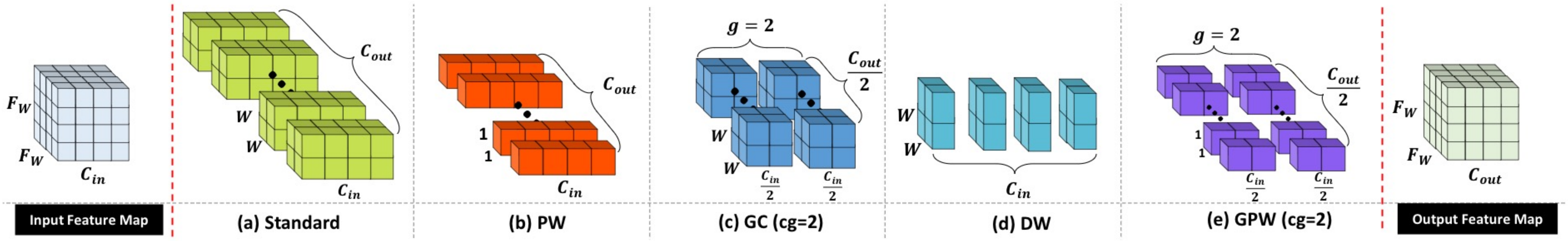
DSXplore: Optimizing Convolutional Neural Networks via Sliding-Channel Convolutions

Yuke Wang, Boyuan Feng, Yufei Ding

UC Santa Barbara, CA, USA

yuke_wang@cs.ucsb.edu

Diverse Convolutions in CNNs



Challenges

- **First, there are limited DSC designs to balance accuracy performance and the size of computation/parameters.**

- Derived from the **DW+PW** design for standard convolution replacement. The more effective DSC schemes that can potentially deliver better accuracy and model size trade-offs still remain uncovered.
- For example, we can further reduce the computation cost and parameter size by **combining the group convolution (GC)** (dividing the input and output channel into the same number of groups and only applying standard convolution within each group) **with PW.**

- **Second, there is a lack of efficient implementation support for new factorized kernels.**

- Rely on the **deep-learning infrastructure with standard/group convolutions** for their factorized kernel implementation.
- For example, the DW convolution can be expressed as the **extreme case of the GC with the number of groups equal the input channels**, while the PW convolution can be expressed as another **special case of standard convolution with the 1×1 kernel spatial dimension**.
- Therefore, the better factorized kernel that **may bring better accuracy and lower computation and memory costs** but not in the above categories cannot leverage the existing convolutional primitives for an effective implementation

Contributions

- **A novel Sliding-channel Convolution (SCC) design.**
  - Balance the accuracy performance and the reduction of computation and memory cost.
  - Enormous design exploration space with parameterized design strategy.
- **Seamless integration with the original Pytorch framework.**
  - Drop-in replacement of the existing DSCs to facilitate the training and inference of an end-to-end fashion.
- **An optimized GPU-implementation tailored for SCC design.**
  - Output-centric forward and input-centric backward optimization
  - Optimization based on the convolutional specialty (cyclic channel) of its filters.
- **Extensive Experiments.**
  - Better accuracy and lower computation/memory cost compared with the existing DSC.

