# **TASO:** Optimizing Deep Learning Computation with Automatic Generation of Graph Substitutions (SOSP'19)

**Authors:**

Zhihao Jia, Oded Padon, James Thomas, Todd Warszawski, Matei Zaharia, Alex Aiken

Presenters: Tianyi Ge, Haojie Ye, Lingyun Guo

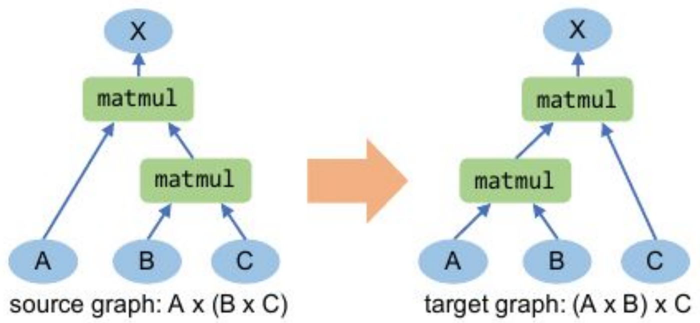
Feb 22, 2021

# Overview

- Background
- Problems
- Contributions
- Methodologies
- Evaluations
- Conclusions
- Discussions

# Background

- DNN frameworks represent NN as a computation graph
    - node := tensor operator
- To optimize a huge computation graph
    - Equivalent subgraph substitution
- Rule-based optimization strategy
    - TensorFlow, PyTorch, TVM, etc.
    - Manually designed substitution
- Previous work
    - MetaFlow [Jia, SysML'19]
        - Cost model: automated cost evaluation



# Problems

## Limitations of Rule-based Optimizations

- Robustness
  - Experts' manual heuristics do not apply to all models
- Scalability
  - Introducing new operators and properties require more rules
  - TensorFlow XLA: ~4K LOC just for convolution
- Performance
  - Human's heuristics might miss subtle optimizations
  - Some optimizations become worse!

https://www.youtube.com/watch?v=YsJICemFBsQ

# Contributions

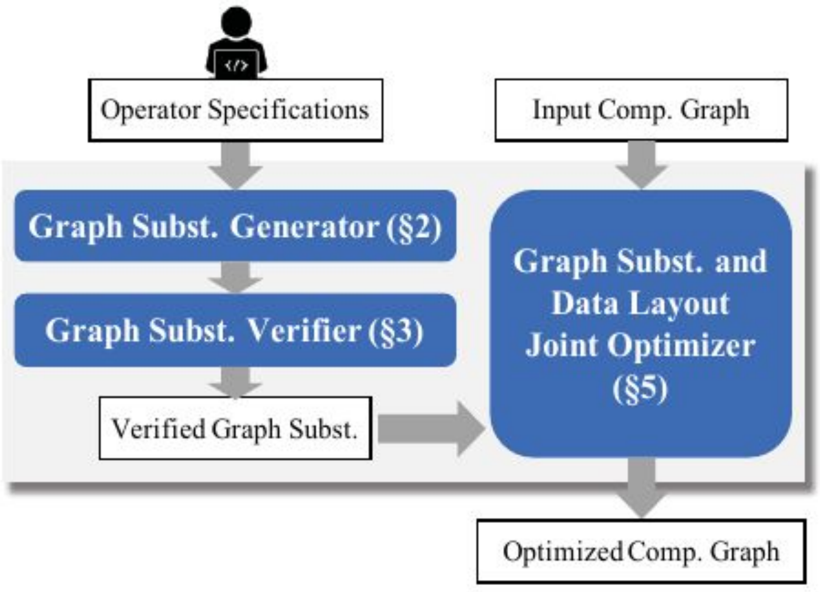
**TASO:** Tensor Algebra SuperOptimizer

- Manually-designed rule (❌) Automated generation and verification (✅)
- Better program performance: improve ML training/inference up to 10x
- Less effort: save codes of optimization up to 38x
  - LOC: 53K (❌) => 1.4K (✅)
- Fast optimization search: minutes
- Framework-agnostic: high portability among different frameworks

# Methodologies

1. Graph substitution generator
   - Operators
   - Graph Fingerprinting
   - Depth-first search
2. Graph substitution verifier
   - Operator Properties
3. Pruning
   - Input tensor renaming
   - Common subgraph
4. Joint Optimizer
   - Cost-based backtracking search

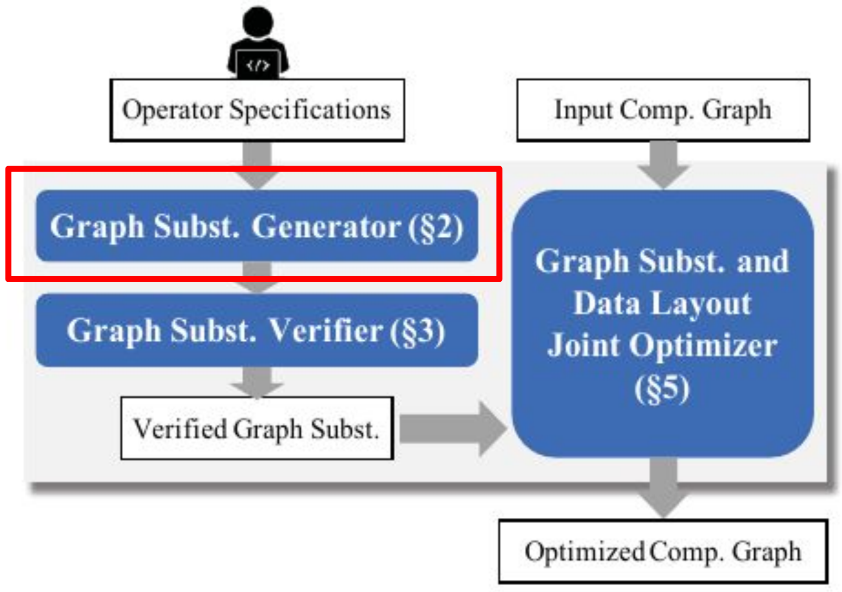
TASO workflow



# Methodologies

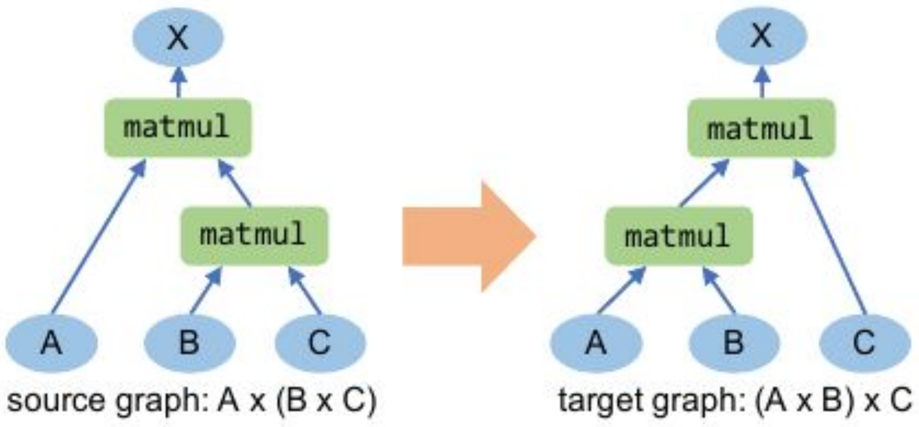
1. Graph substitution generator
    - Operators
    - Graph Fingerprinting
    - Depth-first search
2. Graph substitution verifier
    - Operator Properties
3. Pruning
    - Input tensor renaming
    - Common subgraph
4. Joint Optimizer
    - Cost-based backtracking search

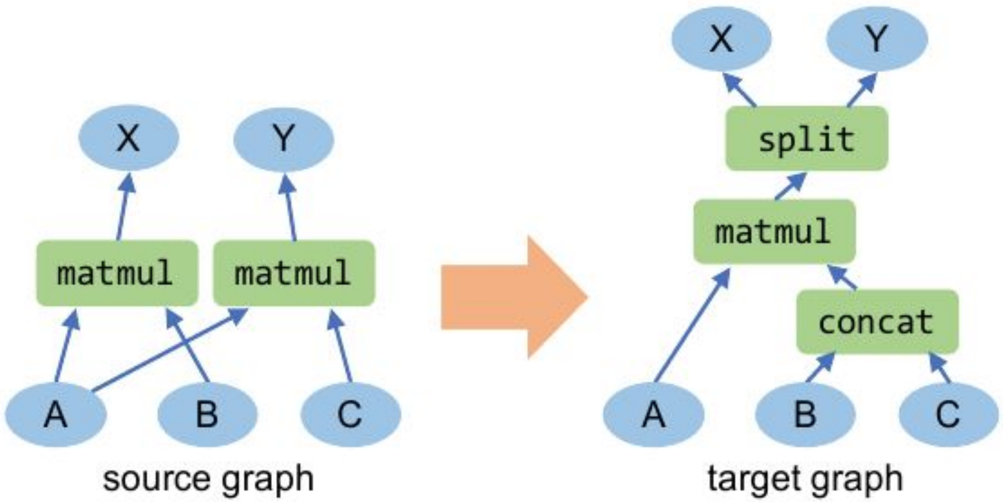
TASO workflow



# Methodologies: Graph substitution generator

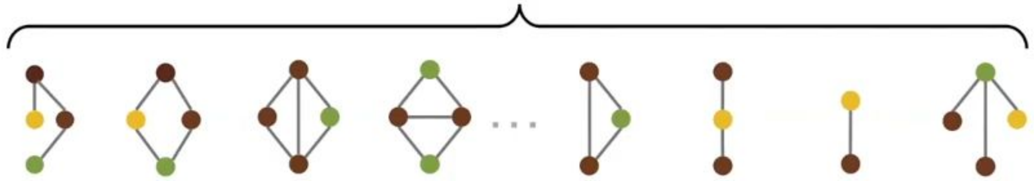
- What is graph substitution?
- Mathematically equivalent computations



# Methodologies: Graph substitution generator

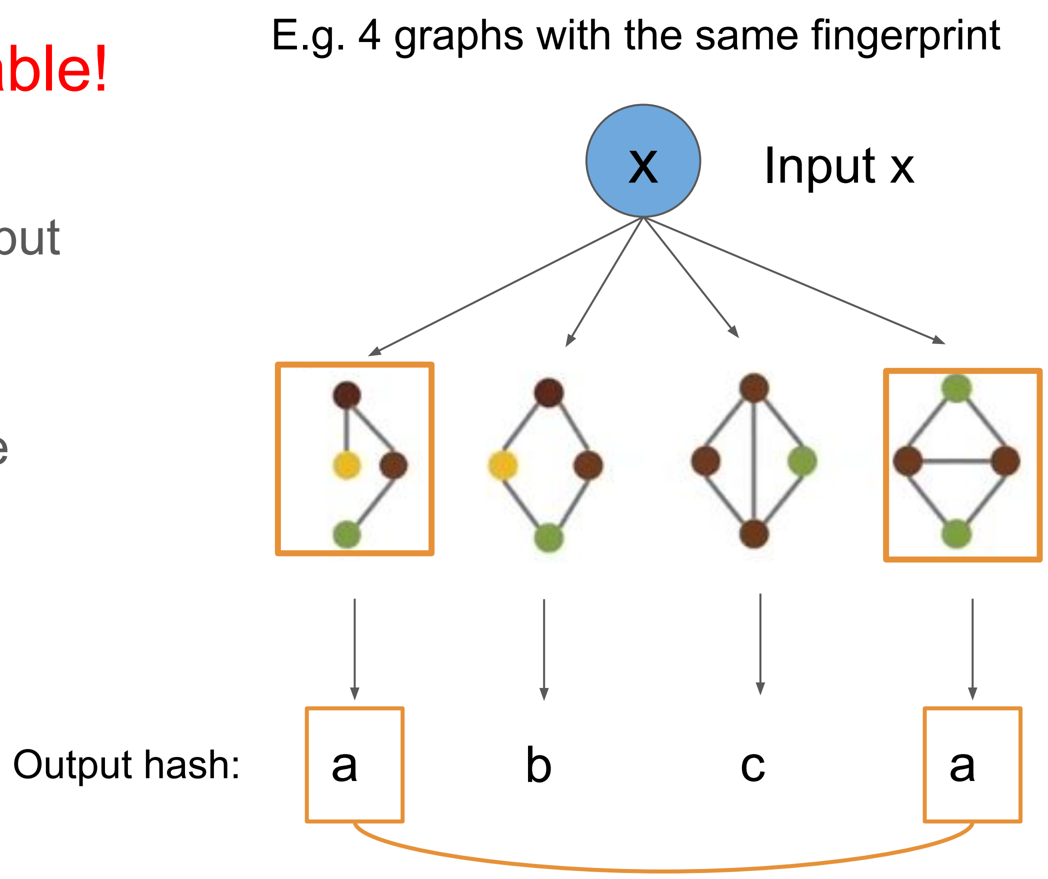
- Enumerate all possibilities over a set of DNN operators
  - Use DFS to find all possible graphs (n-level DFS)
  - Use up to 4 operators for implementation (66M in total)
  - Need to find pairs of equivalent graphs
    - How?




