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**!

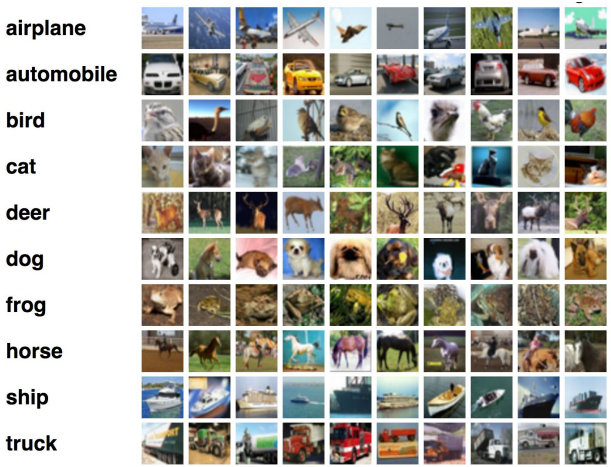
  ✖ **Performance**
  ✖ **Hardware Backend Portability**

- **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



Machine Translation

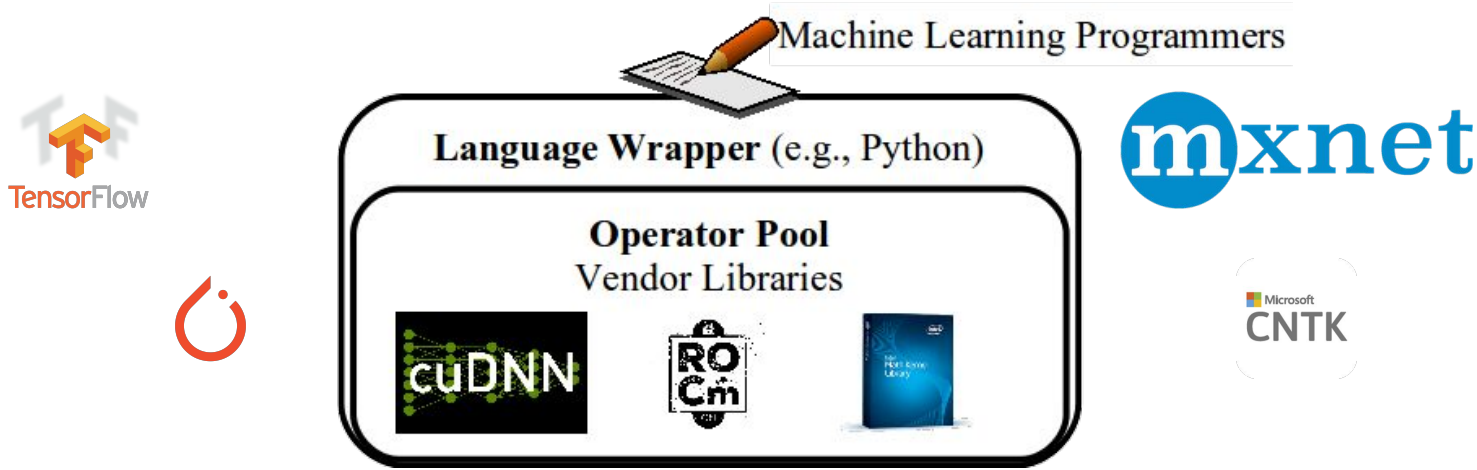
Google Translate

Speech Recognition

hello

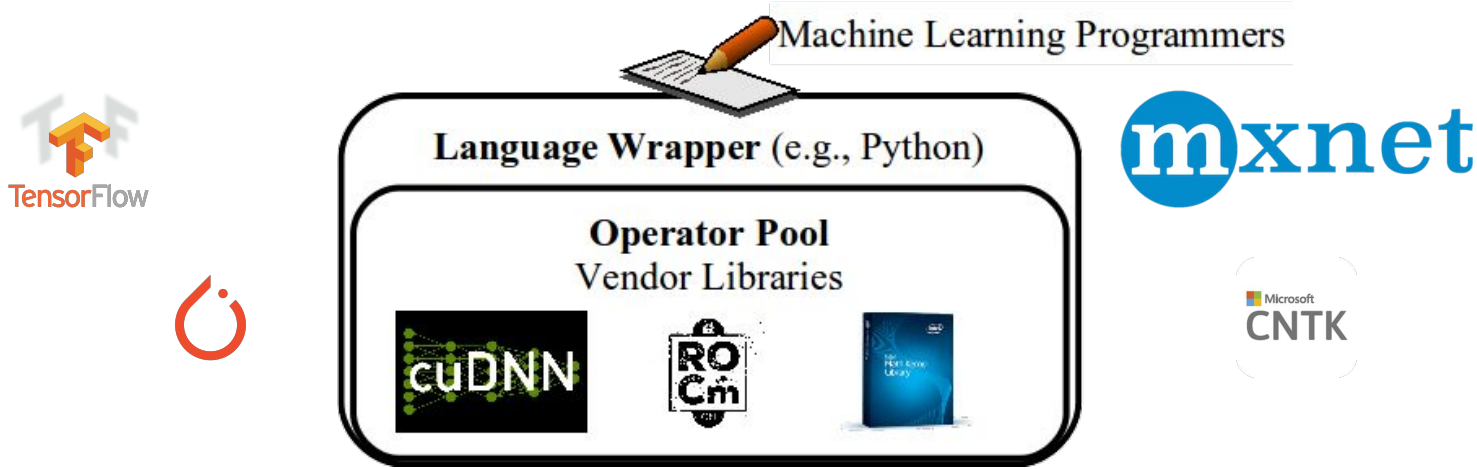
Background: **Machine Learning Frameworks**

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**

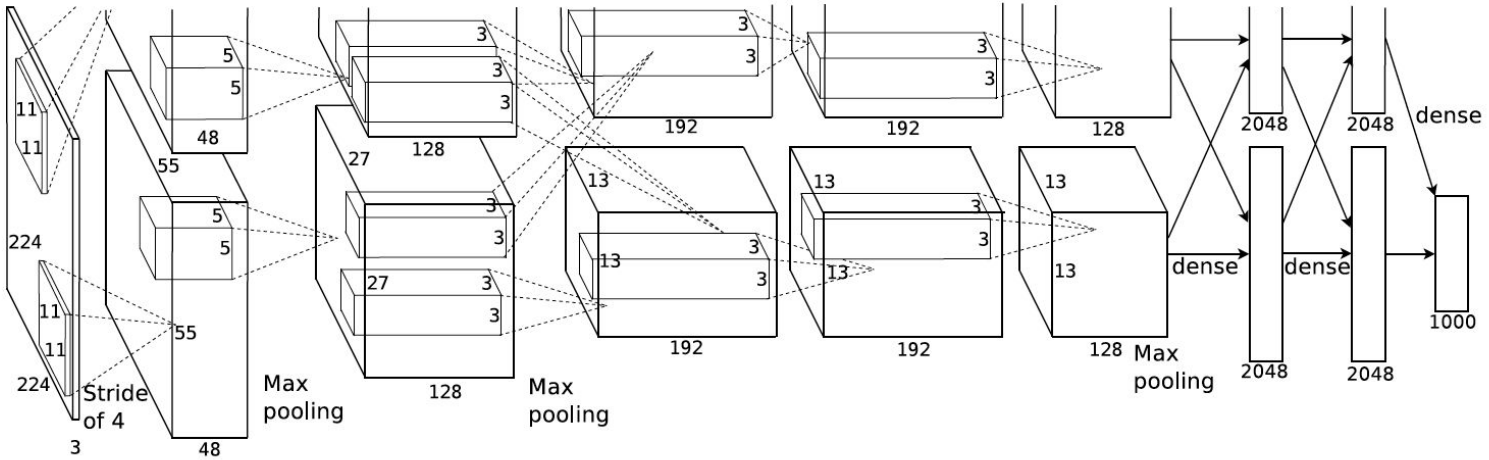
The **Operator Pool** consists of implementations of machine learning operators that are **tuned in depth**, either by framework developers or vendor libraries.



Background: **Machine Learning Frameworks**

Machine learning practitioners build up **Computation Graphs** for development.

Each node is an **instantiation** of the operator with specific parameters.



Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E. Hinton. "Imagenet classification with deep convolutional neural networks." Advances in neural information processing systems. 2012.

