Tensor Virtual Machine

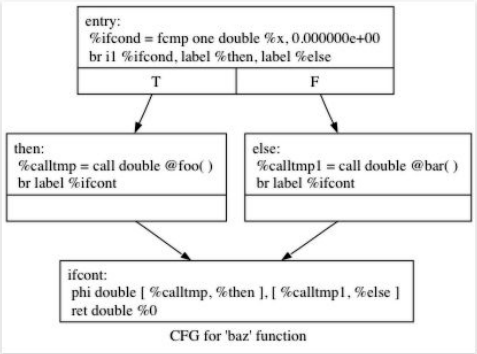
Chen et al.
Presented by Sultan Durrani, Ryan Ziegler

Agenda

- Background and Motivation
- Machine learning compilers
- Existing Work (Halide)
- TVM design and optimizations
- Experimentation and evaluation
- Strengths and Weaknesses
- Future directions

How do (CPU) compilers work?

- “Lower” high-level (i.e. C) code into *basic blocks*
- A basic block contains no control flow
- Basic blocks are connected by edges representing control flow
- We can follow these edges to trace dataflow
- SSA IR: values are written to *exactly* once
- Optimization passes transform the IR while preserving program semantics



Compilers, more generally

- Take an input program, convert it into some representation
- Transform the representation to improve performance while preserving semantics
- Output code (i.e. ASM)
-

```c
int func(int x) {
    return x + 2;
}

int main() {
    func(3);
}
```

Source C Program

```
func(int):
        lea     eax, [rdi+2]
        ret
main:
        xor     eax, eax
        ret
```

Compiled with -O3

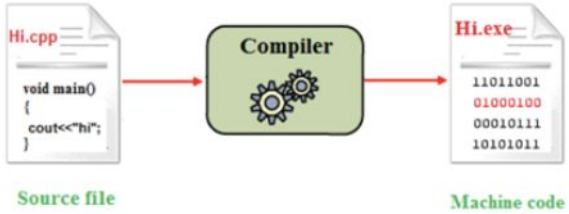
```
func(int):
        push    rbp
        mov     rbp, rsp
        mov     dword ptr [rbp - 4], edi
        mov     eax, dword ptr [rbp - 4]
        add     eax, 2
        pop     rbp
        ret

main:
        push    rbp
        mov     rbp, rsp
        mov     edi, 3
        call    func(int)
        xor     eax, eax
        pop     rbp
        ret
```