| Name | Description | Parameters |
|---|---|---|
| | Tensor Operators | |
| ewadd | Element-wise addition | |
| ewmul | Element-wise multiplication | |
| smul | Scalar multiplication | |
| transpose | Transpose | |
| matmul | Batch matrix multiplication# | |
| conv | Grouped convolution% | stride, padding, activation |
| enlarge | Pad conv. kernel with zeros† | kernel size |
| relu | Relu operator | |
| $\mathrm{pool}_{avg}$ | Average pooling | kernel size, stride, padding |
| $\mathrm{pool}_{max}$ | Max pooling | kernel size, stride, padding |
| concat | Concatenation of two tensors | concatenation axis |
| $\mathrm{split}_{\{0,1\}}$ | Split into two tensors | split axis |
| | Constant Tensors | |
| $C_{pool}$ | Average pooling constant | kernel size |
| $I_{conv}$ | Convolution id. kernel | kernel size |
| $I_{matmul}$ | Matrix multiplication id. | |
| $I_{ewmul}$ | Tensor with 1 entries | |

# Methodologies: Graph fingerprinting

- Pairwise comparison $O(n^2)$? ❌ not scalable!
- Fingerprinting ✅
  - Take hash value of output tensor with <u>small</u> input tensors
  - Roughly group the graphs by their fingerprint
  - Run <u>several rounds of random tests</u> to remove "lucky" cases (efficient and strong pruning)



# Methodologies: Graph fingerprinting

- Fingerprinting ✅
- Details
    a. Input tensors are integers to avoid floating-point errors
    b. The fingerprint is independent of permutation of output tensors
        - $hash_2$ is symmetric: ignorant of output tensor order

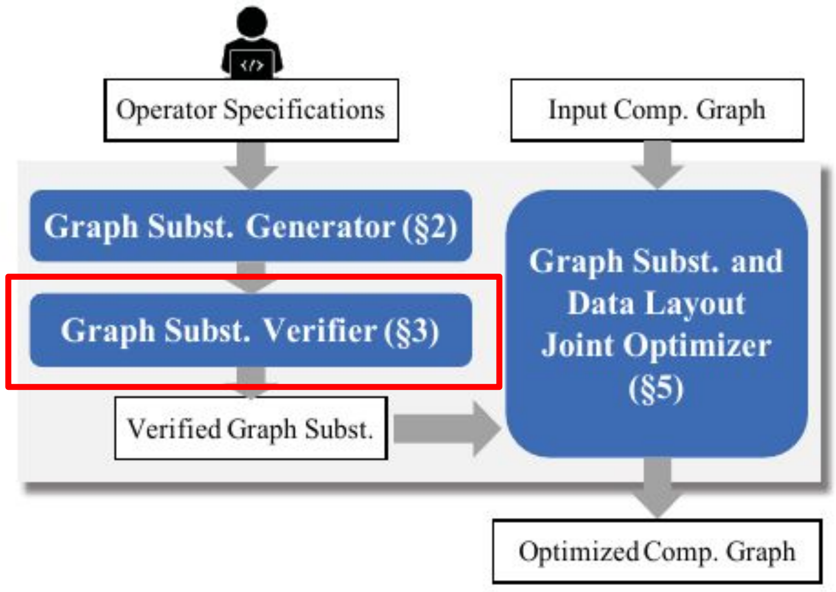
$$\textsc{FingerPrint}(\mathcal{G}) = hash_2(\{hash_1(t_i) \mid i \in \textsc{Outputs}(\mathcal{G})\})$$

- At this stage, TASO generates all 28744 substitutions <5 minutes

# Methodologies

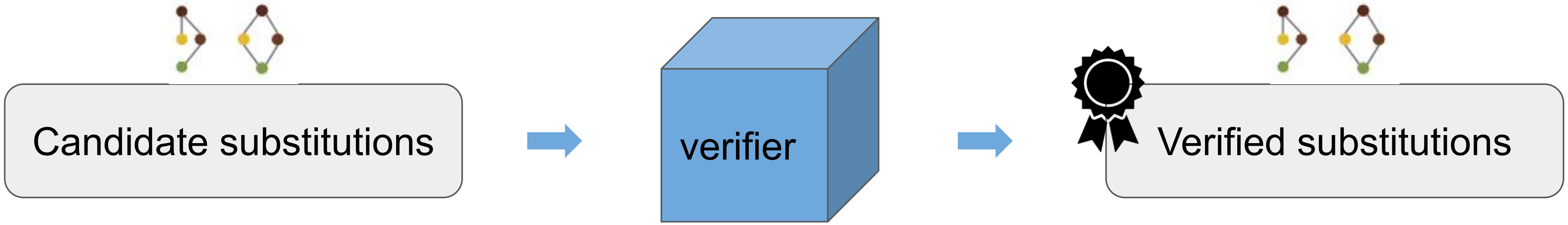
1. Graph substitution generator
    - Operators
    - Graph Fingerprinting
    - Depth-first search
2. Graph substitution verifier
    - Operator Properties
3. Pruning
    - Input tensor renaming
    - Common subgraph
4. Joint Optimizer
    - Cost-based backtracking search

TASO workflow



# Methodologies: Graph substitution verifier

- Multiple rounds of tests do not guarantee correctness of equivalence
- Automated theorem prover
  - Z3 prover
  - Given axioms, Z3 is able to prove/disprove equivalence
- What are the **axioms** here?



# Methodologies: Graph substitution verifier

- What are the **axioms** here? First-order logics
- Z3 can tell if an argument is true based on the given axioms
  - E.g. element-wise addition is associative
- How to find the properties?

| Operator Property |
| --- |
| $\forall x, y, z.\ \text{ewadd}(x, \text{ewadd}(y, z)) = \text{ewadd}(\text{ewadd}(x, y), z)$ |

```
# ewadd is associative
(ForAll([x,y,z], ewadd_0(x,ewadd_0(y, z)) == ewadd_0(ewadd_0(x,y),z)),
 lambda : [(s,s,s) for dim in [2,3,4] for s in product(N, repeat=dim)] ),
```

TASO source code for verifier

https://github.com/jiazhihao/TASO/blob/master/verify/verify.py#L101

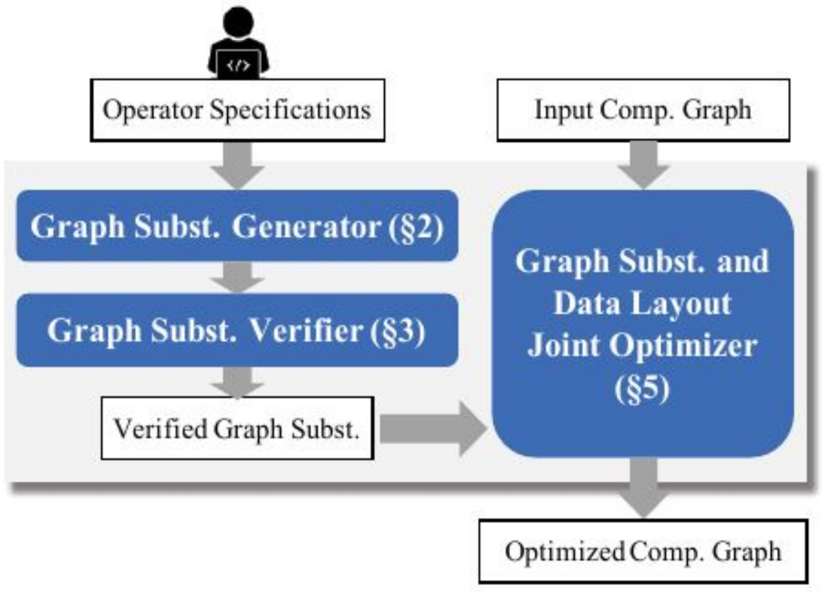
# Methodologies: Graph substitution verifier

- How to find the properties? Iteratively
  - First, manually verify the generated <u>candidate</u> substitutions
  - If failed to verify, use Z3 with bounded tensor size up to 4x4x4x4
  - Add newly verified properties to axioms
- Consistency & Redundancy check

# Methodologies

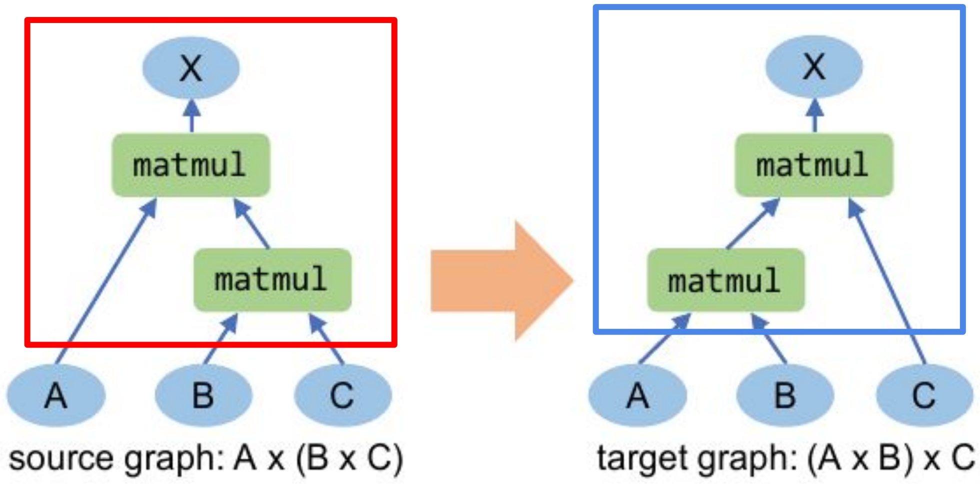
1. Graph substitution generator
    - Operators
    - Graph Fingerprinting
    - Depth-first search
2. Graph substitution verifier
    - Operator Properties
3. Pruning
    - Input tensor renaming
    - Common subgraph
4. Joint Optimizer
    - Cost-based backtracking search

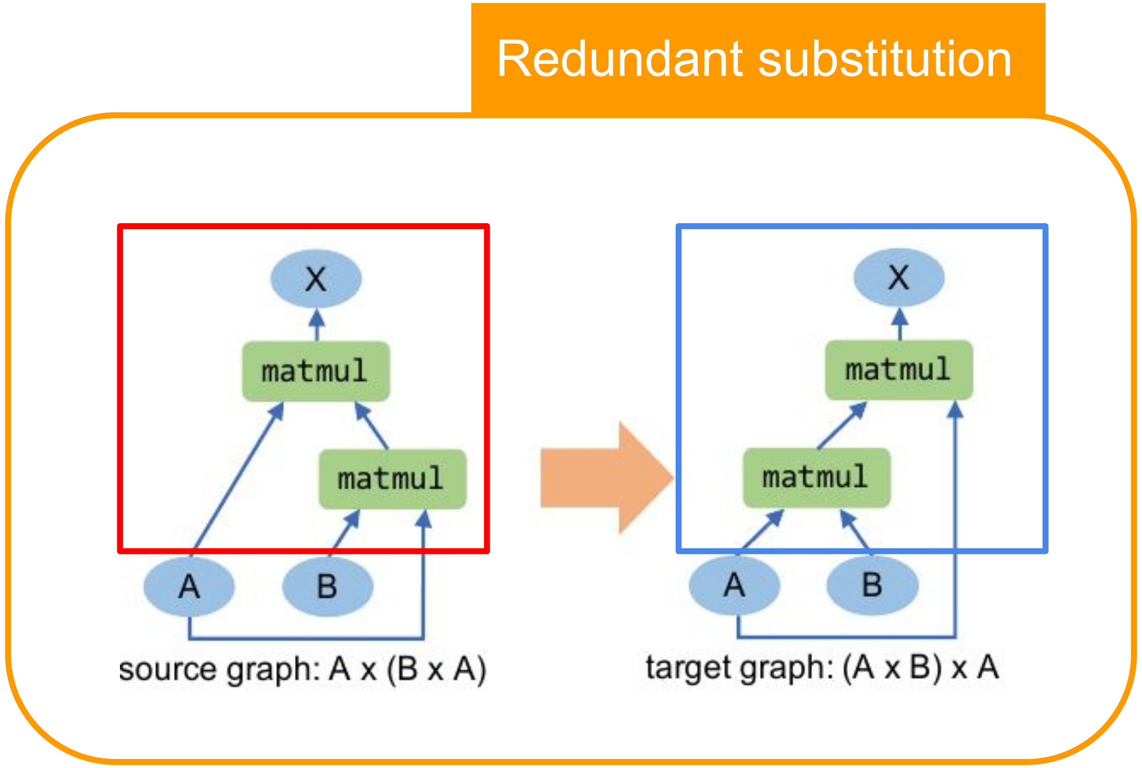
TASO workflow



# Methodologies: Pruning techniques

- Remove redundant substitutions but preserve all the possible ones
- Two types
    a. Rename input tensors (rename C with A)



