Lecture 16:

- Dif. ML frameworks and high-level differences
- ML compilation
  - ↳ Graph substitution
  - ↳ Low-level code generation
  - ↳ ways of automatin

Part 1: Tensorflow vs. Pytorch

- "eager execution"
  - ↳ Pytorch builds computational graphs (set of matrix ops needed to compute FW and BW pass) as code is executing
    - actual functions (e.g., "torch.relu") will call into optimized c/c++/cuda implementations
    - but intermediate state in computation graph is available to developers
    - recall: pytorch constructs autograd graph for BW pass *during FW pass*
- "static graph execution"
  - ↳ main difference: compilation step

added

- no intermediate state available DURING execution (in python) - only would be available in underlying C++/C process

- Tensorflow v1 = graph (static) execution
- Pytorch = eager execution

* now, both frameworks mix both:
  - tensorflow has "eager" execution mode
  - pytorch has JIT compilation (for main training/inference code)
    - but other code (e.g., preparing training data) can still be done eagerly

Part 2: what is ML compilation?

- we have:
  1) new ML models/architectures (think FW/BW functions you are writing for project 2!)
     - or new nn.modules, new operators
  2) Different HW backends!

(GPU DNN backend)

MKL-DNN | cuDNN | ARM-Compute | TPU Backends

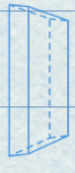


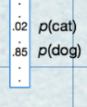
→ right now, dif. libraries exist for each accelerators

* we potentially also want to DISTRIBUTE computation across HW backends

* core question: how do we map dif. models/operators automatically to dif. HW?

- idea: use a "compiler":
- Two stages of compilation:

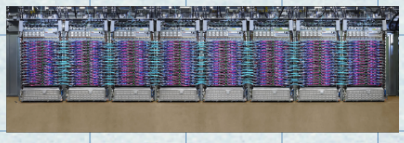


"graph-level substitutions" → High-level IR Optimizations and Transformations

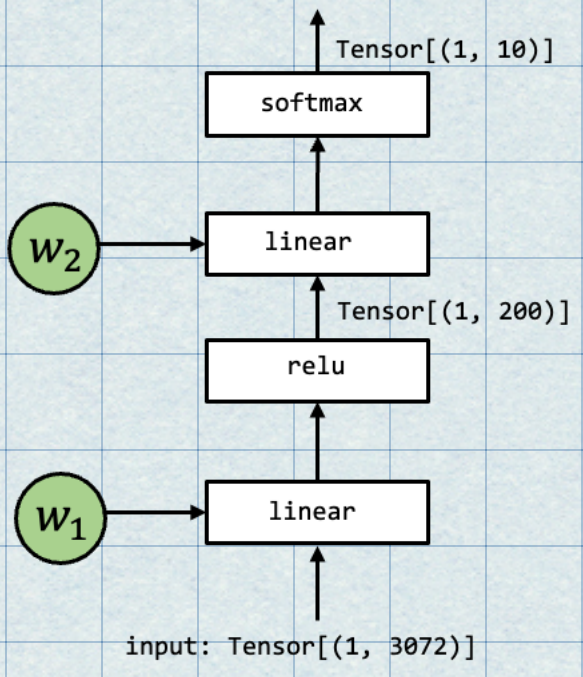
Tensor Operator Level Optimization → "code-level" optimizations

Direct code generation





- Graph-Level optimization



- Tensor: multidimensional array storing input, output and intermediate results of model execution

- Tensor functions: encode computations among input/output
  - ↳ could contain multiple operations

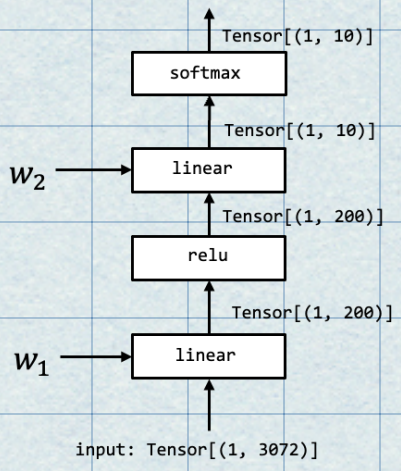
*3 general goals of compilation process:

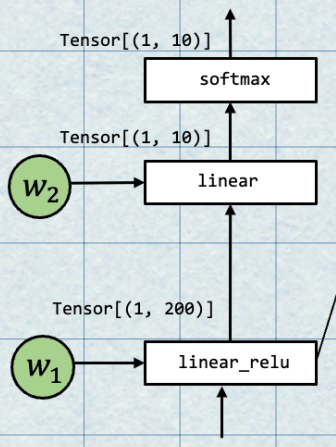
1) minimize memory usage
2) improve execution efficiency
3) scale to multiple, heterogeneous nodes

* Example compilation: replace relu w/ linear_relu

→ recall: linear: input · w_1 (1x3072 · 3072x200)

→ relu: $\max(x_i, 0)$ for all elements in x



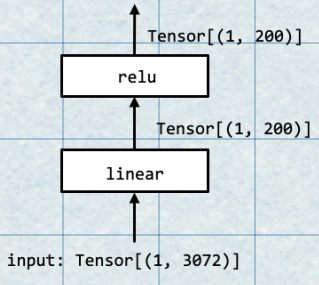


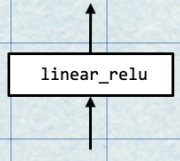

```
def linear_relu(x, w, out):
    for i in range(1):
        for j in range(200):
            out[i, j] = 0
            for k in range(3072):
                out[i, j] += x[i, k] * w[j, k]
            out[i, j] = max(out[i, j], 0)
```

Fuse: do max when already calculating output

- Discussion: why could this help in a GPU kernel?

* A note on terminology: abstraction vs. implementation





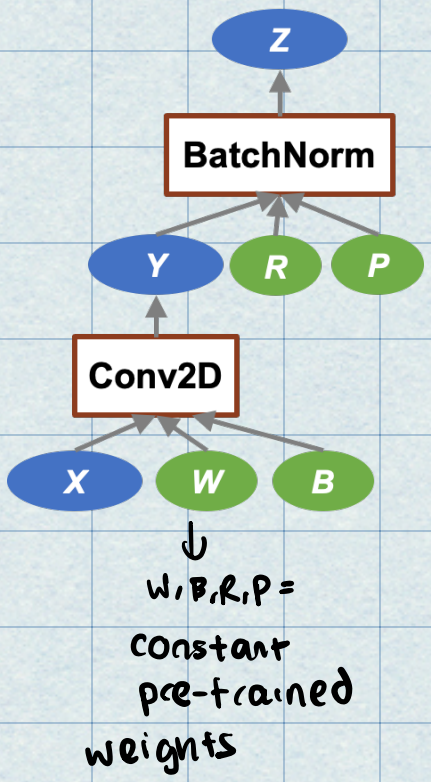

```
def linear_relu(x, w, out):
    for i in range(1):
        for j in range(200):
            out[i, j] = 0
            for k in range(3072):
                out[i, j] += x[i, k] * w[j, k]
            out[i, j] = max(out[i, j], 0)
```

- three ways to represent SAME tensor function, more/most specialized version is IMPLEMENTATION of higher-level ops

* some common "graph-level" optimizations:

1) Fusion

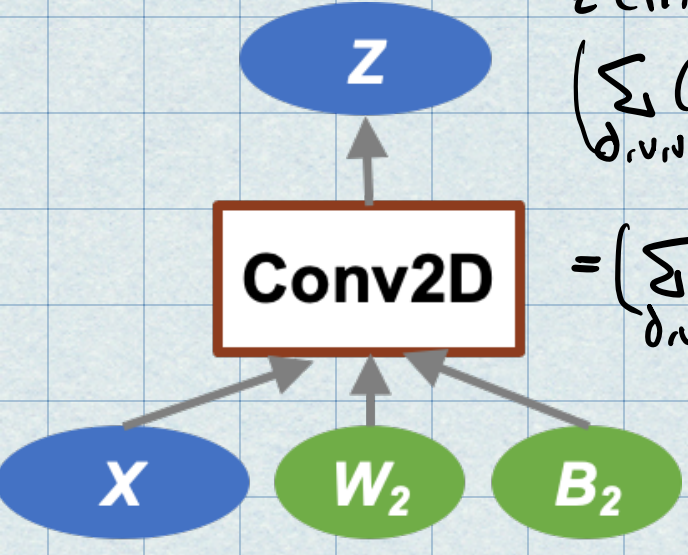
- recall example from beginning of class: fuse batchNorm + Conv



