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

---

13th Symposium on Operating Systems Design and Implementation (OSDI), 2018

Presenter: Constant (Sang-Soo) Park

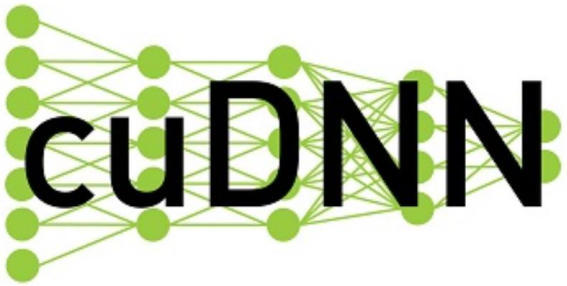
http://esoc.hanyang.ac.kr/people/sangsoo_park/index.html

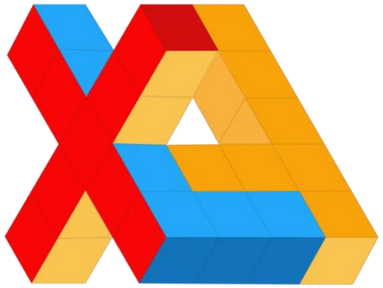
Feb 25, 2021

DL_Compiler Study

Abstract

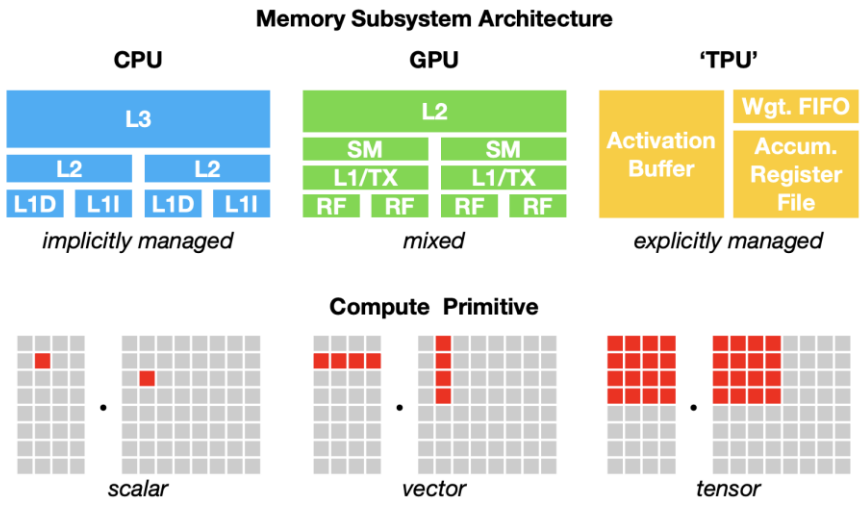
- **Machine learning to wide diversity of hardware devices**
  - Current DL frameworks rely on vendor-specific library, optimization
  - New platform (e.g., FPGA, ASIC) requires **laborious manual effort**
  - TVM is an end-to-end optimization stack that exposes (Graph, Operator-level optimizations)



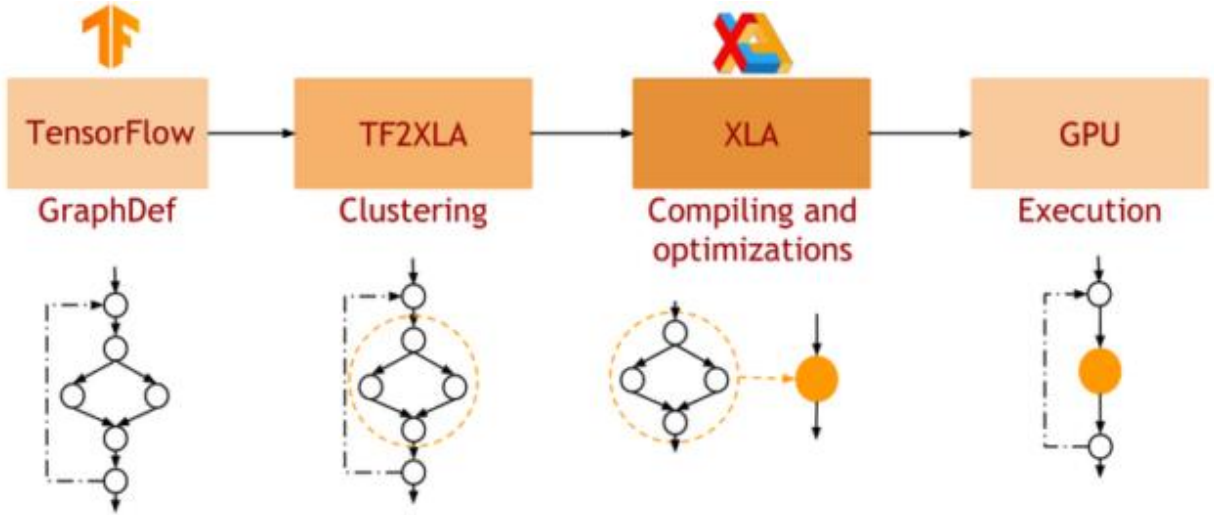


Introduction

- **DL accelerators posing adoption challenge**
  - Diverse hardware characteristics (Memory organization, Compute functional units, etc.,)
  - Present ad-hoc fashion (DL frameworks for various HW back-ends)
  - Easily deploy DL workloads to all kinds of hardware targets (embedded, GPUs, FPGAs)
  - Exposing optimization opportunities across both graph/operator-level



