A Hardware-Software Blueprint for Flexible Deep Learning Specialization

Thierry Moreau

ARM Research Summit Presentation, September16th 2019

sampl

PAUL G. ALLEN SCHOOL
OF COMPUTER SCIENCE & ENGINEERING

tvm.ai

This Talk

- Introduce VTA, the open source DL accelerator compiled through TVM
- Walk through TVM compilation process to get a model running on VTA
- Discuss hardware-software co-design study

We are in the middle of a golden age of DL Specialization

(credit: http://basicmi.github.io/AI-Chip/)

Tech Giants/System

Google
Microsoft
facebook
aws
Apple
IBM
Alibaba Group 阿里巴巴集团
HUAWEI
Baidu 百度
Tesla
Hewlett Pack Enterprise
FUJITSU
DELL
Western Digital
NOKIA
LG

IC Vender/Fabless

intel
SAMSUNG
NVIDIA
QUALCOMM
AMD
XILINX
MEDIATEK
UNISOC
MARVELL
Rockchip 瑞芯微电子
Ambarella
NationalChip

Automated Driving

NXP
Texas Instruments
RENESAS
TOSHIBA
ST

Startup in China

Cambricon 寒武纪科技
地平线 Horizon Robotics
BITMAIN
intelliFusion 云天励飞
Think Force
NextVPU 肇观电子
Canaan
Enflame
INTENGINE 探境科技
清微智能 TSING MICRO
亿智科技
WITHMEM 忆芯科技

Smart Voice

ChipIntelli
Rokid
AISPEECH 思必驰 专注人性化的智能语音
Unisound 云知声

Startup Worldwide

Cerebras
WAVE COMPUTING
Graphcore
SambaNova SYSTEMS
habana
HAILO
thinci
KALRAY
groq
Tachyum
Esperanto TECHNOLOGIES
PEZY Computing
Eta Compute
GREENWAVES TECHNOLOGIES

FPGA/eFPGA

Achronix
flexlogix

Processing in Memory

MYTHIC
SYNTIANT
AREANNA
gyrfalcon technology

Optical Computing

LIGHTELLIGENCE
LIGHTMATTER
Optalysys

Neuromorphic

aiCTX
brainchip
Preferred Networks
Kneron
INNOGRIT
NeuroBlade
NOVUMIND

More at https://basicmi.github.io/AI-Chip/

IP/Design Sevice

arm
SYNOPSYS
Imagination
CEVA
cadence
SiFive
ARTERIS IP
Moortec

Design service with In-house IP

VeriSilicon
BROADCOM
GUC
alchip
FARADAY
eSilicon

Compilers

TensorFlow XLA
GLOW
tvm
NVIDIA TensorRT
plaidML
nGraph
Tiramisu Compiler
ONNC
The Tensor Algebra Compiler (taco)

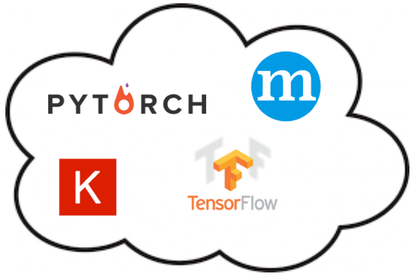
Benchmarks

MLPerf
AI - Benchmark
AI Matrix.
AIIA 中国人工智能产业发展联盟 Artificial Intelligence Industry Alliance
DAWNBench
MLMark An EEMBC Benchmark

Compilation Challenges for Novel Hardware

Architecture/VLSI Researcher

I built a new chip, how can I run some cool models on it?



