Performance Analysis of GPU-based Convolutional Neural Networks

Xiaqing Li[†‡§], Guangyan Zhang[†‡§], H. Howie Huang[¶], Zhufan Wang[†‡], Weimin Zheng[†‡]
[†]Department of Computer Science and Technology, Tsinghua University
[‡]Tsinghua National Laboratory for Information Science and Technology
[§]State Key Lab of Mathematical Engineering and Advanced Computing, Wuxi, China
[¶]Department of Electrical and Computer Engineering, George Washington University
Email: li-xq14@mails.tsinghua.edu.cn, gyzh@tsinghua.edu.cn, howie@gwu.edu
wang.zhufan1993@gmail.com, zwm-dcs@tsinghua.edu.cn

***Abstract*—As one of the most important deep learning models, convolutional neural networks (CNNs) have achieved great successes in a number of applications such as image classification, speech recognition and nature language understanding. Training CNNs on large data sets is computationally expensive, leading to a flurry of research and development of open-source parallel implementations on GPUs. However, few studies have been performed to evaluate the performance characteristics of those implementations. In this paper, we conduct a comprehensive comparison of these implementations over a wide range of parameter configurations, investigate potential performance bottlenecks and point out a number of opportunities for further optimization.**

***Index Terms*—Convolutional neural network, deep learning, GPU, performance evaluation, parallel computing.**

I. Introduction

Convolutional neural networks (CNNs) are important deep learning models that have achieved great successes in large scale image classifications [2], [9], [22], speech recognitions[3], [4] and nature language understanding [5], [6], [7]. This can be attributed to the advanced architecture of CNNs (such as AlexNet, VGGNet, GoogleNet and OverFeat) [2], [12], [15], [22], large labeled training samples [16] and powerful computing devices such as GPUs.

The training cost of CNNs is very high for two reasons. First, CNNs are getting more complicated due to increased depth and parameters. For example, AlexNet, the winner of ILSVRC-2012, has 8 layers (5 convolutional layers and 3 fully-connected layers) and more than 60 million parameters. VGGNet has 19 layers (16 convolutional layers and 3 fully-connected layers) and over 144 million parameters. Another recent model, GoogLeNet, is comprised of 22 layers with about 6.8 million parameters [15]. Training these large-scale CNNs requires thousands of iterations of forward and backward propagations, and therefore is much time-consuming.

Second, the training samples are getting much larger. One of the early CNNs, LeNet-5, was trained to recognize handwritten digits on MNIST data set, which only contains 60,000 images in the training set and 10,000 images in the testing set [8]. CIFAR-10 [11] dataset consists of 60,000 32×32 color images, including 50,000 training images and 10,000 testing images. In contrast, a larger dataset called ImageNet was provided in 2009, including more than 1.2 million high-resolution images. Driven by industry groups like Google, YouTube, Twitter and FaceBook, CNNs require to be trained on some very large datasets (e.g., text, audio and video). Again, training on those large-scale datasets requires significant runtime, and several weeks or months is not uncommon.

To address this challenge, using GPUs to accelerate the training process of CNNs is popular. During CNN training, the computation is inherently parallel and involves a massive amount of floating-point operations, e.g., matrix and vector operations. This computing pattern is well suitable for GPU computing model. Many of emerging deep learning frameworks are highly optimized on GPUs with the CUDA programming interface, including cuda-convnet [2], cuda-convnet2 [18], Theano [19], Torch [20], Decaf [21] and Caffe [23]. Most of these frameworks are open source and support one or multiple GPUs. Moreover, some GPU-optimized libraries are explored to accelerate CNNs, such as cuDNN [24] and fbfft [25].

However, few studies have been performed to enable a comprehensive evaluation on the performance characteristics of those implementations over a wide range of configurations. As our experiments and evaluations will show, each implementation has pros and cons, and there is no single implementation that performs well in all scenarios. The best performance is heavily dependent on different configurations.

The goal of this work is to assist practitioners identifying the implementations that best serve their CNN computation needs in different scenarios, and provide insights and suggestions to practitioners and pinpoint aspects for researchers who are interested in convolution optimization on GPUs. In this paper, we conduct a head-to-head comparison of their runtime to assist identifying the fastest implementation for a wide range of scenarios. Furthermore, we also examine their memory usage and shape limitation during GPU kernel execution. In addition, developing optimization schemes and implementations requires an understanding of how efficiently the computing power of GPUs has been exploited and where the potential performance bottlenecks of those implementations are. We thus conduct a performance profiling to study the intrinsic characteristics of those implementations on GPU over different typical configurations.

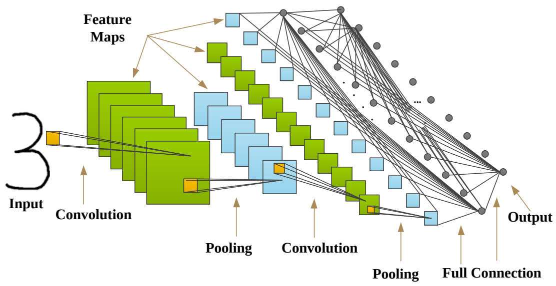

Fig. 1: A simple CNN architecture (LeNet-5).

The rest of this paper is organized as follows. In Section 2, we present an overview of the architecture of CNNs and three convolution strategies. In Section 3, we describe the experimental environment and evaluation methodology. In Section 4, we identify hotspot layers in CNNs and compare different implementations in the running time over a wide range of configurations. In Section 5, we analyze hotspot functions in the hotspot layers and evaluate the performance of each implementation on GPU. Finally, we conclude this paper in Section 6.

II. Background

Understanding the architecture of CNNs better is key to evaluation and optimization of the convolution implementations. In this section, we present an overview of the architecture of CNNs and discuss different convolution strategies that are adopted by typical CNN implementations.

A. Convolutional Neural Networks

The training process of CNNs is a typical feed-forward neural network, which applies BP algorithm to adjust learnable kernels so as to minimize the cost function. Convolutional neural network automatically provides some degree of shift and distortion invariance by three key ideas: local receptive field, shared weight, and pooling [26].

Convolutional layer is the central part in CNNs. In convolutional layer, each neuron of the same feature map applies the same weights over input data at all possible positions to extract the corresponding features. The convolved results are organized into a set of two dimensional feature maps. All of neurons in a feature map share the same weights, which are called *shared weights*. Each neuron of the current layer is connected to a local region of the previous layer. This connectivity with a local region is called a *local receptive filed* [26]. *Pooling* layers are optionally used after convolutional layers, and it aims to reduce the spatial size of feature map and to control the over-fitting problem to some extent.

We take Lenet-5 as a typical example to illustrate the architecture of CNNs. As shown in Figure 1, Lenet-5 is stacked by convolutional layer, pooling layer and two fully connected layers. The input image is first fed to input layer, and then is passed through a stack of convolutional and pooling layers. Repeat convolutions with the methods of local receptive field, shared weight and pooling, until the last convolutional layer holds a set of relatively high-level features. Finally, those high-level features are mapped to a probability vector over ten different classes in last two fully-connected layers.

