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

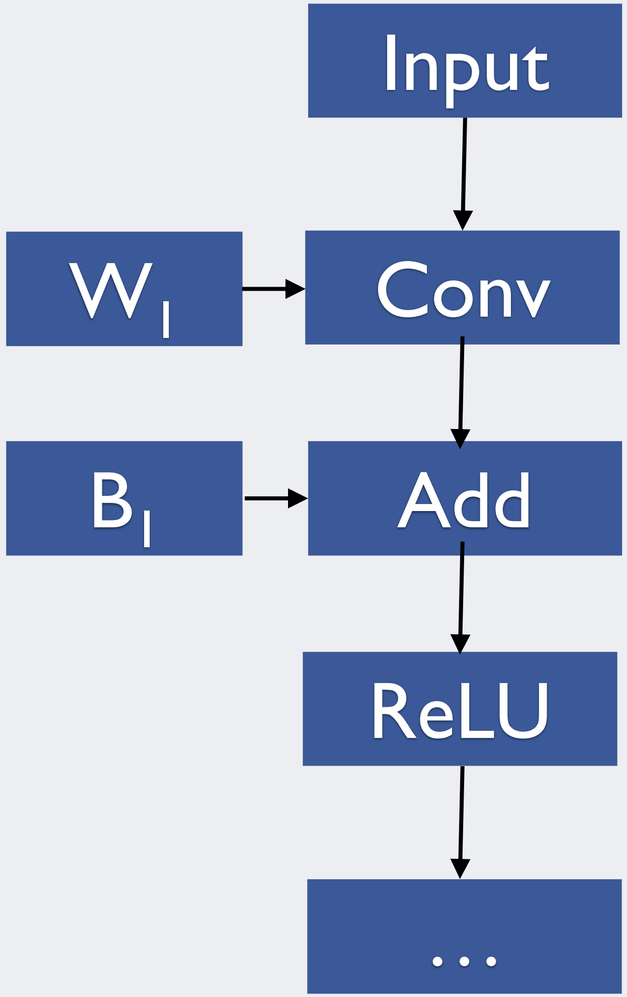








A tale of many layers



Caffe2

```
caffe2.python.brew.conv()
...
```

PYTORCH

```
torch.nn.conv2d()
...
```

TensorFlow *

```
tf.contrib.layers.conv2d()
...
```

NVIDIA cuDNN

```
cudnnConvolutionForward()
...
```

intel MKL

```
dnnConvolutionCreateForwa
rd_F32()
...
```

* 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]

[1] Fast Convolutional Nets With fbfft : A GPU Performance Evaluation, ICLR, 2015

