Accelerating Deep Learning Frameworks with Micro-batches

Yosuke Oyama [1] * Tal Ben-Nun [2] Torsten Hoefler [2] Satoshi Matsuoka [3 1]

September 13, 2018

[1]Tokyo Institute of Technology [2]ETH Zurich [3]RIKEN Center for Computational Science
*oyama.y.aa@m.titech.ac.jp, Presenter

Background

Background

- **Convolution** is one of the key operations in Convolutional Neural Networks (CNNs)

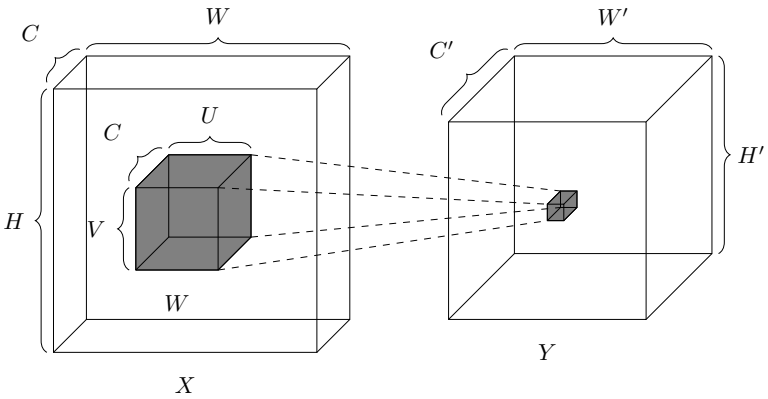


Figure 1: 2D convolution.

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] × X[n, c, h + v, w + u];
```

Background

- NVIDIA cuDNN library provides deep learning primitives for GPUs
  - cuDNN provides several equivalent convolution algorithms

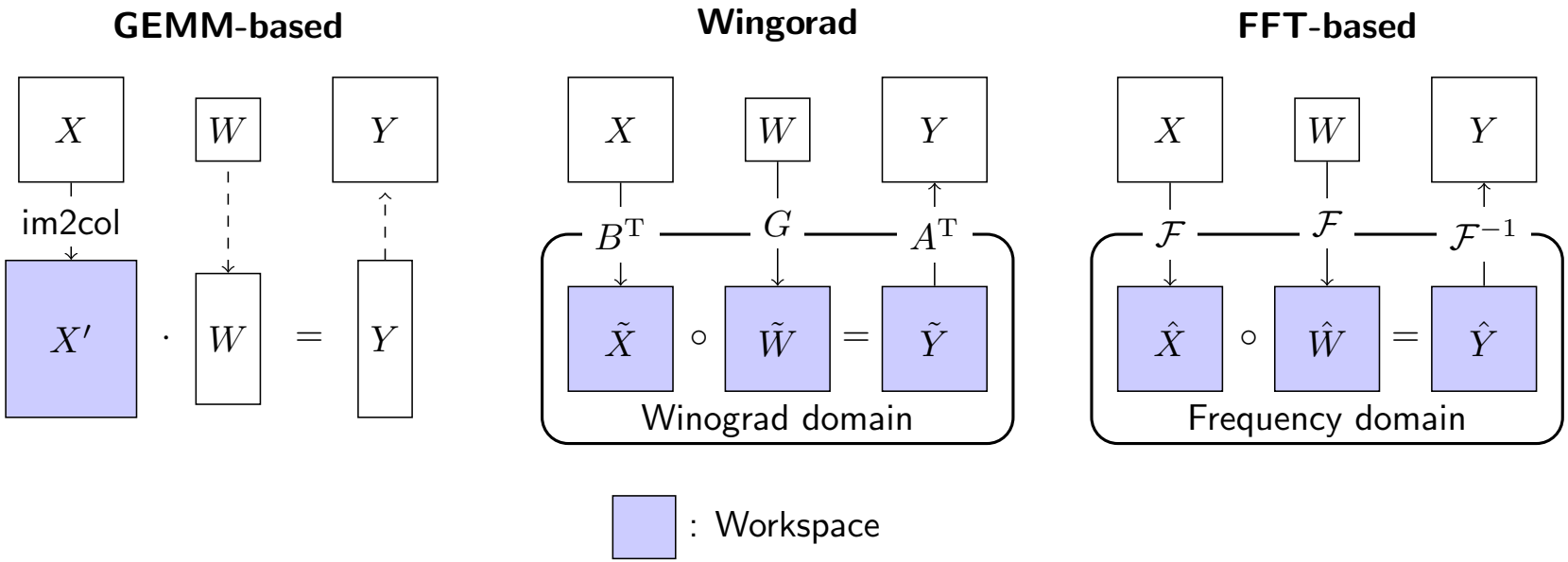


Figure 2: Three different convolution algorithms.

Background

- **Problem statement**: cuDNN may require a workspace as large as the network itself to use efficient convolution algorithms!

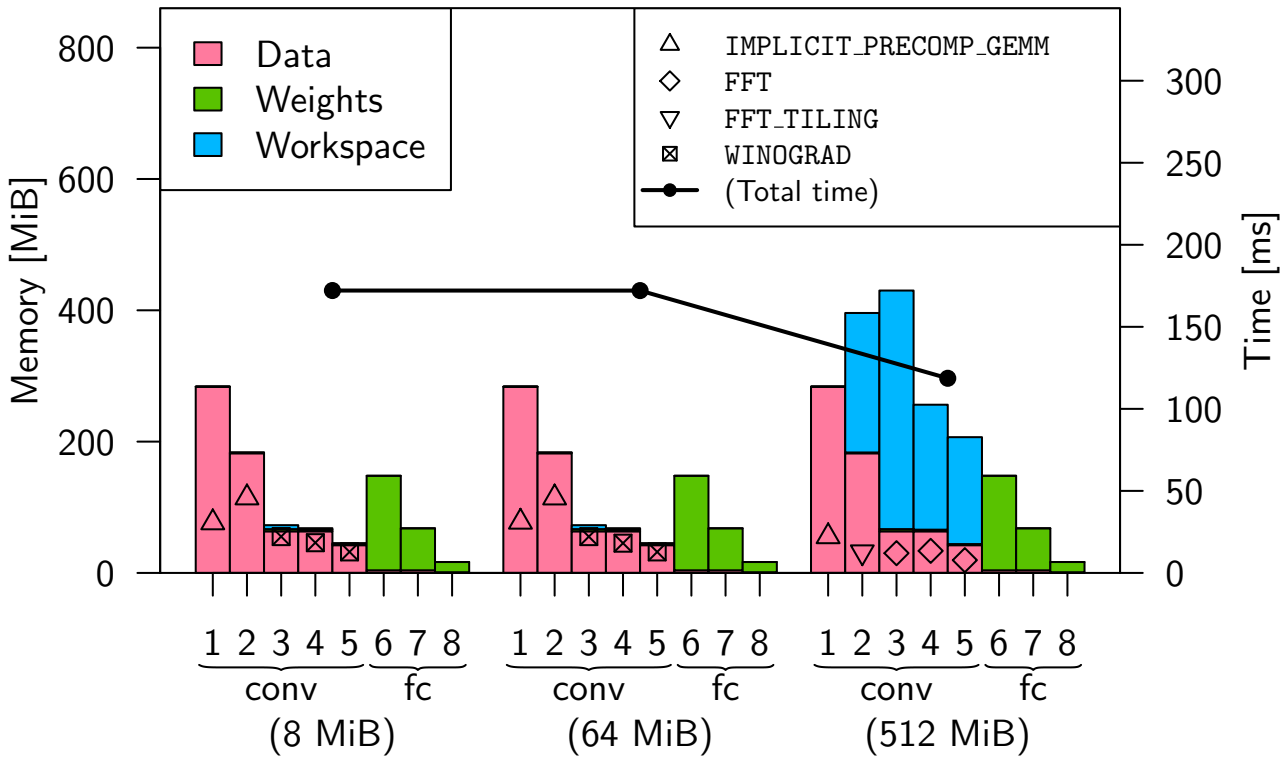


Figure 3: Memory consumption (bars) and computation time (line/points) of AlexNet on P100-SXM2 with different workspace sizes (8, 64, 512 MiB).

Tokyo Tech

Background

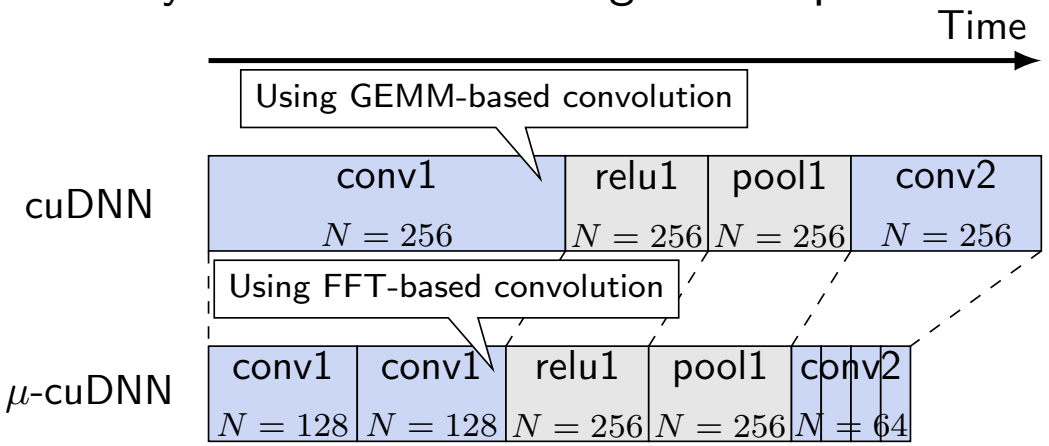
- **Idea**: Loop splitting for the convolution's outermost loop decreases workspace size (as well as computation efficiency)

Algorithm 2 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] × X[n, c, h + v, w + u];
```

Approach and Contribution

- **Approach**: μ-cuDNN, a thin wrapper library for cuDNN, which
  - divides a mini-batch into "micro-batches" by applying loop splitting
  - is based on Dynamic Programming (DP) and Integer Lienar Programming (ILP)
  - provides a Python interface for high-level optimization



- **Contribution**:
  - 1.60x speedup for AlexNet on V100-SXM2 GPU
  - up to 4.54x speedup (1.60x on average) for DeepBench on V100-SXM2 GPU

μ-cuDNN

μ-cuDNN - Software stack

- μ-cuDNN is a wrapper library for cuDNN, which can be called by
  1. a DL framework as low-level performance tuning library
  2. its dedicated Python frontend for high-level performance analysis

