GLP4NN: A Convergence-invariant and Network-agnostic Light-Weight Parallelization Framework for Deep Neural Networks on Modern GPUs

Hao Fu, Shanjiang Tang*
Tianjin University
Jinnan District, Tianjin, China
{haofu,tashj}@tju.edu.cn

Bingsheng He
National University of Singapore
Singapore
hebs@comp.nus.edu.sg

Ce Yu*, Jizhou Sun
Tianjin University
Jinnan District, Tianjin, China
{yuce,jzsun}@tju.edu.cn

ABSTRACT

In this paper, we propose a network-agnostic and convergence-invariant light-weight parallelization framework, namely *GLP4NN*, to accelerate the training of Deep Neural Networks (DNNs) by taking advantage of emerging GPU features, especially concurrent kernel execution. To determine the number of concurrent kernels on the fly, we design an analytical model in the kernel analyzer module and integrate a compact asynchronous resource tracker in the resource tracker module for collecting runtime configurations of kernels with low memory and time overheads. We further develop a runtime scheduler module and a pool-based stream manager for handling GPU work queues in GLP4NN to avoid consuming too many CPU threads or processes while dispatching workloads to GPU devices. In our experiments, we integrate GLP4NN into Caffe to accelerate the batch-based training of four well-known networks on NVIDIA GPUs. Experimental results show GLP4NN is able to achieve a speedup of up to 4X over the original implementation as well as keep the convergence property of networks.

CCS CONCEPTS

• **Computing methodologies → Neural networks**; **Concurrent algorithms**; *Batch learning*; *Machine learning approaches*; Massively parallel algorithms;

KEYWORDS

Network-agnostic, Convergence-invariant, Light-weight Parallelization Framework, GLP4NN, Neural Network, GPU

ACM Reference Format:
Hao Fu, Shanjiang Tang, Bingsheng He, and Ce Yu*, Jizhou Sun. 2018. GLP4NN: A Convergence-invariant and Network-agnostic Light-Weight Parallelization Framework for Deep Neural Networks on Modern GPUs. In *ICPP 2018: 47th International Conference on Parallel Processing, August 13–16, 2018, Eugene, OR, USA*. ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3225058.3225077

*Corresponding Authors: Shanjiang Tang and Ce Yu.

1 INTRODUCTION

In the past few years, we have witnessed successful adoptions of deep neural networks (DNNs) in various domains, such as computer vision [34, 44, 46], multi-modal data analysis [15], and natural language processing [8]. The two key factors behind these remarkable achievements are the immense computing power and the availability of massive training datasets. Both require plenty of computational and memory resources. As a result, modern many-core accelerators like GPUs have become popular in DNN training systems.

Recently, some emerging hardware features have been introduced to new-generation GPU microarchitectures as shown in Table 1. First, concurrent kernel execution is supported and improved since Kepler architecture, by using multiple CUDA streams, Hyper-Q feature or multiple-process service (MPS). Second, more computing resources, such as CUDA cores and registers, are integrated into a single GPU chip, and memory bandwidth is also increasing. Hence, it allows users to dispatch more workloads onto a single GPU device. These hardware features change the basic assumption of the existing *single* kernel-based execution approach and open new space for optimizing DNN training.

Most modern deep learning frameworks are designed with GPU support by adopting a kernel-based execution approach, such as Tensorflow [1], Theano [2], MXNet [4], Caffe [17], DIGITS [28] and SINGA [41]. But DIGITS and all the above frameworks are mainly designed to simplify the implementation of DNN applications. To further optimize DNN applications on GPUs, NVIDIA has issued a set of tools and libraries, including cuDNN [5] and cuBLAS [30], which concentrate on optimizing the performance of kernels for fundamental operations. There are also recent works on accelerating CNNs by reducing arithmetic complexity in convolution layers [22, 25, 40] and optimizing memory access [23]. However, all of the methods above only focus on optimizations of individual GPU kernels in a network layer without considering the new features (in particular, concurrent kernel executions) of modern GPUs, which can result in low resource utilization. As we will show in the paper later, being aware of new features can bring in significant performance improvement for DNNs' training.

Due to differences among GPUs and workloads, there are three challenges in the training phase of DNNs. The first challenge is about the collection of kernel execution configurations on-the-fly. On account of the prosperity of neural network models and layers, workloads may vary from layer to layer or network to network. The difference in the sample size of different datasets aggravates it. Also, these two issues make it hard to analyze the kernel configuration statically. Although there are several excellent profiling tools such as NVIDIA Visual Profiler [27] and Vampir [39] that can be utilized to collect kernel runtime information. But these tools can only be used in an offline manner, and a sample execution is required, which brings in burdens to developers and consumes a large amount of memory and times. Besides, it is very difficult to distinguish kernels belonging to different layers with offline tools.

The second challenge is on how many concurrent kernels on different devices should be launched. There are two issues to be addressed. The first is how to launch kernels in parallel. We can exploit Hyper-Q and MPS features, but these features may occupy too many CPU threads or processes when many kernels should be launched concurrently. The second issue is the setting of the proper number of concurrent kernels on GPUs equipped with various amount of resources on the basis of their runtime information. Moreover, the difference in GPU devices makes it impossible to specify a predefined optimal number of concurrent kernels for all GPU platforms (like those in Table 1).

Table 1: Overview of GPU architecture features.

Architecture	CUDA Streams	Dynamic Parallelism	Max Concurrent Kernels	HBM	Tensor Cores
Tesla	×	×	1	×	×
Fermi	√	×	16	×	×
Kepler	√	√	32	×	×
Maxwell	√	√	16	×	×
Pascal	√	√	128	√	×
Volta	√	√	128	√	√

The third challenge is the guarantee of the convergence property of neural networks. Since a parameter tuning process is always conducted to ensure appropriate convergence property before training a neural network, it is important to retain the effect of this pre-tuning phase and keep the convergence of a network model invariant between the original and optimized implementation.

To address all the above challenges, we propose a light-weight parallelization framework, called GLP4NN. In this framework, we utilize a compact resource tracker based on NVIDIA CUPTI library [31] to gather necessary kernel configurations at runtime within a low overhead. Then all information collected will be passed to the analytical model to figure out the proper number of concurrent kernels that can reside on a GPU under the resource constraint. In addition, instead of using multi-thread technology, we implement a pool-style stream manager to handle concurrent kernel launching.

In summary, this paper makes the following contributions:

- We propose a light-weight parallel framework to accelerate the training of DNN models, which can be integrated into modern deep learning frameworks.
- We make a compact resource tracker to collect kernels' runtime configuration and propose a naive analytical model to maximize the GPU resource utilization.
- We design a stream pool which can be utilized to dispatch concurrent kernels onto GPUs without consuming extra system thread or process resources.
- We prove that GLP4NN conforms to convergence-invariant and network-agnostic.
- We implement the proposed framework based on Caffe and evaluate its performance.

The remainder of this paper is organized as follows. First, the background and motivation of this paper are presented in Section 2. Details of the light-weight parallelization framework and the proposed analytical model are described and analyzed in Section 3. Section 4 shows the experiment results. We will introduce related works in Section 5, and conclude this paper in Section 6.

