15-884: Machine Learning Systems

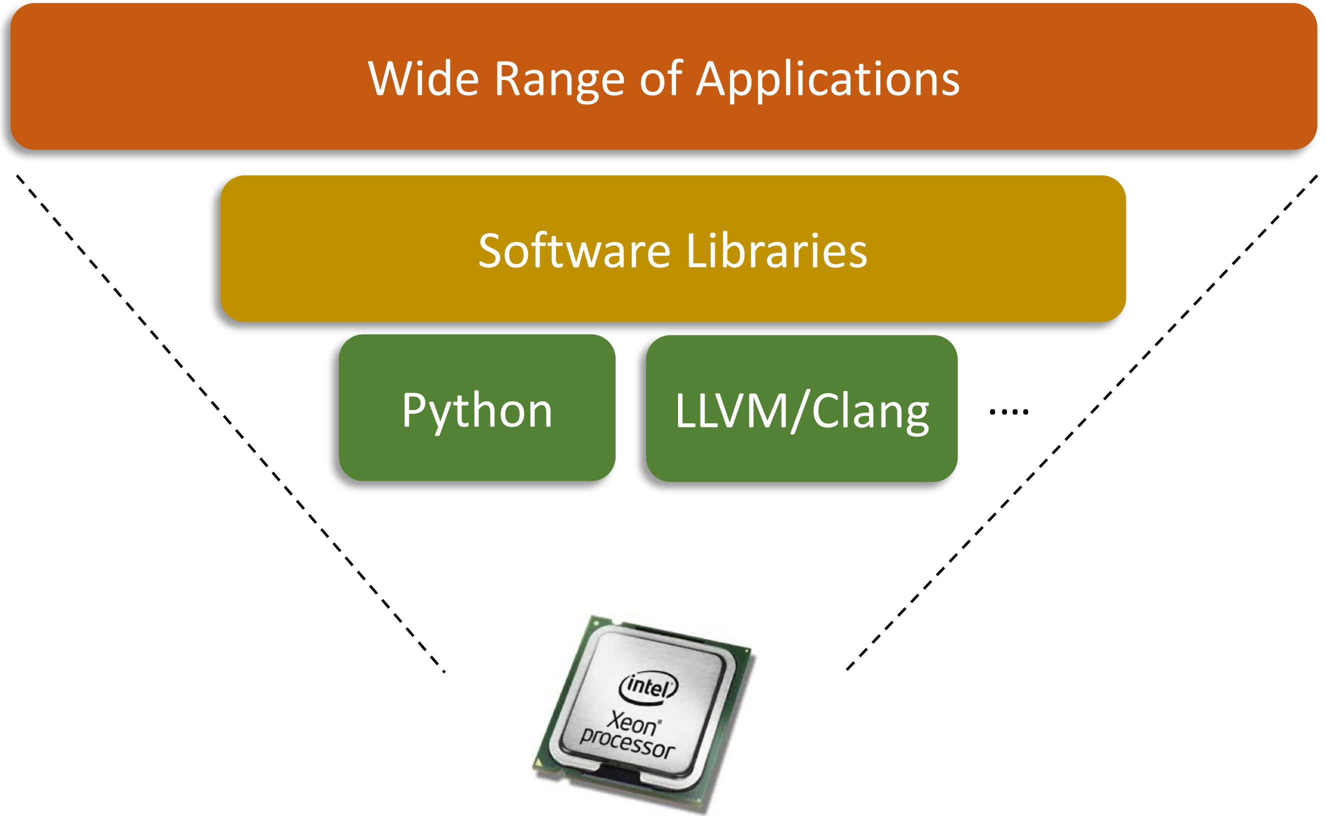
Machine Learning Compilation

Instructor: Tianqi Chen

Carnegie Mellon University
School of Computer Science

catalyst

Software Landscape



- Broad coverage.
- Compute performance is less critical.
- Engineers handles most optimizations