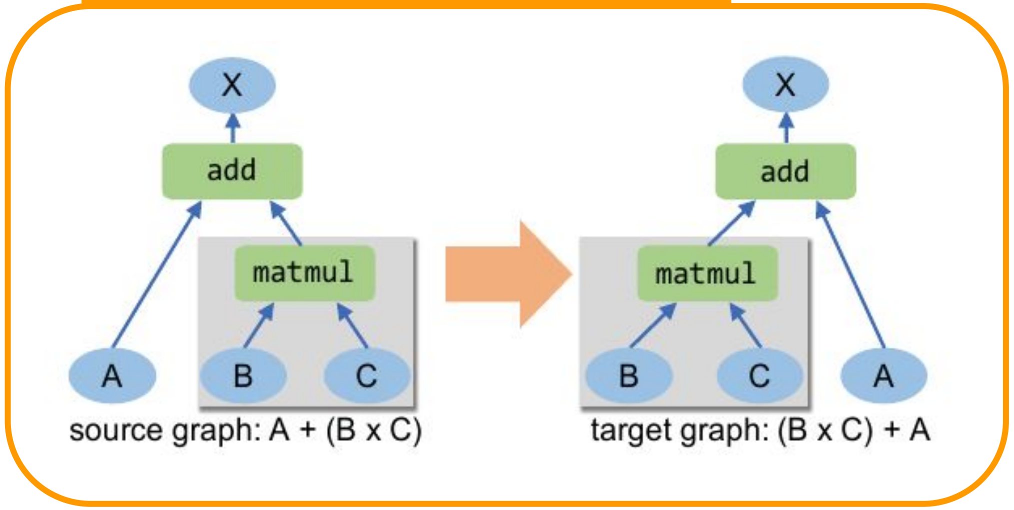



# Methodologies: Pruning techniques

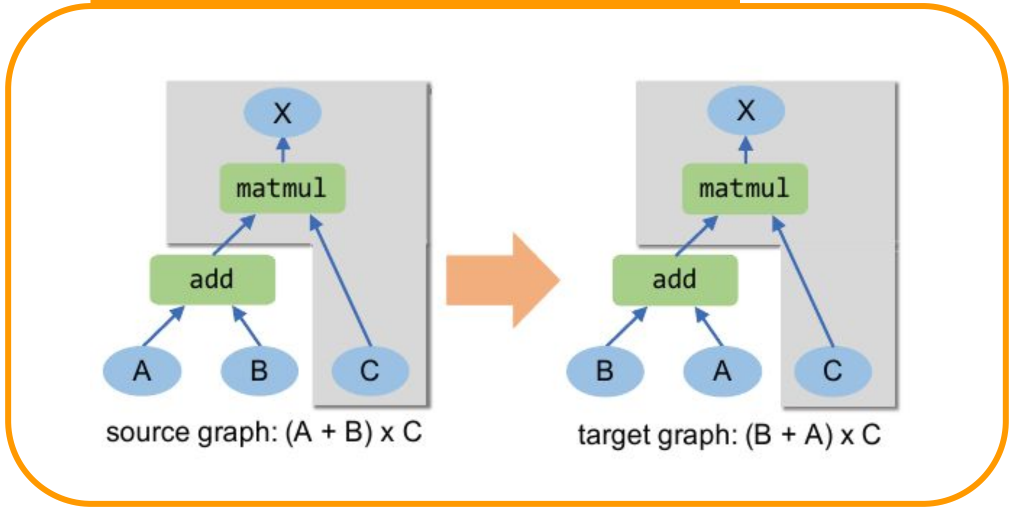
- Two types
  a. Rename input tensors
  b. Common subgraph
     i. E.g. Actually both are commutative laws!

| Pruning Techniques | Remaining Substitutions | Reduction v.s. Initial |
|---|---|---|
| Initial | 28744 | 1× |
| Input tensor renaming | 17346 | 1.7× |
| Common subgraph | 743 | 39× |

Common input subgraph



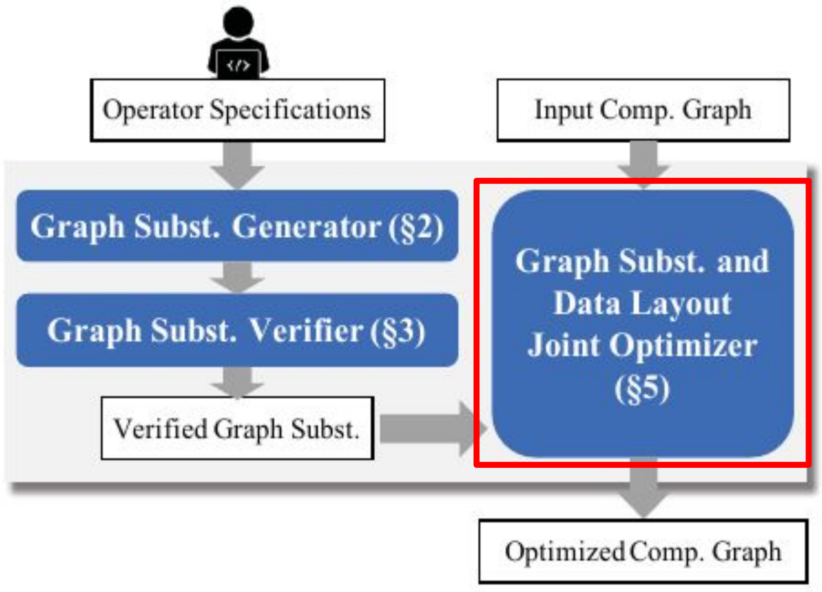

Common output subgraph



# Methodologies

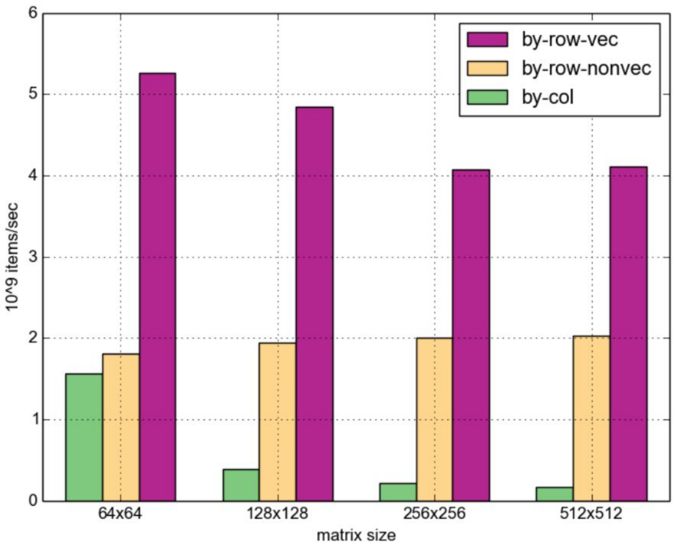
1. Graph substitution generator
    - Operators
    - Graph Fingerprinting
    - Depth-first search
2. Graph substitution verifier
    - Operator Properties
3. Pruning
    - Input tensor renaming
    - Common subgraph
4. Joint Optimizer
    - Cost-based backtracking search

TASO workflow

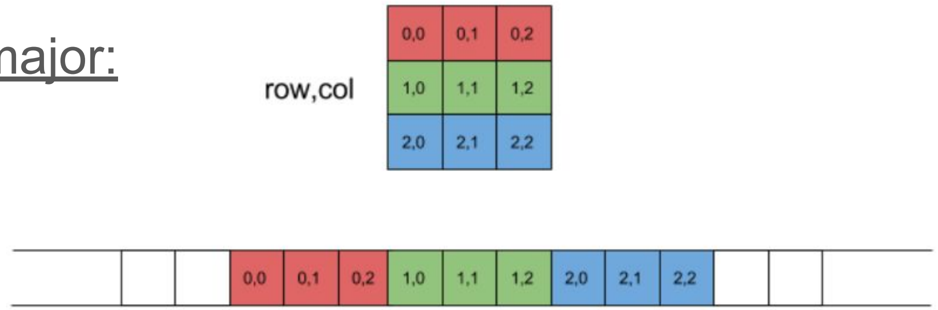


# Methodologies: Search-based graph optimizer

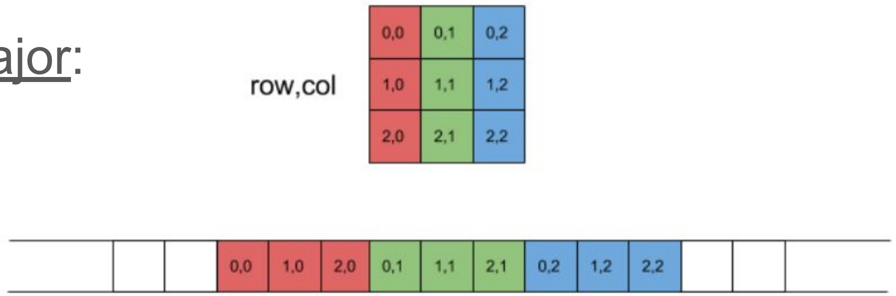
- Combine Data Layout and Substitution
  - Row-major vs. Col-major
  - Ex: Test function `ewadd` by row




Row-major:



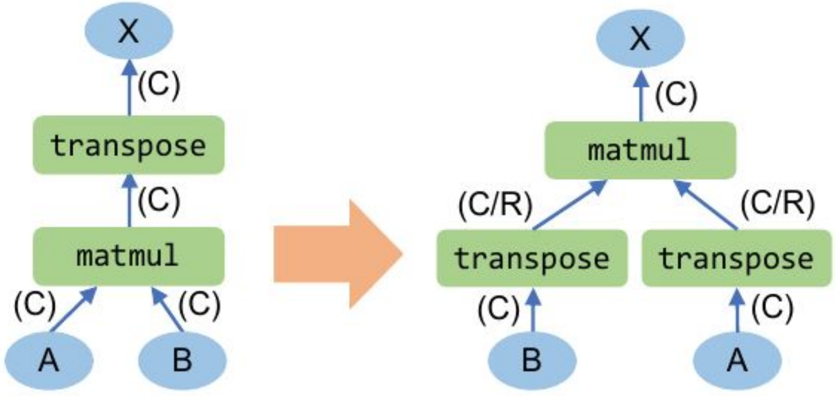

Col-major:




https://eli.thegreenplace.net/2015/memory-layout-of-multi-dimensional-arrays

# Methodologies: Search-based graph optimizer

- Combine Data Layout and Substitution
- Enumerate all layout possibilities



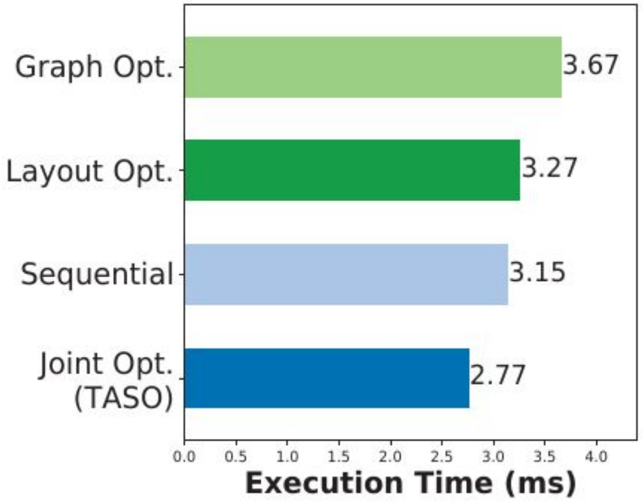

source graph: $(A \times B)^T$ target graph: $(B^T \times A^T)$

**Algorithm 2** Cost-Based Backtracking Search

**Input**: an input graph $\mathcal{G}_{in}$, verified substitutions $\mathcal{S}$, a cost model $\boxed{Cost(\cdot),}$ and a hyper parameter $\alpha$.<br>
**Output**: an optimized graph.<br>
<br>
$\mathcal{P} = \{\mathcal{G}_{in}\}$ // *$\mathcal{P}$ is a priority queue sorted by Cost.*<br>
**while** $\mathcal{P} \neq \{\}$ **do**<br>
    $\mathcal{G} = \mathcal{P}$.dequeue()<br>
    **for** substitution $s \in \mathcal{S}$ **do**<br>
        $\boxed{\textit{// } \textsc{Layout}(\mathcal{G}, s) \textit{ returns possible layouts applying } s \textit{ on } \mathcal{G}.}$<br>
        **for** layout $l \in \textsc{Layout}(\mathcal{G}, s)$ **do**<br>
            // *$\textsc{Apply}(\mathcal{G}, s, l)$ applies $s$ on $\mathcal{G}$ with layout $l$.*<br>
            $\mathcal{G}' = \textsc{Apply}(\mathcal{G}, s, l)$<br>
            **if** $\mathcal{G}'$ is valid **then**<br>
                **if** $Cost(\mathcal{G}') < Cost(\mathcal{G}_{opt})$ **then**<br>
                    $\mathcal{G}_{opt} = \mathcal{G}'$<br>
                **if** $Cost(\mathcal{G}') < \alpha \times Cost(\mathcal{G}_{opt})$ **then**<br>
                    $\mathcal{P}$.enqueue($\mathcal{G}'$)<br>
**return** $\mathcal{G}_{opt}$

# Methodologies: Search-based graph optimizer

- Combine Data Layout and Substitution
- α: controls the search range
  - α = 1 ⇔ greedy search
  - Practically, α = 1.05
- Better E2E inference performance



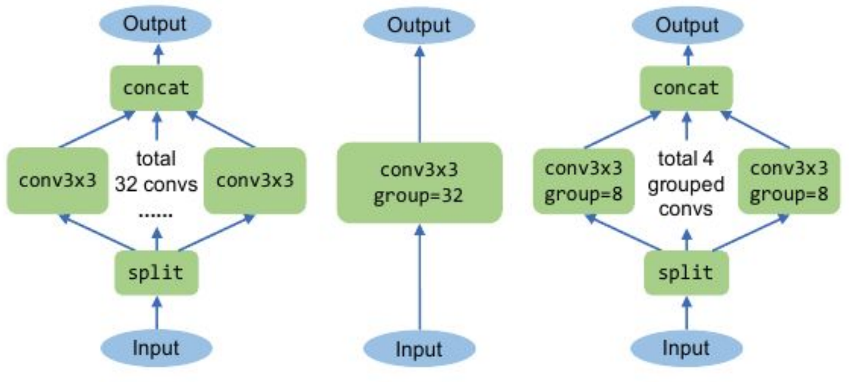

**Algorithm 2** Cost-Based Backtracking Search

**Input**: an input graph $\mathcal{G}_{in}$, verified substitutions $\mathcal{S}$, a cost model $Cost(\cdot)$, and a hyper parameter $\alpha$.
**Output**: an optimized graph.