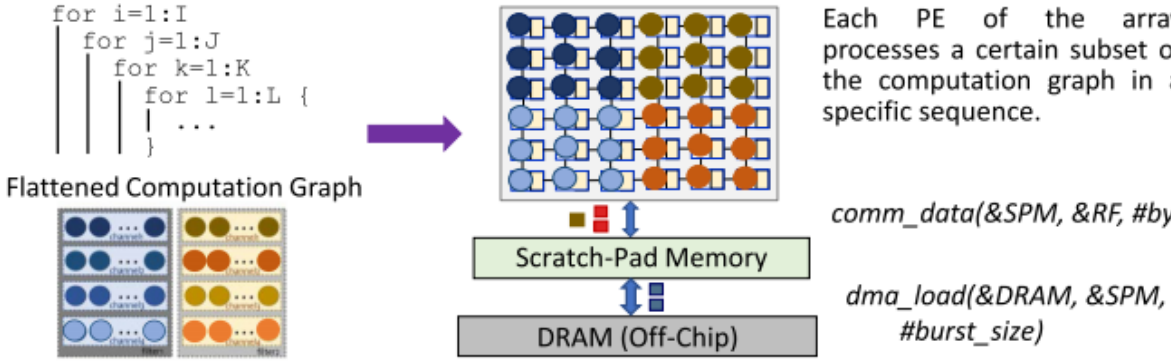
Diverse hardware



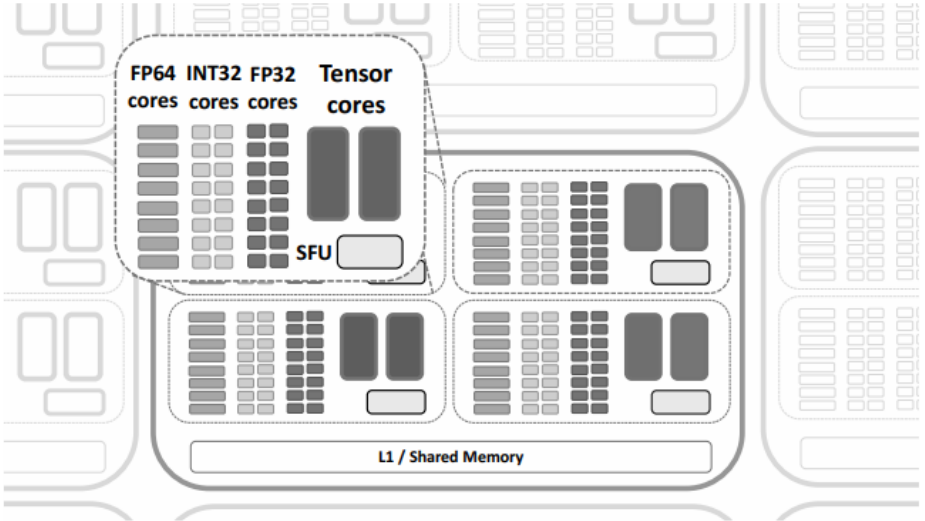
Graph optimization flow in XLA

Fundamental challenges

- **In term of graph level and tensor operator level**
  - High-level dataflow: Strategies to fuse operators and optimize data layouts (Memory access)
  - Memory reuse across threads: Shared memory in GPU
  - Tensorized compute intrinsic: Instructions for vector operations like the GEMM operator in TPU
  - Latency hiding: Hiding memory access latency



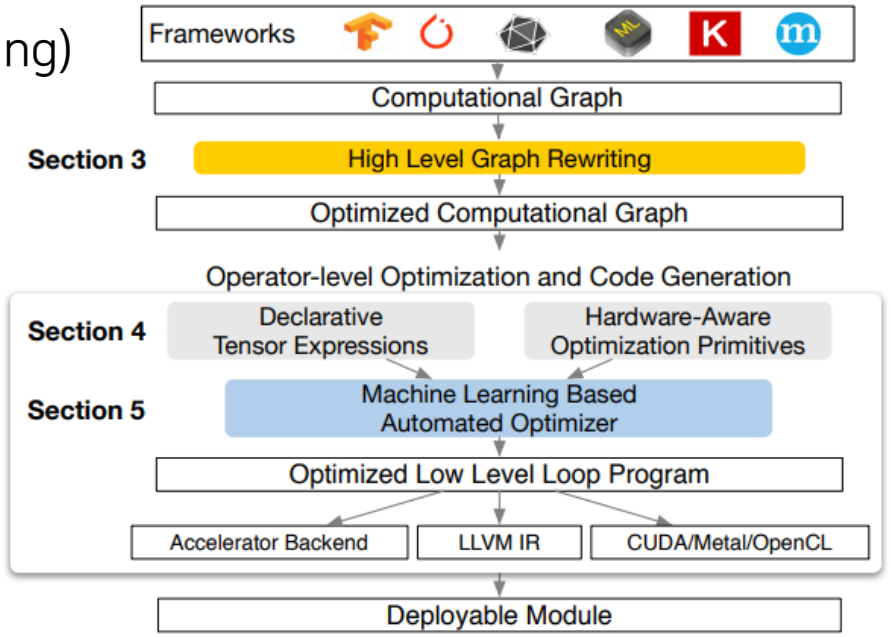
Dataflow and memory mapping[1]



HW diagram of Volta SM architecture[2]

