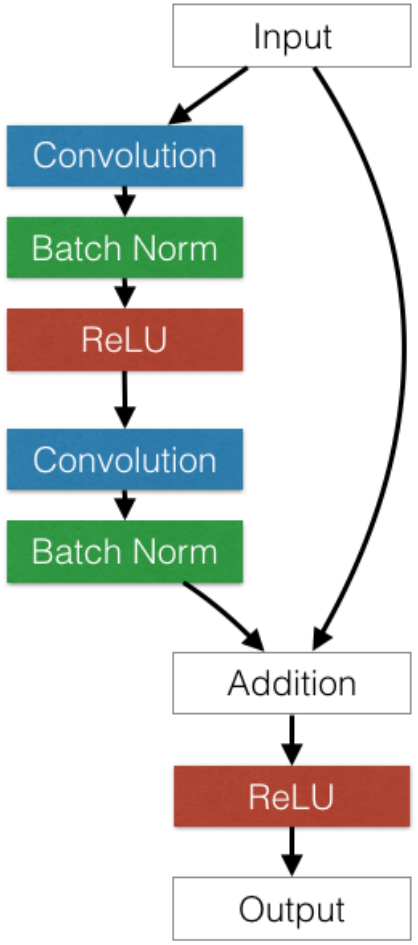
# AI-Systems
# Deep Learning Compilers

Some content has been borrowed from:
- Simon Mo's Lecture (Ai-Sys SP19)
- UW-CSE 599W Systems for ML Class

Joseph E. Gonzalez
Co-director of the RISE Lab
jegonzal@cs.berkeley.edu

# Deep Learning Execution Model

- DL frameworks execute the network by running one operator at a time
  - May **optimize choice** of operator implementation
  - Each operator reads input and produces new output
- Issues?

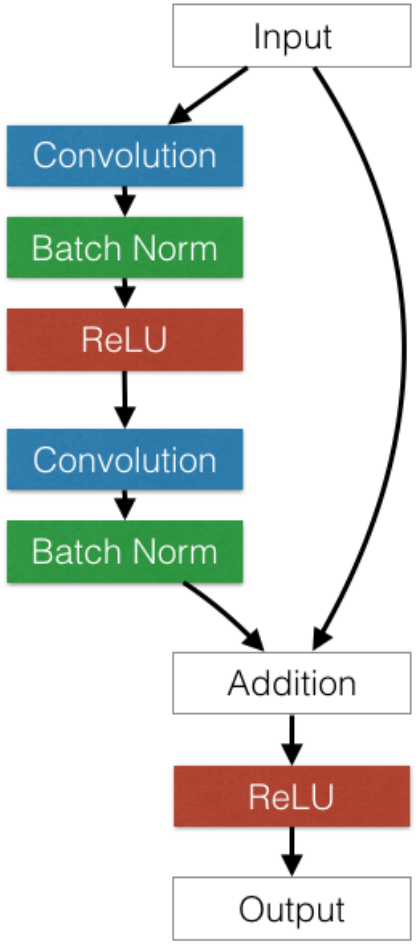



**Example (Conv Op):**
volta_scudnn_winograd_128x128_ldg1_ldg4_relu_tile148t_nt_v1

Image from http://torch.ch/blog/2016/02/04/resnets.html

# Issues with operator at a time execution model

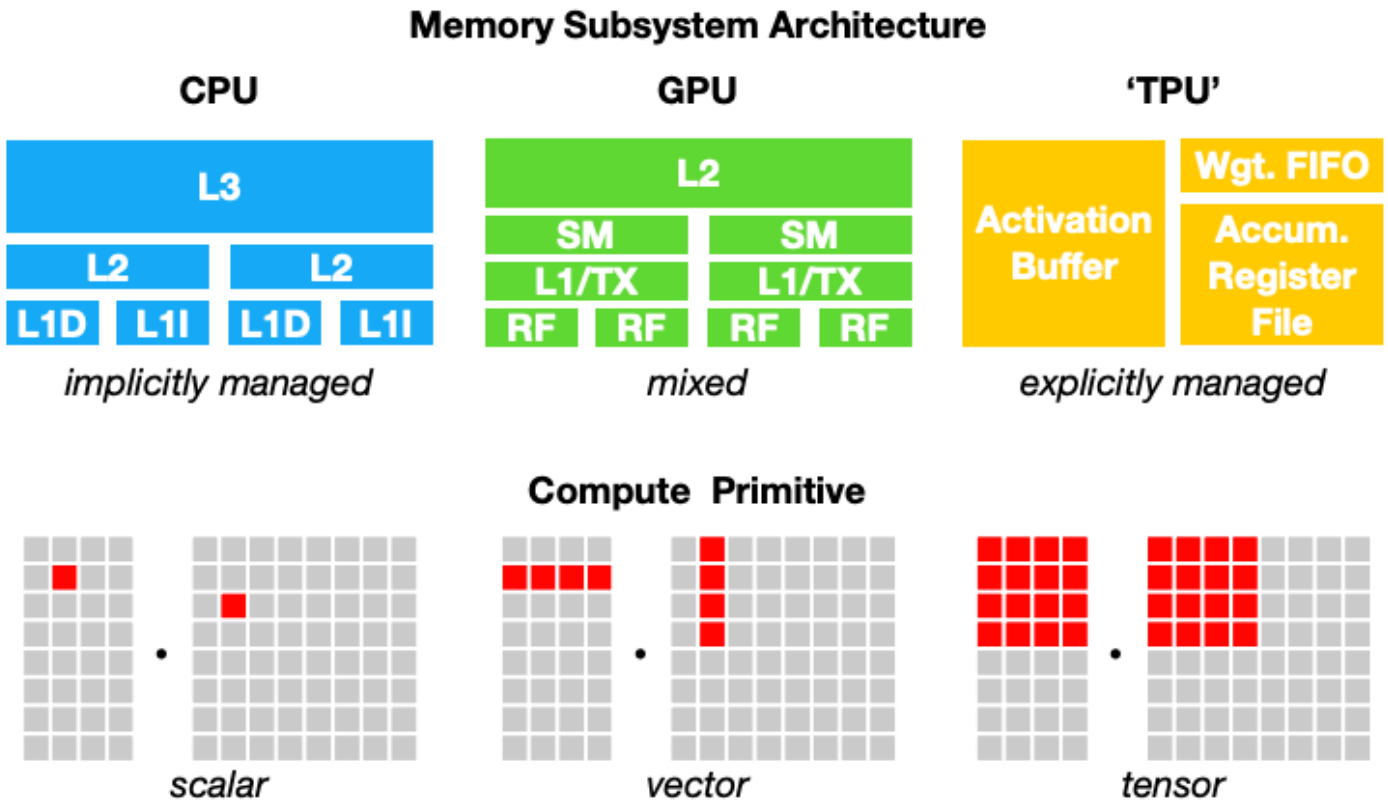
- Interpreted execution
- Multiple scans of data
  - Potentially large temp. memory requirements
- Need optimized implementations of operators
  - Difficult to build new ops.
  - Difficult to target new hardware




