2013 IEEE 25th International Conference on Tools with Artificial Intelligence

Optimized Deep Learning Architectures with Fast Matrix Operation Kernels on Parallel Platform

Ying Zhang
Department of Automation
University of Science and Technology of China
Hefei 230027, China
zhy0927@mail.ustc.edu.cn

Saizheng Zhang
Department of Computer Science
Stony Brook University
Stony Brook, NY 11790
saizheng.zhang@stonybrook.edu

***Abstract*—In this paper, we introduce an optimized deep learning architecture with flexible layer structures and fast matrix operation kernels on parallel computing platform (e.g. NVIDIA's GPU). Carefully designed layer-wise strategies are conducted to integrate different kinds of deep architectures into a uniform neural training-testing system. Our fast matrix operation kernels are implemented in deep architecture's propagation processes. In our experiment, these kernels save 70% time on average comparing with the kernels in NVIDIA's CUBLAS library (widely used by many other neural network toolkits), and help our parallel deep architecture beats the neural structures using CUBLAS kernels in practical problems.**

I. Introduction

Recently, notable research has been devoted in fields of deep learning (or deep architectures) which automatically learns features and patterns at multiple levels of abstraction [1, 2, 6]. Proposed methods include restricted boltzmann machine (RBM) [7, 8], stacked denoising autoencoder (SDAE) [9, 12, 17], deep neural network (DNN) (convolutional network (CNN)) [11, 14, 15], deep belief nets (DBN) [3, 6] and combination of these approaches [4, 13, 16, 18]. Deep architecture allows hierarchical unsupervised feature learning from higher level statistics formed by the composition of lower level patterns, and it can be fine-tuned to memory specific object classes in a more abstractive way. Compared with traditional learning techniques starting from hand-crafted feature extraction stage at low levels (e.g. pixel-level in machine visions) and ending in a classifier [19, 20, 21], deep architectures can capture more complicated, hierarchically launched statistical patterns of inputs, hold high level non-linearties, be adaptive to new areas requiring more complicated model descriptions and often outperform state-of-the-art achieved by hand-made features.

Although deep architecture has made successful steps in size-limited problems (CIFAR and STL10 [22, 23]), it is still challenging to scale the input to realistic size. One important reason for this weak scale-adaptability is that the computing cost of implementing computational tractable algorithms for training/testing with large size inputs is extremely high. In recent years, new parallel computing tools and algorithms have been employed in academic fields. Successful examples can be seen from NVIDIA's Graphic Processing Unit (GPU), a processing core with highly parallel structures and CUDA (Compute Unified Device Architecture) [28], a parallel computing architecture for parallel processing on GPUs. With both GPU and CUDA, researchers can access to the virtual instructions set and different memories of parallel computational elements, perform efficient memory management and implement highly parallel computing strategies.

To construct high-speed deep architecture available for large size inputs, the key point is to speed up the layer-wise parameter calculations/updates during propagation process (i.e. in MLP training, the most time consuming parts are forward propagation and backpropagation). It should be pointed out that propagation process can be modeled in matrix forms and the matrix operation can be accelerated by well-designed parallel schemes. There are already some neural network training systems such as QuickNet [29], while its CPU based structure has little parallelism. Recently, several parallel neural network toolkits are released [25, 26]. Most of these GPU based neural network toolkits use CUBLAS libraries (NVIDIA's GPU-accelerated version of the complete standard BLAS library [34]) for their matrix operations. However, we argue that comparing with our kernels, the performance of matrix kernels in CUBLAS is mediocre in specific propagation process during deep architecture training and testing.

In this paper, our primary concern is to construct an efficient and flexible general deep learning architecture on parallel devices. We give a framework of layer-wise parameter holding to uniformly model layers from different kinds of deep architectures. We utilize efficient dataset storage strategy to achieve a fast and flexible data accessing. We create optimized kernel functions for fast matrix operation. During our well-designed experiments, our fast matrix kernels significantly outperform kernels in NVIDIA's CUBLAS library, and our optimized deep learning architecture equipped with fast kernels beats the traditional CPU neural learning structure and structures using CUBLAS library.

The rest of the paper is organized like this: In Section II we discuss the general mathematical model of deep architectures. In Section III we introduce our optimized parallel deep architecture. In Section IV we give details about the fast matrix operation kernels. In Section V we show our experimental results, and finally in Section VI, we draw our conclusion.

1082-3409/13 $31.00 © 2013 IEEE
DOI 10.1109/ICTAI.2013.21

II. Model Description of Deep Architectures

Deep architecture comes from neural network (or multi-layer perceptron). Traditional backpropagation neural network (BP-NN) only has one hidden layer. However, the shallow BP-NN serves as the basic building block of the denoising autoencoder (DAE), convolutional neural network(CNN) and the restricted boltzmann machine (RBM). Furthermore, these architectures share many structural similarities: 1) They all consist of several layers of interconnected neurons in similar structures, see Fig.1 (a). 2) They all adopt a layer-wise building strategy that layers are hierarchically arranged from lower feature space to higher pattern space. Thus, for a general deep network model M_{deep} we have

$$M_{deep} = \{k, \mathbf{F}_{deep}, \mathbf{\Theta}_{deep}\} \tag{1}$$

k is the depth of M_{deep}, $\mathbf{F}_{deep} = \{f_1, ..., f_k\}$ is the mapping function set describing each layer's architecture, $\mathbf{\Theta}_{deep} = \{\theta_1, ..., \theta_k\}$ is the hypoparameter set of $\mathbf{F}_{deep}$. In the rest of the section, we discuss the detail relations of the different architectures mentioned above, and how they are fitted to the general model.

A multi-layer perceptron (MLP) M_{mlp} consists of an input layer L_{in} (L_0), several hidden layers L_{hid}s (L_is) and an output layer L_{out} (L_{end}). Each layer is made up of neurons like Fig.1 (a). Given a M_{mlp} with the depth of k, any f_i in $\mathbf{F}_{mlp}$ is the same $sigmoid$ function $sigm(\cdot)$, and the parameter set $\mathbf{\Theta}_{mlp}$ has $\{\mathbf{W}_i, \mathbf{b}_i, \ \ i = 1, ..., k\}$. Suppose that $\mathbf{x}_i$ and $\mathbf{y}_i$ are the input and output of layer i, the architecture between L_i and L_{i-1} can be modeled as:

$$\mathbf{y}_i = f_i(\mathbf{x}_i, \mathbf{W}_i, \mathbf{b}_i) = sigm(\mathbf{W}_i\mathbf{x}_i + \mathbf{b}_i) \tag{2}$$