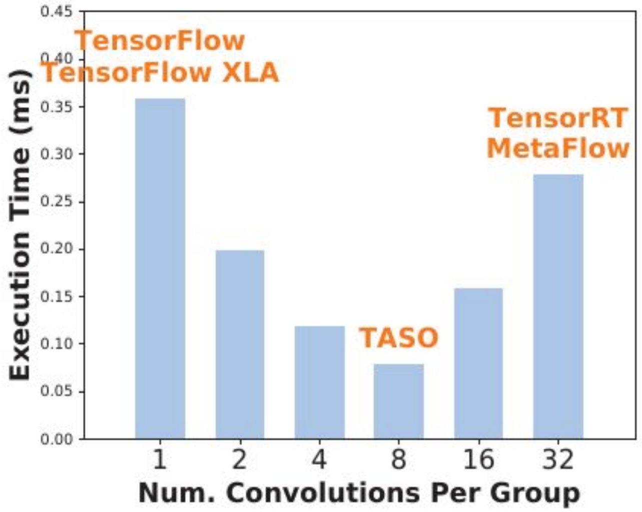
$\mathcal{P} = \{\mathcal{G}_{in}\}$ // *$\mathcal{P}$ is a priority queue sorted by Cost.*
**while** $\mathcal{P} \neq \{\}$ **do**
  $\mathcal{G} = \mathcal{P}$.dequeue()
  **for** substitution $s \in \mathcal{S}$ **do**
    // *LAYOUT($\mathcal{G}, s$) returns possible layouts applying $s$ on $\mathcal{G}$.*
    **for** layout $l \in$ LAYOUT($\mathcal{G}, s$) **do**
      // *APPLY($\mathcal{G}, s, l$) applies $s$ on $\mathcal{G}$ with layout $l$.*
      $\mathcal{G}' =$ APPLY($\mathcal{G}, s, l$)
      **if** $\mathcal{G}'$ is valid **then**
        **if** $Cost(\mathcal{G}') < Cost(\mathcal{G}_{opt})$ **then**
          $\mathcal{G}_{opt} = \mathcal{G}'$
        **if** $Cost(\mathcal{G}') < \boxed{\alpha} \times Cost(\mathcal{G}_{opt})$ **then**
          $\mathcal{P}$.enqueue($\mathcal{G}'$)
**return** $\mathcal{G}_{opt}$

# Evaluations: examples

- TASO optimization of ResNeXt-50
    - Convolutions can be grouped
- TASO gives the best grouping solution in terms of execution time



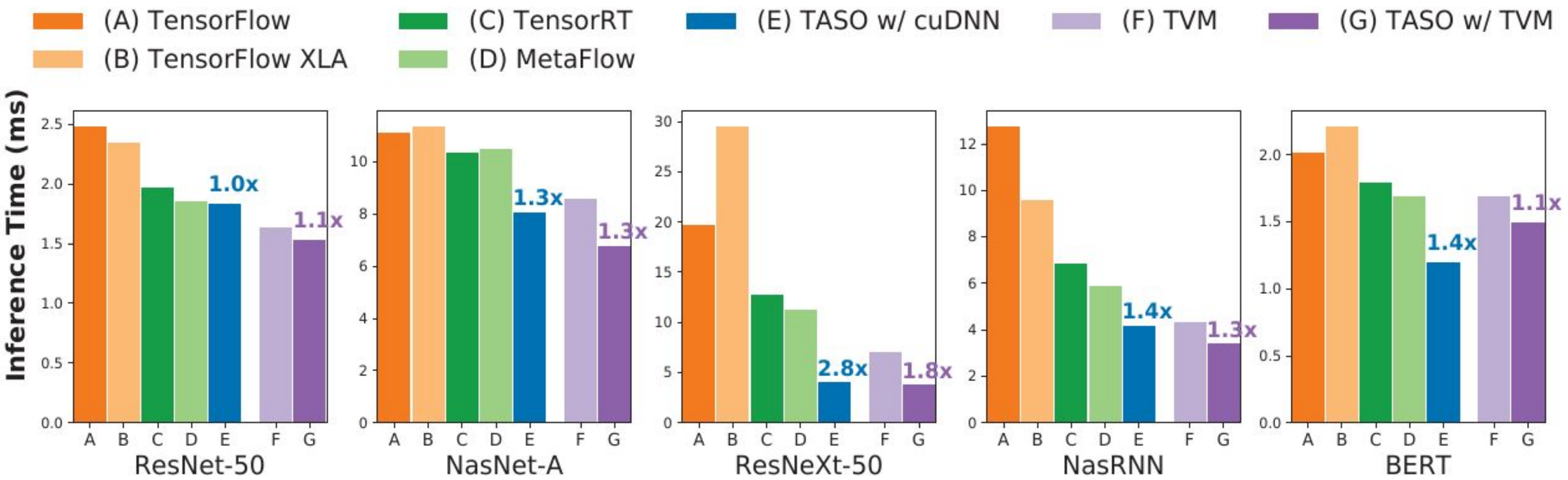

**(a)** Multi-branch convolution. **(b)** Grouped convolution. **(c)** Multi-branch grouped convolution.




**(d)** Performance comparison.

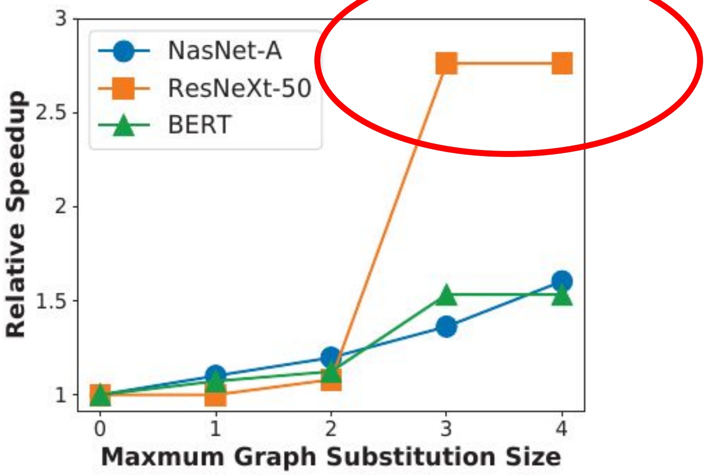
# Evaluations

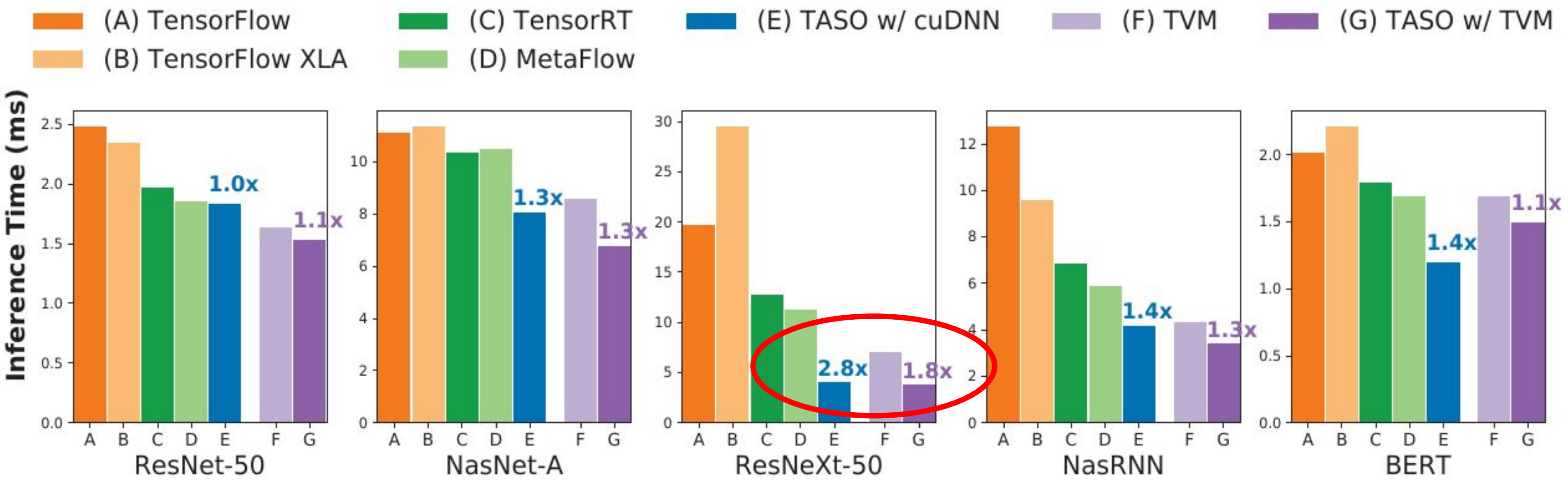
- Case study
  - ResNeXt-50, NasNet-A, NasRNN, BERT
- Experiment setup
  - 8-core Intel E5-2600 CPU
  - 64 GB DRAM
  - NVIDIA Tesla V100



# Evaluations

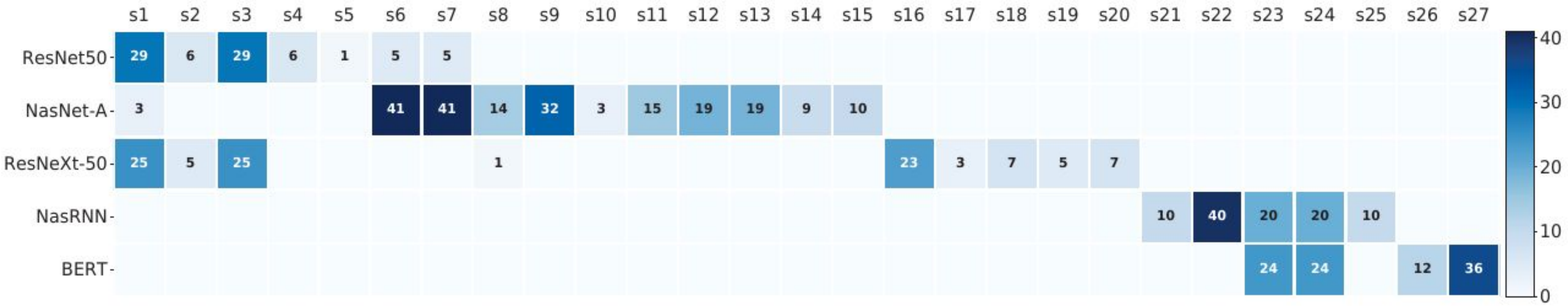
- ResNet-50 has been well-optimized
- ResNeXt-50 has huge space for improvement!



# Evaluations

- Each DNN optimization uses ~4-10 types of substitutions
- Implication: difficult for human experts to develop such optimization



# Conclusions

- Components
  - Subgraph substitution generator
  - Substitution verifier
  - Search-based optimizer
- Better program performance: improve ML training/inference up to 10x
- Less engineering effort: 53K => 1.4K
- Formal verifications
- TASO matches well-optimized DNNs, and outperforms the current solutions on other new DNNs

# Discussions

1. Highly effective pruning strategies
    a. Fingerprinting
    b. Common subgraph
2. Questions about formal verification
    a. How does Z3 (SMT solver) prove a new operator property? Enumeration?
3. Scalability: how to enumerate > 4 operators?
    a. From Jia’s talk, distributed computing might exceed the limit of 4 operators
    b. Distributed generator and verifier
    c. Potential challenges?

# References

[1] Jia, Zhihao, et al. "TASO: optimizing deep learning computation with automatic generation of graph substitutions." *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 2019.

[2] Jia, Zhihao, et al. "Optimizing dnn computation with relaxed graph substitutions." *SysML 2019* (2019).

[3] Automated Discovery of Machine Learning Optimizations
https://www.youtube.com/watch?v=YsJICemFBsQ

[4] Memory layout of multi-dimensional arrays
https://eli.thegreenplace.net/2015/memory-layout-of-multi-dimensional-arrays

[5] https://github.com/Z3Prover/z3

[2] http://theory.stanford.edu/~nikolaj/programmingz3.html

# Ansor: Generating High-Performance Tensor Programs for Deep Learning


Author

Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu,
Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo,
Koushik Sen, Joseph E. Gonzalez, Ion Stoica

Presenter

Haojie Ye, Lingyun Guo, Tianyi Ge

# Outline

## Background

What is tensor program? / Challenges of generating high efficiency tensor program

Current state-of-the-art of tensor program generator

## Design

Sketch and Annotation – Decoupling coarse and fine-grained program sampling

Evolutionary Search – Mutating and tuning the program inspired by biological evolution

Task Scheduler – Task scheduling to meet the code generation latency requirement

## Benchmark & Conclusion

Ansor finds high-performance programs that are ***outside the search space of existing state-of-the-art compiler approaches***, achieving 3.8x and 1.7x over state-of-the-art tensor program generator on Intel CPU and NVIDIA GPUs.

# Background

***What is Tensor/Tensor programs?***

A tensor may be represented as a (potentially multidimensional) array.

Modern DNNs can be represented as directed acyclic graph, or DAG. DAG involves executing tensor programs.

***Why Optimizing tensor programs?***

Converting these mathematical declaration to H/W execution is an open task for the compiler, thus may not be optimal if mapped naively.

