LIVING LIFE ON THE LOW-POWER EDGE

TINY MODELS ON TINY DEVICES

A thesis submitted by

Logan Weber

in partial fulfillment of the requirements for the degree of
Master of Science.

Thesis approved by _____
Date: _____

Paul G. Allen School of Computer Science & Engineering,
University of Washington

Finish it until it's done.

JARED ROESCH

If you're gonna be dumb,
you gotta be tough.

ZACH TATLOCK

Abstract

Insights from researchers working on quantization and alternative numeric formats have challenged longstanding assumptions of the computational horsepower required to run artificial neural networks. With the theoretical bar to entry lowered, the practical bar to entry remains high, as nearly all machine learning (ML) frameworks were built upon the assumption of access to an operating system and computational resources aplenty. To support edge devices, ML frameworks have resorted to engineering-intensive solutions that shoehorn these devices into the existing system in a one-off fashion. Furthermore, there exist **no** frameworks that offer autotuning of operator performance for edge devices, a practical necessity to cope with the everchanging hardware landscape.

We present μTVM, an extension of the TVM compiler stack to target bare-metal devices. μTVM facilitates model execution and automatic optimization with minimal requirements of target devices. As one of our chief design goals, using μTVM feels deceptively similar to using standard TVM, while performance remains competitive with libraries hand-optimized by experts.

Acknowledgements

Of all the brilliant minds that have graced my presence in my time at the University of Washington, there are a select few without whom none of the work in this thesis would have been possible.

Jared Roesch, my mentor on one of my first research projects and my friend thereafter. I am thankful for all of the meetings he started[1] in the doorway to the UW PLSE lab (much to its occupants' annoyance), for his ferociously gung-ho attitude, and for his steady stream of hot takes.

The Sad Plus[2]. What began as a group of fellow disciples of Jared grew into a friend group in its own right. Whether we were *on* a deadline or *off* a deadline, we shared long nights in the PLSE lab listening to Marisa's **only** Spotify playlist for the **1000th** time, and I will always treasure the inspiring atmosphere they fostered.

Zachary Tatlock, my faculty advisor. Zach showed me the ropes of research and taught me by example what it takes to become a force of nature in academia. In every research meeting, Zach brought piercing insights and quelled our tendency towards analysis paralysis.

Tianqi Chen, my mentor for the topic of this thesis: µTVM. I am grateful for his unwavering resolve to make things work, for his ability to constantly humble me, and for guiding the design of µTVM with an expert's eye.

Pratyush Patel, my collaborator on µTVM in the early days. Pratyush was the original driver of µTVM. He dug through the literature, developed a compelling case for why the world needs µTVM, and framed the problem so clearly that I, without any background in microcontrollers, was able to design and implement this infrastructure.

Luis Vega, an always-friendly face to turn to in the SAMPL group. While struggling to bridge the gap between running µTVM on a simulator and running on a real device, Luis offered to show me the fundamentals of working with microcontrollers. I knew nothing about hardware[3], and he was incredibly patient in teaching me the basics. These interactions were brief, but immeasurably helpful.

[1] and finished, of course

[2] with members Josh Pollock, Altan Haan, Marisa Kirisame, and yours truly

[3] and still know nothing about it, for that matter!

Introduction

It was long believed that deep learning models required high-precision (at least 32-bit) IEEE-754 floating point values to achieve decent accuracy, as the theory of deep learning is based on the infinite-precision real numbers. However, work on quantization and alternative numeric representations have invalidated this assumption and shown us there is a sense in which the theory of neural networks is not chained to the reals, but rather some more abstract notion of numeric systems [17, 4, 9, 8, 10]. While this work improved performance and memory usage on desktops, it simultaneously lowered the bar to entry for devices broadly.

In response to these developments, low-cost, AI-powered consumer devices have sprung up, leading to widespread interest in "bare-metal" devices among ML researchers and practitioners [6, 11, 7]. Bare-metal devices are even more stripped-down of computational amenities than embedded devices (such as Raspberry Pi's), because they cannot even run general-purpose operating systems.

While it is already possible for experts to run *some* models on *some* bare-metal devices, optimizing models for diverse sets of devices is challenging, often requiring manually optimized device-specific libraries. And for those platforms without, say, Linux support, there exists no scalable solution for deploying models. Because of this, in order to target new devices, developers must implement one-off custom software stacks for managing system resources and scheduling model execution.

The manual optimization of machine learning software is not unique to the domain of bare-metal devices. In fact, this has been a common theme for developers working with other hardware backends (e.g., GPUs and FPGAs). TVM, an end-to-end deep learning compiler, has proven resilient to the onslaught of new hardware targets, but until now, it couldn't grapple with the unique profile of microcontrollers [2]. To solve the problem in this domain, we've extended TVM to feature a microcontroller backend, called µTVM [4]. µTVM facilitates host-driven execution of tensor programs on bare-metal devices and enables automatic optimization of these programs via AutoTVM [3], TVM's built-in tensor program optimizer.

µTVM consists of three core contributions:

- Low-level interfaces for interaction with bare-metal devices in TVM.
- An implementation of these low-level interfaces for JTAG-compatible bare-metal devices.
- A driver stack, which utilizes these interfaces to support memory management, library linking, model execution, and automatic tensor program optimization.

[4] pronounced "MicroTVM"

It is worth noting that by targeting bare-metal devices, we have accidentally included support for driving hardware accelerators, as they are often subject to the same constraints. Thus, µTVM *also* functions as a tool for faster iteration on hardware designs, since a hardware designer is unlikely to develop an entire operating system for their accelerator, but they *are* quite likely to adapt a compiler toolchain and implement a device communication protocol.

Background

To understand where µTVM fits in the landscape of work on machine learning systems, it helps to understand what bare-metal devices are, the conditions in which deep learning frameworks and operator compilers evolved, and how their evolution led to their shortcomings in serving bare-metal devices.

Bare-Metal Devices

We classify bare-metal devices as devices without a sophisticated OS or software stack. These primarily include embedded boards, such as certain RISC-V boards, Arduinos, soft-cores running on FPGAs, etc. There is an implicit assumption that bare-metal targets can run basic, cross-compiled C programs, although standard library support is not mandatory. The key challenges in programming these devices lies in their highly-constrained amount of memory (usually on the order of a few hundred kilobytes) and low processing power (usually one or two wimpy cores). However, they are useful in cyber-physical domains due to their small size, easy interfacing with commercial sensors, and low energy consumption.

Deep Learning Frameworks

Existing deep learning frameworks provide expressive, high-level interfaces (usually in Python) to execute models. But these typically rely on software stacks that are not supported on bare-metal devices. For example, while TVM compiles code into LLVM IR for CPU backends, there is no publicly available Arduino LLVM, and hence the same code cannot be reused for these devices. Furthermore, space restrictions on bare-metal devices often preclude the possibility of access to dynamic memory allocation (e.g., `malloc`).

Recent deep learning frameworks like TensorFlow Lite (TF Lite) and compiler stacks like TVM **do** provide a solution to program embedded boards such as the Raspberry Pi. However, Raspberry Pi's are relatively powerful computers, given that they run Linux and have gigabytes of memory. Consequently, the above frameworks still depend on some components of the OS stack. For instance, AutoTVM depends on a remote procedure call (RPC) server running on the Raspberry Pi. On the other hand, bare-metal devices do not have such sophisticated software support. Although setting up a minimal RPC server may be feasible, it would require effort to build it from scratch for each bare-metal device, in

addition to consuming the already meager on-board resources. As a result, any effective solution must be customized to target these platforms, while also prioritizing portability.

Researchers have made headway in supporting novel architectures with TF Lite, such as RISC-V devices [15], but the approaches thus far have involved modifying the framework's source code to include hand-optimized operator implementations.

There has been a recent effort by the TensorFlow developers to support microcontrollers, called TensorFlow Lite Micro. TFLite Micro relies on manually optimized libraries, whereas μTVM can directly make use of AutoTVM to automatically generate and optimize programs on bare-metal devices. Furthermore, their approach does not enable unification of microcontroller interaction with regular device interaction, as TensorFlow Lite Micro is a fork of TensorFlow Lite which is already a fork of TensorFlow.