Notice that the output $\mathbf{x}_i = \mathbf{y}_{i-1}$, and L_{end} (L_k)'s output $f_k \circ .. \circ f_{1_{\mathbf{\Theta}_{mlp}}}(\mathbf{x})$ becomes the label given by the MLP. To train M_{mlp} on a given dataset $\mathcal{U}$ and labelset $\mathcal{Z}$, we estimate $\mathbf{\Theta_{mlp}}$ by minimizing a cost function E measuring the discrepancy between M_{mlp}'s outputs $f_k \circ ... \circ f_{1_{\mathbf{\Theta}_{mlp}}}(\mathcal{U})$ and the corresponding labels $\mathcal{Z}$,

$$E = \sum_m || \underset{\mathbf{\Theta}_{mlp}}{f_k \circ ... \circ f_1}(\mathbf{u}_m) - \mathbf{z}_m||_2 \tag{3}$$

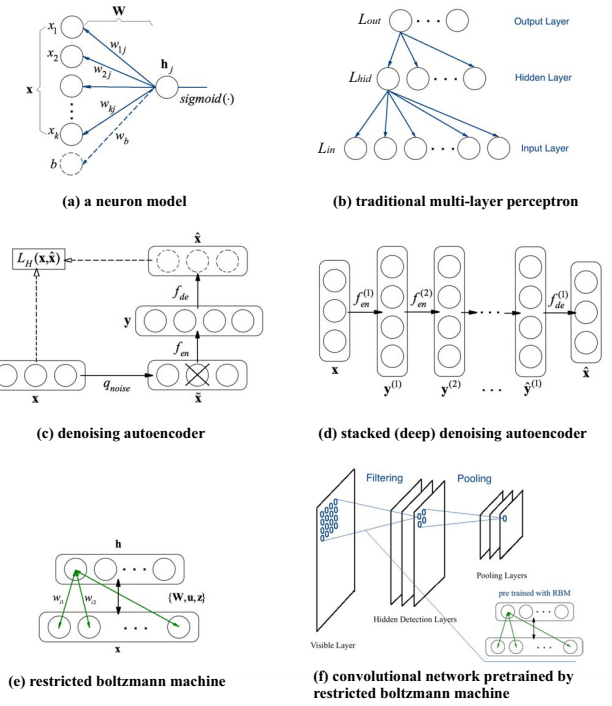
here $\mathbf{u}_m \in \mathcal{U}$ and $\mathbf{z}_m \in \mathcal{Z}$. We use backpropagation for minimization, where we take partial derivatives of E with respect to $\mathbf{W}$ and $\mathbf{b}$ and perform gradient descent [5].

With prior knowledge of MLP, we extend it to denoising autoencoder (DAE) which is an one-hidden-layer MLP added with noises in its input layer. As the basic building block of stacked denoising autoencoder (SDAE), DAE reconstructs the original clean input from its noisy version. Let $\mathbf{x}$ be the original input and $\tilde{\mathbf{x}}$ be the noisy version of $\mathbf{x}$ where $\tilde{\mathbf{x}} = q_{noise}(\mathbf{x})$, a DAE M_{dae} includes the denoising encoder f_{en} and decoder f_{de} ($f_{en}, f_{de} \in \mathbf{F}_{dae}$),

$$\mathbf{y} = f_{en}(\tilde{\mathbf{x}}) = sigm(\mathbf{W}_{en}\tilde{\mathbf{x}} + \mathbf{b}_{en}) \tag{4}$$

$$\hat{\mathbf{x}} = f_{de}(\mathbf{y}) = sigm(\mathbf{W}_{de}\mathbf{y} + \mathbf{b}_{de}) \tag{5}$$

$\hat{\mathbf{x}}$ is the denoising version, $\mathbf{y}$ is the encoded pattern and $\{\mathbf{W}_{en}, \mathbf{b}_{en}, \mathbf{W}_{de}, \mathbf{b}_{de}\}$ becomes the $\mathbf{\Theta}_{dae}$, see Fig.1 (c).



(a) a neuron model

(b) traditional multi-layer perceptron

(c) denoising autoencoder

(d) stacked (deep) denoising autoencoder

(e) restricted boltzmann machine

(f) convolutional network pretrained by restricted boltzmann machine

Fig. 1. Different kinds of deep architectures including multi-layer perceptron, stacked denoising autoencoder, restricted boltzmann machine and deep convolutional network.

SDAE is a hierarchical structure made up of several DAEs in a stacking manner. If a SDAE consists of n DAEs and the kth DAE is made up of $f_{en}^{(k)}$, $f_{de}^{(k)}$, this SDAE's $\mathbf{F}_{sdae}$ can be divided into the encoding part consisting of $f_{en}^{(1)}$ to $f_{en}^{(n)}$ and the decoding part consisting of $f_{en}^{(n)}$ to $f_{de}^{(1)}$, see Fig.1 (d). Considering SDAE's hierarchical characteristics, given an input $\mathbf{x}$ we have:

$$\hat{\mathbf{x}} = f_{de}^{(1)} \circ ... f_{de}^{(n)} \circ f_{en}^{(n)} ... \circ f_{en}^{(1)}(\mathbf{x}) \tag{6}$$

To train these n DAEs, we adopt the similar strategy of [9].

The restricted boltzmann machine (RBM) is a kind of bidirectionally connected network consisting of stochastic processing units, which can also be interpreted as MLP models [7]. The RBM has an input layer $\mathbf{x}$ and a hidden layer $\mathbf{h}$, between which the symmetric connections are described by weights $\mathbf{W}$ and biases $\mathbf{u}$, $\mathbf{z}$. A marginal probability of $\mathbf{x}$ in RBM is defined using an energy model,

$$p(\mathbf{x}) = \sum_{\mathbf{h}} \frac{exp(\mathbf{h}^T\mathbf{W}\mathbf{x} + \mathbf{u}^T\mathbf{x} + \mathbf{z}^T\mathbf{h})}{Z} \tag{7}$$

Z is the partition function and the conditional probabilities of $p(\mathbf{h}|\mathbf{x})$ and $p(\mathbf{x}|\mathbf{h})$ are given as follows:

$$p(\mathbf{h}_i = 1|\mathbf{x}) = sigm(\mathbf{W}_i\mathbf{x} + \mathbf{z}_i) \tag{8}$$

$$p(\mathbf{x}_j = 1|\mathbf{h}) = sigm(\mathbf{W}_j\mathbf{h} + \mathbf{u}_j) \tag{9}$$

here $sigm(\cdot)$ is the $sigmoid$ function. To train a RBM, we use contrastive divergence to estimate the gradient steps of $\mathbf{W}$

(the same to $\mathbf{b}$ and $\mathbf{z}$) [8, 10]:

$$\Delta \mathbf{W}_{ji} = \epsilon \cdot (\langle \mathbf{x}_j \mathbf{h}_i \rangle_{data} - \langle \mathbf{x}_j \mathbf{h}_i \rangle_{recon}) \quad (10)$$

$\langle \mathbf{x}_j \mathbf{h}_i \rangle_{data}$ and $\langle \mathbf{x}_j \mathbf{h}_i \rangle_{recon}$ can be easily obtained from (8) and (9) [10]. Notice that if we arrange several RBMs in stacking manner, we get a deep belief net (DBN).

