Vol.2017-HPC-162 No.22
2017/12/19

Less is More: Accelerating Deep Neural Networks with Micro-Batching

(Unrefereed Workshop Manuscript)

Yosuke Oyama[1,a)] Tal Ben-Nun[2,b)] Torsten Hoefler[2,c)]
Satoshi Matsuoka[1,d)]

Abstract: NVIDIA 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 transparent wrapper library for cuDNN, which divides layer computation into several micro-batches. Based on Dynamic Programming and Integer Linear Programming, μ-cuDNN enables faster algorithms by decreasing the workspace requirements. We demonstrate the effectiveness of μ-cuDNN over the Caffe framework, achieving speedups of 1.63x for AlexNet and 1.21x for ResNet-18. These results indicate that using micro-batches can seamlessly increase the performance of deep learning, while maintaining the same memory footprint.

1. Introduction

Prevalent Deep Neural Networks (DNNs) are becoming increasingly deeper and are trained with large batch sizes. Specifically, state-of-the-art DNNs contain hundreds of layers [1], [2], and utilize batch sizes in the order of thousands [3], [4], [5].

Large batches are also favored by distributed data-parallel deep learning frameworks, because the communication of parameter gradients can be overlapped with their computation. Consequently, the batch size per accelerator (e.g., 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 accelerator memory tends to be used at full capacity in most real-world cases.

This "limited memory scenario" still leaves room to optimize with cuDNN [6], a deep learning kernel library for NVIDIA GPUs. cuDNN provides a variety of computational primitives for deep neural networks, and is widely used in deep learning frameworks, such as Caffe [7] and others [8], [9], [10]. cuDNN provides up to eight different algorithms to perform convolutions, each of which requires different temporary storage (workspace) schemes.

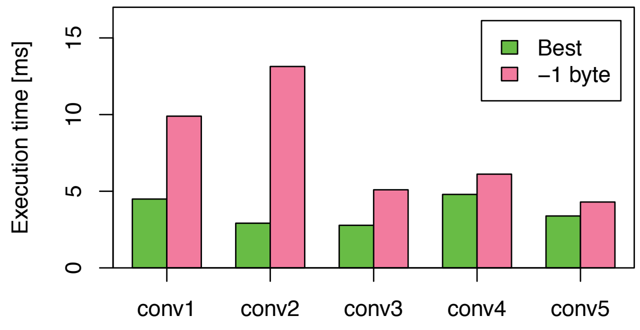
Additionally, cuDNN provides optimization functions to determine the best algorithm for a given maximum workspace size, either with respect to computation time or memory usage. Therefore, if the workspace size requested by a fast algorithm is one byte larger than provided, cuDNN will resort to a slower algorithm that requires less workspace. In fact, the performance impact can be 4.51 times in the 2nd convolutional layer of AlexNet, as shown in **Fig. 1**. This is because there are considerable performance gaps among different convolution algorithms.

In most frameworks, including Caffe, workspace is typically allocated for each layer separately, so the total workspace size can easily exceed the GPU memory size. This problem is amplified as the number of layers increase, since users are required to reduce the workspace limit to prevent running out of memory.

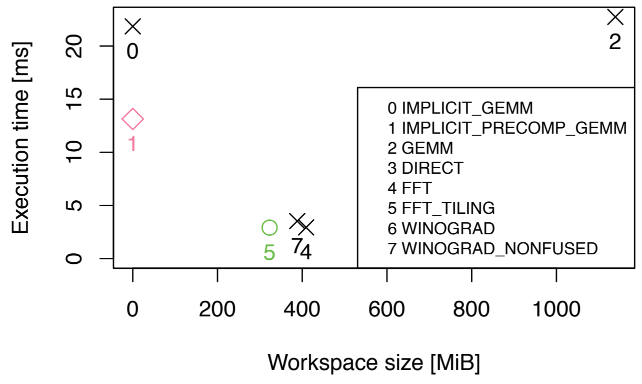
In this paper, we propose μ-cuDNN, a transparent wrapper for cuDNN that attempts to mitigate the aforementioned inefficiency. The contributions of this paper are as follows:

- 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 two different workspace policies, which enable optimization of multiple convolutional layers with inter-dependencies.
- We evaluate μ-cuDNN over the Caffe deep learning framework, showing that it can mitigate the inefficiency of cuDNN even with state-of-the-art Convolutional

[1] Tokyo Institute of Technology, Tokyo, Japan
[2] ETH Zurich, Zurich, Switzerland
[a)] oyama.y.aa@m.titech.ac.jp
[b)] talbn@inf.ethz.ch
[c)] htor@inf.ethz.ch
[d)] matsu@acm.org



(a) Execution time of all layers.



