TVM: An Automated End-to-End Optimizing Compiler for Deep Learning

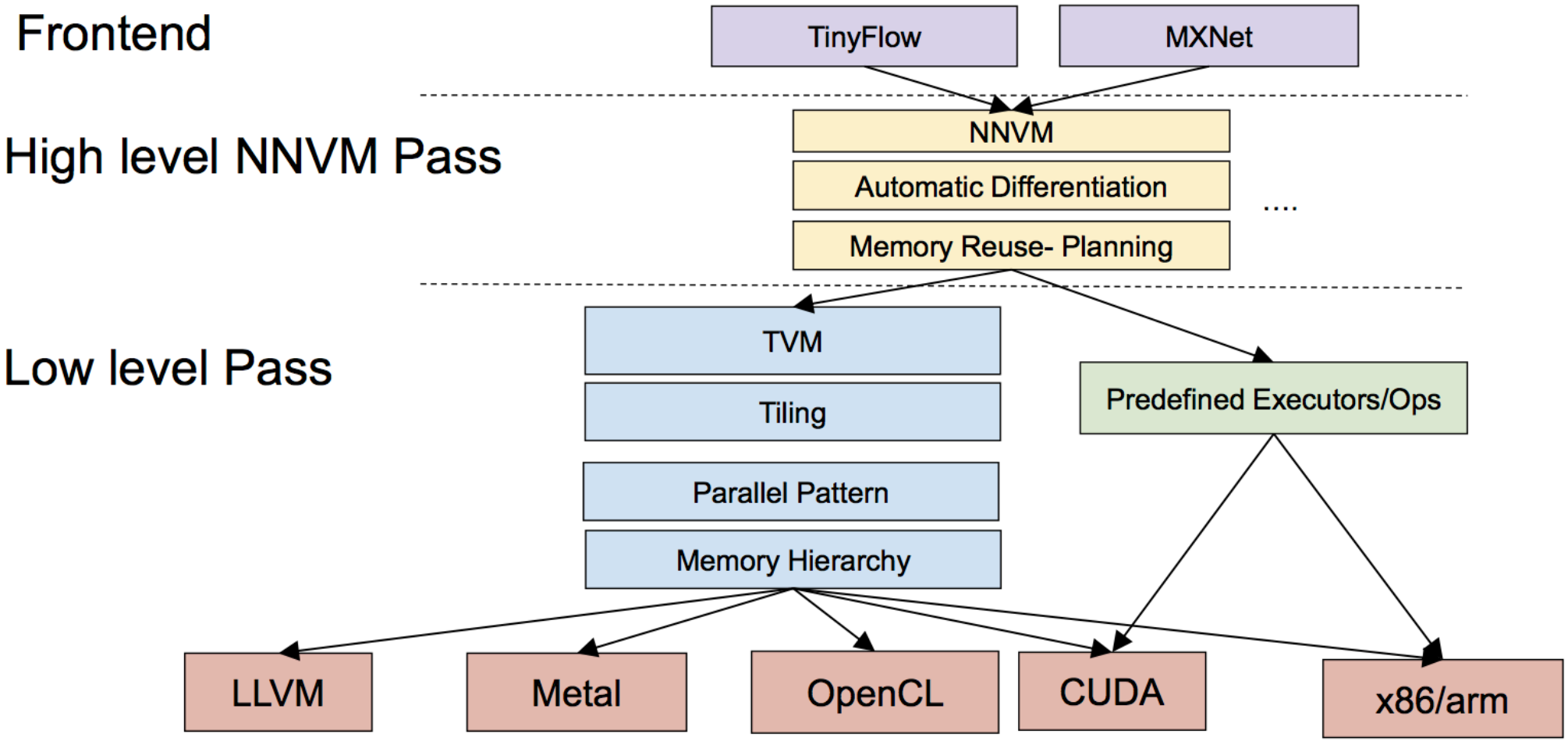
Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan

SYSML 2018

Outline

- Background
- Challenge
- Overview
  - Optimizing Computational Graphs
  - Generating Tensor Operations
  - Automating Optimization
- Evaluation

In Progress: Multi-Level Compilation Pipeline



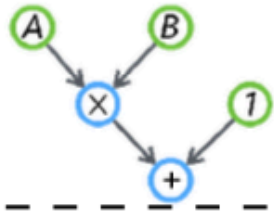
Deep Learning System Research is Exciting but Hard