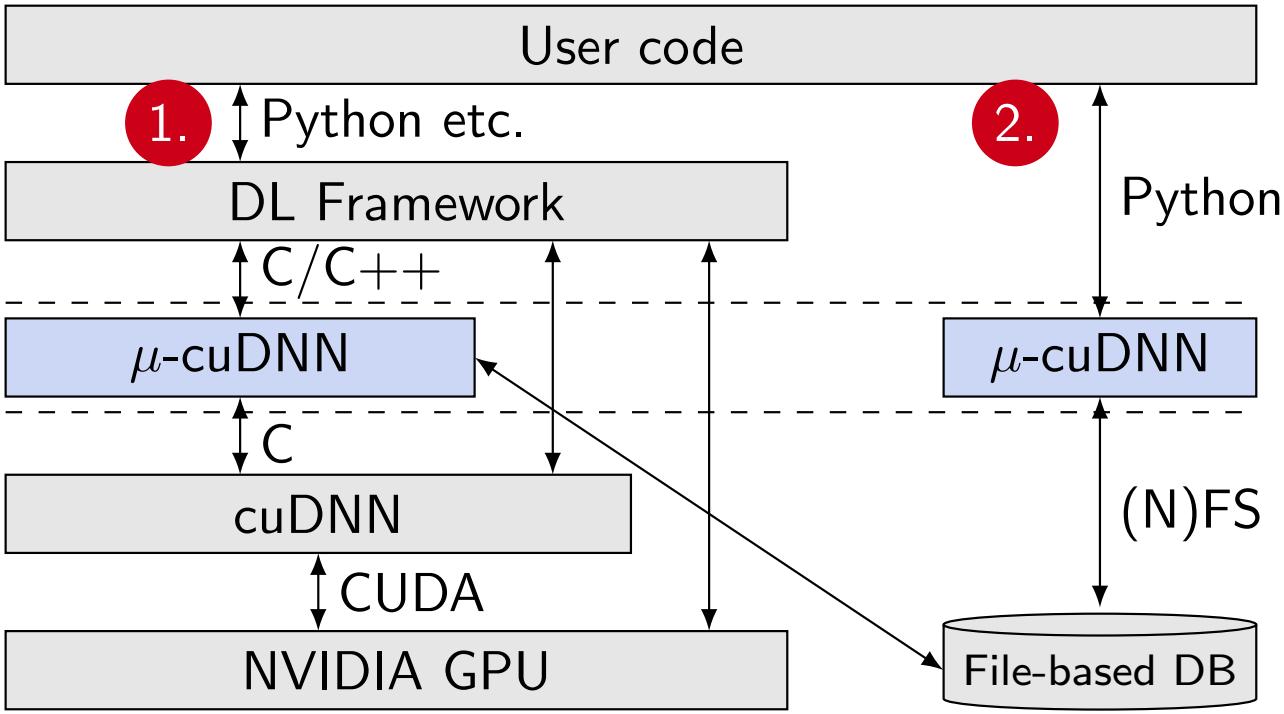


Figure 4: μ-cuDNN software stack.

μ-cuDNN - Methodology

- μ-cuDNN is enabled by replacing cuDNN handle type `cudnnHandle_t`

DL Framework

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

μ-cuDNN

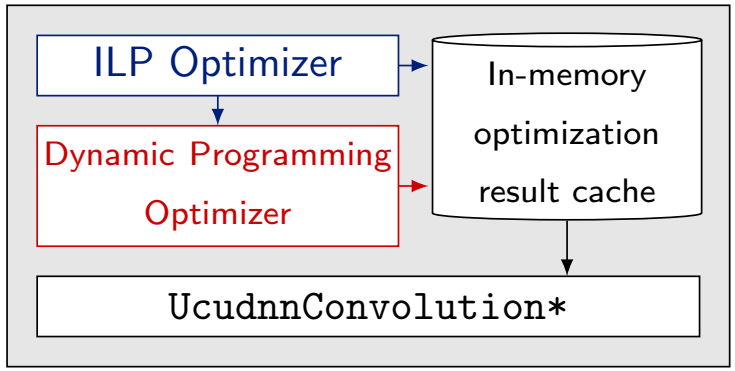


Figure 5: Workflow of μ-cuDNN.

μ-cuDNN - Methodology

- μ-cuDNN is enabled by replacing cuDNN handle type `cudnnHandle_t`

1. The DL framework passes layer's metadata via `cudnnGetConvolution*Algorithm`

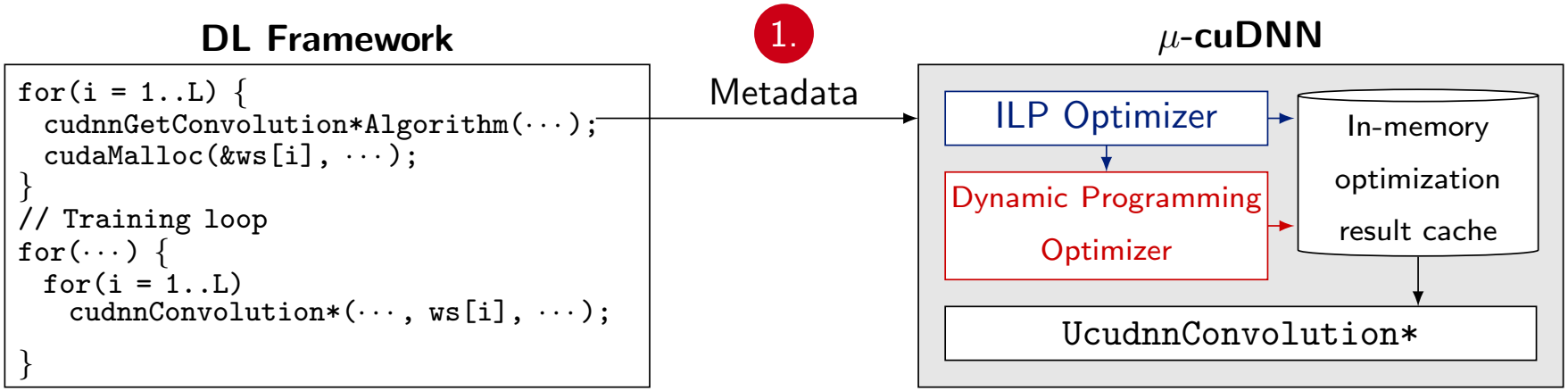


Figure 5: Workflow of μ-cuDNN.

μ-cuDNN - Methodology

- μ-cuDNN is enabled by replacing cuDNN handle type `cudnnHandle_t`

1. The DL framework passes layer's metadata via `cudnnGetConvolution*Algorithm`
2. μ-cuDNN runs ILP (or DP optimizer) and returns resulting workspace size

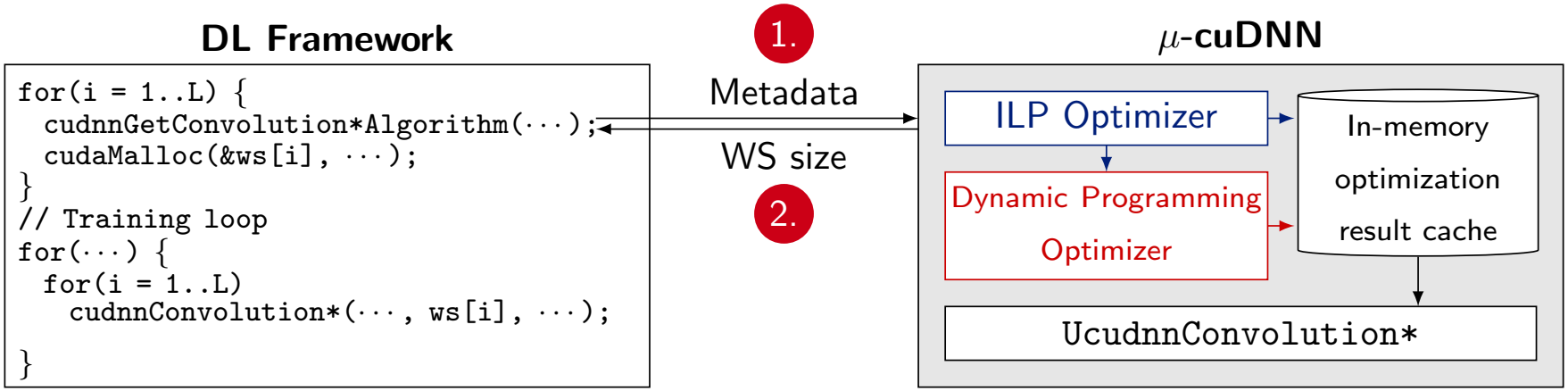


Figure 5: Workflow of μ-cuDNN.

μ-cuDNN - Methodology

- μ-cuDNN is enabled by replacing cuDNN handle type `cudnnHandle_t`

1. The DL framework passes layer's metadata via `cudnnGetConvolution*Algorithm`
2. μ-cuDNN runs ILP (or DP optimizer) and returns resulting workspace size
3. The framework calls `cudnnConvolution*` with the workspace size

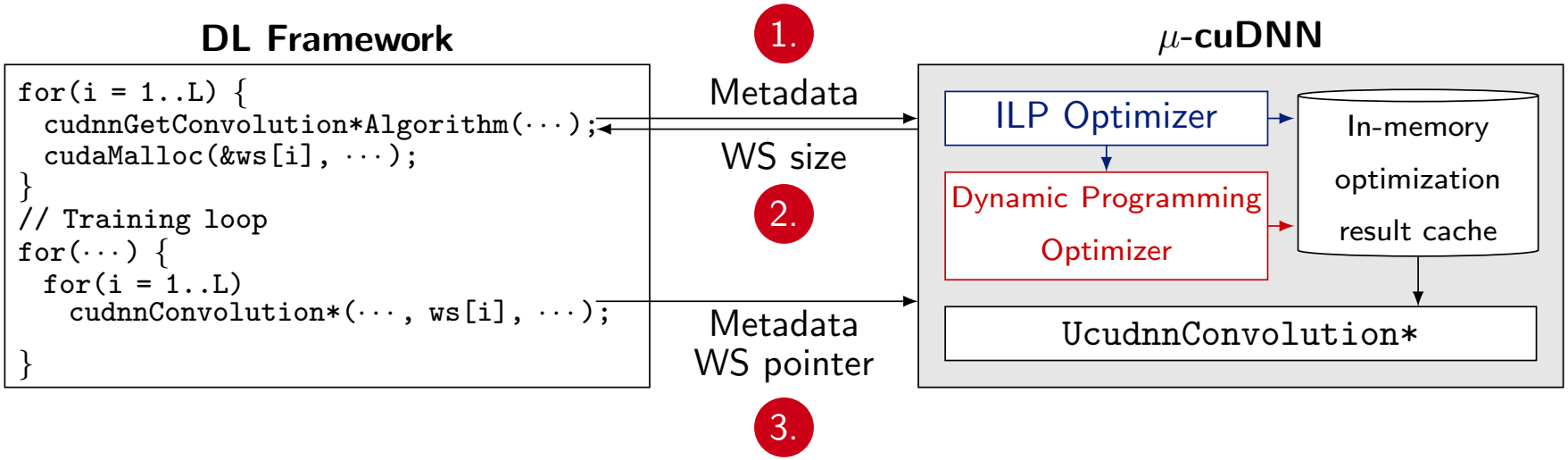


Figure 5: Workflow of μ-cuDNN.

