을 위해 사용한 신경망의 구조는 2개의 은닉층, 1개의 출력노드로 구성되며 인접한 노드들은 모두 연결되어 있고, 입력층의 입력 값으로 그림 4와 같이 입력 영상에서 Y×Y 크기의 마스크내의 검은색으로 표시된 픽셀들의 그레이(gray) 값을 이용하였다. 신경망을 이용하여 입력 영상을 처리함으로써 생성된 출력 영상의 각 픽셀은 대응하는 입력 영상내의 픽셀의 문자 여부를 나타낸다. 그림 4는 신경망의 입력층이 생략되어 있고 대신 입력 픽셀의 모양으로 이를 대신한다. 실험에서 11×11의 윈도우 마스크를 사용하였으며, 33개의 노드를 가진 은닉층을 이용하여 실험하였다.

입력 영상의 값과 Y×Y의 윈도우 마스크와의 연산을 통해 그레이 값을 얻고, 이렇게 분류된 영상 값은 신경망에 보내질 데이터로 수집된다. 이 과정에서 멀티코어를 이용한 병렬화 방식을 이용하는 OpenMP를 이용하여 동일한 시간 동안에 더 많은 데이터를 수집할 수 있게 된다. 그림 5는 이 과정의 의사코드(pseudo code)를 보여주며, 두 개의 쓰레드를 할당하기 위해 컴파일러 지시자 '**#pragma omp section**'이 두 개 포함되어 있다. 이렇게 수집된 데이터는 GPU에 전달되게 된다. 신경망에 전달된 데이터는 GPU를 이용하여 신경망을 수행하게 되며 이 과정의 의사코드는 그림 6을 통해 보이고 있다.

4.2 문자영역 검출의 결과

그림 7은 영상 크기에 따른 수행결과를 보인다. 영상 크기는 다음과 같이 분류하여 실험하였다. 그림 7(a,b) 320×240, (c,d) 571×785, (e,f) 1152×1546, 3가지 영상으로 이용하였다. 그리고 그림 7(a,c,e)는 입력영상이고, (b,d,f)는 CUDA와 OpenMP를 이용하여 신경망을 수행

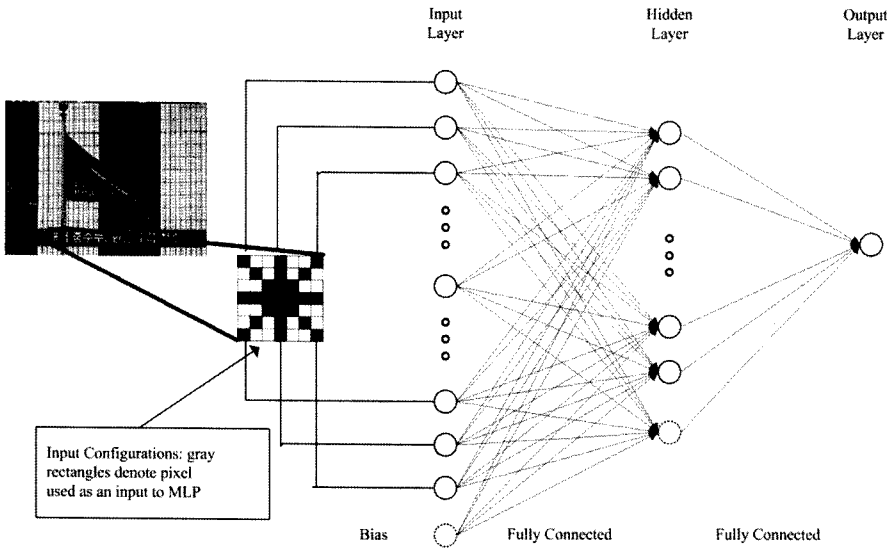


그림 4 신경망의 개략적인 구조

```
ImageLoad(filename);
for(check everyPixel of image)
{
  #pragma omp parallel section                  //Parallel Implementation using OpenMP
  {
      #pragma omp section
      {
          GetConfigMatrix(cpuData1);            //pixel check in window range
      }
      #pragma omp section
      {
          GetConfigMatrix(cpuData2);            //pixel check in window range
      }
  }
  ForwardCUDA(cpuData1,outputCUDAData);         //calculate neural network using CUDA
  SaveOutputData(outputCUDAData);

  ForwardCUDA(cpuData2,outputCUDAData);         //calculate neural network using CUDA
  SaveOutputData(outputCUDAData);
}
```

그림 5 OpenMP의 의사코드

```
cublasSetMatrix(CPUData, CUDAData );    //memory copy from CPU to GPU
                                        //matrix multiplication of first layer
cublasSgemm(Weight0, CUDAData, Result0);//Result 0 = Weight0 * GPUData
Sigmoid(Result0);                       // sigmoid calculation of first layer
                                        // matrix multiplication of second layer
cublasSgemm(Weight1, Result0, Result1); //Result1 = Weight1 * Result0;
Sigmoid(Result1);                       //sigmoid calculation of second layer
cublasGetMatrix(Result1, outputCPUData);//memory copy from GPU to CPU
```

그림 6 CUDA에서 수행되는 의사코드

한 결과 영상이다. 문자영역으로 검출된 부분의 픽셀을 검은색으로 출력하였다.

