ACCELERATE DNN INFERENCE BY INTER-OPREATOR PARALLELISM

Anonymous authors
Paper under double-blind review

ABSTRACT

High utilization is key to achieve high efficiency for deep neural networks. Existing deep learning frameworks has focused on improving the performance of individual operators but ignored the parallelization between operators. This leads to low device utilization especially for complex deep neural networks (DNNs) with many small operations such as Inception and NASNet. To make complex DNNs more efficient, we need to execute parallely. However, naive greedy schedule leads to much resource contention and do not yield best performance. In this work, we propose *D*eep *O*ptimal *S*cheduling (DOS), a general dynamic programming algorithm to find optimal scheduling to improve utilization via parallel execution. Specifically, DOS optimizes the execution for given hardware and inference settings. Our experiments demonstrate that DOS consistently outperform existing deep learning library by 1.2 to 1.4 $\times$ on widely used complex DNNs.

1 INTRODUCTION

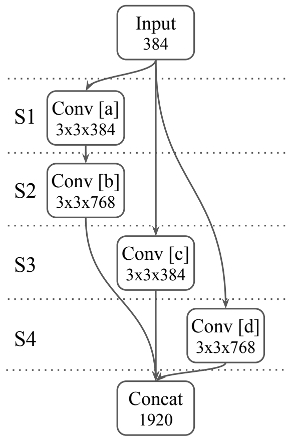
Deep neural networks (DNNs) have achieves state-of-the-art predictive performance across many tasks including computer vision (Krizhevsky et al., 2012; He et al., 2016), machine translation (Sutskever et al., 2014; Devlin et al., 2018) and game playing (Mnih et al., 2013; Silver et al., 2016). The success has resulted in the growth of computational requirements in today's DNN architectures.

To mitigate the increasing computational requirements, it is standard to parallelize computation in a DNN onto many-core accelerators, such as GPUs and TPUs. Existing deep learning frameworks (Abadi et al., 2016; Paszke et al., 2017) exploit *intra-operator parallelism* that parallelize all arithmetic operations within a *single* DNN operator (e.g., matrix multiplication) using vendor-provided libraries such as cuDNN and cuBLAS. Different operators within a DNN architecture are performed sequentially on the hardware device.

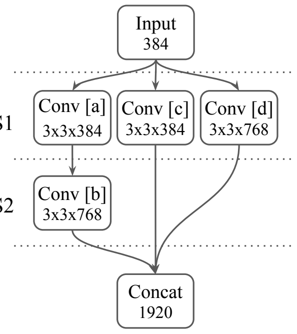
However, as the computational power of many-core devices consistently increases, intra-operator parallelism cannot provide sufficient parallelization opportunities to fully utilize all computation resources provided by a hardware device, resulting in suboptimal runtime performance. For example, for the DNN architecture in Fig. 1(a), the four convolutions [a], [b], [c], and [d] are executed sequentially in existing frameworks. Due to the limited parallelism in each convolution, the sequential schedule results in low GPU utilization (i.e., 24.6% on average).

Recent work explores *inter-operator parallelism* by using different heuristics to execute multiple DNN operators in parallel. For example, MetaFlow (Jia et al., 2019) fuse multiple operators matching a specific pattern into a larger operator to increase operator granularity. As another example, Tang et al. (2018) proposes a *greedy* strategy that directly execute all available DNN operators on a hardware device to maximize resource utilization. These approaches apply different heuristics to optimize local parallelization across a few DNN operators, which does not lead to a globally optimal schedule for the entire DNN architecture. For example, for the input DNN in Fig. 1(a), Fig. 1(b) shows the greedy schedule, which performs convolutions [a], [c], and [d] in parallel in a single stage (S1), and runs convolution [b] in a subsequent stage (S2) upon the completion of (S1). The greedy schedule is suboptimal as S2 does not include sufficient workload.

