Whole-model optimization with Apache TVM

Andrew Reusch
Software Engineer, OctoML
TVM Project Management Committee Member
The Deployment Challenge
The Deployment Challenge
Why is this a challenge?

- Fewer resources
  - Less memory
  - Slower CPU
  - Power constraints

- Traditional ML stacks are giant
Describing the Challenge

libmodel.a
The Transpiler Approach

Map each layer to a library call

<table>
<thead>
<tr>
<th>Layer</th>
<th>Library Call</th>
</tr>
</thead>
<tbody>
<tr>
<td>conv2d</td>
<td>arm_convolve_s8</td>
</tr>
<tr>
<td>relu</td>
<td>arm_relu_q7</td>
</tr>
<tr>
<td>max_pool2d</td>
<td>arm_max_pool_s8</td>
</tr>
<tr>
<td>conv2d</td>
<td>arm_convolve_q7_HWC</td>
</tr>
<tr>
<td>relu</td>
<td>arm_relu_q7</td>
</tr>
<tr>
<td>avg_pool2d</td>
<td>arm_avepool_q7</td>
</tr>
<tr>
<td>dense</td>
<td>arm_fully_connected_s8</td>
</tr>
</tbody>
</table>

Library of optimized layer implementations (e.g. CMSIS-NN)
What may microcontrollers soon look like?

- CPU0
- CPU1
- DMA
- e.g. AXI bus
- SRAM
- FLASH
- (sensors)
- NPU SRAM
- NPU
The Transpiler Approach, with NPU

🤔 which library to use?

- CMSIS-NN on CPU
- NPU (example library)

Libraries of optimized layer implementations:

- `conv2d`
  - `arm_convolve_s8`
  - `npu_convolve`

- `relu`
  - `arm_relu_q7`
  - `npu_relu`

- `max_pool2d`
  - `arm_max_pool_s8`
  - `npu_maxpool`

- `conv2d`
  - `arm_convolve_q7_HWC`
  - `npu_convolve`

- `relu`
  - `arm_relu_q7`
  - `npu_relu`

- `avg_pool2d`
  - `arm_avepool_q7`
  - `n/a`

- `dense`
  - `arm_fully_connected_s8`
  - `npu_dense`
The Transpiler Approach, with complex architecture

1. Which implementation is faster? On which layers?
2. Does my workload fit in NPU SRAM?
3. Is the memory transfer time worth it?
4. Can a second CPU speed up my workflow in parallel?
Apache TVM: Bridging the gap as a DL compiler and runtime

Open-source optimization framework for deep learning.

ML-based Optimizations

Backends for x86, nVidia/CUDA, AMD, ARM, MIPS, RISC-V, etc
TVM is an emerging industry standard

Every “Alexa” wake-up today across all devices uses a model optimized with TVM

“[TVM enabled] real-time on mobile CPUs for free...We are excited about the performance TVM achieves.” More than 85x speed-up for speech recognition model.

Bing query understanding: 112ms (Tensorflow) -> 34ms (TVM).
QnA bot: 73ms->28ms (CPU), 10.1ms->5.5ms (GPU)

“TVM is key to ML Access on Hexagon” - Jeff Gehlhaar, VP Technology

Unified ML compilation stack for CPU, GPU, NPU built with TVM
microTVM - A recap

using TVM to run models on bare-metal devices

libmodel.a
microTVM works in places without...

🚫 Operating Systems
- no files, DLLs, .so, memory mapping, kernels

🚫 Virtual Memory
- No malloc, C++ RAII, exceptions, …

🚫 Advanced Programming Languages
- No opinions on C++, Rust, Python, …
  (But we like those and you could use them!)
The microTVM Vision

>Codify and Automate Model Deployment onto any hardware platform

Including:

- Tailoring the model to fit
- Deciding how to leverage the hardware available
- Optimizing the model for application and platform
- Running, debugging, validating the model end-to-end on-device

```python
x = keras.Conv2d(2, 3, activation='relu')
keras.MaxPool2d(x)
```
Example Model Implementation with microTVM

```c
int32_t tvmgen_default___tvm_main__(int8_t* input_1_int8_buffer_var,
                                   int8_t* output_1_buffer_var,
                                   uint8_t* global_const_workspace_0_var,
                                   uint8_t* global_workspace_1_var) {

    if (tvmgen_default_fused_cast_subtract(input_1_int8_buffer_var, ...) != 0 ) return -1;
    if (tvmgen_default_fused_nn_conv2d_add_cast_multiply_clip(...) != 0 ) return -1;
    // ...
    if (tvmgen_default_fused_nn_softmax_(...,output_1_buffer_var) != 0 ) return -1;
}
```
Example Model Implementation with microTVM

```c
int32_t tvmgen_default_fused_cast_subtract(int8_t* p0, int16_t* T_subtract, ...
{
    for (int32_t i = 0; i < 1024; ++i) {
        for (int32_t j = 0; j < 3; ++j) {
            int32_t cse_var_1 = ((i * 3) + j);
            T_subtract[cse_var_1] = (((int16_t)p0[cse_var_1]) - (int16_t)-128);
        }
    }
    return 0;
}
```
Languages of the TVM Compiler

Model import

Relay Module

Optimize Operators
(AutoTVM)

TIR Module

Generate C/LLVM library

C Source Code or LLVM IR

```python
# [version = "0.0.5"]
def @main(%data : Tensor[(1, 3, 64, 64), int8],
%weight : Tensor[(8, 3, 5, 5), int8]) {
%1 = nn.conv2d(
    %data,
    %weight,
    padding=[2, 2],
    channels=8,
    kernel_size=[5, 5],
    data_layout="NCHW",
    kernel_layout="OIHW",
    out_dtype="int32");
%3 = right_shift(%1, 9);
%4 = cast(%3, dtype="int8");
%4
}
```

```c
int32_t fused_nn_contrib_conv2d_NCHWc_right_shift_cast(
    void* args, void* arg_type_ids, int32_t num_args, void* out_ret_value,
    void* out_ret_tcode, void* resource_handle) {
    void* data_pad = TVMBackendAllocWorkspace(1, dev_id, (uint64_t)13872, T, 8);
    for (int32_t i0_i1_fused_i2_fused = 0;
        i0_i1_fused_i2_fused < 68;
        ++i0_i1_fused_i2_fused) {
        for (int32_t i3 = 0; i3 < 68; ++i3) {
            for (int32_t i4 = 0; i4 < 3; ++i4) {
                ((uint8_t*)data_pad)[(((i0_i1_fused_i2_fused * 204) + (i3 * 3)) + i4)] = (((((2 <=
                    i0_i1_fused_i2_fused) && (i0_i1_fused_i2_fused < 66)) && (2 <= i3)) && (i3 < 66)) ?
                    ((uint8_t*)placeholder)[(((i0_i1_fused_i2_fused*192) + (w*3)) + (w*3)) + w]) :
                    (uint8_t)0);
            }
        }
    }
}
```
Why TIR?

- Separates implementation from definition
  - Relay is functional, TIR is procedural

- TVM can automate optimization of layers
  - AutoTVM and MetaScheduler allow TVM to automatically tile, reorder, loop-unroll, etc
  - Future: Auto-tensorization allows TVM to generate an implementation given knowledge of vector intrinsics

- Define procedural optimizations one level above codegen
Optimization in TVM - Today

Today: mostly split between two worlds

- Relay: focus on graph-level optimizations
- TIR: focus on operator-level implementation

Today: few places in the compiler with joint visibility over the two.
Example: Operator Fusion in Relay

```python
def @main(%data: Tensor[(1, 3, 224, 224), float32]) -> Tensor[(1, 1000), float32] {
    %0 = layout_transform(%data, src_layout="NCHW", dst_layout="NCHW3c");
    %1 = nn.contrib_conv2d_NCHWc(%0, ...);
    %2 = add(%1, meta[relay.Constant][1]);
    %3 = nn.relu(%2);
}
```

Relay Fusion Strategy:

1. Label operators to indicate how they can be fused
2. ...
Example: Operator Fusion in Relay

```python
def @main(%data: Tensor[(1, 3, 224, 224), float32]) -> Tensor[(1, 1000), float32] {
  %0 = layout_transform(%data, src_layout="NCHW", dst_layout="NCHW3c");
  %1 = nn.contrib_conv2d_NCHWc(%0, ...);
  %2 = add(%1, meta[relay.Constant][1]);
  %3 = nn.relu(%2);
}
```

Relay Fusion Strategy:

1. **Label operators to indicate how they can be fused**
2. **Group operators into “fusible” blocks based on rules**
3. **...**
Example: Operator Fusion in Relay

def @main(%data: Tensor[(1, 3, 224, 224), float32]) -> Tensor[(1, 1000), float32] {

    %56 = fn (%p031: Tensor[(1, 3, 224, 224), float32], Primitive=1) {
        layout_transform(%p031, src_layout="NCHW", dst_layout="NCHW3c")
    }
    %57 = %56(%data);

    %58 = fn (%p030: Tensor[(1, 1, 224, 224, 3), float32], Primitive=1) {
        %54 = nn.contrib_conv2d_NCHWc(%p030, ...);
        %55 = add(%54, meta[relay.Constant][55] );
        nn.relu(%55)
    }
    %59 = %58(%57);

    Relay Fusion Strategy:

    1. Label operators to indicate how they can be fused
    2. Group operators into “fusable” blocks based on rules
    3. Express the grouping in Relay using closures
Example: Operator Fusion in Relay

```python
def @main(%data: Tensor[(1, 3, 224, 224), float32]) -> Tensor[(1, 1000), float32] {
    %56 = fn (%p031: Tensor[(1, 3, 224, 224), float32], Primitive=1) {
        layout_transform(%p031, src_layout="NCHW", dst_layout="NCHW3c")
    };%57 = %56(%data);

    %58 = fn (%p030: Tensor[(1, 1, 224, 224, 3), float32], Primitive=1) {
        %54 = nn.contrib_conv2d_NCHWc(%p030, ...);
        %55 = add(%54, meta[relay.Constant][55] );
        nn.relu(%55)
    };%59 = %58(%57);
}
```

Relay Fusion Strategy:

1. Label operators to indicate how they can be fused
2. Group operators into “fusable” blocks based on rules
3. Express the grouping in Relay using closures
4. Send to scheduling to implement the fused blocks

🤔 What is missing here?
Example: Operator Fusion in Relay

```python
def @main(%data: Tensor[(1, 3, 224, 224), float32]) -> Tensor[(1, 1000), float32] {
    %56 = fn (%p031: Tensor[(1, 3, 224, 224), float32], Primitive=1) {
        layout_transform(%p031, src_layout="NCHW", dst_layout="NCHW3c")
    };
    %57 = %56(%data);

    %58 = fn (%p030: Tensor[(1, 1, 224, 224, 3), float32], Primitive=1) {
        %54 = nn.contrib_conv2d_NCHWc(%p030, ...);
        %55 = add(%54, meta[relay.Constant][55] );
        nn.relu(%55)
    };
    %59 = %58(%57);
```

🤔 What if we want to change how we fuse based on...
- the loop structure of the implementation?
- the way Tensor types are represented in memory?
- the whole program?
Introducing Relax

- A new, Relay-inspired Graph-level IR

- Developed in open source alongside TVM

- Language improvements around:
  - Dynamic shape support
  - Blending third-party libraries
  - Iterative, interactive compilation

- Written by my colleagues and others connected to TVM’s community
Coexistence in Relax

```python
@R.function
def main(x: Tensor((10, 20), "float32"), p0: Tensor((), "float32")):
    lv = R.call_tir(add, (x, p0), (10, 20), dtype="float32")
    lv1 = R.call_tir(exp, (lv,), (10, 20), dtype="float32")
    gv = R.call_tir(squeeze, (lv1,), (10, 20), dtype="float32")
    return gv

@add = primfn(v1: handle, v2: handle, v3: handle) -> ()
for (i0: int32, 0, 10) {
    for (i1: int32, 0, 20) {
        v3[ax0, ax1] = (v1[ax0, ax1] + v2[ax0, ax1])
    }
}
```

Graph representation

Implemented operator
Operator Fusion in Relax

```python
@R.function
def main(x: Tensor((10, 20), "float32"), p0: Tensor((,), "float32")):  
  lv = R.call_tir(add, (x, p0), (10, 20), dtype="float32")
  lv1 = R.call_tir(exp, (lv,), (10, 20), dtype="float32")
  gv = R.call_tir(squeeze, (lv1,), (10, 20), dtype="float32")
  return gv

@add = primfn(v1: handle, v2: handle, v3: handle) -> ()
  for (i0: int32, 0, 10) {
    for (i1: int32, 0, 20) {
      v3[ax0, ax1] = (v1[ax0, ax1] + v2[ax0, ax1])
    }
  }

// @exp = ..., @squeeze = ...
```