Graph Compiler

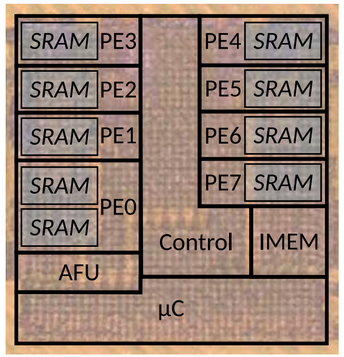
Autotuner

Tensor Compiler

Code Generator

Runtime

Drivers

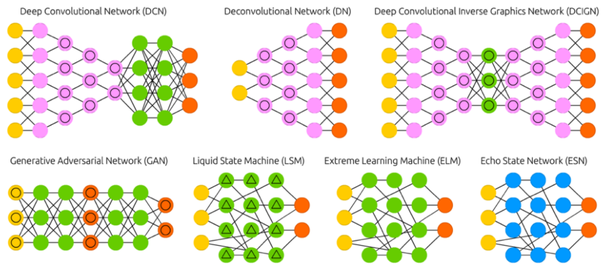


Compilation Challenges for Novel Hardware

Building a software compiler can be a huge engineering burden

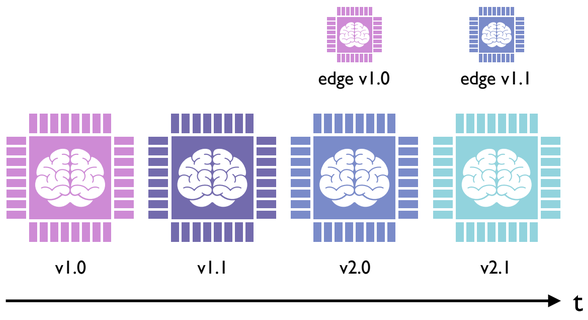
front-ends are numerous

new models get introduced



credit: asimovinstitute.org

continuous hardware design



too many moving parts can make software maintenance a huge burden!

TVM: an open source deep learning system stack for diverse hardware (see tvm.ai)