2 BACKGROUND

In this section, we first describe the training algorithm of neural networks and then show the motivation of this paper.

2.1 Neural Network Training

Algorithm 1 Forward Propagation Algorithm.

```
Input:
  The number of input bottom blobs.
  Dimensions of a bottom blob (N, $D_1$, $D_1$, ·, $D_s$).
Output: Top blob for the network layer.
  for all bottom blobs do
    for $n \leftarrow 1$ to $N$ do
      for $d_1 \leftarrow 1$ to $D_1$ do
        ...
        for $d_i \leftarrow 1$ to $D_i$ do
            top[f($n, d_1, \cdots, d_i$)]=BLAS(weight($d_1, \cdots, d_i$),
          bias($d_1, \cdots, d_i$), bottom[g($n, d_1, \cdots, d_i$)]).
        end for
      end for
    end for
  end for
```

Algorithm 2 Backward propagation Algorithm

```
Input:
  Gradient w.r.t. top blob (N, $D_1$, $D_1$, ·, $D_s$).
Output:
  Gradient w.r.t. to the bottom blob.
  for all top blobs do
    for $n \leftarrow 1$ to $N$ do
      for $d_1 \leftarrow 1$ to $D_1$ do
        ...
        for $d_i \leftarrow 1$ to $D_i$ do
          bottom_gradient[f($n, d_1, \cdots, d_i$)]=
              BLAS(top_gradient[h($n, d_1, \cdots, d_i$) , weight($d_1$,
          $\cdots, d_i$));
        end for
      end for
    end for
  end for
```

The batch training algorithm is widely adopted in neural networks. This algorithm continuously seeks the minimum value of a network loss function epoch by epoch. One epoch consists of processing all samples in the training dataset. During each epoch,

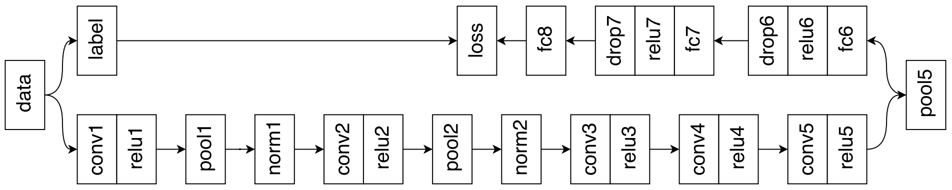


Figure 1: Overview of CaffeNet network.

samples are often processed in a batch manner. The computation process of a sample batch can be divided into two passes, namely *forward* and *backward*. For the *forward* pass, samples traverse the network and calculate an average loss value. While in the *backward* pass, a gradient value is computed and derivatives are propagated across the network for parameter update by applying the chain rule. Both *forward* and *backward* phases are handled layer by layer, and an inter-layer synchronization operation is often required. A brief introduction to the *forward* and *backward* pass of a network layer in Caffe can be found in Algorithm 1 and 2, where *bottom* and *top* are the inputs and outputs of a layer's *forward* pass, N is the batch size, *weight* and *bias* are layer parameters, and **BLAS** represents a linear algebra operation.

2.2 Motivation

To show the impact of concurrent kernel execution on DNN applications, we conduct several experiments in training CaffeNet (Fig. 1), which is a variant of AlexNet [21], with concurrent kernel execution on different GPU platforms. In general, two layers dominate the forward execution during the training of a CNN, i.e., the convolutional and pooling layers [37]. Thus we only present results on convolutional layers in CaffeNet on Tesla P100 GPU as an example when exploiting batch-level parallelism, and pertinent layer parameters can be found in Table 5 in Section 4. As shown in Fig. 3, concurrent kernel execution can help accelerate the training of convolutional layers.

Our work is motivated by the following observations.

Observation 1: The training of a DNN model can be accelerated by utilizing concurrent kernel execution strategy effectively. We have demonstrated a speedup in most convolution layers considered in this paper. Fig. 2 shows the speedup of convolution layers in the CaffeNet model, and Fig. 3 shows that it can help accelerate the computation of those layers by taking advantage of kernel execution overlapping with multiple CUDA streams. As shown in Algorithm 1 and 2, the computation of the backpropagation process is similar to that of the forward process. Hence, only the forward process of a layer is considered in experiments.

Observation 2: The optimal number of CUDA streams varies from GPU to GPU. Fig. 4 shows the optimal number of CUDA streams according to the runtime of the forward pass in each layer. Although programmers can utilize as many CUDA streams as possible, this method may consume a lot of hardware resources, which may result in performance decrease on some GPUs. In addition, since workloads may differ from layer to layer, it is hard for users to set the number of streams for various GPUs. Hence it is necessary to determine the proper number of CUDA streams by considering both the device characteristics and kernel execution configurations.

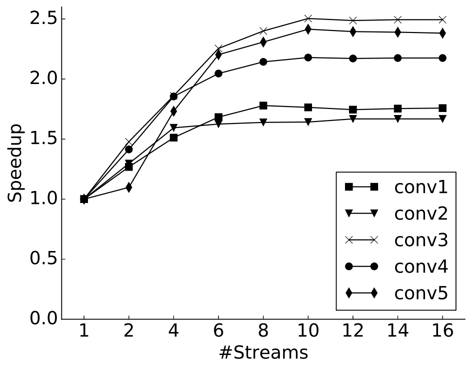


Figure 2: Speedups of CaffeNet's convolution layers on P100.

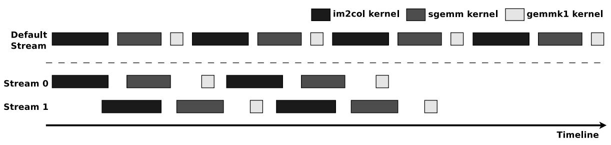


Figure 3: Timeline of kernels in the conv1 layer(MNIST) with multiple CUDA streams.

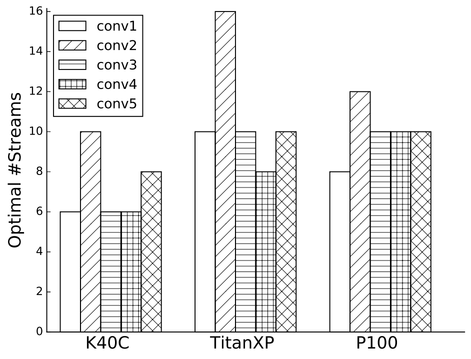


Figure 4: Best observed number of concurrent streams for CaffeNet's layers.

3 FRAMEWORK DESIGN AND ANALYSIS

We first present the overall architecture design of GLP4NN, which is followed by an introduction to the proposed kernel analytical model. At the end of this section, we show the cost analysis of the proposed framework and model.

3.1 Architecture Design of GLP4NN

Fig. 5 shows all basic system components of GLP4NN, which are *resource tracker*, *kernel analyzer*, *stream manager* and *runtime scheduler*, respectively. GLP4NN supports multiple GPUs on the same machine. Each GPU device is assigned with a private kernel analyzer and runtime scheduler, and all GPUs in the same machine share a public resource tracker and stream manager. Fig. 6 shows the workflow among the basic components. We will describe more details about them in the remainder of this section.

