Accelerating Deep Learning Frameworks with Micro-batches

Yosuke Oyama*, Tal Ben-Nun†, Torsten Hoefler† and Satoshi Matsuoka‡ *
*Department of Mathematical and Computing Science, Tokyo Institute of Technology, Tokyo, Japan, oyama.y.aa@m.titech.ac.jp
†Department of Computer Science, ETH Zurich, Zurich, Switzerland, {talbn,htor}@inf.ethz.ch
‡RIKEN Center for Computational Science, Hyogo, Japan, matsu@acm.org

***Abstract*—cuDNN is a low-level library that provides GPU kernels frequently used in deep learning. Specifically, cuDNN implements several equivalent convolution algorithms, whose performance and memory footprint may vary considerably, depending on the layer dimensions. When an algorithm is automatically selected by cuDNN, the decision is performed on a per-layer basis, and thus it often resorts to slower algorithms that fit the workspace size constraints. We present μ-cuDNN, a thin wrapper library for cuDNN that transparently divides layers' mini-batch computation into multiple micro-batches, both on a single GPU and a heterogeneous set of GPUs. Based on Dynamic Programming and Integer Linear Programming (ILP), μ-cuDNN enables faster algorithms by decreasing the workspace requirements. At the same time, μ-cuDNN does not decrease the accuracy of the results, effectively decoupling statistical efficiency from the hardware efficiency. We demonstrate the effectiveness of μ-cuDNN for the Caffe and TensorFlow frameworks, achieving speedups of 1.63x for AlexNet and 1.21x for ResNet-18 on the P100-SXM2 GPU. We also show that μ-cuDNN achieves speedups of up to 4.54x, and 1.60x on average for DeepBench's convolutional layers on the V100-SXM2 GPU. In a distributed setting, μ-cuDNN attains a speedup of 2.20x when training ResNet-18 on a heterogeneous GPU cluster over a single GPU. These results indicate that using micro-batches can seamlessly increase the performance of deep learning, while maintaining the same overall memory footprint.**

***Index Terms*—Deep learning, convolutional neural networks, performance tuning, micro-batch.**

I. Introduction

Prevalent Deep Neural Networks (DNNs) are becoming increasingly deeper and are trained with large batch sizes. Specifically, recent DNNs contain hundreds of layers [1], [2], and utilize batch sizes in the order of thousands [3]–[5].

Large batches are also favored by distributed data-parallel deep learning frameworks, because they improve GPU utilization, as well as hide the communication of parameter gradients in the computation efficiently. Consequently, the batch size per GPU should be large to achieve better scaling. Since the memory usage of a DNN is nearly proportional to the layer size and the batch size, the GPU memory tends to be used at full capacity in most real-world cases.

This "limited memory scenario" is also exhibited in cuDNN [6], a deep learning kernel library for NVIDIA GPUs. cuDNN provides a variety of computational primitives for neural networks, and is widely used in deep learning frameworks, such as Caffe [7] and others [8]–[10]. cuDNN provides up to eight different algorithms to perform convolutions, each of which requires different temporary storage (workspace) schemes. To guide users to determine the best algorithm, cuDNN provides a function `cudnnGetConvolution*Algorithm` ($\star$ is one of convolution types, `Forward`, `BackwardData` and `BackwardFilter`), that benchmarks all the algorithms and chooses the best algorithm, either with respect to computation time or memory usage. However, if the workspace size requested by a fast algorithm is larger than provided, cuDNN will resort to a slower algorithm that requires less workspace. In fact, cuDNN may require workspace sizes that are as large as the network itself to use efficient convolution algorithms, such as FFT-based convolution [11] and Winograd's algorithm [12] (Figure 1).

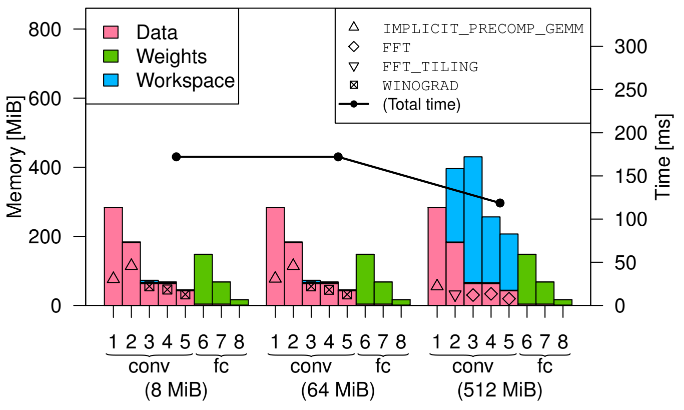


Fig. 1: Per-layer breakdowns of memory consumption (left axis, bars) and computation time of forward and backward passes (right axis, points) of AlexNet's convolutional layers on P100-SXM2. We use three different workspace sizes (8, 64, 512 MiB), and a mini-batch of 256.

In this paper, we propose μ-cuDNN, a transparent wrapper for cuDNN that attempts to mitigate the aforementioned inefficiency. In order to utilize fast convolution algorithms with limited workspace size, μ-cuDNN automatically divides a layer's mini-batch computation into several "micro-batches" and perform multiple convolutions sequentially (Figure 2). μ-cuDNN decouples the statistical efficiency (speed of

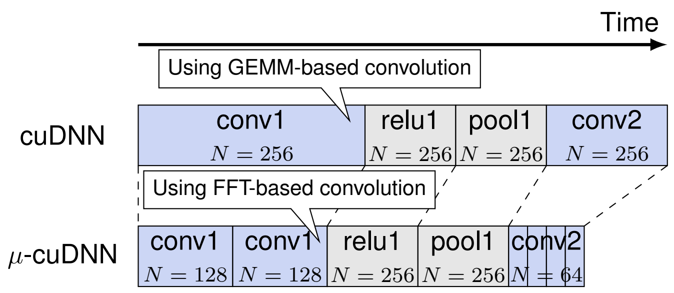


Fig. 2: The conceptual execution timeline of μ-cuDNN. N represents the mini-batch size.

accuracy/loss improvement with fixed number of parameter updates) from the hardware efficiency (speed of computations with fixed number of updates), improving only the latter. *Using micro-batches, μ-cuDNN improves the hardware utilization without incurring any reduction in training accuracy.*

The contributions of this paper are as follows:

- We present a method to automatically divide mini-batch training into several "micro-batches", so that faster algorithms are utilized with tight workspace constraints.
- We propose different workspace allocation policies, which enable optimization of multiple convolutional layers with inter-dependencies.
- We provide a Python interface, compatible with major deep learning frameworks, which suggests high-performing micro-batch divisions over heterogeneous GPU clusters.
- We evaluate μ-cuDNN over two different deep learning frameworks, Caffe and TensorFlow, showing that it can mitigate the inefficiency of cuDNN with several Convolutional Neural Networks (CNNs), AlexNet, ResNet and DenseNet, both in single and multi-node environments.

The source code of μ-cuDNN is available online at https://github.com/spcl/ucudnn.

II. The Anatomy of Convolutional Neural Networks

Convolution operations in CNNs apply multiple filters to a batch of channels of two-dimensional data (Algorithm 1). In particular, input and output tensors are represented as four-dimensional tensors with dimensions (N, C, H, W), where N is the mini-batch size, C is the number of channels, and H and W represent image height and width, respectively. Similarly, the filter tensor is represented as four-dimensional (K, C, V, U) tensor, where K is the number of output channels and V, U represent kernel height and width.