그림 8은 영상의 크기에 따른 수행시간을 보여준다. 그림 8에서 영상의 크기가 작을 경우 수행시간에서 큰 차이를 보이지 않는다. 이는 영상의 크기가 작을 경우 CPU만을 이용해서 신경망을 구현해도 빠르게 수행할 수 있기 때문이다. 하지만 영상의 크기가 커지면 CPU만을 사용하였을 경우보다 CUDA와 함께 이용한 경우 약 6배의 성능 차이를 보였고, OpenMP와 CUDA를 같이 사용한 경우는 CUDA만을 사용한 경우보다 약 4배의 성능향상을 보였다.

그림 9는 OpenMP를 사용할 때의 장점을 보여준다. 그림 9에서 보는 바와 같이 OpenMP를 사용하지 않을 경우(Only CUDA) GPU와 CPU의 수행시간의 차이가 약 8배 정도이며, 이는 CPU의 병목현상으로 인해 GPU를 충분히 활용하지 못하는 문제를 발생시킨다. 반면 OpenMP를 사용할 경우 (OpenMP+CUDA) GPU와 CPU의 수행시간의 차이가 약 2배 정도로, OpenMP를 사용하지 않을 경우 비교해서 수행시간의 차이가 약 4배 정도 줄어들었다. 이로 인해 OpenMP를 함께 사용

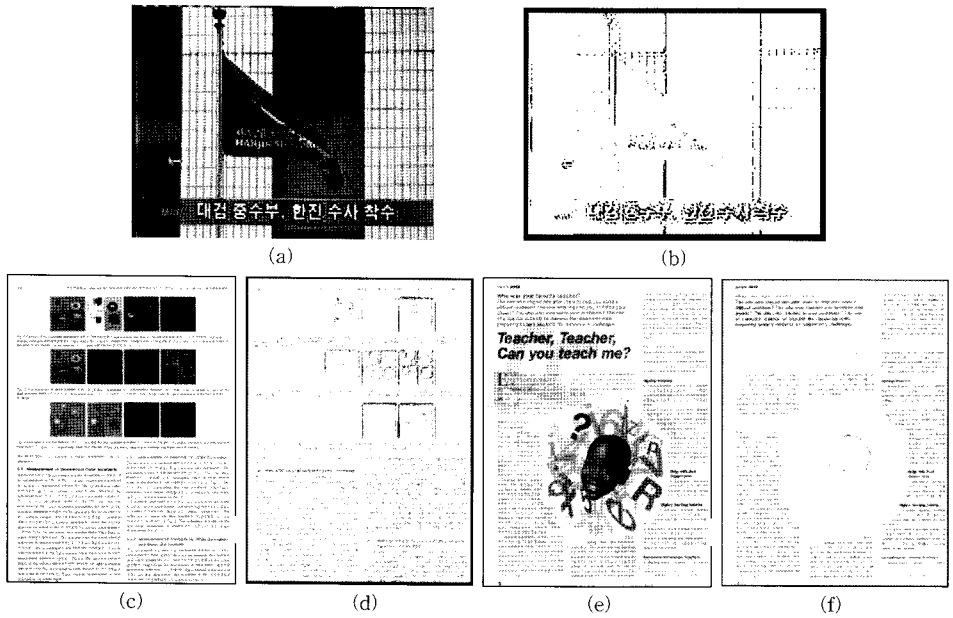
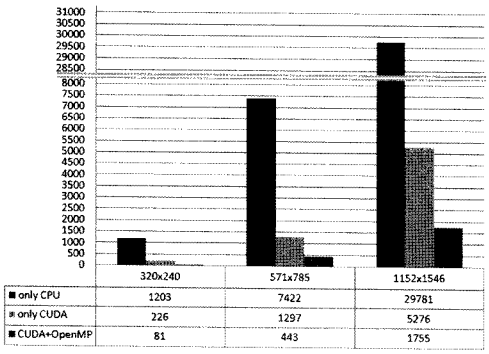


그림 7 결과 영상: (a,c,e) 입력영상, (b,d,f) 문자영역 검출 결과영상



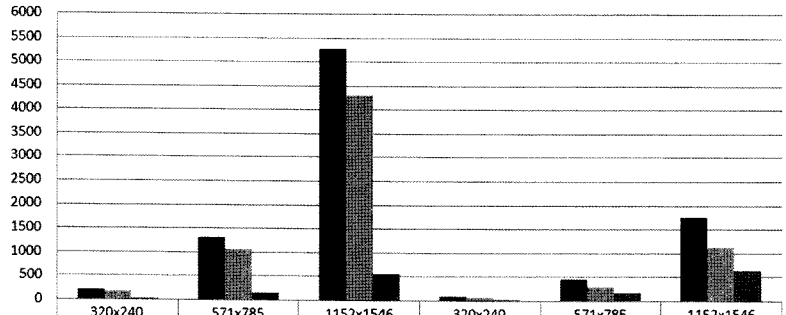
	320x240	571x785	1152x1546
only CPU	1203	7422	29781
only CUDA	226	1297	5276
CUDA+OpenMP	81	443	1755

그림 8 3가지 방법에 대한 총 수행시간(msec.)