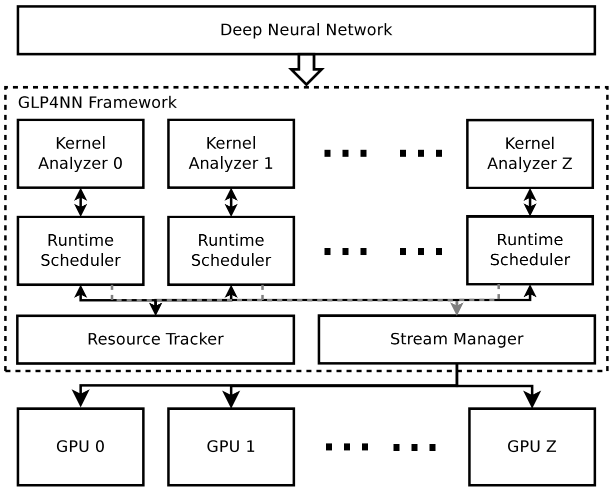


Figure 5: Overview of the GLP4NN framework.

Resource Tracker. This module is responsible for collecting and maintaining kernel execution information at runtime. To avoid gathering trivial kernel runtime information and reduce the profiling and memory overheads, a self-defined compact profiler based on NVIDIA CUPTI library, called *kernel profiler*, is integrated into the proposed framework instead of utilizing existing profiling tools, such as NVIDIA Visual Profiler [27] and Vampir [39]. Then the kernel information collected will be passed to and parsed by the *kernel parser*, which is essential to the proposed model. This module can be customized to support other analytical models.

Kernel Analyzer. The *kernel analyzer* is exploited to analyze the execution configurations of kernels to get the proper number of concurrent kernels with the analytical model proposed. Within this module, *concurrency analyzer* is responsible for kernel analysis and *concurrency maintainer* is used to manage analysis results relative to a specific GPU. The analytical model to be utilized can be customized by developers. This module is invoked by the *runtime scheduler* when the kernel profiling is finished, and its output is utilized to initialize the *stream pool* for launching kernels.

Stream Manager. To support concurrent kernel execution without consuming too many system thread or process resources on the host side, a *stream manager* is designed within the GLP4NN framework. According to CUDA programming guide, a stream is a sequence of commands that are executed in order, and on the other hand, different streams may issue their commands concurrently. A *default stream* is used to maintain the default stream allocated to a CPU thread, and it is often utilized to synchronize all other streams constructed by the same thread or collect kernels' execution information. Hence, a *concurrent stream pool* is designed to maintain all concurrent streams on the current device and the default CUDA stream is utilized to perform synchronization operation based on the scheduling algorithm.

Runtime Scheduler. The *runtime scheduler* module is in charge of invoking the asynchronous resource tracker and obtaining the concurrency configuration from the kernel analyzer to initialize the corresponding stream pool in the stream manager module. Specifically, this module is also responsible for dispatching kernels to GPU streams. In this paper, we take a round-robin scheduling policy for simplicity.

Fig. 6 presents GLP4NN's workflow. Take as an example the forward pass of the conv1 layer in CaffeNet on K40C. There are three kernels needed to be computed, i.e., im2col, sgemm and gemmk. At the beginning, the *runtime scheduler* checks whether configurations of these kernels have been collected. If not, it will invoke the *resource tracker* to gather the profiling information of these kernels, e.g. im2col is initialized with a [18,1,1] grid and 33 registers per thread. Then the information gathered is parsed by the kernel parser and further analyzed by the *kernel analyzer*. In this example, the output of the *kernel analyzer* is 3, which is utilized to initialize the concurrent stream pool. The *runtime scheduler* will take the result into account to dispatch kernels in the following iterations.

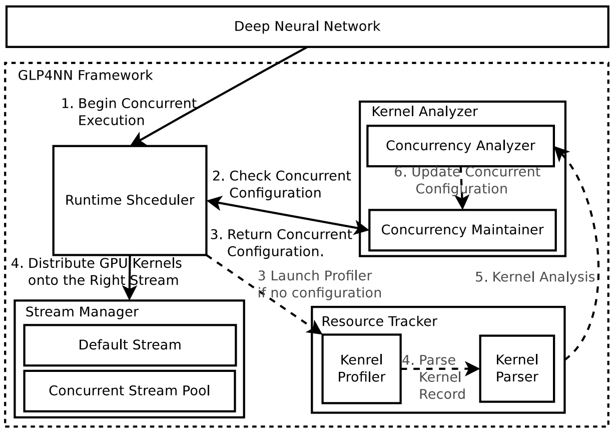


Figure 6: Workflow and all submodules of GLP4NN.

3.2 Analytical Model

Previous performance models on the GPU [35] mainly focus on the analysis of a single kernel. Authors in [32, 47] provide a multiple kernel analysis model to estimate the runtime and IPC of concurrent kernel execution. But there is no guide on how to figure out the proper number of concurrent kernels during the computation of a given network layer for the device utilization improvement. It is difficult for DNN developers to determine an proper number of kernels to be adopted, especially in consideration of a large number of network parameters needed to be tuned and the device diversity. In this paper, we mainly concentrate on designing an analytical model to obtain a proper number of concurrent kernels that can be launched.

Table 2: Notations.

Notation	Definition	Sources
$\#SM$	Number of SMs	Platform Property
C	Concurrency degree	
sm_{max}	Amount of available shared memory per SM	
β_{max}	Amount of resident blocks per SM	
τ_{max}	Amount of resident threads per SM	
$\#\beta_{K_i}$	Number of blocks for K_i	Profiling Input
β_{K_i}	Amount of blocks per SM for K_i	
sm_{K_i}	Amount of shared memory per block for K_i	
τ_{K_i}	Amount of threads per block for K_i	
T_{K_i}	Execution time of kernel K_i	
T_{launch}	Kernel startup time	
$\#K_i$	Total number of kernel K_i launched	Model Output
mem_{tt}	Memory allocated for kernels' timestamps	Others
mem_K	Memory allocated for kernel execution configurations	
mem_{cupti}	Memory required by CUPTI library	
T_p	Profiling time	
T_a	Kernel analysis time	
T_s	Kernel scheduling time	

The basic assumption of this model is that thread blocks are assigned evenly among all SMs on a typical GPU, and thread blocks from different kernels (e.g. β_{K_i} and β_{K_j}, where $i \neq j$) can be placed on the same SM if there are enough resources. For convenience, we summarize notations in Table 2. These notations can be categorized into four groups: *device property*, *profiling input*, *model output* and *others*. *Device property* notations are platform specific and can be easily obtained with the built-in device property or its specification. Parameters from the *profiling input* are provided by the integrated compact profiler or other third-party tools and libraries, such as NVIDIA Visual Profiler [27] and Vampir [39], and those from *model output* are calculated as the output of the proposed model. Notations belonging to *others* are only used in the framework analysis. In the remainder of this section, we focus on the model for NVIDIA GPU, which is more widely used in deep learning applications.

