TurboDL: Improving the CNN Training on GPU With Fine-Grained Multi-Streaming Scheduling

Hai Jin, *Fellow, IEEE*, Wenchao Wu, Xuanhua Shi, *Senior Member, IEEE*, Ligang He, *Member, IEEE*, and Bing Bing Zhou, *Member, IEEE*

Abstract—*Graphics Processing Units* (GPUs) have evolved as powerful co-processors for the CNN training. Many new features have been introduced into GPUs such as concurrent kernel execution and hyper-Q technology. It is challenging to orchestrate concurrency for CNN (*convolutional neural networks*) training on GPUs since it may introduce synchronization overhead and poor resource utilization. Unlike previous research which mainly focuses on single layer or coarse-grained optimization, we introduce a critical-path based, asynchronous parallelization mechanism, and propose the optimization technique for the CNN training that takes into account global network architecture and GPU resource usage together. The proposed methods can effectively overlap the synchronization and the computation in different streams. As a result, the training process of CNN is accelerated. We have integrated our methods into Caffe. The experimental results show that the Caffe integrated with our methods can achieve 1.30X performance speedup on average compared with Caffe+cuDNN, and even higher performance speedup can be achieved for deeper, wider, and more complicated networks.

Index Terms—Deep learning, parallelism optimization, scheduling, GPU

1 Introduction

Deep neural networks (DNN) have been widely applied for solving problems in many practical fields such as image classification, object detection, speech recognition, and language translation. Since training deep neural networks is a very time and resource consuming task, general-purpose *graphics processing units* (GPUs) are often used to accelerate the neural network training process. Several deep learning platforms have been developed to train DNN, especially convolution neural networks on GPUs [1], [2], [3], [4]. It should be noted that although the existing platforms are optimized for current GPUs, they may need to be revised as the GPU architectures evolve in order to make efficient use of the added features in new architectures and retain good performance. This type of re-optimization is a non-trivial task.

Graphics Processing Units (GPUs) have evolved as powerful co-processors for many applications. In addition to the significant increase in on-chip resources, e.g., more and faster compute cores, larger shared memory , and more registers, many new features have also been introduced recently, such as Hyper-Q technology, concurrent kernel execution, and dynamic parallelism. Since the NVIDIA Kepler architecture, concurrent kernel execution is designed to enable the simultaneous running of multiple CUDA streams on GPU. Multiple streams bring several desired advantages. For example, data transfer between host memory and device memory can be carried out in parallel with the kernel computation to effectively overlap the computation with the communication/memory-copy. The execution of multiple kernels can also be interleaved to enhance the resource utilization. Many GPU programs have applied the concurrent kernel technique to enhance the performance. A well known example is the CNN (*convolution neural network*) training framework Caffe+cuDNN [2], [5].

By applying the concurrent kernels, we can also effectively enhance the performance of CNN training which requires frequent synchronization during the computation. A convolution neural network consists of a number of layers, ranging from several to a few hundreds of layers. The BP (*back propagation*) algorithm is widely used for CNN training. BP is an iterative computational algorithm and in each iteration a forward computation is performed layer by layer from the first layer to the last to obtain the losses. Then a backward computation is performed to back-propagate the losses to update weight parameters for each layer in the reverse order.

In the CNN training it may take a model multiple millions of such iterations to converge. During this process, the strict global synchronization is needed. Generally speaking, simply creating multiple streams for a computational layer between two consecutive synchronization points may not give a substantial performance improvement as the accumulative overhead from frequent global synchronization

- *Hai Jin, Wenchao Wu, and Xuanhua Shi are with the National Engineering Research Center for Big Data Technology and System, Services Computing Technology and System Lab, Cluster and Grid Computing Lab, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, Hubei 430074, China. E-mail: {hjin, wcwu, xhshi}@hust.edu.cn.*
- *Ligang He is with the Department of Computer Science, University of Warwick, CV4 7AL Coventry, United Kingdom. E-mail: Ligang.He@warwick.ac.uk.*
- *Bing Bing Zhou is with the School of Computer Science, The University of Sydney, NSW 2006, Australia. E-mail: bing.zhou@sydney.edu.au.*

Manuscript received 15 Oct. 2019; revised 28 Feb. 2020; accepted 23 Mar. 2020. Date of publication 4 May 2020; date of current version 11 Mar. 2021. (Corresponding author: Hai Jin.)
Recommended for acceptance by Xuehai Qian and Yanzhi Wang.
Digital Object Identifier no. 10.1109/TC.2020.2990321

may be high and can have significant impact on the overall performance. Moreover, workloads often vary significantly between inter-layer and inner-layer. Frequent synchronization can result in poor resource utilization on GPU.

However, we argue that in each layer the computational task can be partitioned into several sub-tasks with careful dependency analysis for the whole training DAG. Although all the sub-tasks need to be synchronized at the beginning as they require the output of previous layers as their input, some sub-tasks are just local sub-tasks, the output of which is not passed to next layer as its input. Therefore these outputs do not need to be carried over. More specifically, in the backward computation stage of training, feature maps, weight, and bias in each layer need to be updated independently by a loss function. The gradients for feature maps are carried over as input of next layer, while the computation and update of weight and bias gradient are local to each layer. Therefore, the feasibility of fine-grained task parallelism and the discrepancy of dependency between back-propagation of losses and the updating of weight and bias gradient bring opportunity to promote performance, if handled properly.

Existing studies mainly focus on single layer optimization, coarse-grained parallel or distributed communication reduction. They fall short in matching the characteristics of the network architecture and the dependency with the supporting GPU, which may introduce synchronization overhead and low resource utilization. In this work, we exploit this global dependency discrepancy to embrace the new features of GPU such as concurrent kernel executions, aiming to improve GPU resource utilization while reducing the synchronization cost. First, through the careful dependency analysis of the network modelled as DAG (*directed acyclic graph*). we apply the fine-grained task parallelism supported with multi-stream to the inner-layer. Then, the asynchronous execution is performed on inter-layers to eliminate the synchronization cost and balance the workload between different layers. Last, a critical-path based scheduling strategy is developed to both effectively use contending resources and fully use idle resources. More specifically, we propose the following methods.

1) We apply the fine-grained task parallelism by careful dependency analysis and utilize multiple streams on GPU, each being responsible for one gradient subtask. If one stream does not need all the computing cores, those idle cores can be allocated to other streams so as to increase inner-layer resource utilization.
2) Only the stream which computes the gradient of feature map has to be synchronized at both the beginning and the end of the computation phase in each layer. In order to overlap the synchronization and the computation in different streams, we adopt the asynchronous mechanism, such as *events*, to control the execution of the streams for updating local weight and bias. As a result, the strict global synchronization can be eliminated. With the asynchronous execution, the gradient computation of weight and bias in a layer can be performed concurrently with the computation in its successive layers. Consequently, the workload among different layers can be balanced and resource utilization can be improved.
3) We identify the streams which need strict synchronization discussed above. We call them *critical* streams. The streams which update local parameters (thus do not need synchronization) are called *non-critical* streams. We propose a critical-path based priority scheduling strategy. This way, the execution of non-critical streams can be delayed and performed only on idle computing cores left by the critical streams. By doing so, we can effectively utilize the resources that are either idle or contended in the whole iterations.

We implement our methods into Caffe and develop an efficient deep learning framework called TurboDL. The experimental results show that TurboDL is able to improve the performance by approximately 30 percent.

The rest of this paper is organized as follows. The detail of TurboDL and the optimization methods are presented in Section 2. The system design and implementation are described in Section 3. The experiments with TurboDL are presented and the results are analyzed in Section 4. Related work is discussed in Section 5 and finally, the conclusions and future work are presented in Section 6.

2 Methodology

In this section we elaborate our motivation first and then propose a few simple yet effective optimization methods for the CNN training, which include the fine-grained parallelism in the inner-layer, an asynchronous execution mechanism to eliminate the synchronization cost and balance the inter-layer workload, a critical path based scheduling policy to effectively utilize the resources that are either idle or contended in the whole iterations.

2.1 Motivation

Changes of both neural network models and GPUs architectures drive us to rethink the optimization scope for DNN training. On the one hand, neural networks are becoming wider and deeper with complicated, dense connections and multi-paths due to the introduction of more convolution layers with small kernels [31], depth-wise separable convolution [11], deep residual learning [12], channel shuffle [13], and dense connection [14]. On the other hand, advanced features are gradually introduced to new generation GPUs, such as concurrent kernel execution and hyper-Q technology, which provide users more opportunities to improve the performance of their applications.