TVM: An End-to-End Optimization Stack

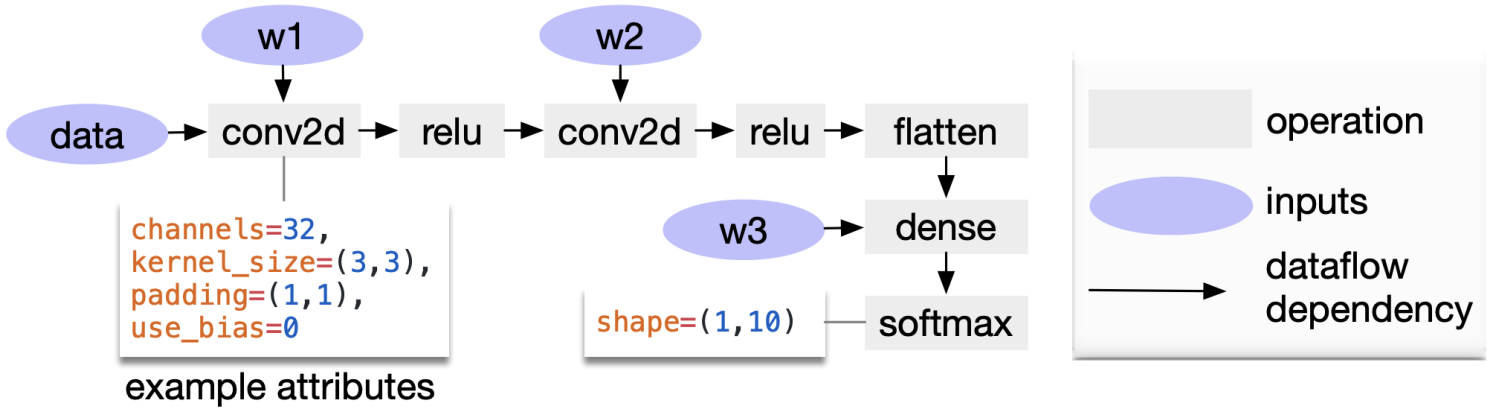
- **End-to-end optimizing compiler stack**
  - To lower and fine-tune DL workloads to diverse HW back-ends
  - Designed to separate: algorithm description, schedule, hardware interface
  - Computation graph optimization layer (High-level dataflow)
  - Tensor optimization layer (Memory reuse, intrinsic, latency hiding)



System overview of TVM

Optimizing Computational Graphs

- **Operation fusion**
  - Combining multiple operators into single kernel w/o intermediate memory
  - 1.2x~2x speedup by reducing memory accesses
  - Injection (one-to-one map), Reduction (Summation)
  - Complex-out-fusable (can fuse element-wise map to output), Opaque (Can not be fused)



Graph representation in TVM

Optimizing Computational Graphs

- **Graph operators**
  - Injective (one-to-one map), reduction
  - Complex-out-fusable (can fuse element-wise map to output)
  - Opaque (can not be fused)

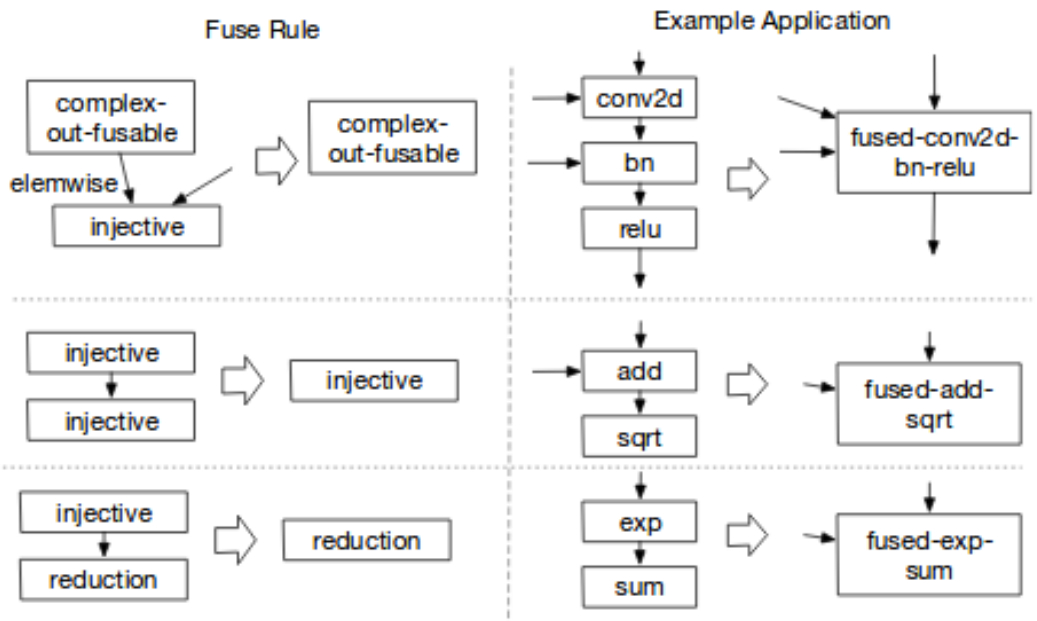


Figure 5: Rules for operation fusion pass.

Optimizing Computational Graphs

- **Data layout transformation**
  - Converting computational graph to internal data layouts for fast execution
  - Preferred data layout for each operator given the constraints dictated by memory hierarchies
  - Not feasible to handcraft operator kernels for various operations desired for each back-end