Relay: High-Level Differentiable IR

TVM: Tensor Expression IR

LLVM | CUDA | Metal

VTA Runtime & JIT Compiler

VTA Meta-ISA

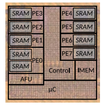
VTA Meta-Architecture

ARM/x86 CPU

GPU

iOS

FPGA

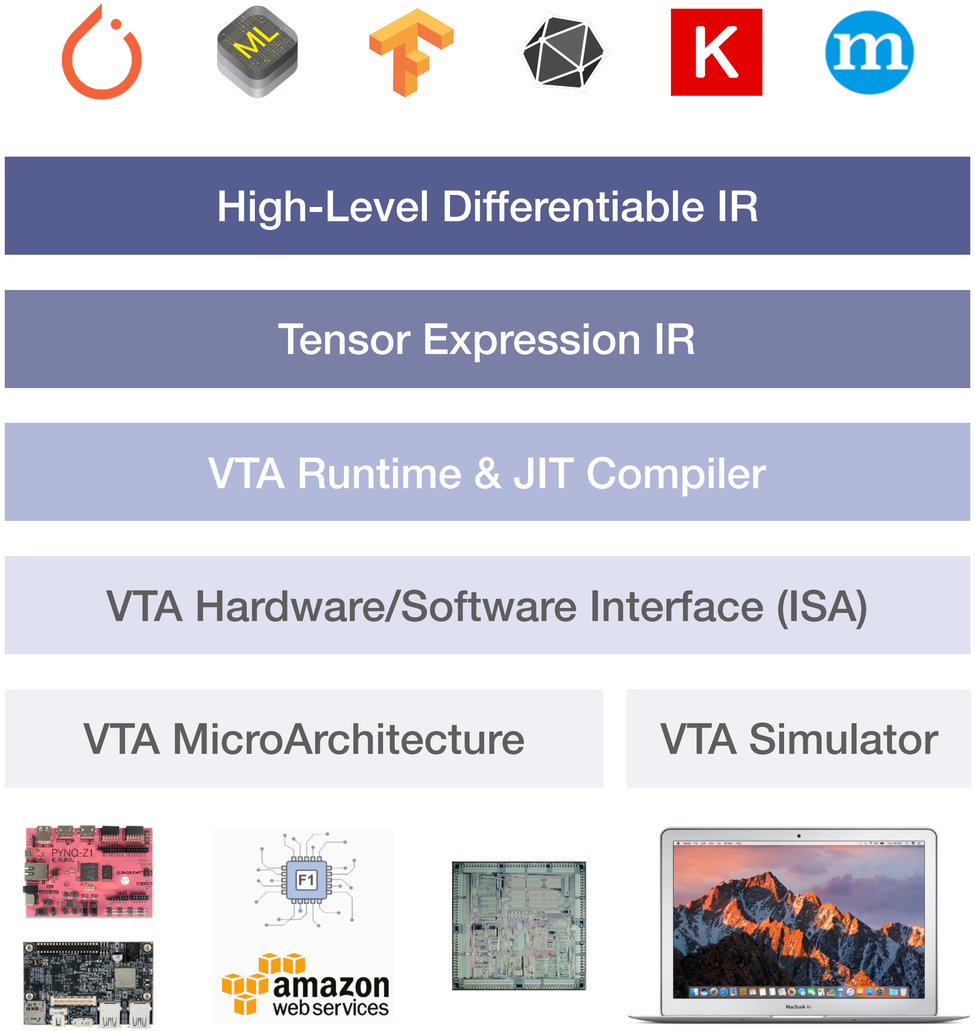


ASIC

- ✅ Model translation to Relay
- ✅ Rich graph-level transformations (quantization etc.)
- ✅ Flexible and automated schedule optimizations
- ✅ Plug-in code-generation backends

Chen et al., OSDI 2018

TVM+VTA Stack Overview

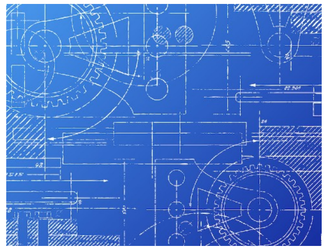


Versatile Tensor Accelerator Stack (VTA)

VTA Backends

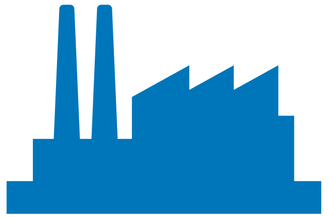
- **Fast SIM**: out-of-the-box testing to write compiler passes
- **Cycle-accurate SIM (TSIM)**: RTL simulation with Verilator
- **FPGA**: full system prototyping

