TVM: AN AUTOMATED END-TO-END OPTIMIZING COMPILER FOR DEEP LEARNING

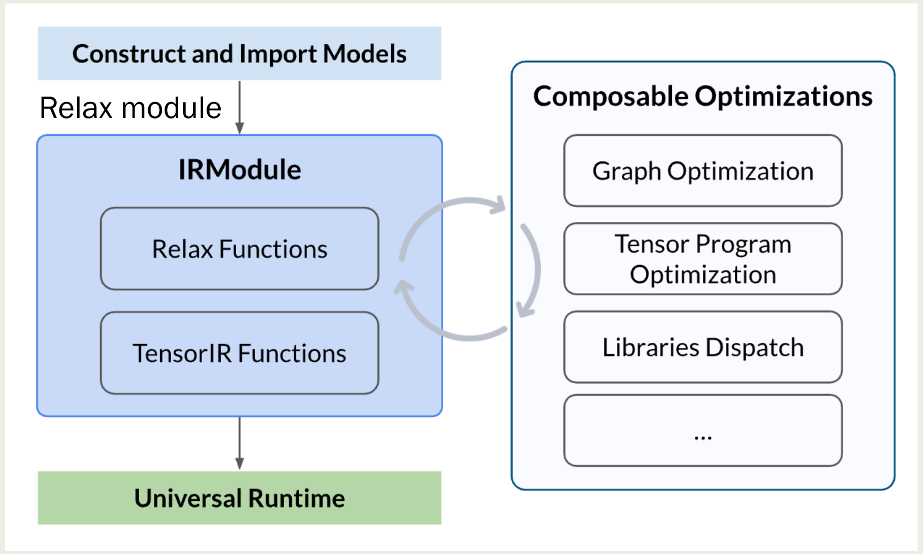
전기정보공학부 김희원

Papers read

- Chen, T., Moreau, T., Jiang, Z., Zheng, L., Yan, E., Shen, H., Cowan, M., Wang, L., Hu, Y., Ceze, L., Guestrin, C., & Krishnamurthy, A. (2018). *TVM: An automated end-to-end optimizing compiler for deep learning*. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI '18)* (pp. 579–594). USENIX Association. https://www.usenix.org/conference/osdi18/presentation/chen
- Chen, T., Zheng, L., Yan, E., Jiang, Z., Moreau, T., Ceze, L., Guestrin, C., & Krishnamurthy, A. (2018). *Learning to optimize tensor programs*. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, & R. Garnett (Eds.), *Advances in Neural Information Processing Systems* (Vol. 31, pp. 3393–3404). Curran Associates, Inc. https://doi.org/10.48550/arXiv.1805.08166