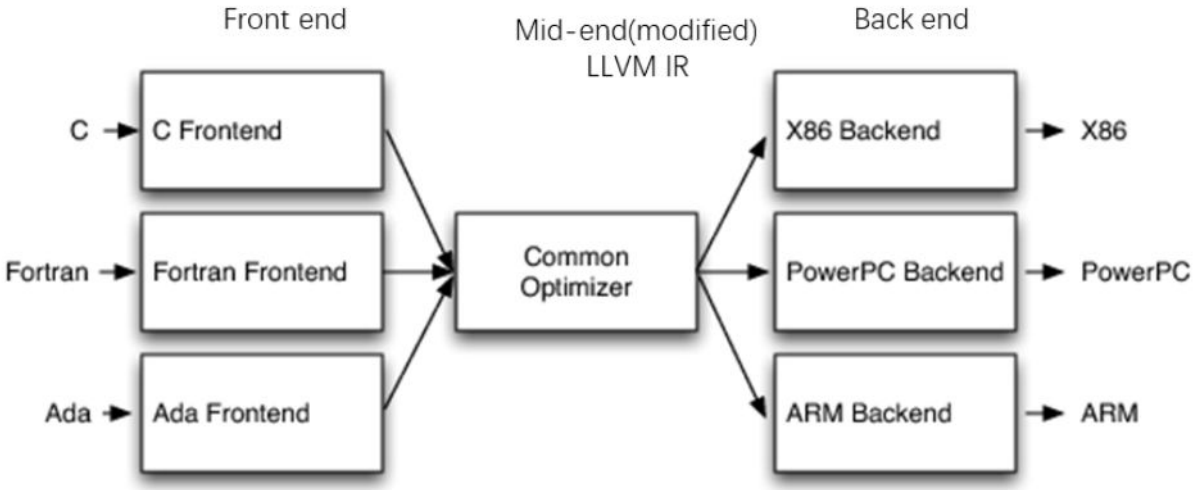
Compiled with -O0

Traditional Compilers

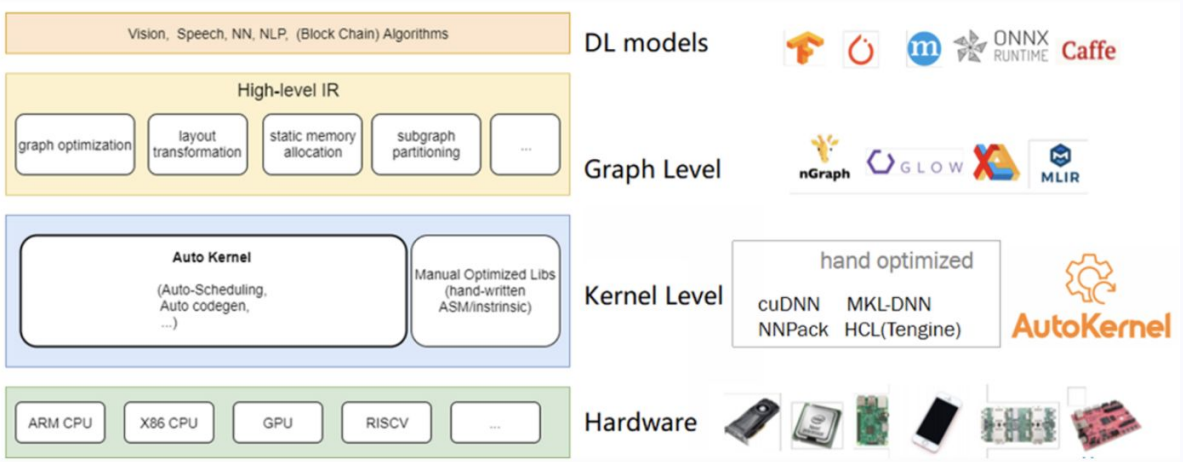
Traditionnal Compiler

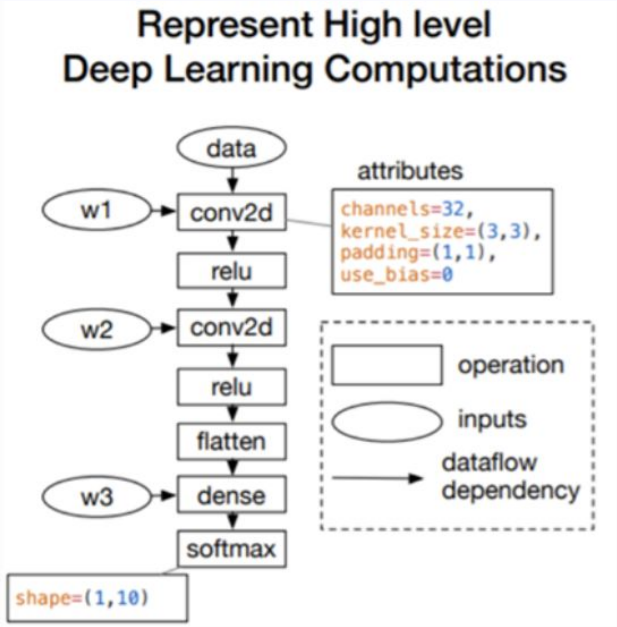


LLVM



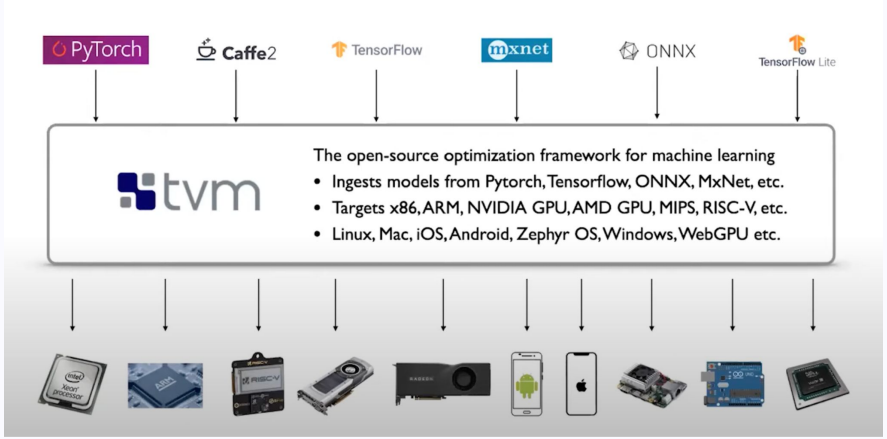
Machine Learning/Deep Learning Compilers





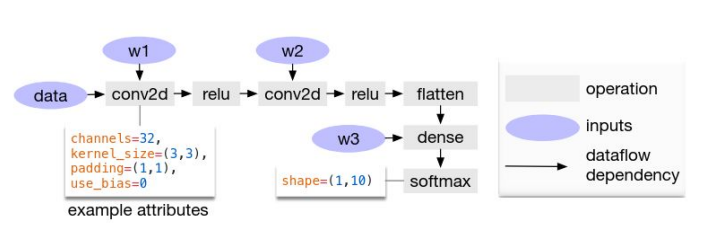
Challenges with Machine Learning compilers

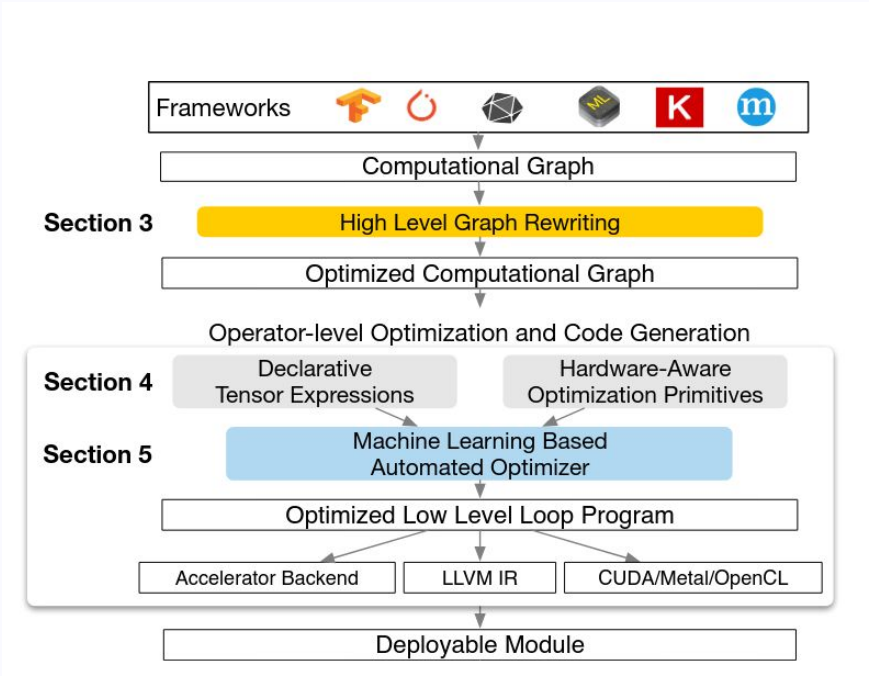
- Need to learn how to use new hardware features and accelerators. For example H100(hopper) introduced wgmma instructions. Different from mma
- Large search space for optimization
  Need to produce efficient code without manual tuning (huge configuration space)