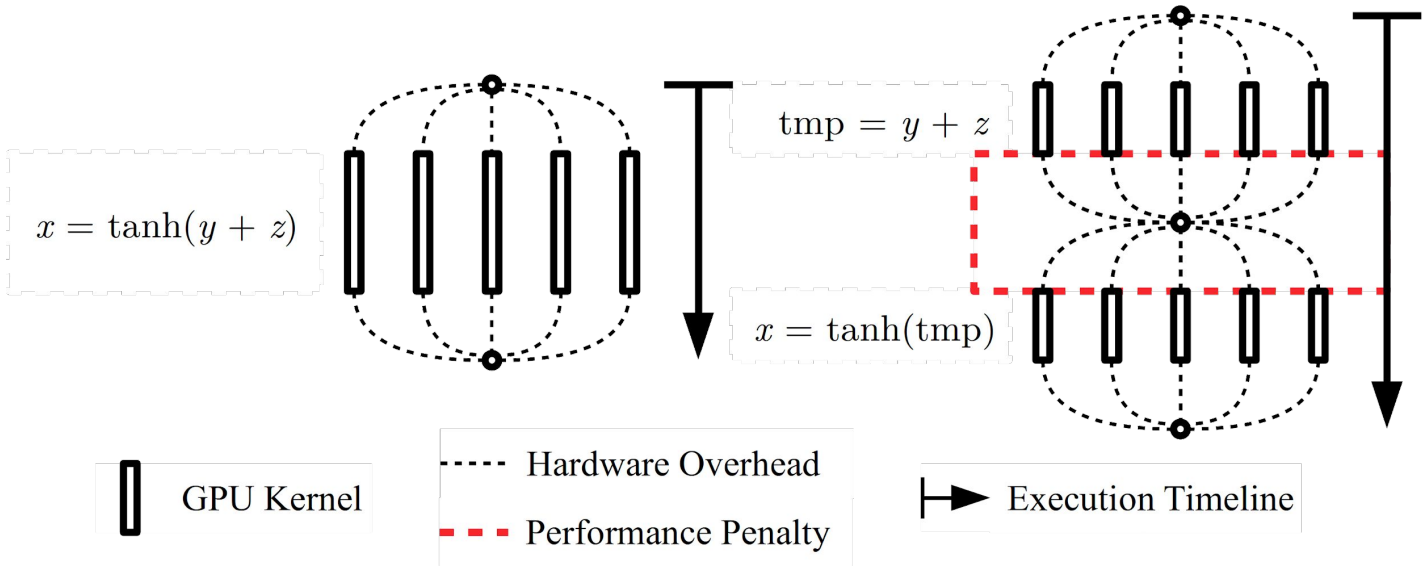
Problem Statement

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

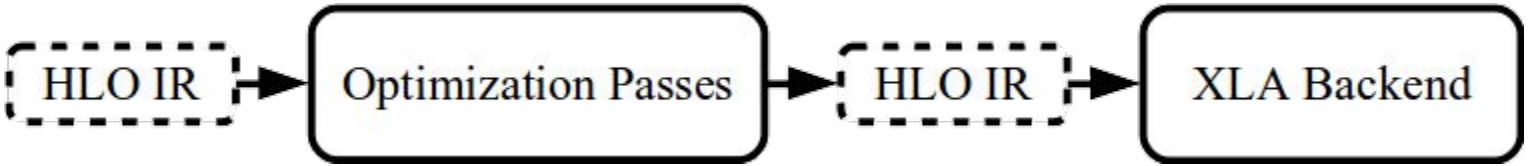
So what are the problems with **Machine Learning Frameworks**?

- ✖ **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, rathen 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**

Vasilache, Nicolas, et al. "Tensor Comprehensions: Framework-Agnostic High-Performance Machine Learning Abstractions." arXiv preprint arXiv:1802.04730 (2018).

