Equality Saturation For Tensor Graph Superoptimization

Authors: Yang, Y., Phothilimthana, P. M., Wang, Y. R., Willsey, M., Roy, S., and Pienaar, J.
Year of Publication: 2021

Presenter: Hetong Shen
CRSid: HS899
TENSAT, a tensor graph superoptimization framework that employs equality saturation on E-Graphs.
TENSAT, a tensor graph superoptimization framework that employs equality saturation on E-Graphs.
TENSAT, a tensor graph superoptimization framework that employs equality saturation on E-Graphs.

- Enumerate through potential substitutions of graphs and find the optimal one
TENSAT, a tensor graph superoptimization framework that employs equality saturation on **E-Graphs**.

- Term Rewriting \((a \cdot 2)/2 \rightarrow a\)

  **Useful**
  
  \[
  (x \cdot y)/z = x \cdot (y/z)
  
  x/x = 1
  \]

  \[
  (a \cdot 2)/2 \rightarrow a \cdot (2/2) \rightarrow a
  \]

  **Not useful**

  \[
  x \cdot 2 = x << 1
  
  x \cdot y = y \cdot x
  \]

  \[
  (a \cdot 2)/2 \rightarrow (a << 1)/2
  \]
TENSAT, a tensor graph superoptimization framework that employs equality saturation on E-Graphs.

- E-graphs: \( (a \cdot 2)/2 \)
- Term rewriting: \( (a \cdot 2)/2 \rightarrow (a\ll1)/2 \)

This e-class represents \((a \cdot 2) / 2\) and \((a \ll 1) / 2\)

This e-class represents \((a \cdot 2)\) and \((a \ll 1)\)
TENSAT, a tensor graph superoptimization framework that employs equality saturation on E-Graphs.

- Grow a E-graph

(a) Initial e-graph contains \((a \times 2)/2\).
(b) After applying rewrite \(x \times 2 \rightarrow x \ll 1\).
(c) After applying rewrite \((x \times y)/z \rightarrow x \times (y/z)\).
(d) After applying rewrites \(x/x \rightarrow 1\) and \(1 \times x \rightarrow x\).
TENSAT, a tensor graph superoptimization framework that employs **equality saturation** on E-Graphs.

- Equality Saturation

\[
x \cdot 2 \rightarrow x << 1
\]
\[
(x \cdot y) / z \rightarrow x \cdot (y / z)
\]
\[
x / x \rightarrow 1
\]
\[
x \cdot 1 \rightarrow x
\]
TENSAT, a tensor graph superoptimization framework that employs **equality saturation** on E-Graphs.

- Equality Saturation

```
Initial Term → E-graph → Optimized term
```

- Find a pattern
- Apply a match
- Restore invariants
Challenges

- When doing graph rewriting to determine the order of applying the rewrite rules:
  - manually curated set of rewrite rules.
  - heuristic.

- However, sequential substitution often leads to sub-optimal:
  - The non-comprehensive set of rewrite rules.
  - The sub-optimal graph substitution heuristic.
  - Rule choice problem

Sequential Substitution
Existing Works

- Graph Rewrite Optimizations
  - TASO
  - NeuRewriter
- Superoptimization
  - Short sequences of low-level instructions
  - Denali
- Equality Saturation Applications
  - Optimize in other fields: ML, CAD simplification, Numerical Accuracy.

TENSAT:
Re-implementation of the TASO compiler using equality saturation
## TENSAT’s Representations

### Representing Tensor Computation Graphs

| Operator | Description                          | Inputs                                                                 | Type signature            |
|----------|--------------------------------------|-----------------------------------------------------------------------|---------------------------|
| ewadd    | Element-wise addition                | input₁, input₂                                                       | (T, T) → T               |
| ewmul    | Element-wise multiplication          | input₁, input₂                                                       | (T, T) → T               |
| matmul   | Matrix multiplication                | activation, input₁, input₂                                           | (N, T) → T               |
| conv     | Grouped convolution                  | strideₜ, strideₜ, pad., act., input, weight                            | (N, N, N, N, T, T) → T   |
| relu     | Relu activation                      | input                                                                 | T → T                    |
| tanh     | Tanh activation                      | input                                                                 | T → T                    |
| sigmoid  | Sigmoid activation                   | input                                                                 | T → T                    |
| poolmax  | Max pooling                          | input, kernel{h, w}, stride{h, w}, pad., act.                         | (T, N, N, N, N, N, N) → T|
| poolavg  | Average pooling                      | input, kernel{h, w}, stride{h, w}, pad., act.                         | (T, N, N, N, N, N, N) → T|
| transpose| Transpose                            | input, permutation                                                    | (T, S) → T               |
| enlarge  | Pad a convolution kernel with zeros  | input, ref-input                                                      | (T, T) → T               |
| concat   | Concatenate                           | axis, input₁, . . . , inputₙ                                           | (N, T, . . . , T) → T     |
| split    | Split a tensor into two              | axis, input                                                           | (N, T, . . . ) → T        |
| split₀   | Get the first output from split      | input                                                                 | TT → T                   |
| split₁   | Get the second output from split     | input                                                                 | TT → T                   |
| merge    | Update weight to merge grouped conv  | weight, count                                                         | (T, N) → T               |
| reshape  | Reshape tensor                       | input, shape                                                          | (T, S) → T               |
| input    | Input tensor                          | identifier                                                           | S → T                    |
| weight   | Weight tensor                         | identifier                                                           | S → T                    |
| noop     | Combine the outputs of the graph     | input₁, input₂                                                       | (T, T) → T               |
TENSAT’s Representations

