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

- Comment

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

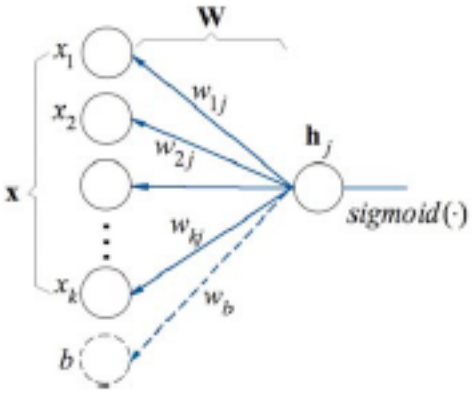
- The author's primary concern is to construct an efficient and flexible general deep learning architecture on parallel devices.

2. Model description of deep architectures

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

BP-NN

- Traditional backpropagation neural network（BP-NN）only has one hidden layer.



(a) a neuron model

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)

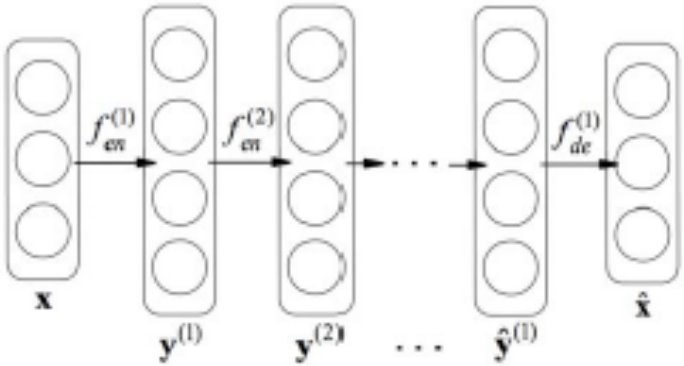
MLP

- A multi-layer perceptron（MLP） M_{mlp} consists of

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

SDAE

SDAE is a hierarchical structure made up of several DAEs in a stacking manner.



(d) stacked (deep) denoising autoencoder

RBM

The restricted boltzman machine（RBM） is a kind of bidirectionally connected network considering of stochastic units.

- Input layer x
- Hidden layer h

between which the symmetric connections are described by weights W and bias 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:

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

RBM

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

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

CN

The Convolutional Network（CN）can be considered as 2D version of MLP.

- In each processing level, it has filtering layer with several convolutional kernels

- A kernel K is described with its weights W_k, bias b_k, and activation function f_k

3. Optimized parallel deep architectures

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

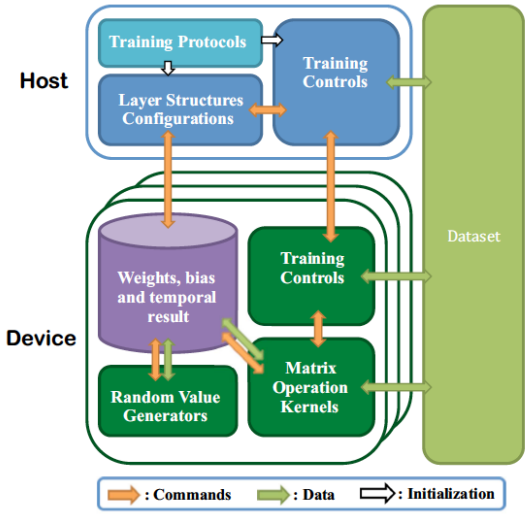
  - The matrix operation can be divide into smaller computing units.

- Layer-wise flexibility requirements are very important for an uniform layer structure to describe different kind of deep learning architecture.

A. Parallel device(GPU) and its relation to deep architectures

Optimized parallel deep learning structure

GPU is organized into series of streaming processors（SP）that share instruction cache and can run thousands of threads per app.



B. Overview of our parallel learning architecture

Parallel deep learning architectures

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

This framework includes

- Data holding D
- Layer operations K

Parallel deep learning architectures

A basic stage of propagation is modeled as follow:

$$[\mathbf{V}'_1; ...; \mathbf{V}'_m] = f_{kernel}(\mathbf{V}_1, ..., \mathbf{V}_n, \langle T^{\mathbf{H}}_{ctl}, T^{\mathbf{E}}_{ctl} \rangle_{prop}, t_1, ..., t_n)$$

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

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

C. Flexible Layer Structures

Optimized matrix based architecture

In author's optimized matrix based architecture, they map M_{deep} to a matrix based model $G_{deep} = \{D, \phi\}$:

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

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.

D. Dataset Storing and Accessing

Data storing approach

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

  - poor data storing strategy could diminish up to 10% of the speed.

Data storing approach

- Two data storing approaches are available

- store the data in separated device memory

for flexible transformation and accessing, but the time cost is high.

- store the whole dataset in a continuous memory block in device for accessing speed, but the flexibility cannot be guaranteed.

=> The author achieve a balance between the speed and flexibility.

4. Fast matrix operation kernels

Optimized matrix operation kernels contribute a lot to the deep architecture.

- design new matrix operation algorithms on parallel devices and use GPU as the platform

A. CUBLAS Library

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.

B. Fast optimized matrix kernels

Fast optimized matrix kernels

- The idea behind their kernels is that they want to maximize the utilization of the cache memory in each block.

- Each block is only used for one row calculation

gNewGemvf

gNewGermvf is for vector-matrix multiplication, which is for example responsible for calculating every layers output and 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, which is for example responsible for calculating the partial derivative of E with the respect of weights W in propagation process.

Without add operation, like step 3 in 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**

5. Experimental results

They conducted three independent experiments to show the structural and speed improvements gained from their optimized layer architectures and optimized matrix kernels.