TensorComprehensions: **Code Generation**

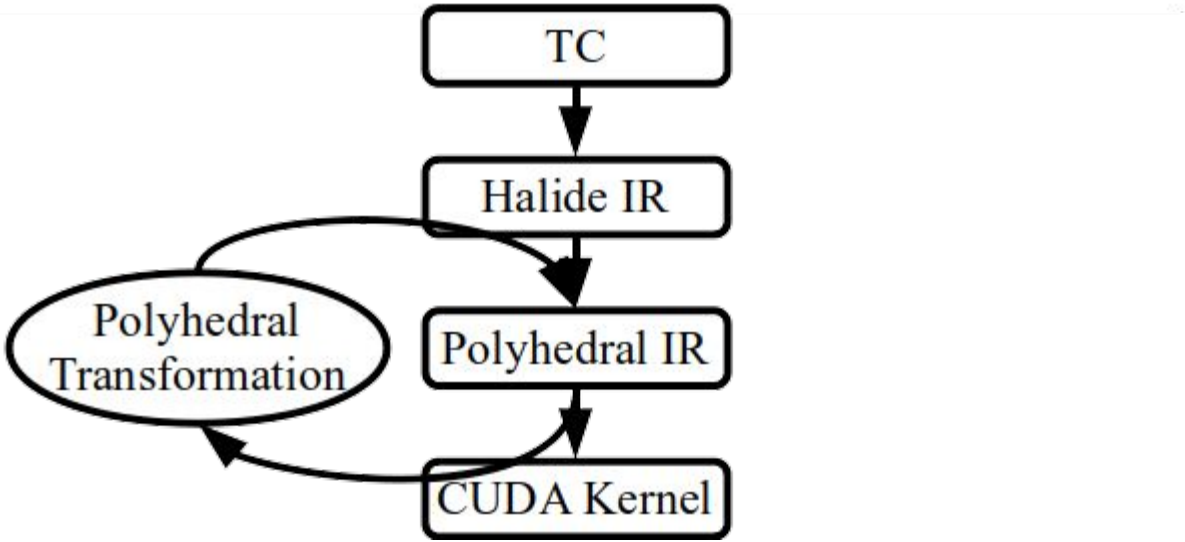
Example: Writing a **Matrix Multiply** using *TensorComprehensions*:

```
# import tc, torch
ee = tc.ExecutionEngine()
# define a new matmul operator
ee.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
```

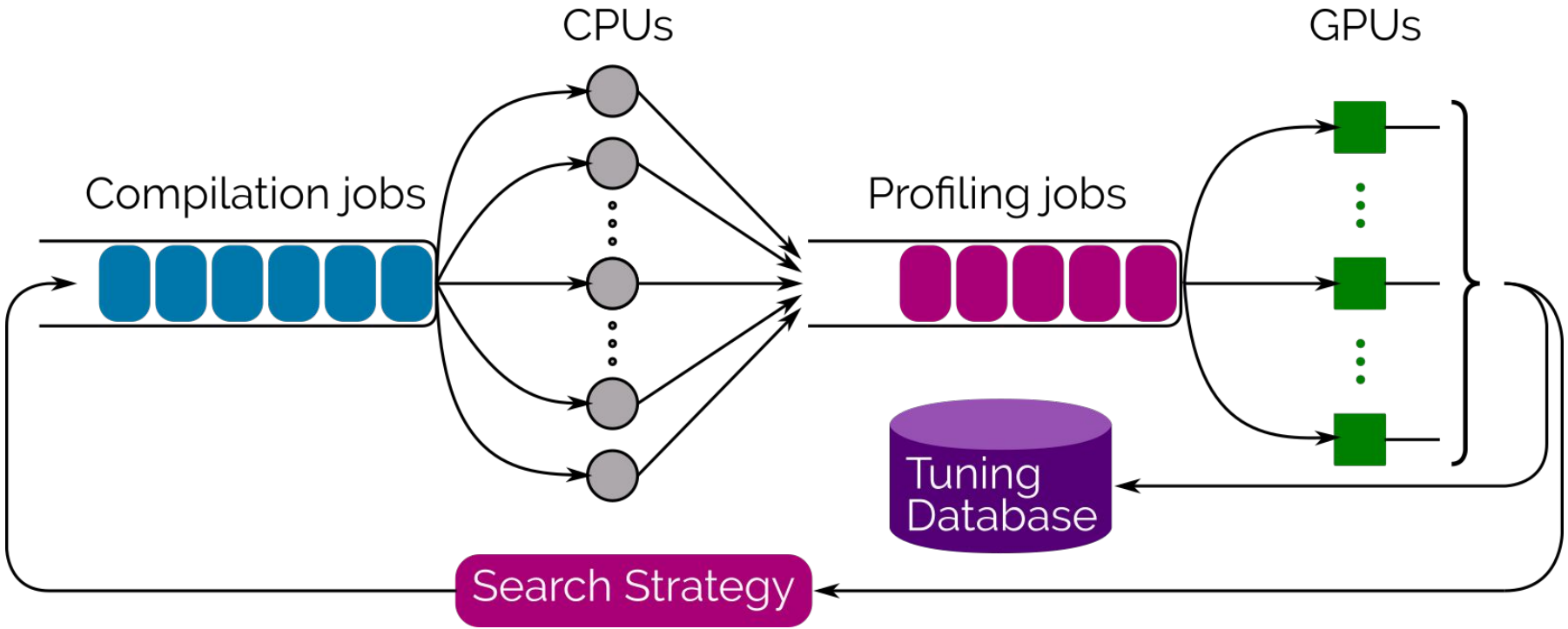
TensorComprehensions: **Code Generation**