Comparison with existing kernels

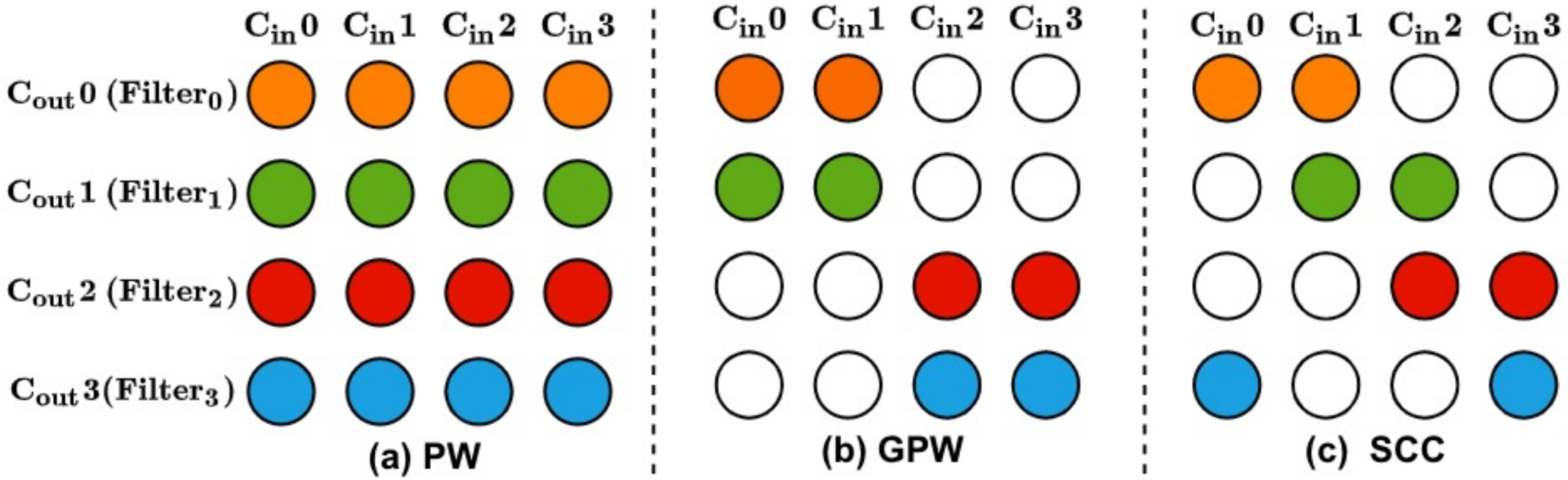


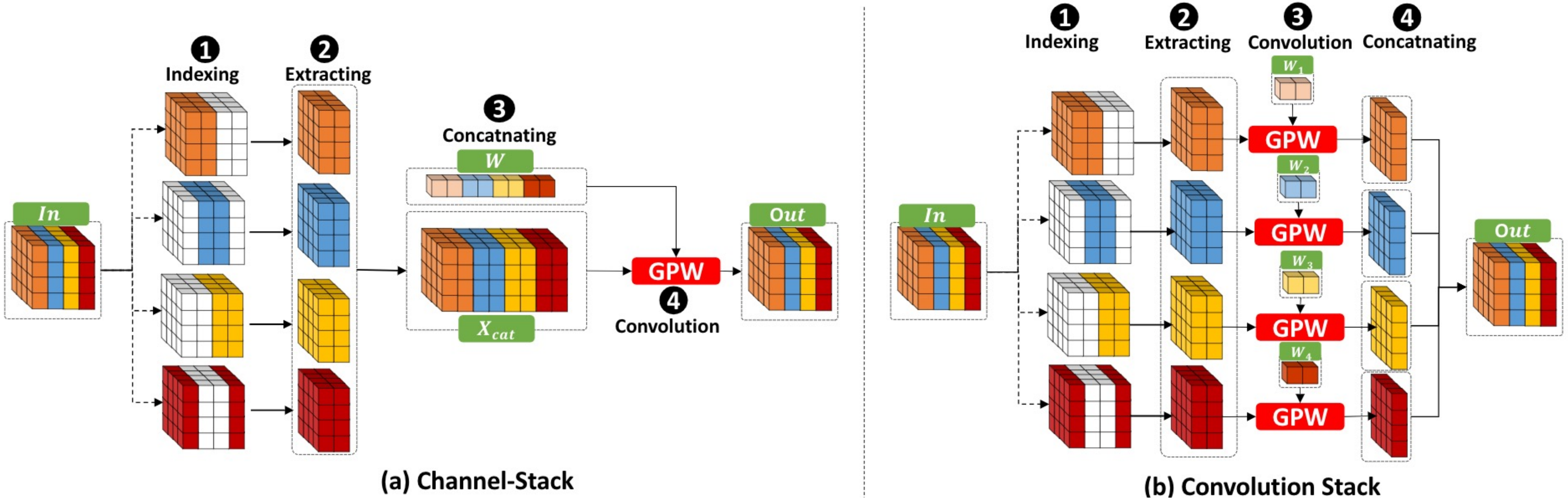
TABLE I: Comparison among SCC, PW, and GPW.