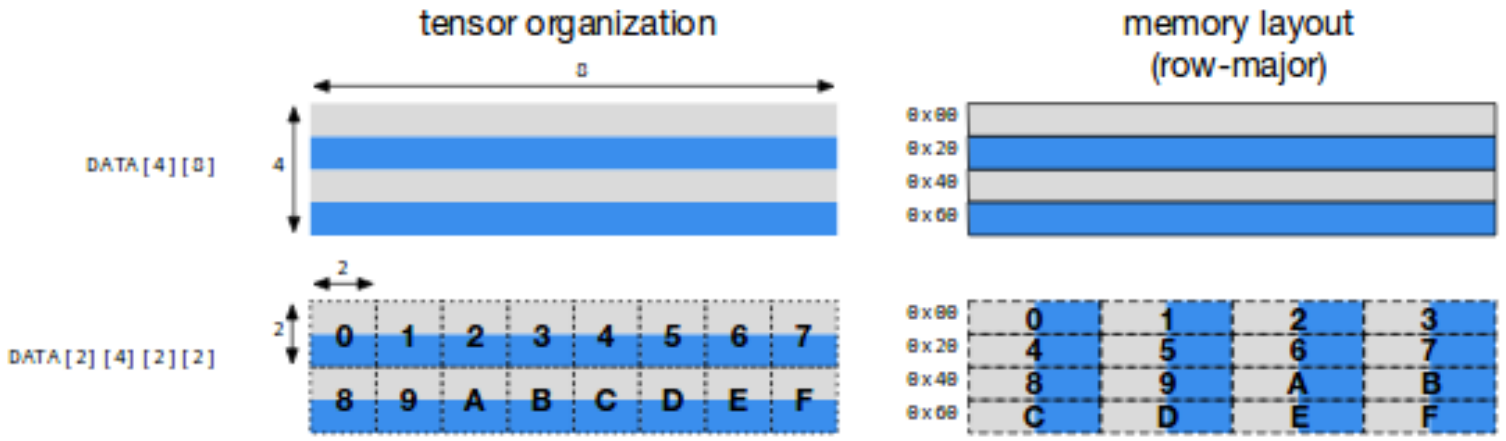


Figure 7: Data layout requirements can be affected by tensorization: here a 2×2 tensorized operation dictates a data layout transformation.

Optimizing Computational Graphs

- **Limitations**
  - Effective when operator library is provided
  - Limited operator fusion with only supporting implantations of fused patterns by HW vendors
  - Not feasible to handcraft operator kernels for massive space of back-end specific operators

Optimizing Tensor Operations

- **Schedule space**
  - Challenging to create high-performance implementations for each hardware back-end
  - Each optimized low-level program is the result of different combinations of scheduling strategies
  - AutoTVM: principle of decoupling compute descriptions from schedule optimizations
  - Schedule primitives: split, tile, fuse, reorder, bind, etc.,

```
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

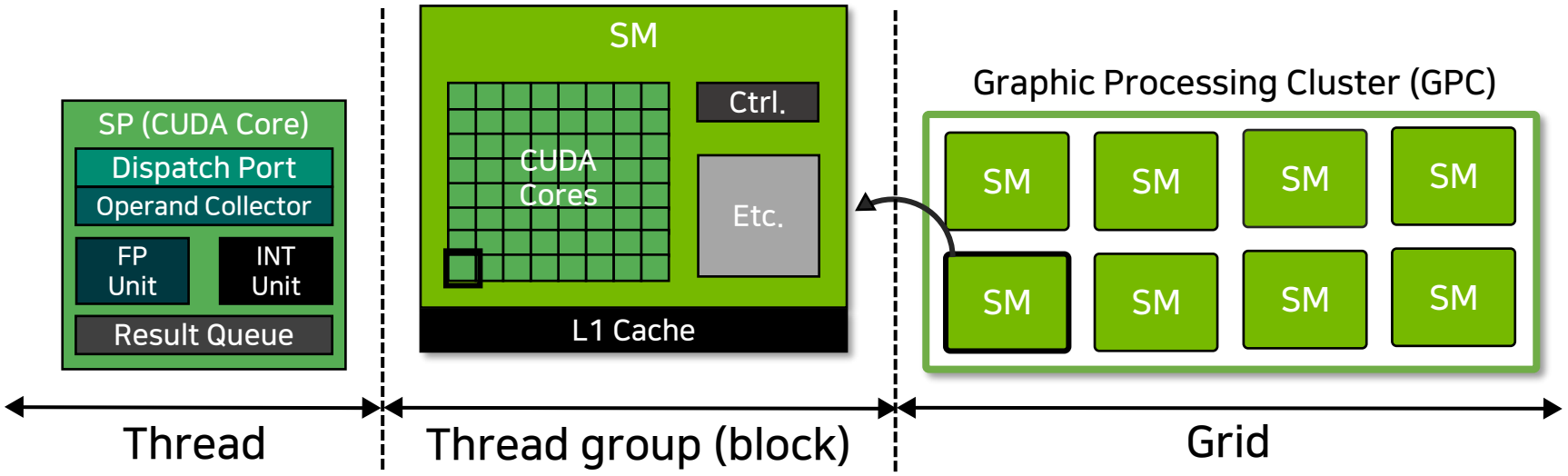
```
s[CL].tensorize(yi, vdla.gemm8x8)
```

```
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 transformation (Default, Loop tiling, Tensor instruction)

Optimizing Tensor Operations

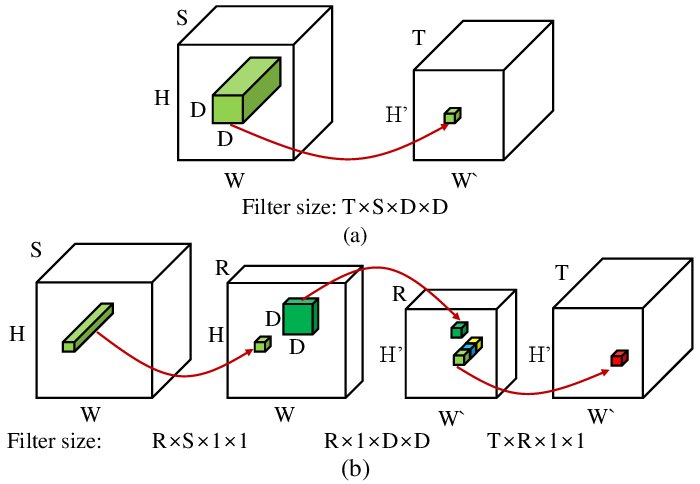
- **Nested parallelism with cooperation**
  - Nested parallelism: fork-join problem (Recursively subdivided into subtasks)
  - Shared-nothing nested parallelism, To fetch data cooperatively with GPU optimization
  - Subdivided into subtasks to exploit the target architecture's multi-level thread hierarchy