• Representing Rewrite Rules
  • Single pattern rewrite rules
  • Multiple pattern rewrite rules

Source: (matmul ?input₁ ?input₂), (matmul ?input₁ ?input₃)
Target: (split₀ (split 1 (matmul ?input₁ (concat₂ 1 ?input₂ ?input₃)))),
(split₁ (split 1 (matmul ?input₁ (concat₂ 1 ?input₂ ?input₃)))))
TENSAT

• Rule choice problem
  • Solution: first generates all rewritten terms, leaving the choice of which term to select to the extraction procedure

• Exploration Phase
• Extraction Phase
Exploration Phase

- Search for matches of all rewrite rules in the current e-graph, and add the target patterns and equivalence relations to the e-graph
  - Single pattern rewrite rules and Multiple pattern rewrite rules

---

Algorithm 1 Applying multi-pattern rewrite rules

**Input:** starting e-graph $G$, set of multi-pattern rewrite rules $\mathcal{R}_m$.  
**Output:** updated e-graph $G$.

1. canonicalized S-expr $e_c = \text{Set}()$
2. for rule $r \in \mathcal{R}_m$ do
3.     for $i = 0, \dotsc, |r| - 1$ do $\triangleright |r|$: #S-exprs in source pattern
4.         $(e, \text{rename\_map}) = \text{CANONICAL}(r.\text{source}[i])$
5.         $e_c.\text{insert}(e)$
6.         $r.\text{map}[i] = \text{rename\_map}$
7.     end for
8. end for
9. for iter = 0, $\dotsc$, MAX ITER do
10.    $M = \text{SEARCH}(G, e_c)$ $\triangleright$ all matches for all patterns
11. for rule $r \in \mathcal{R}_m$ do
12.     for $i = 0, \dotsc, |r| - 1$ do
13.         canonical matches $mc_i = M[r.\text{source}[i]]$
14.         matches $m_i = \text{DECANONICAL}(mc_i, r.\text{map}[i])$
15.     end for
16.     for $(\sigma_0, \dotsc, \sigma_{|r| - 1}) \in m_0 \times \cdots \times m_{|r| - 1}$ do
17.         if COMPATIBLE($\sigma_0, \dotsc, \sigma_{|r| - 1}$) then
18.             APPLY($G, r, \sigma_0, \dotsc, \sigma_{|r| - 1}$)
19.         end if
20. end for
21. end for
22. return $G$
Extraction Phase – 1st Approach Greedy

• Cost Model

• Greedy Extraction:
  • For each e-class, computes the total cost of the subtrees rooted on each of the e-nodes, and picks the e-node with the smallest subtree cost
  • Not guaranteed to extract the graph with the minimum cost
Extraction Phase – 2\textsuperscript{nd} Approach ILP

• ILP Extraction:
  • Objective function and constraints

Minimize: \( f(x) = \sum_i c_i x_i \)

Subject to:

\( x_i \in \{0, 1\}, \quad (1) \)
\( \sum_{i \in e_0} x_i = 1, \)  \( (2) \)
\( \forall i, \forall m \in h_i, x_i \leq \sum_{j \in e_m} x_j, \)  \( (3) \)
\( \forall i, \forall m \in h_i, t_{g(i)} - t_m - \epsilon + A(1 - x_i) \geq 0, \quad (4) \)
\( \forall m, 0 \leq t_m \leq 1, \)  \( (5) \)
Extraction Phase – 2\textsuperscript{nd} Approach ILP

- ILP Extraction:
  - Objective function and constraints
  - Cycles
Extraction Phase – 2\textsuperscript{nd} Approach ILP

- **ILP Extraction:**
  - **Objective function and constraints**
  - **Have cycles vs. no cycles**

Minimize: \( f(x) = \sum_{i} c_i x_i \)

Subject to:

\[
\begin{align*}
x_i & \in \{0, 1\}, \\
\sum_{i \in e_0} x_i & = 1, \\
\forall i, \forall m \in h_i, x_i & \leq \sum_{j \in e_m} x_j,
\end{align*}
\]

Table 5. Effect of whether or not to include cycle constraints in ILP on extraction time (in seconds), on BERT, NasRNN, and NasNet-A. For the cycle constraints, we compare both using real variables and using integer variables for the topological order variables \( t_m \).
Extraction Phase – Comparison

