특집

GPU를 이용한 신경망 구현

오경수·정기철 (숭실대학교)

I. 서 론

"GPGPU(General-Purpose computing on Graphics Processing Units: 그래픽 처리 장치를 통한 일반 목적의 컴퓨팅)는 GPGPU라고도 불리며, 영상을 생성하는 컴퓨터 그래픽스를 위한 계산만 다루는 GPU를 사용하여 CPU에 전통적으로 수행했던 계산을 수행하는 기술이다. 다음은 위키피디아 영문판에서 정의하는 GPGPU의 정의를 번역한 것이다.

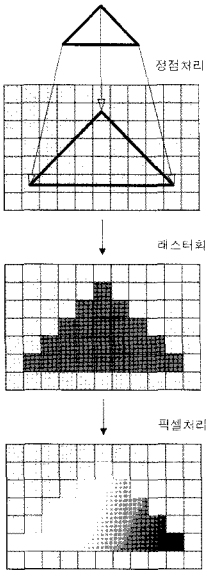
"GPGPU는 GPU를 이용하여 일반적 목적의 프로그램을 제작하는 것이다. GPU는 고성능의 다-코어 처리장치로 매우 복잡한 계산과 다량의 자료를 빨리 처리할 수 있다. 예전에는 컴퓨터 그래픽스용으로만 특별하게 설계되어 프로그램하는 것이 매우 어려웠지만 요즘의 GPU는 c와 같은 많이 일반적인 언어로도 접근 가능한 일반적 목적의 병렬 프로세서이다. CPU에서 동작하던 프로그램을 GPU에서 동작하도록 수정하면 수십, 수백배의 속도 향상을 얻는 경우가 많다."

위 정의와 같이 GPU를 일반적인 연산에 사용하는 GPGPU는 선형대수, 물리엔진, 계산기하, 영상처리, 신호처리, 과학계산 그리고 컴퓨터 비젼등 수많은 복잡한 계산에 활용되고 있다.

이 글에서는 GPU를 이용하여 신경망계산을 빠르게 수행하는 기법에 대해서 설명하려고 한다. 우선 GPU에 대한 배경 지식 및 특징에 대해서 설명하고 GPU를 활용하기에 적합한 으용 분야에 대해서 생각해 보고 그래픽스 파이프라인을 이용한 신경망 구현과 CUDA를 이용한 신경망 구현의 예를 설명하겠다.

II. GPU의 특징

우선 GPU의 특징을 이해하려면 그래픽스 파이프라인의 이해가 필요하다. 3차원 그래픽스는 3차원 점과 그들의 이어짐 정보인 삼각형 메시 정보와 카메라 및 조명, 물체의 반사 특성 등을 입력 받아 가상의 3차원 영상을 출력한다. 이러한 일련의 과정을 하드웨어로 구현한 것이 초기 GPU였고 이러한 일련의 과정을 그래픽스 파이프라인이라고 한다. 그래픽스 파이프라인 중 중요한 부분이 두군데 있는데 정점 처리 부분과 픽셀 처리 부분이다. 정점 처리 부분은 그림에서 보는 바와 같이 입력된 정점으로부터 화면 속에서



〈그림 1〉 렌더링 파이프라인

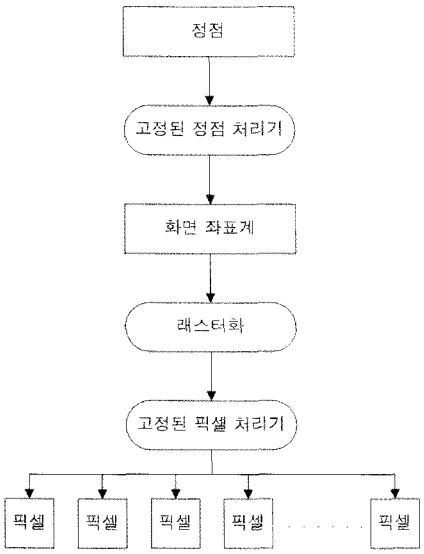
의 위치를 계산하고 그 밝기를 결정하는 부분이다. 픽셀 처리 부분은 다각형이 차지하는 픽셀들의 색상을 텍스처, 조명 결과 등을 활용하여 결정하는 부분이다.