We installed the popular deep learning frameworks, Caffe [2] along with cuDNN, on morden NVIDIA GPUs P100 and K20 and conducted a number of experiments using typical neural networks such LeNet [33], VGG [30], ResNet [12], GoogLeNet [31], CaffeNet [32] to identify the performance issues of these popular frameworks. We utilized the NVIDIA profile tool [27] to obtain the resource usage of GPUs. Although Caffe+cuDNN uses the optimized functions from cuDNN and applies concurrent streams and batch techniques to enhance the resource utilization, we observed poor GPU resource utilization while the kernel occupancy was low (less than 30 percent on average) even when being run with multi-streams. The warp occupancy

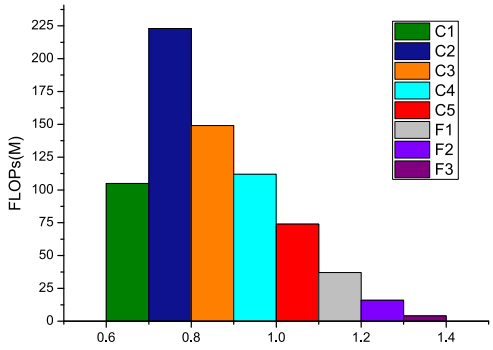


Fig. 1. Flops for main layers in AlexNet. C and F stand for convolution and full connection layer respectively.

for most kernels was lower than 25 percent during the computation. We also observed peaks and valleys in resource utilization and periodic synchronization between the layers during the whole training process (the detailed profiling results are presented in Section 4.3 and the issues for other frameworks are discussed in Section 3.3). Those results indicate that it may not produce a substantial performance improvement by simply applying multiple streams to CNN training but being ignorant of the underlying network structure features and computation characteristics.

There are two issues which impact on the efficiency of resource utilization. A typical CNN consists of multiple different layers such as convolution, polling, batch normalization, and loss. The CNN training is a very time consuming process. In particular, computation for the backward phase is more expensive than that for the forward phase, and most computations are performed in the convolution layers. Since the computation is performed layer by layer, the imbalanced workloads among different layers can significantly affect the resource utilization. For example, in AlexNet as illustrated in Fig. 1, the workload in terms of flops for main layers such as convolution and full-connection can vary by as much as 56X. Even for the same type of convolution, the flops can vary from 74M to 223M. Other neural networks have similar characteristic. Since the output of one layer is the input of the next layer, the parallelization in most existing CNN platforms is coarse-grained and they perform the computation synchronously from one layer to another. Frequent synchronization between layers can also reduce the resource utilization. Based on above observation, we propose the techniques in the following subsections to effectively address these problems.

2.2 Data Dependency Analysis and Fine-Grained Task Partitioning

Multi-stream concurrency is an efficient way to utilize the characteristic of GPUs multi-cores. For the complicated GPU workloads such as CNN training, our studies suggest that the fine-grained parallelism is feasible. Through dependency analysis and data parallelism, we can decompose the task into independent sub-tasks with each sub-task being run by a kernel concurrently, so as to improve resource utilization in GPU.

For the CNN training, previous research mainly focuses on the parallelism 1) between layers such as independent scale convolution layers in inception [31], 2) between different images in a mini-batch, and 3) between different convolution groups in an image. We call those parallel methods the coarse-grained parallelism [35]. Finding fine-grained parallelism between different sub-tasks in a layer for CNN training can potentially improve resource utilization further. Specifically, a convolution layer involves the following computations in CNN training.

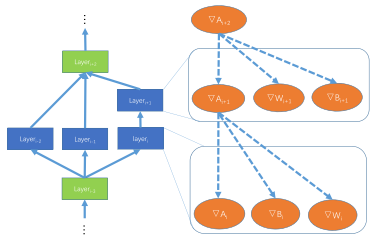


Fig. 2. Fine-grained decomposition of one layer in backward propagation.

$$A_{i+1} = W_i \otimes A_i + B_i \tag{1}$$

$$\nabla W_i = \nabla A_{i+1} \oplus A_i \tag{2}$$

$$\nabla B_i = \sum \nabla A_{i+1} \tag{3}$$

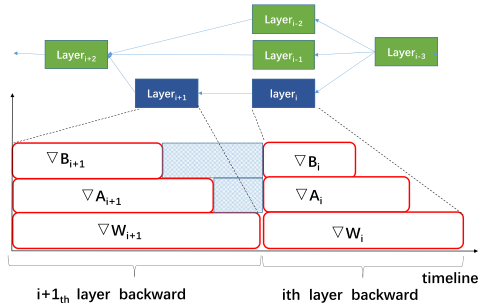
$$\nabla A_i = \nabla A_{i+1} \oplus W_i. \tag{4}$$

The forward convolution computation can be expressed by formula 1. In this formula A_i, W_i, and B_i are activation (also called feature map), weight, and bias, respectively, in convolution layer i. In the backward computation the gradients computation for feature, weight, and bias of layer i, denoted by ∇A_i, ∇W_i, and ∇B_i, respectively, are calculated using Formulas 2, 3, and 4.[1]

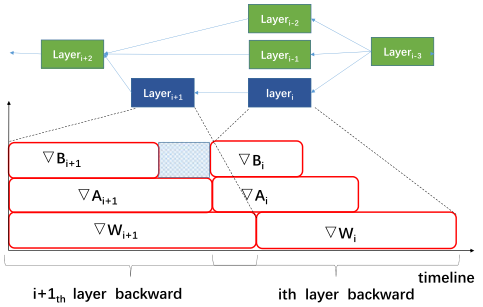
We can see from the above formulas that the computation of ∇A_i, ∇W_i, and ∇B_i all need ∇A_{i+1} from layer $i+1$ in addition to local data A_i and W_i in backward stage , while ∇A_i, ∇W_i, and ∇B_i are independent to each other. More general, several layers of a typical CNN architecture are depicted in Fig. 2 to illustrate the data dependencies between the layers.

In the figure, the rectangles represent the layers (Note it is a non-linear CNN for generality and a linear CNN is just a special case). The orange ellipses on the right represent sub-tasks for ∇A, ∇W, and ∇B in two consecutive layers i and $i+1$, while the data dependencies are denoted by the dashed blue arrows. Using this representation we can easily draw a fine-grained DAG to identify the data dependencies for backward (or forward) computations in CNN training. Based on the derived DAG, we implement those sub-tasks with individual kernels and then create multiple streams, each for running one sub-task in a layer, to realize the concurrent kernel execution. Moreover, the characteristic of the fine-grained dependency provides us the space for further cross-layer optimization, which will be discussed in next subsection. It is

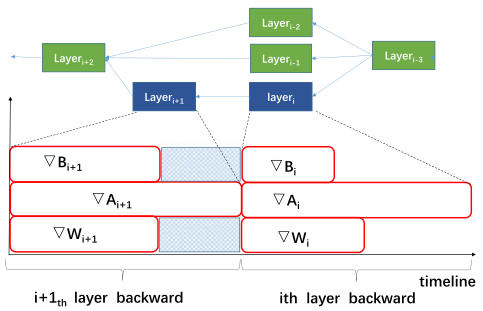
1. For brevity, we have simplified the formulas to highlight their core dependencies, in which $\otimes$ stands for convolution operation, which can be implemented by various methods (such as imtocol and then matrix multiplication), while $\oplus$ stands for the backward convolution process by matrix multiplication or other methods.



(a) Synchronization between layers



(b) Asynchronous execution between layer (1)



(c) Asynchronous execution between layer (2)

Fig. 3. Synchronous/asynchronous execution between layers in backward propagation. Blue striped rectangle indicates the synchronization overhead.

worth noting that except weight definitions, this sub-task decomposition can be applied to many layers with learnable parameters such as convolution, full-connection.

2.3 Asynchronous Execution Between Layers

Now the computational task in each layer is partitioned into several sub-tasks. Although all the sub-tasks will be synchronized at the beginning as they require the output of the previous layers as their input, the synchronization relation of those sub-tasks may only rely on some particular parts of the previous layers (not on the entire layers). The outputs of some sub-tasks in previous layers are kept local and do not need to be carried over to the current layer. We call these sub-tasks local sub-tasks. This insight provides us new scope for further performance improvement. We introduce the pipeline mechanism along with asynchronous execution to further improve the degree of parallelism between layers. By doing so, we can eliminate strict global synchronization in many occasions and also balance the workloads between different layers.

After partitioning the task of a layer into several sub-tasks and identifying the data dependencies, we can apply concurrent kernels to execute the sub-tasks in each layer simultaneously, as shown in Fig. 3a. As mentioned previously, simply running multiple streams may not achieve substantial performance improvement. The main problem is the strict synchronization between layers, that is, a global synchronization is placed at the beginning of layer i to block its computation until the completion of the computations in layer $i+1$. This kind of frequent global synchronization incurs large overhead and causes severe load imbalance across layers. As the consequence, GPUs resources may be wasted, which causes low resource utilization, especially in the backward computation stage.

Fortunately, fine-grained task parallelism not only provides us with the opportunity to increase resource utilization with kernel concurrency but also eliminate a lot of synchronization overhead with asynchronous execution, which is impossible in previous coarse-grained layer-centric approaches. We can see from Fig. 2 that the sub-task for activation gradient map ∇A_i in layer i only needs ∇A_{i+1} as its input and can start immediately when ∇A_{i+1} becomes available from layer $i+1$. Thus it is independent of both ∇W_{i+1} and ∇B_{i+1} and no longer waits for completion for all computation of layer $i+1$. Applying the asynchronous execution mechanism such as *events* to control the sub-task execution, we may largely remove the strict global synchronization as partial sub-tasks in a layer can be fully asynchronous scheduled in a pipeline manner. An ideal case is depicted in Fig. 3b. In the figure the computation of ∇A_{i+1} and ∇A_i are still strictly synchronized. However, the synchronization can effectively overlap the computation of ∇W_{i+1}. The computation, e.g., of ∇W_{i+1} in this example, in layer $i+1$ is also performed simultaneously with the computation in layer i or other layers to achieve the workload balance across the layers.