B. Convolution Strategies

Recently, many deep learning frameworks and libraries have been developed to implement CNN on GPUs, e.g., cuda-convnet [2], cuda-convnet2 [18], Theano [19], Torch [20], Decaf [21], Overfeat [22], Caffe [23], cuDNN [24] and fbfft [25]. Since convolutional layers is the central part of CNNs, researchers devote most efforts into design and optimization of convolutional layers. In order to implement CNN, researchers have explored different kind of convolution strategies. However, mainstream CNN implementations follow three convolution strategies: direct convolution, unrolling-based convolution [32], [24], and FFT(Fast Fourier Transformation)-based convolution. These strategies are depicted as follows.

Direct Convolution. This is the traditional way to compute convolution. During direct convolution, a small window slides within an input feature map and a dot production between the filter bank and local patch of the input feature map is computed. The result of dot production is then passed into a non-linear activation function, e.g., $Sigmoid$ and $Tanh$. Outcome results from this activation function are organized into a new feature map as output. Repeating the above process for each filter bank, we can get a set of two-dimensional feature maps as the output of the convolutional layer. Presentative implementations of direct convolution include cuda-convnet2 [18], and Theano-legacy [31].

Unrolling Based Convolution. Unrolling-based convolution is a very efficient method on GPUs according to [32] [24]. The key idea behind unrolling convolution is to reshape the input and the filter bank to double large matrices. The local regions of input image are unrolled into columns and the filter banks are unrolled into rows using $im2col$. The final convolution can be converted into a clean and efficient matrix-matrix production by using highly-optimized libraries such as cuBLAS on GPUs [32]. Finally, the results should be remapped back to the proper dimension using $col2im$. Many new frameworks and libraries are developed based on this strategy, such as Caffe [23], Torch-cunn [20], Theano-CorrMM [19], and cuDNN [24].

FFT Based Convolution. This strategy is based on the convolution theorem that a discrete convolution in the spatial domain can be converted into the product of the Fourier domain. The performance of FFT-based convolution can be significantly improved thanks to its lower computation complexity. In general, FFT-based convolution can be implemented by three main steps. First, inputs and filter banks are transformed from the spatial domain to the Fourier domain with Fast Fourier Transformation (FFT). Second, those transformed matrices are multiplied in the Fourier domain. Finally, the product results are inversed from the Fourier domain to the spatial domain. This strategy is followed by fbfft [25], and Theano-fft [19].

III. Experimental Methodology

A. Experimental Environment

We evaluate CNN implementations on a CPU-GPU hybrid system. Ubuntu 14.04.1 is installed on a machine with Intel Xeon E5-2620 2.10 GHz 24 processor, 64GB main memory and 1TB hard disk. A single K40c GPU card is used in our experiments. We use openCV 2.4.8 and CUDA Toolkit 7.5.

The K40c GPU card has an excellent computing power due to its many-core architecture, large device memory, high memory bandwidth and floating point throughput. The K40c card consists of 15 Streaming Multiprocessors (SM), each SM with 192 processing units (a.k.a., CUDA cores). Each CUDA core can perform 2 floating-point operations per clock rate, and work at a maximum core clock rate of 745 MHz. Therefore, all the 2880 (15×192) CUDA cores provide a peak single-precision floating point performance of 4.29 TFLOPS.

Each SM has 256KB register files and 48KB on-chip memory. The card is also equipped with 12GB device memory and has 288 GB/s peak memory bandwidth. More details about CUDA and GPU can refer to [1].

B. Evaluation Methodology

We select Caffe [23], Torch-cunn [20], Theano-CorrMM [19], Theano-fft [19], cuDNN [24], cuda-convnet2 [18], and fbfft [25] as representative implementations in our evaluation. It should be noticed that we evaluate cuDNN-v3 in Caffe, fbfft in Torch and cuda-convnet2 with a Torch wrapper provided by convnet-benchmarks [28]. Our evaluation methodology can be categorized into two groups: high-level workload profiling and detailed performance profiling.

For high-level workload profiling, we analyze the workload from two aspects.

- We conduct a hotspot layer analysis for those CNN implementations by profiling four typical CNN models (i.e., ImageNet, GoogleNet, VGG, and Overfeat).
- For hotspot layers, we conduct a head-to-head performance comparison in forms of speed across those seven implementations, with varying batch sizes, input sizes, filter numbers, kernel sizes and strides, and analyze strengths and weaknesses for those implementations in shape limitations.

For detailed performance profiling, we conduct four sets of experiments as follows. The goal is to explore the reasons behind performance differences between those implementations.

- For aforementioned hotspot layers, we identify top kernels that dominate the total runtime.
- We compare peak GPU memory usage for those implementations over a wide range of configurations.
- With the *nvprof* tool [14] provided by NVIDIA, we profile and analyze those top kernels in five important metrics and two events.
- We evaluate the overheads of data transfers between CPU and GPU over five typical configurations.

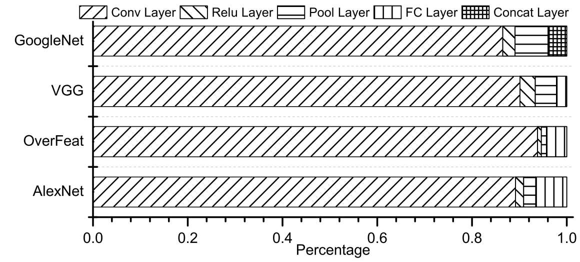

Fig. 2: Runtime breakdown of typical real-life CNN models: GooleNet, VGG, OverFeat and AlexNet.

IV. High-Level Workload Profiling

In this section, we make a high-level workload profiling. First, we break down four popular CNN models to investigate where hotspot layers are during their training iterations. Second, we compare the hotspot layers of those CNN implementations in terms of runtime over a large parameter space.

A. Hotspot Layer Analysis

The hotspot layer analysis can help understanding the flow of CNN applications and identify hotspot layers that dominate the total runtime in CNN models. We break down four popular real-life CNN models, i.e., AlexNet, GoogleNet, OverFeat and VGG, to collect the runtime of each layer and identify the hotspot layers for each model. The runtime we collected is the average runtime of each layer for 10 training iterations. Each training iteration includes one forward propagation and one backward propagation.

Results. As shown in Figure 2, those real-life models are mainly comprised of convolutional layer (Conv Layer), Pooling layer, Relu layer, Fully Connected Layer (FC Layer) and Concat layer (in GooLeNet). Convolutional layer consumes the bulk of total runtime (86%, 89%, 90% and 94% respectively in four CNN models).

