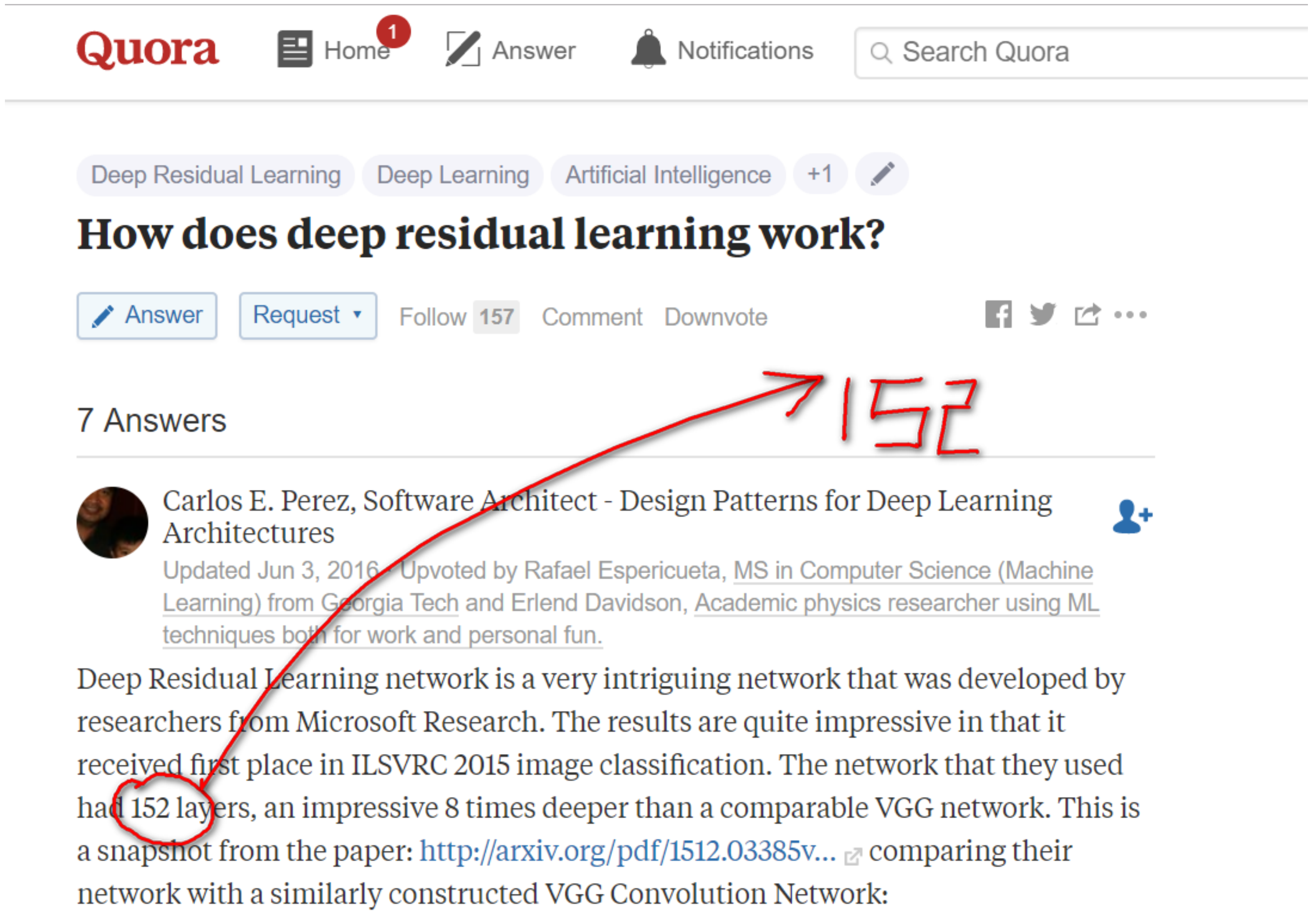
Distributed Training of Deep Neural Networks: Theoretical and Practical Limits of Parallel Scalability

BY Janis Keuper & Franz-Josef Pfreundt

17M38236 SUN NAN

Contents

Deep Neural Network



Quora Home 1 Answer Notifications Search Quora

Deep Residual Learning Deep Learning Artificial Intelligence +1

How does deep residual learning work?

Answer Request Follow 157 Comment Downvote

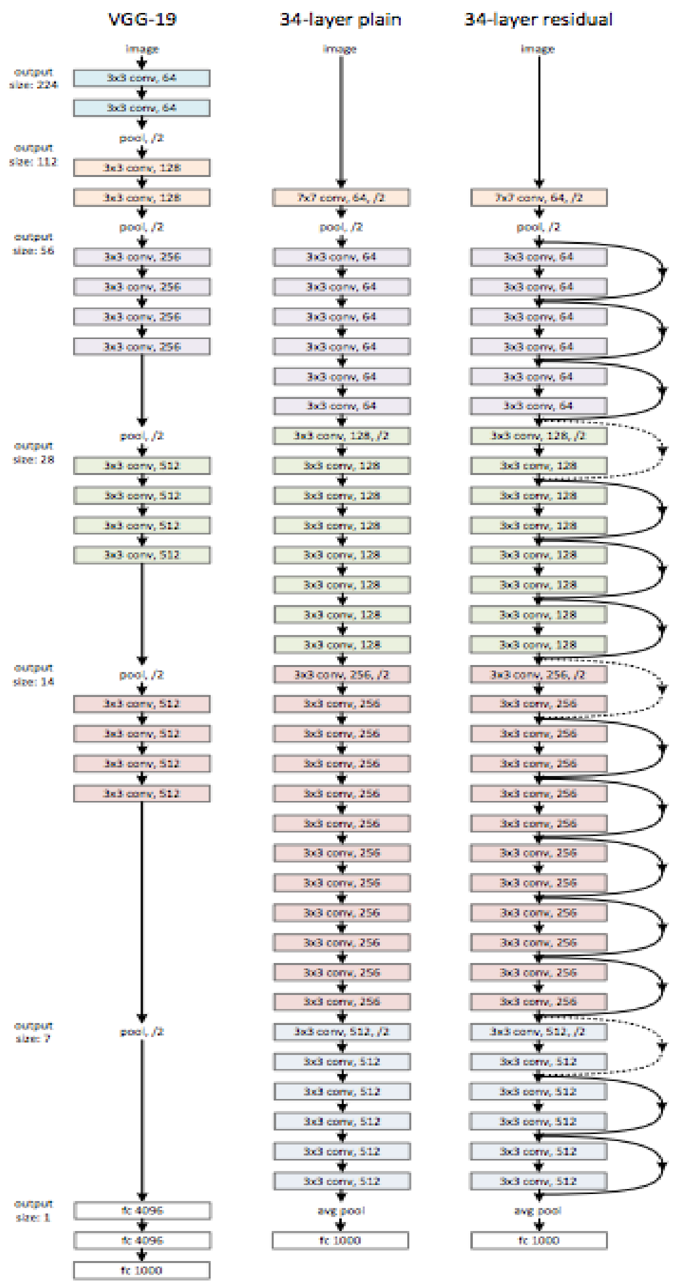
152

7 Answers

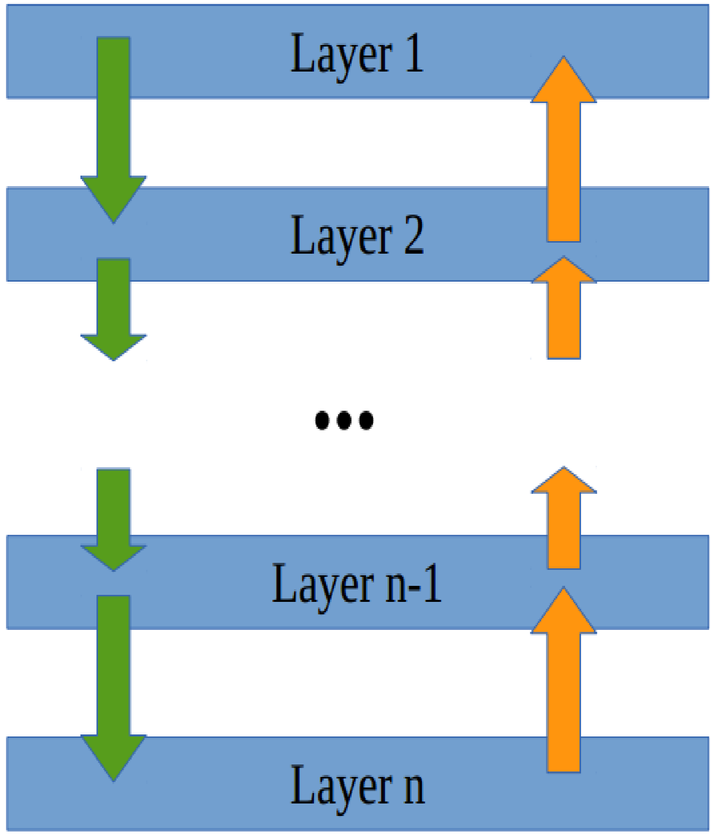
Carlos E. Perez, Software Architect - Design Patterns for Deep Learning Architectures

Updated Jun 3, 2016 · Upvoted by Rafael Espericueta, MS in Computer Science (Machine Learning) from Georgia Tech and Erlend Davidson, Academic physics researcher using ML techniques both for work and personal fun.

Deep Residual Learning network is a very intriguing network that was developed by researchers from Microsoft Research. The results are quite impressive in that it received first place in ILSVRC 2015 image classification. The network that they used had 152 layers, an impressive 8 times deeper than a comparable VGG network. This is a snapshot from the paper: http://arxiv.org/pdf/1512.03385v... comparing their network with a similarly constructed VGG Convolution Network:



Deep Neural Network



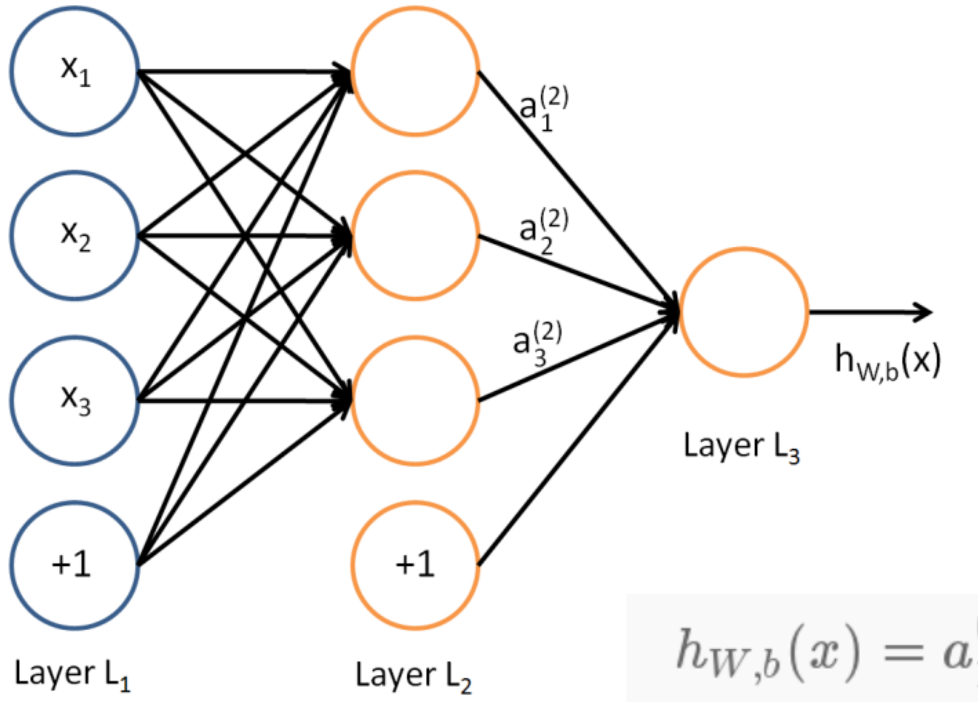
At a very abstract level, DNN means:

1. Graphs, more precisely, directed graphs
2. Nodes, compute entities (Layers)
3. Edges, data flow through graphs (Functions and Weights)

Training the networks means:

Flow the data from the top to the bottom

Deep Neural Network



Training set: $\{(x^{(1)}, y^{(1)}), \ldots, (x^{(m)}, y^{(m)})\}$

$$
\begin{aligned}
a_1^{(2)} &= f(W_{11}^{(1)} x_1 + W_{12}^{(1)} x_2 + W_{13}^{(1)} x_3 + b_1^{(1)}) \\
a_2^{(2)} &= f(W_{21}^{(1)} x_1 + W_{22}^{(1)} x_2 + W_{23}^{(1)} x_3 + b_2^{(1)}) \\
a_3^{(2)} &= f(W_{31}^{(1)} x_1 + W_{32}^{(1)} x_2 + W_{33}^{(1)} x_3 + b_3^{(1)})
\end{aligned}
$$

$$h_{W,b}(x) = a_1^{(3)} = f(W_{11}^{(2)} a_1^{(2)} + W_{12}^{(2)} a_2^{(2)} + W_{13}^{(2)} a_3^{(2)} + b_1^{(2)})$$

Deep Neural Network

Layer types(more than 100):

0. Input Layers
1. Sigmoid
2. Convolutional Layer
3. Pooling
4. ReLU
5. Softmax
6. Loss Layers
7. Local Response Normalization (LRN)

……

Training Deep Neural Network

$$J(W,b;x,y) = \frac{1}{2}\left\|h_{W,b}(x) - y\right\|^2.$$

$$\begin{aligned} J(W,b) &= \left[\frac{1}{m}\sum_{i=1}^{m} J(W,b;x^{(i)},y^{(i)})\right] + \frac{\lambda}{2}\sum_{l=1}^{n_l-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{l+1}}\left(W_{ji}^{(l)}\right)^2 \\ &= \left[\frac{1}{m}\sum_{i=1}^{m}\left(\frac{1}{2}\left\|h_{W,b}(x^{(i)}) - y^{(i)}\right\|^2\right)\right] + \frac{\lambda}{2}\sum_{l=1}^{n_l-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{l+1}}\left(W_{ji}^{(l)}\right)^2 \end{aligned}$$

$$W_{ij}^{(l)} = W_{ij}^{(l)} - \alpha\frac{\partial}{\partial W_{ij}^{(l)}}J(W,b)$$

$$b_i^{(l)} = b_i^{(l)} - \alpha\frac{\partial}{\partial b_i^{(l)}}J(W,b)$$

BP algorithm:

1. Initialize weights W at random
2. Take small random subset X (=batch) of the train data
3. Run X through network (forward feed)
4. Compute Loss
5. Compute Gradient
6. Propagate backwards through the network
7. Update W
8. Repeat until convergence

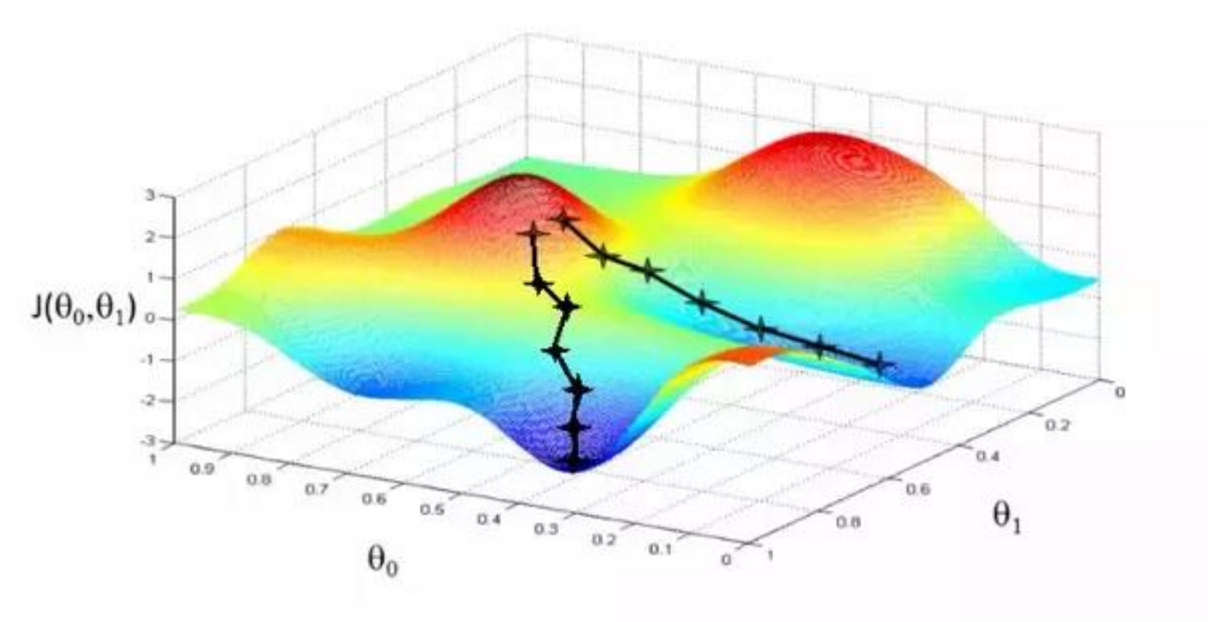
Why Stochastic Gradient Descent

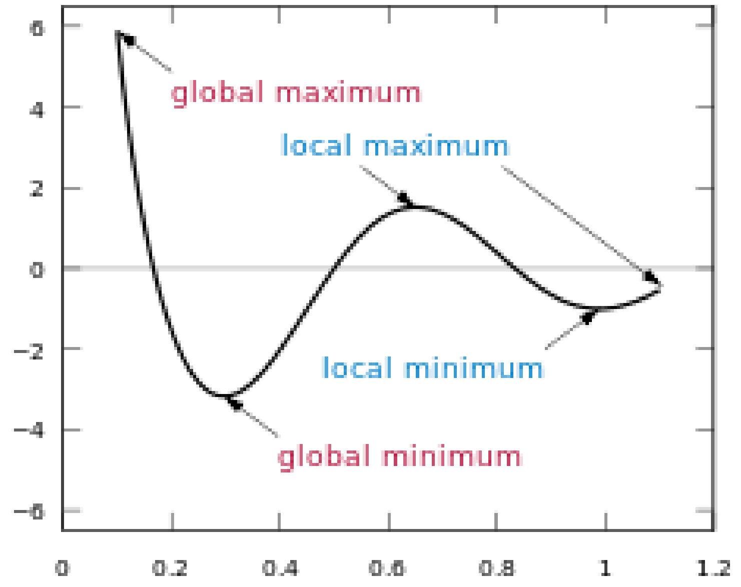
Limitations of gradient descent:

a. Relatively slow close to the minimum

b. Local minima

c. Entire data set is needed for each computation





Why Stochastic Gradient Descent

$$J(\theta) = \frac{1}{2m}\sum_{i=1}^{m}(y^i - h_\theta(x^i))^2$$

$$\frac{\partial J(\theta)}{\partial \theta_j} = -\frac{1}{m}\sum_{i=1}^{m}(y^i - h_\theta(x^i))x_j^i$$

$$\theta_j^{'} = \theta_j + \frac{1}{m}\sum_{i=1}^{m}(y^i - h_\theta(x^i))x_j^i$$

In Gradient Descent:

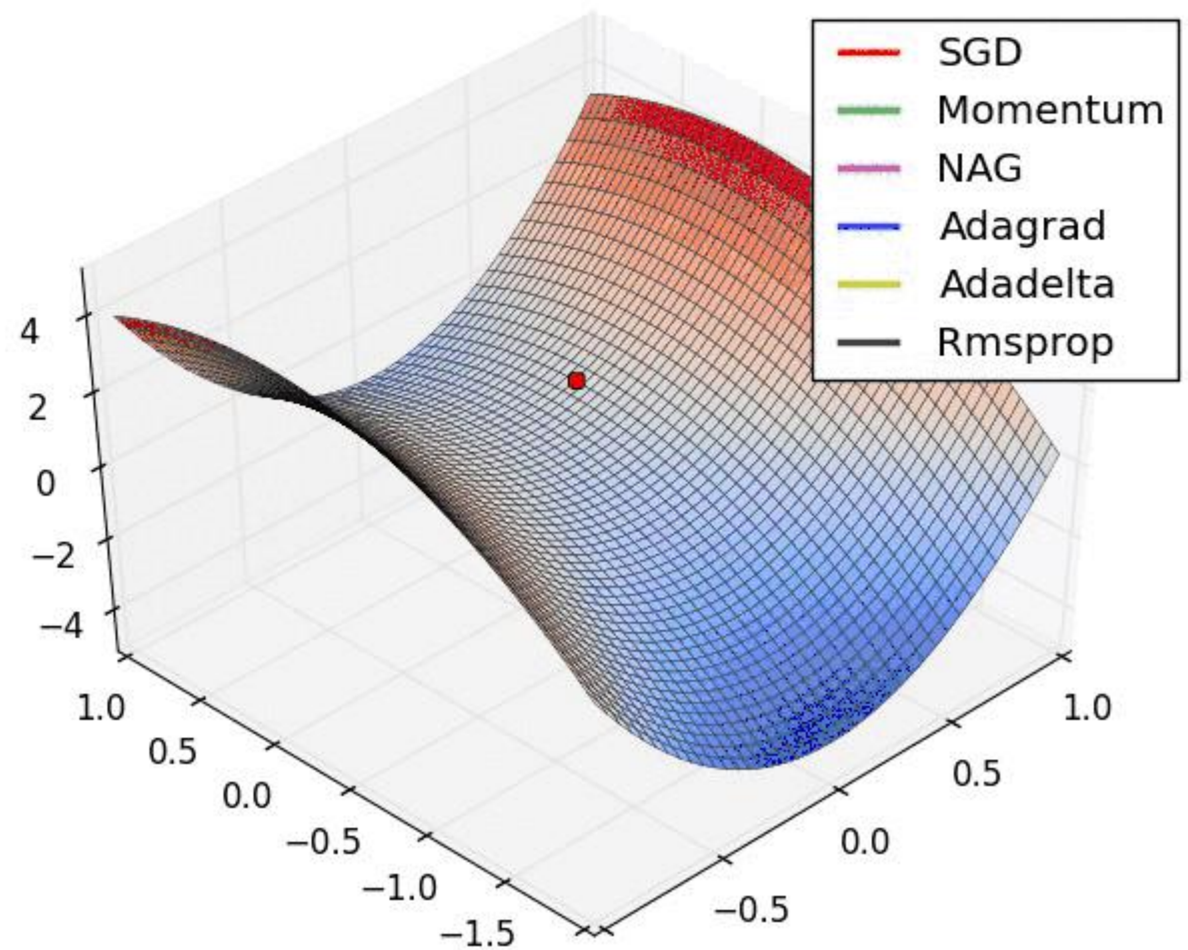
Run every training example before doing an update. When there is a large dataset, you might spend much time on getting something that works.

$$\theta_j^{'} = \theta_j + (y^i - h_\theta(x^i))x_j^i$$

In SGD:

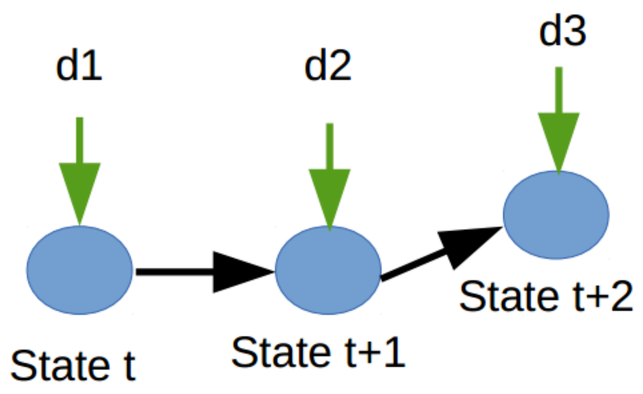
Update every time it finds a training example(*Online Learning*). On large datasets, SGD can converge faster than gradient descent since it performs updates more frequently.

