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

Presenter: Cong Ding, Rainie Li

2021/04/24

Motivation

Frameworks

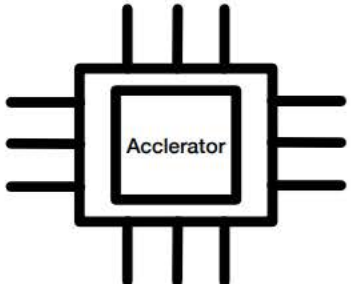
Explosion of models and frameworks

Motivation

Frameworks

Explosion of models and frameworks

Explosion of hardware backends



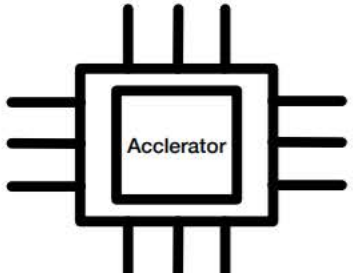
Motivation

Frameworks

Explosion of models and frameworks

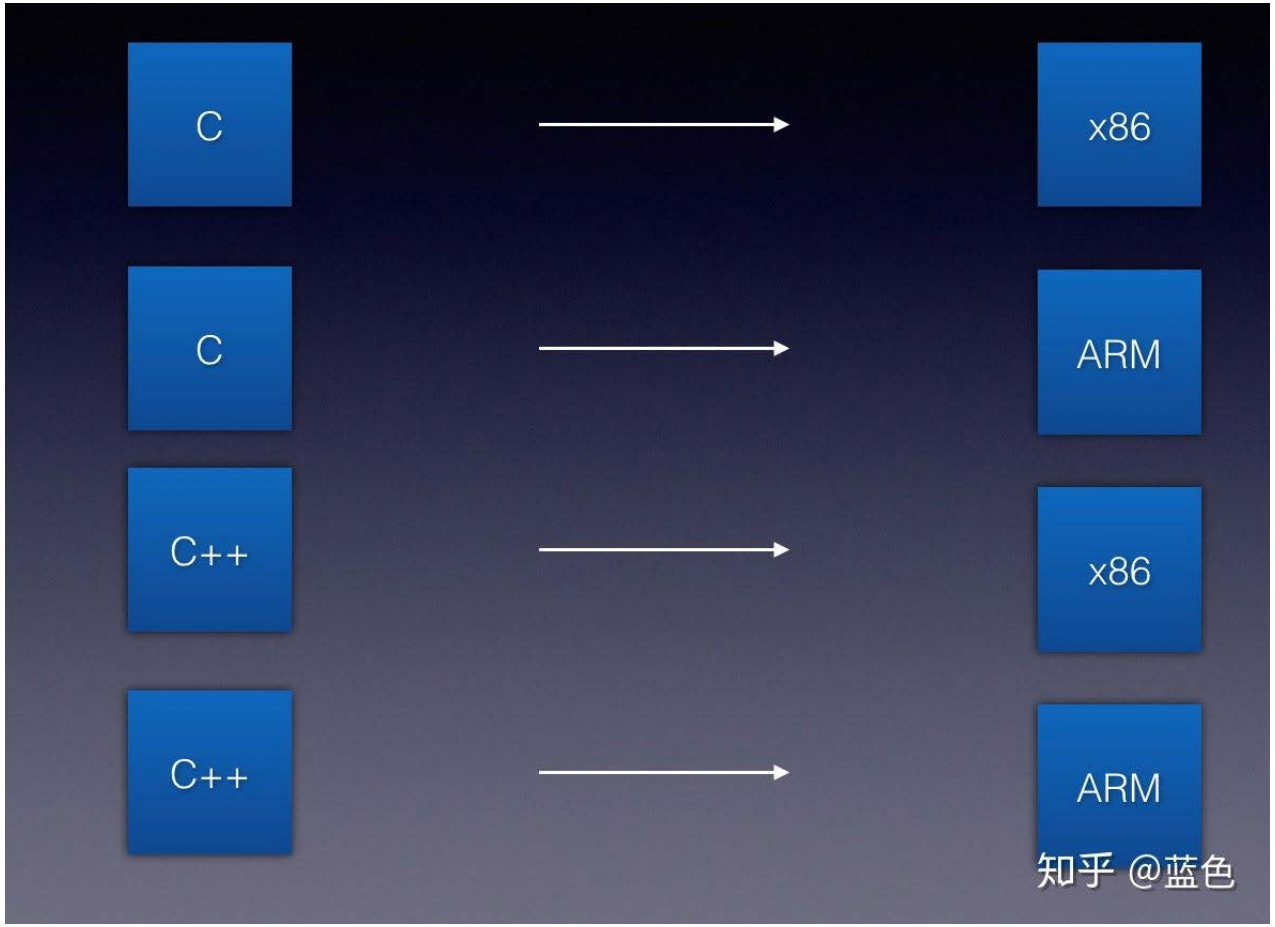
Huge gap between model/frameworks and hardware backends