Two more issues need to be discussed here. First, to make our asynchronous execution of multi-stream work, the initial task partition is very important. We need to carefully partition the task into sub-tasks so that only a subset of the sub-tasks need synchronization at both ends, while different sub-tasks should be independent to each other. Second, so far our optimization technique provides the opportunity to overlap the computation and synchronization across the layers. Fig. 3b shows one possible situation. Another possible situation which is not ideal is depicted in Fig. 3c. We can see from this figure that if ∇A_{i+1} takes the longer time to complete in layer $i+1$, no computation in layer i can start before completion of ∇A_{i+1}. This is effectively the similar situation as the one where strict global synchronization is performed between layers. Another optimization technique to address this problem is discussed in next sub-section.

2.4 Critical Path Based Sub-Task Scheduling

As discussed above, fine-grained inner-layer parallelism and asynchronous inter-layer execution produce a higher degree of parallelism. When GPUs resource is sufficient, these different sub-tasks can execute concurrently in individual streams and kernels, such as ∇A_i, ∇W_i, ∇B_i, ∇W_{i+1}, and ∇B_{i+1} (note that ∇A_{i+1} has completed). However, GPUs resource may not be always sufficient to afford running the kernels in parallel especially when ∇A_{i+1} is longer than

∇W_{i+1}. When the resource shortage occurs, the problem arises how we can schedule different sub-tasks in an efficient manner so that the resource is efficiently utilized and the overall iteration time is minimized.

From the perspective of DAG, different sub-tasks in a layer have different characteristics, some are local sub-tasks with long reuse distance between iterations (specifically, the distance between the gradient computation point in current iteration to the parameter usage point in subsequent iteration for the same layer), while some are on the critical path of the DAG. Based on this observation on the discrepancy of dependency, we propose a novel critical-path based scheduling mechanism, which assigns different priorities to the sub-tasks according to their paths in CNN training.

The execution order of the sub-tasks affects the global iteration time and resource efficiency. If the output of a sub-task (e.g., the one for computing ∇A_{i+1}) in layer $i+1$ needs to be synchronized and used immediately as the input of layer i in the backward phase, the sub-task is deemed as a critical sub-task. The critical sub-tasks then constitute the critical path. The sub-tasks whose outputs are not needed immediately (for example, the sub-tasks for calculating ∇W_i and ∇B_i, whose outputs are only needed in next iteration) are classified as non-critical sub-tasks. The paths containing the non-critical sub-tasks are non-critical paths.

Then we assign different priorities to those paths utilizing path discrepancy. The rationale behind this assignment strategy is that 1) prioritizing the execution of critical tasks will advance the execution of the following layers, and can also effectively overlap the computation of non-critical sub-tasks with the synchronization of the critical sub-tasks; 2) the streams that run the non-critical sub-tasks can take full advantage of idle resources in GPU. The non-critical sub-tasks are not restricted to run in the same layer as the critical sub-tasks, but can be run in other layers in which there are free GPU resources. For example, the computations of ∇W_{i+1} and ∇B_{i+1} are not restricted to layer $i+1$, but can be run in a layer (e.g., layer i) which is the predecessor according to the topological order when there are idle resources in the layer. This strategy not only speeds up the execution of critical sub-tasks, but also significantly improves the resource utilization during the whole training process. Next we will discuss the priority policy in detail by conducting the analysis for the critical paths, starting from the analysis in a single layer to the analysis for the entire network.

Path Analysis in a Single Layer. In a single layer, there are different types of computation paths as follows in the CNN training. 1) Parameter gradient paths such as ∇W_i and ∇B_i. 2) Activation gradient paths, such as ∇A_i. 3) Parameter update paths, which update the weight according to gradient. 4) Data transfer paths, which exchange the gradients in the parameter server [19], [20] architecture with multiple nodes or one node equipped with multiple GPUs.

These computation paths have different characteristics. For example, activation gradient computation paths are critical paths as their outputs are needed immediately by the subsequent layers along with strict synchronization. Therefore, delaying the execution of the sub-tasks on this type of paths will also delay the start of the subsequent layers. Scheduling the sub-tasks in these paths with the high priority can reduce the overall training time.

Parameter gradient paths are non-critical since the execution of a sub-task on these paths can be delayed as long as it can finish before the forward computation in next iteration for the same layer. Therefore, it is possible to postpone the executions of the sub-tasks in these paths until better timing, such as when GPUs resource utilization is low or the critical sub-tasks are in synchronization.

Transfer paths in the parameter server architecture have the similar feature as the parameter gradient paths since the sub-tasks in these paths can overlap their communication with the computation by starting pushing the data to parameter server after the computation of parameter gradient and finishing pulling from parameter server before the computation in next iteration for the same layer.

Take the situations in Figs. 3a and 3b as an example. When the critical-path based scheduling is applied, ∇A_{i+1} is assigned with a high priority as it is on the critical paths according to the above analysis. Therefore thread blocks from kernel ∇A_{i+1} have the higher priority to be scheduled and run on GPUs, which leads to the prompt start of the ith layer. On the contrary, ∇W_{i+1} is on the non-critical paths. Its execution can be postponed until fewer critical sub-tasks are utilizing the GPU resources (e.g., ∇A is in synchronization, or the workload of ∇A is low in some particular layers). Moreover, ∇W_{i+1} as the non-critical sub-task does not have to be restricted to run in layer $i+1$, but can be run concurrently with the sub-tasks in other layers. By doing so, it not only balances the workload among all layers, but also reduces the idle GPU resources in the whole iterations as much as possible. This method reduces the computation time of the critical sub-tasks significantly in the whole training process while imposing no side effect on the execution of non-critical sub-tasks.

Path Analysis of the Entire Network. In the entire network, there are multiple groups of convolution channels for one image or a mini-batch. Moreover, there are multi-scale convolutions and dense connections to previous layers. All these complicated connections among layers increase both the quantity and the complexity of the paths in a network. In this section, we extend the path analysis in a single layer to that in the entire network.

In order to gain insight into this complex network structure, we would like to identify the global critical path in the entire network. We abstract the DAG representing the entire network to a sequence of layer blocks. Each layer block includes of a number of layers. Each block starts from a fork layer and ends with a merge layer, which represents synchronization. Fig. 4 illustrates the structure of an exemplar network in terms of layer blocks. The entire network consists of two layer blocks (blocks 1 and 2) in this figure. Block 1 starts from layer L1 (a fork layer) and ends with layer L6, which is a merge (i.e., synchronization) layer. After block 1 is completed, block 2 then starts from Layer L7 (fork) and ends with Layer 13 (merge). The layer-block structure of this type is ubiquitous in many networks, such as inception unit in GoogLeNet [31], residual block in ResNet [12], and dense block in DenseNet [14].

Since the layer blocks are run in sequence, we only need to analyze the critical paths in each layer block. In a layer block, there are different paths from the fork layer to the merge layer, which we call layer paths since a path consists

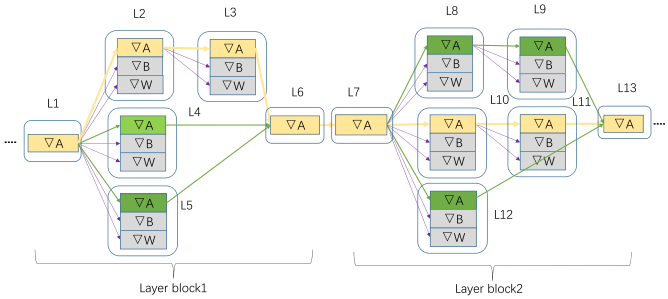


Fig. 4. The path analysis for the entire network.

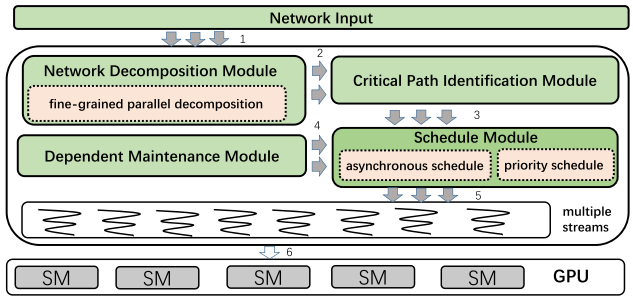


Fig. 5. System architecture.

of layers. We assign different priorities to different layer paths (Note that there are two levels of priorities, the priority for layer path and the priority for sub-tasks as presented in previous subsection). The path priority is assigned as follows. We estimate the computation time of activation gradient (i.e., ∇A_i) in each layer in a path. The total computation time of ∇A_i in all layers of a path is regarded as the distance of the path. The longest layer path in a layer block, which is the critical path in the block, is assigned with the highest priority. Note that when assigning the path priority, we only consider the computation time of ∇A_i, not ∇B_i and ∇W_i, since ∇B_i and ∇W_i can be run fully asynchronously with the layers in other layer blocks. Take Fig. 4 as an example.[2] We assign the highest priority to the yellow layer path in the backward layer block 1, which includes layers $L1$, $L2$, $L3$, and $L6$, since it is the longest layer path in the block.

