Dive into Deep Learning for NLP

6. Deployment with TVM

Haichen Shen

gluon-nlp.mxnet.io

13:15-14:15	Natural Language Processing and Deep Learning Basics
14:15-14:25	Break
14:25-15:15	Context-free Representations with Word Embeddings
15:15-15:55	Machine Translation and Sequence Generation
15:55-16:35	Contextual Representations with BERT
16:35-16:45	Break
16:45-17:15	Model Deployment with TVM

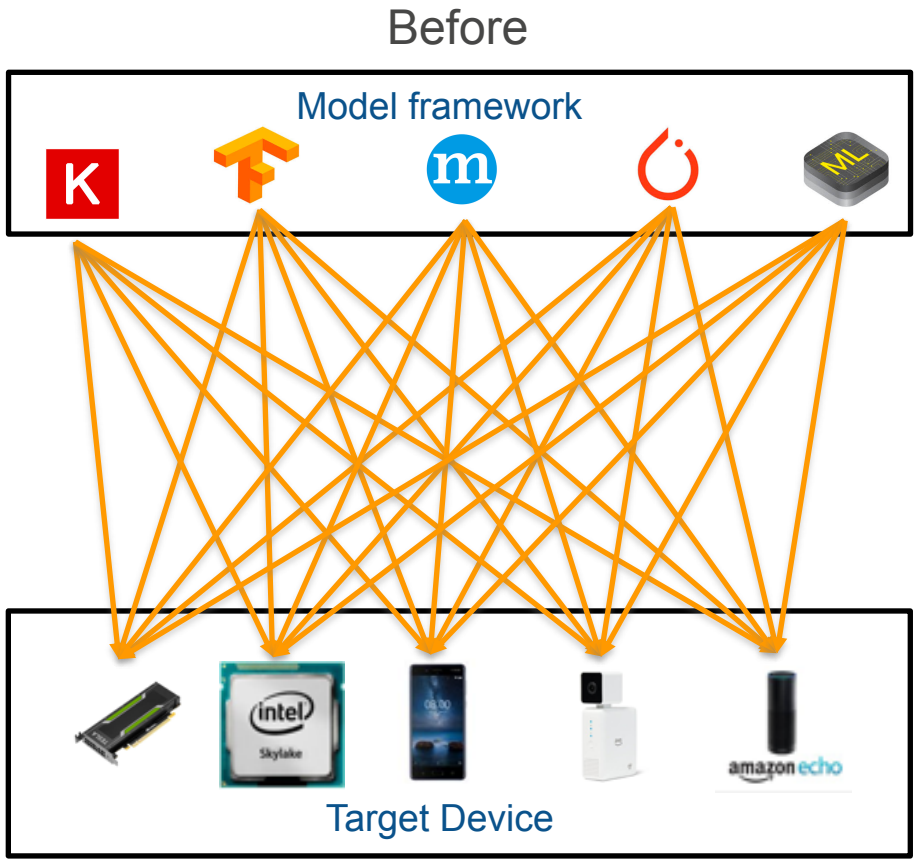
Deep learning inference is complicated

Before

gluon-nlp.mxnet.io

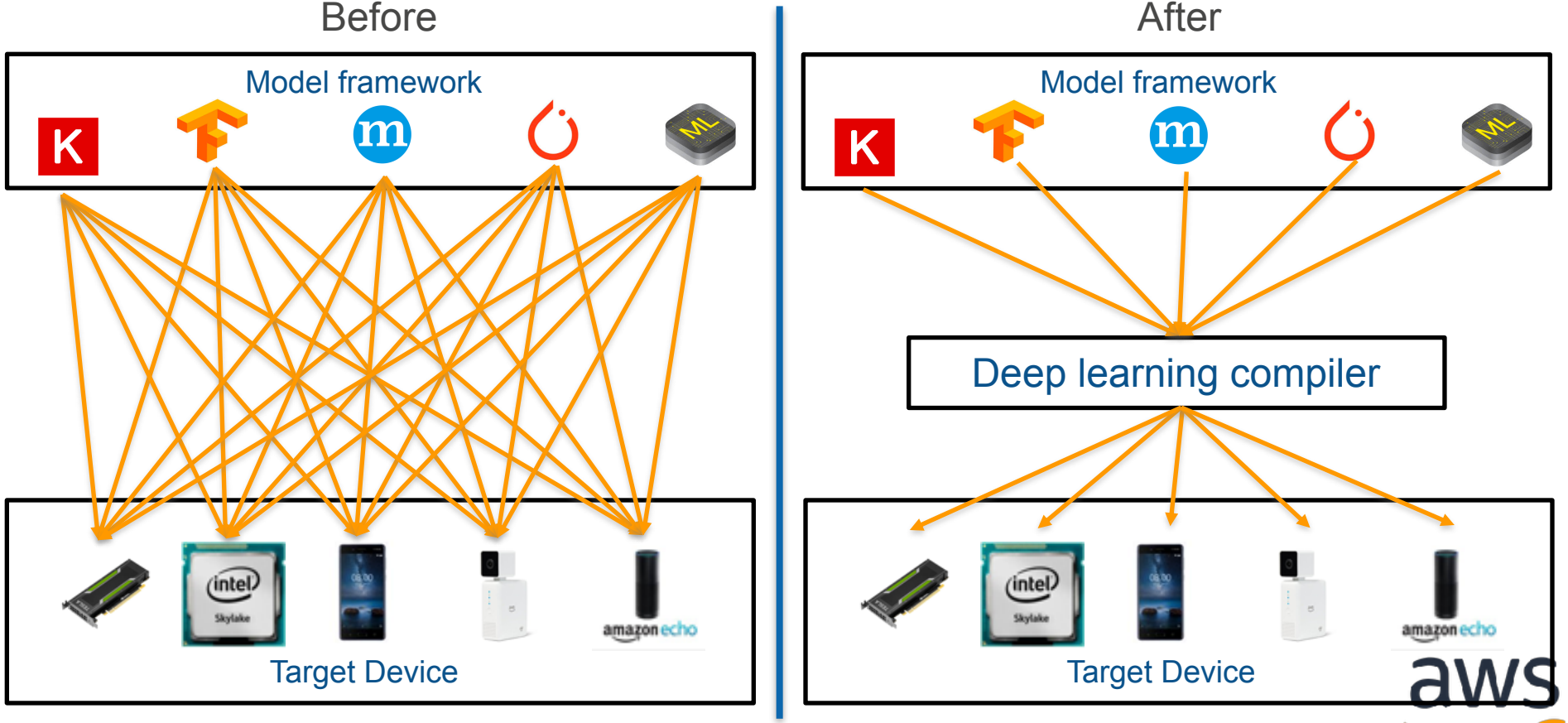
aws

Deep learning inference is complicated



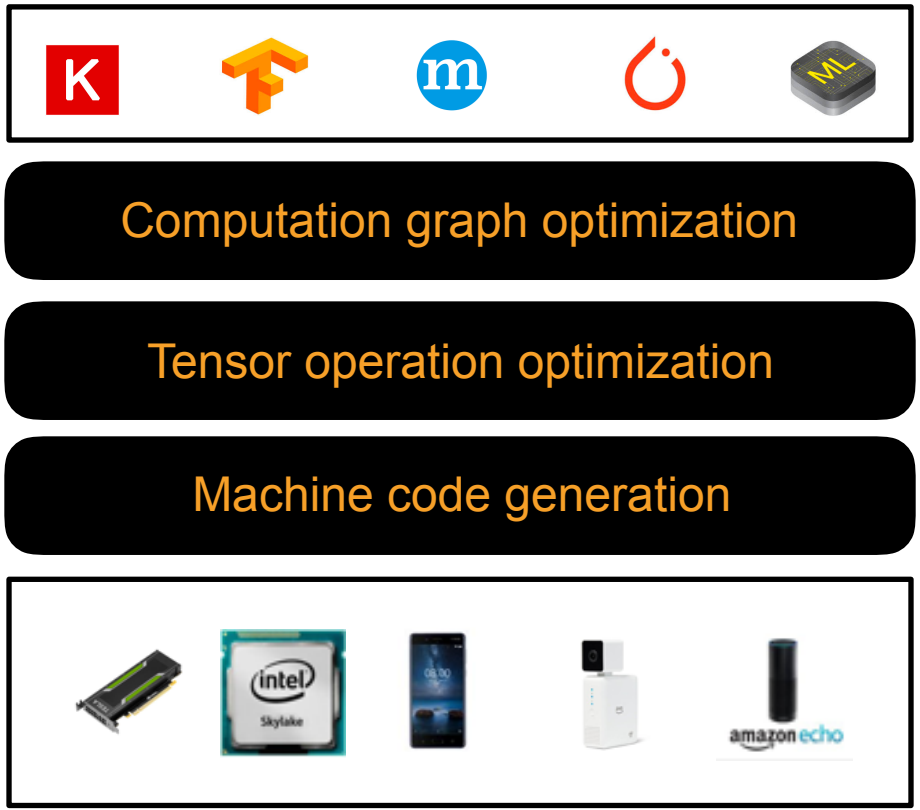
gluon-nlp.mxnet.io

Deep learning inference is complicated



gluon-nlp.mxnet.io

Models and hardware targets are far away!

