TASO: Optimizing Deep Learning Computation with Automatic Generation of Graph Substitutions

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

Presented By
Pranav Talluri
Background

- DNNs are expressed as computation graphs
- Multiple formulations can achieve the same goal, with differing costs
- Introduces the desire to optimise DNN computation graphs
- Before TASO, the specific optimisations were manually designed by human experts
- TASO automates the generation of graph substitutions in order to programmatically optimise DNN graphs
Background
Background
Overview

• TASO automates generation of graph substitutions
• Framework agnostic (cuDNN + TVM)
• Takes operator specifications as an input
• Does so in a few stages:
  • Programmatically generate candidate graph substitutions
    • Generate
    • Quick test to prune impossible substitutions
  • Formally verify validity
  • Cost-based backtracking search to find an optimised graph
    • Includes co-optimisation of data locality
Overview

Operator Specifications -> Graph Subst. Generator (§2) -> Graph Subst. Verifier (§3) -> Verified Graph Subst. -> Joint Optimizer (§5) -> Optimized Comp. Graph
Approach: Generate Substitutions

- Substitution = source, target, mapping
- Configuration parameter dependent operators
- Generation algorithm
  - Enumerate potential graphs
  - Create graphs iteratively
  - Collect fingerprints
  - Test graphs with identical fingerprints
- Special Cases

---

**Algorithm 1** Graph substitution generation algorithm.

1: Input: A set of operators $\mathcal{P}$, and a set of input tensors $\mathcal{I}$.
2: Output: Candidate graph substitutions $\mathcal{S}$.
3: 
4: // Step 1: enumerating potential graphs.
5: $\mathcal{D} = \{\}$ // $\mathcal{D}$ is a graph hash table indexed by their fingerprints.
6: BUILD(1, 0, $\mathcal{I}$)
7: function BUILD($n$, $\mathcal{G}$, $\mathcal{I}$)
8:   if $\mathcal{G}$ contains duplicated computation then
9:     return
10: $\mathcal{D} = \mathcal{D} + \{\text{FINGERPRINT}($\mathcal{G}$), $\mathcal{G}$\}
11: if $n < \text{threshold}$ then
12:   for $op \in \mathcal{P}$ do
13:     for $i \in \mathcal{I}$ and $i$ is a valid input to $op$ do
14:       Add operator $op$ into graph $\mathcal{G}$.
15:       Add the output tensors of $op$ into $\mathcal{I}$.
16:       BUILD($n + 1$, $\mathcal{G}$, $\mathcal{I}$)
17:     Remove operator $op$ from $\mathcal{G}$.
18:     Remove the output tensors of $op$ from $\mathcal{I}$.
19: 
20: // Step 2: testing graphs with identical fingerprint.
21: $\mathcal{S} = \{\}$
22: for $\mathcal{G}_1$, $\mathcal{G}_2 \in \mathcal{D}$ with the same FINGERPRINT(·) do
23:   if $\mathcal{G}_1$ and $\mathcal{G}_2$ are equivalent for all test cases then
24:     $\mathcal{S} = \mathcal{S} + (\mathcal{G}_1, \mathcal{G}_2)$
25: return $\mathcal{S}$
Approach: Formal Verification

• Verify generated substitutions
• Operator properties expressed in FOL
  • Manually written and reviewed
  • Further validated using symbolic execution
  • Properties are added when required
  • Checked for consistency and redundancies are removed
• Uses Z3 (SMT Solver)
• Shapes of tensors are not modelled
• Data layout not included
## Approach: Formal Verification

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Tensor Operators</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ewadd</td>
<td>Element-wise addition</td>
<td></td>
</tr>
<tr>
<td>ewmul</td>
<td>Element-wise multiplication</td>
<td></td>
</tr>
<tr>
<td>smul</td>
<td>Scalar multiplication</td>
<td></td>
</tr>
<tr>
<td>transpose</td>
<td>Transpose</td>
<td></td>
</tr>
<tr>
<td>matmul</td>
<td>Batch matrix multiplication¹</td>
<td></td>
</tr>
<tr>
<td>conv</td>
<td>Grouped convolution²</td>
<td></td>
</tr>
<tr>
<td>enlarge</td>
<td>Pad conv. kernel with zeros³</td>
<td></td>
</tr>
<tr>
<td>relu</td>
<td>Relu operator</td>
<td>stride, padding, activation</td>
</tr>
<tr>
<td>poolavg</td>
<td>Average pooling</td>
<td>kernel size</td>
</tr>
<tr>
<td>poolmax</td>
<td>Max pooling</td>
<td></td>
</tr>
<tr>
<td>concat</td>
<td>Concatenation of two tensors</td>
<td></td>
</tr>
<tr>
<td>split₀</td>
<td>Split into two tensors</td>
<td></td>
</tr>
<tr>
<td>split₁</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Constant Tensors</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$C_{pool}$</td>
<td>Average pooling constant</td>
<td>kernel size</td>
</tr>
<tr>
<td>$I_{conv}$</td>
<td>Convolution id. kernel</td>
<td></td>
</tr>
<tr>
<td>$I_{matmul}$</td>
<td>Matrix multiplication id.</td>
<td></td>
</tr>
<tr>
<td>$I_{ewmul}$</td>
<td>Tensor with 1 entries</td>
<td></td>
</tr>
<tr>
<td>$I_{ewmul}$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Approach: Formal Verification