Analysis. Convolutional layer involves large amount of computation-intensive operations and requires substantial amount of computing resources. Especially for modern advanced CNN models, the computing cost of convolutional layers is getting much higher due to the increasingly more filters and layers, smaller strides and their combinations [17]. Therefore, we primarily focus on evaluating the performance of convolutional layer in this paper.

B. Runtime Comparison

We run five groups of experiments in terms of runtime that is averaged over 10 iterations on GPUs, to compare the total runtime of a single convolutional layer of the seven implementations (Caffe, cuDNN, cuda-convnet2, Theano-CorrMM, Theano-fft, Torch-cunn and fbfft) with respect to different size of mini-batch, input image, filter number, kernel size and stride. For a better performance comparison, the total runtime we test here does not include the time of network initialization and data preparation. We organize those 5 parameters into a 5-tuple (b, i, f, k, s) similar to [35]. In order to investigate

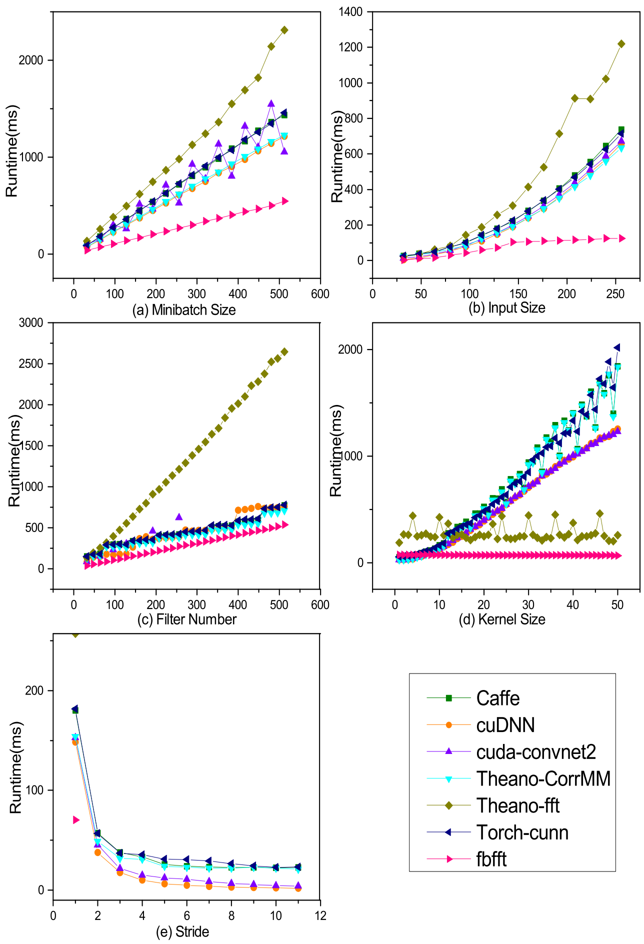

Fig. 3: Runtime comparison for seven convolutional implementations on GPU with varing configurations.

how each parameter impacts on the overall performance of convolutional layer, our evaluation is divided into five groups. Each group only tests one kind of the parameters, and the other four parameters are fixed. All input images and kernels are square and we have a basic configuration 5-tuple (64, 128, 64, 11, 1). According to five different parameters, we have five groups of 5-tuples: (b, 128, 64, 11, 1), (64, i, 64, 11, 1), (64, 128, f, 11, 1), (64, 128, 64, k, 1) and (64, 128, 64, 11, s). Taking the first tuple for example, we test a changeable mini-batch by fixing the other four parameters. In addition, we also observe the shape limitations for each implementation during the runtime comparison.

Results. Figure 3(a and b) shows the speed of the seven implementations in different mini-batch size and input size, which ranges from 32 to 512 and 32 to 256 with multiple of 32 and 16 respectively. The runtime clearly presents the advantage of fbfft over other implementations (from 1.4$\times$ to 9.7$\times$) in all given mini-batch and input sizes, while Theano-fft results in the slowest speed. For unrolling-based convolution, cuDNN has consistent superior performance in all given mini-batch and input sizes. The performance of cuda-convnet2 is not stable with different mini-batch sizes. It performs well only for those cases when mini-batch size is a multiple of 128.

In Figure 3(c), filter number ranges from 32 to 512 with multiple of 16. In this configuration space, fbfft is consistently faster than other implementations (from 1.19$\times$ to 5.1$\times$), while Theano-fft still results in the worst performance. Cuda-convnet2 cannot support all given filter numbers in our experiment and thus its runtime on GPU is reported with dots in Figure 3(c). For unrolling-based convolution, Theano-CorrMM slightly outperforms its counterparts with large filter numbers (greater than 160 in our experiment).

In Figure 3(d), for small kernel size (smaller than 7 in our experiment) cuDNN and Theano-CorrMM result in better performance than others. For example, the speed advantage of cuDNN over fbfft is from 1.21$\times$ to 2.62$\times$. But with the increasing of the kernel size (greater than 7), the runtime of fbfft tends to be a constant value and the performance advantage is becoming increasingly obvious. For example, fbfft is becoming increasingly faster than cuDNN (from 1.15$\times$ to 19$\times$). In addition, the performances of cuda-convnet2 and cuDNN are very close with all given kernel sizes.

In Figure 3(e), fbfft outperforms other implementations when stride is size of 1. Because $fbfft$ and $Theano-conv2d\_fft$ only support stride size of 1, their runtime is denoted as an spot in the figure. For greater stride (greater than 1), cuDNN results in the best performance.

Analysis. The speed of each implementation varies with different configurations and there is no single implementation that is the fastest for all given scenarios in our experiments. We summarize the main observations from runtime comparison as follows:

- fbfft is the overall fastest convolutional implementation and cuDNN performs the second best in most scenarios.
- For small kernels (smaller than 7 in our experiment), cuDNN outperforms fbfft. Otherwise, fbfft is faster than cuDNN.
- For unrolling-based convolution, cuDNN is the overall fastest implementation. But for large filter numbers (greater than 160 in our experiment), Theano-CorrMM slightly outperforms cuDNN.
- cuda-convnet2 performs well only for certain cases, such as for mini-batch sizes of multiple of 128.

In most scenarios, the speed of fbfft is much faster due to its low arithmetic complexity compared with unrolling-based convolution and direct convolution. cuDNN is much slower than fbfft when computing convolution with a large kernel size (large than 7 in our experiment). But for a small kernel size (smaller than 7 in our experiment), fbfft is a bit slower than cuDNN. In essence, this arises from the differences between their convolution strategies. fbfft can benefit significantly from dramatic reduction of arithmetic complexity when running on a large kernel size. But for a small kernel, the computational cost of fbfft is higher than other counterparts, which leads to a lower speed. It is important to note that fbfft and Theano-fft share the similar convolution strategy, but they present a clear difference in performance. Because of different implementation techniques, fbfft is much faster than Theano-fft. Cuda-convnet2 was optimized for mini-batch sizes of a multiple of 128, and thus performs well only in those cases.