After assigning the priorities to the layer paths, the method presented in path analysis in a single layer is then applied to assign the priorities to individual sub-tasks in each layer. With this hierarchical priority assignment, our scheduling strategy is aware of the underlying network structure as a DAG and is able to work effectively in the ever-increasingly complicated networks as they evolve towards the trend with denser connections and more paths.

3 System Design and Implementation

We apply our methods discussed in previous section into the CNN training. Then based on Caffe [2], we develop a highly efficient neural network training framework, called TurboDL. In this section, we first present the architecture of our system by introducing the main components. Then we describe how we integrate our three methods into Caffe [2] and also discuss a few issues.

3.1 System Overview

Fig. 5 shows the system architecture of TurboDL. The system contains four main modules: network decomposition module, critical path identification module, schedule module, and dependency maintenance module. Network decomposition module receives the network as input, then decomposes the network into fine-grained sub-tasks. The critical path identification module identifies the critical paths in the decomposed DAG by taking into account network structure and computation dependency. The schedule module schedules the kernels of different paths to run in parallel on GPUs with different priorities (specially, with multiple streams). The dependency maintenance module maintains the synchronization through events and callback to guarantee the correct training.

Network Decomposition Module. The network decomposition module automatically processes the network file (such as prototxt file). By doing so, TurboDL does not require any change in users' program codes and therefore TurboDL is transparent to the users. It collects the whole network information including connection relatives of each layer, layer type, input tensor dimensions, whether the parameters of each layer needs to be updated, memory footprint through automatic inference, GPUs information such as the number of cores, register, shared memory capacity, peak flops. Note that we use the same methods from [10], [17] to estimate the calculation time of each layer and each sub-task, and revise it with profiling the stand-alone running of one iteration since it has been shown that the CNN training has the characteristic of repetitive computation and predictability [34].

Based on these information, we further decompose the network into fine-grained sub-tasks in each layer such as parameter gradient computation, activation gradient computation, and parameter updating, and implement these sub-tasks in individual kernels. Then edges among these fine-grained sub-tasks are reconstructed according to their dependencies. In our current implementation, the sub-task decomposition process for a layer is manual. We analyze the mathematical relationship between the input and output of the layer, obtain independent computational subsets for each input through the automatic derivative function. If an unknown layer is encountered, it can be decomposed recursively into basic known operations such as linear layer, and the dependency among the subtasks corresponding to each part of the input is then recorded. After the decomposition, the entire fine-grained DAG is constructed automatically according to the dependency.

We batch all the images in a mini-batch to form a big kernel as cuDNN [5]. Thus, it reduces the cost of kernel launching because a bigger matrix multiplication is more efficient than multiple small matrix multiplications [6]. We rebuild a more accurate DAG in which each vertex represents a computation sub-task with its estimated time and each edge represents their reading after writing dependency. Therefore it is completely different from the original DAG in Caffe [2], MXNet [1], and TensorFlow [4]. If the vertices of an edge in the DAG are in different paths, the edge leads to the automatic injection of synchronization codes. The DAG construction is conducted automatically. First, the popular layers

2. Note that some layers may not have W or B such as the MaxPooling layer.

(about 15 typical layers) are decomposed into sub-tasks (individual kernels) offline. Then when the network is provided online, the DAG is re-factored layer by layer according to the topological order. In particular, the fine-grained nodes and edges are added based on the dependency between the layers. This process is conducted only once for the whole training with thousands of iterations. Moreover, it is scalable as its complexity depends on the number of layers in the network, which is usually in the range of dozen to hundreds.

Algorithm 1. Critical Path Identification and Construction Algorithm

Inpput: G:fine-grained graph; β, α:relative priority adjustment parameter
Output: $paths$:paths consisting of all sub-tasks with priority

for each $node \in G$ **do**
  compute $DD(node)$ using SSP;
end for
topological sorting for each node in G
segment G into $layer\_blocks$
for each $block$ in $layer\_blocks$ **do**
  for each $layer\_path$ in $block$ **do**
    $t(layer\_path) \leftarrow \sum time(\nabla A_i)(i \in layer\_path)$
  end for
  $Sort(layer\_path, t)$
  assign priority to $layer\_path$ according to order
  for each $layer\_path$ in $blocks$ **do**
    create paths[sub-task.type.num]
    for each $layer$ in $layer\_path$ **do**
      for each sub_task in $layer$ **do**
        AssignPath($sub\_task, DD(sub\_task), paths$)
        if DD(sub_task)==1 **then**
          $sub\_task.path \leftarrow criticalpath$
        end if
      end for
    end for
    for $path \in paths$ **do**
      if $path.sub\_task.type == \nabla B$ **then**
        $path.priority \leftarrow layer\_path.priority - \beta$
      else
        if $path.sub\_task.type == \nabla W$ **then**
          $path.priority \leftarrow layer\_path.priority - \alpha$

        else
          $path.priority \leftarrow layer\_path.priority$
        end if
      end if
    end for
  end for
end for

Critical Path Identification Module. After the fine-gradient DAG graph is built, more opportunities for inner-layer and inter-layer parallelization can be exploited. The critical path identification module is responsible for path identification, classification, and construction. We implement the critical path analysis and hierarchical priority assignment for each layer block. The critical-path identification algorithm is outlined in Algorithm 1. In the algorithm, DD (*dependency distance*) is the metrics to quantify the importance of different sub-tasks, defined by the distance from a certain node (sub-task) to its first reused node across iterations, which is similar

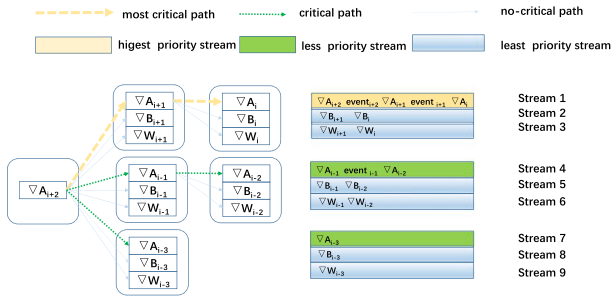


Fig. 6. A scheduling example with multiple streams.

to the aforementioned reuse distance. It is worth noting that when there are group convolution applied to a convolution layer, we only need to build the streams for each convolution group as they can be executed in parallel. Thus we omit it in our algorithms for simplicity. Through the critical path identification module, we only need to build as many steams as the number of the layer paths, rather than the total number of sub-tasks for all the layers. This method can reduce the overhead of streams creation, management while retaining the correct dependency relation.

Schedule Module. The schedule module is responsible for scheduling at run-time and implementing our asynchronous and critical-path based schedule strategy. It first builds multiple streams with different priorities using the CUDA API $cudaStreamCreateWithPriority$ and assigns the paths to the streams. At run-time, the schedule module schedules the vertex in the DAG to the corresponding streams and all vertexes in the same path will be assigned to the same stream. These vertexes in the same path can safely maintain dependency and correctness since the kernels in the same stream are executed sequentially. Moreover, different vertexes in different paths can be executed concurrently across the streams as long as there are GPU resources available. If the kernels in different streams have a dependency edge, the dependency maintenance module will apply the event-based lightweight synchronization and callback mechanism to enforce the dependency, which will be presented in next subsection.

As illustrated in Fig. 6, we use a typical partial network architecture with six layers as an example. In the figure, the backward-propagation of the layers is decomposed into nine independent computing paths. Calculations of ∇A_{i+2}, ∇A_{i+1}, ∇A_i are on the critical layer path. Thus we assign them to the stream with the highest priority. While $\nabla A_{i-1}, \nabla A_{i-2}$ are on the less critical path, they have higher sub-task priorities than other sub-tasks for calculating ∇W and ∇B. ∇W and ∇B are in the stream with the least priority. Since ∇A_i is dependent on ∇A_{i+1}, we put them into the same stream. ∇W_i, ∇A_i, ∇B_i have no dependency with ∇W_{i+1} and ∇B_{i+1}, the execution of ∇W_{i+1} and ∇B_{i+1} can be delayed and executed in parallel with other kernels in the subsequent phases until when GPU resource utilization is low. Because ∇A_{i+1} is in the stream with the highest priority, the waiting time of ∇A_i can be reduced, which, as a result, shortens one iteration time.

Dependency Maintenance Module. We assign different sub-tasks in different streams to fully exploit the parallelism. However, problems arise when the reading-after-writing dependency is broken, which may cause the incorrect training results. In Fig. 6, ∇W_i and ∇B_i depend on the results of

∇A_{i+1}. If this dependency is not met, ∇W_i and ∇B_i will get the stale result from last iteration, which can affect the convergence speed. To synchronize ∇A_{i+1} before ∇W_i and ∇B_i, we apply the CUDA $event$ and light-weight synchronization between the streams. When visiting the DAG, if there is a dependency edge between the kernels in different paths, we use CUDA API $cudaEventCreate$ to create an event and record the event at the synchronization point in the source stream by using $cudaEventRecord$, and then use $cudaStreamWaitEvent$ to synchronize and wait in the target stream. For example, we assign $event_{i+1}$ to $stream1$ after kernel ∇A_{i+1}, and synchronize this event before issuing ∇W_i and ∇B_i to the $stream2$ and $stream3$. Moreover, the callback functions are added by calling $cudaStreamAddCallback$ for W_i and B_i. This way, we can update the parameters further asynchronously according to their gradients. In order to reduce the synchronization cost, we use $cudaStreamWaitEvent$ instead of $cudaDeviceSynchronize$ because the former is light while the later synchronizes all the streams, which incurs a higher cost.