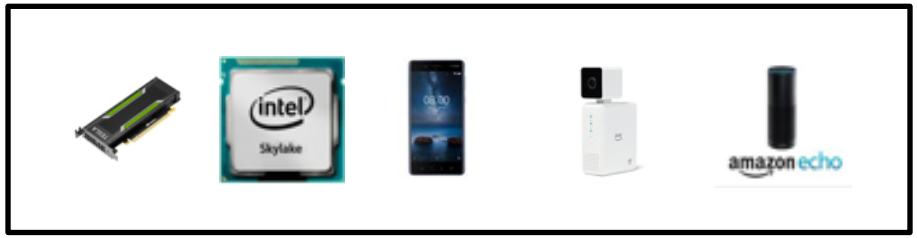


aws

End-to-end deep learning compiler

tvm

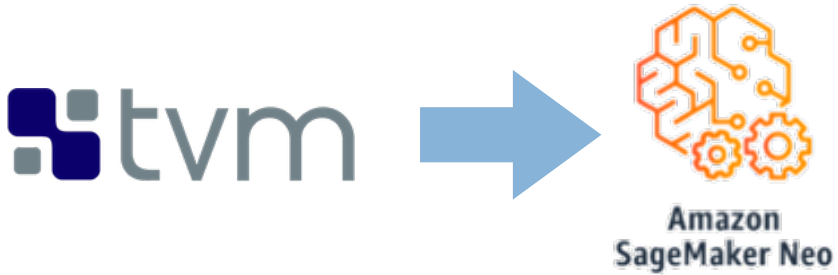
end-to-end optimization



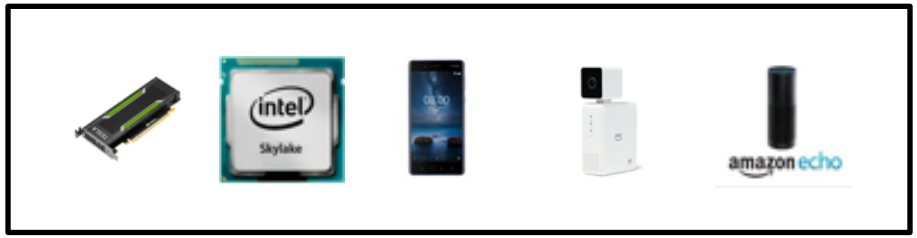
gluon-nlp.mxnet.io

aws

End-to-end deep learning compiler



end-to-end optimization



aws

TVM overview

Relay: High-Level Differentiable IR

Tensor Expression IR

LLVM, CUDA, Metal, …

Edge FPGA

Cloud FPGA

ASIC

aws

TVM overview

Relay: High-Level Differentiable IR

Tensor Expression IR

LLVM, CUDA, Metal, …

Edge FPGA | Cloud FPGA | ASIC

Optimization

- Graph optimization
- Tensor optimization
- AutoTVM

aws

TVM overview

Relay: High-Level Differentiable IR

Tensor Expression IR

LLVM, CUDA, Metal, …

Edge FPGA | Cloud FPGA | ASIC

Optimization

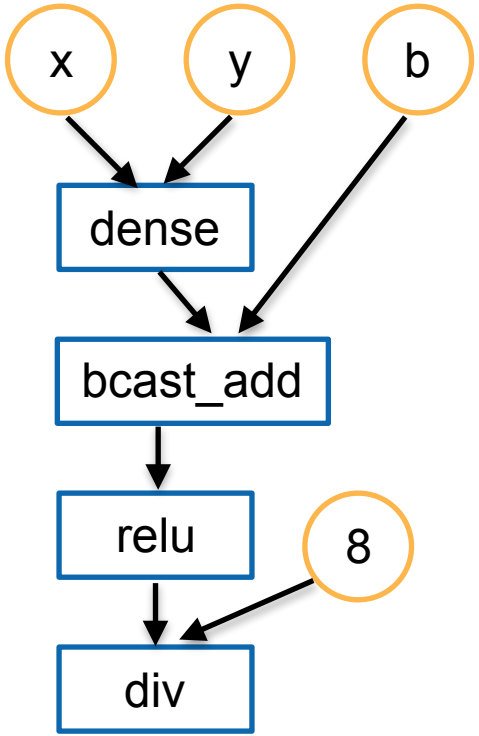
- Graph optimization
- Tensor optimization
- AutoTVM