Low-Level Tensor Compilers

Low-level tensor compilers (or operator compilers) are focused on the production of high-performance operators which implement compute-intensive operations such as matrix multiplication or convolution. Operator compilers perform code generation for sets of scalar loop nests, but only represent a restricted subset of a whole program, ignoring details such as memory allocation/management, data structures, closures, and arbitrary control flow. The most notable designs are either inspired by the compute-schedule split introduced by Halide [13] or the polyhedral framework, as used by Tensor Comprehensions [18] and Diesel [5].

TVM

TVM began as an operator compiler based on the compute-schedule split paradigm, but has since grown into an end-to-end optimizing compiler for deep learning that bridges the gap between productivity-focused DL frameworks and efficiency-oriented hardware backends. It has done so by introducing a high-level differentiable intermediate representation, called Relay, atop its low-level tensor expression IR [14], along with runtime targets for Relay to compile to [16]. Relay expresses entire models (static **or** dynamic) without requiring a host language, allowing previously disparate mechanisms in ML frameworks (shape inference, operator fusion, backpropagation) to be written as mere compiler passes. The entire TVM stack can be succinctly visualized by Figure 1.

AutoTVM

Since its release in 2018, AutoTVM has become a killer feature of TVM. It works by making use of simulated annealing and machine learning to optimize tensor programs. A full explanation of the algorithm AutoTVM uses is beyond the scope of this thesis, so for

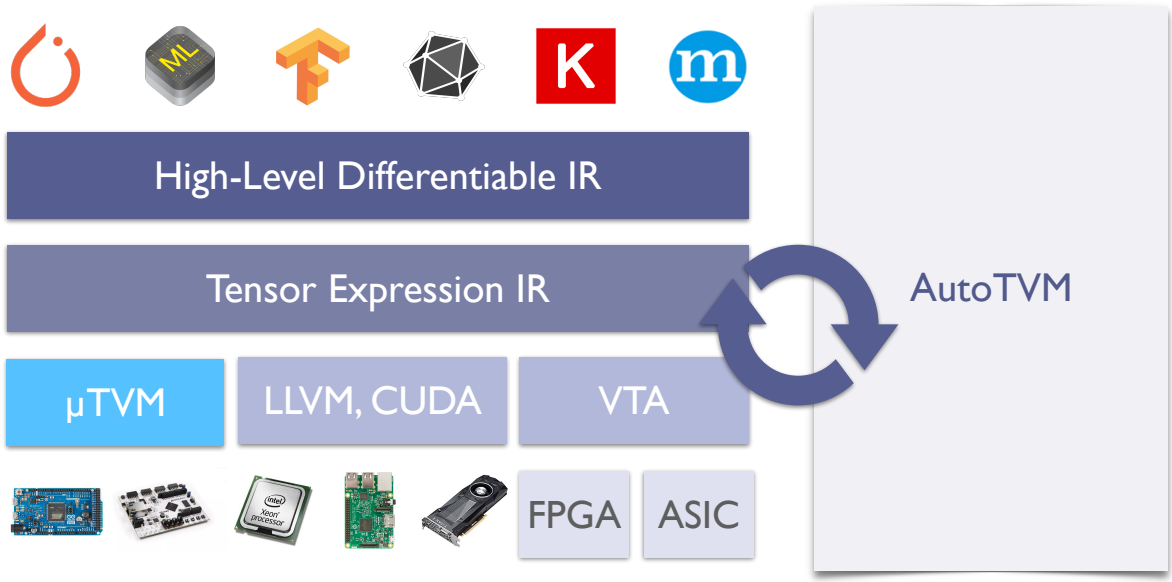

Figure 1: A high-level depiction of the TVM stack, and where µTVM fits

our purposes, we treat it as a black-box optimizer and only describe its *interaction* with the µTVM runtime.

Following the flow of Figure 2, an autotuning loop begins with some operator (e.g., convolution) for which we would like to generate an optimized implementation. To start, AutoTVM provides some initial implementation which makes no promise to be performant. This implementation is fed through the µTVM runtime and loaded onto the device. Then, some sample inputs are generated and the program is run and timed with those inputs. This timing data is then sent to AutoTVM, which it uses to score the implementation. AutoTVM then generates a new (often better) implementation and repeats the timing/scoring process. This loop is repeated until the performance meets some user-defined threshold.

Design

The design of µTVM was ultimately motivated by the question,

> **Can we make *bare-metal* device interaction feel like *regular* device interaction?**

However, we did not wish to fall prey to the same temptations encountered by others in previous approaches—namely, of constructing a bespoke vertical for a *particular* microcontroller. So we constrained our design with another question:

> **What can we do with features common to all microcontrollers?**

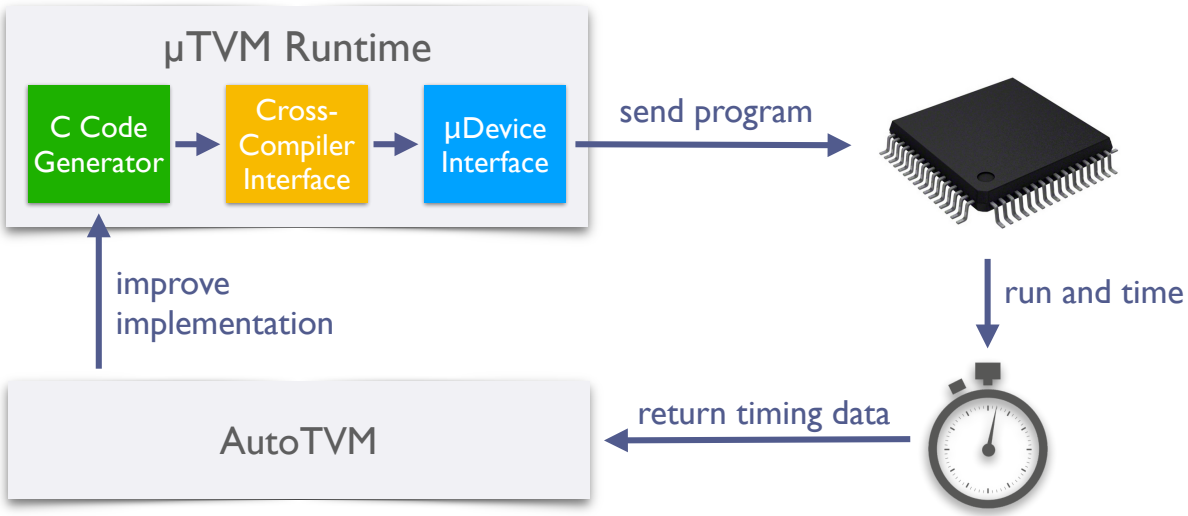

Figure 2: The AutoTVM optimization loop, when the target device is a microcontroller

Fueled by these design considerations, we have built a runtime infrastructure that operates under the smallest reasonable set of requirements that can be expected of a device. In particular, users need only provide:

1. A C cross-compiler toolchain for their device
2. A method for reading/writing to device memory and executing code on the device
3. A specification containing the device's memory layout and general architectural characteristics
4. A code snippet that prepares the device for function execution

Most bare-metal devices have support for C and JTAG (a debugging protocol), so (1) and (2) usually come for free! Furthermore, (3) and (4) are often very small asks (examples in Figure 3).

Once these requirements are met for a device, the delta from CPU-targeted TVM and microcontroller-targeted TVM is miniscule, as depicted in Figure 4.

Device Sessions

Given the networked nature of microcontroller interaction, we slightly deviate from standard TVM code by introducing the concept of `MicroSessions`.

