VTA: Open & Flexible DL Acceleration

Thierry Moreau

TVM Conference, Dec 12th 2018

sampl

PAUL G. ALLEN SCHOOL OF COMPUTER SCIENCE & ENGINEERING

tvm.ai

TVM Stack

High-Level Differentiable IR

Tensor Expression IR

LLVM | CUDA | Metal

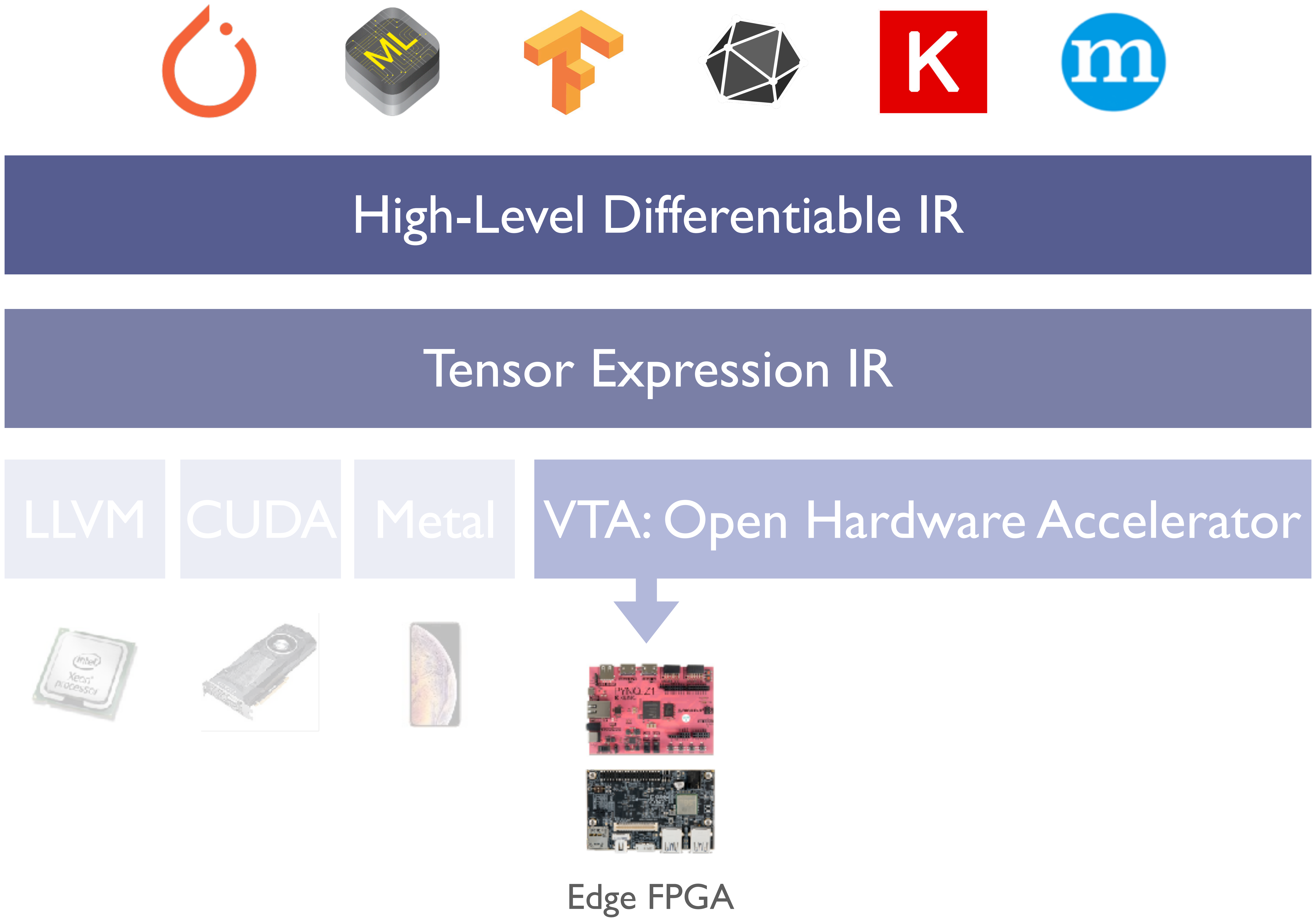
TVM Stack

High-Level Differentiable IR

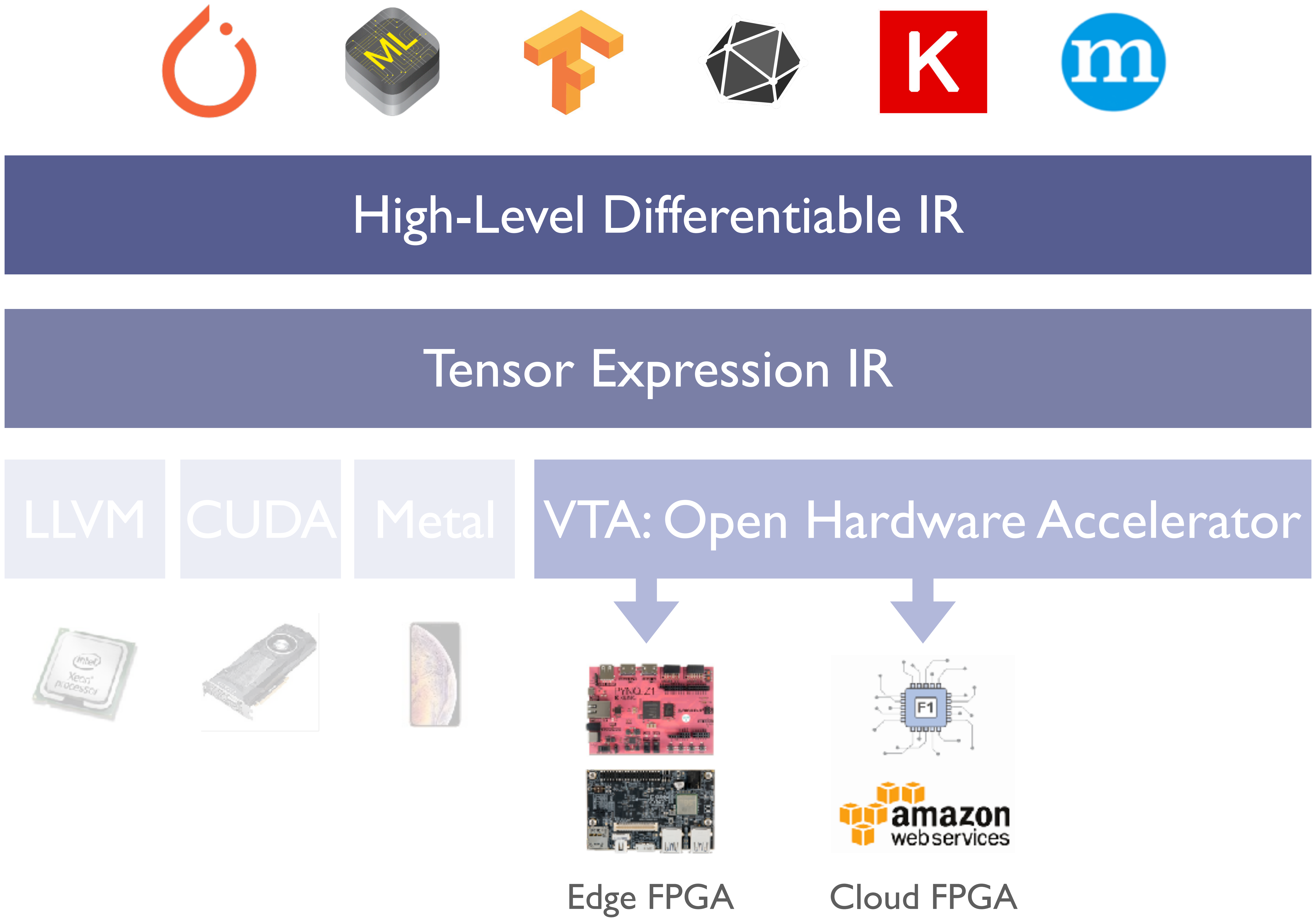
Tensor Expression IR

LLVM | CUDA | Metal | VTA: Open Hardware Accelerator

TVM Stack


High-Level Differentiable IR
Tensor Expression IR
LLVM
CUDA
Metal
VTA: Open Hardware Accelerator
Edge FPGA

TVM Stack



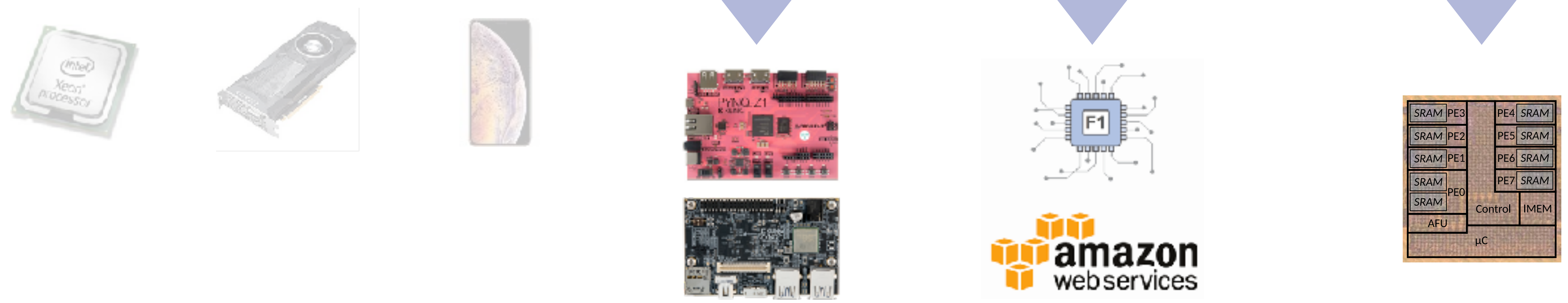
High-Level Differentiable IR
Tensor Expression IR
LLVM
CUDA
Metal
VTA: Open Hardware Accelerator
Edge FPGA
Cloud FPGA

TVM Stack

High-Level Differentiable IR

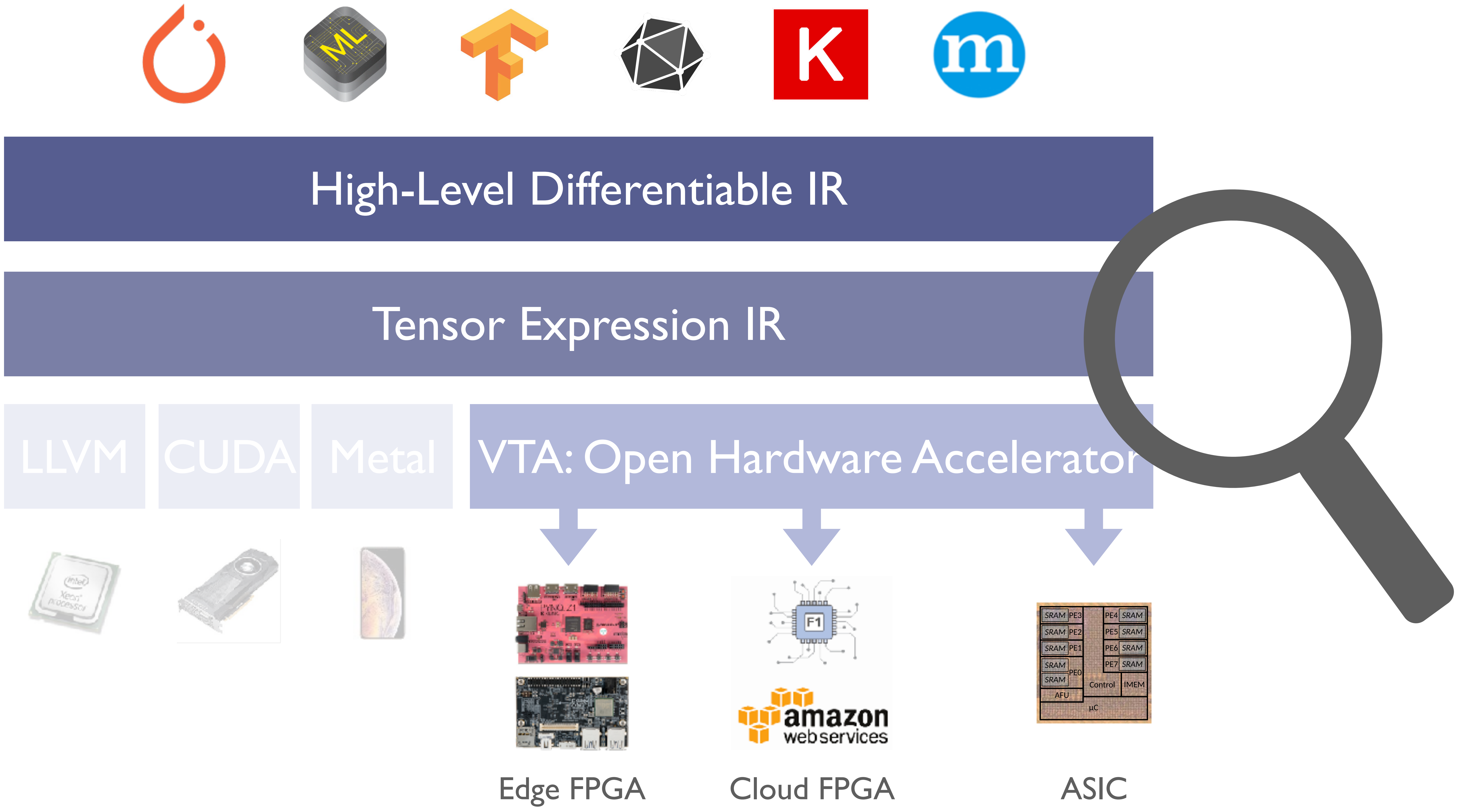
Tensor Expression IR

LLVM | CUDA | Metal | VTA: Open Hardware Accelerator

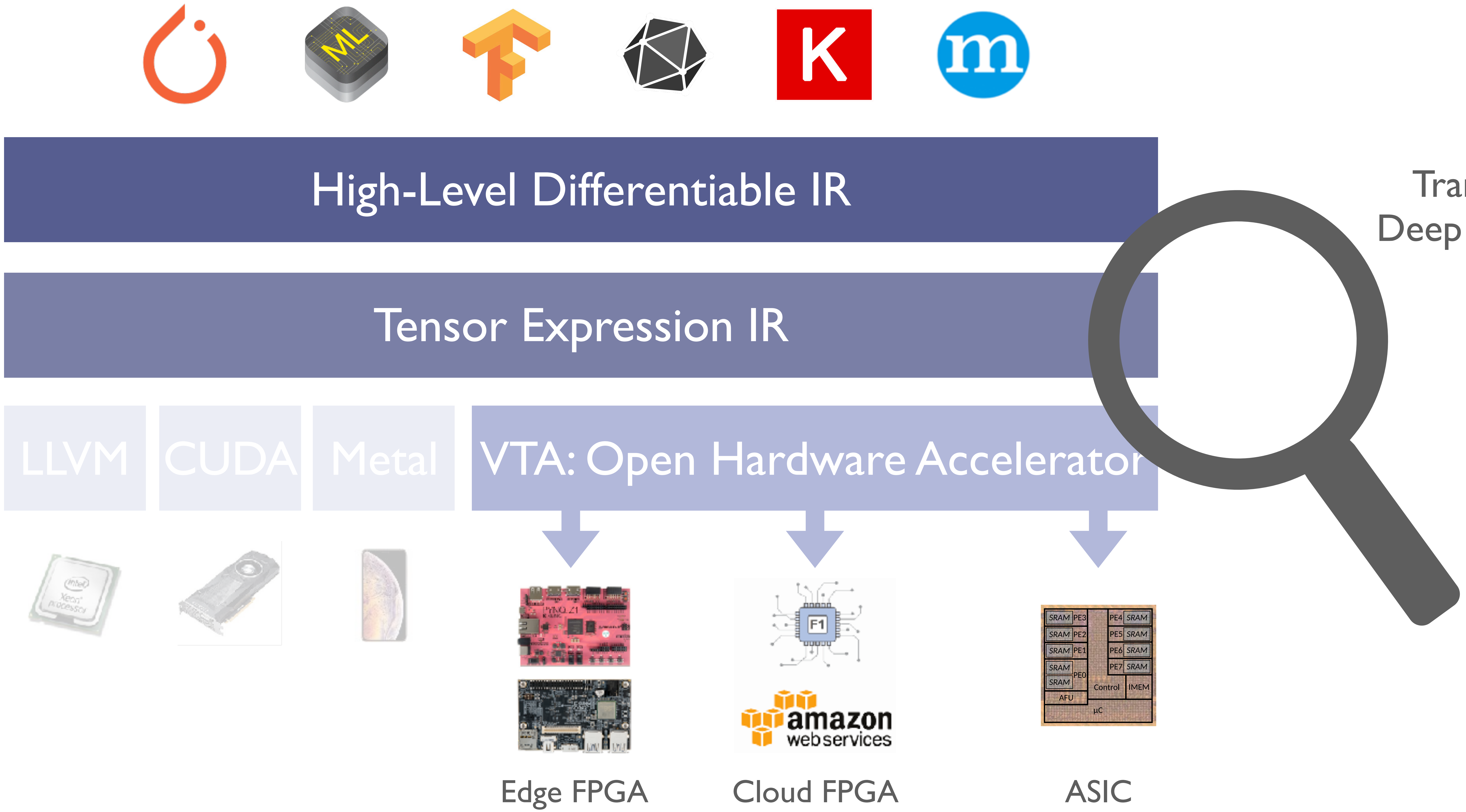


Edge FPGA | Cloud FPGA | ASIC

TVM Stack


High-Level Differentiable IR
Tensor Expression IR
LLVM
CUDA
Metal
VTA: Open Hardware Accelerator
amazon web services
Edge FPGA
Cloud FPGA
ASIC