(b) Execution time vs. execution time of conv2. ○ and ◇ represent the "Best" and the "-1 byte" respectively.

Fig. 1: Execution time of cuDNN 7.0.1 forward convolution of single-column AlexNet [11] with different workspace sizes. The "Best" case always chooses the fastest algorithm regardless of workspace size, while in the "-1 byte" case the maximum workspace size is limited to 1 byte less than the best algorithm.

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:               $Y[n,k,h,w]$ += $W[k,c,v,u] \times X[n,c,h+v,w+u]$;
```

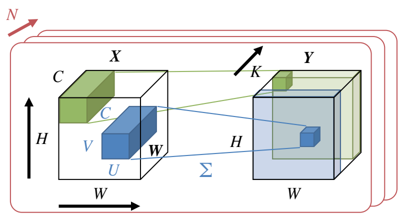


Fig. 2: Two-dimensional convolution. Each element of Y is set to be a sum of element-wise products between partial $C \times V \times U$ area of X and one filter from W.

Neural Networks (CNNs), such as AlexNet and ResNet.

2. The Anatomy of Convolutional Neural Networks

Convolution operations in Convolutional Neural Networks (CNNs) apply multiple filters to a batch of channels of two-dimensional data (Algorithm 1, **Fig. 2**). 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 X is multiplied and accumulated to one element of the output tensor Y. The remaining loops iterate over all elements of Y. The key observation is that in order to solve the problem described in Section 1, there is no dependency 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 splitting to the mini-batch loop, so that we can reduce the resident workspace size.

In cuDNN, there are three operations related to the two-dimensional convolution; `Forward` for forward computation (Fig. 2), `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 dW. 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.

3. 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], [12]. 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

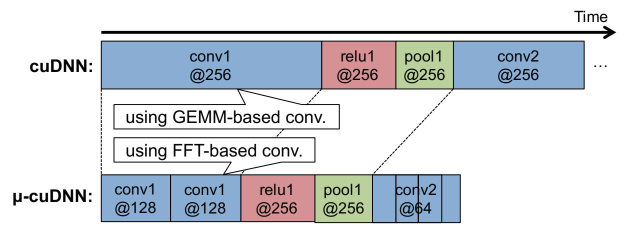


Fig. 3: The conceptual timeline of μ-cuDNN. "@256" means that each computation is executed with batch-size of 256. μ-cuDNN splits one convolution operation into one or more disjoint subsets of the mini-batch.

algorithms (**Fig. 3**).

3.1 μ-cuDNN Methodology

μ-cuDNN library employs one of two workspace utilization policies to optimize micro-batches for convolution kernels (**Fig. 4**):

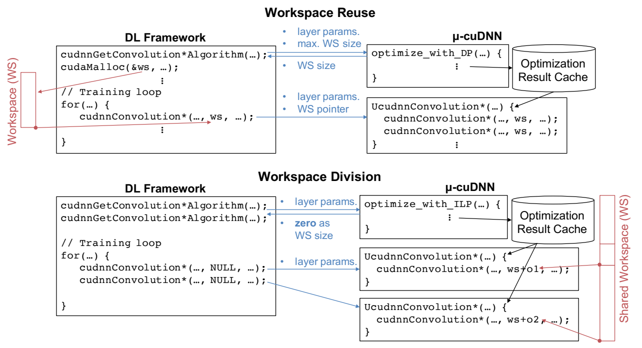
- **Workspace Reuse (WR)**: WR uses one workspace for each layer separately, sharing the space between the internal micro-batches. In this scheme, each layer is assumed to use the workspace exclusively, hence the size of the single workspace is exactly the maximum workspace size.
- **Workspace Division (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 [13], to run concurrently with larger workspaces. In WD, the actual workspace is managed by μ-cuDNN rather than the deep learning framework, 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), mini-batch size, and maximum workspace size. The output of μ-cuDNN is a division of the mini-batch, and "micro-configurations"; a pair of micro-batch size and a convolution algorithm 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 mini-batch size of 256 is equally divided into three micro-batches and each of them uses algorithm X, the configuration is represented as $\{(64, X), (64, X), (64, X), (64, X)\}$. Also we define concatenation of two lists as $+$, such as $\{a, b\} + \{c, d\} = \{a, b, c, d\}$ and $\{a\} + \phi = \{a\}$.

3.2 WR Algorithm

The goal of the WR policy is to optimize the total execution time with mini-batch size of B using Dynamic Programming (DP), given by:

$$T(B) = \min \left\{ \begin{array}{l} T_\mu(B), \\ \min_{b=1,2,\ldots,B-1} T(b) + T(B-b) \end{array} \right\},$$