Frameworks

CNTK

Caffe2

Computational graph

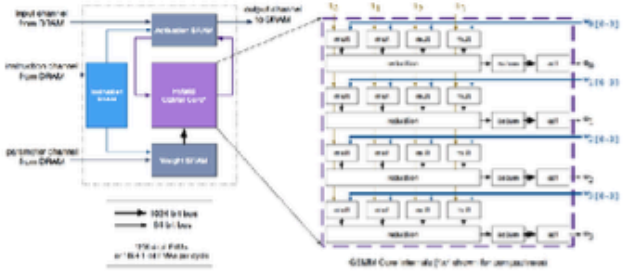


Operator Libraries **cuDNN, NNPack, MKL-DNN**

Hardware

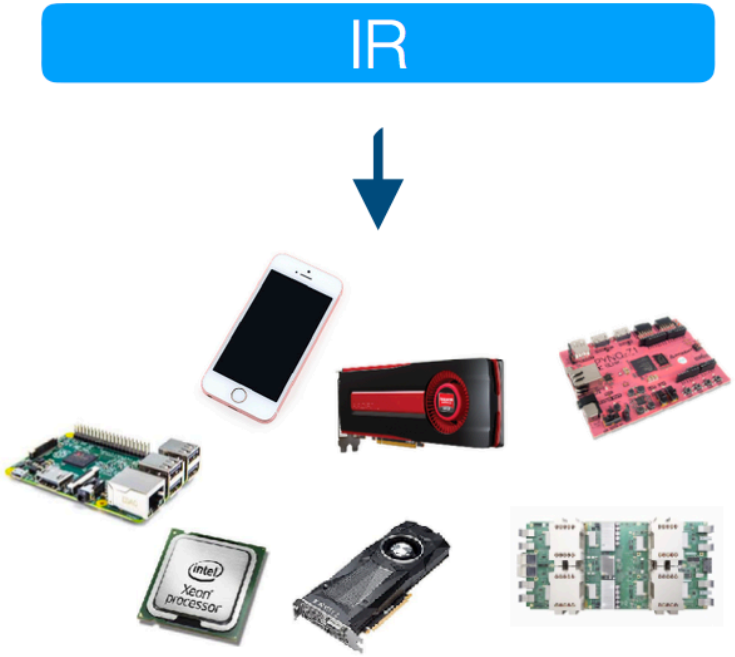
Need entire software stack on top of it!

Layout transformation
Quantization
Operator kernel optimization
Benchmarking
....



Built a new accelerator

Challenge: Hardware Diversities

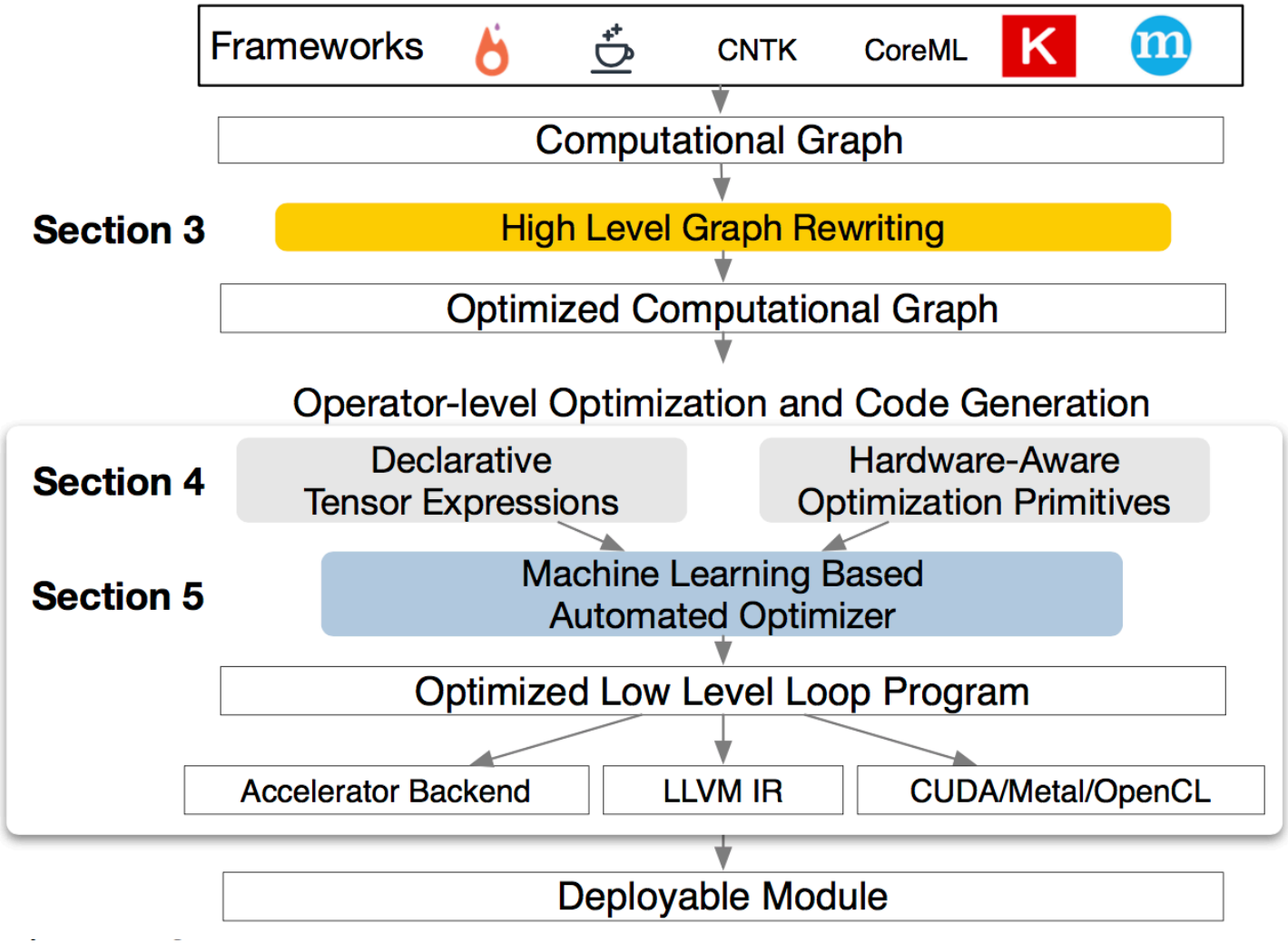


	CPU	GPU	Accelerators
Memory subsystem	L3 L2 L2 L1D L1I L1D L1I *implicitly managed*	L2 SM SM TX/L1 TX/L1 RF RF RF RF *mixed*	Unified Buffer FIFO Acc *explicitly managed*
Compute primitives	*scalar*	*vector*	*tensor*
Data type	`fp32`	`fp16`	`int8`