Explosion of hardware backends



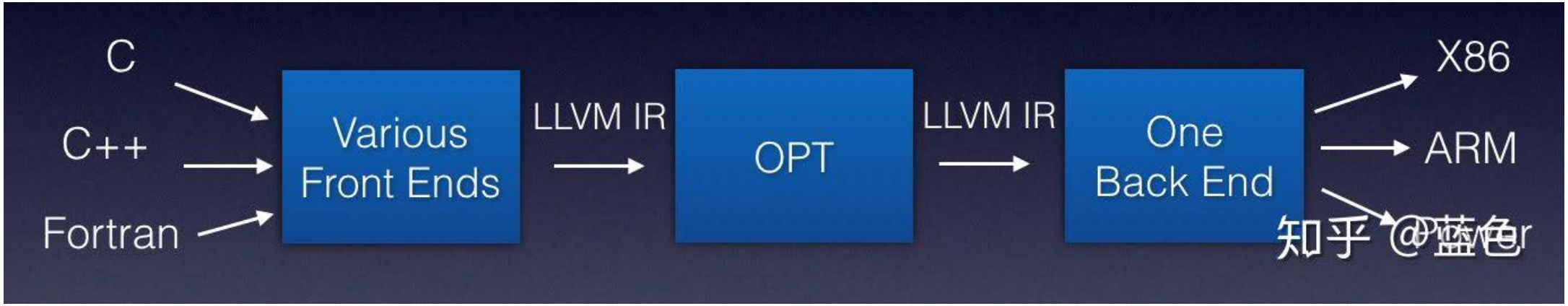
Motivation

- Compiler

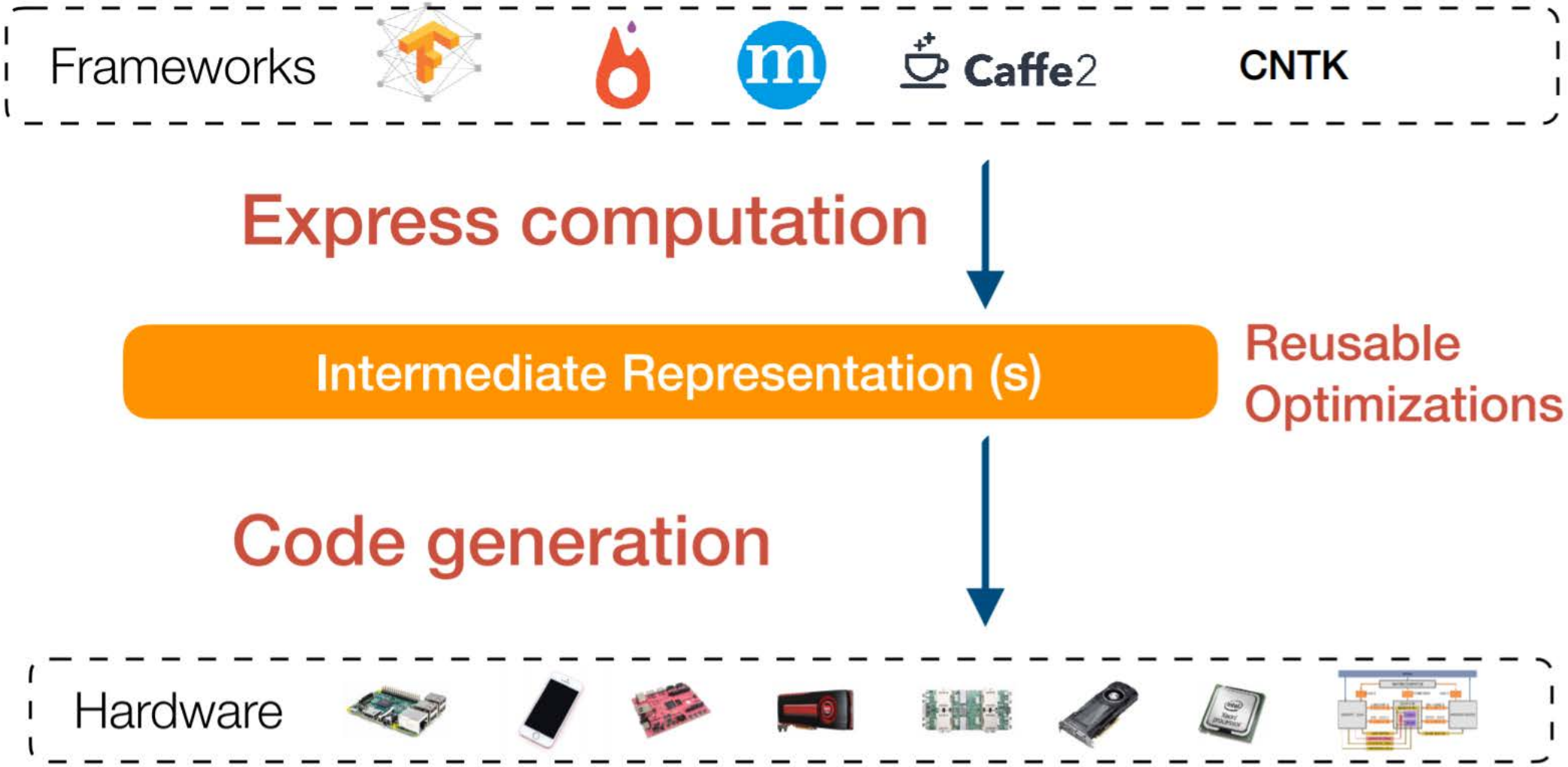


Motivation

- Intermediate Representation (IR)
  - Machine independent
  - Exposes optimization opportunities



Motivation


Frameworks
Caffe2
CNTK
Express computation
Intermediate Representation (s)
Reusable
Optimizations
Code generation
Hardware

Motivation

Frameworks

Caffe2

CNTK

Computational Graph Optimization

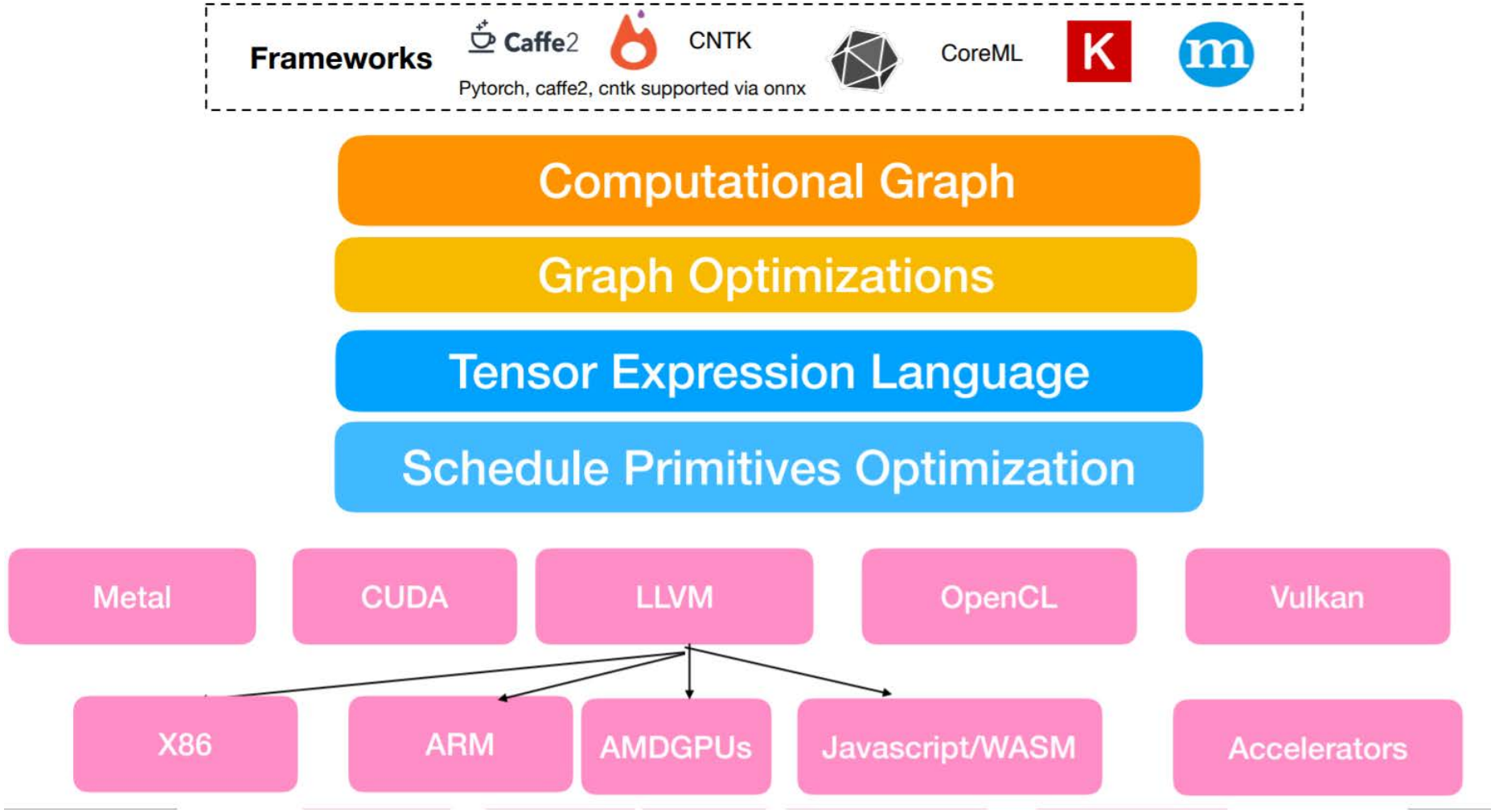
need to build and optimize operators for each hardware, variant of layout, precision, threading pattern ...