The two-dimensional convolution is composed of seven-nested loops (Algorithm 1). The innermost three loops compute the actual convolution, where one element of the input tensor $\mathbf{X}$ is multiplied and accumulated to one element of the output tensor $\mathbf{Y}$. The remaining loops iterate over all elements of $\mathbf{Y}$. The key observation is that in order to solve

Algorithm 1 Pseudo-code of two-dimensional convolution.

```
1: for($n = 0$; $n < N$; $n$++)                 // Mini-batch loop
2:   for($k = 0$; $k < K$; $k$++)               // Output channel loop
3:     for($h = 0$; $h < H$; $h$++)             // Height loop
4:       for($w = 0$; $w < W$; $w$++)           // Width loop
5:         for($c = 0$; $c < C$; $c$++)         // Input channel loop
6:           for($v = 0$; $v < V$; $v$++)       // Kernel width loop
7:             for($u = 0$; $u < U$; $u$++)     // Kernel height loop
8:               $\mathbf{Y}[n,k,h,w]$ += $\mathbf{W}[k,c,v,u] \times \mathbf{X}[n,c,h+v,w+u]$;
```

the problem described in Section I, there is no dependence inside the mini-batch loop between different iterations. This is intuitive because in training or inference we compute parameter gradients or outputs with respect to different data samples, so this is equivalent to computing N different CNNs concurrently. This observation motivates us to apply loop tiling to the mini-batch loop, so that we can reduce the resident workspace size.

The only exception to the inter-sample independence is the computation of parameter gradients;

$$\frac{\partial L}{\partial \mathbf{W}} = \frac{1}{N} \sum_{n=1}^{N} \frac{\partial L_n}{\partial \mathbf{Y}_n} * \mathbf{X}_n,$$

where L and L_n is the loss function with respect to a mini-batch and a sample n respectively, and $*$ is the convolution operation [11]. The semantics of this computation are, however, not violated by the loop splitting, if each of the iterations is performed sequentially.

In cuDNN, there are three operations related to the two-dimensional convolution; `Forward` for forward computation, `BackwardData` for computing neuron errors in back-propagation, `BackwardFilter` for computing parameter gradients in back-propagation.

Although `Forward` and `BackwardData` can directly be divided into several micro-batches, `BackwardFilter` cannot, since there are output dependencies on the accumulated parameter gradients tensor $\partial L / \partial \mathbf{W}$. However, we can still divide the loops by running `BackwardFilter` multiple times while accumulating the results, i.e., output scale $= 1$ in cuDNN. Therefore, loop splitting can be achieved by repeating cuDNN kernels one or more times for any convolution-related operation, regardless of the underlying method.

III. μ-cuDNN

μ-cuDNN is a transparent C++ wrapper library for cuDNN, which can easily be integrated into most deep learning frameworks [7], [8], [10], [13] (Figure 3). μ-cuDNN can be called either by a deep learning framework as a low-level performance tuning library for cuDNN, or its dedicated Python frontend for high-level performance analysis, as described in Section III-E.

To enable μ-cuDNN, the only modification that needs to be performed to the code is to replace the cuDNN handle type `cudnnHandle_t` with `UcudnnHandle_t`. The μ-cuDNN handle object is an opaque type that wraps the original

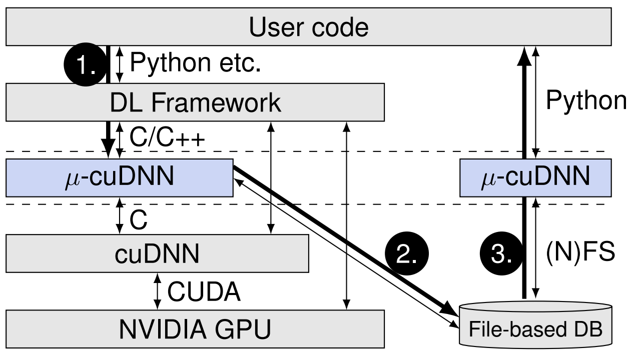


Fig. 3: μ-cuDNN software stack. Users can obtain performance metrics via 1. to 3., as described in Section III-E.

type, such that users can call any cuDNN function. When a convolution operation or a benchmarking function is called with the μ-cuDNN handle object, the μ-cuDNN internally computes the optimal configurations, and returns a virtual algorithm ID and required workspace size. This mechanism enables users to call μ-cuDNN with minimal modification to the original code. For example, only three lines of code need to be modified to introduce μ-cuDNN to Caffe (v1.0).

The implementation of μ-cuDNN is based on overloading a subset of cuDNN functions, where the memory of the μ-cuDNN handle type is structured to act as the cuDNN internal handle for the other calls. We define a cast operator from the μ-cuDNN handle to cuDNN handle so that the framework automatically adopts this method. Using this technique, μ-cuDNN delegates most of the functions to cuDNN, but overrides functions related to the convolutional layers.

μ-cuDNN caches the optimized configurations and the benchmark results into memory and optional file-based database respectively, to skip unnecessary recomputations. This is especially beneficial for networks that replicate convolutional layers of the same size, such as ResNet [2]. In addition, the file-based caching enables offline benchmarking, as well as sharing the results among a GPU cluster via a networked file system.

A. μ-cuDNN Methodology

The key concept of μ-cuDNN is that it automatically divides a mini-batch to several batches (referred to as "micro-batches" in this paper) and optimizes their sizes, to utilize faster convolution algorithms. Our μ-cuDNN library employs one of two workspace utilization policies to optimize micro-batches for convolution kernels (Figure 4):

- **Workspace Reuse (WR)**: WR allocates one workspace per layer, sharing the space between the internal micro-batches. In this scheme, each layer is assumed to use the workspace exclusively, hence the total size of the workspaces cannot be determined before runtime.
- **Workspace Division (WD)**: WD allocates one workspace per network, and assigns different segments to each convolutional layer. WD enables small groups of convolution operations, as in the Inception module [14], to run concurrently with larger workspaces. In WD, the actual workspace is managed by μ-cuDNN rather than the deep learning framework. This is because conventional frameworks allocate each workspace separately, lacking a global view of the entire network's workspace requirements.

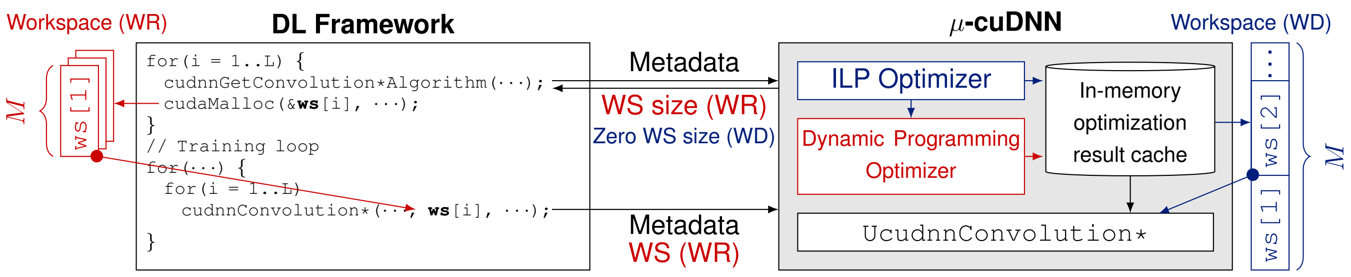
WR and WD both rely on the parameters of one or more convolution kernel(s), the mini-batch size, and the maximum workspace size. The output of μ-cuDNN is a division of the mini-batch, and "micro-configurations"; a pair of a convolution algorithm and micro-batch size for each convolution micro-batch. In this paper, we define "configuration" of a segmented convolution kernel as "a list of micro-configurations". For example, if a kernel with a mini-batch size of 256 is equally divided into four micro-batches and each of them uses algorithm X, the configuration is represented as $\{(X, 64), (X, 64), (X, 64), (X, 64)\}$. Also, we define concatenation of two lists as $+$, such as $\{a, b\} + \{c, d\} = \{a, b, c, d\}$ and $\{a\} + \emptyset = \{a\}$.