Why Stochastic Gradient Descent



Parallelization

Parallelization of SGD is very hard, it is an inherently sequential algorithm



1. Start at a state t (point in a billion dimensional space)
2. Introduce t to data batch d1
3. Compute an update (based on the objective function)
4. Apply the update → t+1

Parallelization

Things we can do:

1.Make faster updates -> **Inner parallelization**

2.Make larger updates -> **Outer parallelization**

Parallelization

Inner parallelization

Def. Use parallel algorithms to compute the forward and backward operations within the layers of the DNN

1. Dense matrix multiplication

    a. Open-source BLAS(Basic Linear Algebra Subprograms)

    b. Intel® MKL(CPU)

    c. NVIDIA® cuBLAS(GPU)

    ……

Parallelization

Dense matrix:

$$A = \begin{pmatrix} 3 & 3 & 2 & 4 & 0 & 3 & 2 & 1 & 1 & 4 \\ 1 & 1 & 3 & 0 & 3 & 3 & 0 & 1 & 4 & 2 \\ 2 & 2 & 2 & 3 & 0 & 3 & 4 & 2 & 4 & 4 \\ 1 & 2 & 0 & 4 & 0 & 0 & 4 & 0 & 2 & 4 \\ 3 & 1 & 0 & 1 & 1 & 1 & 4 & 2 & 0 & 1 \\ 0 & 4 & 4 & 0 & 4 & 0 & 2 & 1 & 3 & 4 \\ 4 & 0 & 4 & 0 & 1 & 2 & 2 & 3 & 0 & 4 \\ 4 & 0 & 0 & 4 & 2 & 4 & 2 & 4 & 1 & 4 \\ 1 & 4 & 3 & 0 & 2 & 1 & 4 & 4 & 2 & 0 \\ 3 & 3 & 1 & 0 & 0 & 2 & 0 & 1 & 3 & 1 \end{pmatrix}$$

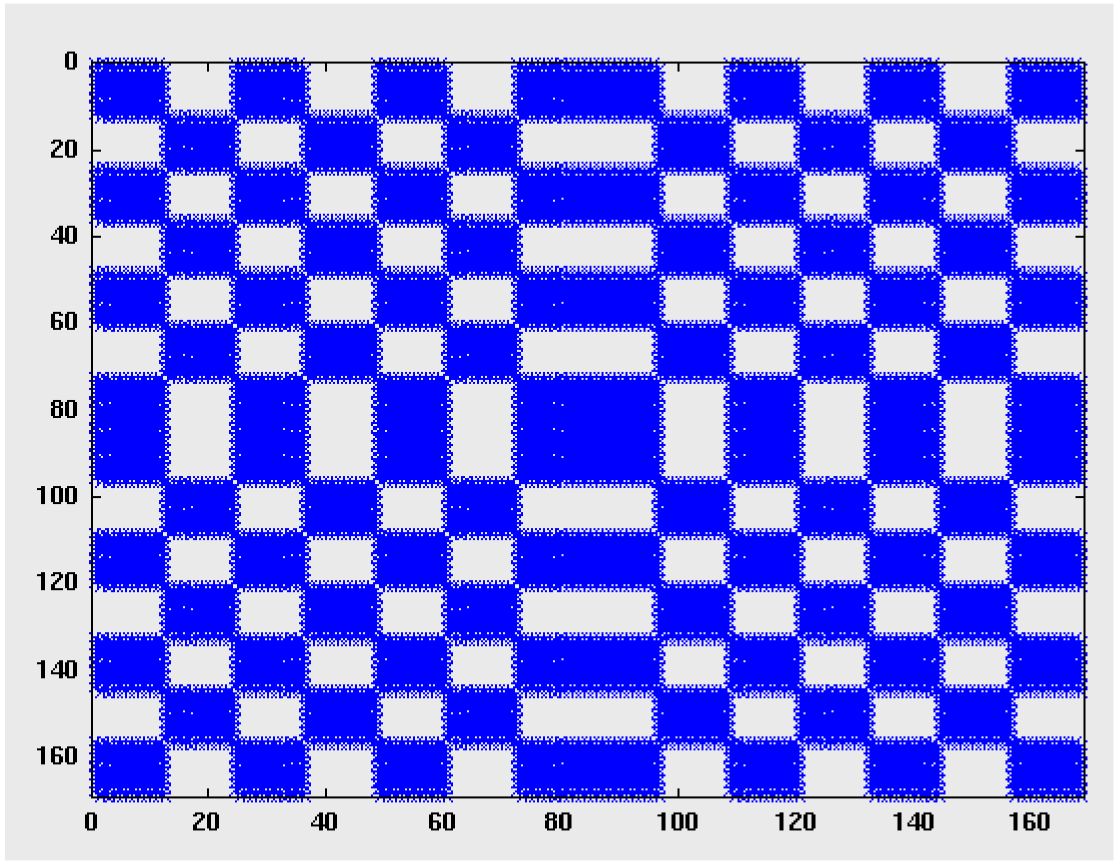
Is it a dense matrix?
Probably YES

1. Definition
Whether or not we have only a few non-zero entries

2. Basic rule
Never store the whole matrix in the memory(also brings about O(n^2) in multiplication), GPU&CPU doesn’t have “enough” memory

3. Block matrix

Parallelization



A 168×168 element block matrix with 12×12, 12×24, 24x12, and 24×24 sub-Matrices. Non-zero elements are in blue, zero elements are grayed.

Parallelization

Inner parallelization

Def. Use parallel algorithms to compute the forward and backward operations within the layers of the DNN

2. Task parallelization for special Layers

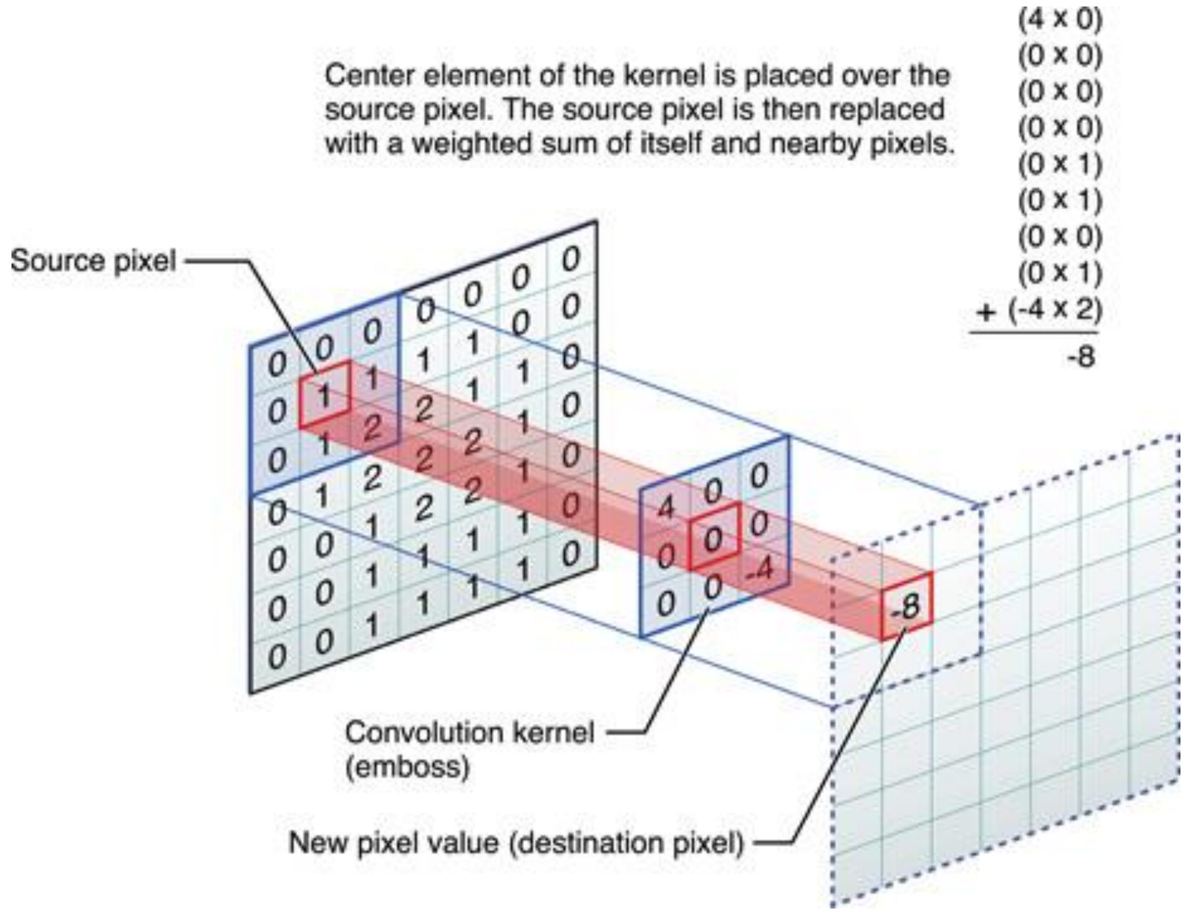
    NVIDIA® Cuda-CNN for fast convolutions

Parallelization

A quick overview of convolution operation in CNN:

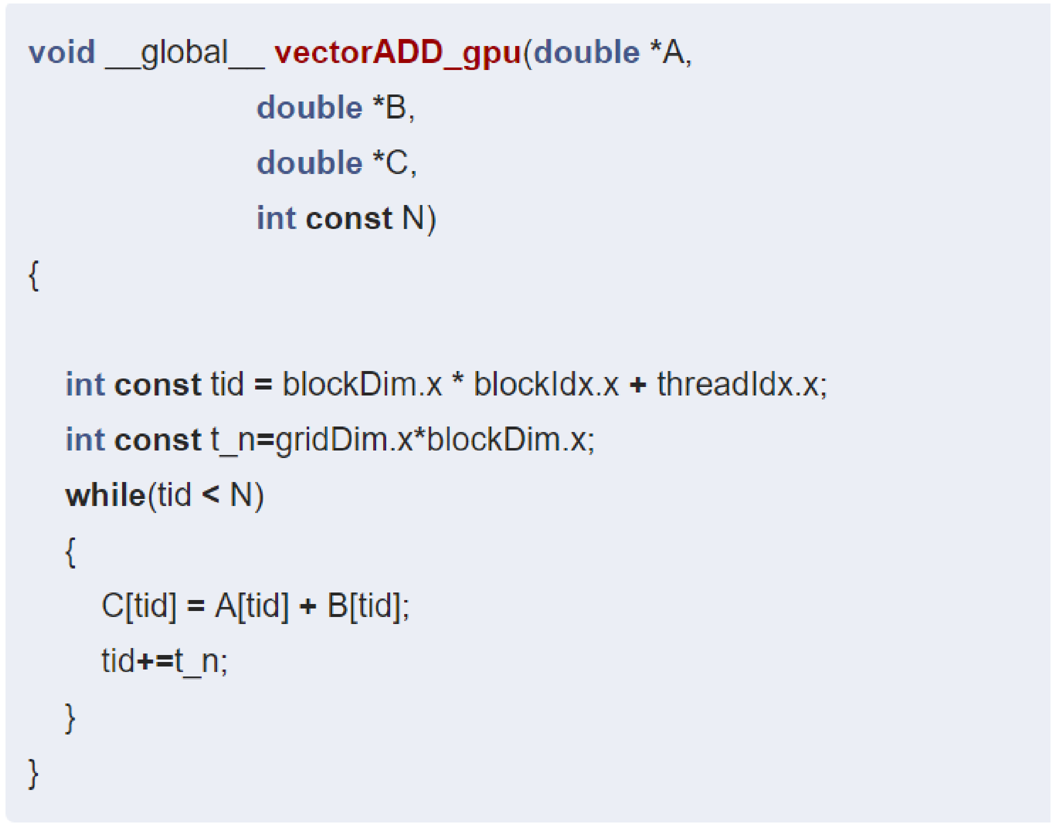
$$(f * g)(t) \overset{\text{def}}{=} \int_{\mathbb{R}^n} f(\tau) g(t-\tau)\, d\tau$$

$$(f * g)[n] = \sum_{m=-M}^{M} f[n-m] g[m].$$


Center element of the kernel is placed over the source pixel. The source pixel is then replaced with a weighted sum of itself and nearby pixels.
(4 x 0)
(0 x 0)
(0 x 0)
(0 x 0)
(0 x 1)
(0 x 1)
(0 x 0)
(0 x 1)
+ (-4 x 2)
-8
Source pixel
Convolution kernel (emboss)
New pixel value (destination pixel)