Summary. From the perspective of speed, fbfft is the fastest implementation to train a CNN model with large kernels. For small kernels, cuDNN would be a good choice. Moreover, for a model with small kernel and large filter number, Theano-CorrMM slightly outperforms other implementations.

From the perspective of shape restrict, unrolling-based implementations are most flexible in configuration selection as they support any possible shapes. Cuda-convnet2 only supports square input images and square kernels, its mini-batch size must be a multiple of 32 and its filter number must be a multiple of 16. FFT-based convolutions (i.e., fbfft and Theano-fft) are applicable to any configuration shapes except that their stride must be 1.

V. Detailed Performance Profiling

In this section, we primarily focus on the performance profiling of convolutional layer in each implementation. First step, we conduct a detailed hotspot kernel analysis to look more closely at the inside of each convolutional implementation. Secondly, we evaluate the memory usage for each convolution implementation. Thirdly, we report a comprehensive profiling and analysis of the GPU performance for those convolution implementations. Finally, we evaluate the data transfer overhead between CPU and GPU.

A. *Hotspot Kernels in Convolutional Layer*

A convolutional layer in each implementation consists of multiple kernels and it is worthwhile to figure out which kernel determines the overall performance of convolutional layers. The analysis of hotspot kernels helps understanding and identifying which kernels dominate the total runtime in convolutional layer.

For different configurations, the convolutional layer in the same implementation shows the similar hotspot kernel results. We thus choose one set of configuration (64, 128, 64, 11, 1), which indicates that a square input of size 128, 64 mini-batch size, 64 filters, square kernel of size 11 and stride of size 1, as the representative to analyze hotspot kernels. Based on the profiling results, we group the similar kernels who have the same functionalities into one. Take GEMM (General Matrix to Matrix Multiplication) as an example, all different kernels that are responsible for matrix-matrix or matrix-vector multiplications are classified into GEMM.

Results. Figure 4 shows the hotspot kernels developed for convolutional layer of each implementation in terms of percentages. As we can see, different convolution strategies result in totally different hotspot kernel results. Even for the same convolution strategy, the kernels can be clearly different due to different implementation methods. According to Figure 4(a,b,c), for unrolling-based convolution, Caffe, Torch-cunn and Theano-CorrMM have similar hotspot kernel results, in which GEMM operations take up 87%,83%,80% of their total runtime respectively. But the hotspot kernel results of cuDNN

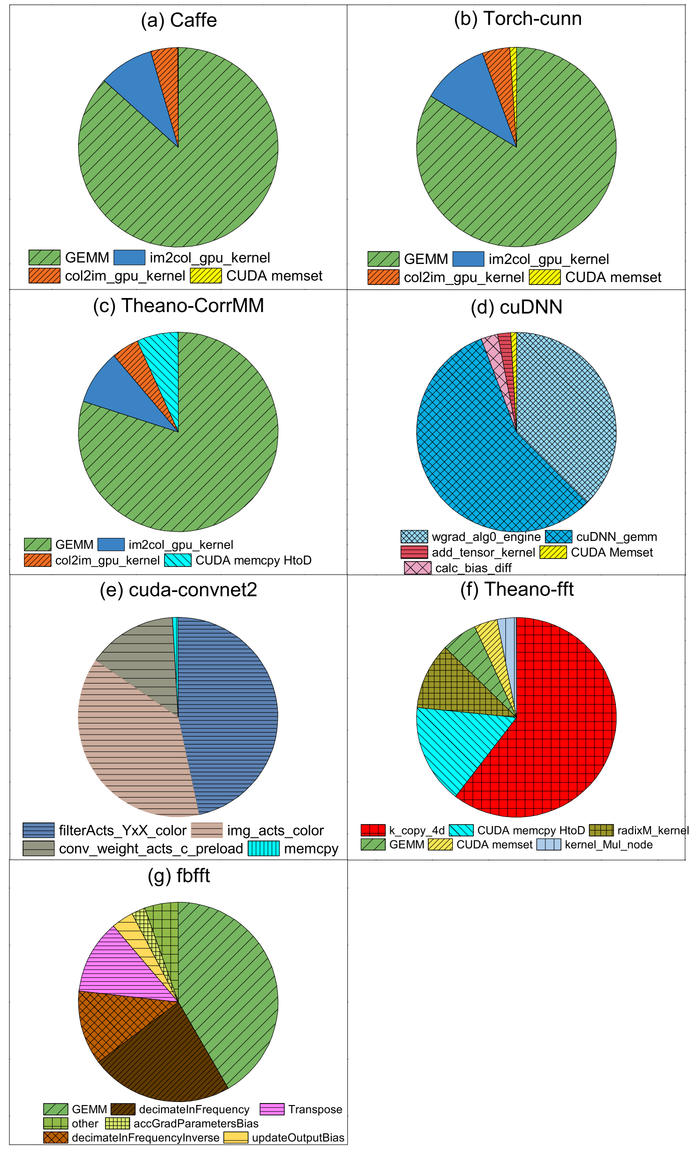

Fig. 4: Runtime breakdowns of convolutional layers in different implementations.

are totally different with its counterparts (Caffe, Torch-cunn and Theano-CorrMM) due to its different kernel implementations. As shown in Figure 4(d), $wgrad\_alg0\_engine$ and $cuDNN\_gemm$ dominate the runtime of cuDNN. cuda-convnet2 computes for convolutional layers directly, which is mainly achieved by three kernels: $filterActs\_YxX\_olor$, im_acts_color and $conv\_weight\_acts\_c\_preload$.

Analysis. We summarize some observations as follows:

- GEMM operations are the essence of convolutional layers. Especially in unrolling-based convolution, GEMMs are dominant of the total runtime, followed by unrolling operations.
- For FFT-based convolution, GEMM, FFT transform, FFT inverse and data transposition account for most of the

runtime in fbfft. On the contrary, most of the runtime is spent on data preparation and data transfer between CPU and GPU in Theano-fft.

For unrolling-based convolution, in Caffe, Torch-cunn and Theano-CorrMM, $im2col\_gpu\_kernel$ and $col2im\_gpu\_kernel$ mainly take up the rest of the runtime. $im2col\_gpu\_kernel$ is used to unroll the input data and filters to double large matrixes and then the traditional convolution can be converted into a clean matrix-matrix multiplication by using highly-optimized GEMM libraries. The $col2im\_gpu\_kernel$ is used to convert the multiplication result back to the right format, the same as the format before unrolling. In cuDNN, the unrolling operations and matrix-matrix multiplications are optimized by using shared memory and tiled matrix multiplication [24], which is mainly achieved by $wgrad\_alg0\_engine$ and $cuDNN\_gemm$ kernels.