According to the CUDA programming model, each GPU kernel consists of a number of thread blocks, and the thread block is the basic unit to be scheduled onto a SM for computation. $K = \{K_1, K_2, \ldots, K_N\}$ represents the set of kernels that should be executed. The objective of the proposed model is to maximize the occupancy ratio (OR_{SM}) of GPUs, which is defined as the ratio of active warps on a *SM* (ω_{SM}^{active}) to the maximum number of active warps supported by a *SM* (ω_{SM}):

$$OR_{SM} = \frac{\omega_{SM}^{active}}{\omega_{SM}} \tag{1}$$

The number of active warps allowable per *SM* in Eq. 1 is calculated through dividing the total number of active threads per *SM* (τ_{total}) by the warp size (θ), which is 32 in current GPUs.

$$\omega_{SM}^{active} = \frac{\tau_{total}}{\theta} \tag{2}$$

Therefore, the objective is to maximize the number of active threads per *SM*, which is the sum of products of active allowable blocks per *SM* and the number of threads per block for kernel K_i.

$$\tau_{total} = \sum_{i=1}^{N} (\tau_{K_i} \times \beta_{K_i}) \tag{3}$$

There are several factors which may limit the number of kernels that can be launched concurrently, including threads, registers, shared memory and the concurrency degree. According to CUDA documents, additional variables in a thread can be spilled to its local memory when there is no sufficient register space. Normally, the size of thread-specific local memory, which is 512KB per thread on some GPU devices, is much larger than the number of registers allocated. Hence, we treat registers as a kind of *soft* constraint, while others are *hard* constraints. In our analytical model, we only focus on *hard* constraints.

Shared memory per SM. There is a fixed amount of shared memory available on an *SM*, which can also be configured via CUDA APIs. The size of shared memory per thread block is defined as the sum of its static shared memory and dynamic shared memory allocated. As a result, the total size of shared memory allocated for all blocks resident on a *SM* should not exceed sm_{max}.

$$0 \le \sum_{K_i \in K} (sm_{K_i} \times \beta_{K_i}) \le sm_{max} \tag{4}$$

Threads per SM. The maximum number of active threads supported by an *SM* is determined by the hardware. The total number of threads that can be launched should conform to the following equation.

$$0 \le \sum_{K_i \in K} (\tau_{K_i} \times \beta_{K_i}) \le \tau_{max} \tag{5}$$

Concurrency degree. Let C denote the maximum number of concurrent kernels that can executed in the GPU, which is generally different across different types of GPUs. The final number of kernels that can be launched in parallel should be no more than C.

$$1 \le \sum_{K_i \in K} \#K_i \le C \tag{6}$$

By assuming that only a single host thread is responsible for dispatching kernels, T_{K_i}/T_{launch} can be used as a rough estimate of the maximum number of kernel K_i that can be launched concurrently. Considering resource constraints, the number of kernel K_i that can be executed in parallel should conform to the equation below.

$$\begin{aligned}\#K_i \le min\{\lceil \frac{T_{K_i}}{T_{launch}} \rceil, \frac{\tau_{max} \times \#SM}{\tau_{K_i} \times \#\beta_{K_i}}, \\ \frac{sm_{max} \times \#SM}{sm_{K_i} \times \#\beta_{K_i}}\}\end{aligned} \tag{7}$$

Besides, Eq. 8 can be utilized to estimate the number of blocks to be placed onto a single stream multiprocessor for kernel K_i.

$$\beta_{K_i} = \lfloor \#\beta_{K_i} / \#SM \rfloor \tag{8}$$

As described above, the computation of τ_{total} is a kind of mixed integer linear programming problem, which can be solved easily with many modern well-optimized libraries. The final number of streams to be initialized is calculated by Eq. 9.

$$C_{out} = \sum_{i=1}^{N} \#K_i \tag{9}$$

3.3 Framework Analysis

In this part, we first prove that GLP4NN conforms to the network-agnostic and convergence-invariant properties. Then, we demonstrate that the proposed framework is lightweight.

3.3.1 Model Analysis. There are a number of studies on the model analysis of deep learning frameworks [16, 24, 37], both in distributed and centralized systems. In this paper, we consider the *network-agnostic* and *convergence-invariant* property proposed in [37].

Network-agnostic. Network-agnostic means that an optimization strategy does not rely on any particular data layout nor any specialized and highly optimized libraries for neural layers. GLP4NN is a light-weight framework, which aims at accelerating the training or inference of neural networks by exploiting features of new-generation GPUs, especially the concurrent kernel execution. It mainly works on the kernel level by launching kernels concurrently while preserving kernel dependencies. As described in Section 2.1, the batch training algorithm is widely utilized in various networks, including CNNs and RNNs, and samples from the same batch can be independently processed in parallel, which is called batch-level parallelism. In this paper, we apply the proposed framework to the training of convolutional layers by exploiting the batch-level parallelism, which corresponds to the loop in line 2 of Algorithms 1 and 2. It can be easily extended to other network layers adopting the batch training method. Hence, it conforms to the *network-agnostic* property.

Convergence-invariant. Convergence-invariant refers to that an optimization strategy or approach does not change any parameters in a neural network. For DNN developers, this is an important advantage. Before the training of a DNN model, a parameter pre-tuning process is often conducted to ensure an appropriate convergence. It is required to keep the effect of this process. Since GLP4NN framework can be utilized at the batch level as mentioned above, it neither changes the computation inside a kernel nor breaks kernel dependencies. Thus, no network parameters will be changed and the convergence rate will keep invariant between the original and GLP4NN-based implementation. It implies that the training of a neural network with GLP4NN can converge to a stable state, i.e. a local optimal state as the execution without GLP4NN. An experiment result can be found in Section 4.2.3.

3.3.2 Cost Analysis. In this section, we analyze that our approach has very low space and time overheads through a cost model.

Space analysis. To collect and analyze the kernel execution information, additional memory spaces are needed to be allocated. The total memory space mem_{total} allocated by this framework can be defined in Eq. 10. mem_{tt} and mem_K can be computed as in Eq. 11. All these three kinds of memory are allocated in the host memory, which will not influence the DNN's training on GPUs, and are safe to be released after kernel analysis finished.

$$mem_{total} = mem_{tt} + mem_K + mem_{cupti} \tag{10}$$

$$\begin{cases} mem_{tt} = \sum_{i=1}^{N} (mem_{tt_i} * \#K_i) \\ mem_K = \sum_{i=1}^{N} (mem_{K_i} * \#K_i) \end{cases} \tag{11}$$

Time analysis. The one-time overhead of GLP4NN can be divided into three parts: T_p is the profiling time needed by the resource tracker, and T_a is the execution time of the kernel analysis procedure. T_s is the time cost of the scheduling algorithm. As in the naive implementation of the proposed framework in this paper, a static scheduling algorithm is adopted and T_s can be safely ignored. Furthermore, T_a is done on the host side and can overlap with GPU workload by further careful optimization.

$$T_{total} = T_p + T_a + T_s \tag{12}$$

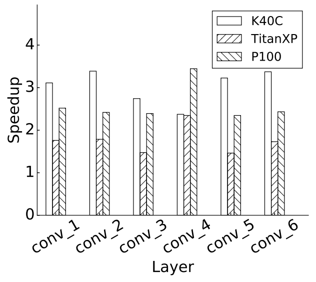
4 EXPERIMENTS

To evaluate the efficiency and effectiveness of the proposed light-weight parallelization framework, we provide an implementation of GLP4NN with the integration into the Caffe framework [17], GLP4NN-Caffe[1]. Caffe is one of the most popular DNN frameworks nowadays. Our experiments have been conducted on NVIDIA GPUs with four well-known neural networks. Especially, we use GNU Linear Programming Kit to solve the MIP problem arisen from the proposed analytical model.