Hardware

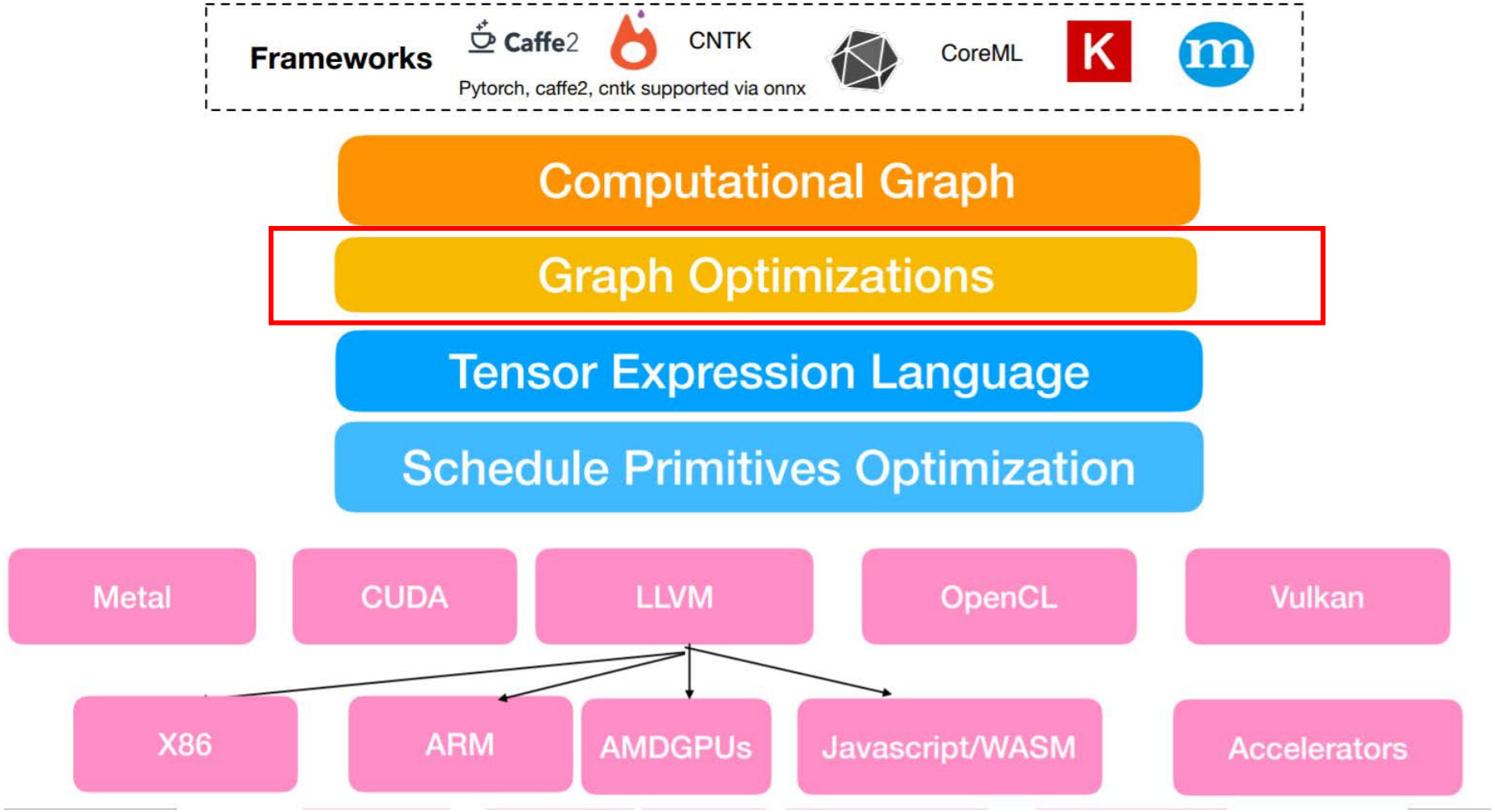
Motivation

- Computational Graph as IR
  - Operator specific
  - Engineering intensive
  - Hardware dependent
  - Too high level for optimizations
- TVM: Tensor Expression as IR
  - Expressive
  - Easier to perform optimizations on
  - Automated
  - Generalized to various backends

Global View


Frameworks
Caffe2
CNTK
CoreML
Pytorch, caffe2, cntk supported via onnx
Computational Graph
Graph Optimizations
Tensor Expression Language
Schedule Primitives Optimization
Metal
CUDA
LLVM
OpenCL
Vulkan
X86
ARM
AMDGPUs
Javascript/WASM
Accelerators

Global View


Frameworks
Caffe2
CNTK
CoreML
Pytorch, caffe2, cntk supported via onnx
Computational Graph
Graph Optimizations
Tensor Expression Language
Schedule Primitives Optimization
Metal
CUDA
LLVM
OpenCL
Vulkan
X86
ARM
AMDGPUs
Javascript/WASM
Accelerators

Graph Optimizations

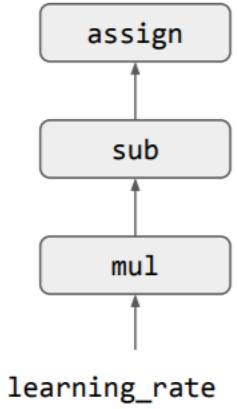
- Operator Fusion
- Constant Folding
- Static Memory Planning Pass
- Data Layout Transformations

Graph Optimizations

- Operator Fusion (reduce memory access)

```
w = w - lr * grad
```

Computation



Sequential Kernel Execution

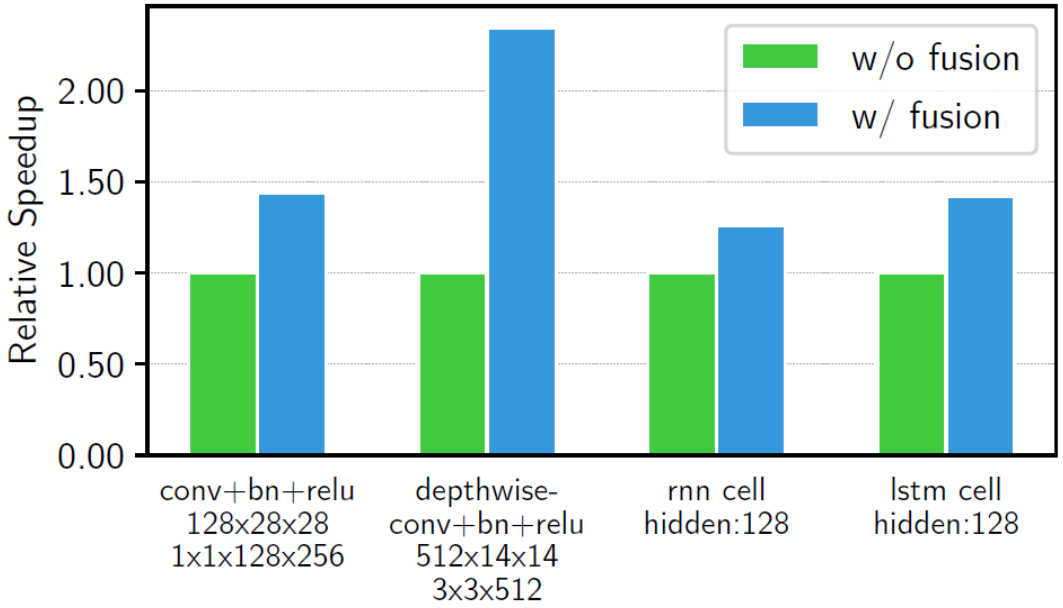
```
for (int i = 0; i < n; ++i) {
   temp1[i] = lr * grad[i]
}
for (int i = 0; i < n; ++i) {
   temp2[i] = w[i] - temp1[i]
}
for (int i = 0; i < n; ++i) {
   w[i] = temp2[i]
}
```

Fused Kernel Execution

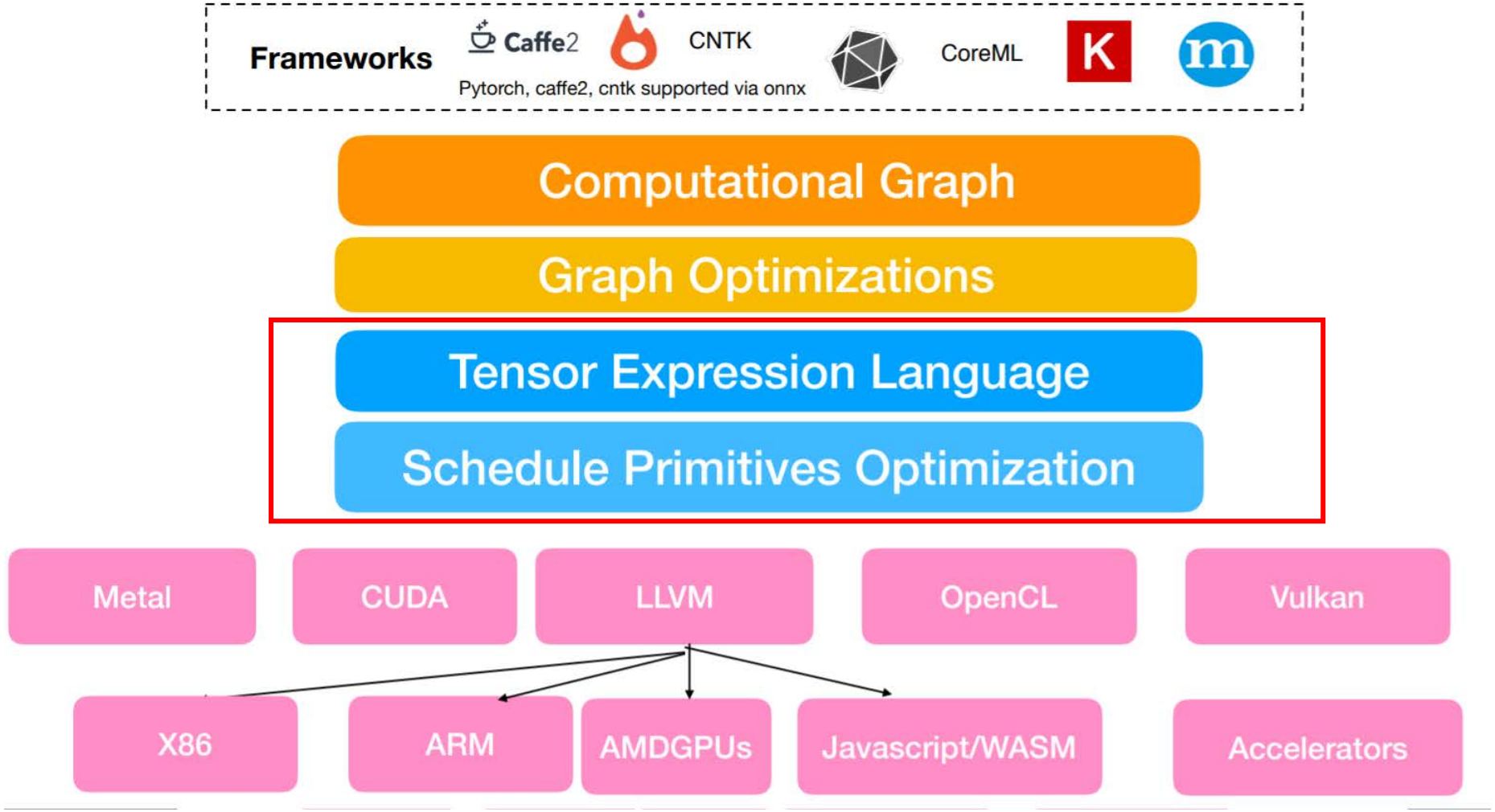
```
for (int i = 0; i < n; ++i) {
   w[i] = w[i] - lr * grad[i]
}
```