Convolution	**FLOPs**	**Params.**	**Acc.**
PW†	High	High	High
GPW*	Low	Low	Low
SCC	**Low**	**Low**	**High**

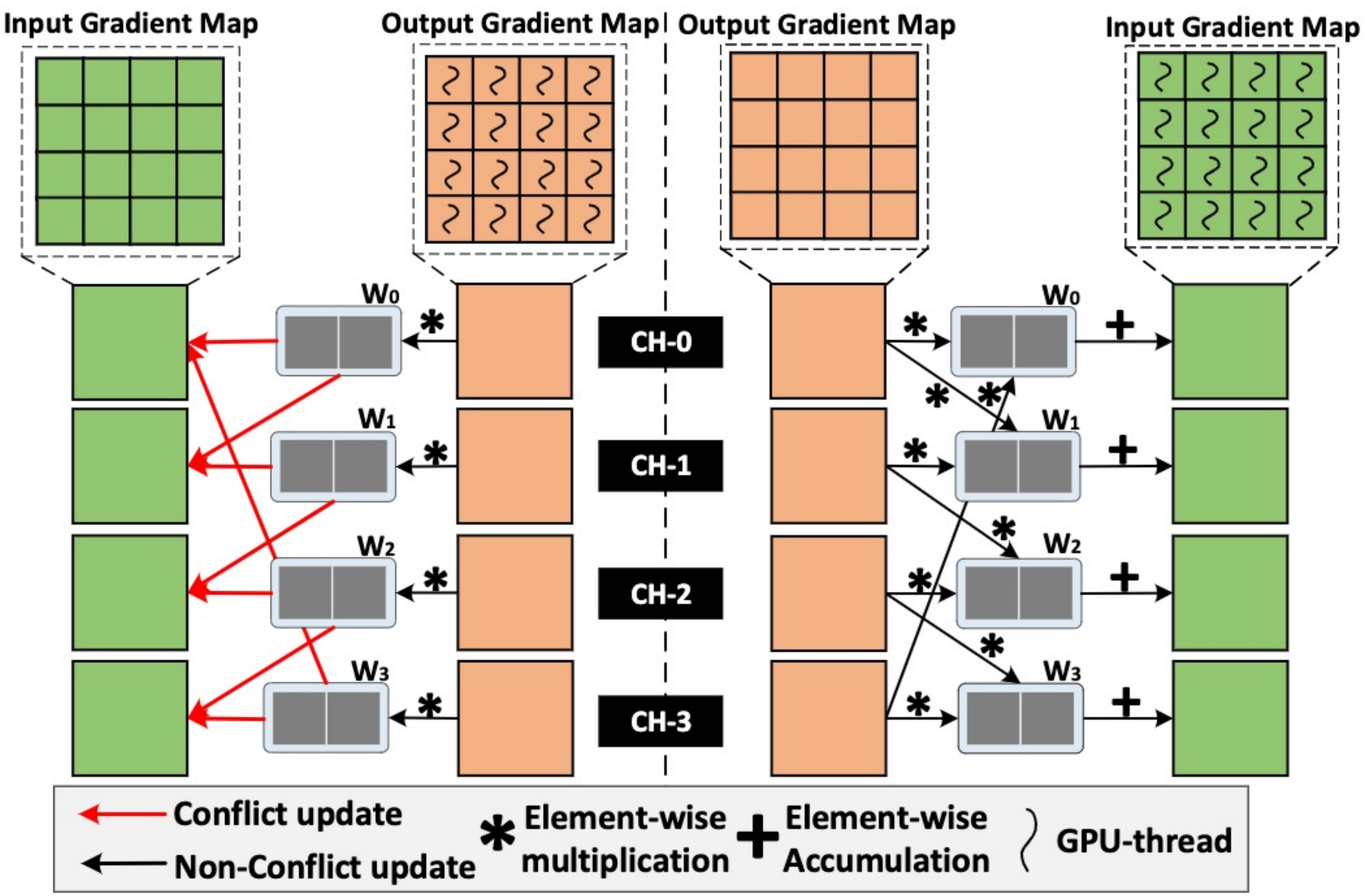
†: PW can be seen as SCC with 1 group with 100% channel overlapping of adjacent filters.