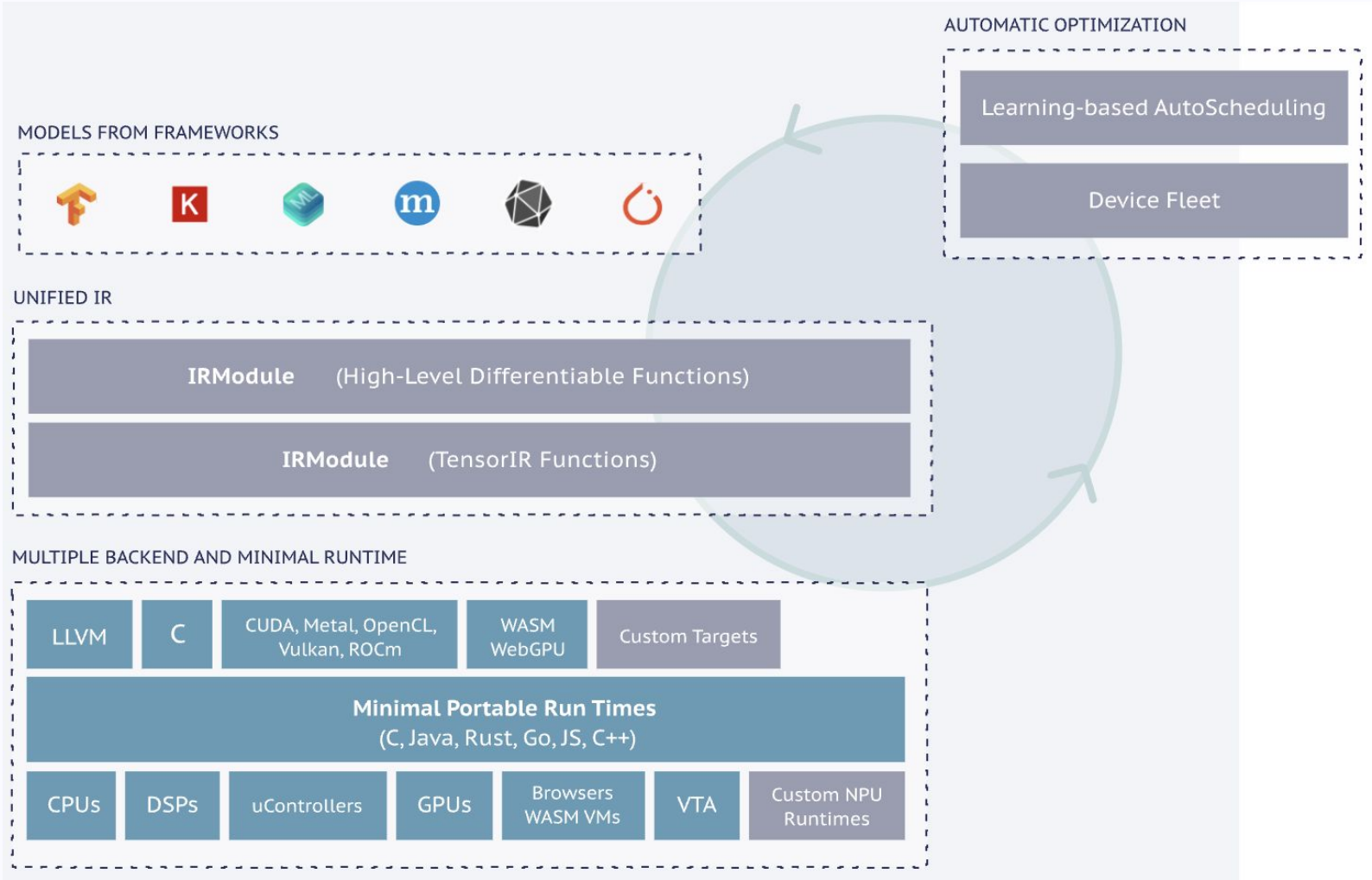


TVM Overview

- TVM takes the IR of ML frameworks and generates a compute graph
- The compute graph contains operators as nodes and edges between them representing data dependencies






AUTOMATIC OPTIMIZATION
Learning-based AutoScheduling
Device Fleet
MODELS FROM FRAMEWORKS
UNIFIED IR
IRModule (High-Level Differentiable Functions)
IRModule (TensorIR Functions)
MULTIPLE BACKEND AND MINIMAL RUNTIME
LLVM
C
CUDA, Metal, OpenCL, Vulkan, ROCm
WASM WebGPU
Custom Targets
Minimal Portable Run Times
(C, Java, Rust, Go, JS, C++)
CPUs
DSPs
uControllers
GPUs
Browsers WASM VMs
VTA
Custom NPU Runtimes

TVM Optimizations Overview

- Optimizing Tensor Operations
- Optimizing Computation Graphs
- Automating Optimizations with ML cost model

Halide

- Problem: highly parallel matrix operations are difficult to express

```
void blur(const Image<uint16_t> &in, Image<uint16_t> &bv) {
 Image<uint16_t> bh(in.width(), in.height();

  for (int y = 0; y < in.height(); y++)
   for (int x = 0; x < in.width (); x++)
    bh(x, y) = (in(x-1, y) + in(x, y) + in(x+1, y))/3;

  for (int y = 0; y < in.height(); y++)
   for (int x = 0; x < in.width (); x++)
    bv(x, y) = (bh(x, y-1) + bh(x, y) + bh(x, y+1))/3;
}
```

