TVM

An End-to-End Optimizing Compiler for Deep Learning

Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Meghan Cowan, Haichen Shen, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, Arvind Krishnamurthy

TVM

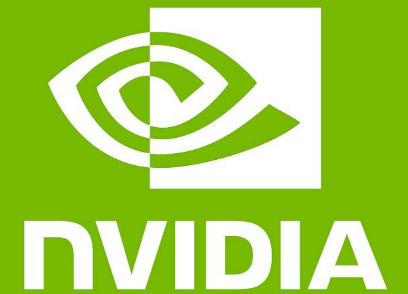
An End-to-End Optimizing Compiler for Deep Learning

Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Meghan Cowan, Haichen Shen, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, Arvind Krishnamurthy

TVM

An End-to-End Optimizing Compiler for Deep Learning

Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Meghan Cowan, Haichen Shen, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, Arvind Krishnamurthy



TVM

An End-to-End Optimizing Compiler for Deep Learning

Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Meghan Cowan, Haichen Shen, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, Arvind Krishnamurthy

TVM

2018

XGBoost: A Scalable Tree Boosting System

Tianqi Chen
University of Washington
tqchen@cs.washington.edu

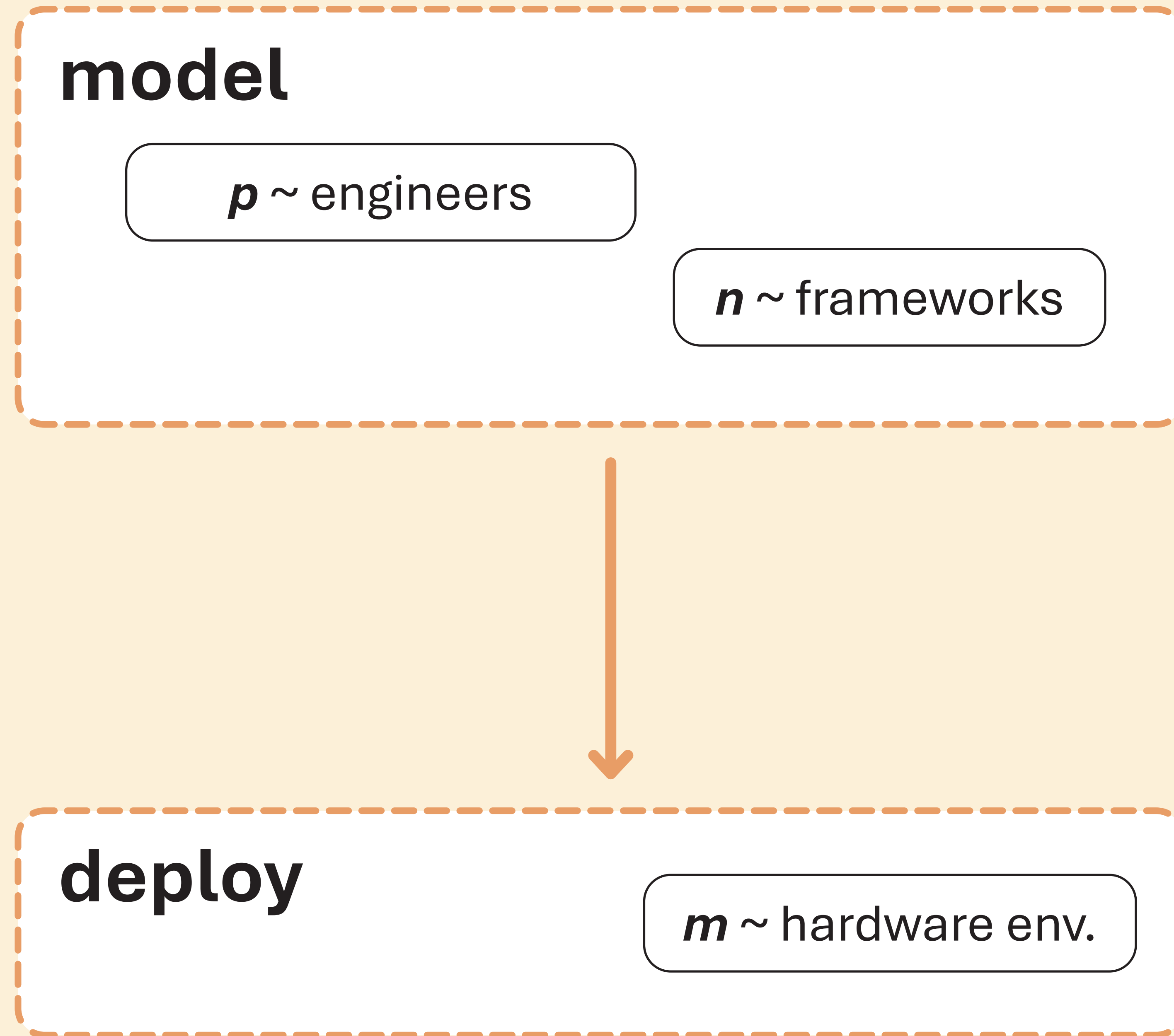
Carlos Guestrin
University of Washington
guestrin@cs.washington.edu

2016

An End-to-End Optimizing Compiler for Deep Learning

Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Meghan Cowan, Haichen Shen, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, Arvind Krishnamurthy

imagine ...



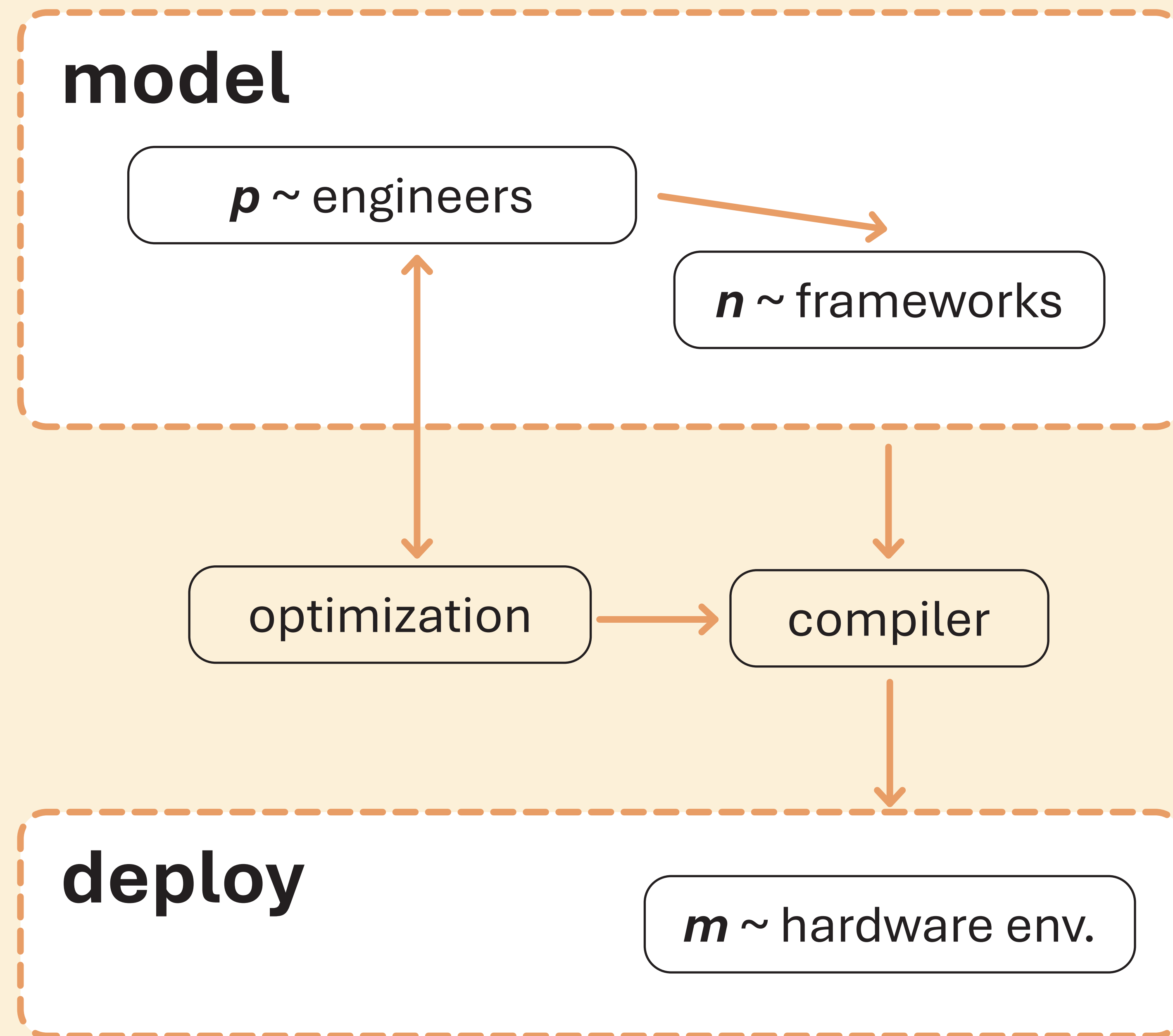
imagine ...

in a case where:

p is finite

n is infinite

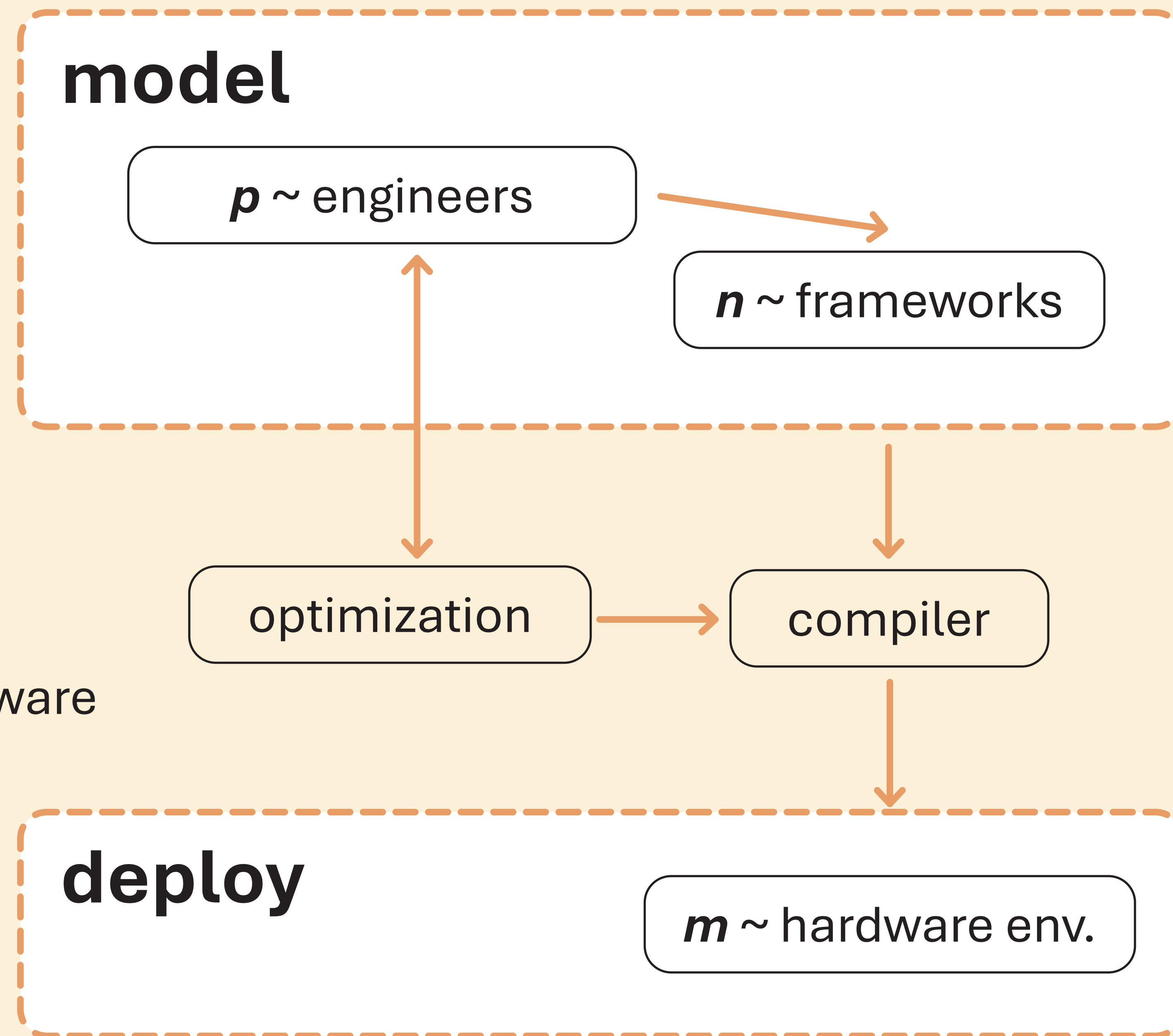
m is evolving / growing ...



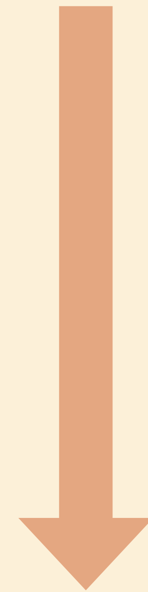
imagine ...

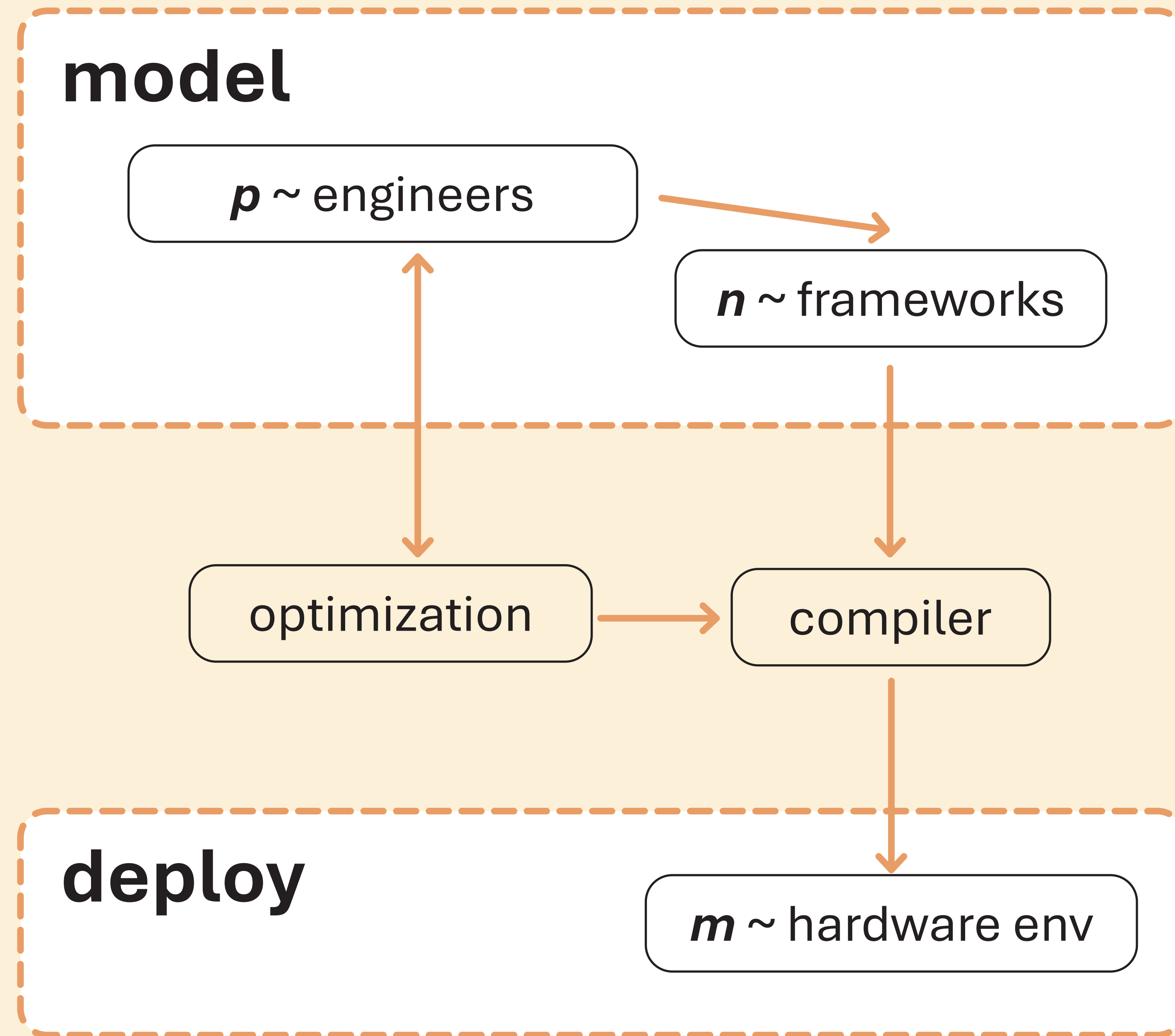
in a case where:

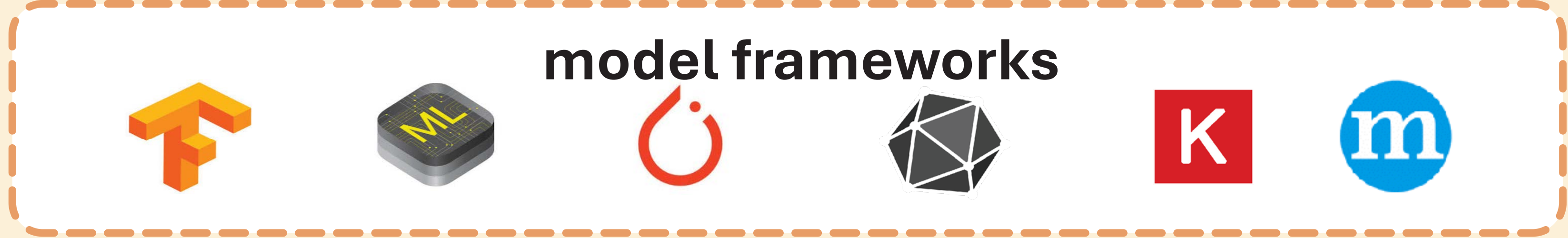
the goal is to optimize each hardware environment ...

