High Performance Computing

Yoshifumi Motoyama
Tokyo Institute of Technology
Dept. of mathematical and computing sciences
Matsuoka Lab.

Review Paper

- Optimized Deep Learning Architectures with Fast Matrix Operation Kernels on Parallel Platform
  - Ying Zhang
    - University of Science and Technology of China
  - Saizheng Zhang
    - Stony Brook University

Tools with Artificial Intelligence (ICTAI), 2013 IEEE 25th International Conference on <http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=6734837>

outline

1. Introduction
2. Model description of deep architectures
3. Optimized parallel deep architectures
4. Fast matrix operation kernels
5. Experimental results
6. Conclusion

- Comments

1. Introduction

- Recently, notable research has been devoted in fields of deep learning.
- Deep architecture allows hierarchical unsupervised feature learning from higher level statistics formed by the composition of lower level patterns, and it can be fine-turned to memory specific object classes in a more abstractive way.

1. Introduction

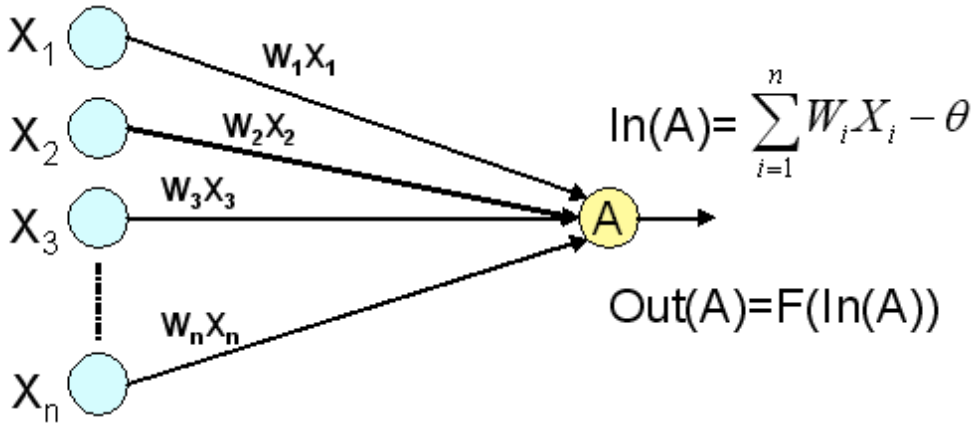
The author's primary concern is to construct an efficient and flexible general deep learning architecture on parallel devices.

2. Model description of deep architectures

Deep architecture comes from neural network (or multi-layer perceptron).

NN(Neural Network)

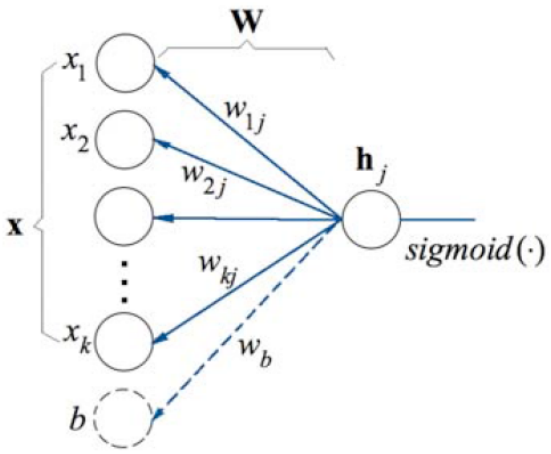
NN(Neural Network) is a biologically-inspired programming paradigm which enables a computer to learn observational data.



BP-NN

BP-NN(BackPropagation Neural Network) only has one hidden layer.