For FFT-based convolution, the computation of convolutional layers is mainly achieved by three steps in fbfft. Firstly, the kernel $decimateInFrequency$ uses DIF algorithm to transform input and weight data from spatial domain to frequency domain. Secondly, the $Transpose$ kernel is used to convert the $BDHW$ layout into $HWBD$ and then conducts Cgemm matrix multiplications. Thirdly, the $Transpose$ kernel converts the Cgemm results back to $BDHW$ layout and performs an inverse FFT by using $decimateInFrequencyInverse$ [25].

Summary. GEMM is the essence of convolutional layers in unrolling-based implementations, which indicates that kernels responsible for GEMM computing are the first-order modules to be optimized. So are FFT and Cgemm in fbfft.

B. Memory Usage

For most applications at present, memory is not the primary limitations, and while the fastest algorithm is considered as the best algorithm. As a result, a common way to rank order algorithms is using their computing speeds as a criterion. However, GPU cannot afford a large memory-consuming application due to its limit device memory. Thus memory usage also should be considered as a significant portion on GPUs.

Results. We use *nvidia-smi* to monitor memory usage on GPU for each implementation. Figure 5 shows the peak memory consumption of the seven convolutional implementations by varying different parameters that are similar to runtime comparison. In all given scenarios of our experiments, cuda-convnet2 have the lowest consumption of GPU memory (from 125 MB to 2076 MB), followed closely by Torch-cunn (from 170 MB to 2093 MB). While the other three unrolling-based implementations, cuDNN, Caffe and Theano-CorrMM, are of a relatively higher consumption (from 155MB to 3810MB, from 136MB to 3809MB and from 130MB to 3709MB respectively). On the contrary, FFT-based convolution have the highest consumption of GPU memory. Taking fbfft as an example, it consumes a large amount of GPU memory, from 1632 MB to 10866 MB in our experiments. There are also several abnormal memory consumptions in FFT-based

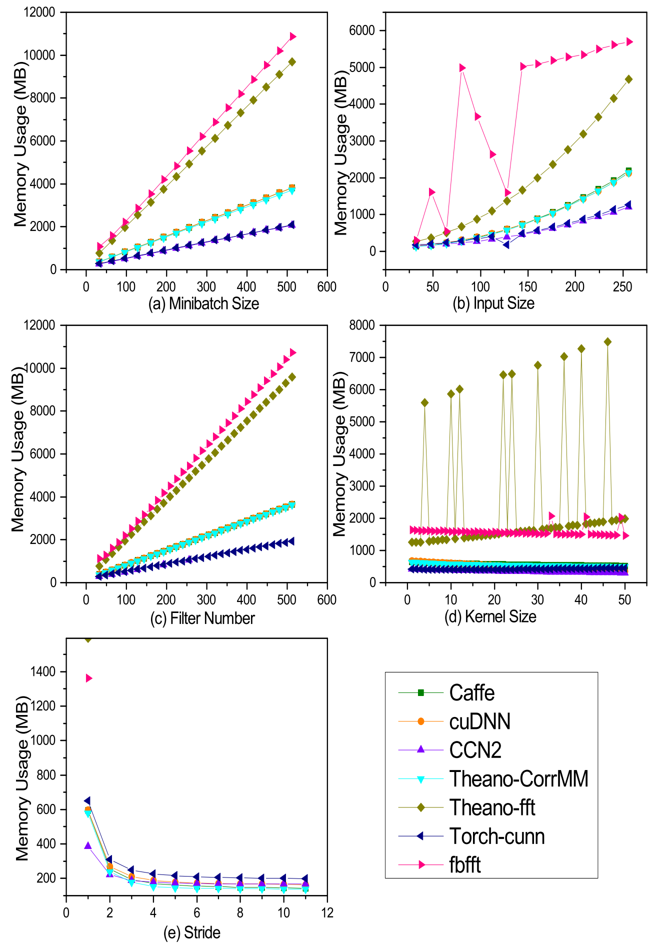

Fig. 5: Memory usage comparison for seven convolutional implementations on GPU with varing configurations.

implementations. Figure 5 (b) shows that there are dramatic fluctuations in memory usage of fbfft over certain input size. The same fluctuation also can be observed in fbfft and Theano-fft in Figure 5 (d). Such abnormal memory usage can lead to program crush which we will investigate as part of future work.

Analysis. We summarize main observations from the above results as follows:

- cuda-convnet2 is the most memory efficient one in all scenarios given in our experiment.
- Torch-cunn is the overall most memory efficient implementation in unrolling-based convolution, while with the increase of kernel size, cuDNN becomes the most memory efficient implementation.
- fbfft requires the most memory, followed by Theano-fft.

Cuda-convnet2 computes the convolution directly and thus does not need temporary memory to keep intermediate data. Compared with cuda-convnet2, Caffe, Theano-CorrMM and Torch-cunn require extra memory to store the unrolled matrices using, but there are still slight differences of memory usage due to different data layouts and programming techniques between them. Although cuDNN does not need extra memory

TABLE I: Convolution configurations for benchmarking

Layers	Configuration (b,i,f,k,s)
Conv1	(128,128,96,11,1)
Conv2	(128,128,96,3,1)
Conv3	(128,32,128,9,1)
Conv4	(128,16,128,7,1)
Conv5	(128,13,384,3,1)

for unrolling, it consumes more memory than other unrolling-based implementations to achieve a better performance.

On the contrary, low computational complexity of FFT-based implementations and highly optimized CUDA codes bring an excellent speed to fbfft, however, at the expense of an unreasonable memory consumption. The main reason is that FFT-based implementations require substantial amounts of temporary memories to keep the intermediate data such as input and filter data of the Fourier domain, and they also need extra memory for zero-padding to extend filter bank to be the same size of input. Therefore, when choosing a CNN implementation, a trade-off between speed and memory consumption needs to be considered.

Summary. Cuda-convnet2 is well suitable for cases when the memory is limited. Otherwise, fbfft is a great choice to compute for convolutional layer. If a good balance between memory, speed and flexibility is needed, cuDNN is most likely the best choice.

C. GPU Performance Evaluation

In this subsection, we conduct a detailed runtime profiling study based on *nvprof* CUDA tool. Metrics and events are collected by using *nvprof* to analyze kernel performance during kernel execution. An event collects hardware counter values during kernel execution and a metric is computed based on one or more event values to identify characteristics of an CUDA application [14]. To investigate the performance differences among seven different convolution implementations, we use the following metrics to profile GPU performance [14]:

- *achieved_occupancy* is the ratio of the average active warps per active cycle to the maximum number of warps supported on a SM.
- *ipc* is the instructions executed per cycle.
- *warp_execution_efficiency* is the Ratio of the average active threads per warp to the maximum number of threads per warp supported on a multiprocessor expressed as percentage.
- *gld_efficiency* and *gst_efficiency* is the ratio of requested global memory load/store throughput to required global memory load/store throughput.
- *shared_efficiency* is the ratio of requested shared memory throughput to required shared memory throughput.