AI Software Landscape

ML Models

Transformer, ResNet, LSTM …

Data

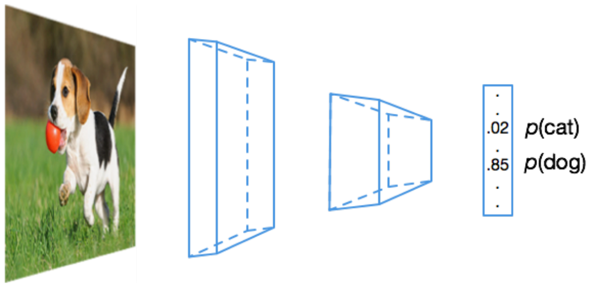
…

- Diverse and fast evolving models
- Big data
- Specialized compute acceleration

Compute

ML System Optimization Problem

ML Models



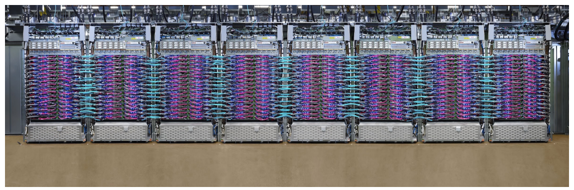
- Specialized libraries for each backend (labor intensive)
- Non-automatic optimizations

MKL-DNN

cuDNN

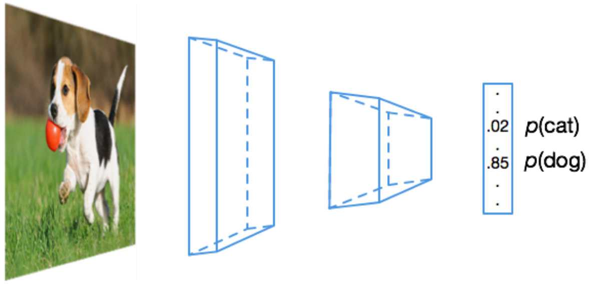
ARM-Compute

TPU Backends



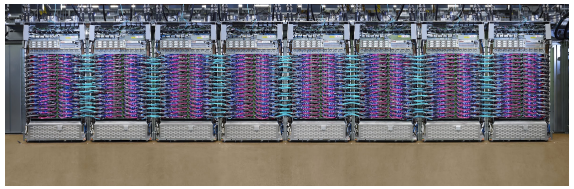
ML Compilation

ML Models



ML Compiler

Direct code generation

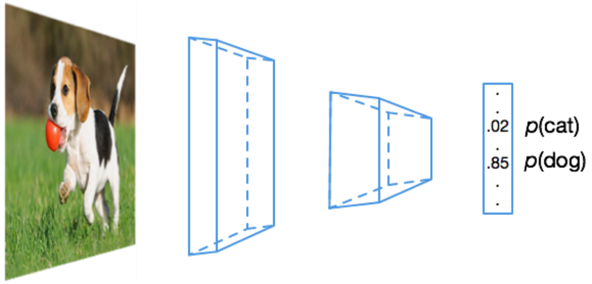


Discussion

- What would an end-to-end compilation flow for a ML model looks like
- What are the possible challenges ?