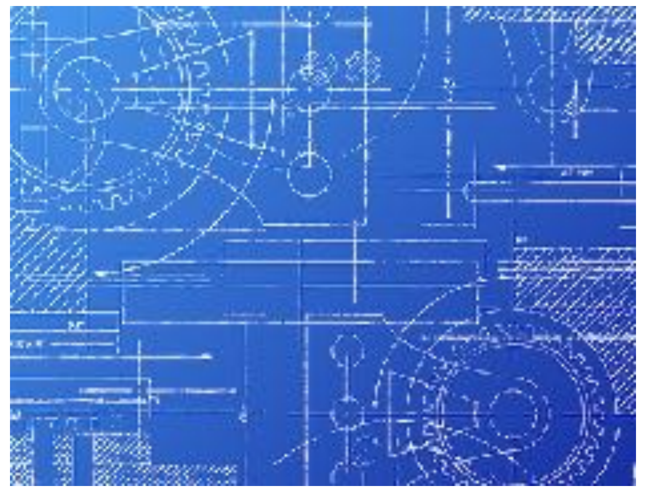
TVM Stack



Transparent End-to-End
Deep Learning System Stack

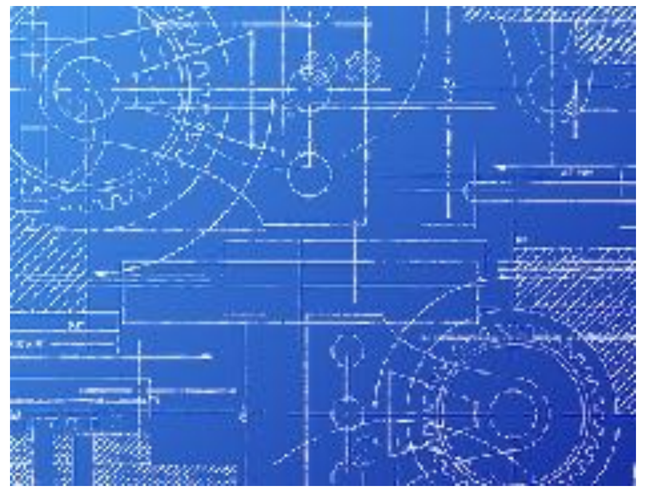
TVM+VTA Stack Goals

TVM+VTA Stack Goals



- Blue-print for a complete deep learning acceleration stack

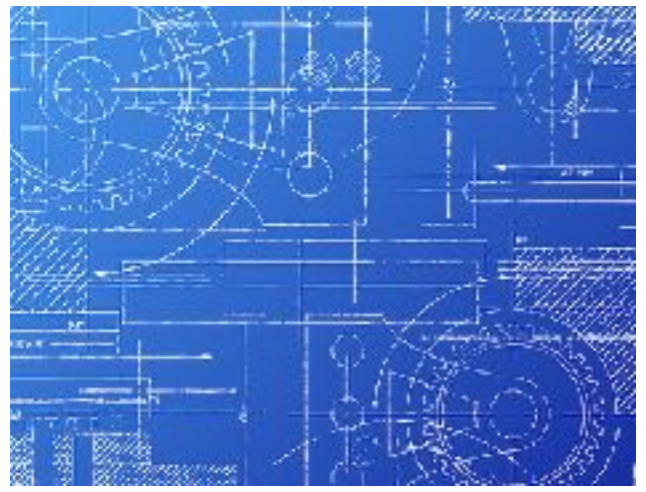
TVM+VTA Stack Goals



- Blue-print for a complete deep learning acceleration stack

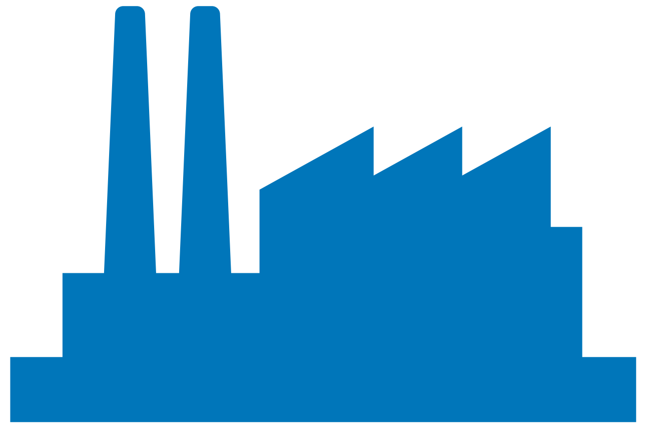
- Experimentation framework for cross-stack deep learning optimizations

TVM+VTA Stack Goals



- Blue-print for a complete deep learning acceleration stack

- Experimentation framework for cross-stack deep learning optimizations



- Open-source community for industrial-strength deep learning acceleration

VTA Overview

Extensible Hardware Architecture

Programmability Across the Stack

Facilitates HW-SW Co-Design

VTA Overview

Extensible Hardware Architecture

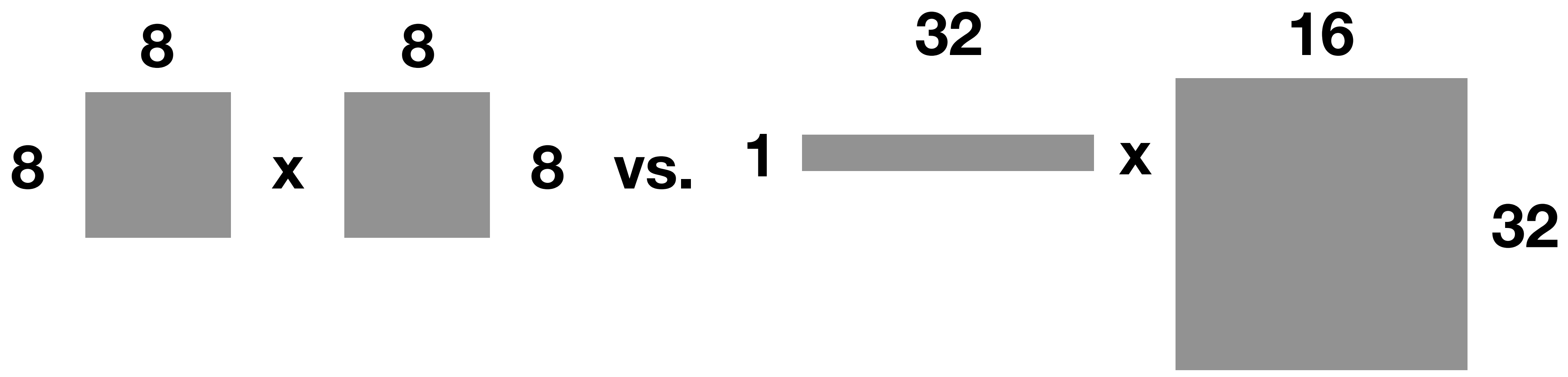
Programmability Across the Stack

Facilitates HW-SW Co-Design

VTA: General DL Architecture

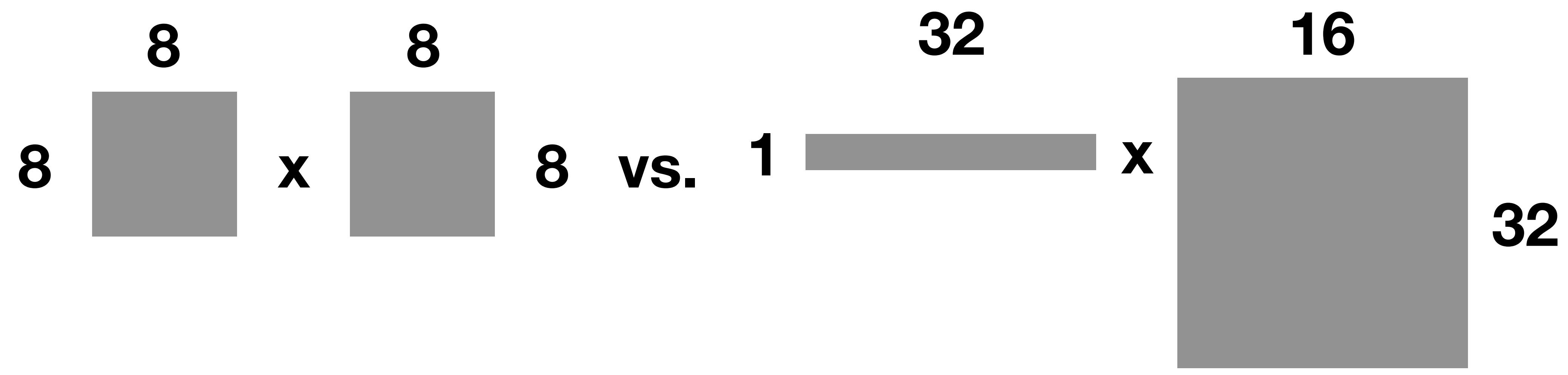
VTA: General DL Architecture

Tensor Intrinsic



VTA: General DL Architecture

Tensor Intrinsic

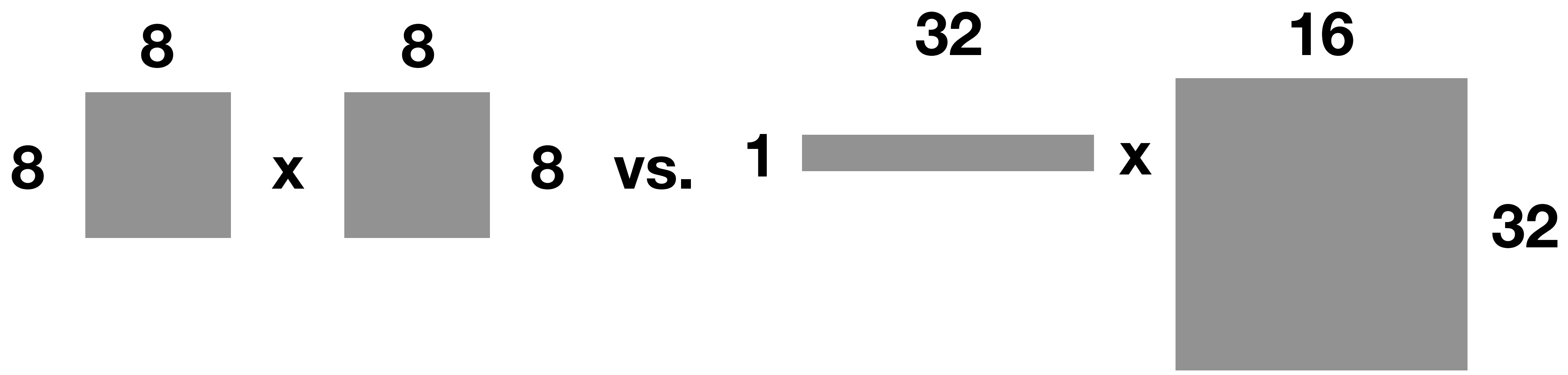


Hardware Datatype

`<16 x i8>` vs. `<32 x i4>`

VTA: General DL Architecture

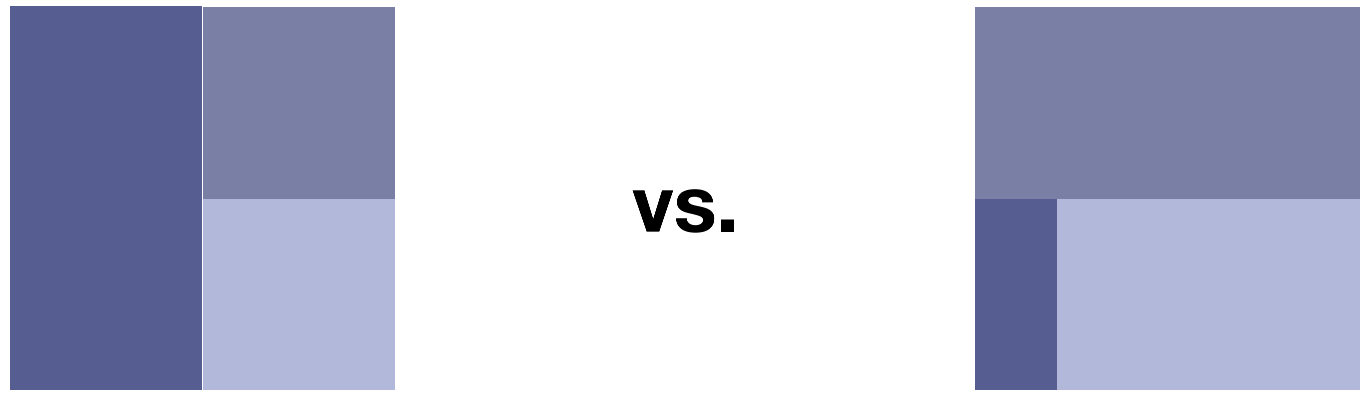
Tensor Intrinsic



Hardware Datatype

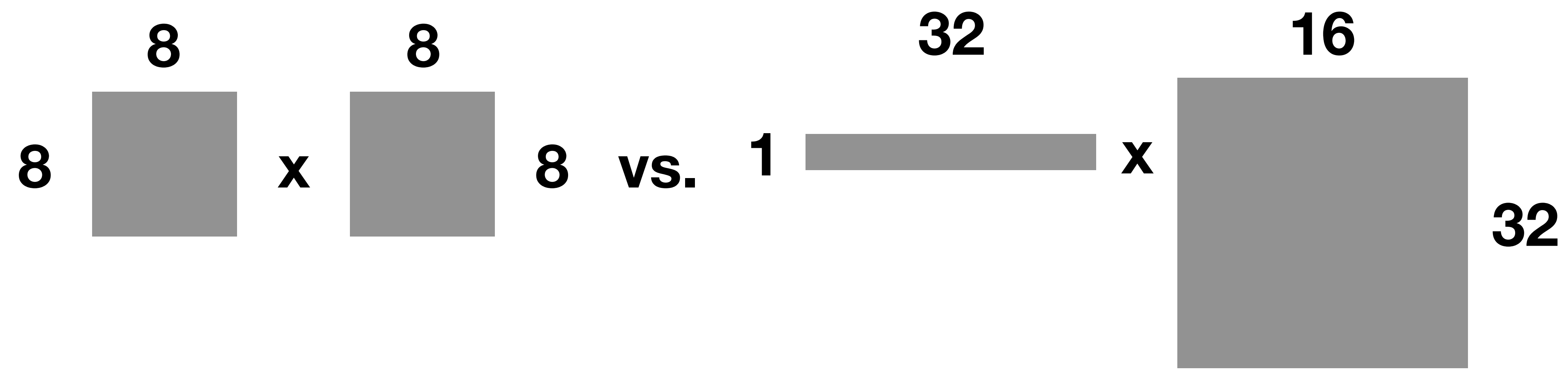
<16 x i8> vs. <32 x i4>

Memory Subsystem



VTA: General DL Architecture

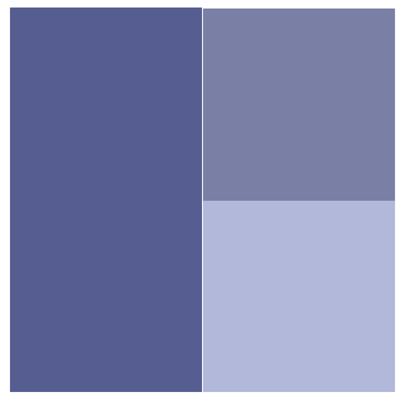
Tensor Intrinsic



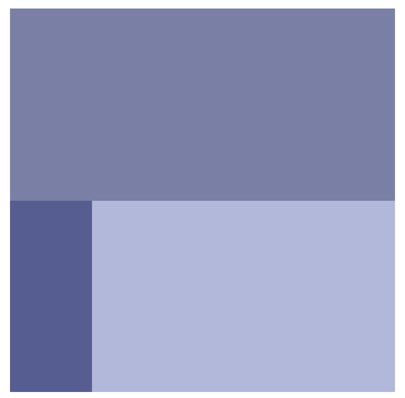
Hardware Datatype

`<16 x i8>` vs. `<32 x i4>`

Memory Subsystem



vs.



Operation Support

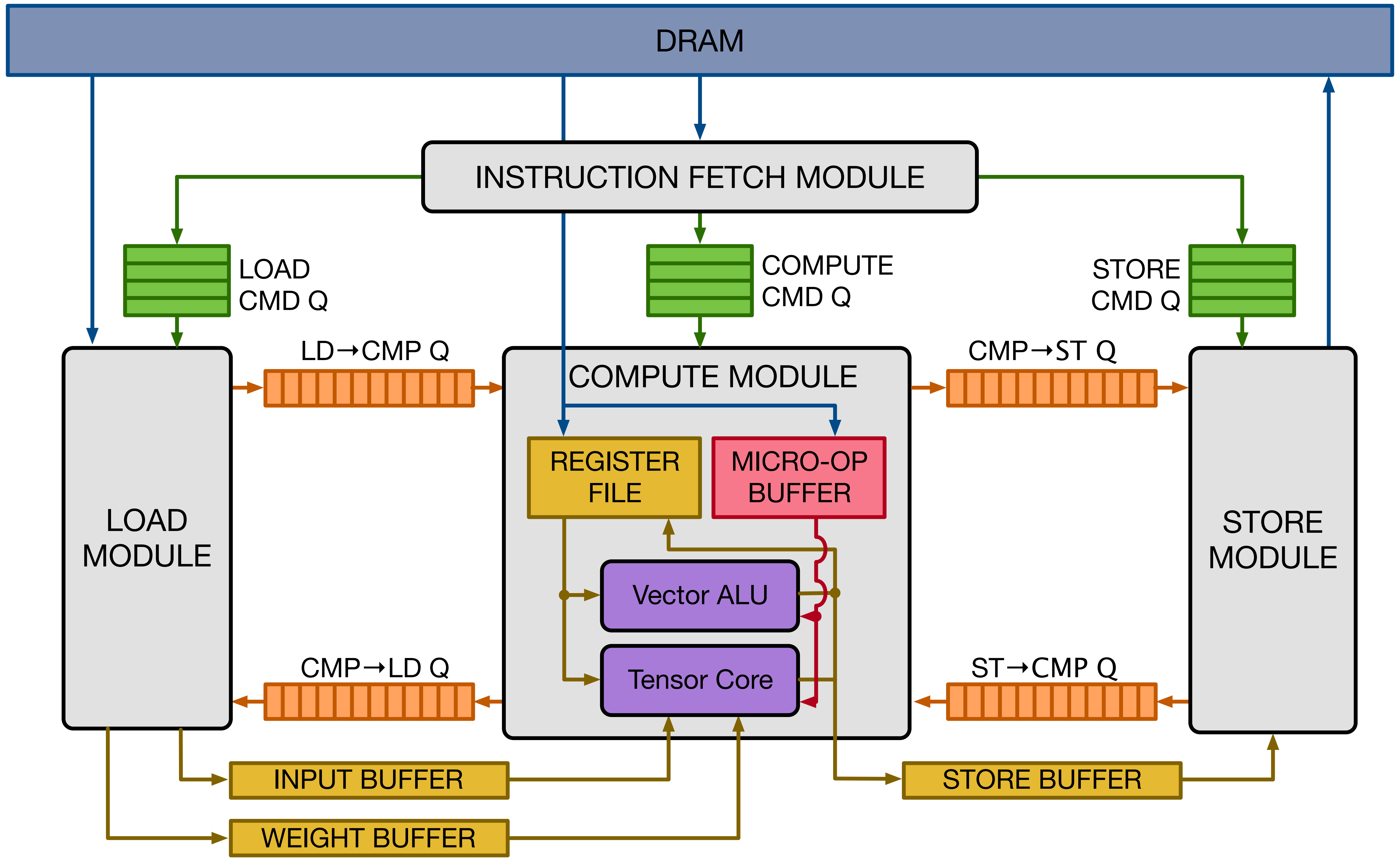
`{ADD, MUL, SHL, MAX}` vs. `{ADD, SHL, MAX}`