The Convolutional Network (CN) can be considered as $2D$ version of MLP [14]. In each processing level, it has a filtering layer with several convolutional kernels (a kernel K is described with its weights $\mathbf{W}_K$, bias $\mathbf{b}_K$ and an activation function f_K, similar to MLP) and a pooling layer for subsampling layer-wise features. Some researcher also proposed convolutional RBM (CRBM), which integrated CN's convolutional features and RBM's statistical advantages.

As we can see, these architectures have strong mathematical relations during their building, training and testing stages. MLP, SDAE, CN's convolutional kernels and RBM's layer probability model share the same computational structure where $sigm(\cdot)$ serves as the activation/probability function and the corresponding weight $\mathbf{W}_{\{mlp,dae,rbm,cn\}}$ and biases $\mathbf{b}_{\{mlp,dae,rbm,cn\}}$ are in matrix form. More importantly, all the layer-wise operations can be modeled in matrix form. Therefore, these layer-wise calculations during training and testing can be modeled as matrix operations for computational efficiency.

III. Optimized Parallel Deep Architectures

The mathematical model of deep architecture showed above is the guideline for designing fast computational algorithms and flexible layer-wise structures. Notice that the matrix operations in propagation process of training and testing are available for employing parallel strategies, since the matrix operation can be divide into smaller computing units in parallel [30]. This makes parallel device available to employ its parallel schemes to achieve fast matrix operation speed. Furthermore, layer-wise flexibility requirements are also very important for an uniform layer structure to describe different kind of deep learning architectures. In this section we first discuss parallel device's characteristics (using GPU as an example) that are critical for high parallel deep network training and testing, then we introduce our flexible parallel deep architectures.

A. Parallel device (GPU) and Its Relation to Deep Architectures

The design philosophy of GPU is shaped by demands of performing massive number of floating-point calculations in a high parallel way. A general GPU is organized into series of streaming processors (SP) that share instruction cache and can run thousands of threads per application. GPU programming offers convenient ways to launch parallel kernels in threads and to integrate the memory management both in hosts and devices. At this stage, GPU significantly outperforms CPU when performing matrix operations due to its parallel hardware architecture. Considering of the tradeoff between flexibility and speed, we do not involve the whole propagation process into one GPU kernel function, which is usually a strategy used by some other toolkits.

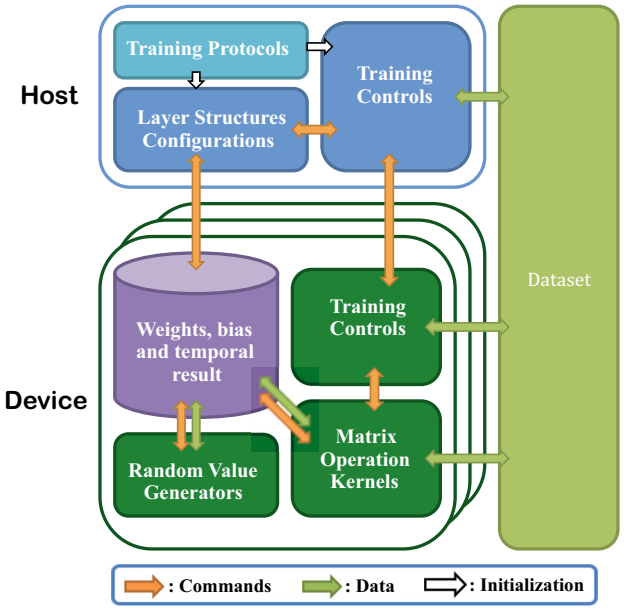


Fig. 2. Our optimized parallel deep learning structure. It is devided into host part for controlling and device part for arranging computing resources.

B. Overview of Our Parallel Deep Learning Architectures

Our parallel deep learning architecture contains both the host stage $\mathbf{H} = \{T_{config}, T_{ctl}^{\mathbf{H}}\}$ and the device stage $\mathbf{E} = \{R, K(\phi), \mathbf{D}, T_{ctl}^{\mathbf{E}}\}$. The matirx operations in different deep architectures can be modeled in an uniform framework under device's parallel hardware structure. This framework includes data holding $\mathbf{D}$ and layer operations K (or ϕ, we show details in the following subsection). Furthermore, the random value generator R used for stochastic operations in RBM/CRBM and parameter initialization are on the device. It is reasonable to hold training control $T_{ctl}^{\mathbf{H}}$ and basic configuration of layer architecture T_{config} on host side, simply due to their very limited time cost. Fast data saving and accessing is necessary for the reason that the deep network retrieves data through the whole dataset in every training and testing epoch. Suppose that we have $f_{kernel} \in K$ and a $g_{rand} \in R$, given $\mathbf{V}_i \in \mathbf{D}$, $i = 1, ..., n$ and its configuration $t_i \in T_{config}$, $i = 1, ..., n$ a basic stage of propagation is modeled as follows:

$$[\mathbf{V}'_1; ...; \mathbf{V}'_m] = f_{kernel}(\mathbf{V}_1, ..., \mathbf{V}_n, \langle T_{ctl}^{\mathbf{H}}, T_{ctl}^{\mathbf{E}} \rangle_{prop}, t_1, ..., t_n) \quad (11)$$

where $\langle \cdot, \cdot \rangle_{prop}$ gives the propagation descriptions of $\mathbf{V}$ based on $T_{ctl}^{\mathbf{H}}$ and $T_{ctl}^{\mathbf{E}}$ (e.g. safety checks, processing orders in the whole propagation). For parameter initialization of $\mathbf{V}_i$, we have:

$$\mathbf{V}_i = g_{rand}(t_i, \langle T_{ctl}^{\mathbf{H}}, T_{ctl}^{\mathbf{E}} \rangle_{rand}, \mathbf{V}_i) \quad (12)$$

where $\langle \cdot, \cdot \rangle_{rand}$ gives the random generating descriptions of $\mathbf{V}$ based on $T_{ctl}^{\mathbf{H}}$ and $T_{ctl}^{\mathbf{E}}$. The random operations in RBM/CRBM

can use the similar strategy, where the $\mathbf{V}_i$ includes input layer $\mathbf{x}$ and hidden layer $\mathbf{h}$ and small changes appear in $\langle\cdot,\cdot\rangle_{rand}$. The whole structure is showed in Fig.2.