Runtime

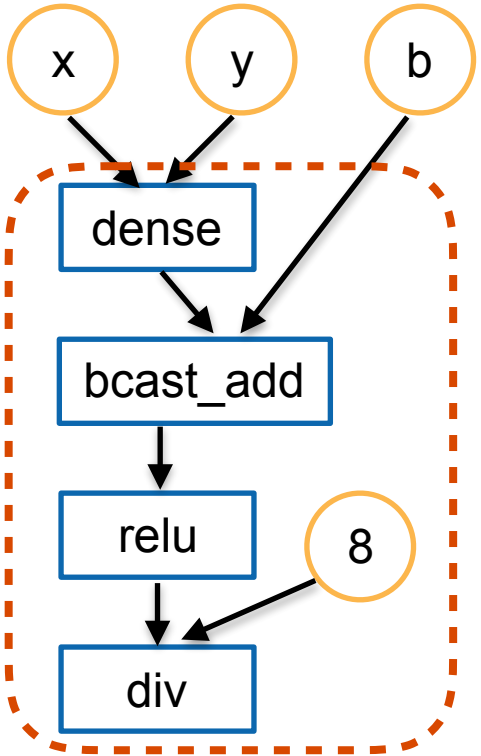
- Graph runtime
- VM
- Interpreter

aws

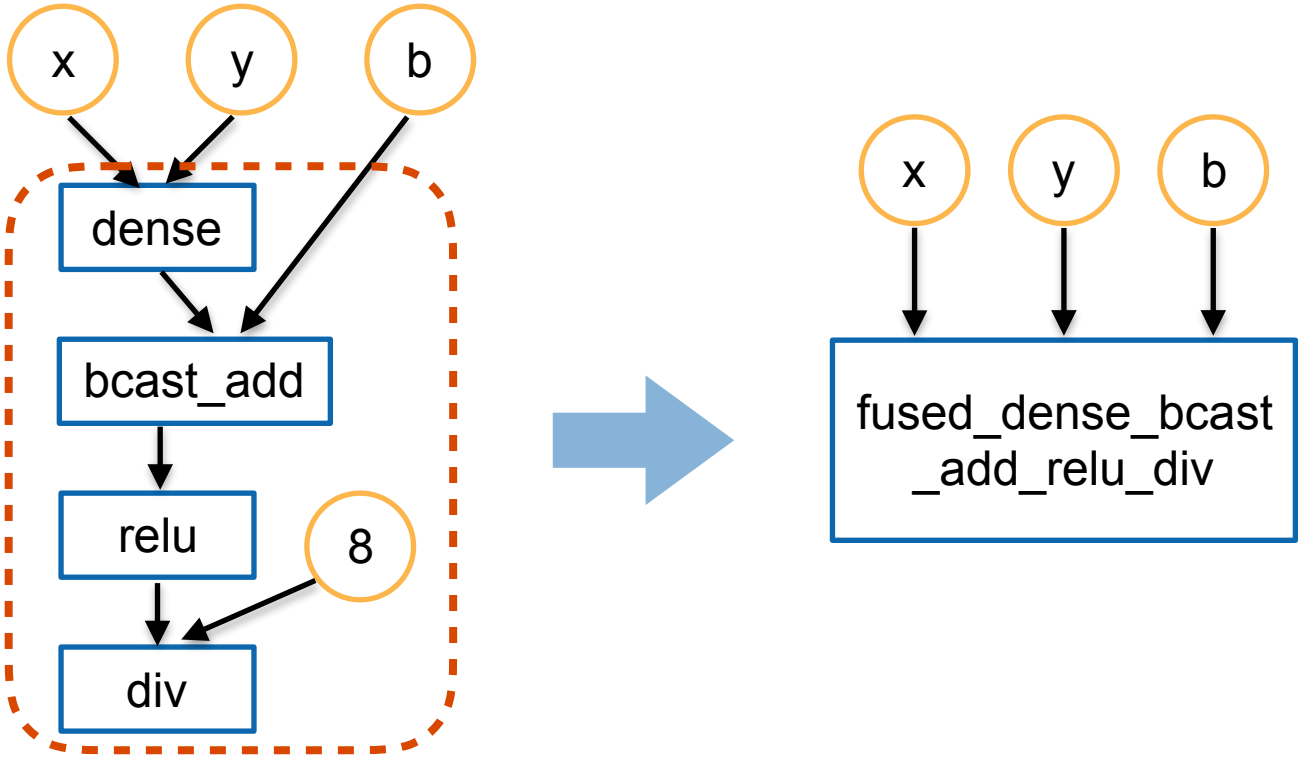
Graph level optimization



Graph level optimization



Graph level optimization

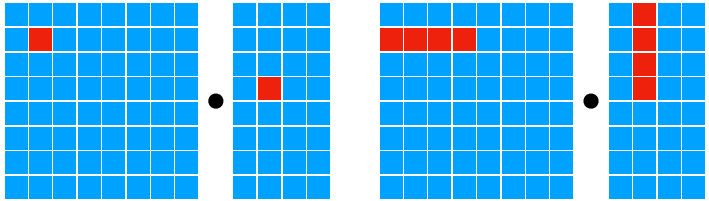


Tensor level optimization

Hardware

CPUs

Compute Primitives



scalar *vector*

Memory Subsystem

aws

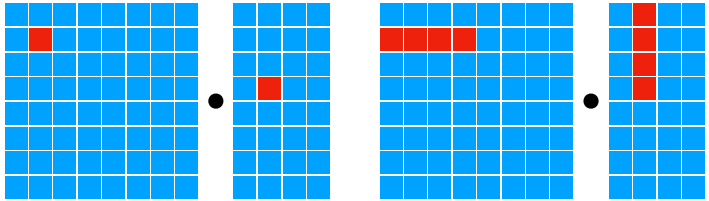
Tensor level optimization

Hardware

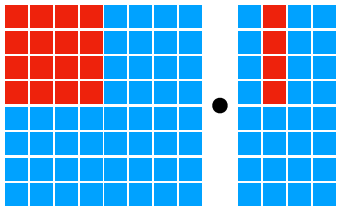
CPUs

GPUs

Compute Primitives



scalar *vector*



tensor

Memory Subsystem

aws

Matrix multiplication

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

Matrix multiplication

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

Vanilla Code

```
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]
```

Matrix multiplication

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

Vanilla Code

```
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]
```

```
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]
```

Matrix multiplication

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

Vanilla Code

```
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]
```

```
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 yo in range(64):
  for xo in range(64):
    C[yo*16:yo*16+16][xo*16:xo*16+16] = 0
    for ko in range(64):
      for yi in range(16):
        for xi in range(16):
          for ki in range(16):
            C[yo*16+yi][xo*16+xi] +=
              A[ko*16+ki][yo*16+yi] * B[ko*16+ki][xo*16+xi]
```

Matrix multiplication

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

Vanilla Code

```
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]
```

```
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 xo in range(64):
  for yo in range(64):
    C[yo*16:yo*16+16][xo*16:xo*16+16] = 0
    for ko in range(64):
      for ki in range(16):
        for yi in range(16):
          for xi in range(16):
            C[yo*16+yi][xo*16+xi] +=
              A[ko*16+ki][yo*16+yi] * B[ko*16+ki][xo*16+xi]
```

```
for yo in range(64):
  for xo in range(64):
    C[yo*16:yo*16+16][xo*16:xo*16+16] = 0
    for ko in range(64):
      for yi in range(16):
        for xi in range(16):
          for ki in range(16):
            C[yo*16+yi][xo*16+xi] +=
              A[ko*16+ki][yo*16+yi] * B[ko*16+ki][xo*16+xi]
```