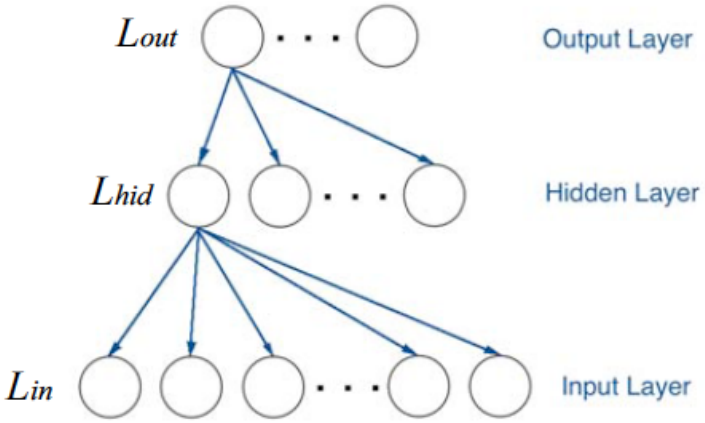
The shallow BP-NN serves as the basic building block of the denoising autoencoder(DAE), conventional neural network(CNN) and the restricted boltzman machine(RNM)



MLP

MLP(Multi-Layer Perceptron) consists of

- input layer $L_{in}(L_0)$
- several hidden layers $L_{hid}s(L_is)$
- Output layer $L_{out}(L_{end})$



MLP

Given a M_{mlp} with the depth of k, any f_i in F_{mlp} is the same sigmoid function sigm(·), and the parameter set Θ_{mlp} has $\{W_i, b_i, i = 1, \cdots, k\}$.

Suppose that x_i and y_i, are input and output of layer i, the architecture between L_i and L_{i-1} can be modeled as:

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

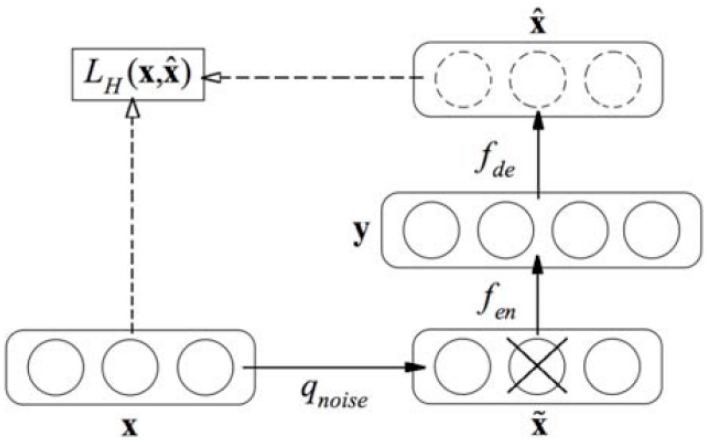
To train M_{mlp}, estimate Θ_{mlp} by minimizing a cost function E measuring the discrepancy between M_{mlp}'s outputs $f_k \circ \ldots \circ f_1{}_{\mathbf{\Theta}_{mlp}}(\mathcal{U})$ and corresponding labels Z,

$$E = \sum_m || f_k \circ \ldots \circ f_1(\mathbf{u}_m) - \mathbf{z}_m||_2$$

DAE

DAE(Denoising AutoEncoder) is an one-hidden-layer MLP added with noises in its input layer.

DAE reconstructs the original clean input from its noisy version.



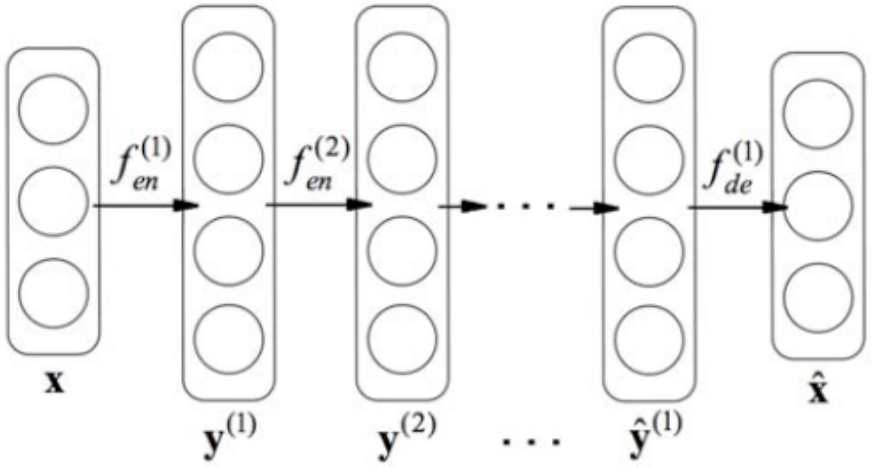
Let x be the original input and $\tilde{\mathbf{x}}$ be the noisy version of x where $\tilde{\mathbf{x}} = q_{noise}(\mathbf{x})$, a DAE M_{dae} includes the denoising encoder f_{en} and decoder $f_{de,}$