**Example (Conv Op):**
volta_scudnn_winograd_128x128_ldg1_ldg4_relu_tile148t_nt_v1

Image from http://torch.ch/blog/2016/02/04/resnets.html

# Hardware for Deep Learning




- **Heterogenous hardware**:
  - Need to optimize workload for different hardware.
- **Layered Memory Hierarchy**:
  - Complex scheduling space
- **Parallel Compute Primitives**
  - Threads
  - SIMD/Vector parallelism
  - Specialized primitives (e.g., Tensor Cores)

# Challenges of Implementing Ops.

Basic gemm operation from TVM paper

```
A = t.placeholder((1024, 1024))
B = t.placeholder((1024, 1024))
k = t.reduce_axis((0, 1024))
C = t.compute((1024, 1024), lambda y, x:
                t.sum(A[k, y] * B[k, x], axis=k))
s = t.create_schedule(C.op)
```

```
for y in range(1024):
  for x in range(1024):
    C[y][x] = 0
    for k in range(1024):
      C[y][x] += A[k][y] * B[k][x]
```

**+ Loop Tiling**

```
yo, xo, ko, yi, xi, ki = s[C].tile(y, x, k, 8, 8, 8)
```

```
for yo in range(128):
  for xo in range(128):
    C[yo*8:yo*8+8][xo*8:xo*8+8] = 0
    for ko in range(128):
      for yi in range(8):
        for xi in range(8):
          for ki in range(8):
            C[yo*8+yi][xo*8+xi] +=
               A[ko*8+ki][yo*8+yi] * B[ko*8+ki][xo*8+xi]
```

**+ Cache Data on Accelerator Special Buffer**

```
CL = s.cache_write(C, vdla.acc_buffer)
AL = s.cache_read(A, vdla.inp_buffer)
# additional schedule steps omitted …
```

**+ Map to Accelerator Tensor Instructions**

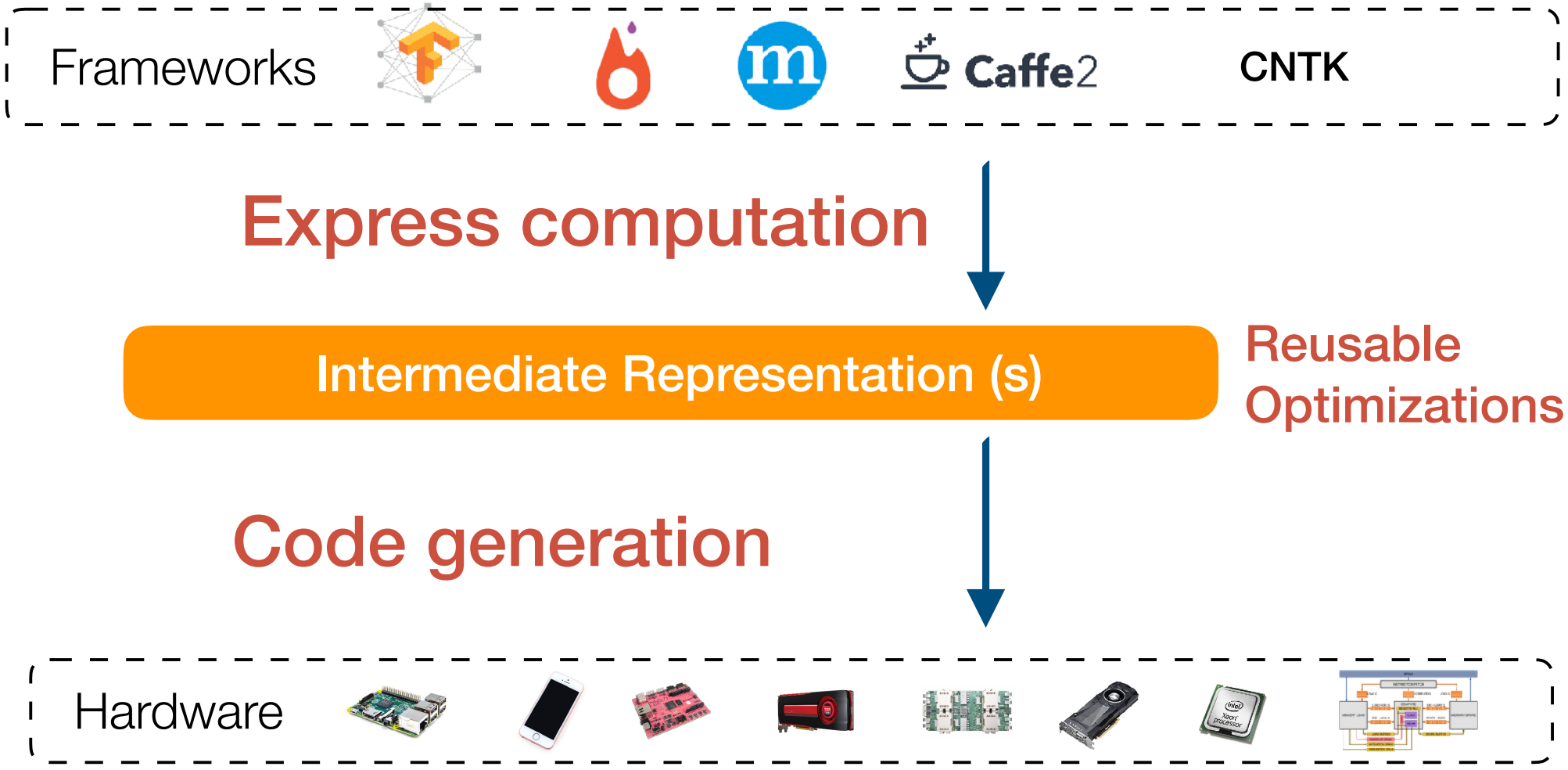
```
s[CL].tensorize(yi, vdla.gemm8x8)
```

```
inp_buffer AL[8][8], BL[8][8]
acc_buffer CL[8][8]
for yo in range(128):
  for xo in range(128):
    vdla.fill_zero(CL)
    for ko in range(128):
      vdla.dma_copy2d(AL, A[ko*8:ko*8+8][yo*8:yo*8+8])
      vdla.dma_copy2d(BL, B[ko*8:ko*8+8][xo*8:xo*8+8])
      vdla.fused_gemm8x8_add(CL, AL, BL)
    vdla.dma_copy2d(C[yo*8:yo*8+8,xo*8:xo*8+8], CL)
```