VTA Hardware Architecture

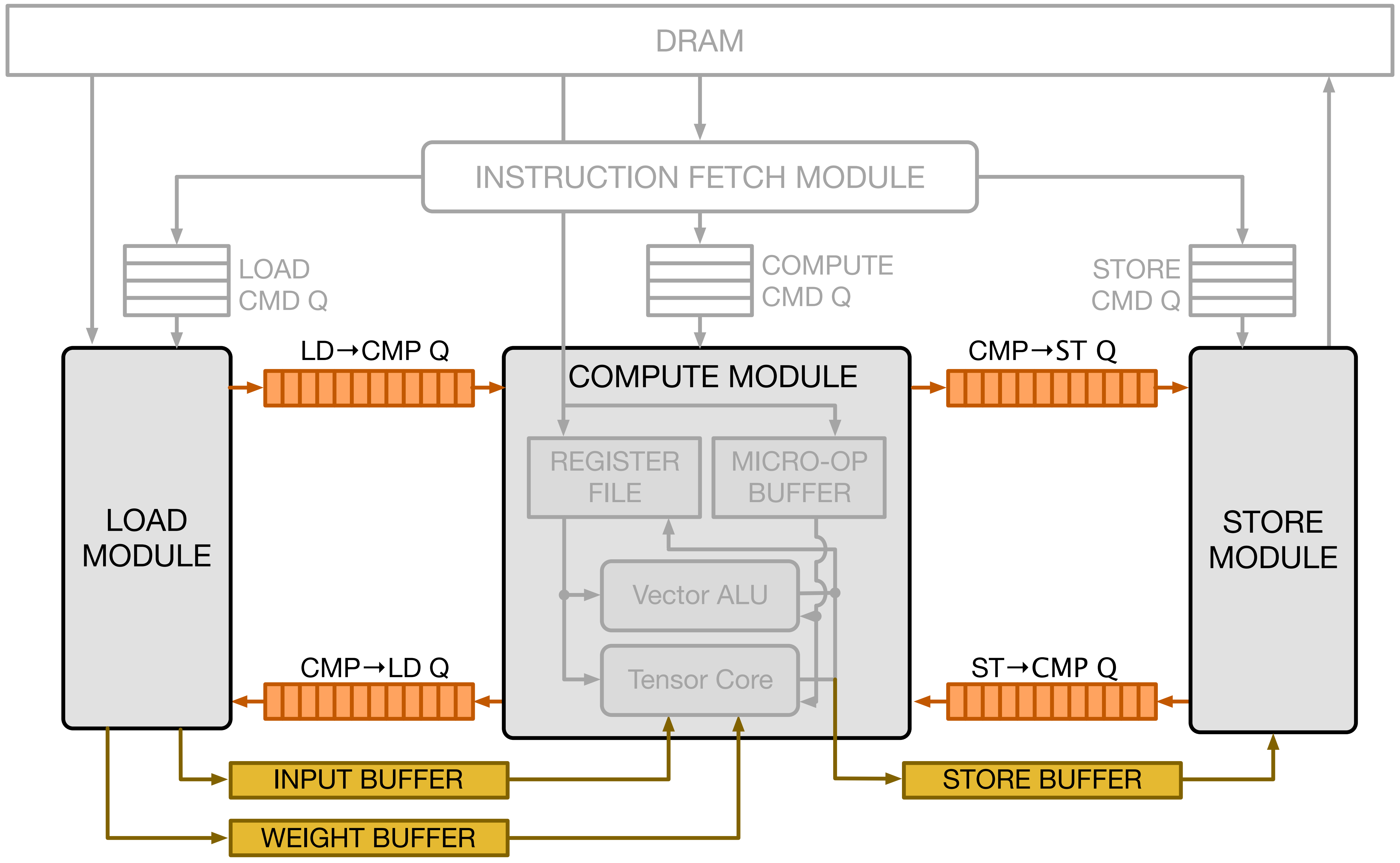
Philosophy: simple hardware, provide software-defined flexibility

VTA Hardware Architecture

Philosophy: simple hardware, provide software-defined flexibility



VTA Hardware Architecture

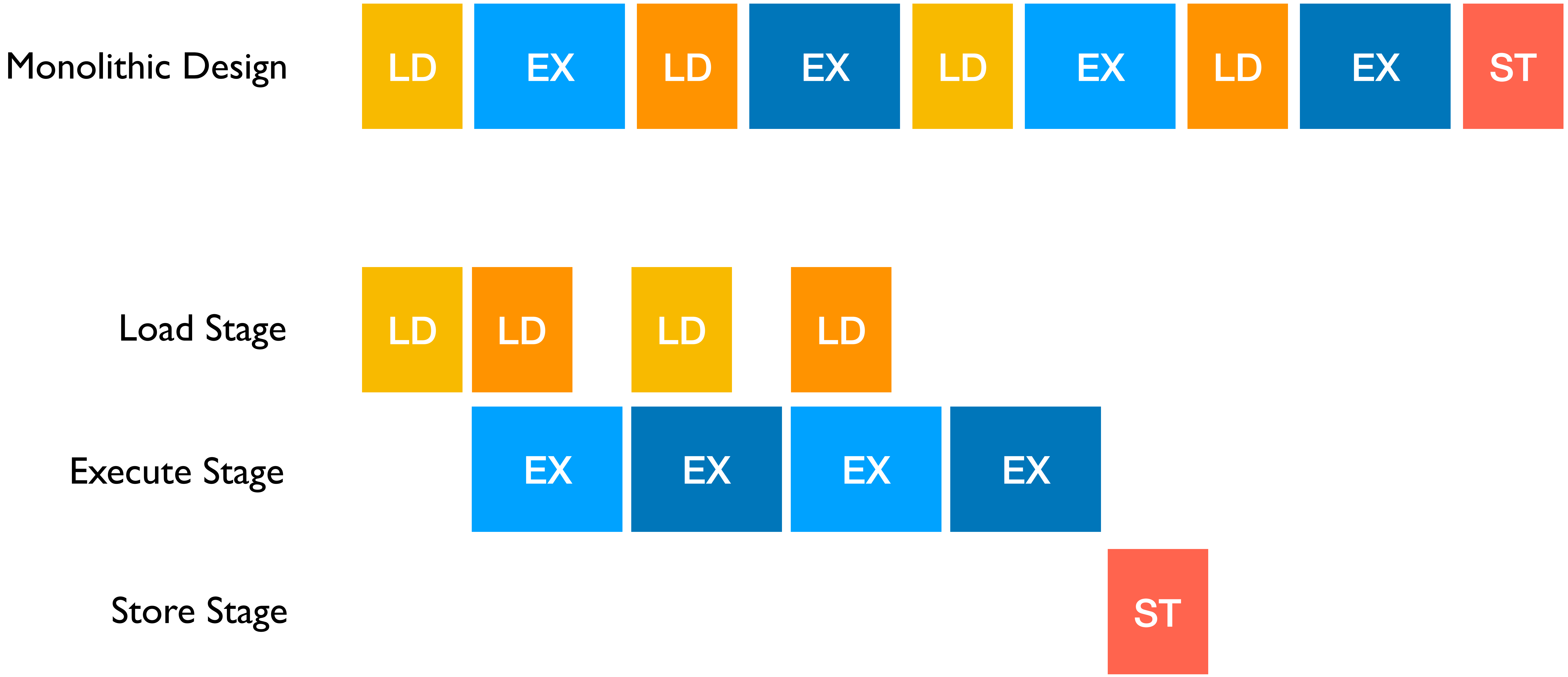


Pipelining Tasks to Hide Memory Latency

Monolithic Design

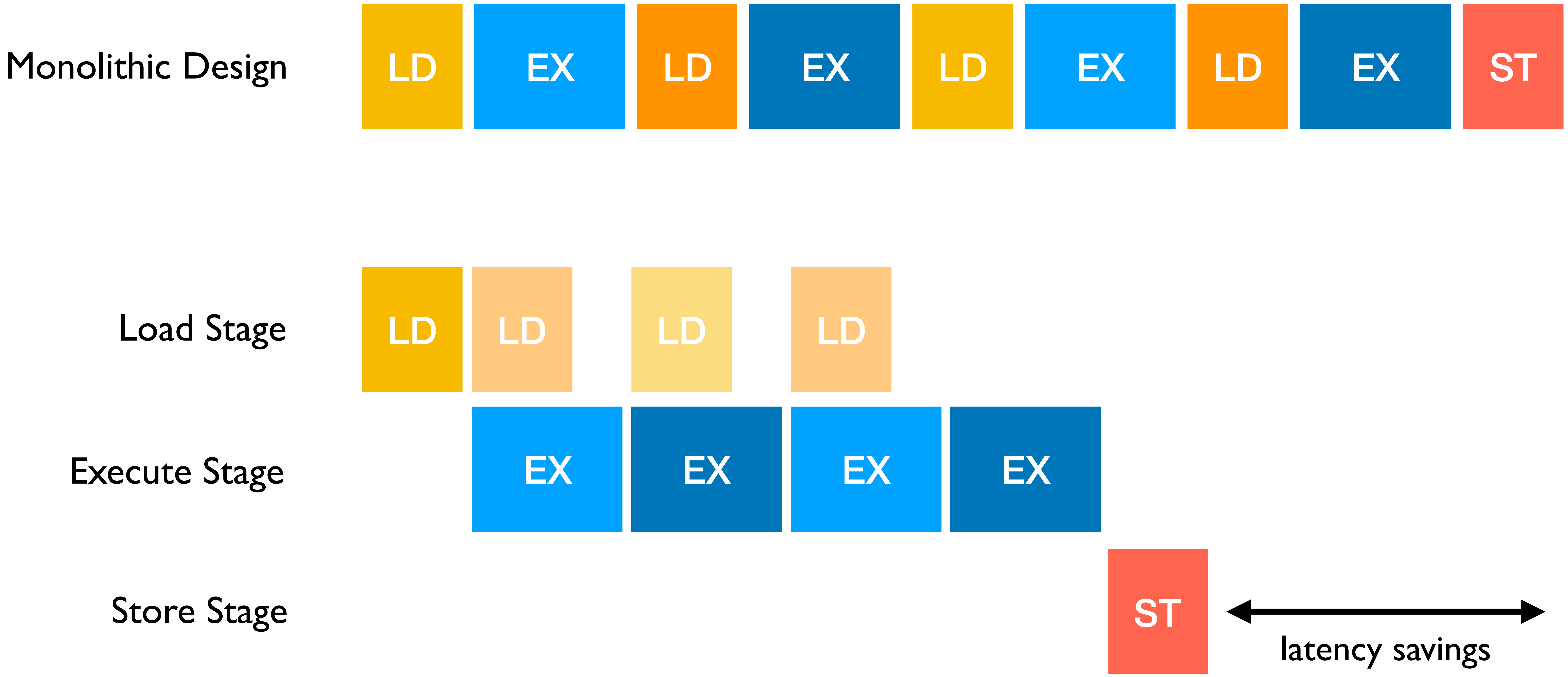
LD: load
EX: compute
ST: store

Pipelining Tasks to Hide Memory Latency



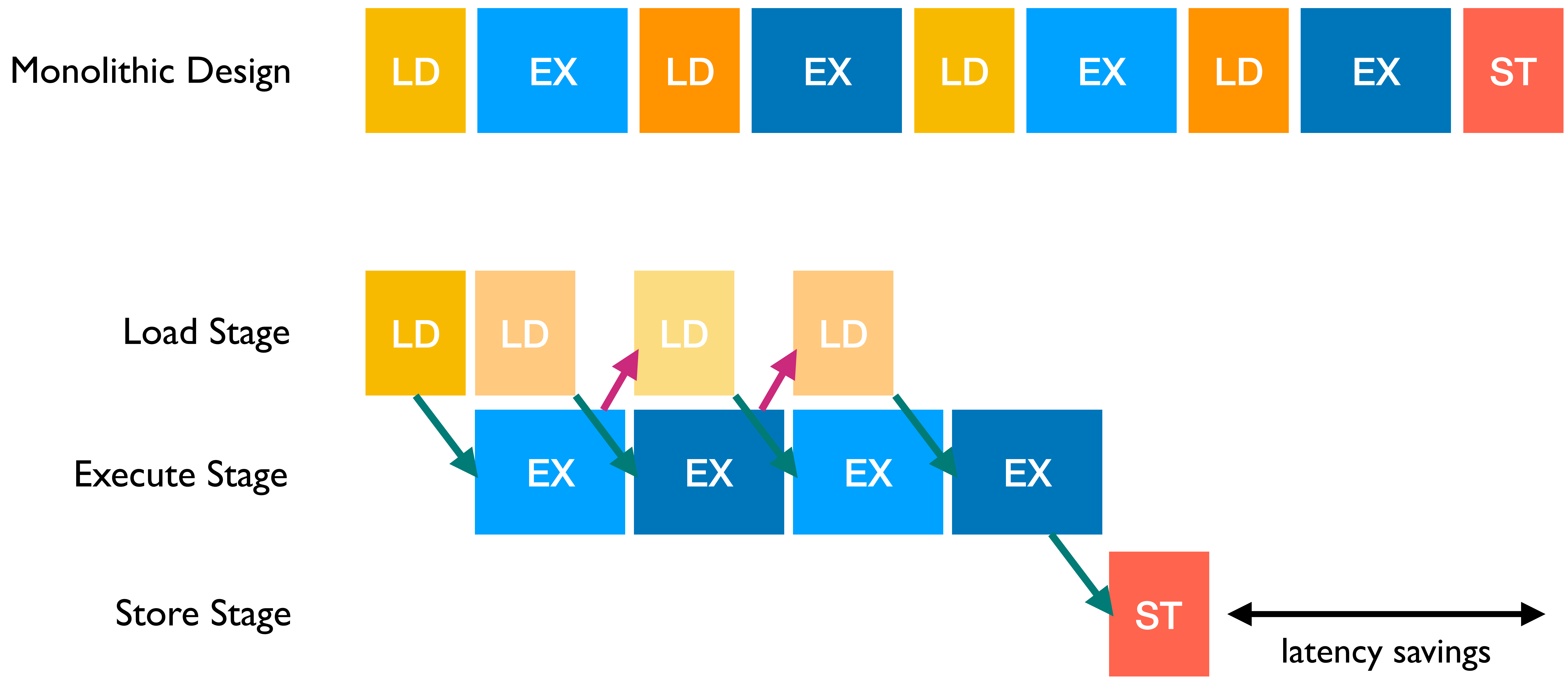
LD: load
EX: compute
ST: store

Pipelining Tasks to Hide Memory Latency



LD: load
EX: compute
ST: store

Pipelining Tasks to Hide Memory Latency



low-level synchronization between tasks is explicitly managed by the software

LD: load
EX: compute
ST: store

Two-Level ISA Overview

Provides the right tradeoff between expressiveness and code compactness

Two-Level ISA Overview

Provides the right tradeoff between expressiveness and code compactness

- Use CISC instructions to perform multi-cycle tasks

DENSE	ALU	LOAD	STORE

Two-Level ISA Overview

Provides the right tradeoff between expressiveness and code compactness

- Use CISC instructions to perform multi-cycle tasks

DENSE	ALU	LOAD	STORE

- Use RISC micro-ops to perform single-cycle tensor operations

Two-Level ISA Overview

Provides the right tradeoff between expressiveness and code compactness

- Use CISC instructions to perform multi-cycle tasks

DENSE	ALU	LOAD	STORE

- Use RISC micro-ops to perform single-cycle tensor operations

```
R0: R0 + GEMM(A8, W3)
```

Two-Level ISA Overview

Provides the right tradeoff between expressiveness and code compactness

- Use CISC instructions to perform multi-cycle tasks

DENSE	ALU	LOAD	STORE

- Use RISC micro-ops to perform single-cycle tensor operations

```
R0: R0 + GEMM(A8, W3)
R2: MAX(R0, ZERO)
```

VTA RISC Micro-Kernels

VTA RISC Micro-Kernels

multiple RISC instructions define a **micro-kernel**,
which can be invoked by a CISC instruction

VTA RISC Micro-Kernels

multiple RISC instructions define a **micro-kernel**,
which can be invoked by a CISC instruction

```
CONV2D: layout=NCHW, chan=128, kernel=(3,3), padding=(1,1), strides=(1,1)
```

VTA RISC Micro-Kernels

multiple RISC instructions define a **micro-kernel**,
which can be invoked by a CISC instruction

```
CONV2D: layout=NCHW, chan=128, kernel=(3,3), padding=(1,1), strides=(1,1)
```

```
CONV2D: layout=NCHW, chan=256, kernel=(1,1), padding=(0,0), strides=(2,2)
```

VTA RISC Micro-Kernels

multiple RISC instructions define a **micro-kernel**,
which can be invoked by a CISC instruction

```
CONV2D: layout=NCHW, chan=128, kernel=(3,3), padding=(1,1), strides=(1,1)
```

```
CONV2D: layout=NCHW, chan=256, kernel=(1,1), padding=(0,0), strides=(2,2)
```

```
CONV2D_TRANSPOSE: ...
```

VTA RISC Micro-Kernels

multiple RISC instructions define a **micro-kernel**,
which can be invoked by a CISC instruction

```
CONV2D: layout=NCHW, chan=128, kernel=(3,3), padding=(1,1), strides=(1,1)
```

```
CONV2D: layout=NCHW, chan=256, kernel=(1,1), padding=(0,0), strides=(2,2)
```

```
CONV2D_TRANSPOSE: ...
```

```
GROUP_CONV2D: ...
```

VTA RISC Micro-Kernels

micro-kernel programming gives us
software-defined flexibility

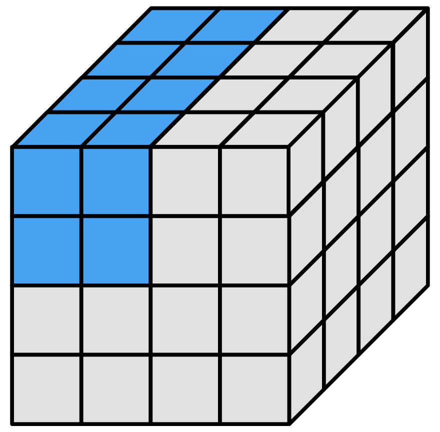
DCGAN

ResNet50

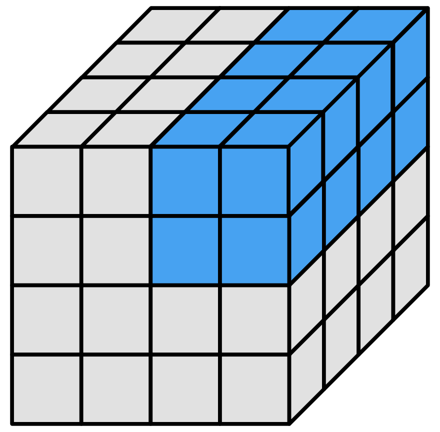
How is VTA Programmed?