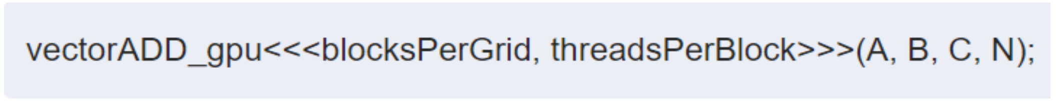
Parallelization

How to accelerate the convolution with Cuda?



```
void __global__ vectorADD_gpu(double *A,
                              double *B,
                              double *C,
                              int const N)
{

   int const tid = blockDim.x * blockIdx.x + threadIdx.x;
   int const t_n=gridDim.x*blockDim.x;
   while(tid < N)
   {
      C[tid] = A[tid] + B[tid];
      tid+=t_n;
   }
}
```



```
vectorADD_gpu<<<blocksPerGrid, threadsPerBlock>>>(A, B, C, N);
```

blockDim.x: Number of thread
blockIdx.x: Index of block
threadIdx.x: Index of thread

Kernel function will compute tid(thread‘s ID) elements in A,B and C in each thread simultaneously

Parallelization

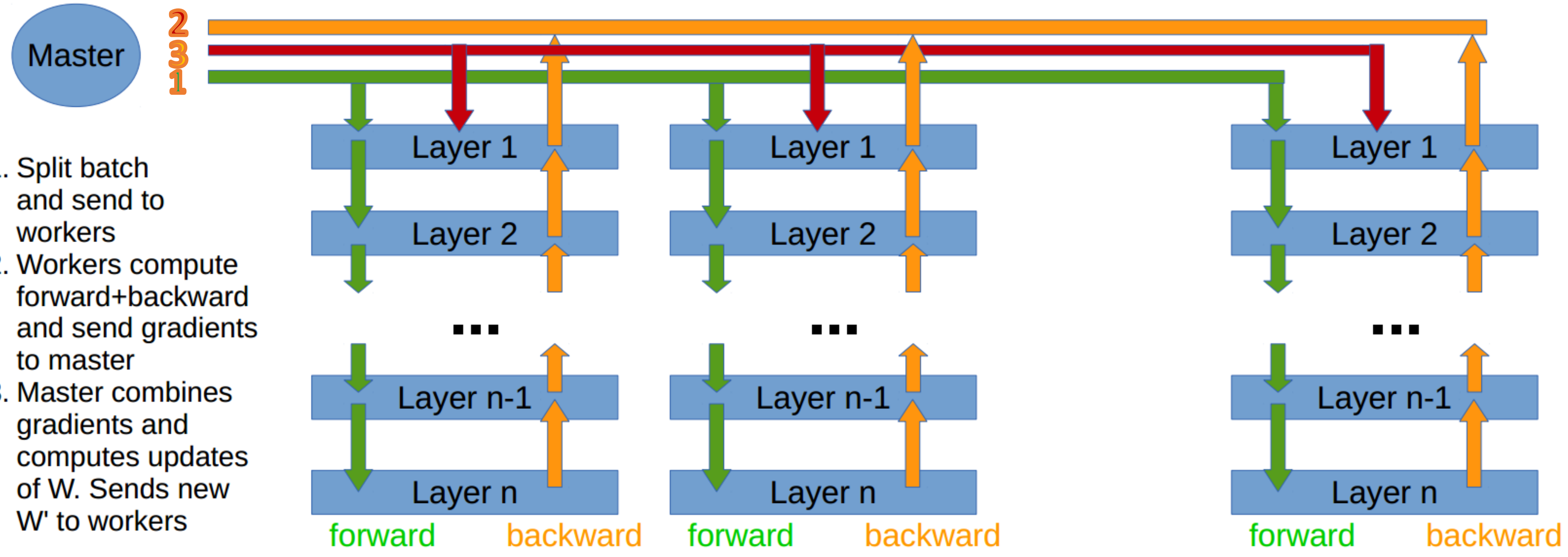
Similarly, Cuda use the parallelization to reduce time in calculating convolution

FFT(Fast Fourier Transform)
O(n^2) -> O($n \log n$)

Parallelization

Outer parallelization

Def. Use parallel algorithms to compute the forward and backward operations over the distributed batches

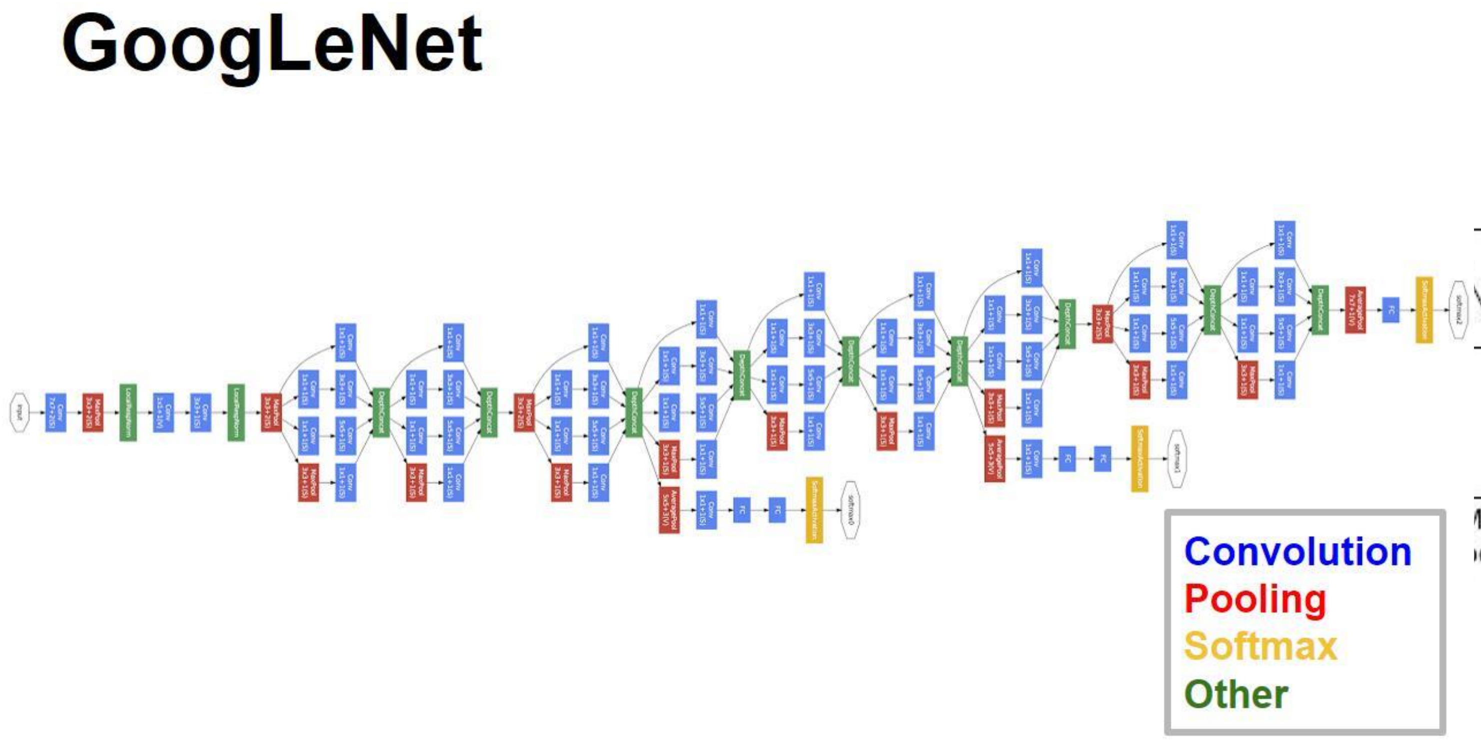


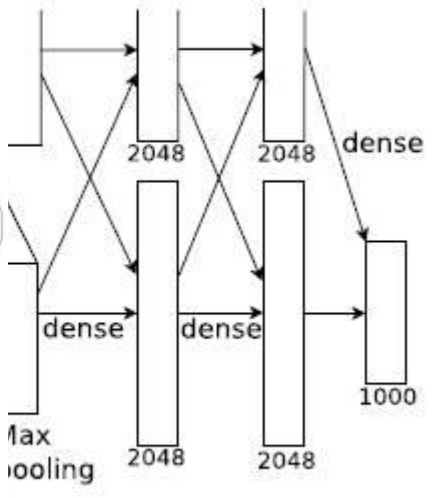
1. Split batch and send to workers
2. Workers compute forward+backward and send gradients to master
3. Master combines gradients and computes updates of W. Sends new W' to workers

Two famous CNN models for evaluation

1. AlexNet (8 layers and 60M parameters):

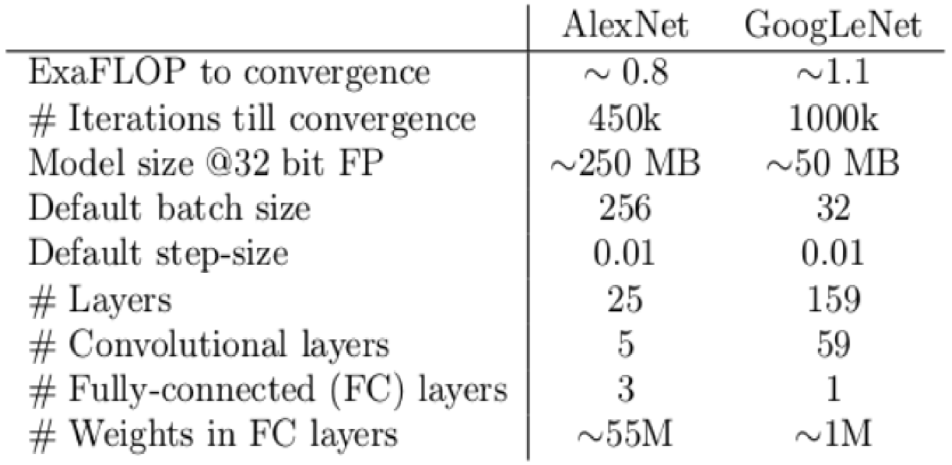
2. GoogleNet (more layers and more convolutions)





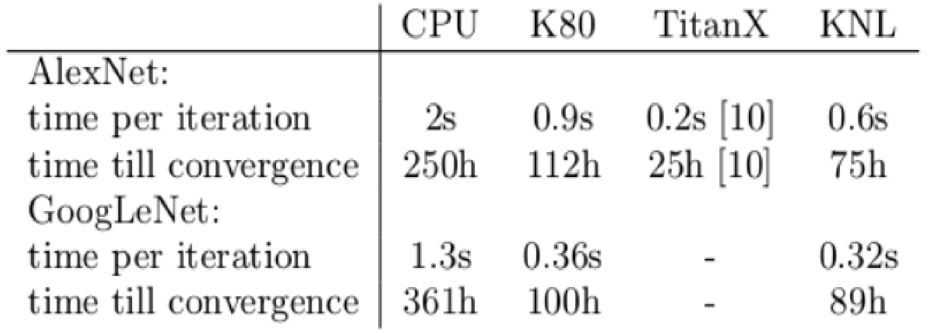
Evaluation

AlexNet vs GoogleNet on the ImageNet 2D Image labeling and object detection benchmark:



	AlexNet	GoogLeNet
ExaFLOP to convergence	~ 0.8	$\sim$1.1
# Iterations till convergence	450k	1000k
Model size @32 bit FP	$\sim$250 MB	$\sim$50 MB
Default batch size	256	32
Default step-size	0.01	0.01
# Layers	25	159
# Convolutional layers	5	59
# Fully-connected (FC) layers	3	1
# Weights in FC layers	$\sim$55M	$\sim$1M

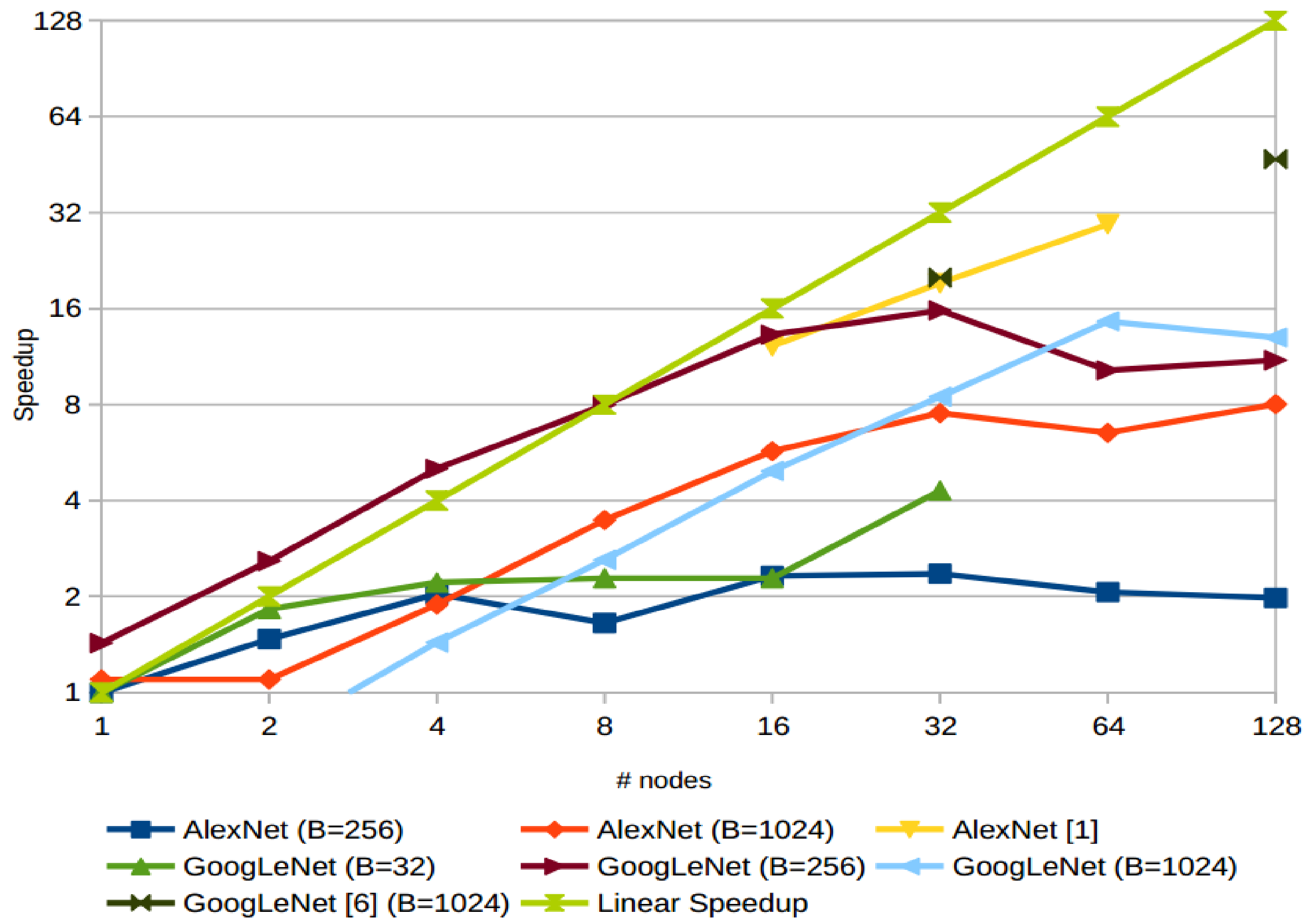
Table 2: Properties of the Deep Neural Networks used for the following benchmarks.



	CPU	K80	TitanX	KNL
AlexNet:				
time per iteration	2s	0.9s	0.2s [10]	0.6s
time till convergence	250h	112h	25h [10]	75h
GoogLeNet:				
time per iteration	1.3s	0.36s	-	0.32s
time till convergence	361h	100h	-	89h

Table 1: Approximate computation times for AlexNet with batch size $B = 256$ and 450k iterations and GoogLeNet with $B = 32$ and 1000k iterations. KNL (Xeon Phi "Knights Landing") results with MKL17. TitanX with Pascal GPU. See section 1.2.3.

Evaluation

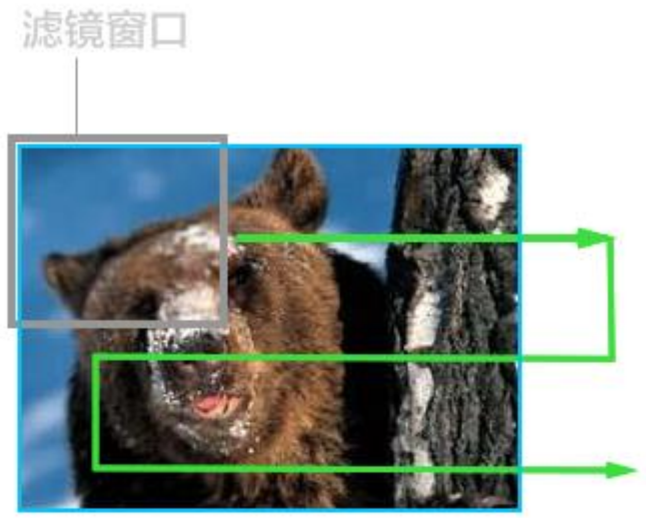


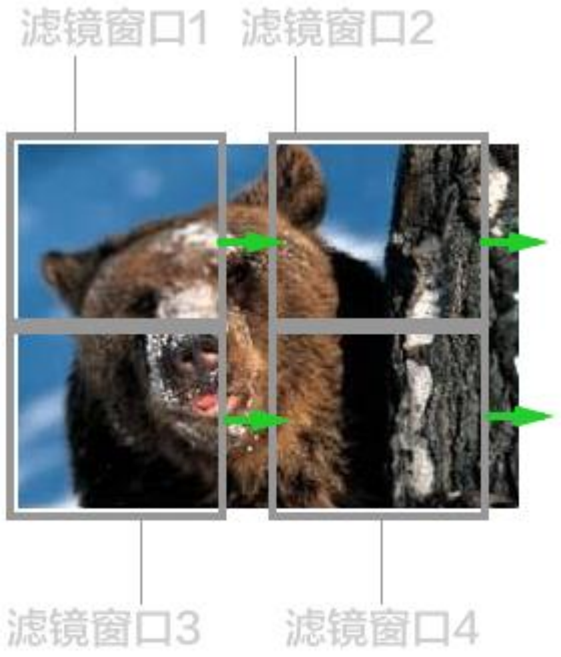
A HPC cluster with nodes holding a dual Xeon E5-2680 v3 CPU (12 cores @ 2.50GHz), a NVIDIA Tesla K80 GPU

GPU

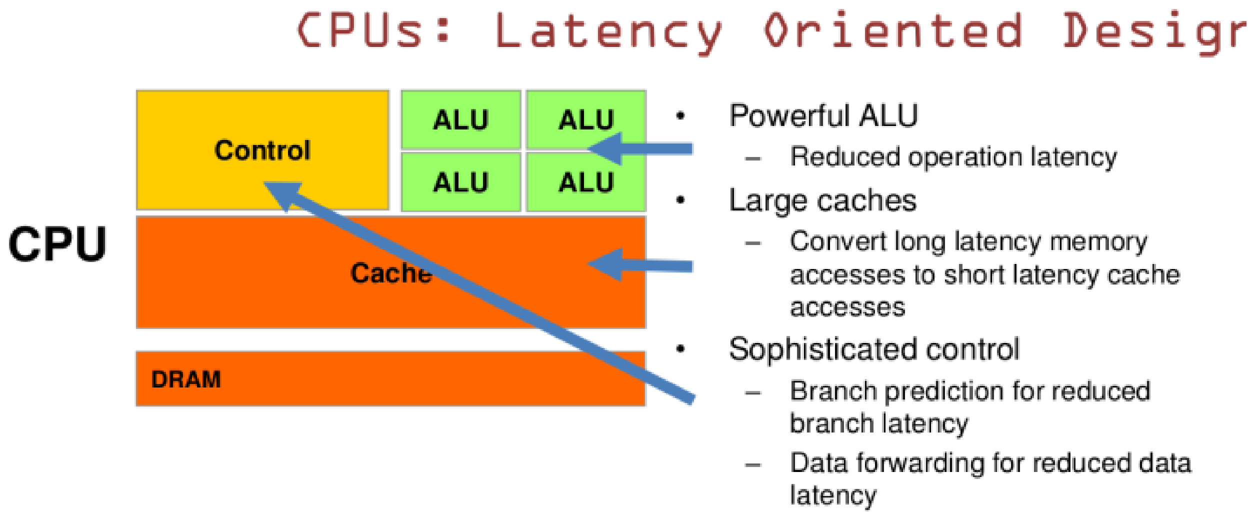
Graphics Processing Unit

1. Rendering
2. Work independently, no relations between each other

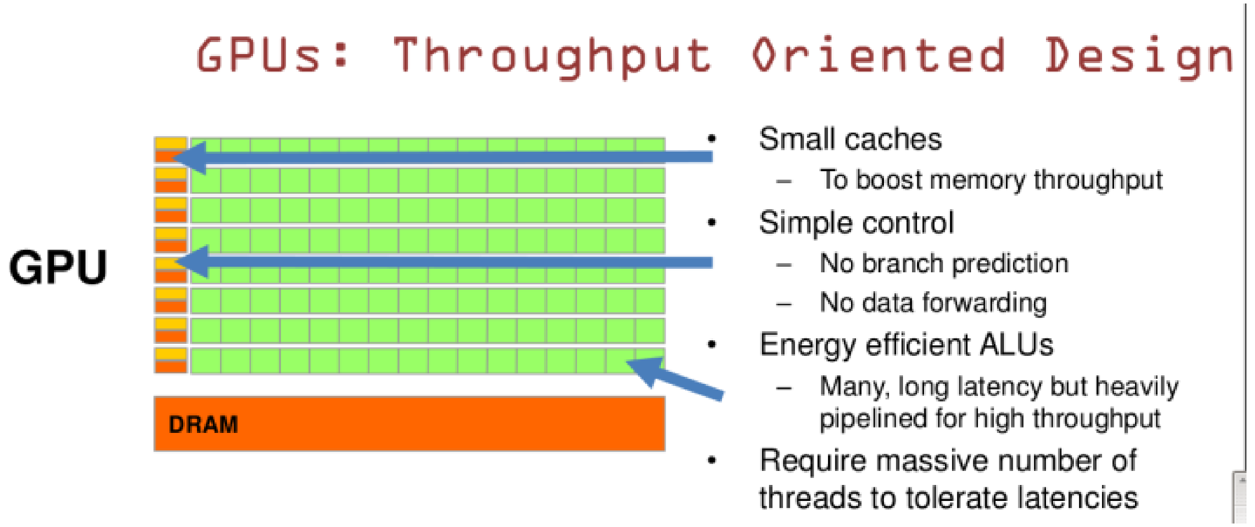




GPU


CPUs: Latency Oriented Desigr

- Powerful ALU
  - Reduced operation latency
- Large caches
  - Convert long latency memory accesses to short latency cache accesses
- Sophisticated control
  - Branch prediction for reduced branch latency
  - Data forwarding for reduced data latency