Challenge: Producing efficient code

- Huge configuration space
  - Memory access
  - Threading pattern
  - Loop tiles and ordering
  - Caching
  - Unrolling
  - …

Overview


Frameworks
CNTK
CoreML
Computational Graph
Section 3
High Level Graph Rewriting
Optimized Computational Graph
Operator-level Optimization and Code Generation
Section 4
Declarative
Tensor Expressions
Hardware-Aware
Optimization Primitives
Section 5
Machine Learning Based
Automated Optimizer
Optimized Low Level Loop Program
Accelerator Backend
LLVM IR
CUDA/Metal/OpenCL
Deployable Module

End-User Example

Deploy:

```
import tvm as t
# Use keras framework as example, import model
graph, params = t.frontend.from_keras(keras_model)
target = t.target.cuda()
graph, lib, params = t.compiler.build(graph, target, params)
```

Runtime:

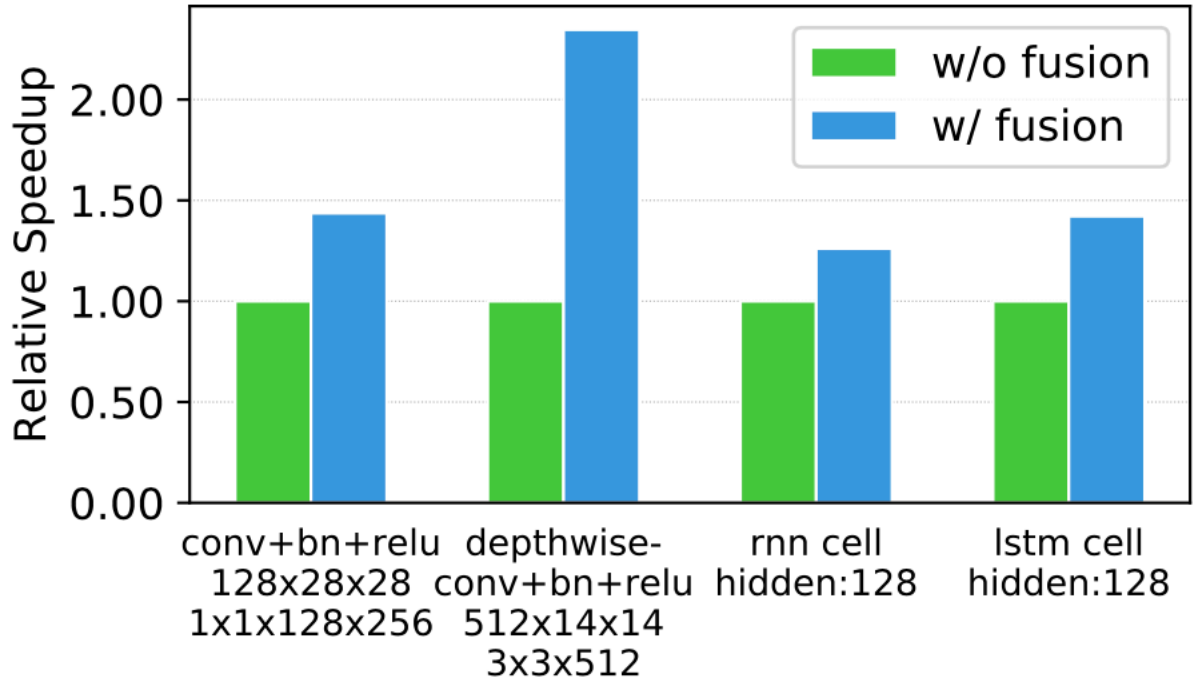
```
import tvm.runtime as t
module = runtime.create(graph, lib, t.cuda(0))
module.set_input(**params)
module.run(data=data_array)
output = tvm.nd.empty(out_shape, ctx=t.cuda(0))
module.get_output(0, output)
```

Optimizing Computational Graphs

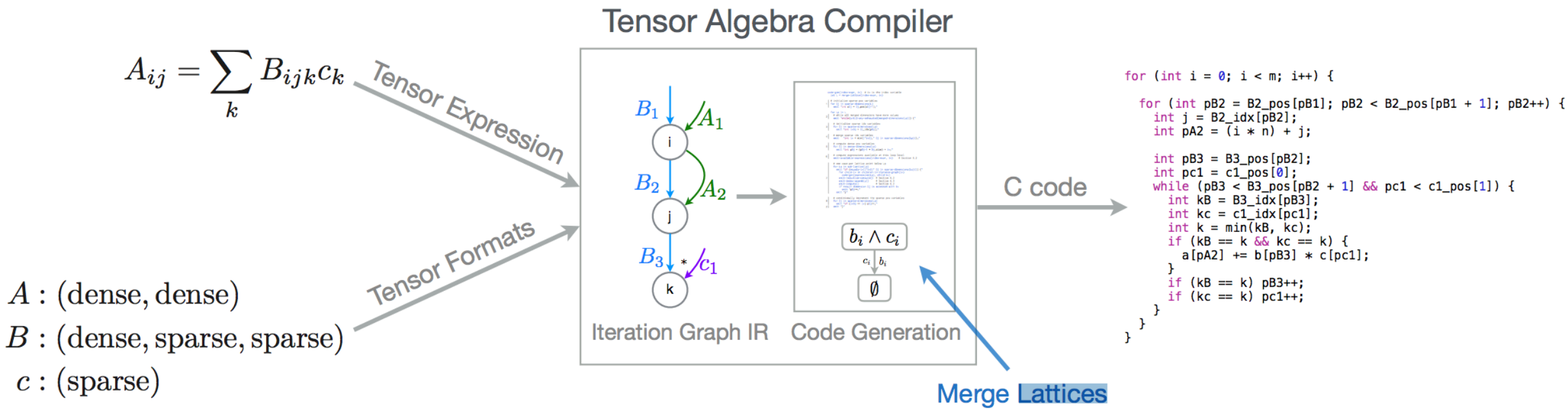
- Operator Fusion
  - Injective (one-to-one map, e.g. add)
  - Reduction (e.g. sum)
  - Complex-out-fusable (can fuse element-wise map to output, e.g. conv2d)
  - Opaque (cannot be fused, e.g. sort)
- Data Layout Transformation