GPU architecture with multi-level thread hierarchy[3]

Optimizing Tensor Operations

- **Tensorization**
  - Analogous to vectorization for SIMD architecture, but significant different
  - Tensor operators like matrix-matrix multiplication or 1D convolution
  - Vectorization (Fixed length), Tensorization (Multi-dimension, Variable length/data layouts)
  - Extensible solution: N×N tensor hardware intrinsic (Handcraft micro-kernel)




```
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)
```

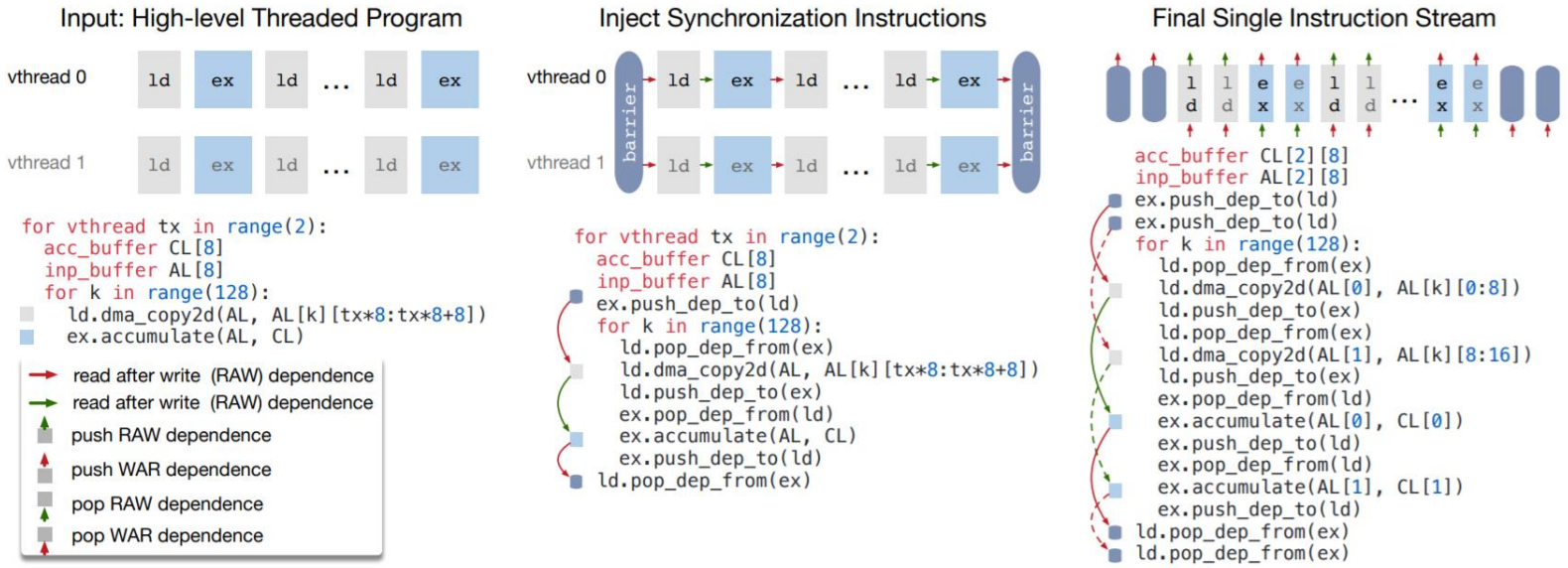
declare behavior

lowering rule to generate hardware intrinsics to carry out the computation

Tensorization (Left: Variable tensors, Light: 8×8 micro tensor hardware intrinsic)

Optimizing Tensor Operations

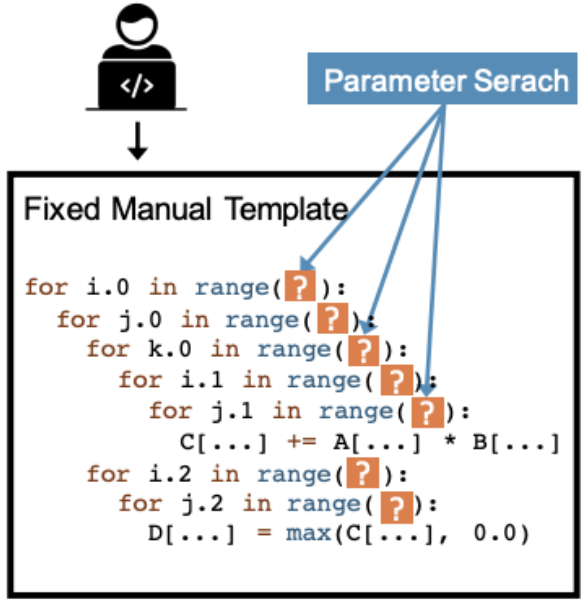
- **Explicit memory latency hiding**
  - Overlapping memory operations with computation to maximize utilization
  - CPU (Multi-threading), GPU (Context switching of warps), TPU (Decoupled access-execute, DAE)



Memory latency hiding with Virtual threads

Automating Optimization

- **Schedule space specification**
  - Schedule template specification API to declare knobs in schedule space
  - Template specification: incorporation of development's domain-specific knowledge
  - Manual definition optimization and turning search space (size of tiling)




```python
def matmul_v1(N, L, M, dtype):
 A = te.placeholder((N, L), name="A", dtype=dtype)
 B = te.placeholder((L, M), name="B", dtype=dtype)
 ......
 # Define search space
 cfg.define_knob("tile_y", [1, 2, 4, 8, 16])
 cfg.define_knob("tile_x", [1, 2, 4, 8, 16])

 # Schedule according to config
 yo, yi = s[C].split(y, cfg["tile_y"].val)
 xo, xi = s[C].split(x, cfg["tile_x"].val)
 ......
 return s, [A, B, C]
```