Relax Fusion Strategy:

1. Decide which layers to fuse
2. ...
Operator Fusion in Relax

```python
@R.function
def main(x: Tensor((10, 20), "float32"), p0: Tensor(), "float32"):
    lv1 = fused_add_exp(x, p0)
    gv = R.call_tir(squeeze, (lv1,), (10, 20), dtype="float32")
    return gv

@R.function
def fused_add_exp():
    lv = R.call_tir(add, (x, p0), (10, 20), dtype="float32")
    lv1 = R.call_tir(exp, (lv,), (10, 20), dtype="float32")
    return lv1

// @add = primfn(...) ...
```

Relax Fusion Strategy:

1. Decide which layers to fuse
2. Extract those layers into a function
3. ...
Operator Fusion in Relax

```python
@R.function
def main(x: Tensor((10, 20), "float32"), p0: Tensor((,), "float32")):
    lv1 = R.call_tir(fused_add_exp, (x, p0), (10, 20), dtype="float32")
    gv = R.call_tir(squeeze, (lv1,), (10, 20), dtype="float32")
    return gv

@fused_add_exp = primfn(x: handle, p0: handle, out: handle) {
    T_add = alloc_buffer(float32[10, 20])
    for (i0: int32, 0, 10) {
        for (i1: int32, 0, 20) {
            T_add[ax0, ax1] = (x[ax0, ax1] + p0[])
            out[i0_2, i1_2] = @tir.exp(T_add[i0_2, i1_2], dtype=float32)
        }
    }
}
```

Relax Fusion Strategy:

1. Decide which layers to fuse
2. Extract those layers into a function
3. Fuse the two implementations together
What changed?

- We have a lot more information available to help inform our decision
- The decision of “what to fuse” can now be a user-facing decision
- We can still use the pattern-based method from before, if needed
- But we can also explore some new possibilities...
What can we now do?

- **Memory-based fusions**
  - If debugging, don’t fuse if there is memory to spare

- **Fuse complex operations together (i.e. stripe two convolutions)**

- **On microTVM: tensorize requantization operators**
  - i.e. multiply/shift with SIMD instructions
Other features of Relax

- Dynamic or symbolic shapes
  
  foo: Tensor[(m, 224, 224, 3)]
  bar: Tensor[(m * 224 * 224 * 3, )] = flatten(foo)

- Natural calls to third-party libraries
  
  foo = R.call_packed("my_custom_op", bar)

- Natural python interface
  
  @R.function
  def my_model(foo : Tensor[(1, 2)) -> int32:
    return R.call_tir("conv2d", (foo, [1]), ...)
Status of Relax

- A proposed addition to TVM’s mainline codebase
- Initially prototyped alongside TVM by members of the TVM community
- Still early in development, but seeking feedback from early adopters
Upcoming work: Relax + AoT

- Right now, Relax requires a runtime VM
  - Limits use in a tinyML setting

- In development: Relax + Ahead-of-Time compilation

- We expect Relax techniques to broadly apply when optimizing for heterogenous architectures
Getting Involved

- microTVM and Relax are community-driven efforts
  - we welcome new contributions!

- Check out Relax (incubating at https://github.com/tlc-pack/relax)

- We welcome new ideas and questions on our Discuss forum and Discord, and new contributors to join our weekly TVM Community Meeting
  
  https://tvm.apache.org/community
By the way, we’re hiring!

OctoML started in July 2019 - $130M in venture capital, now 100+ people and growing!

We’re hiring Software Engineers, Product Managers, Researchers, and more!

https://octoml.ai/careers

Or email me: areusch@octoml.ai
Thank you!

- Much of the work summarized today, in particular Relax, was authored and contributed by members of the TVM Community.

https://tvm.apache.org/community
Copyright Notice

This presentation in this publication was presented at the tinyML® EMEA Innovation Forum 2022. The content reflects the opinion of the author(s) and their respective companies. The inclusion of presentations in this publication does not constitute an endorsement by tinyML Foundation or the sponsors.

There is no copyright protection claimed by this publication. However, each presentation is the work of the authors and their respective companies and may contain copyrighted material. As such, it is strongly encouraged that any use reflect proper acknowledgement to the appropriate source. Any questions regarding the use of any materials presented should be directed to the author(s) or their companies.

tinyML is a registered trademark of the tinyML Foundation.