Operator Fusion in XLA

Authors: Daniel Snider, Ruofan Liang (University of Toronto)

Presented by: Eric Zhao

Agenda

01	Motivation
02	Problem
03	Architecture
04	Key Ideas
05	Evaluation
06	Limitations
07	Future Work

01 Motivation

- ML workloads are compute-intensive, and rely on GPUs/TPUs
- Hand-optimized CUDA kernels found to outperform general frameworks
- ML compilers (like XLA) promise automatic optimization

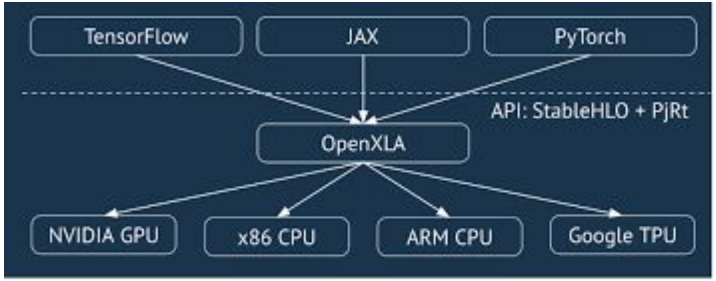
- Motivation behind this paper:
  - XLA’s compiler optimizations are mostly undocumented

02 Problem

- No transparency
  - In what cases does XLA fuse?
- Performance discrepancy
  - Hand-tuned CUDA performance > XLA performance
- Rules
  - Limits fusion
- Goals of the paper: analyze XLA’s source code and test fusion in practice

03 Architecture

- Input: Python (JAX/TF)
- Compilation pipeline: traced -> HLO IR -> optimizations -> GPU kernels
- Fusion is one of the final passes
- Kernel scheduling decides launch order



03 Architecture: XLA Optimization Pipeline

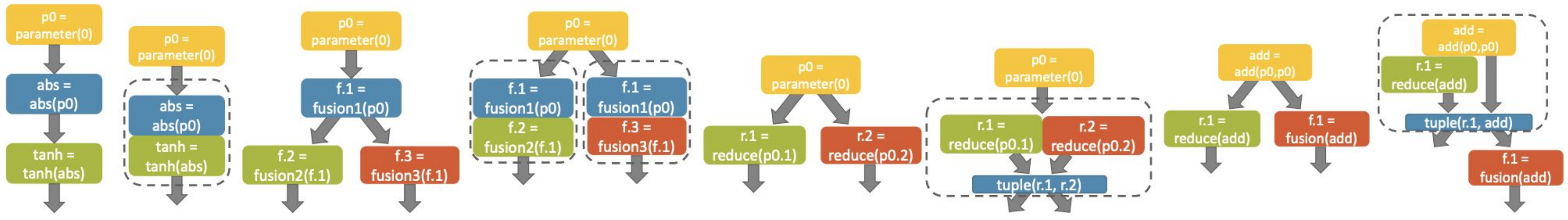
- Single-processor multi-data partitioning
- Optimization
  - Canonicalization and simplification
  - Layout assignment
- Fusion passes
  - Vertical, horizontal, and multi-output
- Post-fusion optimization (reduce, gather)
- GPU code generation

04 Key Ideas: Introduction

1. Source code analysis of XLA fusion logic
2. JAX Cartpole RL environment
3. Code modifications in XLA to test fusion
4. Key benchmarks: PyTorch/TorchScript, CPU, CUDA

04 Key Ideas: Types of Fusion

- Instruction fusion: producer-consumer
- Fusion merger: merges existing fusion
- Multi-output fusion: sibling or producer-consumer
- Horizontal fusion: combine small kernels with different shapes



(a) Instruction Fusion. (b) Fusion Merger. (c) Sibling Fusion. (d) Producer-consumer Fusion.

Fig. 1. Fusion strategies in XLA.