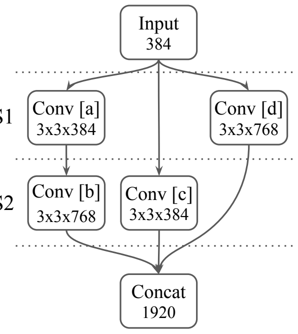
Discovering optimized schedules to parallelize a DNN model on a specific hardware device is challenging, as an optimal schedule depends on both thee DNN model settings and the hardware



(a) Sequential Schedule
Latency 0.499 ms
Device Utilization 24.6%



(b) Greedy Schedule
Latency 0.414 ms
Device Utilization 29.6%



(c) Our Schedule
Latency 0.356 ms
Device Utilization 34.4%

Figure 1: Different schedules for parallelizing a DNN architecture on a V100 GPU. DNN operators scheduled to run in parallel are placed at the same height. Both sequential and greedy schedules result in resource under utilization and suboptimal runtime performance. Balanced schedule yields the best latency and device utilization.

architecture. For example, our evaluation shows that changing the batch sizes of a DNN model or using different hardware devices can both result in different optimal schedule for a DNN model.

In this work, we propose *Deep Optimal Scheduling* (DOS), which accelerates DNN computation by combining intra- and inter-operator parallelism. To discover an efficient *schedule* to parallelize an input DNN model on a specific hardware device, DOS employs a cost model to estimate the runtime performance of different schedules based on profiling of representative executions. DOS uses a dynamic programming based search algorithm to explore the space of potential schedules, and discovers a *globally optimal* schedule under the cost model.

We evaluate DOS on five real-world DNN models, and show that DOS outperforms existing deep learning frameworks by increasing the end-to-end inference time by 1.2 to 1.4$\times$. We also validate DOS under different hardware and inference settings and experiments show that DOS can find specialized schedule for different inference settings and devices. Experiments demonstrate that DOS consistently outperform existing implementations.

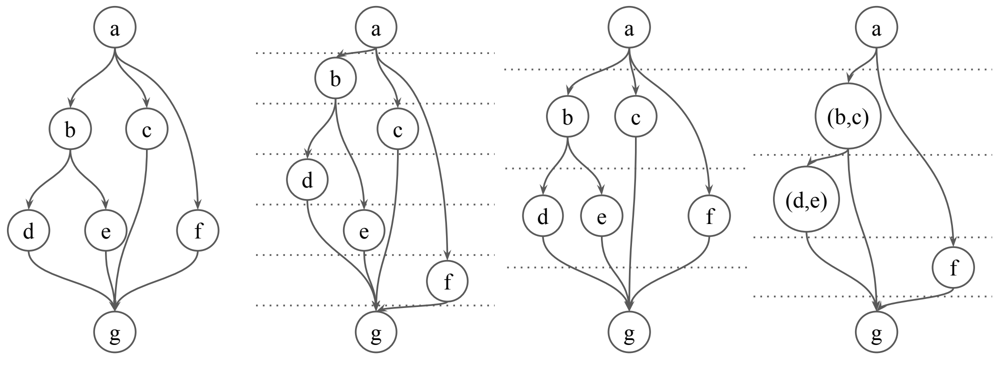
To summarize, our contributions are:

- We point out the long ignored utilization issues in DNN computing, especially for DNNs with complex topology.
- We propose a novel schedule algorithm (DOS) to find the best parallel execution for complex DNNs and show that this technique can generally and consistently boost the inference.
- We apply DOS to different devices and settings and show that the different platforms actually requires different schedule. Our searched schedule consistently outperforms existing deep learning libraries.

2 Background and Related work

2.1 Architectures of Deep Neural Networks

With bloom of deep learning, a series of studies have been made on how to design efficient DNN architectures. In early studies, sequential structures are more widely used because they are easier to design and deployment (*e.g.* AlexNet (Krizhevsky et al., 2012), VGG (Simonyan & Zisserman, 2014)) and ResNet (He et al., 2016)). Moreover, GPU memory is very limited at that time, even it is observed that DNNs with multi-branch have better performance under same FLOPs (Szegedy et al., 2015; 2016). These models are less preferred because large memory consumption in training.