Every piece of functionality in µTVM relies on having an open session with the target device. This requirement forces our hand in creating a syntactic distinction from normal TVM code—namely, this one:

```
...
with micro.Session(DEV_CONFIG):
    ...
```

```
device_config = {
    # Unique identifier for the device
    'device_id': 'arm.stm32f746xx',
    # Prefix of each binary (e.g., arm-none-eabi-gcc)
    # in the cross-compilation toolchain
    'toolchain_prefix': 'arm-none-eabi-',
    # First address of RAM
    'base_addr': 0x20000000,
    # Dictionary of desired ELF section sizes (in bytes)
    'section_sizes': {
        'text': 18000,
        'rodata': 100,
        'data': 100,
        ...
    },
    # Device word size
    'word_size': 4,
    # Whether to use Arm's thumb ISA
    'thumb_mode': True,
    # Method of communication with the device
    'comms_method': 'openocd',
    # OpenOCD server address (if 'comms_method' is 'openocd')
    'server_addr': '127.0.0.1',
    # OpenOCD server port (if 'comms_method' is 'openocd')
    'server_port': 6666,
}
```

```
.syntax unified
.cpu cortex-m7
.fpu softvfp
.thumb

.section .text.UTVMInit
.type UTVMInit, %function
UTVMInit:
  # Enable FPU.
  ldr r0, =0xE000ED88
  ldr r1, [r0]
  ldr r2, =0xF00000
  orr r1, r2
  str r1, [r0]
  dsb
  isb
  # Set stack pointer.
  ldr sp, =_utvm_stack_pointer_init
  # Jump to main function of uTVM
  # device runtime.
  bl UTVMMain
.size UTVMInit, .-UTVMInit
```

Figure 3: **(Left)** Example of a device configuration dictionary for an Arm STM32F746-series microcontroller. **(Right)** Example of a device initialization code snippet for an Arm STM32F746-series microcontroller. The symbol `_utvm_stack_pointer_init` is exported by µTVM while linking this code snippet with the device runtime; it points to the end of the stack section in device memory.

```python
# Grab ResNet-18 expressed in Relay.
resnet = get_resnet()

# Compile from Relay into an operator
# graph description, operator source,
# and model parameters.
graph, c_mod, params = relay.build(
  resnet, target='llvm', params=weights)

# Create graph runtime from operator
# graph, operator module, and device
# context.
graph_mod = graph_runtime.create(
  graph, micro_mod, tvm.cpu(0))
# Set model weights.
mod.set_input(**params)
# Execute with `image` as the input.
mod.run(data=image)
# Show the prediction.
print(get_prediction(mod))
```

```python
# Generate a configuration for
# the microcontroller.
DEV_CONF = stm32f746xx.generate_config(
  '127.0.0.1', 6666)
# Grab ResNet-18 expressed in Relay.
resnet = get_resnet()
# Begin a uTVM session with the device.
with micro.Session(DEV_CONF):
  # Compile from Relay into an operator
  # graph description, operator source,
  # and model parameters.
  graph, c_mod, params = relay.build(
    resnet, target='c', params=weights)
  # Convert generated module into
  # microcontroller-compatible module
  # and load onto the device.
  micro_mod = create_micro_mod(
    c_mod, DEV_CONF)
  # Create graph runtime from operator
  # graph, operator module, and device
  # context.
  graph_mod = graph_runtime.create(
    graph, micro_mod, tvm.micro_dev(0))
  # Set model weights.
  mod.set_input(**params)
  # Execute with `image` as the input.
  mod.run(data=image)
  # Show the prediction.
  print(get_prediction(mod))
```

Figure 4: A side-by-side comparison of running ResNet-18 in TVM on a CPU (left) and on an Arm STM32F746-series microcontroller (right). The key differences are **(1)** we need to create a device configuration that tells **µ**TVM which OpenOCD server to connect to, **(2)** we need to create a session with the device using this configuration, and **(3)** we need to convert the raw C module generated by Relay into a **µ**TVM-compatible module (via `create_micro_mod`).

Every line inside this `with` block can call standard TVM functions, but when a µTVM device context is used (e.g., `tvm.micro_dev(0)`), standard TVM functions, such as allocating a tensor or executing a model, now run through the µTVM infrastructure.

When such a `with` block is entered, it initializes a connection with the device, using whichever communication method was specified in `DEV_CONFIG` (usually OpenOCD). The µTVM device runtime is then cross-compiled, using whichever cross-compiler was specified in `DEV_CONFIG`. Finally, space for the compiled binary is allocated by the host, and the binary is loaded onto the device using the opened connection. Once the runtime is situated on the device, loading tensors and modules becomes possible.

Module Creation

One of the core abstractions in TVM is that of a module. A module stores a set of related functions for a particular device/runtime target. Given that microcontrollers don't normally have operating systems, µTVM needs to do a lot of extra work to maintain this high-level abstraction. To see what's going on, we'll trace through the process of creating and loading a µTVM-compatible module.

Suppose we have a `micro.Session` open with our device and a TVM schedule that implements 2D convolution. If we want to load it onto our microcontroller, we need it to emit C code. To do so, we just need to set the build target as C, which routes the build process through TVM's C code generation backend (see Figure 5).

```
# First, we construct a Relay function
# that computes `x + y`.
func = relay.fromtext("""
v0.0.4
fn (%x: Tensor[(1024,), float32],
    %y: Tensor[(1024,), float32]) {
  add(%x, %y)
}
""")
# Disable vectorization, since uTVM does not
# currently support it natively.
with tvm.build_config(disable_vectorize=True):
  # Set `target` to 'c', and the build
  # is now routed to C code generation.
  _, src_mod, _ = relay.build(
    func, target='c', params={})
# Now, we can see the generated source.
print(src_mod.get_source())
```

```
int32_t add(
    void* args,
    void* arg_type_ids,
    int32_t num_args) {
  void* A_val = ((TVMValue*) args)[0].v_handle;
  void* B_val = ((TVMValue*) args)[1].v_handle;
  void* C_val = ((TVMValue*) args)[2].v_handle;
  TVMArray A_arr = ((TVMArray*) A_val)[0];
  TVMArray B_arr = ((TVMArray*) B_val)[0];
  TVMArray C_arr = ((TVMArray*) C_val)[0];
  float* A = (float*) A_arr.data;
  float* B = (float*) B_arr.data;
  float* C = (float*) C_arr.data;
  for (int32_t i = 0; i < 1024; ++i) {
    C[i] = A[i] + B[i];
  }
  return 0;
}
```

Figure 5: **(Left)** TVM code to produce C source code from a Relay function **(Right)** C source code resulting from running the code on the left (modified to be human-readable)

Once we've generated C source code and stored it in a TVM module, that module still resides on the *host* machine. In order to load it onto the *device*, we run it through one of the core functions in the µTVM infrastructure: `create_micro_mod` (see Figure 4 for example usage).

This function cross-compiles the C source within the module, allocates room for the resulting binary (so it can coexist with the runtime or other modules resident in device memory), then sends each section of the binary to its allocated slot on the device. Once the module binary is snug in device memory, function pointers within the binary are patched to give the module access to helper functions in the device runtime (e.g., for allocating scratchpads)—a primitive form of dynamic linking. Additionally, symbol mappings from the module binary are stored to allow obtaining remote module handles by name (e.g., `conv_func = micro_mod['conv2d']`). When a named access is attempted, the symbol table is searched for the name, and if found, a wrapper around the symbol's address is returned that will call into µTVM's function execution mechanisms when given arguments (e.g., `conv_func(activations, weights, output)`).

Tensor Creation

Before we can call an on-device operator, we first need to allocate both input ***and*** output tensors (functions are generated in destination-passing style). Within a `micro.Session` block, on-device tensors can be created with the snippet below:

```
data_np = ...  # Generate data as NumPy array.
data = tvm.nd.array(data_np, ctx=tvm.micro_dev(0))
```

Based on its data type (e.g., `int8`, `float32`, etc.) and shape, the tensor's size in bytes is calculated, and the host allocates a region of memory of that size in the device's heap. The backing NumPy data is then loaded into the allocated region, and the user is given a remote handle to the tensor.

Function Calls

Operator execution is perhaps the trickiest part of this system. To simplify its presentation, we'll first cover strict execution (where operators are executed as soon as they're called), then lazy execution (where operators are only executed once their results are needed)—the latter is how the system actually works.

Strict Execution

Since functions are called in destination-passing style (e.g., `conv_func(activations, weights, output)`), all required tensors are already allocated on the device, so we only need to send *metadata* to the device (device address, shape, and data type), to tell it which of its resident tensors to use. The runtime representation of a function call (Figure 6) includes this metadata, as well as the address of the function being called. Before constructing this representation, the metadata needs to be serialized into the arguments section on the device that exists expressly for this purpose.

```
/*
 * Task Structure for uTVM
 */
typedef struct {
  /* Pointer to function to call for this task */
  int32_t (*func)(void*, void*, int32_t);
  /* Array of argument tensors */
  TVMValue* arg_values;
  /* Array of datatype codes for each argument */
  int* arg_type_codes;
  /* Number of arguments */
  int32_t num_args;
} UTVMTask;
```

Figure 6: Struct declaration for µTVM function call metadata

In the strict setting, there is a single global `UTVMTask` instance that we, from the host side, write into. Once we have written to the task, the runtime has everything it needs to execute the function, and we can begin execution at the runtime's entry point. The runtime will perform some lightweight initialization, run our operator, then return control to the host.