*** The mathmetical expression:**

$$C[i,j] = \sum_k A[i,k] \times B[k,j]$$

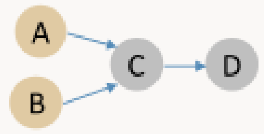
$$D[i,j] = \max(C[i,j], 0.0)$$

where $0 \le i,j,k < 512$

*** The corresponding naive program:**

```
for i in range(512):
  for j in range(512):
    for k in range(512):
      C[i, j] += A[i, k] * B[k, j]
for i in range(512):
  for j in range(512):
    D[i, j] = max(C[i, j], 0.0)
```

*** The corresponding DAG:**

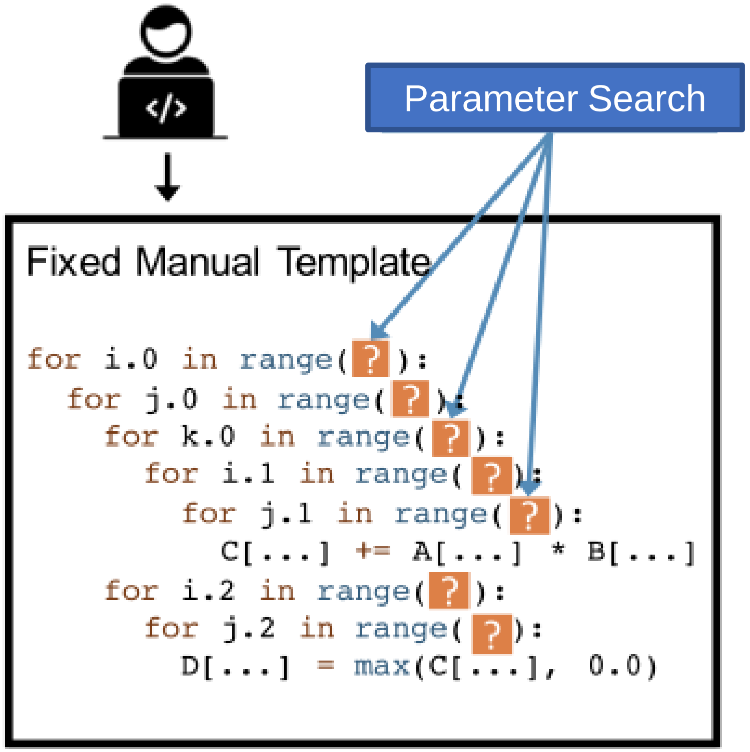



Matrix Multiplication $C_{i,j} = \sum_k A_{i,k} B_{k,j}$

```
C = compute((N, M), lambda i, j: sum(A[i, k]*B[k, j], [k]))
```

# State-of-the-art tensor program compilers

Mathematical declaration to H/W execution is an open task for the compiler (Heterogeneity makes the exploration space even larger). Mutating and tuning the tensor program can make a big difference in computation throughput.




(a) Template-guided Search

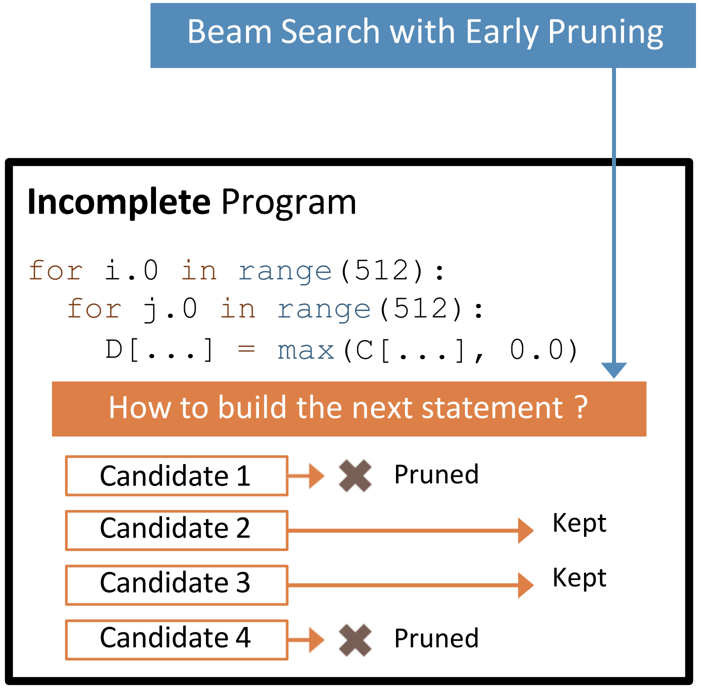
## Template-guided search (e.g., TVM)

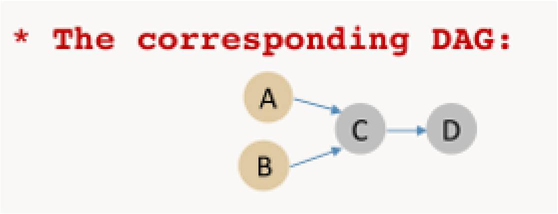
Use **templates** to define the search space for every operator

### Drawbacks

- Not fully-automated -> Requires huge manual effort, cannot adapt to flexible operations
- Limited search space -> Does not achieve optimal performance

# State-of-the-art tensor program compilers




## Sequential Construction Based Search (e.g., Halide)

Use **beam search** to generate the Sequential Construction Based Search

### Drawbacks

- Intermediate candidates are pruned by analyzing incomplete programs -> The cost model cannot do accurate prediction
- Sequential order -> The error accumulates with increased number of layers in the network

# Outline

## Background

What is tensor program? / Challenges of generating high efficiency tensor program

Current state-of-the-art of tensor program generator

## Design

Sketch and Annotation – Decoupling coarse and fine-grained program sampling

Evolutionary Search – Mutating and tuning the program inspired by biological evolution

Task Scheduler – Task scheduling to meet the code generation latency requirement

## Benchmark & Conclusion

Ansor finds high-performance programs that are ***outside the search space of existing state-of-the-art approaches***, achieving 3.8x and 1.7x over state-of-the-art tensor program generator on Intel CPU and NVIDIA GPUs.

# Design

**How to allocate resource for many search tasks?**

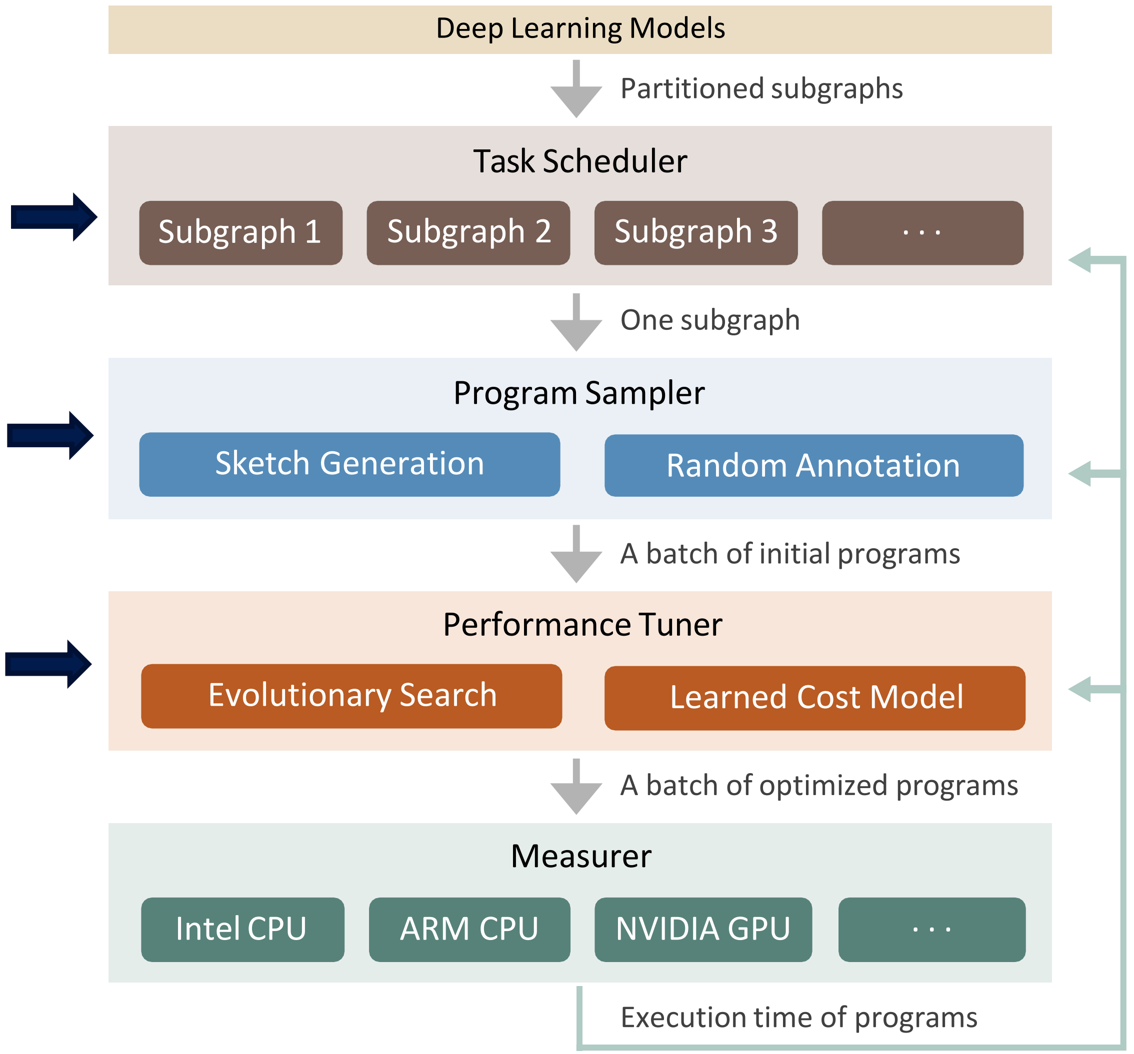
- Utilize a task scheduler to prioritize important tasks

**How to build a large search space automatically?**

- Use a hierarchical search space

**How to search efficiently?**

- Sample complete programs and fine-tune them



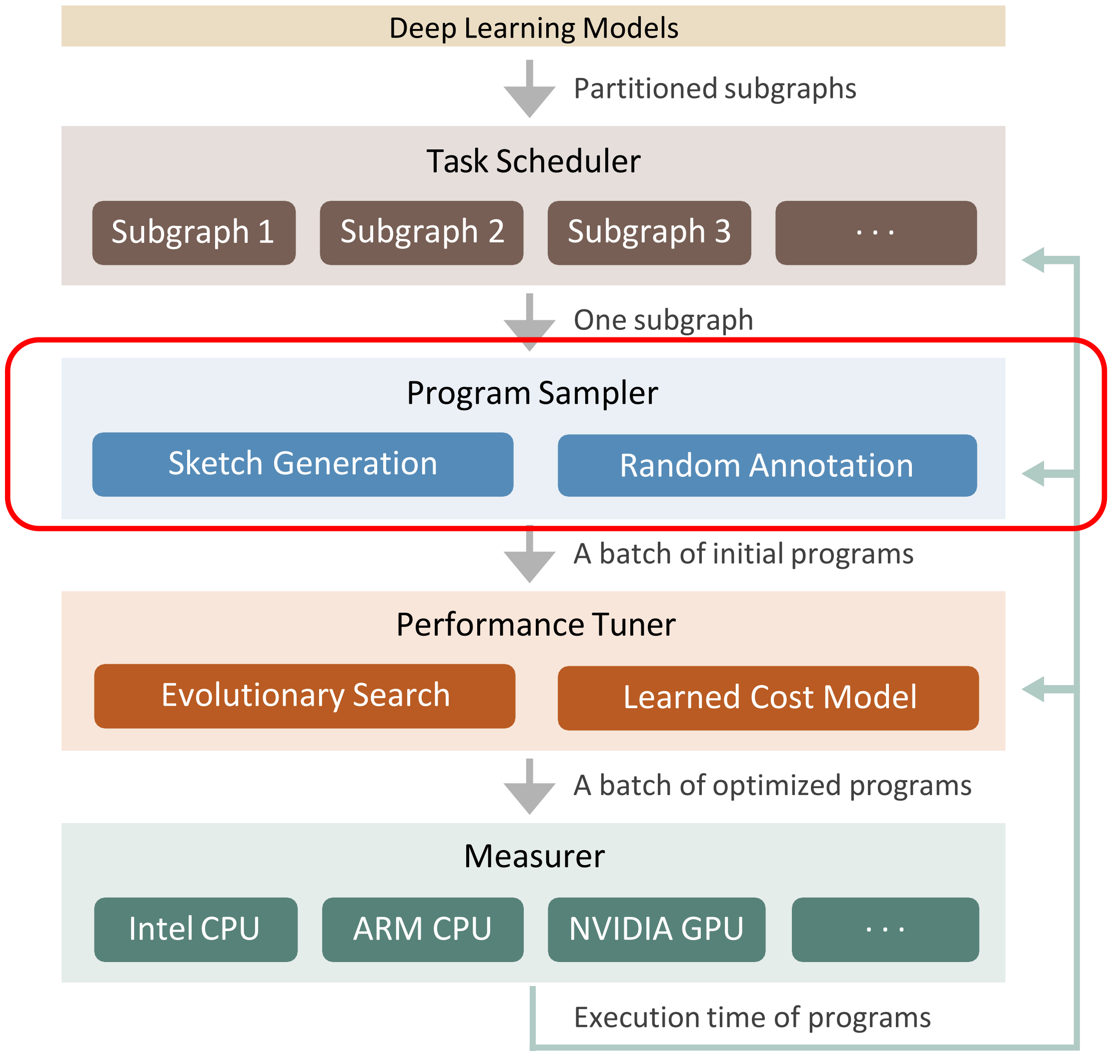
# Design

Goal:

- Automatically construct **a large search space** and **uniformly sample** from the design space

Approach:

- **Sketch**: a few good high-level structures
- **Annotation**: millions of low-level details



# Rule-based Sketch

Example Input 1:

*** The mathmetical expression:**

$C[i,j] = \sum_k A[i,k] \times B[k,j]$

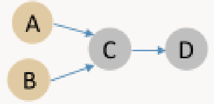
$D[i,j] = \max(C[i,j], 0.0)$

where $0 \leq i,j,k < 512$

*** The corresponding naive program:**

```
for i in range(512):
  for j in range(512):
    for k in range(512):
      C[i, j] += A[i, k] * B[k, j]
for i in range(512):
  for j in range(512):
    D[i, j] = max(C[i, j], 0.0)
```

*** The corresponding DAG:**




$$Input\ 1 \rightarrow \sigma(S_0, i=4) \xrightarrow{\text{Rule 1}} \sigma(S_1, i=3) \xrightarrow{\text{Rule 4}} \sigma(S_2, i=2) \xrightarrow{\text{Rule 1}} \sigma(S_3, i=1) \xrightarrow{\text{Rule 1}} Sketch\ 1$$

Write **naive tensor programs** with nested loops. Start **coarse-grained** program sketch on DAG from backwards.