(a) Computation Graph (b) Sequential Execution (c) Concurrent Execution (d) Merge Kernel

Figure 2: Illustration of two methods of parallel kernel execution

Recently, using machine learning techniques to automatically design neural network architectures has drawn increasingly interests (Zoph & Le, 2016), In neural architecture search (NAS), again complex DNNs appear superior performance (Zoph et al., 2018; Real et al., 2019; Xie et al., 2019). However, in production simple sequential models are more preferred (Tan et al., 2019; Cai et al., 2019) because complex DNNs suffer from low device utilization and the theoretical improvement cannot be translated into real speedup. Our work aims to bridge the gap and justify the efficiency of complex DNNs.

2.2 Related Work

Intra-operator parallelism. Existing deep learning frameworks (Abadi et al., 2016; Paszke et al., 2017) generally exploit intra-operator parallelism that parallelize all arithmetic operations within a single DNN operator (e.g., matrix multiplication), and performs different DNN operators sequentially on the hardware device. Intra-operator parallelism cannot provide sufficient parallelizable computation for today's hardware devices with increasing computational power, and therefore results in under utilized computation resources.

Inter-operator parallelism. Recent work has proposed different approaches to exploit inter-operator parallelism. For example, (Tang et al., 2018) proposes a greedy strategy that directly execute all available DNN operators on a hardware device to maximize resource utilization. TensorRT and TVM supports rule-based graph optimizations that transforms the computation graph of a DNN architecture, such as merging normalization and activation into linear operators (Abadi et al., 2015; Paszke et al., 2017; Chen et al., 2018). Though these techniques can be also applied to complex DNNs, they do not fundamentally solve the low utilization problem.

The work most related to our target is relaxed graph substitution (Jia et al., 2019) which explores functional preserving substitutions to transform a computational graph to. By merging kernels with same input and type of operations, it can be used to improve the utilization and redundancy of complex DNNs. However, such transformations are based on hand craft rules and restricted to specific layers, thus not general enough.

3 Problem Definition

Computation Graph. Deep neural network is usually represented as a computation graph $G = (S, E)$, where S is the set of operations and E is the set of edges. A computation graph is a directed acyclic graph (DAG). Each kernel in the graph represents an operation such as convolution, normalization, addition and *etc*. Each edge (u, v) in the graph bridges the output of its source kernel u and the input of its target kernel v.

Stage. In inter-operator parallel execution, multiple kernels will be launched at the same time. We name the group of concurrently launched kernels as *stage*. For example, Fig.2c contains 2 two stages: one consists of kernel b and c and the other includes kernel d, e and f, with extra input stage (kernel a) and output stage (kernel b).

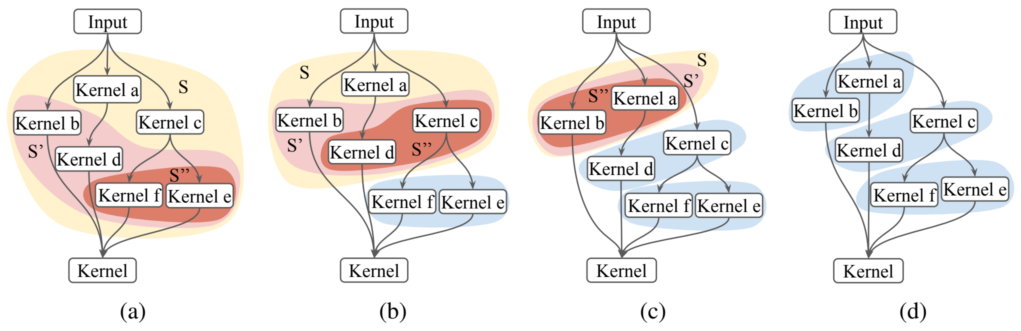


Figure 3: How DOS finds a specific schedule on given DAG. Set S, S' and S'' are marked in the figures and the blue areas are the stages. In (a), S' contains kernels *b, d, f, e* and DOS will enumerate all its non-empty subset as the last stage. When DOS chooses subset *f, e* as a stage as shown in Fig. 3a, the problem becomes to find the best schedule for kernel *a, b, c, d* as shown in Fig. 3b. Repeat the process then DOS will get the final schedule $\{a, b\}, \{c, d\}, \{e, f\}$ as shown in Fig 3d.

Parallel Strategy. There are two possible parallel strategies: concurrent execution (Fig.2c) and operation merging (Fig.2d). *Concurrent execution* will run all operations in the stage simultaneously. *Operation merging* combines the operations of the same type into a bigger one during scheduling and executes it during inference.