μ-cuDNN - Methodology

- μ-cuDNN is enabled by replacing cuDNN handle type `cudnnHandle_t`

1. The DL framework passes layer's metadata via `cudnnGetConvolution*Algorithm`
2. μ-cuDNN runs ILP (or DP optimizer) and returns resulting workspace size
3. The framework calls `cudnnConvolution*` with the workspace size
4. μ-cuDNN internally calls the convolution function one or more times

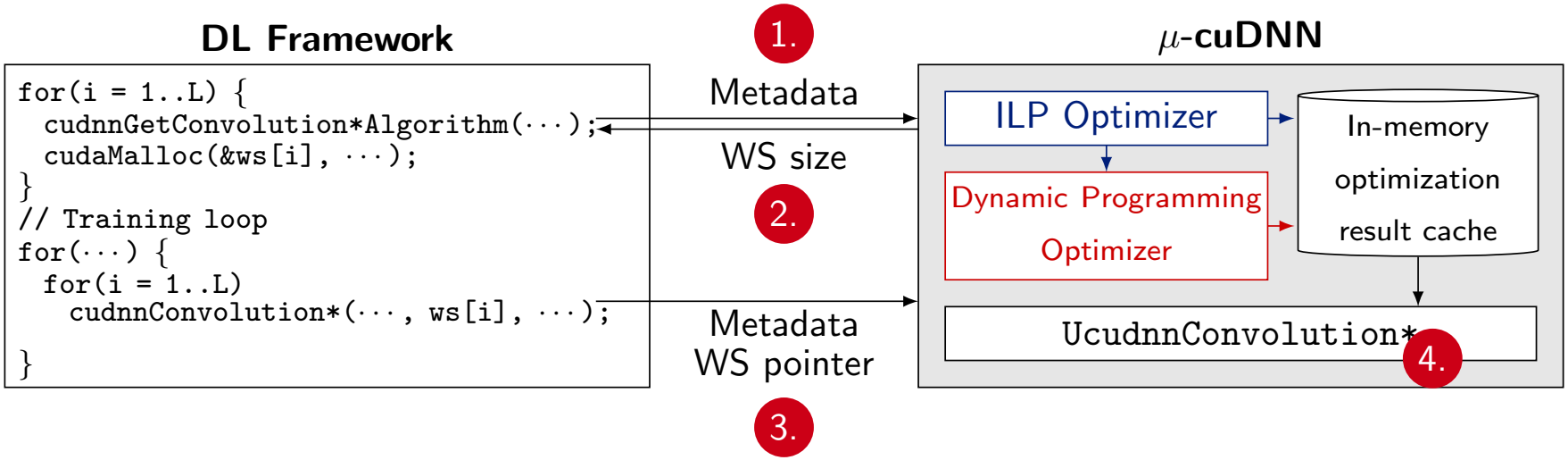


Figure 5: Workflow of μ-cuDNN.

Tokyo Tech

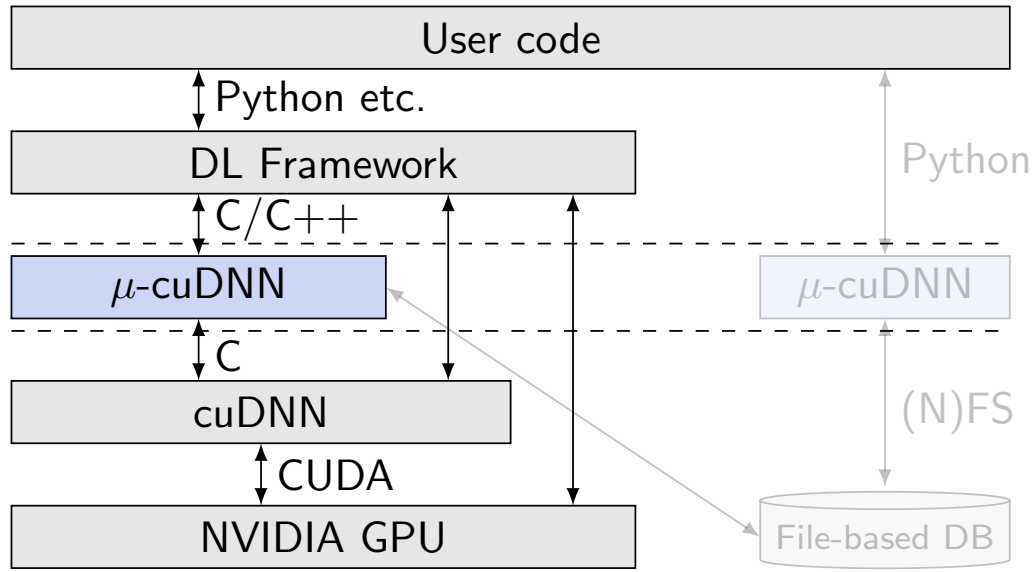
Workspace policies

- μ-cuDNN employs one of two workspace utilization policies
  - **Workspace Reuse (WR)**: Each layer reuses a private workspace
  - **Workspace Division (WD)**: Each layer uses a part of an unified workspace

	WR	WD
Maximum total WS size	$\mathcal{O}(\#$ of layer) ☹	constant ☺
Optimizer	DP	DP+ILP
WS owner	DL framework	μ-cuDNN

μ-cuDNN

WR


User code
Python etc.
DL Framework
C/C++
μ-cuDNN
C
cuDNN
CUDA
NVIDIA GPU
Python
μ-cuDNN
(N)FS
File-based DB

Tokyo Tech

Workspace policies - WR

- **Problem**: Given a mini-batch size B and the fastest execution time $T_\mu(b)$ $(b = 1, 2, \cdots, B)$, compute $T(B)$ where

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

Workspace policies - WR

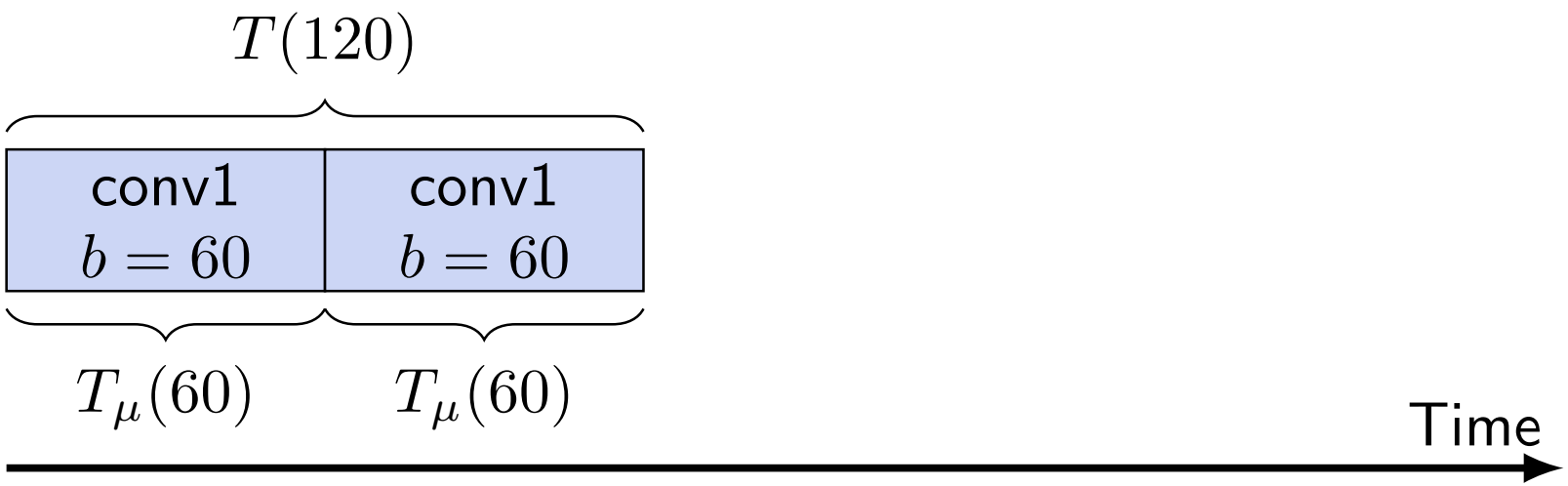
- **Problem**: Given a mini-batch size B and the fastest execution time $T_\mu(b)$ $(b = 1, 2, \cdots, B)$, compute $T(B)$ where

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

Workspace policies - WR

- **Problem**: Given a mini-batch size B and the fastest execution time $T_\mu(b)$ $(b = 1, 2, \cdots, B)$, compute $T(B)$ where

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

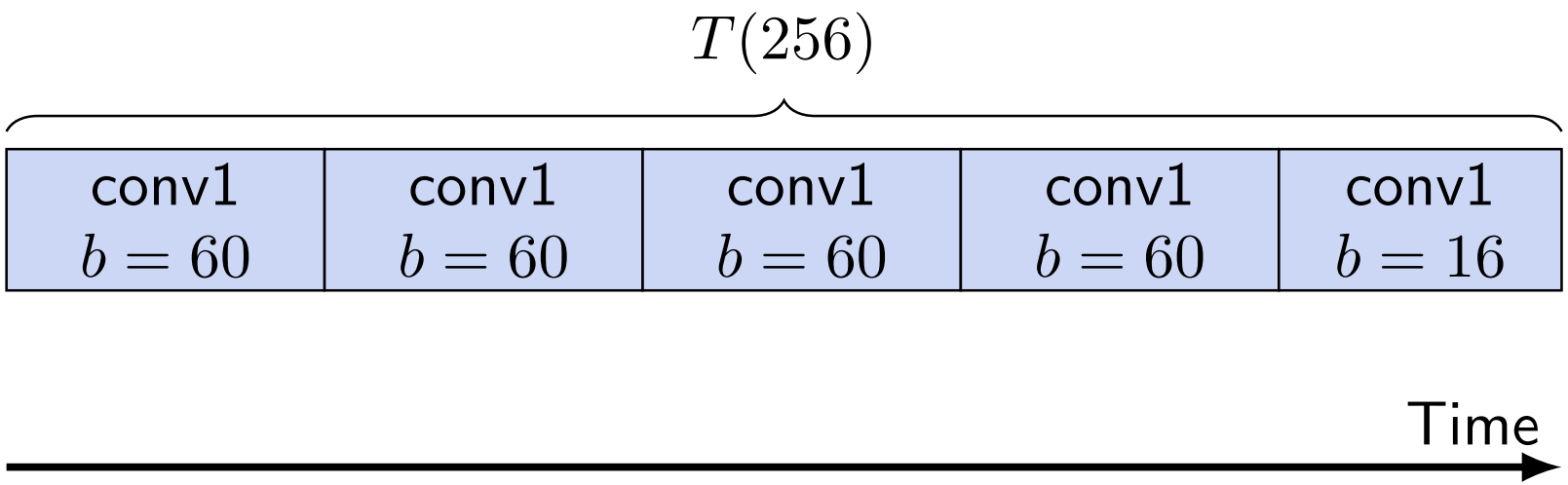


Tokyo Tech

Workspace policies - WR

- **Problem**: Given a mini-batch size B and the fastest execution time $T_\mu(b)$ $(b = 1, 2, \cdots, B)$, compute $T(B)$ where

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



Workspace policies - WR

- **Solution**: Use Dynamic Programming:

Algorithm 3 DP-based solution of WR policy.