How is VTA Programmed?

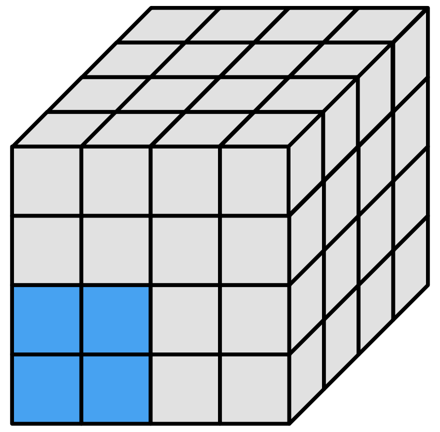
```
// Pseudo-code for convolution program for the VIA accelerator
// Virtual Thread 0
0x00: LOAD(PARAM[ 0-71])                                          // LD@TID0
0x01: LOAD(ACTIV[ 0-24])                                          // LD@TID0
0x02: LOAD(LDBUF[ 0-31])                                          // LD@TID0
0x03: PUSH(LD->EX)                                                // LD@TID0
0x04: POP (LD->EX)                                                // EX@TID0
0x05: EXE (ACTIV[ 0-24],PARAM[ 0-71],LDBUF[ 0-31],STBUF[ 0- 7])  // EX@TID0
0x06: PUSH(EX->LD)                                                // EX@TID0
0x07: PUSH(EX->ST)                                                // EX@TID0
0x08: POP (EX->ST)                                                // ST@TID0
0x09: STOR(STBUF[ 0- 7])                                          // ST@TID0
0x0A: PUSH(ST->EX)                                                // ST@TID0
```



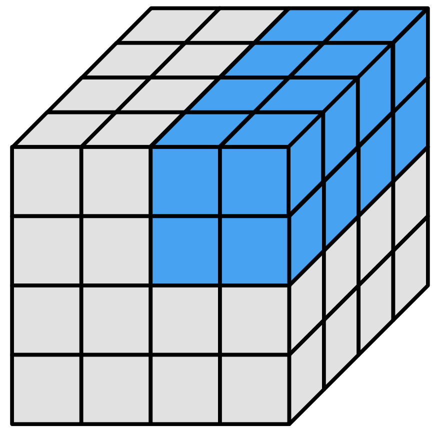
```
// Virtual Thread 1
0x0B: LOAD(ACTIV[25-50])                                          // LD@TID1
0x0C: LOAD(LDBUF[32-63])                                          // LD@TID1
0x0D: PUSH(LD->EX)                                                // LD@TID1
0x0E: POP (LD->EX)                                                // EX@TID1
0x0F: EXE (ACTIV[25-50],PARAM[ 0-71],LDBUF[32-63],STBUF[32-39])  // EX@TID1
0x10: PUSH(EX->LD)                                                // EX@TID1
0x11: PUSH(EX->ST)                                                // EX@TID1
0x12: POP (EX->ST)                                                // ST@TID1
0x13: STOR(STBUF[32-39])                                          // ST@TID1
0x14: PUSH(ST->EX)                                                // ST@TID1
```



```
// Virtual Thread 2
0x15: POP (EX->LD)                                                // LD@TID2
0x16: LOAD(PARAM[ 0-71])                                          // LD@TID2
0x17: LOAD(ACTIV[ 0-24])                                          // LD@TID2
0x18: LOAD(LDBUF[ 0-31])                                          // LD@TID2
0x19: PUSH(LD->EX)                                                // LD@TID2
0x1A: POP (LD->EX)                                                // EX@TID2
0x1B: POP (ST->EX)                                                // EX@TID2
0x1C: EXE (ACTIV[ 0-24],PARAM[ 0-71],LDBUF[ 0-31],STBUF[ 0- 7])  // EX@TID2
0x1D: PUSH(EX->ST)                                                // EX@TID2
0x1E: POP (EX->ST)                                                // ST@TID2
0x1F: STOR(STBUF[ 0- 7])                                          // ST@TID2
```



```
// Virtual Thread 3
0x20: POP (EX->LD)                                                // LD@TID3
0x21: LOAD(ACTIV[25-50])                                          // LD@TID3
0x22: LOAD(LDBUF[32-63])                                          // LD@TID3
0x23: PUSH(LD->EX)                                                // LD@TID3
0x24: POP (LD->EX)                                                // EX@TID3
0x25: POP (ST->EX)                                                // EX@TID2
0x26: EXE (ACTIV[25-50],PARAM[ 0-71],LDBUF[32-63],STBUF[32-39])  // EX@TID3
0x27: PUSH(EX->ST)                                                // EX@TID3
0x28: POP (EX->ST)                                                // ST@TID3
0x29: STOR(STBUF[32-39])                                          // ST@TID3
```



(a) Blocked convolution program with multiple thread contexts

```
// Convolution access pattern dictated by micro-coded program.
// Each register index is derived as a 2-D affine function.
// e.g. idx_rf = a_rf y+b_rf x+c_rf^0, where c_rf^0 is specified by
//      micro op 0 fields.
for y in [0…i)
  for x in [0…j)
    rf[idx_rf^0] += GEVM(act[idx_act^0], par[idx_par^0])
    rf[idx_rf^1] += GEVM(act[idx_act^1], par[idx_par^1])
    …
    rf[idx_rf^n] += GEVM(act[idx_act^n], par[idx_par^n])
```

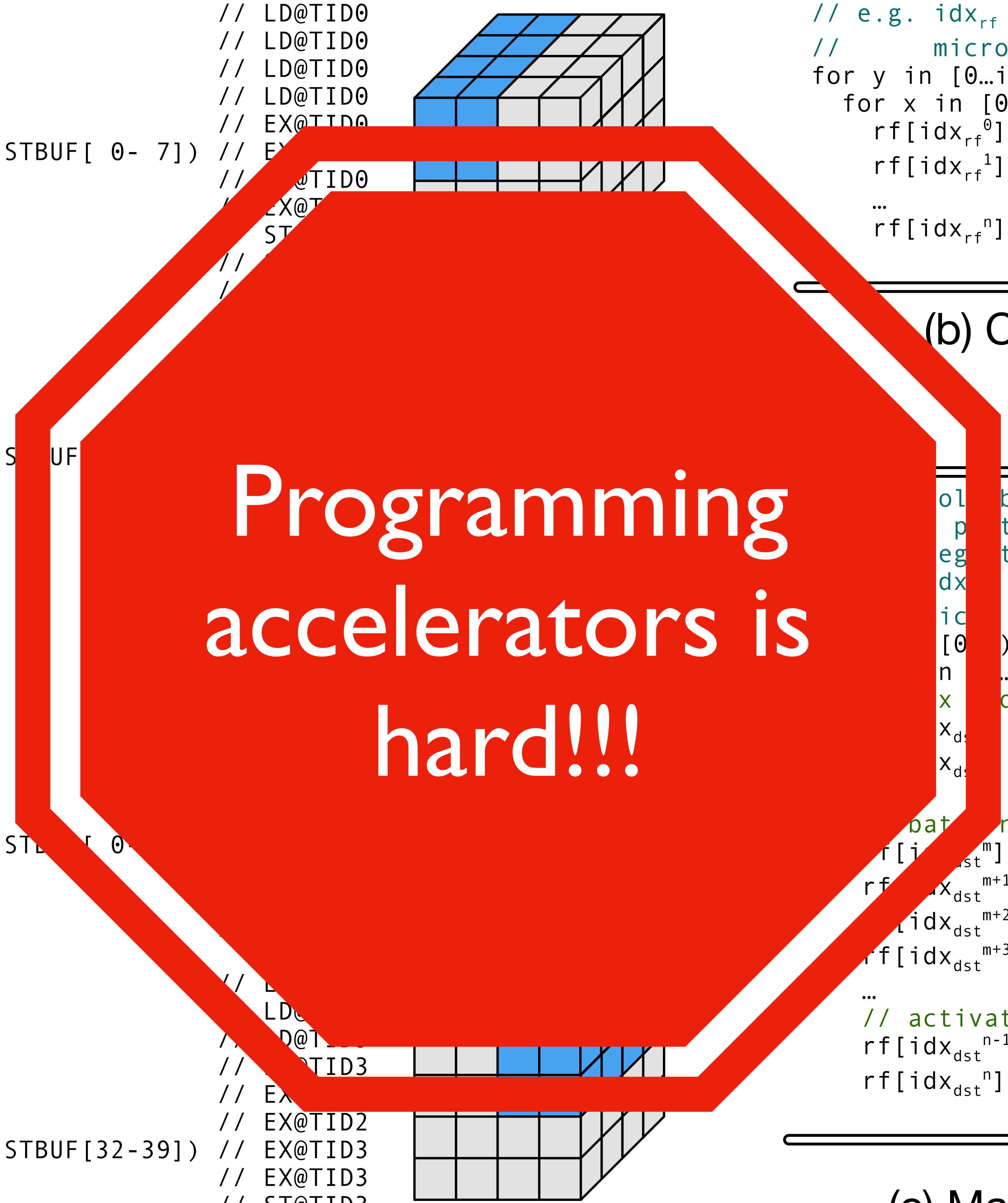
(b) Convolution micro-coded program

```
// Max-pool, batch normalization and activation function
// access pattern dictated by micro-coded program.
// Each register index is derived as a 2D affine function.
// e.g. idx_dst = a_dst y+b_dst x+c_dst^0, where c_dst^0 is specified by
//      micro op 0 fields.
for y in [0…i)
  for x in [0…j)
    // max pooling
    rf[idx_dst^0] = MAX(rf[idx_dst^0], rf[idx_src^0])
    rf[idx_dst^1] = MAX(rf[idx_dst^1], rf[idx_src^1])
    …
    // batch norm
    rf[idx_dst^m] = MUL(rf[idx_dst^m], rf[idx_src^m])
    rf[idx_dst^m+1] = ADD(rf[idx_dst^m+1], rf[idx_src^m+1])
    rf[idx_dst^m+2] = MUL(rf[idx_dst^m+2], rf[idx_src^m+2])
    rf[idx_dst^m+3] = ADD(rf[idx_dst^m+3], rf[idx_src^m+3])
    …
    // activation
    rf[idx_dst^n-1] = RELU(rf[idx_dst^n-1], rf[idx_src^n-1])
    rf[idx_dst^n] = RELU(rf[idx_dst^n], rf[idx_src^n])
```

(c) Max pool, batch norm and activation micro-coded program

How is VTA Programmed?

```
// Pseudo-code for convolution program for the VIA accelerator
// Virtual Thread 0
0x00: LOAD(PARAM[ 0-71])                                          // LD@TID0
0x01: LOAD(ACTIV[ 0-24])                                          // LD@TID0
0x02: LOAD(LDBUF[ 0-31])                                          // LD@TID0
0x03: PUSH(LD->EX)                                                // LD@TID0
0x04: POP (LD->EX)                                                // EX@TID0
0x05: EXE (ACTIV[ 0-24],PARAM[ 0-71],LDBUF[ 0-31],STBUF[ 0- 7])  // E
0x06: PUSH(EX->LD)                                                   @TID0
0x07: PUSH(EX->ST)
0x08: POP (EX->ST)
0x09: STOR(STBUF[ 0- 7])
0x0A: PUSH(ST->EX)
// Virtual Thread 1
0x0B: LOAD(ACTIV[25-50])
0x0C: LOAD(LDBUF[32-63])
0x0D: PUSH(LD->EX)
0x0E: POP (LD->EX)
0x0F: EXE (ACTIV[25-50],PARAM[ 0-71],LDBUF[32-63],S
0x10: PUSH(EX->LD)
0x11: PUSH(EX->ST)
0x12: POP (EX->ST)
0x13: STOR(STBUF[32-39])
0x14: PUSH(ST->EX)
// Virtual Thread 2
0x15: POP (EX->LD)
0x16: LOAD(PARAM[ 0-71])
0x17: LOAD(ACTIV[ 0-24])
0x18: LOAD(LDBUF[ 0-31])
0x19: PUSH(LD->EX)
0x1A: POP (LD->EX)
0x1B: POP (ST->EX)
0x1C: EXE (ACTIV[ 0-24],PARAM[ 0-71],LDBUF[ 0-31],ST
0x1D: PUSH(EX->ST)
0x1E: POP (EX->ST)
0x1F: STOR(STBUF[ 0- 7])
// Virtual Thread 3
0x20: POP (EX->LD)
0x21: LOAD(ACTIV[25-50])
0x22: LOAD(LDBUF[32-63])
0x23: PUSH(LD->EX)                                                   TID3
0x24: POP (LD->EX)                                                // EX
0x25: POP (ST->EX)                                                // EX@TID2
0x26: EXE (ACTIV[25-50],PARAM[ 0-71],LDBUF[32-63],STBUF[32-39])  // EX@TID3
0x27: PUSH(EX->ST)                                                // EX@TID3
0x28: POP (EX->ST)                                                // ST@TID3
0x29: STOR(STBUF[32-39])                                          // ST@TID3
```



(a) Blocked convolution program with multiple thread contexts

```
// Convolution access pattern dictated by micro-coded program.
// Each register index is derived as a 2-D affine function.
// e.g. idx_rf = a_rf y+b_rf x+c_rf^0, where c_rf^0 is specified by
//      micro op 0 fields.
for y in [0…i)
  for x in [0…j)
    rf[idx_rf^0] += GEVM(act[idx_act^0], par[idx_par^0])
    rf[idx_rf^1] += GEVM(act[idx_act^1], par[idx_par^1])
    …
    rf[idx_rf^n] += GEVM(act[idx_act^n], par[idx_par^n])
```

(b) Convolution micro-coded program

```
ol batch normalization and activation function
 p tern dictated by micro-coded program.
eg ter index is derived as a 2D affine function.
dx  = a_dst y+b_dst x+c_dst^0, where c_dst^0 is specified by
ic  op 0 fields.
[0 )
n  …j)
x  oling
x_d   = MAX(rf[idx_dst^0], rf[idx_src^0])
x_d   = MAX(rf[idx_dst^1], rf[idx_src^1])

bat norm
f[i  st^m] = MUL(rf[idx_dst^m], rf[idx_src^m])
rf  x_dst^m+1] = ADD(rf[idx_dst^m+1], rf[idx_src^m+1])
 [idx_dst^m+2] = MUL(rf[idx_dst^m+2], rf[idx_src^m+2])
f[idx_dst^m+3] = ADD(rf[idx_dst^m+3], rf[idx_src^m+3])
…
// activation
rf[idx_dst^n-1] = RELU(rf[idx_dst^n-1], rf[idx_src^n-1])
rf[idx_dst^n] = RELU(rf[idx_dst^n], rf[idx_src^n])
```

(c) Max pool, batch norm and activation micro-coded program

VTA Overview

Extensible Hardware Architecture

Programmability Across the Stack

Facilitates HW-SW Co-Design

Latency Hiding: An Example of Cross-Stack Design

programmer friendly construct

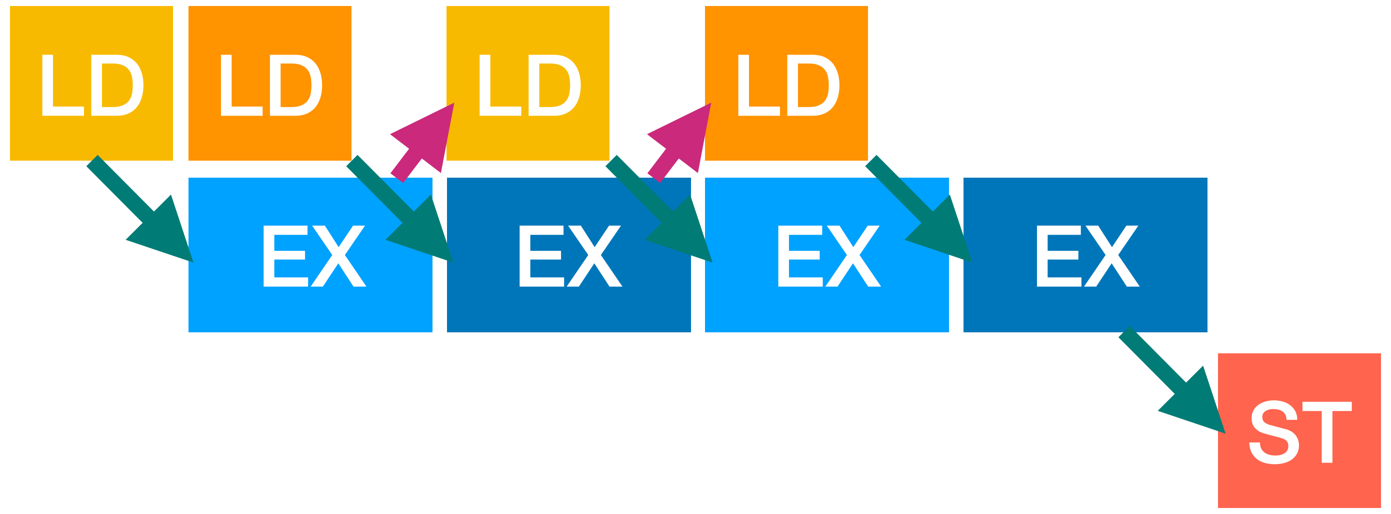
```
// Virtual Threading
tx, co = s[OUT_L].split(co, factor=2)
s[OUT_L].bind(tx, thread_axis("cthread"))
```