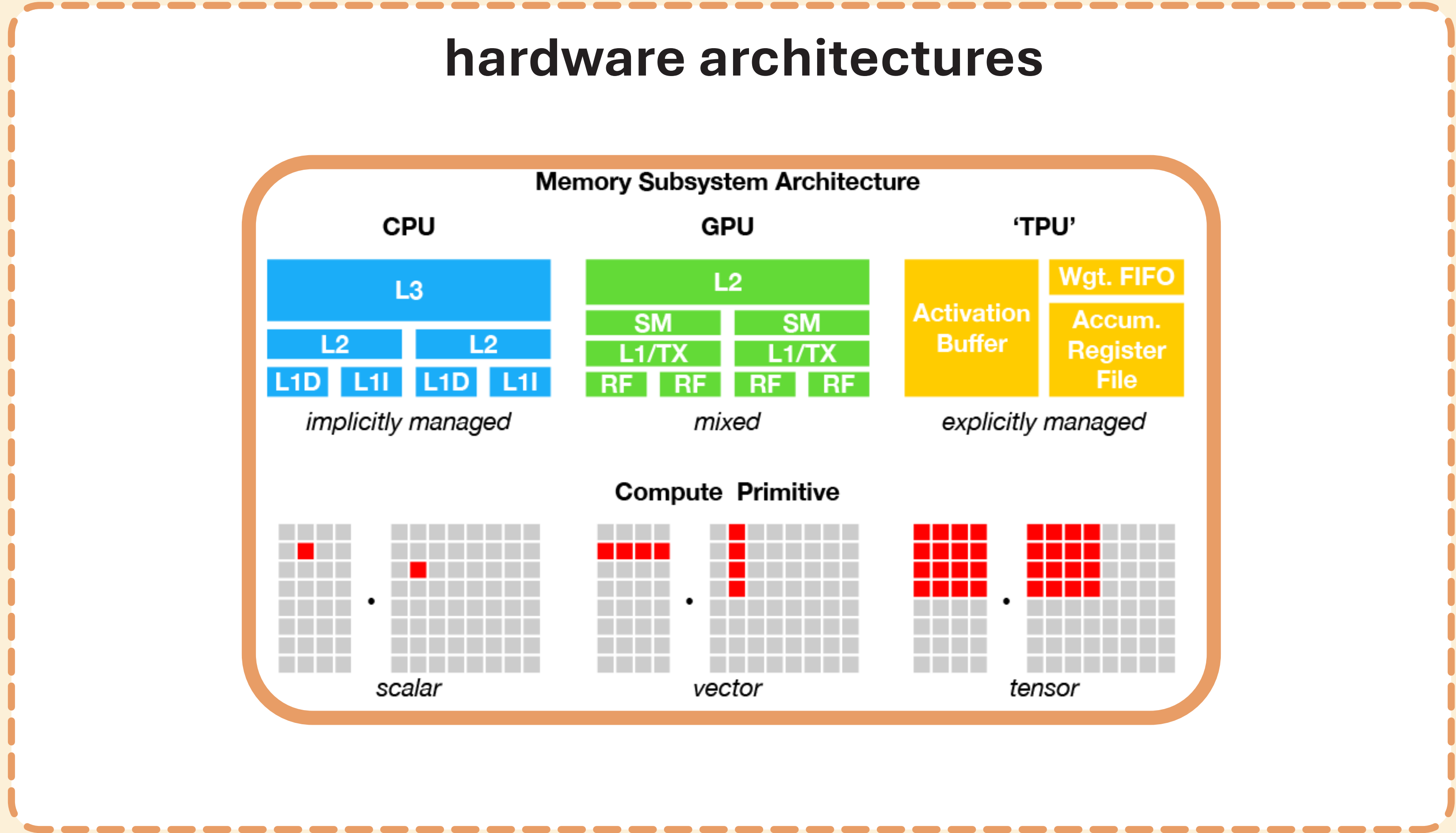


TVM was designed to solve this bottleneck …






model frameworks
ML
K
m


hardware architectures
Memory Subsystem Architecture
CPU
GPU
'TPU'
L3
L2
L2
L1D
L1I
L1D
L1I
implicitly managed
L2
SM
SM
L1/TX
L1/TX
RF
RF
RF
RF
mixed
Activation Buffer
Wgt. FIFO
Accum. Register File
explicitly managed
Compute Primitive
scalar
vector
tensor

INPUT ~ naive algorithm

```
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)
```

INPUT →

TVM optimization

OUTPUT ~ optimized naive algorithm

```
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)
```

INPUT

```
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)
```

TVM

```
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

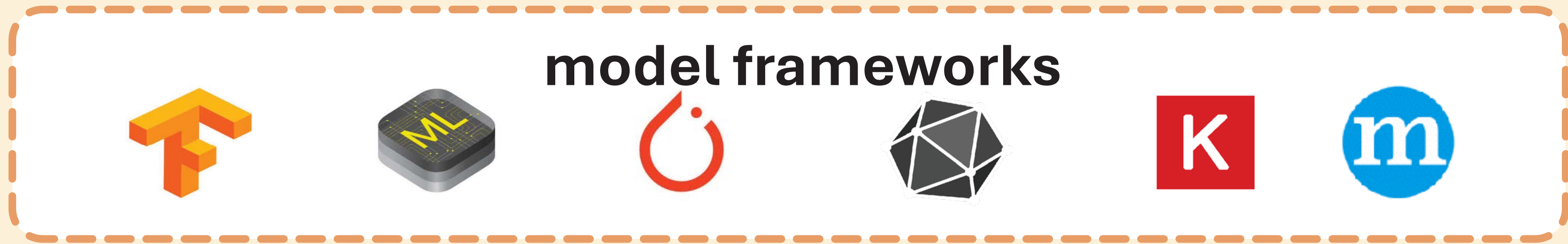
```
CL = s.cache_write(C, vdla.acc_buffer)
AL = s.cache_read(A, vdla.inp_buffer)
# additional schedule steps omitted …
```

+ Map to Accelerator Tensor Instructions

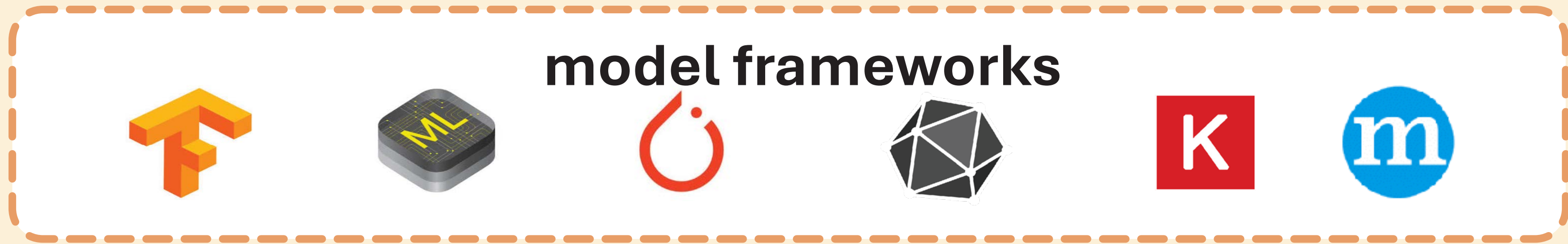
```
s[CL].tensorize(yi, vdla.gemm8x8)
```

OUTPUT