Naive image blur

```
void fast_blur(const Image<uint16_t> &in, Image<uint16_t> &bv) {
 __m128i one_third = _mm_set1_epi16(21846);
 #pragma omp parallel for
 for (int yTile = 0; yTile < in.height(); yTile += 32) {
  __m128i a, b, c, sum, avg;
  __m128i bh[(256/8)*(32+2)];
  for (int xTile = 0; xTile < in.width(); xTile += 256) {
   __m128i *bhPtr = bh;
   for (int y = -1; y < 32+1; y++) {
    const uint16_t *inPtr = &(in(xTile, yTile+y));
    for (int x = 0; x < 256; x += 8) {
       a = _mm_loadu_si128((__m128i*)(inPtr - 1));
       b = _mm_loadu_si128((__m128i*)(inPtr + 1));
       c = _mm_load_si128 ((__m128i*)(inPtr));
     sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
     avg = _mm_mulhi_epi16(sum, one_third);
     _mm_store_si128(bhPtr++, avg);
     inPtr += 8;
   }}
   bhPtr = bh;
   for (int y = 0; y < 32; y++) {
    __m128i *outPtr = (__m128i *)(&(bv(xTile, yTile+y)));
    for (int x = 0; x < 256; x += 8) {
       a = _mm_load_si128(bhPtr + (256 * 2) / 8);
       b = _mm_load_si128(bhPtr + 256 / 8);
       c = _mm_load_si128(bhPtr++);
     sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
     avg = _mm_mulhi_epi16(sum, one_third);
     _mm_store_si128(outPtr++, avg);
}}}}}
```

Hand-optimized blur

Halide

- Solution: separate the algorithm from the *schedule* (tiling behavior, vectorization [width], loop ordering, etc)
- Halide takes an algorithm and schedule, and generates code implementing the schedule

```
Func halide_blur(Func in) {
 Func bh, bv;
 Var x, y, xi, yi;

 // The algorithm
 bh(x, y) = (in(x-1, y) + in(x, y) + in(x+1, y))/3;
 bv(x, y) = (bh(x, y-1) + bh(x, y) + bh(x, y+1))/3;

 // The schedule
 bv.tile(x, y, xi, yi, 256, 32)
   .vectorize(xi, 8).parallel(y);
 bh.compute_at(bv, x).vectorize(x, 8);

 return bv;
}
```

Why Halide?

- It’s easier to optimize an algorithm decoupled from an execution schedule
- Algorithms can be expressed more concisely

```
Func halide_blur(Func in) {
 Func bh, bv;
 Var x, y, xi, yi;

 // The algorithm
 bh(x, y) = (in(x-1, y) + in(x, y) + in(x+1, y))/3;
 bv(x, y) = (bh(x, y-1) + bh(x, y) + bh(x, y+1))/3;

 // The schedule
 bv.tile(x, y, xi, yi, 256, 32)
   .vectorize(xi, 8).parallel(y);
 bh.compute_at(bv, x).vectorize(x, 8);

 return bv;
}
```

Why not Halide?

- Execution schedules have a high impact on algorithm runtime
- Difficult to optimize ML because schedules are not graph-level (i.e. cannot use schedules for fusion)
- Lower-level: C++ embedded DSL

```
Func halide_blur(Func in) {
 Func bh, bv;
 Var x, y, xi, yi;

 // The algorithm
 bh(x, y) = (in(x-1, y) + in(x, y) + in(x+1, y))/3;
 bv(x, y) = (bh(x, y-1) + bh(x, y) + bh(x, y+1))/3;

 // The schedule
 bv.tile(x, y, xi, yi, 256, 32)
   .vectorize(xi, 8).parallel(y);
 bh.compute_at(bv, x).vectorize(x, 8);

 return bv;
}
```

TVM：Tensor Expression DSL

```
A = t.placeholder((1024, 1024))
B = t.placeholder((1024, 1024))
k = t.reduce_axis((0, 1024))
C = t.compute((1024, 1024), lambda y, x:
                t.sum(A[k, y] * B[k, x], axis=k))
s = t.create_schedule(C.op)
```

```
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]
```

Very similar to Halide

- Specify the algorithm
- Specify the schdule

Cooperation

- Traditional nested parallelism: threads do not access one another's memory
- Cooperative parallelism: all threads fetch data they all need, allows for sharing common data
- TVM implements memory *scopes*: a compute stage can be marked as shared, and the compiler will generate cooperative code

```
for thread_group (by, bx) in cross(64, 64):
  for thread_item (ty, tx) in cross(2, 2):
    local CL[8][8] = 0
    shared AS[2][8], BS[2][8]
    for k in range(1024):
      for i in range(4):
        AS[ty][i*4+tx] = A[k][by*64+ty*8+i*4+tx]
      for each i in 0..4:
        BS[ty][i*4+tx] = B[k][bx*64+ty*8+i*4+tx]
      memory_barrier_among_threads()
      for yi in range(8):
        for xi in range(8):
          CL[yi][xi] += AS[yi] * BS[xi]
      for yi in range(8):
        for xi in range(8):
          C[yo*8+yi][xo*8+xi] = CL[yi][xi]
```