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

$\hat{\mathbf{x}}$ is the denoising version

SDAE

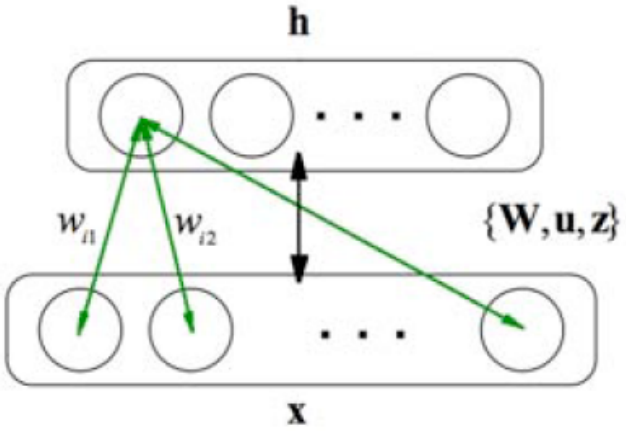
SDAE(Stacked Denoising AutoEncoder) is a hierarchical structure made of several DAEs in stacking manner.



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

RBM

RBM(Restricted Boltzmann Machine) is a kind of bidirectionally connected network consisting of stochastic processing units.



The RBM has an input layer x and a hidden layer h, between which the symmetric connections are described by weights W and biases u, z.

RBM

A marginal probability of x in RBM is defined using an energy model :

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

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

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

To train a RBM, they use contrastive divergence to estimate the gradient step of W:

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

3. Optimized parallel deep architectures

The matrix operations in propagation process of training and testing are available for employing parallel strategy.

- The matrix operation can be divide into smaller computing units.

Overview of parallel learning architecture

In this paper, Parallel deep learning architecture contains both the host stage $\mathbf{H} = \{T_{config}, T_{ctl}^{\mathbf{H}}\}$ and the device stage E = $\{R, K(\phi), \mathbf{D}, T_{ctl}^{\mathbf{E}}\}$.

- The matrix operations in different deep architectures can be modeled in an uniform framework. This framework includes

- data holding D
- layer operations K
- the random value generator R for stochastic operations in RBM/CRBM

$T_{ctl}^{\mathbf{H}}$: training control

T_{config} : basic configuration of layer architecture

Suppose that :

$f_{kernel} \in K$, $g_{rand} \in R$, given $V_i \in D$, $i = 1, \cdots, n$

A basic stage of propagation is modeled as follow :

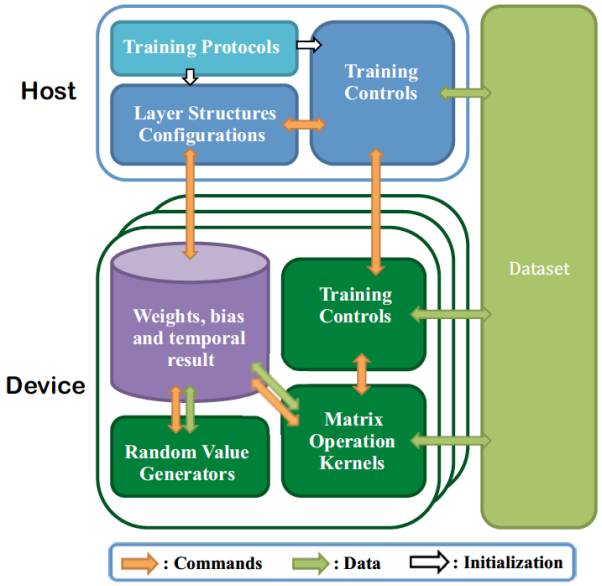
$$[\mathbf{V}'_1; ...; \mathbf{V}'_m] = f_{kernel}(\mathbf{V}_1, ..., \mathbf{V}_n, \langle T_{ctl}^{\mathbf{H}}, T_{ctl}^{\mathbf{E}} \rangle_{prop}, t_1, ..., t_n)$$