Graph Optimizations

- Operator Fusion (reduce memory access)
  - Injective
  - Reduction
  - Complex-out-fusable
  - Opaque
- Data Layout Transformation (access locality)
  - Hardware-related



Global View


Frameworks
Caffe2
CNTK
CoreML
Pytorch, caffe2, cntk supported via onnx
Computational Graph
Graph Optimizations
Tensor Expression Language
Schedule Primitives Optimization
Metal
CUDA
LLVM
OpenCL
Vulkan
X86
ARM
AMDGPUs
Javascript/WASM
Accelerators

Tensor Expression

- tensor expression language in tvm

```
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

Tensor Expression

- Compute/Schedule Decoupling

```
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

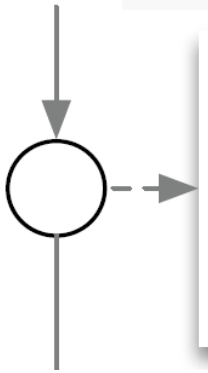
result shape

No Execution Details!

Schedule Primitives

- Preserve logical equivalence
- Apply schedule primitives incrementally

```
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]
```

Schedule Primitives

- Preserve logical equivalence
- Apply schedule primitives incrementally

+ 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]
```

Schedule Primitives

+ Cache Data on Accelerator Special Buffer

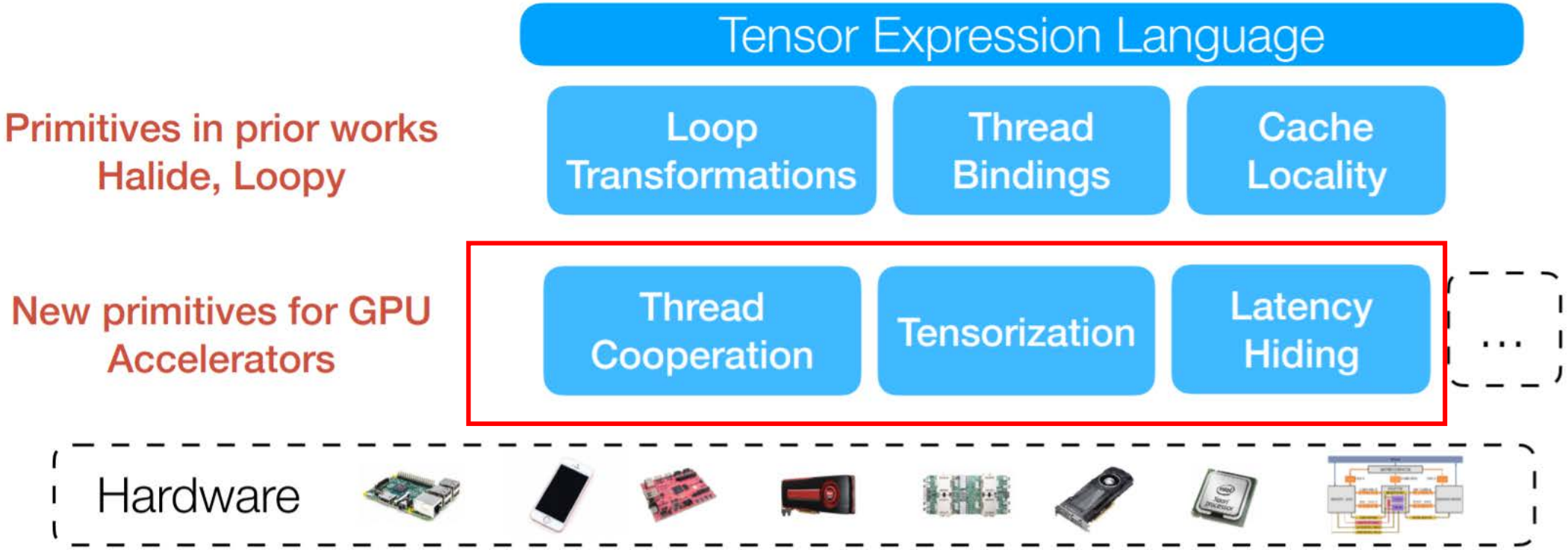
```
CL = s.cache_write(C, vdla.acc_buffer)
AL = s.cache_read(A, vdla.inp_buffer)
# additional schedule steps omitted …
```

+ 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 Primitives



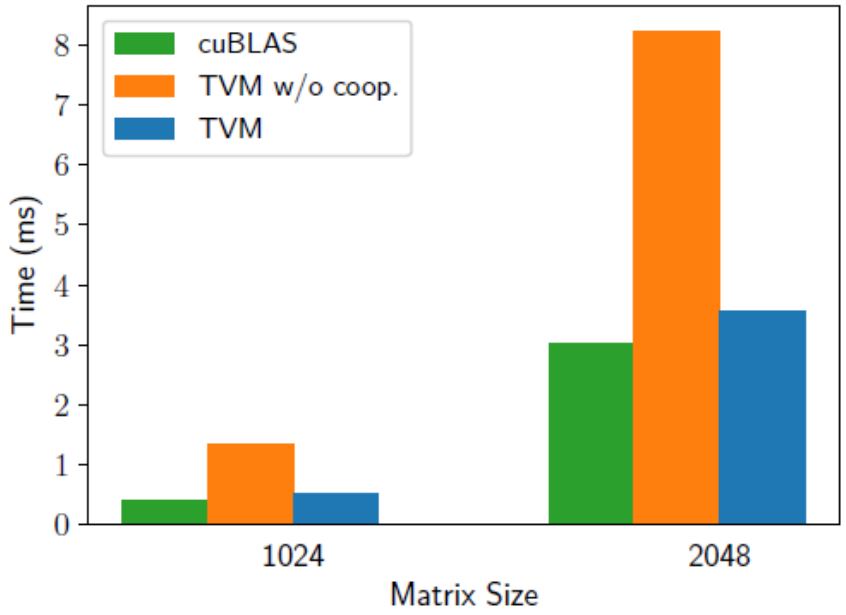
Schedule Primitives

- Thread 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



Schedule Primitives

- Tensorization

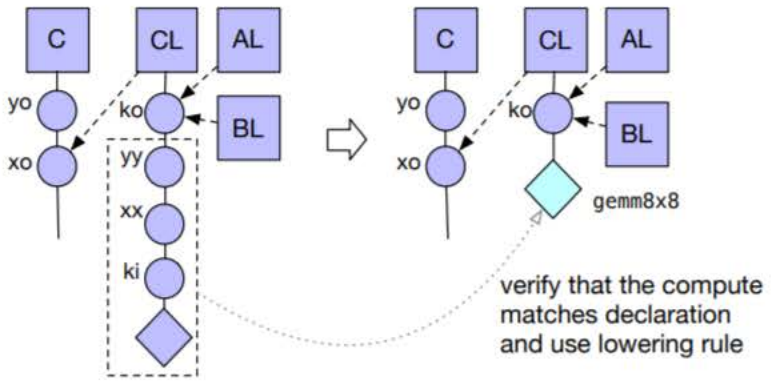
Hardware designer: declare tensor instruction interface

```
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

Tensorize: transform program to use tensor instructions



More details at https://tvm.apache.org/docs/tutorials/language/tensorize.html

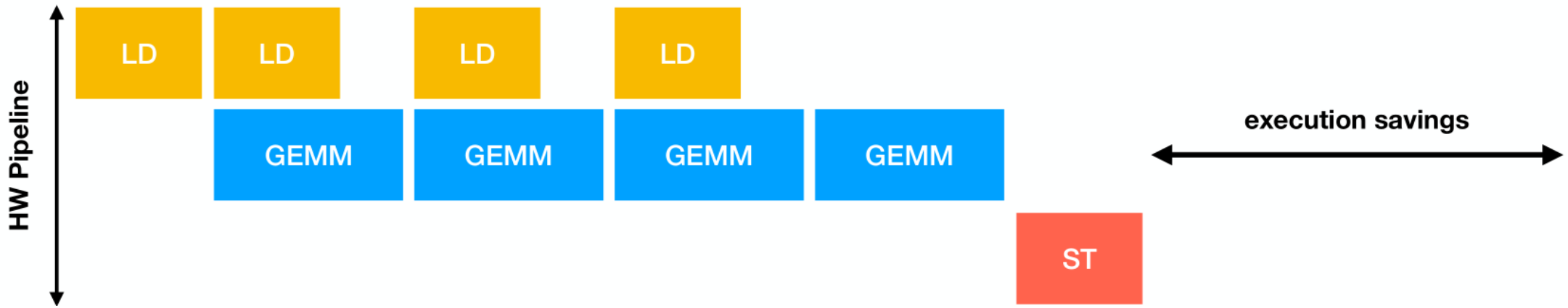
Schedule Primitives

- Latency Hiding
  - Without latency hiding, we are wasting compute/memory resources