ML Compilation

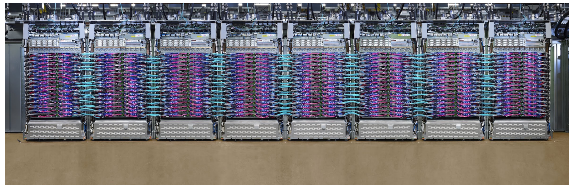
ML Models



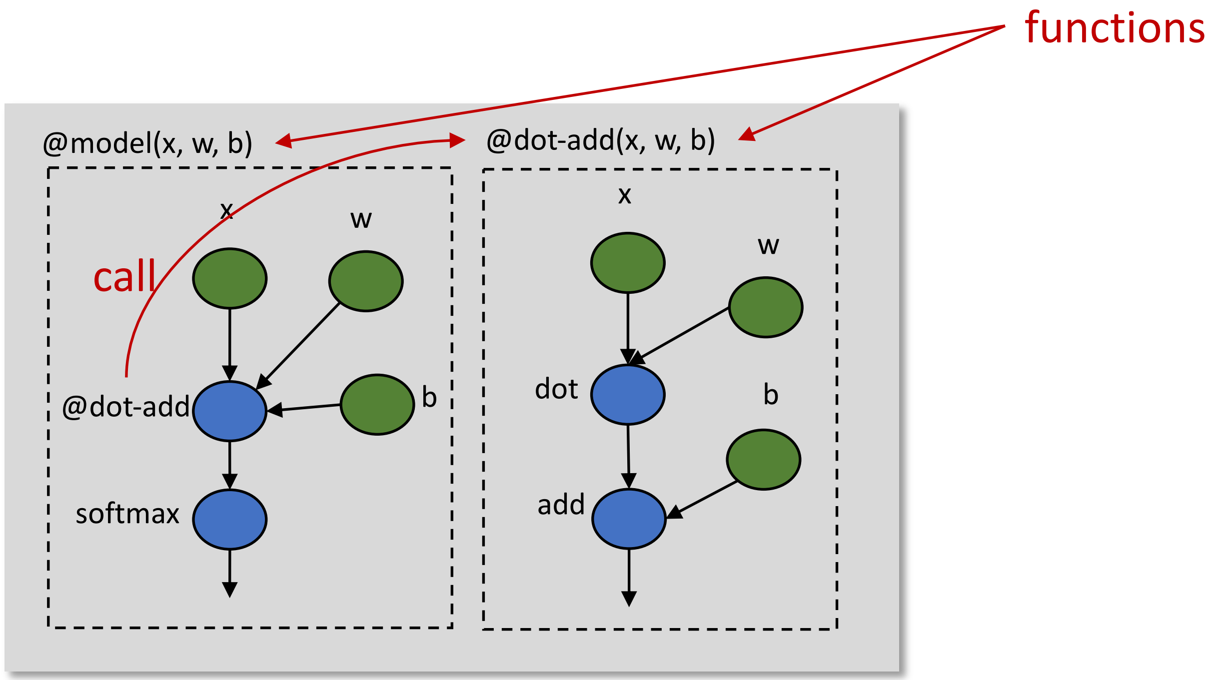
High-level IR Optimizations and Transformations