schedule → schedule transformation - - → corresponding low-level code

- Need to reason about:
  - Loop order and Tiling
  - Memory layout
  - Relation to other operations
- Relationship to memory hierarchy and specialized hardware

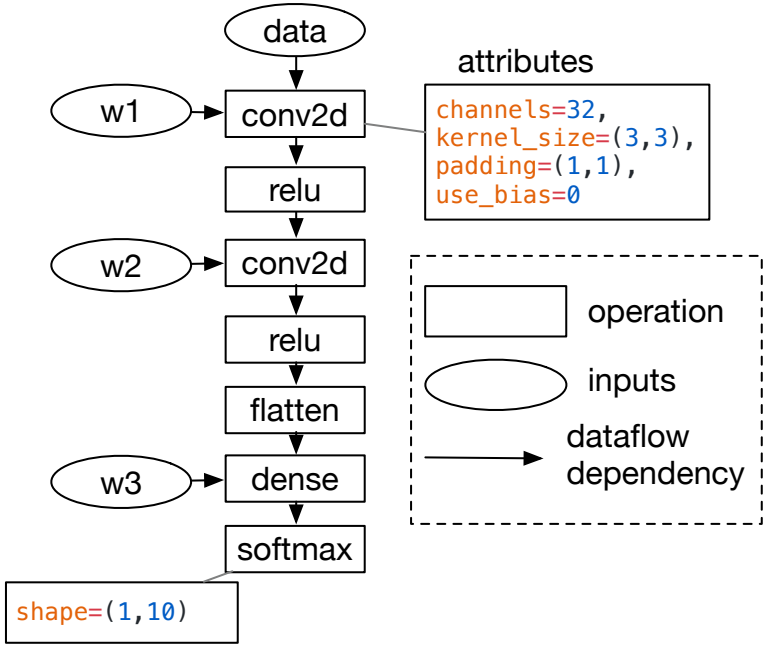
# Compiler's Perspective



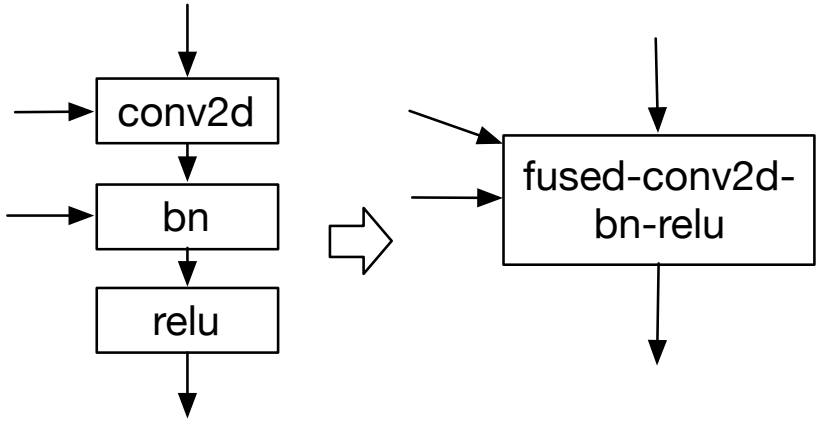

Slide borrowed from UW-CSE 599W Systems for ML Class

# Computation Graph as IR

**Represent High level Deep Learning Computations**

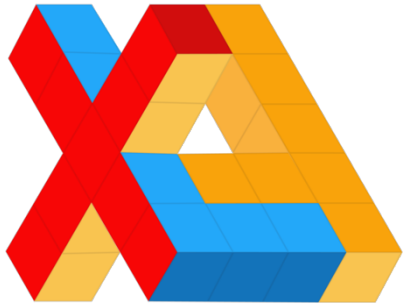



**Effective Equivalent Transformations to Optimize the Graph**



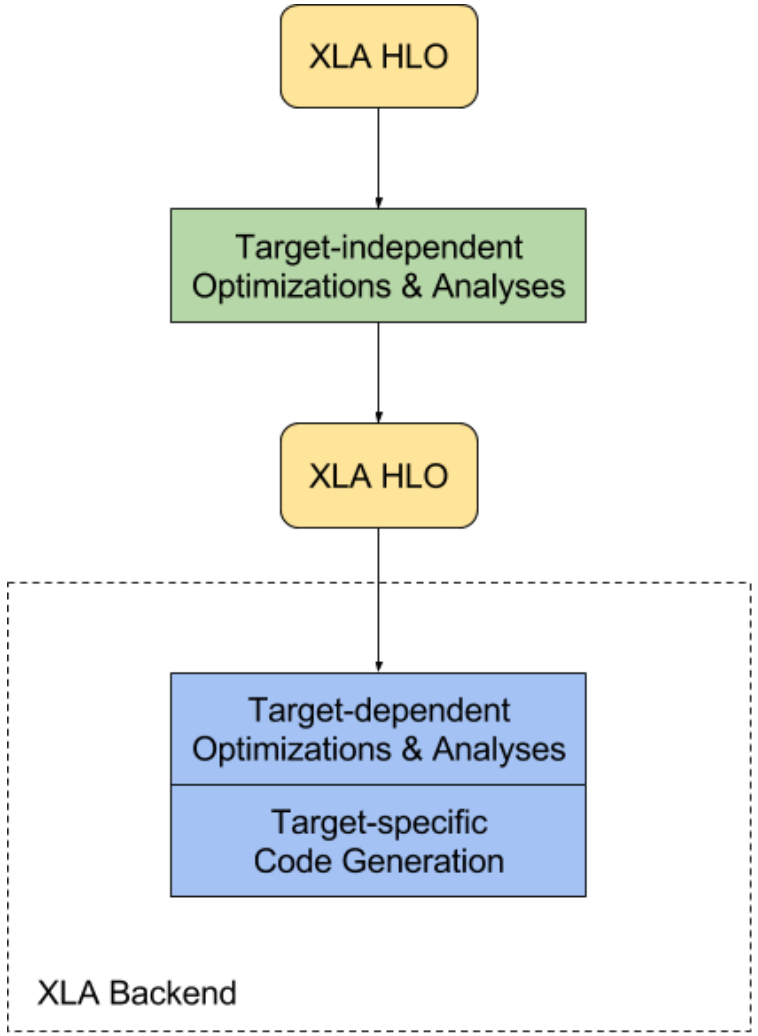

**Approach taken by: TensorFlow XLA, Intel NGraph, Nvidia TensorRT**

Slide borrowed from UW-CSE 599W Systems for ML Class



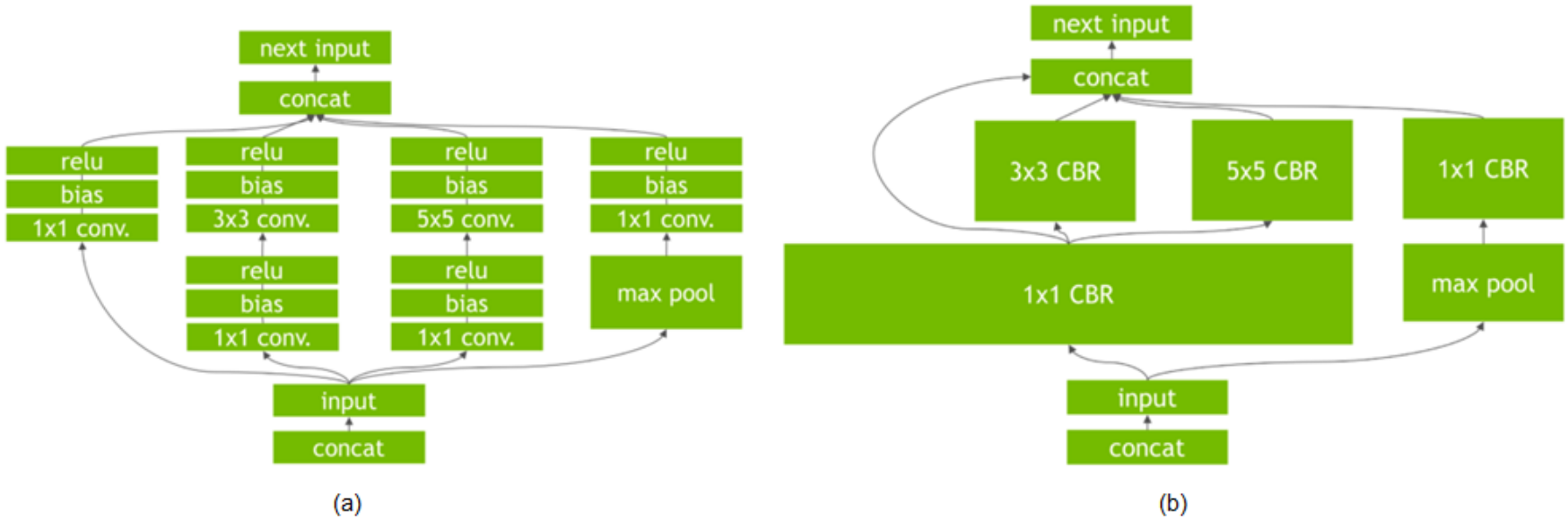
# TensorFlow XLA Compiler

- XLA HLO is an IR composed to tensor operations
- Generates optimized binaries to evaluate models
  - Fuses kernels → eliminating reads and writes to slow memory
  - Optimized data layout
  - Reduced environment size