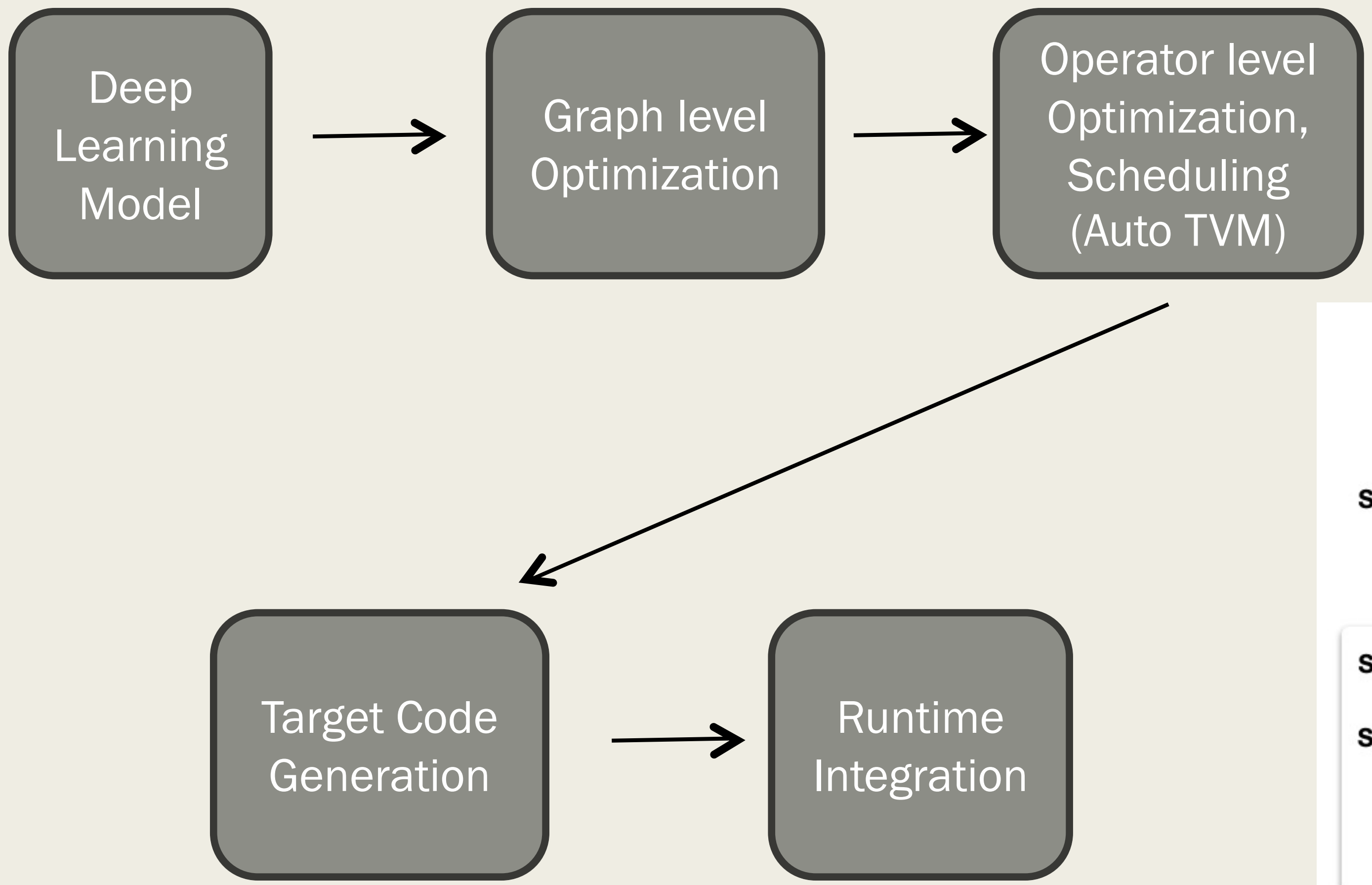
Overview

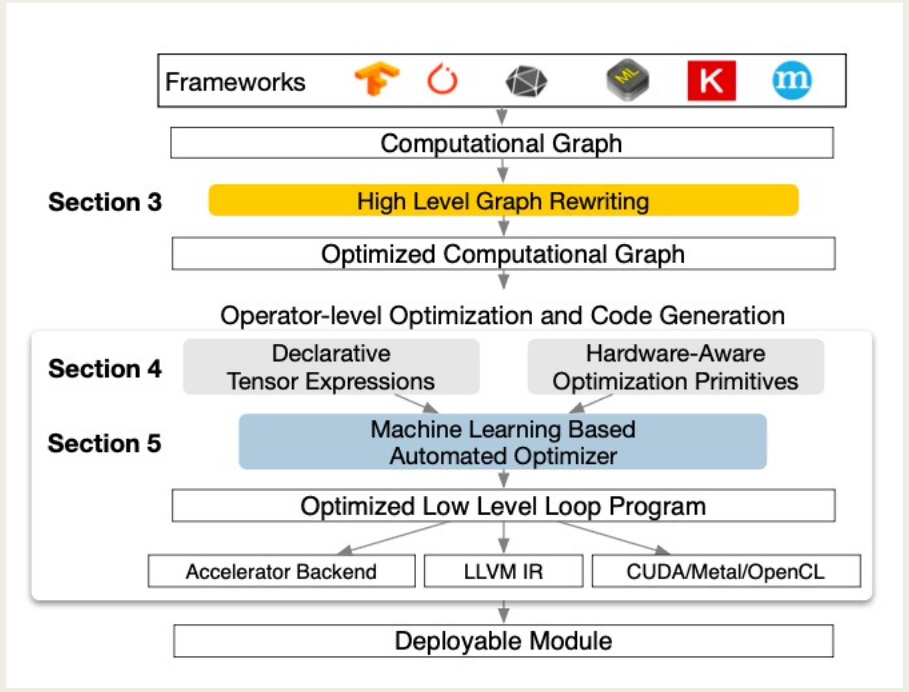
- End-to-end deep learning compiler stack
- Translates high-level models to low level optimized code
- Supports diverse backends: CPU, GPU, FPGA
- Graph-level optimization, operator-level optimization: auto-scheduling via AutoTVM