```
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)
```

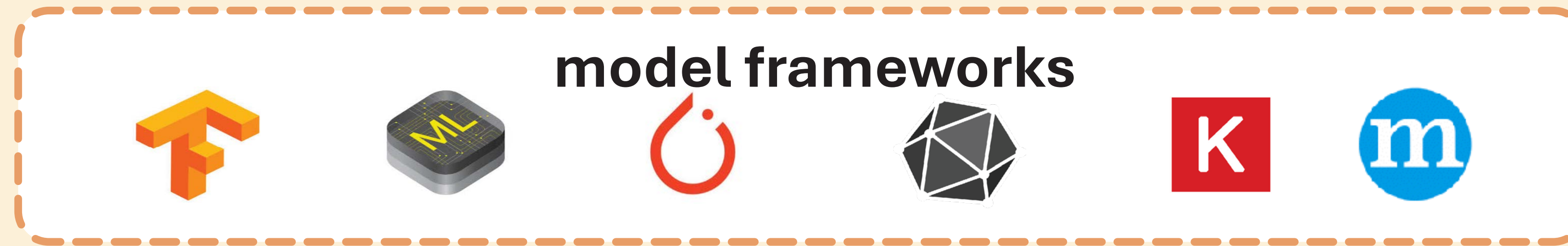


TVM



TVM

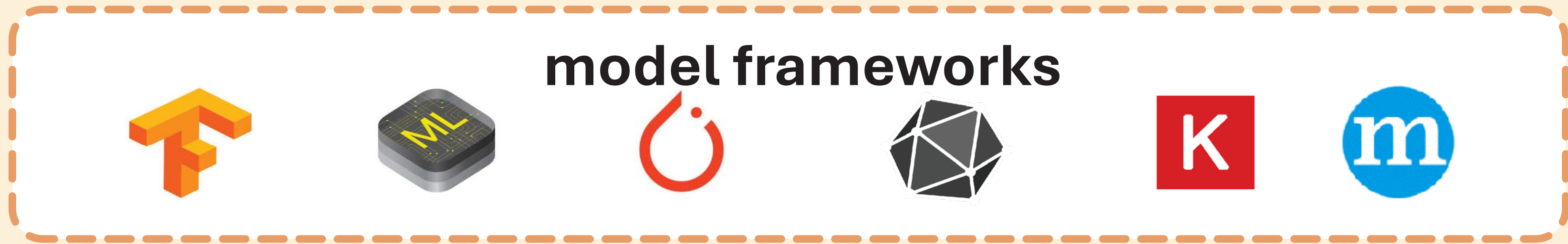
exposes computational graphs



TVM

optimizes tensor operations

exposes computational graphs



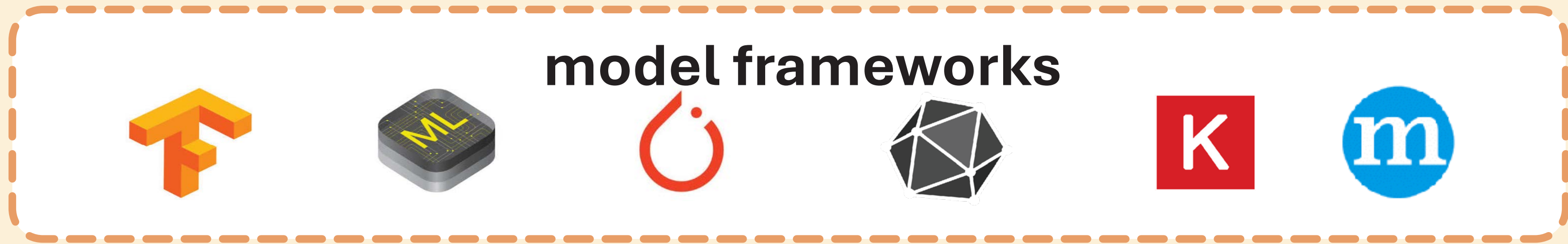
TVM

automates optimization

optimizies tensor operations

exposes computational graphs

hardware deployable module



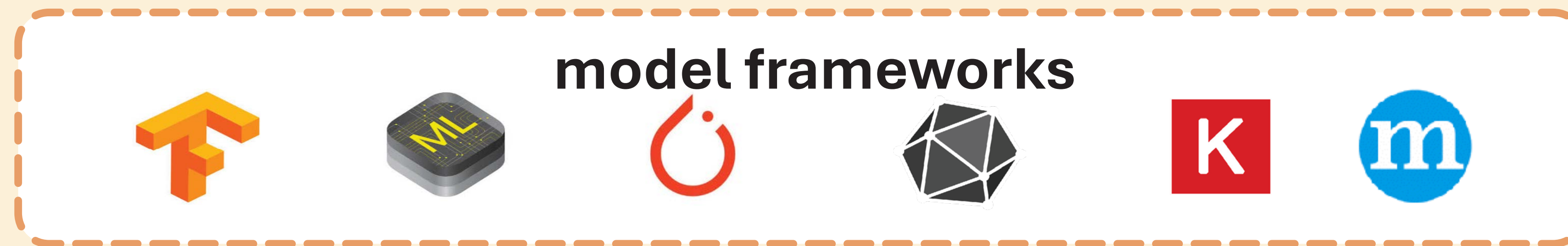
TVM

compiles

automates optimization

optimizes tensor operations

exposes computational graphs

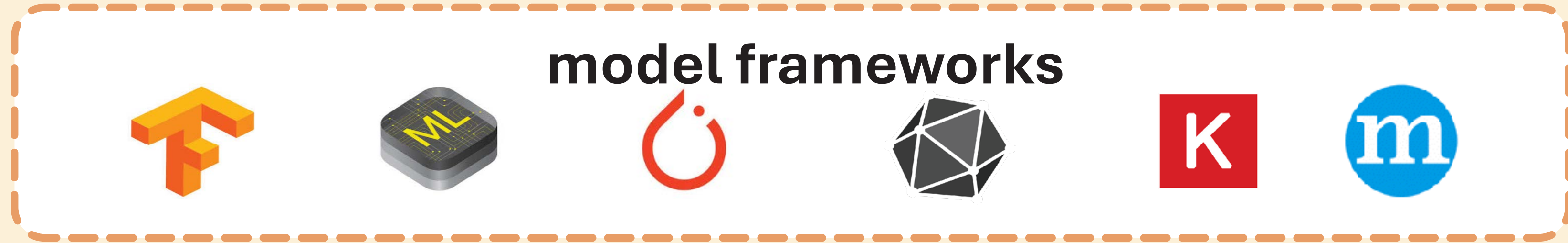


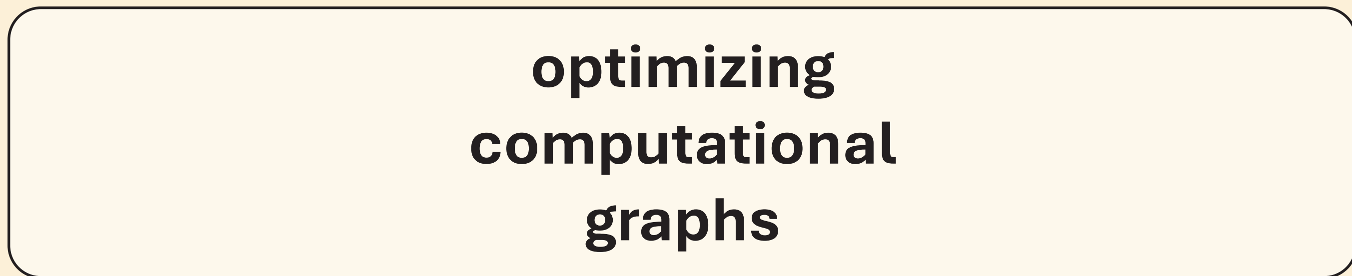
TVM

a compiler that ...

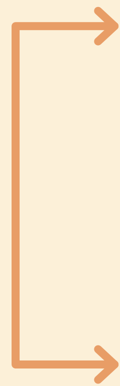
exposes computational graphs
searches for optimized tensor operations
automates the entire pipeline => ML
generates low-level programs for deployment

input

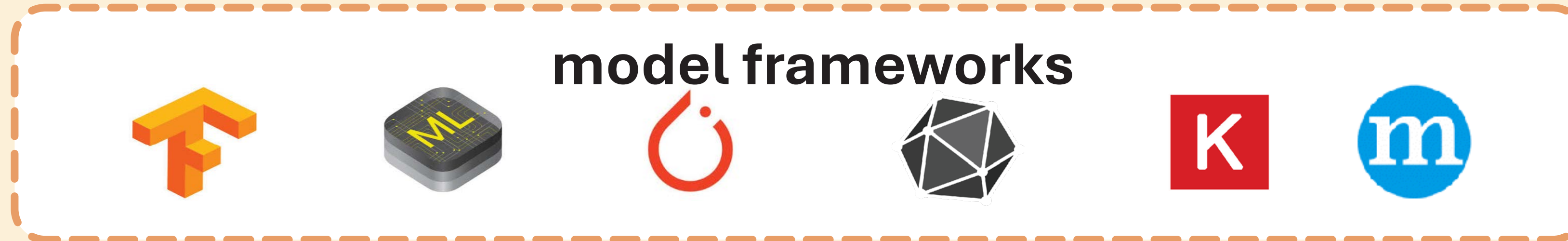




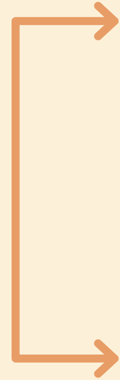