1: **for** $b = 1$ to B **do**
2: $\quad \hat{b_\mu} \leftarrow \underset{b_\mu=1,2,\ldots,b}{\operatorname{argmin}} \{T_\mu(b_\mu) + T(b - b_\mu)\}$
3: $\quad T(b) \leftarrow T_\mu(\hat{b_\mu}) + T(b - \hat{b_\mu})$
4: $\quad c(b) \leftarrow \{c_\mu(\hat{b_\mu})\} + c(b - \hat{b_\mu})$
5: **end for**
6: **return** $c(B)$ // **Configuration**; a list of (algorithm ID, batch size)

$$c(256) = \{(\text{FFT}, 100), (\text{FFT}, 100), (\text{GEMM}, 56)\}$$

conv1	conv1	conv1
$c_\mu(100) = (\text{FFT}, 100)$	$c_\mu(100) = (\text{FFT}, 100)$	$c_\mu(56) = (\text{GEMM}, 56)$

Time

Tokyo Tech

Workspace policies - WR

- In practice, μ-cuDNN uses one of three different micro-batch size granularities
  - μ-cuDNN with the `undivided` option acts as cuDNN
- μ-cuDNN increases the number of algorithms by exploiting higher computation precision (`PSEUDO_HALF`) than specified (`TRUE_HALF`) without decreasing the accuracy

Table 1: Micro-batch size policies.

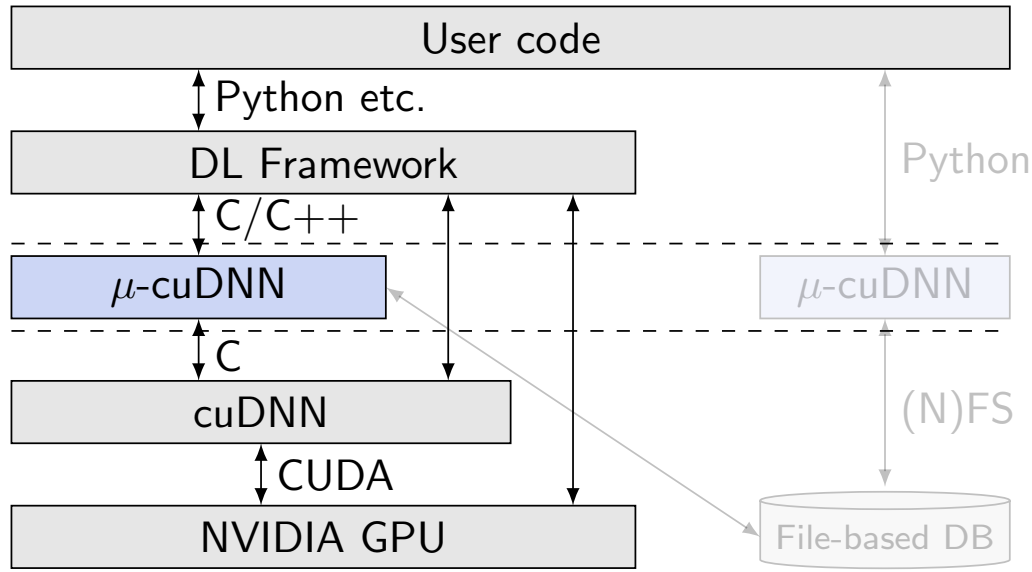
	Micro-batch size set
`all`	$\{1, 2, 3, \cdots, B\}$
`powerOfTwo`	$\{2^0, 2^1, 2^2, \cdots, B\}$
`undivided`	$\{B\}$

Table 2: cuDNN's convolution datatypes.

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

μ-cuDNN

WD


User code
Python etc.
DL Framework
C/C++
μ-cuDNN
C
cuDNN
CUDA
NVIDIA GPU
Python
μ-cuDNN
(N)FS
File-based DB

Tokyo Tech

Workspace policies - WD

- **Problem**:

$$\underset{f:\mathcal{K}\to C_k}{\text{argmin}} \quad \sum_{k\in\mathcal{K}} f(k)T_k(c)$$

$$\text{s.t.} \quad \sum_{k\in\mathcal{K}} f(k)M_k(c) \leq M$$

- where
  - M: Total workspace limit
  - $\mathcal{K}$: A set of convolution kernels
  - C_k: A set of configurations of kernel k
  - $T_k(c)$: The fastest execution time of kernel k and configuration c

Tokyo Tech

Workspace policies - WD

- **Problem**:

$$\underset{f:\mathcal{K}\to C_k}{\operatorname{argmin}} \quad \sum_{k\in\mathcal{K}} f(k)T_k(c)$$

Minimization of the total computation time

$$\text{s.t.} \quad \sum_{k\in\mathcal{K}} f(k)M_k(c) \leq M$$

- where
  - M: Total workspace limit
  - $\mathcal{K}$: A set of convolution kernels
  - C_k: A set of configurations of kernel k
  - $T_k(c)$: The fastest execution time of kernel k and configuration c

Tokyo Tech

Workspace policies - WD

- **Problem**:

$$\underset{f:\mathcal{K}\to C_k}{\operatorname{argmin}} \quad \sum_{k\in\mathcal{K}} f(k)T_k(c)$$

$$\text{s.t.} \quad \sum_{k\in\mathcal{K}} f(k)M_k(c) \le M$$

Total memory comsumption constraint

- where
  - M: Total workspace limit
  - $\mathcal{K}$: A set of convolution kernels
  - C_k: A set of configurations of kernel k
  - $T_k(c)$: The fastest execution time of kernel k and configuration c

Workspace policies - WD

- **Problem**:

$$\underset{f:\mathcal{K}\to C_k}{\text{argmin}} \quad \sum_{k\in\mathcal{K}} f(k)T_k(c)$$

$$\text{s.t.} \quad \sum_{k\in\mathcal{K}} f(k)M_k(c) \leq M$$

- where
  - M: Total workspace limit
  - $\mathcal{K}$: A set of convolution kernels
  - C_k: A set of configurations of kernel k
  - $T_k(c)$: The fastest execution time of kernel k and configuration c

- **Solution**: Use ILP:

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

$$\text{s.t.} \quad \sum_{k\in\mathcal{K}} \sum_{c\in C_k} M_k(c)x_{k,c} \leq M$$

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

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

- where $f(k) = c \Leftrightarrow x_{k,c} = 1$

Tokyo Tech

Workspace policies - WD

- **Problem**:

$$\underset{f:\mathcal{K}\to C_k}{\text{argmin}} \quad \sum_{k\in\mathcal{K}} f(k)T_k(c)$$

$$\text{s.t.} \quad \sum_{k\in\mathcal{K}} f(k)M_k(c) \le M$$

- where
  - M: Total workspace limit
  - $\mathcal{K}$: A set of convolution kernels
  - C_k: A set of configurations of kernel k
  - $T_k(c)$: The fastest execution time of kernel k and configuration c

- **Solution**: Use ILP:

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

$$\text{s.t.} \quad \sum_{k\in\mathcal{K}} \sum_{c\in C_k} M_k(c)x_{k,c} \le M$$

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

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

- where $f(k) = c \Leftrightarrow x_{k,c} = 1$

Workspace policies - WD

- **Problem**:

$$\underset{f:\mathcal{K}\to C_k}{\text{argmin}} \quad \sum_{k\in\mathcal{K}} f(k)T_k(c)$$

$$\text{s.t.} \quad \sum_{k\in\mathcal{K}} f(k)M_k(c) \le M$$

- where
  - M: Total workspace limit
  - $\mathcal{K}$: A set of convolution kernels
  - C_k: A set of configurations of kernel k
  - $T_k(c)$: The fastest execution time of kernel k and configuration c

- **Solution**: Use ILP:

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

$$\text{s.t.} \quad \sum_{k\in\mathcal{K}} \sum_{c\in C_k} M_k(c)x_{k,c} \le M$$

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

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

- where $f(k) = c \Leftrightarrow x_{k,c} = 1$

Workspace policies - WD

- **Problem**:

$$\underset{f:\mathcal{K}\to C_k}{\text{argmin}} \quad \sum_{k\in\mathcal{K}} f(k)T_k(c)$$

$$\text{s.t.} \quad \sum_{k\in\mathcal{K}} f(k)M_k(c) \le M$$

- where
  - M: Total workspace limit
  - $\mathcal{K}$: A set of convolution kernels
  - C_k: A set of configurations of kernel k
  - $T_k(c)$: The fastest execution time of kernel k and configuration c

- **Solution**: Use ILP:

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

$$\text{s.t.} \quad \sum_{k\in\mathcal{K}} \sum_{c\in C_k} M_k(c)x_{k,c} \le M$$

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

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

Assign one configuration for each layer

- where $f(k) = c \Leftrightarrow x_{k,c} = 1$

Workspace policies - WD

- We only use configurations on the Pareto front to enumerate "desirable" configurations C_k:

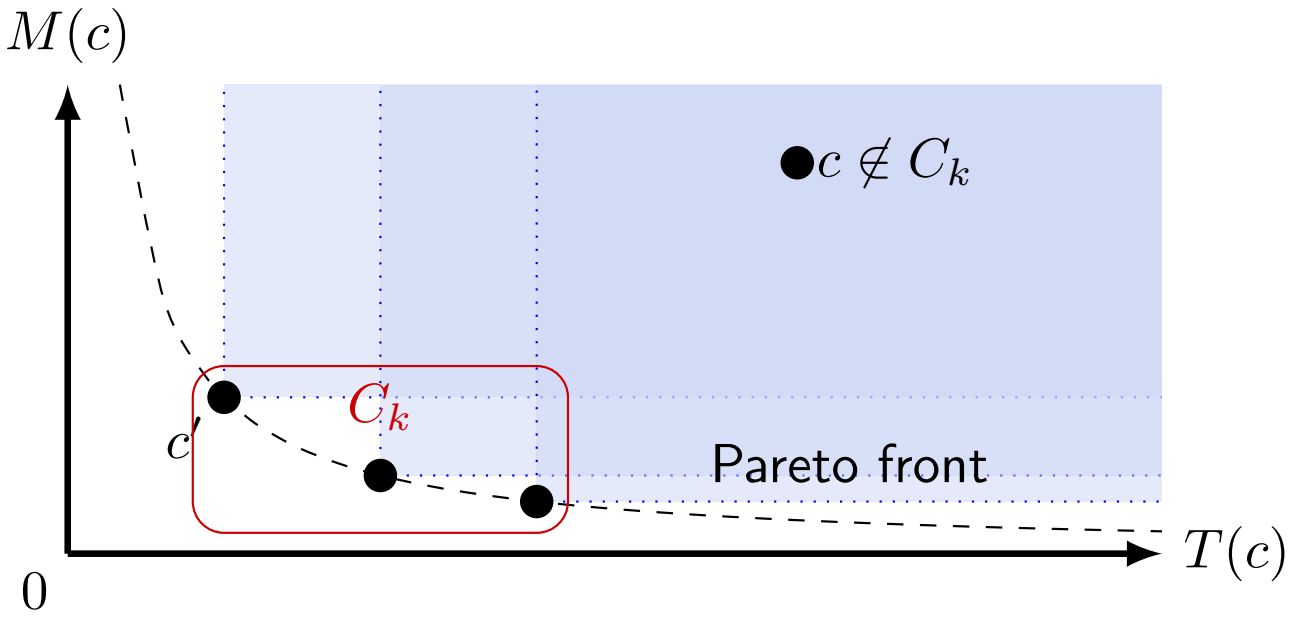


Figure 6: Pareto front of configurations.

Tokyo Tech

Workspace policies - WD

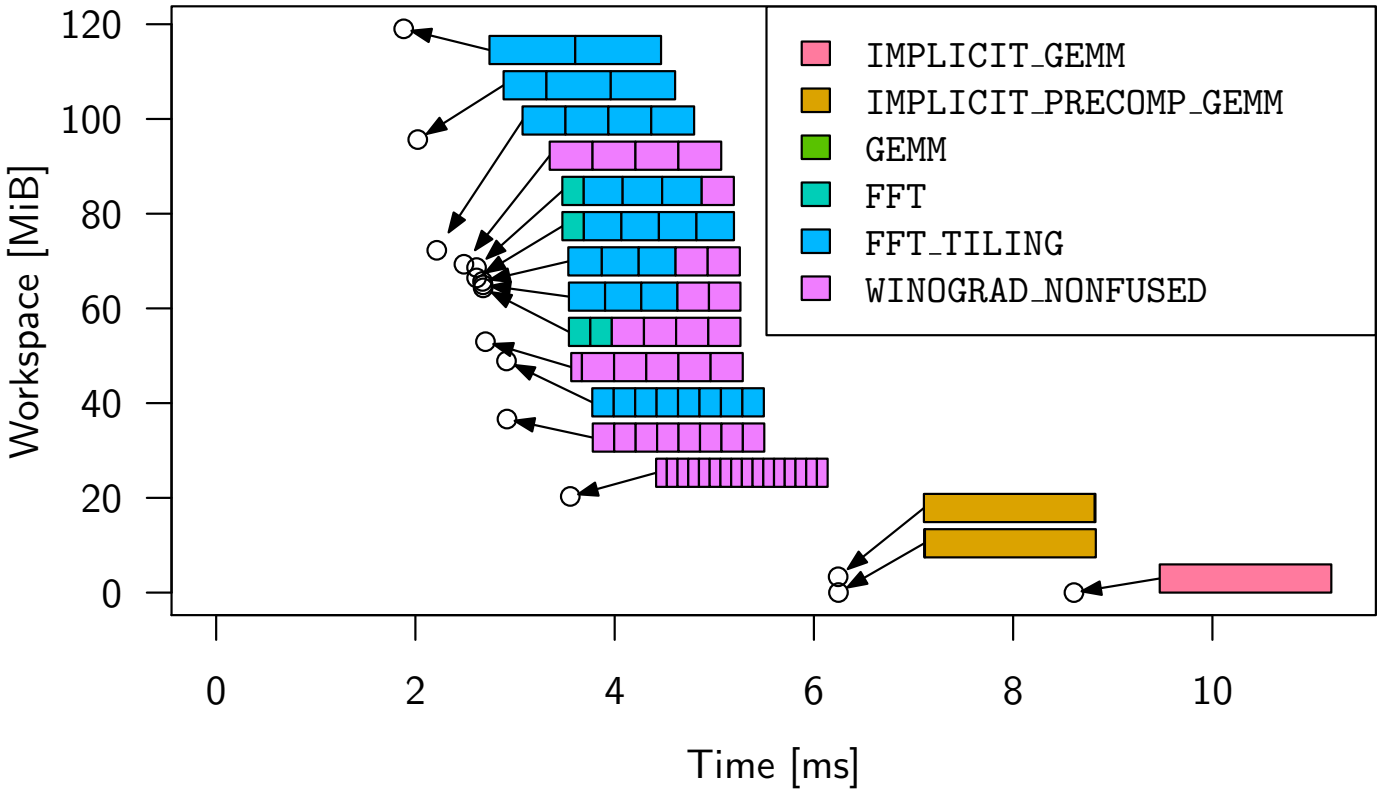
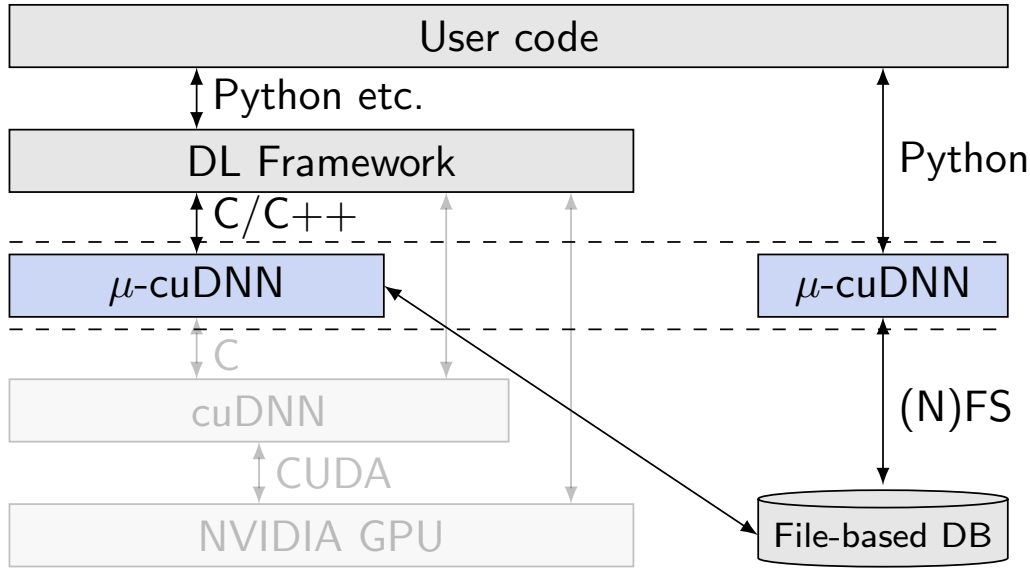


Figure 7: Pareto front of AlexNet's "conv2" layer on P100-SXM2.

μ-cuDNN

High-level optimization frontend



High-level optimization frontend

- μ-cuDNN's Python interface performs framework-independent performance analysis
  - by passing layers' metadata via file-based database (1.,2.,3.)

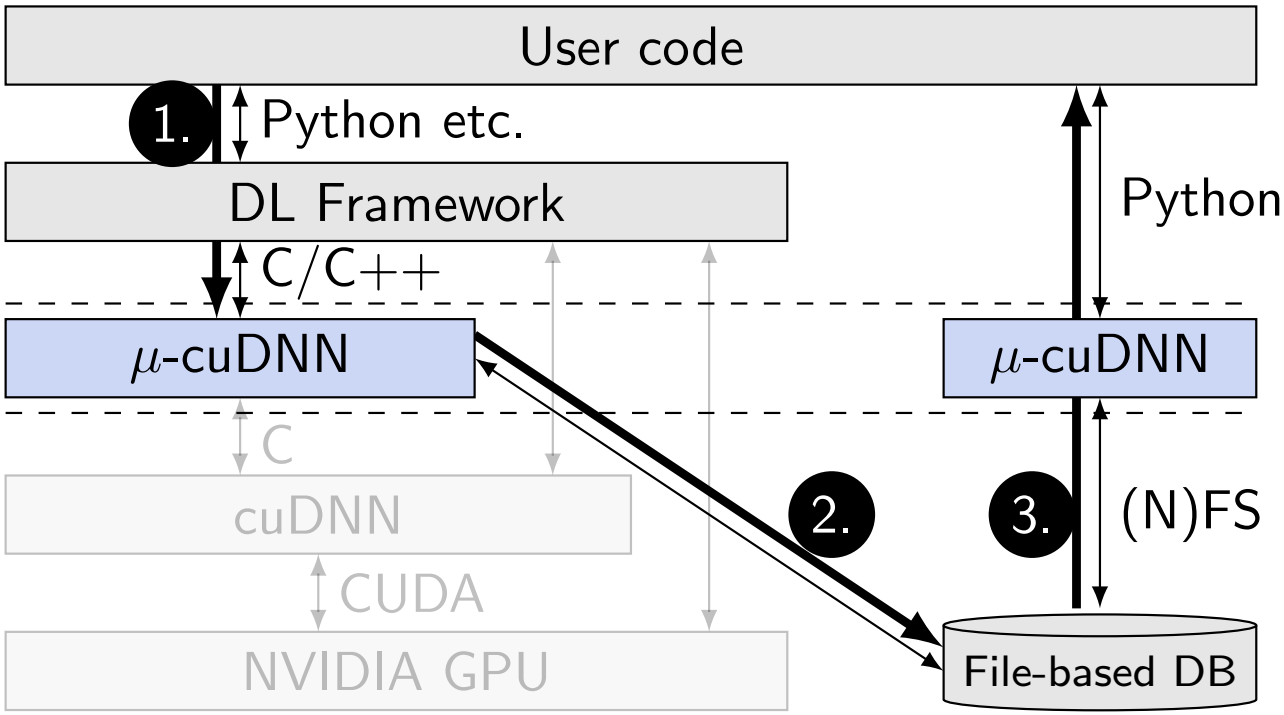


Figure 8: μ-cuDNN software stack.

