IOS: Inter-Operator Scheduler for CNN Acceleration

Yaoyao Ding[1,2] Ligeng Zhu[3] Zhihao Jia[4] Gennady Pekhimenko[1,2]
Song Han[3]

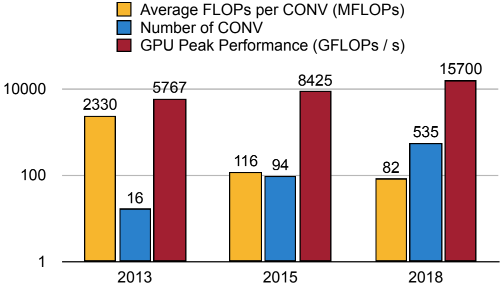
[1]University of Toronto [2]Vector Institute [3]Massachusetts Institute of Technology [4]Carnegie Mellon University

Presenter: Shiwei Zhang

Introduction

GPU Under-utilization in CNN

A recent trend in CNN design is to replace a single branch of convolutions with multiple branches of convolutions. As a result, the number of convolutions grows while the computation FLOPs in each convolution becomes smaller.



Existing Approaches

- **Intra-operator parallelism** (TVM) executes arithmetic operations within a single operator in parallel. However, the degree of parallelism within an operator is limited, especially when the `Conv` operations are becoming smaller.
- **Graph transformation** (MetaFlow, TASO) explores merging and substituting operators to enable more parallelism. However, the possible merging are limited to same type of operators.
- **Inter-operator scheduling** (Graphi, Rammer, Nimble) schedules some operators to run concurrently. However, they use simple heuristics and don't lead to global optimal.

IOS: Inter-Operator Scheduler

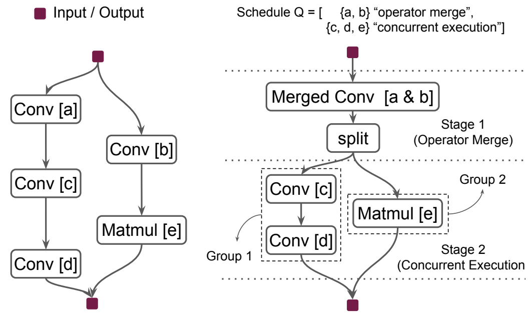
This paper introduces **IOS**, a novel **dynamic programming** algorithm to find a highly optimized schedule for **inter-operator parallelization**.

Problem Definition

Graph and Stage

A CNN is defined as a DAG $G = (V, E)$, where V is the set of operators, and E is the edge set representing dependencies.

The computation graph is partitioned into multiple **stages**. Stages are executed sequentially and the operators in the same stage are executed according to a certain **parallelization strategy**.



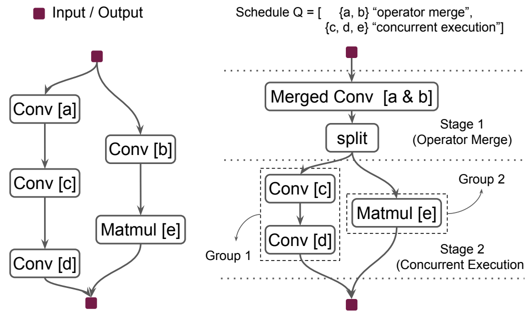
(1) Computation Graph

(2) A Feasible Schedule Q

Parallelization Strategy

Operator merge merges multiple operators of the **same type** together. For example, an 3x3 `Conv` can be merged with a 5x5 `Conv`, by padding and concatenating the kernels.

Under **concurrent execution**, the operators in the stage that have no dependencies are executed concurrently with multiple CUDA streams.



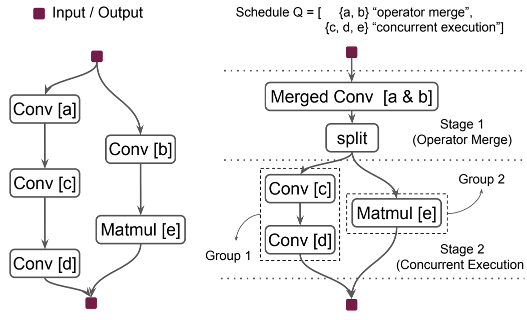
(1) Computation Graph

(2) A Feasible Schedule Q

Schedule

A Schedule $Q = \{(S_1, T_1), (S_2, T_2), \dots\}$ is an assignment of operators S_i to the i-th stage and the parallelization strategy T_i of the i-th stage.