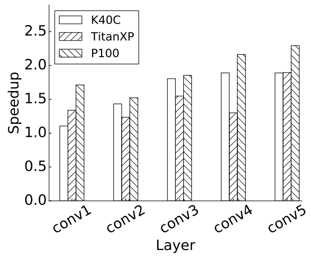
Table 3: Hardware profile.

	Machine configuration		
GPU	K40C	P100	Titan XP
Generation	Kepler	Pascal	Pascal
Core Count	15×192	56×64	30×128
Clock Rate (GHz)	0.745	1.189	1.455
Memory Size (GB)	12	12	12
Memory Bandwidth (GB/s)	288	549	547.7
Memory Type	GDDR5	HBM2.0	GDDR5X
L1 Cache/ Shared Memory per SM	48KB	64KB	48KB
CPU	Xeon E5-2620	Xeon E5-2640	Xeon E5-2650
CPU Cores	6	2×10	12
CPU Clock (GHz)	2.4	2.4	2.2

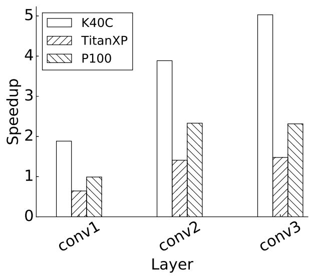
[1]https://github.com/Hao-Tju/GLP4NN_Caffe.



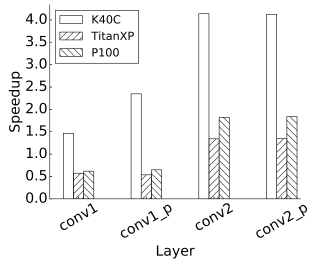
(a) GoogLeNet network.



(b) CaffeNet network.



(c) CIFAR10 network.



(d) Siamese network.

Figure 7: Speedup of GLP4NN-Caffe over naive-Caffe per training iteration on GPUs.

4.1 Experiment Setup

Hardware. We evaluate the proposed framework on NVIDIA GPUs, including Tesla K40C, Telsa P100 and Titan XP. Details about these devices can be found in Table 3. The first GPU is built on the Kepler microarchitecture, while the others are built on the Pascal microarchitecture. We use the GCC compiler suite version g++ 4.8.5 on P100 and Titan XP, and g++ 5.4.0 on K40C. The glpk version utilized in the experiments are glpk-4.63 on P100 and Titan XP, and glpk-4.57 on K40C. For the GPU programming, we utilize the CUDA toolkit 8.0 in all experiments. In this study, we make use of a single GPU into consideration in this paper, but this framework can also be applied to a multi-GPU platform or even distributed environments by optimizing workloads on a single GPU.

Workloads. We evaluate the proposed framework with four famous networks, CIFAR10 [20], Siamese [17], CaffeNet [17] and GoogLeNet [36], on MNIST [11], CIFAR-10 [20] and ImageNet 2012 [34] datasets. Since the convolution layer is one of the most time-consuming layers in CNNs, we only apply GLP4NN to optimize the computation of convolution layers in this paper. Layers' configurations can be found in Table 5, where N is the batch size, C_i is the depth or number of input feature maps, H and W are the height and width of a feature map, F_h and F_w represent the height and width of the convolution filter kernel, C_o is the number of output feature maps or filters, S specifies the stride which controls how the filter convolves around the input volume, and P is the number of pixels to add to each side of the input. There are 22 layers in GoogLeNet with total 59 convolutional units, and we select 6 units from them for convenience. All these networks are available within the Caffe framework [17]. Detailed description of the datasets is shown in Table 4.

4.2 Evaluation on The Proposed Framework

4.2.1 Overall Comparison. Fig. 7 shows speedups of GLP4NN-Caffe over the original Caffe for the four networks, and the number of streams calculated by the proposed analytical model for each convolution layer is presented in Fig. 8. As shown in Fig. 7, the proposed framework can achieve a better performance than original Caffe in most convolution layers.

The speedup of GLP4NN-Caffe for Siamese network on K40C is better than that on TitanXP and P100. By analyzing execution timelines, we find that kernels within these networks' layers are finished in a much shorter time on TitanXP and P100 than on K40C, which results in a low probability of overlapping execution according to Eq. 7.

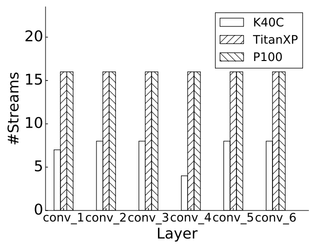
Table 4: Test datasets.

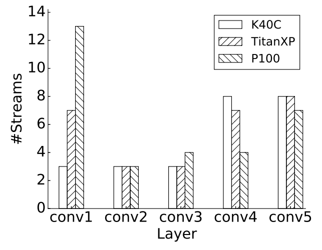
Datasets	Training Images	Test Images	Pixels	Classes
MNIST [11]	60,000	10,000	28×28	10
Cifar10 [20]	50,000	10,000	32×32	10
ImageNet [34]	1.2 million	150,000	256×256	1000

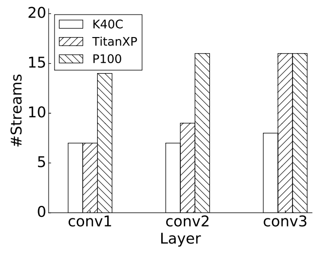
Table 5: Layers of DNNs used in this paper.

Layer	N	C_i	H/W	C_o	F_w/F_h	S	P	Net
conv1	100	3	32	32	5	1	2	CIFAR10
conv2	100	32	16	32	5	1	2	
conv3	100	32	8	64	5	1	2	
conv1	64	1	28	20	5	1	0	Siamese
conv2	64	20	12	50	5	1	0	
conv1_p	64	1	28	20	5	1	0	
conv2_p	64	20	12	50	5	1	0	
conv1	256	3	227	96	11	4	0	CaffeNet
conv2	256	96	27	256	5	1	2	
conv3	256	256	13	384	3	1	1	
conv4	256	384	13	384	3	1	1	
conv5	256	384	13	256	3	1	1	
conv_1	32	160	7	320	3	1	1	GoogLeNet
conv_2	32	832	7	32	1	1	0	
conv_3	32	832	7	384	1	1	0	
conv_4	32	192	7	384	3	1	1	
conv_5	32	832	7	192	1	1	0	
conv_6	32	832	7	48	1	1	0	

Yet in some cases, such as conv1 layer in CIFAR10 network and conv1 and conv1_p layer in Siamese network, the performance of GLP4NN-Caffe is worse than the original implementation. To understand the reason of performance degradation in these layers, Fig. 9 shows the elapsed time of CIFAR10 network on TitanXP and that of Siamese network on Tesla P100, where C and S in the legend are short for CIFAR10 and Siamese, respectively. From this figure, we can find that both conv1 in CIFAR10 and conv1/conv1_p in Siamese can be finished within about 2ms, which may be too short for launch much concurrent kernels. The prior kernel has


(a) GoogLeNet network.