Lazy Execution

In practice, executing operators as soon as the user requests to becomes prohibitively expensive, as communication overhead begins to dominate. We can improve the throughput of our system by delaying evaluation until the user needs the results of the call (triggered by copying a tensor from the device or calling `Sync` on the TVM side).

The device-side runtime needs to be modified to accomodate this mechanism by storing an array of `UTVMTasks` and looping over these tasks during execution, instead of storing and executing a single `UTVMTask`.

Thus, with lazy execution, `UTVMTasks` are enqueued on the host side as the user or runtime calls functions. Then when execution is triggered by the conditions mentioned above, these tasks are written to the device, execution begins, and the device-side runtime handles the rest.

AutoTVM with MicroTVM

So far, the runtime we've described doesn't seem very useful for *model deployment*, since it relies so heavily on a host machine. This is intentional, and this first version of the runtime has in fact been designed with a different goal in mind: **AutoTVM support**.

In general, AutoTVM proposes candidate kernels, runs them on the target backend with random inputs, then uses the timing results to improve its search process. Given that

AutoTVM only cares about single operator executions, we have designed the runtime to be operator-oriented, as opposed to being model-oriented. In the case of µTVM though, communication with the device will usually dominate the execution time. Lazy execution is an important feature, because it allows us to run the same operator many times without returning control to the host, so the communication cost is amortized over each run, and we get a better idea of the performance profile.

Because AutoTVM requires rapid iteration on large numbers of candidate kernels, µTVM infrastructure only makes use of RAM currently. However, for a self-hosted runtime (Section), we will surely need to make use of both flash memory and RAM.

The Hosted Graph Runtime

Although the hosted runtime was designed for AutoTVM, we can still run full models (as long as they don't have any control flow). This functionality comes for free just by using TVM's graph runtime, but with a µTVM context.

In this mode of execution, the model is expressed as a graph, where each node is an operator, and each directed edge represents the direction in which a tensor flows. In order to execute this graph, inputs are given and the execution order of operators is defined as a topological sort of the graph.

When used with a µTVM context, the graph runtime loads all operators in the model graph onto the device, and the host drives the overarching control flow of the execution. An added benefit of lazy execution with the graph runtime is that it allows for entire models to be run on the device without returning control flow to the host until the very end (see Figure 7). In fact, the only reliance on the host with the graph runtime is for tensor allocation and operator scheduling (which is just a topological sort of the dependence graph).

Note that entire tensors are not being transmitted between every operator execution. Instead, tensors remain resident in device memory for as long as they are alive in the graph, and it is actually *metadata* that is being transmitted. This metadata simply describes which tensors to use for a particular operator invocation.

Implementation

There are four key components in the µTVM runtime that allow us to plug into TVM:

- `LowLevelDeviceAPI`: Provides read, write, and execute capabilities for a device.
- `MicroSession`: Provides functionality to load compiled libraries onto the device and to execute functions (using a `LowLevelDeviceAPI` backend).
- `MicroDeviceAPI`: Performs on-device memory management and provides helper functions to copy memory to and from the device for TVM objects (using a `MicroSession`

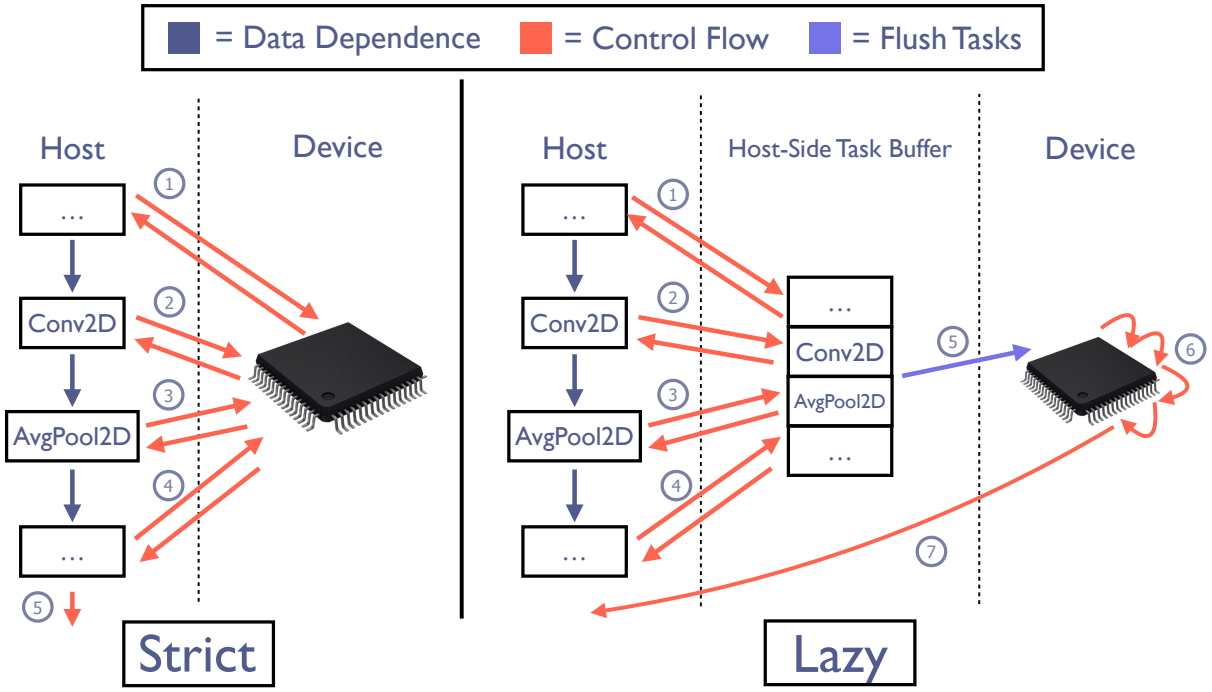

Figure 7: Comparison of device communication while using the graph runtime with strict µTVM task execution vs. lazy task execution. The ordering of events is shown via numbering.

instance).

- `MicroModule`: Provides functionality to load compiled libraries onto the device and to execute functions (using a `MicroSession` instance).

The point of providing `MicroDeviceAPI` and `MicroModule` implementations is primarily to fit neatly into TVM's APIs for device-agnostic interaction, but most of the heavy lifting is done in `MicroSession`, where all of the device bookkeeping and core functionality is implemented. The implementations of `MicroDeviceAPI` and `MicroModule` consist of mostly plumbing that feeds into functions provided by `MicroSession`. This relationship is shown graphically in Figure 8.

`LowLevelDeviceAPI`

Before we can do anything of use with bare-metal devices, we need to build abstractions for the core operations involved in any device interaction—namely, reading memory, writing memory, and executing code from a specified address (detailed in Figure 9).

With the ability to read, write, and execute memory, we can define the mechanics of `MicroDeviceAPI` and `MicroModule` in terms of these primitives.

