Performance Analysis of CNN Frameworks for GPUs

Heehoon Kim*‡ Hyoungwook Nam†‡ Wookeun Jung* Jaejin Lee*

*Department of Computer Science and Engineering,
†College of Liberal Studies,
Seoul National University, Korea
{heehoon, hyoungwook, wookeun}@aces.snu.ac.kr, jaejin@snu.ac.kr
http://aces.snu.ac.kr

‡These two authors contributed equally to this work as the first authors.

***Abstract*—Thanks to modern deep learning frameworks that exploit GPUs, convolutional neural networks (CNNs) have been greatly successful in visual recognition tasks. In this paper, we analyze the GPU performance characteristics of five popular deep learning frameworks: Caffe, CNTK, TensorFlow, Theano, and Torch in the perspective of a representative CNN model, AlexNet. Based on the characteristics obtained, we suggest possible optimization methods to increase the efficiency of CNN models built by the frameworks. We also show the GPU performance characteristics of different convolution algorithms each of which uses one of GEMM, direct convolution, FFT, and the Winograd method. We also suggest criteria to choose convolution algorithms for GPUs and methods to build efficient CNN models on GPUs. Since scaling DNNs in a multi-GPU context becomes increasingly important, we also analyze the scalability of the CNN models built by the deep learning frameworks in the multi-GPU context and their overhead. The result indicates that we can increase the speed of training the AlexNet model up to 2X by just changing options provided by the frameworks.**

I. Introduction

Deep neural networks (DNNs) have been very successful in various machine learning tasks, such as visual recognition [1]–[3], speech recognition [4], and machine translation [5]. Among others, the convolutional neural network (CNN) proposed by LeCun *et al.* [6] is one of the earliest successful DNN models that were used to classify images. CNN models equipped with deep learning techniques (*e.g.*, ReLU activation, dropout layers, data augmentation, etc.) outperform previous machine learning techniques in various visual recognition challenges, such as ILSVRC [7] and PASCAL [8].

A larger and deeper CNN with more parameters usually results in a better accuracy. However, a larger CNN requires more processing power, and training it using a typical computer is impractical. Fortunately, computations in a CNN can easily be represented as tensor or matrix operations that can be efficiently parallelized. Thus, GPUs' massively parallel processing power makes CNNs to be trained efficiently, and most of popular deep learning frameworks support GPU acceleration by default [9]–[13].

The most popular deep learning library for such frameworks is cuDNN [14] developed by NVIDIA, and most of the popular deep learning frameworks use it as the backend for GPUs. An approach to speed up CNNs is reducing the time complexity of convolution algorithms. Fast Fourier Transform (FFT) algorithms [15], [16] and Winograd's minimal filtering algorithm [17] successfully reduce the algorithm complexity of the convolution computation in a CNN. However, while the efficiency of a CNN on a single GPU has been improved a lot, its efficiency on multiple GPUs still shows poor scalability [18].

Users choose a deep learning framework to build their CNN models based on language interfaces, operating system support, performance, ease of use, etc.. Ideally, the execution time of the same CNN model should be the same across all the frameworks when the input is the same. However, in reality, this is not true; a CNN model built with a framework delivers more than twice the performance compared to the same model built with a different framework [19], [20].

In this paper, we analyze the performance characteristics of CNN models built with different deep learning frameworks and libraries. For clarity, a *framework* refers to a full collection of libraries to build DNN models and a *library* refers to a GPU library such as cuDNN used in the framework. We choose five most popular deep learning frameworks, Caffe [10], CNTK [13], TensorFlow [11], Theano [9], and Torch [12]. The popularity criterion is the number of GitHub stars [21].

We choose a representative CNN model, AlexNet [1], to compare the five frameworks and to obtain performance characteristics. Identically structured AlexNet models are built and trained using different frameworks. All five frameworks use cuDNN as the GPU backend. Cuda-convnet [22] is another GPU library used with those frameworks. In addition, we compare three different convolution algorithms of cuDNN with the direct convolution algorithm of Cuda-convnet.

Our comparative study provides useful insights to both end users and developers of a DNN framework. In the case of end users, their main interest is the accuracy and speed of the network in training and testing, not the optimization of the implementation. They sometimes do not know that their

978-1-5386-3890-3/17/$31.00 ©2017 IEEE

model built with the DNN framework is slower than those with other frameworks. Moreover, they do not know the reasons of slowdown either. On the other hand, the developers want to know where the bottlenecks occur to optimize their framework implementation. Both end users and developers want to improve the performance of DNN models built by their framework.

We identify which part of the GPU implementation of a framework is the performance bottleneck. This provides the developers with optimization possibilities. This will also help end users to choose a proper framework for their CNN models.

The contributions of this work are as follows:

- We analyze differences in the performance characteristics of the five frameworks in a single GPU context. Unlike previous approaches, we measure layer-wise execution times as well as the processing time of an input batch. Based on the measurement, we identify performance limiting factors of each framework.
- We show the performance characteristics of different convolution algorithms. Based on the characteristics, we provide possible optimization techniques to implement efficient CNN models using the algorithm libraries.
- We show the performance characteristics of the deep learning frameworks in multi-GPU contexts. We also provide possible techniques to improve the scalability of the frameworks in the multiple GPU context.

Based on the characteristics obtained, we increase the speed of training the AlexNet model up to 2X by just changing options provided by the framework compared to the same model built with default options. We do not modify the source code of the framework at all. Thus, the end users can easily adopt our techniques to their CNN model building processes.

II. Background and Related work

In this section, we briefly describe the five major deep learning frameworks and a typical organization of CNNs. We also introduce some related studies that perform comparative studies of deep learning frameworks.

A. Deep Learning Frameworks

There are five major deep learning frameworks that are frequently used by machine learning users to build deep learning models: Caffe [10], CNTK (Computational Network Toolkit) [13], TensorFlow [11], Theano [9], and Torch [12]. Table I summarizes the frameworks used in this paper (we will describe the data parallelism and model parallelism later in this section).

Visual recognition challenge winners are usually implemented with Caffe [2], [3], [23]. However, the flexibility of Caffe is somehow limited. Introducing a new feature to a layer requires re-building the source code. CNTK developed by Microsoft is widening its user base even though it was introduced most recently in 2016. TensorFlow was publicly introduced in 2015 and is now the most popular deep learning framework in GitHub [21]. Theano is one of the earliest deep learning frameworks. Pylearn2 [24], Keras [25], and Lasagne [26] are popular DNN frameworks that use Theano as their backend. However, the multi-GPU support of Theano is still in an experimental stage. Torch is a scientific computing framework based on LuaJIT [12] and also one of the earliest DNN frameworks. NVIDIA's self-driving car project [27] and Deepmind's Deep Q Learning model [28] were built on top of it.