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**



TensorComprehensions: **Auto-tuning**

```
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

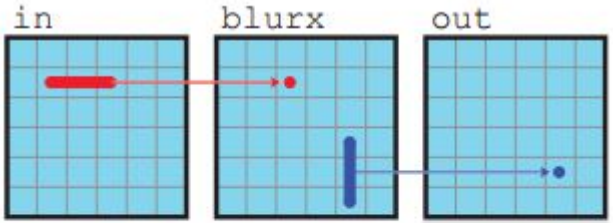
Consider a image processing pipeline consisted of a two-stage blur filter:

```
blurx(x, y) =    in(x-1, y) +    in(x, y) +    in(x+1, y)

  out(x, y) = blurx(x, y-1) + blurx(x, y) + blurx(x, y+1)
```

What are the possible implementations?

Case Study: Two-stage Blur Filter

Parallelism, **Locality**, **No Redundant Computation**

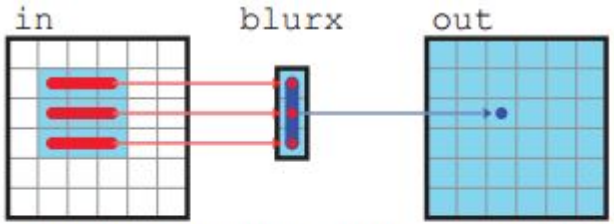


breadth first: each function is entirely evaluated before the next one.

```
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**

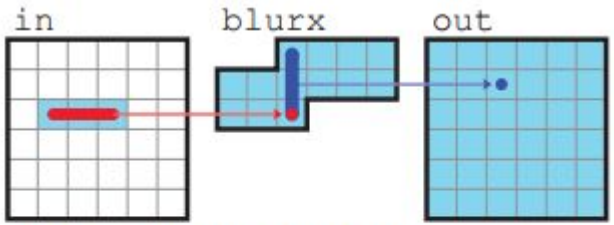


total fusion: values are computed on the fly each time that they are needed.

```
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]
    out[y][x] = blurx[0] + blurx[1] + blurx[2]
```

Case Study: Two-stage Blur Filter

Parallelism, **Locality**, **No Redundant Computation**

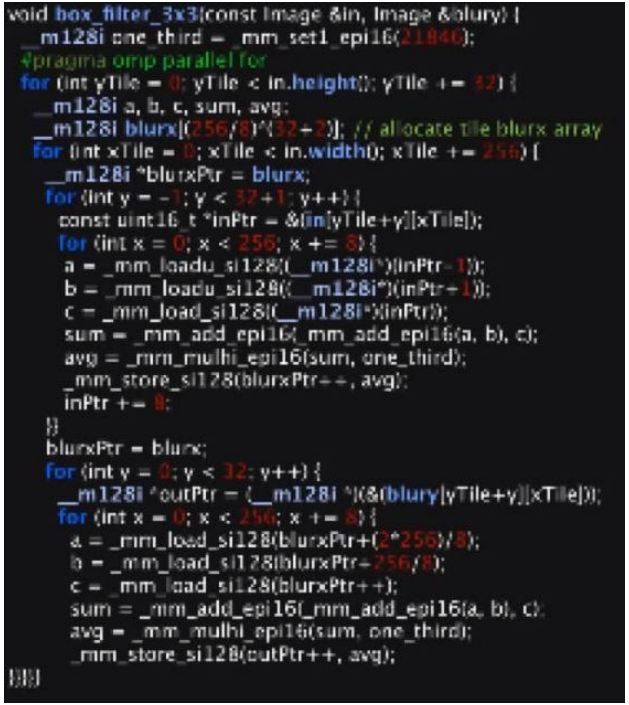


sliding window: values are computed when needed then stored until not useful anymore.

```
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.