```
cudnnGetConvolution*Algorithm(…);
cudaMalloc(&ws, …);
                    ⋮
// Training loop
for(…) {
   cudnnConvolution*(…, ws, …);
                    ⋮
}
```

```
optimize_with_DP(…) {
         ⋮
}
```

```
UcudnnConvolution*(…) {
  cudnnConvolution*(…, ws, …);
  cudnnConvolution*(…, ws, …);
}        ⋮
```

```
cudnnGetConvolution*Algorithm(…);
cudnnGetConvolution*Algorithm(…);

// Training loop
for(…) {
   cudnnConvolution*(…, NULL, …);
   cudnnConvolution*(…, NULL, …);

}
```

```
optimize_with_ILP(…) {
         ⋮
}
```

```
UcudnnConvolution*(…) {
  cudnnConvolution*(…, ws+o1, …);
}
```

```
UcudnnConvolution*(…) {
  cudnnConvolution*(…, ws+o2, …);
}
```

Fig. 4: Overview of WR and WD. μ-cuDNN optimizes micro-batch sizes and internally calls cuDNN functions, via the cuDNN interfaces.

where $T_\mu(b)$ is the fastest execution time of one convolution kernel with micro-batch size of b, within the workspace constraint. If the first row of the definition of $T(B)$ is smaller than the second row, 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 (**Fig. 5**).

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 is mainly two-fold, where the mini-batch size is B, and user-given maximum workspace size is M:

(1) For $b = 1, 2, \cdots, B$, 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) and its execution time as $T_\mu(b)$.

(2) For $b = 1, 2, \cdots, B$, computes $T(b)$, the fastest execution time for micro-batch size of b, and $c(b)$, the corresponding configuration, as follows (where $T(0) = 0, c(0) = \phi$). $T(b)$ 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 $c(B)$.

3.3 WD Algorithm

In the WD scheme, configurations for multiple convolution kernels are optimized, 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

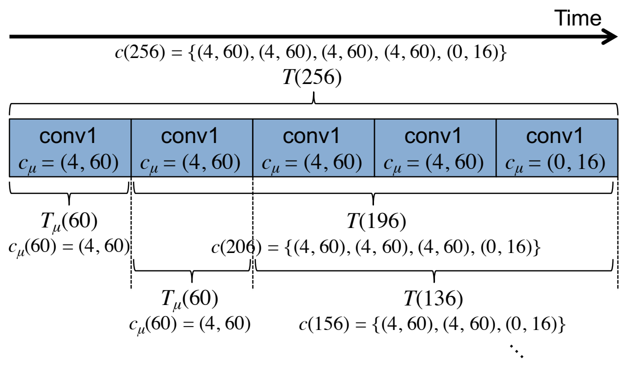


Fig. 5: DP-based optimization of WR. Here it is assumed that algorithm 4 with micro-batch size of 60 ($c_\mu(60) = (4, 60)$) achieves better computation efficiency, hence it is repeatedly used.

(**Fig. 6**). Given the set of kernels K and sets of available configurations C_k of kernel k, WD is solved in Equation 1.

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

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

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

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

Where $M_k(c)$ and $T_k(c)$ are the workspace size and execution time of kernel k with configuration c, respectively. Equation 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 Equation 3.

3.3.1 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 $|A|^B$ (where A is set of algorithms) configurations for each kernel, which makes the problem impractically large.

Here we introduce a pruning algorithm to remove undesirable configurations from all possible configurations, without returning any sub-optimal solutions. The resulting desirable configuration set C_k is then input to the ILP (Equation 1-4) to solve the entire problem.

First, we modify the DP algorithm 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

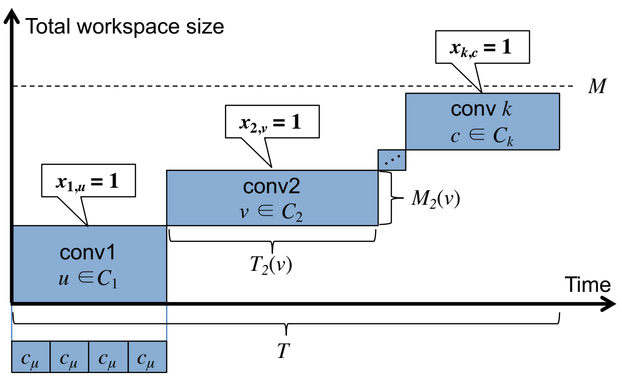


Fig. 6: ILP-based optimization of WD. The problem is stacking "time × memory" rectangles of configurations diagonally, and obtaining the minimum total width T, provided that the total height is lower than M. Each configuration $u, v, \ldots, c$ is composed of one or more micro-configurations such as c_μ.

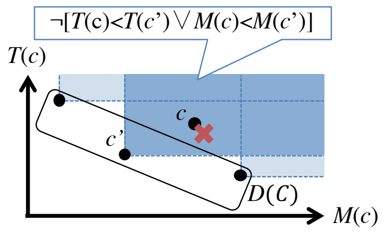


Fig. 7: The concept of desirable set. Here c cannot be in $D(C)$ because a c' exists for which the condition $T(c) < T(c') \lor M(c) < M(c')$ is not satisfied.