IOS finds a schedule Q^* that minimizes a cost function c for a given graph G, i.e., $Q^* = \text{argmin}_Q c(G, Q)$. In this work, c is defined as the latency of running G following schedule Q.



(1) Computation Graph

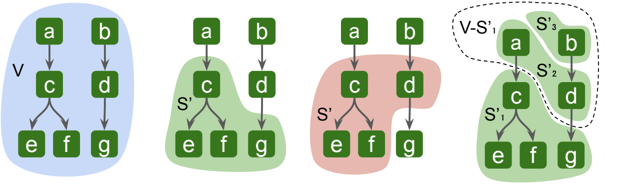
(2) A Feasible Schedule Q

Methods

Main Idea

For an **ending** S' of S, we have:

$$\text{cost}[S] = \min_{S'}(\text{cost}[S - S'] + \text{stage\_latency}[S'])$$



(1) Operators V

(2) S' is an ending of V

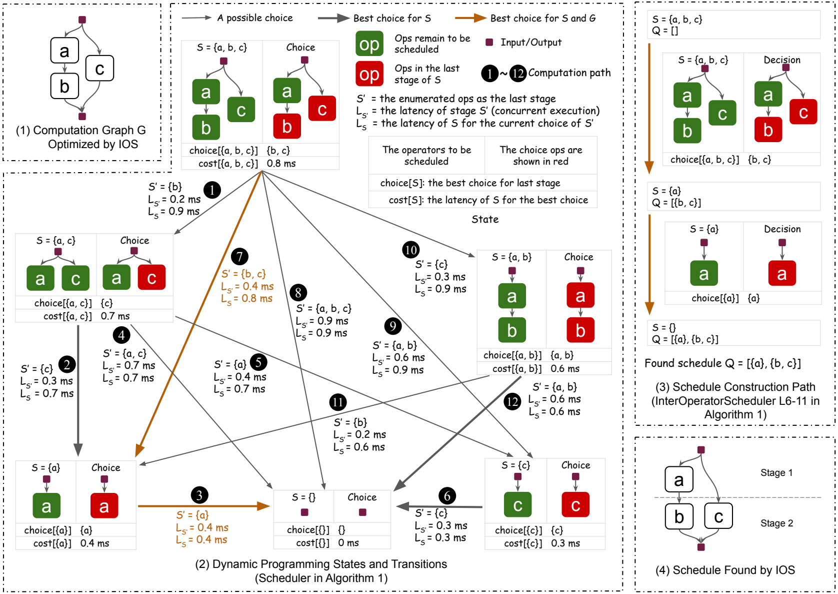
(3) S' is **not** an ending of V

(4) Partition graph by endings recursively

Core Function

```
function Scheduler($S$)
    if cost[$S$] $\neq \infty$ then
        return cost[$S$]
    for all ending $S'$ of $S$ satisfying pruning strategy $P$ do
        $L_{S'}, T_{S'}$ = GenerateStage($S'$)
        $L_S$ = Scheduler($S - S'$) $+ L_{S'}$
        if $L_S <$ cost[$S$] then
            cost[$S$] $= L_S$
            choice[$S$] $= (S', T_{S'})$
    return cost[$S$]
```

Example



A possible choice → Best choice for S → Best choice for S and G

🟩 op Ops remain to be scheduled

🟥 op Ops in the last stage of S

■ Input/Output

1 ~ 12 Computation path

S′ = the enumerated ops as the last stage

$L_{S'}$ = the latency of stage S′ (concurrent execution)

L_S = the latency of S for the current choice of S′

The operators to be scheduled	The choice ops are shown in red
choice[S]: the best choice for last stage	
cost[S]: the latency of S for the best choice	