Generated sketch 1

```
for i.0 in range(TILE_I0):
  for j.0 in range(TILE_J0):
    for i.1 in range(TILE_I1):
      for j.1 in range(TILE_J1):
        for k.0 in range(TILE_K0):
          for i.2 in range(TILE_I2):
            for j.2 in range(TILE_J2):
              for k.1 in range(TILE_I1):
                for i.3 in range(TILE_I3):
                  for j.3 in range(TILE_J3):
                    C[...] += A[...] * B[...]
        for i.4 in range(TILE_I2 * TILE_I3):
          for j.4 in range(TILE_J2 * TILE_J3):
            D[...] = max(C[...], 0.0)
```

| No | Rule Name | Condition | Application |
|---|---|---|---|
| 1 | Skip | $\neg IsStrictInlinable(S,i)$ | $S' = S; i' = i-1$ |
| 2 | Always Inline | $IsStrictInlinable(S,i)$ | $S' = Inline(S,i); i' = i-1$ |
| 3 | Multi-level Tiling | $HasDataReuse(S,i)$ | $S' = MultiLevelTiling(S,i); i' = i-1$ |
| 4 | Multi-level Tiling with Fusion | $HasDataReuse(S,i) \land HasFusibleConsumer(S,i)$ | $S' = FuseConsumer(MultiLevelTiling(S,i),i); i' = i-1$ |
| 5 | Add Cache Stage | $HasDataReuse(S,i) \land \neg HasFusibleConsumer(S,i)$ | $S' = AddCacheWrite(S,i); i = i'$ |
| 6 | Reduction Factorization | $HasMoreReductionParallel(S,i)$ | $S' = AddRfactor(S,i); i' = i-1$ |
| ... | User Defined Rule | ... | ... |

# Rule-based Sketch

Example Input 2:

*** The mathmetical expression:**

$B[i,l] = \max(A[i,l], 0.0)$

$C[i,k] = \begin{cases} B[i,k], & k < 400 \\ 0\ , & k \geq 400 \end{cases}$

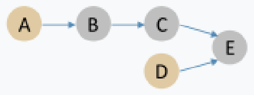
$E[i,j] = \sum_{k} C[i,k] \times D[k,j]$

where $0 \leq i < 8,\ 0 \leq j < 4,$
$0 \leq k < 512, 0 \leq l < 400$

*** The corresponding naive program:**

```
for i in range(8):
  for l in range(400):
    B[i, l] = max(A[i, l], 0.0)
for i in range(8):
  for k in range(512):
    C[i, k] = B[i, k] if k < 400 else 0
for i in range(8):
  for j in range(4):
    for k in range(512):
      E[i, j] += C[i, k] * D[k, j]
```

*** The corresponding DAG:**




$$Input\ 2 \rightarrow \sigma(S_0, i=5) \xrightarrow{\text{Rule 5}} \sigma(S_1, i=5) \xrightarrow{\text{Rule 4}}$$
$$\sigma(S_2, i=4) \xrightarrow{\text{Rule 1}} \sigma(S_3, i=3) \xrightarrow{\text{Rule 1}}$$
$$\sigma(S_4, i=2) \xrightarrow{\text{Rule 2}} \sigma(S_5, i=1) \xrightarrow{\text{Rule 1}} Sketch\ 2$$

**Multiple rules** can be applied to one state to generate **multiple succeeding states.** But the number of sketches is **less than 10** for a typical subgraph.

Generated sketch 2

```
for i in range(8):
  for k in range(512):
    C[i, j] = max(A[i,k], 0.0) if k<400 else 0
for i.0 in range(TILE_I0):
  for j.0 in range(TILE_J0):
    for i.1 in range(TILE_I1):
      for j.1 in range(TILE_J1):
        for k.0 in range(TILE_K0):
          for i.2 in range(TILE_I2):
            for j.2 in range(TILE_J2):
              for k.1 in range(TILE_I1):
                for i.3 in range(TILE_I3):
                  for j.3 in range(TILE_J3):
                    E.cache[...] += C[...] * D[...]
        for i.4 in range(TILE_I2 * TILE_I3):
          for j.4 in range(TILE_J2 * TILE_J3):
            E[...] = E.cache[...]
```

| No | Rule Name | Condition | Application |
|---|---|---|---|
| 1 | Skip | $\neg IsStrictInlinable(S,i)$ | $S' = S; i' = i-1$ |
| 2 | Always Inline | $IsStrictInlinable(S,i)$ | $S' = Inline(S,i); i' = i-1$ |
| 3 | Multi-level Tiling | $HasDataReuse(S,i)$ | $S' = MultiLevelTiling(S,i); i' = i-1$ |
| 4 | Multi-level Tiling with Fusion | $HasDataReuse(S,i) \wedge HasFusibleConsumer(S,i)$ | $S' = FuseConsumer(MultiLevelTiling(S,i),i); i' = i-1$ |
| 5 | Add Cache Stage | $HasDataReuse(S,i) \wedge \neg HasFusibleConsumer(S,i)$ | $S' = AddCacheWrite(S,i); i = i'$ |
| 6 | Reduction Factorization | $HasMoreReductionParallel(S,i)$ | $S' = AddRfactor(S,i); i' = i-1$ |
| ... | User Defined Rule | ... | ... |

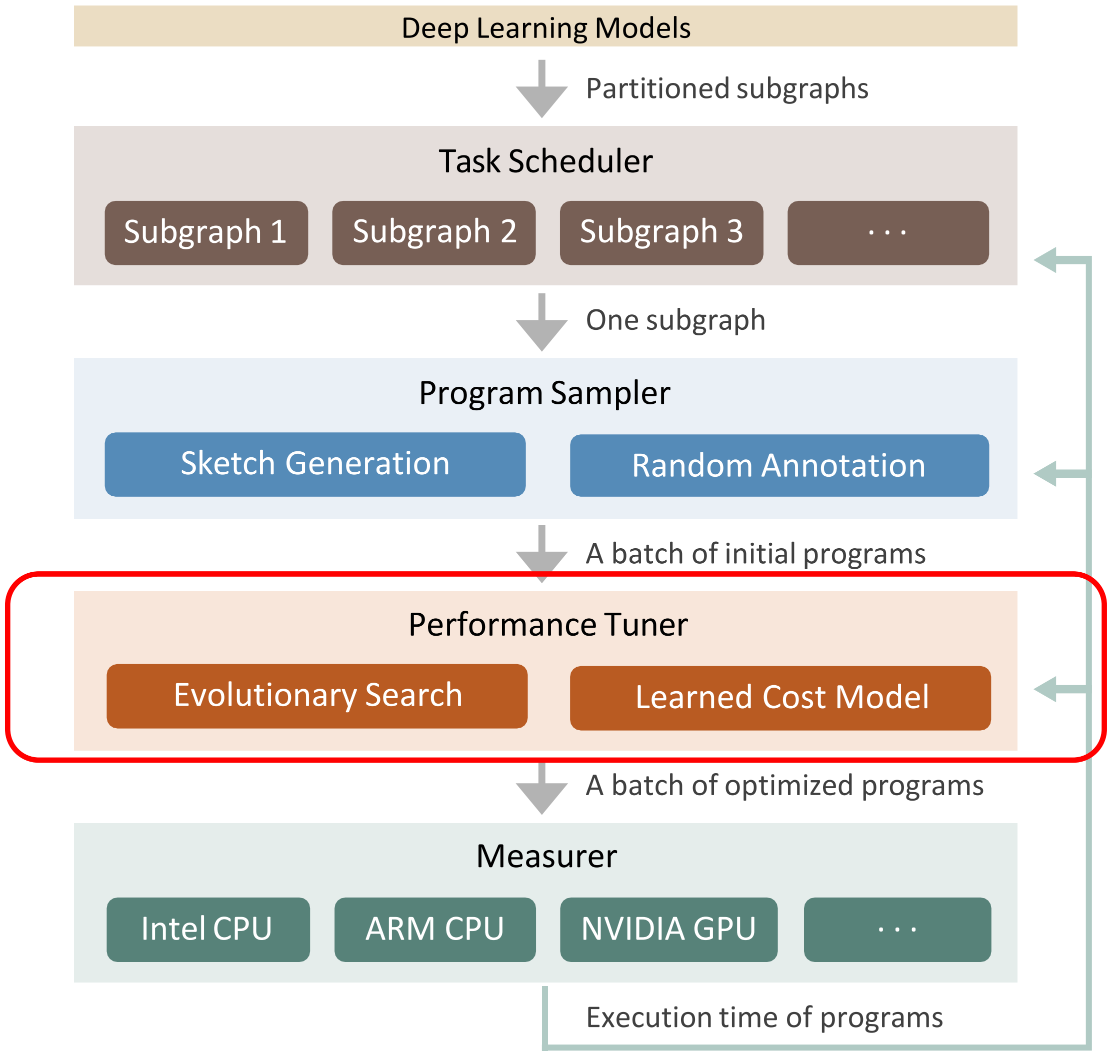
# Fine-grained Search for Best Annotation

Problem:

- **Random annotation** ensures a large search space, but **does not guarantee the performance**

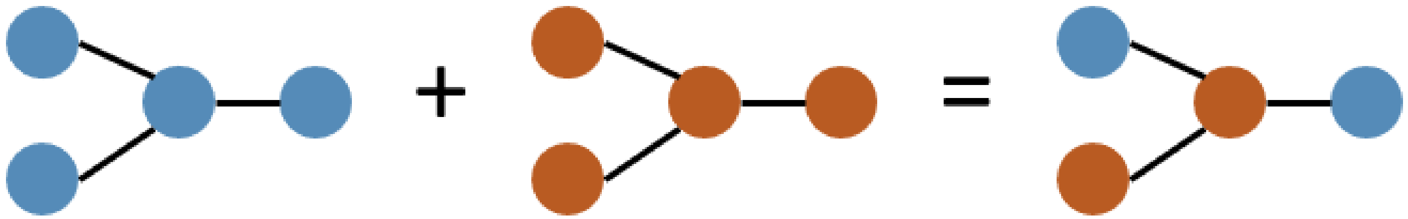
Approach:

- Perform **evolutionary search** to find heuristically a better program



# Evolutionary Search

- Random sampling in the design space does not guarantee the performance
- **Mutation**
  - Randomly mutate tile size
  - Randomly mutate parallel/unroll/vectorize factor and granularity
  - Randomly mutate computation location
- **Crossover**



# Task Scheduler

**Problem:**

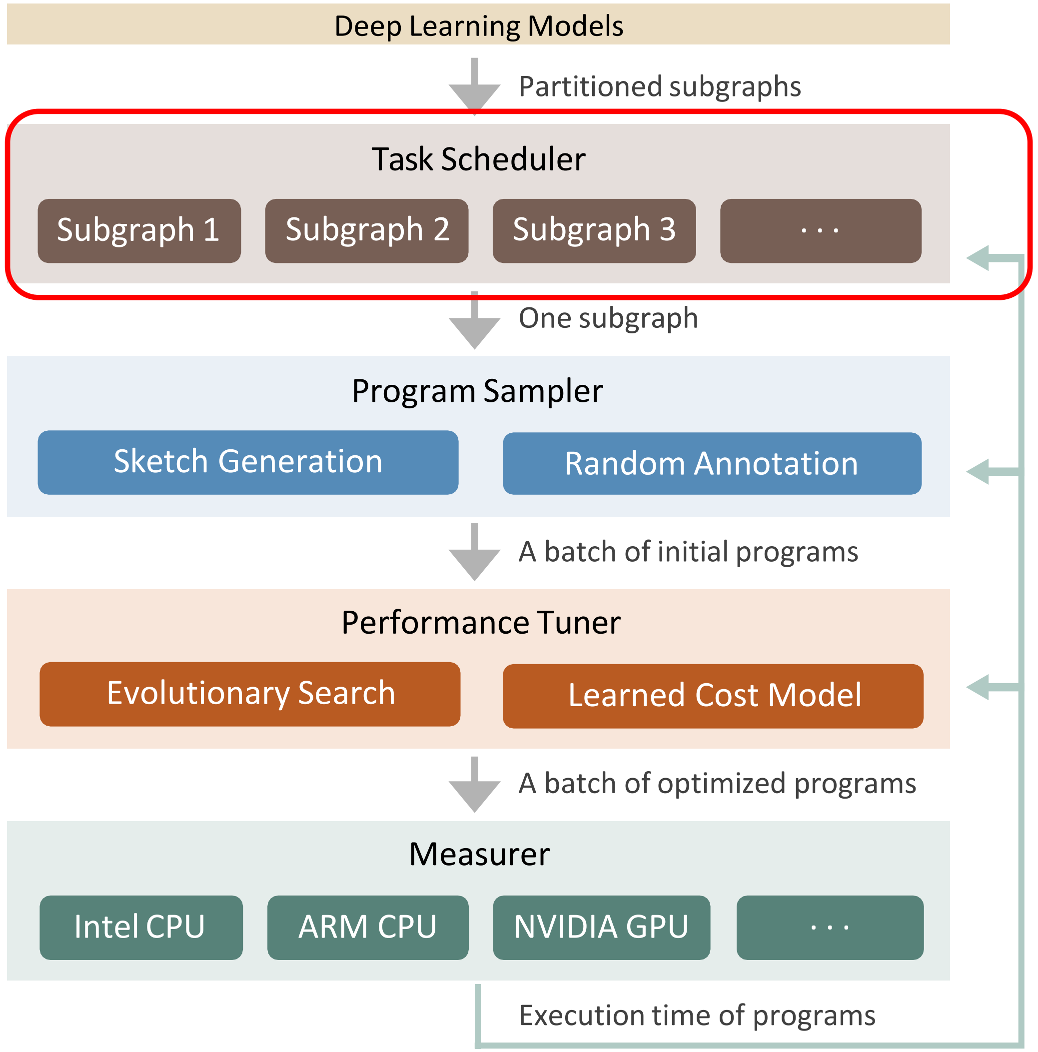
Tuning some layers may not **improve the end-to-end DNN performance**:

(1) the subgraph is not a performance bottleneck

(2) subgraph performance is not sensitive to tuning

**Approach:**

- Form an objective function
- Reschedule time budget after a time window



# Task Scheduler

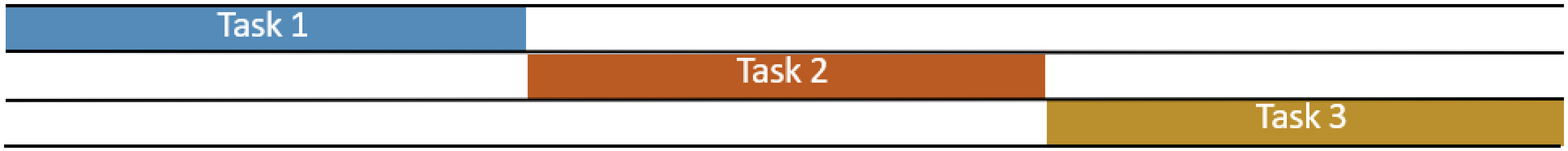
Let the subgraph ***i*** latency (the compiler has achieved so far) to be $g_i(t)$. Let the end-to-end cost of the DNNs be a function of the time of the sub-graphs $f(g_1(t),g_2(t),\ldots,g_3(t))$. Our objective is to minimize the end-to-end cost:

$$f_1 = \sum_{j=1}^{m} \sum_{i \in S(j)} w_i \times g_i(t)$$

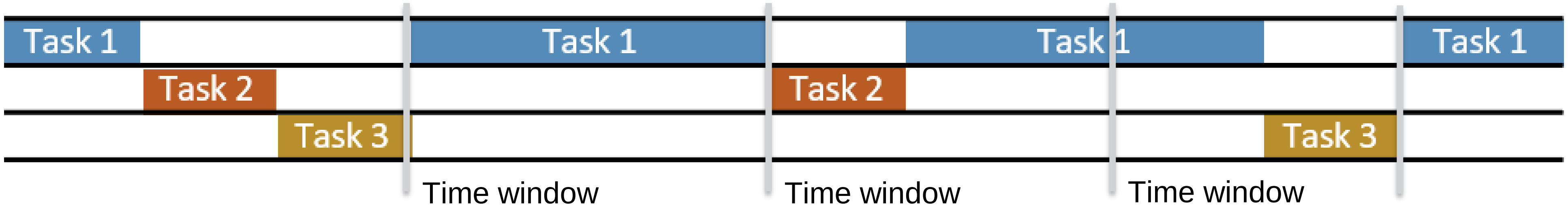
$$f_2 = \sum_{j=1}^{m} \max\left(\sum_{i \in S(j)} w_i \times g_i(t), L_j\right)$$

$$\frac{\partial f}{\partial t_i} \approx \frac{\partial f}{\partial g_i}\left(\alpha \frac{g_i(t_i) - g_i(t_i - \Delta t)}{\Delta t} + (1-\alpha)\left(\min\left(-\frac{g_i(t_i)}{t_i}, \beta \frac{C_i}{\max_{k \in N(i)} V_k} - g_i(t_i)\right)\right)\right)$$

**Existing systems**: sequential optimization with a fixed allocation




**Ansor task scheduler**: slice the time and prioritize important subgraphs



# Outline

## Background

What is tensor program? / Challenges of generating high efficiency tensor program

Current state-of-the-art of tensor program generator

## Design

Sketch and Annotation – Decoupling coarse and fine-grained program sampling

Evolutionary Search – Mutating and tuning the program inspired by biological evolution

Task Scheduler – Task scheduling to meet the code generation latency requirement

## Benchmark & Conclusion

Ansor finds high-performance programs that are ***outside the search space of existing state-of-the-art approaches***, achieving 3.8x and 1.7x over state-of-the-art tensor program generator on Intel CPU and NVIDIA GPUs.

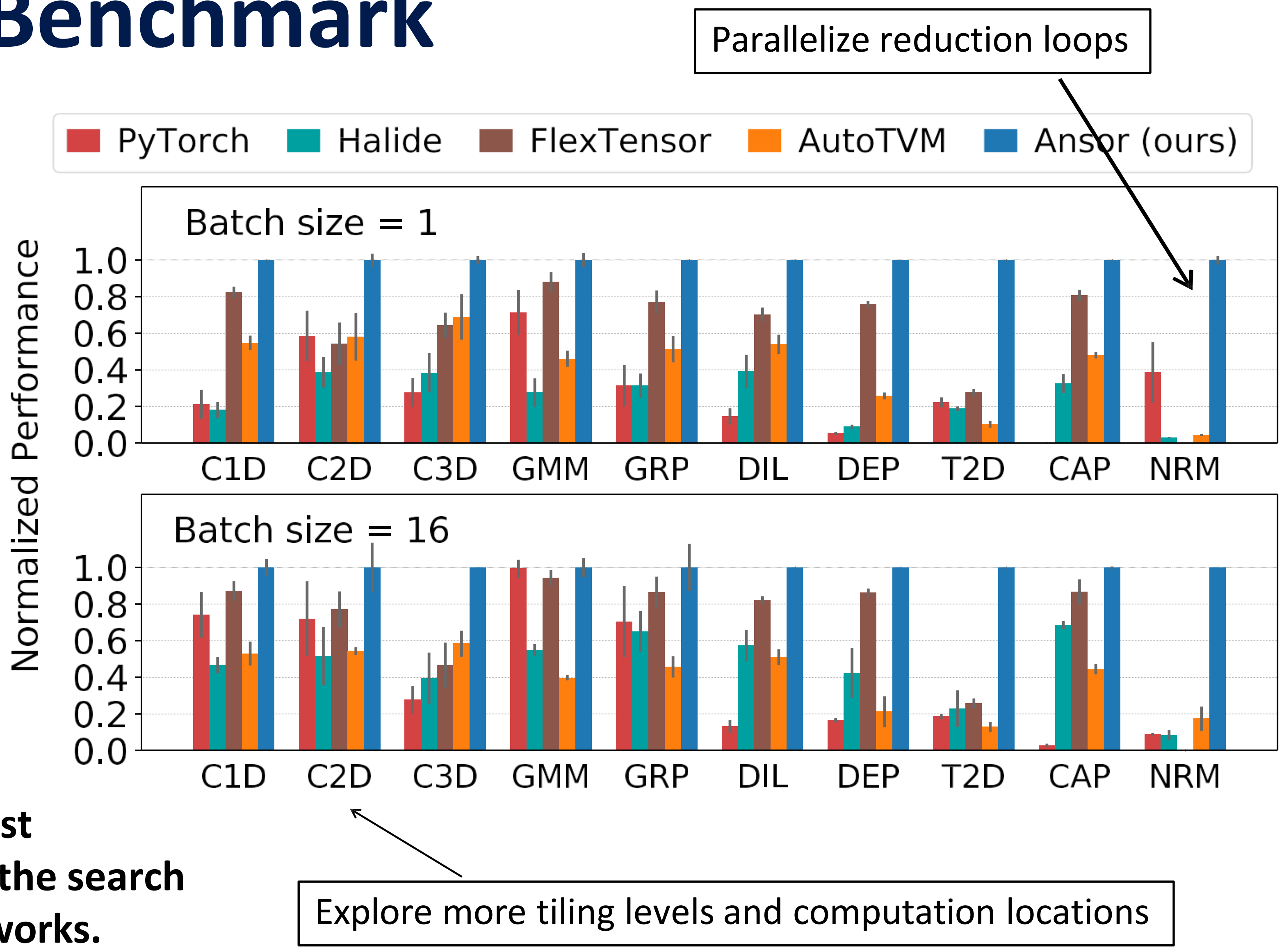
# Single Operator Benchmark

**Platform:**

Intel-Platinum 8124M (18 cores)

**Operators:**

conv1d (C1D), conv2d (C2D),
conv3d (C3D), matmul (GMM)
group conv2d (GRP),
dilated conv2d (DIL)
depthwise conv2d (DEP),
conv2d transpose (T2D),
capsule conv2d (CAP),
matrix 2-norm (NRM)




**Takeaway: For most test cases, the best programs found by Ansor are outside the search space of existing search-based frameworks.**

# Single Subgraph Benchmark

**Platforms:**

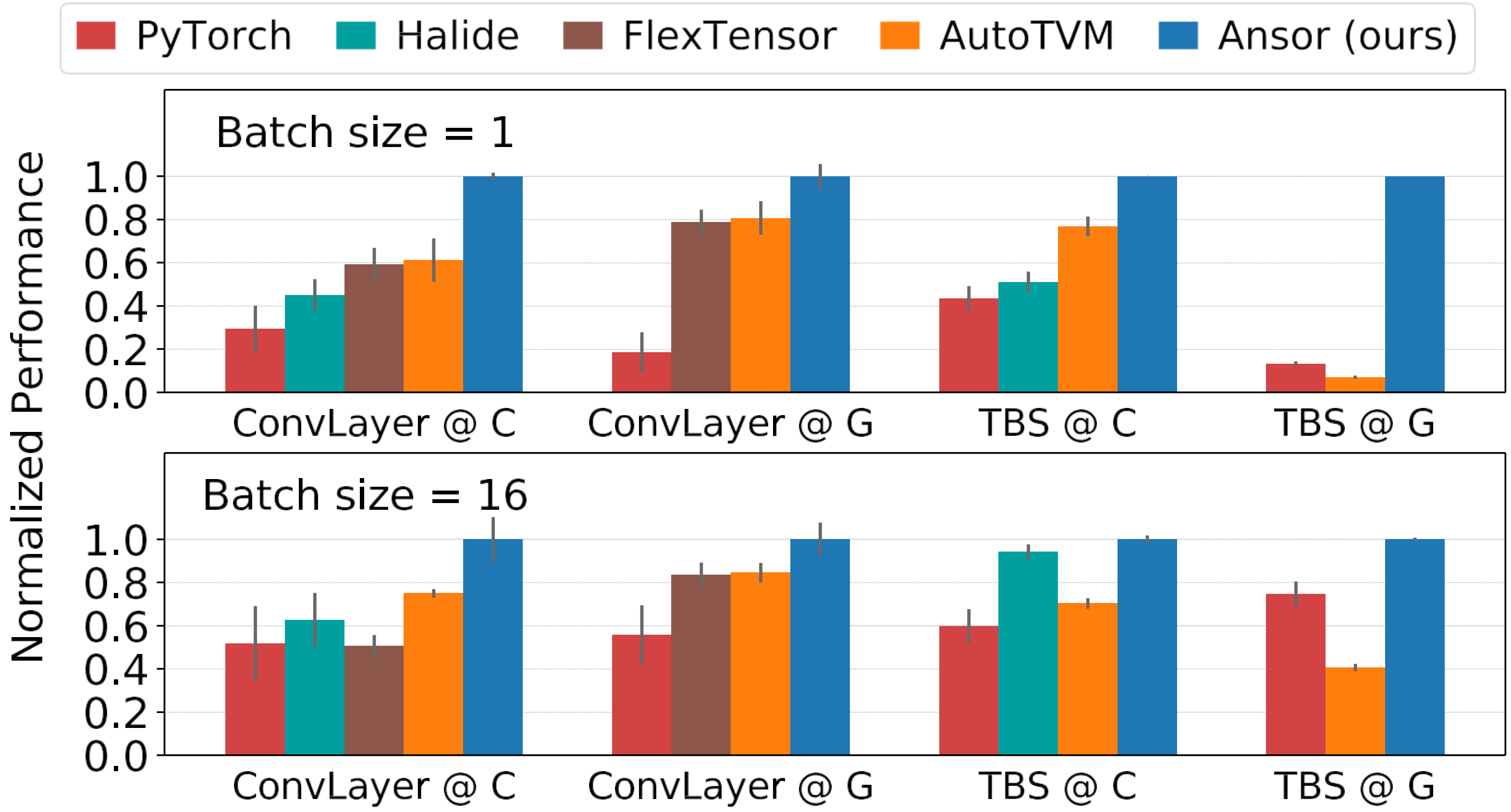
"@C" for Intel CPU (8124M)

"@G" for NVIDIA (V100)

**Subgraphs:**

ConvLayer = conv2d + bn + relu

TBS = transpose + batch_matmul + softmax




**Takeaway:**

Comprehensive coverage of the search space gives 1.1**x** -14.2**x** speedup against the best alternative.