- User still needs to implement optimized tensor ops for each architecture
  - Smaller set then all of TF



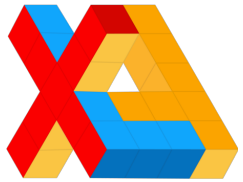
# Nvidia TensorRT

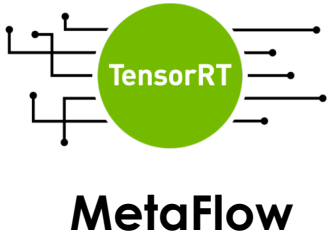
- Nvidia's platform for optimizing deep neural networks
  - Quantization of weights
  - Data layout and kernel section
  - Fuses kernels -- Vertically (conv, relu) and horizontally (reuse inputs)



# Intermediate Representation (IR) Approaches

## Computation Graph







DAG Optimization:
- Operator Fusion
- No-op Elimination

Typically leverage pre-existing tensor operations

## Tensor Loop Algebra

**Halide**

Tensor Comprehensions

- Optimize loop order, tiling, and memory layout across operators in DAG
- Support new operator design

# Halide: Compiling Image Processing Pipelines

Key Innovation:

- Decouples ***algorithm from the compute***
- ***So we can express operator in a simple language***

# Halide: Compiling Image Processing Pipelines

Key Innovation:
- Decouples ***algorithm from the compute***
- User only needs to provide the algorithm, and optionally the schedule.