3.2 Integrate Into Caffe

We integrate the proposed methods into Caffe, and modify the layer schedule code in the Net class, which is responsible for layer schedule and kernel execution. Unlike Caffe-cuDNN, we create the streams in Net globally rather than create the streams per-layer. By doing so, we can facilitate the global scheduling, easily enforcing our priority assignment in different paths and adding the dependencies if necessary. In the per-layer implementation, we modify the forward and especially backward process to decompose the layer into the fine-grain sub-tasks and implement them in individual kernels, which can then be launched independently.

3.3 Discussion

Some Implement Issues. In this paper, we assign different priorities to the sub-tasks in order to efficiently utilize the GPUs resources across different layers. We implement this method based on the accurate global dependency analysis, the support of multiple streams, and hierarchical priority assignment. There are multiple levels of priority (e.g., there are three priorities, which are highest, less, and least priority, in Fig. 6). However, in our testbed P100, there are only two priorities (-1 and 0). To solve this problem, we propose two methods, In method one, we assign the kernel with least priority to the hardware stream with the priority of 0 (low priority). Both the kernels with the highest and the less priority are assigned to the stream with priority -1. But we control the running order of these two kernels by controlling when the kernels are put into the run-queue of the stream as well as their positions in the queue.

In method two, we add additional synchronization to simulate priority difference between multiple streams with the same setting of original priority. For example in Fig. 6, we set stream 1, 4, 7 with high priority (eg, -1 for P100), and add a synchronization from ∇A_{i+1} to ∇A_{i-2} and ∇A_{i-3}. This method makes the overall priority of stream 1 higher than streams 4, 7. Although both methods may introduce some overhead, our hierarchical path and priority analysis can be applied to the higher-end or future generation GPU devices, in which more priority levels are supported.

Extension to Multiple GPUs. Our method is general and it can be easily extended to the system with multiple GPUs. In the system setting with multiple GPUs, the main difference is an extra parameter propagation process across multiple GPUs through the communication scheme such as using the CPU to act as the Parameter Server and collect the local updates from multiple GPUs [19], [20], or using the all-reduce communication to aggregate local updates [40]. The computation process in each GPU is the same as in Section 2. We can use the following mechanism to further improve the performance of running TurboDL in the multi-GPU setting. In the backward propagation phase, we can use the no-wait propagation optimization presented in [16] to communicate the gradients. Namely, once the gradient of a layer is calculated, we assign the kernel for the gradient propagation to an individual stream and move on to calculate the gradient of the previous layer. By doing so, we can effectively overlap the communication and the computation of the gradients by making use of the GPU support for concurrent communication and computation.

In next iteration, the forward phase of a layer cannot start until its parameters have been updated. Thus we can add a synchronization event between the parameter updating and the forward computation. By using the fine-grained subtask synchronization within a layer across iterations as above, we can effectively eliminate the global synchronization at the beginning of each iteration. This way, we can fully overlap communication and computation, and make next iteration start earlier. Moreover, we can assign different priorities to individual communication streams (for communicating the gradients of the layers). When the reuse distance of a gradient (the reuse distance is defined at the beginning of the second paragraph in Section 2.4) is shorter, the communication stream for sending this gradient can be assigned with a higher priority. This way, we can reduce the waiting time of subtasks.

Extension to Other Applications. Although in this paper, we mainly utilize CNN as the example to show the effectiveness of our methods, our methods have an excellent potential to be applied to other complicated multi-stage applications, such as database query processing, and other network architectures, such as RNN, tree neural network, generative adversarial network, *Graph-based Convolutional Neural network* (GCN). This is because the characteristics of those workloads indicate that the critical path analysis and the asynchronous execution proposed in this work should also be applicable to improve the performance, although more complicated dependencies may have to be taken into account. For example, in RNN, we can build the critical paths not only for the layers but also for the steps within a layer to capture the time dependency. For GCN, we need to analyze the critical path and apply the pipeline and the asynchronous optimization between the layers among the consecutive graph levels. We leave these detailed extension work in these aspects to future work.

Applicability to Other Frameworks. It is worth noting that our methods are equally applicable to other data-flow based deep learning frameworks such as MXNet [1], TensorFlow [4], Caffe2 [3], and PyTorch [41]. Those graph-based analysis systems utilize the data-flow based mechanism to schedule the operators. As long as the inputs for an operator are ready,

the operator can be scheduled to execute. Even though those systems apply many techniques to optimize the graph execution such as operator fusion, common subexpression elimination, they lack the fine-grained decomposition within the operators and ignore the dependency difference between fine-grained subtasks. Also they do not consider the critical path in the network, but use a simple topological order. Due to their coarse-grained nature, those systems also incur the unnecessary synchronization overhead between layers (or operators), lower the resource usage, and cause the limited degree of parallelism as what happen in Caffe-cuDNN. These three optimization methods developed in TurboDL can be applied to those systems too. Extending the existing framework to integrate our method includes the following work: 1) modifying the schedule module of the existing framework, 2) conducting fine-grained decomposition based on the existing DAG, 3) utilizing automatic derivative function and kernels inside the operation to build a fine-grained DAG, 4) analysing the cross-iteration dependency distance to prioritize different streams for different paths before scheduling. The applicability of our methods have been demonstrated by the implementation of the methods in Caffe for fast prototype implementation.

4 Evaluation

To evaluate the efficiency of our proposed optimization methods, we integrate those methods into Caffe [2], a popular deep learning framework used in many deep learning research due to its easy programming feature. Users only need to write a network file to construct their custom network and specify the hyper-parameters. Moreover, Caffe can utilize highly optimized deep learning library such as cuDNN [5] to accelerate the computation for most layers optimized for NVIDIA GPU. Our experiments are conducted on different NVIDIA GPUs with many well-known networks.

4.1 Experiment Setup

Cluster Configuration. We conduct our experiments on two different GPUs, P100 and K20, those two GPUs have different number of cores from diverse architecture generations (Pascal and Kepler respectively) and with different computation capacity. Both of them support multiple streams and hardware-managed connection from Hyper-Q, which allow the parallel execution of more kernels without the effect of false dependency. Our testing platform is equipped with two eight-core Xeon-2670 2.60 GHz CPUs with 32 GB memory, we run our system on CenterOS 7.5, with the gcc version 4.9. We use cuda 9.0 with cudnn V7.4.2 for all the experiments. The experiments are conducted on a single NVIDIA P100 GPU except the experiments in Fig. 10, which are conducted on K20 and P100.

Workload Configuration. Our experiments use three famous datasets for image classification. (1) MNIST, which contains 60K images for training and 10K for testing with 10 categories. (2) CIFAR-10, which contains 50K colored images for training and 10K for testing with 10 categories. (3) ILSVRC-12, which contains 1.28 million training set and 50K test set with 1000 categories. We test our system efficiency with various kind of networks: (1) LeNet [33], (2) AlexNet [29], (3) CaffeNet [32], (4) VGG [30], (5) GoogLeNet [31], (6) ResNet [12]. These networks are the most typical deep convolution networks in the field of image classification.

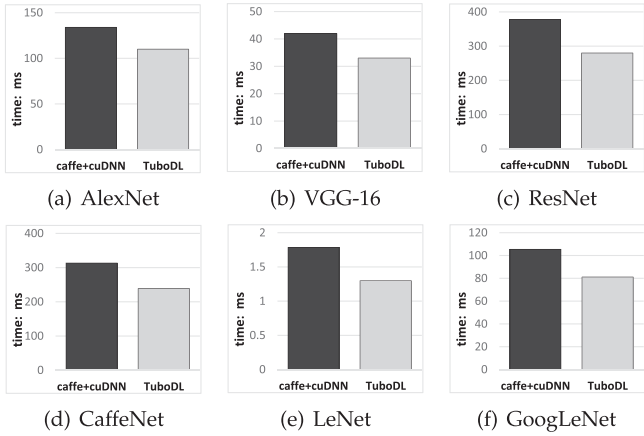


Fig. 7. End-to-end training time of one iteration. X-axis is different systems (Caffe-cuDNN, TurboDL), and Y-axis is the time of one iteration in milli-seconds.

System Configuration. We use the latest version of Caffe (V1.0) as our system prototype and integrate our methods into it (the improved system is called TurboDL). We conduct the experiments to compare TurboDL with the Caffe accelerated with cuDNN [5] (denoted by Caffe-cuDNN).