*: GPW can be seen as SCC with m groups with 0% channel overlapping of adjacent filters. m is a parameter that can be determined by users.

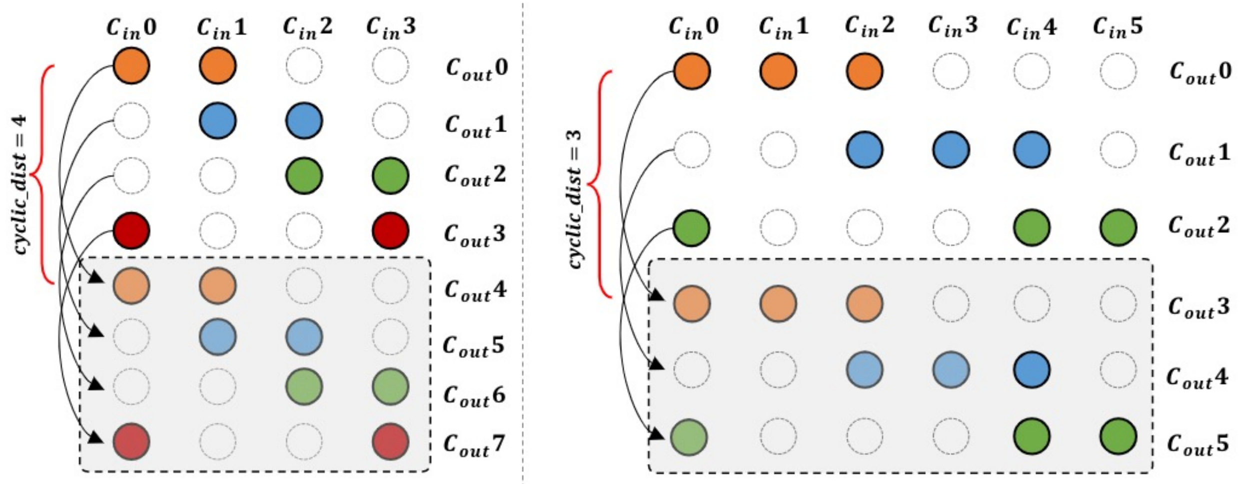
Pytorch-based SCC Design



Optimized CUDA SCC Kernel



Channel-cyclic Optimizations



(a) $C_{in} = 4, \mathrm{cg} = 2, \mathrm{co} = 50\%$ (b) $C_{in} = 6, \mathrm{cg} = 2, \mathrm{co} = 33\%$