transform

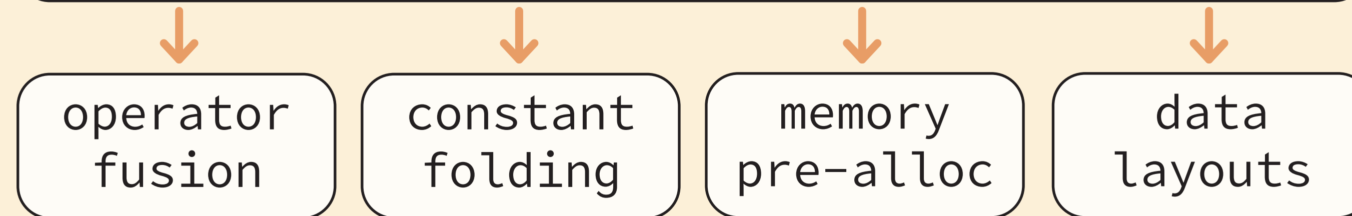


input

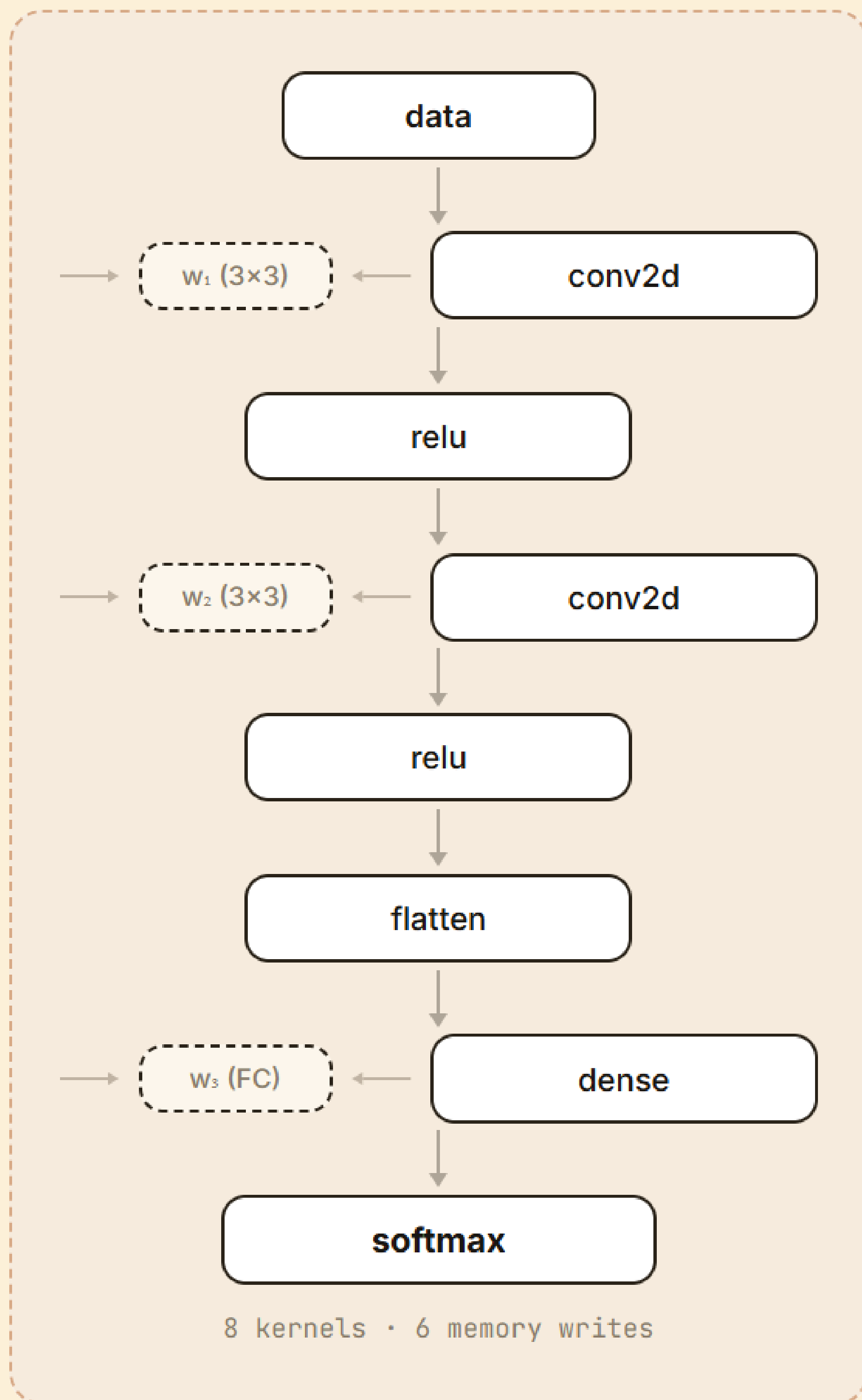


transform



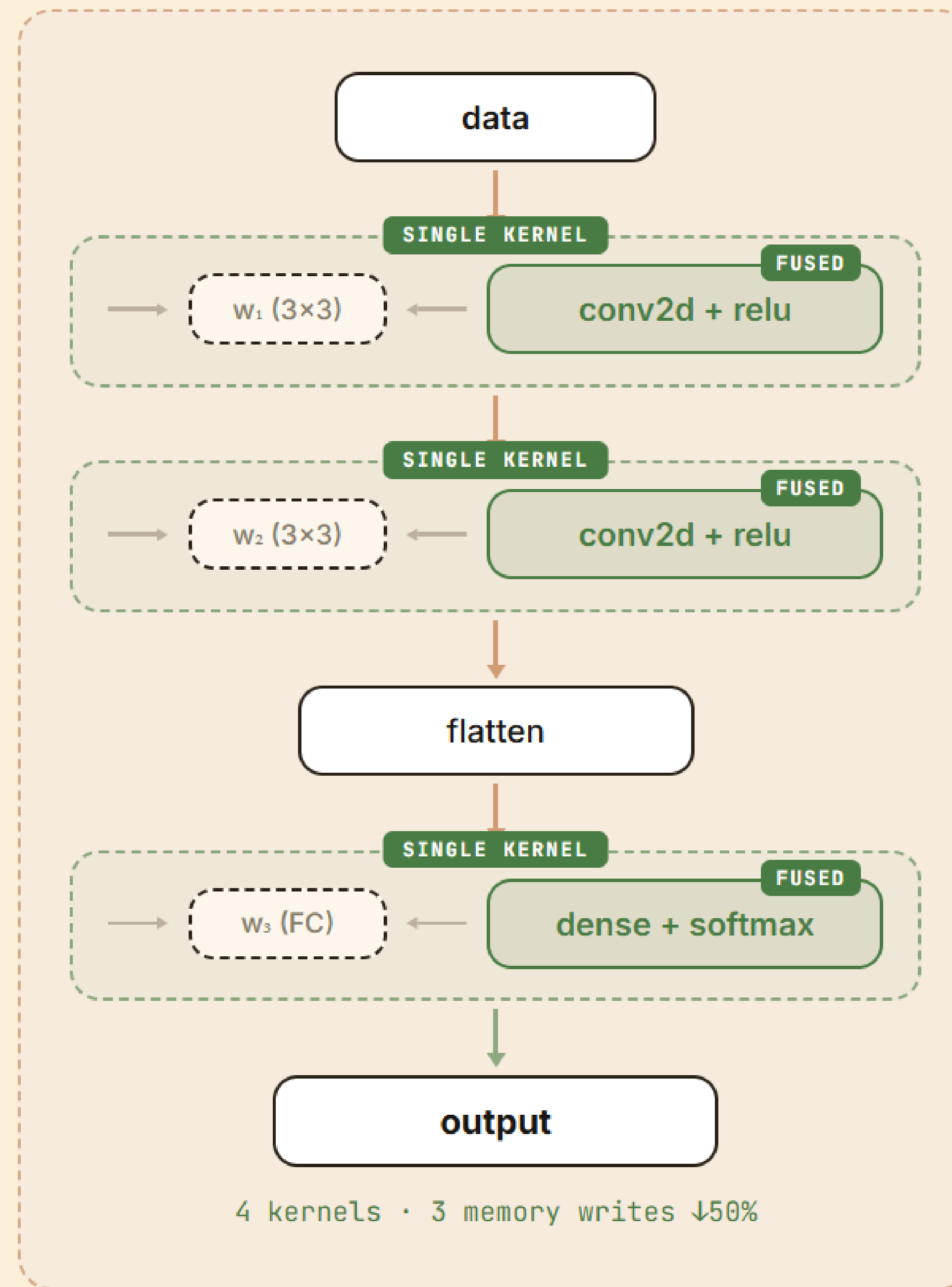


naive graph

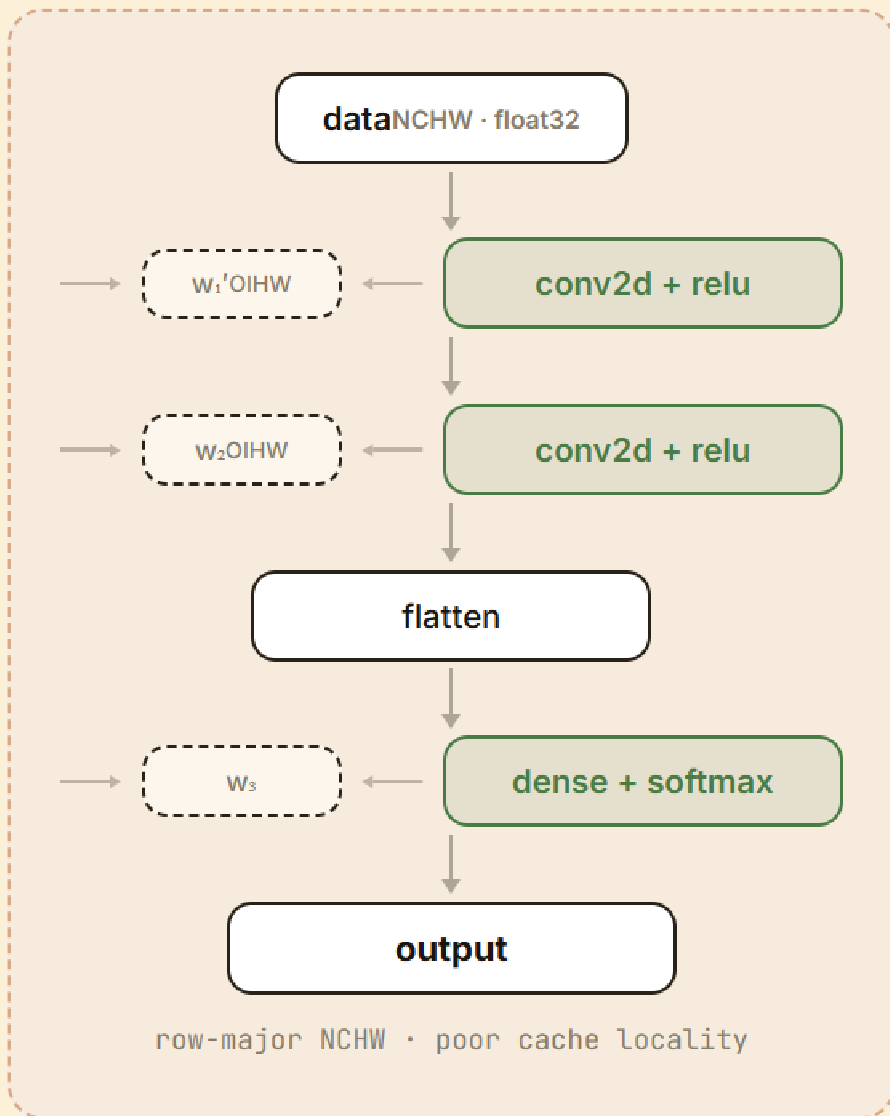


operator fusion

optimized graph

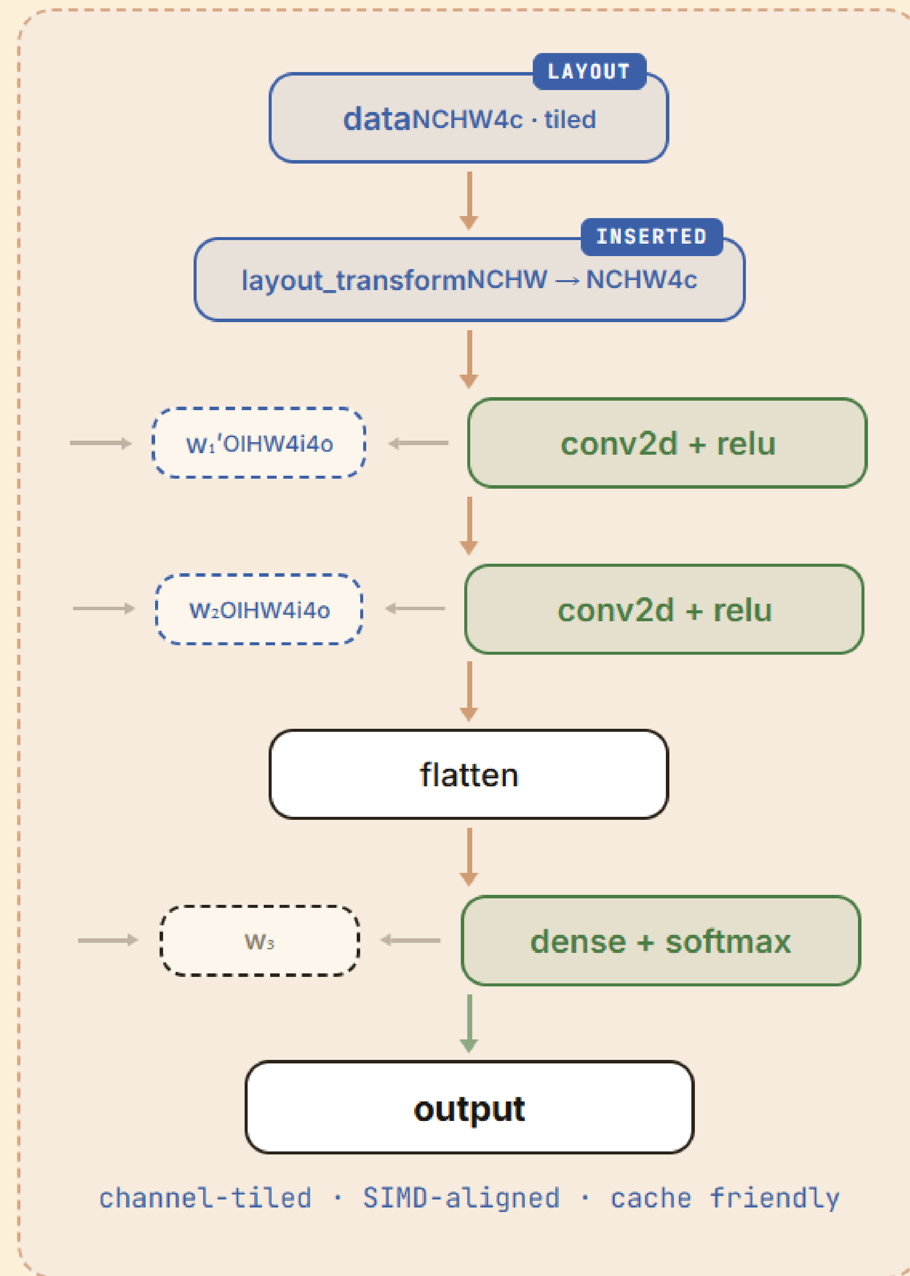


naive graph

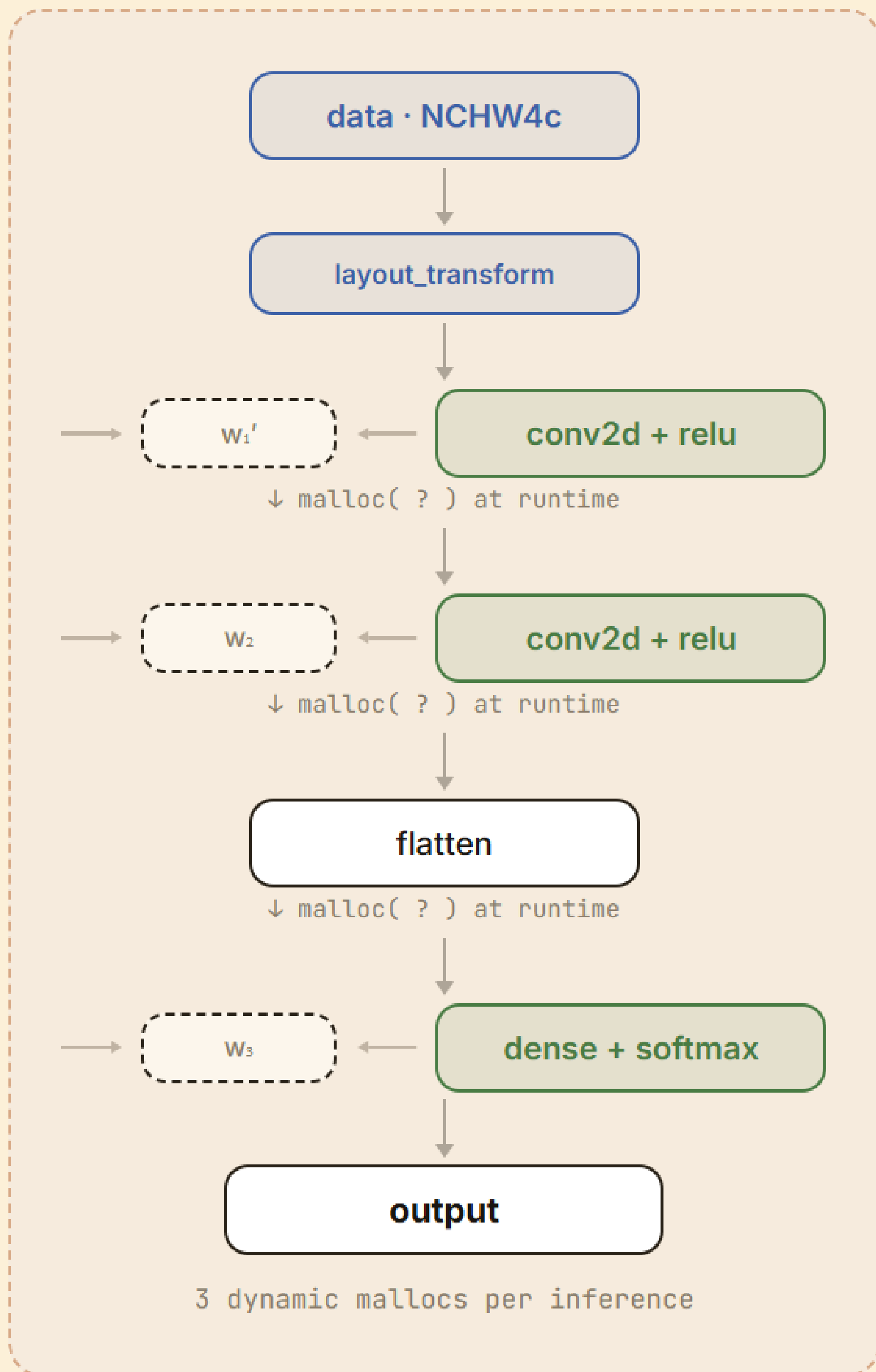


data layout transformation

optimized graph

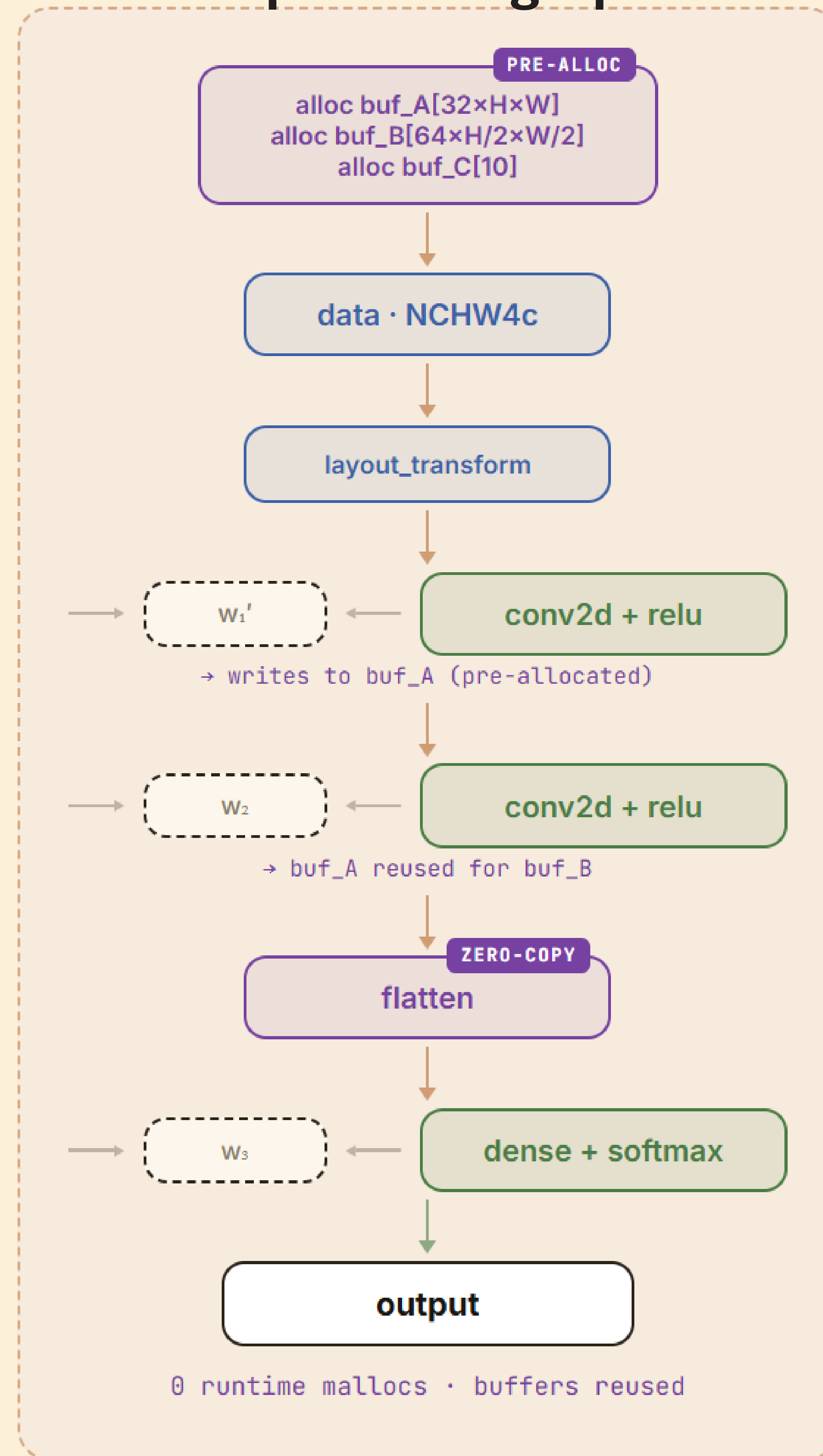


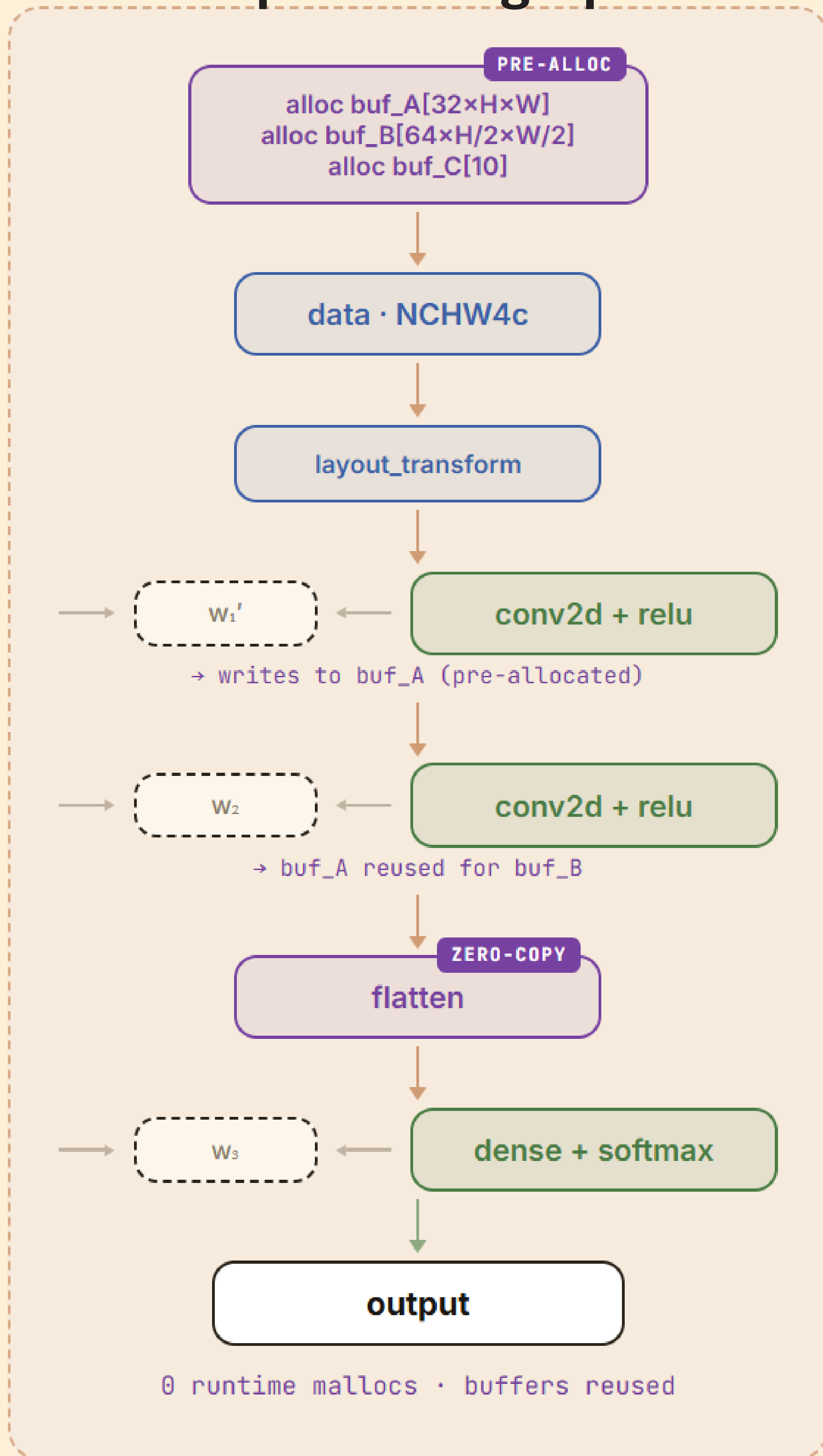
naive graph

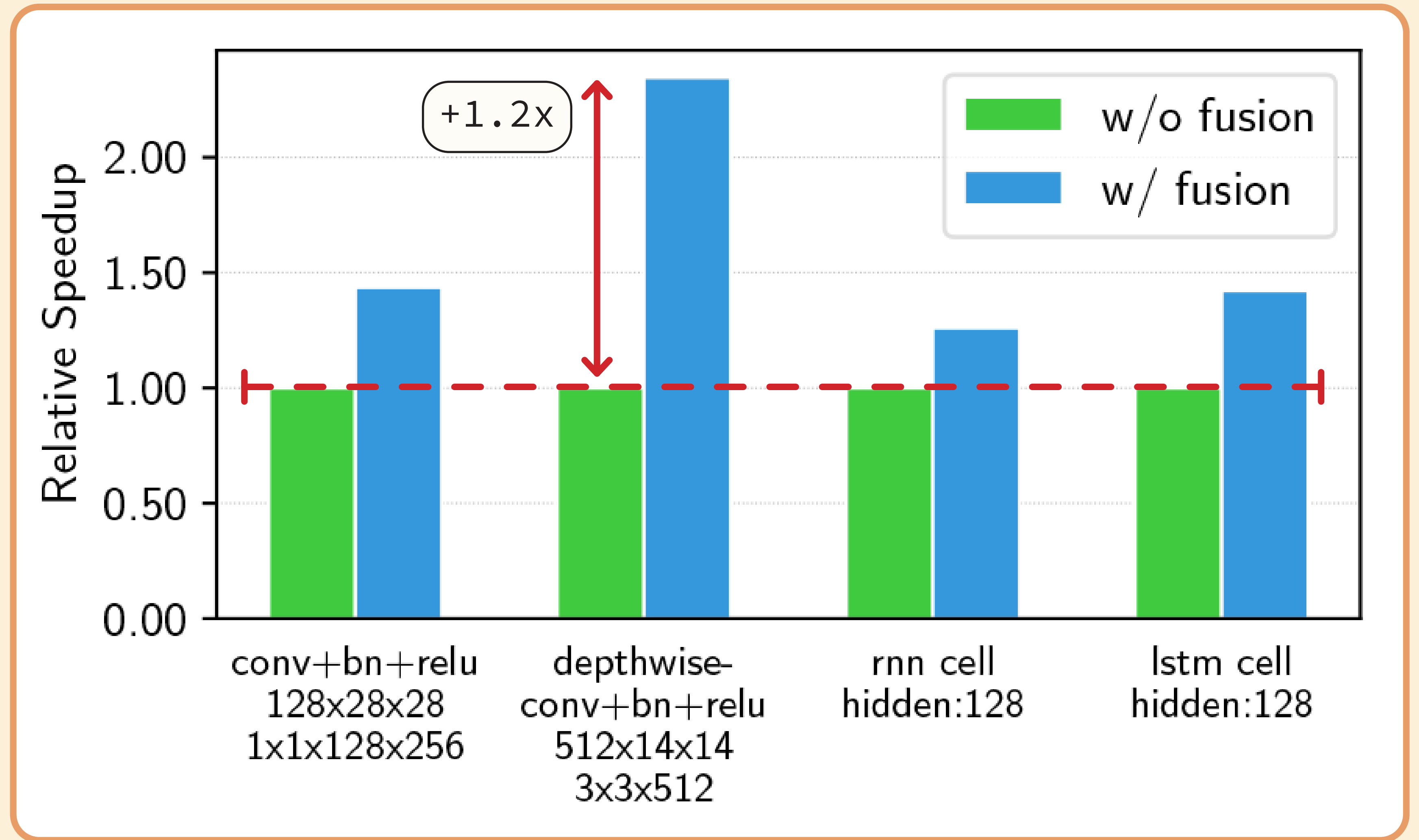


memory planning

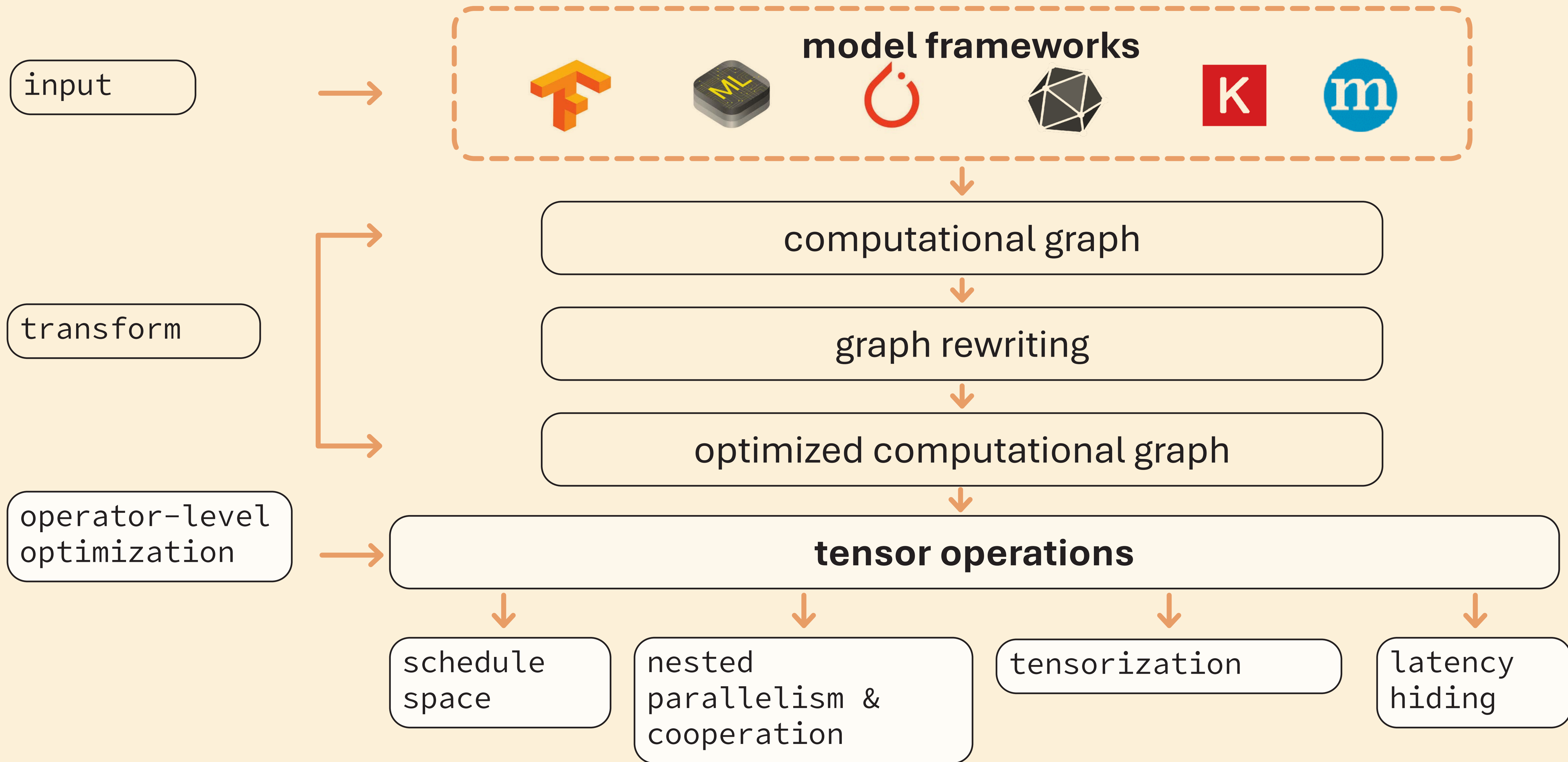
optimized graph







1.2x to 2x speedup by operator reducing memory access


input
model frameworks
K
transform
computational graph
graph rewriting
optimized computational graph
operator-level optimization
tensor operations
schedule space
nested parallelism & cooperation
tensorization
latency hiding

tensor expression language and schedule space