4.2 Running Time Evaluation

Fig. 7 shows the end-to-end training time of six different networks on different systems. The batch sizes for these six networks are CaffeNet(256), VGG-16(100), LeNet(64), ResNet (32), GoogLeNet(32), AlexNet(128), respectively. Our proposed methods outperform Caffe-cuDNN in all those cases. TurboDL can achieve significant speedup (5.3X on average) compared with the original Caffe (its running time is not presented as it does not represent the best performance in the state of the art). This result supports our claim that our fine-grained parallel method can benefit from combining concurrent kernel execution with the network computation characteristic as original Caffe runs different sub-tasks in a layer in sequence. We achieve the 1.30X performance speedup on average compared with Caffe-cuDNN and LeNet (1.38X), VGG-16 (1.27X), Alexnet (1.21X), GoogleNet (1.30X), ResNet (1.35X), CaffeNet (1.31X), respectively. The speedup is lower than the speedup over original Caffe because Caffe-cuDNN utilizes multiple streams for parallel execution and the batching technique with the dedicated implementation for each kernel on NVIDIA GPUs. However, our methods eliminate the synchronization overhead between layers and exploit the characteristic of the network architecture to accelerate the critical computations, thus further reduce one iteration time.

From the figures, we observe that TurboDL shows different speedup for different networks. For example, our methods achieve higher speedup on ResNet (35 percent) compared with VGG (27 percent). This is due to the difference between their network architectures. ResNet has more complicated dense connections (e.g., residual paths) and deeper layers (e.g., tens to hundreds layers). Thus it provides more opportunity for concurrent execution of kernels and more valley of GPU resource utilization from different

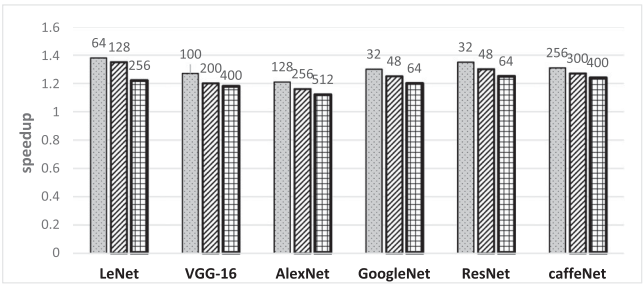


Fig. 8. The end-to-end training time speedup of one iteration for different batch-size. The number on each data bar represents the batch size.

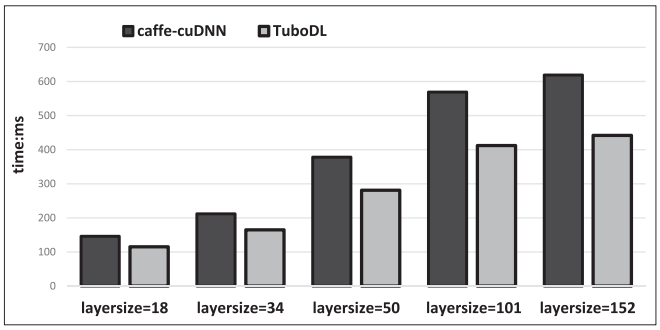
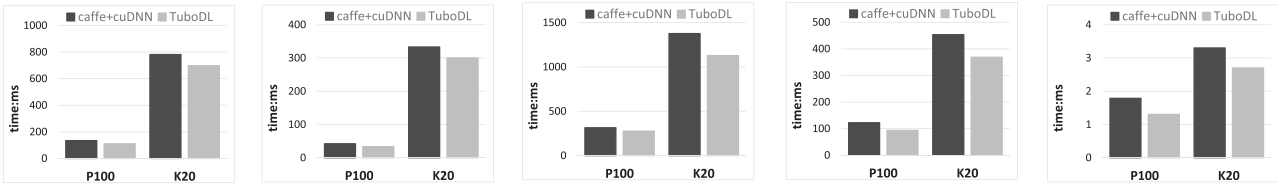


Fig. 9. The running time of one iteration of ResNet with different layer size.

layers, which can be used by other non-critical sub-tasks. This result also indicates that our approaches will bring greater benefits for future networks which are shown to bear the trend of having denser and more paths and even having the combination of multiple models.

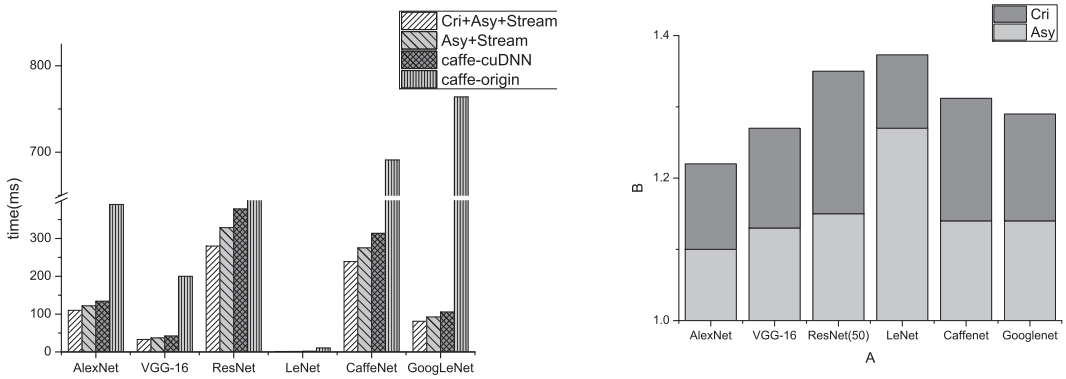
Fig. 8 presents the speedup of TurboDL with different batch-size of LeNet, VGG-16, AlexNet, GoogLeNet, ResNet, CaffeNet on P100. Compared with Caffe-cuDNN, our performance advantage decreases when the batch size increases for those networks. For example, the performance advantage decreases from 27 to 18 percent for VGG-16 when the batch size increases from 100 to 400. This is to be expected because when the batch size increases, the computation complexity of sub-tasks also increases linearly as we batch all images in a mini-batch to a bigger kernel for efficiency, making kernels harder to execute in parallel and leaving little idle resource for inner-layer processing. Therefore the sub-tasks in non-critical paths will block the whole iteration due to the resource shortage. Even in this situation, our methods outperform Caffe-cuDNN because of the better usage of the cross-layer resources. Although large batch improves the computation efficiency, it hinders the statistical efficiency and requires more epochs to converge [39].

Fig. 9 shows the running time of one iteration when the number of layers increases from 18, 34, 50, 101 to 152 with the batch size being 32, 32, 32, 28, and 18, respectively. We reduce the batch size for the last two cases because of the out of memory problem on P100). Our methods achieve the speedup of 1.27X, 1.29X, 1.35X, 1.38X, and 1.40X compared with Caffe-cuDNN, which shows the increasing trend because the path experienced by one iteration becomes longer when the number of layers increases. Since TurboDL is based on the critical path and the asynchronous execution of sub-tasks, it can fill the idle resources between the layers with non-critical calculations, resulting in higher performance. We also measure the performance on different GPUs, K20 and P100. The results are presented in Fig. 10. Our methods achieve better performance compared with Caffe-cuDNN on different GPUs. The speedup on P100 is higher than on K20. This is because there are much less cores and less support of hardware parallelism on K20. The results



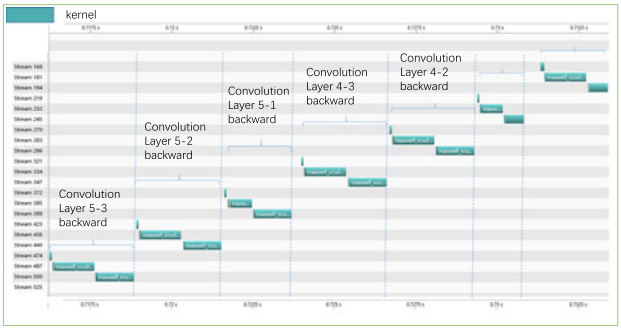
(a) AlexNet (b) VGG-16 (c) CaffeNet (d) GoogLeNet (e) LeNet

Fig. 10. The comparison of running time between different GPUs (P100 and K20).

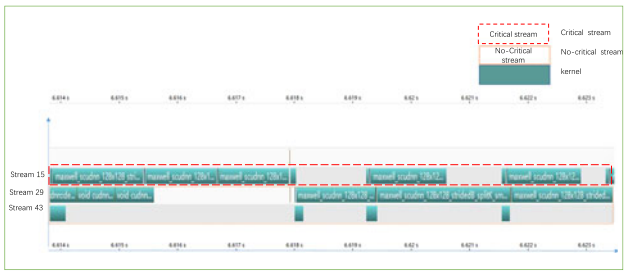


(a) Runtime time of different strategies (b) Speedup of different strategy

Fig. 11. (a). Running time of different strategy combinations for one iteration. (b). Improvement of different strategy combinations compared with the baseline performance obtained by Caffe-cuDNN (speedup=1). "Asy" stands for asynchronous parallel, and "Cri" stands for critical path based scheduling.



(a) caffe-cuDNN



(b) TurboDL

Fig. 12. Profiling results of VGG-16. The X-axis is the time line, and the colored rectangles represent different kernels.

suggest that TurboDL can gain more benefits when the GPUs are more powerful and have higher parallel capacity.