1. 고정 파이프 라인(fixed pipeline)

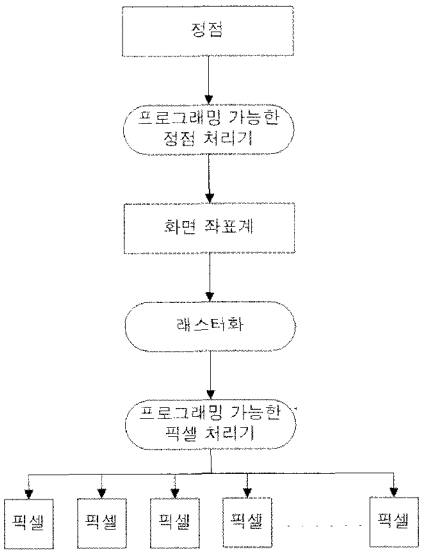
고정 파이프라인은 정점처리부분과 픽셀 처리 부분의 알고리즘이 고정된 방식이다. 초기의 GPU들은 모두 이러한 방식을 따랐고 이러한 방식에서는 알고리즘의 입력만을 제어해서 결과를 얻을 수밖에 없다. 예를 들어 정점의 위치를 결정하기 위해서 행렬형태의 변환 정보만을 제어할 수 있고 정점의 화면상의 위치를 결정하는 알고리즘은 정점의 값에 행렬을 곱하는 알고리즘으로 고정되어 있다.

2. 프로그램 가능한 파이프 라인

프로그램 가능한 파이프 라인은 정점 처리 부분과 픽셀 처리 부분의 알고리즘을 고정된 알고리즘을 사용하는 것이 아니라 알고리즘을 프로그래머가 c와 유사한 형태의 프로그램을 작성해서 컴파일 한 후 GPU에 전달하면 GPU가 이를 읽어서 수행하는 방식이다. 알고리즘을 바꿀수 있으므로 고정 파이프라인에 비해서 매우 다양한 응용이 가능하다.



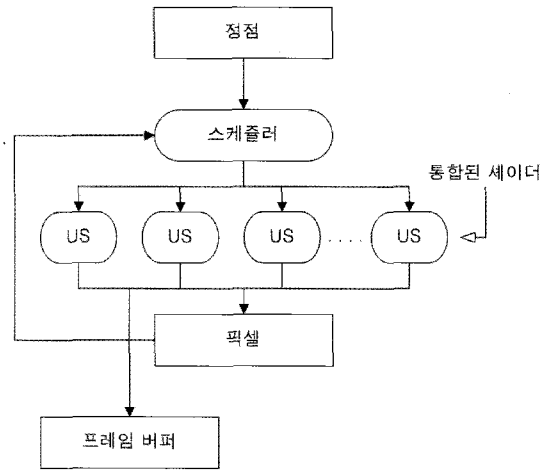
〈그림 2〉 고정된 파이프라인



〈그림 3〉 프로그래밍 가능한 파이프라인

3. 통합화된 프로세서

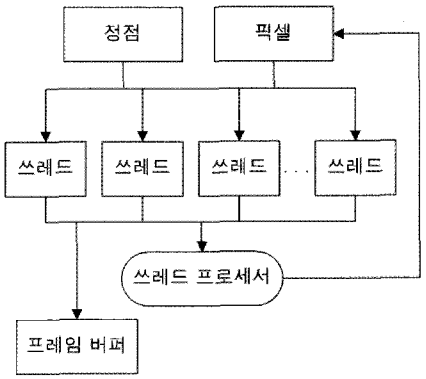
통합화된 프로세서(unified processor)개념에 관한 설명은 소프트웨어 엔지니어 입장에서는 관심없을 수 있는 문제이고 하드웨어 구조에 관한 이야기 이다. 프로그램 가능한 파이프라인에서도 정점 처리기와 픽셀 처리기는 요구되는 기능이 달라서 보통 다른 하드웨어 모듈로 구현되었다. 이렇게 정점처리기와 픽셀 처리기가 하드웨어적으로 다르게 구현이 되면 정점처리와 픽셀처리에 필요한 계산량에 따라 어느 한쪽은 사용율이 낮게 되어 낭비 되는 상황이 많이 발생한다. 이를 개선하기 위한 것이 통합화된 프로세서이다. 통합화된 프로세서는 정점처리기와 픽셀처리기에서 요구되는 기능을 모두 갖춘 하드웨어 모듈이고 필요에 따라 정점처리기 픽셀 처리기 모두의 역할을 할 수 있다. 소프트웨어 엔지니어는 통합화된 프로세서이건 그렇지 않건 간에 정점처리 알고리즘과 픽셀 처리 알고리즘을 작성해서 GPU에 전달해주면 GPU에서 하드웨어적으로 통합화된 프로세서들을 사용하여 알고리즘을 처리하게 된다.