State

a
b
c
S = {a, b, c}
Choice
choice[{a, b, c}] {b, c}
cost[{a, b, c}] 0.8 ms
1 S′ = {b} $L_{S'}$ = 0.2 ms L_S = 0.9 ms
S = {a, c}
choice[{a, c}] {c}
cost[{a, c}] 0.7 ms
7 S′ = {b, c} $L_{S'}$ = 0.4 ms L_S = 0.8 ms
8 S′ = {a, b, c} $L_{S'}$ = 0.9 ms L_S = 0.9 ms
10 S′ = {c} $L_{S'}$ = 0.3 ms L_S = 0.9 ms
S = {a, b}
choice[{a, b}] {a, b}
cost[{a, b}] 0.6 ms
2 S′ = {c} $L_{S'}$ = 0.3 ms L_S = 0.7 ms
4 S′ = {a, c} $L_{S'}$ = 0.7 ms L_S = 0.7 ms
5 S′ = {a} $L_{S'}$ = 0.4 ms L_S = 0.7 ms
9 S′ = {a, b} $L_{S'}$ = 0.6 ms L_S = 0.9 ms
11 S′ = {b} $L_{S'}$ = 0.2 ms L_S = 0.6 ms
12 S′ = {a, b} $L_{S'}$ = 0.6 ms L_S = 0.6 ms
S = {a}
choice[{a}] {a}
cost[{a}] 0.4 ms
3 S′ = {a} $L_{S'}$ = 0.4 ms L_S = 0.4 ms
S = {}
choice[{}] {}
cost[{}] 0 ms
6 S′ = {c} $L_{S'}$ = 0.3 ms L_S = 0.3 ms
S = {c}
choice[{c}] {c}
cost[{c}] 0.3 ms
S = {a, b, c} Q = []
Decision
choice[{a, b, c}] {b, c}
S = {a} Q = [{b, c}]
choice[{a}] {a}
S = {} Q = [{a}, {b, c}]
Stage 1
Stage 2

(1) Computation Graph G Optimized by IOS

(2) Dynamic Programming States and Transitions (Scheduler in Algorithm 1)

Found schedule Q = [{a}, {b, c}]

(3) Schedule Construction Path (InterOperatorScheduler L6-11 in Algorithm 1)

(4) Schedule Found by IOS

Time Complexity

Definition: d is the width of G, if we can find at most d operators in G such that there is no path connecting any two of them.

Theorem: The time complexity of IOS is $\mathcal{O}(\binom{n/d+2}{2}^d)$, which can be relaxed to $\mathcal{O}((n/d+1)^{2d})$, where n is the number of operators in G and d is its width.

Model	n	d	$\binom{n/d+2}{2}^d$	$\#(S, S')$	#Schedules
Inception V3	11	6	2.6×10^4	4.9×10^3	3.8×10^6
Randwire	33	8	3.7×10^9	1.2×10^6	9.2×10^{22}
NasNet	18	8	5.2×10^6	3.1×10^5	7.2×10^{12}
SqueezeNet	6	3	2.2×10^2	51	1.3×10^2

Pruning

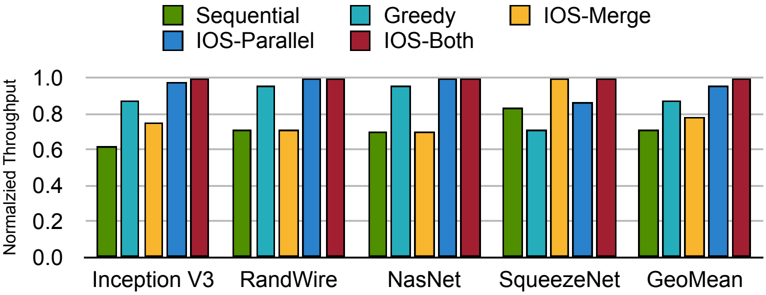
IOS without pruning can find the optimal strategy for the benchmarked graphs in 4 hours. To further reduce the search time, IOS introduces two parameters r and s. $P_{r,s}(S, S') =$ True if and only if S' has at most s groups and each group has at most r operators.

Evaluation

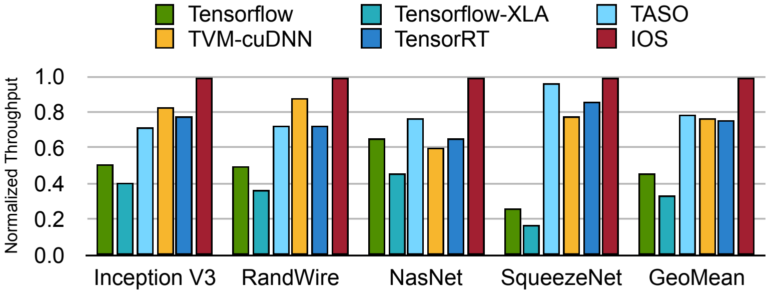
Experiments Setup

- Hardware: NVIDIA Tesla V100
- Execution Engine: A cuDNN-based C++ execution engine.
- Models: Inception V3, RandWire, NasNet-A, and SqueezeNet
- Baselines: TensorRT and TVM
- Pruning Parameters: $r = 3$ and $s = 8$