(b) CaffeNet network.


(c) CIFAR10 network.

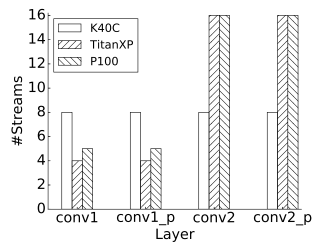

(d) Siamese network.

Figure 8: Number of streams configuration for different networks.

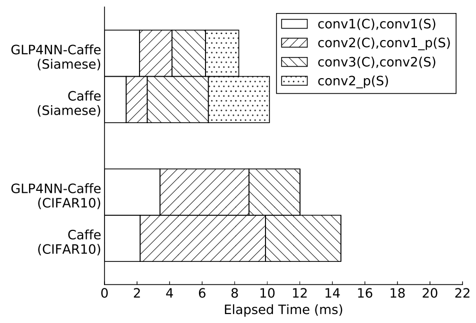


Figure 9: Comparison between GLP4NN-Caffe and Caffe for CIFAR10 on TitanXP and Siamese on P100.

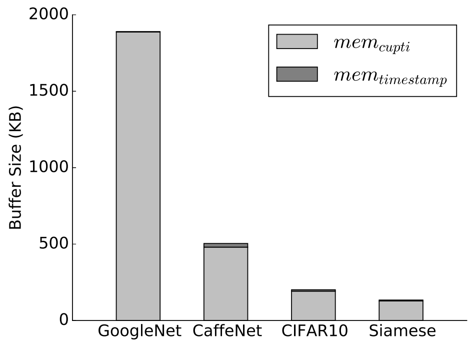


Figure 10: Memory consumption of GLP4NN on GPUs.

finished before the next kernel can execute. However, the overall performance of these two networks has still been improved.

In conclusion, the GLP4NN framework can be helpful for accelerating the training of DNN models automatically on different generations of GPUs. The performance obtained by exploiting this framework may vary based on devices and workloads.

4.2.2 Evaluation on GLP4NN's overhead. Space analysis. Fig. 10 shows the memory size occupied by the proposed GLP4NN framework. mem_{tt} and mem_K is irrelevant to GPU devices, and they only depend on the number of kernels being recorded and analyzed. mem_{cupti} is decided by the CUPTI runtime system, which is much larger than the other two parts in our experiments.

Time analysis. As described in Section 3.3.2, There are two kinds of time cost in the naive implementation of the GLP4NN, as shown in Table 6. T_p, which is proportion to the number of kernels collected, relies on the CUPTI library, whereas T_a is related the GLPK libraries and host CPU. The last column shows the ratio of T_{total} and the total training time, which is always less than 0.1%. Therefore, one-time overhead of GLP4NN is much smaller than the training time of a network, and can be safely ignored.

4.2.3 Convergence Evaluation. To show that GLP4NN conforms to the convergence-invariant property, an example experiment is conducted and the corresponding result is shown in Fig. 11. It can be seen that GLP4NN-Caffe has a similar convergence rate with original Caffe and can achieve a roughly equal local optimal status. The difference between GLP4NN-Caffe and Caffe is caused by the shuffle process while fetching training batch samples.

5 RELATED WORK

In recent years, many frameworks [1, 2, 4, 7, 10, 14, 17, 28, 41] have been released to promote the developement of DNN applications by supporting various accelerators, especially GPUs. The main objective of those frameworks is to provide an easy-to-use and scalable tool for users to design and train DNN models with massive training datasets.

Numerous studies have been done on accelerating the training of DNNs in a distributed environment. Two common parallelism adopted in various frameworks are *data parallelism* and *model parallelism*. In data parallelism, training datasets are divided and processed on multiple accelerators or machines. While in model parallelism, a DNN model is partitioned and trained [13, 18, 19, 45]. To maintain shared data among devices, the *parameter server* architecture [16, 24] is adopted in many frameworks, which can support various synchronization patterns such as BSP [3], stale synchronous parallel (SSP) [6, 9, 12] and value-bounded asynchronous parallel [42].

To speed up the training of DNN applications on GPUs, many works have been done in recent years. NVIDIA has published a set of GPU-powered libraries, including cuDNN [5], cuBLAS [30], etc. These libraries can be directly used by developers easily but may get a performance degradation with small-scale data. Moreover, NVIDIA also presented NCCL [29] to reduce the communication cost between multiple GPUs without considering the computing efficiency of applications. Authors in [43] proposed multiple optimizations for gradient computation on GPUs. Apart from these works, [22, 25, 40] concentrate on reducing arithmetic complexity in convolution layers, while [23, 26, 33] aim at optimizing the memory access and management of neural layers. Nevertheless, these strategies require programmers to update existing codes and only works for a single kernel. GLP4NN is designed to improve the efficiency of concurrent kernel executions in order to improve the efficiency of DNN training.

Some emerging hardware features have been introduced to new-generation GPUs, including concurrent kernel execution as well as more processing elements and active blocks per SM, which bring new spaces for application optimization on GPUs. In [37], authors proposed a coarse-grain parallelization strategy, and authors in [48] developed node-level parallelization for DNNs based on a conditional independent graph (CIG). All the above two parallelization strategy are based on OpenMP technology and may occupy too many CPU threads, which will eliminate the potential of CPU-GPU cooperations, especially in cloud environment [38]. Moreover, these strategies still require programmers to determine the number of threads to be executed, and introduces extra hyper-parameters for programmers to tune.

6 CONCLUSION AND FUTURE WORK

In this paper, we proposed GLP4NN, a light-weight parallelization framework for deep neural networks on GPUs, consisting of four independent modules. We implemented a *resource tracker* module to track and analyze kernels' configurations at runtime. Then the collected information is processed by the *kernel analyzer* module to obtain the proper number of kernels to be launched in parallel with the proposed analytical model in Section 3.2. Moreover, we designed a *stream manager* module to support concurrent kernel execution and CUDA stream management. Finally, we also integrated a *runtime scheduler* into GLP4NN to launch kernels concurrently and interact with other modules. Experiment results show that GLP4NN-Caffe can achieve a speedup of up to 4X over the Caffe.