Note that this interface abstracts over the device communication protocol, so although OpenOCD is *currently* the standard interface, when a new, faster protocol becomes

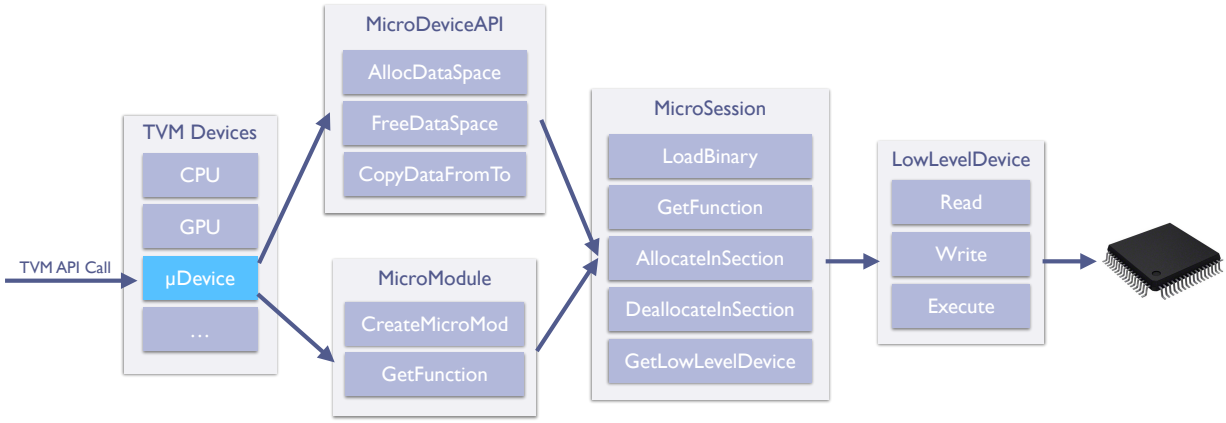

Figure 8: Diagram showing how TVM API calls flow through the µTVM runtime and ultimately reach the device

```
class LowLevelDeviceAPI {
   virtual void Read(TargetPtr addr,
                     void* buffer,
                     size_t num_bytes) = 0;
   virtual void Write(TargetPtr addr,
                      void* buffer,
                      size_t num_bytes) = 0;
   virtual void Execute(TargetPtr func_addr,
                        TargetPtr breakpoint_addr) = 0;
};
```

Figure 9: The C++ `LowLevelDeviceAPI` interface. Providing an implementation for the three methods above is all that is required to satisfy this component of the µTVM interface. The `TargetPtr` data type is a wrapper around a raw pointer, so the type system prevents us from accidentally using host pointers as device pointers and vice versa.

widely available, µTVM is future-proof in this respect, as an implementation of the `LowLevelDeviceAPI` for this new protocol serves as a drop-in replacement.

To date, there are two low-level device implementations, which we now describe.

`HostLowLevelDeviceAPI`

The host low-level device API is primarily for debugging purposes, because bare-metal devices often have uninformative error messages and limited debugging support. By using this device, we can at least rule out device communication as the problem when anything goes wrong.

The implementation of this low-level device simply allocates a region of memory on the host machine and allows for reading, writing, and execution just by using C++ operations (reading from addresses, writing to addresses, and calling functions, respectively).

`OpenOCDLowLevelDeviceAPI`

The OpenOCD low-level device API is the instance we care about for most real-world devices. As the name suggests, this instance uses OpenOCD, which technically stands for "Open On-Chip Debugger", though we've abused it for its capabilities in performing basic read/write/execute operations. To use it, we first connect it to the device, which it communicates with using the widely-supported JTAG debugging standard. The host then interfaces with OpenOCD by connecting with a socket and sending commands in Tcl—a high-level scripting language—over the network. OpenOCD then translates these Tcl commands into the appropriate JTAG commands (Figure ??). With this design, the user must have OpenOCD open and configured separately from TVM, which we admit is not optimal.

The implementations for reading and writing are very simple, because they map directly onto the read/write interface in OpenOCD . The trickiest functionality to implement is code execution.

OpenOCD provides an interface for executing code at an arbitrary address, but it doesn't wait for the device to finish executing the function before returning. Even so, if we hit the `return` statement in `main`, what happens? We have nowhere to return to *on the device*, so we want to transfer control back to the host, rather than letting the device spin indefinitely. To achieve this transfer of control, we use OpenOCD's breakpoint interface. To create a breakpoint, you specify a memory address, and a breakpoint is set there. Now, upon execution, the device will halt when it reaches that address, and control returns to the host.

`MicroSession`

The micro session class acts as the orchestrator of interactions with the device. The session handles both allocation of binaries in the device's memory and execution of functions with

given arguments (including the encoding of those arguments on the device).

A simple linear allocator is used for every ELF section (i.e., text, data, bss, etc.) to remap sections on binaries to the allocated regions on the device.

Encoding of arguments is done by matching on the type of each argument and dispatching to the appropriate encoder for that type. Each encoder writes each field of the argument into a host-side temporary buffer, accounting for both the endianness of the target device and for any data alignment restrictions. Once all arguments have been encoded in this buffer, the buffer is flushed to the arguments section of the device.

`MicroDeviceAPI`

The micro-device API extends the already-existing `DeviceAPI` class in TVM. The purpose of the *general* device API class is to provide device-agnostic memory management primitives. The micro-device API achieves this by using methods in the low-level device API in its implementation.

One of the key constraints we needed to consider when designing **μ**TVM was memory. Most bare-metal devices are very limited on memory, and because of this, we arrived at a somewhat peculiar implementation for allocating memory.

With any dynamic allocation scheme, there is the space overhead of maintaining the bookkeeping data structures, but there is also overhead for the procedures themselves that manipulate these data structures. In order to avoid wasting precious device space, the host manages the memory on the device. One of the implications of this is that tensors can be passed around as usual on the host machine, but the tensor's data field is a pointer to *device* memory. Currently, we use a linear allocator for the sake of simplicitly, but we intend to implement more complex memory management in the future.

With this API implemented, standard TVM objects can now be allocated and passed around, with the actual data being stored on a bare-metal device.

Tensor Creation

When a tensor is created, the call flows to the `MicroDeviceAPI`, and space is allocated on the device via `AllocDataSpace`. The allocation call then flows to `AllocateInSection`, where the section is the heap, and the session's allocation bookkeeping is updated (it's important to note there is no interaction with the device as a result of this call). With space allocated for the tensor, its data is then loaded onto the device via `CopyDataFromTo`, which calls `GetLowLevelDevice` to get access to the `Write`. Once `Write` finishes, the user receives a TVM tensor object storing metadata (address of data on device, data type, and shape) for the tensor that is now resident in device memory.

Since multiple sessions can be open simultaneously, but only **one** session can be active at any given moment, **μ**TVM tensors additionally store a smart pointer to the session they

belong to. This prevents premature session destruction when a context switch from one session to another occurs, because the reference count to the session does not drop to zero.

`MicroModule`

The micro module class allows an ELF object file to be loaded, linked, and then dumped onto the device by using a micro session. Once the linked binary is dumped, TVM `PackedFuncs` that map to functions on the device can be created and called. When they are called, the address of the function and the arguments used to call it are passed to the micro session, so the arguments can be encoded and written onto the device.

While the micro-device API allows us to create objects in TVM, the micro module class allows us to load functions from a file and to call those functions, as if they were any other function in TVM.

Module Creation

Following Figure 8, when a module is created, the call flows to `CreateMicroMod`, which routes to `LoadBinary`. `LoadBinary` then allocates space for each section of the binary and dumps each section of the binary into its allocated slot (according to the memory layout in Figure 10). In the now-loaded binary, function pointers corresponding to utility functions (e.g., scratchpad allocation/deallocation) are patched to point to implementations in the device runtime. The mapping from function symbols to their device addresses are then stored, so functions can later be retrieved by name via TVM's `GetFunction` API call. An overview of this process can be seen in Figure 10.

Function Calls

The micro module class further wraps the low-level `Execute` method by handling the sending of arguments. This functionality is implemented in the micro module, because it has access to the binary and can search for symbols (e.g., function names and global variables).

By the time we're sending arguments to the device, whatever tensors we're operating on will have already been allocated in the device's memory, so all we need to do is tell the device *where* the arguments are.

Since the runtime uses lazy execution, instead of eagerly serializing argument metadata and `UTVMTask` data, we accumulate function call metadata on the host side, before flushing it to the device.