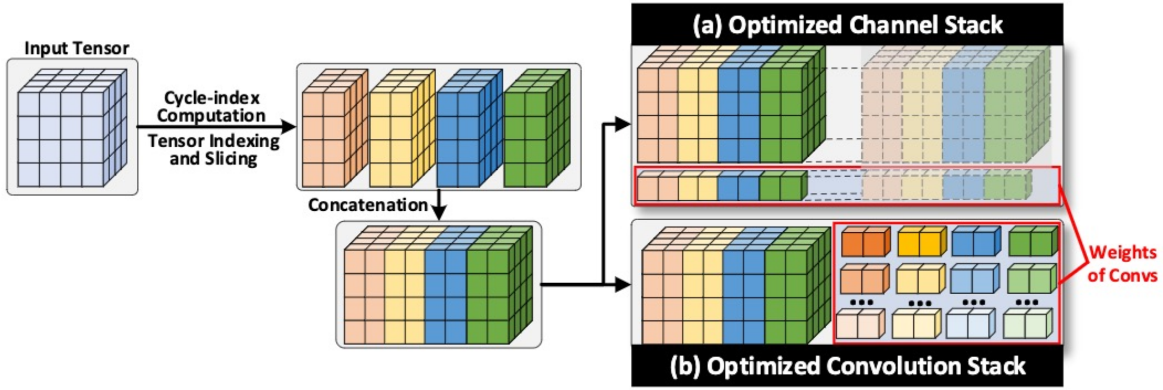


Fig. 6: Case study of cyclic-channel optimization on Pytorch-based implementations for $C_{in} = 4$, $cg = 2$, $co = 50\%$.

Algorithm 1: Compute channel indexes of one cycle.

```
channel_map = {} ;
group_width = input_channel//num_groups;
start, end = 0, group_width;
start_v, end_v = start, end;
cyclic_dist = 0;
for oid = 0; oid < output_channel; oid++ do
    item = (start, end);
    if item not in channel_map then
        channel_map.add(item);
        cyclic_dist++;
    end
    else
        break;
    end
    start_v = end_v − int(overlap ∗ group_width);
    end_v = start_v + group_width;
    start = start_v % input_channel;
    end = end_v % input_channel;
end
```

Algorithm 2: DSXplore Channel-cyclic Optimization

```
thread_id = blockIdx.x ∗ blockDim.x + threadIdx.x;
opt_channel_id = get_output_channel(thread_id);
idx = output_channel_id % cyclic_dist;
start, end = channel_map[idx];
```

Experiment: **Accuracy**

TABLE II: Accuracy comparison of CNNs on CIFAR-10.

Model	**Implementation**	**MFPLOS**	**Param. (M)**	**Acc. (%)**
VGG16	Origin	314.67	14.73M	92.64
	DSXplore	94.39	0.85M	92.60
VGG19	Origin	399.75	20.02M	93.88
	DSXplore	114.22	1.16M	92.71
MobileNet	Origin	67.31	3.19M	92.05
	DSXplore	45.29	1.63M	92.02
ResNet18	Origin	581.63	11.17M	95.75
	DSXplore	298.63	0.54M	94.81
ResNet50	Origin	2036.01	23.52M	95.82
	DSXplore	1469.64	12.81M	95.67

TABLE III: Accuracy comparison (**ImageNet**) for ResNet50.

Network	MFLOPs	Param.	Acc.(%)
Origin	4130	23.67M	76.56
DSXplore	**2550**	**14.34M**	**75.91**

Experiment: Accuracy (Cont'd)

TABLE IV: Comparison of different settings on MobileNet.

Network	MFLOPs	Param.	Acc.(%)
Baseline (DW+PW)	67.31	3.19M	92.05
DW+GPW-cg2	**45.29**	**1.63M**	**90.11**
DW+GPW-cg4	**34.28**	**0.84M**	**88.88**
DW+GPW-cg8	**28.78**	**0.45M**	**82.69**
DW+SCC-cg2-co25%	**45.29**	**1.63M**	**92.02**
DW+SCC-cg2-co50%	**45.29**	**1.63M**	**91.36**
DW+SCC-cg4-co25%	**34.28**	**0.84M**	**90.63**
DW+SCC-cg4-co50%	**34.28**	**0.84M**	**90.60**
DW+SCC-cg8-co25%	**28.78**	**0.45M**	**88.92**
DW+SCC-cg8-co50%	**28.78**	**0.45M**	**89.23**