**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)
```

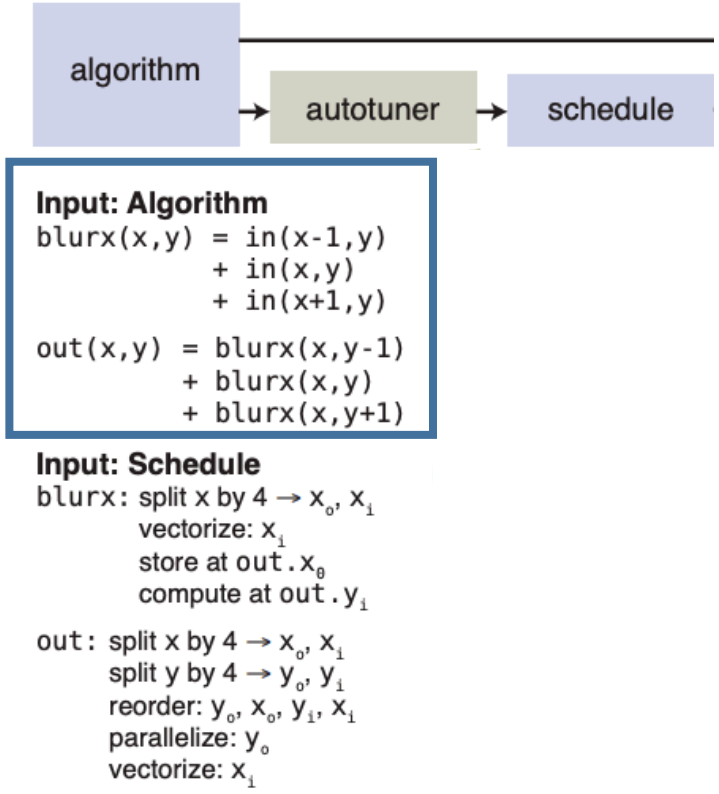
**Input: Schedule**

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

# Halide: Compiling Image Processing Pipelines

Key Innovation:
- Decouples ***algorithm from the compute***
- User only needs to provide the algorithm

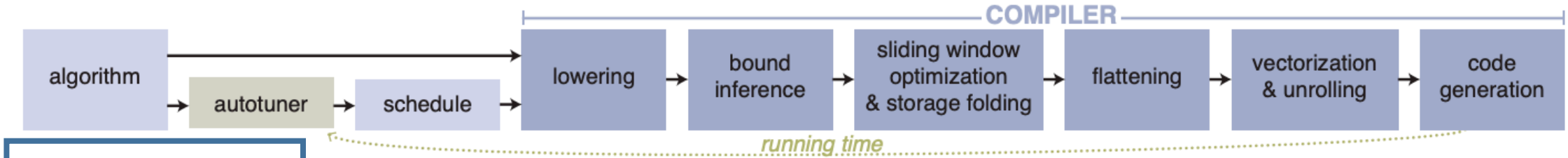
Auto-tuner can select the optimal "schedule"
- How to split the axis?
- How to vectorize?



# Halide: Compiling Image Processing Pipelines

Key Innovation:
- Decouples ***algorithm from the compute***
- User only needs to provide the algorithm

Auto-tuner can select the optimal "schedule"
- How to split the axis?
- How to vectorize?




**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**

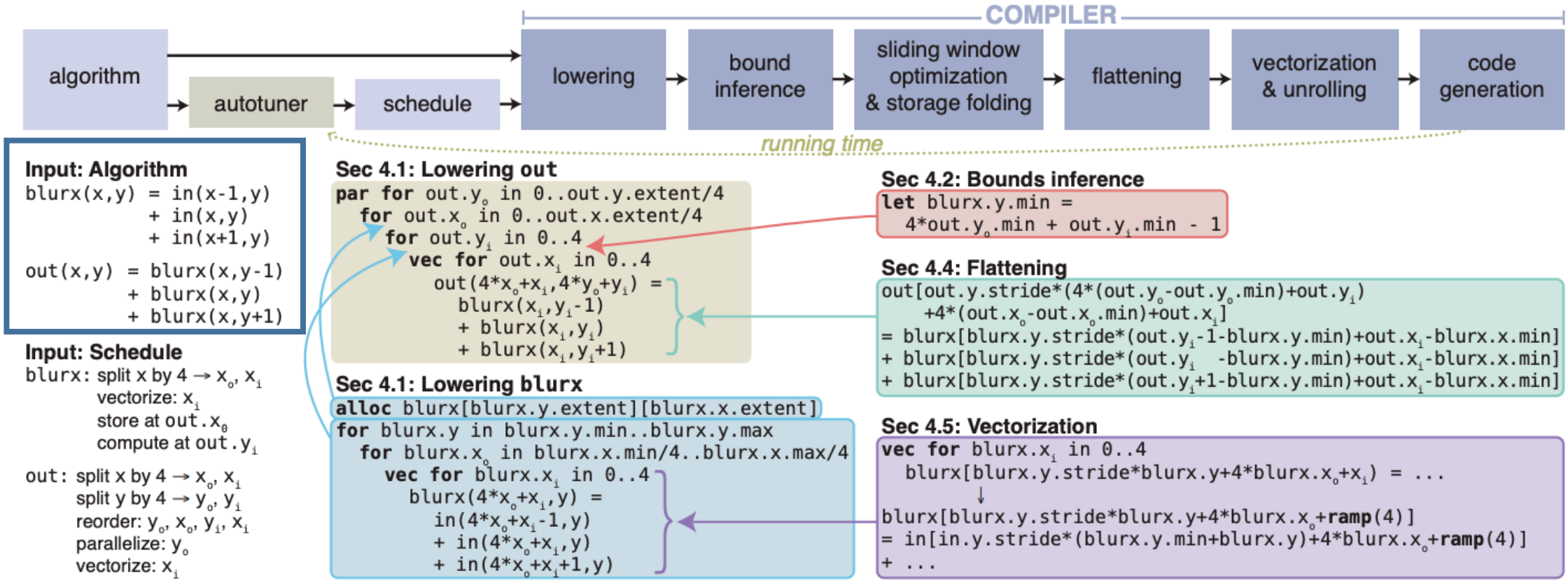
`blurx`: split x by 4 → $x_o$, $x_i$
vectorize: $x_i$
store at `out`.$x_o$
compute at `out`.$y_i$

`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: $x_i$

# Halide: Compiling Image Processing Pipelines

Key Innovation:
- Decouples ***algorithm from the compute***
- User only needs to provide the algorithm

Auto-tuner can select the optimal "schedule"
- How to split the axis?
- How to vectorize?



# Halide DSL

```
Func blur_3x3(Func input) {
  Func blur_x, blur_y;
  Var x, y, xi, yi;

  // The algorithm - no storage or order
  blur_x(x, y) = (input(x-1, y) + input(x, y) + input(x+1, y))/3;
  blur_y(x, y) = (blur_x(x, y-1) + blur_x(x, y) + blur_x(x, y+1))/3;