Optimizing Computational Graphs

- Operator Fusion



The Tensor Algebra Compiler (taco)




```
for (int i = 0; i < m; i++) {

  for (int pB2 = B2_pos[pB1]; pB2 < B2_pos[pB1 + 1]; pB2++) {
    int j = B2_idx[pB2];
    int pA2 = (i * n) + j;

    int pB3 = B3_pos[pB2];
    int pc1 = c1_pos[0];
    while (pB3 < B3_pos[pB2 + 1] && pc1 < c1_pos[1]) {
      int kB = B3_idx[pB3];
      int kc = c1_idx[pc1];
      int k = min(kB, kc);
      if (kB == k && kc == k) {
        a[pA2] += b[pB3] * c[pc1];
      }
      if (kB == k) pB3++;
      if (kc == k) pc1++;
    }
  }
}
```

Halide

```
Func blur_3x3(Func input) {
  Func blur_x, blur_y;
  Var x, y, xi, yi;

  // The algorithm - no storage or order
  blur_x(x, y) = (input(x-1, y) + input(x, y) + input(x+1, y))/3;
  blur_y(x, y) = (blur_x(x, y-1) + blur_x(x, y) + blur_x(x, y+1))/3;

  // The schedule - defines order, locality; implies storage
  blur_y.tile(x, y, xi, yi, 256, 32)
        .vectorize(xi, 8).parallel(y);
  blur_x.compute_at(blur_y, x).vectorize(x, 8);

  return blur_y;
}
```

Generating Tensor Operations

- Tensor Expression and Schedule Space

```
m, n, h = t.var('m'), t.var('n'), t.var('h')
A = t.placeholder((m, h), name='A')
B = t.placeholder((n, h), name='B')
k = t.reduce_axis((0, h), name='k')
C = t.compute((m, n), lambda y, x:
      t.sum(A[k, y] * B[k, x], axis=k))
```

computing rule

result shape

```
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]
```

+ Loop Tiling

```
yo, xo, ko, yi, xi, ki = s[C].tile(y, x, k, 8, 8, 8)
```

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

+ Cache Data on Accelerator Special Buffer

```
CL = s.cache_write(C, vdla.acc_buffer)
AL = s.cache_read(A, vdla.inp_buffer)
# additional schedule steps omitted …
```

+ Map to Accelerator Tensor Instructions

```
s[CL].tensorize(yi, vdla.gemm8x8)
```

```
inp_buffer AL[8][8], BL[8][8]
acc_buffer CL[8][8]
for yo in range(128):
  for xo in range(128):
    vdla.fill_zero(CL)
    for ko in range(128):
      vdla.dma_copy2d(AL, A[ko*8:ko*8+8][yo*8:yo*8+8])
      vdla.dma_copy2d(BL, B[ko*8:ko*8+8][xo*8:xo*8+8])
      vdla.fused_gemm8x8_add(CL, AL, BL)
    vdla.dma_copy2d(C[yo*8:yo*8+8,xo*8:xo*8+8], CL)
```

schedule → schedule transformation ⇢ corresponding low-level code

Generating Tensor Operations

- Nested Parallelism with Cooperation

```
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

Generating Tensor Operations

- Tensorization

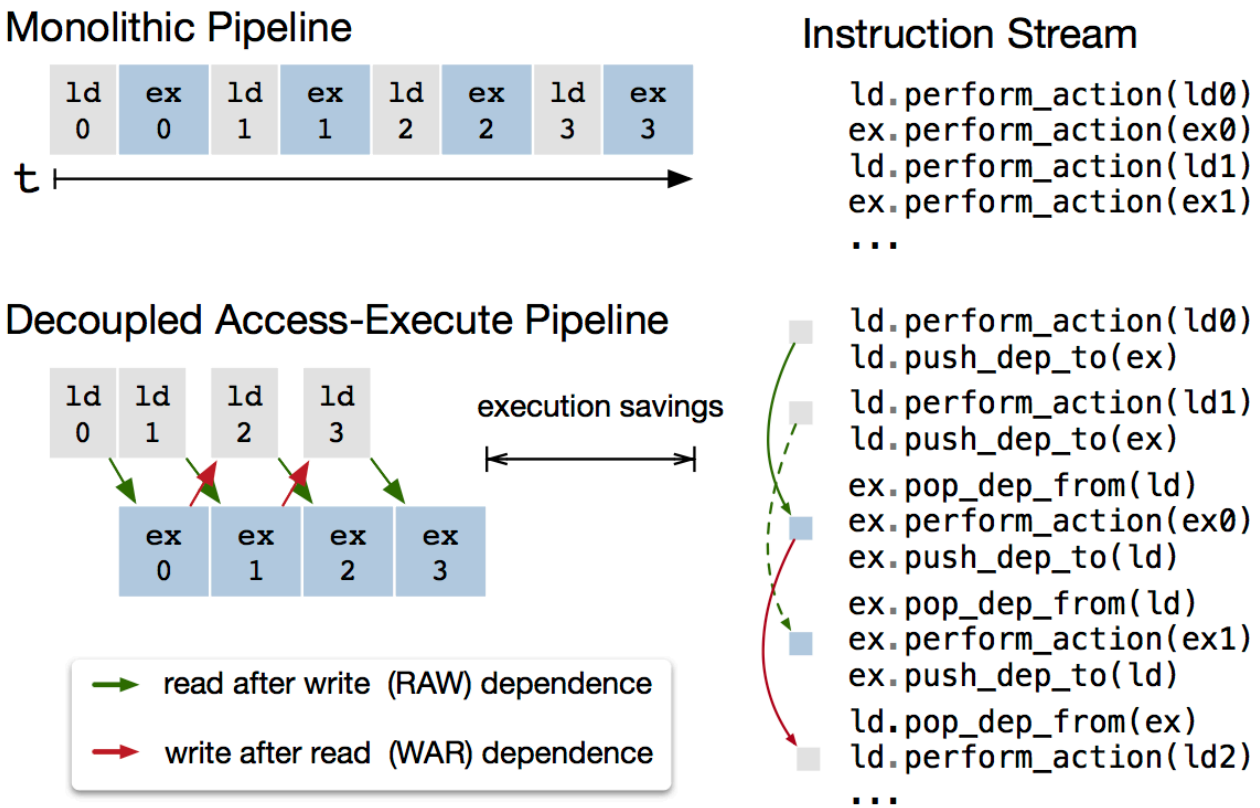
```
w, x = t.placeholder((8, 8)), t.placeholder((8, 8))
k = t.reduce_axis((0, 8))
y = t.compute((8, 8), lambda i, j:
              t.sum(w[i, k] * x[j, k], axis=k))