Experiment: **Performance**

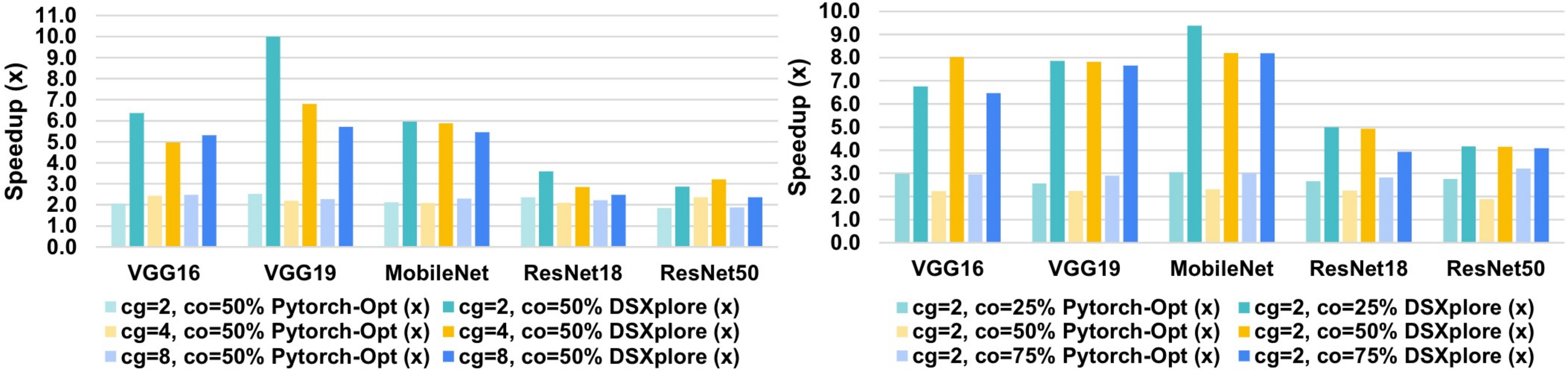


Fig. 7: Runtime performance comparison on CIFAR10. Note that speedup is normalized *w.r.t.* **Pytorch-Base** Implementation.

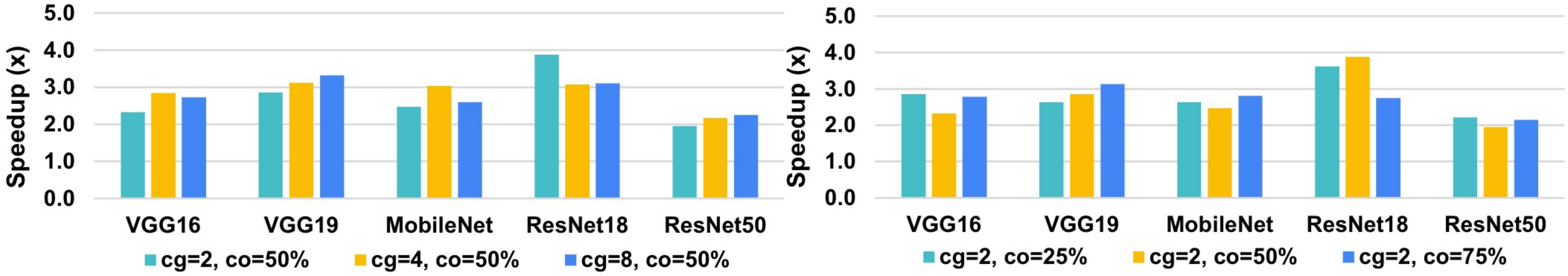


Fig. 8: Runtime performance comparison on ImageNet. Note that speedup is normalized *w.r.t.* **Pytorch-Opt** Implementation.