There are tremendous feasible schedules for a given computation graph and it is too costly to manually explore. Among all the possible schedules, the most intuitive one is to greedily assign all currently available kernels into a stage. However, as shown in Fig. 1, schedule without global information leads to low utilization and thus sub-optimal. Moreover, there are various hardware platforms and inference settings, all these, would affect the scheduling for a computation graph. This brings extra challenge to find a good schedule.

Schedule. We define the *schedule* of a computation graph $G = (S, E)$ as following

$$L = (S_1, S_2, \dots, S_k, g_1, g_2, \dots, g_k)$$
$$\forall u \in S_i, v \in S_j, (u, v) \in E \ (i < j).$$

where $S_1, S_2, \dots, S_k$ are *stages* of the schedule and g_i is the parallel strategy for S_i. The computation graph G under schedule L would be executed stage by stage. Let c be a cost function defined on computation graph G and schedule L. We aim to find a schedule L^* to minimize the cost function for given computation graph G, i.e $c(G, L*) = \min_L c(G, L)$. In this work, the cost function $c(G, L)$ is define as the latency of execution.

4 METHODS

In this section, we present our method to find schedule for given computation graph. In Sec. 4.1, we introduce the cost model. In Sec. 4.2, we propose a dynamic programming method to find the optimal schedule under our cost model.

4.1 COST MODEL

We introduce the cost model c to evaluate the runtime performance. For given stage S_i and corresponding parallel strategy, we take the measured latency as the cost model. We here make a mild assumption the latency of a stage is not affected by the previous. This assumption holds for most DSPs. Therefore, we can add up all the stage latency predicted by cost model as an approximation of the graph latency under schedule L, for a given computation graph G and schedule L.

4.2 DEEP OPTIMAL SCHEDULER

To address the problem, we propose DOS to schedule an optimal schedule under given cost model, hardware and inference settings.

DOS starts with state S, which is the set of operators in computation G. For a given state S, let S' be the set of kernels in S such that none of the successors of nodes in S' are in S. For all the schedules

Algorithm 1 Deep Optimal Scheduling

```
    Input: Computation Graph: $G$    (let $S$ be the node set of $G$);
    Output: A schedule for $G$.
 1: $\text{cost}[S'] = \infty,\ \text{choice}[S'] = \emptyset$ where $S' \subset S$ and $S' \neq \emptyset$    ▷ Initialize the cost to be large.
 2: $\text{cost}[\emptyset] = 0,\ \text{choice}[\emptyset] = \emptyset$
 3: procedure DOS($S$)
 4:     if $\text{cost}[S] \neq \infty$ then
 5:         return $\text{cost}[S]$    ▷ We have already got the result for set $S$
 6:     end if
 7:     for $S'' \subset S'$ where $S' = \{s \mid \forall s' \in \text{successors}(s)\ s' \notin S\}$ do
 8:         $T = \text{DOS}(S - S'') + min(\text{latency}_{\text{merge}}(S''), \text{latency}_{\text{concurrent}}(S''))$
 9:         if $T < \text{cost}[S]$ then    ▷ A better schedule is found
10:             $\text{cost}[S] = T$    ▷ Update best latency.
11:             $\text{choice}[S] = S''$    ▷ Record the choice.
12:         end if
13:     end for
14:     return $\text{cost}[S]$
15: end procedure
```

of S, the nodes of the last stage must be a subset of S'. Then we enumerate all the non-empty subsets S'' of S'. Once we choose S'' to be the last stage of S, the problem to find the optimal schedule of S will be reduced to find the optimal schedule of $S - S''$, which is a sub-problem of the original one. By enumerating all the subset S'' of S', we can find the optimal schedule for S. For each stage S_i, the parallel strategy with lower latency will be chosen.

DOS uses `cost[`S`]` to record the sum of latency of all the stages in the optimal schedule for S. and uses `choice[`S`]` to record the last stage in the optimal schedule. $\texttt{latency}_{\text{merge}}$ and $\texttt{latency}_{\text{concurrent}}$ are the cost model used by DOS to get the latency. We can get the final schedule for the computation graph from the `choice` array.