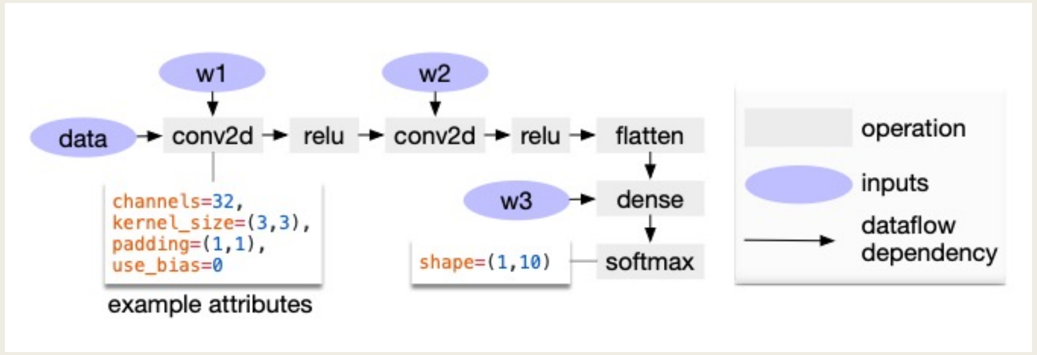
https://tvm.apache.org/docs/how_to/tutorials/e2e_opt_model.html

TVM Compiler Pipeline

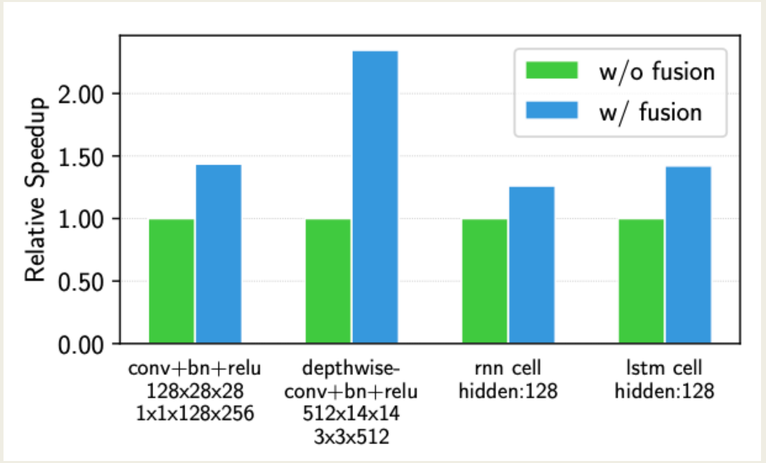



Frameworks
Computational Graph
Section 3
High Level Graph Rewriting
Optimized Computational Graph
Operator-level Optimization and Code Generation
Section 4
Declarative Tensor Expressions
Hardware-Aware Optimization Primitives
Section 5
Machine Learning Based Automated Optimizer
Optimized Low Level Loop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

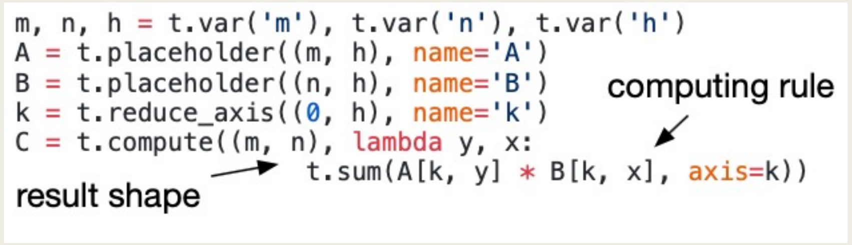
1. Graph-Level Optimizations



- **Computational Graph**
  Node = operation / Edge = dataflow dependency
- Graph-level optimization methods
  - ***Operator Fusion:*** *combine multiple operators into a single kernel*
  - ***Constant Folding:*** *precompute statically determinable values at compile-time*
  - ***Static Memory Planning:*** *pre-allocate memory for each intermediate tensor*
  - ***Data Layout Transformation:*** *reorder data in back-end-friendly forms for memory efficiency*



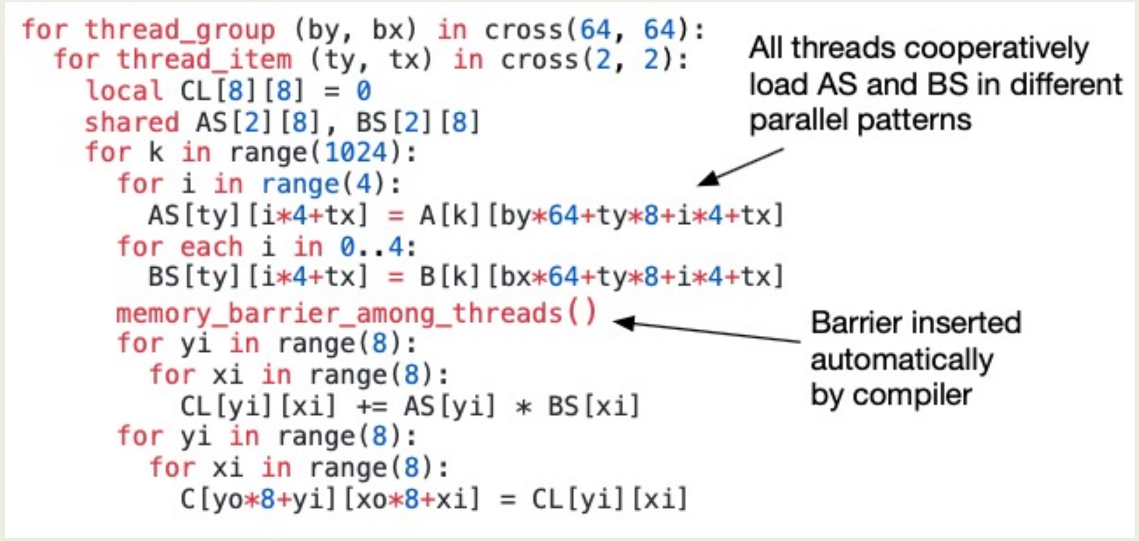
2. Operator-Level Optimizations



```
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))
```