  - By exploiting pipeline parallelism, we can hide memory latency

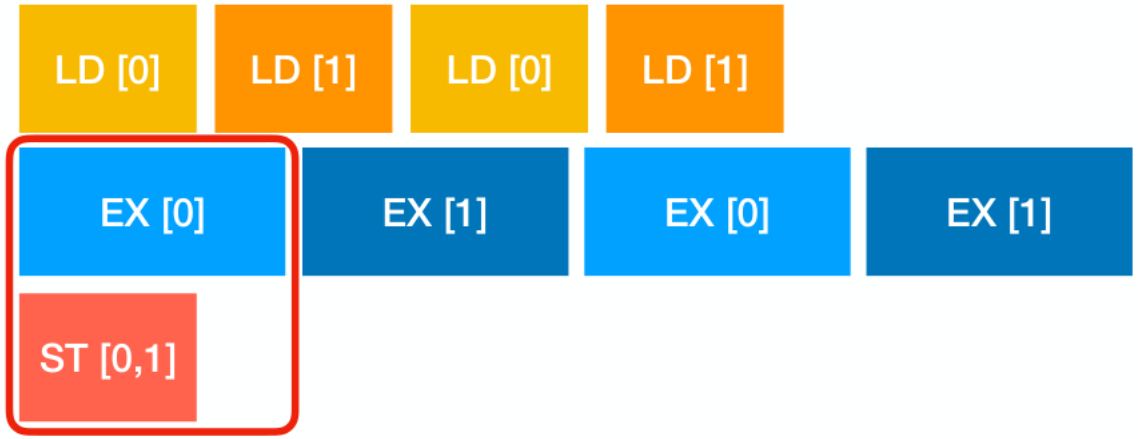
Schedule Primitives

- Latency Hiding
  - Concurrent tasks need to access non-overlapping regions of memory
  - Data dependences need to be explicit!



Schedule Primitives

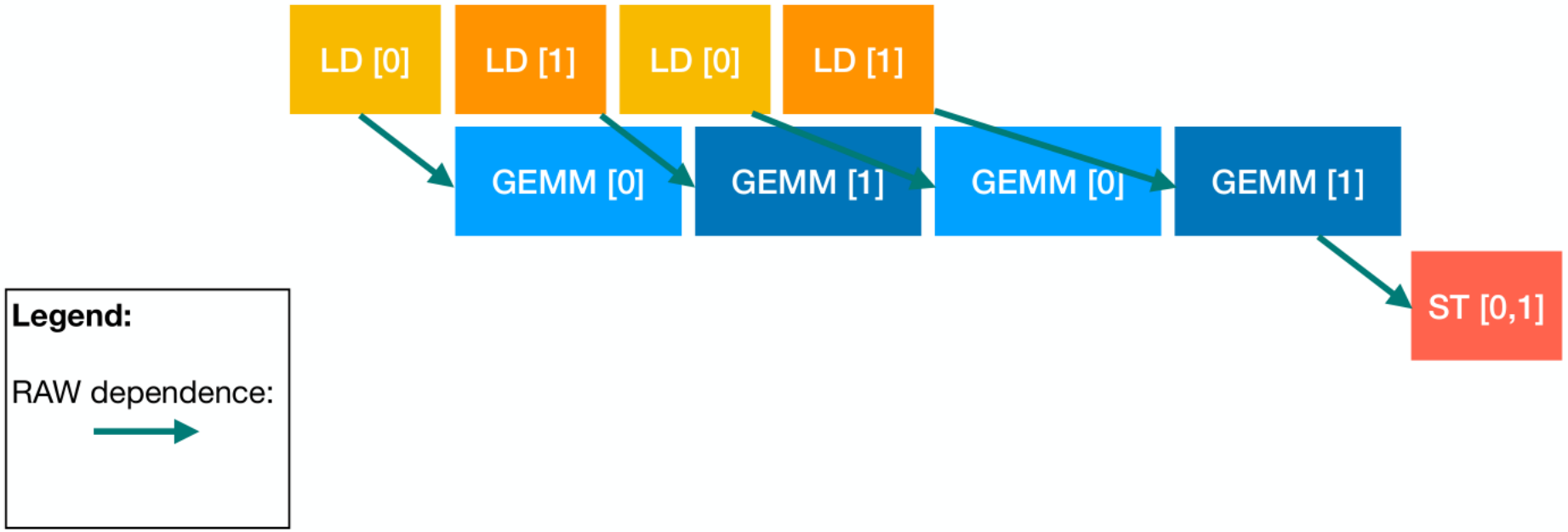
- Latency Hiding
  - RAW & WAR Dependencies



Without RAW dependence tracking, operations execute as soon as the stage is idle.

Schedule Primitives

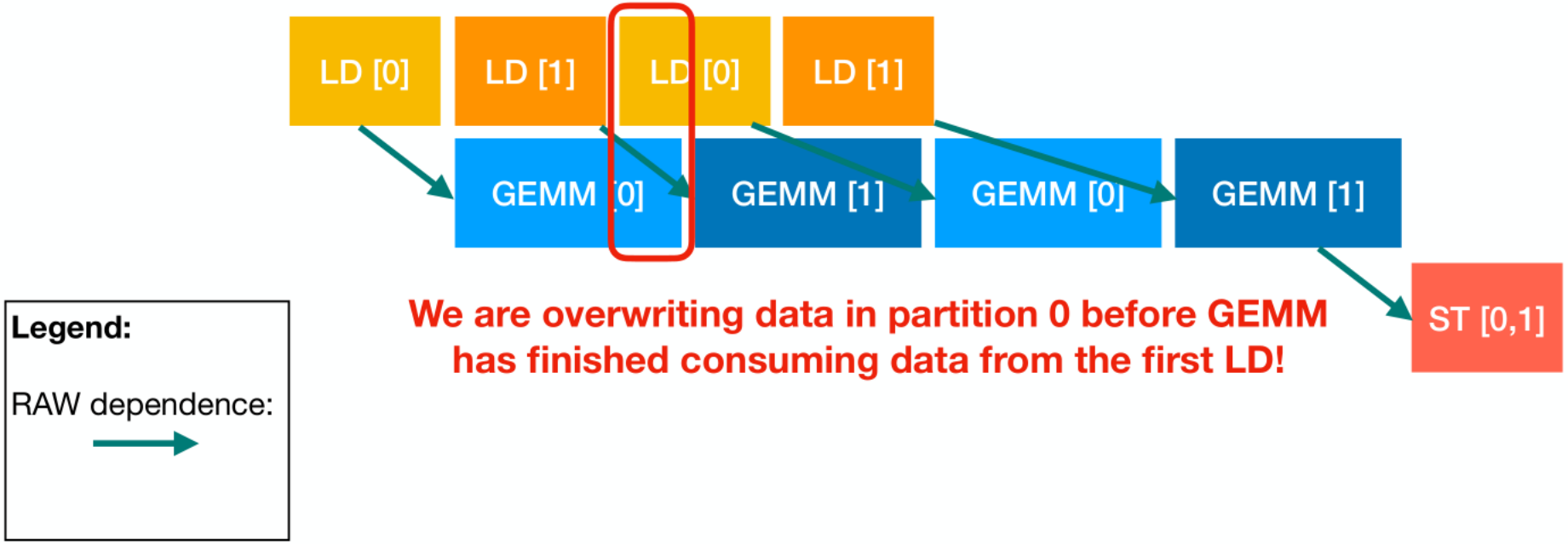
- Latency Hiding
  - RAW & WAR Dependencies



