Recent Trend in Machine Learning Compilers: A Survey

Bojian Zheng, Shang Wang, Gennady Pekhimenko

EcoSystem Research Group
Department of Computer Science, University of Toronto
bojian,wangsh46,pekhimenko@cs.toronto.edu
Abstract: Why Machine Learning Compilers?

- Machine Learning Frameworks are NOT the end of story!
  - Performance
  - Hardware Backend Portability
- Machine Learning Compilers create the separation between what to compute (Algorithms) and how to compute (Schedules).
Background: **Machine Learning**

**Machine Learning** has applications in many domains.

- **Image Classification**
  - airplane
  - automobile
  - bird
  - cat
  - deer
  - dog
  - frog
  - horse
  - ship
  - truck

- **Speech Recognition**

- **Machine Translation**
  - Google Translate
Background: Machine Learning Frameworks

Machine Learning Applications are usually developed under Machine Learning Frameworks (Tensorflow, PyTorch, MXNet, CNTK), which can be viewed as a language wrapper on top of the Operator Pool.
Background: **Machine Learning Frameworks**

The **Operator Pool** consists of implementations of machine learning operators that are **tuned in depth**, either by framework developers or vendor libraries.
Background: **Machine Learning Frameworks**

Machine learning practitioners build up **Computation Graphs** for development. Each node is an **instantiation** of the operator with specific parameters.

---

Problem Statement

So what are the problems with Machine Learning Frameworks?

✖ Performance
- Implementing NEW operators using EXISTING ones are costly.
- EXISTING operators CANNOT guarantee best performance for every input case (dimensions, data types, ...).

✖ Hardware Backend Portability
- Many hardwares targeting machine learning workloads.
- E.g., CPUs, GPUs, TPU, FPGAs, ASICs …
- Different hardwares require different management policy.
- Hard for those frameworks to be ported to new hardwares.
Problem Statement: Performance

“Implementing NEW operators using EXISTING ones are costly.”

Extra Overheads for Splitting/Joining Threads & Reading/Writing tmp
Problem Statement

So what are the problems with Machine Learning Frameworks?

✖ Performance
  ○ Implementing NEW operators using EXISTING ones are costly.
  ○ EXISTING operators CANNOT guarantee best performance for every input case (dimensions, data types, ...).

✖ Hardware Backend Portability
  ○ Many hardwares targeting machine learning workloads.
  ○ E.g., CPUs, GPUs, TPU, FPGAs, ASICs …
  ○ Different hardwares require different management policy.
  ○ Hard for those frameworks to be ported to new hardwares.
Tensorflow XLA

Tensorflow XLA consists of two separate components:

- **Ahead-Of-Time** Compiler `tfcompile`:
  - **Major Goal**: Reduce size of executables on mobile devices.
  - **Idea**: Applications ONLY take subroutines that they need, rather than the entire Operator Pool.

- **Just-In-Time** Compiler XLA:
  - **Major Goal**: Improve performance and portability.
  - **Idea**: Similar Design with LLVM
  - Currently, **Kernel Fusion** is the major optimization that XLA does.
TensorComprehensions

- TensorComprehensions is mainly focusing on runtime performance optimization on GPUs.

- More specifically, it auto-tunes each individual operator.
  - This is different from auto-scheduling that optimizes across multiple different operators.

- Two Major Highlights: (1) Code Generation (2) Auto-tuning

TensorComprehensions: Code Generation

Example: Writing a Matrix Multiply using `TensorComprehensions`:

```python
# import tc, torch
e = tc.ExecutionEngine()
# define a new matmul operator
e.define("""
def mm(float(M, K) A, float(K, N) B) -> (C)
{
    C(m, n) +=! A(m, kk) * B(kk, n)
}""")
A = torch.randn(3, 4)
B = torch.randn(4, 5)
C = ee.mm(A, B) # use the newly defined operator
```
TensorComprehensions: Code Generation

Code Generation of TensorComprehensions relies on PPCG, the Polyhedral Parallel Code Generator for CUDA. Features: (1) Polyhedral Scheduling (2) GPU Mapping (3) Memory Promotion
TensorComprehensions: Auto-tuning

Compilation jobs → CPUs → Profiling jobs → Tuning Database → Search Strategy → GPUs
TensorComprehensions: **Auto-tuning**

```python
def avgpool(float(B, C, H, W) input) -> (output) {{
    output(b, c, h, w) += input(b, c, h * {sH} + kh, w * {sW} + kw)
    where kh in 0:{kH}, kw in 0:{kW}
}}
```

Tensor Comprehension for 2D Average Pooling
Case Study: Two-stage Blur Filter

Consider a image processing pipeline consisted of a two-stage blur filter:

\[
\begin{align*}
\text{blur}(x, y) &= \text{in}(x-1, y) + \text{in}(x, y) + \text{in}(x+1, y) \\
\text{out}(x, y) &= \text{blur}(x, y-1) + \text{blur}(x, y) + \text{blur}(x, y+1)
\end{align*}
\]