When the graph has a large number of nodes, we can split the graph into multiple connected sub-graphs to apply DOS algorithm separately. Most networks such as Inception V3 and RandWire naturally split the graph into blocks so we can use the blocks as the subgraphs to apply DOS algorithm.

5 Experiment

DOS is frame-agnostic algorithm for arbitrary computation graphs. In our experiment, the runtime is built upon existing deep learning libraries cuDNN (Chetlur et al., 2014). DOS accepts user defined cost function to minimize. We adpat the measured latency as the optimization target.

5.1 Inference Speedup

In this section, we show that our proposed method can effectively speed up the inference of existing models, even for the ones with highly complicated topology. We benchmarked our methods on four representative convolution neural networks, Inception V3 (Szegedy et al., 2016), Randwire (Xie et al., 2019), NasNet-A (Zoph et al., 2018) and SqueezeNet (Iandola et al., 2016). For each network, we measure the latency of sequential schedule, greedy schedule, optimized schedule, and the latency on existing frameworks including TVM[*] (Chen et al., 2018), Tensorflow (Abadi et al., 2016) and TensorRT (TensorRT). We measured the inference time on a single NVIDIA Tesla V100 GPU. The results are shown in Figure 4. DOS outperforms the sequential schedule and greedy schedule baselines by up to 1.4x, even surpassing TensorRT and TVM implementation on all of the four networks. DOS enjoys great stability, it is the optimization method that consistently outperforms the sequential baseline.

[*]The backend of TVM is cuDNN.

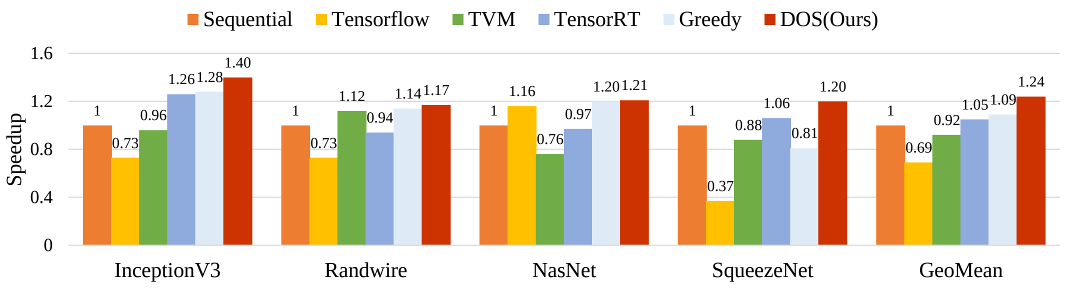


Figure 4: End-to-end inference comparison among sequential schedule, Tensorflow, TVM, TensorRT, Greedy schedule and DOS optimization. Sequential schedule, greedy schedule and DOS share the same runtime and the only difference among them is the schedule.

5.2 Specialize for Different Hardware Accelerators

We explore DOS optimization on different devices. Different devices have different hardware configurations (*e.g.*, cache size, frequency, number of compute cores, etc). Therefore, we may need different scheduling schemes to make full use of the hardware resources. We use DOS to optimize the schedule of Inception V3 on three devices: NVIDIA Titan XP, Tesla V100 and GeForce RTX 2080Ti. We applied DOS on each device respectively to generate a specific scheduling. The results are shown in Table 1. For each of the devices, the scheduling searched on the current device achieves the highest inference speed and hardware utilization. For example, on NVIDIA RTX 2080Ti GPU, the specialized scheduling can achieve 6% and 8% faster speed compared to the ones transferred from Titan XP and Tesla V100.

We provide the details of DOS's optimized schedules for NVIDIA Titan XP and GeForce RTX 2080Ti in Fig. 5. The operations on the same height belong to a stage. The two schedules show different patterns: on Titan XP, DOS prefers to place similar operators within the same stage, *e.g.*, convolutions with similar kernel sizes are scheduled in the same stage, while average pool is placed in a separate stage. While on 2080Ti with larger computation capacity, DOS learns to move convolutions from late stages into earlier ones to increase the hardware utilization.