B. WR Algorithm

The goal of the WR policy is to minimize $T(B)$, the total execution time with mini-batch size of B using Dynamic Programming (DP), where $T(b)$ is defined as follows:

$$T(b) = \min \left\{ \begin{array}{l} T_\mu(b), \\ \min_{b'=1,2,\ldots,b-1} T(b') + T(b - b') \end{array} \right\},$$

where $T_\mu(b)$ is the fastest execution time of one convolution kernel with a micro-batch size of b, within the workspace constraint. If the first row of the definition of $T(b)$ is smaller



```
for(i = 1..L) {
  cudnnGetConvolution*Algorithm(···);
  cudaMalloc(&ws[i], ···);
}
// Training loop
for(···) {
  for(i = 1..L)
    cudnnConvolution*(···, ws[i], ···);
}
```

Fig. 4: Overview of μ-cuDNN. μ-cuDNN optimizes micro-batch sizes and internally calls cuDNN functions, with either a per-layer workspace (WR) or a part of a global workspace (WD) of a size limit M.

than the second row, μ-cuDNN does not have to divide the batch. Otherwise, it is beneficial to divide the batch into two or more parts, applying the process recursively.

The key point of WR is that the optimal micro-configuration size is deterministic and independent from other kernels. This is because in this case, we assume that multiple kernels do not run simultaneously.

The algorithm of WR comprises three steps, where the mini-batch size is B, and given maximum workspace size is M:

1) For $b = 1, 2, \cdots, B$, WR benchmarks all available convolution algorithms of micro-batch size of b with maximum workspace size of M, using cuDNN. We define the fastest micro-configuration as $c_\mu(b) = (a, b)$ (where a is the fastest algorithm).
2) For $b = 1, 2, \cdots, B$, WR computes $T(b)$, the fastest execution time for micro-batch size of b and $c(b)$, as follows (where $T(0) = 0, c(0) = \emptyset$). $T(b)$ and $c(b)$ are memorized and reused for further iterations.

$$\begin{aligned} \hat{b_\mu} &\leftarrow \operatorname*{argmin}_{b_\mu = 1,2,\ldots,b} \{T_\mu(b_\mu) + T(b - b_\mu)\} \\ T(b) &\leftarrow T_\mu(\hat{b_\mu}) + T(b - \hat{b_\mu}) \\ c(b) &\leftarrow \{c_\mu(\hat{b_\mu})\} + c(b - \hat{b_\mu}) \end{aligned}$$

3) Outputs the optimal configuration $c(B)$.

In practice, μ-cuDNN provides "micro-batch size policies", which determine what micro-batch sizes are benchmarked at the step 1 of the WR algorithm, as follows:

- **all** uses all micro-batch sizes $b \in \{1, 2, 3, \cdots, B\}$. Although this always finds the best solution, it takes $\mathcal{O}(B)$ time for the benchmark.
- **powerOfTwo** uses only power-of-two batch sizes $b \in \{2^0, 2^1, 2^2, \cdots, B\}$. This saves a considerable amount of time since it only costs $\mathcal{O}(\log B)$ time for the benchmark.
- **undivided** uses only the original mini-batch size $b \in \{B\}$. In WR, this option always selects the same configuration as cuDNN, hence this option is only useful to evaluate the overhead of μ-cuDNN.

These policies can be specified via an environment variable or through a special library function in μ-cuDNN. Furthermore, μ-cuDNN supports parallel micro-configuration evaluation, where the aforementioned micro-batches are distributed to different GPUs on the same computing node and tested concurrently. This function assumes that the node contains multiple GPUs of the same kind.

C. WD Algorithm

In the WD scheme, configurations for multiple convolution kernels are optimized, while at the same time the total workspace size should be less than the total workspace limit that users specify. Therefore, WD is a more complex problem than WR, since the configuration of each convolution kernel is no longer independent from others, due to the total workspace size constraint.

To solve this problem, we formulate a 0-1 Integer Linear Programming (ILP)-based optimization algorithm (Figure 5).

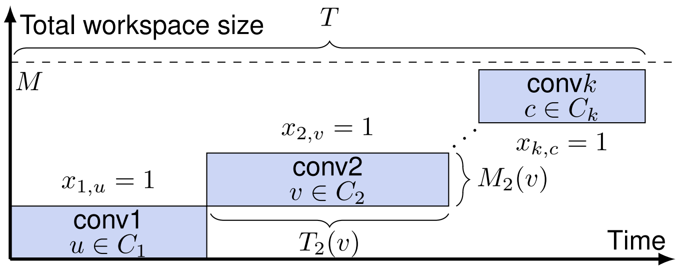


Fig. 5: ILP-based optimization of WD.

Given the set of kernels $\mathcal{K}$ and sets of available configurations C_k of kernel k, WD is solved by minimizing Eq. (1):

$$\min \quad T = \sum_{k \in \mathcal{K}} \sum_{c \in C_k} T_k(c) x_{k,c} \tag{1}$$

$$\text{subject to} \quad \sum_{k \in \mathcal{K}} \sum_{c \in C_k} M_k(c) x_{k,c} \leq M \tag{2}$$

$$\sum_{c \in C_k} x_{k,c} = 1 \ (\forall k \in \mathcal{K}) \tag{3}$$

$$x_{k,c} \in \{0, 1\} \ (\forall k \in \mathcal{K}, \forall c \in C_k), \tag{4}$$

where $M_k(c)$ and $T_k(c)$ are the workspace size and execution time of kernel k with configuration c, respectively. Eq. (2) limits the total workspace size to the user-specified size M. μ-cuDNN uses configuration c on kernel k if and only if $x_{k,c} = 1$, and exactly one of them is selected for each kernel k, according to the constraint in Eq. (3).

1) Desirable Configuration Selection: The challenging problem of the above ILP-based algorithm is that if all possible configurations are evaluated (i.e., all combinations of the number of micro-batch and algorithms), the search-space is in the order of $|A|(|A| + 1)^{B-1}$ configurations for each kernel, which is resulted from $x(b) = \sum_{b_\mu=1}^{b} |A| x(b - b_\mu)$, where $x(b)$ is the number of configurations of a micro-batch size b, A is set of algorithms and B is the mini-batch size. This huge search-space makes the problem impractically large.

Below, we compute a Pareto front to remove undesirable configurations from all possible configurations, without returning any sub-optimal solutions. The resulting Pareto front C_k is then input to the ILP to solve the entire problem.

First, we modify the DP algorithm from WR (Section III-B) to output a set of configurations, rather than the fastest configuration, as follows:

$$C(b) = D\left(\bigcup_{b_\mu = 1,2,\ldots,b} \ \bigcup_{c_\mu \in C_\mu(b_\mu)} \ \bigcup_{c \in C(b - b_\mu)} (\{c_\mu\} + c) \right),$$

where $C_\mu(b)$ is a set of available micro-configurations of micro-batch size of b, and D is a pruning function described below. Note that this outputs $c(B)$ of the WR algorithm as one of its elements; $c(b) \in C(b)$ and $c_\mu(b) \in C_\mu(b)$ for any b.

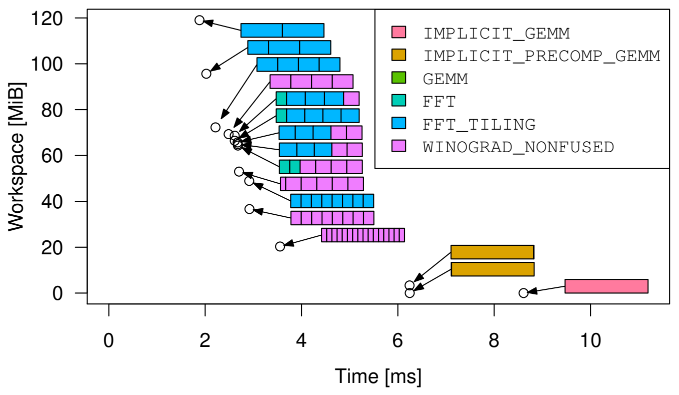


Fig. 6: A Pareto front (in circles) and the corresponding compositions of AlexNet's "conv2" layer (Forward) on P100-SXM2 with a maximum workspace size of 120 MiB, and a mini-batch size of 256. Colored bars corresponding to data points represent the division of the mini-batch and the chosen micro-batch algorithms.