C. Flexible Layer Structures

In Section II, we proposed a general mathematical model of deep architecture that $M_{deep} = \{k, \mathbf{F}_{deep}, \mathbf{\Theta}_{deep}\}$. In our optimized matrix based architecture, we map M_{deep} to a matrix based model $G_{deep} = \{\mathcal{D}, \phi\}$:

$$M_{deep} \rightarrow G_{deep} : \{k, \mathbf{\Theta}_{deep}\} \rightarrow \mathcal{D}, \mathbf{F}_{deep} \rightarrow \phi_{prop} \quad (13)$$

where $\mathcal{D}$ is the multi-dimensional vector set storing layer-wise parameters in matrix version, and ϕ is the set of operations over $\mathcal{D}$. ϕ includes all possible operations launched in hosts and devices. As we mentioned before, deep architectures SDAE, DBN, CNN and DNN share a uniform layer structure holding. This uniformity requires specific flexibility such as compacting high-dimensional vectors into low-dimensional ones with all their dimensional and structural information reserved, or accessing low-dimensional sub-matrices of the high-dimensional vectors. A transform function set $\mathcal{T}$ should be established to satisfy these transform demands without losing useful information. In general, for a series of transform functions $\tau_i \in \mathcal{T}$ and k-dimensional vectors $\mathbf{v}_d \in \mathcal{D}_{[d]}$,

$$\begin{gathered}\mathbf{v}_1 = \tau_1(\mathbf{v}_0), \;\; \mathbf{v}_0 \in \mathcal{D}_{[m]}, \mathbf{v}_1 \in \mathcal{D}_{[p_1]} \\ \cdots \\ \mathbf{v}_k = \tau_k(\mathbf{v}_{k-1}), \;\; \mathbf{v}_{k-1} \in \mathcal{D}_{[p_{k-1}]}, \mathbf{v}_k \in \mathcal{D}_{[p_k]}\end{gathered} \quad (14)$$

There should exists an inverse function $\hat{\tau} : \mathcal{D}_{[m]} \rightarrow \mathcal{D}_{[p_k]}$, $\hat{\tau} \in \mathcal{T}$ satisfying the following restriction:

$$\begin{aligned}\mathbf{v}_0 &= \hat{\tau}(\mathbf{v}_k) \\ &= \hat{\tau}(\tau_k(...\tau_1(\mathbf{v}_0)))\end{aligned} \quad (15)$$

Here, we build a flexible class $\mathcal{C}$ and the corresponding memory operation set $\phi_{\mathcal{C}} \subseteq \phi$ to hold all these mathematic characteristics. Firstly, $\phi_{\mathcal{C}}$ has a field $I_{\mathcal{C}}$ with its subfield $I_{\mathcal{C}}\_size$ holding the size information and $I_{\mathcal{C}}\_datalayout$ holding the real data layout information in host and device. When performing transformation and matrix operation, whether this vector should be treated as a $1D$ vector or $2D$ matrix or some high-dimensional forms is decided by $I_{\mathcal{C}}\_datalayout$. But the original information can still be saved safely in $I_{\mathcal{C}}\_size$. Secondly, $\mathcal{C}$ has a field $P_{\mathcal{C}}$ to save all the control knowledge, such as its host/device labels (labels can be changed during training process to achieve more flexible implements) and operation flags which serves as the critical operation control. Thirdly, $\mathcal{C}$ has host and device memory pointers for memory management. For $\phi_{\mathcal{C}}$, it includes the following methods:

1) Multi-dimensional memory allocation approaches based on the vectors $I_{\mathcal{C}}$ field.

2) $I_{\mathcal{C}}$ modification logics for conveniently changing vector structure.

3) Memory release methods.

4) Memory copy methods among devices.

5) Compacted methods for vector dimension reduction.

6) Sub-matrix and sub-slide extraction methods for specific requirements (such as accessing the red channel of an RGB image, or accessing the feature maps in convolutional networks).

D. Dataset Storing and Accessing

The dataset's accessing speed can be the bottleneck of the training and testing performance. In our experiment, a poor data storing strategy could diminish up to 10% of the speed. At least two data storing approaches are available: we can store the data in separated device memory pieces for flexible transformation and accessing, but the time cost could be high. Or we can store the whole dataset in a continuous memory block in device for fast accessing speed, but the flexibility cannot be guaranteed. In our architecture, we achieve a balance between the speed and flexibility. Although we implement only one big memory block for data storing and accessing, but we treat every pattern memory unit (a pattern memory unit could be $1D$ array or a $2D$ sub-block with the padding, which stores only one pattern) as a virtual object of class $\mathcal{C}$. That is, although we put the input pattern into a continuous memory block with low dimension (up to 2 dimension), we can still access it as a high-dimensional vector through correctly setting its subfield $I_{\mathcal{C}\_size}$, and its subfield $I_{\mathcal{C}\_datalayout}$ will match the real data layout in that pattern memory unit. For instance, if we have a vector with the size of $M \times N \times K$, which can be considered as a $K - dimensional$ vector, we will store it in a pattern memory unit whose data layout size is $(M + length\,of\,padding) \times K \times N$, but in its subfield $I_{\mathcal{C}}\_size$, it will be recorded as a three-dimensional vector with the width of M, height of N and depth of K. Notice that all the matrix operations are based on the $I_{\mathcal{C}}\_size$.

IV. Fast Matrix Operation Kernels

Optimized matrix operation kernels contribute a lot to the deep architecture's propagation speed acceleration. So we design new matrix operation algorithms on parallel devices and use GPU as the parallel platform to show our algorithms. NVIDIA has already released the Basic Linear Algebra Subroutines (CUBLAS) library, which is available for researchers to perform high speed parallel computing on their matrix-based programs. However, these kernels are in average slower than our fast matrix operation kernels on specific tasks of deep learning, and under some circumstances (i.e. a large difference exists between the number of rows and the number of columns), the speed difference gap between CUBLASs functions and ours is big. In the following, we will give a short review of CUBLASs library and then present our optimized algorithms for fast matrix operations.

A. CUBLAS Library

The NVIDIA CUDA Basic Linear Algebra Subroutines (CUBLAS) library is a GPU-accelerated version of the complete standard BLAS library that delivers 6 to 17 times faster performance than the latest MKL BLAS [34]. To use CUBLAS

library, the corresponding GPU memory space should be allocated before required matrices and vectors utilization. After the memory allocation, we call the sequence of desired CUBLAS functions, and upload the results from the GPU memory space back to CPU. The CUBLAS kernels implemented during propagation include *Sgemv(CUDA)* and *Sgemv^T(CUDA)*, for vector-matrix multiplication, and *Sger(CUDA)* for vector-vector multiplication.

B. Fast Optimized Matrix Kernels

The idea behind our kernels is that we want to maximize the utilization of the cache memory in each block, see [28] for details about GPU's memory holding. We hope to load the input vector and matrix only once to reduce the time of accessing GPU's global memory. We store all the intermediate results in cache memory which is much faster to access than the global memory. Each block is only used for one row calculation, and we use index-exponential-declination strategy to perform the add operations for calculating the inner productions in the first warp in that block. We utilize M warps ($N \times M$ threads) in each block.