B. Convolutions

A convolution is an operation between two functions. Its value shows the degree of similarity between the two functions f and g. Convolutions can also be naturally defined for discrete values. We can define the convolution of two finite sequences $f[n]$ $(0 \leq n \leq N-1)$ and $g[j]$ $(0 \leq m \leq M-1)$ as follows:

$$(f * g)[n] = \sum_{m=0}^{M-1} f[n+m]g[m] \quad (1)$$

The convolution operation can be extended to multiple dimensions. A two-dimensional (2D) convolution between filter (a.k.a. kernel) $F[r][s]$ $(0 \leq r \leq R-1, 0 \leq s \leq S-1)$ and data $D[h][w]$ $(0 \leq h \leq H-1, 0 \leq w \leq W-1)$ can be described with the following equation:

$$(D * F)[h][w] = \sum_{r=0}^{R-1} \sum_{s=0}^{S-1} D[h+r][w+s]F[r][s] \quad (2)$$

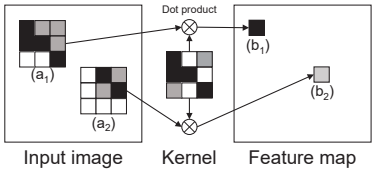


Fig. 1: 2D convolution.

Such a 2D discrete convolution is widely used in image processing and sometimes called an *image convolution*. An image, $D[h][w]$ in Equation 2, is treated as a function of 2D pixel coordinates in an image convolution. $F[h][w]$ in Equation 2 is called a filter or a kernel. The result of the 2D convolution generates a feature map as shown in Figure 1. The size of a filter ($R \times S$) is typically much smaller than the input image size ($H \times W$). For a given filter, input image regions (*e.g.*, a_1 and a_2) with the same size and dimension as the kernel are point-wisely multiplied with the kernel. A *feature map* consists of all the results (*e.g.*, b_1 and b_2) of such multiplications. A high value of the convolution implies that the corresponding region in the input image has a high degree of similarity to the kernel.

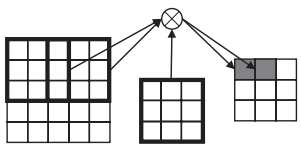


Fig. 2: Convolution with a stride of two.

TABLE I: Deep Learning Frameworks Used

Framework	Version	Multi-GPUs (parallelism)	Operating system	User interface	Libraries used
Caffe	1.0.0-rc3	Data	Linux	MATLAB, protobuf, Python	CUDA 7.5, cuDNN 5.0.5
CNTK	1.7.2	Data	Linux, Windows	BrainScript, C++, C#	CUDA 7.5, cuDNN 5.0.5
TensorFlow	0.10.0rc0	Data, Model	Linux	Python, C++	CUDA 7.5, cuDNN 5.0.5
Theano	0.8.2	–	Linux	Python	CUDA 7.5, Lasagne 0.2, cuDNN 5.0.5
Torch	7	Data, Model	Linux	LuaJIT	CUDA 7.5, cuda-convnet3, cuDNN 5.0.5, cuDNN 5.1.3, ccn2.torch

*All the frameworks spupport a single GPU.

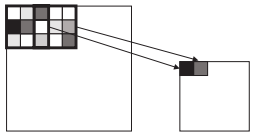


Fig. 3: 3×3 max pooling with a stride of two.

Since the convolution operation contains a large amount of computation, there are many techniques introduced to reduce it. A representative technique is applying the 2D convolution to the input image with a stride as shown in Figure 2. A strided convolution performs down-sampling by sampling only every s pixel in each direction of the input image, where s is the value of the stride. Thus, the resulting feature-map size is smaller than the input image size.

Another technique is *pooling*. Pooling also performs down-sampling and reduces the amount of computation. It identifies a representative pixel for a pixel region to reduce the size of the input. For example, Figure 3 shows 3×3 *max pooling* with a stride of two. It divides the input in 3×3-pixel regions with a stride of two. For each region, it selects a pixel that has the maximum value in the region as the representative.

C. Convolutional Neural Networks

A *convolutional neural network* (CNN) is an artificial neural network using convolutional filters to extract features from its input. In a CNN, a layer that performs 2D convolutions is called a *convolutional layer*. Since a filter extracts a feature from the input image, a typical convolution layer extracts multiple features from an input image using $N(\geq 1)$ filters, resulting in N feature maps. Each of the feature maps is also called a *channel*. The training stage of the CNN makes it learn a filter for each of the features.

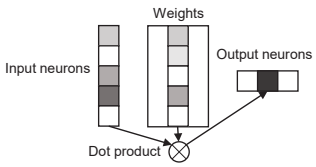


Fig. 4: Computation in the fully connected layer.

A *fully connected layer* in a CNN combines the results of convolutions. As shown in Figure 4, it consists of input neurons, output neurons, and weights that represent the relationship between the input and output. It performs matrix (the weights) and vector (the input) multiplication and generates the output.

AlexNet. Figure 5 shows a representative CNN, AlexNet [1]. It is one of the earliest successful CNNs that perform image recognition tasks using the ImageNet dataset [7]. It uses five convolution layers (conv1, conv2, conv3, conv4, and conv5) and three max pooling layers (pool1, pool2, and pool3) to extract features. In addition, there are three fully connected layers (fc1, fc2, and fc3) for image classification. Each layer uses the rectified linear unit (ReLU) for nonlinear neuron activation.

TABLE II: Configuration of AlexNet

Layer name	Kernel size (pooling size) / stride	Output size	Number of parameters	Number of FP operations
conv1	11 x 11 / 4	96 x 55 x 55	35K	55G
pool1	3 x 3 / 2	96 x 27 x 27		
conv2	5 x 5 / 1	256 x 27 x 27	614K	227G
pool2	3 x 3 / 2	256 x 13 x 13		
conv3	3 x 3 / 1	384 x 13 x 13	885K	65G
conv4	3 x 3 / 1	384 x 13 x 13	1.3M	98G
conv5	3 x 3 / 1	256 x 13 x 13	885K	65G
pool3	3 x 3 / 2	256 x 6 x 6		
fc1		4096	37M	74M
fc2		4096	16M	32M
fc3		1000	4M	8M
softmax		1000		

Convolution layers act as feature extractors while fully connected layers act as classifiers. Convolution layers extract feature map from the input data. Each convolution layer generates a feature map in 3D tensor format and feeds it into the next layer. Then the fully connected layers take the last feature map as a flattened vector input and create a 1000 dimensional vector as an output. After normalization with softmax layer, each dimension in normalized output vector refers to the probability of the image being associated with each image class.