- Tensor expression language for $C = A^T \cdot B$:
- Doesn’t specify execution details such as loop structure- separates scheduling from computation

- *1) Nested parallelism with cooperation*: data parallel tasks are parallelized. **“Shared-nothing nested parallelism”**: not shared between working thread siblings.



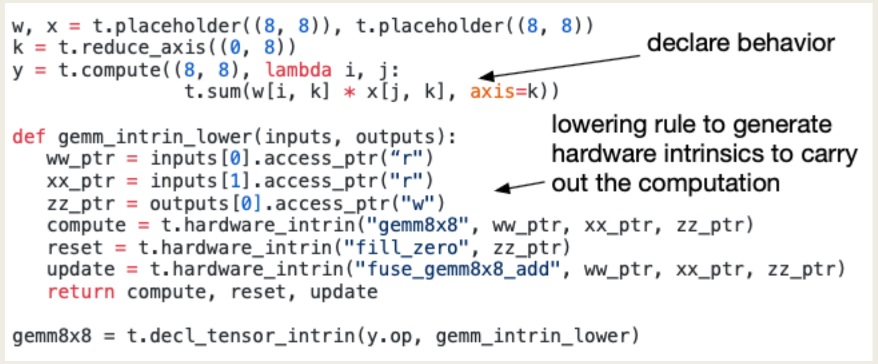
```
for thread_group (by, bx) in cross(64, 64):
  for thread_item (ty, tx) in cross(2, 2):
    local CL[8][8] = 0
    shared AS[2][8], BS[2][8]
    for k in range(1024):
      for i in range(4):
        AS[ty][i*4+tx] = A[k][by*64+ty*8+i*4+tx]
      for each i in 0..4:
        BS[ty][i*4+tx] = B[k][bx*64+ty*8+i*4+tx]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

AS, BS: shared arrays

2. Operator-Level Optimizations

- *2) Tensorization*:
  - Computations → Tensor operations
  - *TVM made tensorization <u>extensible by separating</u> the target hardware intrinsic (instructions) from the schedule.*

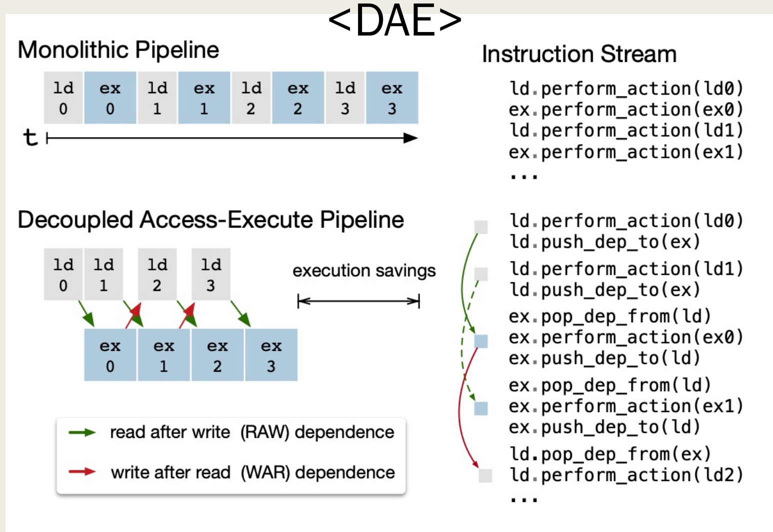


```
w, x = t.placeholder((8, 8)), t.placeholder((8, 8))
k = t.reduce_axis((0, 8))
y = t.compute((8, 8), lambda i, j:
                    t.sum(w[i, k] * x[j, k], axis=k))