Second, we define the "desirable configuration set" $D(C) \subset C$ as a Pareto front in the two-dimensional (execution time $\times$ workspace size) space:

$$D(C) = \{c \in C | \forall c' \in C\ [T(c) < T(c') \vee M(c) < M(c')]\},$$

where $T(c)$ and $M(c)$ stand for execution time and required workspace size of a configuration c. This definition implies that any $c \in D(C)$ is the fastest configuration among any of the elements of $D(C)$ using a workspace size of $M(c)$ or less. Conversely, if an element $c \in C$ is not in $D(C)$, there is an element that is both faster than c and requires less workspace, hence there is no reason to choose c.

The above pruning drastically reduces the number of variables of Eq. (1), and enables solving the ILP for state-of-the-art deep CNNs in practical time. For instance, the maximum number of desirable configurations of AlexNet's layers we examined in Section IV-B2 was 68, which is much smaller than the exponential order, $|A|(|A|+1)^{B-1} = 1.9 \times 10^{216}$, where $|A| = 6$ (the number of `BackwardFilter` algorithms), $B = 256$. Figure 6 illustrates a Pareto front of one convolutional layer of AlexNet.

2) Implementation Details: To perform WD optimization, μ-cuDNN must know the number of convolutional layers and corresponding layer parameters in advance, i.e., before running any kernel. In the current cuDNN API, however, the parameters are passed one layer at a time, and thus there is no way to obtain all the parameters collectively from deep learning frameworks.

To overcome this issue, we assume that the deep learning framework calls `cudnnGetConvolution*Algorithm` one time for each layer prior to the computation of the entire network (e.g., training, inference). This is the most straightforward use of the cuDNN interface, as memory (including workspace) is usually allocated before initiating computations. When the function is called, μ-cuDNN pushes the kernel parameters to an internal list, and returns a dummy result. Note that the returned results satisfy the semantics given by the cuDNN interface, so the framework will not raise errors nor allocate its own workspaces. When `cudnnConvolution*` is called for the first time, μ-cuDNN executes the optimization algorithm (namely, WD). We use the GNU Linear Programming Kit (GLPK) [15] as the ILP solver.

TABLE I: Convolution configurations of cuDNN 7.1.3.

Configuration	Data Type	Compute Type	Algorithms GEMM	Winograd	FFT
`TRUE_HALF`	`half`	`half`	✓	✓	
`PSEUDO_HALF`	`half`	`float`	✓	✓	✓
`FLOAT`	`float`	`float`	✓	✓	✓

D. Exploiting Higher Computation Precision

μ-cuDNN increases the number of available algorithms by exploiting higher computation precision than the framework specifies. In particular, cuDNN permits three types of configurations on convolutional layers that use either single- or half-precision, which determine a data type in memory and computation precision (Table I).

Table I shows that while `TRUE_HALF` can exploit either GEMM (GEneral Matrix-Matrix multiply)-based convolution or Winograd's algorithm, `PSEUDO_HALF` can select FFT-based convolution in addition to them. Thus, when `TRUE_HALF` is specified by the framework μ-cuDNN takes algorithms of `PSEUDO_HALF` into account, in order to exploit potential speedups by FFT-based convolution and well-optimized implementations. Since computing precision is increased, the function chosen by μ-cuDNN does not incur higher numerical error than `TRUE_HALF`. In addition, changing computation precision does not require explicit memory realignment, hence no additional overhead will be introduced except for a change in the workspace size.

E. High-level Optimization Frontend

In this section, we introduce μ-cuDNN's Python interface. Since the convolutional layers' parameters are passed through μ-cuDNN, it is capable of simplifying the analysis and optimization of the convolutional layers, regardless of the underlying deep learning framework. Indeed, almost all deep learning frameworks exploit cuDNN, and thus this interface provides widely-applicable utilities to end users.

μ-cuDNN first transparently stores benchmark results, coupled with layer parameters, in an SQLite database (1. and 2. of Figure 3). The database contains a table of performance metrics of the cuDNN's benchmarking functions for specific GPUs, a table of layer IDs, and corresponding layer parameters. Then, the Python interface loads the results from the database, without involving the framework itself (3.). This scheme decouples framework-specific codes from the analysis process itself. This file-based caching is also beneficial to eliminate repetition of benchmarking for the optimization scheme described in Section III-B.

```
1  import ucudnn
2  import framework as f
3
4  mb = 256
5  gpus = ["K80", "K80", "K20Xm"]
6  f.CNN(gpus=gpus, minibatch=[mb, mb, ...]).run_once()
7  f.CNN(gpus=gpus,
8        minibatch=ucudnn.best_batch_size(mb, gpus)).run()
```

Fig. 7: Sample code for heterogeneous cluster optimization.

One concern when storing network parameters is that the cuDNN interface does not provide unique identifiers for layers, such as its name. To solve this problem, μ-cuDNN requires an extra "layer ID" argument to be passed to the μ-cuDNN functions. This requirement can easily be fulfilled in most frameworks, as internal layer identifiers are already kept.

As a part of the Python interface, we provide a function to minimize training time by assigning different micro-batch sizes to heterogeneous GPUs (Figure 7). The motivation behind this function is that researchers tend to have access to clusters of heterogeneous accelerators, and highly utilizing such clusters may speed up training considerably.

In Figure 7, the function in line 8 provides an uneven batch size for each GPU so that the time to perform forward and backward passes of synchronous SGD becomes uniform among GPUs, increasing load balancing among GPUs. Since gradient communication is typically overlapped with computation, especially in large batch training [4], we omit the extra communication term in the objective, formulating the problem as follows:

$$\begin{aligned}
\text{min} \quad & \max_{g \in G} \left\{ \sum_{b \in \mathcal{B}} t_{g,b} x_{g,b} \right\} \\
\text{subject to} \quad & \sum_{b \in \mathcal{B}} x_{g,b} \leq 1 \quad (\forall g \in G) \\
& \sum_{g \in G} \sum_{b \in \mathcal{B}} b x_{g,b} = B \\
& x_{g,b} \in \{0, 1\} \quad (\forall g \in G, \forall b \in \mathcal{B}),
\end{aligned}$$

where G is a set of GPUs, $\mathcal{B}$ is a set of available batch sizes for each GPUs, B is the mini-batch size, and $t_{g,b}$ is time to perform forward and backward passes on GPU g with a batch size of b. μ-cuDNN uses a micro-batch size of b on GPU g if and only if $x_{g,b} = 1$. It is reasonable to restrict b to be a multiple of a power of two in order to reduce the number of configurations. If $\sum_{b \in \mathcal{B}} x_{g,b} = 0$ for a given GPU g, the ILP failed to find a fast configuration with g and it will not participate in training.

IV. Performance Evaluation

We evaluate the performance of μ-cuDNN for three different GPU architectures, NVIDIA Tesla K80 [17], P100-SXM2 [18] and V100-SXM2 [19] on the TSUBAME-KFC/DL, TSUBAME 3.0 supercomputers, and an NVIDIA DGX-1, respectively. We also use a spare Tesla K20Xm and GTX 750Ti on TSUBAME-KFC/DL in Section IV-C. The specifications of these systems are listed in Table II. Note that a K80 GPU contains two GK210 chips, and we show performance results of a single GK210 as "K80".

TABLE II: Evaluation environment specification.