• Greedy vs. ILP Extraction:
  • Greedy extraction is slow: it makes the choices on which node to pick separately and greedily, without considering the interdependencies between the choices.
  • ILP Guaranteed to give a valid graph (no cycles) with the lowest cost

| Graph Runtime (ms) | Original | Greedy | ILP |
|--------------------|----------|--------|-----|
| BERT               | 1.88     | 1.88   | 1.73|
| NasRNN             | 1.85     | 1.15   | 1.10|
| NasNet-A           | 17.8     | 22.5   | 16.6|
Bottle Neck and Cycle Filtering

- Vanilla cycle filtering:
- Efficient cycle filtering in exploration phase:
  - Pre-filtering
  - Post processing

Algorithm 2 Exploration phase with efficient cycle filtering

Input: starting e-graph $G$, set of rewrite rules $R$.
Output: updated e-graph $G$, filter list $l$.

1: $l = \emptyset$
2: for iter = 0, ..., MAX_ITER do
3:    descendants map $d = \text{GETDESCENDANTS}(G, l)$
4:    matches = $\text{SEARCH}(G, R, l)$
5:    for match $\in$ matches do
6:       if not $\text{WILLCREATECYCLE}$(match, $d$) then
7:          $\text{APPLY}(G, \text{match})$
8:       end if
9:    end for
10:   while true do
11:    cycles = $\text{DFSGETCYCLES}(G, l)$
12:    if len(cycles) == 0 then
13:       break
14:    end if
15:    for cycle $\in$ cycles do
16:       $\text{RESOLVECYLE}(G, l, \text{cycle})$
17:    end for
18:   end while
19: end for
20: return $G, l$
Bottle Neck and Cycle Filtering

- Vanilla cycle filtering vs. Efficient cycle filtering

| Exploration time (s) | $k_{\text{multi}}$ | Vanilla | Efficient |
|----------------------|--------------------|---------|-----------|
| BERT                 |                    |         |           |
| 1                    | 0.18               | 0.17    |           |
| 2                    | 32.9               | 0.89    |           |
| NasRNN               |                    |         |           |
| 1                    | 1.30               | 0.08    |           |
| 2                    | 2932               | 1.47    |           |
| NasNet-A             |                    |         |           |
| 1                    | 3.76               | 1.27    |           |
| 2                    | $>3600$            | 8.62    |           |

*Table 6. Comparison between vanilla cycle filtering and efficient cycle filtering, on the exploration phase time (in seconds) for BERT, NasRNN, and NasNet-A.*
Evaluation – Set Up

• TENSAT Implementation:
  • Developed in Rust
  • Equality saturation library egg

• ILP solver:
  • Utilized SCIP
Evaluation – Set Up

The models evaluated:

• BERT (Devlin et al., 2019)
• ResNeXt-50 (Xie et al., 2017)
• NasNet-A (Zoph et al., 2018)
• NasRNN (Zoph & Le, 2017)
• Inception v3 (Szegedy et al., 2016)
• VGG-19 (Liu & Deng, 2015)
• SqueezeNet (Iandola et al., 2017)

• Limit the number of nodes in the e-graph $N_{max} = 50000$
• Limit number of iterations for exploration $k_{max} = 15$
Evaluation – Speed Up

- TASO vs TENSAT
- Equality saturation covers a much larger space of equivalent graphs than sequential backtracking search.
- K: K multi
- Inception: Optimizer can achieve a better speedup given longer optimization time.

*Figure 4.* Speedup percentage of the optimized graph with respect to the original graph: TASO vs TENSAT. Each setting (optimizer × benchmark) is run for five times, and we plot the mean and standard error for the measurements.
Evaluation – Optimization Time

• TASO vs TENSAT
• TENSAT can not only cover a much larger search space, but also in less time

Figure 5. Optimization time (log scale): TASO v.s. TENSAT. “TASO total” is the total time of TASO search. “TASO best” indicates when TASO found its best result; achieving this time would require an oracle telling it when to stop.
Evaluation – Varying Iterations of Multi-Pattern Rewrites

- Effect of varying the number of iterations of multi-pattern rewrites $k_{multi}$
- Squeeze-Net: discrepancy between the cost model and the real graph runtime.
Novelty
• Uses e-graph for tensor graph superoptimization
• Introduces multi pattern write rules
• Efficient cycle filtering in exploration phase

Downside
• Limitation in Scalability:
  • Multi-pattern rules for tensor graph: grow the e-graph extremely rapidly
  • Can only explore up to a certain number of iterations of multi-pattern rewrites.
  • E-graph becomes too large for the extraction phase
• Parallelism:
  • Uses cost model as TASO, which is suitable for GPU (one operator when executing graph)
Impact and Future directions

• Tackle Limitation in Scalability:
  • Selectively apply rules during exploration
  • Utilize ML techniques
• Achieve Parallelism:
  • Some hardware may execute multiple kernels in parallel
  • Needs a different cost model, such as a learned method to perform extractions
• Applications:
  • TENSAT’s optimization time is small enough that can be integrated into a default compilation flow