Experiment: **Additional Studies (cont'd)**

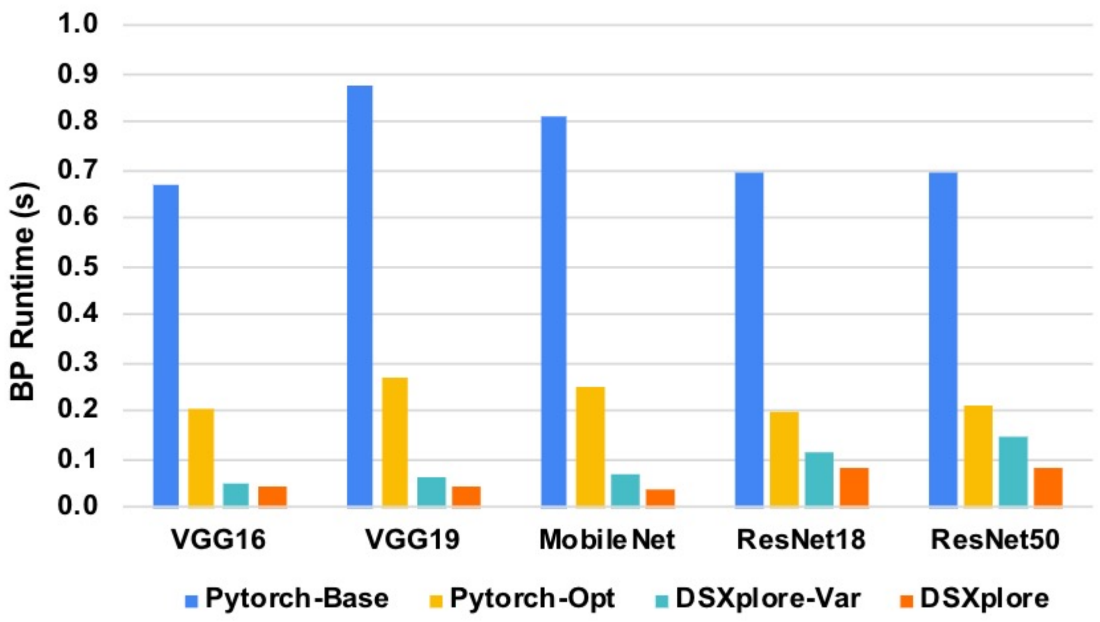


Fig. 9: Back-propagation optimization.

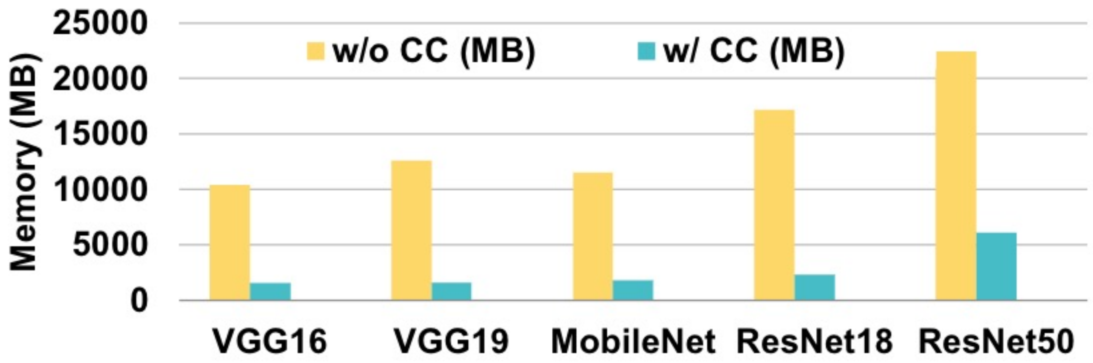


Fig. 10: Channel-cyclic optimization.