b.

Second, we define the "desirable configuration set" function $D(C) \subset C$ as follows (**Fig. 7**):

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

where $T(c)$ and $M(c)$ is execution time and required workspace size of a configuration set c.

This definition implies that any $c \in D(C)$ is the fastest configuration among any of the elements of $D(C)$ which use 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 faster than c and requires less workspace, hence there is no reason to choose c, namely "undesirable". **Fig. 8** shows desirable configurations of one layer of AlexNet.

We define the implementation of D as follows. When a new element c is added to C, the function (i) searches faster elements from desirable configuration candidates, (ii) checks that $M(c)$ is less than that of the slowest element of the elements, (iii) adds c to candidates, and (iv) removes candidates less desirable than c. This algorithm drastically reduces the number of variables of Equation 1, and enables to solve the ILP for state-of-the-art deep CNNs in practical time.

3.4 μ-cuDNN Implementation

To enable μ-cuDNN, the only modification that needs

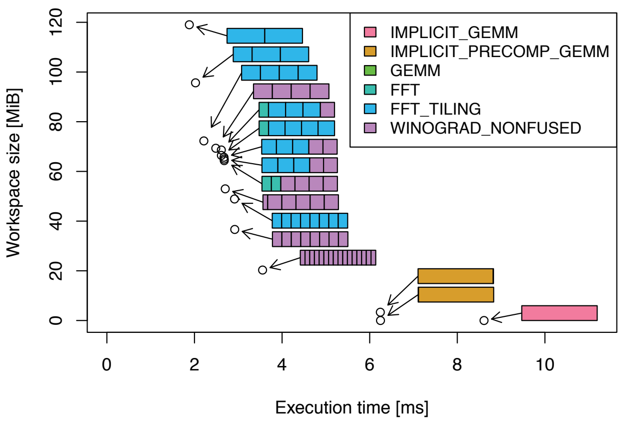


Fig. 8: Desirable configurations of AlexNet conv2 (Forward) on NVIDIA Tesla P100-SMX2 with a maximum workspace size of 120 MiB, and a mini-batch size of 256. Colored bars corresponding to data points represent the division of the mini-batch and the chosen micro-batch algorithms. For example, the top-left point divides the mini-batch into two micro-batches of 128 and utilizes the `FFT_TILING` algorithm.

to be performed to the code is replacing the cuDNN handle type `cudnnHandle_t` with `Ucudnn`···. The μ-cuDNN handle object is an opaque type which wraps the original type, such that users can call any cuDNN functions with the handle object. When the convolution operation or benchmarking function is called with the μ-cuDNN handle object, the μ-cuDNN library internally computes the optimal configurations and returns a virtual algorithm ID and zero required workspace size. This mechanism enables users to call μ-cuDNN with minimal modification to the original code. Indeed, the number of lines to be modified to introduce μ-cuDNN to Caffe (v1.0) is approximately three.

The implementation of μ-cuDNN is based on overloading a subset of cuDNN functions, and defining a cast operator from `Ucudnn`··· to `cudnnHandle_t` which the μ-cuDNN handle object internally holds. Using this technique, μ-cuDNN delegates most of the functions to cuDNN, but overrides those functions related to the convolutional layers.

The optimization algorithm in μ-cuDNN is fundamentally based on the methodology described in Section 3.1. In practice, μ-cuDNN provides a "batch size policy", which determines what micro-batch sizes are benchmarked at the step 1 of the WR algorithm, as follows:

- `all` uses all batch sizes $b \in \{1, 2, 3, \cdots, B\}$. Although this always finds the optimal solution, it takes $O(B)$ 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 $O(\log B)$ 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 via an environment variable, in which 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 homogeneous GPUs.

In addition, the optimization results are statically cached, which skips unnecessary recomputations. This is especially beneficial for networks that replicate convolutional layers of the same size, such as ResNet [2].

3.5 Implementation of WD Optimization