Tokyo Tech

High-level optimization frontend

- We provide a function to minimize training time of data-parallel training by assigning different micro-batch sizes to heterogeneous GPUs
  - We ignore time to perform inter-GPU all-reduce since communication is typically overlapped with computation

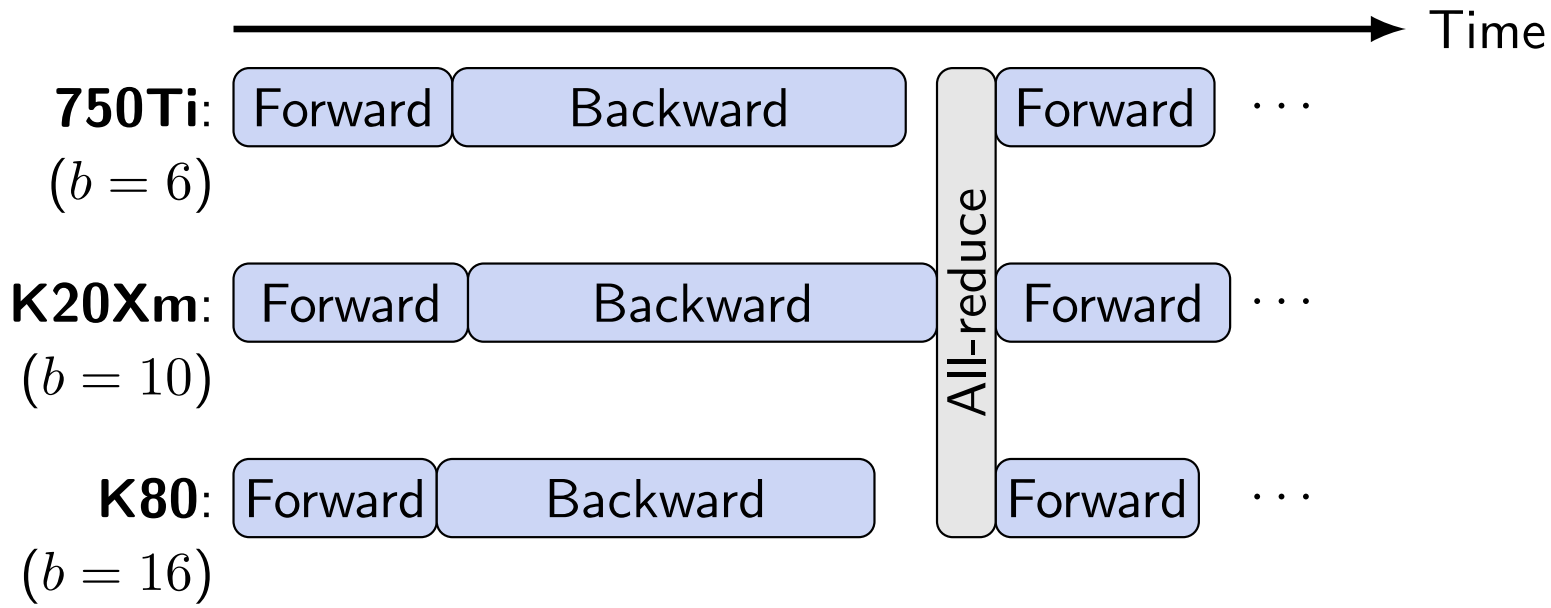


Figure 9: Data-parallel training on a heterogeneous GPU cluster.

High-level optimization frontend

- **Solution**: Given
  - G: A set of GPUs
  - B: A mini-batch size
  - $t_{g,b}$: Computation time on GPU g with a batch size of b
  - $\mathcal{B}$: A batch-size set,

  compute

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

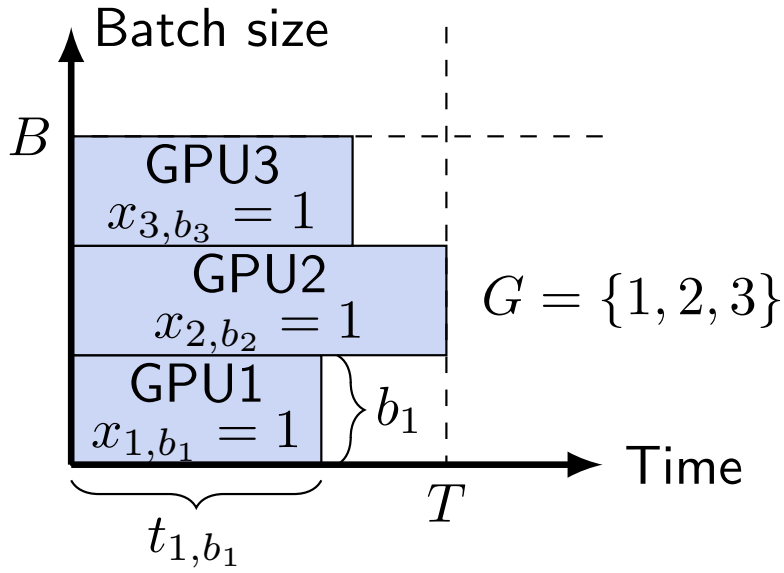


Figure 10: Illustration of the ILP problem.

High-level optimization frontend

- **Solution**: Given
  - G: A set of GPUs
  - B: A mini-batch size
  - $t_{g,b}$: Computation time on GPU g with a batch size of b
  - $\mathcal{B}$: A batch-size set,

compute

$$\min \quad \max_{g \in G} \left\{ \sum_{b \in \mathcal{B}} t_{g,b} x_{g,b} \right\}$$

Minimize the slowest GPU

$$\text{s.t.} \quad \sum_{b \in \mathcal{B}} x_{g,b} \leq 1 \ (\forall g \in G)$$

$$\sum_{g \in G} \sum_{b \in \mathcal{B}} b x_{g,b} = B$$

$$x_{g,b} \in \{0, 1\} \ (\forall g \in G, \forall b \in \mathcal{B})$$

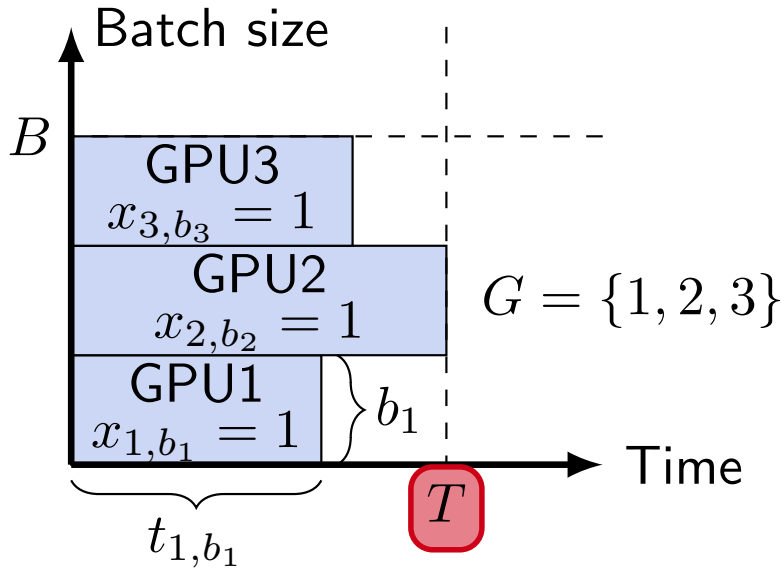


Figure 10: Illustration of the ILP problem.

High-level optimization frontend

- **Solution**: Given
  - G: A set of GPUs
  - B: A mini-batch size
  - $t_{g,b}$: Computation time on GPU g with a batch size of b
  - $\mathcal{B}$: A batch-size set,

compute

$$\min \quad \max_{g \in G} \left\{ \sum_{b \in \mathcal{B}} t_{g,b} x_{g,b} \right\}$$

$$\text{s.t.} \quad \sum_{b \in \mathcal{B}} x_{g,b} \leq 1 \ (\forall g \in G)$$

Select one batch size for each GPU

$$\sum_{g \in G} \sum_{b \in \mathcal{B}} b x_{g,b} = B$$

$$x_{g,b} \in \{0, 1\} \ (\forall g \in G, \forall b \in \mathcal{B})$$

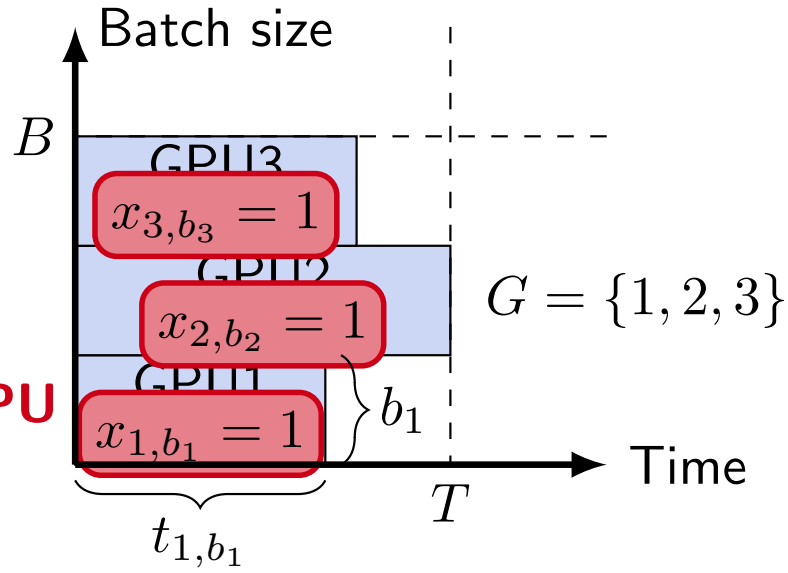


Figure 10: Illustration of the ILP problem.