Latency Hiding: An Example of Cross-Stack Design

programmer friendly construct

```
// Virtual Threading
tx, co = s[OUT_L].split(co, factor=2)
s[OUT_L].bind(tx, thread_axis("cthread"))
```

low-level pipelined execution



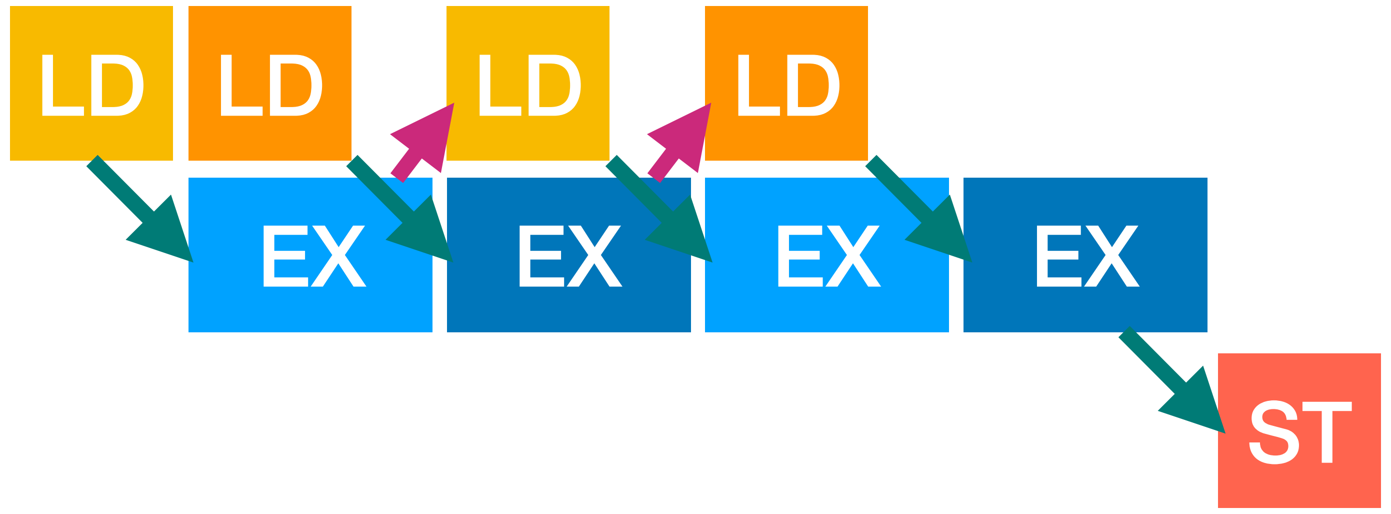
Latency Hiding: An Example of Cross-Stack Design

programmer friendly construct

?

low-level pipelined execution

```
// Virtual Threading
tx, co = s[OUT_L].split(co, factor=2)
s[OUT_L].bind(tx, thread_axis("cthread"))
```



Latency Hiding: An Example of Cross-Stack Design

programmer friendly construct

low-level pipelined execution

Latency Hiding: An Example of Cross-Stack Design

programmer friendly construct

↓

Tensor Expression Optimizer (TVM) — inserts dependence ops based on thread scope

↓

low-level pipelined execution

Latency Hiding: An Example of Cross-Stack Design

programmer friendly construct

Tensor Expression Optimizer (TVM) — inserts dependence ops based on thread scope

VTA Runtime & JIT Compiler — generates instruction stream

low-level pipelined execution

Latency Hiding: An Example of Cross-Stack Design

programmer friendly construct

↓

Tensor Expression Optimizer (TVM)	inserts dependence ops based on thread scope
VTA Runtime & JIT Compiler	generates instruction stream
VTA Hardware/Software Interface (ISA)	exposes explicit dependences

↓

low-level pipelined execution

Latency Hiding: An Example of Cross-Stack Design

programmer friendly construct

Tensor Expression Optimizer (TVM)	inserts dependence ops based on thread scope
VTA Runtime & JIT Compiler	generates instruction stream
VTA Hardware/Software Interface (ISA)	exposes explicit dependences
VTA MicroArchitecture	execution predicated on dependences

low-level pipelined execution

Latency Hiding: An Example of Cross-Stack Design

programmer friendly construct

↓

Tensor Expression Optimizer (TVM)	inserts dependence ops based on thread scope
VTA Runtime & JIT Compiler	generates instruction stream
VTA Hardware/Software Interface (ISA)	exposes explicit dependences
VTA MicroArchitecture	execution predicated on dependences

↓

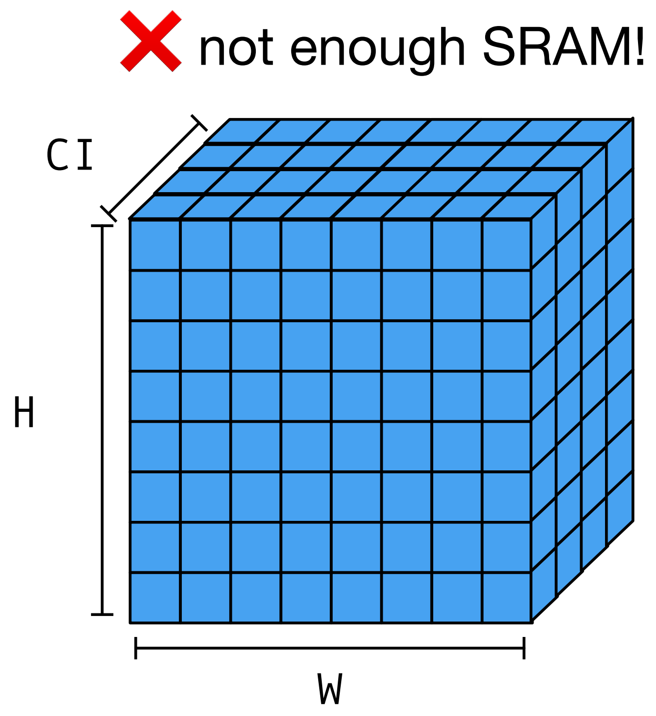
low-level pipelined execution

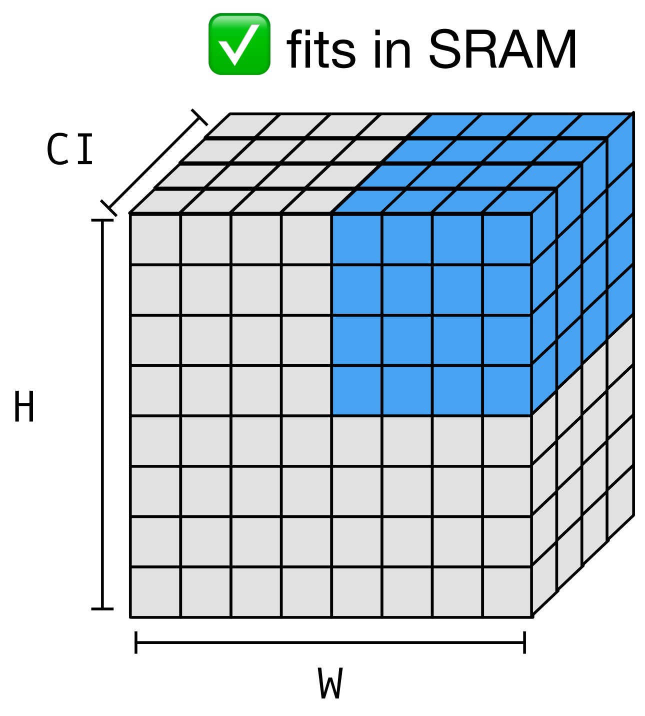
9-60% better compute utilization

VTA Helped inform ASIC Support in TVM

VTA Helped inform ASIC Support in TVM

1. How do we partition work and explicitly manage on-chip memories?

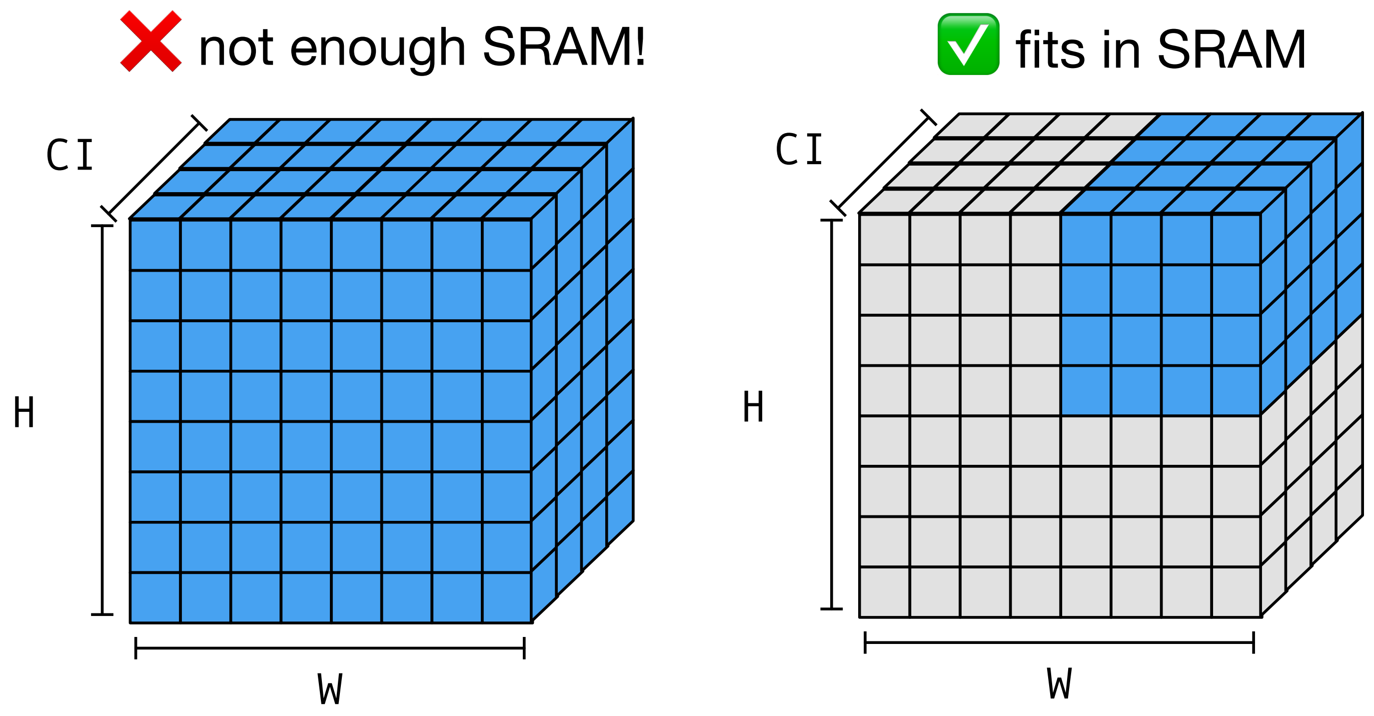





```
// Tile
yo, xo, yi, xi = s[OUT].tile(y, x, 4, 4)
// Scoped cache read
INP_L = s.cache_read(INP, vta.inp, [OUT])
s[INP_L].compute_at(s[OUT], xo)
```

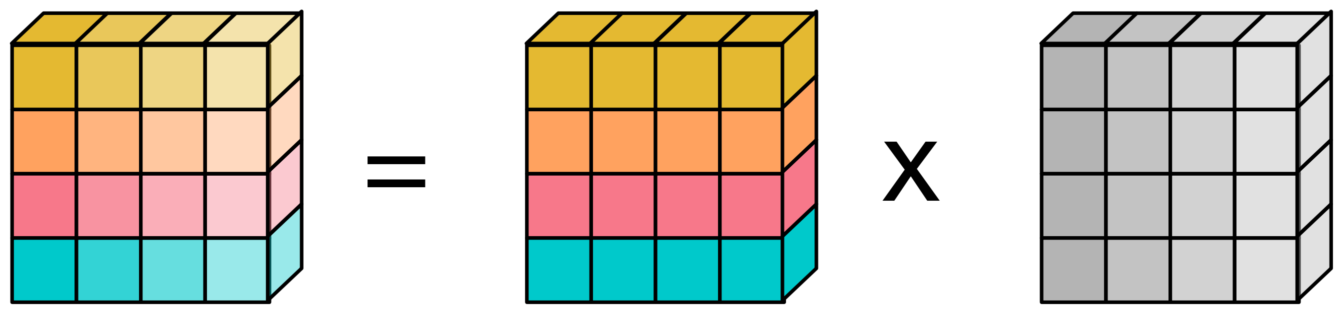
VTA Helped inform ASIC Support in TVM

1. How do we partition work and explicitly manage on-chip memories?




```
// Tile
yo, xo, yi, xi = s[OUT].tile(y, x, 4, 4)
// Scoped cache read
INP_L = s.cache_read(INP, vta.inp, [OUT])
s[INP_L].compute_at(s[OUT], xo)
```

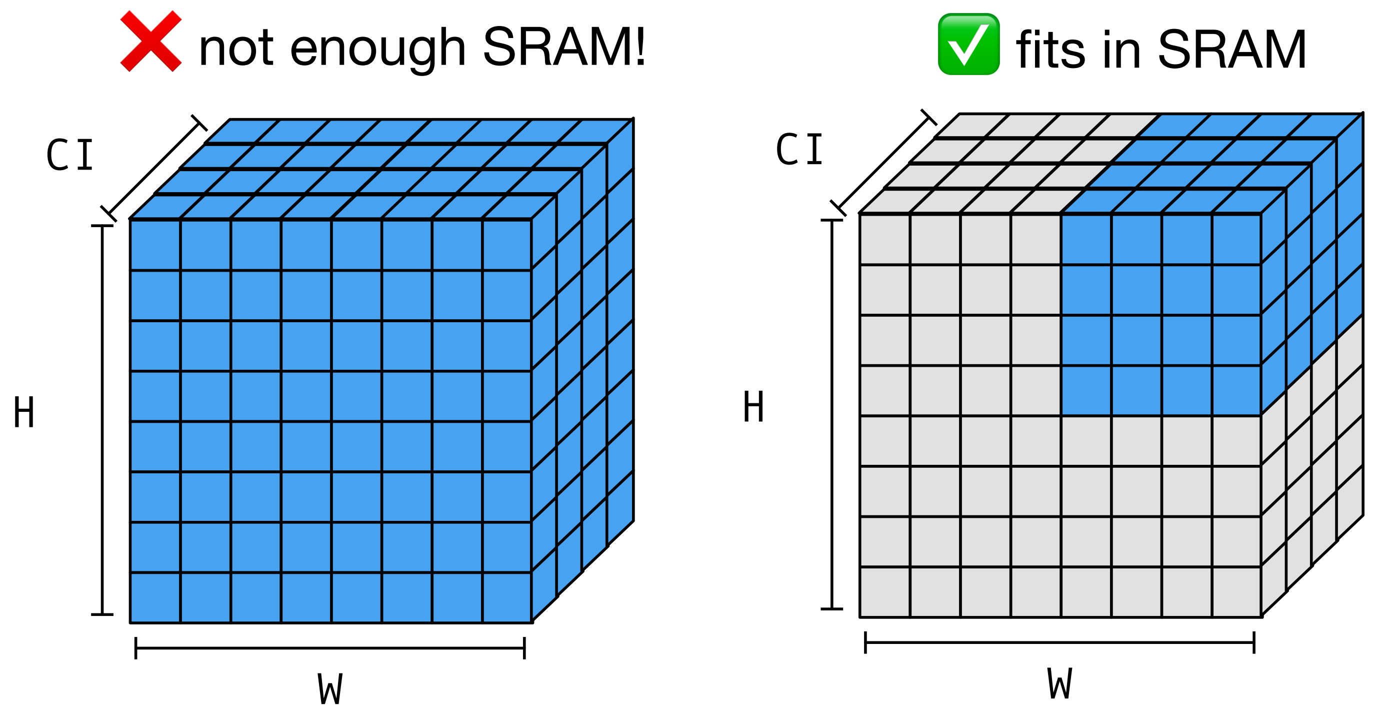
2. How do we take advantage of tensor computation intrinsics?




```
// Tensorize
s[OUT_L].tensorize(ni)
```

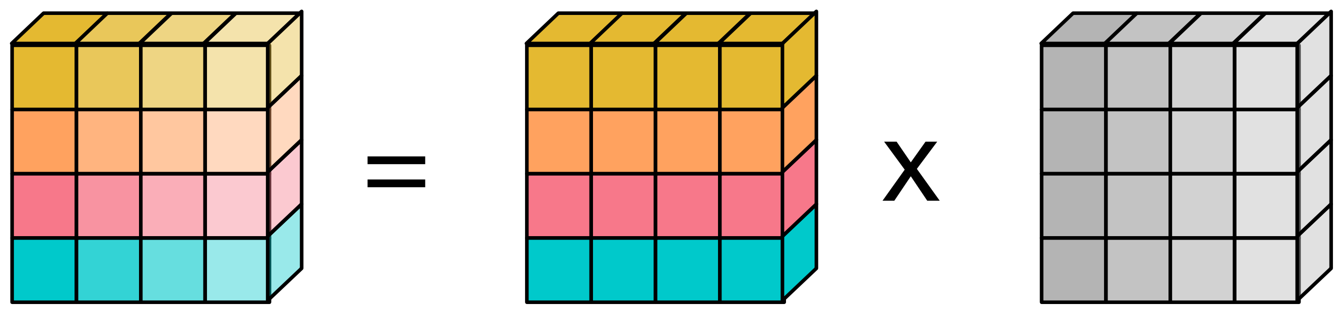
VTA Helped inform ASIC Support in TVM

1. How do we partition work and explicitly manage on-chip memories?



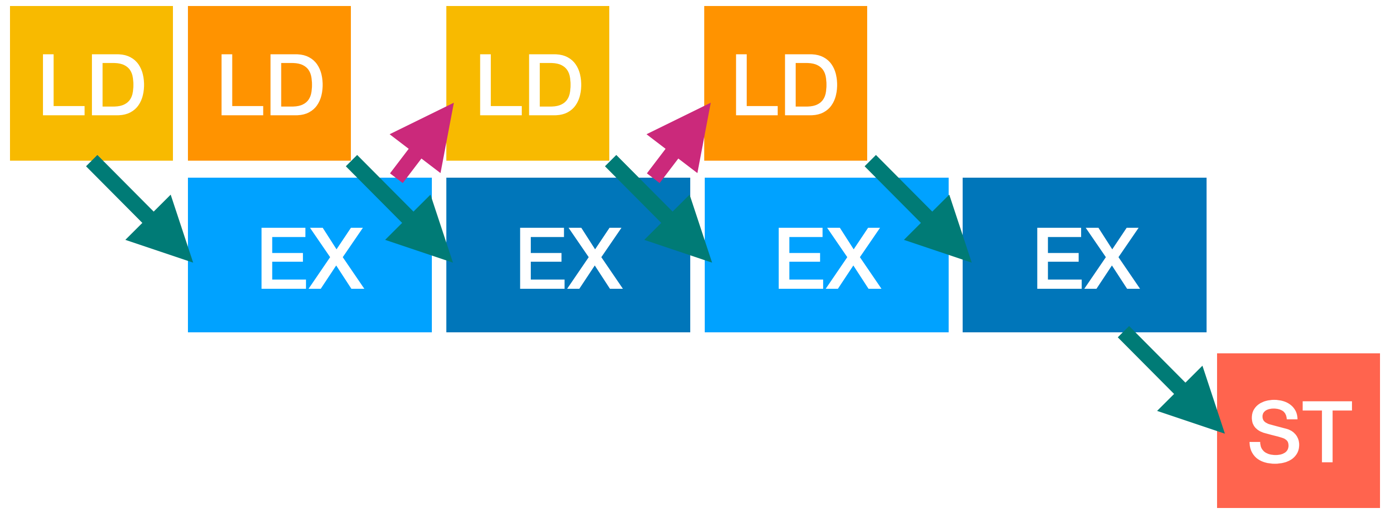

```
// Tile
yo, xo, yi, xi = s[OUT].tile(y, x, 4, 4)
// Scoped cache read
INP_L = s.cache_read(INP, vta.inp, [OUT])
s[INP_L].compute_at(s[OUT], xo)
```