All threads cooperatively load AS and BS in different parallel patterns

Barrier inserted automatically by compiler

Tensorization

- This is analogous to vectorization on SIMD architectures
- Input instructions are multi dimensional which dictate specific layouts
- Not restricted to a fixed set of primitives, each DL accelerator could potentially have their own flavors of Tensor instructions
- TVM makes tensorization extensible, decouple hardware intrinsic from schedule
- Adds a tensorize primitive to make use of hand crafted micro kernels

```
w, x = t.placeholder((8, 8)), t.placeholder((8, 8))
k = t.reduce_axis((0, 8))
y = t.compute((8, 8), lambda i, j:
              t.sum(w[i, k] * x[j, k], axis=k))

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
```

declare behavior

lowering rule to generate hardware intrinsics to carry out the computation

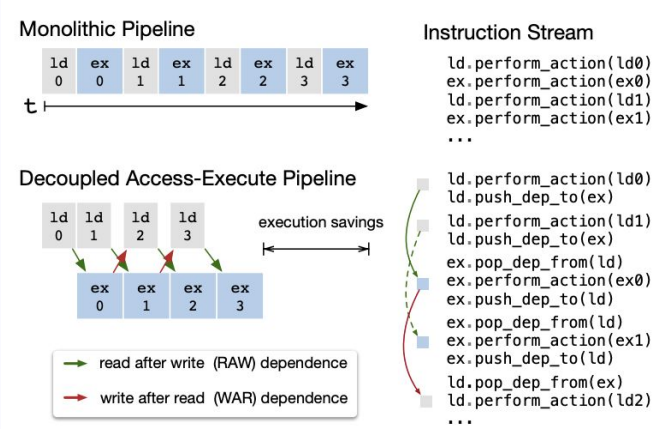
Row\Col	0 1	2 3	4 5	6 7
0	T0: {c0, c1}	T1: {c0, c1}	T2: {c0, c1}	T3: {c0, c1}
1	T4: {c0, c1}	T5: {c0, c1}	T6: {c0, c1}	T7: {c0, c1}
2				
..				
7	T28: {c0, c1}	T29: {c0, c1}	T30: {c0, c1}	T31: {c0, c1}

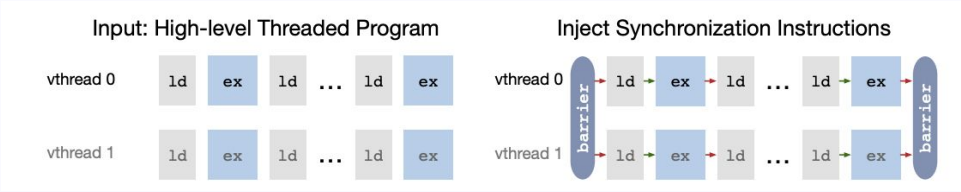
%laneid:{fragments}

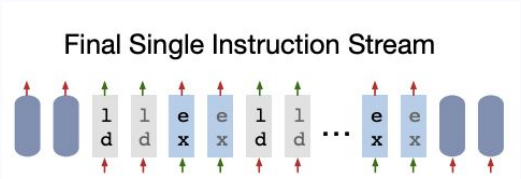
Figure 34: *MMA .m8n8k16 fragment layout for accumulator matrix C/D with* `.s32` *type*

Explicit Memory latency hiding

- Refers to overlapping memory operations with computations
- In CPU, can be achieved via hardware prefetching or SMT
- In CUDA, we have async memory copies (TMA on H100)
- TVM adds virtual threading to transform the program to a single instruction stream

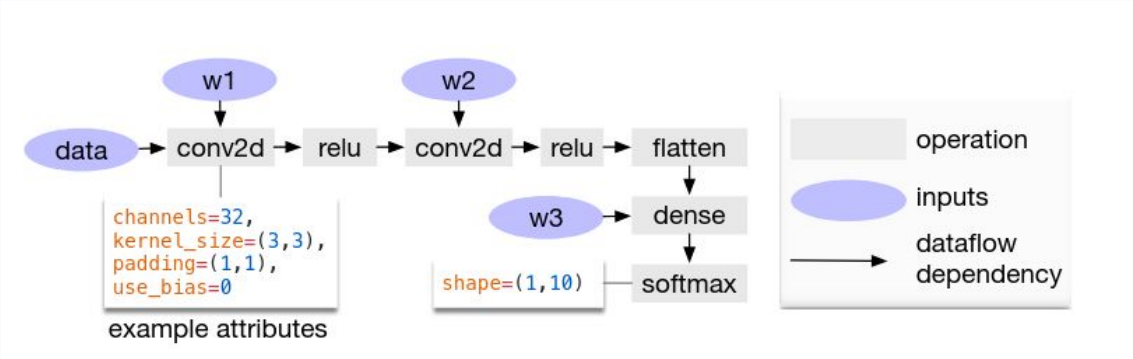






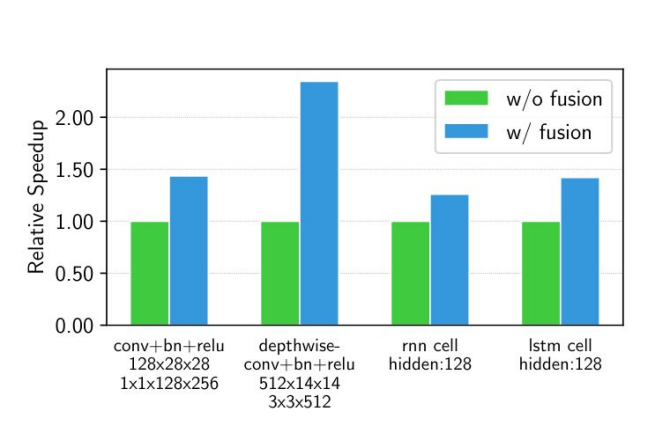
Computational graph Optimizations

- Operator implementations are unspecified
- Only: inputs, operations, dependencies
- All dimensions typically known statically
- A computation graph is analogous to a Halide *algorithm*