〈그림 4〉 통합화된 프로세서

4. CUDA프로그래밍 모델

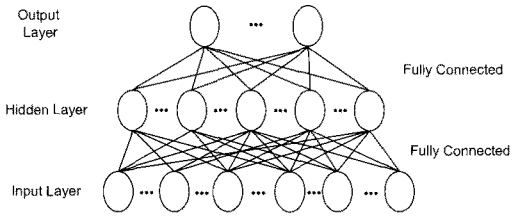
CUDA는 GPU를 그래픽스에 대한 지식 없이도 효율적으로 사용할 수 있게 하는 c언어 개발환경이다. CUDA환경에서는 소프트웨어 관점에서도 정점 처리기, 픽셀 처리기라는 개념이 없어지고 임의의 알고리즘을 작성하면 GPU내부에 있는 쓰레드들로 분산되어서 동시에 처리되게 된다. 병렬로 처리되어야 할 부분을 얼마나 정확히 지정하느냐에 따라 얻을 수 있는 효율성이 좌우된다.



〈그림 5〉 CUDA 프로그래밍 모델

III. 신경망과 GPU

Hopfield network, Adaptive Resonance Theory(ART), Multilayer Perceptron(MLP), Self Organizing Feature Map(SOFM) 등, 많은 종류의 인공신경망 구조가 제안되고 사용되고 있지만, 핵심이 되는 기본적인 구조는 매우 유사하다. 이들은 기본적인 연산을 수행하는 단위인 노드(node)들의 집합인 레이어(층, layer)로 구성되며, 각 노드들과 레이어들은 일반적으로 서로 독립적으로 연산을 수행하기 때문에 병렬 구현 시 많은 장점이 있다.



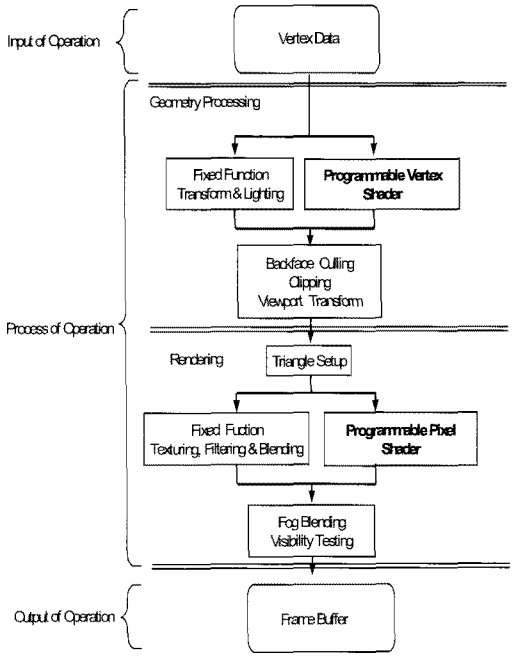
〈그림 6〉 전형적인 MLP의 구조

<그림 6>은 일반적인 다층 퍼셉트론(multilayer perceptron : MLP)의 예 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 노드 수 등에서 변화가 있을 수 있지만, 기본적으로 각 노드는 연결가중치벡터와 해당 노드의 입력벡터의 내적연산을 수행한 후, 식 활성함수 연산을 수행하는 구조로 진행된다.

이와 같이 신경망의 가장 핵심이 되는 연산이 벡터들 간의 내적 연산이고, 이러한 연산은 다수의 입력데이터를 동시에 처리함으로써 GPU 연산이 성능의 우위를 보일 수 있는 행렬 간의 곱연산으로 이루어 질 수 있다는 점이, 신경망의 GPU 구현의 장점이라 할 수 있다.

IV 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 연



〈그림 7〉 렌더링 파이프라인의 전형적인 예

산이 많은 분야인 렌더링을 위해서 개발되었기 때문에, 반복된 연산을 많이 가지는 분야에서는 CPU보다 성능을 향상시킬 수 있다.

GPU를 이용해서 일반적인 연산을 하는 과정은 <그림 7>과 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.