Template example in AutoTVM

Automating Optimization

- **AutoTVM overview**
  - Schedule explorer examines schedule space using ML-based cost model
  - And explorer chooses experiments to run on device cluster via RPC (Remote procedure call)

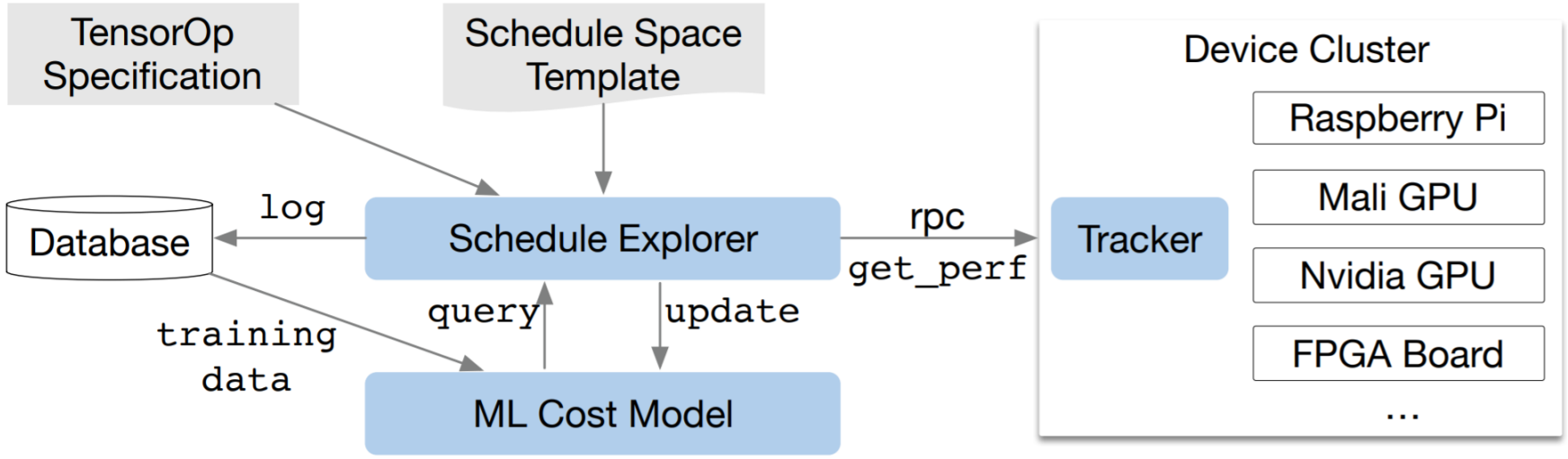
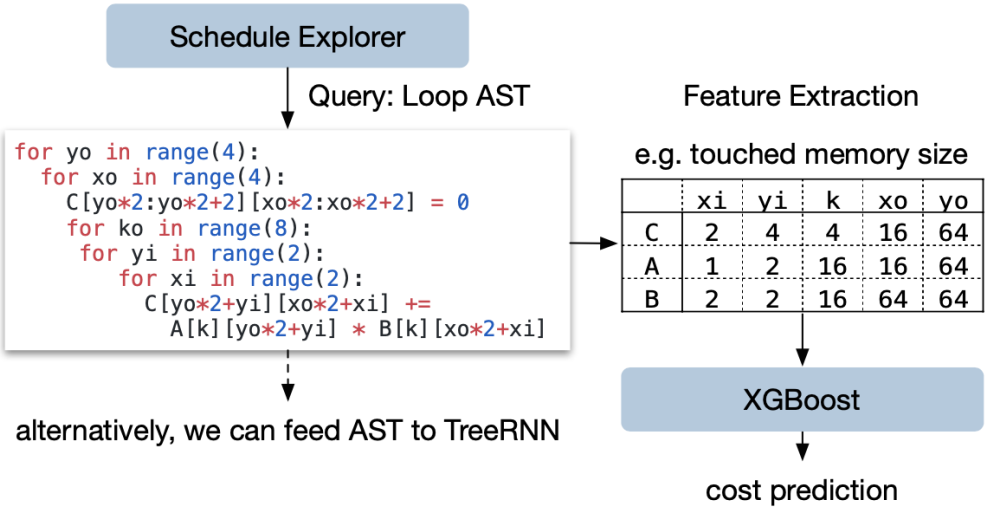
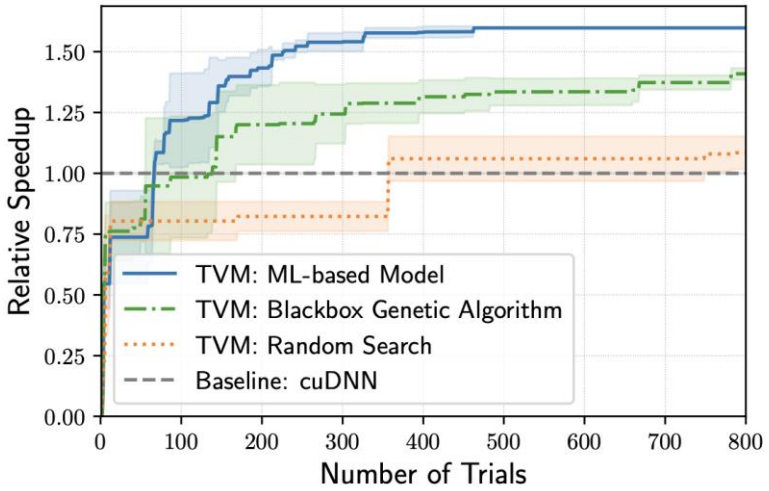


