Tensor Comprehensions

Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zach DeVito, **William S. Moses**, Sven Verdoolaege, Andrew Adams, Albert Cohen

LLVM Workshop at CGO 2018
February 24, 2018
The Tensor Comprehensions Team
A tale of many layers

- **Input**
- **Conv**
  - \( W_1 \)
- **Add**
  - \( B_1 \)
- **ReLU**
- ... (more layers)

**Caffe2**
- `caffe2.python.brew.conv()`
- ... (more functions)

**cuDNN**
- `cudnnConvolutionForward()`
- ... (more functions)

**PyTorch**
- `torch.nn.conv2d()`
- ... (more functions)

**Intel** and **MKL**
- `dnnConvolutionCreateForward_F32()`
- ... (more functions)

**TensorFlow**
- `tf.contrib.layers.conv2d()`
- ... (more functions)

* TF also can compile via XLA, discussed later
Someone has a clever idea

• Suppose a ML researcher invents a new layer: hconv
• He/she can implements it two ways:
  • Inefficiently cobbling together existing operators [slow]
  • Write optimized GPU/CPU kernel [difficult, time-consuming]
• Even when the operator exists, it often misses peak-performance, lacking cross-operator-optimization and data-shape/size tuning [1]

“Abstraction without regret”

• To make development efficient, we need abstractions that provide productivity without sacrificing performance.
• Given the enormous number of potential kernels, suggests a dynamic-code-generation approach.
Prior work

• “Direct generation” such as active library [2] or built-to-order (BTO) [3] provide usability, but miss optimization
• DSLs such as Halide [4] provide usability, and permit scheduling transformations, though manually specify.
• Compilers like XLA [5] or Latte [6] optimize and fuse operators, though performance lacking as the language can’t represent complex schedules crucial to GPU/others.

Tensor Comprehensions

- High-level DSL to express tensor computations by extending Einstein-notation.
- End-to-End compilation flow capable of lowering tensor comprehensions to efficient GPU code (CPU in progress)
- Collection of polyhedral compilation algorithms with a specific domain and target orientation
- Autotuning framework built off JIT compilation and caching
- Integration into ML Frameworks (Caffe2, Pytorch)
Tensor Comprehensions

Tensor Comprehensions

Polyhedral Transformations

Polly

Tapir/LLVM

Cilk/OpenMP

Halide IR

Polyhedral IR (ISL)

CUDA Kernel

CUDA Module

Exec

Range Inference and Specialization

ATen

libTHC.so
Tensor Comprehensions

In Progress

Polyhedral Transformations

Polyhedral IR (ISL)

Halide IR

Range Inference and Specialization

Tapir/LLVM

CUDA Kernel

Polly

CUDA Module

Cilk/OpenMP

libTHC.so

ATen

Exec

In Progress

In Progress
Optimizations at the appropriate time

- High-level polyhedral: broader scheduling optimizations (mapping, tiling, fusion, etc)
- Halide: Expression simplification / optimizations
- Tapir/LLVM [7] <-> Polly [8]: Runtime-level optimization/scheduling (coarsening, vectorization), instruction-level optimization (i.e. LICM, fuse instructions)

  https://github.com/wsmoses/Tapir-LLVM
  https://github.com/wsmoses/Tapir-Polly
TC language

Concise, emits 1000’s of optimized LOC

```python
def mv(float(M,K) A, float(K) x) -> (C) {
    C(i) +=! A(i,k) * x(k)
}

    O1(n, o, h, w) +=! I(n, c, h + kh, w + kw) * W1(o, c, kh, kw)
    O1(n, o, h, w) = fmax(O1(n, o, h, w), 0) // relu
    O2(n, d, h, w) +=! O1(n, d, h + kh, w + kw) * W2(d, o, kh, kw)
    O2(n, d, h, w) = fmax(O2(n, d, h, w), 0)
    O3(n, e, h, w) +=! O2(n, c, h + kh, w + kw) * W3(e, d, kh, kw)
    O3(n, e, h, w) = fmax(O3(n, e, h, w), 0)
}
```

Iteration bounds inferred

Variables only on one side are reduced
Polyhedral + TC

- High Level Polyhedral IR (ISL) => Easy Transformations
- Schedule heuristic folds into a single kernel
- Schedule tiled to facilitate the mapping and reuse of memory hierarchy of GPU/CPU
- GPU mapping borrows from PPCG, with extensions for more complex/imperfectly nested structures
- Memory promotion into shared cache
ISL scheduling

```python
def sgemm(float a, float b float(N,M) A, float(M,K) B) -> (C) {
    C(i,j) = b  //S(i,j)
    C(i,j) += a * A(i,k) * B(k,j)  //T(i,j,k)
}
```