$$z(n,c,h,w) = y(n,c,h,w) \cdot R(c) + P(c)$$

$$y(n,c,h,w) = \sum_{d,u,v} \left(x(n,d,h+u,w+v) \cdot W(c,d,u,v) \right) + B(n,c,h,w)$$

- idea: fuse into a SINGLE conv2d w/ dif. pre-trained weights



substitute for y

$$z(n,c,h,w) = \left(\sum_{d,u,v} \left(X(n,d,h+u,v) \cdot W(c,d,u,v) \right) \cdot B(n,c,h,w) \right) \cdot R(c) + P(c)$$

$$= \left(\sum_{d,u,v} X(n,d,h+u,v) \cdot W_2(c,d,u,v) \right) + B_2(n,c,h,w)$$

$W_2 = W \cdot R$

$B_2 = B \cdot R + P$

2) Data-Layout transformations

- how do we store data in given HW?

↳ a DL accelerator might be optimized for 4x4 matrix operations
- so data should be tiled into 4x4 chunks

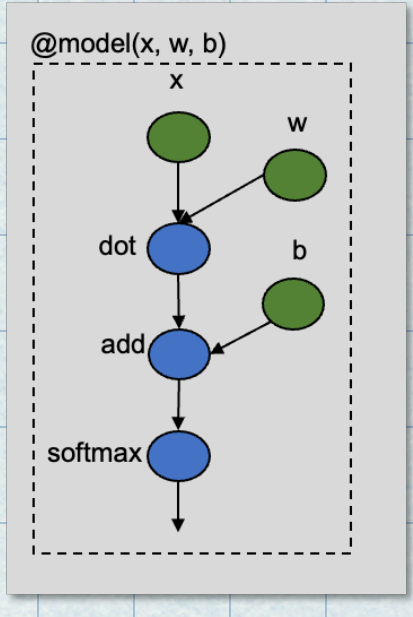
3) others:
- constant folding: pre-compute graph parts that can be determined statically
- static memory planning pass: pre-allocates memory to hold each tensor

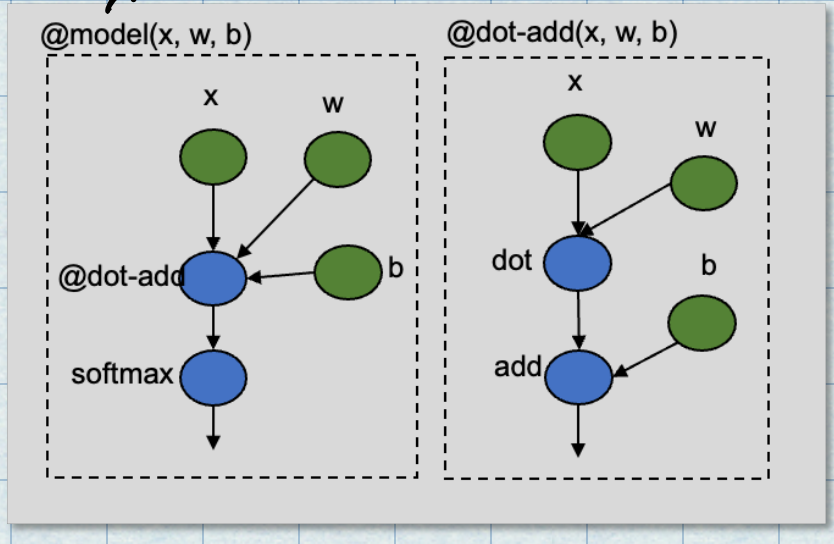
* similar optimizations are commonly made when compiling code to LLVM!

* next: LOW-LEVEL CODE OPTIMIZATIONS

- consider following operator graph (where fusion of dot, add → dot-add has been applied)

@ denotes functions





- what would "optimized" fused dot add look like?