Devices	TitanXP	V100	2080Ti
TitanXP	**6.01 ms**	6.27 ms	6.27 ms
V100	4.86 ms	**4.65 ms**	5.08 ms
2080Ti	4.33 ms	4.43 ms	**4.09** ms

Table 1: Specialized Schedules for Devices

Batchsize	1	8	16
1	**4.66 ms**	4.73 ms	5.09 ms
8	10.98 ms	**10.55 ms**	10.96 ms
16	17.43 ms	16.94 ms	**16.56 ms**

Table 2: Specialized Schedules for Batch Sizes

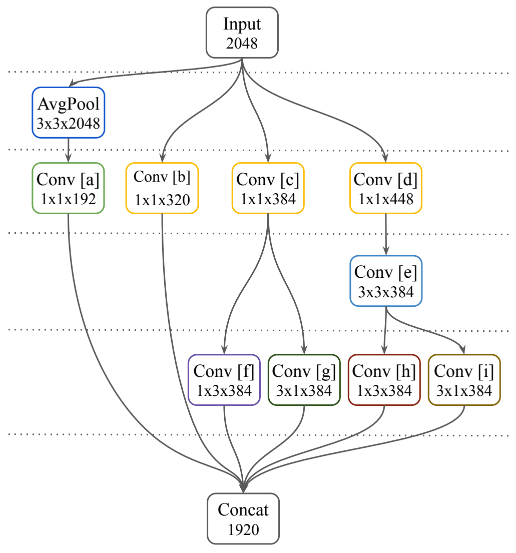
Table 3: Specialized schedules for different devices and batch sizes. Each row represents the real setting to execute the network and each column represents the setting DOS optimized for.

5.3 Specialize for different batch size

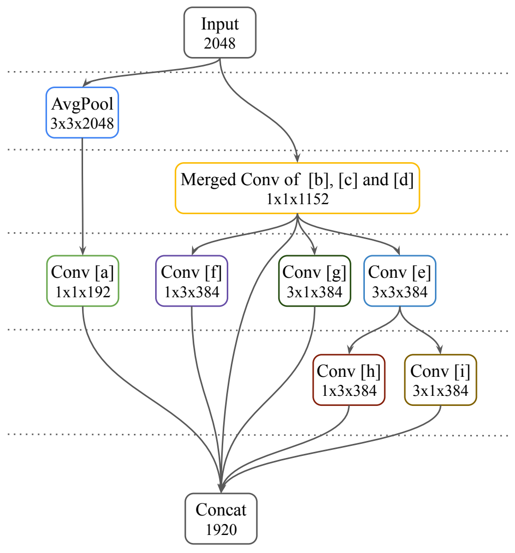
Apart from different hardware, in real-life cases, we also need to handle different batch sizes during inference. For example, for real-time recognition in autonomous driving, we usually use a batch size of 1 to reduce latency; while on cloud computing, a larger batch size is preferred to increase the throughput and hardware utilization.

DOS can also optimize the computation graph when batch size is greater than 1. We optimize the Inception V3 with batch size 1, 2, 4, 8 on V100. Fig. 6 shows the result. DOS outperforms the other five baselines consistently and can still have 1.2 speed up comparing to sequential schedule when batch size equals to 8. The speedup ratio generally decreases as the batch size increases, as sequential schedule has a larger parallelism with increased batch size.

The computational requirements changes proportional to the batch size. So the same computation graph with different batch sizes need different specialized schedules. We optimize Inception V3 with batch size 1, 8, 16, and find that DOS can also find the specialized schedule for each batch size. The result is reported in Table 2. The schedule optimized for specific batch size outperforms the schedule optimized for other batch sizes when the computation graph is executed on that specific batch size.



(a) Schedule for TitanXP



(b) Schedule for 2080Ti

Figure 5: Specialized schedules of a InceptionV3 Block for TitanXP and 2080Ti. From the schedule on TitanXP, we can find that DOS prefers to place similar operators in a stage: put avgpool in a separate stage and convolutions with similar kernel size are scheduled in one stage. When the computation capability increases, DOS prefers to move convolutions in late stages into early stages to increase device utilization.