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

gemm8x8 = t.decl_tensor_intrin(y.op, gemm_intrin_lower)
```

declare behavior

lowering rule to generate hardware intrinsics to carry out the computation

Generating Tensor Operations

- Explicit Memory Latency Hiding



Generating Tensor Operations

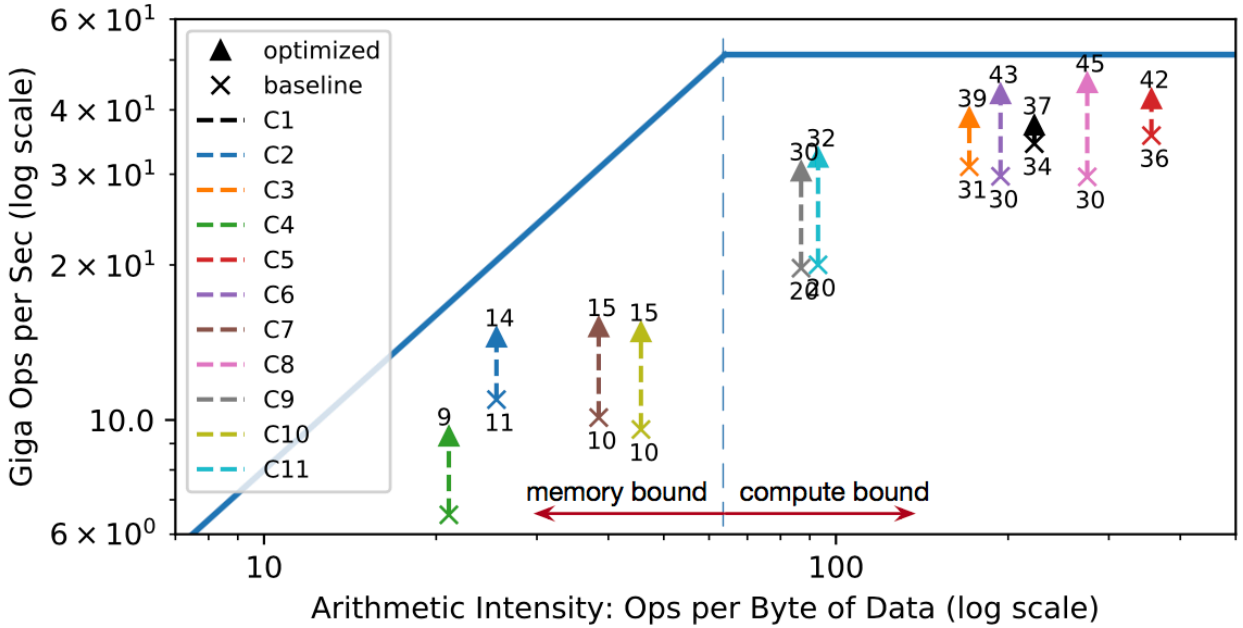


Figure 9: Roofline [42] of an FPGA-based deep learning accelerator running ResNet inference. With latency hiding enabled by TVM, the performance of the benchmarks are brought closer to the roofline, demonstrating higher compute and memory bandwidth efficiency.

Automating Optimization

- modifying the loop order
- optimizing for the memory hierarchy