Here we use *gNewGemvf* and *gNewGerf* to illustrate our strategy. *gNewGemvf* is for vector-matrix multiplication, which is for example responsible for calculating every layers output and the partial derivative of the objective function with the respect of layer parameters. Its optimized calculating algorithm is showed in Algorithm 1. Here we have:

$$\mathbf{y} = \mathbf{A} \cdot \mathbf{x} \tag{16}$$

The key point is that in each block we only perform one row calculating. Inside the block j we set $N \times M$ threads and each thread i calculates the multiplication of $\mathbf{x}[i]$ and $\mathbf{A}[j][i]$ in $\mathbf{buff}[i]$. For index i_0 that is greater than $N \times M$, we calculate $\mathbf{x}[i_0] \cdot \mathbf{A}[j][i_0]$ and save the result in $\mathbf{buff}[i_0 \ mod \ N \times M]$.

Algorithm 1 Vector-Matrix Multiplication (in block j)

Ensure: Cache memory of temporal array $\mathbf{buff}[N \times M]$ is allocated.
1: 1. $\mathbf{buff}[i] \leftarrow 0$, where $i = 1, ..., N \times M$
2: 2. **Parallelly do** in each thread i:
3: Load $\mathbf{x}[i]$ and $\mathbf{A}[j][i]$
4: $\mathbf{buff}[i] \leftarrow \mathbf{buff}[i] + \mathbf{x}[i] \cdot \mathbf{A}[j][i]$
5: **if** $i > blocksize(or N \times M)$ **then**
6: **repeat**
7: $\mathbf{buff}[i \ mod \ N \times M] \leftarrow \mathbf{buff}[i \ mod \ N \times M] + \mathbf{x}[i] \cdot \mathbf{A}[j][i]$
8: $i \leftarrow i + N \times M$
9: **until** $i > size(\mathbf{A}[j])$
10: **end if**
11: 3. **Parallelly do** in N threads in the first warp of the block:
12: **if** $i > N$ **then**
13: **repeat**
14: $\mathbf{buff}[i \ mod \ N] \leftarrow \mathbf{buff}[i \ mod \ N] + \mathbf{buff}[i]$
15: **until** $i < N$
16: **end if**
17: $n \leftarrow log_2 N$
18: **repeat**
19: **if** $i \in [2^{n-1}, 2^n]$ **then**
20: $\mathbf{buff}[i \ mod \ 2^{n-1}] \leftarrow \mathbf{buff}[i \ mod \ 2^{n-1}] + \mathbf{buff}[i]$
21: **end if**
22: $n \leftarrow n - 1$
23: **until** $n < 0$
24: 4. $\mathbf{y}[j] \leftarrow \mathbf{buff}[0]$

After we finish the multiplication, we add all buff's elements together in the first warp in an index-exponential-declining way, see step 3 of Algorithm 1.

Algorithm 2 Vector-Vector Multiplication (in block j)

1: **Parallelly do** in each thread i:
2: Load $\mathbf{x}[i]$ and $\mathbf{y}[j]$ and the jth row of A
3: *buff_y* $\leftarrow \mathbf{y}[j]$
4: $\mathbf{A}[j][i] = \mathbf{A}[j][i] + \mathbf{x}[i] \cdot$ *buff_y*;
5: **if** $i > N \times M$ **then**
6: **repeat**
7: $A[j][i \ mod \ N \times M] = A[j][i \ mod \ N \times M] + x[i] \cdot$ *buff_y*
8: $i \leftarrow i + N \times M$
9: **until** $i > size(\mathbf{A}[j])$
10: **end if**

gNewGerf is for vector-vector multiplication, which is for example responsible for calculating the partial derivative of E with the respect of weights W in propagation process. Algorithm 2 shows our optimized strategy for *gNewGerf*, its parallel strategy is similar to what we used in *gNewGemvf*, but without "add" operation like step 3 in Algorithm 1. Here we have:

$$\mathbf{A} = \mathbf{y} \cdot \mathbf{x}^T \tag{17}$$

V. Experimental Results

In this section, we conduct three independent experiments to show the structural and speed improvements gained from our optimized layer architectures and optimized matrix kernels. The first experiment compares the pure speed performance of our matrix kernels with CUBLAS library and CPU based matrix kernels (adopt strategies from QuickNets). The second experiment is performed on *MINST* dataset [27] to evaluate the comprehensive performance of our new GPU based deep learning structure and matrix kernels. The third experiment consider a real problem of face occlusion recognition on ORL and AR databases [31, 32] using stacked denoising autoencoder and deep neural network.

A. Pure Kernel Speed Comparison

In the first experiment, we focus on the pure performance of our kernels without implementing them into deep architecture's propagation process. We have three new created kernels *gNewGemvf*, *gNewGemvf^T* and *gNewGerf* employed for vector-matrix multiplication, vector-matrix multiplication (transposed version) and vector-vector multiplication. Our kernels are compared with CUBLASs kernels *Sgemv(CUDA)*, *Sgemv^T(CUDA)* and *Sger(CUDA)*, and the corresponding kernels in CPU. For *gNewGemvf* and *gNewGemvf^T*, tests are performed on square matrices scaled from 256 to 4096, and on rectangular matrices with the size of $128 \times N$and $256 \times N$, where N ranges from 256 (or 512) to 16384. The vector's size equals to the number of the columns of the matrix. We implement 100000 iterations for each kernel, recorded the total running time and repeat the test to gain an average results. the time saving is evaluate like follows:

$$\alpha_{saving} = \frac{T_{CUBLAS/CPUs} - T_{ours}}{T_{CUBLAS/CPUs}} \times 100\% \tag{18}$$

TABLE I
SPEED COMPARISON OF VECTOR-MATRIX MULTIPLICATION (NORMAL AND TRANSPOSED) (100000 ITERATIONS)