Tensor Operator Level Optimization

Direct code generation

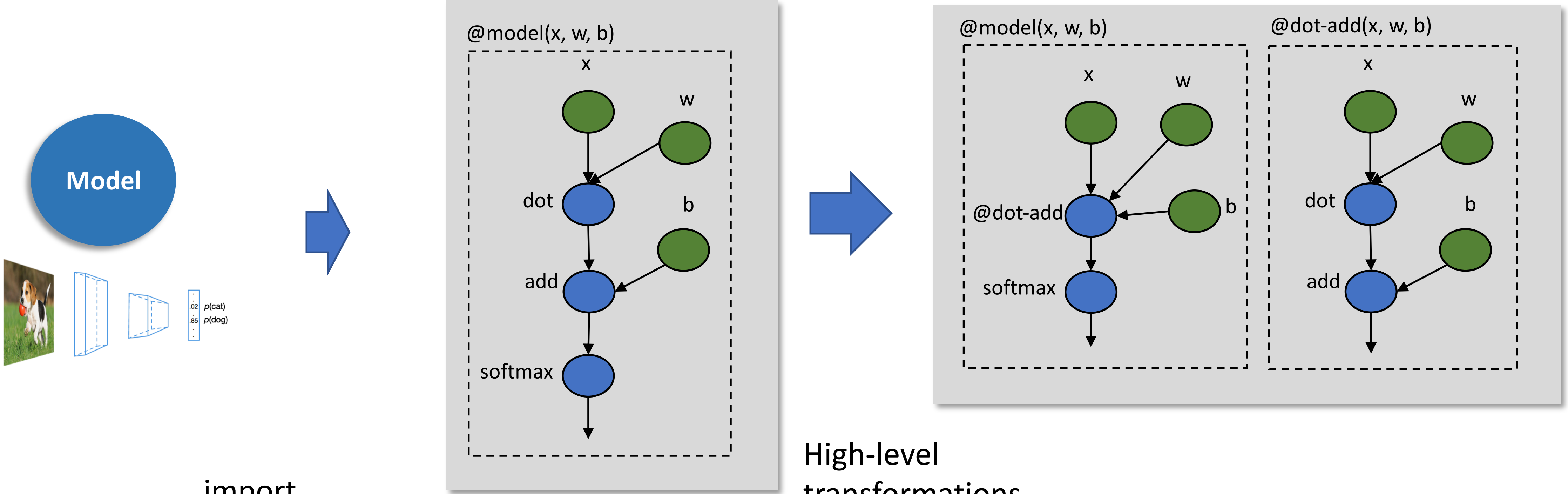


Compiler Representation of a ML Model



IRModule: a collection if interdependent functions

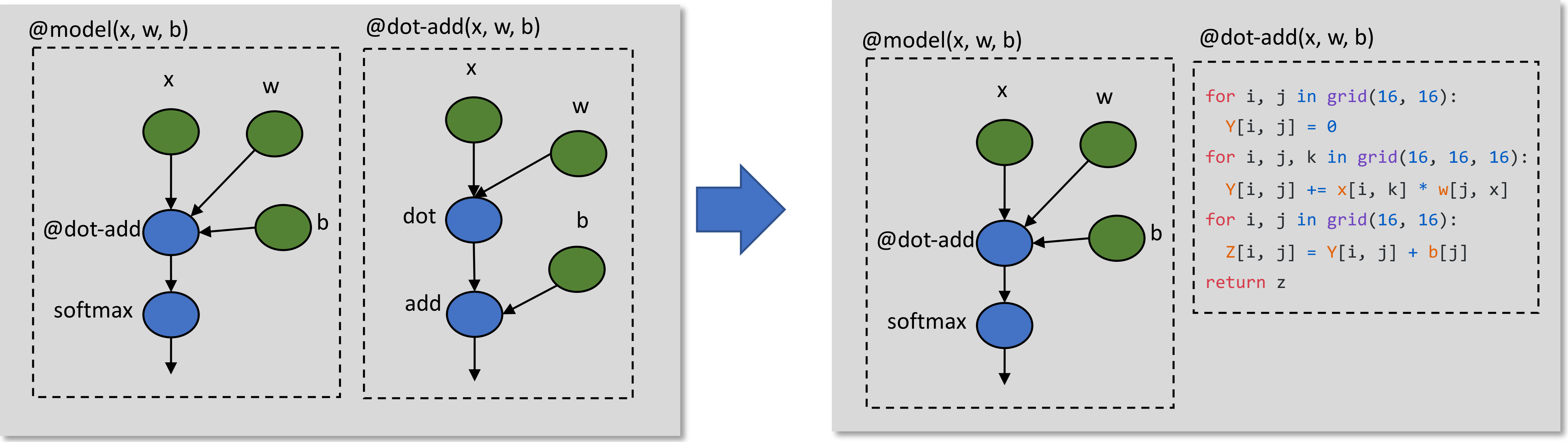
Example Compilation Flow: High-Level Transformations



import

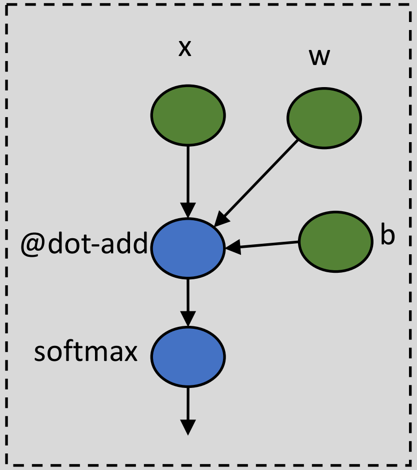
High-level transformations

Example Compilation Flow: Lowering to Loop IR



Example Compilation Flow: Low Level Transformations

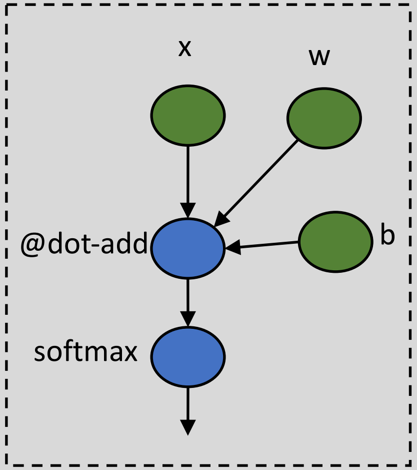
@model(x, w, b)



@dot-add(x, w, b)

```
for i, j in grid(16, 16):
  Y[i, j] = 0
for i, j, k in grid(16, 16, 16):
  Y[i, j] += x[i, k] * w[j, x]
for i, j in grid(16, 16):
  Z[i, j] = Y[i, j] + b[j]
return Z
```

@model(x, w, b)