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

- *3) Explicit Memory Latency Hiding:* The process of <u>overlapping memory operations with computation</u> to maximize utilization of memory and compute resources.
  - *CPU:* simultaneous multithreading or hardware prefetching
  - *GPU*: rapid context switching of many warps of threads
  - *TPU*: decoupled access-execute (DAE) architecture

<DAE>



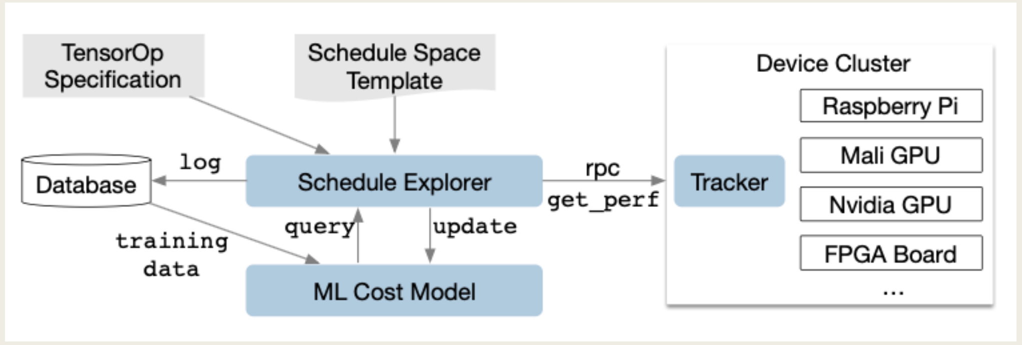
Monolithic Pipeline
ld 0
ex 0
ld 1
ex 1
ld 2
ex 2
ld 3
ex 3
t
Decoupled Access-Execute Pipeline
execution savings
read after write (RAW) dependence
write after read (WAR) dependence
Instruction Stream

```
ld.perform_action(ld0)
ex.perform_action(ex0)
ld.perform_action(ld1)
ex.perform_action(ex1)
...
```

```
ld.perform_action(ld0)
ld.push_dep_to(ex)
ld.perform_action(ld1)
ld.push_dep_to(ex)
ex.pop_dep_from(ld)
ex.perform_action(ex0)
ex.push_dep_to(ld)
ex.pop_dep_from(ld)
ex.perform_action(ex1)
ex.push_dep_to(ld)
ld.pop_dep_from(ex)
ld.perform_action(ld2)
...
```

2. Operator-Level Optimizations- Auto Scheduling via AutoTVM

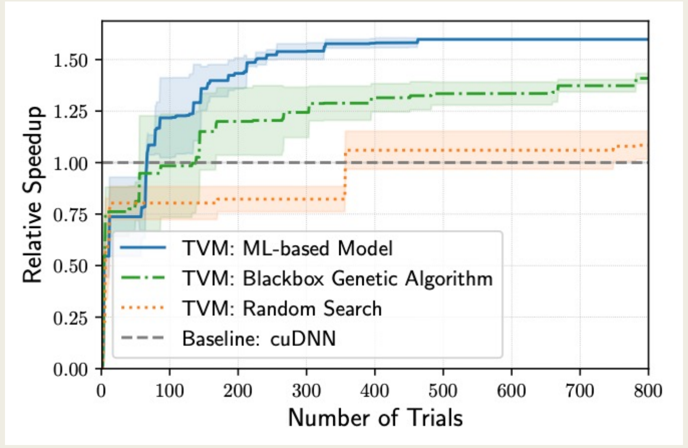
- **Motivation:**
  - Systems relied on manually optimized libraries (ex) cuDNN) → significant engineering cost was needed to deploy them to new hardware targets.
  - Use learning to automatically optimize operators.