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 노드 수 등에서 변화가 있을 수 있지만,

기본적으로 각 노드는 식 (1)과 같이 연결가중치 벡터와 해당 노드의 입력벡터의 내적연산을 수행한 후, 식 (2)와 같은 활성함수 연산을 수행한다.

$$m_j = \sum w_{ji} x_i + b_j \quad (1)$$

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

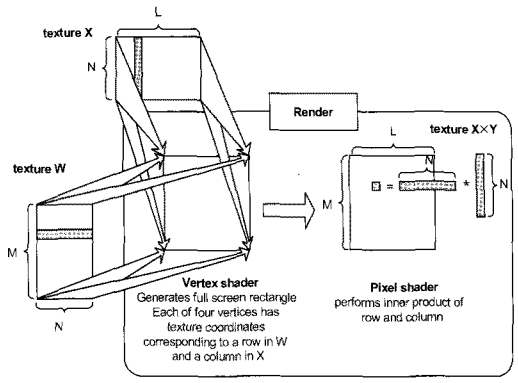
식 (1)과 식 (2)에서 첨자 j는 출력 노드를 의미하며, i는 j-번째 노드와 연결되어 있는 하위층의 노드이다. wji는 j-번째 노드와 i-번째 노드를 연결하는 연결가중치, xi는 입력값, bj는 j-번째 노드의 바이어스 항(bias term), rj는 j-번째 노드의 최종 출력값을 의미한다. MLP의 첫 번째 은닉층의 노드들부터 이와 같은 연산을 수행하여 차례로 출력층까지 계산한다. MLP 이외의 다른 종류의 신경망들도 기본적으로 각 노드는 MLP의 노드들과 유사한 연산을 수행하기 때문에 다른 신경망에도 이러한 구조의 연산이 쉽게 적용될 수 있다.

신경망의 각 '노드'에서의 내적 연산은 입력벡터와 연결가중치벡터를 축적함으로써 행렬의 곱연산으로 변환할 수 있다. 이렇게 분리되어 수행되는 노드들에서의 내적 연산들을 하나의 행렬곱으로 해석함으로써 GPU의 병렬구조를 효율적으로 활용하여 신경망을 구현할 수 있다. 그리고 바이어스 항의 덧셈과 시그모이드 연산들은 픽셀세이더로 쉽게 구현가능하다. 이로써 신경망의 각 '층'의 연산은 수식 (3), (4), (5)와 같이 기술할 수 있다. 위의 연산은 행렬연산, 바이어스 항 덧셈연산, 활성함수인 시그모이드 연산의 순으로 수행된다.

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

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

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

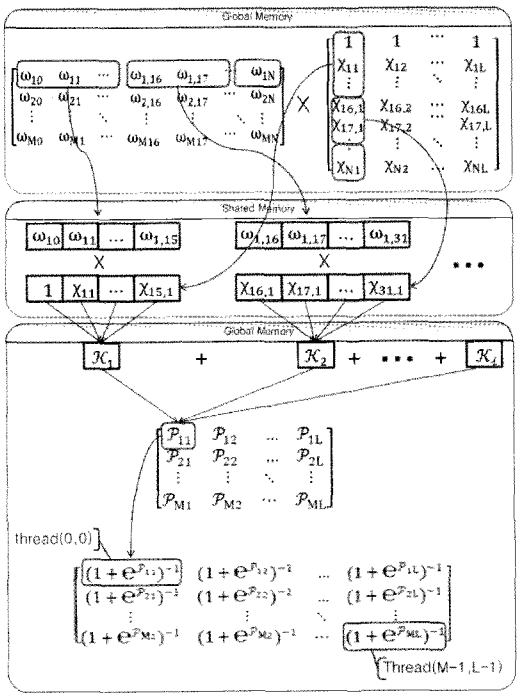


〈그림 8〉 GPU를 이용한 행렬 곱 연산의 개괄도

행렬의 곱을 위해서 Moravanszky[3]에 의해 제안된 방법을 사용한다. <그림 8>은 GPU 내에서의 행렬 곱 연산의 개괄도이다. 두개의 행렬을 texture W와 texture X로 변환하고 렌더링 연산을 수행한다. 행렬 곱의 출력을 위해 전체 화면을 덮는 사각형을 렌더링한다. 정점세이더는 사각형의 각 정점의 위치 외에 텍스처 좌표를 출력하는데, 각각의 정점은 두개의 텍스처 좌표를 가지고 있다. 그 중 하나는 texture W의 행(row) 좌표이고, 또 다른 하나는 texture X의 열(column) 좌표이다. 예를 들어, 왼쪽 상단의 좌표는 texture W의 첫 번째 행의 텍스처 좌표와 texture X의 첫 번째 열 텍스처 좌표를 가지며, 오른쪽 상단의 좌표는 texture W의 첫 번째 행의 텍스처 좌표와 texture X의 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 W의 행과 texture X의 열 사이의 내적연산을 수행한다. 행렬곱의 결과는 texture W×X에 저장된다.