04 Key Ideas: Cartpole RL Case Study

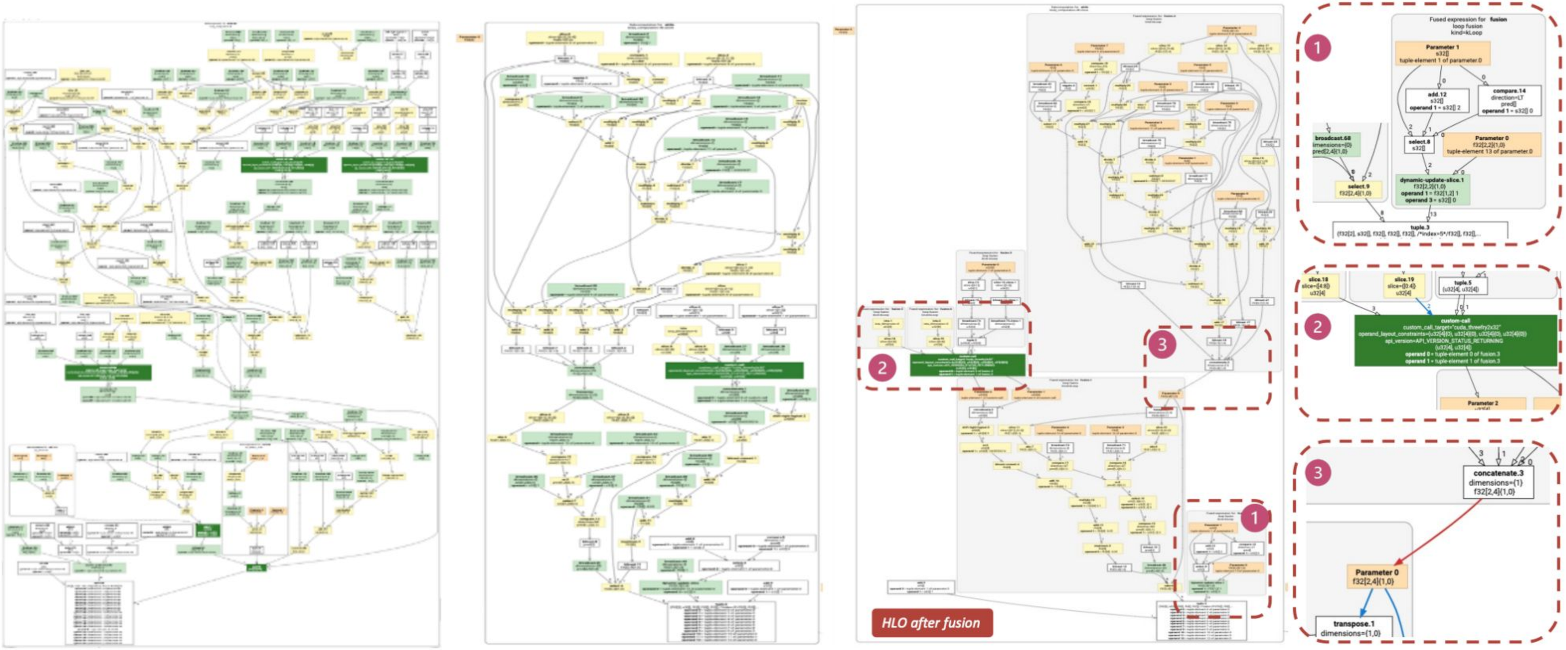
- Inspired by JAX-based RL simulation framework
- Setup: 2048 parallel environments, 10,000 steps
- Baseline: naive JAX-XLA implementation
- Evaluation: RTX 2080Ti (CUDA 11.2)

```python
def dynamics(self, state, action):
    x, x_dot, theta, theta_dot = state
    force = self.force_mag if action == 1 else -self.force_mag
    costheta = np.cos(theta)
    sintheta = np.sin(theta)
    temp = (force + self.polemass_length * theta_dot**2 * sintheta)\
         / self.total_mass
    thetaacc = (self.gravity * sintheta - costheta * temp) / (
        (4.0/3.0 - self.masspole * costheta**2 / self.total_mass)
        * self.length)
    xacc = temp - self.polemass_length * thetaacc * costheta\
         / self.total_mass
    x = x + self.tau * x_dot
    x_dot = x_dot + self.tau * xacc
    theta = theta + self.tau * theta_dot
    theta_dot = theta_dot + self.tau * thetaacc
    return np.array([x, x_dot, theta, theta_dot])

def step(self, action):
    self.state = self.dynamics(self.state, action)
    [x, x_dot, theta, theta_dot] = self.state.transpose()
    done = np.where((np.abs(x) > self.x_threshold) |
        (np.abs(theta) > self.theta_threshold_radians), 1, 0)
    self.state = self.reset_some(done)
    reward = np.ones(done.shape)
    return self.state, reward, done, {}
```

Fig. 2. The JAX code for the Cart-pole environment update step.

04 Key Ideas: Cartpole RL Case Study



(a) Before any optimization.

(b) Before fusion optimization

(c) After the fusion optimizations; right: 3 fusion boundaries for the fused kernels.

Fig. 3. HLO computation graph for Cartpole update step.