```python
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)
```

```python
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)
```

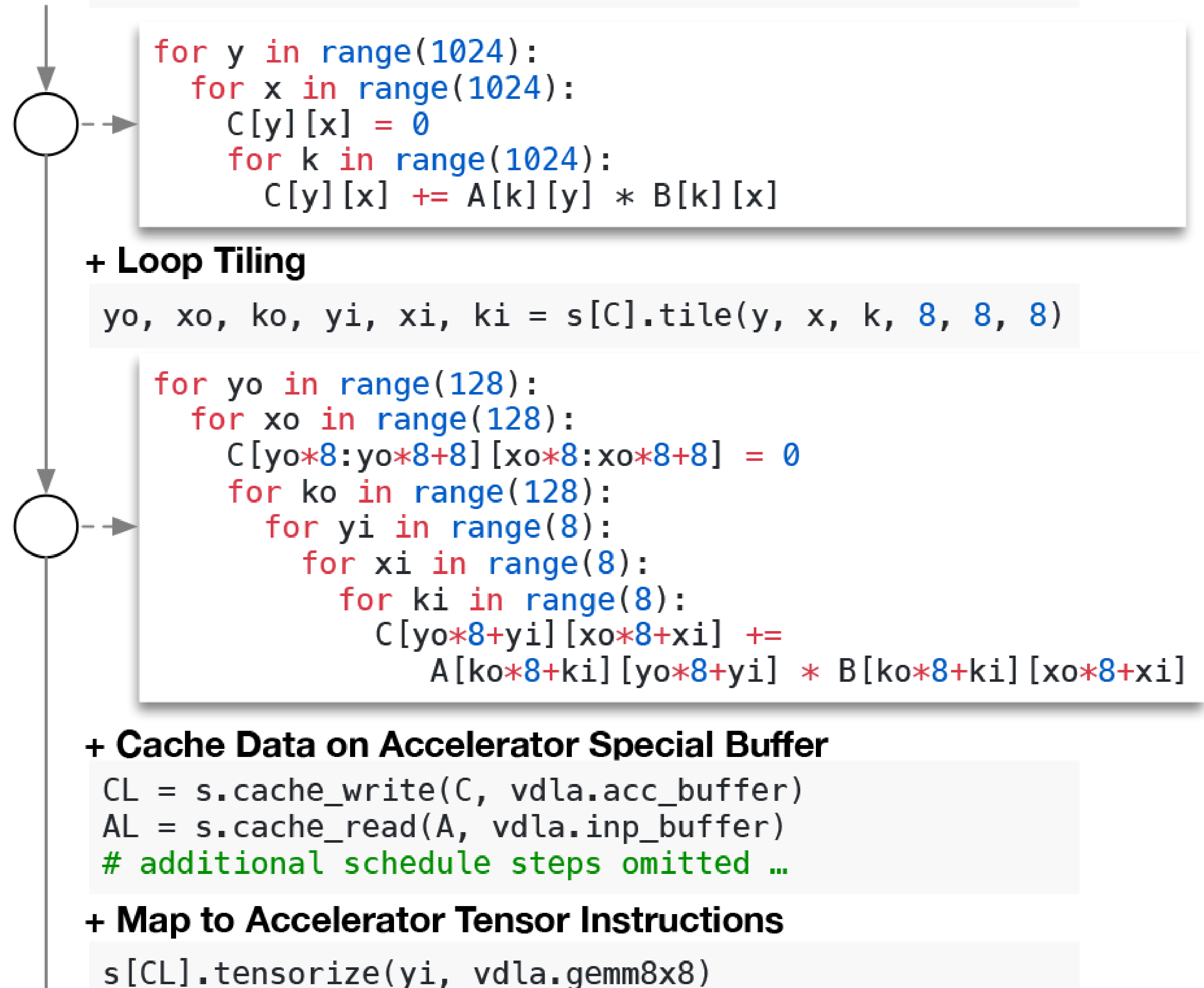
tensor expression language and schedule space

```
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)
```

```
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 and schedule space

schedule primitives



tensor expression language and schedule space

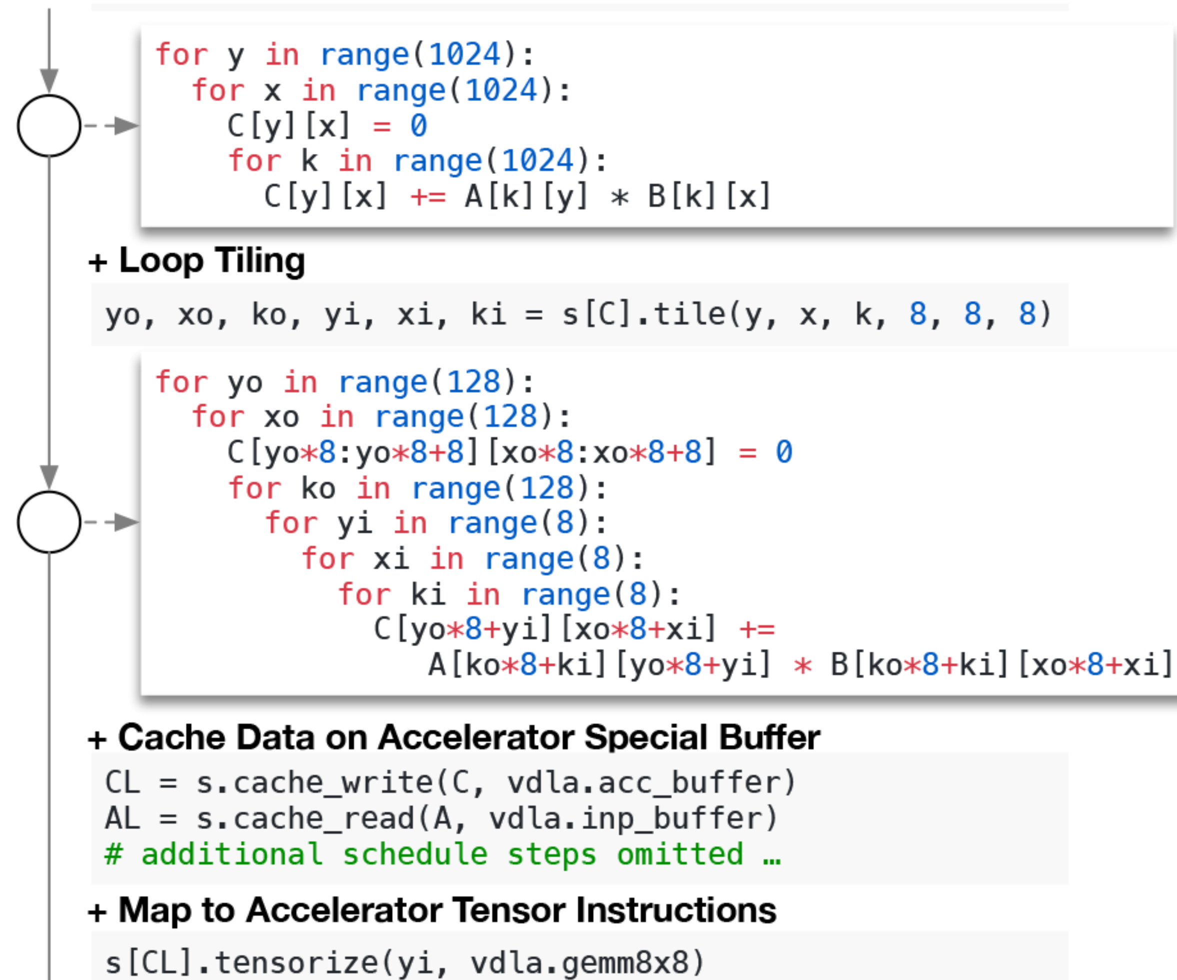
optimized tensor algorithm

```
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)
```

naive

```
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)
```

schedule primitives



optimized

```
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)
```

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]
```

memory scopes

nested parallelism with cooperation

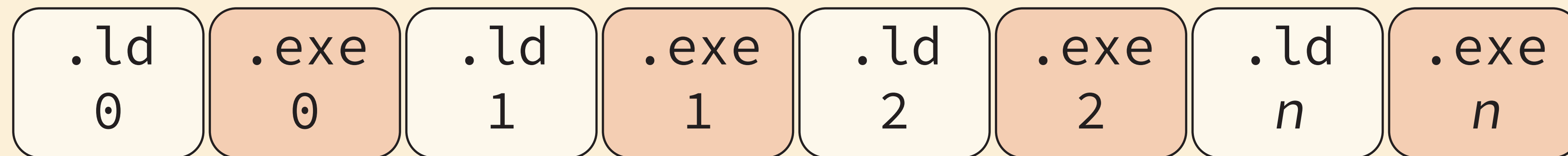
```
for thread_group (by, bx) in cross(64, 64):
  for thread_item (ty, tx) in cross(2, 2):
    local CL[8][8] = 0
    shared AS[2][8], BS[2][8]
    for k in range(1024):
      for i in range(4):
        AS[ty][i*4+tx] = A[k][by*64+ty*8+i*4+tx]
      for each i in 0..4:
        BS[ty][i*4+tx] = B[k][bx*64+ty*8+i*4+tx]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

memory scopes

explicit memory declaration

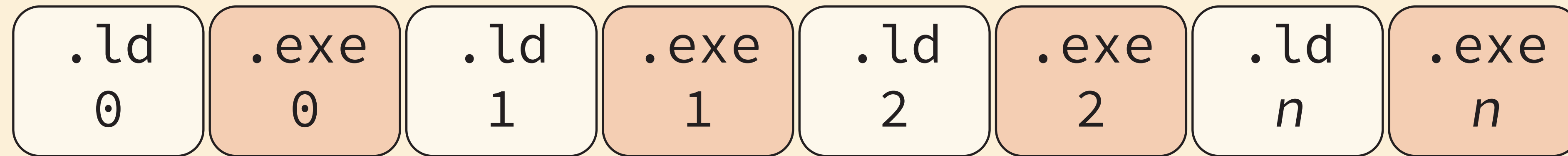
memory barrier is an automated insertion