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

1st half is now dot-alone would be done → DOT

2nd is add-alone → ADD

- what about? (Discuss...)

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

→ eliminate O(ng) out data

- Loop splitting

- why would we do this?

- perhaps we can keep tiles of data local

```
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]
```

to gpu shared s.m. memory, depending on GPU memory size

- Loop reorder

```
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]
```

] depending on data layout, one might be more effective

- How do we automate BOTH low-level code optimization and and graph substitution?

↳ think of it as a search problem!

* TASO, SOSP 19

- replace manually generated graph optimizations w/ automated

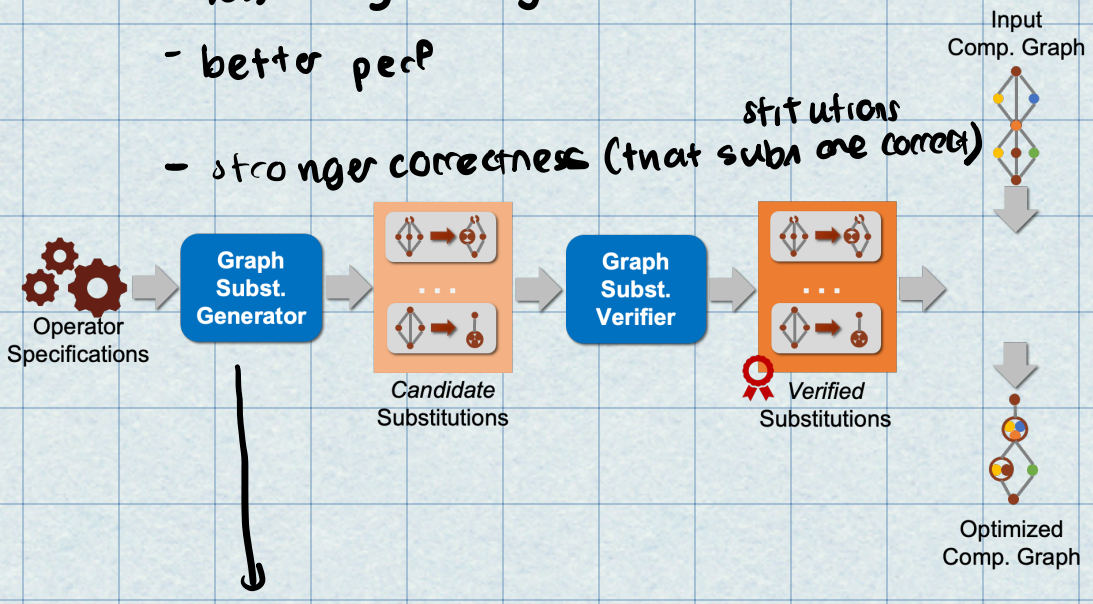
generation + verification of

graph substitutions for tensor

algebra

- less engineering effort
- better perf
- stronger correctness (that subs are correct)

stitutions



1. enumerate some # of potential graphs
2. Prune redundant subtitutions (where names are aliased)
3. prune common subgraphs

- then verify all candidates w/ theorem prover

* dif. from how TF applies optimizations!

- greedily apply substitutions vs. automatically find optimal program

TVM, OSDI 2018

- compiler for low-level code gen on dif. backends
- idea: write code in "high-level" tensor IR that expresses operations on tensors
- have "HW" search space that knows about mapping operators to specific accelerators w/ specific optimizations:
  - loop transformations
  - thread bindings (which parts of data are assigned to which gpu threads?)
  - cache locality
  - thread cooperation
  - tensorization
  - pipelining / latency hidings

* use ML-based learning optimizer to SEARCH
* now is apache open source project

Resources:

CMU CS-15-442 class slides on ML compilation: https://mlsyscourse.org/slides/07-machine-learning-compilation.pdf

CMU CS-15-442 class slides on graph optimizations: https://mlsyscourse.org/slides/08-graph-level-optimizations.pdf

TVM paper: https://www.usenix.org/conference/osdi18/presentation/chen (Tianqi Chen)

TASO paper: https://dl.acm.org/doi/pdf/10.1145/3341301.3359630 (Zhihao Jia)