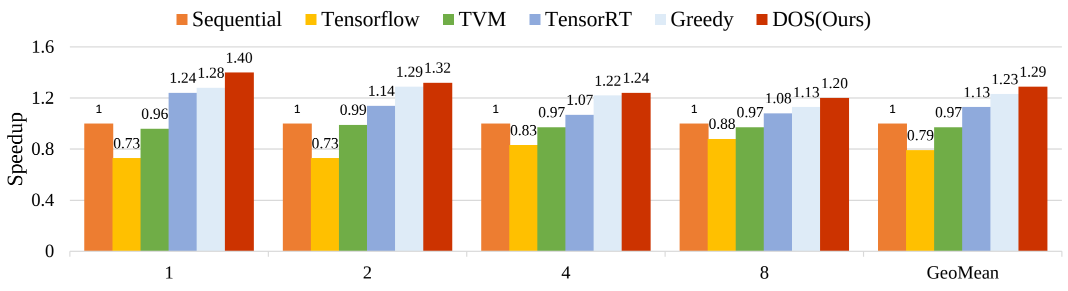


Figure 6: Speedup of Inception V3 with different batch sizes

5.4 Search Time Comparision

We report the optimization time for different networks. The optimized graph of SqueezeNet is the same as Metaflow. And when we only do the operator merging in DOS, we can also get the same graph as Metaflow in about 5 seconds. All the graphs can be optimized within 8 minutes.

Models	SqueezeNet	InceptionV3	NasNet	RandWire
Metaflow($\alpha = 1.01$)	4 secs	24 secs	40mins	$>$ 1 day
Metaflow($\alpha = \infty$)	5 secs	60 secs	4 hours	$>$ 1 day
DOS	4 secs	10 secs	3 mins	8 mins

Table 4: Search Time Comparison between DOS and Metaflow. DOS can have reasonable search time even for DNNs with complex topological structure.

5.5 Block Performance

We also compare the performance of each block in InceptionV3 under sequential schedule and DOS optimized schedule in Fig. 8. The figure shows that the optimized schedule is consistently faster than sequential schedule, which leads to an end-to-end 1.4x speedup.

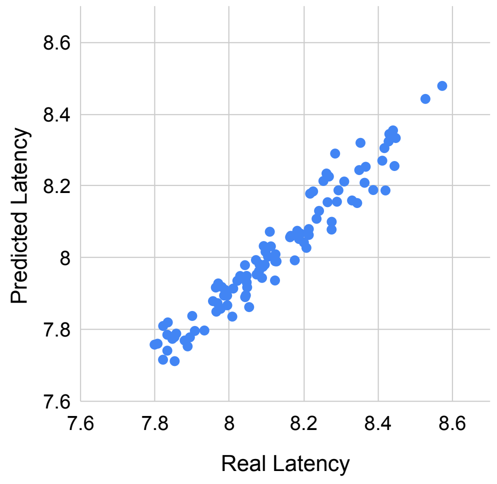


Figure 7: Assumption Validation

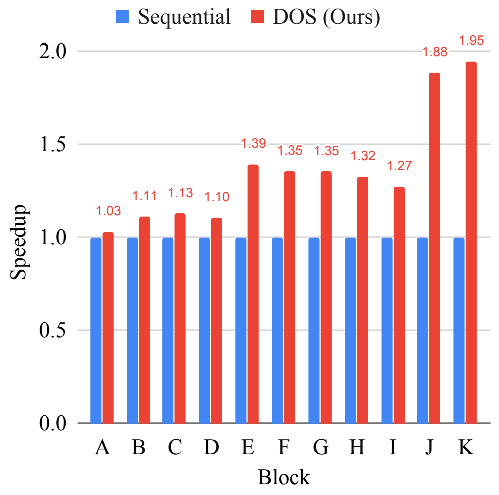


Figure 8: Speedup of each block in InceptionV3

5.6 Performance comparison between DOS and TensorRT

Table 5 compares the different metrics between TensorRT and DOS. For InceptionV3 and SqueezeNet, DOS merged kernels, which reduce the memory access and number of kernel launches. Although DOS's optmized schedule for SqueezeNet merge operations and increase the FLOPs, the over all latency reduced.

