Automatic Code Generation
TVM Stack

CSE 599W Spring

TVM stack is an active project by saml.cs.washington.edu and many partners in the open source community
The Gap between Framework and Hardware

Each backend to a new software stack on top of it!
Compiler’s Perspective to this Problem

- Frameworks: TensorFlow, PyTorch, mxnet, Caffe2, CNTK

Express computation

Intermediate Representation (s)

Reversible Optimizations

Code generation

Hardware: CPU, GPU, FPGA, ASIC
Computational 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
XLA: Tensorflow Compiler

- Constant shape dimension
- Data layout is specific
- Operations are low level tensor primitives
  - Map
  - Broadcast
  - Reduce
  - Convolution
  - ReduceWindow
  - ...

Source: Google
TensorRT: Rule based Fusion

Source: Nvidia
Simple Graph-based Element-wise Kernel Generator

```c
extern "C" __global__ fusion_kernel (uint32_t num_element, 
    float *x0, float *x1, float *x2, float *y) {
    int global_idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (global_idx < num_element)
        y[global_idx] = (x0[global_idx] * x1[global_idx]) + x2[global_idx];
}
```
Two min Discussion

What are pros and cons of computational graph based approach
The Remaining Gap

Frameworks

need to build and optimize operators for each hardware, variant of layout, precision, threading pattern …

Computational Graph Optimization

Hardware
Tensor Level Optimizations

Frameworks

- TensorFlow
- PyTorch
- mxnet
- Caffe2
- CNTK

Computational Graph Optimization

Tensor Expression Language

```python
C = t.compute((m, n),
    lambda i, j: t.sum(A[i, k] * B[j, k], axis=k))
```
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))

Tensor Index Expression

Compute $C = \text{dot}(A, B.T)$
Tensor Expressions are Expressive

Affine Transformation

\[
\text{out} = \text{tvm.compute((n, m), } \lambda i, j: \text{tvm.sum(data}[i, k] \times w[j, k], k))
\]

\[
\text{out} = \text{tvm.compute((n, m), } \lambda i, j: \text{out}[i, j] + \text{bias}[i])
\]

Convolution

\[
\text{out} = \text{tvm.compute((c, h, w), } \lambda i, x, y: \text{tvm.sum(data}[kc,x+kx,y+ky] \times w[i,kx,ky], [kx,ky,kc])}
\]

ReLU

\[
\text{out} = \text{tvm.compute(shape, } \lambda *i: \text{tvm.max(0, out}(\ast i))
\]
Emerging Tools Using Tensor Expression Language

Halide: Image processing language

Loopy: python based kernel generator

TACO: sparse tensor code generator

Tensor Comprehension
Schedule: Tensor Expression to Code

Tensor Expression Language

\[
C = t.\text{compute}((m, n), \\
\quad \lambda i, j: t.\text{sum}(A[i, k] \ast B[j, k], \text{axis}=k))
\]

Key Idea: Separation of Compute and Schedule introduced by Halide
Example Schedule Transformation

```python
C = tvm.compute((n,), lambda i: A[i] + B[i])
s = tvm.create_schedule(C.op)
```

```python
for (int i = 0; i < n; ++i) {
    C[i] = A[i] + B[i];
}
```
Example Schedule Transformation

```python
C = tvm.compute((n,), lambda i: A[i] + B[i])
s = tvm.create_schedule(C.op)
xo, xi = s[C].split(s[C].axis[0], factor=32)
```

```python
for (int xo = 0; xo < ceil(n / 32); ++xo) {
    for (int xi = 0; xi < 32; ++xi) {
        int i = xo * 32 + xi;
        if (i < n) {
            C[i] = A[i] + B[i];
        }
    }
}
```
Example Schedule Transformation

```python
C = tvm.compute((n,), lambda i: A[i] + B[i])
s = tvm.create_schedule(C.op)
xo, xi = s[C].split(s[C].axis[0], factor=32)
s[C].recorder(xi, xo)
```

```python
for (int xi = 0; xi < 32; ++xi) {
    for (int xo = 0; xo < ceil(n / 32); ++xo) {
        int i = xo * 32 + xi;
        if (i < n) {
            C[i] = A[i] + B[i];
        }
    }
}
```
Example Schedule Transformation

```python
C = tvm.compute((n,), lambda i: A[i] + B[i])
s = tvm.create_schedule(C.op)
xo, xi = s[C].split(s[C].axis[0], factor=32)
s[C].recorder(xi, xo)
s[C].bind(xo, tvm.thread_axis("blockIdx.x"))
s[C].bind(xi, tvm.thread_axis("threadIdx.x"))

int i = threadIdx.x * 32 + blockIdx.x;
if (i < n) {
  C[i] = A[i] + B[i];
}
```
Key Challenge: Good Space of Schedule

Should contain any knobs that produces a logically equivalent program that runs well on backend models

Must contain the common manual optimization patterns

Need to actively evolve to incorporate new techniques
Two Min Discussions

What are useful program transformation that can be used as schedule primitive
TVM Schedule Primitives

Still constantly evolving

Primitives in prior works
Halide, Loopy

New primitives for GPU
Accelerators

Tensor Expression Language
- Loop Transformations
- Thread Bindings
- Cache Locality
- Thread Cooperation
- Tensorization
- Latency Hiding

Hardware

...
Schedule Space Exploration

Tensor Expression Language

\[ C = t.\text{compute}((m, n), \lambda i, j: t.\text{sum}(A[i, k] \times B[j, k], axis=k)) \]

Schedule 1  Schedule 2  Schedule 3  ...