In order to evaluate how much each optimization strategy in Section 2 contributes to the final performance gain, we conduct the experiment with different combinations of the methods to break down where the gains come from. The running time of different strategies is presented in Fig. 11a for LeNet, VGG-16, AlexNet, GoogLeNet, ResNet, and CaffeNet. Since the effectiveness of fine-grained parallelism with multiple streams can be verified by significant improvement over original Caffe. In Fig. 11b, we only depict other two strategies with Caffe-cuDNN as the baseline. For example, when only inner-layer asynchronous parallelism and inner-layer multiple streams are applied on ResNet (50), our methods can achieve the speedup of about 1.15X over Caffe-cuDNN. After adding the critical-path based scheduling, the speedup increases to 1.35X. The result shows both methods contribute to the final performance improvement for different networks.

4.3 Resource Evaluation

Fig. 12 shows the resource profiling results for VGG-16 on Caffe-cuDNN and our system TurboDL. We only present VGG since other networks have similar results. We obtain the line of the execution time by using the NVIDIA visual profiler tools [27]. For Caffe-cuDNN, even if there are multiple streams the profiling result in Fig. 12a shows that: 1) Periodic synchronizations between successive layers exist throughout the whole backward process. 2) Workloads vary significantly from layer to layer. 3) The kernel concurrency is very low in each layer. We also profile major kernels in the training phase. We find that the warp occupancy for most kernels is low (about 25 percent).

The result regarding TurboDL is presented in Fig. 12b, we can see that the kernel concurrency is very high as most kernels from no-critical paths overlap with the kernels in critical paths. At the same time, there are not periodic synchronizations between layers, and the kernels on non-critical paths mostly overlap with other kernels from different layers. This complete asynchronous execution is achieved thus TurboDL can utilize resources from different layers and balance the workloads between different layers as much as possible. Moreover, from the figure, the critical paths are mostly busy, suggesting that the critical paths are prioritized to execution than other less or non-critical paths computations, which results in the shorter overall iteration time. Last, there are tens to hundreds of streams on Caffe-cuDNN (e.g., 38 streams on VGG-16), while there are only a few streams on TurboDL (e.g., 3 for VGG-16). Fewer streams lead to less overhead of creating, managing, and scheduling streams. We also observe the inefficiency of single kernel execution because of the contention of register allocation and underutilization of share memory. We leave these to future work.

TABLE 1
The Convergence Comparison

Iteration	Loss		Iteration	Loss	
	Caffe-cuDNN	TurboDL		Caffe-cuDNN	TurboDL
0	7.21	7.20	1000	1.77	1.75
200	2.52	2.52	1200	1.66	1.62
400	2.29	2.31	1400	1.75	1.62
600	2.07	2.05	1600	1.63	1.59
800	1.91	1.98	1800	1.59	1.58

4.4 Convergence Evaluation

In order to verify that our proposed methods guarantee the correctness through only retaining the least but necessary dependencies, we compare the convergence speed every 200 iterations for VGG-16 with cifar10 dataset as an example. Table 1 shows that the loss decreases as the iteration increases. Our methods have nearly the same loss rate as Caffe-cuDNN, which shows the correctness of our system. Moreover, our methods have shorter time with the same convergence for one iteration. Therefore the overall training time to convergence with thousands of iterations can be reduced. Both systems show uneven convergence trend for model parameters with a long tail effect. This provides the opportunity for future optimization, which we also plan to leave for future work.

We test the inference accuracy on those models trained by TurboDL and Caffe-cuDNN with the identical setting (such as dataset, iteration number, hyper-parameter). The result is the same (the accuracy result is not presented here due to the space limitation). Note that the data dependencies between layers are correctly maintained in TurboDL and consequently, the accuracy of the results is ensured, which is achieved not by modifying the algorithm, but by improving the inner-layer parallelism, removing unnecessary inter-layer synchronization cost, and optimizing the scheduling of fine-grained sub-tasks for the iterations of the whole network. Moreover, our fine-grained decomposition, critical path identification, and priority assignment are all included in the pre-processing phase, which can be amortized by thousands of iterations.

5 Related Work

GPU Memory Optimization for DL. As GPU memory becomes a bottleneck when training large-scale network, several methods are proposed to solve this problem. vDNN [9] and GeePS [8] propose a mitigation strategy to offload intermediate results to host memory and pre-fetch them before needed in backward. Chen *et al.* [21] propose re-computation to trade off computation cost and memory occupancy. MXNet [1] provides a re-use strategy to share memory space for different data through lifetime analysis. However, large models limit practical deployment such as in embedded devices. Many researches try to construct models with less parameters and fewer computation complexity which reduce model size from several thousand MB to several MB, making GPU memory optimization less particularly important.

Dynamic Graph Optimization for DL. Recent deep learning models are moving toward dynamic neural network structure, making training inefficiently for hard-to-batching problem. To solve this, TensorFlow Fold [22] proposes dynamic batching technology to batch different inter-input and inner-input operations. Cavs [7] introduces GAS model from graph processing system and represents dynamic network as static vertex function with a dynamic instance-specific graph to avoid repeated graph construction cost, and facilitates to incorporate static graph optimization technologies. Yu *et al.* [24] propose a programming model for distributed machine learning with dynamic control flow support on TensorFlow. Jeong *et al.* [23] add recursion to the existing programming framework and exploit efficient parallel among nodes. Gao *et al.* [25] propose cellular batching to improve latency and throughput for RNN inference. Those methods try to improve the expressiveness and efficiency for dynamic and recursive networks as they are the important trend for machine learning model in realistic deployment. Our methods are orthogonal to these methods in the literature and can also be applied to these dynamic networks to improve their efficiency.

Training Optimization of Computation for DL. Our work focuses on the CNN training phase with complicated bi-directional dependency between forward and backward computation and iterative characteristic. Li *et al.* [15] try to improve training performance by increasing memory access efficiency. They transform the storage dimension of data and apply kernel fusion technology. In order to add parallelism of operations, data-flow based execution model has been applied to many deep learning frameworks such as TensorFlow [4], MXNet [1], however, those frameworks apply a simple topological order to schedule coarse-grained operations, ignoring the path importance characteristics for the network structure, thus leading to suboptimal performance.

Many hardware manufacturers have proposed efficient linear algebra and deep learning libraries like Intel MKL [26], cuBLAS [28], cuDNN [5], however, those libraries mainly focus on single layer, coarse-grained operation optimization which fall short in small matrix computation [6] and can not be aware of network feature to adapt to the underlying GPU hardware feature. PipeDream [36] proposes a generalized pipeline method with the model being parallelized on multiple workers, which is inherently coarse-grained. Peng *et al.* [37] propose a generic communication scheduler by introducing a unified abstraction and a Dependency Proxy mechanism. These two works focus on the distributed DNN training in which the communication cost is a bottleneck, while our work focuses on increasing the computation efficiency of a single GPU.

Jia *et al.* [38] propose a computation graph optimizer that automatically generates graph substitutions with a formal theorem prover and a cost-based backtracking search to find an optimized graph. Our work is orthogonal to the work discussed above, and acts as a complementary tool to further enhance the performance. The most similar and latest work with us is GLP4NN [18], which tries to improve parallelism by incorporating concurrent kernel execution. However, it only optimizes single layer not the whole training process, which is completely different and orthogonal to our work as we conduct whole network architecture optimization by fine-grained, asynchronous execution, and critical path based schedule with priority.

6 Conclusion

In this paper, we develop TurboDL, an efficient deep learning framework to accelerate the deep learning training. Three key optimization methods are developed in TurboDL, including i) a fine-grained parallel mechanism, which decomposes the computation of a layer to expose more parallel opportunity through accurate dependency analysis, ii) an asynchronous execution strategy to eliminate synchronization cost, break the isolation between layers, overlap computation and balance workloads, and iii) a critical-path based schedule mechanism to reduce overall running time. Although in this paper we mainly utilize CNN as the example to show the effectiveness of our methods, our methods have an excellent potential to be applied to other complicated multi-stage applications. We leave these detailed extension of our work to future work.

Acknowledgments

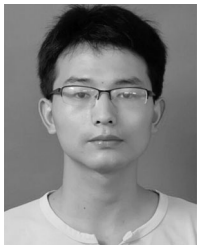
This work was supported by the National Key Research and Development Program under Grant 2017YFC0803700 and NSFC Grant 61832006.

References

[1] T. Chen *et al.*, "MXNet: A flexible and efficient machine learning library for heterogeneous distributed systems," 2015, *arXiv:1512.01274*.
[2] Y. Jia *et al.*, "Caffe: Convolutional architecture for fast feature embedding," in *Proc. ACM Int. Conf. Multimedia*, 2014, pp. 675–678.
[3] Caffe2, 2017, Accessed: Sep. 5, 2019. [Online]. Available: https://caffe2.ai/
[4] M. Abadi *et al.*, "TensorFlow: A system for large-scale machine learning," in *Proc. USENIX Conf. Operating Syst. Des. Implementation*, 2016. pp. 265–283.
[5] S. Chetlur *et al.*, "cuDNN: Efficient primitives for deep learning," 2014, *arXiv:1410.0759*.
[6] M. Zhang, S. Rajbhandari, W. Wang, and Y. He, "DeepCPU: Serving RNN-based deep learning models 10X faster," in *Proc. USENIX Conf. Usenix Annu. Techn. Conf.*, 2018, pp. 951–965.
[7] S. Xu *et al.*, "Cavs: An efficient runtime system for dynamic neural networks," in *Proc. USENIX Conf. Usenix Annu. Tech. Conf.*, 2018, pp. 937–950.
[8] H. Cui, H. Zhang, G. R. Ganger, P. B. Gibbons, and E. P. Xing, "GeePS: Scalable deep learning on distributed GPUs with a GPU-specialized parameter server," in *Proc. Eur. Conf. Comput. Syst.*, 2016, pp. 4:1–4:16.