04 Key Ideas: Remove cuRAND Kernels (baseline)

- Removed the unfusable ”cuda_threefry” cuRAND kernel
- Precomputed randomness to avoid unfusable kernel
- Removed 3 parent kernels
- Resulted in 1.87x speedup

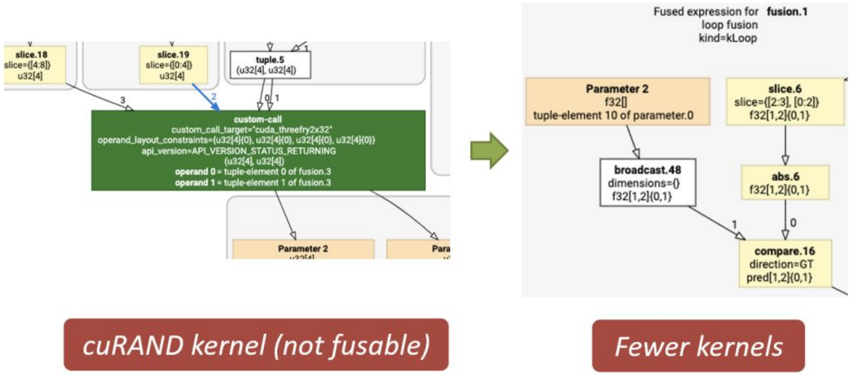


Fig. 4. We replaced the cuRAND kernel (in green) with precomputed random values to bring our cartpole implementation closer to a single fully fused kernel.

04 Key Ideas: Fusion via XLA Modification

- Relaxed constraint on fusion duplication
  - CodeDuplicationTooHigh()
- Allowed more consumers per op
- Resulted in 10% speedup

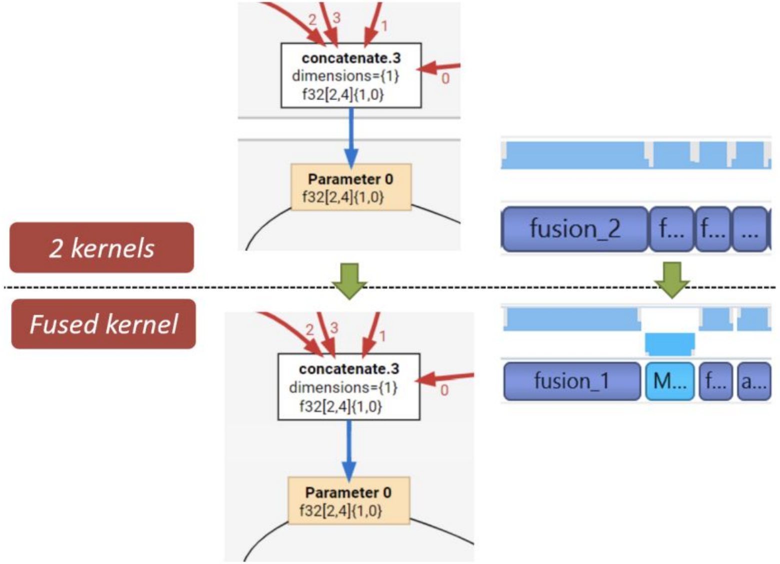


Fig. 6. We modified XLA so that it would fuse this concatenation operation into its child operation.

04 Key Ideas: Remove Concatenation

- Rewrite code to avoid unnecessary concat
  - Pass state values manually
- Enabled full fusion of simulation kernel
- 3.41x speedup

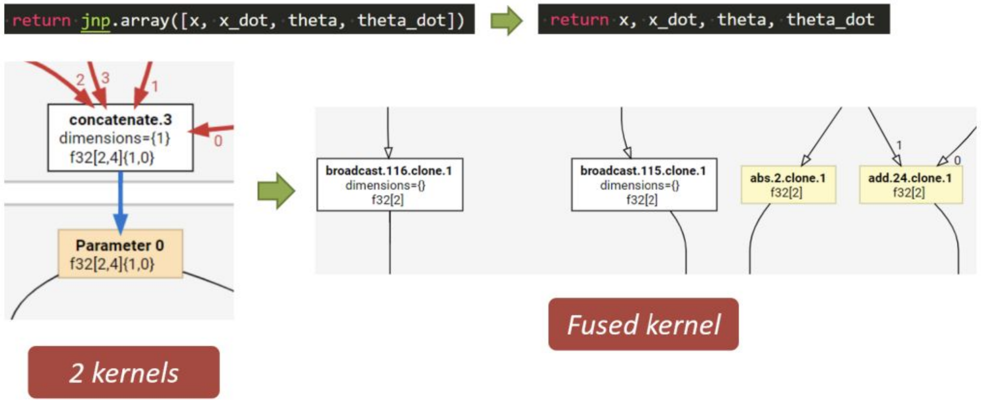


Fig. 7. We improved our code to remove the concatenation operation which allowed XLA to fuse together two cartpole kernels into one.