Experiment: **Additional Studies (con'd)**

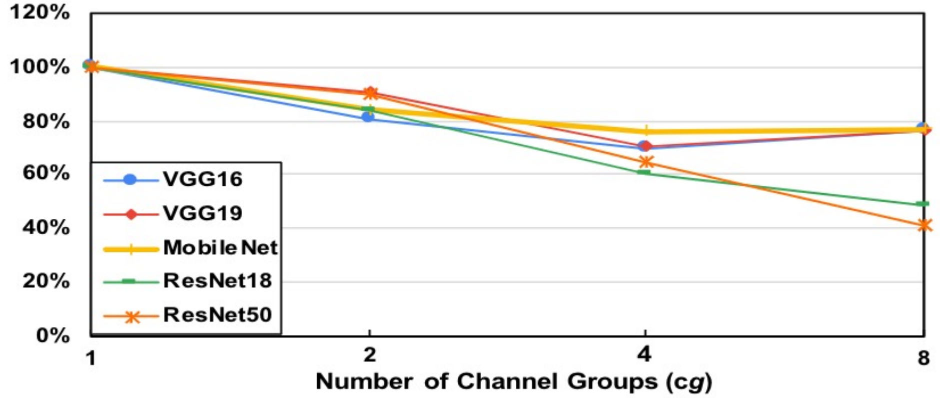


Fig. 11: The performance impact of the number of groups (cg). Note that we set $co = 50\%$ and the runtime is normalized *w.r.t* the performance at $cg = 1$.

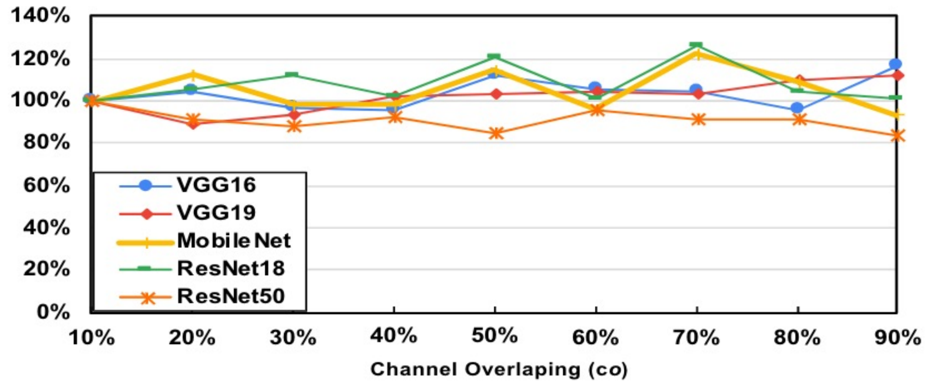


Fig. 12: The performance impact of the input-channel overlapping ratio (co). Note that we set $cg = 2$ and the runtime is normalized *w.r.t* the performance at $co = 10\%$.

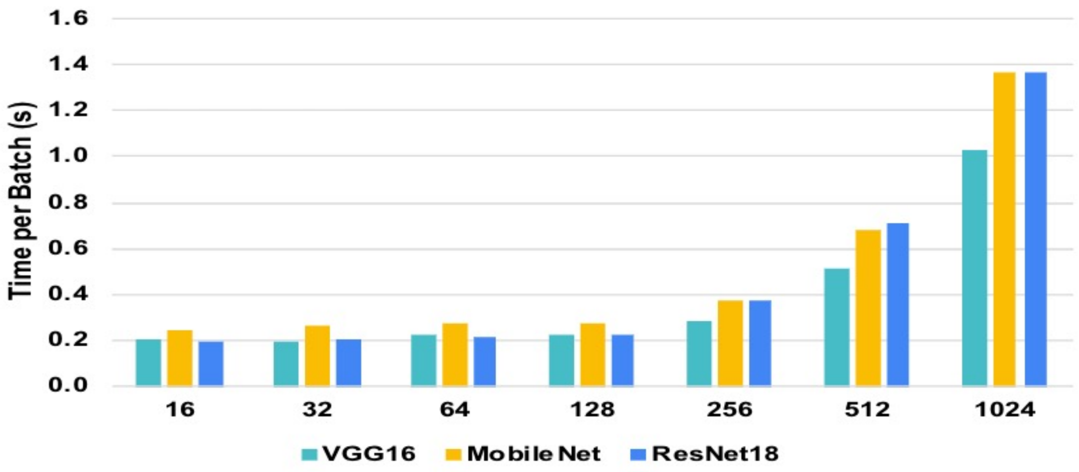


Fig. 13: Impact of batch size on training performance.

Thank You

Q & A

[Github] https://github.com/YukeWang96/DSXplore.git