When flushing, the argument tensors for each task are serialized in a way that conforms to the `TVMValue` struct and the argument tensors' type IDs are serialized, then both are written contiguously in the arguments section. With all of the argument data structures now on the device, `UTVMTask` structs (Figure 6) are constructed that correspond to each of

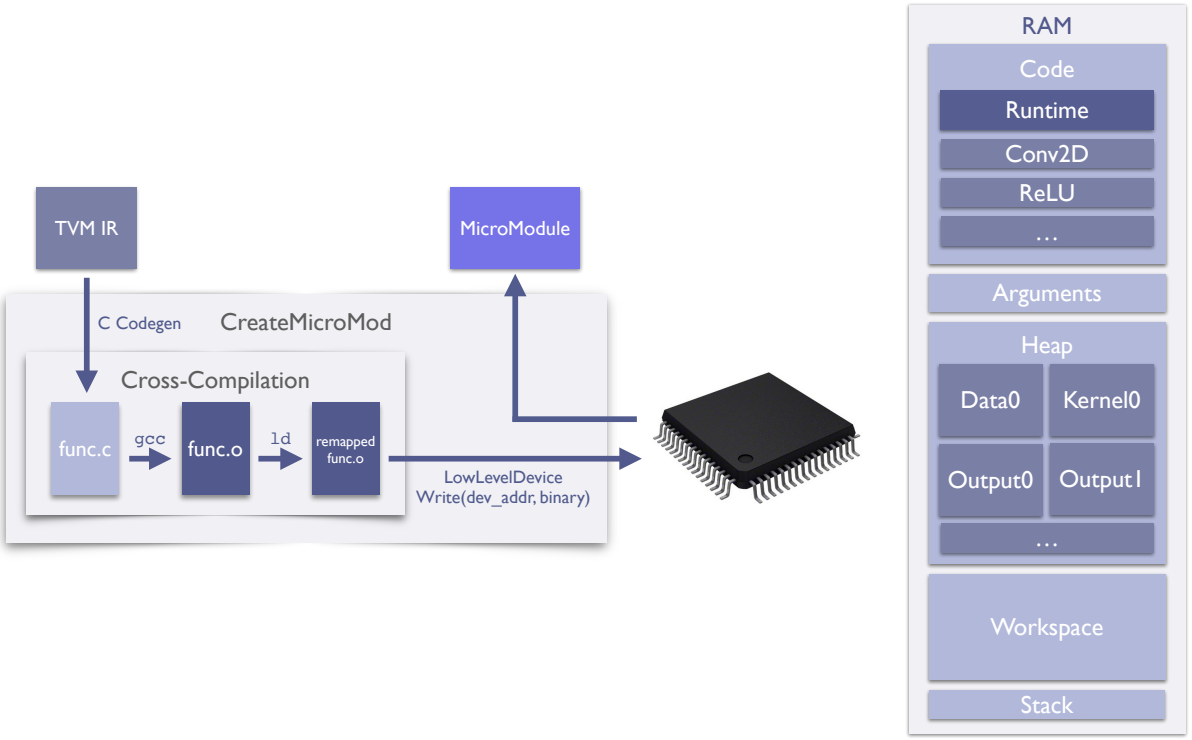

Figure 10: **(Left)** Sequence of steps involved in creating a `MicroModule` **(Right)** The µTVM device memory layout in RAM

the accumulated tasks, the array of tasks is serialized to `utvm_tasks`, and `utvm_num_tasks` is set to the number of tasks written (Figure 11).

Once the tasks have been written, we use the `Execute` method in `LowLevelDeviceAPI` with the address of `UTVMInit` as the function address and `UTVMDone` as the breakpoint address. Recall that `UTVMInit` performs device-specific initialization (e.g., enabling the floating point unit or setting the stack pointer register) before calling into `UTVMMain`.

In order to retrieve the return value, we don't need to do anything special, because we currently require functions to be written in destination-passing style.

Evaluation

With this infrastructure in place, we sought to answer the following questions:

1. Is µTVM truly device-agnostic?
2. How much effort is required to experiment with optimizations using µTVM?

To evaluate (1), we ran our experiments on three diverse targets:

- An Arm STM32F746NG development board, featuring a Cortex-M7 processor
- Spike, a functional RISC-V ISA simulator

```
volatile UTVMTask utvm_tasks[TASK_QUEUE_SIZE] = {};
volatile uint32_t utvm_num_tasks = 0;

// Called by UTVMInit, after device-specific initialization
// is finished.
void UTVMMain() {
  for (int i = 0; i < utvm_num_tasks; i++) {
    utvm_tasks[i].func(
      (void*) utvm_tasks[i].arg_values,
      (void*) utvm_tasks[i].arg_type_codes,
      utvm_tasks[i].num_args);
  }
  UTVMDone();
}

// Dummy function to signal execution is finished for device
// backends which require breakpoints.
void UTVMDone() {}
```

Figure 11: Simplified version of the µTVM on-device runtime. When task execution is forced, all accumulated arguments and task structs are written into the arguments section and `utvm_tasks`, respectively. Then, a breakpoint is set at the location of the symbol `UTVMDone` (retrieved using the cross-compiler binutils). Device execution then begins at `UTVMMain`, and when execution finishes, the breakpoint will be triggered, and control will be returned to the host.

- The µTVM x86 emulated device, which creates a memory arena on the host machine that is interfaced with as if it is a bare-metal device

To evaluate (2), we explored optimizations for the Arm board that give the biggest bang for the buck. For single-operator comparisons, we only consider convolution, since it dominates the run time.

As a point of comparison, we pulled a quantized CIFAR-10 CNN from a tutorial by Arm. In the tutorial, CMSIS-NN (a library of highly optimized kernels by Arm experts) was used as the operator library, which allowed us to directly compare the results of µTVM with CMSIS-NN on the Arm board.

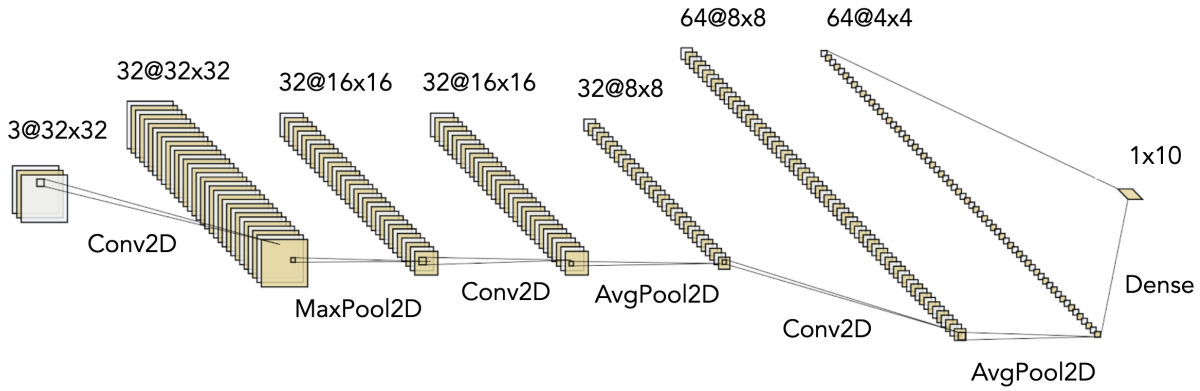

Figure 12: Diagram of CIFAR-10 CNN

Methodology

In our experiments, we used a fork of November 2019 TVM (commit 2571449), version 5.6.0 of CMSIS-NN (commit b5ef1c9) and version 8 of Arm's GCC toolchain (revision 273027). The host machine used in our experiments ran Ubuntu Linux 18.04.4 LTS and sported an AMD Ryzen Threadripper 2990WX 32-Core Processor with 62GB of RAM. All evaluation scripts for this thesis are contained in a GitHub repository.

Single Operator

The out-of-the-box (untuned) performance of µTVM isn't great. There are, however, generic optimizations that we can apply to close the gap. On most hardware, unrolling loops, reordering loop axes, and tiling data layouts can give significant performance improvements when tuned carefully. We implemented these optimizations by defining a TVM schedule, which we then autotuned, achieving the "µTVM Non-SIMD Tuned" results in Figure 14.

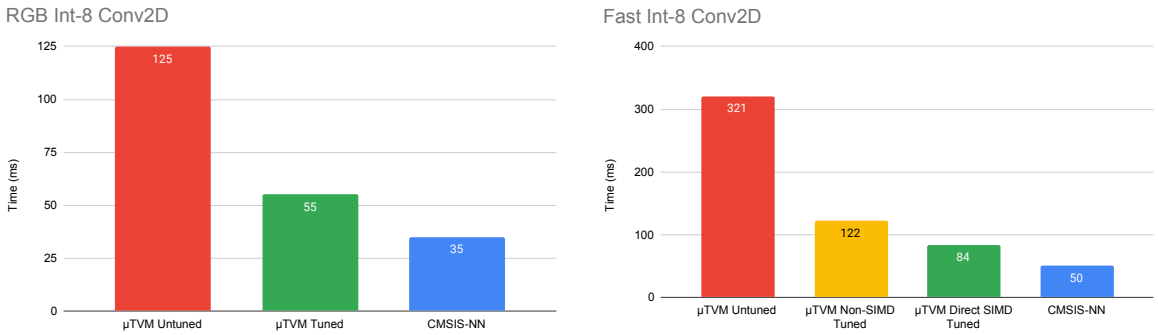

Figure 13: 2D convolution comparisons with CMSIS-NN on an Arm STM32F746NG. **(Left)** Data shape is `(N=1, H=32, W=32, C=3)` and kernel shape is `(H=5, W=5, I=3, O=32)`. Corresponds to the 1st convolution in the CNN. **(Right)** Data shape is `(N=1, H=8, W=8, C=32)` and kernel shape is `(H=5, W=5, I=32, O=64)`. Corresponds to the 3rd convolution in the CNN.