Matrix multiplication

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

Billions of possible optimization choices

Vanilla Code

```
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]
```

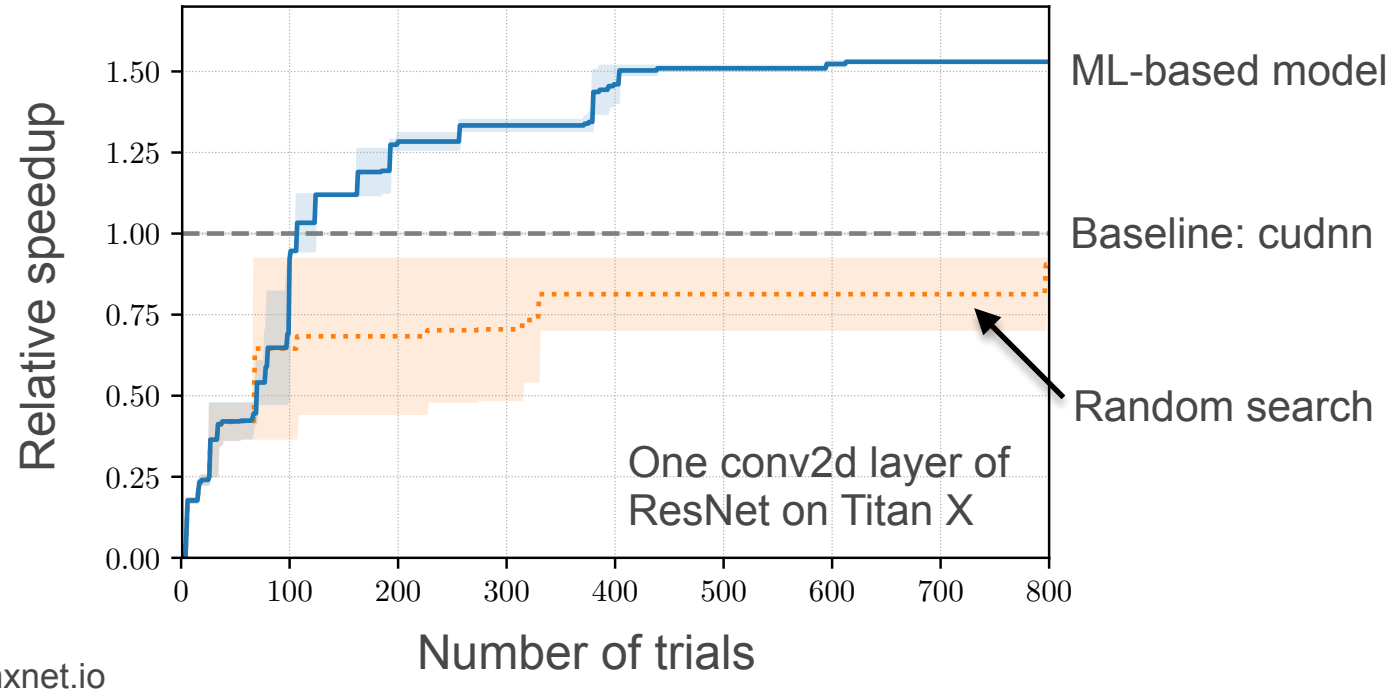
```
for yo in range(128):
  for xo in range(128):
    C[yo*8:yo*8+8][xo*8:xo*8+8] = 0
    for ko in range(128):
      for yi in range(8):
        for xi in range(8):
          for ki in range(8):
            C[yo*8+yi][xo*8+xi] +=
              A[ko*8+ki][yo*8+yi] * B[ko*8+ki][xo*8+xi]
```

```
for xo in range(64):
  for yo in range(64):
    C[yo*16:yo*16+16][xo*16:xo*16+16] = 0
    for ko in range(64):
      for ki in range(16):
        for yi in range(16):
          for xi in range(16):
            C[yo*16+yi][xo*16+xi] +=
              A[ko*16+ki][yo*16+yi] * B[ko*16+ki][xo*16+xi]
```

```
for yo in range(64):
  for xo in range(64):
    C[yo*16:yo*16+16][xo*16:xo*16+16] = 0
    for ko in range(64):
      for yi in range(16):
        for xi in range(16):
          for ki in range(16):
            C[yo*16+yi][xo*16+xi] +=
              A[ko*16+ki][yo*16+yi] * B[ko*16+ki][xo*16+xi]
```

AutoTVM: ML-based tensor optimization

- Use machine learning to learn hyper-parameter in the *schedule template*



gluon-nlp.mxnet.io

Using TVM for deployment

- Model coverage

Task	Model
Image classification	Resnet, VGG, Inception, MobileNet, DenseNet, etc.
Object detection	SSD, Yolo, Faster R-CNN
NLP	(bidirectional) LSTM/GRU/RNN, Transformer, BERT

- Device coverage
  - Intel CPU, ARM CPU, Nvidia GPU, Mali GPU, Inferentia, etc.

BERT for QA performance on ARM and Intel CPU

Max seq length	Instance	$/hour	TVM latency (ms)
128	a1.4x	0.408	251.59
256	a1.4x	0.408	536.15
128	c5.9x	1.53	59.82
256	c5.9x	1.53	88.52

gluon-nlp.mxnet.io

aws

Demo of BERT for QA

06_deployment/deployment.ipynb