A. Pure kernel speed comparison

Pure kernel speed comparison

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

New created kernels

The author create three kernels

- gNewGermvf for vector-matrix multiplication
- gNewGermvf^{T} for vector-matrix multiplication (transposed version)
- gNewGerf for vector-vector multiplication

They are compared with CUBLASs kernels

- Sgemv(CUDA)
- Sgemv^{T}(CUDA)
- Sger(CUDA)

And the corresponding kernel in CPU

Tests are performed on square matrices scaled 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 is evaluate like follows:

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

Test results

The average time saving is about +77.7% to +96.2% respectively.

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	$+65.7, +98.5$

B. Performance Comparison on MNIST Dataset

Performance comparison on MNIST Dataset

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

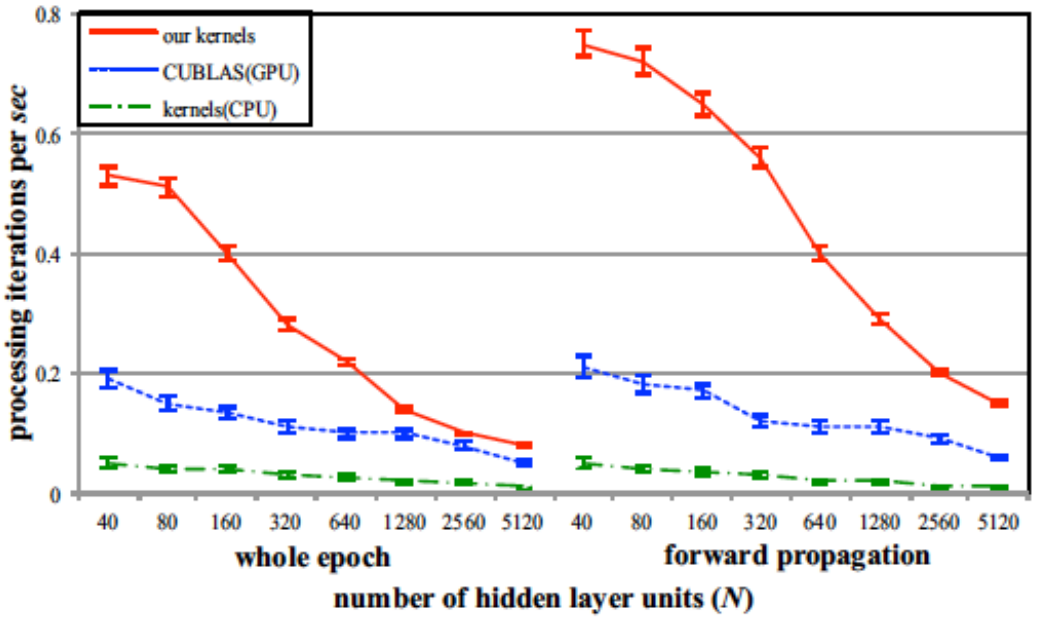
Evaluate way

- First, they evaluate the time cost of the entire training epoch that includes both forward and back propagation.

- Second, they consider only the forward propagation process, which purely consists of their kernels.

Test results

- First : +200% faster speed
- Second : +300% faster speed



C. Comprehensive Evaluation on ORL/AR face database

Experiment performed on ORL and AR face database

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.

ORL Face Database

The ORL face database consists of 400 grayscale face images of 40 people with size of 92×112 pixels.

-very limited facial expression changes.

Compared with AR face database, its image size is smaller and the amount of images is also relatively small.

AR Face Database

The AR face database contains

- more than 4000 face images (= 126 individuals with different facial expressions)
- illumination conditions
- occlusions (sunglasses and scarves)

- There are 26 pictures taken in two different sessions for each individual, and 14 of them are clean faces.

The result on ORL/AR database

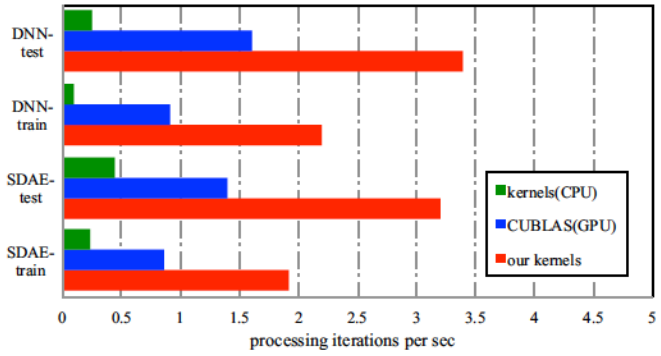


Fig. 4. We test our optimized architecture equipped with our fast matrix kernels on real face recognition problems, this result is on ORL database. The recognition process is divided into restoration part using SDAE and recognition part using DDN. Three kinds of kernels are compared.

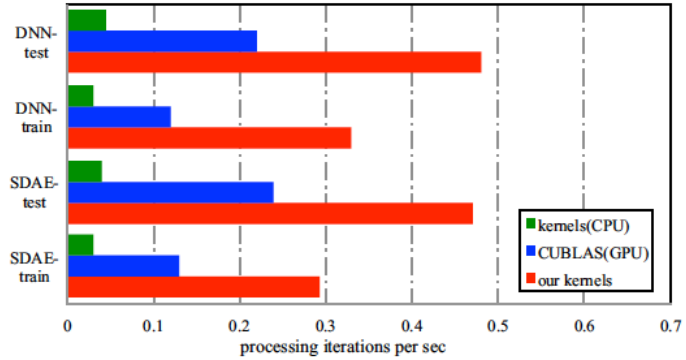


Fig. 5. The result on AR database. As we can see, the iterations per sec is less than ORL database, this is due to its larger amount of training data (4000+ images comparing with 400 images in ORL database).

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.