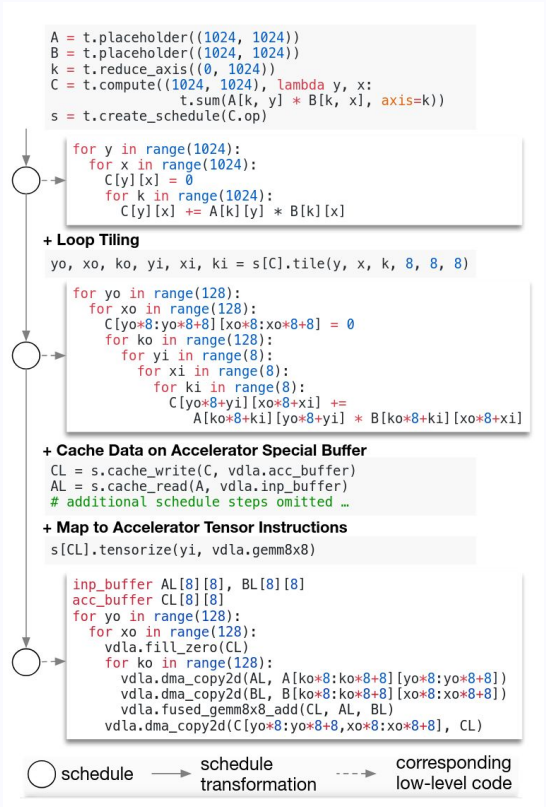
Operator Fusion

- Operator fusion refers to combining multiple operators into one
- Operators are fused following four rules:
  - Injective operators can be fused
  - Reductions may be fused to an injective operator
  - “Complex” operators (i.e. conv2d) can be fused with element-wise maps after
  - “Opaque” operators (i.e. sort) cannot be fused



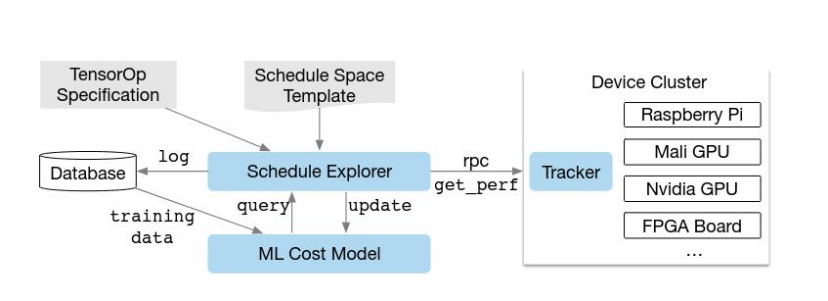
Constant folding and data layout transformation

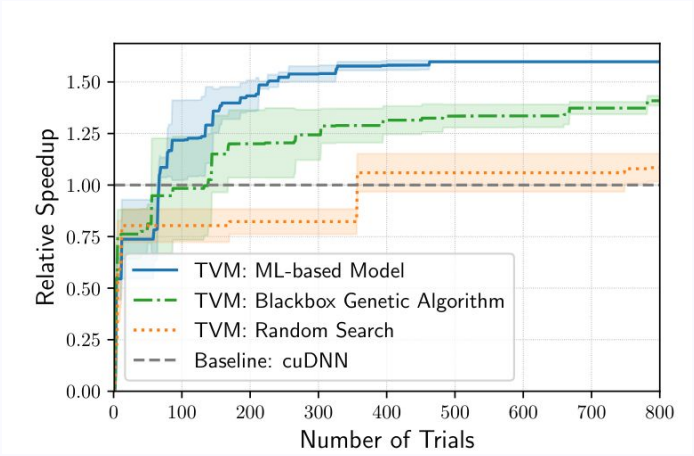
- Constant folding: if some operators have static inputs, compute their output at compile time
- Data layout transformation: adjust how tensors are stored (row major, blocks, ...) depending on target device(s)
- Memory planning: adjust memory layout based on characteristics of target device (CPU, GPU, custom) to ensure locality



Cost model

- How do we decide what optimizations to make?
- Solution: use ML to determine the projected cost (positive or negative) of making a specific optimization
  - Use simulated annealing to perform optimization using model as a cost metric





Experimentation

- TVM workload optimization over multiple platforms
- TVM vs existing DL frameworks
- TVM support for new DL operations and workloads
- TVM support for specialized accelerators