Although AlexNet is old and most state-of-the-art CNN applications do not use AlexNet model, it has been frequently used for benchmarking purpose because it contains most of contemporary CNN components. On the other hand, other frequently used CNN models such as GoogLeNet and VGGNet lack some components in AlexNet. GoogLeNet [29] do not use fully connected layer in order to reduce parameters. VGGNet use only 3x3 kernels for convolution layers while AlexNet use 5x5 and 11x11 kernels as well. Layer-wise performance analysis of AlexNet provides sufficient information to predict performance behavior of other CNN applications.

The detailed configuration of the AlexNet model in this paper is presented in Table II. The original AlexNet model includes local response normalization (LRN) layers, but we exclude them in this paper since LRN is very rarely used in current CNN applications.

Training a CNN. Training a CNN is a supervised learning process using training input images and correct class labels that are associated with the images. Since training a CNN is the most time consuming process, it is very important to reduce the training time of a CNN by efficiently implementing the CNN.

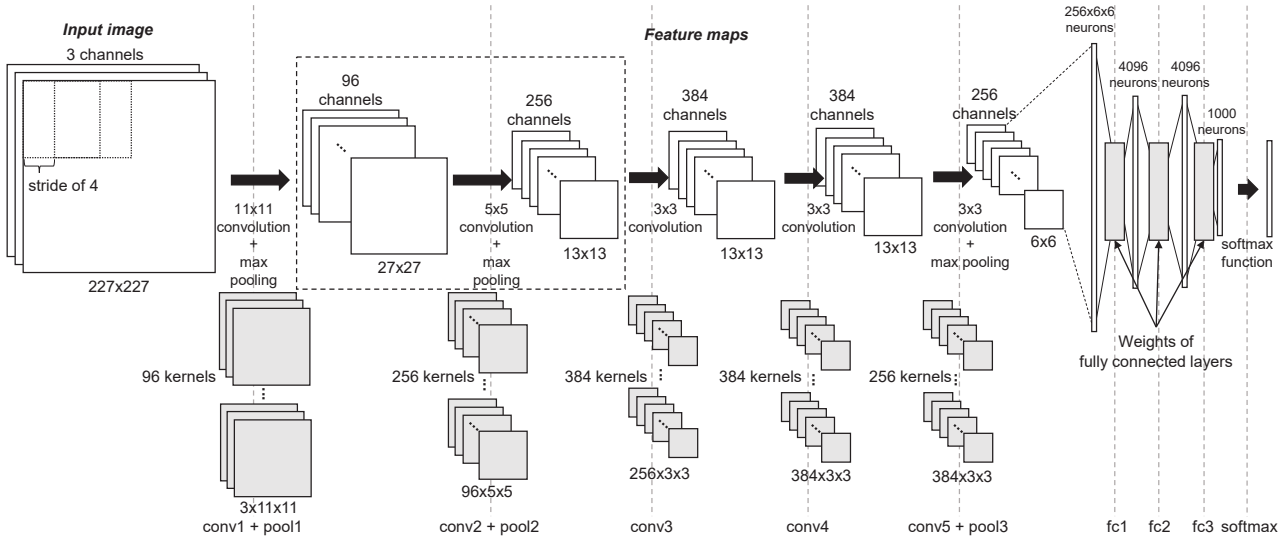


Fig. 5: Organization of AlexNet.

The training phase has two stages: *forward computation* and *backward computation*. For a given training input image, the forward computation stage goes through the CNN layers from the first to the last and obtains the output from the last layer. For example, AlexNet determines the class to which the training input image belongs. After computing the difference between the output and the correct label, the backward computation stage obtains the values that need to be added to the network parameters (*e.g.*, weights) of the last fully connected layer by computing gradients. After updating the parameters in the last layer with the gradients, these values are backward propagated to the previous layers (*i.e.*, *backpropagation* is performed) to adjust their parameters in the backward computation stage. The gradients are computed in the direction of minimizing the difference between the output of a layer and the correct answer for each layer.

Batch processing. When a CNN is trained, training input images are divided into sets of images, each of which is called a *batch*. A batch is processed by the CNN at a time. This is because the stochastic gradient decent technique (SGD) [30] that is typically used to obtain the gradients can be easily parallelized across different images in the same batch.

4D tensor format. Since multiple 2D feature maps (or images) are typically processed in a layer at a time, the feature map data in a CNN can be treated as four-dimensional tensors $< N, C, H, W >$, where N, C, H, and W are the number of images in a batch, the number of channels (*i.e.*, feature maps), the height of a feature map (or an image), and the width of a feature map (or an image), respectively. Since the 4D tensor data are stored in the order of the dimensions N, C, H, and W, they are called $NCHW$ 4D tensors. The computational complexity of convolving a $NCHW$ tensor with $K \times R \times S$ 3D kernel is $O(K \times CRS \times NHW)$.

D. Convolution Algorithms

Direct convolution. Since the efficiency of computing a convolution is important to CNNs, several methods have been developed to efficiently implement the convolution operation. Directly computing the convolution (we call it *direct convolution* in this paper) using Equation 2 is the most straightforward way to perform convolution. Cuda-convnet [22] is a widely used direct convolution library.

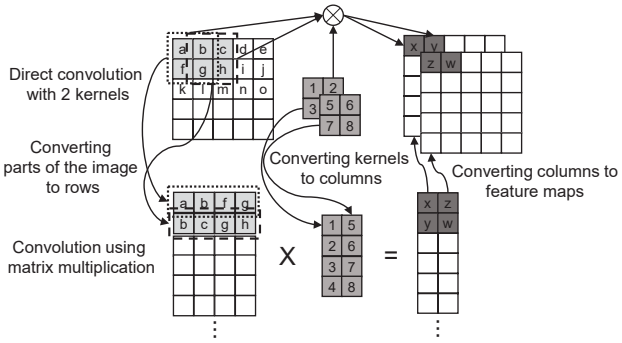


Fig. 6: Convolution using matrix multiplication.