37

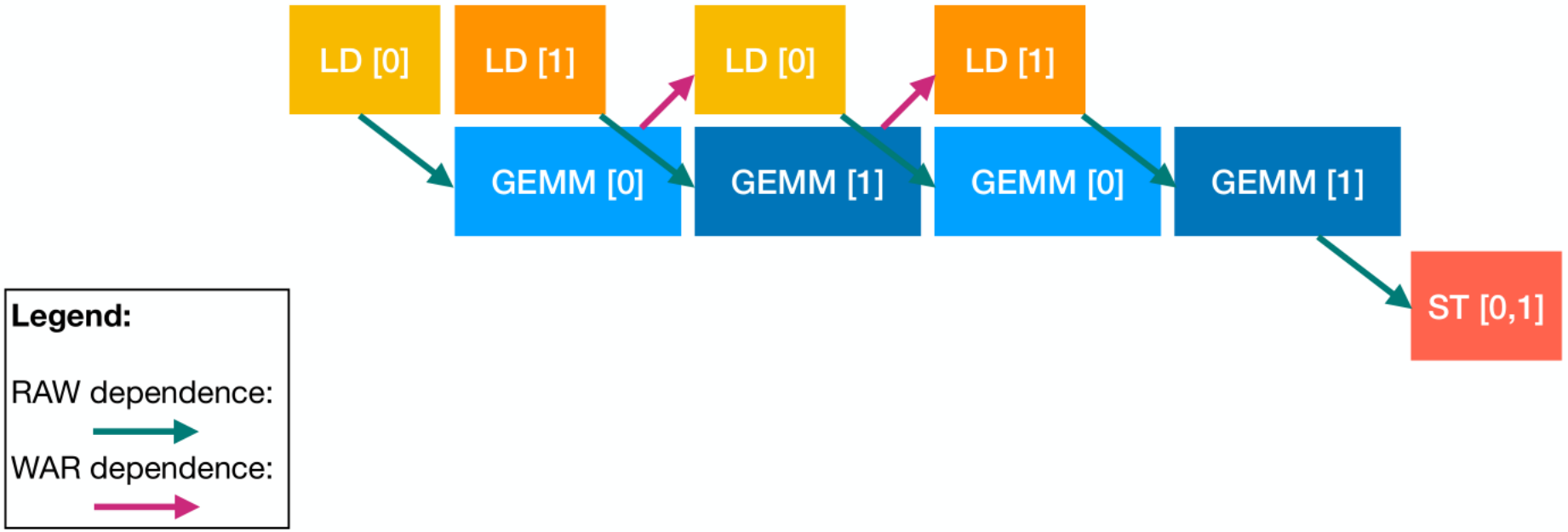
Schedule Primitives

- Latency Hiding
  - RAW & WAR Dependencies



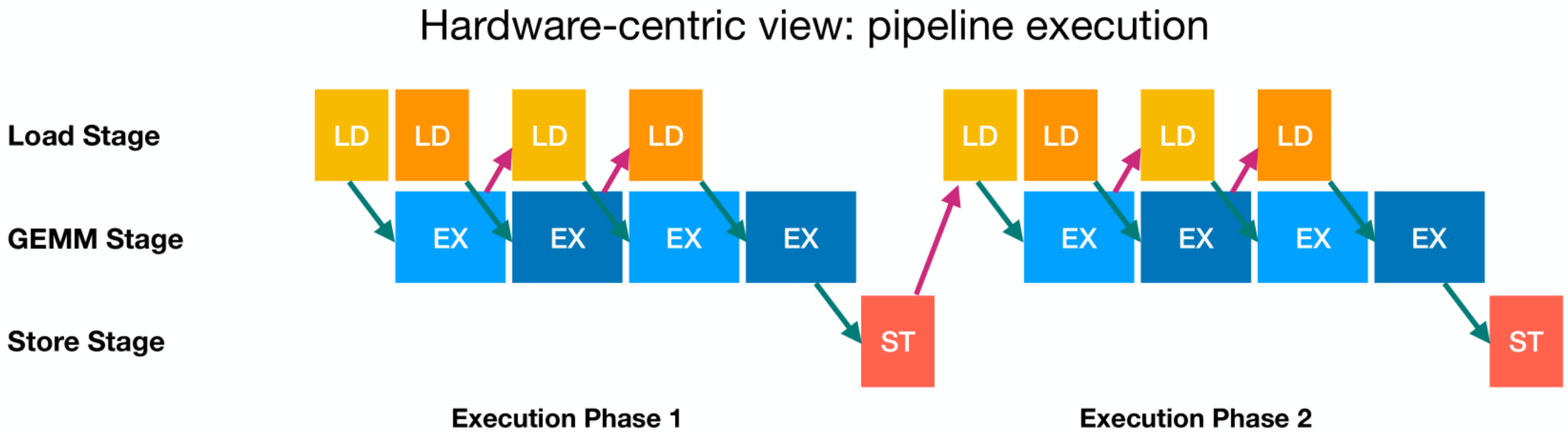
Schedule Primitives

- Latency Hiding
  - RAW & WAR Dependencies



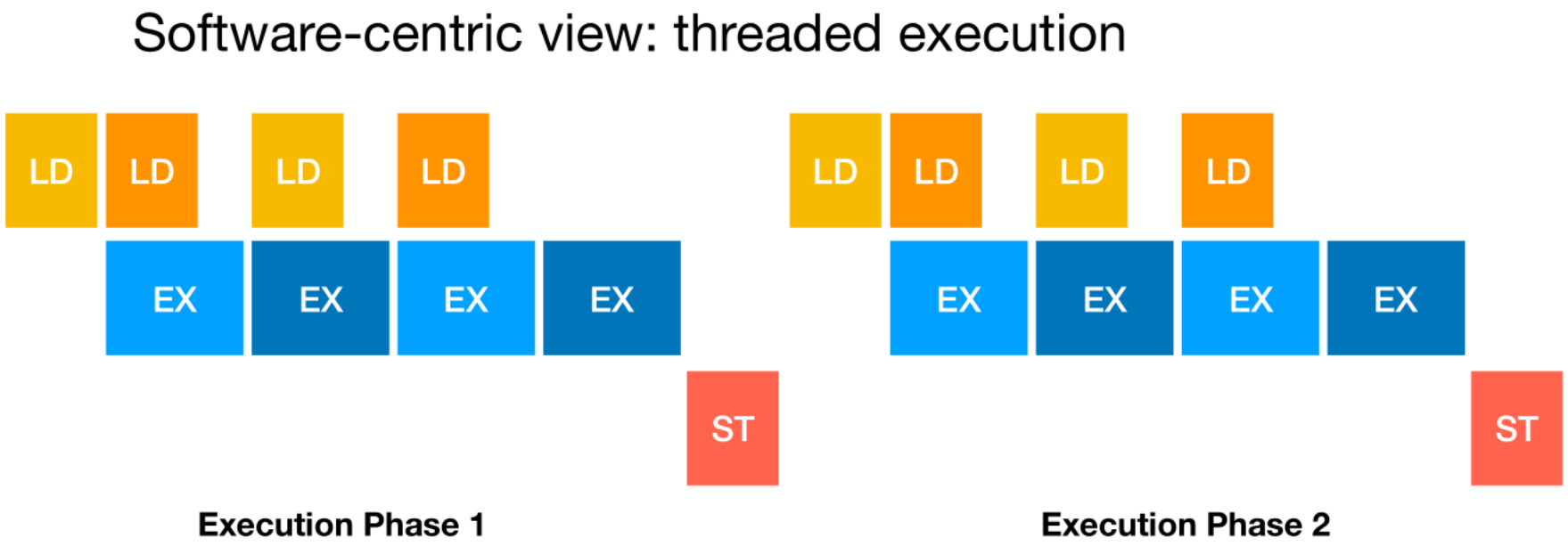
Schedule Primitives

- Virtual Threading
  - Hardware-centric view



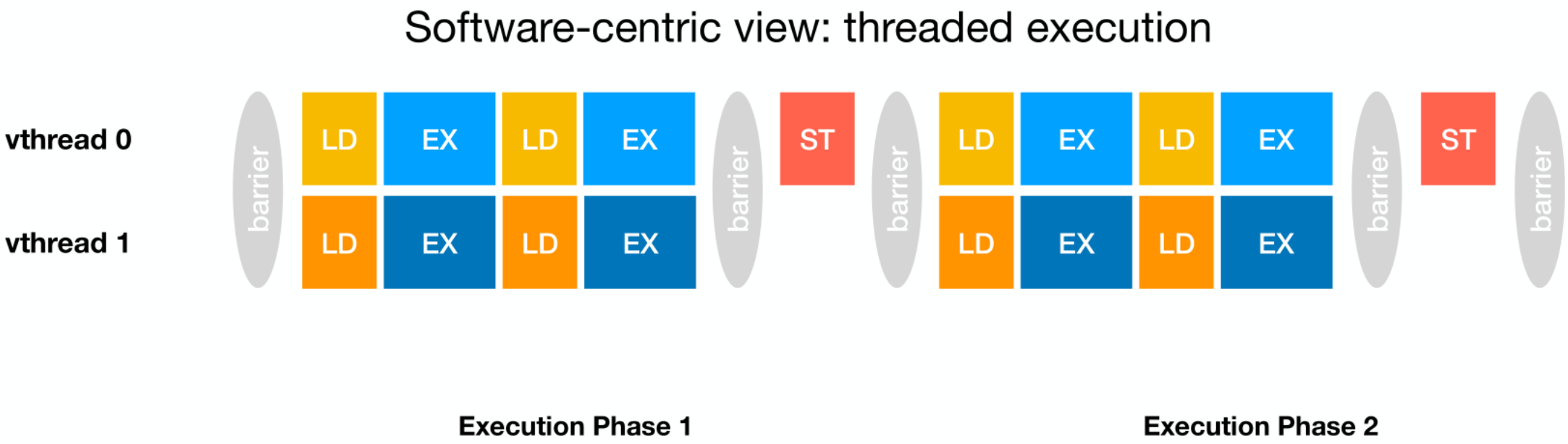
Schedule Primitives

- Virtual Threading
  - take advantage of pipeline parallelism with virtual threading