Server class GPU

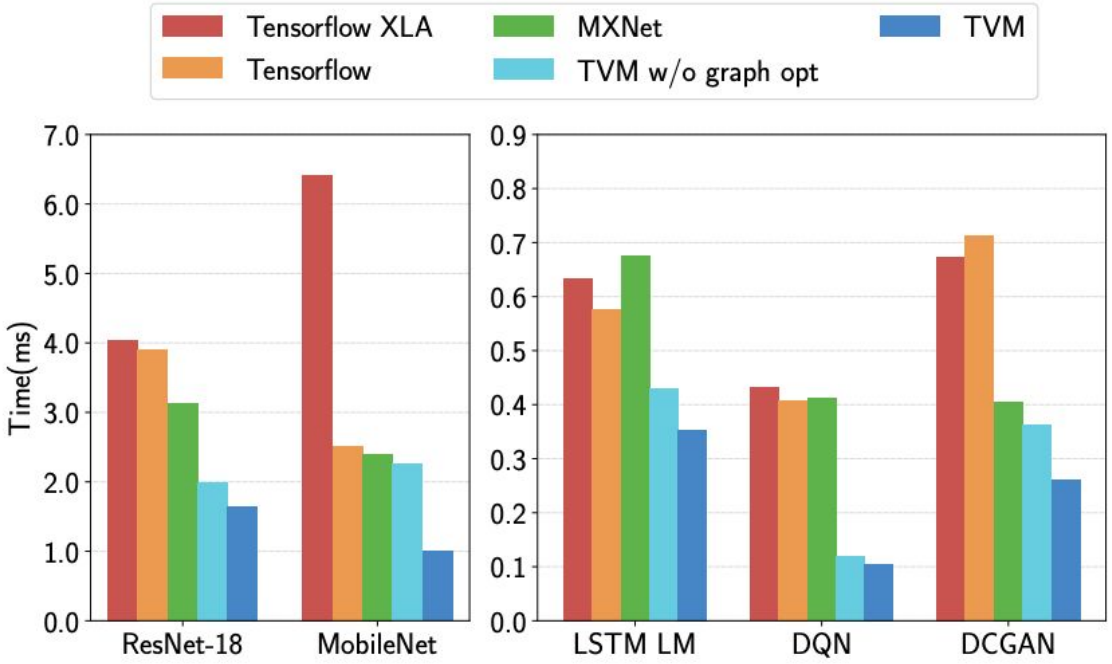


Figure 14: GPU end-to-end evaluation for TVM, MXNet, Tensorflow, and Tensorflow XLA. Tested on the NVIDIA Titan X.

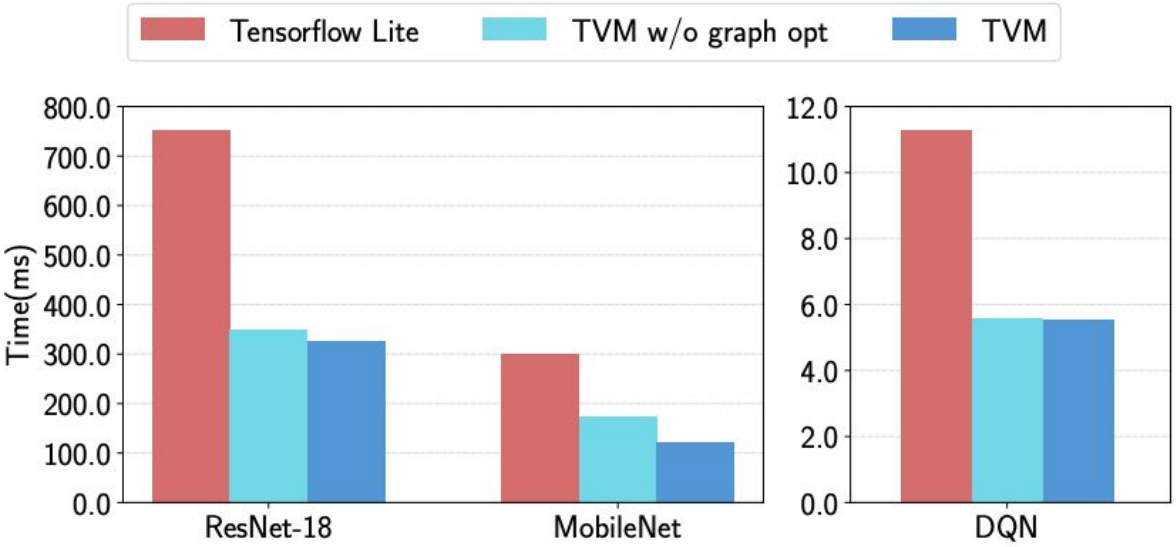


Figure 16: ARM A53 end-to-end evaluation of TVM and TFLite.

Embedded CPU

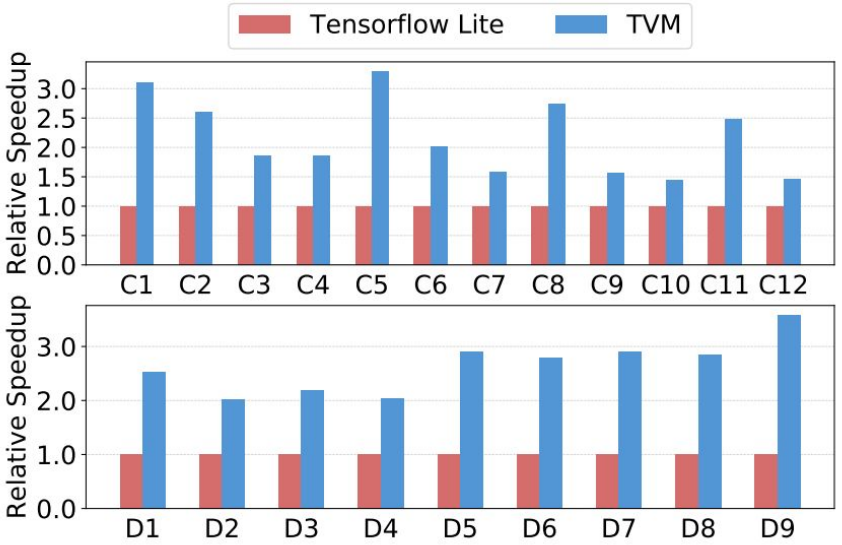


Figure 17: Relative speedup of all conv2d operators in ResNet-18 and all depthwise conv2d operators in mobilenet. Tested on ARM A53. See Table 2 for the configurations of these operators.

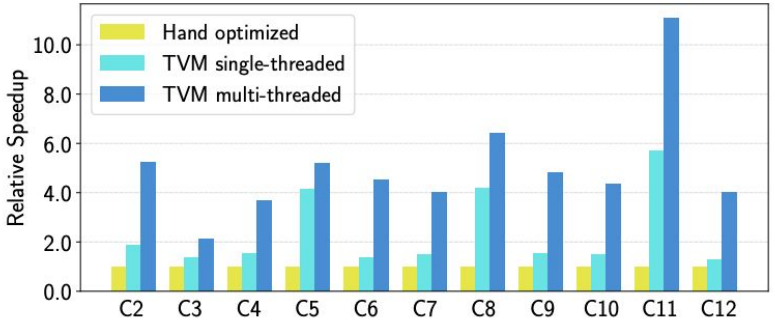


Figure 18: Relative speedup of single- and multi-threaded low-precision conv2d operators in ResNet. Baseline was a single-threaded, hand-optimized implementation from Caffe2 (commit: 39e07f7). C5, C3 are 1x1 convolutions that have less compute intensity, resulting in less speedup by multi-threading.