	TSUBAME-KFC/DL	TSUBAME 3.0	NVIDIA DGX-1
CPU (Intel Xeon)	E5-2620 × 2	E5-2680 v4 × 2	E5-2698 v4 × 2
GPU (NVIDIA Tesla)	K80 × 4 - 8.73 SP TFlop/s - 24 GiB GDDR5 (480 GiB/s BW)	P100-SXM2 × 4 - 10.6 SP TFlop/s - 16 GiB HBM2 (732 GiB/s BW)	V100-SXM2 × 8 - 15.7 SP TFlop/s - 16 GiB HBM2 (900 GiB/s BW)
OS	CentOS 7.3.1611	SUSE Linux Enterprise Server 12 SP2	Ubuntu 16.04.3
CUDA	8.0.61 / 9.1.85	8.0.44	9.0.176
cuDNN	6.0 / 7.1.2	6.0	7.1.2
GLPK	4.63	4.63	N/A
Caffe	1.0	1.0	NVCaffe v0.16.5 [16]
TensorFlow	N/A	1.4.1	N/A

Table III summarizes the experimental configurations evaluated in this section. Throughout the evaluation, unless explicitly mentioned, we use single-precision floating point format and store tensors in the (N, H, C, W) storage order. We use three different deep learning frameworks for evaluations: Caffe [7], its NVIDIA branch (NVCaffe) [16], and TensorFlow [8]. All of them support recent versions of cuDNN (6 or 7). We use a built-in benchmarking command (Caffe's "time" command) or an official benchmarking script (from TensorFlow models repository [20]) to measure the execution time of forward and backward passes, and show the sum of forward and backward passes together. In the following sections, unless explicitly mentioned, each forward-backward pass is measured 50 times on both Caffe and TensorFlow.

For neural networks, we use AlexNet [1], ResNet [2], and DenseNet [21]. For evaluations on Caffe, we use the AlexNet model defined in Caffe, ResNet-18, and ResNet-50 from NVCaffe. We modify data prefetching size from 4 to 16 for AlexNet and ResNet-18 for TSUBAME 3.0. For evaluations on TensorFlow, we use the definitions in an official benchmarking repository [22].

As for workspace limit, unless explicitly mentioned, we use 8 MiB and 64 MiB for each layer, which are the default workspace size limits of Caffe and Caffe2 [13] respectively. In addition, we use 512 MiB of workspace per layer to investigate the case where sufficiently large workspace is provided. To shorten the benchmarking time, we use several GPUs on the

TABLE III: Evaluation settings. Neural architectures "DB", "A", "R", and "D" represent DeepBench, AlexNet, ResNet, and DenseNet respectively.

	Workspace policy	Workspace limit [MiB]	Neural Architecture	GPU(s)
Layer Micro-benchmark	WR	64	DB	K80, P100, V100
Caffe	WR, WD	8, 64, 512	A, R	K80, P100, V100
TensorFlow	WR	64	A, R, D	P100
Heterogeneous cluster optimization	WR	64	R	K80, K20Xm, 750Ti

TABLE IV: Number of convolutional layers in DeepBench.

Network type	Kernel size	Batch size	# of layers
DeepSpeech	10×5, 20×5	4, 8, 16, 32	8
OCR	3×3	16	4
Face Recognition	1×1, 3×3	8, 16	23
Vision	1×1, 3×3, 5×5, 7×7	8, 16	19
Speaker ID	3×3, 5×5	16	8
ResNet	1×1, 3×3	8, 16	32

same node with the parallel evaluation function of μ-cuDNN, mentioned in Section III.

A. Layer Microbenchmark

DeepBench [23] is a set of frequently used layer (e.g., convolution kernels) configurations in deep learning, proposed by Baidu. DeepBench defines the parameters of 94 convolutional layers, including those of ResNet (Table IV).

Figure 8 shows the speedup of each convolutional layer against cuDNN on three different GPUs. In this section, we use the latest version of cuDNN (7.1.2), and set the workspace limit to 64 MiB. Also, we multiply the batch size by 4 from the original size.

In Figure 8, not only does μ-cuDNN accelerate the frequently used 3×3 and 5×5 kernels for all the GPUs, it also achieves up to 4.54x speedup (1.60x on average) on a V100-SXM2 GPU, when computation is done with half-precision and Tensor Cores. In this case, μ-cuDNN adopts the `PSEUDO_HALF` configuration for 65 kernels (69% of the kernels), and `TRUE_HALF` for the others. This observation demonstrates that μ-cuDNN successfully avoids new implementations that are potentially inefficient. In addition, Tensor Core-enabled convolutions tend to consume a large amount of memory (up to 437.73 MiB, 64.6 MiB on average on V100-SXM2) since GEMM-based convolution is more efficient but require a large workspace to rearrange the elements of an input tensor. This workspace requirement is, however, naturally resolved by μ-cuDNN. More importantly, μ-cuDNN was able to accelerate 3×3 kernels in half-precision, which are usually adopted in recent CNNs, by 1.16x on P100-SXM2 and 1.73x on V100-SXM2, on average respectively.

B. Deep Learning Frameworks

1) Caffe (WR): Figure 9 shows timing breakdowns of Caffe on AlexNet with three different GPUs. We only highlight convolutional layers since the others (e.g., pooling) are out of the scope of this paper.

One important observation from Figure 9 is that the performance improvement of μ-cuDNN over cuDNN (which is equivalent to `undivided`) is significant when the moderate amount of workspace is set by users. For instance, if the workspace size per kernel is 64 MiB, μ-cuDNN with the `all` option achieves 1.81x speedup with respect to the entire iteration, and 2.10x with respect to convolutions alone, than `undivided` on K80. This is because μ-cuDNN successfully enables cuDNN to use faster algorithms, as in Figure 10. In addition, a similar speedup is achieved on P100-SXM2 (1.40x for the entire iteration, and 1.63x for convolutions alone), and on V100-SXM2 (1.45x for the entire iteration, and 1.60x for convolutions alone).

In the case where workspace size is limited to 8 MiB, μ-cuDNN cannot attain any performance improvement, because of WR's per-layer workspace allocation scheme, which are too small to utilize. Indeed, on P100-SXM2, only one kernel of `all` option seems to increase the utilization of the workspace over `undivided`.

On the other hand, when the workspace size limit is too large (512 MiB) on K80 and P100-SXM2 GPUs, performance difference between cuDNN and μ-cuDNN is negligible. This is because there is no benefit from dividing the mini-batch, as all algorithms fit into the workspace constraints. However, this workspace limit consumes a considerable amount of workspace memory: While the `undivided` option consumes 2.87 GiB in total, `all` with 64 MiB limit only consumes 0.70 GiB, although with 4% overhead caused by the choice of micro-batch algorithms.

From the viewpoint of the time to optimization, including kernel benchmarking and solving DP, `powerOfTwo` considerably outperforms `all`. In particular, with 64 MiB workspace on P100-SXM2, `all` takes 34.16 s, whereas `powerOfTwo` takes 3.82 s. This result and Figure 9 imply that `powerOfTwo`

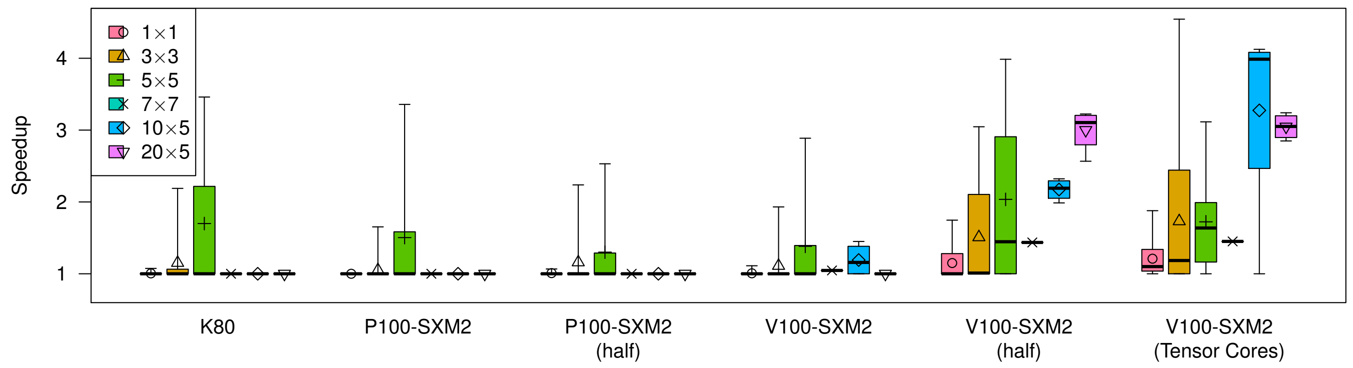


Fig. 8: Relative speedups of DeepBench's forward convolution against cuDNN. The whiskers and the points represent minimum/maximum speedups and their means respectively. We use 64 MiB workspace size.