Arm-Specific Optimizations

With CMSIS-NN, the first convolution maps to their RGB convolution implementation (specifically for usage in input layers) and the latter two map to their "fast" convolution implementation. Our performance was already competitive with CMSIS-NN's RGB convolution after generic optimizations, so we focused on improving our comparison with their fast convolution.

After further analysis, we realized they were achieving significant speedups via single-instruction multiple-data (SIMD) operations offered by the Cortex-M7. Following the strategy detailed in their paper [12], we implemented a general matrix multiplication (GEMM) microkernel that uses these SIMD intrinsics. Then, using TVM's tensorization mechanism, we injected our microkernel in the core of the schedule and "autotuned around it". Though first-class support for the intrinsics in TVM's code generation facilities is the best move in the long run, tensorization provided a "quick-and-dirty" solution to supporting SIMD.

Tensorization works by defining a microkernel that can be inserted into the innermost loop of a TVM operator. Using this mechanism, adding SIMD support for the Arm board was as simple as defining a microkernel in C (found here) that mirrored the implementation in their paper. We defined a schedule that used this microkernel (found here), autotuned it, achieving the "µTVM SIMD tuned" results in Figure 14.

While we were able to use the SIMD microkernel for direct convolution, CMSIS-NN uses what they call "partial im2col" as their implementation strategy. Im2col is a technique for converting 2D convolution into GEMM, a great performance win when highly optimized GEMM subroutines are available [1]. Their variation of im2col offers a tradeoff between performance and memory usage. Instead of manifesting the entire im2col matrix at once, *partial* im2col generates only a few columns at a time. Then, each batch can be sent to a

SIMD GEMM function.

Our hypothesis was that, among other optimizations, we could find the optimal batch size via autotuning. In practice, we found partial im2col to be significantly slower than our direct convolution implementation, so we don't include it in the rest of our results.

End-To-End

After exploring optimizations for convolution, we set out to measure their effects on end-to-end performance. For the Arm board, we collected untuned results, results that were tuned **without** any use of SIMD, results that were tuned **with** SIMD, and results using CMSIS-NN. For Spike and the emulated host device, we only collected untuned results and generic tuned results.

On the Arm STM32-series board, we were able to improve performance by over 3× compared to the initial untuned operators, and we achieved results much closer to CMSIS-NN. Additionally, we were able to significantly improve performance on Spike and the x86 emulated device. Though the RISC-V and x86 ***numbers*** don't mean much, they show we can use the same infrastructure (µTVM) to optimize performance on vastly different architectures.

Discussion

In this section, we discuss performance optimizations we could pull from CMSIS-NN and TVM-specific optimizations we could implement to make µTVM (and TVM overall) more competitive.

Batch expansion of `int8` weights into `int16`, to cut down on duplicate expansion for SIMD, would drastically improve performance. To compare, CMSIS-NN's `arm_convolve_HWC_q7_fast` implementation is able to simultaneously use SIMD to expand from `int8` to `int16` **and** generate the partial im2col buffer, by calling `arm_q7_to_q15_reordered_no_shift`. However, with our GEMM microkernel, we expand weights *inside* of the microkernel, rather than factoring out this expansion.

A similarly valuable optimization is reordering-aware kernels. As mentioned before, CMSIS-NN uses `arm_q7_to_q15_reordered_no_shift` , which is faster than `arm_q7_to_q15_no_shift`, but the mechanics of the SIMD expansion intrinsic used in the former leaves the data in a jumbled format (see Figures 3 and 4 of [12]). However, `arm_nn_mat_mult_kernel_q7_q15_reordered`, is written to accomodate inputs that have been reordered by this intrinsic by reordering indexing calculations accordingly, saving a considerable amount of time.

Direct convolution being faster than partial im2col came as quite a surprise to us, and we believe it stems from a number of small performance problems that add up. One problem is that present-day TVM has trouble reasoning through reshapes and tensorization,

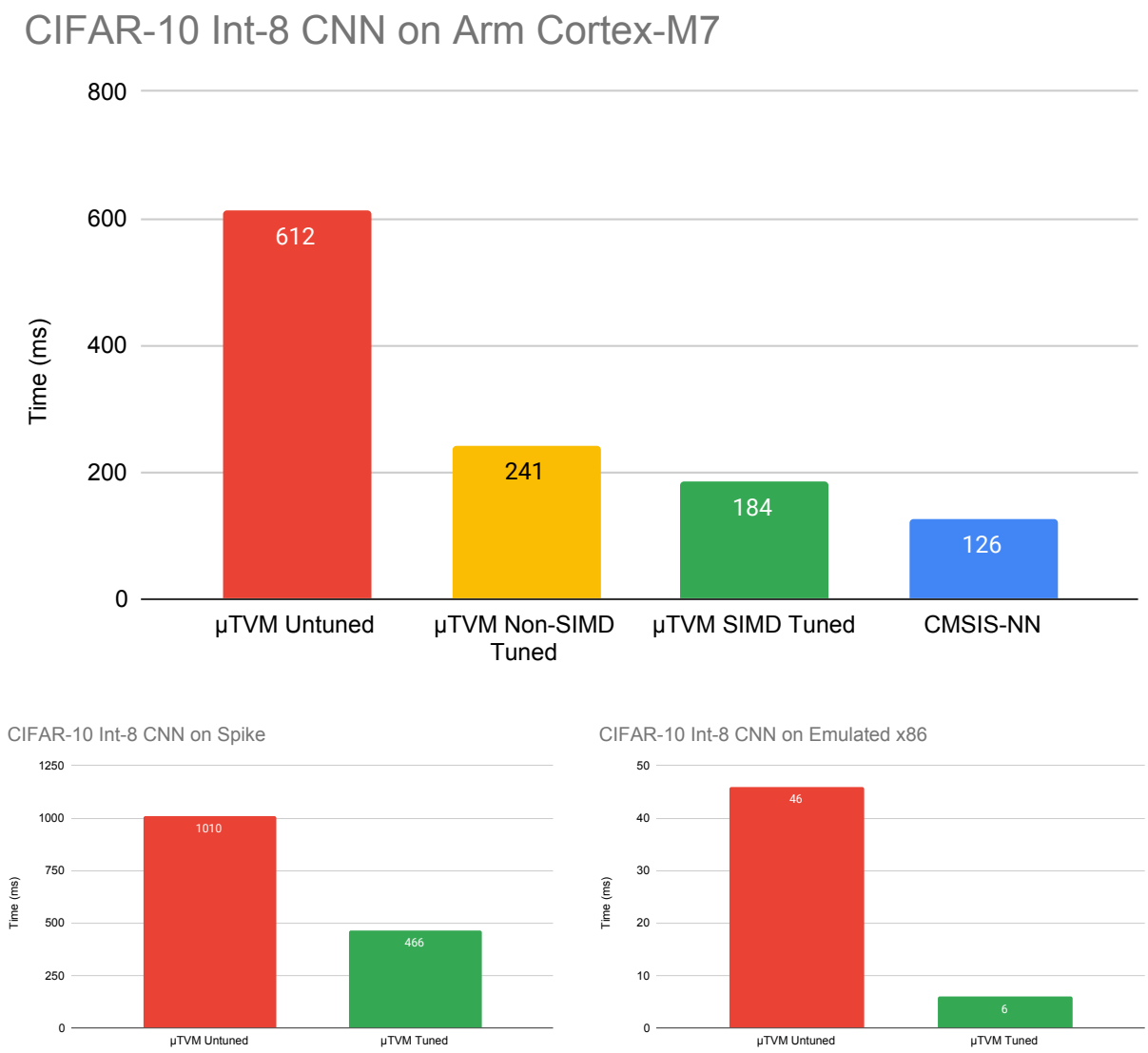

Figure 14: `int8`-quantized CIFAR-10 CNN results. (**Top**) Untuned vs. tuned vs. CMSIS-NN results on Arm STM32F746NG (**Bottom Left**) Untuned vs. tuned results on Spike, a RISC-V ISA simulator (**Bottom Right**) Untuned vs. tuned results on **µ**TVM's emulated x86 host device

which means we can't inline reshapes near the GEMM microkernel in the partial im2col schedule, and we get unnecessary workspace allocs and tensor copies.