Using matrix multiplication. cuDNN [14], a DNN library from NVIDIA, treats the convolution as matrix multiplication (*e.g.*, GEMM [31]). Figure 6 illustrates the process. It convolves a 5×5 image with two 2×2 kernels and obtains two 5×5 feature maps. When the input image has multiple channels, a row in the input matrix contiguously contains pixels from the multiple channels. When the number of input channels, the size of a kernel, the number of kernels, and the input image size are C, $R\times S$, K, and $H\times W$, respectively, the sizes of the input matrix and the kernel matrices become $CRS \times WH$ and $K \times CRS$, respectively. Since the complexity of the matrix multiplication

is $O(K \times CRS \times WH)$, when the number of images in a batch is N, the complexity becomes $O(K \times CRS \times NWH)$.

Interestingly, the computational complexity of this matrix multiplication method is the same as that of the direct convolution. However, matrix multiplication can be easily parallelized using highly efficient BLAS libraries [31]. Moreover, it enables exploiting the GPU local memory that has low latency and high bandwidth. cuDNN performs matrix multiplication by applying tiling to the matrices in the GPU local memory. This method scales well with a small batch size and can be used on all types of convolution layers.

Using FFT. The convolution operation can be implemented using a fast Fourier transform (FFT) algorithm to reduce computational complexity [15]. The complexity of FFT convolution is $O(K \times CWW \times \log W)$ that does not depend on the kernel size, $R \times S$. However, it requires more memory space because filters need to be padded to the dimension of the input. Another restriction is that it can be applied only to convolutions with the stride of one.

Using the Winograd algorithm. Winograd convolution is based on GEMM [31]. It reduces the complexity using Winograd's minimal filtering algorithm [17]. It is similar to the well known Strassen's algorithm [32] for matrix multiplication. It reduces multiplication operations significantly when the kernel size is fixed. Thus, a different kernel size requires a different minimal filtering algorithm. The minimal filtering algorithm for 4×4 tiled matrix can reduce 12 multiplications to 6. Nesting the minimal filtering algorithm twice would reduce the algorithm complexity by a factor of 4 [17]. cuDNN 5.1 supports Winograd convolution only for the filter sizes of 3×3 and 5×5.

E. Multi-GPU Support

Multi-GPU support for DNNs can be implemented by exploiting *data parallelism* or *model parallelism* [33]. Exploiting data parallelism means that input images in a batch are divided and distributed across multiple GPUs and processed by all the GPUs. Thus, all the GPUs have the entire network parameters. In the backward computation stage, each GPU computes its own gradients for the inputs assigned to it, then a single device (a CPU or a GPU) combines the gradients computed by all the GPUs and performs the SGD. Network parameters are updated with the result of the SGD, and they are distributed across the GPUs again.

The communication cost in data parallelism depends on the number of network parameters. Since AlexNet has 62M parameters and their size is 250MB, each iteration (an iteration processes a batch) needs to transfer approximately 250MB of data per GPU. Quantization methods that approximate the value of a floating-point number with an integer can reduce the size of the data transfer [34]. CNTK provides 1-bit SGD method that quantizes 32-bit gradient values into a single bit with some accuracy loss [35].

On the other hand, the model parallelism makes users to divide and distribute the network itself across GPUs. Since parameter updates can be done on each GPU, only a small amount of data needs to be communicated between GPUs. A convolution layer using carefully designed model parallelism typically outperforms data parallelism [18].

TensorFlow and Torch support both data parallelism and model parallelism. While Caffe and CNTK support only data parallelism, the multi-GPU support of Theano is still in an experimental stage. Thus, we compare only the efficiency of data parallelism between Caffe, TensorFlow, Torch and CNTK for multiple GPUs.

F. Related Work

Since it has been only a few years since deep learning frameworks were introduced to public, not that many previous studies compare their performance. Recently, some comparative studies for the deep learning frameworks are performed by Bahrampour *et al.* [19] and Shi *et al.* [20]. However, they show only the entire execution times of various DNN models built by the deep learning frameworks. They do not identify performance bottlenecks and reasons for performance differences. Moreover, they use cuDNN version R4 that does not support the Winograd convolution algorithm, and their work does not cover CNTK that was most recently introduced to public.

A benchmark result for some CNN frameworks is publicly available on GitHub [36]. It reports forward and backward computation times of a CNN model. The latest result was produced with cuDNN version R4, while the most recent cuDNN version is R5.1. This paper uses cuDNN version R5 and R5.1.

III. Experimental Methodology

To see what is exactly happening under the hood, we conduct three experiments with AlexNet models built by aforementioned five deep learning frameworks. First, we measure the execution time of the AlexNet model for each framework to compare their characteristics. Unlike previous work, we measure layer-wise and GPU kernel-wise execution times as well as the execution time of a single batch. Second, we obtain performance characteristics of different convolution algorithms by profiling each GPU kernel implementation. Finally, we measure the training speed of each AlexNet implementation with multiple GPUs. Based on the experiment, we suggest possible optimization techniques to improve the scalability of CNNs.

TABLE III: System Configuration

CPU	2 x Intel Xeon E5 2650@2.0GHz
GPU	4 x NVIDIA Titan X (GM200)
Main memory	128GB DDR3
GPU memory	4 x 12GB GDDR5
Operating system	CentOS 7.2.1511 (Linux 3.10.0-327)

Our experiments are performed with a system described in Table III. The versions of the deep learning frameworks and libraries used are described in Table I. Torch is currently the only framework that officially supports cuDNN R5.1. In addition, the newest Cuda-convnet3 is supported only by the ccn2.torch module in Torch. The comparison between the frameworks is done with cuDNN R5 and the comparison between different

convolution algorithms is performed using cuDNN R5.1 with Torch7.

AlexNet models are trained for many iterations and profiled. After they are stabilized, an iteration for a single batch is selected for our analysis. Model parameters are carefully equalized across different frameworks to remove the effect from things other than the frameworks themselves.

IV. Charactrization on a Single GPU

In this section, we characterize the five deep learning frameworks on a single GPU. The measurement of the layer-wise execution time of the AlexNet model for each framework is shown in Figure 7. The batch size of 256 was used in the experiment. The string in parentheses after the framework name stands for the framework configuration when the AlexNet model is built. No parentheses means the default options. A bar shows the breakdown of the total execution time into that of each layer. A layer name suffixed with f stands for the forward computation stage, and b for the backward computation stage.