where $<\cdot, \cdot>_{prop}$ gives the propagation descriptions of V based on $T_{ctl}^{\mathbf{H}}$ and $T_{ctl}^{\mathbf{E}}$.

For parameter initialization of V_i :

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

The whole structure is as follows :



Flexible Layer Structures

In their optimized matrix based architecture, they map M_{deep} to a matrix based model $G_{deep} = \{D, \Phi\}$:

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

where D is the multi-dimensional vector set storing layer-wise parameters in matrix version, and Φ is the set of operations over D.

Φ includes all possible operations launched in hosts and devices.

Data storing approach

The dataset's accessing speed can be the bottleneck of the training and testing performance.

Two data storing approaches

i. store the data in separated device memory pieces for flexible transformation and accessing. But time cost could be high.
ii. store the whole dataset in a continuous memory block in device for fast accessing speed. But flexibility cannot be guaranteed.

-> balance between the speed and flexibility.

4. Fast matrix operation kernels

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.

gNewGemvf

gNewGemvf is for vector-matrix multiplication.

- calculating every layers output

- the partial derivative of objective function with the respect of layer parameters

The key is that each block they only perform one row calculating.

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. $\mathbf{y}[j] \leftarrow \mathbf{buff}[0]$

gNewGerf

gNewGerf is for vector-vector multiplication.

- calculating the partial derivative of E with the respect of weights W in propagation process

without add operation

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**

5. Experimental results

i. Compares the pure speed performance of their matrix kernels with CUBLAS library and CPU based matrix kernels.
ii. Performed on MNIST dataset to evaluate the comprehensive performance of their new GPU based matrix kernels.
iii. Consider a real problem of face occlusion recognition on ORL/AR databases using SDAE and DNN.

A. Pure Kernel Speed Comparison

First, they focus on the pure performance of their kernels without implementing them into deep architecture's propagation process.

Tests are performed on square matrices from 256 to 4096, and on rectangular matrices with the size of 128 × N and 256 × N, where N ranges from 256 (or 512) to 16384.

The time saving evaluate like follows:

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

Result

- Vector-Matrix multiplication

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

gNewGemvf achieves the average time saving about +77.7% and +96.2% respectively.

gNewGemvf^T achieves the average time saving about +29.7% and +96.7% respectively.

- Vector-vector multiplication

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

The average time saving is about +9.0% and 92.9% respectively.

B. Performance Comparison on MNIST Dataset

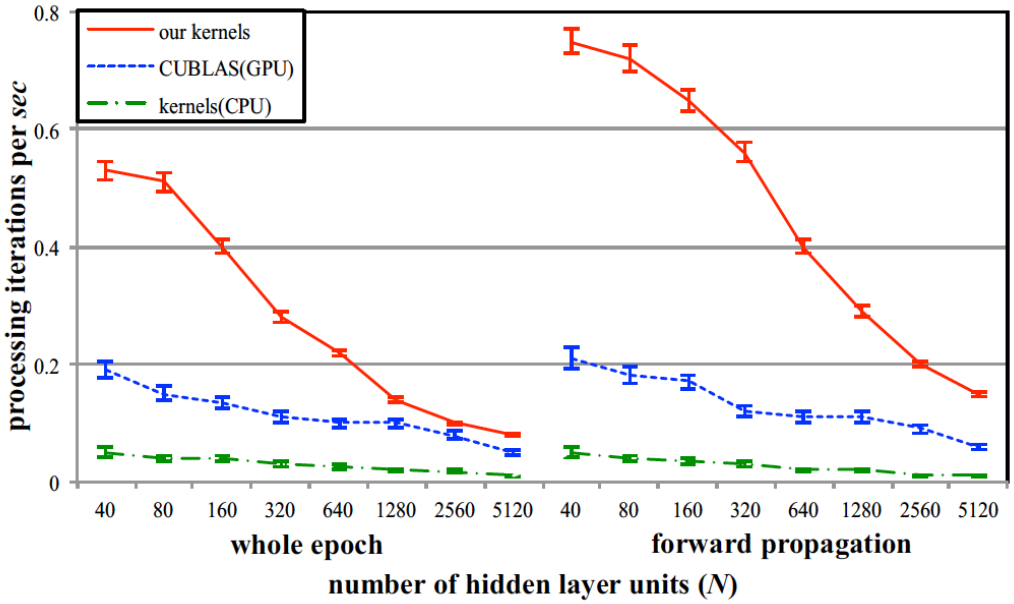
The second experiment compares the propagation speed differences between MLPs using CUBLAS/CPU kernels.