  // The schedule - defines order, locality; implies storage
  blur_y.tile(x, y, xi, yi, 256, 32)
        .vectorize(xi, 8).parallel(y);
  blur_x.compute_at(blur_y, x).vectorize(x, 8);

  return blur_y;
}
```

- Functional Language
- Embed in C++
- Much Simpler than writing threaded or CUDA program
- Downside:
  - Still requires domain experts to tune it
  - Not built for Deep Learning
    - TC: Assume infinite input range, cannot be optimized for fixed ops.
    - TVM: No special memory scope; no custom hardware intrinsics

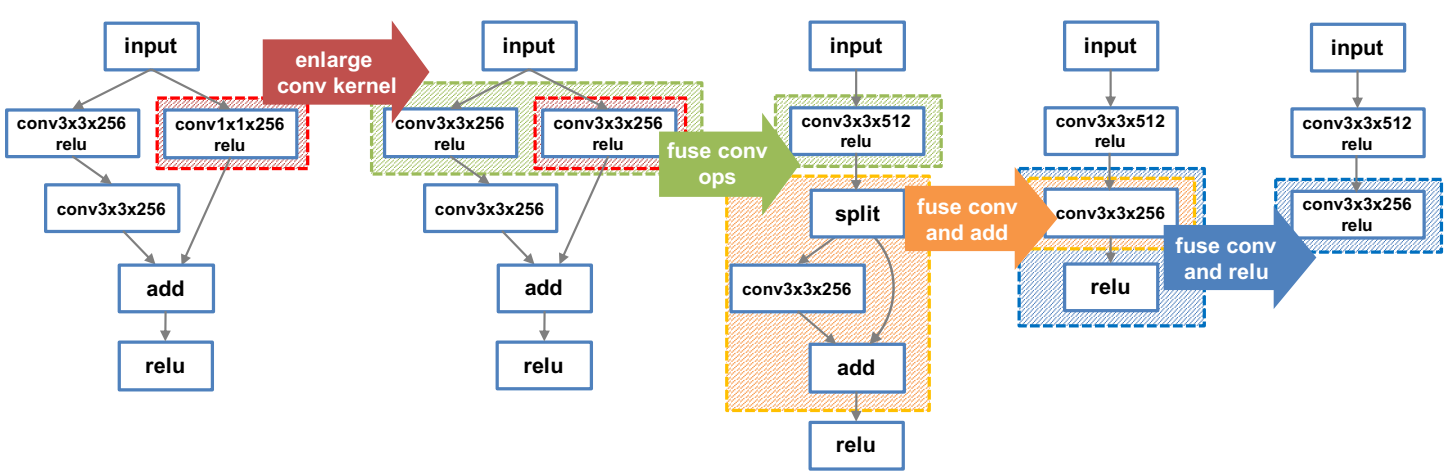
# Reading This Week

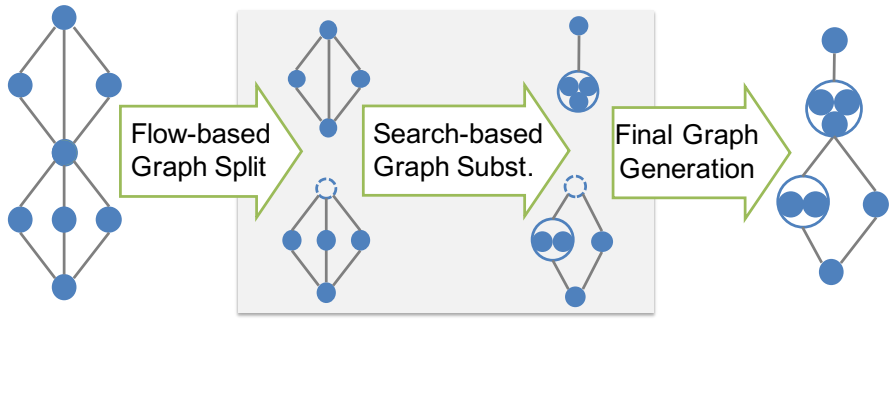
# Reading for the Week

- Optimizing DNN Computation with Relaxed Graph Substitutions (SysML'19)
  - Improves the graph search but does not modify individual ops.
  - Leverages basic cost model
- TVM: An Automated End-to-End Optimizing Compiler for Deep Learning (OSDI'18)
  - Optimizes graph and then individual tensor operations
  - Uses learning based approach
- Learning to Optimize Halide with Tree Search and Random Programs (TOG'19)
  - Schedule optimization in Halide using hybrid learning based approach

# MetaFlow (SysMI'19)

- Optimizes graph only by transforming groups of operators into revised versions of existing operators

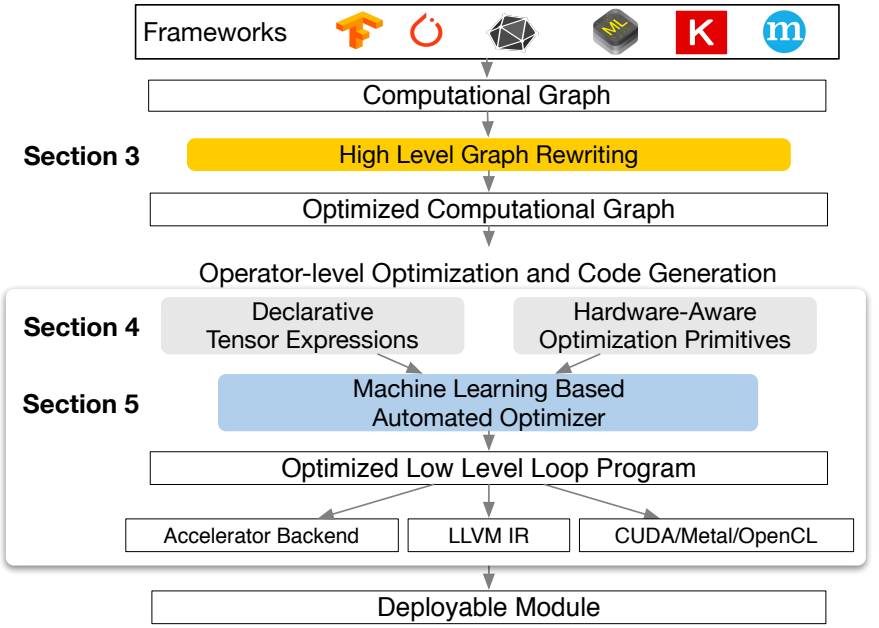



- **Key Insights:**
  - Use "**backtracking**" search to allow for less myopic opt.
  - **Cluster ops.** using dep. flow analysis to identify subgraphs
  - **Static operator impl.** have predictable costs

# TVM

- Originally derived from Halide
  - Leverages similar IR and separation of algorithm from schedule