latency hiding

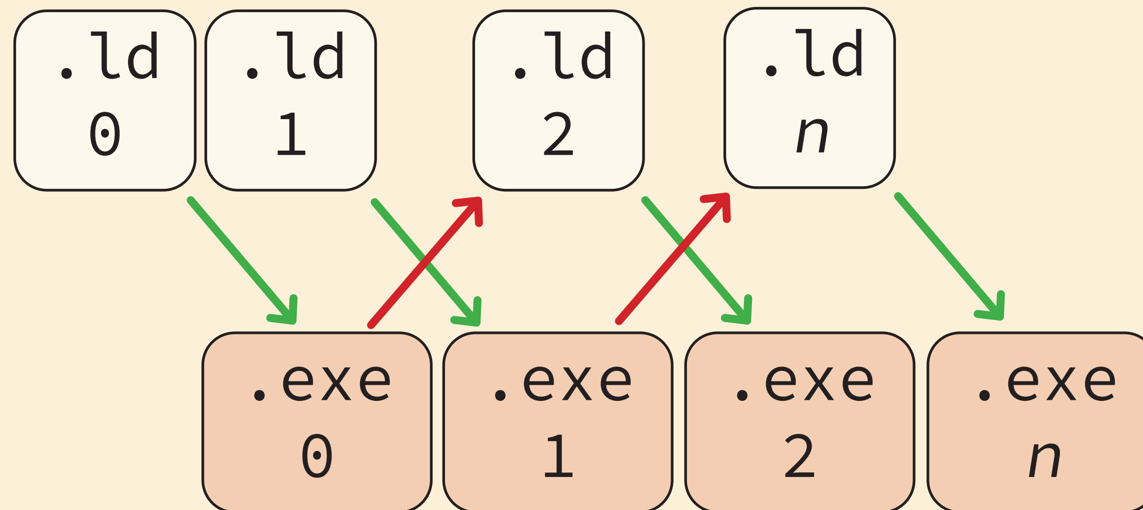


unoptimized pipeline

latency hiding

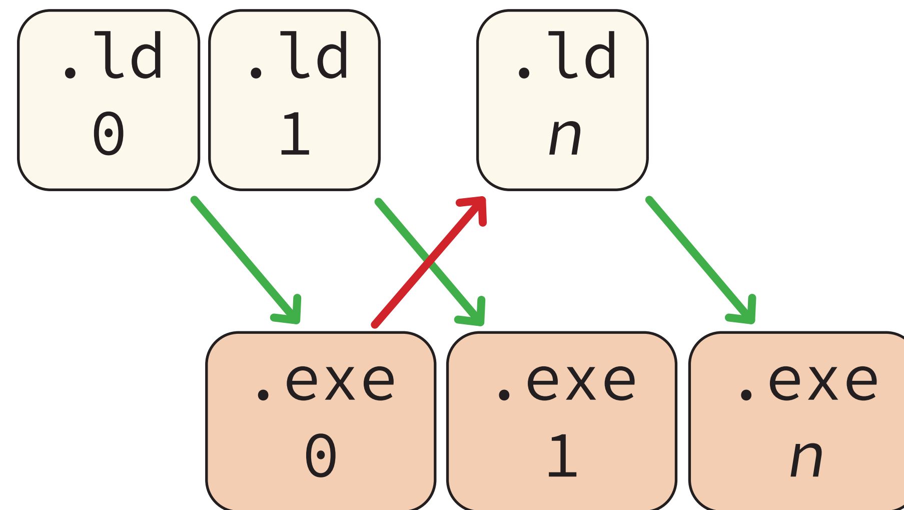


unoptimized pipeline

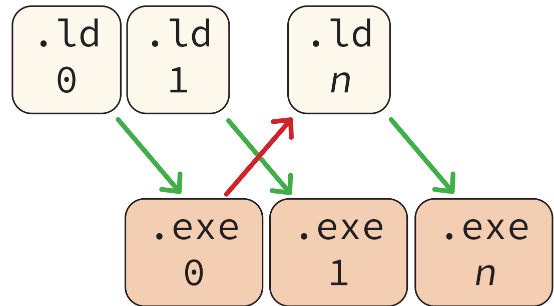


with latency hiding

latency hiding



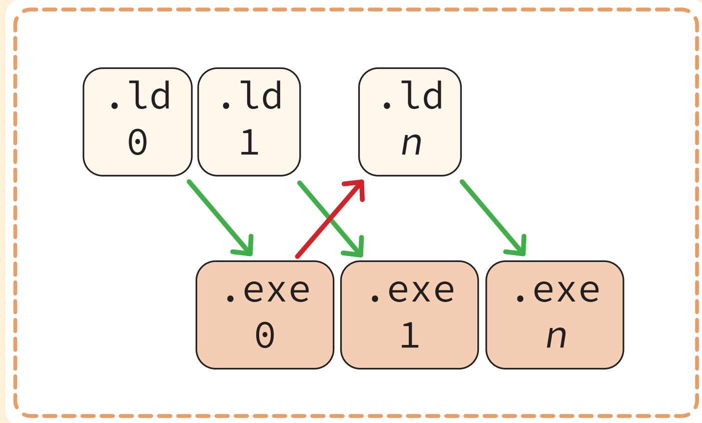
latency hiding

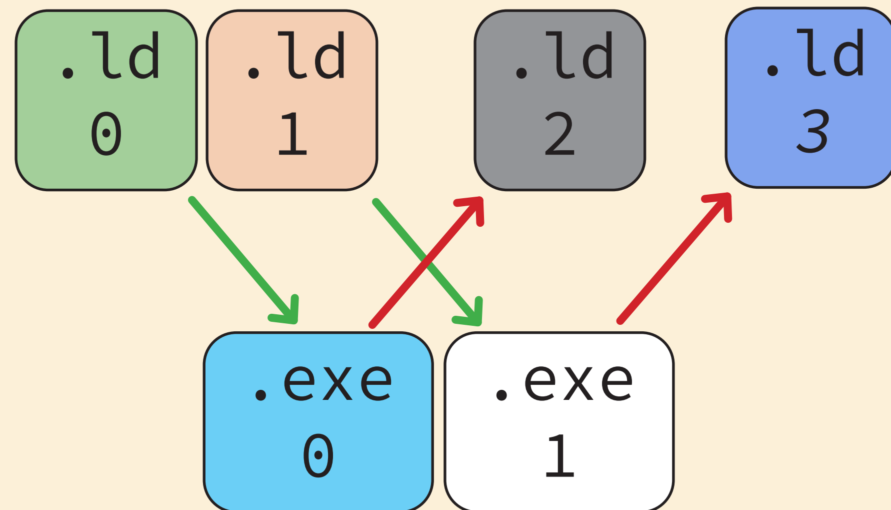


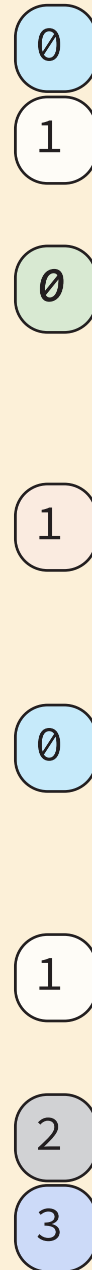

```
for vthread tx in range(2):
    acc_buffer CL[8]
    inp_buffer AL[8]
    for k in range(128):
        ld.dma_copy2d(AL,AL[k] [tx*8:tx*8+8])
        ex.accumulate(AL, CL)
```

```
for vthread tx in range(2):
    acc_buffer CL[8]
    inp_buffer AL[8]
    ex.push_dep_to(ld)
    for k in range(128):
        ld.pop_dep_from(ex)
        ld.dma_copy2d(AL,AL[k] [tx*8:tx*8+8])
        ld.push_dep_to(ex)

        ex.pop_dep_from(ld)
        ex.accumulate(AL, CL)
        ex.push_dep_to(ld)

    ld.pop_dep_from(ex)
```

latency hiding

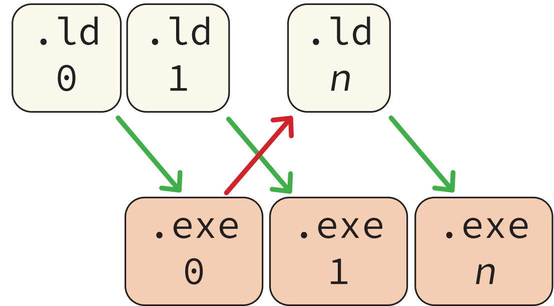


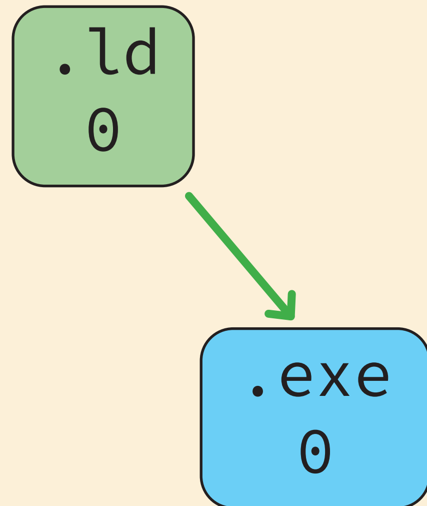





```
acc_buffer CL[2][8]
inp_buffer AL[2][8]
ex.push_dep_to(ld)
ex.push_dep_to(ld)
for k in range(128):
    ld.pop_dep_from(ex)
    ld.dma_copy2d(AL[0],AL[k][0:8])
    ld.push_dep_to(ex)

    ld.pop_dep_from(ex)
    ld.dma_copy2d(AL[1],AL[k][8:16])
    ld.push_dep_to(ex)

    ex.pop_dep_from(ld)
    ex.accumulate(AL[0], CL[0])
    ex.push_dep_to(ld)

    ex.pop_dep_from(ld)
    ex.accumulate(AL[1], CL[1])
    ex.push_dep_to(ld)

ld.pop_dep_from(ex)
ld.pop_dep_from(ex)
```

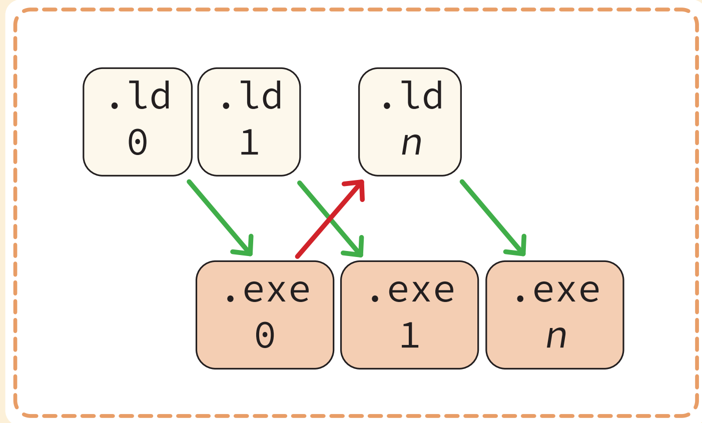
latency hiding

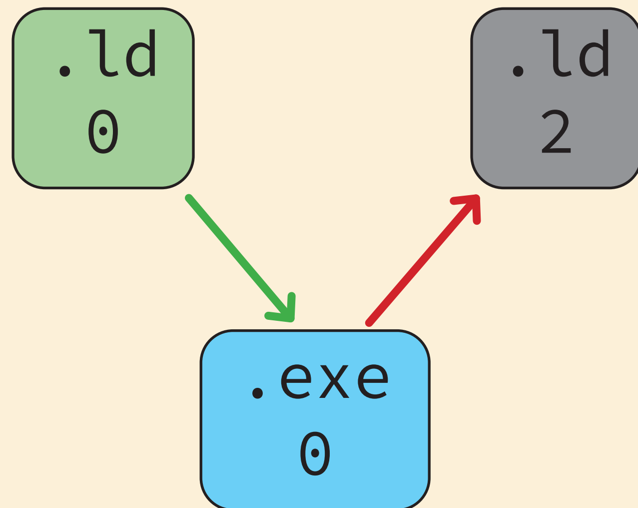




```
acc_buffer CL[2][8]
inp_buffer AL[2][8]
ex.push_dep_to(ld)
ex.push_dep_to(ld)
for k in range(128):
    ld.pop_dep_from(ex)
    ld.dma_copy2d(AL[0],AL[k][0:8])
    ld.push_dep_to(ex)

    ld.pop_dep_from(ex)
    ld.dma_copy2d(AL[1],AL[k][8:16])
    ld.push_dep_to(ex)

    ex.pop_dep_from(ld)
    ex.accumulate(AL[0], CL[0])
    ex.push_dep_to(ld)

    ex.pop_dep_from(ld)
    ex.accumulate(AL[1], CL[1])
    ex.push_dep_to(ld)

ld.pop_dep_from(ex)
ld.pop_dep_from(ex)
```

latency hiding





0

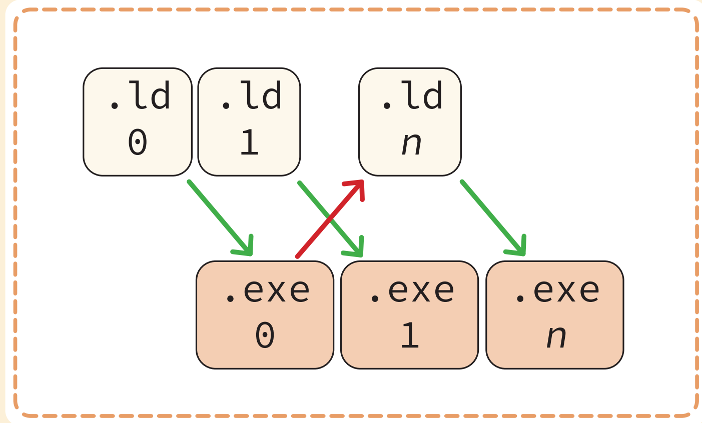
0

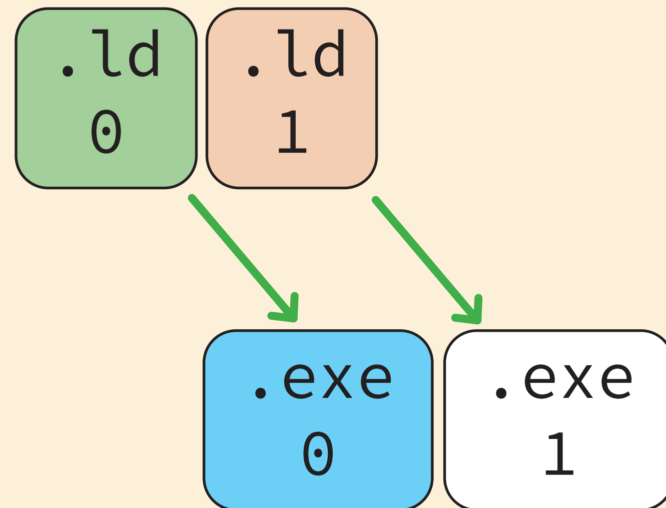
0

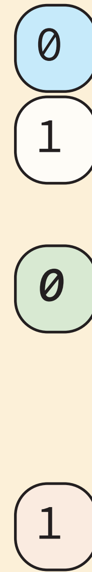
2

```
acc_buffer CL[2][8]
inp_buffer AL[2][8]
ex.push_dep_to(ld)
ex.push_dep_to(ld)
for k in range(128):
    ld.pop_dep_from(ex)
    ld.dma_copy2d(AL[0],AL[k][0:8])
    ld.push_dep_to(ex)

    ld.pop_dep_from(ex)
    ld.dma_copy2d(AL[1],AL[k][8:16])
    ld.push_dep_to(ex)

    ex.pop_dep_from(ld)
    ex.accumulate(AL[0], CL[0])
    ex.push_dep_to(ld)

    ex.pop_dep_from(ld)
    ex.accumulate(AL[1], CL[1])
    ex.push_dep_to(ld)

ld.pop_dep_from(ex)
ld.pop_dep_from(ex)
```