- schedule- specific parameters such as the tiling size and the loop unrolling factor
- …

billions of possible configurations on the real world deep learning workloads

Automating Optimization

Method Category	Data Cost	Model Bias	Need hardware info	Learn from history
Blackbox auto-tuning	high	none	no	no
Predefined cost model	none	high	yes	no
ML based cost model	**low**	**low**	**no**	**yes**

Table 1: Comparison of automation methods. Model bias refers to inaccuracy due to modeling.

Automating Optimization

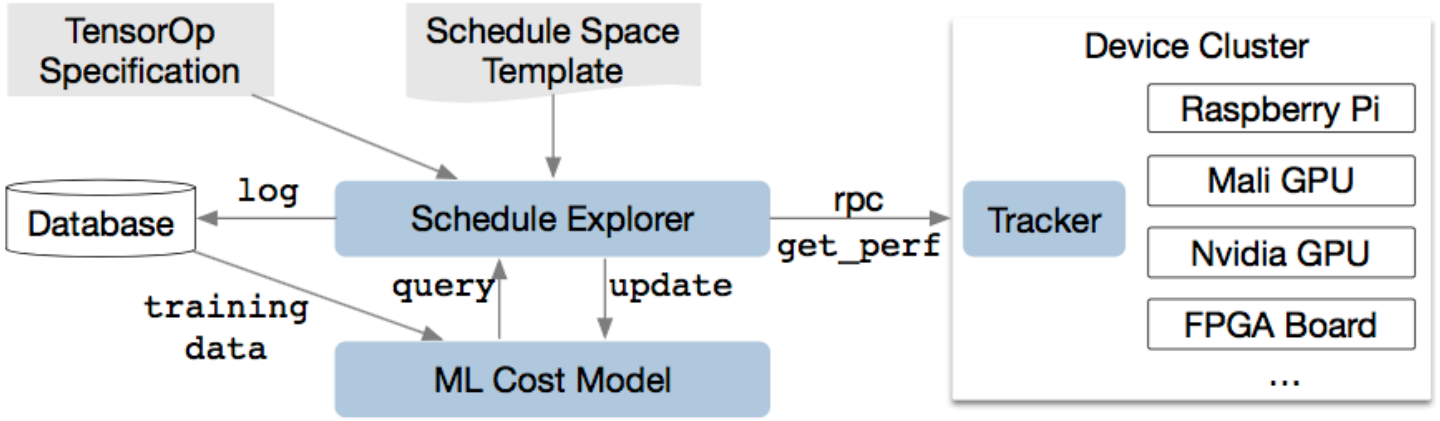


Figure 10: Overview of automated optimization framework.

Automating Optimization

- XGBoost
  - Memory access count
  - Reuse ratio of each memory buffer