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. Due to the specific implementation of Caffe, we add a μ-cuDNN library call after network initialization, which ignores subsequent `cudnnGetConvolution*Algorithm` calls.

When `cudnnGetConvolution*Algorithm` 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 and will not 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) [14] as the ILP solver.

Table 1: Evaluation Environment Specification.

	TSUBAME-KFC/DL	TSUBAME 3
CPU	Intel Xeon E5-2620×2	Intel Xeon E5-2680 v4×2
GPU	NVIDIA Tesla K80×4 - 8.73 SP TFlop/s - 24 GiB GDDR5 memory (480 GiB/s bandwidth)	NVIDIA Tesla P100-SMX2×4 - 10.6 SP TFlop/s - 16 GiB HBM2 memory (732 GiB/s bandwidth)
OS	CentOS 7.3.1611	SUSE Linux Enterprise Server 12 SP2
CUDA	8.0.61	8.0.44
cuDNN	6.0	6.0
GLPK	4.63	4.63

4. Performance Evaluation

We evaluate the performance of μ-cuDNN for two different GPU architectures, NVIDIA Tesla K80 [15] and NVIDIA Tesla P100-SMX2 [16], on the TSUBAME-KFC/DL and TSUBAME 3 supercomputers, respectively. The specifications of these supercomputers are listed in **Table 1**.

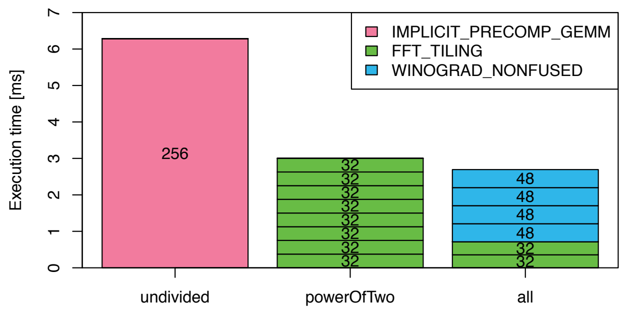


Fig. 9: Benchmark results of `cudnnConvolutionForward` of AlexNet's "conv2" layer on NVIDIA Tesla P100-SMX2. We use 64 MiB workspace size and a mini-batch size of 256. Numbers on each rectangle represent micro-batch sizes.

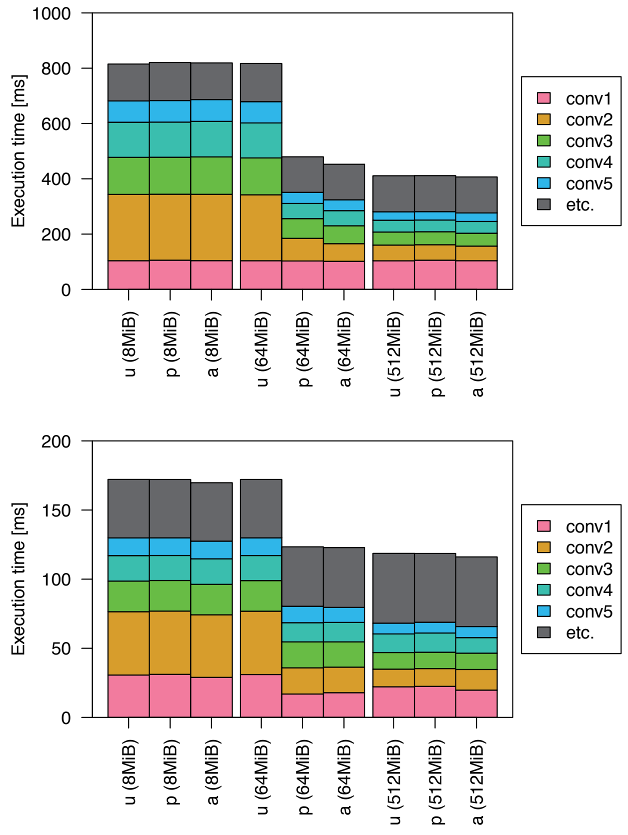


Fig. 10: Benchmark results of AlexNet on NVIDIA Tesla K80 (top) and P100-SMX2 (bottom) with different workspace sizes (8, 64, 512MiB). The labels "u", "p" and "a" represent `undivided`, `powerOfTwo`, and `all`, respectively. We use a mini-batch size of 256.

For overall evaluation, we use the Caffe [7] deep learning framework (version 1.0). We use single-precision floating point format and store tensors in $NCHW$ format.

For neural networks, we use the AlexNet model defined in Caffe, ResNet-18, and ResNet-50 from the NVIDIA branch of Caffe [17]. We also modify data prefetching size from 4 to 16 for AlexNet and ResNet-18 for TSUBAME 3.

For workspace limit for the evaluation, we use 8 MiB and 64 MiB for each layer, which are the default workspace size limits of Caffe and Caffe2 [12] respectively. In addition, we use 512 MiB to investigate the case where sufficiently large workspace is provided. To shorten the benchmarking time, we use all GPUs in the same node with the parallel evaluation function of μ-cuDNN, mentioned in Section 3.4.

4.1 Convolution Kernel Optimization Using WR

Fig. 9 shows the execution time of forward convolution (`cudnnConvolutionForward`) of AlexNet "conv2" layer on NVIDIA Tesla P100-SMX2. With workspace size of 64 MiB, the GEMM (GEneral Matrix-Matrix multiply)-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 [18] is more efficient although it requires excessive amount of workspace 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 constraints, using 48.9 MiB over micro-batches of size 32. Without micro-batches, FFT-based convolution requires 213 MiB for workspace with batch size of 256, which makes it infeasible to use.

The `all` option also enables μ-cuDNN to use Winograd convolution [19], an algorithm that is especially efficient for small convolution kernels, achieving 2.33 times speedup over `undivided` in total.

4.2 CNN Optimization Using WR