```
import tvm

m, n, h = tvm.var('m'), tvm.var('n'), tvm.var('h')
A = tvm.placeholder((m, h), name='A')
B = tvm.placeholder((n, h), name='B')


k = tvm.reduce_axis((0, h), name='k')
C = tvm.compute((m, n), lambda i, j: tvm.sum(A[i, k] * B[j, k], axis=k))
```

**Inputs**

**Shape of C**

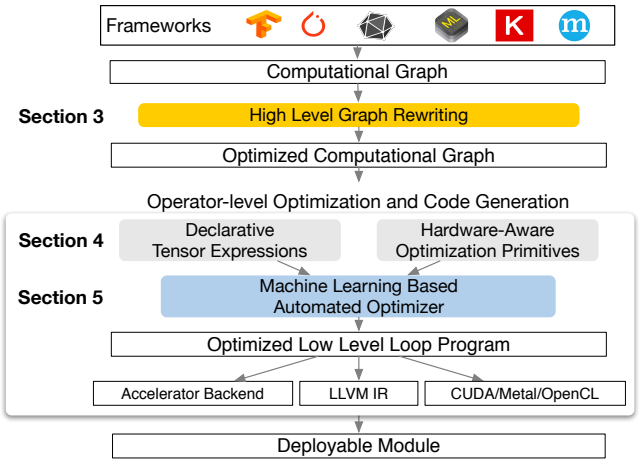
**Computation Rule**

Guess what this describes?

```
out = tvm.compute((c, h, w),
  lambda i, x, y: tvm.sum(data[kc,x+kx,y+ky] * w[i,kx,ky], [kx,ky,kc]))
```

# TVM

- Enables declaring new hardware intrinsics
  - Simplifies adding support for new hardware



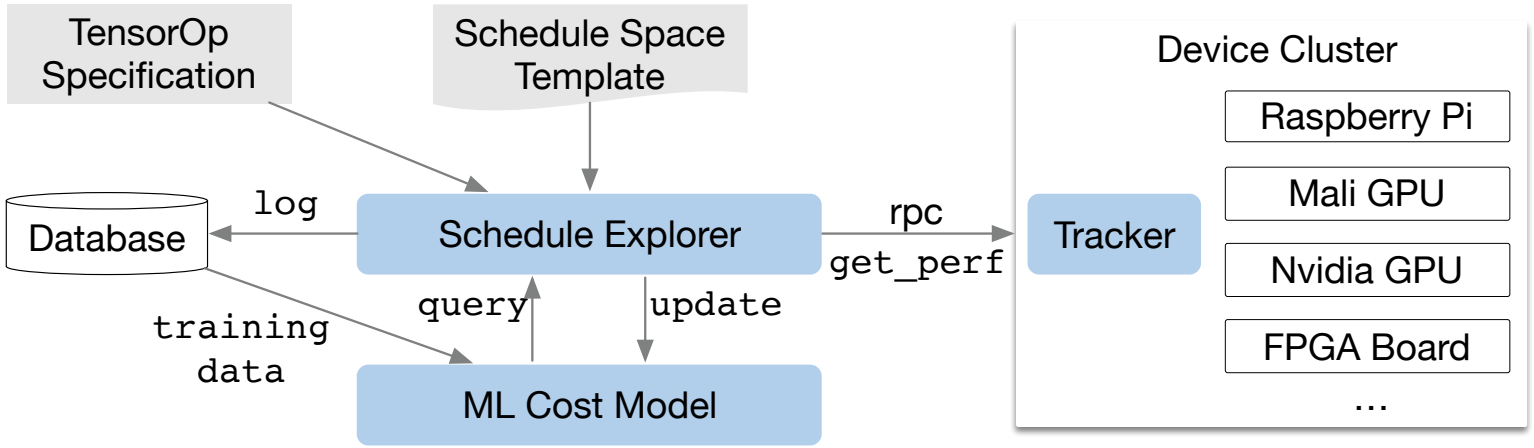

```
w, x = t.placeholder((8, 8)), t.placeholder((8, 8))
k = t.reduce_axis((0, 8))
y = t.compute((8, 8), lambda i, j:
              t.sum(w[i, k] * x[j, k], axis=k))

def gemm_intrin_lower(inputs, outputs):
   ww_ptr = inputs[0].access_ptr(“r")
   xx_ptr = inputs[1].access_ptr("r")
   zz_ptr = outputs[0].access_ptr("w")
   compute = t.hardware_intrin("gemm8x8", ww_ptr, xx_ptr, zz_ptr)
   reset = t.hardware_intrin("fill_zero", zz_ptr)
   update = t.hardware_intrin("fuse_gemm8x8_add", ww_ptr, xx_ptr, zz_ptr)
   return compute, reset, update

gemm8x8 = t.decl_tensor_intrin(y.op, gemm_intrin_lower)
```

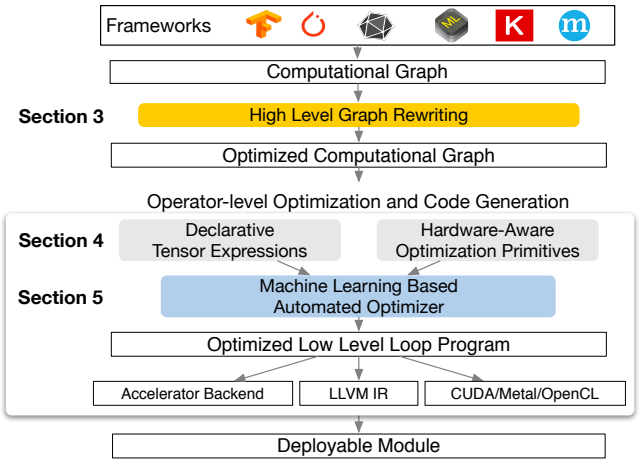
declare behavior

lowering rule to generate hardware intrinsics to carry out the computation

# TVM

- Learning based auto-tuner