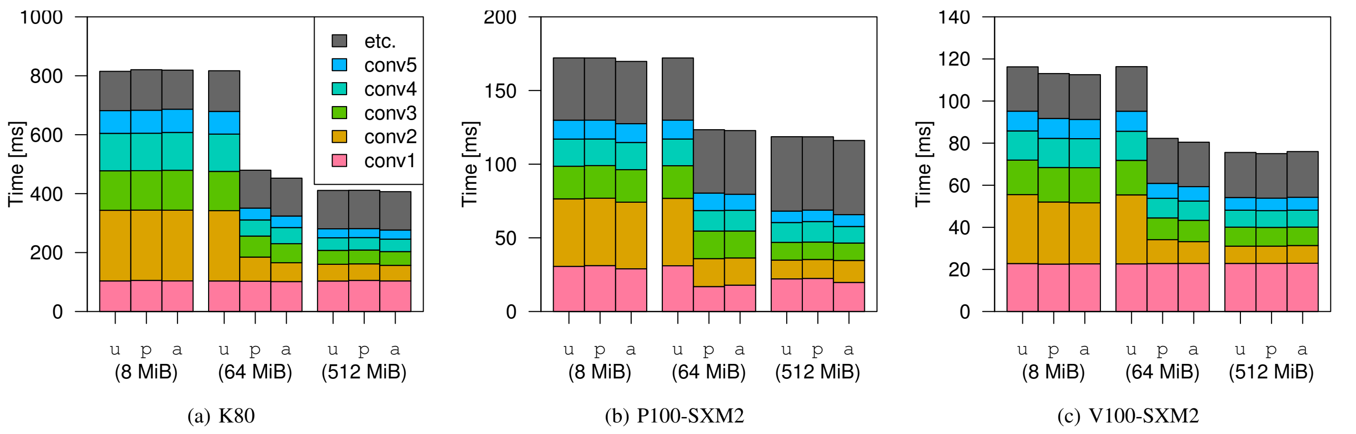


Fig. 9: Benchmark results of AlexNet on three different GPUs with different workspace sizes (8, 64, 512 MiB). The labels "u", "p" and "a" represent `undivided`, `powerOfTwo`, and `all`, respectively. We use a mini-batch size of 256.

is a reasonable choice to test the computation efficiency of new CNNs quickly. Note that we can reuse the same benchmarking results for different hyperparameters to save time, since the hyperparameters do not affect the computational performance of the convolution operations. Generally, the overhead of μ-cuDNN is negligible with respect to the entire training time, since it is only run once, whereas the forward and backward passes are repeated hundreds of thousands of times.

Figure 10 shows the execution time of forward convolution (`cudnnConvolutionForward`) of the "conv2" layer in AlexNet on P100-SXM2. With workspace size of 64 MiB, the GEMM-based algorithm is the one chosen by cuDNN, requiring only 4.3 KiB for workspace if the mini-batch is not divided. On the other hand, FFT-based convolution is more efficient, although it requires excessive amount of workspace (213 MiB) to store the images and filters in the frequency domain. μ-cuDNN with `powerOfTwo` option successfully enables the use of FFT within the workspace size sizes, using 48.9 MiB over micro-batches of size 32. The `all` option also enables μ-cuDNN to use Winograd convolution, an algorithm that is especially efficient for small convolution kernels, achieving 2.33x speedup over `undivided` in total.

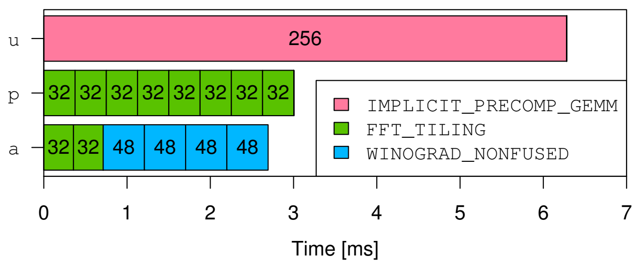


Fig. 10: Benchmark results of forward convolution of AlexNet's "conv2" layer on P100-SXM2. We use 64 MiB workspace size and a mini-batch size of 256. Numbers on each rectangle represent micro-batch sizes.

Note that in our work we assume that CNNs are robust against numerical errors introduced by changing a convolution algorithm, as supported by previous work [12].

2) Caffe (WD): Figure 11 shows the benchmark results of using the WD algorithm. The adjoined bars have the same workspace limit in total: For example, since AlexNet has five convolutional layers and each layer has three kernels (`Forward`, `BackwardData` and `BackwardFilter`), we place the result with 120 MiB WD workspace next to that of 8 MiB WR workspaces.

In Figure 11, we can see that the training time decreases as the workspace constraints increase in both WR and WD. At the same time, WD successfully manages the global memory requirements better, attaining higher performance with the same overall memory footprint (Figure 12). Specifically, when 120 MiB workspace in total is provided for AlexNet, the entire execution time with WD optimization and `all` option is 1.24x

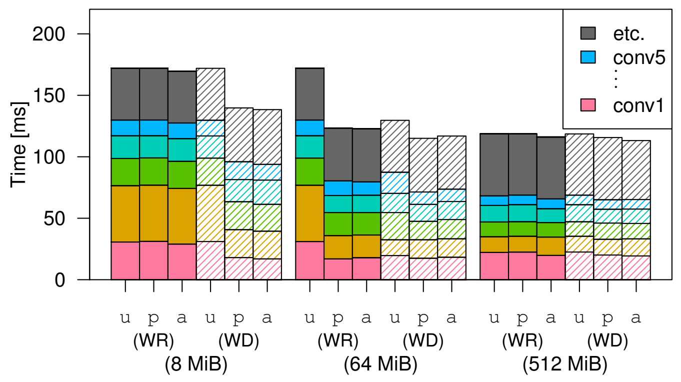


Fig. 11: Benchmark results of AlexNet on P100-SXM2 with different workspace sizes and policies, WR (solid) and WD (shaded). We use a mini-batch size of 256 for AlexNet and 32 for ResNet-50. Note that the adjoined bars have the same workspace limit in total.

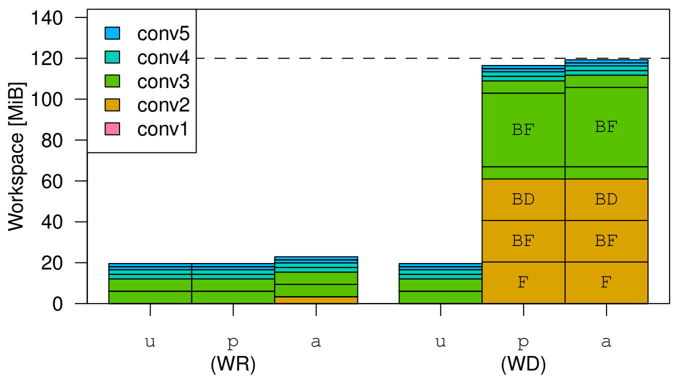


Fig. 12: Assigned workspace division of AlexNet on P100-SXM2. "BD", "BF" and "F" represent kernel types (`BackwardData`, `BackwardFilter` and `Forward` respectively). We use a mini-batch size of 256 for AlexNet. We set a workspace limit of 8 MiB for WR, and a total workspace limit of 120 MiB (shown as a dashed line) for WD.

faster than the WR with `undivided` option for the entire iteration (or 1.38x for convolution). WD also outperforms the baseline with 960 MiB workspace by 1.24x in total, which can use 8 times more memory for workspace.