V. CUDA를 이용한 MLP 구현

CUDA(compute unified device architecture)는 C 언어를 기반으로 만들어진 nVIDIA에서 개발한 GPU 프로그래밍 언어이다. 기존에 GPU를 이용하여 수학적인 연산을 하기 위해선 그래픽 함수를 이용한 쉐이딩 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 사전지식 없이 쉽게 GPU를 이용한 일반연산을 구현할 수 있다. 이는 일반적인 수학 연산자를 데이터를 병렬로 처리할 수 있는 GPU 상에서 쉽



〈그림 9〉 CUDA를 이용한 신경망 연산

고 효과적으로 현할 수 있다는 뜻이다. 또한 기존에 쉐이딩 언어를 사용함으로써 발생할 수 있는 오버헤드가 제거됨으로써 더욱 효율적인 연산 수행을 가능하게 한다.

CUDA도 GPU를 이용하는 언어이기 때문에 하드웨어의 구조적인 원인에서 오는 문제점을 가지고 있다. CPU와 GPU간에 메모리를 공유하여 사용할 수 없다는 것이다. 즉, GPU에서 연산을 하기 위해선 메인 메모리로부터 데이터를 전송 받아 연산을 수행하여야 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.

<그림 8>은 CUDA를 이용한 행렬의 곱과 활성화 함수(식 5)를 보여 준다 CUDA에서 큰 특징 중 하나인 공유 메모리 영역을 이용하여 행렬의 곱을 계산함으로써 더 효율적인 연산이 가능하다. 전역 메모리에 접근하는데 대략 400~600 사이클의 메모리 지연이 발생하게 된다. 하지만 공유 메모리를 사용할 때 메모리 지연이 4 사이클 밖에 발생하지 않기 때문에 훨씬 효율적인 연산이 가능하다. sigmoid 연산은 행렬의 크기만큼 쓰레드를 생성하여 각각의 쓰레드에서 위와 같은 수식을 독립적으로 수행함으로써 병렬처리를 수행한다.

V. 결 론

지금까지 GPU를 이용한 신경망의 구현에 대한 여러 사항들에 대해서 알아보았다. GPU는 병렬처리가 가능한 간단한 범용 프로세서로 볼 수 있고 가능하다면 그래픽스 파이프라인을 잘 활용하는 것이 효율성을 높일수 있는 방법이다. 신경망은 내적 연산을 많이 사용하고 다른 연산들도 복잡하지 않아서 GPU로 구현이 용이하다. 다만 GPU의 병렬성을 최대한 활용해서 속도향상을 이루려면 많은 양의 데이터를 입력을 필요로 하는 신경응용분야를 찾아야 한다.

참고문헌

[1] Kyoung-Su Oh, and Keechul Jung, 'GPU implementation of neural networks,' Journal of Pattern Recognition, Volume 37, Issue 6, June 2004, Pages 1311-1314

[2] 장홍훈, 박안진, 정기철, CUDA와 OpenMP를 이용한 신경망 구현, DICTA, 2008.

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

저자소개

오 경 수

1994년 2월 서울대학교 계산통계학과 학사
1996년 2월 서울대학교 전산과학과 학사
2001년 8월 서울대학교 전기컴퓨터공학과 박사
2000년 1월 ~ 2000년 12월 조이멘트 개발팀장

주관심 분야 : 실시간 렌더링, 전역조명 렌더링, 게임개발

정 기 철

1994년 2월 경북대학교 컴퓨터공학과 학사
1996년 2월 경북대학교 컴퓨터공학과 석사
2000년 2월 경북대학교 컴퓨터공학과 박사
2003년 3월 ~ 2009년 5월 숭실대학교 IT대학 미디어학부 부교수
2006년 3월 ~ 2008년 2월 숭실대학교 정보과학대학원 미디어학과 주임교수
2001년 10월 ~ 2002년 10월 Postdoc, PRIP Lab, Department of CSE, Michgan State University

주관심 분야 : HCI, 인공지능, 패턴인식, 영상처리/컴퓨터비젼