What are the possible implementations?
Case Study: Two-stage Blur Filter

Parallelism, Locality, No Redundant Computation

```
breadth first: each function is entirely evaluated before the next one.
```

```
alloc blurx[2048][3072]
for each y in 0..2048:
    for each x in 0..3072:
        blurx[y][x] = in[y][x-1] + in[y][x] + in[y][x+1]
 alloc out[2046][3072]
for each y in 1..2047:
    for each x in 0..3072:
        out[y][x]=blurx[y-1][x] + blurx[y][x] + blurx[y+1][x]
```
Case Study: Two-stage Blur Filter

Parallelism, Locality, No Redundant Computation

```
alloc out[2046][3072]
for each y in 1..2047:
    for each x in 0..3072:
        alloc blurx[-1..1]
        for each i in -1..1:
            blurx[i]= in[y-1+i][x-1]+in[y-1+i][x]+in[y-1+i][x+1]
```
Case Study: Two-stage Blur Filter

Parallelism, Locality, No Redundant Computation

```
alloc out[2046][3072]
alloc blurx[3][3072]
for each y in -1..2047:
    for each x in 0..3072:
        blurx[(y+1)%3][x]=in[y+1][x-1]+in[y+1][x]+in[y+1][x+1]
        if y < 1: continue
        out[y][x] = blurx[(y-1)%3][x]
            + blurx[ y % 3 ][x]
            + blurx[(y+1)%3][x]
```
Case Study: Two-stage Blur Filter

Finding optimal kernel by manually trying different combinations.
Halide

- Separation between **functional correctness** and **optimization**:
  - Algorithm: What to compute.
  - Schedule: How to compute.
- (Original) Usage: stencil computations and stream programs.
- Application: Image processing (graphics).
- Motivation: Writing optimized image processing kernels is **very hard**!
A Halide Program

**Input: Algorithm**

\[ \text{blurx}(x,y) = \text{in}(x-1,y) \]
\[ + \ \text{in}(x,y) \]
\[ + \ \text{in}(x+1,y) \]

\[ \text{out}(x,y) = \text{blurx}(x,y-1) \]
\[ + \ \text{blurx}(x,y) \]
\[ + \ \text{blurx}(x,y+1) \]

**Input: Schedule**

- \text{blurx}: split \( x \) by 4 → \( x_0, x_1 \)
- vectorize: \( x_1 \)
- store at \( \text{out}.x_0 \)
- compute at \( \text{out}.y_1 \)

- \text{out}: split \( x \) by 4 → \( x_0, x_1 \)
- split \( y \) by 4 → \( y_0, y_1 \)
- reorder: \( y_0, x_0, y_1, x_1 \)
- parallelize: \( y_0 \)
- vectorize: \( x_1 \)
But how to schedule?

- Heuristic Cost Model

\[
\text{Cost} = (\text{Number of arithmetic operations}) + (\text{Number of memory accesses}) \times (\text{LOAD COST})
\]

\[
\text{Benefit}(A, B) = \text{Cost}(A) + \text{Cost}(B) - \text{Cost}(A, B)
\]

- Greedy Grouping until no positive benefit

TVM

- Like Halide (not exactly), but in machine learning (ML) domain.
- ML operators (conv2d, relu, etc.) implemented as **Algorithm + Schedule**.
  - Targeted at various hardware backends.
  - Optimal schedules are different for different hardware backends.

TVM

- **Graph Level Optimization:**
  - NNVM
  - Specific Handwritten Rules

- **Operator Level Optimization:**
  - TVM
  - More schedule primitives targeted at various hardware backends.
Operator Schedule Auto-tuning

- Define a template schedule.
- Automatic Schedule Tuning:
  - Cost Model: Gradient Tree Boosting & TreeRNN
  - Exploration: Simulated Annealing Algorithm
Operator Schedule Auto-tuning

# Schedule.
y, x = s[C].op.axis
k = s[C].op.reduce_axis[0]
# Get the config object.
cfg = autotvm.get_config()
# Define search space.
cfg.define_knob("tile_y", [1, 2, 4, 8, 16])
cfg.define_knob("tile_x", [1, 2, 4, 8, 16])
# Schedule according to config.
yo, yi = s[C].split(y, cfg['tile_y'].val)
xo, xi = s[C].split(x, cfg['tile_x'].val)
s[C].reorder(yo, xo, k, yi, xi)
Halide v.s. TVM

- Halide: Auto-scheduling for the entire pipeline.
- TVM:
  - Break DAG into operators.
  - Auto-tuning for each operator
  - Hopefully a bag of optimal operators will yield optimal performance.

Note: “Auto-tuning” does NOT imply “Auto-scheduling”!
- In this sense, “auto-tuning” is a simpler problem than “auto-scheduling”.
- But both aspects are currently under active research.

- The schedule space of TVM is a subset of Halide.
  - TVM: each operator scheduled at root.
Conclusion

- **Machine Learning Frameworks** are NOT the end of story!
  - ✗ Performance and ✗ Hardware Backend Portability
- **Machine Learning Compilers** come for rescue!
  - ○ Tensorflow XLA, TensorComprehensions, Halide, TVM
  - ○ Not covered today: Glow, Tiramisu, Relay, Diesel, R-Stream, etc.
- **Key Idea**: *separation between what to compute (Algorithms) and how to compute (Schedules)*
- Therefore, this is an interesting research field to pursue!
Thank you!