Fig. 10 shows two timing breakdowns of Caffe (obtained using Caffe's "time" command) on AlexNet with two different GPUs. The "time" command measures the execution time of forward and backward passes of each layer multiple times and outputs the average time, excluding the first iteration. In this section, unless explicitly mentioned, each layer is measured 50 times and each segment of bar plots shows the sum of forward and backward passes of the corresponding layer. Additionally, we only highlight convolutional layers since the others (e.g., pooling) are out of the scope of this paper.

One important observation from Fig. 10 is that the performance improvement of μ-cuDNN over cuDNN (which is equivalent to `undivided`) is significant when proper amount of workspace is provided. For instance, if the workspace size per kernel is 64 MiB, μ-cuDNN with the `all` option achieves 1.81 times speedup with respect to the entire iteration, and 2.10 times with respect to convolutions alone, than `undivided` on the K80 GPU. This is because μ-cuDNN successfully enables cuDNN to use faster algorithms, as in the example from Section 4.1. In addition, a similar speedup is achieved on P100-SMX2 GPU (1.40 times for the entire iteration, and 1.63 times for convolutions alone).

In the case where workspace size is limited to 8 MiB, μ-cuDNN cannot attain any performance improvement, because even if the mini-batch is finely divided the specified workspace is too small to utilize. Indeed, on P100, only one kernel of `all` option seems to increase the utilization of the workspace over `undivided`.

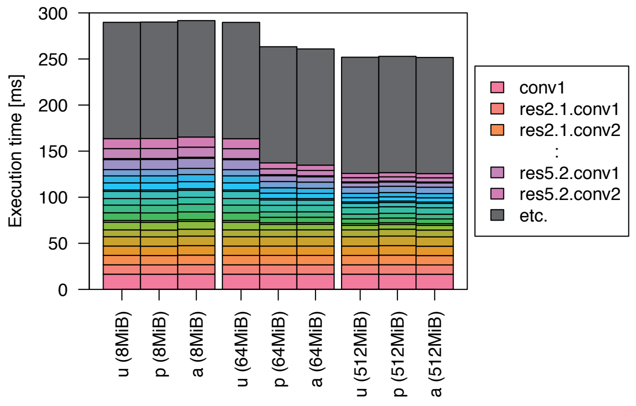


Fig. 11: Benchmark results of ResNet-18 on NVIDIA Tesla P100-SMX2 with different workspace sizes. We use a mini-batch size of 128. Each convolutional layer name is based on the model definition.

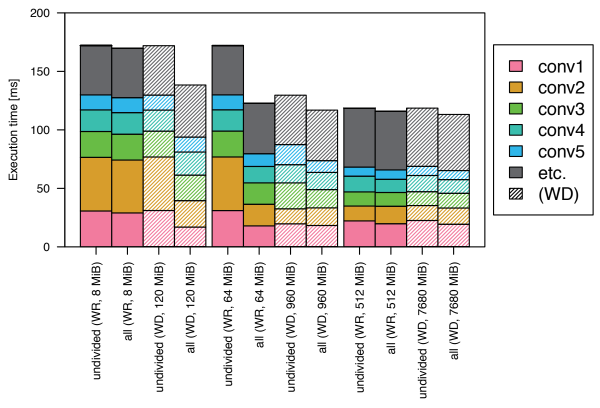
On the other hand, when the workspace size limit is too large (512MiB), μ-cuDNN cannot outperform cuDNN as well. This is because there is no benefit from dividing the mini-batch, as all algorithms fit into the workspace constraints. However, this workspace limit consumes a considerable amount of workspace memory: While `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 GPU, `all` takes 34.16 s, whereas `powerOfTwo` takes 3.82 s. This result and Fig. 10 imply that `powerOfTwo` is a reasonable choice to test the computation efficiency of new CNNs quickly. Generally, the overhead of μ-cuDNN is negligible with respect to the entire training time, in which the forward and backward passes are repeated hundreds of thousands of times.

Fig. 11 presents the performance breakdown of ResNet-18 on Caffe with two GPUs. Even with ResNet-18, in which considerable amount of time is spent on non-convolutional layers (shown in gray in Fig. 11), μ-cuDNN achieves 1.11 times speedup (and 1.21 times for convolutions) using `all` with workspace limit of 64 MiB. Note that we cannot measure an `undivided` workspace of 512 MiB for each convolutional layer on ResNet-18, due to the GPU memory limit; the reason we can run the benchmark here is that the workspace size μ-cuDNN specifies is significantly lower than the limit, and Caffe allocates as much as actually needed.

4.3 CNN Optimization Using WD

Fig. 12 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, BackwardFilter), we place the result with 120MiB WD workspace next to that of 8 MiB WR workspaces.



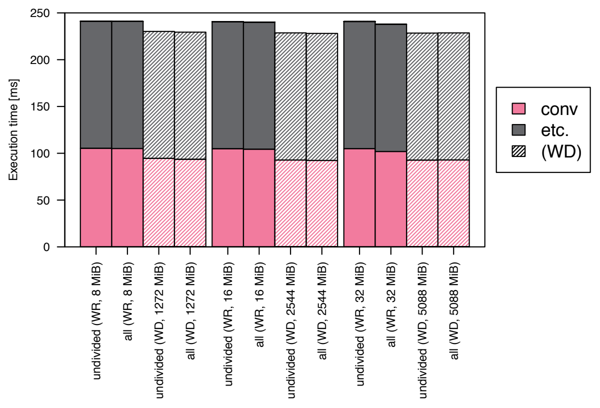


Fig. 12: Benchmark results of AlexNet (top) and ResNet-50 (bottom) on NVIDIA Tesla P100-SMX2 with different workspace sizes and policies (WR and WD). We use a mini-batch size of 256 for AlexNet and 32 for ResNet-50. Note that the adjoined bars have the same workspace limit in total.