A. Options for Convolution Algorithms

Caffe. Caffe does not have any explicit option to choose convolution algorithms. Instead, it exploits cuDNN's heuristics that try to determine the best suited algorithm under the given specification of the model. Surprisingly, Caffe does not have an appropriate memory management technique yet. As a result, algorithms that require a smaller memory space, such as GEMM and Winograd, can be inappropriately selected resulting in low performance even if Caffe exploits cuDNN's heuristics.

TensorFlow. TensorFlow also does not provide any option to choose convolution algorithms. Unlike Caffe, however, it executes all available algorithms in the first run by itself. Then, the fastest algorithm for each layer is executed in subsequent runs. The FFT algorithm is chosen for all the convolution layers except conv1, where FFT cannot be used because of the stride size (4). Since the set of algorithms that can be chosen is fixed and hardwired, recently added options of cuDNN R5.1 are not included in the set.

Theano. On the other hand, Theano provides full accesses to the choices of convolution algorithms. Users can specify a specific convolution algorithm globally or in a layer-by-layer manner. There are two types of GEMM implementations in cuDNN: explicit and implicit. Theano selects the explicit GEMM by default. When a convolution algorithm is given as a global option, and the algorithm does not match the condition for a layer, the implicit GEMM is chosen as a fallback for the layer. However, the implicit GEMM is usually slower than the explicit GEMM. Thus, it is better to give layer-wise option to make Theano not to choose the implicit GEMM. Theano (FFT) in Figure 7 stands for the AlexNet model built by Theano with a global option for the FFT algorithm. However, conv1 cannot use the FFT algorithm because of its stride size. Instead, the implicit GEMM is chosen for conv1. This is the reason why conv1 of Theano (FFT) in Figure 7 is slower than those of Theano (guess_once), Theano (time_once), and even Theano.

Unlike Caffe, Theano properly exploits cuDNN's heuristics when the guess_once option is given. The guess_once option makes Theano behave like Caffe where cuDNN's heuristics determine the best suited algorithm. The time_once option in Theano exploits cuDNN's another functionality that executes all available convolution algorithms and choose the fastest one. Unlike TensorFlow, the set of algorithms in Theano is not hardwired. When the time_once option is on, Theano uses GEMM in the first layer. It uses FFT or Winograd for the rest of the convolution layers. When the guess_once option is on, the selection of an algorithm for each layer is slightly different from time_once, but the execution time is almost the same as that of time_once. This implies that cuDNN's heuristics are quite reliable.

TABLE IV: Other GPU Kernels

	Caffe	CNTK	TensorFlow	Theano	Torch
Bias addition	cuDNN 2.64 ms	CNTK 4.74 ms	TensorFlow 2.84 ms	Theano 7.88 ms	cuDNN 2.63 ms
ReLU activation	cuDNN 2.56 ms	CNTK 2.26 ms	Eigen 2.43 ms	Theano 4.59 ms	cuDNN 2.56 ms

Although Theano executes the fastest algorithm for convolution computation, it does not have the fastest convolution layer because of other computations for bias addition and ReLU activation. Table IV lists computations included in a convolution layer, but not directly participate in convolution. It also shows the sources of the implementations. Their execution time is measured in conv1. Theano uses GPU implementations that are dynamically compiled at run time, and they are noticeably slower than those in other frameworks.

Torch. Like Theano, Torch provides a full control over algorithm choices. Its cuDNN backend, cudnn.torch, has the cudnn.benchmark option that is the same as time_once in Theano. When benchmark option is on, Torch is the fastest among the frameworks.

CNTK. CNTK does not provide any options for algorithm choices. Instead, similar to Theano and Torch, CNTK uses cuDNN's functionality to choose the fastest kernel by executing all the convolution algorithms. CNTK is not the fastest though because its bias addition implementation makes it run slow.

B. Effect of Data Layout

We also find that data layout affects performance a lot. The most noticeable one is the tensor format. Caffe, CNTK, Theano, and Torch use the $NCHW$ 4D tensor format mentioned in Section II-C.

Even though TensorFlow supports the $NCHW$ format, it seems to prefer the $NHWC$ format. Sample code in TensorFlow uses the $NHWC$ format, and many function in TensorFlow, such as conv2d, uses it as the default argument format. Since channel data are stored in the last dimension in the $NHWC$ format, the performance will be better off using it when there are innermost channel-wise operations. However, some convolution algorithms, such as FFT, in cuDNN support only $NCHW$ format. Moreover, by default, TensorFlow performs conversion between $NHWC$ and $NCHW$ even if it

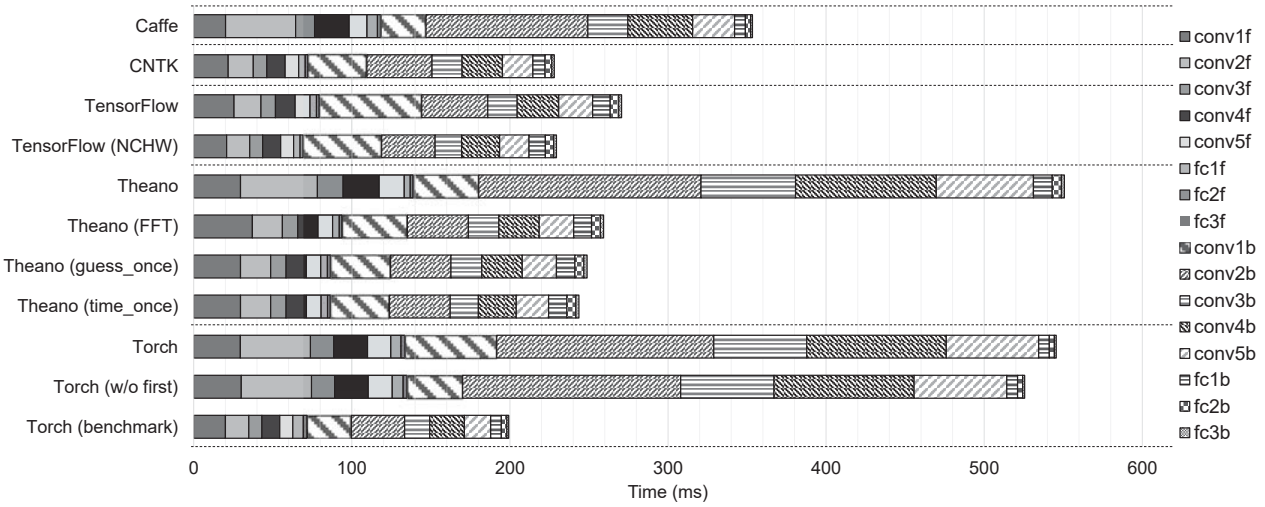


Fig. 7: The execution time of training AlexNet with a batch of input images for different deep learning frameworks.

uses algorithms that support the $NHWC$ format. Thus, using the $NHWC$ tensor format introduces significant conversion overhead before and after convolution computation. In Figure 7, just changing the tensor format from $NHWC$ to $NCHW$ leads to about 15% performance improvement.