Matrix Size	*gNewGemvf* (sec)	*Sgemv(CUDA)* (sec)	*gemv(CPU)* (sec)	Time Saving α_{saving} %	*gNewGemvf*^{T} (sec)	*Sgemv*^{T}*(CUDA)* (sec)	*gemv*^{T}*(CPU)* (sec)	Time Saving α_{saving} %
256×256	$\mathbf{0.30 \pm 0.01}$	1.88 ± 0.06	10.81 ± 0.37	$+84.0, +97.2$	$\mathbf{0.29 \pm 0.01}$	0.30 ± 0.01	10.78 ± 0.35	$+3.3, +97.3$
512×512	$\mathbf{1.22 \pm 0.08}$	5.97 ± 0.15	42.83 ± 0.38	$+79.6, +97.2$	1.10 ± 0.03	$\mathbf{1.04 \pm 0.05}$	40.54 ± 0.44	$-5.8, +97.4$
1024×1024	$\mathbf{4.36 \pm 0.12}$	12.17 ± 0.11	176.54 ± 1.71	$+64.2, +97.5$	4.36 ± 0.11	$\mathbf{3.45 \pm 0.08}$	175.78 ± 1.65	$-26.4, +97.5$
2048×2048	$\mathbf{13.27 \pm 0.11}$	25.55 ± 0.17	405.01 ± 2.30	$+48.1, +96.7$	14.81 ± 0.18	$\mathbf{12.63 \pm 0.13}$	407.10 ± 3.09	$-17.3, +96.4$
4096×4096	$\mathbf{47.86 \pm 0.23}$	58.52 ± 0.38	1778.52 ± 4.32	$+18.2, +97.3$	51.46 ± 0.30	$\mathbf{48.36 \pm 0.29}$	1790.10 ± 4.39	$-6.4, +97.1$
128×256	$\mathbf{0.30 \pm 0.03}$	1.89 ± 0.28	2.80 ± 0.28	$+84.1, +89.2$	$\mathbf{0.26 \pm 0.02}$	0.29 ± 0.02	3.08 ± 0.11	$+10.3, +91.6$
128×512	$\mathbf{0.53 \pm 0.02}$	4.54 ± 0.08	6.76 ± 0.39	$+88.3, +92.2$	$\mathbf{0.54 \pm 0.03}$	0.64 ± 0.05	6.70 ± 0.35	$+15.6, +91.9$
128×1024	$\mathbf{0.67 \pm 0.04}$	9.81 ± 0.14	13.08 ± 0.24	$+93.2, +94.9$	$\mathbf{0.67 \pm 0.09}$	1.06 ± 0.18	13.38 ± 0.40	$+36.8, +95.0$
128×2048	$\mathbf{1.22 \pm 0.03}$	26.17 ± 0.08	26.22 ± 0.50	$+95.3, +95.3$	$\mathbf{0.95 \pm 0.08}$	2.42 ± 0.21	25.58 ± 1.32	$+60.7, +96.3$
128×4096	$\mathbf{2.04 \pm 0.02}$	26.58 ± 0.12	54.09 ± 1.01	$+92.3, +96.2$	$\mathbf{1.54 \pm 0.09}$	4.60 ± 0.33	56.80 ± 1.21	$+66.5, +97.3$
128×8192	$\mathbf{3.58 \pm 0.05}$	51.80 ± 0.20	110.81 ± 1.30	$+93.1, +96.8$	$\mathbf{2.66 \pm 0.16}$	8.78 ± 0.37	111.04 ± 2.30	$+69.7, +97.6$
128×16384	$\mathbf{6.52 \pm 0.09}$	53.30 ± 0.28	214.30 ± 2.19	$+87.8, +97.0$	$\mathbf{5.02 \pm 0.20}$	17.22 ± 0.31	216.32 ± 2.70	$+70.8, +97.7$
256×512	$\mathbf{0.53 \pm 0.02}$	4.54 ± 0.08	10.31 ± 0.28	$+88.3, +94.9$	$\mathbf{0.51 \pm 0.02}$	0.86 ± 0.04	29.71 ± 0.28	$+40.7, +94.7$
256×1024	$\mathbf{0.67 \pm 0.04}$	9.81 ± 0.14	27.16 ± 0.60	$+93.2, +97.6$	$\mathbf{0.59 \pm 0.02}$	$\mathbf{0.59 \pm 0.01}$	28.65 ± 0.33	$0.0, +97.9$
256×2048	$\mathbf{1.22 \pm 0.03}$	26.17 ± 0.08	58.29 ± 0.83	$+95.3, +97.9$	$\mathbf{1.18 \pm 0.08}$	1.70 ± 0.21	57.07 ± 0.75	$+30.5, +97.9$
256×4096	$\mathbf{2.04 \pm 0.02}$	26.58 ± 0.12	96.20 ± 1.80	$+92.3, +97.9$	$\mathbf{2.24 \pm 0.07}$	5.33 ± 0.29	94.19 ± 1.37	$+58.0, +97.6$
256×8192	$\mathbf{3.58 \pm 0.05}$	51.80 ± 0.20	200.01 ± 2.57	$+93.1, +98.2$	$\mathbf{3.39 \pm 0.10}$	8.98 ± 0.41	205.88 ± 3.10	$+62.2, +98.4$
256×16384	$\mathbf{6.52 \pm 0.09}$	53.30 ± 0.28	399.35 ± 3.89	$+87.8, +98.4$	$\mathbf{6.28 \pm 0.22}$	18.30 ± 0.41	409.55 ± 5.61	$+65.7, +98.5$

TABLE II
SPEED COMPARISON OF VECTOR-VECTOR MULTIPLICATION (100000 ITERATIONS)

Vector Size	*gNewGerf* (sec)	*Sger(CUDA)* (sec)	*ger(CPU)* (sec)	Time Saving α_{saving} %
$256, 256$	$\mathbf{0.43 \pm 0.03}$	0.46 ± 0.04	5.61 ± 0.11	$+6.5, +92.3$
$512, 512$	$\mathbf{1.75 \pm 0.05}$	1.81 ± 0.08	21.88 ± 0.37	$+3.3, +92.0$
$1024, 1024$	$\mathbf{5.70 \pm 0.29}$	5.91 ± 0.23	87.71 ± 2.21	$+3.6, +93.5$
$2048, 2048$	$\mathbf{21.48 \pm 0.33}$	22.05 ± 0.34	159.03 ± 2.80	$+2.6, +86.5$
$4096, 4096$	$\mathbf{47.81 \pm 0.97}$	49.30 ± 1.14	627.16 ± 6.11	$+3.0, +92.4$
$128, 256$	$\mathbf{0.19 \pm 0.01}$	0.23 ± 0.01	3.04 ± 0.08	$+17.4, +93.8$
$128, 512$	$\mathbf{0.41 \pm 0.03}$	0.47 ± 0.02	5.67 ± 0.23	$+12.8, +92.8$
$128, 1024$	$\mathbf{0.58 \pm 0.02}$	0.78 ± 0.04	10.92 ± 0.43	$+25.6, +94.7$
$128, 2048$	$\mathbf{1.62 \pm 0.09}$	1.85 ± 0.12	21.73 ± 0.49	$+12.4, +92.5$
$128, 4096$	$\mathbf{3.04 \pm 0.18}$	3.30 ± 0.21	44.38 ± 0.91	$+7.9, +93.1$
$128, 8192$	$\mathbf{6.17 \pm 0.15}$	6.82 ± 0.20	84.51 ± 1.28	$+9.5, +92.7$
$256, 512$	$\mathbf{0.54 \pm 0.02}$	$\mathbf{0.54 \pm 0.04}$	11.09 ± 0.21	$0.0, +95.1$
$256, 1024$	$\mathbf{1.13 \pm 0.03}$	1.28 ± 0.06	21.57 ± 0.76	$+11.7, +94.8$
$256, 2048$	$\mathbf{2.67 \pm 0.05}$	2.98 ± 0.11	43.71 ± 1.04	$+10.4, +93.9$
$256, 4096$	$\mathbf{5.78 \pm 0.20}$	6.16 ± 0.31	85.65 ± 1.82	$+6.17, +93.3$
$256, 8192$	$\mathbf{10.49 \pm 0.28}$	11.83 ± 0.34	157.26 ± 2.68	$+11.3, +93.3$