i. evaluate the time cost of the entire training epoch that includes both forward and back propagation.
ii. consider only the forward propagation process, which purely consists of their kernels.

MNIST Dataset

MNIST handwritten digit dataset, which consists of 60000 grey scale image of handwritten numbers from 0 to 9 with the pixel size of 28 × 28 = 784.

Result



Achieve an average + 200% faster speed than CUBLAS/CPU kernels

Their kernels gain at 300 + % outperformance

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 and DNN for recognition.
- Real size images 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.

C. Comprehensive Evaluation on ORL/ AR face databases

- ORL Face Database:
- consists of 400 grayscale face image of 40 people with the size of 92 × 112 pixels.
- very limited facial expression changes.
- no occluded face in the original dataset.

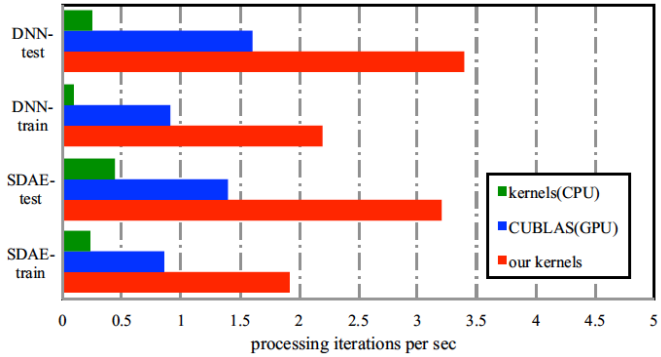
  -> manually add mask noise on it.

- AR Face Database:
- contains more than 4000 face images (= 126 individuals) with different facial expressions, illumination conditions and occlusions (sunglasses and scarves)
- 26 pictures taken in two different sessions for each individual, and 14 of them are clean faces.

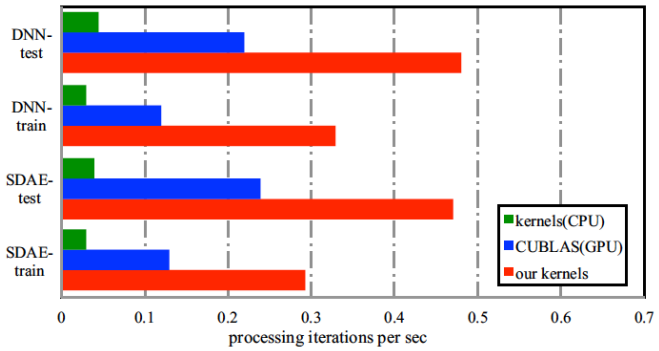
-> use the cropped version which contains only face areas with the size of 120 × 165.

Result on ORL/AR database

- ORL Database



- AR Database



The average speed up is around 100% on both ORL/AR database comparing with using CUBLAS kernels.

6. Conclusion

- The experimental results denote that their kernels achieve significant speed outperformance compared with CUBLAS/CPU kernels.
- Parallel device's better speed adaptability on specific tasks could be achieved with carefully designed kernel strategies.