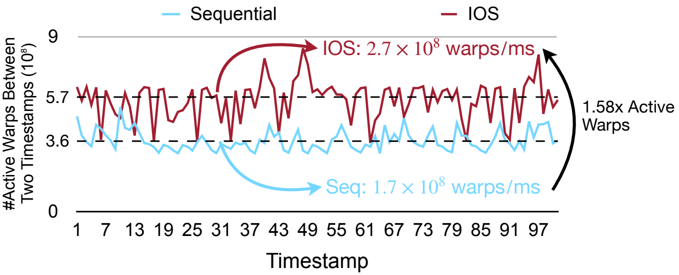
Comparison of Different Schedules



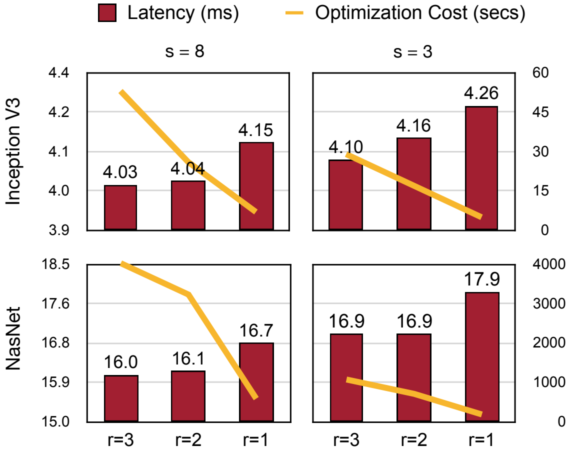
Comparison of cuDNN-based Frameworks



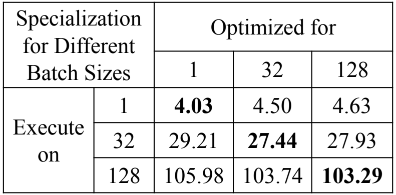
More Active Warps Improve Utilization



Schedule Pruning Reduces Search Time

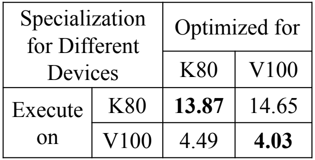


Specialized Scheduling is Beneficial



Specialization for Different Batch Sizes		Optimized for		
		1	32	128
Execute on	1	**4.03**	4.50	4.63
	32	29.21	**27.44**	27.93
	128	105.98	103.74	**103.29**

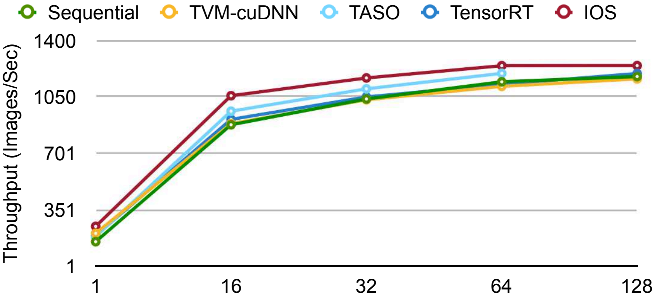
(1) Specialization for Batch Sizes



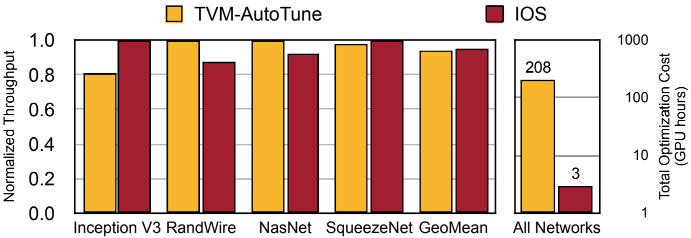
Specialization for Different Devices		Optimized for	
		K80	V100
Execute on	K80	**13.87**	14.65
	V100	4.49	**4.03**

(2) Specialization for Devices

Consistent Improvement for Different Batch Sizes



Intra- and Inter-Operator Parallelism



Summary

Conclusion

Strength

- IOS introduces the concept of **stage**, which enables dynamic programming.
- The algorithm description is detailed and the open-sourced code is clean.

Limitation

- The paper omits the detail about the profiler. IOS needs the cost of running multiple operators concurrently, which is not easy to simulate.
- The strategy space is very limited (compared with related works).
- They do not compare with similar works (Rammer etc.) in experiments.

Takeaways

- Dynamic programming works well with the DAG structure of neural networks. Other works have explored using dynamic programming for saving memory, distributed training, etc.
- We can design a reduced search space and use an efficient algorithm to find the global optimal.

Thank you!