```
void box_filter_3x3(const Image &in, Image &blury) {
  __m128i one_third = _mm_set1_epi16(21846);
  #pragma omp parallel for
  for (int yTile = 0; yTile < in.height(); yTile += 32) {
    __m128i a, b, c, sum, avg;
    __m128i blurx[(256/8)*(32+2)]; // allocate tile blurx array
    for (int xTile = 0; xTile < in.width(); xTile += 256) {
      __m128i *blurxPtr = blurx;
      for (int y = -1; y < 32+1; y++) {
        const uint16_t *inPtr = &(in(xTile, yTile+y));
        for (int x = 0; x < 256; x += 8) {
          a = _mm_loadu_si128((__m128i*)(inPtr-1));
          b = _mm_loadu_si128((__m128i*)(inPtr+1));
          c = _mm_load_si128((__m128i*)(inPtr));
          sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
          avg = _mm_mulhi_epi16(sum, one_third);
          _mm_store_si128(blurxPtr++, avg);
          inPtr += 8;
        }
      }
      blurxPtr = blurx;
      for (int y = 0; y < 32; y++) {
        __m128i *outPtr = (__m128i *)(&(blury(xTile, yTile+y)));
        for (int x = 0; x < 256; x += 8) {
          a = _mm_load_si128(blurxPtr+(2*256)/8);
          b = _mm_load_si128(blurxPtr+256/8);
          c = _mm_load_si128(blurxPtr++);
          sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
          avg = _mm_mulhi_epi16(sum, one_third);
          _mm_store_si128(outPtr++, avg);
}}}
```

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**!

J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe, “Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines,” in Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation, ser. PLDI ’13. New York, NY, USA: ACM, 2013, pp. 519–530.

A Halide Program

Input: Algorithm

```
blurx(x,y) = in(x-1,y)
           + in(x,y)
           + in(x+1,y)

out(x,y) = blurx(x,y-1)
         + blurx(x,y)
         + blurx(x,y+1)
```

Input: Schedule

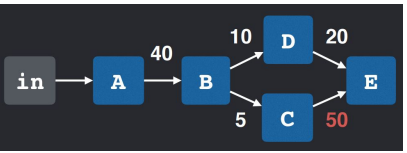
```
blurx: split x by 4 → x_o, x_i
       vectorize: x_i
       store at out.x_0
       compute at out.y_i

out:   split x by 4 → x_o, x_i
       split y by 4 → y_o, y_i
       reorder: y_o, x_o, y_i, x_i
       parallelize: y_o
       vectorize: x_i
```

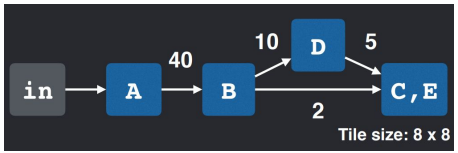
But how to schedule?

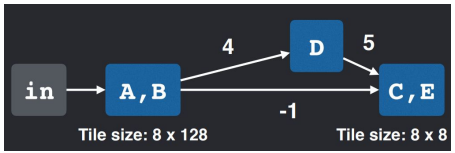
- Heuristic Cost Model