C. Unnecessary Backpropagation

conv1 in AlexNet does not have its previous layer. Thus, there is no need to perform the backpropagation of gradients computed in conv1 . Caffe, CNTK, and Theano automatically omit the backpropagation. Torch omits it when gradInput of conv1 is set to nil.

On the contrary, TensorFlow always performs the backpropagation of conv1 to prevent non-reproducibility caused by race conditions (gate_gradients parameter in tf.train.Optimizer.compute_gradients). This makes TensorFlow slower than Torch even if Torch has almost the same algorithms as those of TensorFlow for the convolution layers. For this reason, in Figure 7, we see that the execution time of conv1b in TensorFlow (NCHW) is significantly different from that in Torch (benchmark). However, the backpropagation can be disabled at the expense of reproducibility and the accuracy of gradient computation, although we left it enabled in our experiment for Figure 7.

Overall, we observe that slight changes in framework options result in large performance differences. Options for convolution algorithm choices, data layout, and disabling useless backpropagation are most influential. We can increase the speed of training a CNN model up to 2X by just changing framework options. We do not need to modify the source code of the framework at all. For example, Theano (FFT), Theano (guess_once), and Theano (time_once) are 2X faster than Theano. Torch (benchmark) are also 2X faster than Torch.

V. Characterization of Convolution Algorithms

In this section, we compare the performance of different convolution algorithms on a single GPU. These algorithms were introduced in Section II-D. We used Cuda-convnet for Direct, and cuDNN for GEMM, FFT, and Winograd.

A. Execution Time of Convolution Algorithms

Figure 8 shows execution time comparisons between different convolution algorithms (GEMM, Direct, FFT, and Winograd) on a single GPU. We vary the batch size from 32 to 256. Since the convolution layers take most of the execution time in a CNN, using an efficient convolution algorithm is an important design decision. All comparisons are done with the AlexNet model built by Torch7 because it is the only framework that officially supports the newest versions of cuDNN and cuda-convnet3. Randomly generated images are used for the input to remove the effect of I/O latency. The forward and backward computation times are measured and averaged for 100 iterations (*i.e.*, batches).

Winograd and FFT perform better than Direct or GEMM most of the time as shown in Figure 8a and Figure 8b. Since many recent CNN models use 3×3 kernel for convolution layers [2], the forward computation times of conv3, conv4 and conv5 are separately measured in Figure 8c.

The total training time (the forward computation time + the backward computation time) of FFT is the best for all batch sizes. However, for 3×3 convolution kernels with a small batch size, Winograd performs better than FFT while FFT scales better with a large batch size. In addition, Direct (*i.e.*, Cuda-convnet) scales poorly when the batch size is smaller than 128 while GEMM scales almost linearly.

Figure 8d shows peak GPU memory usage for each convolution algorithms. FFT occupies the most GPU memory, using about 20% more memory than GEMM.

B. Layer-wise Analysis

Figure 9 shows layer-wise analysis of different convolution algorithms on a single GPU. The batch size is set to 256, and the NVIDIA nvprof profiler is used to gather statistics. FFT and

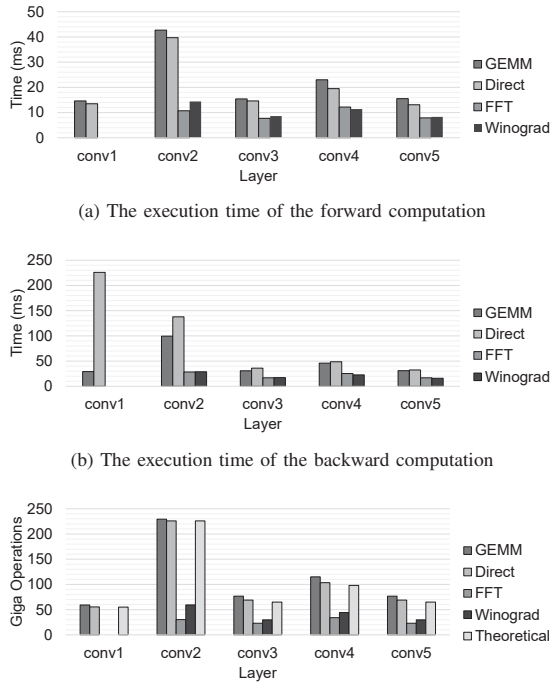
(a) The execution time of the forward computation

(b) The execution time of the backward computation

(c) The execution time of the forward computation from conv3 to conv5

(d) Peak GPU memory usage

Fig. 8: Execution time and memory usage between different convolution algorithms.



(a) The execution time of the forward computation

(b) The execution time of the backward computation

(c) FP operation count in the forward computation

Fig. 9: Layerwise analysis of different convolution algorithms.

Winograd cannot be applied to the conv1 since it has a stride size of 4. As shown in Figure 9b, the main performance limiter of Direct (*i.e.*,cuda-convnet3) is the backward computation in conv1 layer. The reason of the slowdown is low parallelism in Direct that implement conv1 layer. While NVIDIA Titan X has 3072 CUDA cores, the number of CUDA threads in the implementation is only 1024. This makes Direct slower than other algorithms in conv1.

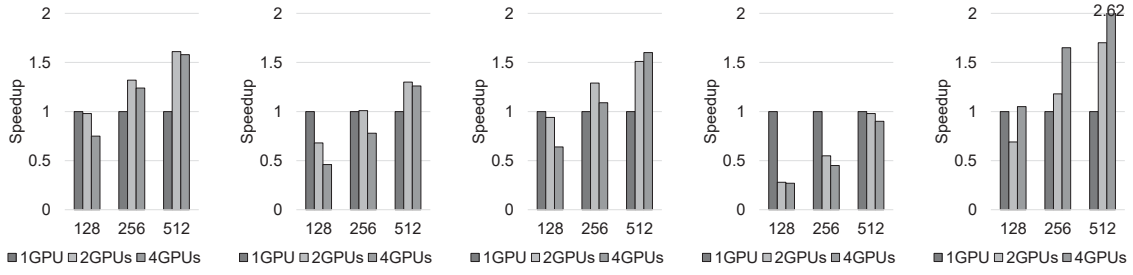
Floating-point operation counts. Figure 9c compares floating-point (FP) operation counts of different algorithms in the forward computation. The count in the backward computation is double the count in forward computation. The numbers are measured with the NVIDIA nvprof profiler. The result tells us that FFT is usually the fastest because of much less FP operation count (*i.e.*, much less algorithm complexity). Especially, FFT is much faster than others in conv2 with 5×5 convolution filter because the complexity of FFT does not depend on the filter size. Winograd also reduces FP operation count by more than a half. Thus, its performance is comparable to FFT in the layers with 3×3 convolution filters (conv3, conv4, and conv5).