Illustration of AutoTVM

Automating Optimization

- **ML-based cost model**
  - Taking lowed loop program as input and Predicting (0.67ms) its running time in hardware back-end
  - ML model trained using runtime measurement data, no requirement of hardware information
  - AST (Abstract syntax tree), TreeRNN (Summarizing loop's AST), XGBoost (Gradient tree boosting)
  - Feature: memory access account, reuse ratio of each buffer at each loop level, loop annotations

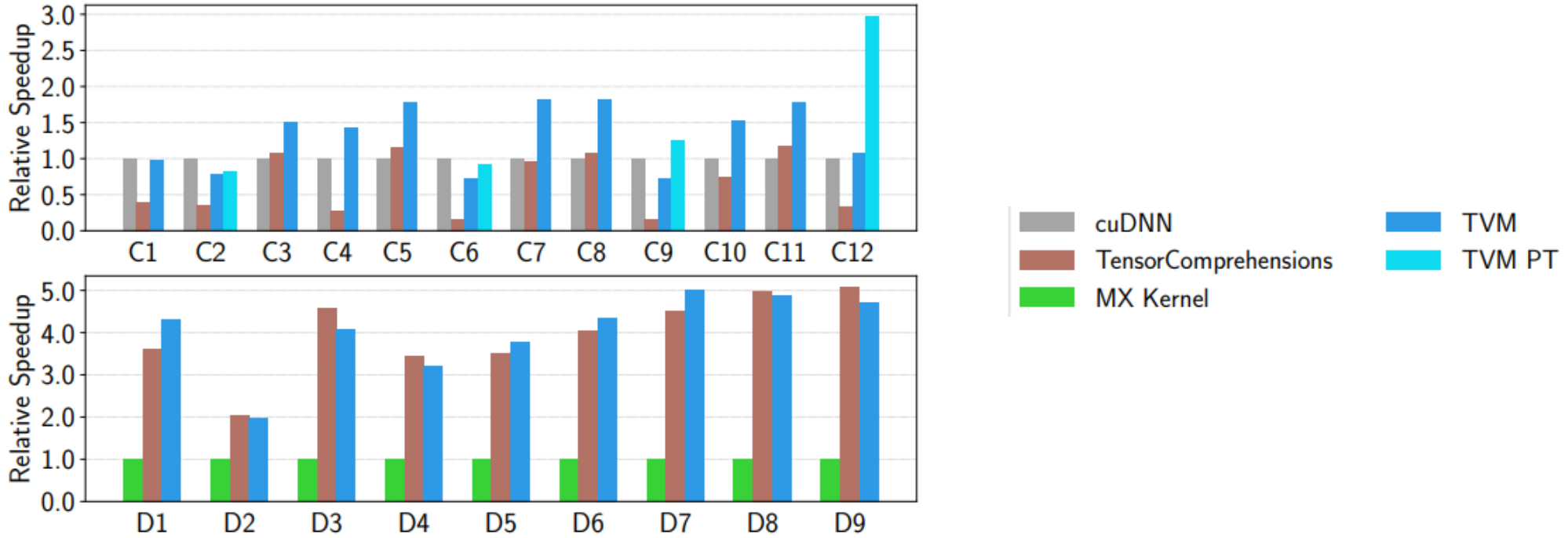




ML cost model (Left), Comparison of different automation (Right)

Performance

- **Server-class GPU evaluation**
  - TVM, MXNet (v1.1), TensorFlow (v1.7), TensorFlow XLA on TItanX/cuDNN
  - Tensor Comprehension (Auto-tuning framework)
  - TVM's improvements are mainly due to exploration of large schedule space/ML-based search algorithm



Performance

- **Embedded CPU evaluation**
  - TVM on ARM Cortex A53 with TFLite
  - TVM operators that outperform hand-optimized TFLite versions for both neural workloads

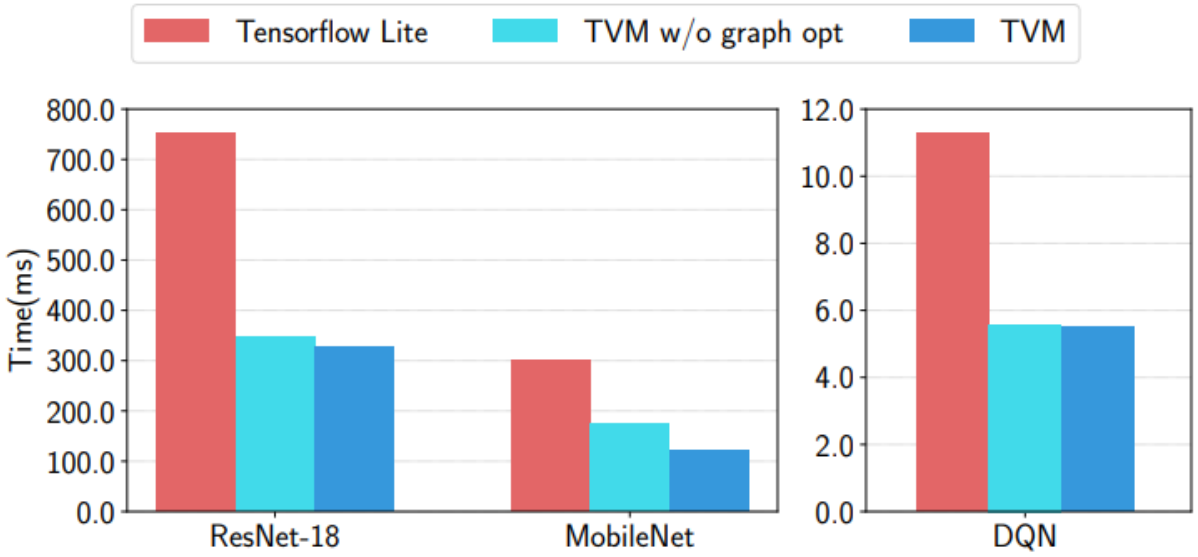


Figure 16: ARM A53 end-to-end evaluation of TVM and TFLite.