```
Cost = (Number of arithmetic operations)
     + (Number of memory accesses) x (LOAD COST)
Benefit(A, B) = Cost(A) + Cost(B) - Cost(A, B)
```

- Greedy Grouping until no positive benefit



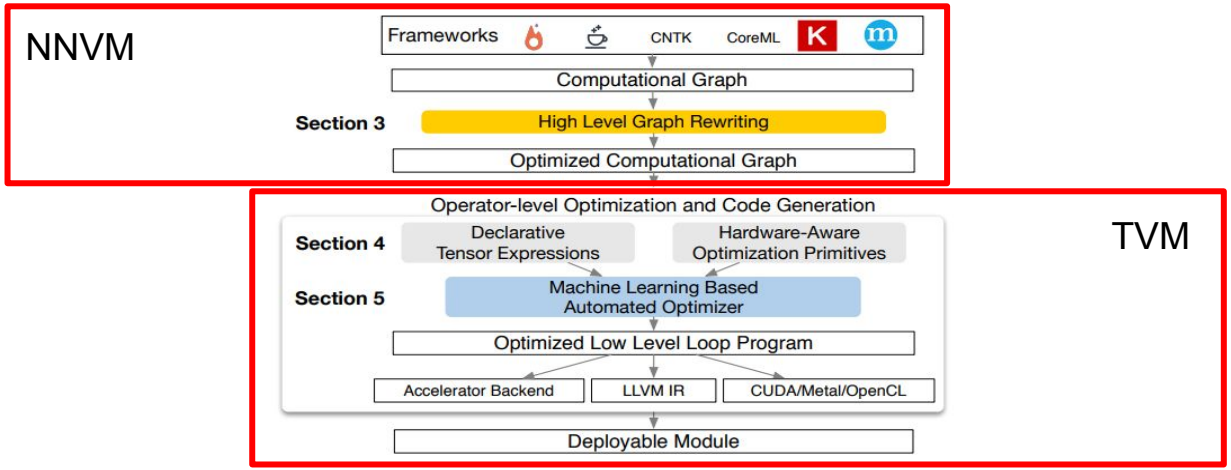




Ravi Teja Mullapudi, Andrew Adams, Dillon Sharlet, Jonathan Ragan-Kelley, and Kayvon Fatahalian. 2016. Automatically Scheduling Halide Image Processing Pipelines. ACM TOG (2016).

TVM

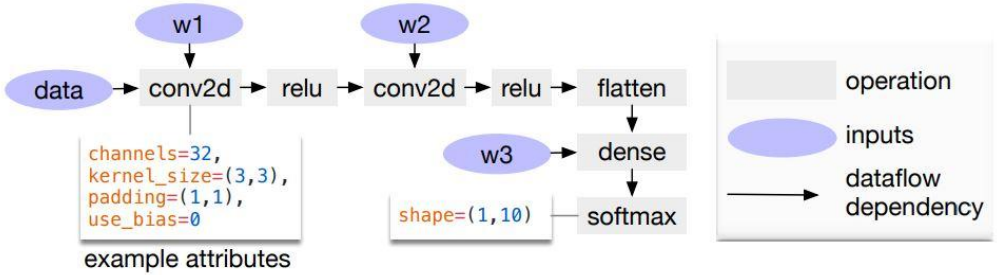
- Like Halide (not exactly), but in machine learning (ML) domain.
- ML operators (conv2d, relu, etc.) implemented as **Algorithm + Schedule**.
  - Targeted at various hardware backends.
  - Optimal schedules are different for different hardware backends.



T. Chen et al. TVM: End-to-End Compilation Stack for Deep Learning. In SysML, 2018.

TVM

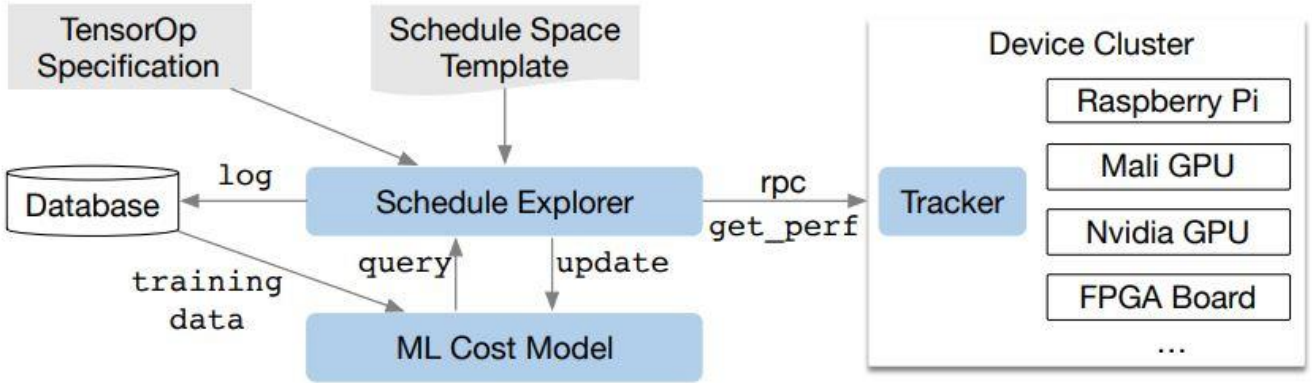
- Graph Level Optimization:
  - NNVM
  - Specific Handwritten Rules



- 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!