Kernel 1  Kernel 2  Kernel 3  ...

Make use of an AutoTuner
Extending Compute Primitives

Symbolic Loop: \( Y = \text{cumsum}(X) \)

```python
import tvm

m = tvm.var("m")
n = tvm.var("n")
X = tvm.placeholder((m, n), name="X")

s_state = tvm.placeholder((m, n))
s_init = tvm.compute((1, n), lambda _, i: X[0, i])
s_update = tvm.compute((m, n), lambda t, i: s_state[t-1, i] + X[t, i])

Y = tvm.scan(s_init, s_update, s_state, inputs=[X])
```
New Hardware Challenges

- IR
- CPU: scalar
- GPU: vector
- Accelerators: tensor
- Memory subsystem:
  - L3
  - L2 implicitly managed
  - SM
  - TX/L1 mixed
  - Unified Buffer explicitly managed
- Data type:
  - fp32
  - fp16
  - int8
Tensorization Challenge

Hardware designer:
declare tensor instruction interface

```python
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

verify that the compute matches declaration and use lowering rule

Tensorize:
transform program to use tensor instructions
Two Min Discussions

We talked a lot of tensor expression language what are the possible drawbacks about what we talked about so far
Global View of TVM Stack

Frameworks: Caffe2, CNTK, CoreML

Computational Graph

Graph Optimizations

Tensor Expression Language

Schedule Primitives Optimization

Accelerators: CUDA, ARM, Vulkan, Metal, LLVM, OpenCL, X86, AMDGPUs, Javascript/WASM
import tvm
import nnvm.frontend
import nnvm.compiler

graph, params = nnvm.frontend.from_keras(keras_resnet50)
graph, lib, params = nnvm.compiler.build(graph, target)
module = runtime.create(graph, lib, tvm.gpu(0))
module.set_input(**params)
module.run(data=data_array)
output = tvm.nd.empty(out_shape, ctx=tvm.gpu(0))
module.get_output(0, output)

Deployable Module

input

prediction tabby, tabby cat

On languages and platforms you choose
Program Your Phone with Python from Your Laptop

RPC Server on Embedded Device

Compiler Stack

```python
lib = t.build(s, [A, B],
    'llvm -target=armv7l-none-linux-gnueabihf',
    name='myfunc')
remote = t.rpc.connect(host, port)
lib.save('myfunc.o')
remote.upload('myfunc.o')
f = remote.load_module('myfunc.o')
ctx = remote.cpu(0)
a = t.nd.array(np.random.uniform(size=1024), ctx)
b = t.nd.array(np.zeros(1024), ctx)
remote_timer = f.time_evaluator('myfunc', ctx, number=10)
time_cost = remote_timer(a, b)
np.testing.assert_equal(b.asnumpy(), expected)
```

- upload module to remote
- get remote function
- copy data to remote
- get remote array handle
- run function on remote
- get profile statistics back
- copy data back to host for correctness verification
Some Fun Results

Compare TVM Stack solution to Existing solutions which relies on manually optimized libraries
End to End Performance across Hardwares

Nvidia GPUs

Time (ms)

<table>
<thead>
<tr>
<th></th>
<th>K80</th>
<th>GTX1080</th>
</tr>
</thead>
<tbody>
<tr>
<td>ResNet</td>
<td>10.0</td>
<td>7.5</td>
</tr>
<tr>
<td>MobileNet</td>
<td>12.5</td>
<td>10.0</td>
</tr>
</tbody>
</table>

Raspberry Pis

Time cost of Inference on Raspberry PI

<table>
<thead>
<tr>
<th></th>
<th>ResNet18</th>
<th>MobileNet</th>
</tr>
</thead>
<tbody>
<tr>
<td>MXNet</td>
<td>2250 ms</td>
<td>750 ms</td>
</tr>
<tr>
<td>NNVM Compiler</td>
<td>750 ms</td>
<td>650 ms</td>
</tr>
</tbody>
</table>

Credit: Leyuan Wang(AWS/UCDavis), Yuwei Hu(TuSimple), Zheng Jiang(AWS/FDU), Lianmin Zheng(SJTU)
End to End Performance on Mobile GPUs (ARM Mali)

Credit: Lianmin Zheng (SJTU)
Support New Accelerators as Well
A Lot of Open Problems

Some examples questions:

Optimize for NLP models like RNN, attention

High dimensional convolutions

Low bit and mix precision kernels

More primitive support for accelerators
Tutorials from tvmlang.org