In the future, several works could be done to further improve the performance of the GLP4NN framework. First, we will continue to improve the performance of the analytical model to get better concurrent kernel execution configurations by considering and supporting complex kernel dependencies, such as the dataflow-like dependency model in Tensorflow. Second, since there are always many kernels needed to be launched concurrently, kernel reordering and kernel fusion technologies may be helpful to gain better training performance of neural networks models, especially for small kernels. Third, we will try to provide a distributed implementation of the proposed framework to support modern distributed DNN frameworks.

ACKNOWLEDGMENTS

This work is supported by the National Natural Science Foundation of China (61602336, 11573019), the Joint Research Fund in Astronomy (U1731423, U1531111) under cooperative agreement between the National Natural Science Foundation of China (NSFC) and Chinese Academy of Sciences(CAS), and a MoE AcRF Tier 2 grant (MOE2017-T2-1-122) in Singapore. Additionally, we would like to acknowledge the financial support provided by the China Scholarship Council (NO.201706250162) during a visit of Hao Fu to the National University of Singapore and the device donation from NVIDIA Corporation.

Table 6: One-time overhead of GLP4NN on GPUs.

Model	GPU	T_p (ms)	T_a (ms)	T_{total} (ms)	Ratio
GoogLeNet	K40C	1.042	116.456	117.498	$< 0.1\%$
	P100	1.696	107.829	109.525	
	Titan XP	1.165	124.811	125.976	
CaffeNet	K40C	14.006	12.375	26.381	
	P100	9.392	12.894	22.286	
	Titan XP	11.645	12.663	24.308	
CIFAR10	K40C	0.131	9.975	10.106	
	P100	1.514	6.461	7.975	
	Titan XP	1.230	6.551	7.781	
Siamese	K40C	1.667	6.783	8.450	
	P100	1.762	7.718	9.480	
	Titan XP	1.942	7.763	9.705	

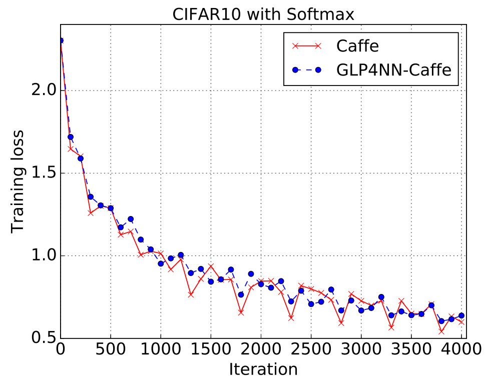


Figure 11: Training CIFAR10 networks on P100.

REFERENCES

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. TensorFlow: A System for Large-Scale Machine Learning.. In *OSDI*, Vol. 16. 265–283.
[2] James Bergstra, Frédéric Bastien, Olivier Breuleux, Pascal Lamblin, Razvan Pascanu, Olivier Delalleau, Guillaume Desjardins, David Warde-Farley, Ian Goodfellow, Arnaud Bergeron, et al. 2011. Theano: Deep learning on gpus with python. In *NIPS 2011, BigLearning Workshop, Granada, Spain*, Vol. 3. Citeseer.
[3] Jianmin Chen, Rajat Monga, Samy Bengio, and Rafal Jozefowicz. 2016. Revisiting Distributed Synchronous SGD. In *International Conference on Learning Representations Workshop Track*. https://arxiv.org/abs/1604.00981
[4] Tianqi Chen, Mu Li, Yutian Yi, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. 2015. MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems. *arXiv preprint arXiv:1512.01274* (2015).
[5] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. 2014. cuDNN: Efficient Primitives for Deep Learning. *arXiv preprint arXiv:1410.0759* (2014).
[6] James Cipar, Qirong Ho, Jin Kyu Kim, Seunghak Lee, Gregory R. Ganger, Garth Gibson, Kimberly Keeton, and Eric P. Xing. 2013. Solving the Straggler Problem with Bounded Staleness. In *HotOS*, Vol. 13. 22–22.
[7] Ronan Collobert, Koray Kavukcuoglu, and Clément Farabet. 2011. Torch7: A matlab-like environment for machine learning. In *BigLearn, NIPS Workshop*.

[8] Alexis Conneau, Holger Schwenk, Loïc Barrault, and Yann Le Cun. 2017. Very deep convolutional networks for text classification. In *Proceedings of the 15th Conference of the European Chapter of the Association for Computational Linguistics*, Vol. 1. 1107–1116.

[9] Henggang Cui, James Cipar, Qirong Ho, Jin Kyu Kim, Seunghak Lee, Abhimanu Kumar, Jinliang Wei, Wei Dai, Gregory R. Ganger, Phillip B. Gibbons, Garth A. Gibson, and Eric P. Xing. 2014. Exploiting Bounded Staleness to Speed Up Big Data Analytics. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. USENIX Association, 37–48.

[10] Henggang Cui, Hao Zhang, Gregory R. Ganger, Phillip B. Gibbons, and Eric P. Xing. 2016. GeePS: Scalable Deep Learning on Distributed GPUs with a GPU-specialized Parameter Server. In *Proceedings of the Eleventh European Conference on Computer Systems (EuroSys '16)*. ACM, New York, NY, USA, Article 4, 16 pages. https://doi.org/10.1145/2901318.2901323

[11] Yann Le Cun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. 1998. Gradient-based learning applied to document recognition. *Proc. IEEE* 86, 11 (1998), 2278–2324.

[12] Wei Dai, Abhimanu Kumar, Jinliang Wei, Qirong Ho, Garth A. Gibson, and Eric P. Xing. 2015. High-Performance Distributed ML at Scale through Parameter Server Consistency Models. In *AAAI*. 79–87.

[13] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Marćaurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, Quoc V. Le, and Andrew Y. Ng. 2012. Large scale distributed deep networks. In *Advances in Neural Information Processing Systems*. Curran Associates, Inc., 1223–1231.

[14] Facebook. 2017. Caffe2. Retrieved March 31, 2018 from https://caffe2.ai/

[15] Fangxiang Feng, Xiaojie Wang, and Ruifan Li. 2014. Cross-modal Retrieval with Correspondence Autoencoder. In *Proceedings of the 22Nd ACM International Conference on Multimedia (MM '14)*. ACM, New York, NY, USA, 7–16. https://doi.org/10.1145/2647868.2654902

[16] Qirong Ho, James Cipar, Henggang Cui, Seunghak Lee, Jin Kyu Kim, Phillip B. Gibbons, Garth A. Gibson, Greg Ganger, and Eric P. Xing. 2013. More effective distributed ml via a stale synchronous parallel parameter server. In *Advances in Neural Information Processing Systems*. Curran Associates, Inc., 1223–1231.

[17] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. 2014. Caffe: Convolutional Architecture for Fast Feature Embedding. In *Proceedings of the 22Nd ACM International Conference on Multimedia (MM '14)*. ACM, New York, NY, USA, 675–678. https://doi.org/10.1145/2647868.2654889

[18] Jin Kyu Kim, Qirong Ho, Seunghak Lee, Xun Zheng, Wei Dai, Garth A. Gibson, and Eric P. Xing. 2016. STRADS: A Distributed Framework for Scheduled Model Parallel Machine Learning. In *Proceedings of the Eleventh European Conference on Computer Systems (EuroSys '16)*. ACM, New York, NY, USA, Article 5, 16 pages. https://doi.org/10.1145/2901318.2901331

[19] Alex Krizhevsky. 2014. One weird trick for parallelizing convolutional neural networks. *arXiv preprint arXiv:1404.5997* (2014).