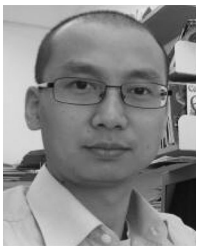
[9] M. Rhu, N. Gimelshein, J. Clemons, A. Zulfiqar, and S. W. Keckler, "vDNN: Virtualized deep neural networks for scalable, memory-efficient neural network design," in *Proc. Annu. IEEE/ACM Int. Symp. Microarchit.*, 2016, pp. 18:1–18:13.
[10] M. Song, Y. Hu, H. Chen, and T. Li, "Towards pervasive and user satisfactory CNN across GPU microarchitectures," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit.*, 2017, pp. 1–12.
[11] F. Chollet, "Xception: Deep learning with depthwise separable convolutions," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2017, pp. 1800–1807.
[12] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2016, pp. 770–778.
[13] X. Zhang, X. Zhou, M. Lin, and J. Sun, "ShuffleNet: An extremely efficient convolutional neural network for mobile devices," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2018, pp. 6848–6856.
[14] G. Huang, Z. Liu, L. van der Maaten, and K. Q. Weinberger, "Densely connected convolutional networks," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2017, pp. 2261–2269.
[15] C. Li, Y. Yang, M. Feng, S. Chakradhar, and H. Zhou, "Optimizing memory efficiency for deep convolutional neural networks on GPUs," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2016, pp. 633–644.
[16] H. Zhang *et al.*, "Poseidon: An efficient communication architecture for distributed deep learning on GPU clusters," in *Proc. USENIX Conf. Usenix Annu. Tech. Conf.*, 2017, pp. 181–193.
[17] M. Song *et al.*, "Bridging the semantic gaps of GPU acceleration for scale-out CNN-based big data processing: Think big, see small," in *Proc. Int. Conf. Parallel Architectures Compilation*, 2016, pp. 315–326.
[18] H. Fu, S. Tang, B. He, C. Yu, and J. Sun, "GLP4NN: A convergence-invariant and network-agnostic light-weight parallelization framework for deep neural networks on modern GPUs," in *Proc. Int. Conf. Parallel Process.*, 2018, pp. 33:1–33:10.
[19] M. Li *et al.*, "Scaling distributed machine learning with the parameter server," in *Proc. USENIX Conf. Operating Syst. Des. Implementation*, 2014, pp. 583–598.
[20] E. P. Xing *et al.*, "Petuum: A new platform for distributed machine learning on big data," in *Proc. ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, 2015, pp. 1335–1344.
[21] T. Chen, B. Xu, C. Zhang, and C. Guestrin, "Training deep nets with sublinear memory cost," 2016, *arXiv:1604.06174*.
[22] M. Looks, M. Herreshoff, D. Hutchins, and P. Norvig, "Deep learning with dynamic computation graphs," 2017, *arXiv: 1702.02181*.
[23] E. Jeong, J. S. Jeong, S. Kim, G.-I. Yu, and B.-G. Chun, "Improving the expressiveness of deep learning frameworks with recursion," in *Proc. Eur. Conf. Comput. Syst.*, 2018, pp. 19:1–19:13.
[24] Y. Yu *et al.*, "Dynamic control flow in large-scale machine learning," in *Proc. Eur. Conf. Comput. Syst.*, 2018, pp. 18:1–18:15.
[25] P. Gao, L. Yu, Y. Wu, and J. Li, "Low latency RNN inference with cellular batching," in *Proc. Eur. Conf. Comput. Syst.*, 2018, pp. 31:1–31:15.
[26] Intel MKL, 2012, Accessed: Sep. 5, 2019. [Online]. Available: https://software.intel.com/en-us/mkl
[27] NVIDIA Visual Profiler, 2008, Accessed: Sep. 5, 2019. [Online]. Available: https://developer.nvidia.com/nvidia-visual-profiler
[28] NVIDIA cuBLAS, 2013, Accessed: Sep. 5, 2019. [Online]. Available: https://developer.nvidia.com/cublas
[29] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," in *Proc. Int. Conf. Neural Inf. Process. Syst.*, 2012, pp. 1097–1105.
[30] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," 2014, *arXiv:1409.1556*.
[31] C. Szegedy *et al.*, "Going deeper with convolutions," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2015, pp. 1–9.
[32] CaffeNet, 2014, Accessed: Sep. 5, 2019. [Online]. Available: https://github.com/BVLC/caffe/tree/master/models/bvlc_reference_caffenet
[33] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," in *Proc. IEEE*, vol. 86, no. 11, pp. 2278–2324, Nov. 1998.
[34] M. Sivathanu, T. Chugh, S. S. Singapuram, and L. Zhou, "Astra: Exploiting predictability to optimize deep learning," in *Proc. Int. Conf. Architectural Support Program. Languages Operating Syst.*, 2019, pp. 909–923.
[35] M. G. Tallada, "Coarse grain parallelization of deep neural networks," in *Proc. ACM SIGPLAN Symp. Principles Practice Parallel Program.*, 2016, pp. 1:1–1:12.
[36] D. Narayanan *et al.*, "PipeDream: Generalized pipeline parallelism for DNN training," in *Proc. ACM Symp. Operating Syst. Principles*, 2019, pp. 1–15.
[37] Y. Peng *et al.*, "A generic communication scheduler for distributed DNN training acceleration," in *Proc. ACM Symp. Operating Syst. Principles*, 2019, pp. 16–29.
[38] Z. Jia, O. Padon, J. Thomas, T. Warszawski, M. Zaharia, and A. Aiken, "TASO: Optimizing deep learning computation with automatic generation of graph substitutions," in *Proc. ACM Symp. Operating Syst. Principles*, 2019, pp. 47–62.
[39] A. Koliousis, P. Watcharapichat, M. Weidlich, L. Mai, P. Costa, and P. Pietzuch, "CROSSBOW: Scaling deep learning with small batch sizes on multi-GPU servers," in *Proc. Very Large Data Base Endowment*, vol. 12, no. 11, pp. 1399–1412, 2019.
[40] A. Sergeev and M. D. Balso, "Horovod: Fast and easy distributed deep learning in TensorFlow," 2018, *arXiv: 1802.05799*.
[41] A. Paszke *et al.*, "Automatic differentiation in PyTorch," in *Proc. Int. Conf. Neural Inf. Process. Syst. Workshop*, 2017.

Hai Jin (Fellow, IEEE) received the PhD degree in computer engineering from the Huazhong University of Science and Technology, in 1994. He is a Cheung Kung Scholars chair professor of computer science and engineering with the Huazhong University of Science and Technology (HUST), in China. In 1996, he was awarded a German Academic Exchange Service Fellowship to visit the Technical University of Chemnitz in Germany. He worked at The University of Hong Kong between 1998 and 2000, and as a visiting scholar at the University of Southern California between 1999 and 2000. He was awarded Excellent Youth Award from the National Science Foundation of China in 2001. He is the fellow of CCF and a member of ACM.



Wenchao Wu is currently working toward the PhD degree with the National Engineering Research Center for Big Data Technology and System, Services Computing Technology and System Lab, Cluster and Grid Computing Lab, School of Computer Science and Technology, Huazhong University of Science and Technology, China. He is doing research on deep learning system.

Xuanhua Shi (Senior Member, IEEE) is currently a professor with the School of Computer Science and Technology, Huazhong University of Science and Technology, China. He is the deputy director of the National Engineering Research Center for Big Data Technology and System (NERC-BDTS). His current research interests include focus on cloud computing, big data processing, and AI systems. He published more than 100 peer-reviewed publications (such as ASPLOS, VLDB, the *ACM Transactions on Computer Systems*, the IEEE *Transactions on Parallel Distributed Systems*). He received research support from a variety of governmental and industrial organizations, such as the National Science Foundation of China, Ministry of Science and Technology, Ministry of Education, European Union.



Ligang He (Member, IEEE) is currently a reader with the Department of Computer Science, University of Warwick, United Kingdom. His research interests include parallel and distributed computing, high performance computing, and bigdata analysis. He has published more than 100 papers in journals (such as the *IEEE Transactions on Parallel Distributed Systems*, *ACM Computing Surveys*, *Journal of Parallel and Distributed Computing*, *Journal of Computer and System Sciences*) and conferences (such as IPDPS, ICPP, ICWS, VLDB, SC).

Bing Bing Zhou (Member, IEEE) received the graduate's degree in electronic engineering from the Nanjing Institute of Technology in China, in 1982, and the PhD degree in computer science from Australian National University, Australia, in 1989. He is currently an associate professor with the School of Computer Science, University of Sydney, Australia. He has been a member of the technical program committees of several international conferences and acted as a reviewer for many international journals, including the *IEEE Transactions on Computers* and the *IEEE Transactions on Parallel and Distributed Systems*.