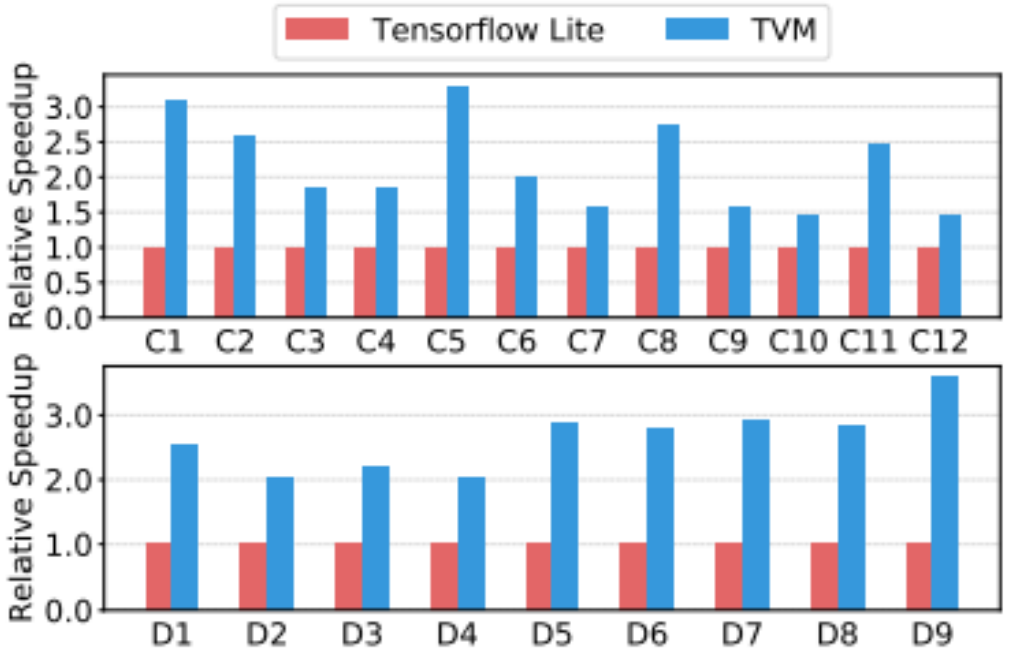


Figure 17: Relative speedup of all conv2d operators in ResNet-18 and all depthwise conv2d operators in mo-

Performance

- **Embedded GPU evaluation**
  - Firefly-RK3399 with Mali-T860MP4 GPU
  - ARM Compute Library (FP16/FP32)

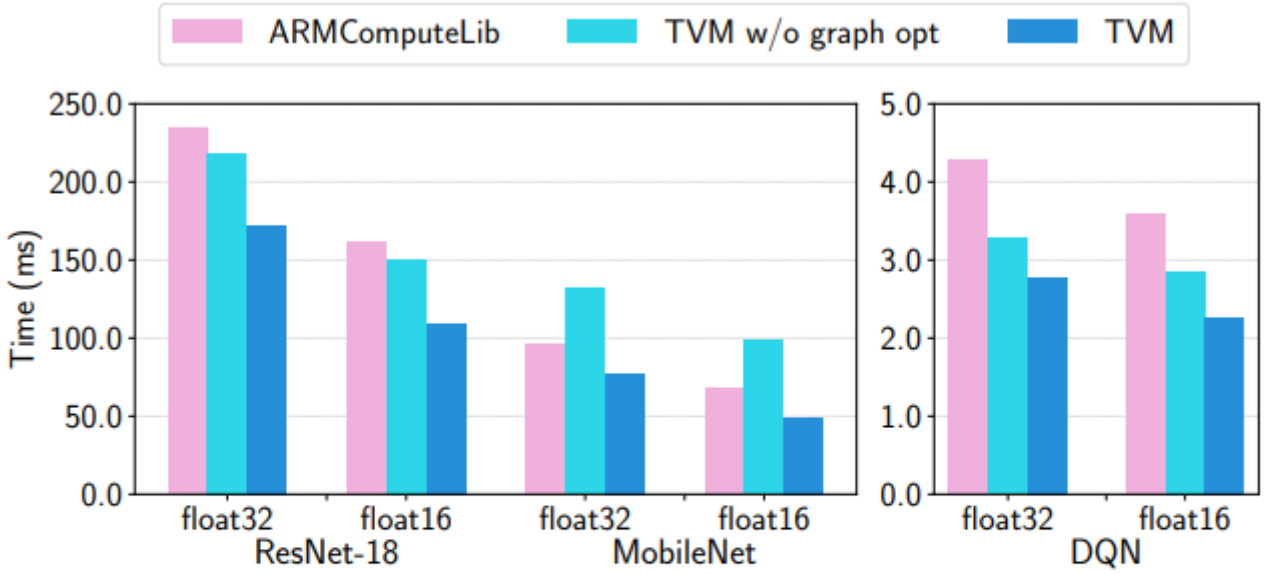


Figure 19: End-to-end experiment results on Mali-T860MP4. Two data types, float32 and float16, were evaluated.

Reference

[1] dMazeRunner, https://github.com/MPSLab-ASU/dMazeRunner

[2] NVIDIA Tensor Core Programmability, Performance & Precision

[3] 딥러닝을 빠르게 하는 방법: Tensor Core, MODUCON19

Thank you