  - One-hot encoding of loop annotations
    - “vecvtorize”
    - “unroll”
    - “parallel”
  - …
- TreeRNN
  - Summarize the loop program’s AST

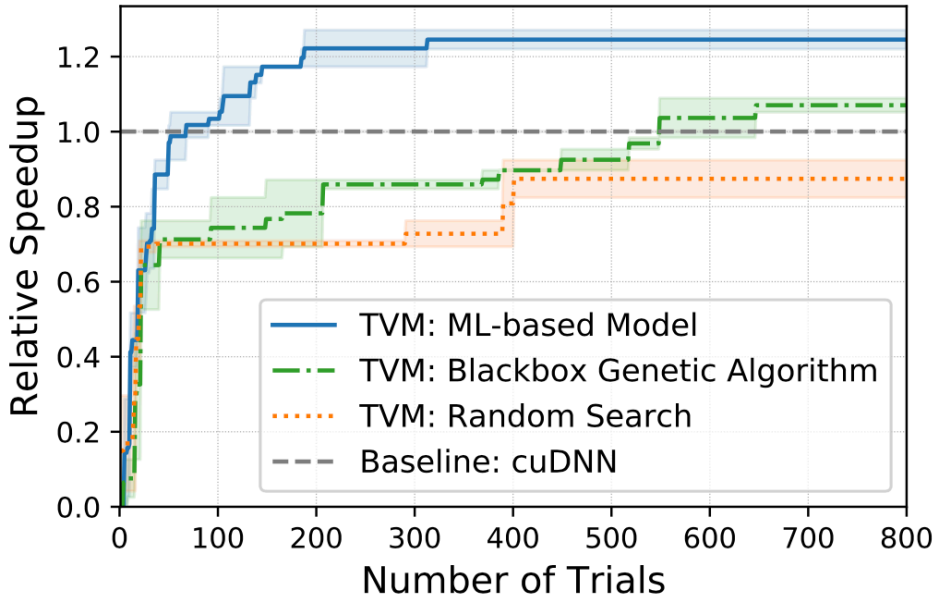


Figure 11: Comparison of different automation methods for a conv2d operator in ResNet-18 on TITAN X. The ML-based model starts with no training data and uses the collected data to improve itself. The Y-axis is the speedup relative to cuDNN. We observe a similar trend for other workloads.

Evaluation

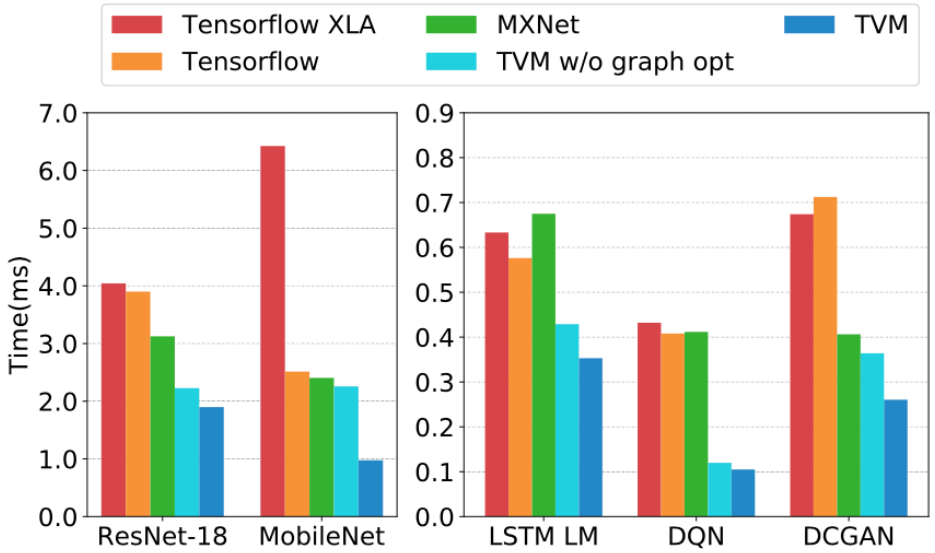


Figure 13: GPU end-to-end evaluation among TVM, MXNet, Tensorflow, and Tensorflow XLA. Tested on NVIDIA Titan X.

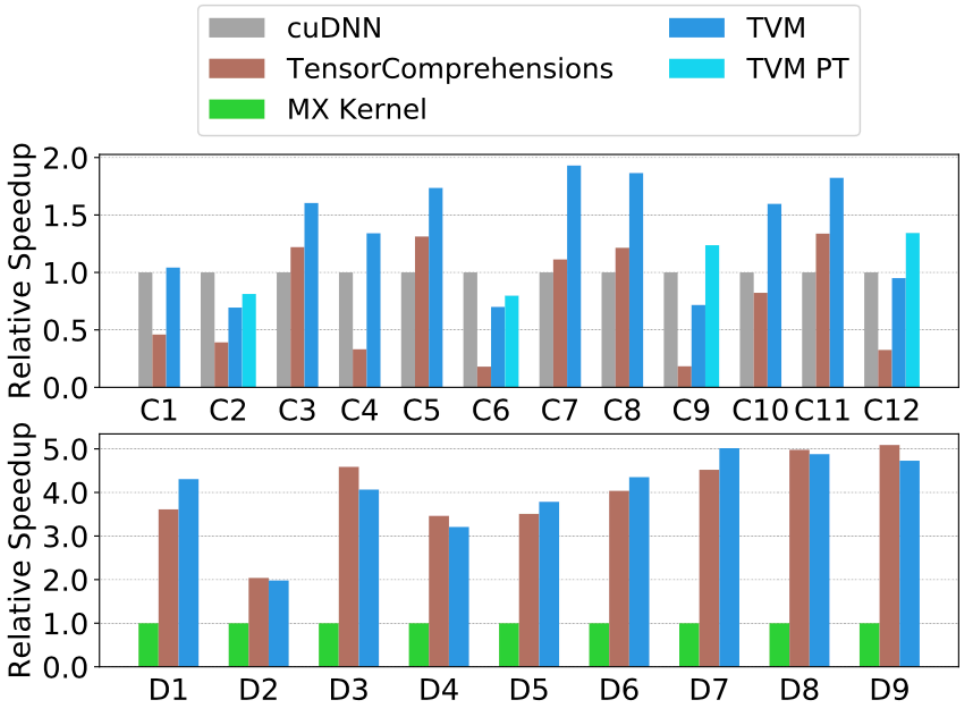