2. How do we take advantage of tensor computation intrinsics?




```
// Tensorize
s[OUT_L].tensorize(ni)
```

3. How do we hide memory access latency?




```
// Virtual Threading
tx, co = s[OUT_L].split(co, factor=2)
s[OUT_L].bind(tx, thread_axis("cthread"))
```

VTA Overview

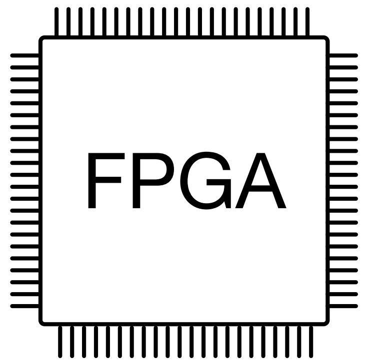
Extensible Hardware Architecture

Programmability Across the Stack

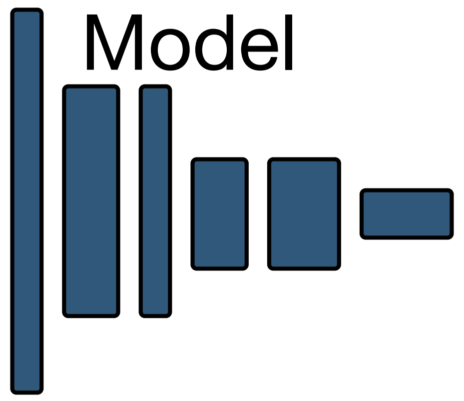
Facilitates HW-SW Co-Design

Hardware Exploration with VTA

HW / SW Constraints



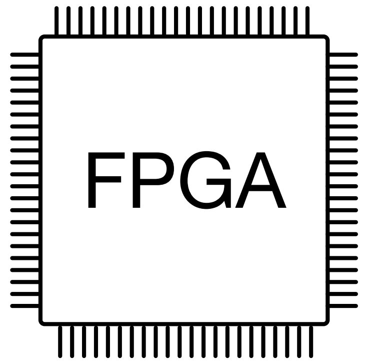

```
# BRAMs
DRAM channels
logic resources
```



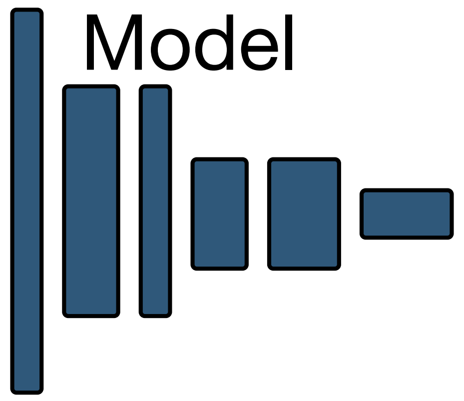

```
batch size
data types
channel width
```

Hardware Exploration with VTA

HW / SW Constraints




```
# BRAMs
DRAM channels
logic resources
```




```
batch size
data types
channel width
```

VTA Design Space

Architecture Knobs

- GEMM Intrinsic: e.g. (1,32) x (32,32) vs. (4,16) x (16,16)
- # of units in tensor ALU : e.g. 32 vs. 16
- BRAM allocation between buffers, register file, micro-op cache

Circuit Knobs

- Circuit Pipelining: e.g. for GEMM core between [11, 20] stages
- PLL Frequency Sweeps: e.g. 250 vs. 300 vs. 333MHz

Hardware Exploration with VTA

HW / SW Constraints

FPGA

- # BRAMs
- DRAM channels
- logic resources

Model

- batch size
- data types
- channel width

VTA Design Space

Architecture Knobs

- GEMM Intrinsic: e.g. (1,32) x (32,32) vs. (4,16) x (16,16)
- # of units in tensor ALU : e.g. 32 vs. 16
- BRAM allocation between buffers, register file, micro-op cache

Circuit Knobs

- Circuit Pipelining: e.g. for GEMM core between [11, 20] stages
- PLL Frequency Sweeps: e.g. 250 vs. 300 vs. 333MHz

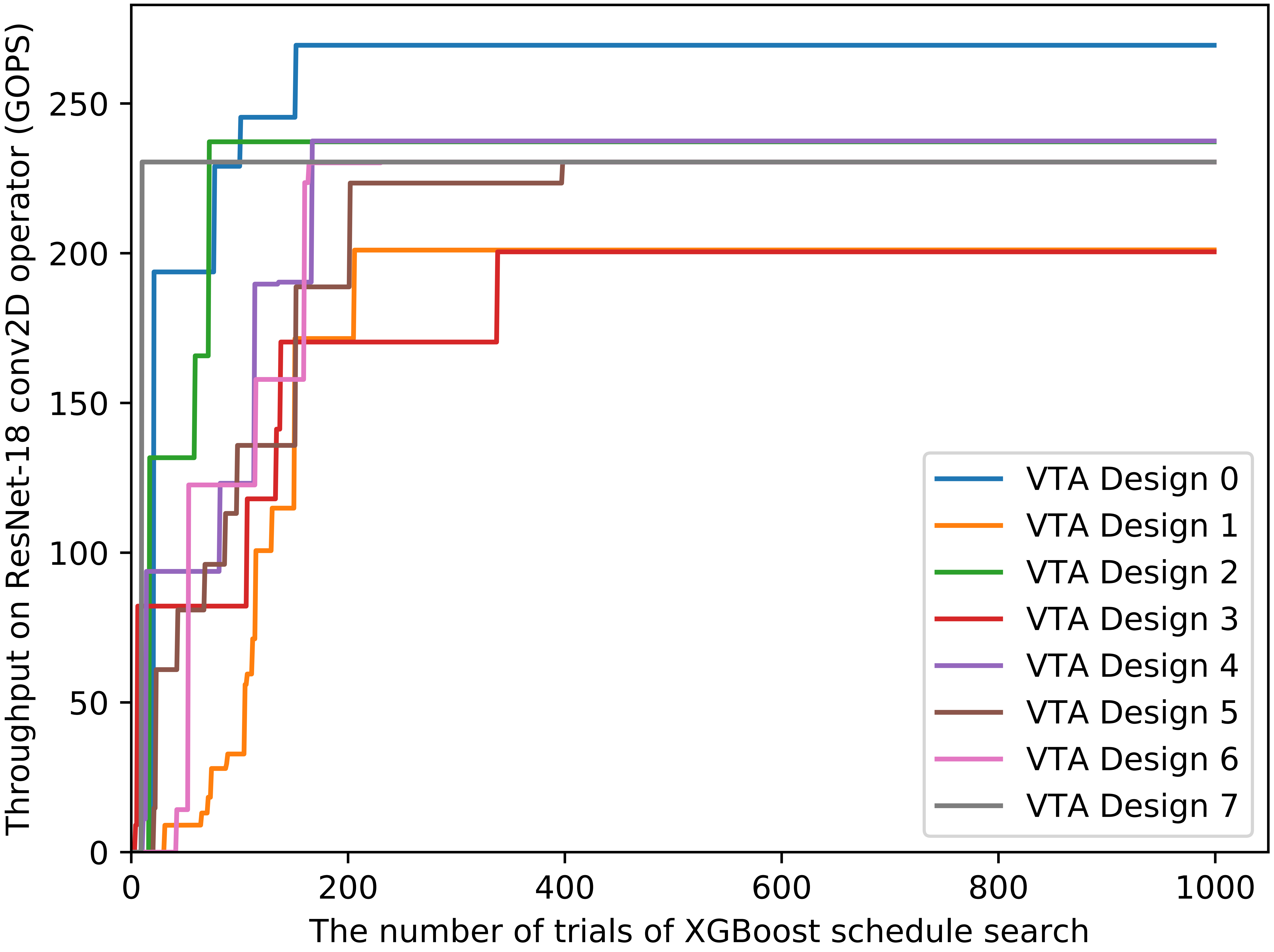
VTA Candidate Designs

- #1 Design AAA @ 307GOPs
- #2 Design BBB @ 307GOPs
- #3 Design CCC @ 307GOPs
- #4 Design DDD @ 256GOPs

Needs to pass place & route and pass timing closure

AutoTVM for Conv2D on Hardware Candidates

AutoTVM for Conv2D on Hardware Candidates



Schedule Exploration with VTA

VTA Candidate Designs

#1 Design AAA @ 307GOPs

#2 Design BBB @ 307GOPs

#3 Design CCC @ 307GOPs

#4 Design DDD @ 256GOPs

Needs to pass place & route and pass timing closure

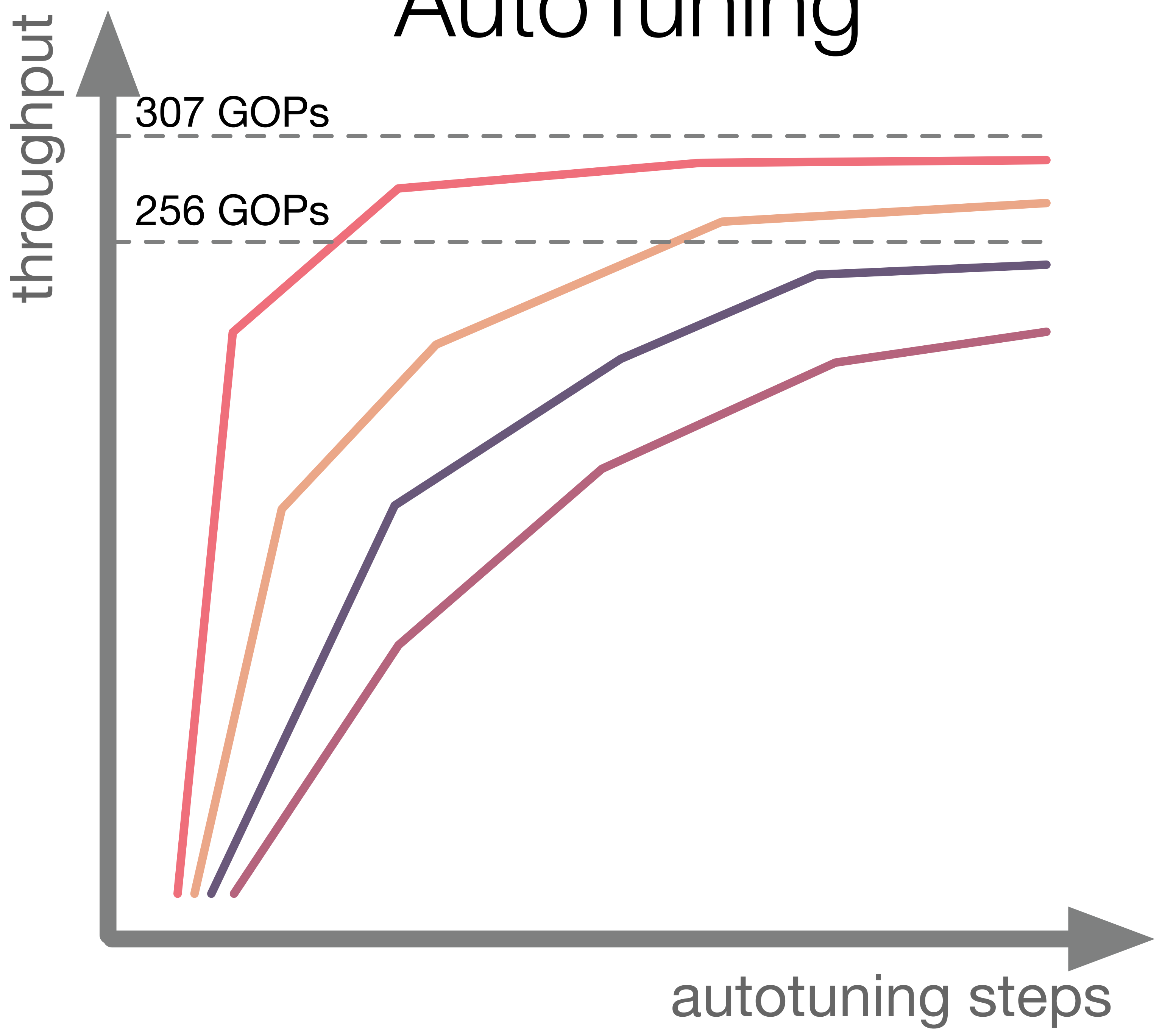
Schedule Exploration with VTA

VTA Candidate Designs

- #1 Design AAA @ 307GOPs
- #2 Design BBB @ 307GOPs
- #3 Design CCC @ 307GOPs
- #4 Design DDD @ 256GOPs

Needs to pass place & route
and pass timing closure

Operator Performance AutoTuning



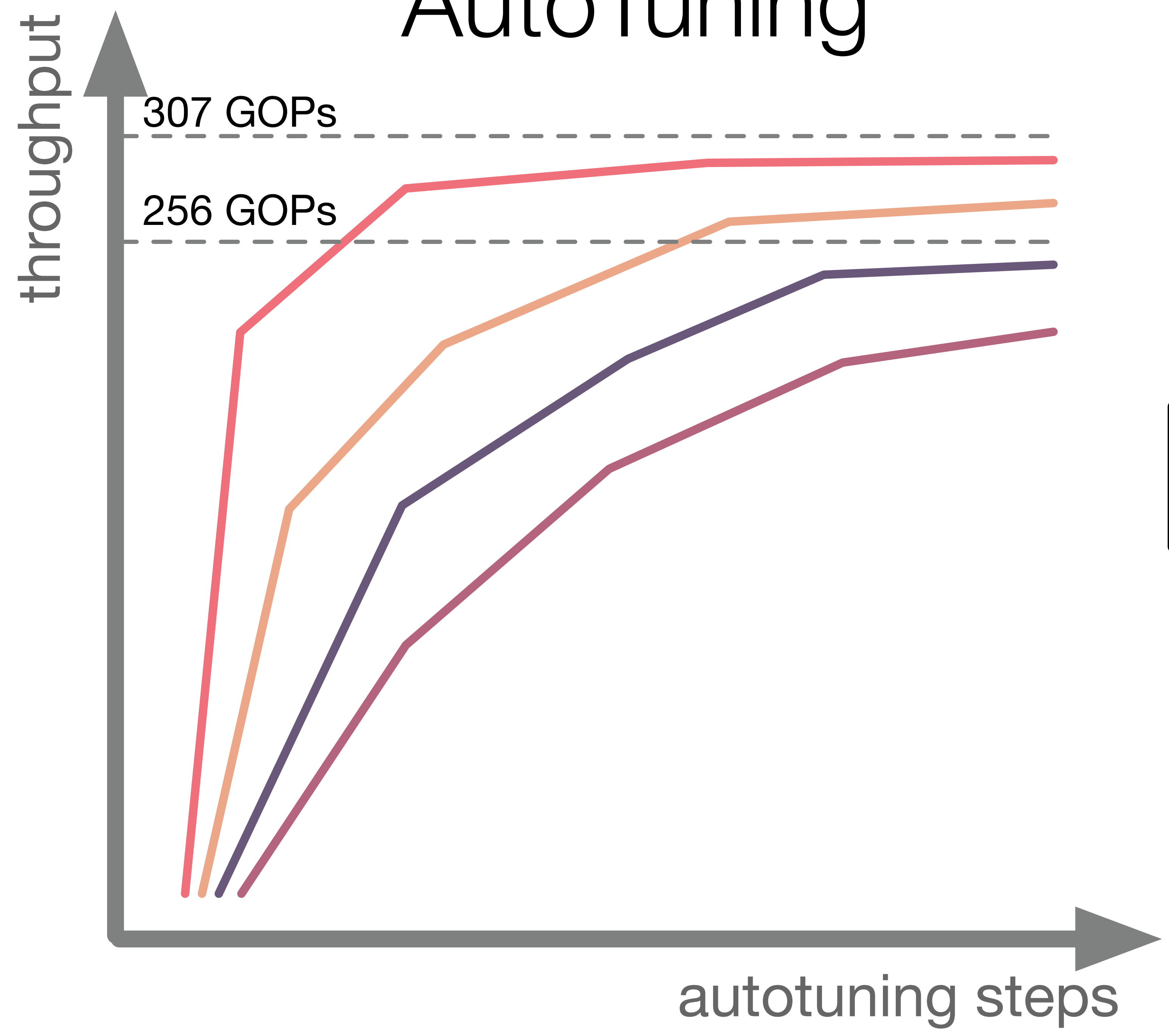
Schedule Exploration with VTA

VTA Candidate Designs

- #1 Design AAA @ 307GOPs
- #2 Design BBB @ 307GOPs
- #3 Design CCC @ 307GOPs
- #4 Design DDD @ 256GOPs