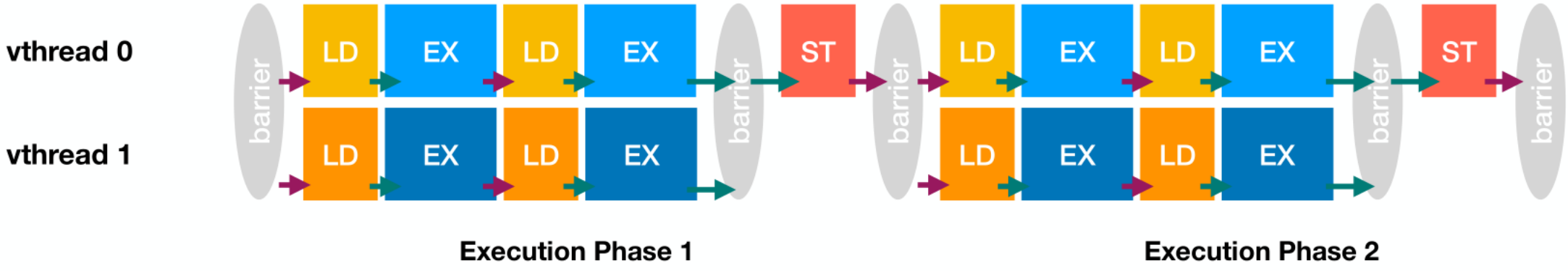
Schedule Primitives

- Virtual Threading
  - take advantage of pipeline parallelism with virtual threading



Schedule Primitives

Software-centric view: threaded execution



Legend:

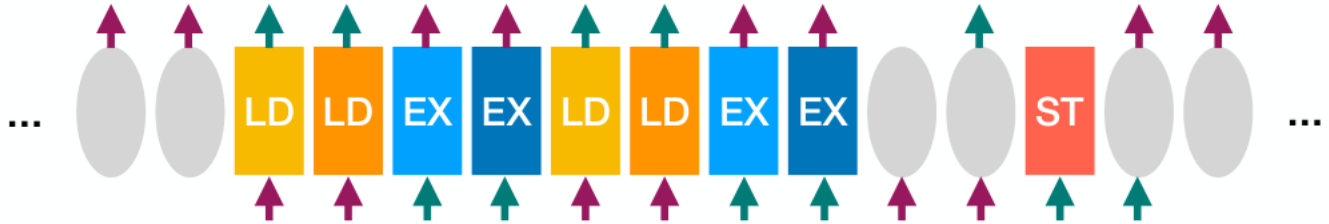
RAW dependence: →

WAR dependence: →

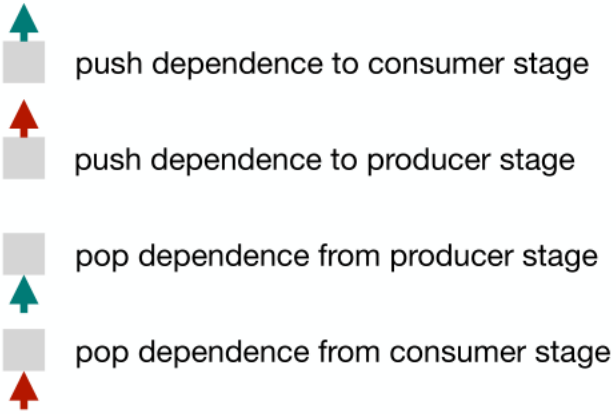
- Benefit #1: dependences are automatically inserted between successive stages within each virtual thread
- Benefit #2: barriers insert dependences between execution stages to guarantee sequential consistency

Schedule Primitives

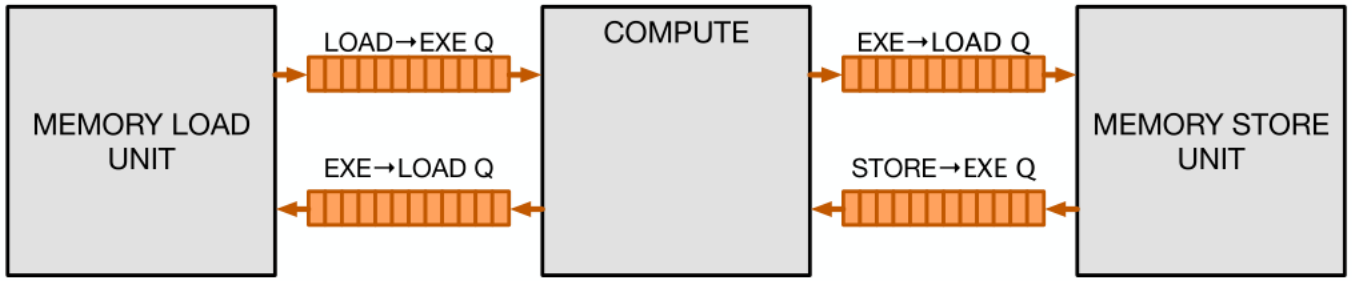
Final step: virtual thread lowering into a single instruction stream



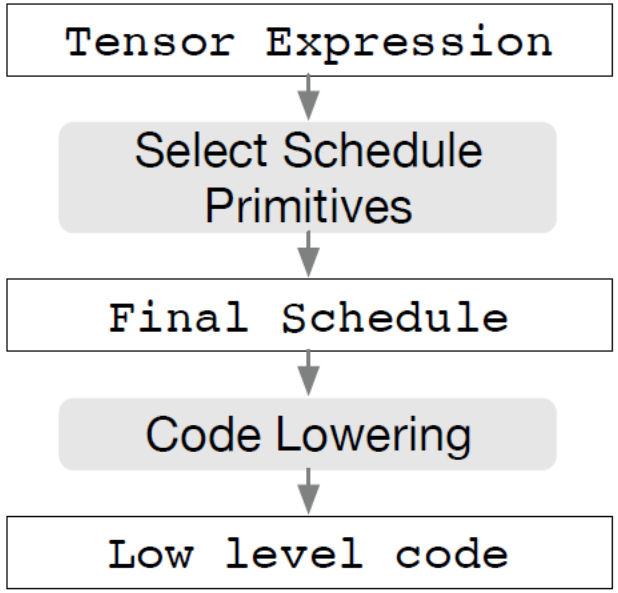
Legend



Push and pop commands dictate how to interact with the hardware dependence queues



Schedule Primitives



Schedule primitives used in various hardware backends	CPU Schedule	GPU Schedule	Accel. Schedule
[Halide] Loop Transformations	✓	✓	✓
[Halide] Thread Binding	✓	✓	✓
[Halide] Compute Locality	✓	✓	✓
[TVM] Special Memory Scope		✓	✓
[TVM] Tensorization	✓	✓	✓
[TVM] Latency Hiding			✓

References:

[1] TVM paper, slides, tutorial.