Five representative convolutional configurations that are commonly used for benchmarking of convnet in [27] are used in our performance profiling to capture various behaviors and performance characteristics. But in order to result in a performance difference, we select a new configuration as the second layer in our evaluation, which has a large input and a small kernel. As shown in Table I, the five configurations, to a certain extent, can represent the configurations of real-life CNNs. For example, in the first two configurations in Table I, the kernel size are quite small compared to the input size, which is common in the first few layers of a real-life CNN models. The last few layers of real-life CNN models often have more filters and large kernel size relative to input image [2], [24].

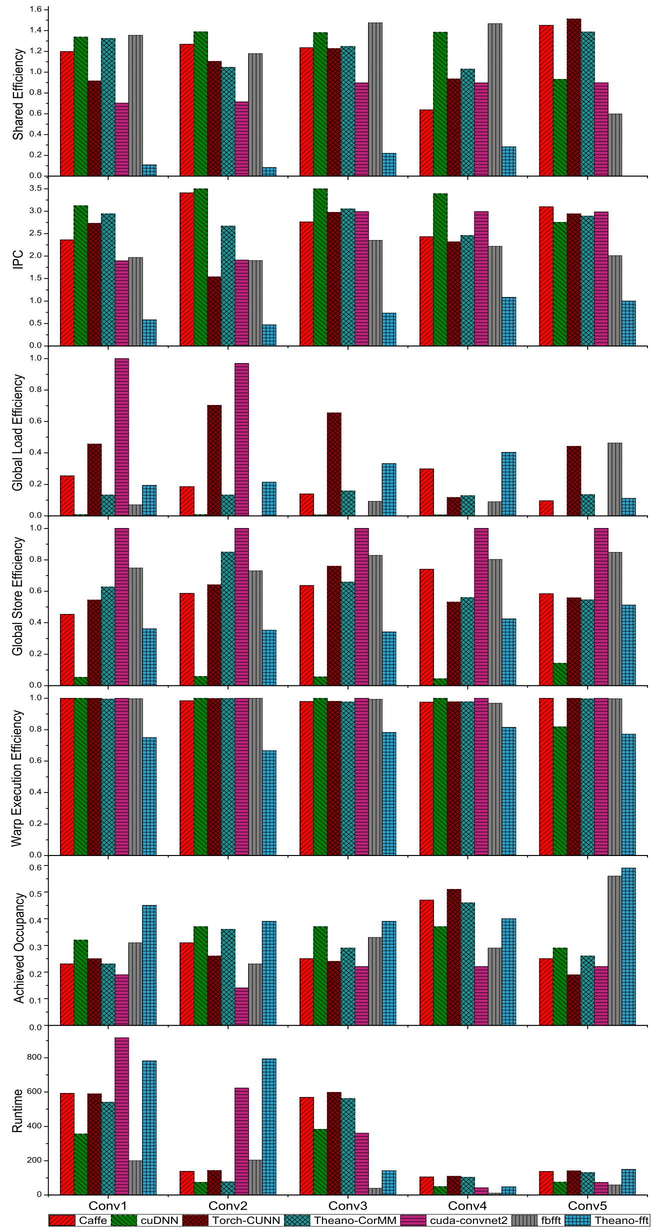

Fig. 6: GPU performance profiling. From bottom to top are runtime (ms), achieved occupancy (%), warp execution efficiency (%), global store/load efficiency (%), IPC and shared memory efficiency (%), respetively.

As shown in Figure 4, there are multiple kernels in each implementation and each kernel performs different functionalities of convolutional layer. In order to evaluate an overall performance of each implementation on GPU, we prefer to profile metrics and events for top kernels of each implementation and then take a weighted average of those top kernels to get the final estimate of performance metrics for that implementation. The weight of each kernel is determined by the percentage of its runtime in the whole implementation.

All seven convolutional implementations in our experiments are based on three convolution strategies. Each strategy adopts different methodology and thus has different computational complexity. For a fair comparison, we analyze the performance metrics according to three convolution strategies. Figure 6 presents the performance profiling results in terms of speed and five metrics for top kernels of each convolutional implementation by running over five convolution configurations in Table I.

The runtime part in Figure 6 shows that cuDNN is the fastest implementation in unrolling-based convolution and fbfft is the fastest one in FFT-based convolution. As expected, their better performance can be supported by their better metric values. For unrolling-based implementation, the metrics (ipc, $achieved\_occupancy$ and $shared\_efficiency$) of cuDNN are overall better than the metrics of its counterparts (Caffe, Torch-cuNN and Theano-CorMM). For FFT-based convolution, the metrics of fbfft are much better than the metrics of Theano-fft. Cuda-convnet2 also has efficient metric profiling results.

1) Observation in achieved occupancy: As shown in achieved occupancy part in Figure 6, most implementations have relatively low achieved occupancy (less than 30%). Especially, the achieved occupancy in cuda-convnet2 is lower than the average level, from 14% to 22%.

Analysis. As shown in Figure 6, cuDNN has an overall better performance and is somehow related to its higher percentages (from 29% to 37%) of achieved occupancy compared with other unrolling implementations. However, a higher occupancy does not mean a better performance. For FFT-based convolution, Theano-fft has higher percentages (39% to 59%) but worse performance. The achieved occupancy in cuda-convnet2 is low, from 14% to 22%, which indicated that top kernels in cuda-convnet2 does not generate enough threads to hide potential latency.

The essence of GPU performance lies in whether the problem can be computed in a high degree of parallel and whether the limited resources on GPUs are allocated reasonably. Threads on GPUs are grouped into warps (32 threads per warp on Tesla K40c) and these warps execute in parallel on GPUs. The context of each warp can be switched almost zero-overhead by GPU hardware scheduler. Long access latencies can be hidden by this zero-overhead context switching when there are enough parallel threads running on GPUs. The resources (e.g., registers and shared memories) are very limited on each SM of GPUs. Reasonable use of those fast memory resources can significantly improve the performance. But using them too much can reduce the total active warps on GPU, which can leads to low occupancy and performance degradation.

To investigate the reason behind low occupancy, we profile the usage of register and shared memory for each implementation as reported in Table II. cuda-convnet2 has the overall lowest percentages of achieved occupancy, which is mainly correlated to its bad usage of register and shared memory. Tesla K40c provides 65536 registers of the maximum amount per SM, while 116 registers are used in cuda-convnet2 by each thread. As a result, the theoretical active threads are only 564 (17 active warps), which is far less than device maximum active threads 2048(64 active warps) per SM and thus leads to a low $achieved\_occupancy$. Similarly, too much use of shared memory can also lead to low occupancy. On the contrary, little use of register and shared memory may contribute to a high achieved occupancy, which can also bring in bad performance due to long access latency from global memory. As shown in Figure 6, although the occupancy of Theano-fft is higher than that of fbfft, its performance is far worse than that of fbfft. One of the main reason to explain that is the little use of register and shared memory.