Furthermore, even for ResNet-50, which has 10 times more convolutional layers than AlexNet, WD achieves 1.05x speedup for the entire iteration (or 1.14x for convolutions) with 2,544 MiB of total workspace, outperforming the original version (which consumes 5,088 MiB) in terms of memory footprint as well. In addition, the ILP for ResNet-50 is still small enough to be solved in practical time. When the workspace limit is set to 5,088 MiB, the number of variables is 562, and the GLPK solver takes 5.46 ms to solve it. Even in the most complicated case we tested, where ResNet-152 with a mini-batch size of 128 is used, it only takes 716 ms.

The main reason that WD outperforms WR is that in WR, if μ-cuDNN fails to find better algorithms and micro-batch sizes to fully utilize the assigned workspace, μ-cuDNN must abandon that workspace slot and cannot allocate it to other kernels. On the other hand, in WD, characteristics of different desirable workspace sizes of different kernels (Figure 6) are implicitly considered in the ILP-based optimization framework. Therefore, μ-cuDNN can assign larger proportional workspaces to time-consuming layers, if it is expected that the kernels will be considerably faster with a larger workspace.

In Figure 12, μ-cuDNN with the WD policy spares most of the workspace for "conv2" and "conv3" (93.7%), which are the most time-consuming layers in the baseline. In contrast, μ-cuDNN doesn't allocate workspace of over 3 MiB for "conv4" and "conv5", although μ-cuDNN lists some faster and desirable configurations than the baseline. For instance, the fastest configuration of "conv5" (forward), which uses FFT-based convolution with two micro-batches, is 1.29x faster than baseline, although this configuration uses 109 MiB of workspace. This observation implies that the WD does not unnecessarily allocate workspace for a specific layer but chooses the best global combination, as defined by the ILP.

TABLE V: TensorFlow benchmark results on P100-SXM2.

Policy	Time [ms] (Speedup)		
	AlexNet	ResNet-50	DenseNet
`undivided`	229.0	318.0	639.0
`powerOfTwo`	186.0 (1.23)	302.0 (1.05)	579.0 (1.10)
`all`	185.0 (1.24)	302.0 (1.05)	574.0 (1.11)

3) TensorFlow: Table V summarizes the speedups of the second version of AlexNet [24], ResNet-50, and DenseNet-40 on P100-SXM2. We use a mini-batch size of 256 for AlexNet and DenseNet, and 64 for ResNet-50.

We set the (input width, output width) to $(224, 1000)$ for AlexNet and ResNet-50, or $(32, 10)$ for DenseNet-40, which are used for training the ILSVRC2012 dataset [25] or the CIFAR-10 dataset [26], respectively. We also set k of DenseNet-40, the number of feature maps of each convolutional layer, to 40 to obtain better computational efficiency.

Since TensorFlow 1.4.1 does not provide any workspace limits to μ-cuDNN via cuDNN's benchmarking functions before actual convolutions, we manually provide a workspace limit of 64 MiB to μ-cuDNN. μ-cuDNN achieves 1.24x speedup for AlexNet, 1.05x for ResNet-50, and 1.11x for DenseNet. These results prove that μ-cuDNN has good performance portability between different deep learning frameworks that depend on cuDNN. Note that we do not expect the same speedups between Caffe (Figure 9) and TensorFlow (Table V). This is because TensorFlow uses different parameters that μ-cuDNN cannot control (such as padding widths), and a considerable part of the time is spent on non-convolutional computation, which are implemented differently in each framework.

C. Heterogeneous Cluster Optimization

In this section, we demonstrate the μ-cuDNN Python interface by combining three different GPUs from the TSUBAME-KFC/DL supercomputer: Tesla K20Xm, Tesla K80, and GTX 750Ti (Table VI). The K20Xm and K80 are Kepler generation GPUs, whereas the 750Ti is a Maxwell generation GPU, not intended for high performance computing.

We first run Caffe's "time" command on each node to collect performance metrics to a database. Since we employ a file-based database, it is easily collected on a Networked File System (NFS). Then, we use μ-cuDNN's optimization function in Python, which is explained in Section III-E.

TABLE VI: GPU specification.

	FP32 TFlop/s	Memory size [GiB]
Tesla K20Xm	3.95	6
Tesla K80 (GK210 $\times$ 2)	8.73	24
GTX 750Ti	1.31	2

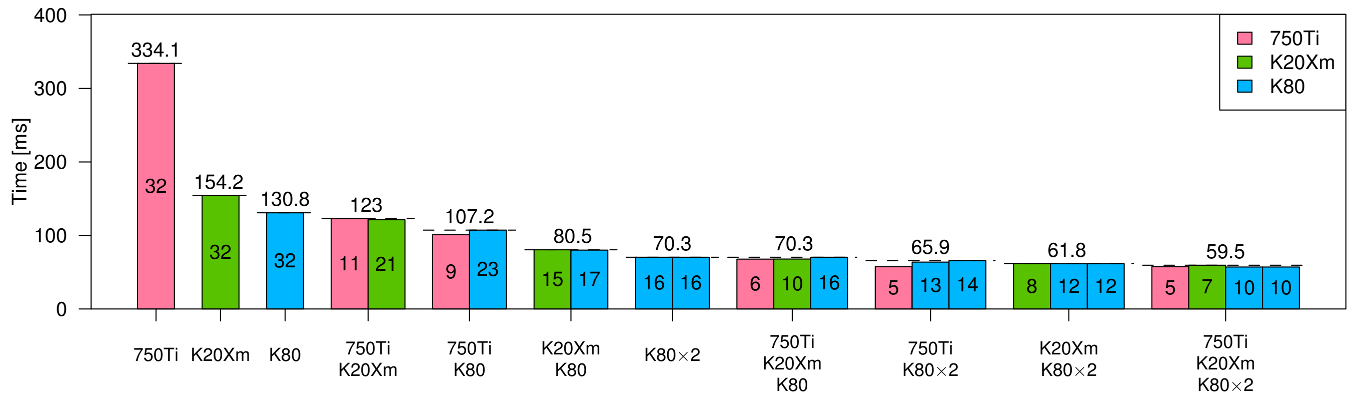


Fig. 13: Estimated time of forward-backward passes of ResNet-18 on heterogeneous GPUs. Numbers on each bar represent batch sizes. The objective is to minimize a maximum of GPUs' time, shown as dashed lines.

Figure 13 shows the estimated time of forward-backward passes of ResNet-18 on heterogeneous GPUs. By combining two GK210 chips of a K80 GPU and a whole K20Xm GPU, forward-backward passes become 2.12x faster than that of a single GK210. When a K20Xm and a K80 are combined, μ-cuDNN assigns uneven batch sizes, 8 and $\{12, 12\}$ respectively. In addition, if a user sets even batch sizes for GPUs, 750Ti and K20Xm for example, the 750Ti will become a bottleneck (168.43 ms vs 83.03 ms) and thus it will incur a slowdown of 1.37x than μ-cuDNN. Furthermore, a combination of all the GPUs yields 2.20x speedup against the baseline. Note that the time to perform MPI all-reduce over MVAPICH2 2.3a with a message size of 1 MiB on 3 nodes takes 2.63 ms, which can be easily hidden by the computation. Therefore, this example illustrates the potential speedups by heterogeneous GPUs for training of a single CNN.