“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

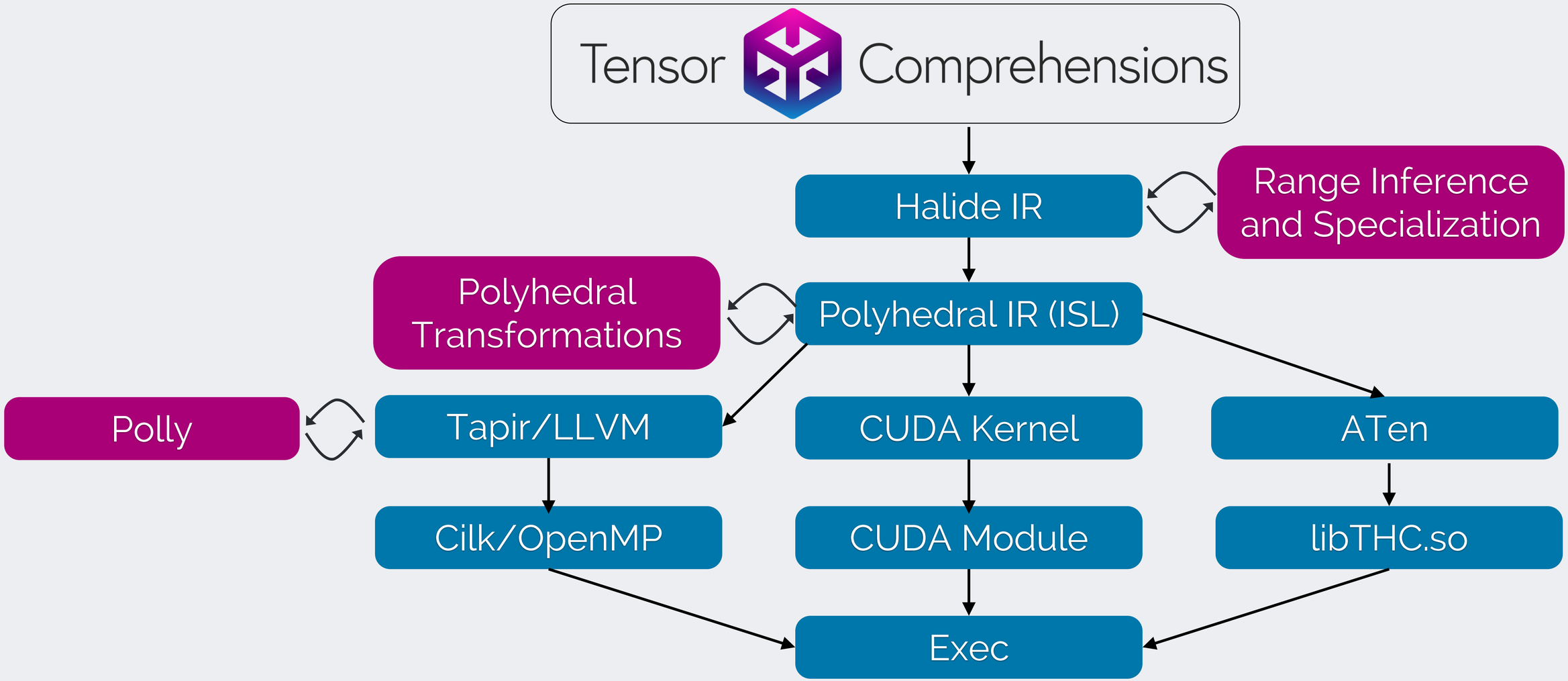
- “Direct generation” such as active library [2] or built-to-order (BTO) [3] provide usability, but miss optimization
- DSLs such as Halide [4] provide usability, and permit scheduling transformations, though manually specify.
- Compilers like XLA [5] or Latte [6] optimize and fuse operators, though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

[2] Run-time code generation in C++ as a foundation for domain-specific optimization, 2003
[3] Automating the generation of composed linear algebra kernels, 2009
[4] Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines, 2013
[5] Xla:domain-specific compiler for linear algebra to optimizes tensorflow computations, 2017
[6] Latte: A language, compiler, and runtime for elegant and efficient deep neural networks, 2016