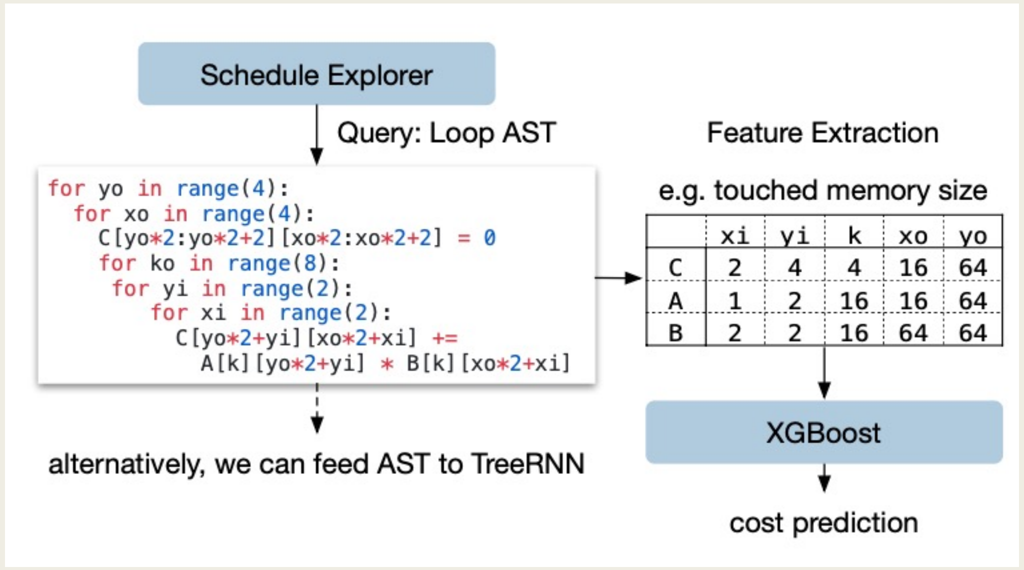


- **1) Schedule Space Specification**
  - Considers as many configurations as possible (billions)
  - Developers can declare knobs themselves,
    or, a generic master template for each hardware back-end is created beforehand

2. Operator-Level Optimizations- Auto Scheduling via AutoTVM

- 2) ML-Based Cost Model
  - Statistical approach instead of considering all factors affecting performance
  - The model is updated experimentally as we explore more configurations.
  - The gradient tree boosting model (based on XGBoost) does prediction in 0.67ms (on average), which was the fastest among different automation methods for a conv2d operator in ResNet-18 on TITAN X.





```
for yo in range(4):
  for xo in range(4):
    C[yo*2:yo*2+2][xo*2:xo*2+2] = 0
    for ko in range(8):
     for yi in range(2):
        for xi in range(2):
          C[yo*2+yi][xo*2+xi] +=
            A[k][yo*2+yi] * B[k][xo*2+xi]
```

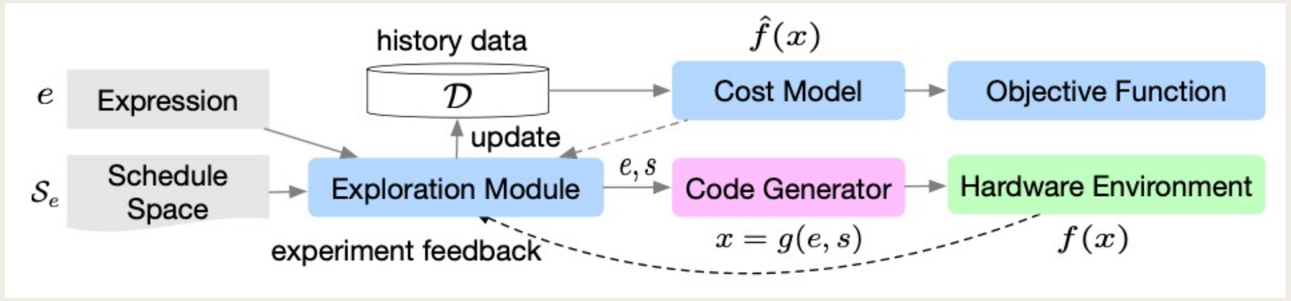
	xi	yi	k	xo	yo
C	2	4	4	16	64
A	1	2	16	16	64
B	2	2	16	64	64

2. Operator-Level Optimizations- Auto Scheduling via AutoTVM

- **3) Schedule Exploration**
  - Runs a <u>parallel Simulated Annealing (SA)</u> algorithm for large search spaces.
    - Randomly walks to a nearby configuration:
    - Accepts if cost decreases.
    - Rejects with some probability if cost increases.

- **4) Distributed Device Pool**
  - Implements an RPC-based <u>distributed device pools</u> for optimization.