Figure 14: Relative speedup of all conv2d operators in ResNet-18 and all depthwise conv2d operators in MobileNet. Tested on TITAN X. See Table 2 for the configurations of these operators. We also include a weight pre-transformed Winograd [24] for 3x3 conv2d (TVM PT).

Evaluation

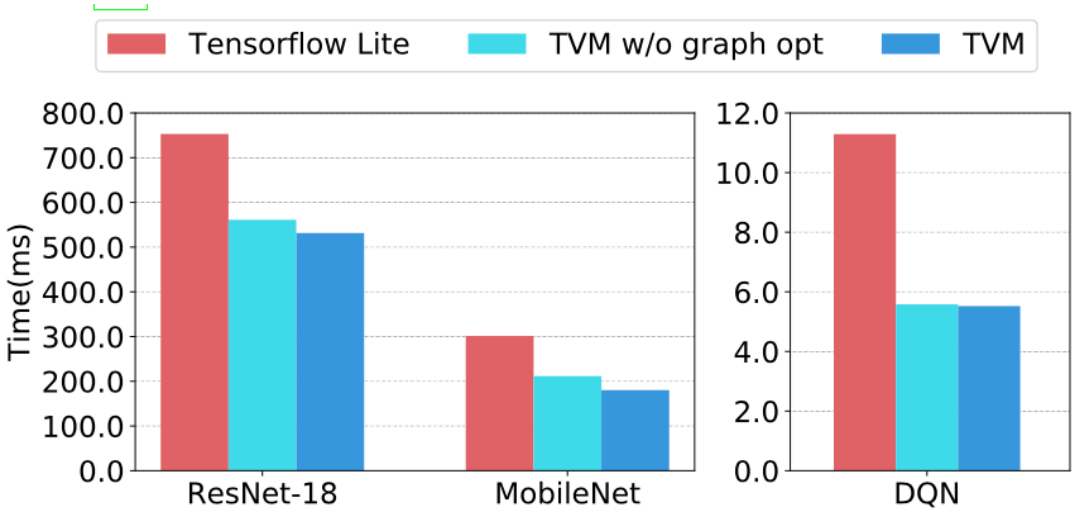


Figure 15: ARM A53 end-to-end evaluation of TVM and TFLite.

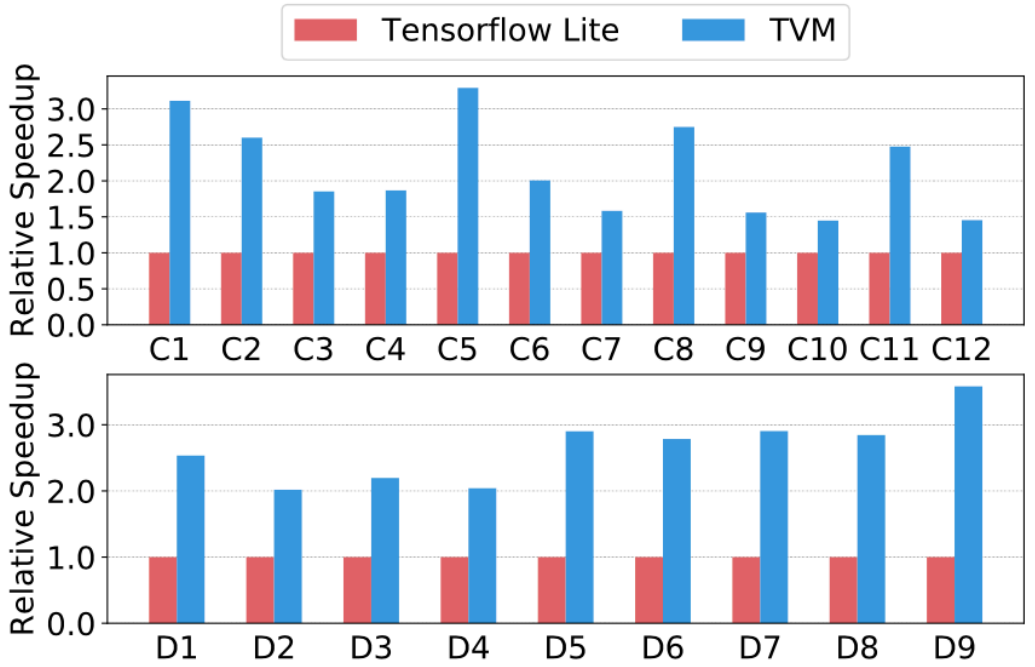
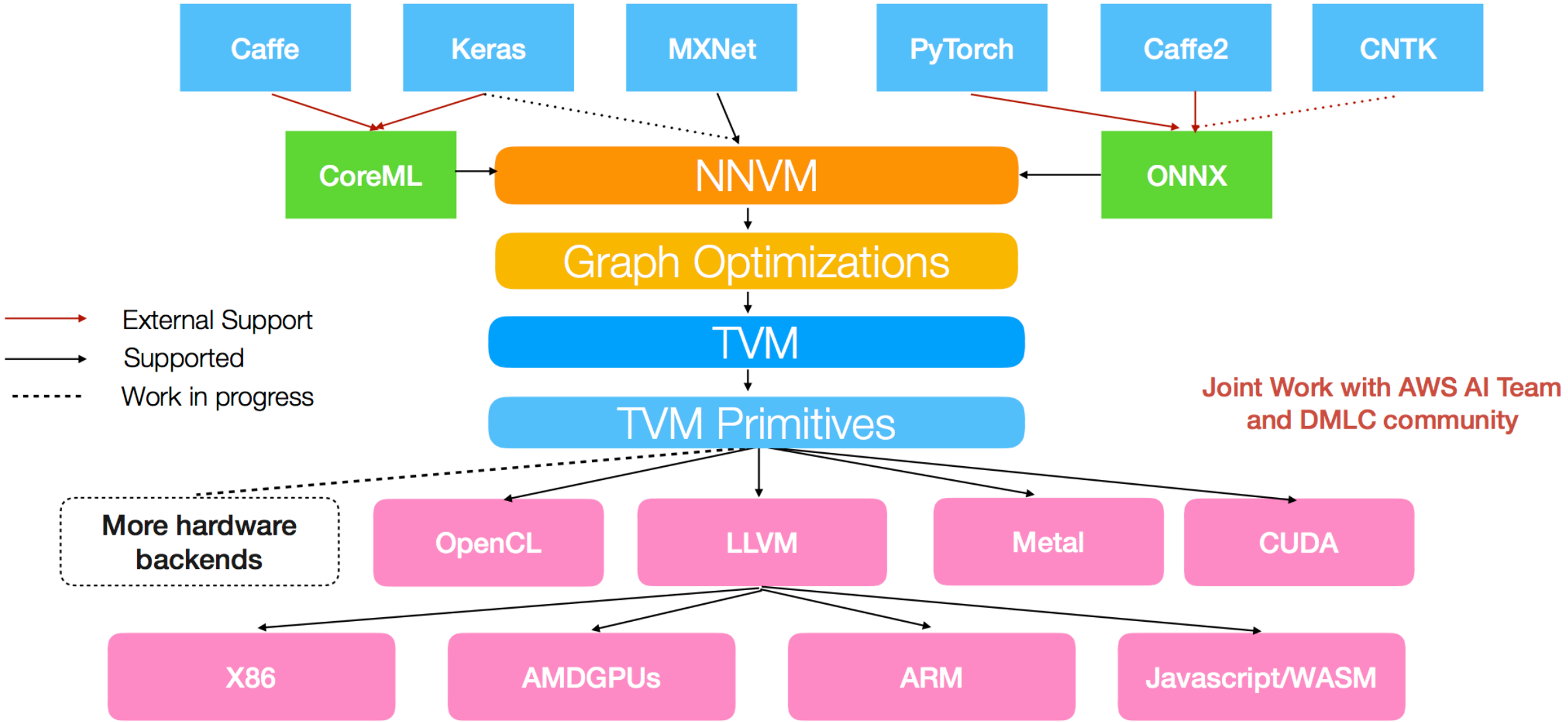


Figure 16: Relative speedup of all conv2d operators in ResNet-18 and all depthwise conv2d operators in mobilenet. Tested on ARM A53. See Table 2 for the configurations of these operators.

NNVM Compiler: Open Compiler for AI Systems



Q&A