04 Key Ideas: Loop Unrolling

- Precomputed randomness to avoid unfusable kernel
- Removed 3 parent kernels
- Resulted in 1.87x speedup

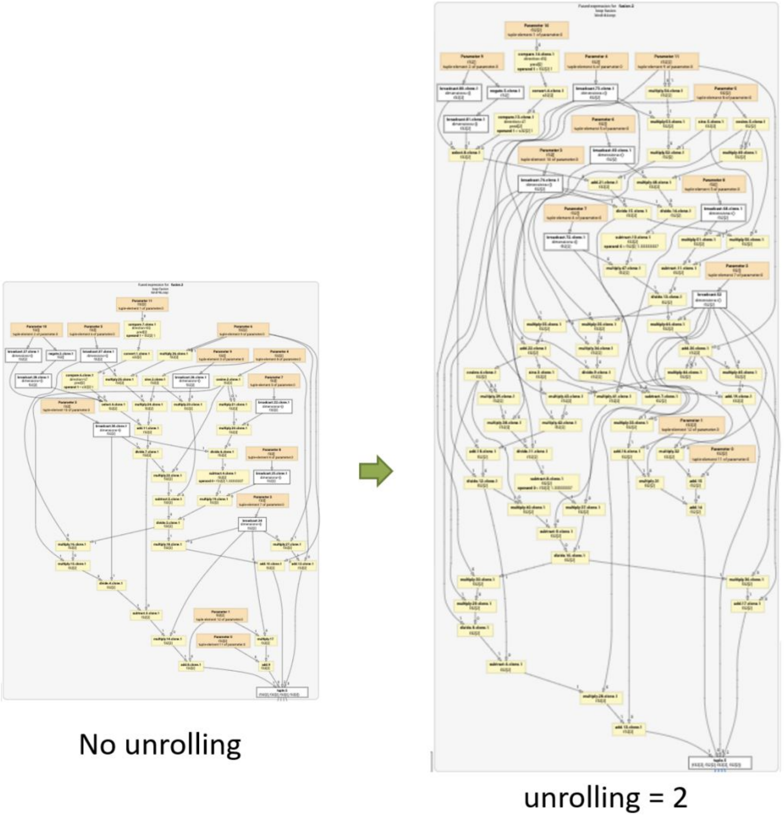


Fig. 8. Illustration of XLA's HLO IR computational graph before and after unrolling. Operations in the loop body become duplicated.

05 Evaluation

- TorchScript: 1.97x faster with single fused kernel
- CPU backend: better at small parallelism
- CUDA: ultimate performance with no framework overhead

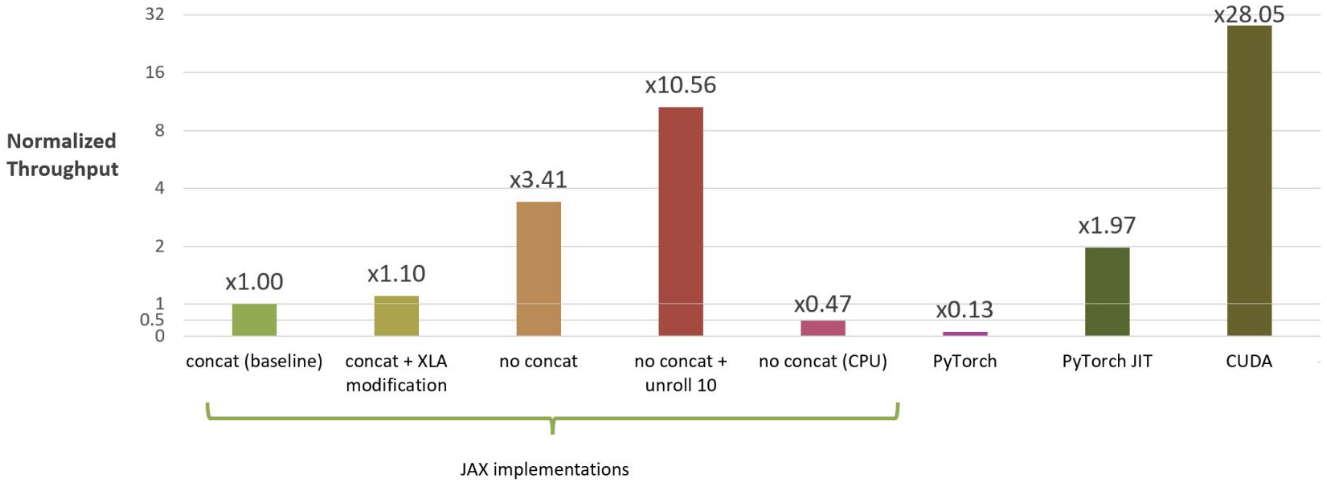


Fig. 5. Normalized throughput of different implementations of cartpole simulation.

06 Limitations

- Highly dependent on frontend Python code quality
- Custom CUDA kernels block fusion
- Conservative rules
- Compile time vs. runtime

07 Future work

1. Fusing simulation with with neural networks
2. Auto-tuning loop unrolling
3. Fusion in DL training

Thank you!

Questions?