**Fuse**
- Band node: (partial) execution
- Filter node: partition iteration space
- Sequence node: order-dependent collection of nodes
ISL scheduling

- ISL’s scheduling algorithm
  - Works by solving a linear program
  - Uses *affine clustering*, computing schedule for each strongly-connected components then scheduling those together
Extending ISL scheduling

• Extended ISL’s scheduler to allow additional constraints
  • Affine constraint added to the LP
  • Supply clustering decision for graph component combining
• Clustering allows for conventional minimum and maximum fusion targets AND maximum fusion that preserves at least three nested parallel loops (i.e. for mapping to CUDA blocks / threads)
Memory promotion

• Cache indirectly accessed arrays
  
  \[0[l+\text{Idx}[i][j]][k] \Rightarrow \text{shared}_0[1][i][j][k]\]

• Only done when 0 and Idx are only read (not written)

• Promote directly accesses if tile of fixed size, elements reused, and >= 1 access without memory coalescing

• Promote indirectly accessed arrays in same way (ignore coalescing)
Autotuning

• Even with heuristics, there’s a large space of options
• Derive schedule (and other parameters) by searching via genetic algorithm with fixed search-time.
How well does it work?
End-to-end benchmarks

Baseline CUDA 8.0, CUBLAS 8.0, CUDNN 6.0, CUB recent

8 Pascal nodes with 2 socket, 14 core Intel(R) Xeon(R) CPU E5-2680 v4 @ 2.40GHz, with 8 Tesla P100-SXM2 GPUs and 16GB of memory each. Median runtime out of a batch of 1000
Autotuning time out O(hours)
TC overview

“Natural ML math running faster than libraries”

• Productive environment to develop ML
• Comparable or better than hand-coded operators
• Perform *true* kernel fusion, with optimization
• Specialize to specific architecture and sizes
• Autotuning “unlocks” much of polyhedral benefits
Future work

• Share best implementations, for any architecture
• Port to more architectures & accelerators, leveraging highly optimized primitives
• Implement symbolic automatic differentiation directly
• Allow sparse, vector and mixed-precision types
• Support more dynamic control flow and ML architectures
• Integrate with other frameworks
TC overview

“Natural ML math running faster than libraries”

• Available stand-alone and in Caffe2/PyTorch bindings [public in a few days]
• Open source:
  https://github.com/facebookresearch/tensorcomprehensions
• Paper:
  https://arxiv.org/abs/1802.04730
Questions?
Backup Slides
import torch
A = torch.randn(3, 4)
B = torch.randn(4, 5)
C = torch.mm(A, B)

Figure 12: JIT compile, tune, or hit the compilation cache, then run

Figure 11: Build execution engine

def mm(float(M,K) A, float(K,N) B) -> (C) {
    C(m,n) += A(m,kk) * B(kk,n)
}

import tc
ee = tc.ExecutionEngine()

# Defining the model

def prodModel(float(E1,D) LUT1, int(B,L1) I1, float(E2,D) LUT2, int(B,L2) I2) -> (01,02) {
    01(i,j) += LUT1[I1(i,k),j]
    02(i,j) += LUT2[I2(i,k),j]
}

def MLPL(float(B,M) I, float(O,N) W1, float(O) B1) -> (01) {
    01(b,n) = B1(n)
    01(b,n) += I(b,m) * W1(n,m)
    01(b,n) = fmaxf(01(b,n), 0)
}

def MLP3(float(B,M) I, float(O,N) W2, float(O) B2, float(P,O) W3, float(P) B3, float(Q,P) W4, float(Q) B4) -> (01,02,03,04) {
    02(b,o) = B2(o)
    02(b,o) += 01(b,n) * W2(o,n)
    02(b,o) = fmaxf(02(b,o), 0)
    03(b,p) = B3(p)
    03(b,p) += 02(b,o) * W3(p,o)
    03(b,p) = fmaxf(03(b,p), 0)
    04(b,q) = B4(q)
    04(b,q) += 03(b,p) * W4(q,p)
    04(b,q) = fmaxf(04(b,q), 0)
}

def 2LUT(float(E1,D) LUT1, int(B,L1) I1, float(E2,D) LUT2, int(B,L2) I2) -> (01,02) {
    01(i,j) += LUT1[I1(i,k),j]
    02(i,j) += LUT2[I2(i,k),j]
}

Figure 17: Full production model (pseudo-code)