GPUs: Throughput Oriented Design

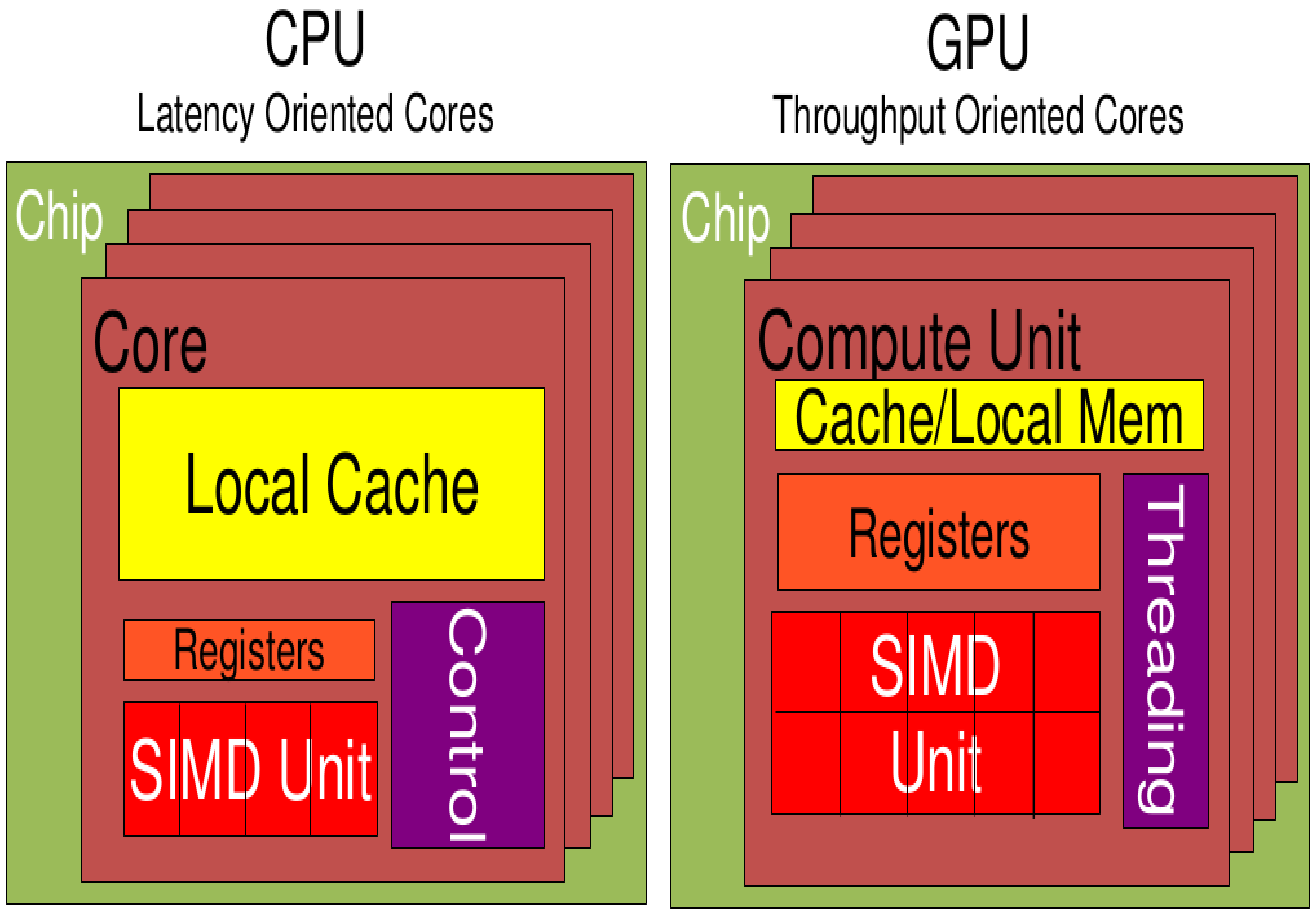
- Small caches
  - To boost memory throughput
- Simple control
  - No branch prediction
  - No data forwarding
- Energy efficient ALUs
  - Many, long latency but heavily pipelined for high throughput
- Require massive number of threads to tolerate latencies

GPU vs. CPU

1. More threads(not cores) and registers
2. Cache is used to improve thread's performance, not to store the data
3. Different coding methods
4. More SIMD(single instruction multiple data) Unit
5. Compute-intensive & Parallelized calculations

"One professor vs. Thousands of primary school students"

GPU

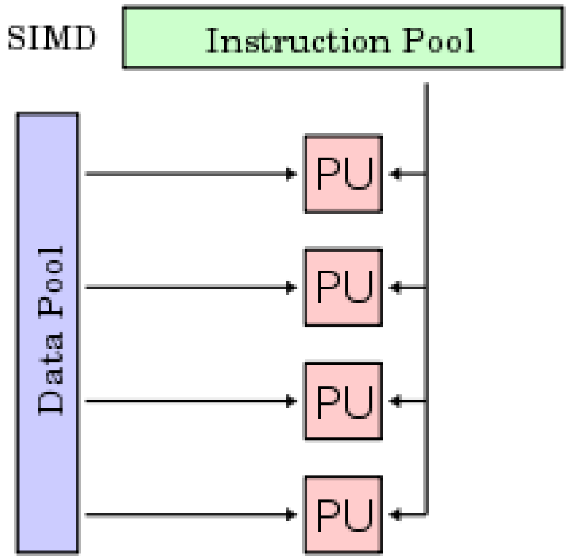


GPU vs. CPU

1. More threads(not cores) and registers
2. Cache is used to improve thread's performance, not to store the data
3. Different coding methods
4. More SIMD(single instruction multiple data) Unit
5. Compute-intensive & Parallelized calculations

"One professor vs. Thousands of primary school students"

GPU



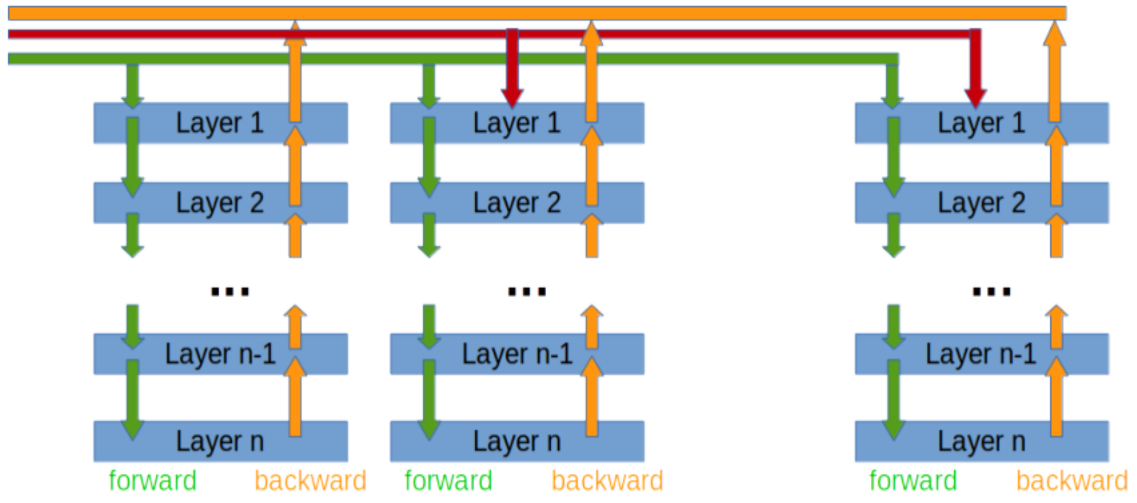
GPU vs. CPU

1. More threads(not cores) and registers
2. Cache is used to improve thread’s performance, not to store the data
3. Different coding methods
4. More SIMD(single instruction multiple data) Unit
5. Compute-intensive & Parallelized calculations

“One professor vs. Thousands of primary school students”

Limitation Ⅰ

Distributed SGD is heavily Communication Bound:

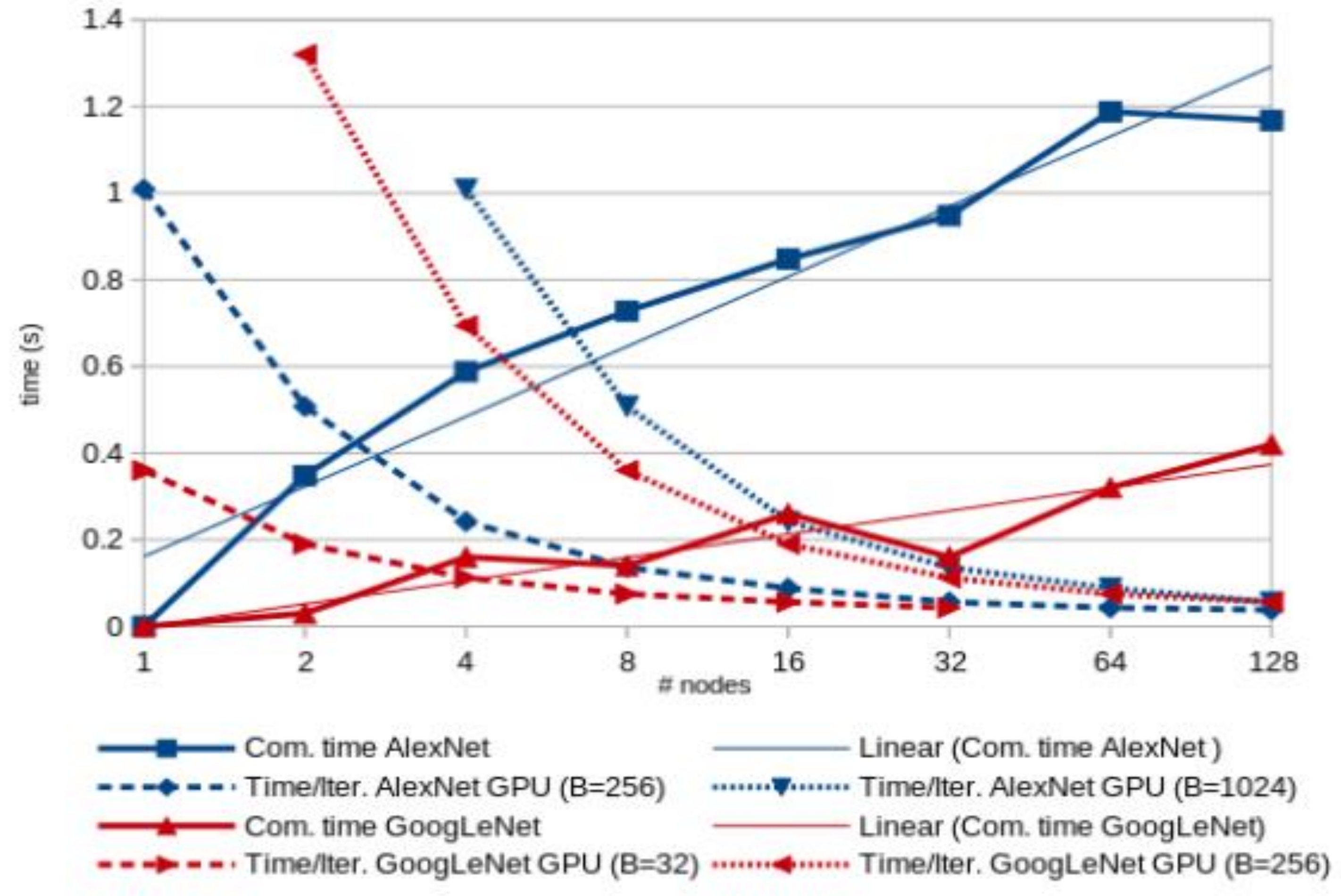


Network Bandwidth is limited:

1. Model size can be hundreds of MB(Transfer Data)

2. GPU Iteration time

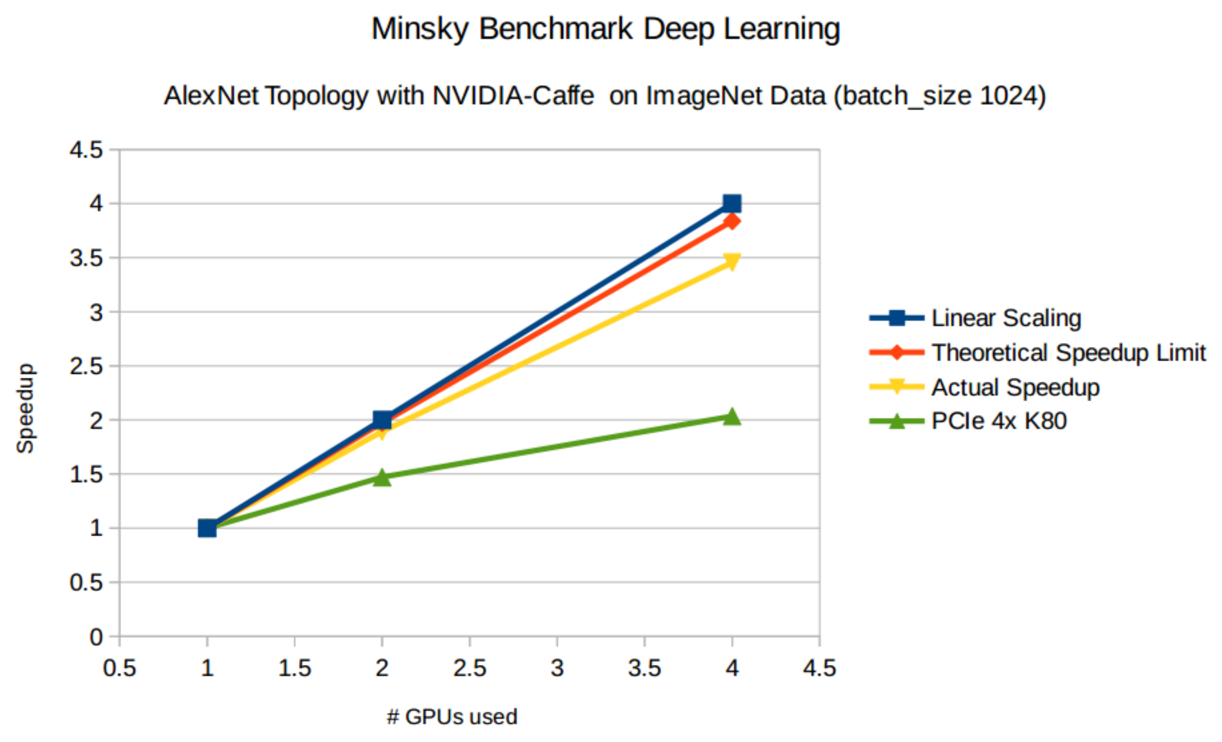
Limitation Ⅰ



Communication overhead for different models and batch sizes. The scalability stalls when the compute times drop below the communication times, leaving compute units idle. Hence becoming an communication bound problem.

Limitation Ⅰ

Hardware solutions:



IBM Minsky

- 4x P100
- 2x10 core Power8 (160 hw threads)
- NVLink between all components

NVLink spec: ~40GB/s (single direction)

Limitation Ⅰ

Algorithm solutions：

(I) Re-design of the network eliminating unused weights

a. Avoid fully connected Layers for smaller models

(II) Limit the numerical precision of the model weights

a. Reduce Floating Point precision (8 Bit is enough)

(III) Reduce / Avoid Communication

a. Compression

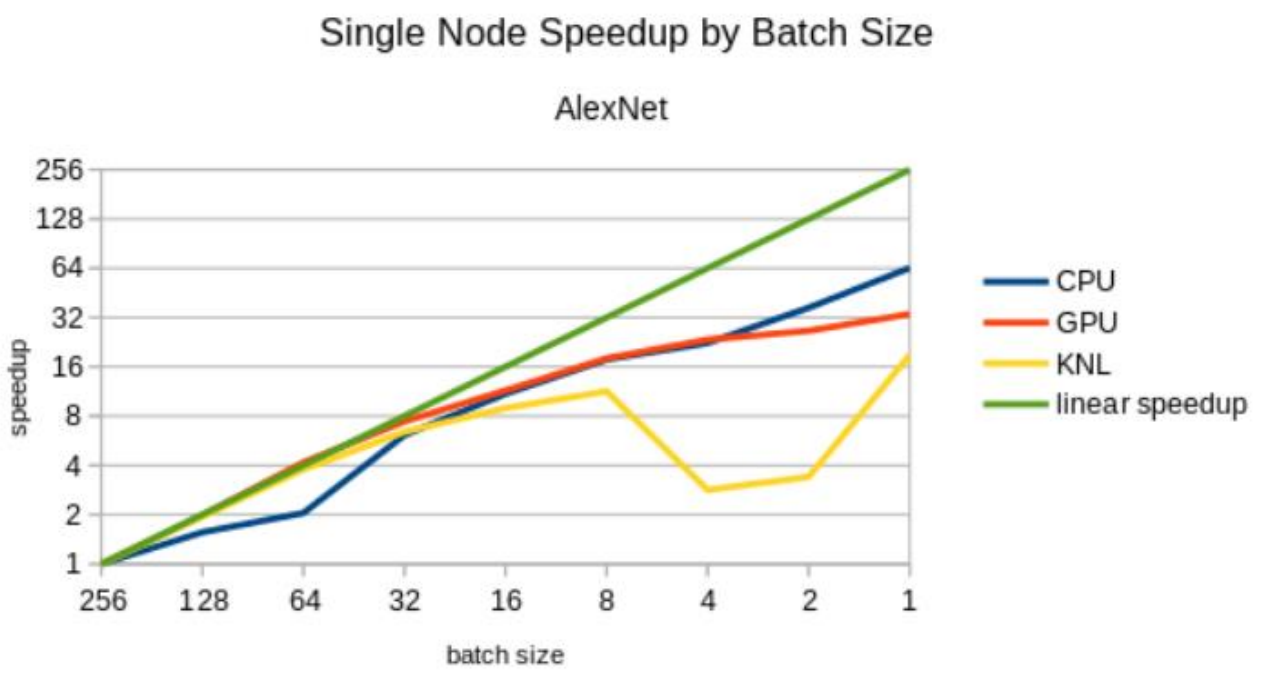
b. Transmit key information

Limitation Ⅰ

But we are still not there, why?

Assume we have already solved all the problems in communication, or Free Communication…

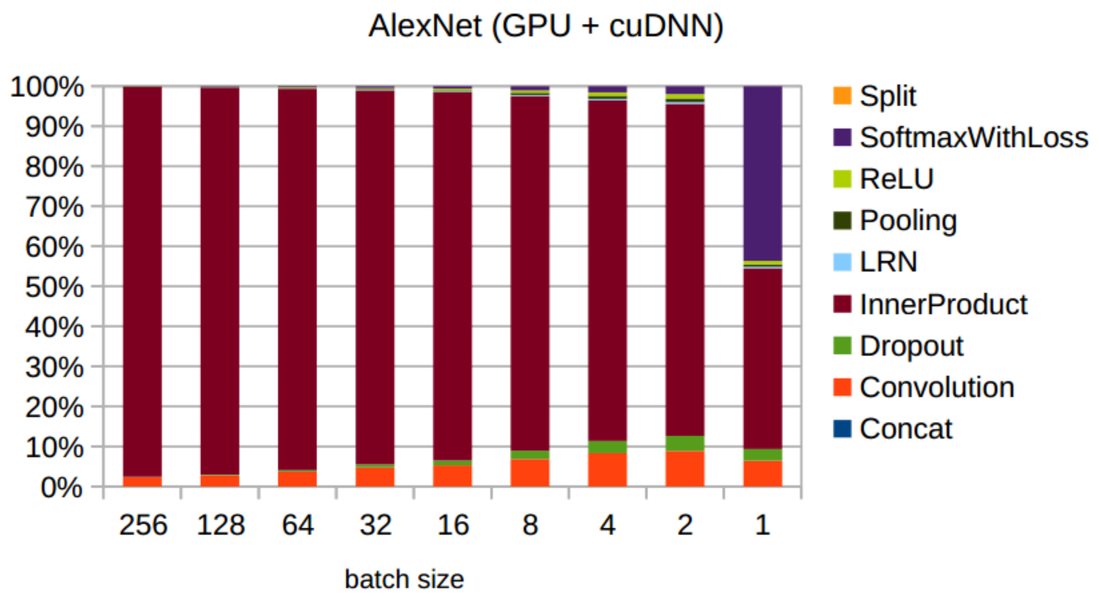
Limitation I

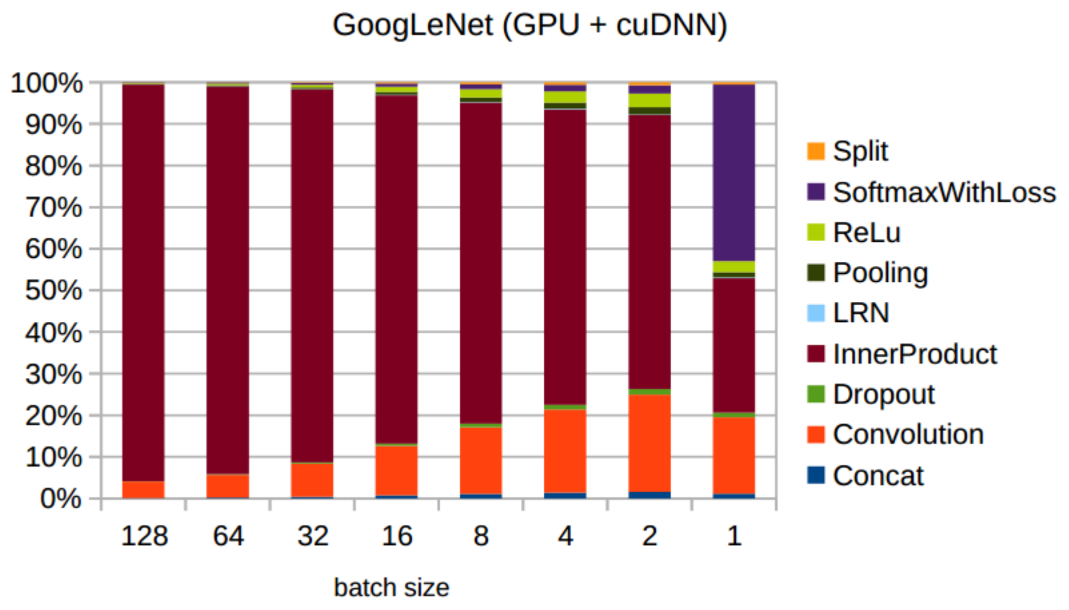


Simulated by measuring the ***compute times at a single node*** *at decreasing batch sizes*