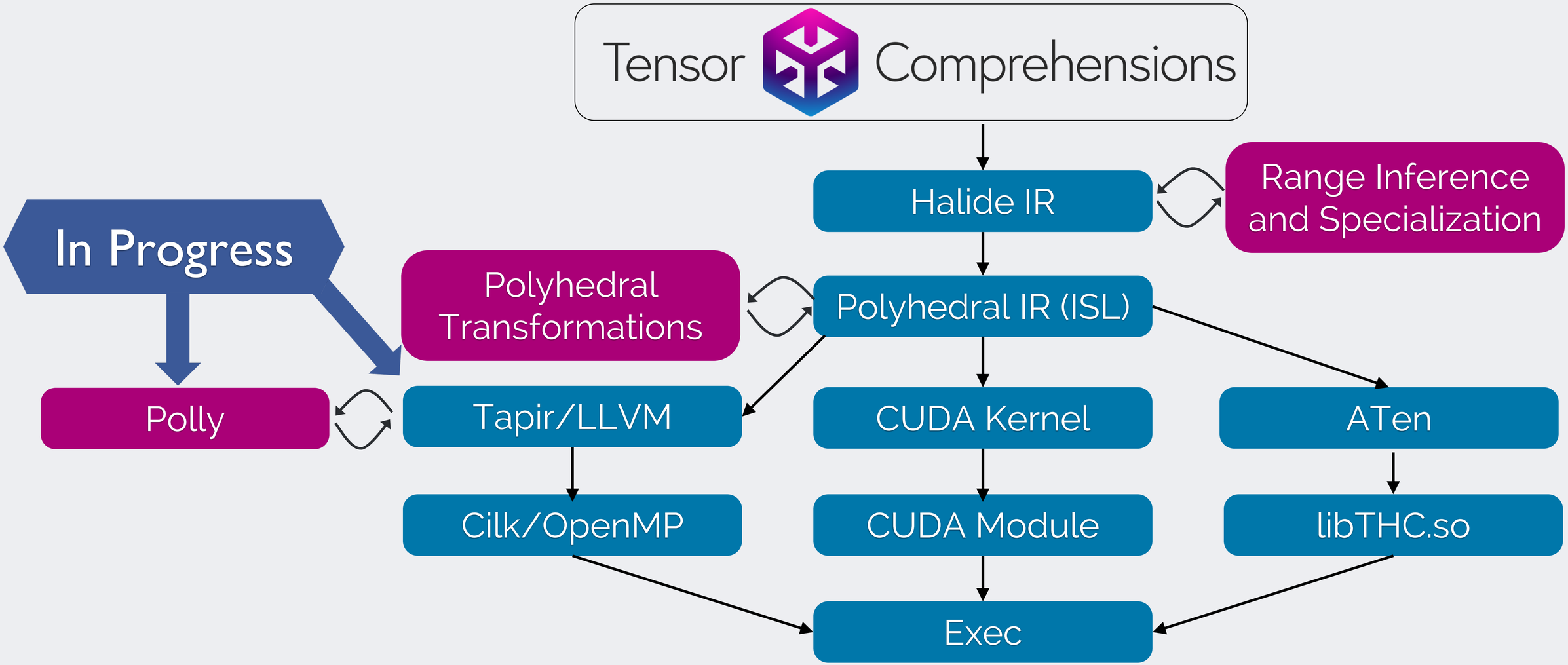
Tensor Comprehensions

- 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



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)

[7] Tapir: Embedding Fork-Join Parallelism into LLVM's Intermediate Representation, 2017
https://github.com/wsmoses/Tapir-LLVM

[8] Polly - Performing polyhedral optimizations on a low-level intermediate representation, 2012
https://github.com/wsmoses/Tapir-Polly

TC language

Concise, emits 1000's of optimized LOC

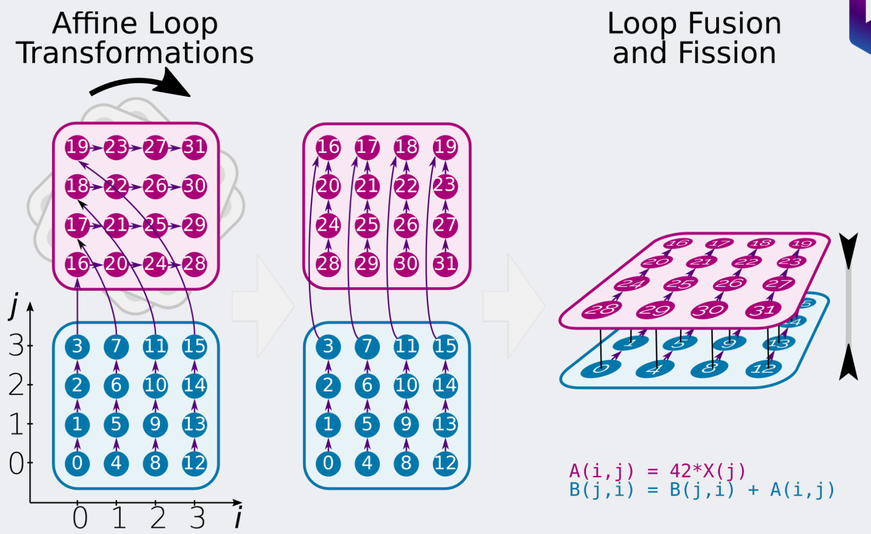
```
def mv(float(M,K) A, float(K) x) -> (C) {
  C(i) +=! A(i,k) * x(k)
}
```

```
def conv3(float(N,C,H,W) I, float(O,C,H,W) W1, float(D,O,H,W) W2, float(E,D,H,W) W3) -> (O1, O2, O3) {
  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