latency hiding

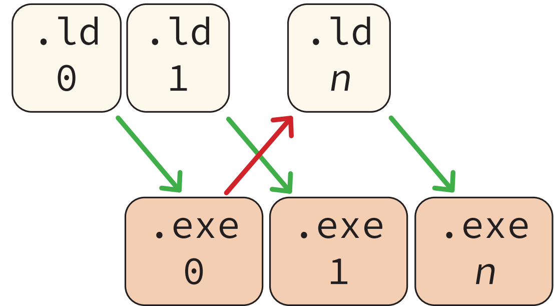


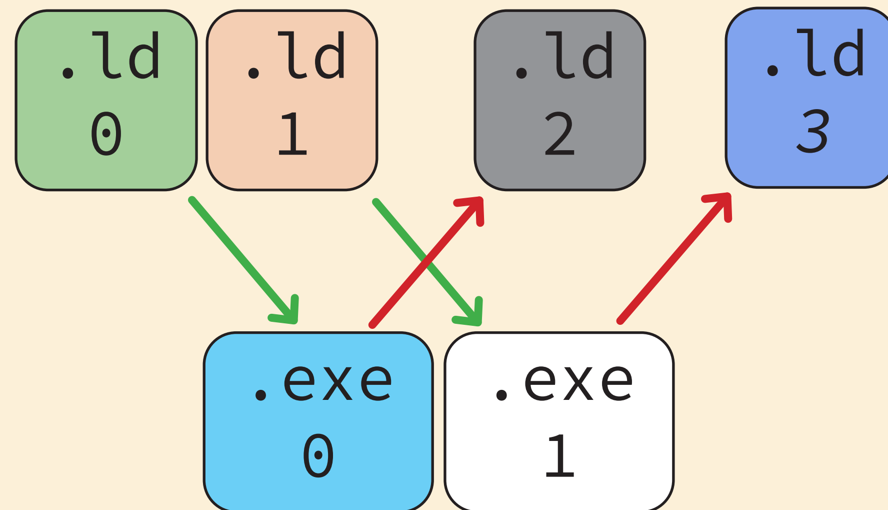


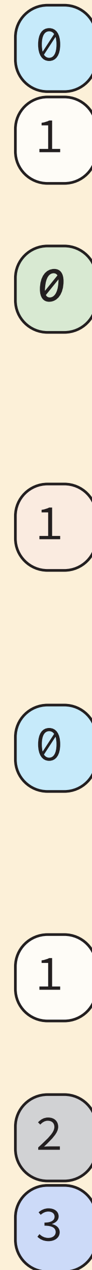



```
acc_buffer CL[2][8]
inp_buffer AL[2][8]
ex.push_dep_to(ld)
ex.push_dep_to(ld)
for k in range(128):
    ld.pop_dep_from(ex)
    ld.dma_copy2d(AL[0],AL[k][0:8])
    ld.push_dep_to(ex)

    ld.pop_dep_from(ex)
    ld.dma_copy2d(AL[1],AL[k][8:16])
    ld.push_dep_to(ex)

    ex.pop_dep_from(ld)
    ex.accumulate(AL[0], CL[0])
    ex.push_dep_to(ld)

    ex.pop_dep_from(ld)
    ex.accumulate(AL[1], CL[1])
    ex.push_dep_to(ld)

ld.pop_dep_from(ex)
ld.pop_dep_from(ex)
```

latency hiding

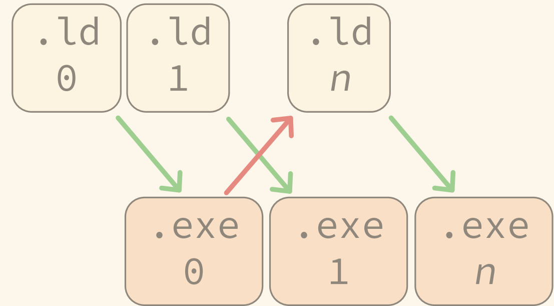


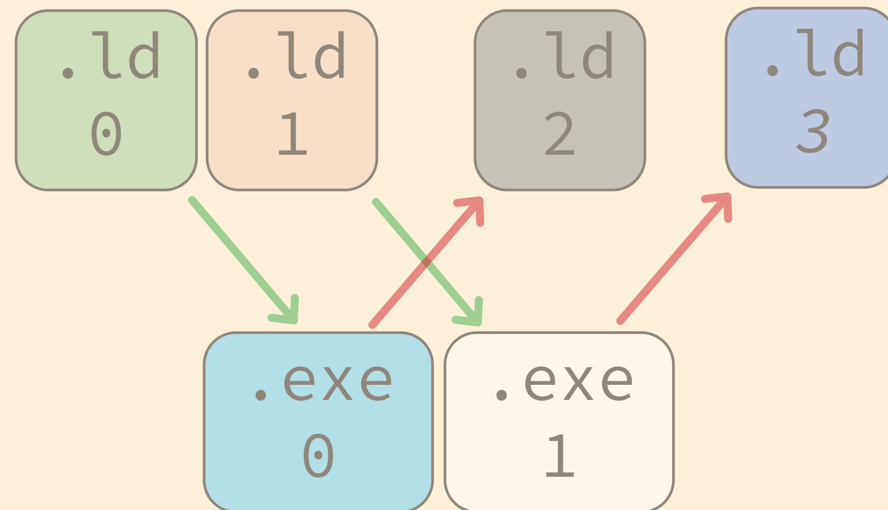




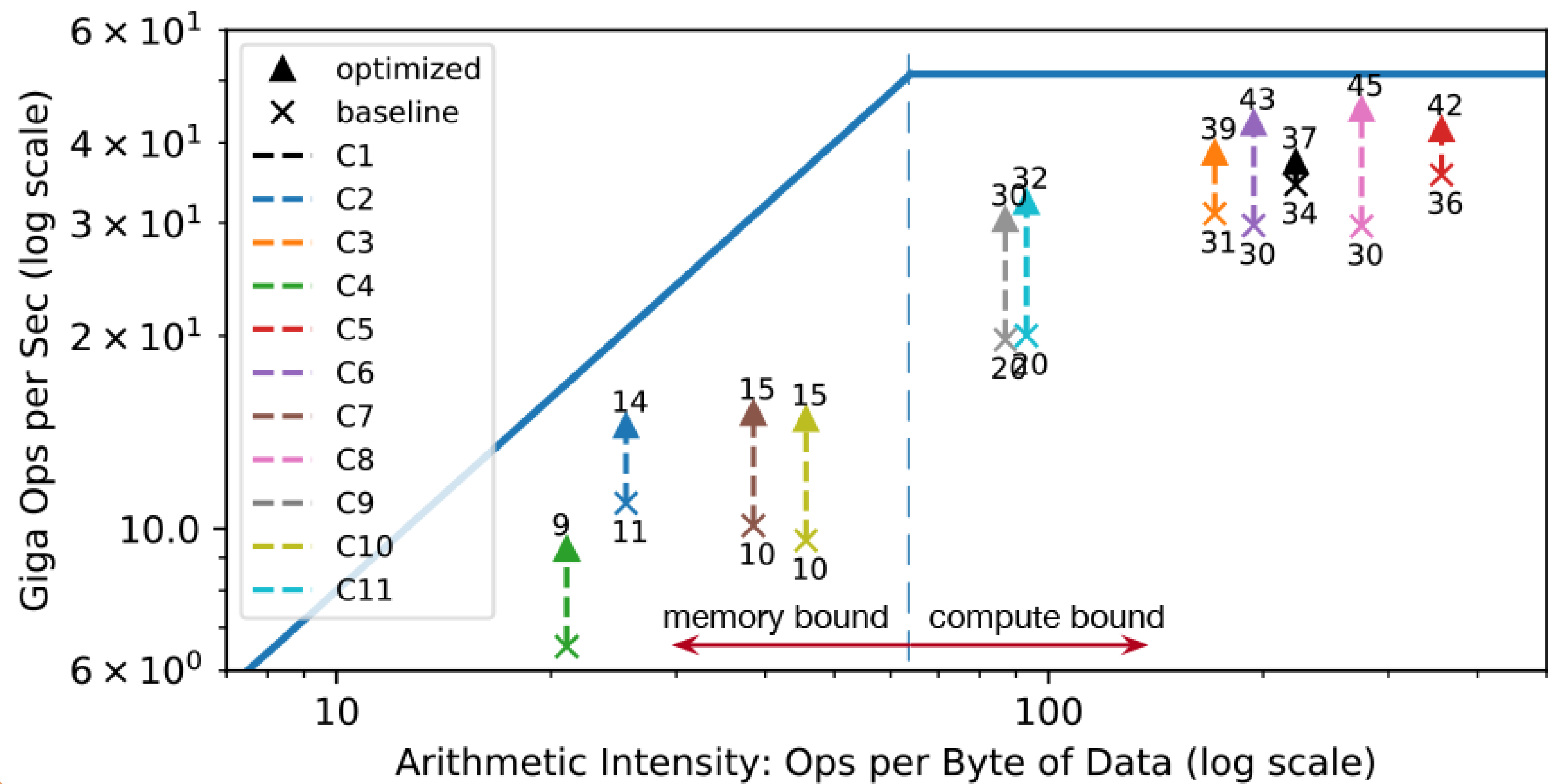
```
acc_buffer CL[2][8]
inp_buffer AL[2][8]
ex.push_dep_to(ld)
ex.push_dep_to(ld)
for k in range(128):
    ld.pop_dep_from(ex)
    ld.dma_copy2d(AL[0],AL[k][0:8])
    ld.push_dep_to(ex)

    ld.pop_dep_from(ex)
    ld.dma_copy2d(AL[1],AL[k][8:16])
    ld.push_dep_to(ex)

    ex.pop_dep_from(ld)
    ex.accumulate(AL[0], CL[0])
    ex.push_dep_to(ld)

    ex.pop_dep_from(ld)
    ex.accumulate(AL[1], CL[1])
    ex.push_dep_to(ld)

ld.pop_dep_from(ex)
ld.pop_dep_from(ex)
```

latency hiding

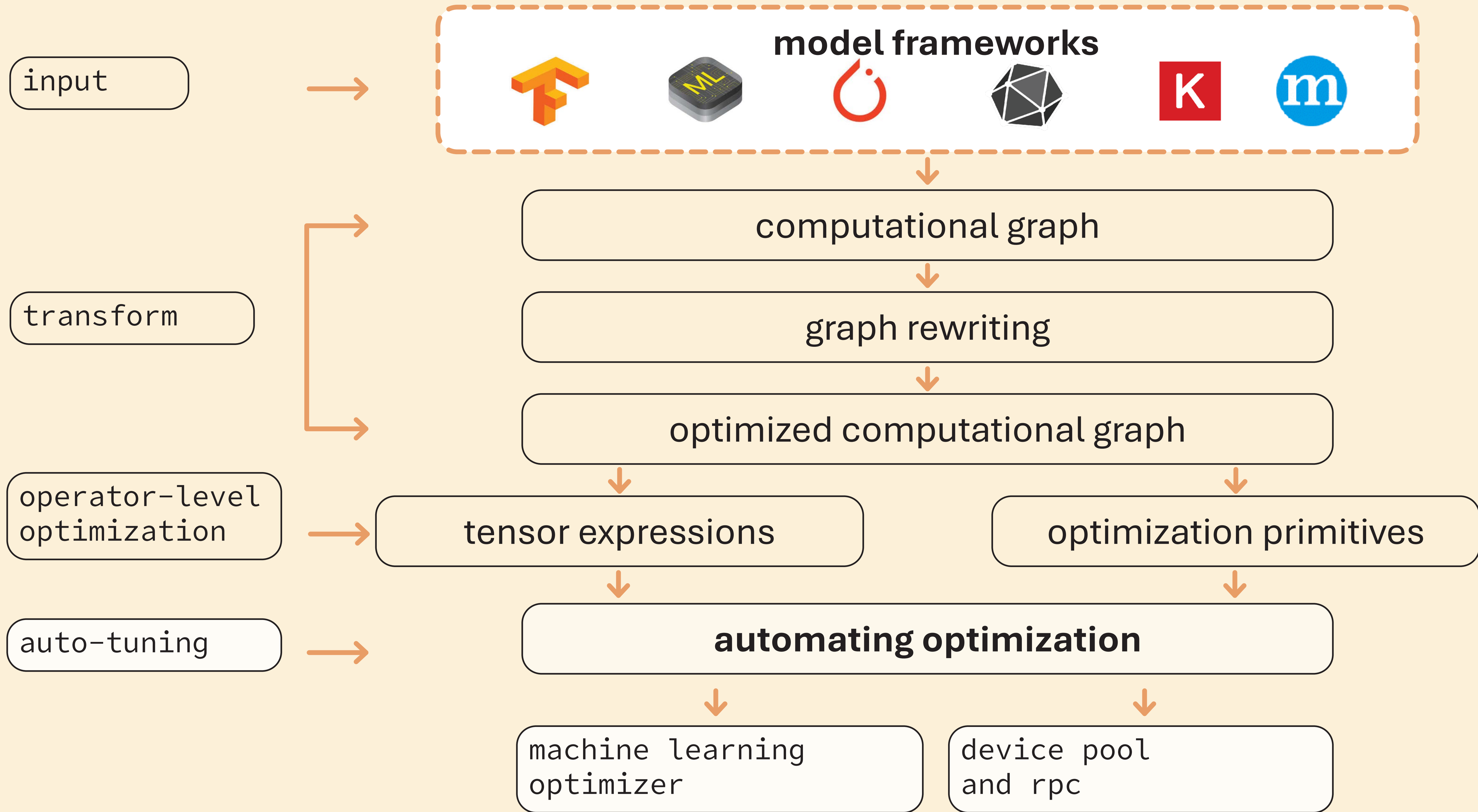




Name	Operator	H,W	IC,OC	K,S
C1	conv2d	224, 224	3,64	7, 2
C2	conv2d	56, 56	64,64	3, 1
C3	conv2d	56, 56	64,64	1, 1
C4	conv2d	56, 56	64,128	3, 2
C5	conv2d	56, 56	64,128	1, 2
C6	conv2d	28, 28	128,128	3, 1
C7	conv2d	28, 28	128,256	3, 2
C8	conv2d	28, 28	128,256	1, 2
C9	conv2d	14, 14	256,256	3, 1
C10	conv2d	14, 14	256,512	3, 2
C11	conv2d	14, 14	256,512	1, 2
C12	conv2d	7, 7	512,512	3, 1



performance on all ResNet layers increased from 70% to 88% with latency hiding


input
model frameworks
ML
K
m
transform
computational graph
graph rewriting
optimized computational graph
operator-level
optimization
tensor expressions
optimization primitives
auto-tuning
automating optimization
machine learning
optimizer
device pool
and rpc