Notice that this running time includes the time for safety checks performed on CPU side and the time for kernel launching. The results are showed in TABLE I. we can see that our *gNewGemvf* achieves great outperformance in both square matrix operation and rectangular matrix operation compared with CUBLAS kernels and CPU version. The average time saving is about $+77.7\%$ and $+96.2\%$ respectively. *gNewGemvf*^{T} is a little bit slower than *Sgemv*^{T}*(CUDA)* on square matrix-vector calculation. However, under the condition that the gap between number of column and the number of row is big, *gNewGemvf*^{T} becomes much faster than *Sgemv*^{T}*(CUDA)*. The average time saving is about $+29.7\%$ and $+96.7\%$ respectively. For *gNewGerf*, the result is showed in TABLE II. Tests are performed on different sizes of vectors scaled from 128 to 8192. From TABLE II we can find that the average time saving gained from *gNewGerf* compared with *Sger(CUDA)* and *ger(CPU)* is $+9.0\%$ and $+92.9\%$ respectively.

B. Performance Comparison on MNIST Dataset

The second experiment compares the propagation speed differences between MLPs using our kernels and ones using CUBLAS/CPU kernels. Test is performed on MNIST handwritten digit dataset, which consists of 60000 grey scale images of handwritten numbers from 0 to 9 with the pixel size of $28 \times 28 = 784$. Single hidden layer neural networks with the structure of $\{784, N, 10\}$ are evaluated, where N ranges from 10 to 1280. The whole training time includes the running time of CPU codes, this is for a better evaluation of the integrated performance of the hosts and devices. We divide the experiment into two phases. First we evaluate the time cost of the entire training epoch that includes both forward propagation and backpropagation with global learning rate and without momentum scheme for a better evaluation of the matrix operation speed. The result is showed in the left of Fig. 3. It is clear that MLP equipped with our kernels achieve an average $+200\%$ faster speed than ones using CUBLAS/CPU kernels. Second we consider only the forward propagation

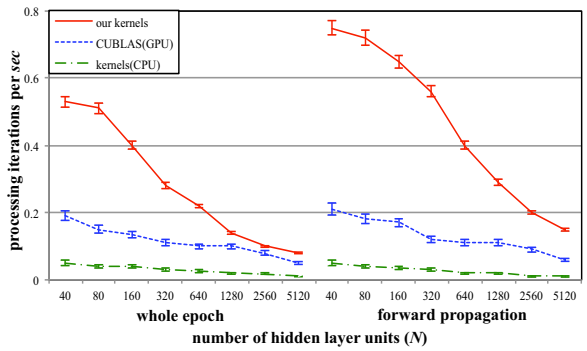


Fig. 3. We compare our kernels with kernels in CUBLAS/CPU on *MNIST* dataset. Experiments are performed on both the whole training process and forward propagation process.

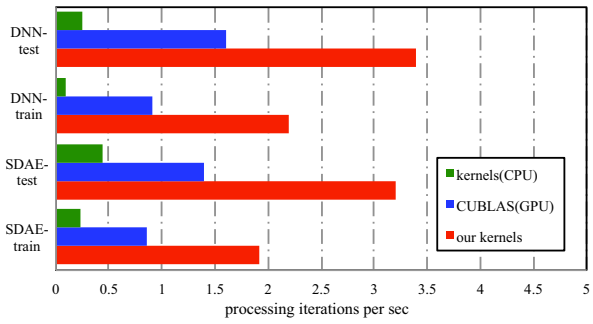


Fig. 4. We test our optimized architecture equipped with our fast matrix kernels on real face recognition problems, this result is on ORL database. The recognition process is divided into restoration part using SDAE and recognition part using DDN. Three kinds of kernels are compared.

process, which purely consists of our kernels. The result is showed in the right side of Fig. 3. In this case, our kernels gain at least $300+\%$ outperformance in most N.

C. Comprehensive Evaluation on ORL/AR face databases

The third experiment considers a practical problem of occluded face recognition using deep learning. The recognition architecture consists of a SDAE for occluded regions restoration (we treat the occluded regions as noises, similar strategy is showed in [33]) and a DNN (pretrained using RBMs) for recognition. Real-size images (ranging in hundreds pixels) are first go through the SDAE trained using clean face images to recover themselves. Then recovered images are sent to the DNN for final recognition. Experiment is performed on ORL and AR face databases.

1) ORL Face Database: The ORL face database consists of 400 grayscale face images of 40 people with the size of 92×112 pixels. These faces are in very limited facial expression changes. Compare with AR face database, its image size is smaller and the amount of images is also relatively small. There is no occluded face in the original dataset, we manually add mask noise on it, like [9].

2) AR Face Database: The AR face database contains more than 4000 face images corresponding to 126 individuals with different facial expressions, illumination conditions and occlusions (sunglasses and scarves). There are 26 pictures taken in two different sessions for each individual, and 14 of them are clean faces. We use the cropped version of AR database which contains only the face areas with the size of 120×165.

Notice that with image resizing, the image sizes of these two databases are much closer to reality than image sizes of traditional MLP training datasets. We separately evaluate the performance of deep architectures using our/CUBLAS/CPU kernels in restoration stage (SDAE) and recognition stage (DNN). In each stage, both the time of training process and the time of testing process are considered. The SDAE has the structure of $\{N, p_1N, p_2N, p_3N, L\}$, N is the number of input units, here it equals to 92×110 or 120×165. $p_iN,\ i=1,2,3$

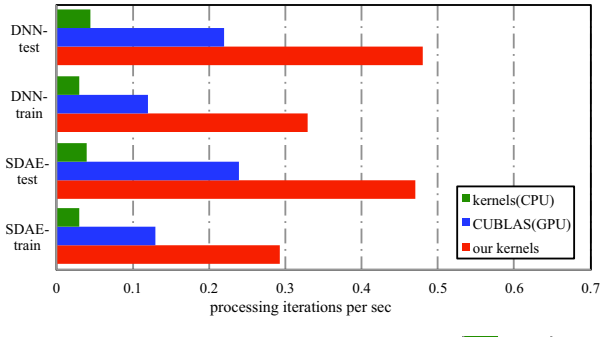


Fig. 5. The result on AR database. As we can see, the iterations per sec is less than ORL database, this is due to its larger amount of training data (4000+ images comparing with 400 images in ORL database).

are the sizes of hidden layers, here we set $p_i \approx 0.8$ for $i=1,2,3$ to gain a reasonable recognition rate (around 90% with the $\sim 30\%$ occlusion level) comparing with state-of-the-art methods. DNN simulates the same layer structure of SDAE and uses SDAE's layer-wise parameters for weights initialization. During propagation process of SDAE/DNN's training and testing, all the available matrix operations are replaced by our/CUBLAS/CPU kernels. Our optimized deep structure is used together with our fast kernels only. Results are showed in Fig. 4 and Fig. 5. As we can see, our optimized deep architectures equipped with our fast kernels continue to achieve notable outperformance comparing with deep architectures with CUBLAS/CPU kernels. The average speed up is around 100% on both ORL and AR databases comparing with architectures using CUBLAS kernels.