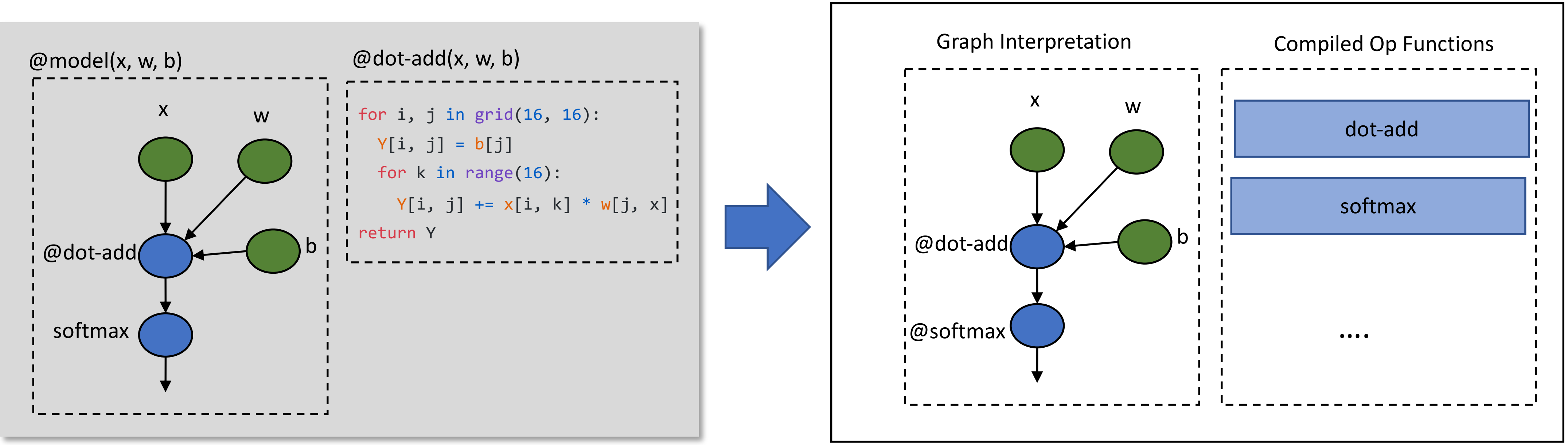


@dot-add(x, w, b)

```
for i, j in grid(16, 16):
  Y[i, j] = b[j]
  for k in range(16):
    Y[i, j] += x[i, k] * w[j, x]
return Y
```

Low-level transformations

Example Compilation Flow: CodeGen and Execution

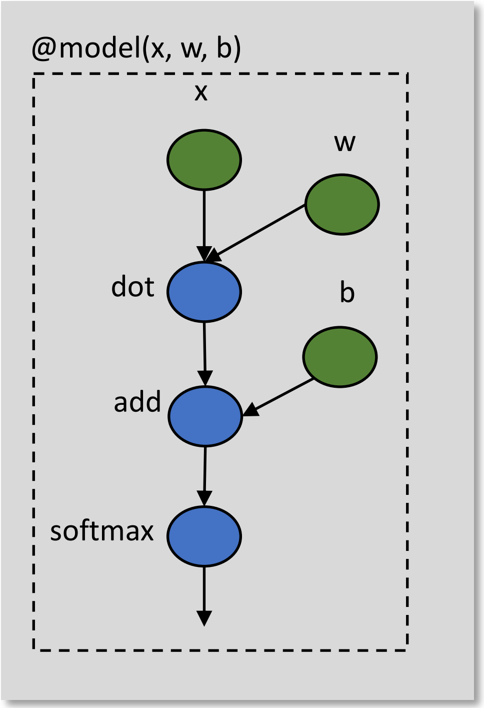


Runtime Execution

Discussion

- What are possible ways to represent a function in ML
- The possible set of optimizations we can perform in each type of representations.

High-level IR and Optimizations

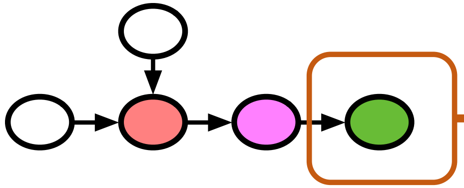


- Computation graph(or graph-like) representation
- Each node is a tensor operator(e.g. convolution)
- Can be transformed (e.g. fusion) and annotated (e.g. device placement)
- Most ML frameworks have this layer

Covered in previous lectures

Tensor Operator Level Optimizations

Low-level Code Optimization



Specification

```
C = tvm.compute((m, n),
   lambda y, x: tvm.sum(A[k, y] * B[k, x], axis=k))
```

Search Space of Possible Program Optimizations

Low-level Program Variants

```
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)
```

```
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]
```

```
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]
```