ML-Based cost model

schedule exploration

query: Loop AST

`AST: Abstract Syntax Tree`

feature extraction

`loop annotations`

`memory access counts`

backend tracker

ML-Based cost model

schedule exploration

↓

query: Loop AST

`AST: Abstract Syntax Tree`

↓

feature extraction

`loop annotations`

`memory access counts`

↓

backend tracker

TreeRNN ← query: Loop AST

feature extraction → XGBoost

ML-Based cost model

schedule exploration

query: Loop AST

`AST: Abstract Syntax Tree`

TreeRNN

feature extraction

`loop annotations`

`memory access counts`

XGBoost

backend tracker

distributed device pool and RPC

RPC: REMOTE PROCEDURE CALL

schedule exploration

↓

query: Loop AST

↓

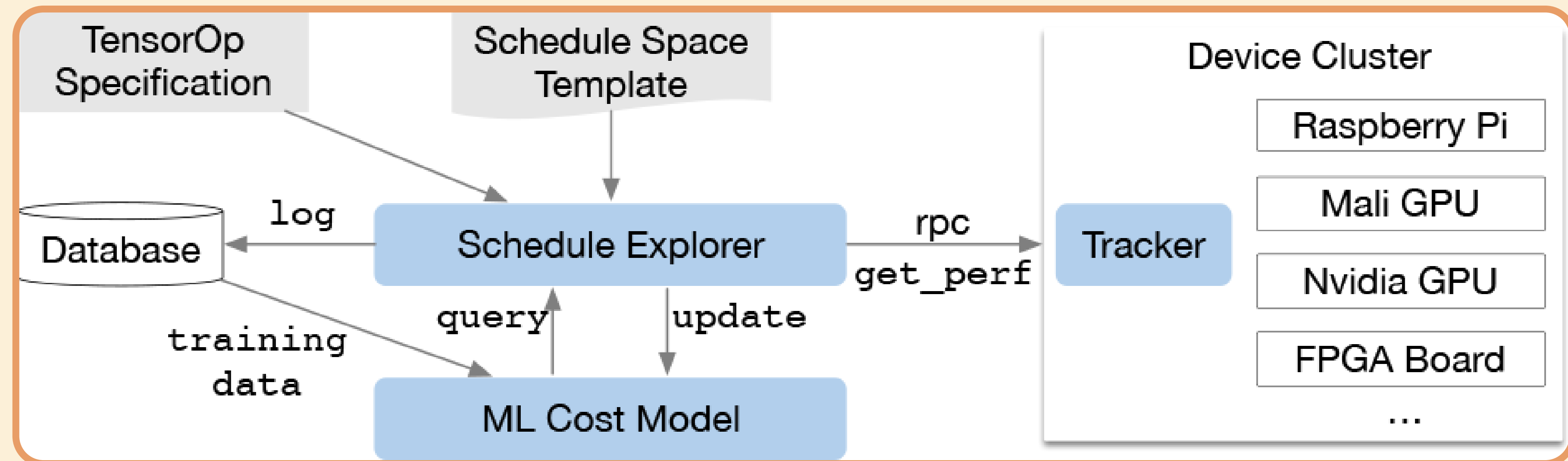
feature extraction

loop annotations

memory access counts

↓

backend tracker



performance

schedule exploration

↓

query: Loop AST

↓

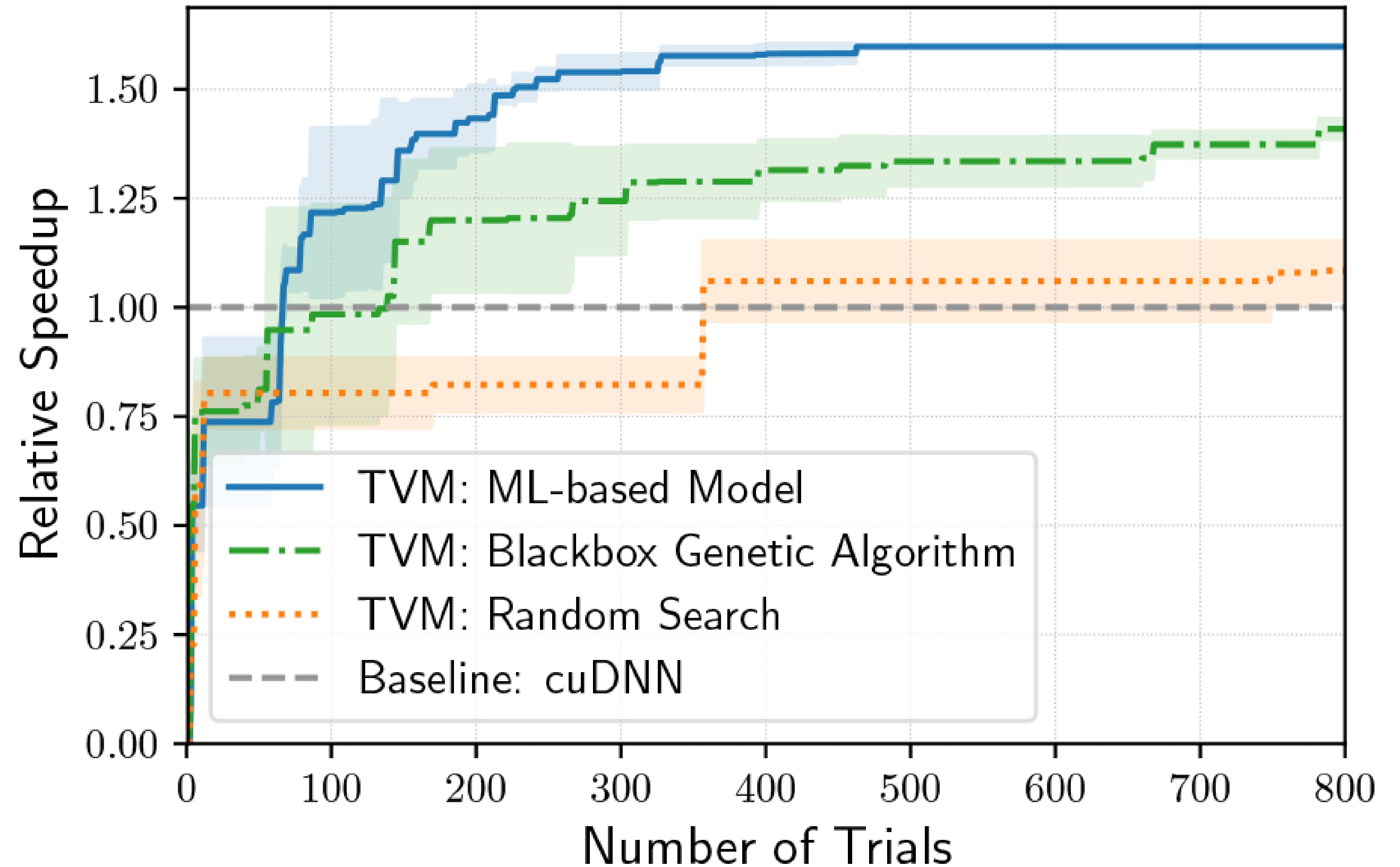
feature extraction

loop annotations

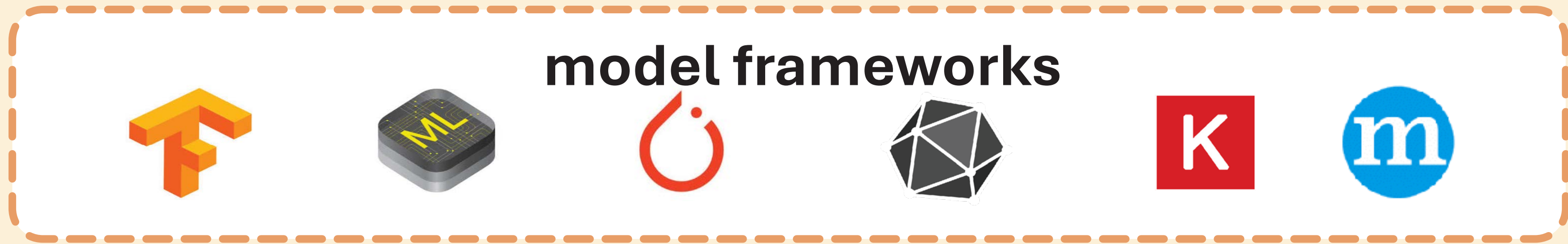
memory access counts

↓

backend tracker



1.5x speed up at 0.67ms delta per prediction (XGBoost)



TVM

computational graph

graph rewriting

optimized computational graph

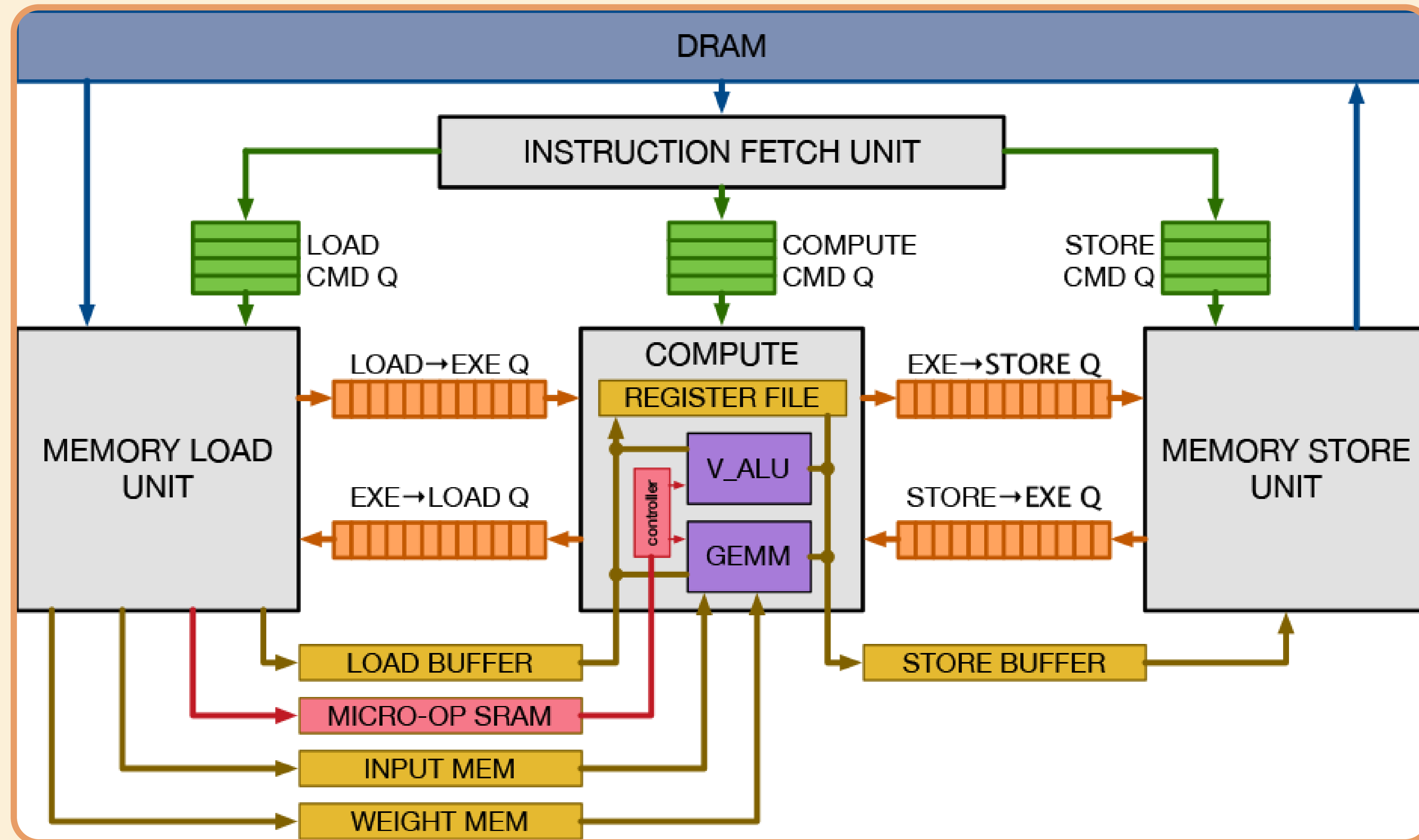
tensor expressions

optimization primitives

ML Automated Optimizer

optimized low level program

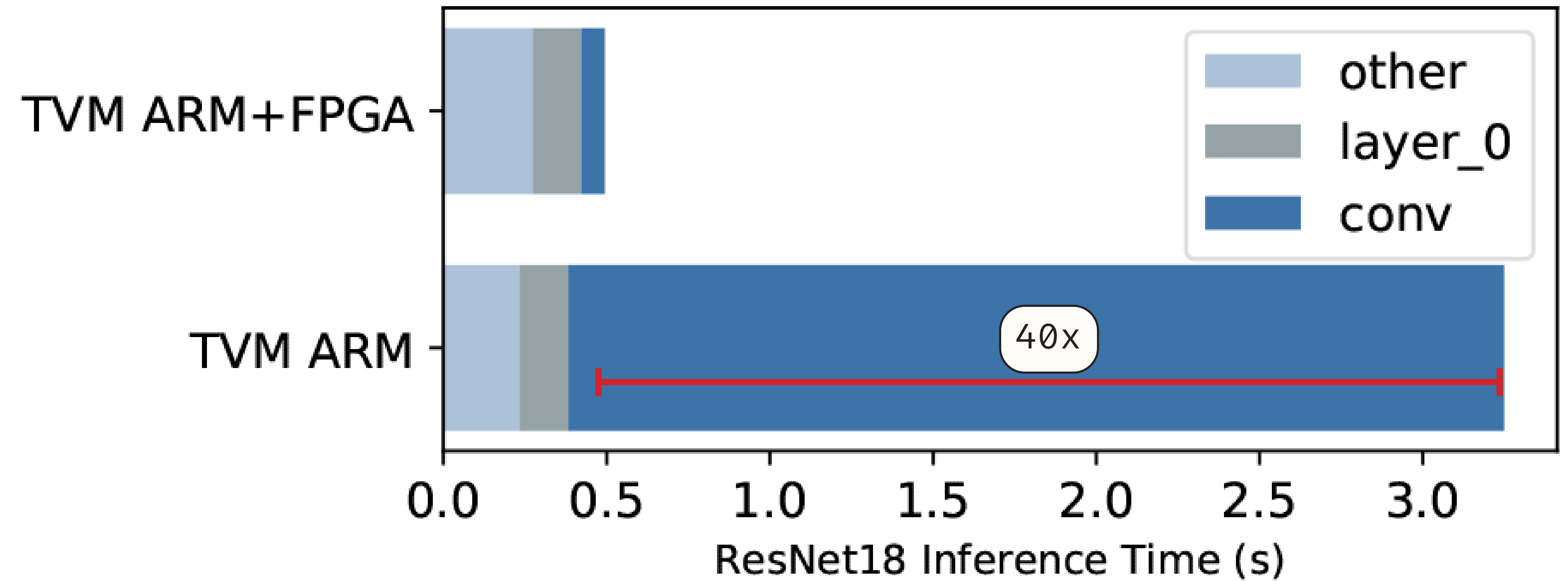
evaluation: vanilla deep learning accelerator



VDLA Hardware design overview

evaluation: vanilla deep learning accelerator

ResNet
cnn test



FPGA performance boost by 40x

```
faster computation;
```

user responsiveness

more work per time = scale models faster

```
portability:
```

deploy models to more hardware backends

```
cost:
```

automation reduces cost of development

possible next steps

Ansor: Generating High-Performance Tensor Programs for Deep Learning

Lianmin Zheng, *UC Berkeley;* Chengfan Jia, Minmin Sun, and Zhao Wu, *Alibaba Group;* Cody Hao Yu, *Amazon Web Services, Inc;* Ameer Haj-Ali, *UC Berkeley;* Yida Wang, *Amazon Web Services;* Jun Yang, *Alibaba Group;* Danyang Zhuo, *UC Berkeley and Duke University;* Koushik Sen, Joseph E. Gonzalez, and Ion Stoica, *UC Berkeley*

https://www.usenix.org/conference/osdi20/presentation/zheng

possible next steps

Ansor: Generating High-Performance Tensor Programs for Deep Learning

Lianmin Zheng, *UC Berkeley;* Chengfan Jia, Minmin Sun, and Zhao Wu, *Alibaba Group;* Cody Hao Yu, *Amazon Web Services, Inc;* Ameer Haj-Ali, *UC Berkeley;* Yida Wang, *Amazon Web Services;* Jun Yang, *Alibaba Group;* Danyang Zhuo, *UC Berkeley and Duke University;* Koushik Sen, Joseph E. Gonzalez, and Ion Stoica, *UC Berkeley*

https://www.usenix.org/conference/osdi20/presentation/zheng

ORCA: A Distributed Serving System for Transformer-Based Generative Models

Gyeong-In Yu and Joo Seong Jeong, *Seoul National University;* Geon-Woo Kim, *FriendliAI and Seoul National University;* Soojeong Kim, *FriendliAI;* Byung-Gon Chun, *FriendliAI and Seoul National University*

https://www.usenix.org/conference/osdi22/presentation/yu

possible next steps

Ansor: Generating High-Performance Tensor Programs for Deep Learning

Lianmin Zheng, *UC Berkeley;* Chengfan Jia, Minmin Sun, and Zhao Wu, *Alibaba Group;* Cody Hao Yu, *Amazon Web Services, Inc;* Ameer Haj-Ali, *UC Berkeley;* Yida Wang, *Amazon Web Services;* Jun Yang, *Alibaba Group;* Danyang Zhuo, *UC Berkeley and Duke University;* Koushik Sen, Joseph E. Gonzalez, and Ion Stoica, *UC Berkeley*

https://www.usenix.org/conference/osdi20/presentation/zheng

Orca: A D
Transfor

Gyeong-In
Geon-Woo Kim, *Frien*
Byung-G

https:/

Transfer Learning for Sequence Generation: from Single-source to Multi-source

Xuancheng Huang[1], Jingfang Xu[4], Maosong Sun[1,3], and Yang Liu[1,2,3]*
[1]Dept. of Comp. Sci. & Tech., BNRist Center, Institute for AI, Tsinghua University
[2]Institute for AI Industry Research, Tsinghua University, Beijing, China
[3]Beijing Academy of Artificial Intelligence
[4]Sogou Inc., Beijing, China