VI. Conclusion

In this paper, we presented an optimized deep learning architecture with flexible data/layer structures and fast parallel matrix operation kernels. The experimental results denote that our kernels achieve significant speed outperformance compared with CUBLAS/CPU kernels. In real problem solving

such as digit and face recognition, our optimized architecture equipped with own-created kernels gains better comprehensive performance than learning schemes using CUBLAS or CPU kernels. This suggests that parallel device's better speed adaptability on specific tasks could be achieved with carefully designed kernel strategies.

REFERENCES

[1] Y. Bengio, "Learning Deep Architectures for AI", *Foundations and Trends in Machine Learning*, vol.2(1), pp. 1-127 2009.

[2] D. Erhan, Y. Bengio, A. Courville, P.A. Manzagol, P. Vincent, and S. Bengio, "Why does unsupervised pre-training help deep learning?", *The Journal of Machine Learning Research*, vol. 11, pp. 625-660, 2010.

[3] Y. Bengio, P. Lamblin, D. Popovici, and H. Larochelle, "Greedy layer-wise training of deep networks", in *NIPS*, 2007.

[4] Y.-L. Boureau, F. R. Bach, Y. LeCun, and J. Ponce,"Learning mid-level features for recognition", in *CVPR*, 2010.

[5] Y. LeCun, L. Bottou, G. Orr, and K. Muller, "Efficient BackProp", *Neural Networks: Tricks of the Trade*, vol. 1524, pp. 9-50, 1998.

[6] G.E. Hinton, and R.R. Salakhutdinov, "Reducing the Dimensionality of Data with Neural Networks", *Science*, vol. 313, pp. 504-507, 2006.

[7] D.H. Ackley, G.E. Hinton, and T.J. Sejnoeski, "A Learning Algorithm for Boltzmann Machines", *Cognitive Science*, vol. 9, pp. 147-169, 1985.

[8] G.E. Hinton, "A Practical Guide to Training Restricted Boltzmann Machines", *Neural Networks: Tricks of the Trade*, vol. 7700, pp. 599-619.

[9] P. Vincent, H. Larochelle, I. Lajoie, Y. Bengio, and P. Manzagol, "Stacked Denoising Autoencoders: Learning Useful Representations in a Deep Network with a Local Denoising Criterion", *Journal of Machine Learning Research*, vol. 11, pp. 3371-3408, 2010.

[10] G.E. Hinton, S. Osindero, and Y. Teh, "A fast learning algorithm for deep belief nets", *Neural Computation*, vol. 18(7), pp. 1527-1554, 2006.

[11] G.B. Huang, H. Lee, and E. Learned-Miller, "Learning Hierarchical Representations for Face Verification with Convolutional Deep Belief Networks", in *CVPR*, 2012.

[12] P. Vincent, "A connection between score matching and denoising autoencoders." *Neural computation*, vol.23.7, pp. 1661-1674, 2011.

[13] P. Luo, X.G. Wang, and X.O. Tang, "Hierarchical Face Parsing via Deep Learning", in *CVPR*, 2012.

[14] H. Lee, R. Grosse, R. Ranganath, and A. Y. Ng, "Convolutional deep belief networks for scalable unsupervised learning of hierarchical representations", in *ICML*, 2009.

[15] H. Lee, R. Grosse, R. Ranganath, and A. Y. Ng, "Unsupervised learning of hierarchical representations with convolutional deep belief networks", *Communications of the ACM*, vol. 54(10), pp. 95-103, 2011.

[16] Q. V. Le, M. Ranzato, R. Monga, K. Chen, M. Devin, G. S. Corrado, J. Dean, and A. Y. Ng, "Building high-level features using large scale unsupervised learning", in *ICML*, 2012.

[17] P. Vincent, H. Larochelle, Y. Bengio, and P. Manzagol, "Extracting and composing robust features with denoising autoencoders", in *ICML*, 2008.

[18] D. C. Ciresan, U. Meier, L. M. Gambardella, and J. Schmidhuber, "Deep big simple neural nets for handwritten digit recognition", *Neural Computation*, vol. 22(12), pp. 3207-3220, 2010.

[19] J. Yang, K. Yu, Y. Gong, and T. S. Huang, "Linear spatial pyramid matching using sparse coding for image classification", in *CVPR*, 2009.

[20] D. G. Lowe, "Distinctive image features from scale-invariant keypoints", *IJCV*, vol. 60(2), pp. 91-110, 2004.

[21] J. Wright, A. Y. Yang, A. Ganesh, S. S. Sastry, and Y. Ma, "Robust Face Recognition via Sparse Representation", *TPAMI*, vol. 31(2), pp. 210–227, 2008.

[22] Alex Krizhevsky, "Learning Multiple Layers of Features from Tiny Images", *Technical Report*, 2009.

[23] A. Coates, H. Lee, and A. Y. Ng, "An Analysis of Single Layer Networks in Unsupervised Feature Learning", *AISTATS*, 2011.

[24] G. E. Hinton, "Training products of experts by minimizing contrastive divergence", *Neural Computation*, vol. 14(8), pp. 1711-1800, 2002.

[25] D. Strigl, K. Kofler and S. Podlipnig, "Performance and Scalability of GPU-based Convolutional Neural Networks", in *Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)"*, 2010.

[26] X. Sierra-Canto, F. Madera-Ramirez and V. Uc-Cetina, "Parallel Training of a Backpropagation Neural Network Using CUDA", *International Conference on Machine Learning and Applications (ICMLA)*, 2010.

[27] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. "Gradient-based learning applied to document recognition", *Proceedings of the IEEE*, vol. 86(11), pp. 2278–2324, 1998.

[28] http://www.nvidia.com/object/cuda_home_new.html.

[29] http://www1.icsi.berkeley.edu/Speech/qn.html.

[30] J. Gunnels, C. Lin, G. Morrow, and R. van de Geijn, "A flexible class of parallel matrix multiplication algorithms", *the 1st Merged International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing*, pp. 110-116, 1998.

[31] F. Samaria and A. Harter, "Parameterisation of a stochastic model for human face identification", *2nd IEEE Workshop on Applications of Computer Vision*, 1994.

[32] A. Martinez, R. Benavente, "The AR face database", CVC Tech. Report 24, 1998.

[33] J.Y. Xie, L.L. Xu, and E.H. Chen,"Image Denoising and Inpainting with Deep Neural Networks", in *NIPS*, 2012.

[34] https://developer.nvidia.com/cublas.