Needs to pass place & route and pass timing closure

Operator Performance AutoTuning



Deliverable

Model

Graph Optimizer

custom

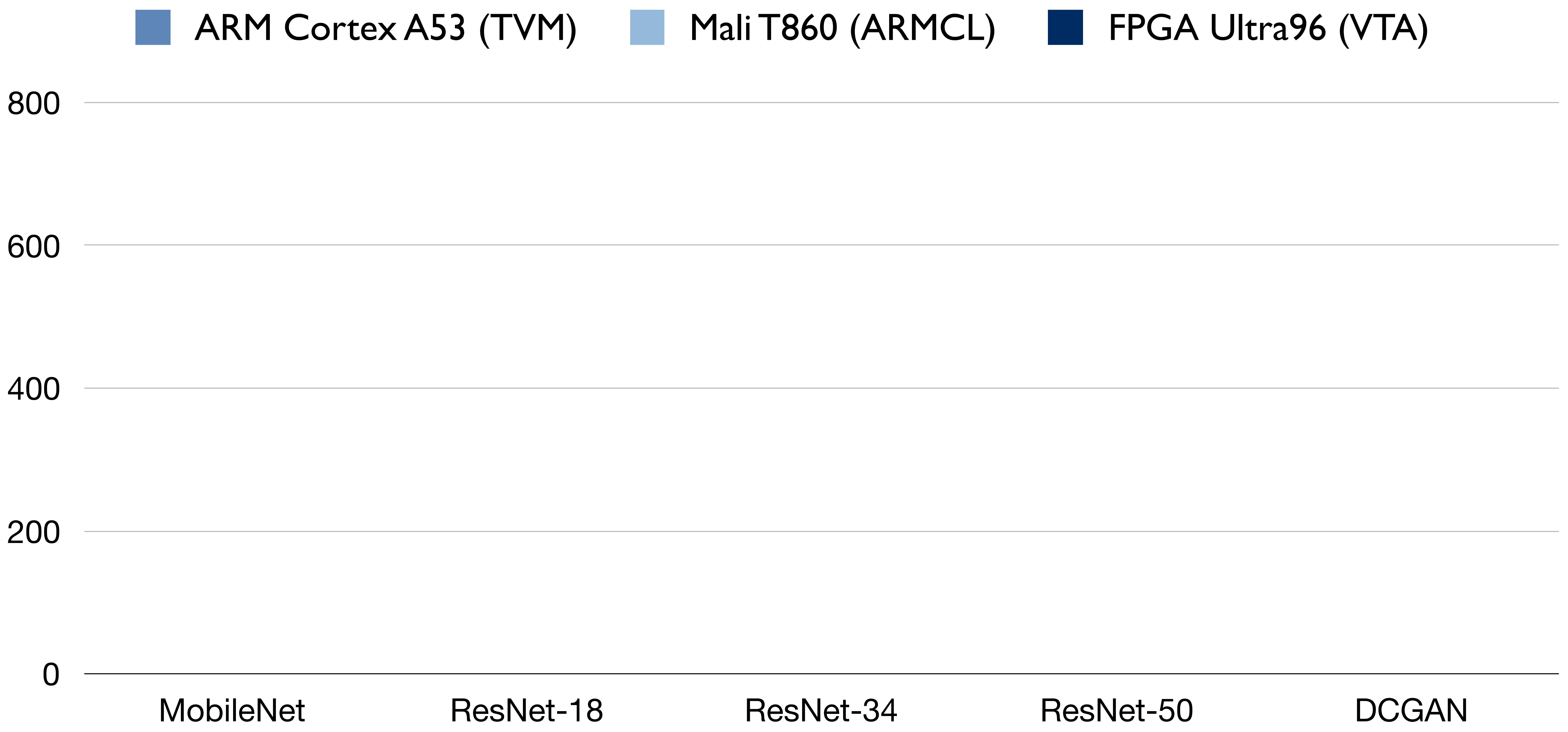
Tuned Operator Lib

VTA Design BBB

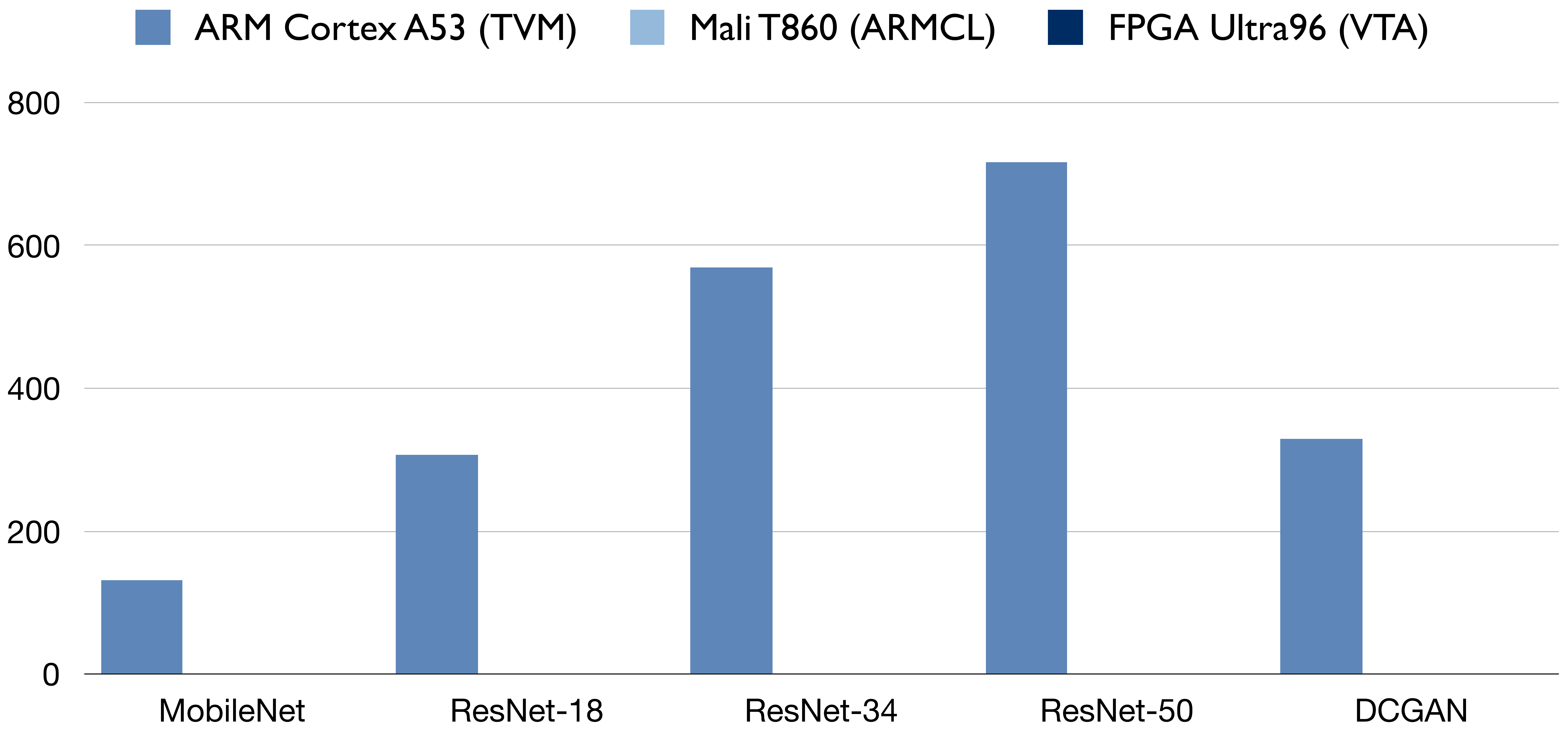
FPGA

End-to-end Performance

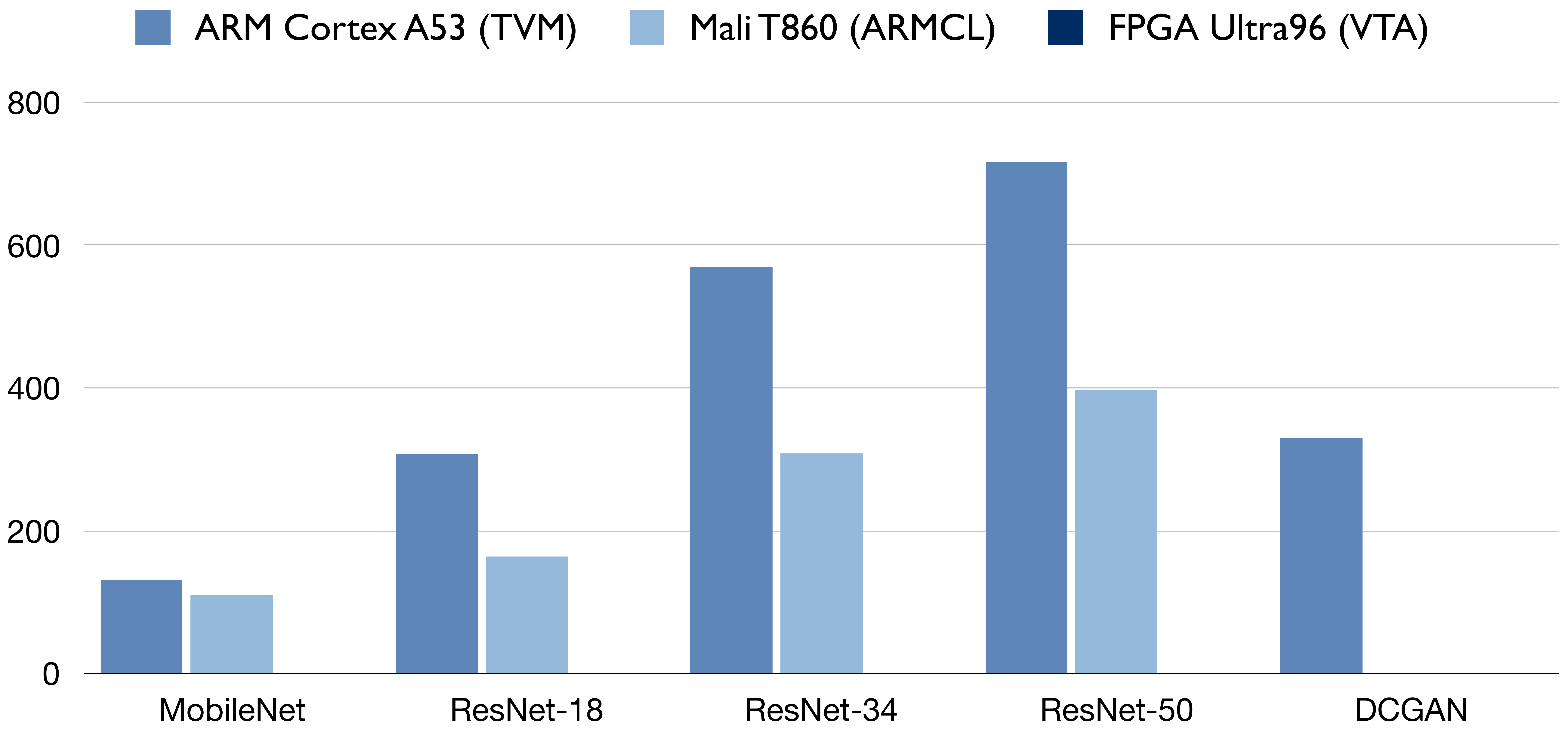
End-to-end Performance



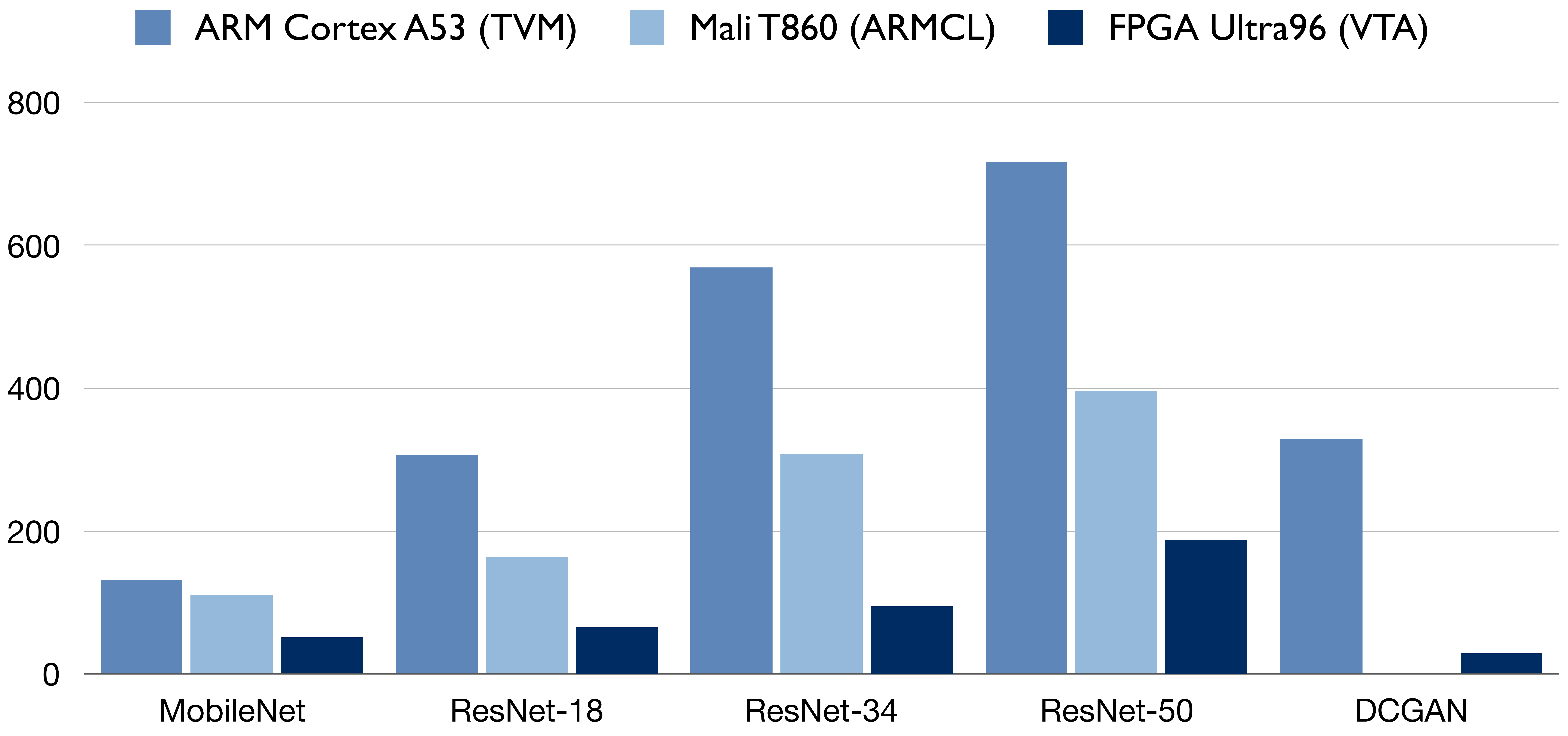
End-to-end Performance



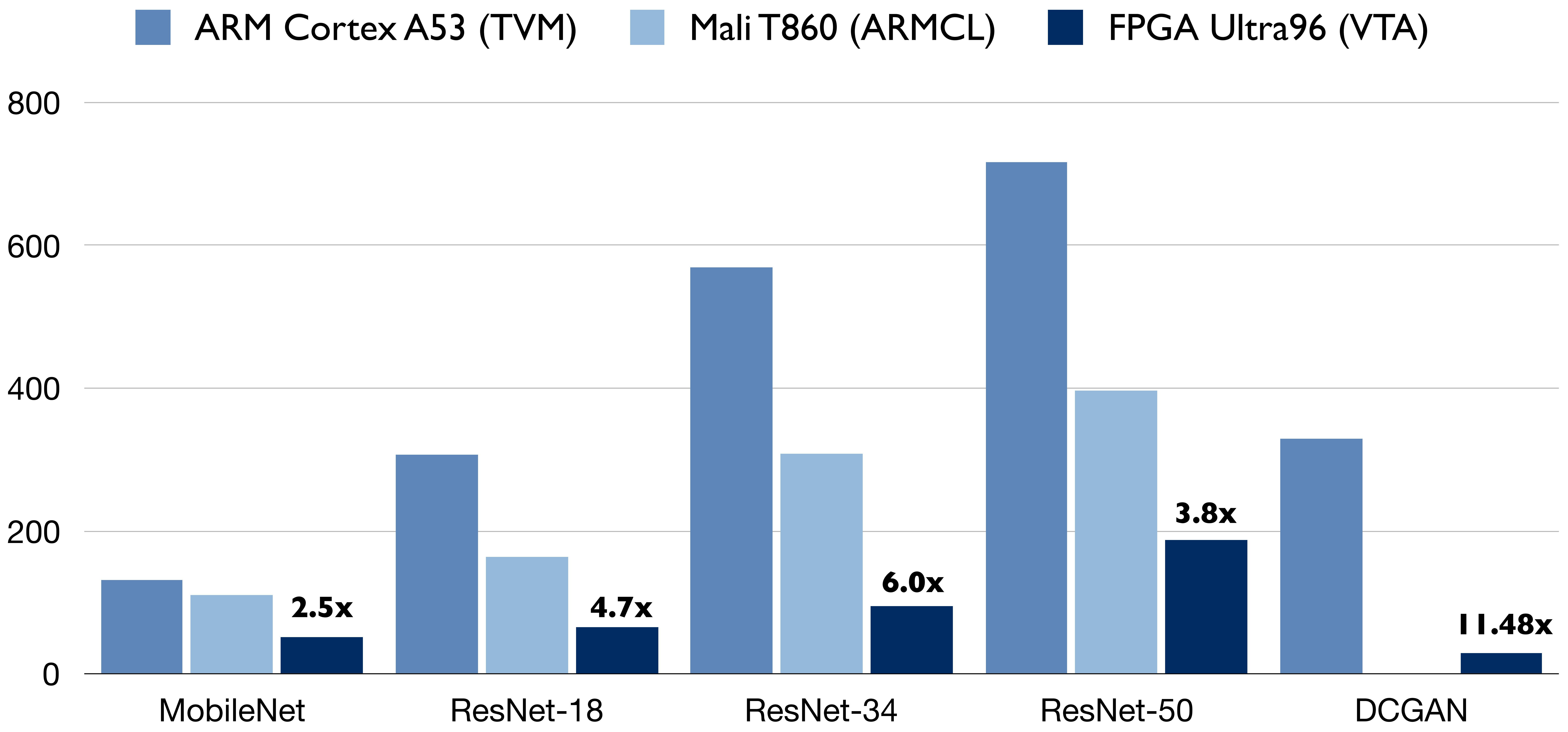
End-to-end Performance



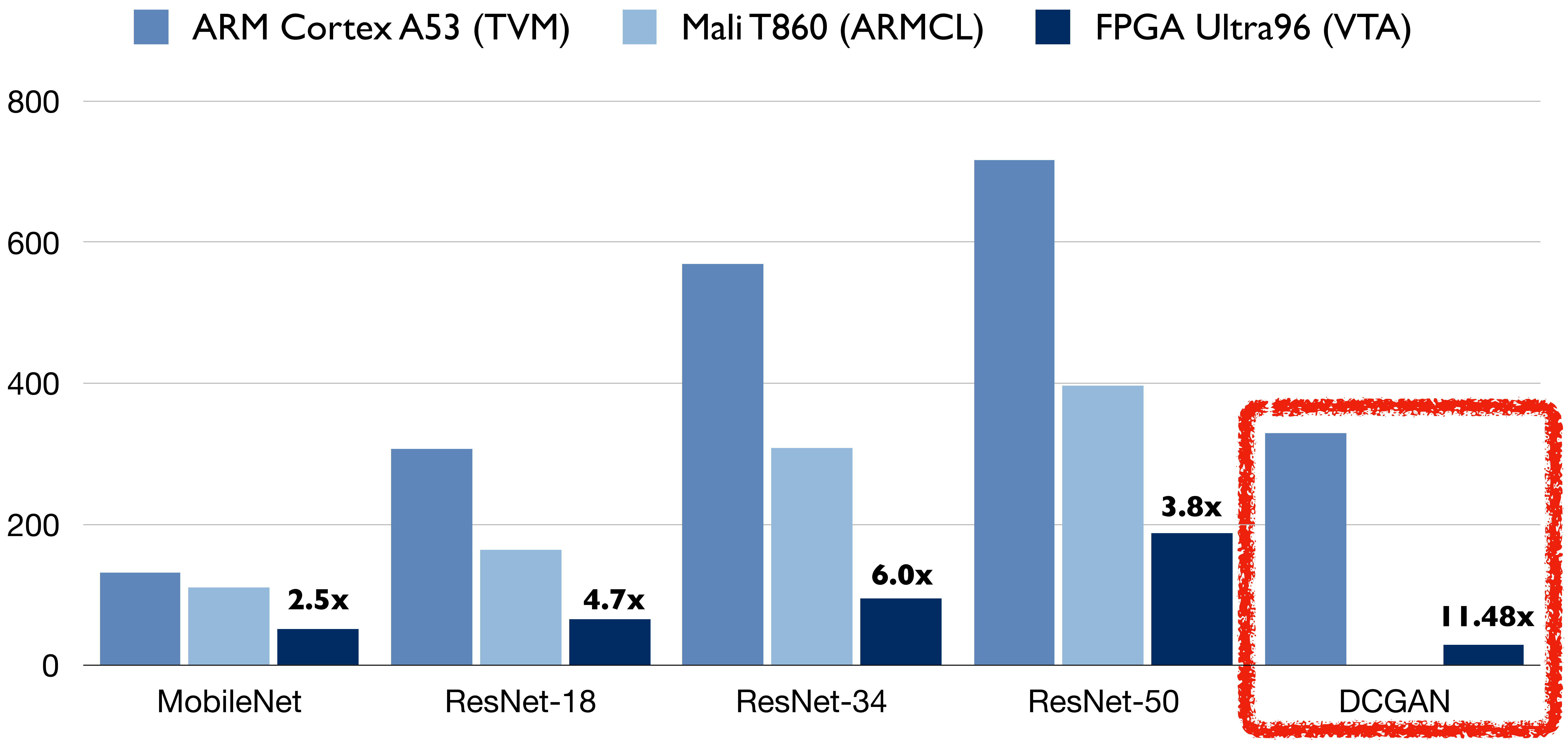
End-to-end Performance



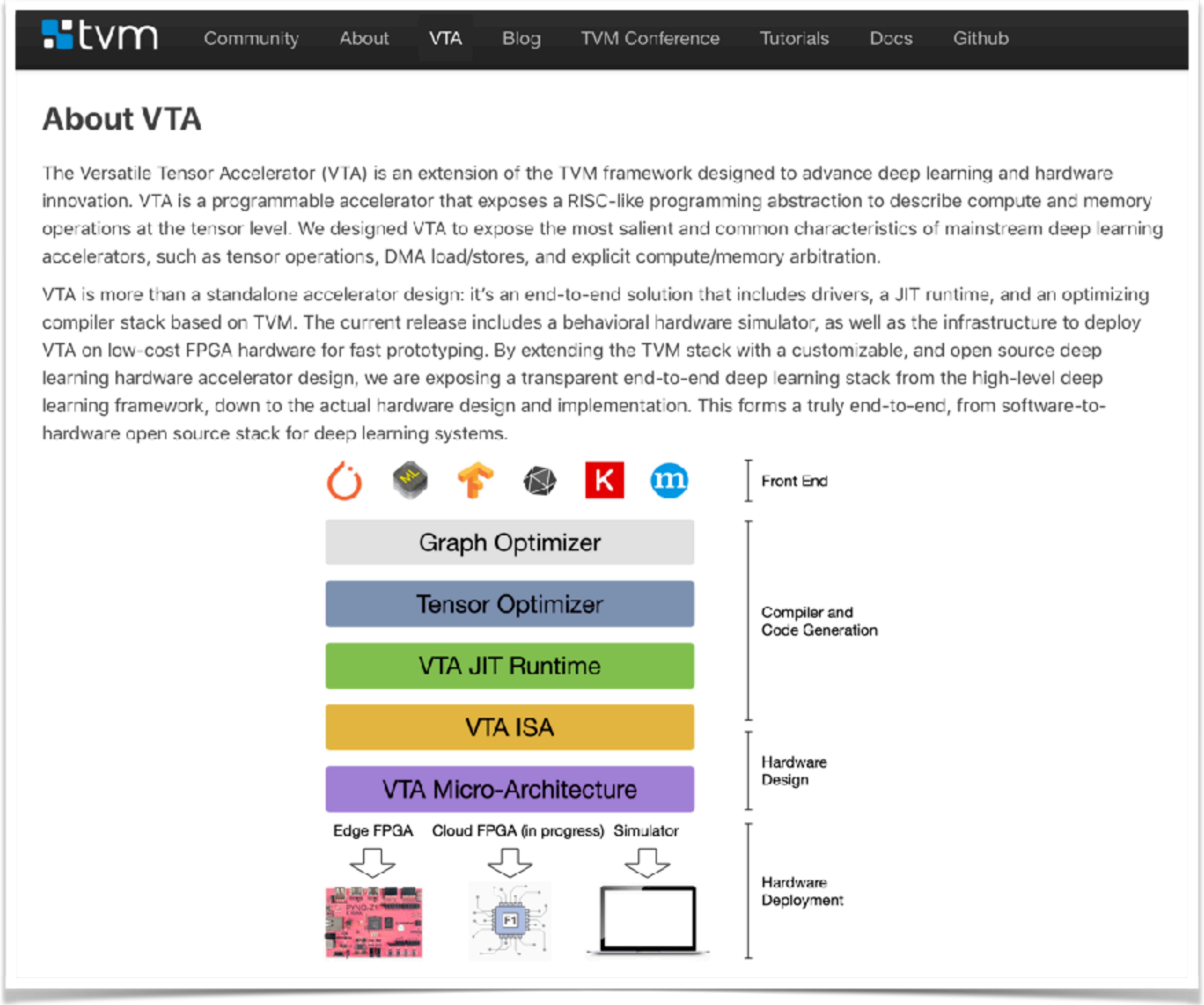
End-to-end Performance



End-to-end Performance



VTA Released in the Summer


tvm Community About VTA Blog TVM Conference Tutorials Docs Github

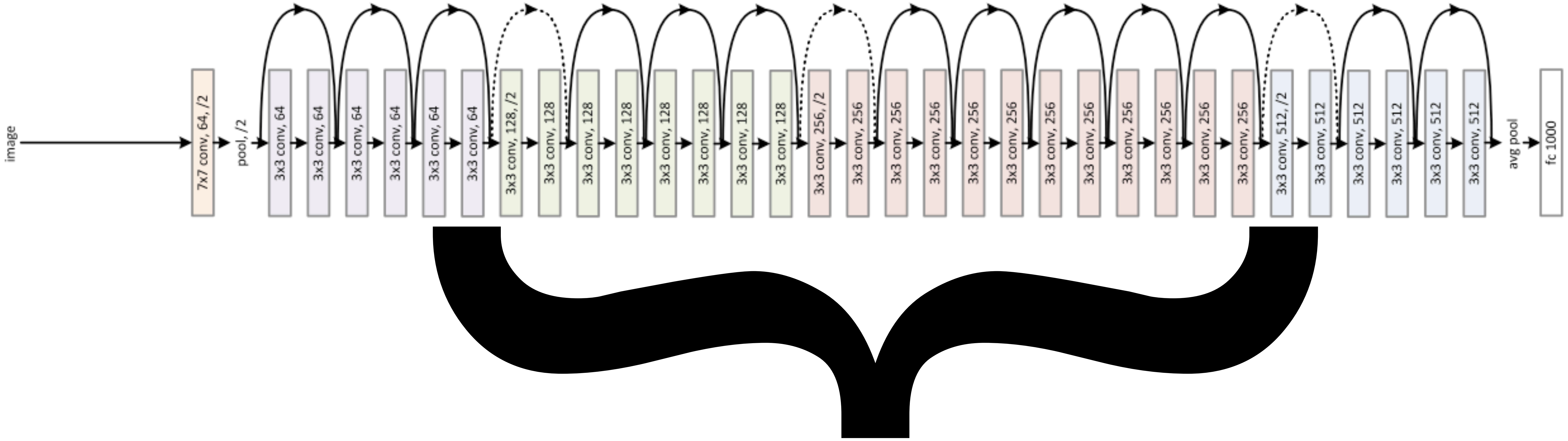
About VTA

The Versatile Tensor Accelerator (VTA) is an extension of the TVM framework designed to advance deep learning and hardware innovation. VTA is a programmable accelerator that exposes a RISC-like programming abstraction to describe compute and memory operations at the tensor level. We designed VTA to expose the most salient and common characteristics of mainstream deep learning accelerators, such as tensor operations, DMA load/stores, and explicit compute/memory arbitration.

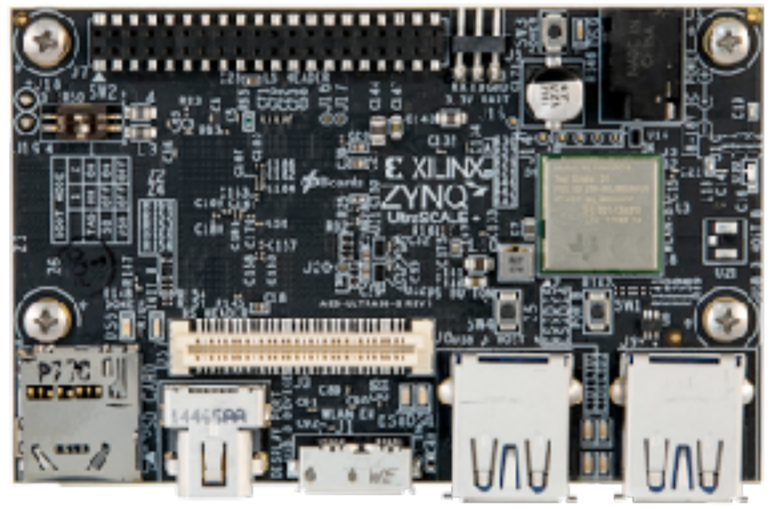