할 경우 CPU에서 발생되는 병목현상을 최소화할 수 있으며, 기존 GPU만을 이용한 경우[1]와 비교해서 약 4배 정도 빠른 수행시간을 보였다.

5. 결 론

본 논문에서는 GPU로 보내어질 대용량의 데이터 생성에 이용되는 OpenMP와 GPU의 병렬처리를 통해 보다 빠르고 효율적인 신경망 구현 방법을 제안하였다. GPU를 위해 사용된 CUDA는 컴퓨터 그래픽스에 대한 상세한 사전지식을 필요로 하지 않는 C언어 스타일의 코드로 작성할 수 있으며, 그래픽 API 함수 없이도 GPU



	320x240	571x785	1152x1546	320x240	571x785	1152x1546
	only CUDA			CUDA+OpenMP		
Total	211	1297	5276	82	443	1755
CPU	172	1060	4309	61	283	1114
GPU	22	138	561	25	157	630

그림 9 OpenMP를 사용할 때와 하지 않을 때의 구간별 시간 비교(msec.)

를 이용한 연산이 가능하여 보다 쉽고 효율적으로 GPU 연산을 수행할 수 있다. 또한 여러 개의 데이터를 병렬로 처리할 수 있도록 도와주는 OpenMP는 GPU에 전송하기 위한 대용량의 데이터 생성에서 오는 오버헤드를 최소화하였다. 우리는 신경망을 CUDA와 OpenMP를 사용하여 구현하였다. 특징값 추출에는 CPU의 병렬처리를 이용하여 수행하였고, 주요 신경망의 연산인 내적 연산은 GPU를 이용하는 CUDA를 사용하여 수행하였다. 실험 결과를 보면 GPU를 사용하는 CUDA만을 사용하는 것보다 CPU의 병렬처리를 동시에 사용하여 신경망을 수행하는 것이 효율적이라는 것을 확인할 수 있다.

참 고 문 헌

[1] K.S. Oh and K. Jung, "GPU Implementation of Neural Network," Pattern Recognition, Vol.37, Issue 6, pp. 1311-1314, 2004.

[2] K. Moreland and E. Angel, "The FFT on a GPU," Proceedings of SIGGRAPH Conference on Graphics Hardware, pp. 112-119, 2003.

[3] J. Mairal, R. Keriven, and A. Chariot, "Fast and Efficient Dense Variational Stereo on GPU," Proceedings of International Symposium on 3D Data Processing, Visualization, and Transmission, pp. 697-704, 2006.

[4] I. Geys and L.V. Gool, "View Synthesis by the Parallel Use of GPU and CPU," Image and Vision Computing, Vol.25, Issue 7, pp. 1154-1164, 2007.

[5] R. Yang and G. Welch, "Fast Image Segmentation and Smoothing using Commodity Graphics Hardware," Journal of Graphics Tools, Vol.17, Issue 4, pp. 91-100, 2002.

[6] http://ati.amd.com/developer/

[7] http://developer.nvidia.com/object/cg_toolkit.html/

[8] http://www.opengl.org/documentation/glsl/

[9] http://graphics.stanford.edu/projects/brookgpu/

[10] http://www.nvidia.com/object/cuda_home.html/

[11] http://www.openmp.org/

[12] K. Jung, "Neural Network-based Text Localization in Color Images," Pattern Recognition Letters, Vol.22, Issue 4, pp. 1503-1515, 2001.

박 안 진

2006년 숭실대학교 미디어학과 석사. 2009년 숭실대학교 미디어학과 공학박사. 2009년 방문연구원, Center for Service Research, AIST, Japan. 관심분야는 컴퓨터비전, 영상분할, 신경망

장 홍 훈

2002년~현재 숭실대학교 IT대학 미디어학부 4학년 재학 중. 관심분야는 영상처리/컴퓨터비전, 패턴인식

정 기 철

1996년 경북대학교 컴퓨터공학과 공학석사. 1999년 방문연구원, Intelligent User Interface Group, DFKI(The German Research Center for Artificial Intelligence, GmbH), Germany. 1999년 방문연구원, Machine Understanding Division, Electro Technical Laboratory, Japan. 2000년 경북대학교 컴퓨터공학과 공학박사. 2000년~2002년 박사후 연구원, PRIP Lab., Michigan State University, USA. 2007년 방문교수, SIAT(School of Interactive Art and Technology), SFU, Canada. 2006년~2007년 숭실대학교, 정보과학대학원, 디지털컨텐츠학과, 학과장. 2003년~현재 숭실대학교 IT대학 미디어학부 교수. 관심분야는 HCI, Interactive Contents, 실감형게임, 영상처리/컴퓨터비전, 패턴인식, 증강현실, 인공지능