Elements of Low-level Loop Representation

@dot-add(x, w, b)

```
for i, j in grid(16, 16):
  Y[i, j] = 0
for i, j, k in grid(16, 16, 16):
  Y[i, j] += x[i, k] * w[j, x]
for i, j in grid(16, 16):
  Z[i, j] = Y[i, j] + b[j]
```

Multi-dimensional buffer

Loop nests

Array computation

Transforming Loops: Loop Splitting

Code

```
for x in range(128):
  C[x] = A[x] + B[x]
```

```
for xo in range(32):
  for xi in range(4):
     C[xo * 4 + xi]
      = A[xo * 4 + xi] + B[xo * 4 + xi]
```

Transformation

```
x = get_loop("x")
xo, xi = split(x, 4)
```

Transforming Loops: Loop Reorder

Code

```
for xo in range(32):
  for xi in range(4):
     C[xo * 4 + xi]
      = A[xo * 4 + xi] + B[xo * 4 + xi]
```

```
for xi in range(4):
  for xo in range(32):
    C[xo * 4 + xi]
      = A[xo * 4 + xi] + B[xo * 4 + xi]
```

Transformation

```
x = get_loop("x")
xo, xi = split(x, 4)
reorder(xi, xo)
```

Transforming Loops: Thread Binding

Code

```
for xi in range(4):
  for xo in range(32):
    C[xo * 4 + xi]
      = A[xo * 4 + xi] + B[xo * 4 + xi]
```

```
def gpu_kernel():
  C[threadId.x * 4 + blockIdx.x] = . . .
```

Transformation

```
x = get_loop("x")
xo, xi = split(x, 4)
reorder(xi, xo)
bind_thread(xo, "threadIdx.x")
bind_thread(xi, "blockIdx.x")
```

Discussion

- What are other possible transformations.
- How to support specialized hardware (GPUs, accelerators)

Hardware for ML Becoming Specialized

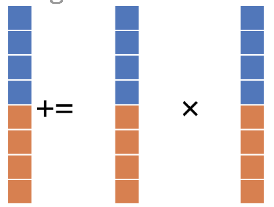
Generic FMA

```
# semantics
C[0] += A[0] * B[0]

# implementation
llvm.fmuladd.f32
```

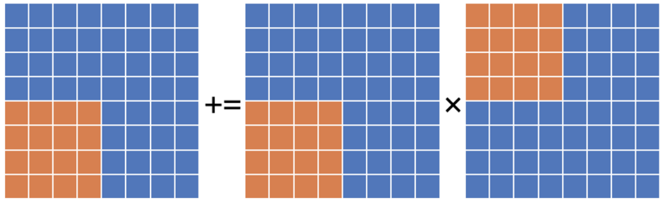
Vector FMA

```
# semantics
for i in range(4):
  C[i] += A[i] * B[i]
# implementation
llvm.fmuladd.v4f32
# diagram
```



Nvidia Tensor Core and NPUs

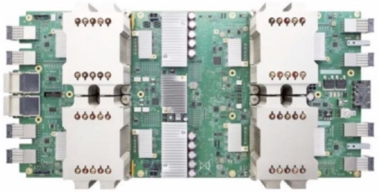
```
# semantics
for y, x, k in grid(16, 16, 16):
  C[y, x] += A[y, k] * B[k, x]
# implementation
nvvm.wmma.m16n16k16.mma.row.row.f32.f32
# diagram
```

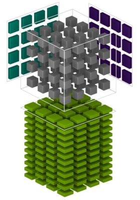


Scalar unit

SIMD,
vector units

Specialized
tensor instructions





Elements of Tensorized Program

```
for ic.outer, kh, ic.inner, kw in grid(...):

    for ax0 in range(...):
      load_matrix_sync(A.shared.wmma.matrix_a, 16, 16, 16, ...)

    for ax0 in range(...):
      load_matrix_sync(W.shared.wmma.matrix_b, 16, 16, 16, ...)

    for n.c, o.c in grid(...):
      wmma_sync(Conv.wmma.accumulator,
                A.shared.wmma.matrix_a,
                W.shared.wmma.matrix_b, ...)

for n.inner, o.inner in grid(...):
  store_matrix_sync(Conv.wmma.accumulator, 16, 16, 16)
```