```
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)
}
```

$$\text{Domain} \left[\begin{array}{ll} \{\mathtt{S}(i,j) & \mid 0 \le i < N \wedge 0 \le j < K\} \\ \{\mathtt{T}(i,j,k) & \mid 0 \le i < N \\ & \quad \wedge 0 \le j < K \wedge 0 \le k < M\} \end{array} \right]$$
$$\text{Sequence}$$
$$\text{Filter}\{\mathtt{S}(i,j)\}$$
$$\text{Band}\{\mathtt{S}(i,j) \to (i,j)\}$$
$$\text{Filter}\{\mathtt{T}(i,j,k)\}$$
$$\text{Band}\{\mathtt{T}(i,j,k) \to (i,j,k)\}$$

Fuse

$$\text{Domain} \left[\begin{array}{ll} \{\mathtt{S}(i,j) & \mid 0 \le i < N \wedge 0 \le j < K\} \\ \{\mathtt{T}(i,j,k) & \mid 0 \le i < N \wedge 0 \le j < K \wedge 0 \le k < M\} \end{array} \right]$$
$$\text{Band} \left[\begin{array}{ll} \{\mathtt{S}(i,j) & \to (i,j)\} \\ \{\mathtt{T}(i,j,k) & \to (i,j)\} \end{array} \right]$$
$$\text{Sequence}$$
$$\text{Filter}\{\mathtt{S}(i,j)\}$$
$$\text{Filter}\{\mathtt{T}(i,j,k)\}$$
$$\text{Band}\{\mathtt{T}(i,j,k) \to (k)\}$$

Sequence node: order-dependent collection of nodes

Band node: (partial) execution

Filter node: partition iteration space

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

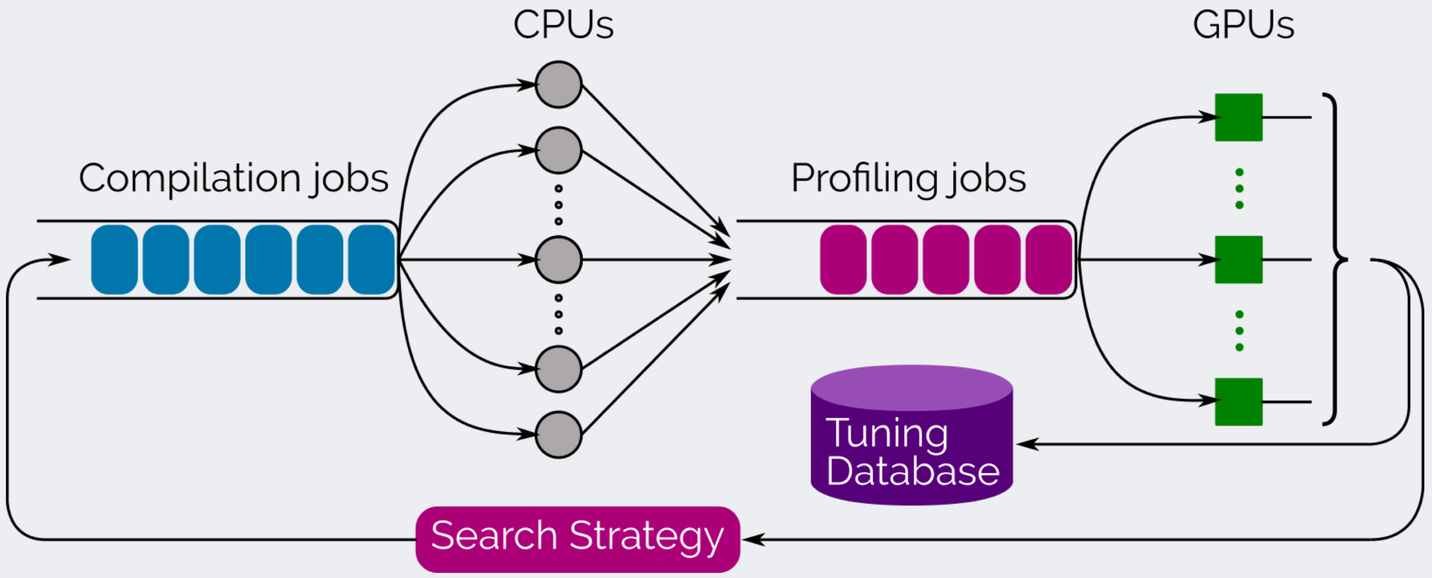
```
O[l+Idx[i][j]][k] => shared_O[l][i][j][k]
```

- Only done when `O` 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

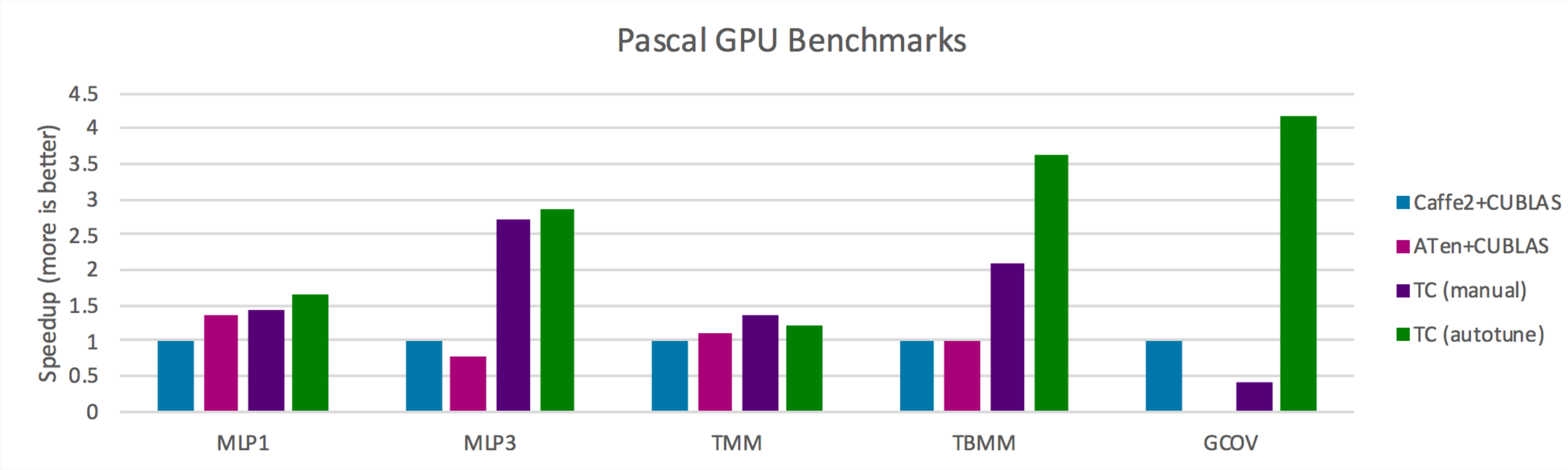
- Even with heuristics, there's a large space of options
- Derive schedule (and other parameters) by searching via genetic algorithm with fixed search-time.



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

TC in Practice