Summary. Occupancy is limited by three potential factors: register usage, shared memory usage and block size. It is important that GPU-based CNN implementations carefully balance these factors to improve the overall performance.

2) Observation in global memory access efficiency: As shown in Figure 6, most of implementations have relatively low percentages of global load and store efficiency, less than 20% for global load efficiency and less than 60% for global store efficiency.

Analysis. Global memory is the largest memory space on GPUs and most data is firstly initialized on it. It also has the highest access latency. The metric $gld\_$ and $gst\_efficiency$ can be measured to evaluate how efficient the threads within a kernel write or read on global memory. When global load or store efficiency is less than 100%, it indicates that there exists request replays in global memory access due to inappropriate access pattern, such as unaligned or non-coalesced memory access.

As we can see from Figure 6, Caffe, Torch-cunn, Theano-CorrMM and Theano-fft have very low global memory load efficiencies, especially for Theano-CorrMM (from 11.64% to 15.79%), mainly because of non-coalesced accesses during their kernel executions. In cuDNN, for some top kernels that are responsible for the operations of unrolling and matrix multiplication use little global memory. Instead, most of the computation for convolutional layers in cuDNN is conducted on shared memory only. Therefore, the global access efficiency of those top kernels is 0%. However, for other top kernels that pre-compute for convolution in cuDNN are conducted on global memory and result in low global load and store efficiencies, which mainly contributes to the overall low global access efficiency of cuDNN in Figure 6. Similarly, low global memory access efficiencies of fbfft is also due to little use of global memory by certain top efficient kernels.

TABLE II: Register numbers per thread and shared memory usage per block of different implementations.

Implementation	Registers	Shared Memory(KB)
Caffe	86	8.5
cuDNN	80	8.4
Torch-cunn	84	8.1
Theano-CorrMM	72	7
cuda-convnet2	116	16
fbfft	106	10
Theano-fft	2	4.5

Summary. It is desirable to use on-chip memory as much as possible, combined with aligned and coalesced access, to improve the efficiency of global memory access. Further optimization for different GPU-based CNN is required in this area.

3) Observation in shared memory efficiency: As shown in Figure 6, Theano-fft have the lowest percentages (from 8.16% to 20%) of shared efficiency, while other implementations have relatively higher percentages.

Analysis. Shared memory is divided into banks on GPUs and bank conflict (or broadcast) occurs when multiple threads in a warp simultaneously access the same bank. When a bank conflict occurs, the accesses to the same bank are serialized by shared memory system, which leads to a significant performance decrease. A low shared efficiency implies that there are bank conflicts during kernel execution.

Observed from the runtime part in Figure 6, Theano-fft is much slower than its counterpart (fbfft), which can be supported by the evidence of their shared efficiencies. As shown in Figure 6, the shared efficiency of Theano-fft is much lower than that of fbfft, which indicates that there are many bank conflicts during its kernel execution and thus leads to a worse performance. For unrolling based implementations, cuDNN has the overall highest percentages of shared efficiency (over 130% in most cases). The shared efficiency in Caffe, Torch-cunn and Theano-CorrMM is still excellent because most of their GEMM operations are computed by using cuBlas that is highly optimized on shared memory and thus they are of a relatively higher shared memory efficiency.

Summary. Shared memory is a particularly important resource to optimize GPU kernels. Bank conflicts are the primary concern to improve the performance of Theano-fft. One should carefully design kernels so as to avoid multiple threads in the same warp access the same bank. Moreover, memory padding is another way to avoid bank conflict for some access patterns.

4) Observation in warp execution efficiency (WEE): Most of the implementations achieve an excellent WEE (over 97%), while Theano-fft has a much lower WEE.

Analysis. The metric of WEE is used to measure how efficient the threads execute in a warp. The maximum number of threads per warp in K40c is 32. All the 32 threads in a warp execute the same instructions. If the threads in a warp take different control paths, it is assumed to be divergent.

Figure 6 shows that the WEE is quite low for Theano-fft on different configurations (from 66% to 81%), which indicates

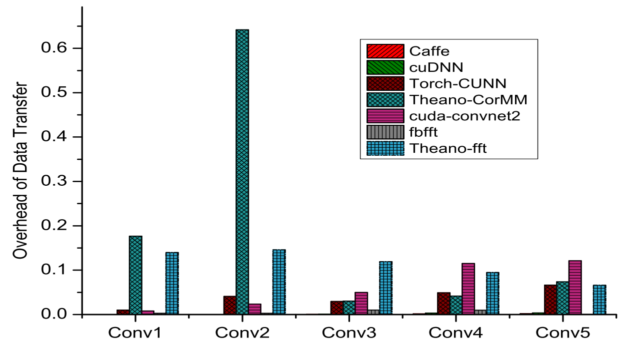

Fig. 7: Data transfer overheads of different implementations over five configurations.

that there are warp divergent branches and the kernels in Theano-fft suffer from diminished SIMD efficiency caused by serialized execution of divergent branches.

Summary. The metric of WEE provides a thread-level insights of GPU performance. For Theano-fft, control divergence can be reduced by redesigning its kernel codes to avoid the use of control flow statement (e.g., if-else) as little as possible or converting the control statement into non-control statement. Otherwise, we have to rearrange the data access patterns of Theano-fft to increase WEE.

D. Overhead of Data Transfer Between CPU and GPU

The data transfer overhead between CPU and GPU can be crucial to the performance if that takes up too much of total runtime. Therefore, programmers must be conscious of the overhead of data transfer in CPU-GPU hybrid system.

Results. Figure 7 provides quantitative analysis of the relative time spent on data transfer of each convolution implementation across five typical different convolutional configurations. As shown, cuDNN, Caffe and fbfft have the lowest percentage (almost 0%) of data transfer time, while Torch-cunn, cuda-convnet2 and Theano-fft have relatively higher percentage (from 1% to 15%). Interestingly, Theano-CorrMM in the second configuration (Conv2) has a significant data transfer overhead (more than 60% of its total runtime).

Analysis. The difference in data transfer overhead is not fixed, but changes with the different configurations. Even with the same convolution strategy, the overhead of data transfer is not identical at all due to different program techniques. Take Caffe as example, before starting to compute convolution, a data prefetching thread is used to hide the latency from CPU-GPU data transfer.

Summary. A big overhead of data transfer between CPU and GPU can lead to an overall performance degradation. To improve that, one can reduce the transfer overhead by the following methods.

- Using pinned memory to improve the bandwidth.
- Using asynchronous transfer to hide the latency from CPU-GPU data transfer.

- Lowing the transfer frequency by organizing many small data transfers to a large data transfer.

VI. Conclusion

Training CNNs on large data sets is computation intensive, leading to a flurry of research and development of open-source parallel implementations on GPUs. The goal of this work is to assist practitioners identifying the most appropriate CNN implementations for different scenarios, and provide insights and suggestions to practitioners and pinpoint aspects for researchers who are interested in convolution optimization on GPU.