Overall, FFT and Winograd outperform GEMM or Direct because of their reduced algorithm complexity. FFT scales better with a large batch and filter size while Winograd scales better with a small batch and filter size. Carefully choosing a convolution algorithm can make the CNN more than 4X faster in convolution layers.

VI. Multi-GPU Support

In this section, we characterize the performance of the AlexNet models on multiple GPUs. The AlexNet models are trained on one, two, and four GPUs. We exploit the data parallelism (explained in Section II-E) already available in the frameworks. Since Theano does not support multiple GPUs, we compare only Caffe, CNTK, TensorFlow, and Torch. To train the AlexNet models, we select the fastest option based on the characteristics on a single GPU in Section IV

To exploit data parallelism, a GPU (say GPU_0) gathers intermediate gradient values to update network parameters stored in other GPUs (GPU_1, GPU_2, and GPU_3, assuming a total of four GPUs). If gradients are gathered sequentially by going through each GPU, the number of communications for gradient transfer between GPU_0 and other GPUs is $O(N-1)$, where N is the number of GPUs. However, if gradients are gathered with



(a) Caffe (b) Torch (c) TensorFlow (d) CNTK (e) CNTK 1bit-SGD

Fig. 10: Speedup (over a single GPU) of multi-GPU training of the AlexNet models.

the parallel reduction algorithm [37], it is $O(\log N)$ under the assumption that data transfers can be parallelized if the source and destination GPUs are distinct. Since a different GPU uses a different set of PCI-E lanes, our system supports this parallel reduction algorithm.

Ideally, the training time of AlexNet is expected to be reduced to $\frac{1}{N}$ when N GPUs are used. However, this is not true because of the data transfer caused by the gradient data exchange. Since the size of the gradient data in our AlexNet model is about 250MB, the gradient transfer between two GPUs takes about 45ms. This is not negligible considering that one forward and backward computation takes about 200ms with a batch size of 256.

Figure 10 shows the speedup obtained by multi-GPU training of our AlexNet models with different batch sizes and numbers of GPUs. We see that a bigger batch size makes the speedup higher. A batch size of 128 makes Caffe and Torch on two or four GPUs much slower than on a single GPU. A bigger batch size makes the computation time in each GPU longer. However, the number and size of data transfer is fixed because the number of network parameters remains the same. Thus, it is beneficial to use a bigger batch size.

We also see that the scalability of each framework is quite poor. The speedup of using four GPUs is slower than or comparable to that of using two GPUs for all frameworks. The reason is, gradient transfer when using four GPUs still takes twice longer than using two GPUs, even though we use the $O(logN)$ algorithm to transfer data between GPUs.

Torch and TensorFlow have comparable execution time on a single GPU. However, TensorFlow achieves higher speedup than Torch. Since TensorFlow handles gradients of each layer as an individual tensor, the gradients of each layer are transferred as soon as the backward computation of that layer finishes. On the other hand, Torch (and Caffe) references gradients of the entire network as a whole. Thus, it starts data transfer after all backward computations finish. That is, TensorFlow has more data-transfer-and-computation overlap.

CNTK is special in terms of multi-GPU support. Although its data transfers can be parallelized, gradients are gathered by the CPU, and this takes much longer than the computation time in GPUs. Thus, using any number of GPUs is slower than a single GPU. However, CNTK can exploit 1bit-SGD [35]. When 1bit-SGD is used, only one bit per gradient is transferred, reducing data transfers and CPU computation by a factor of 32. This makes CNTK scale almost linearly at the cost of slow convergence.

Overall, we observe that the current multi-GPU scalability of the frameworks has much room to be improved. TensorFlow-like data transfer and computation overlapping will be helpful to improve the performance. Reducing the size of gradients by approximating the exact value with less accuracy (*e.g.*, using the half-precision FP format or only 1-bit like CNTK) will also improve scalability a lot. Reducing the number of gradients by resizing and pruning the CNN model will also work, especially in fully-connected layers because they typically contain more than 90% of network parameters.

VII. Conclusions and Future work

In this study, we characterize deep learning frameworks and libraries to build CNN models using the AlexNet model trained on a single GPU and multiple GPUs. Previous studies, such as [7] and [36], address performance differences are caused by framework differences. However, the performance characteristics indicate that difference between the frameworks are mainly caused by backends, especially convolution algorithms in the convolution layers. By just adjusting options provided by the frameworks, we achieve about 2X speedup without modifying any source code. Thus, providing options for convolution algorithm choices would be an important criterion to determine a deep learning framework to use. We also find that data layout and disabling useless backpropagation also affect the performance a lot. Using FFT and the Winograd algorithm in convolution is more beneficial than using direct computation and matrix multiplication. Carefully choosing a convolution algorithm can make a CNN model more than 4X faster in convolution layers. Our multi-GPU analysis indicates that the scalability of data parallelism in CNN models is bounded by the data transfer overhead of network parameters between GPUs. Thus, reducing the amount of parameter transfer would achieve better scalability when exploiting data parallelism. We also observe that the current multi-GPU scalabilities of the frameworks have much room to be improved. Unlike the single-GPU context, framework differences are more important in

the multi-GPU context since the frameworks choose different approaches for parallelization.

We are also interested in a recurrent neural network (RNN), which is mainly composed of fully connected layers and expected to have different characteristics from CNNs. Analyzing the characteristics of RNN is one of our future research topics.

ACKNOWLEDGMENT

This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Ministry of Science, ICT & Future Planning (MSIP) (No. 2013R1A3A2003664), PF Class Heterogeneous High Performance Computer Development through the NRF funded by the MSIP (NRF-2016M3C4A7952587), and BK21 Plus for Pioneers in Innovative Computing (Dept. of Computer Science and Engineering, SNU) through the NRF funded by the Ministry of Education (21A20151113068). ICT at Seoul National University provided research facilities for this study.