```
import tc
ee = tc.ExecutionEngine()
ee.define("""
  def mm(float(M,K) A,
         float(K,N) B) -> (C) {
    C(m,n) +=! A(m,kk) * B(kk,n)
  }
""")
```

Figure 11: Build execution engine

```
import torch
A = torch.randn(3,4)
B = torch.randn(4,5)
C = ee.mm(A, B)
```

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

```
def 2LUT(float(E1,D) LUT1, int(B,L1) I1,
         float(E2,D) LUT2, int(B,L2) I2) -> (O1,O2) {
  O1(i,j) +=! LUT1(I1(i,k),j)
  O2(i,j) +=! LUT2(I2(i,k),j)
}
def MLP1(float(B,M) I, float(O,N) W1, float(O) B1) -> (O1) {
  O1(b,n)  = B1(n)
  O1(b,n) += I(b,m) * W1(n,m)
  O1(b,n)  = fmaxf(O1(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) -> (O1,O2,O3,O4) {
  O2(b,o)  = B2(o)
  O2(b,o) += O1(b,n) * W2(o,n)
  O2(b,o)  = fmaxf(O2(b,o), 0)
  O3(b,p)  = B3(p)
  O3(b,p) += O2(b,o) * W3(p,o)
  O3(b,p)  = fmaxf(O3(b,p), 0)
  O4(b,q)  = B4(q)
  O4(b,q) += O3(b,p) * W4(q,p)
  O4(b,q)  = fmaxf(O4(b,q), 0)
}

def prodModel(float(E1,D) LUT1, int(B,L1) I1,
              float(E2,D) LUT2, int(B,L2) I2,
              float(B,WX) I3, float(WY,WX) W,
              float(N,M) W1, float(N) B1,
              float(O,N) W2, float(O) B2,
              float(P,O) W3, float(P) B3,
              float(Q,P) W4, float(Q) B4)
    -> (C1,C2,C3,I,O1,O2,O3,O4) {
  (C1,C2)     = 2LUT(LUT1,I1,LUT2,I2)
  C3(b,wy)   += I3(b,wxx) * W(wy,wxx)
  I(b,m)      = concat(C1, C2, C3) # not implemented yet
  O1          = MLP1(I, W1, B1)
  (O2,O3,O4) = MLP3(O1,W2,B2,W3,B3,W4,B4)
  # O4 goes out to binary classifier, omitted here
}
```

Figure 17: Full production model (pseudo-code)