# End-to-End Network Performance

**Platforms:**

Intel CPU (8124M)

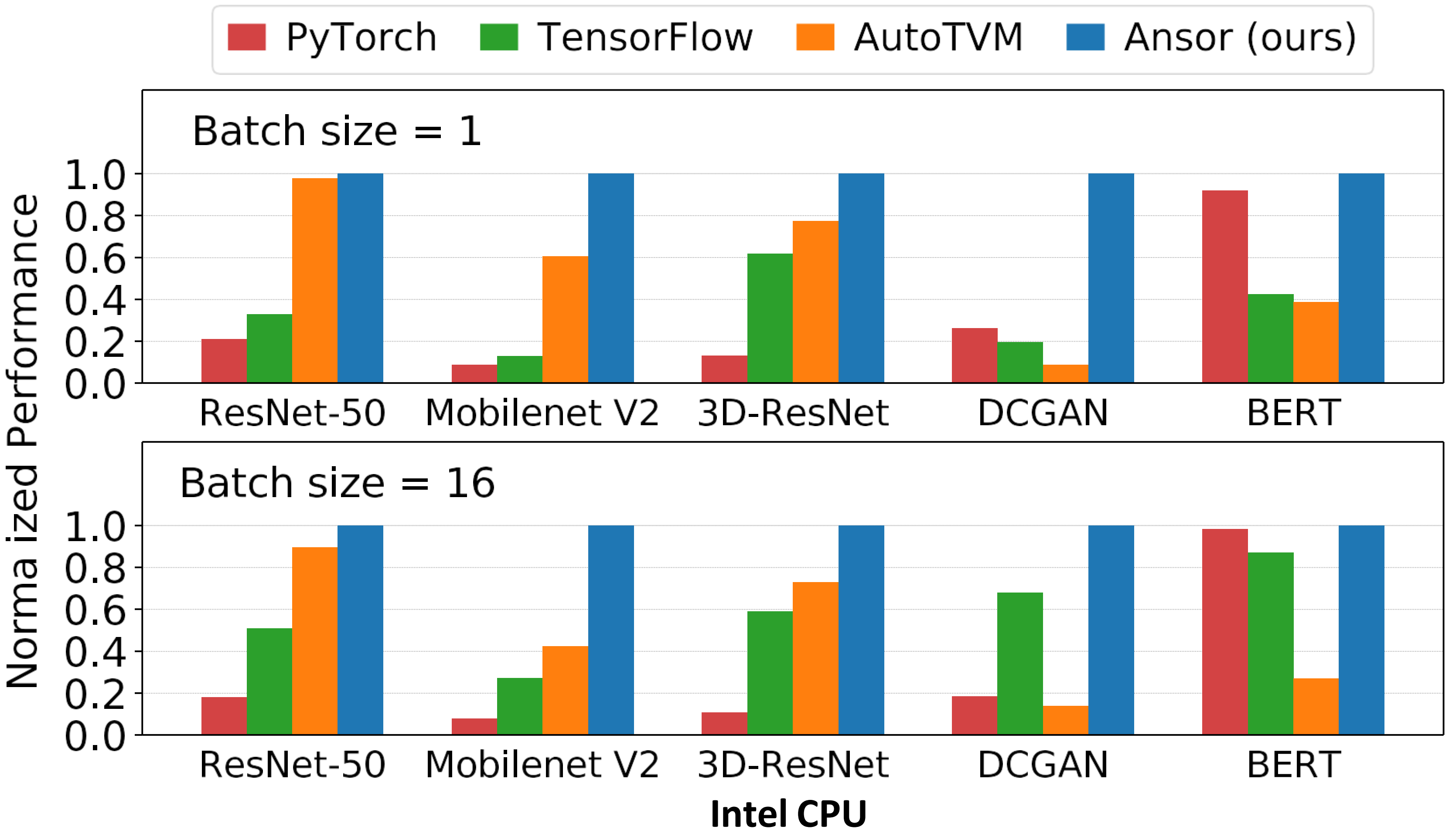
NVIDIA GPU (V100)

**Networks:**

ResNet-50, Mobilenet V2, 3D-ResNet, DCGAN, BERT

**Takeaway:**

Ansor performs best or equally the best in all test cases with up to **3.8x** speedup on Intel CPU



# End-to-End Network Performance

**Platforms:**

Intel CPU (8124M)

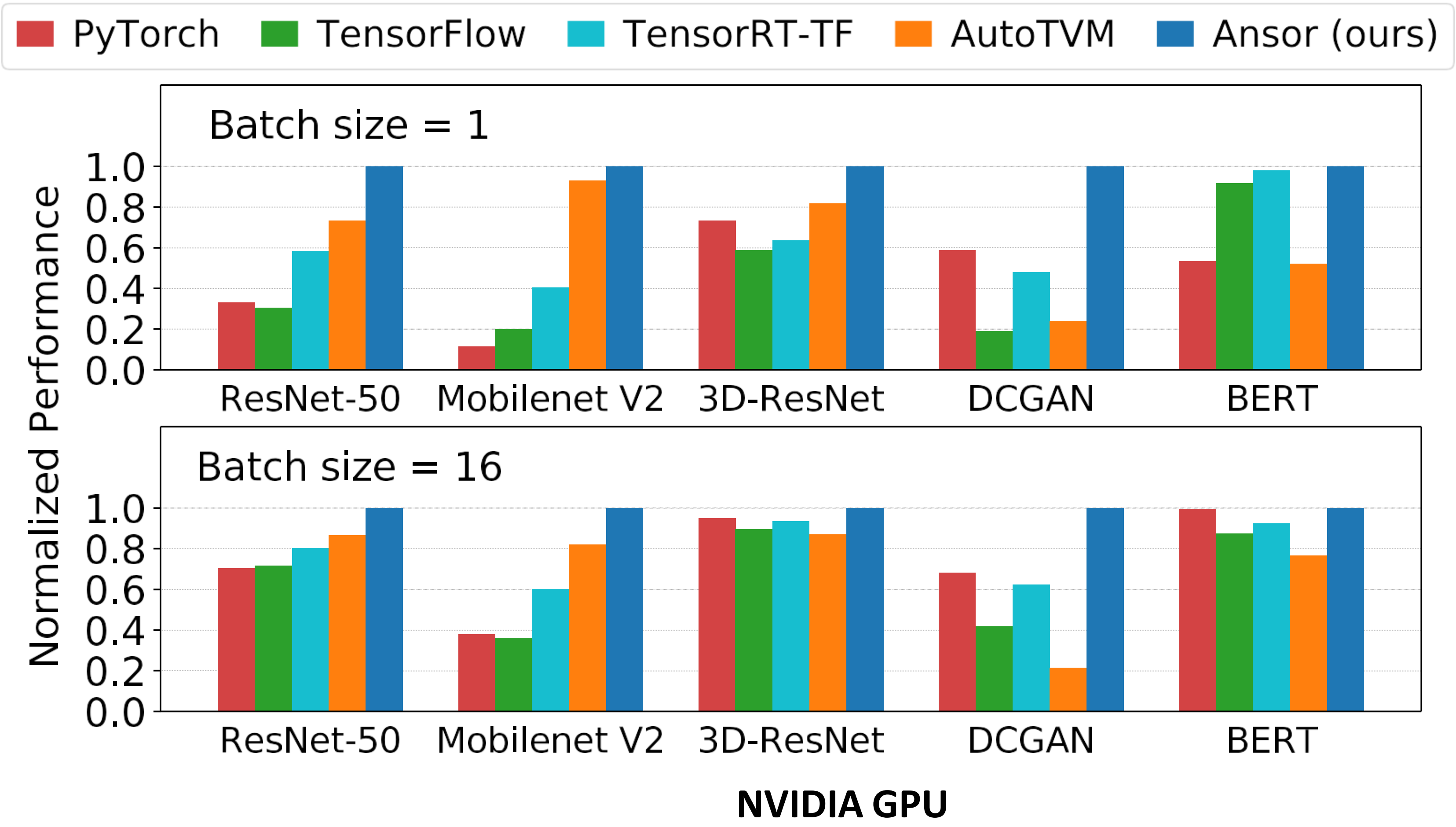
NVIDIA GPU (V100)

**Networks:**

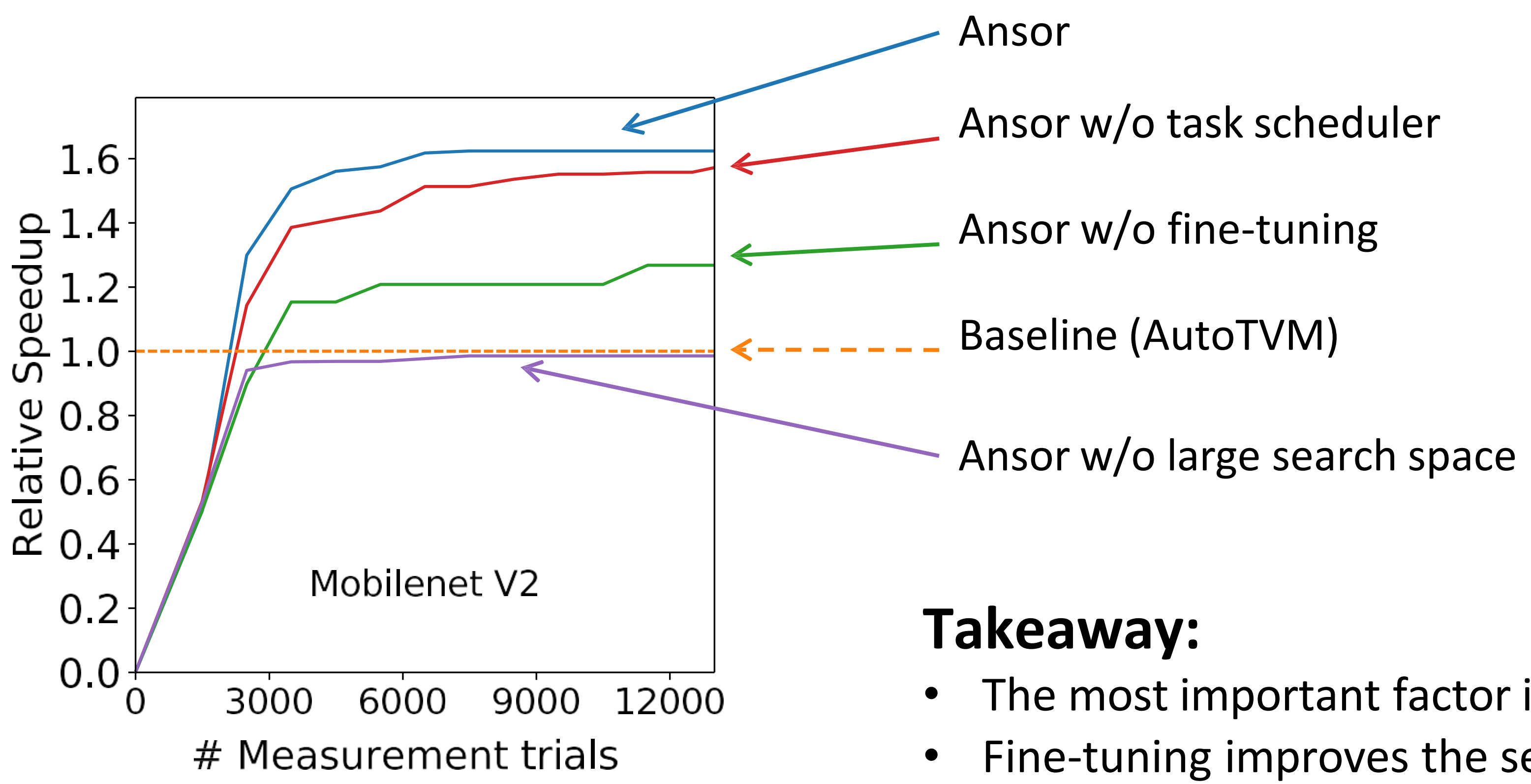
ResNet-50, Mobilenet V2, 3D-ResNet, DCGAN, BERT

**Takeaway:**

Ansor performs best or equally the best in all test cases with up to **1.7x** speedup on NVIDIA GPU



# Ablation Study




**Takeaway:**

- The most important factor is the **large search space**
- Fine-tuning improves the search results significantly
- Task scheduler accelerates the search
- Match the performance of AutoTVM with 10x less search time

# Reference


Chen, Tianqi, et al. "{TVM}: An automated end-to-end optimizing compiler for deep learning." *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*. 2018.

Zheng, Lianmin, et al. "Ansor: Generating high-performance tensor programs for deep learning." *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*. 2020. https://www.usenix.org/system/files/osdi20-zheng.pdf

Ragan-Kelley, Jonathan, et al. "Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines." *Acm Sigplan Notices* 48.6 (2013): 519-530.

Pytorch compiler: https://github.com/pytorch/glow

Kim, YeongSeog, W. Nick Street, and Filippo Menczer. "Feature selection in unsupervised learning via evolutionary search." *Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining*. 2000.

Habana AI chip, https://newsroom.intel.com/news-releases/intel-ai-acquisition/