High-level optimization frontend

- **Solution**: Given
  - G: A set of GPUs
  - B: A mini-batch size
  - $t_{g,b}$: Computation time on GPU g with a batch size of b
  - $\mathcal{B}$: A batch-size set,

compute

$$\min \quad \max_{g \in G} \left\{ \sum_{b \in \mathcal{B}} t_{g,b} x_{g,b} \right\}$$

$$\text{s.t.} \quad \sum_{b \in \mathcal{B}} x_{g,b} \leq 1 \ (\forall g \in G)$$

$$\sum_{g \in G} \sum_{b \in \mathcal{B}} b x_{g,b} = B$$

The total batch size should be equal to the mini-batch size

$$x_{g,b} \in \{0, 1\} \ (\forall g \in G, \forall b \in \mathcal{B})$$

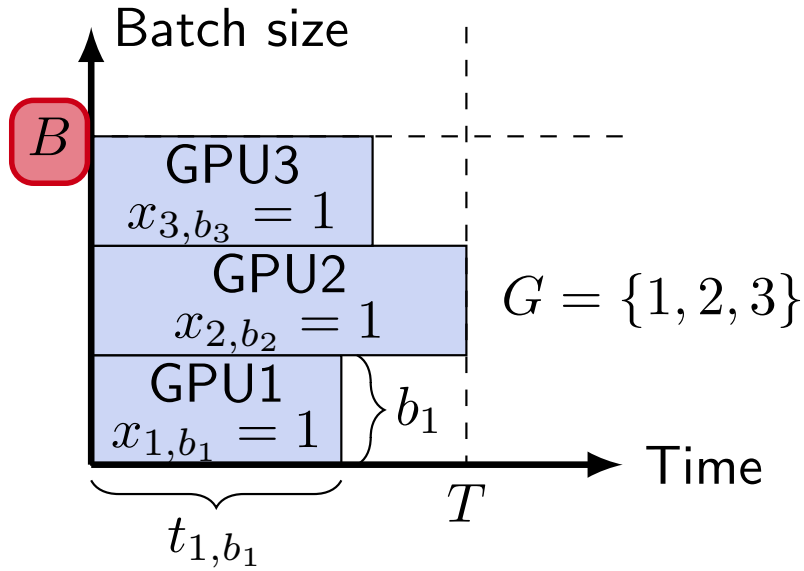


Figure 10: Illustration of the ILP problem.

Performance evaluation

Tokyo Tech

Evaluation environment

- **GPUs**: **K80**, **P100-SXM2**, **V100-SXM2**, K20Xm, 750Ti
- **cuDNN**: 7.1 (or 6.0 for Caffe and TensorFlow)
- **Frameworks**: Caffe 1.0, TensorFlow 1.4.1
- **LP solver**: GNU Linear Programming Kit (GLPK) 4.63

Table 3: GPU specification.

	Generation	TFlop/s FP32	TFlop/s FP16[1]	Memory [GiB]	Tensor cores	Host
K80	Kepler	8.73	-	24	-	KFC[2]
P100-SXM2	Pascal	10.6	21.2	16	-	T3[3]
V100-SXM2	Volta	15.7	125	16	✓	DGX-1
GTX 750Ti	Maxwell	1.31	-	2	-	KFC
K20Xm	Kepler	3.95	-	6	-	KFC

[1] including Tensor Cores' mixed-precision arithmetic
[2] TSUBAME-KFC/DL supercomputer
[3] TSUBAME 3.0 supercomputer

Tokyo Tech

Single convolutional layer

- μ-cuDNN achieves **2.33x** speedup on AlexNet's "conv2" layer by utilizing both FFT-based convolution and Winograd's algorithm
  - GEMM-based convolution requires only 4.3 KiB for workspace but slow
  - FFT-based convolution is faster than GEMM but it requires 213 MiB of workspace with a mini-batch size of 256

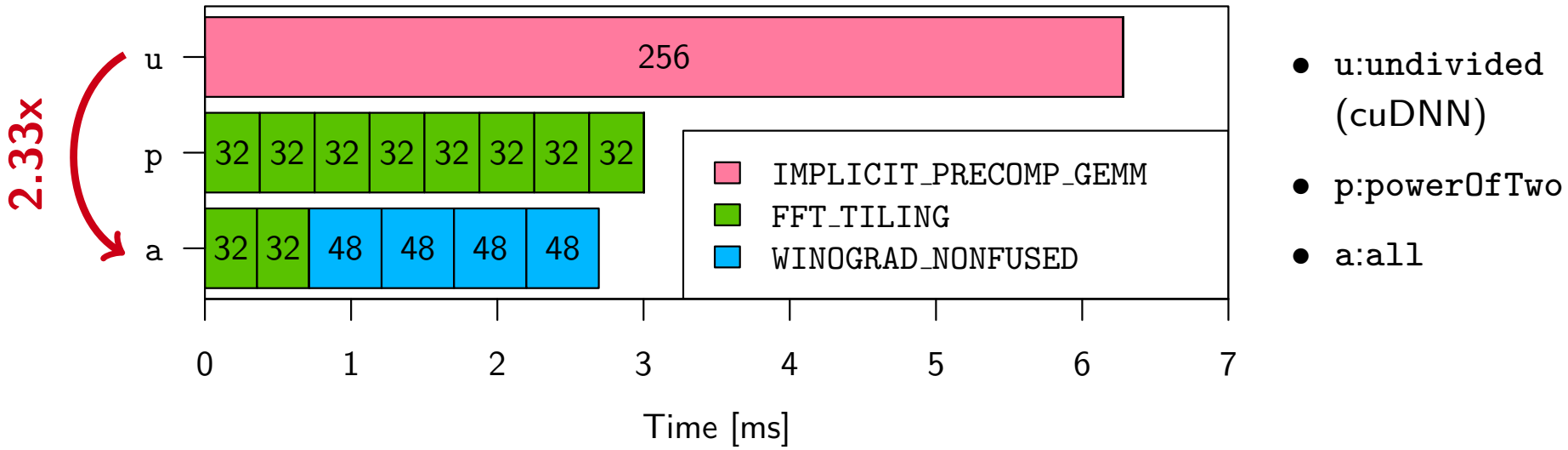


Figure 11: Time (bars) and micro-batch sizes (labels in bars) of forward convolution of AlexNet's "conv2" layer on P100-SXM2.

DeepBench

- We evaluate μ-cuDNN with DeepBench's 94 convolutional layers
  - μ-cuDNN achieves up to **4.54x** speedup (**1.60x** on average) on a V100-SXM2-GPU with Tensor Cores
    - μ-cuDNN exploits `PSEUDO_HALF` in 69% of the layers
  - μ-cuDNN also achieves **1.16x**, **1.73x** average speedups for 3×3 kernels on P100 and V100 respectively

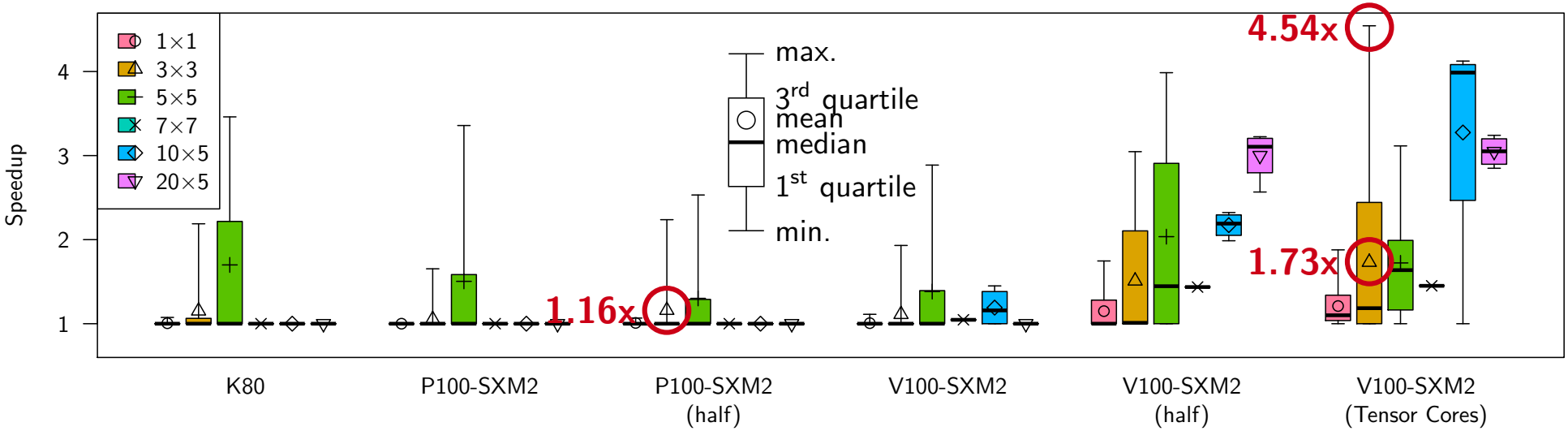
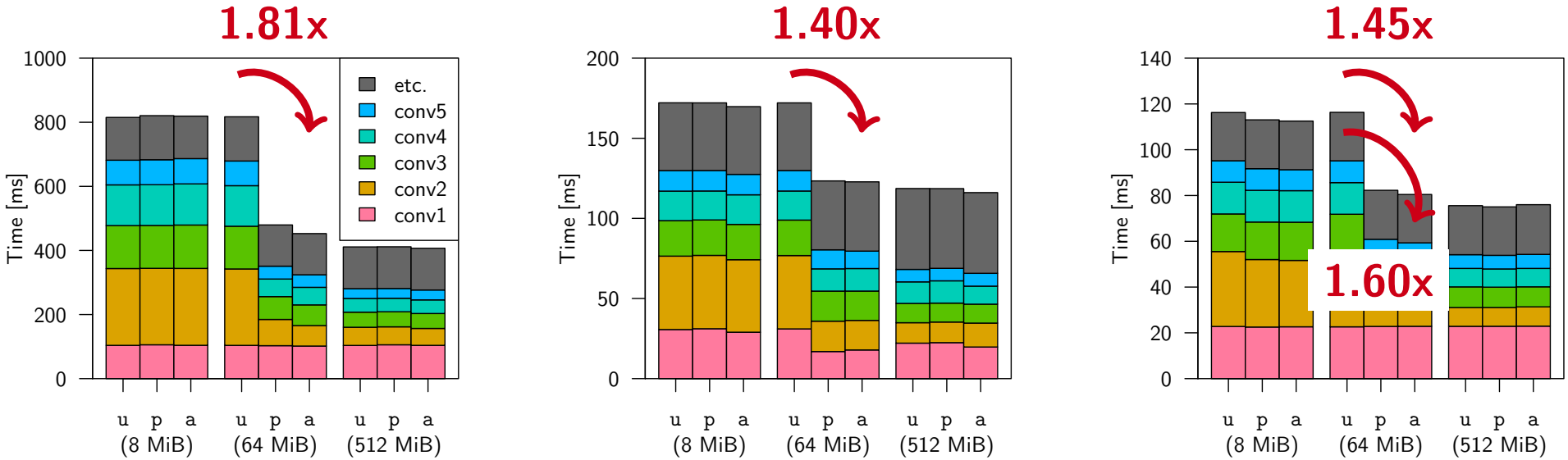


Figure 12: Relative speedup of DeepBench's forward convolution against cuDNN.