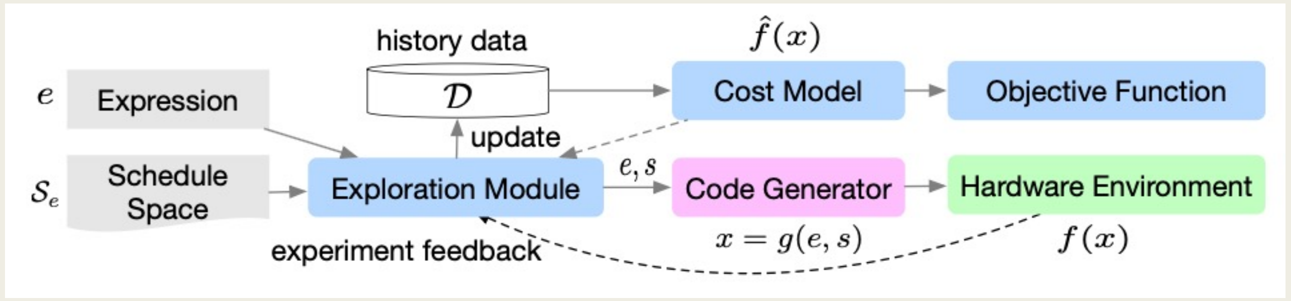
AutoTVM in detail



- **1) Statistical Cost Model** ($\hat{f}(x) \approx f(x)$ for a low-level program x)
  - *Extract features*
    - Loop Structure Information: memory access count, data reuse ratio…
    - Generic Annotations: vectorization, unrolling, thread binding…
  - ***i) XGBoost***: requires precise feature engineering + fast prediction
  - ***ii) TreeGRU:*** feature engineering not required + training speed is slow → used with batching and GPU

- 2) Training Objective Function
  - Run time statistics data $D = \{e_i, s_i, c_i\}$
  - Rank loss function $\sum_{i,j} \log\left(1 + e^{-sign(c_i - c_j)\left(\hat{f}(x_i) - \hat{f}(x_j)\right)}\right)$.
    Used because <u>relative</u> order of program run time is important

AutoTVM in detail



- 3) Exploration Module
  - Select the top-performing batch of candidates to update $\hat{f}$.
  - Simulated Annealing

- 4) Accelerating Optimization via Transfer Learning
  - The cost model is shared using the common transferable representation (invariant to the source/target domains)
  - Transferable representation:
    - GBT: Z: context feature matrix such that $Z_{ik} = i - th\ feature\ of\ loop\ k$
      Relation feature between features i, j: $R_t^{(ij)} = \max_{k \in \{k | Z_{kj} < \beta_t\}} Z_{ki}$
    - TreeGRU: $\hat{f}(x) = \hat{f}^{(global)}(x) + \hat{f}^{(local)}(x)$
      $\hat{f}^{(global)}$ is traind on D′ (= historical data from previously seen workloads)

Comparison to Other Systems

- XLA:
  - *Focuses on <u>graph-level optimizations</u>: too high-level*
- Halide:
  - *Introduced the idea of <u>separating computing and scheduling</u>*
    → TVM adopted this + focuses on new scheduling challenges of DL workloads
- ATLAS, FFTW:
  - *Autotuning (Single kernel, predefined parameters)*
- Tensor Comprehension:
  - *Black-box auto-tuning + polyhedral optimizations via Genetic Algorithm*

Evaluation

- Evaluated TVM on 4 types of platforms:
  - *1) server-class GPU, 2) embedded GPU, 3) embedded CPU, 4) FPGA SoC*
- Benchmarks:
  - *ResNet, MobileNet, LSTM Language Model, Deep Q Network(DQN), Deep Convolutional Generative Adversarial Networks(DCGAN)*
- Compared with:
  - *Existing DL frameworks such as MxNet, TensorFlow*

Evaluation

- 1) Server-class GPU (Nvidia Titan X):
  - **End-to-end performance:** TVM outperforms, due to joint graph optimization + Auto TVM
  - **Operator-level optimization performance:** TVM mostly outperforms, due to large schedule space + ML-based search

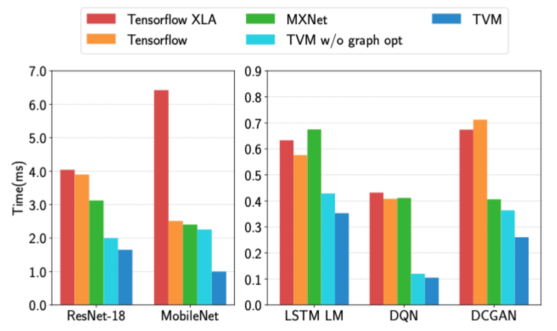


Figure 14: GPU end-to-end evaluation for TVM, MXNet, Tensorflow, and Tensorflow XLA. Tested on the NVIDIA Titan X.

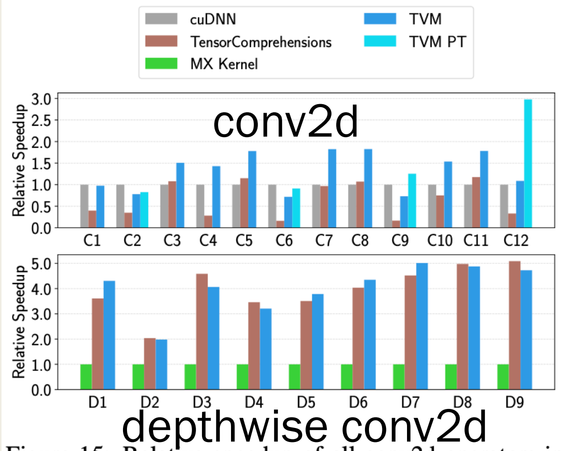


Figure 15: Relative speedup of all conv2d operators in ResNet-18 and all depthwise conv2d operators in MobileNet. Tested on a TITAN X. See Table 2 for operator configurations. We also include a weight pretransformed Winograd [25] for 3x3 conv2d (TVM PT).