```
for y in range(8):
  for x in range(8):
    C[y][x]=0
    for k in range(8):
      C[y][x]+=A[k][y]*B[k][x]
```

(a) Low level AST

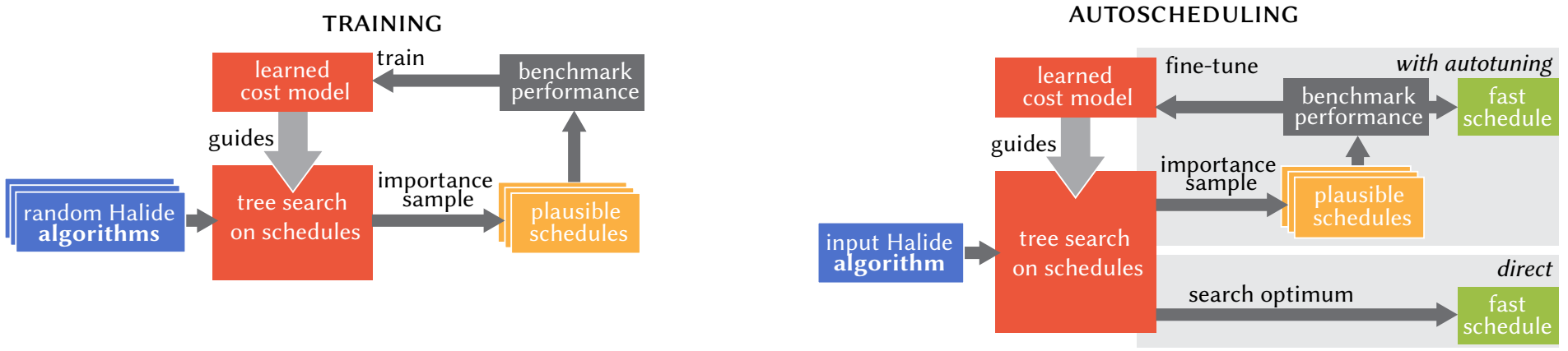
| | touched memory | | | outer loop length | |
|---|---|---|---|---|---|
| | C | A | B | | |
| y | 64 | 64 | 64 | y | 1 |
| x | 8 | 8 | 64 | x | 8 |
| k | 1 | 8 | 8 | k | 64 |

(b) Loop context vectors

- Parametrized the AST
- Use Gradient Boosted Trees (GBT) to optimize a "rank loss" to predict the relative order of program runtime

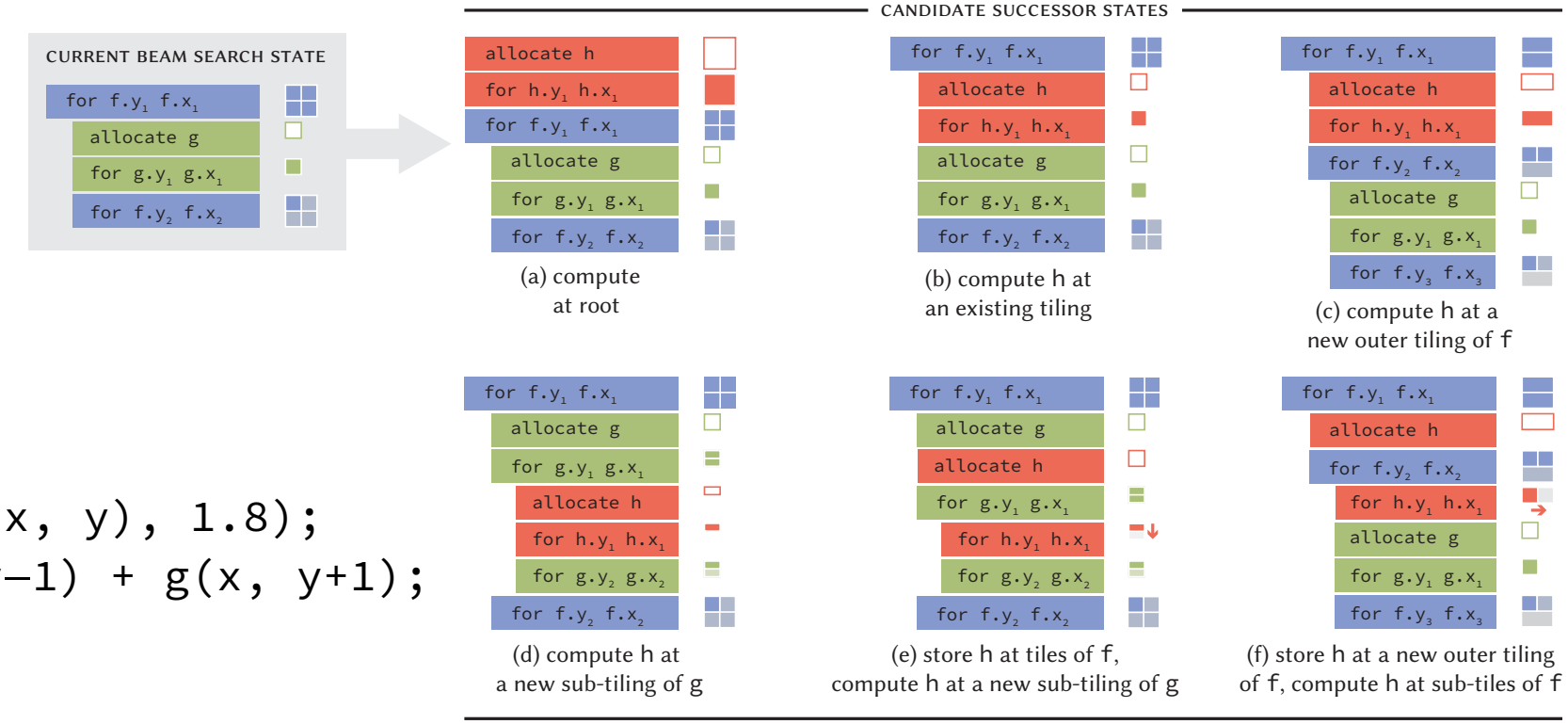
# Learning to Optimize Halide with Tree Search and Random Programs

- Published in ACM Transactions of Graphics (2019)
  - Halide grew out of graphics community
- Addresses missing scheduler optimizer + auto-tuner
  - Adopts learning based approach



# Learning to Optimize Halide with Tree Search and Random Programs

- **Beam search** of rich **schedule space**
  - Beam search ~ breadth first search with pruning
  - Search is constructed inductively from final stage in pipeline




```
h(x, y) = ...;
g(x, y) = pow(h(x, y), 1.8);
f(x, y) = g(x, y−1) + g(x, y+1);
```

# Done!