In this paper, we compare the performance of seven popular CNN implementations by running them over a wide range of parameter space on Tesla K40c. We investigate the merits and shortcomings for those implementations in terms of speed, memory and shape limitation. No single implementation is the best for all scenarios and we have to make trade-offs between speed and memory usage in these implementations.

We present a detailed performance analysis for those implementations and explore potential bottlenecks and acceleration opportunities. We can conclude that no single metric can determine the performance and we have to tune several metrics to achieve the best performance. Moreover, a deep understanding of the algorithm and hardware characteristic is extremely important to accelerate these implementations.

Acknowledgment

This work was supported by the National Natural Science Foundation of China under Grants 61170008 and 61272055, the National Grand Fundamental Research 973 Program of China under Grant No. 2014CB340402, and the Open Project Program of the State Key Laboratory of Mathematical Engineering and Advanced Computing.

References

[1] NVIDIA, CUDA C Programming Guide, 2015.
[2] Krizhevsky, A., Sutskever, I., and Hinton, G. E. ImageNet classification with deep convolutional neural networks.In NIPS, pp. 1106-1114, 2012.
[3] Sainath, T., Mohamed, A.-R., Kingsbury, B. & Ramabhadran, B. Deep convolutional neural networks for LVCSR. In Proc. Acoustics, Speech and Signal Processing 8614-8618, 2013.
[4] Dario Amodei, Rishita Anubhai, Eric Battenberg, Carl Case, Jared Casper, Bryan Catanzaro, Jingdong Chen, Mike Chrzanowski, Adam Coates, Greg Diamos, Erich Elsen, et al. Deep Speech 2:End-to-End Speech Recognition in English and Mandarin. arXiv:1512.02595, 2015.
[5] Zhang, X., Zhao, J., LeCun, Y. Character-level Convolutional Networks for Text Classification, arXiv: 1509.01626, 2015.
[6] Daojian Zeng, Kang Liu, Siwei Lai, Guangyou Zhou, and Jun Zhao. Relation classification via convolutional via convolutional deep neural network. In Proceedings of COLING 2014, pages 2335-2344, August 2014.
[7] Nguyen, T. H., Grishman, R. Relation Extraction: Perspective from Convolutional Neural Networks. Workshop on Vector Modeling for NLP, pp. 39-48, 2015.
[8] Y. LeCun and C. Cortes. MNIST Handwritten Digit Database, http://yann.lecun.com/exdb/mnist, August 2009.
[9] Zeiler, M. D. and Fergus, R. Visualizing and understanding convolutional networks. CoRR, abs/1311.2901, 2013. Published in Proc. ECCV, pp. 818-833, 2014.
[10] Simonyan, Karen, and Andrew Zisserman. "Two-stream convolutional networks for action recognition in videos." In Advances in Neural Information Processing Systems, pp. 568-576. 2014.
[11] A. Krizhevsky, V. Nair, and G. Hinton. CIFAR-10 Dataset. https://www.cs.toronto.edu/ kriz/cifar.html.
[12] Simonyan, Karen, and Andrew Zisserman. "Very deep convolutional networks for large-scale image recognition." arXiv preprint arXiv:1409.1556, 2014.
[13] NVIDIA. https://developer.nvidia.com/cuda-zone.
[14] Docs Nvidia 2015 Profiler User Programming Guide CUDA Toolkit documentation.
[15] Szegedy, Christian, et al. "Going deeper with convolutions." Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 1-9, 2015.
[16] Deng, Jia, et al. "Imagenet: A large-scale hierarchical image database." Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on. IEEE, 2009.
[17] He, Kaiming, and Jian Sun. "Convolutional neural networks at constrained time cost." Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 5353-5360, 2015.
[18] Krizhevsky, A. One weird trick for parallelizing convolutional neural networks. CoRR, abs/1404.5997, 2014.
[19] James Bergstra, Olivier Breuleux, Frederic Bastien, Pascal Lamblin, Razvan Pascanu, Guillaume Desjardins, Joseph Turian, DavidWarde-Farley, and Yoshua Bengio. Theano: a cpu and gpu math expression compiler. In SciPy, volume 4, page 3, 2010.
[20] Ronan Collobert, Koray Kavukcuoglu, and Clement Farabet. Torch: A matlab-like environment for machine learning. In BigLearn, NIPS Workshop, 2011.
[21] J. Donahue, Y. Jia, O. Vinyals, J. Homan, N. Zhang, E. Tzeng, and T. Darrell. Decaf: A deep convolutional activation feature for generic visual recognition. CoRR, abs/1310.1531, 2013.
[22] Sermanet, Pierre, et al. "Overfeat: Integrated recognition, localization and detection using convolutional networks." arXiv preprint arXiv:1312.6229, 2013.
[23] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick,Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast featureembedding. arXiv preprint arXiv:1408.5093, 2014
[24] Chetlur, Sharan, et al. "cudnn: Efficient primitives for deep learning." arXiv preprint arXiv:1410.0759, 2014.
[25] Nicolas Vasilache, Jeff Johnson, Michael Mathieu, Soumith Chintala, Serkan Piantino, Yann LeCun. FAST CONVOLUTIONAL NETS WITH fbfft : A GPU PERFORMANCE EVALUATION. arXiv: 1412.7580. 2015.
[26] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. Nature, 521:436-444, 2015.
[27] convnet-benchmarks. https://github.com/soumith/convnet-benchmarks, 2014.
[28] cuda-convnet2.torch. https://github.com/soumith/cuda-convnet2.torch, 2014.
[29] NVidia, C. U. D. A. "CUDA Profiler Users Guide (Version 6.5): NVIDIA." Santa Clara, CA, USA, 2014.
[30] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, Gradient-Based Learning Applied to Document Recognition, in Proc. of the IEEE, vol. 86, no. 11, 1998, pp. 2278-2324.
[31] http://deeplearning.net/software/theano/index.html
[32] Kumar Chellapilla, Sidd Puri, Patrice Simard, et al. High performance convolutional neural networks for document processing. In Workshop on Frontiers in Handwriting Recognition, 2006.
[33] Ronan Collobert, Koray Kavukcuoglu, and Clement Farabet. Torch: A matlab-like environment for machine learning. In BigLearn, NIPS Workshop, 2011.
[34] James Bergstra, Olivier Breuleux, Frederic Bastien, Pascal Lamblin, Razvan Pascanu, Guillaume Desjardins, Joseph Turian, DavidWarde-Farley, and Yoshua Bengio. Theano: a cpu and gpu math expression compiler. In SciPy, volume 4, page 3, 2010.
[35] Michael Mathieu, Mikael Henaff, and Yann LeCun. Fast training of convolutional networks through ffts. arXiv preprint arXiv:1312.5851, 2013.