Caffe - WR policy

- μ-cuDNN on Caffe framework achieves **1.45x** speedup (and **1.60x** w.r.t. convolutions alone) on V100-SXM2 GPU
  - achieves less speedups with tiny workspace (8 MiB) or huge workspace (512 MiB), due to lack of effectiveness of micro-batching
- μ-cuDNN achieves similar speedups on TensorFlow



(a) K80 **(b)** P100-SXM2 **(c)** V100-SXM2

Figure 13: Benchmark results of AlexNet on three different GPUs with different workspace sizes (8, 64, 512 MiB).

Tokyo Tech

Caffe - WD policy

- μ-cuDNN on Caffe framework achieves **1.38x** and **1.14x** speedup for convolutional layers of AlexNet and ResNet-50 on P100-SXM2 GPU
- Time to solve the ILP problem was negligible (5.46 ms for ResNet-50)

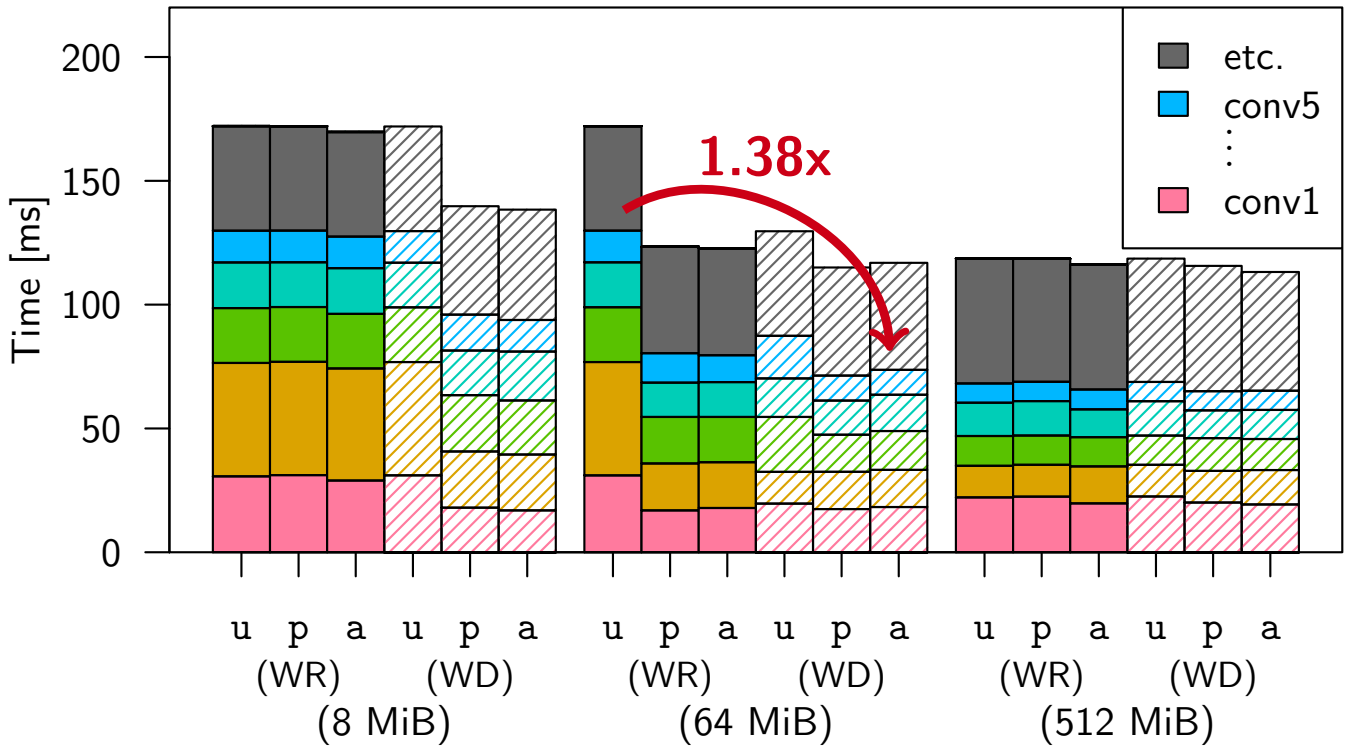


Figure 14: Benchmark results of AlexNet on P100-SXM2 with different workspace sizes and policies.

Tokyo Tech

Caffe - WD policy

- cuDNN with the WD policy spares most of the workspace for "conv2" and "conv3"
  - which are the most time-consuming and can be accelerated by efficient convolution algorithms

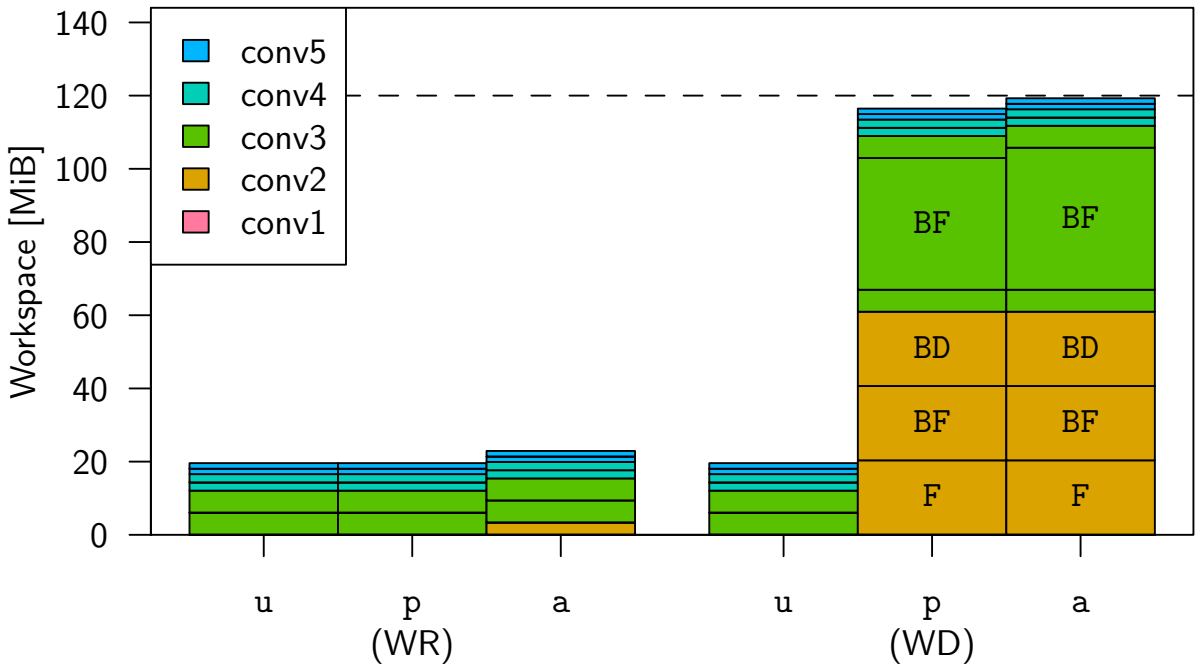


Figure 15: Assigned workspace division of AlexNet on P100-SXM2.

Case study: Heterogeneous cluster optimization

- We estimate time of forward-backward passes of ResNet-18 on three GPUs: 750Ti, K20Xm and K80
  - The GPUs accelerate training of ResNet-18 by **2.20x** than one K80 GPU chip using the same mini-batch size
  - Time to perform all-reduce was negligible
    - 2.63 ms, 1 MiB data, 3 nodes, MVAPICH 2.3a

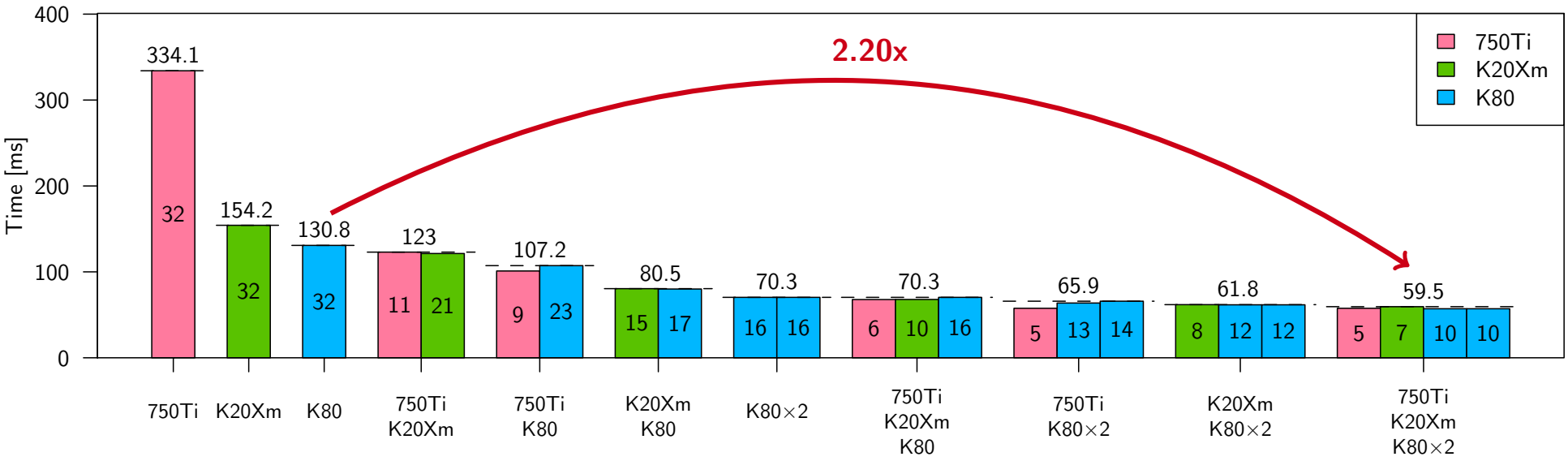


Figure 16: Estimated time of forward-backward passes of ResNet-18 on heterogeneous GPUs.

Tokyo Tech

Conclusion

- μ-cuDNN is a "free-lunch" auto-tuning library for cuDNN
  - does not violate the semantics of computation
    - even improves computation precision for better performance in some cases
  - is independent from the underlying framework
  - generalizes performance analysis around convolutional layers

μ-cuDNN is available online at
`https://github.com/spcl/ucudnn`