VTA is more than a standalone accelerator design: it's an end-to-end solution that includes drivers, a JIT runtime, and an optimizing compiler stack based on TVM. The current release includes a behavioral hardware simulator, as well as the infrastructure to deploy VTA on low-cost FPGA hardware for fast prototyping. By extending the TVM stack with a customizable, and open source deep learning hardware accelerator design, we are exposing a transparent end-to-end deep learning stack from the high-level deep learning framework, down to the actual hardware design and implementation. This forms a truly end-to-end, from software-to-hardware open source stack for deep learning systems.

VTA Demonstration

Based on of the box FPGA demo & tutorials that you can try on your own!

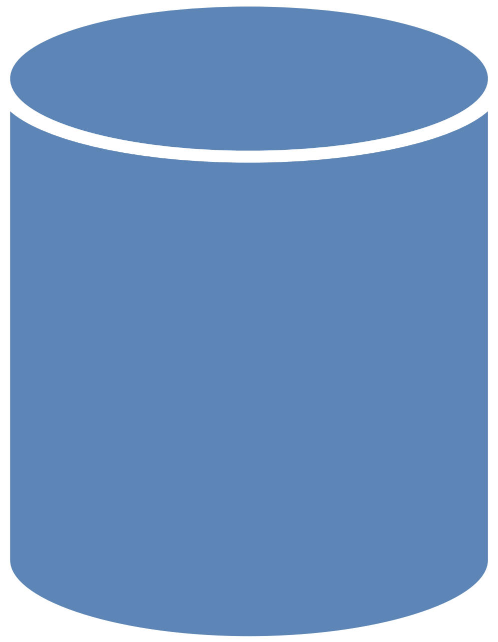


“cat”



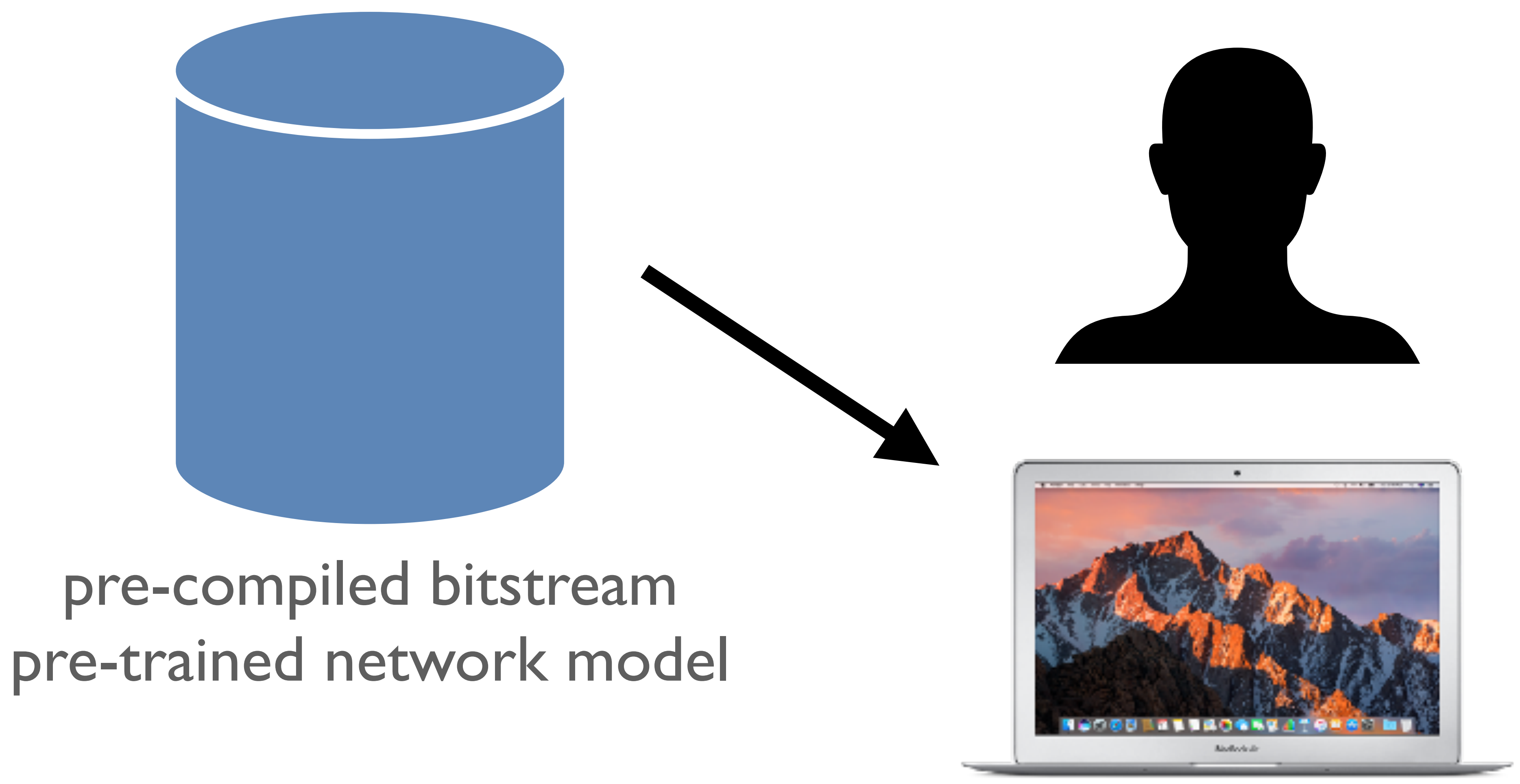
VTA Demonstration

VTA Demonstration

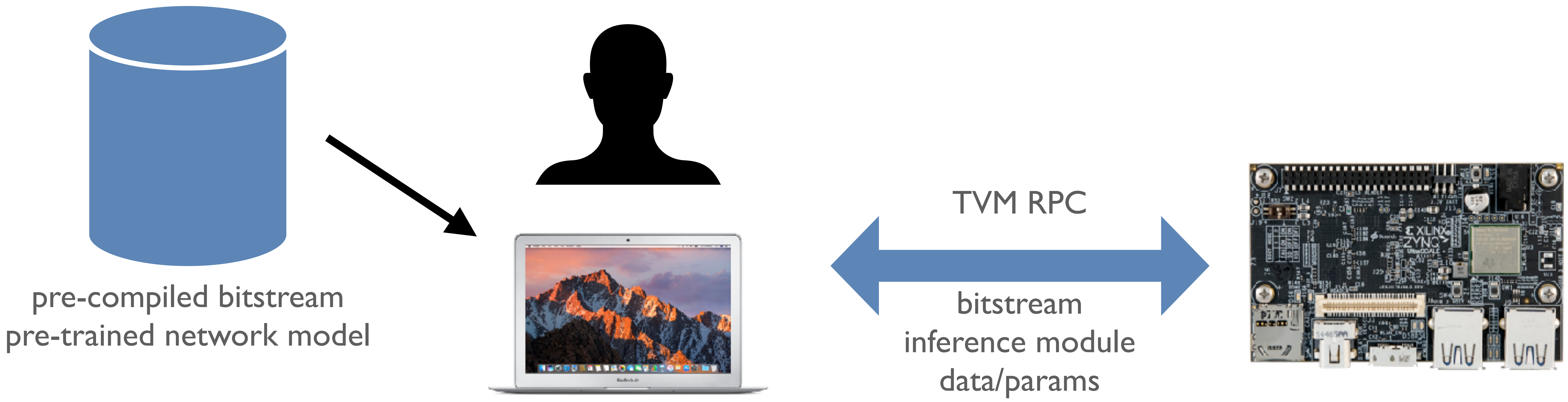


pre-compiled bitstream
pre-trained network model

VTA Demonstration



VTA Demonstration



VTA Demonstration

1. CPU Only Inference (ResNet34, W8)

2. VTA Inference (ResNet34, W8)

3. Fast VTA Inference (ResNet18, W4)

VTA Demonstration

1. CPU Only Inference (ResNet34, W8): 2.6 FPS

2. VTA Inference (ResNet34, W8): 10 FPS

3. Fast VTA Inference (ResNet18, W4): 19 FPS

TVM 0.5 VTA Release Features

TVM 0.5 VTA Release Features

- FPGA Support: Ultra96, ZCU102, Intel DE10Nano
- TOPI Operator Library & AutoTVM support
- Relay graph conversion front end, push-button 8bit quantization

2019 VTA Timeline

2019 VTA Timeline

- Q1:
  - Chisel Generator for ASIC backends
  - Initial Datacenter FPGA Prototype
- Q2:
  - Novel Numerical Representation Support (Posit)
  - Initial Training Prototype

More at tvm.ai/vta