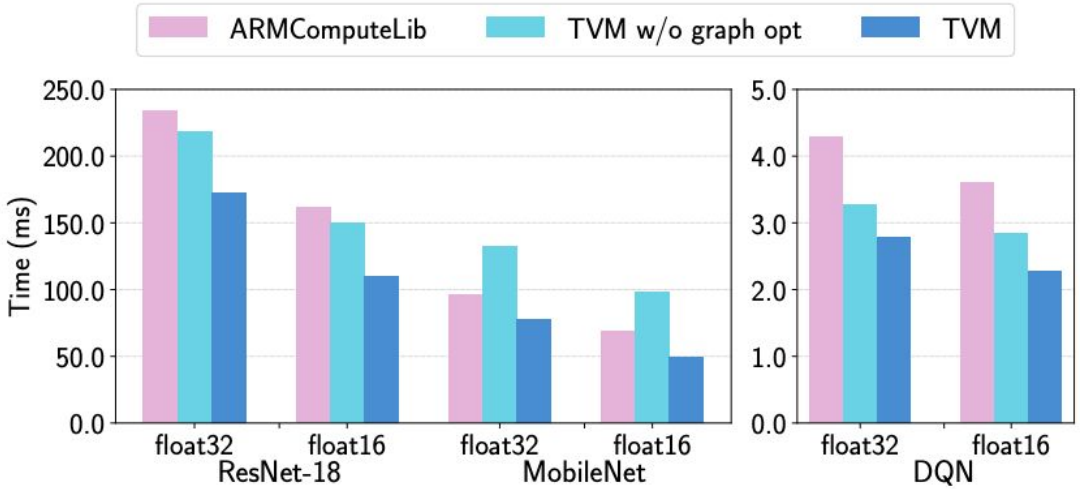


Figure 19: End-to-end experiment results on Mali-T860MP4. Two data types, float32 and float16, were evaluated.

Embedded GPU

TVM showcase on a custom accelerator

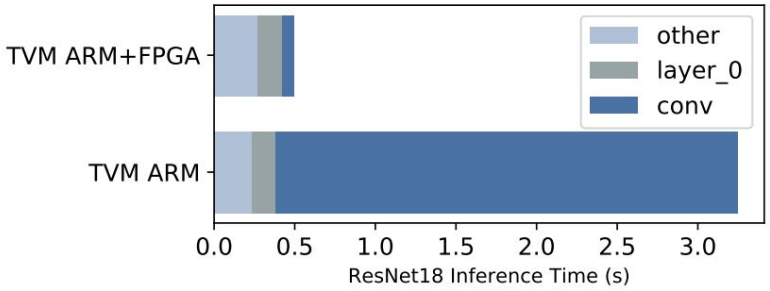


Figure 21: We offloaded convolutions in the ResNet workload to an FPGA-based accelerator. The grayed-out bars correspond to layers that could not be accelerated by the FPGA and therefore had to run on the CPU. The FPGA provided a 40x acceleration on offloaded convolution layers over the Cortex A9.

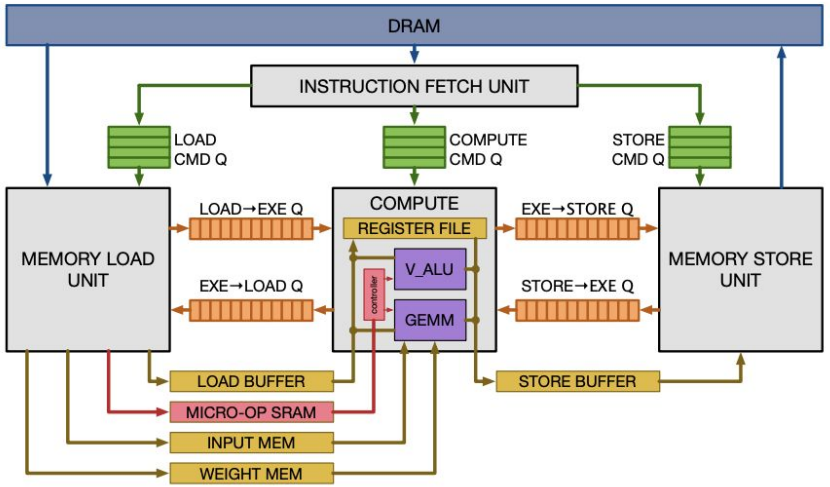


Figure 20: VDLA Hardware design overview.

Strengths

- Can generate code for many backends including new accelerators (ex FPGA based one)
- Open source implementation
- Supports popular frameworks like pytorch, tensorflow etc
- Demonstrates performance at par or even better in cases than hand tuned kernel libraries

Related Work

- Halide
- TensorFlow XLA
- FFTW and ATLAS

Limitations of TVM

- ML cost model requires training, which can be slow or costly
  - Optimization search space is extremely wide
- For better performance, more sophisticated operator fusion decision making required.
- Fragmented code base. Model definition in python while operators in Cuda/C++, programmer needs to be familiar with both and also have to learn TVM expression
- Advanced optimizations require understanding of TVM IR, no easy way to do operator extensibility

Future Work

- Improve Graph level optimization by using a using some better heuristics for Operator Fusion. Can do static analysis on the kernels and keep note of instructions, blocks, loads, math functions, barriers etc
- Work on some pruning strategies to reduce the search space for the ML cost model
- Support for heterogeneous optimizations. Graph partitioning across multiple different devices, and fuse + map operators based on device affinity