V. Related Work

Li et. al [27] propose a heuristic to tune each tensor memory layout to utilize either GEMM-based or FFT-based convolution efficiently. The proposed heuristic is, however, based on the authors' performance observation using conventional convolutional layers and specific GPU architecture, and thus there is no guarantee that the algorithm always provides the best memory alignment for any neural network and GPU architecture. On the other hand, since μ-cuDNN uses the techniques of dynamic programming and integer linear programming, it is guaranteed that μ-cuDNN provides the best performance that the library can produce.

Rhu et al. [28] propose a memory management technique that offloads neuron activations, parameters, and errors from the GPU memory to the CPU memory during forward-/backward-propagation, so that larger models can be trained with the same memory constraint. However, as Figure 1 shows, even in such memory-efficient implementations or similar memory management techniques [29] μ-cuDNN is expected to save the peak memory usage of each layer.

Zlateski et al. [30] propose ZNNi, an FFT-based convolution algorithm, and mention a technique similar to micro-batching to reduce the temporal memory usage by FFT. We generalize the schema so that micro-batching can be applied to any convolution algorithm, obtain the best computational performance for the given layer configurations, as well as maintain high portability between different deep learning frameworks.

VI. Conclusion

In this paper, we proposed μ-cuDNN, a wrapper library for cuDNN, which divides the mini-batch to utilize high-performance convolution algorithms with limited amount of memory for workspaces. We have shown that μ-cuDNN can easily be integrated into existing deep learning frameworks, and works well with several recent CNNs, which are composed of many convolutional layers. In addition, μ-cuDNN provides a framework-independent interface that is useful for empirical performance optimization within a single machine and load balancing across a heterogeneous cluster.

The performance of μ-cuDNN demonstrated in our work suggests that other layer types can be optimized as well, if they can be decomposed and computed by different algorithms. This is because μ-cuDNN does not use any special properties of the convolution operator, apart from gradient accumulation.

In addition, the result of WD optimization (Figure 12) provides us with the insight that allocating the same workspace memory for each convolutional layer is not necessarily effective, and dynamic, adaptive assignment performs better. This observation should be beneficial for advanced deep learning frameworks that dynamically manage GPU memory to store tensors such as neuron data, weights and their gradients, for further memory optimization.

Acknowledgment

This research was supported by the ETH Postdoctoral Fellowship (for T. B. N.), Student Summer Research Fellowship (for Y. O.), JST CREST Grant Number JPMJCR1303, JP-MJCR1687, and JSPS KAKENHI Grant Number JP18J22858, Japan. Part of this work is conducted as research activities of AIST - TokyoTech Real World Big-Data Computation Open Innovation Laboratory (RWBC-OIL).

References

[1] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks," *Advances in Neural Information Processing Systems 25 (NIPS 2012)*, Dec 2012.

[2] K. He, X. Zhang, S. Ren, and J. Sun, "Deep Residual Learning for Image Recognition," in *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR 2016)*, Jun 2016, pp. 770–778.

[3] P. Goyal, P. Dollár, R. B. Girshick, P. Noordhuis, L. Wesolowski, A. Kyrola, A. Tulloch, Y. Jia, K. He, and P. Dollar, "Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour," *CoRR*, vol. abs/1706.0, Jun 2017, http://arxiv.org/abs/1706.02677.

[4] T. Akiba, S. Suzuki, and K. Fukuda, "Extremely Large Minibatch SGD: Training ResNet-50 on ImageNet in 15 Minutes," *CoRR*, vol. abs/1711.04325, Nov 2017, https://arxiv.org/abs/1711.04325.

[5] S. L. Smith, P.-J. Kindermans, and Q. V. Le, "Don't Decay the Learning Rate, Increase the Batch Size," *CoRR*, vol. abs/1711.00489, Nov 2017, https://arxiv.org/abs/1711.00489.

[6] NVIDIA. NVIDIA cuDNN. https://developer.nvidia.com/cudnn. Accessed on 2017-11-23.

[7] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional Architecture for Fast Feature Embedding," *arXiv preprint arXiv:1408.5093*, 2014.

[8] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," https://www.tensorflow.org/, Nov 2015.

[9] Theano Development Team, "Theano: A Python framework for fast computation of mathematical expressions," *arXiv e-prints*, vol. abs/1605.02688, May 2016, http://arxiv.org/abs/1605.02688.

[10] S. Tokui, K. Oono, S. Hido, and J. Clayton, "Chainer: a Next-Generation Open Source Framework for Deep Learning," in *Proceedings of Workshop on Machine Learning Systems (LearningSys) in The Twenty-ninth Annual Conference on Neural Information Processing Systems (NIPS 2015)*, Dec 2015.

[11] M. Mathieu, M. Henaff, and Y. Lecun, "Fast training of convolutional networks through FFTs," in *International Conference on Learning Representations (ICLR 2014)*, Apr 2014.

[12] A. Lavin and S. Gray, "Fast Algorithms for Convolutional Neural Networks," in *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR 2016)*, Jun 2016, pp. 4013–4021.

[13] Facebook. Caffe2. https://caffe2.ai/. Accessed on 2017-11-23.

[14] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR 2015)*, vol. 07-12-June, Jun 2015.

[15] A. Makhorin. GLPK (GNU Linear Programming Kit). https://www.gnu.org/software/glpk/. Accessed on 2017-11-23.

[16] NVIDIA. NVIDIA Caffe. https://github.com/NVIDIA/caffe. Accessed on 2017-11-23.

[17] ——. Tesla K80 HPC and Machine Learning Accelerator. http://www.nvidia.com/object/tesla-k80.html. NVIDIA. Accessed on 2017-11-23.

[18] ——. Tesla P100 Most Advanced Data Center Accelerator. http://www.nvidia.com/object/tesla-p100.html. Accessed on 2017-11-23.

[19] ——. NVIDIA Tesla V100. https://www.nvidia.com/en-us/data-center/tesla-v100/. Accessed on 2018-3-1.

[20] The TensorFlow Authors. tensorflow/models. https://github.com/tensorflow/models. Accessed on 2018-3-1.

[21] G. Huang, Z. Liu, L. van der Maaten, and K. Q. Weinberger, "Densely connected convolutional networks," in *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR 2017)*, Jul 2017.

[22] The TensorFlow Authors. tensorflow/benchmarks. https://github.com/tensorflow/benchmarks. Accessed on 2018-3-1.

[23] Baidu Research. DeepBench. https://github.com/baidu-research/DeepBench. Accessed on 2018-5-3.

[24] A. Krizhevsky, "One weird trick for parallelizing convolutional neural networks," *arXiv preprint*, vol. abs/1404.5, Apr 2014, http://arxiv.org/abs/1404.5997.

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

[26] A. Krizhevsky, "Learning Multiple Layers of Features from Tiny Images," https://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf, Tech. Rep., Apr 2009.

[27] C. Li, Y. Yang, M. Feng, S. Chakradhar, and H. Zhou, "Optimizing Memory Efficiency for Deep Convolutional Neural Networks on GPUs," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '16)*. Piscataway, NJ, USA: IEEE Press, Nov 2016, pp. 54:1–54:12.

[28] 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 *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2016)*, Oct 2016.

[29] K. Shirahata, Y. Tomita, and A. Ike, "Memory reduction method for deep neural network training," in *2016 IEEE 26th International Workshop on Machine Learning for Signal Processing (MLSP 2016)*, Sep 2016.

[30] A. Zlateski, K. Lee, and H. S. Seung, "ZNNi: Maximizing the Inference Throughput of 3D Convolutional Networks on CPUs and GPUs," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '16)*, Nov 2016, pp. 854–865.