Models	Latency (ms)		Memory (MB)		Kernels		FLOPs		Utilization	
	TRT	DOS	TRT	DOS	TRT	DOS	TRT	DOS	TRT	DOS
IncetptionV3	5.11	**4.61**	82	**68**	119	**103**	5.50	5.50	1.08	**1.19**
Randwire	8.91	**7.15**	576	576	406	406	7.67	7.67	0.86	**1.07**
NasNet	23.58	**18.99**	1187	1187	965	965	22.00	22.00	0.93	**1.16**
SqueezeNet	0.84	**0.74**	57	**48**	50	**38**	**1.16**	1.25	1.38	**1.69**

Table 5: The comparison of latency (ms), memory access (MB), number of kernel launches, flops (GFLOPs) and the device utilization (TFLOPs / sec) between TensorRT (TRT in the table) and DOS. Here the device utilization is calculated as the effective FLOPs per second.

5.7 Assumption Validation

In Sec. 4.1, we assume that, for given schedule, the latency of the whole computation graph equals to the sum of latency of each stage. We sample many random schedules and evaluate real latency of the whole computation graph and predict the latency by adding up latency of all stages. The result of the real latency and predicted latency is shown in Fig. 7. The point-wise relative order accuracy is 0.937, thus the assumption is reasonable.

6 Conclusion

With the increasing of computational power, sequential execution of DNNs can not provide sufficient parallelzation opportunities to fully utilize all the computation resources of hardware device. And recent work that explores inter-operators parallelism use heuristics to execute operators parallely, which do not utilize the global information of the networks and can be fall into sub-optimal optimization. In this work, we propose DOS, which combines intra- and inter-operator parallelism and utilize the global information of the networks. Experiments show that DOS can accelerate the performance of widely used DNN architectures from 1.2 to 1.4x.

References

Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. URL http://tensorflow.org/. Software available from tensorflow.org.

Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pp. 265–283, 2016.

Han Cai, Ligeng Zhu, and Song Han. ProxylessNAS: Direct neural architecture search on target task and hardware. In *International Conference on Learning Representations*, 2019. URL https://arxiv.org/pdf/1812.00332.pdf.

Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. {TVM}: An automated end-to-end optimizing compiler for deep learning. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, pp. 578–594, 2018.

Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cudnn: Efficient primitives for deep learning. *arXiv preprint arXiv:1410.0759*, 2014.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. *CoRR*, abs/1810.04805, 2018. URL http://arxiv.org/abs/1810.04805.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778, 2016.

Forrest N Iandola, Song Han, Matthew W Moskewicz, Khalid Ashraf, William J Dally, and Kurt Keutzer. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and¡ 0.5 mb model size. *arXiv preprint arXiv:1602.07360*, 2016.

Zhihao Jia, James Thomas, Todd Warszawski, Mingyu Gao, Matei Zaharia, and Alex Aiken. Optimizing dnn computation with relaxed graph substitutions. 2019.

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pp. 1097–1105, 2012.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. In *NIPS-W*, 2017.

Esteban Real, Alok Aggarwal, Yanping Huang, and Quoc V Le. Regularized evolution for image classifier architecture search. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pp. 4780–4789, 2019.

David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484, 2016.

Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, pp. 3104–3112, 2014.

Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 1–9, 2015.

Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 2818–2826, 2016.

Mingxing Tan, Bo Chen, Ruoming Pang, Vijay Vasudevan, Mark Sandler, Andrew Howard, and Quoc V Le. Mnasnet: Platform-aware neural architecture search for mobile. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 2820–2828, 2019.

Linpeng Tang, Yida Wang, Theodore L Willke, and Kai Li. *arXiv preprint arXiv:1807.09667*, 2018.

TensorRT. Nvidia tensorrt: Programmable inference accelerator. URL https://developer.nvidia.com/tensorrt.

Saining Xie, Alexander Kirillov, Ross Girshick, and Kaiming He. Exploring randomly wired neural networks for image recognition. *arXiv preprint arXiv:1904.01569*, 2019.

Barret Zoph and Quoc V Le. Neural architecture search with reinforcement learning. *arXiv preprint arXiv:1611.01578*, 2016.

Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V Le. Learning transferable architectures for scalable image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 8697–8710, 2018.