We also may have achieved better tuning results if we enabled direct support for Arm's SIMD intrinsics, rather than using tensorization, since tensorization only allows us to tune the *size* of the matmul microkernel, but we can't tune internal parameters at all. Finer-grained tuning could be quite important, as partial im2col relies on a good GEMM implementation.

Another micro-optimization in CMSIS-NN's convolution kernel is splitting the convolution into 3×3 tiles, allowing fewer padding checks in the non-corner tiles (and no padding checks in the center tile).

An additional benefit of implementing µTVM in TVM is that in cases such as this where an external operator that surpasses TVM's performance is readily available, that operator can be plugged in directly via TVM's Bring Your Own Codegen framework.

Future Work

Self-Hosted Runtime: The Final Frontier

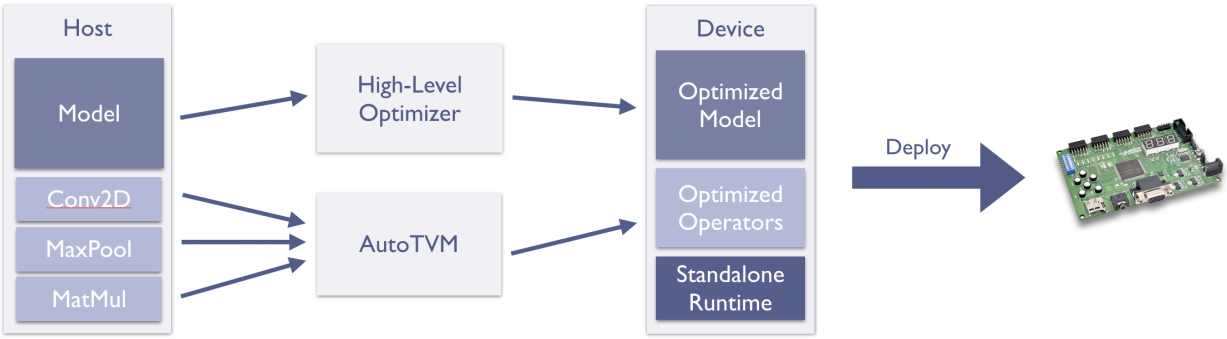

Figure 15: The envisioned µTVM optimization and deployment pipeline

While end-to-end results are already obtainable with the current runtime, deployment of these models in a standalone capacity is currently still on our roadmap. As mentioned in earlier sections, the AutoTVM-oriented runtime relies on the host to allocate tensors and to schedule function execution. However, to be useful at the edge, we need a pipeline through µTVM that generates a **single** binary to be run on a bare-metal device. Users could then easily integrate fast ML into their applications by including this binary.

Self-hosted runtime aside, there are plenty of other interesting research avenues to pursue.

Thus far, we've used the vanilla AutoTVM algorithm, but it would be interesting to make some domain-specific tweaks to autotuning. For example, setting a desired operator

size and penalizing implementations that surpass the desired size would help to generate leaner binaries.

To cover the lowest common denominator of edge devices, we have focused on targeting the JTAG protocol for facilitating device interaction. However, JTAG is a protocol designed for debugging purposes. Futhermore, communicating with OpenOCD using Tcl commands is also quite expensive. While the OpenOCD/JTAG combo have worked for prototyping purposes, we believe there is an opportunity here to design a leaner communication protocol that obviates the need for this combo and that scales and unifies interaction with bare-metal devices more effectively. TCP/IP, for example, is not a desirable protocol, as the runtime required to implement the protocol could itself exceed the memory bounds of most target microcontrollers. Conveniently, when this hypothetical new protocol comes, µTVM is futureproof in this respect, because the `LowLevelDeviceAPI` abstracts over the device communication protocol. Although OpenOCD is *currently* the standard interface, all that's required to support this new protocol is an implementation of the `LowLevelDeviceAPI`, and it would then serve as a drop-in replacement.

Conclusion

In this thesis, we've described µTVM, an extension to TVM that targets bare-metal devices. µTVM makes novel use of existing low-level hardware tools and serves as a platform for performing research... ON THE EDGE!

References

[1] Chellapilla, K., Puri, S., and Simard, P. Y. High performance convolutional neural networks for document processing.

[2] Chen, T., Moreau, T., Jiang, Z., Shen, H., Yan, E. Q., Wang, L., Hu, Y., Ceze, L., Guestrin, C., and Krishnamurthy, A. TVM: end-to-end optimization stack for deep learning. *CoRR abs/1802.04799* (2018).

[3] Chen, T., Zheng, L., Yan, E. Q., Jiang, Z., Moreau, T., Ceze, L., Guestrin, C., and Krishnamurthy, A. Learning to optimize tensor programs. *CoRR abs/1805.08166* (2018).

[4] Courbariaux, M., Bengio, Y., and David, J.-P. Training deep neural networks with low precision multiplications.

[5] Elango, V., Rubin, N., Ravishankar, M., Sandanagobalane, H., and Grover, V. Diesel: Dsl for linear algebra and neural net computations on gpus. In *Proceedings of the 2Nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages* (New York, NY, USA, 2018), MAPL 2018, ACM, pp. 42–51.

[6] Gopinath, S., Ghanathe, N., Seshadri, V., and Sharma, R. Compiling kb-sized machine learning models to constrained hardware. https://www.microsoft.com/en-us/research/uploads/prod/2018/10/paper.pdf.

[7] Gupta, C., Suggala, A. S., Gupta, A., Simhadri, H. V., Paranjape, B., Kumar, A., Goyal, S., Udupa, R., Varma, M., and Jain, P. Protonn: Compressed and accurate knn for resource-scarce devices. https://www.microsoft.com/en-us/research/uploads/prod/2017/06/protonn.pdf.

[8] Gustafson, and Yonemoto. Beating floating point at its own game: Posit arithmetic. *Supercomput. Front. Innov.: Int. J. 4*, 2 (June 2017), 71–86.

[9] Hubara, I., Courbariaux, M., Soudry, D., El-Yaniv, R., and Bengio, Y. Quantized neural networks: Training neural networks with low precision weights and activations. *The Journal of Machine Learning Research 18*, 1 (2017), 6869–6898.

[10] Johnson, J. Rethinking floating point for deep learning, 2018.

[11] Kumar, A., Goyal, S., and Varma, M. Resource-efficient machine learning in 2 kb ram for the internet of things. http://manikvarma.org/pubs/kumar17.pdf.

[12] Lai, L., Suda, N., and Chandra, V. CMSIS-NN: efficient neural network kernels for arm cortex-m cpus. *CoRR abs/1801.06601* (2018).

[13] Ragan-Kelley, J., Barnes, C., Adams, A., Paris, S., Durand, F., and Amarasinghe, S. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2013), PLDI '13, ACM, pp. 519–530.

[14] Roesch, J., Lyubomirsky, S., Kirisame, M., Weber, L., Pollock, J., Vega, L., Jiang, Z., Chen, T., Moreau, T., and Tatlock, Z. Relay: A high-level compiler for deep learning, 2019.

[15] Sahaya Loui, M., Azad, Z., Delshadtehrani, L., Gupta, S., Warden, P., Reddi, V., and Joshi, A. Towards deep learning using tensorflow lite on risc-v.

[16] Shen, H., Roesch, J., Chen, Z., Chen, W., Wu, Y., Li, M., Sharma, V., Tatlock, Z., and Wang, Y. Nimble: Efficiently compiling dynamic neural networks for model inference, 2020.

[17] Soudry, D., Hubara, I., and Meir, R. Expectation backpropagation: Parameter-free training of multilayer neural networks with continuous or discrete weights. In *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 1* (Cambridge, MA, USA, 2014), NIPS'14, MIT Press, p. 963–971.

[18] Vasilache, N., Zinenko, O., Theodoridis, T., Goyal, P., DeVito, Z., Moses, W. S., Verdoolaege, S., Adams, A., and Cohen, A. Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions, 2018.