[20] Alex Krizhevsky and Geoffrey Hinton. 2009. Learning multiple layers of features from tiny images. (2009).

[21] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. 2012. ImageNet Classification with Deep Convolutional Neural Networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1 (NIPS'12)*. Curran Associates Inc., USA, 1097–1105. http://dl.acm.org/citation.cfm?id=2999134.2999257

[22] Andrew Lavini and Scott Gray. 2016. Fast Algorithms for Convolutional Neural Networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 4013–4021.

[23] Chao Li, Yi Yang, Min Feng, Srimat Chakradhar, and Huiyang Zhou. 2016. Optimizing memory efficiency for deep convolutional neural networks on GPUs. In *High Performance Computing, Networking, Storage and Analysis, SC16: International Conference for*. 633–644.

[24] Mu Li, David G. Andersen, Jun Woo Park, Alexander J. Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J. Shekita, and Bor-Yiing Su. 2014. Scaling distributed machine learning with the parameter server. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, Vol. 14. 583–598.

[25] Michael Mathieu, Mikael Henaff, and Yann LeCun. 2013. Fast training of convolutional networks through ffts. *arXiv preprint arXiv:1312.5851* (2013).

[26] Chen Meng, Minmin Sun, Jun Yang, Minghui Qiu, and Yang Gu. 2017. Training Deeper Models by GPU Memory Optimization on TensorFlow. (2017).

[27] NVIDIA. 2008. NVIDIA Visual Profiler. Retrieved March 31, 2018 from https://developer.nvidia.com/nvidia-visual-profiler

[28] NVIDIA. 2015. DIGITS. Retrieved March 31, 2018 from https://developer.nvidia.com/digits

[29] NVIDIA. 2016. NVIDIA Collective Communications Library. Retrieved March 31, 2018 from https://developer.nvidia.com/nccl

[30] NVIDIA. 2017. cuBLAS Library. Retrieved March 31, 2018 from https://developer.nvidia.com/cublas

[31] NVIDIA. 2017. NVIDIA CUDA Profiling Tools Interface. Retrieved March 31, 2018 from https://developer.nvidia.com/cuda-profiling-tools-interface

[32] Johns Paul, Jiong He, and Bingsheng He. 2016. GPL: A GPU-based Pipelined Query Processing Engine. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD '16)*. ACM, New York, NY, USA, 1935–1950. https://doi.org/10.1145/2882903.2915224

[33] Minsoo Rhu, Natalia Gimelshein, Jason Clemons, Arslan Zulfiqar, and Stephen W. Keckler. 2016. vDNN: Virtualized deep neural networks for scalable, memory-efficient neural network design. In *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*. 1–13.

[34] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. 2015. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)* 115, 3 (Dec. 2015), 211–252.

[35] Jaewoong Sim, Aniruddha Dasgupta, Hyesoon Kim, and Richard Vuduc. 2012. A Performance Analysis Framework for Identifying Potential Benefits in GPGPU Applications. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '12)*. ACM, New York, NY, USA, 11–22. https://doi.org/10.1145/2145816.2145819

[36] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. 2015. Going Deeper With Convolutions. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

[37] Marc Gonzalez Tallada. 2016. Coarse Grain Parallelization of Deep Neural Networks. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '16)*. ACM, New York, NY, USA, Article 1, 12 pages. https://doi.org/10.1145/2851141.2851158

[38] Shanjiang Tang, BingSheng He, Shuhao Zhang, and Zhaojie Niu. 2016. Elastic Multi-resource Fairness: Balancing Fairness and Efficiency in Coupled CPU-GPU Architectures. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '16)*. IEEE Press, Piscataway, NJ, USA, Article 75, 12 pages. http://dl.acm.org.libproxy1.nus.edu.sg/citation.cfm?id=3014904.3015005

[39] Vampir.eu. 2007. Vampir. Retrieved March 31, 2018 from https://www.vampir.eu/

[40] Nicolas Vasilache, Jeff Johnson, Michael Mathieu, Soumith Chintala, Serkan Piantino, and Yann LeCun. 2014. Fast convolutional nets with fbfft: A GPU performance evaluation. *arXiv preprint arXiv:1412.7580* (2014).

[41] Wei Wang, Gang Chen, Anh Tien Tuan Dinh, Jinyang Gao, Beng Chin Ooi, Kian-Lee Tan, and Sheng Wang. 2015. SINGA: Putting Deep Learning in the Hands of Multimedia Users. In *Proceedings of the 23rd ACM International Conference on Multimedia (MM '15)*. ACM, New York, NY, USA, 25–34. https://doi.org/10.1145/2733373.2806232

[42] Jinliang Wei, Wei Dai, Abhimanu Kumar, Xun Zheng, Qirong Ho, and Eric P. Xing. 2013. Consistent bounded-asynchronous parameter servers for distributed ML. *arXiv preprint arXiv:1312.7869* (2013).

[43] Zeyi Wen, Bingsheng He, Kotagiri Ramamohanarao, Shengliang Lu, and Jiashuai Shi. 2018. Efficient Gradient Boosted Decision Tree Training on GPUs. In *Parallel and Distributed Processing Symposium (IPDPS), 2018 IEEE International*. Vancouver, British Columbia, Canada.

[44] Zuxuan Wu, Yu-Gang Jiang, Jun Wang, Jian Pu, and Xiangyang Xue. 2014. Exploring Inter-feature and Inter-class Relationships with Deep Neural Networks for Video Classification. In *Proceedings of the 22Nd ACM International Conference on Multimedia (MM '14)*. ACM, New York, NY, USA, 167–176. https://doi.org/10.1145/2647868.2654931

[45] Eric P. Xing, Qirong Ho, Wei Dai, Jin Kyu Kim, Jinliang Wei, Seunghak Lee, Xun Zheng, Pengtao Xie, Abhimanu Kumar, and Yaoliang Yu. 2015. Petuum: A new platform for distributed machine learning on big data. *IEEE Transactions on Big Data* 1, 2 (2015), 49–67.

[46] Li Xu, Jimmy SJ. Ren, Ce Liu, and Jiaya Jia. 2014. Deep Convolutional Neural Network for Image Deconvolution. In *Advances in Neural Information Processing Systems*. Curran associates, Inc., 1790–1798.

[47] Jianlong Zhong and Bingsheng He. 2014. Kernelet: High-throughput GPU kernel executions with dynamic slicing and scheduling. *IEEE Transactions on Parallel and Distributed Systems* 25, 6 (2014), 1522–1532.

[48] Fugen Zhou, Fuxiang Wu, Zhengchen Zhang, and Minghui Dong. 2017. Node-Level Parallelization for Deep Neural Networks with Conditional Independent Graph. *Neurocomputing* 267 (2017), 261–270.