In Fig. 12, we can see that the training time decreases as the workspace constraints increase in both WR and WD, but also that WD successfully manages the global memory requirements better, attaining higher performance with the same overall memory footprint (see **Fig. 13** for breakdown). Specifically, when 120 MiB workspace in total is provided for AlexNet, the entire execution time with WD optimization and `all` option is 1.24 times faster than the baseline (WR with `undivided` option) for the entire iteration (or 1.38 times for convolution). WD also outperforms the baseline with 960 MiB workspace in total, which can use 8 times more memory for workspace, by 1.24 times in total execution time.

Furthermore, even for ResNet-50, which has more fine-grained convolutional layers than AlexNet, WD achieves 1.05 times speedup for the entire iteration (or 1.14 times for convolution) with 2544 MiB of total workspace, outperforming the original version (which consumes 5088 MiB) in terms of memory footprint as well. In addition, the ILP for ResNet-50 is still small scale to solve in practical time. When the workspace limit is set to 5088 MiB, the number of 0-1 variables is 562, and the GLPK solver takes

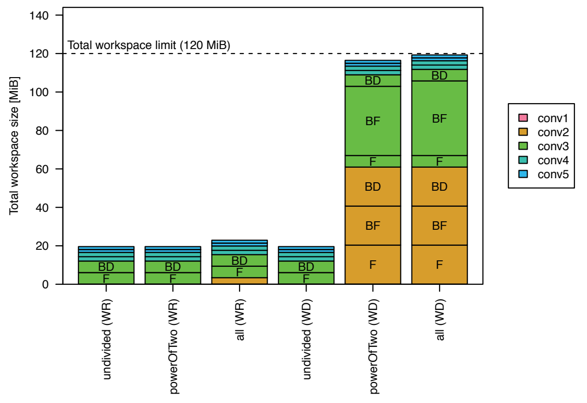


Fig. 13: Assigned workspace division of AlexNet on NVIDIA Tesla P100-SMX2. "F", "BF", "BD" represent kernel types (Forward, BackwardFilter, BackwardData respectively). We use a mini-batch size of 256 for AlexNet. We set a workspace limit of 8 MiB for WR, and a total workspace limit of 120 MiB for WD.

5.46 ms to solve it.

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 (Fig. 8) 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 larger workspace.

In Fig. 13, 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 (WR, `undivided`). 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 1.29 times 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 combination, as defined by the ILP.

5. Related Work

Li et. al [20] indicate that the performance of various convolution algorithms, such as GEMM-based or FFT-based convolution, is heavily affected by the underlying memory layout of layer tensors. In this paper, the authors propose a heuristic to automatically tune each tensor memory layout, as well as an efficient memory layout transformation implementation that is inserted between convolutional layers if needed. 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 deep neural network and GPU architecture. On the other hand, since μ-cuDNN uses the techniques of dynamic programming and integer linear programming, it is mathematically guaranteed that μ-cuDNN provides the best performance that the library can produce, provided that each convolution is independent from the others. In addition, μ-cuDNN can be applied for new convolution algorithms and GPU architectures without additional analysis, since μ-cuDNN treats the algorithms and GPU architectures as black boxes.

Shirahata et al. [21] propose a memory reduction algorithm that reuses memory space of neuron activations and parameters during back propagation. The presented algorithm nearly halves the GPU memory consumption, and the authors imply that one of the main purposes of the algorithm is to enable training with larger mini-batches, which may cancel the memory footprint reduction. Thus μ-cuDNN complements the memory reduction algorithm to increase the mini-batch size.

6. 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 workspace. We have shown that μ-cuDNN works well even with recent CNNs, which are composed of many convolutional layers, and can easily be integrated into existing deep learning frameworks.