Limitation Ⅰ

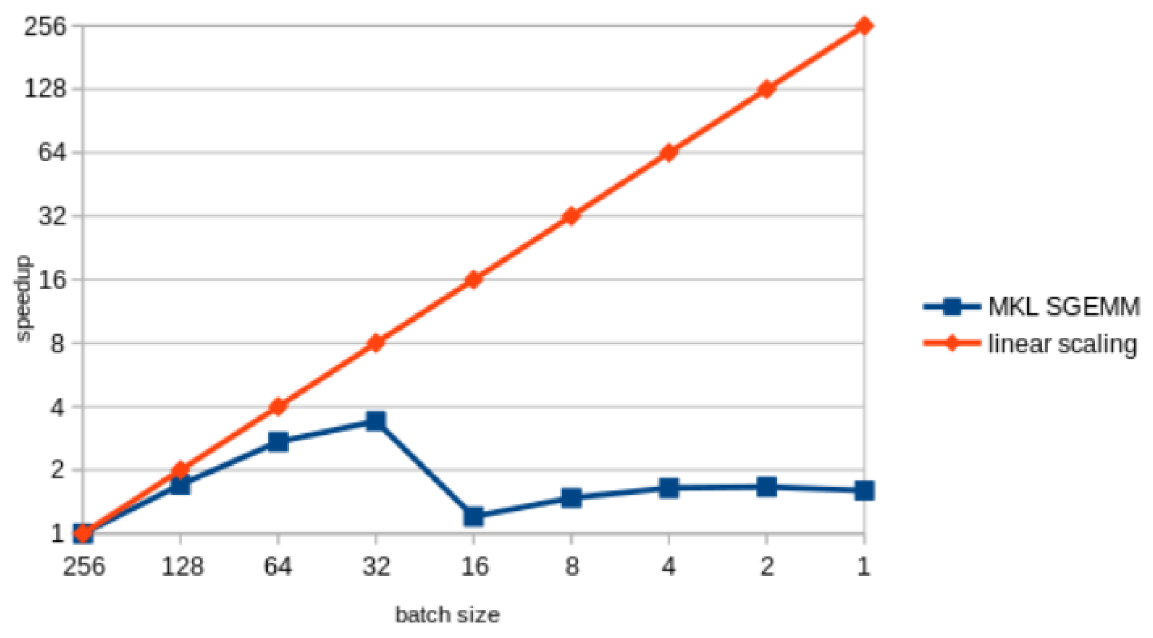
Compute time by Layer





Evaluation of the relative ***compute time for each layer type*** *(several layers of the same type are accumulated) per training iteration on a single node GPU based.*

Limitation Ⅰ



Impact of the batch size b for matrix multiplications with the shape b $\times$ *4096* $*$ *4096* $\times$ *9192*

“1 million neurons with 256 training samples ”

Limitation Ⅱ

Parallelizing "Skinny" Matrix Multiplication:

One problem, but very basically: Batch size decreases with distributed scaling

For skinny matrices there is simply not enough work for efficient internal parallelization over many threads

Limitation Ⅱ

Solution:

Increase Batch size(advantage)

a. Enhance the utilization of memory, also improves parallel efficiency

b. Iterations of the whole epoch is reduced, which means a great speedup in dealing with the same amount of data compared to the small size

c. Faster in determining the gradient direction

Limitation Ⅱ

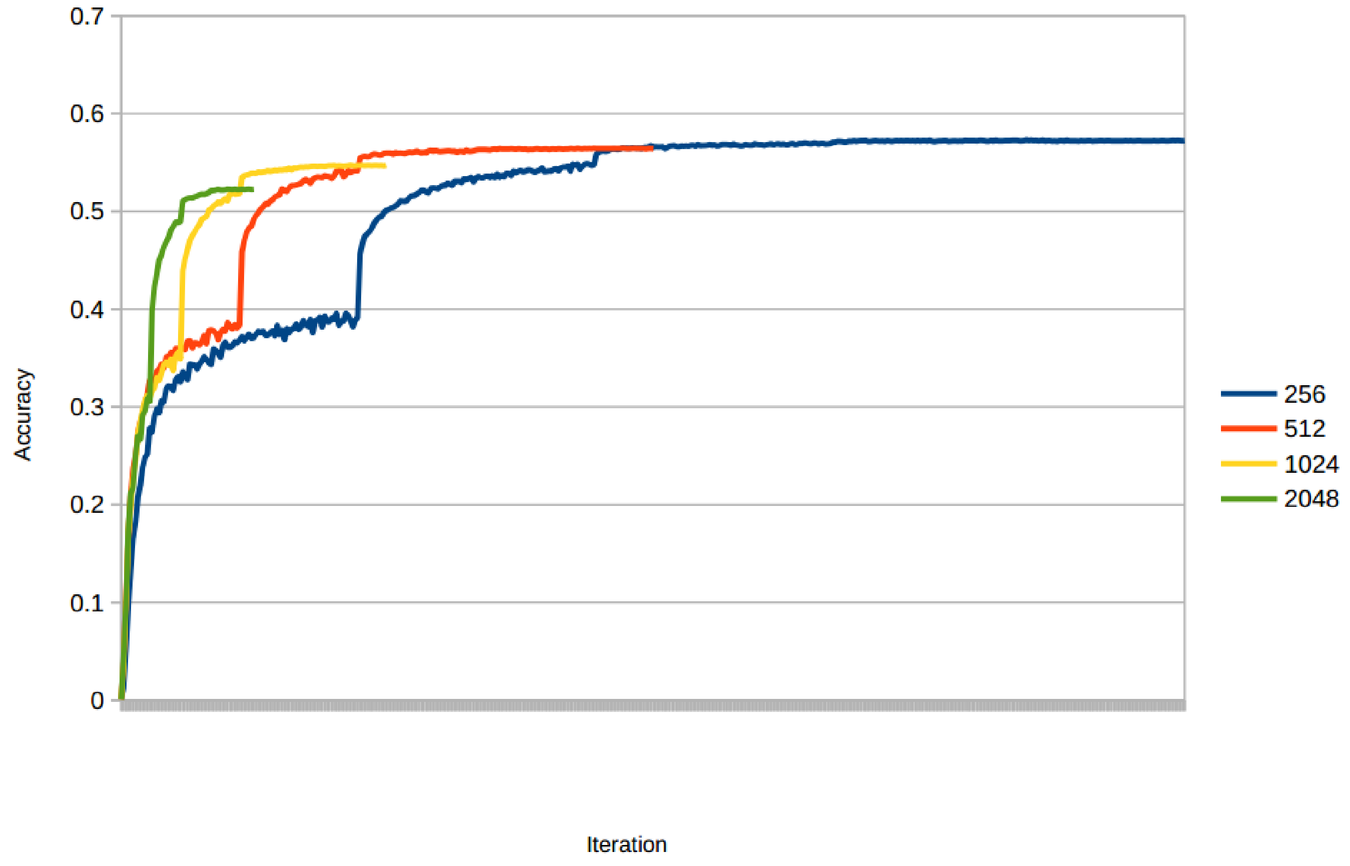
Solution:

Increase Batch size(disadvantage)

a. Memory capacity is limited

b. Loss of accuracy

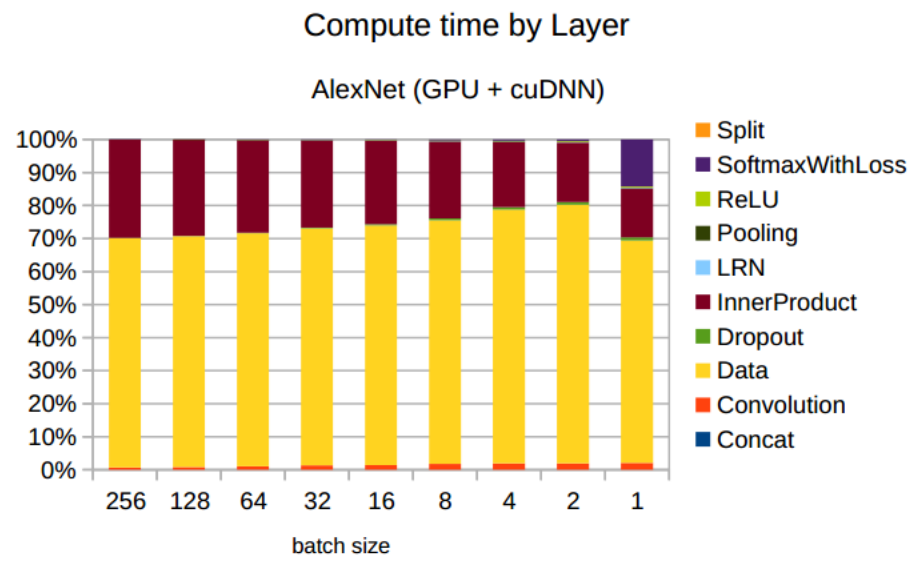
c. Direction is a tiny issue

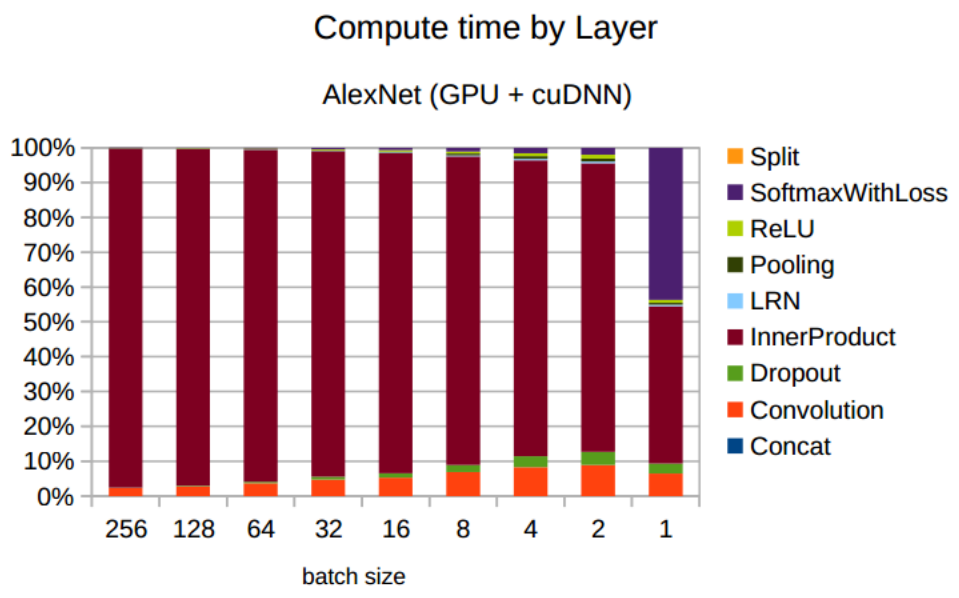


Full validation accuracy plot for AlexNet with different large batch sizes. Settings [B = 256; ε= 0.01; iter = 450k], [B = 512; ε=0.02; iter = 225k], [B = 1024; ε = 0.04; iter = 112k], [B = 2048; ε =0.08; iter = 56k]. ε is Step sizes

Limitation Ⅲ

Distributed File Systems(I/O):





Results shown for SINGLE node access to a Lustre working directory (HPC Cluster, FDR-Infiniband)

Results shown for SINGLE node Data on local SSD.

“Loading Data, very fundamentally but you have to spend time on it”

Limitation Ⅲ

1. Network bandwidth is already exceeded by the SGD communication(I/O)

AlexNet needs 100 epochs(=full pass of the training data) till convergence, resulting in 100 ╳ 150GB= 15TB of total data traffic compared to 450000╳ 250MB╳2(n- 1) in gradient and update communication

2. Worst possible file access pattern:

Access many small files at random

An example on local multi-GPU computations:

Single SSD (>0.5 GB/s) to slow to feed >= 4 GPUs

Limitation Ⅲ

Solutions:

Local SSDs, but more problems to solve

Conclusions

Situations:

1. The main problem with training DNNs via distributed SGDs is that the computation load per iteration is too low.
2. This problem will further increase with faster compute units (GPUs).

Possible solutions:

1. Change Network to handle the over-fitting problem for large Batch sizes
2. Alternative optimization methods (SGD is not the only way)

Suggestions

1. Avoid "fat" layers with too many parameters:

a. For CNNs, go deeper with convolutions (As MS does in their modules)

b. Use less fully connected layers

2. Revise network

3. Optimize meta-parameters for larger batch-sizes:

a. Better scalability(At the very beginning)

b. Better I/O performance

Thank you!