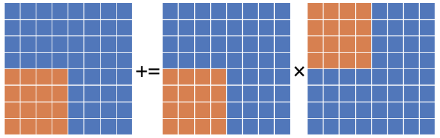
Example Snippet: Conv2D on TensorCore

Optimized loop nests with thread binding

Multi-dimensional data load into **specialized memory buffer**

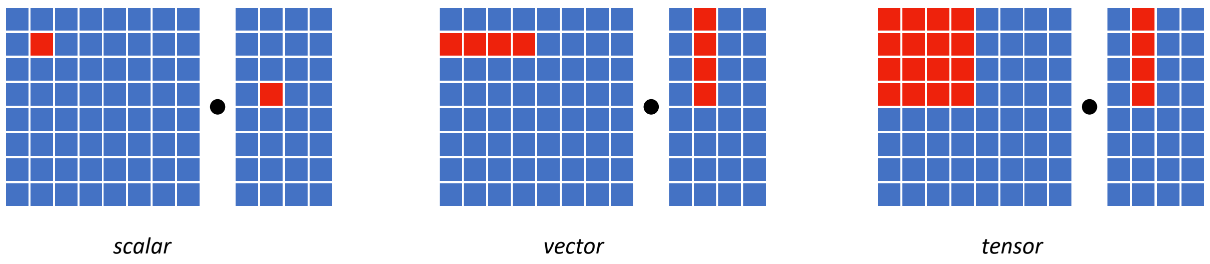
Opaque tensorized computation body

16x16 matrix multiplication



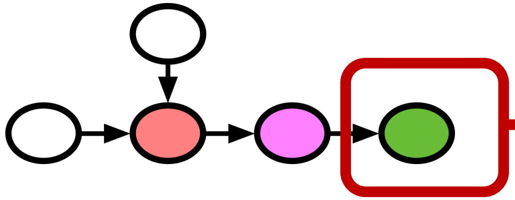
Tensorization Challenge

Compute primitives



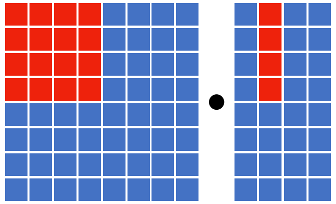
Challenge: build compiler for all kinds of compute primitives

Tensorization Challenge



Specification

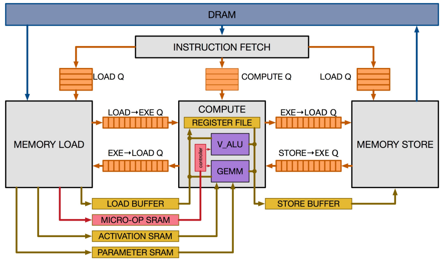
```
C = tvm.compute((m, n),
    lambda y, x: tvm.sum(A[k, y] * B[k, x], axis=k))
```



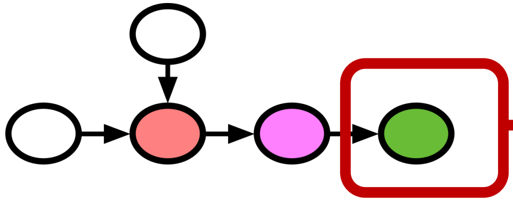
```
A = tvm.placeholder((8, 8))
B = tvm.placeholder((8,))
k = tvm.reduce_axis((0, 8))
C = tvm.compute((8, 8),
   lambda y, x: tvm.sum(A[k, y] * B[k], axis=k))
```

HW Interface Specification

Tensorization



Big Space of Possible Transformations



Specification

```
C = tvm.compute((m, n),
    lambda y, x: tvm.sum(A[k, y] * B[k, x], axis=k))
```

Huge space of possible choices

Loop Transformations

Thread Bindings

Cache Locality

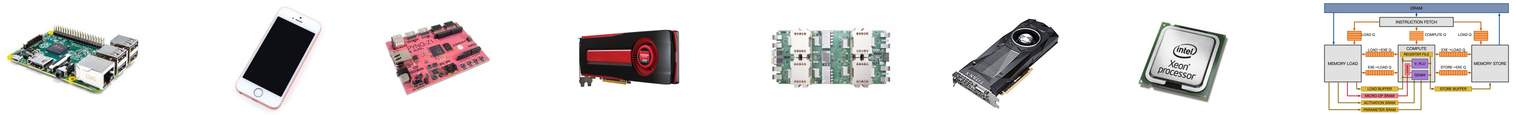
Thread Cooperation

Tensorization

Latency Hiding

...

Hardware



Summary

- Collections of function as basic unit in ML Compilation
- Effective transformation primitives for low-level optimizations
- Need automation (next lecture)