The performance of μ-cuDNN demonstrated in this paper suggests that other layer types can be optimized as well, if they can be computed by different algorithms. This is because μ-cuDNN does not use any special properties of convolutions, apart from gradient accumulation.

In addition, the result of WD optimization (Fig. 13) provides us with an 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 Tal Ben-Nun), Student Summer Research Fellowship (for Yosuke Oyama), and JST CREST Grant Number JPMJCR1303, JPMJCR1687, Japan.

References

[1] Krizhevsky, A., Sutskever, I. and Geoffrey E., H.: ImageNet Classification with Deep Convolutional Neural Networks, *Advances in Neural Information Processing Systems 25 (NIPS2012)*, pp. 1–9 (online), DOI: 10.1109/5.726791 (2012).

[2] He, K., Zhang, X., Ren, S. and Sun, J.: Deep Residual Learning for Image Recognition, *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 770–778 (online), DOI: 10.1109/CVPR.2016.90 (2016).

[3] Goyal, P., Dollár, P., Girshick, R. B., Noordhuis, P., Wesolowski, L., Kyrola, A., Tulloch, A., Jia, Y., He, K. and Dollar, P.: Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour, *CoRR*, Vol. abs/1706.0 (online), available from ⟨http://arxiv.org/abs/1706.02677⟩ (2017).

[4] Akiba, T., Suzuki, S. and Fukuda, K.: Extremely Large Minibatch SGD: Training ResNet-50 on ImageNet in 15 Minutes, *ArXiv e-prints*, (online), available from ⟨https://arxiv.org/abs/1711.04325⟩ (2017).

[5] Smith, S. L., Kindermans, P.-J. and Le, Q. V.: Don't Decay the Learning Rate, Increase the Batch Size, *ArXiv e-prints*, (online), available from ⟨https://arxiv.org/abs/1711.00489⟩ (2017).

[6] NVIDIA: NVIDIA cuDNN, https://developer.nvidia.com/cudnn.

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

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

[9] Theano Development Team: Theano: A Python framework for fast computation of mathematical expressions, *arXiv e-prints*, Vol. abs/1605.02688 (online), available from ⟨http://arxiv.org/abs/1605.02688⟩ (2016).

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

[11] Krizhevsky, A.: One weird trick for parallelizing convolutional neural networks, *arXiv preprint*, Vol. abs/1404.5, pp. 1–7 (online), available from ⟨http://arxiv.org/abs/1404.5997⟩ (2014).

[12] Facebook: Caffe2, https://caffe2.ai/.

[13] Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V. and Rabinovich, A.: Going deeper with convolutions, *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, Vol. 07-12-June, pp. 1–9 (online), DOI: 10.1109/CVPR.2015.7298594 (2015).

[14] Makhorin, A.: GLPK (GNU Linear Programming Kit), https://www.gnu.org/software/glpk/.

[15] NVIDIA: Tesla K80 HPC and Machine Learning Accelerator, http://www.nvidia.com/object/tesla-k80.html.

[16] NVIDIA: Tesla P100 Most Advanced Data Center Accelerator, http://www.nvidia.com/object/tesla-p100.html.

[17] NVIDIA: NVIDIA Caffe, https://github.com/NVIDIA/caffe.

[18] Mathieu, M., Henaff, M. and Lecun, Y.: Fast training of convolutional networks through FFTs, *International Conference on Learning Representations (ICLR2014), CBLS, April 2014* (2014).

[19] Lavin, A. and Gray, S.: Fast Algorithms for Convolutional Neural Networks, *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 4013–4021 (2016).

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

[21] Shirahata, K., Tomita, Y. and Ike, A.: Memory reduction method for deep neural network training, *2016 IEEE 26th International Workshop on Machine Learning for Signal Processing (MLSP)*, pp. 1–6 (online), DOI: 10.1109/MLSP.2016.7738869 (2016).