- 2) Embedded CPU (ARM Cortex A53, Quad Core):
  - **End-to-end performance:** TVM outperforms
  - **Operator-level optimization performance:** TVM outperforms hand-optimized versions

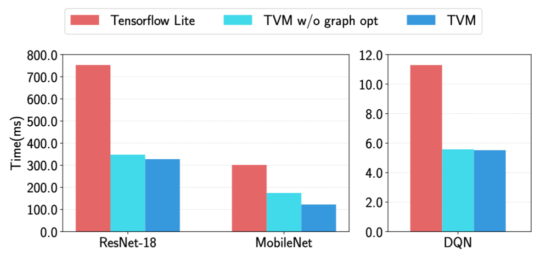


Figure 16: ARM A53 end-to-end evaluation of TVM and TFLite.

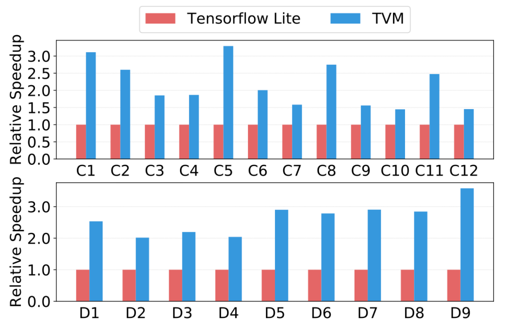
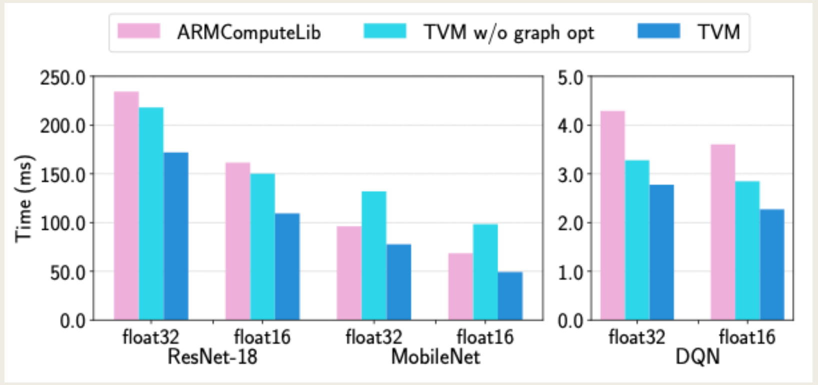


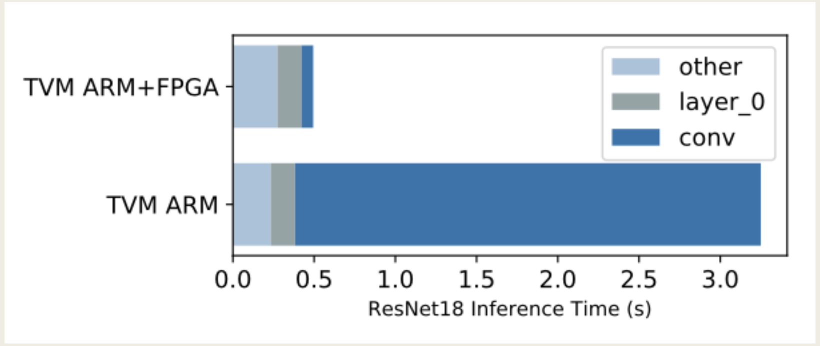
Figure 17: Relative speedup of all conv2d operators in ResNet-18 and all depthwise conv2d operators in mobilenet. Tested on ARM A53. See Table 2 for the configurations of these operators.

Evaluation

- **3) Embedded GPU (ARM Mali-T860MP4 GPU on Firefly-RK3399 board):**
  - **End-to-end performance:** TVM <u>outperforms</u>



- **4) FPGA (Vanilla Deep Learning Accelerator-VDLA):**
  - TVM <u>outperforms</u> by 40x speedup.
  - The overall performance wasn't estimated since some parts were executed using CPU (gray parts).



Conclusion

- TVM proposed an end-to-end compilation stack for DL across diverse HW backends.
- Graph-level optimization + Operator-level optimization (Auto TVM)
- TVM is evaluated to outperform existing benchmarks on diverse backend platforms.