REFERENCES

[1] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, 2012, pp. 1097–1105.

[2] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *CoRR*, vol. abs/1409.1556, 2014.

[3] R. Girshick, J. Donahue, T. Darrell, and J. Malik, "Rich feature hierarchies for accurate object detection and semantic segmentation," in *Computer Vision and Pattern Recognition*, 2014.

[4] O. Abdel-Hamid, A. r. Mohamed, H. Jiang, L. Deng, G. Penn, and D. Yu, "Convolutional neural networks for speech recognition," *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, vol. 22, no. 10, pp. 1533–1545, Oct 2014.

[5] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," *CoRR*, vol. abs/1409.0473, 2014. [Online]. Available: http://arxiv.org/abs/1409.0473

[6] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, Nov 1998.

[7] O. R. et al., "Imagenet large scale visual recognition challenge," *CoRR*, vol. abs/1409.0575, 2014. [Online]. Available: http://arxiv.org/abs/1409.0575

[8] M. Everingham, L. Van Gool, C. K. I. Williams, J. Winn, and A. Zisserman, "The pascal visual object classes (voc) challenge," *International Journal of Computer Vision*, vol. 88, no. 2, pp. 303–338, Jun. 2010.

[9] R. Al-Rfou, G. Alain, A. Almahairi, and et al., "Theano: A python framework for fast computation of mathematical expressions," *CoRR*, vol. abs/1605.02688, 2016. [Online]. Available: http://arxiv.org/abs/1605.02688

[10] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," *arXiv preprint arXiv:1408.5093*, 2014.

[11] M. A. et al., "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, software available from tensorflow.org. [Online]. Available: http://tensorflow.org/

[12] R. Collobert, K. Kavukcuoglu, and C. Farabet, "Torch7: A matlab-like environment for machine learning," in *BigLearn, NIPS Workshop*, 2011.

[13] D. Y. et al., "An introduction to computational networks and the computational network toolkit," Tech. Rep., October 2014. [Online]. Available: https://www.microsoft.com/en-us/research/publication/an-introduction-to-computational-networks-and-the-computational-network-toolkit/

[14] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer, "cudnn: Efficient primitives for deep learning," *CoRR*, vol. abs/1410.0759, 2014. [Online]. Available: http://arxiv.org/abs/1410.0759

[15] M. Mathieu, M. Henaff, and Y. LeCun, "Fast training of convolutional networks through ffts," *CoRR*, vol. abs/1312.5851, 2013. [Online]. Available: http://arxiv.org/abs/1312.5851

[16] N. Vasilache, J. Johnson, M. Mathieu, S. Chintala, S. Piantino, and Y. LeCun, "Fast convolutional nets with fbfft: A GPU performance evaluation," *CoRR*, vol. abs/1412.7580, 2014. [Online]. Available: http://arxiv.org/abs/1412.7580

[17] A. Lavin, "Fast algorithms for convolutional neural networks," *CoRR*, vol. abs/1509.09308, 2015. [Online]. Available: http://arxiv.org/abs/1509.09308

[18] O. Yadan, K. Adams, Y. Taigman, and M. Ranzato, "Multi-gpu training of convnets," *CoRR*, vol. abs/1312.5853, 2013. [Online]. Available: http://arxiv.org/abs/1312.5853

[19] S. Bahrampour, N. Ramakrishnan, L. Schott, and M. Shah, "Comparative study of caffe, neon, theano, and torch for deep learning," *CoRR*, vol. abs/1511.06435, 2015. [Online]. Available: http://arxiv.org/abs/1511.06435

[20] S. Shi, Q. Wang, P. Xu, and X. Chu, "Benchmarking state-of-the-art deep learning software tools," *CoRR*, vol. abs/1608.07249, 2016. [Online]. Available: http://arxiv.org/abs/1608.07249

[21] Github. [Online]. Available: https://github.com

[22] cuda-convnet. [Online]. Available: https://code.google.com/p/cuda-convnet/

[23] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei, "ImageNet Large Scale Visual Recognition Challenge," *International Journal of Computer Vision (IJCV)*, vol. 115, no. 3, pp. 211–252, 2015.

[24] I. J. Goodfellow, D. Warde-Farley, P. Lamblin, V. Dumoulin, M. Mirza, R. Pascanu, J. Bergstra, F. Bastien, and Y. Bengio, "Pylearn2: a machine learning research library," *arXiv preprint arXiv:1308.4214*, 2013. [Online]. Available: http://arxiv.org/abs/1308.4214

[25] F. Chollet, "Keras," https://github.com/fchollet/keras, 2015.

[26] Lasagne documentation. [Online]. Available: https://lasagne.readthedocs.io/

[27] M. B. et al., "End to end learning for self-driving cars," *CoRR*, vol. abs/1604.07316, 2016. [Online]. Available: http://arxiv.org/abs/1604.07316

[28] V. e. a. Mnih, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, Feb. 2015.

[29] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. E. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," *CoRR*, vol. abs/1409.4842, 2014. [Online]. Available: http://arxiv.org/abs/1409.4842

[30] L. Bottou, "Online algorithms and stochastic approximations," in *Online Learning and Neural Networks*, D. Saad, Ed. Cambridge, UK: Cambridge University Press, 1998, revised, oct 2012. [Online]. Available: http://leon.bottou.org/papers/bottou-98x

[31] C. Nvidia, "Cublas library," *NVIDIA Corporation, Santa Clara, California*, vol. 15, p. 27, 2008.

[32] S. Winograd, *Arithmetic complexity of computations*. Siam, 1980, vol. 33.

[33] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, M. aurelio Ranzato, A. Senior, P. Tucker, K. Yang, Q. V. Le, and A. Y. Ng, "Large scale distributed deep networks," in *Advances in Neural Information Processing Systems 25*, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, Eds. Curran Associates, Inc., 2012, pp. 1223–1231.

[34] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding," *CoRR*, vol. abs/1510.00149, 2015. [Online]. Available: http://arxiv.org/abs/1510.00149

[35] F. Seide, H. Fu, J. Droppo, G. Li, and D. Yu, "1-bit stochastic gradient descent and application to data-parallel distributed training of speech dnns," in *Interspeech 2014*, September 2014.

[36] S. Chintala. convnet-benchmarks. [Online]. Available: https://github.com/soumith/convnet-benchmark://github.com/soumith/convnet-benchmarks

[37] M. Harris *et al.*, "Optimizing parallel reduction in cuda," *NVIDIA Developer Technology*, vol. 2, no. 4, 2007.