[2] UW CSE 599W: Systems for ML. http://dlsys.cs.washington.edu/

[3] https://zhuanlan.zhihu.com/p/50529704

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

TWO CHALLENGES FOR TVM

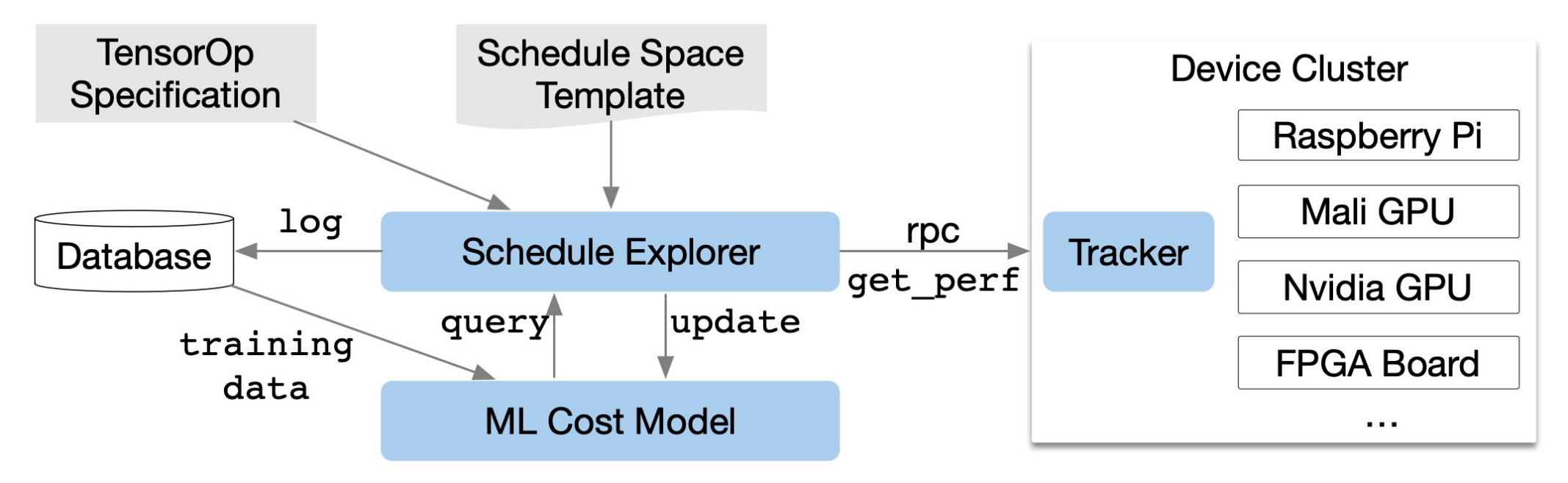
1. Leveraging Specific Hardware Features and Abstractions
2. Large Search Space for Optimization ✓

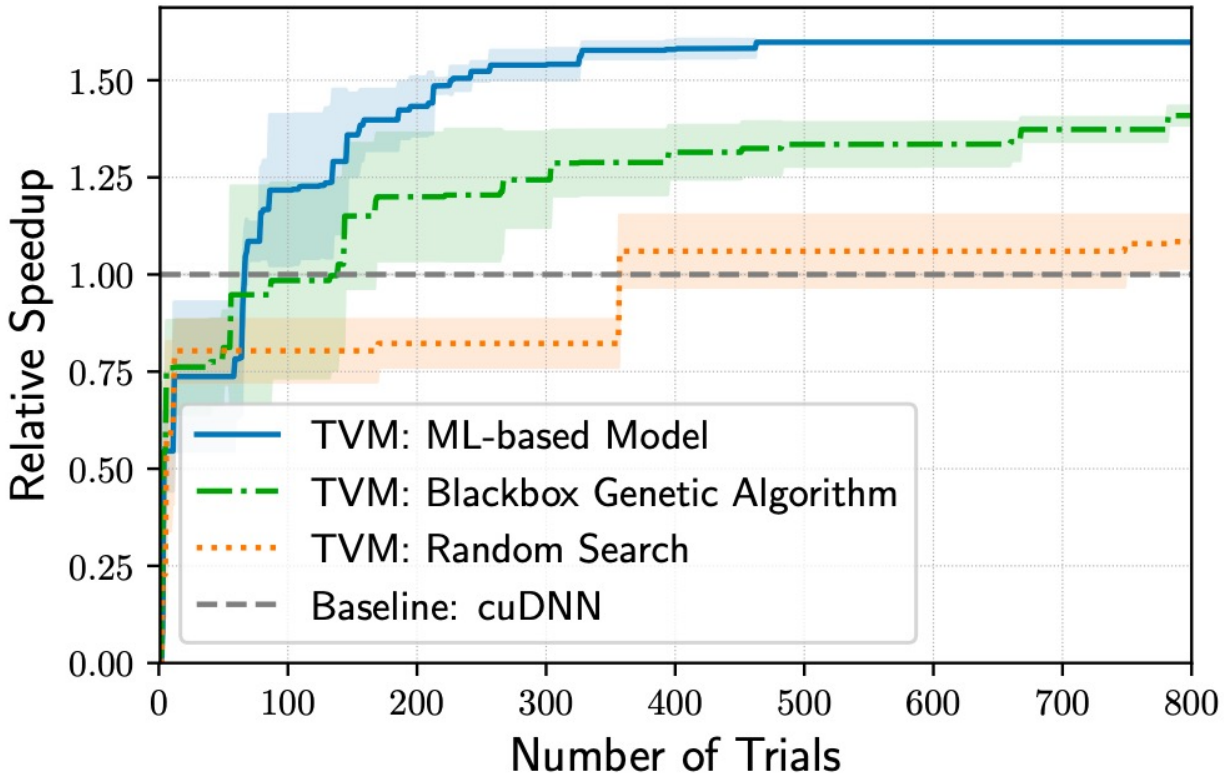
Combination of choices:
memory access,
threading pattern,
novel hardware primitives → loop tils and ordering,
caching,
unrolling

Cost Model?

AUTOMATED SCHEDULE OPTIMIZER

a schedule explorer： proposes promising new configurations
a machine learning cost model ： predicts the performance of a given configuration





a conv2d operator in ResNet-18 on TITAN X

MACHINE LEARNING MODEL DESIGN CHOICES

quality and speed

Speed-wise
Overhead:
Model prediction Time
Model refitting time
Hardware Performance Measuring Time

gradient tree boosting model (based on XGBoost)
(similar result as TreeRNN but faster)

Quality-wise:
objective function → relative order → a rank objective

Feature-wise:
the memory access count,
reuse ratio of each memory buffer at each loop level
a one-hot encoding of loop annotations such as "vectorize", "unroll", and "parallel."

SCHEDULE EXPLORATION

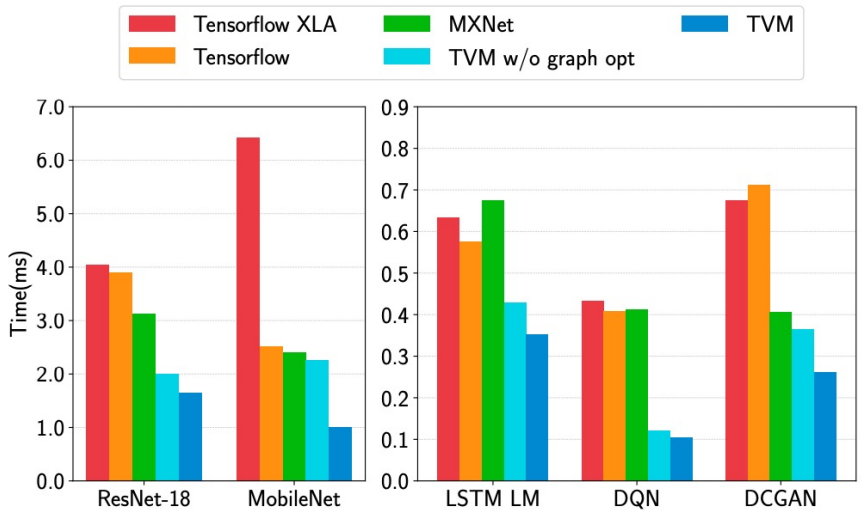
parallel simulated annealing algorithm

DISTRIBUTED DEVICE POOL AND RPC

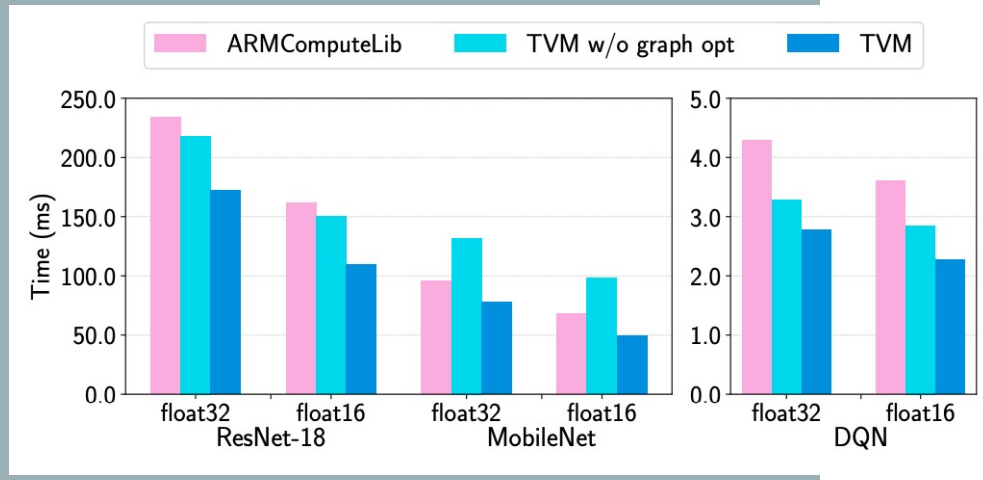
automates the compile, run, and profile steps across multiple devices

EVALUATION

Nvidia GPU



ARM GPU



ARM CPU