<table>
<thead>
<tr>
<th>Operator Property</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\forall x, y, z. \ ewadd(x, \ ewadd(y, z)) = \ ewadd(\ewadd(x, y), z)$</td>
<td>$\ewadd$ is associative</td>
</tr>
<tr>
<td>$\forall x, y. \ \ewadd(x, y) = \ewadd(y, x)$</td>
<td>$\ewadd$ is commutative</td>
</tr>
<tr>
<td>$\forall x, y, z. \ \ewmul(x, \ ewmul(y, z)) = \ ewmul(\ewmul(x, y), z)$</td>
<td>$\ewmul$ is associative</td>
</tr>
<tr>
<td>$\forall x, y. \ \ewmul(x, y) = \ewmul(y, x)$</td>
<td>$\ewmul$ is commutative</td>
</tr>
<tr>
<td>$\forall x, y, z. \ \ewmul(\ewadd(x, y), z) = \ ewadd(\ewmul(x, z), \ewmul(y, z))$</td>
<td>Distributivity</td>
</tr>
<tr>
<td>$\forall x, y, w. \ \smul(x, \ smul(y, w)) = \smul(x, \smul(y, w))$</td>
<td>$\smul$ is associative</td>
</tr>
<tr>
<td>$\forall x, y, w. \ \smul(\ewadd(x, y), w) = \ ewadd(\smul(x, w), \smul(y, w))$</td>
<td>Distributivity</td>
</tr>
<tr>
<td>$\forall x, y, w. \ \smul(\ewmul(x, y), w) = \ ewmul(\smul(x, w), \smul(y, w))$</td>
<td>Operator commutativity</td>
</tr>
<tr>
<td>$\forall x. \ \transpose(\transpose(x)) = x$</td>
<td>$\transpose$ is its own inverse</td>
</tr>
<tr>
<td>$\forall x, y. \ \transpose(\ewadd(x, y)) = \ewadd(\transpose(x), \transpose(y))$</td>
<td>Operator commutativity</td>
</tr>
<tr>
<td>$\forall x, y. \ \transpose(\ewmul(x, y)) = \ewmul(\transpose(x), \transpose(y))$</td>
<td>Operator commutativity</td>
</tr>
<tr>
<td>$\forall x, w. \ \transpose(\smul(x, w)) = \smul(\transpose(x), \transpose(w))$</td>
<td>Operator commutativity</td>
</tr>
<tr>
<td>$\forall x, y, z. \ \matmul(x, \ matmul(y, z)) = \ matmul(\matmul(x, y), z)$</td>
<td>$\matmul$ is associative</td>
</tr>
<tr>
<td>$\forall x, y, w. \ \matmul(\ewadd(x, y), w) = \ ewadd(\matmul(x, w), \matmul(y, w))$</td>
<td>$\matmul$ is linear</td>
</tr>
<tr>
<td>$\forall x, w. \ \matmul(\ewmul(x, y), w) = \matmul(\ewmul(x, \smul(y, w)))$</td>
<td>$\matmul$ is linear</td>
</tr>
</tbody>
</table>

...
Approach: Pruning Redundant Substitutions

- Redundant substitutions are subsumed by more general, valid substitutions
- Input tensor renaming
- Common subgraph
Approach: Joint Optimisation

• Utilises MetaFlow cost-based backtracking search algorithm
• Considers data layout optimisation opportunities
• Joint optimisation uncovers otherwise impossible optimisations
• Costs are given by execution times of specific operators
• Cycle removal
• Alpha parameter prunes search space

Algorithm 2 Cost-Based Backtracking Search

1: **Input:** an input graph $G_{in}$, verified substitutions $S$, a cost model $\text{Cost}(\cdot)$, and a hyper parameter $\alpha$.
2: **Output:** an optimized graph.
3:
4: $P = \{G_{in}\}$ // $P$ is a priority queue sorted by $\text{Cost}$.
5: while $P \neq \{\}$ do
6:   $G = P$.dequeue()
7:   for substitution $s \in S$ do
8:     // $\text{LAYOUT}(G, s)$ returns possible layouts applying $s$ on $G$.
9:     for layout $l \in \text{LAYOUT}(G, s)$ do
10:        // $\text{APPLY}(G, s, l)$ applies $s$ on $G$ with layout $l$.
11:        $G' = \text{APPLY}(G, s, l)$
12:        if $G'$ is valid then
13:           if $\text{Cost}(G') < \text{Cost}(G_{opt})$ then
14:              $G_{opt} = G'$
15:           if $\text{Cost}(G') < \alpha \times \text{Cost}(G_{opt})$ then
16:              $P$.enqueue($G'$)
17:            return $G_{opt}$
Evaluation: Optimisation

- Setup – tested on 5 DNNs
- Successful automatic optimisation – inference time reduction
  - cuDNN: 1.3x to 2.8x
  - TVM: 1.1x to 1.8x
Evaluation: Substitutions

- NasNet was produced using neural architecture search
- Unconventional optimisations were discovered
Evaluation: Substitutions

- Different DNNs used different optimisations, showing usefulness of TASO
- Scalability
  - Larger operator substitutions could be useful
Evaluation: Substitutions

- Joint optimisation
  - Better than individual or sequential
  - $(A \times B) \rightarrow ((B^T \times A^T)^T)$ with $B^T$ in row-major and $A^T$ in column-major
  - Phase ordering?
- Relatively quick - <10 minutes for each DNN

![Execution Time Graph](image-url)
Review

Positives
• Novel idea
• Successful execution
  • Improves DNN performance
  • Reduces human effort
  • Extensible framework
• Seminal work in an exciting research area
  • Graph transformation backend still in use

Negatives
• Reliant on user provided operator properties
• Scalability of generator
• Phase ordering problem + search procedure
• Cost model has issues
Future Works

• Future works have built on this approach
• PET
  • Partially equivalent optimisations
• TENSAT
  • Equality saturation
• X-RLflow
  • RL approach to searching optimisation space
• REGAL
  • Transfer knowledge