VTA Goals



Blue-print for a complete deep learning acceleration stack

Experimentation framework for cross-stack deep learning optimizations



Open-source community to facilitate tech transfer and innovation

VTA Overview

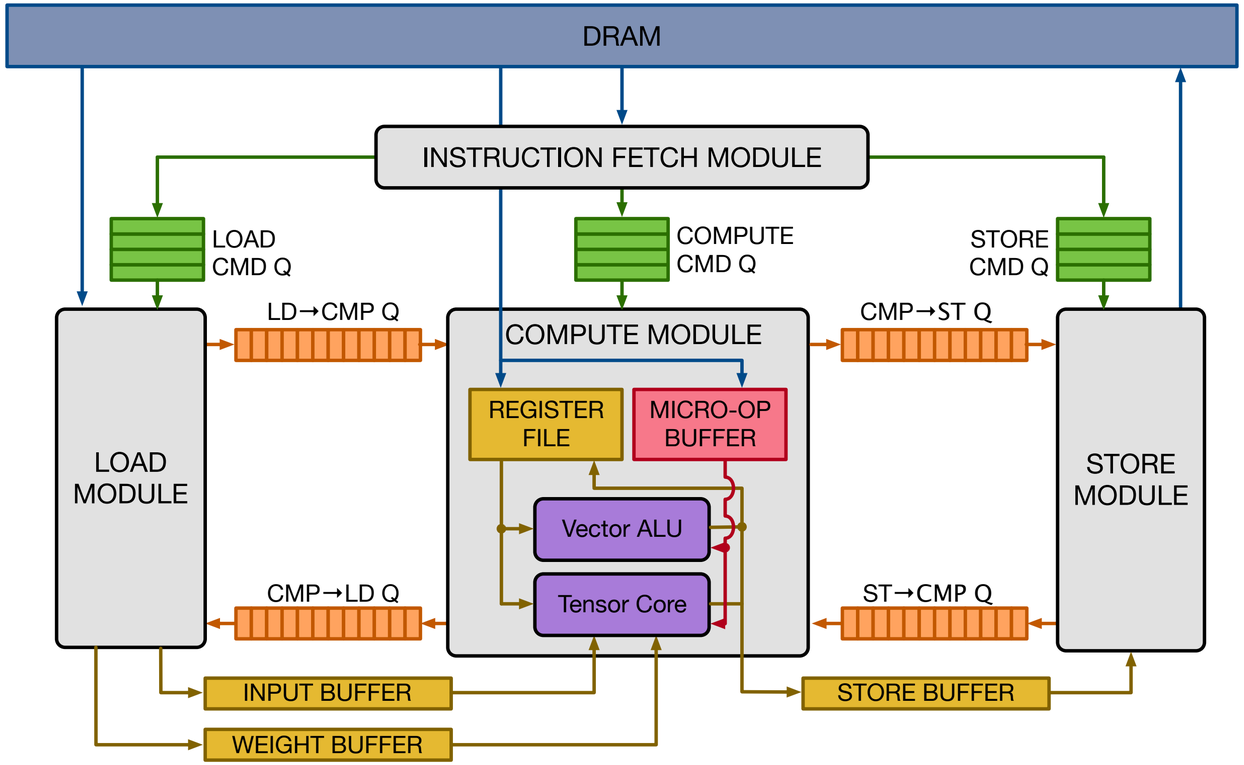
Flexible Hardware Architecture

Programmability Challenges

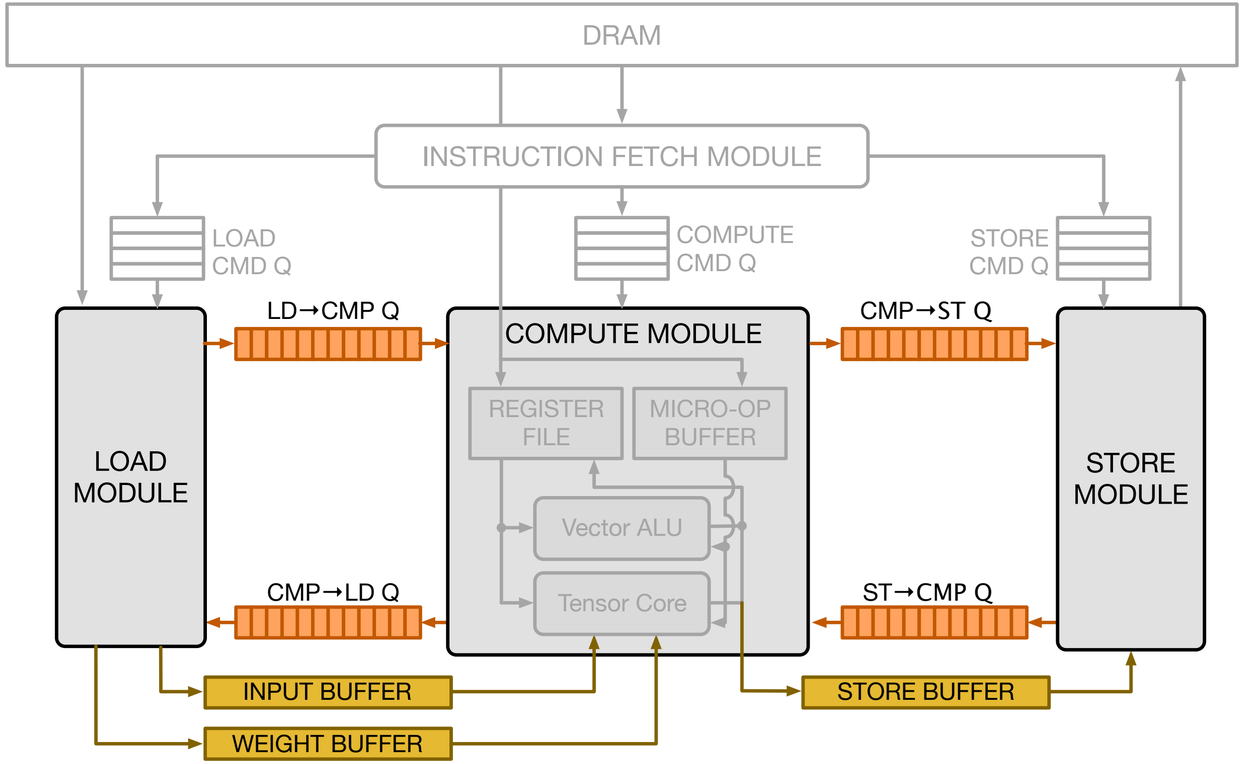
Hardware-Software Co-Design

VTA Hardware Architecture

Philosophy: simple hardware, provide software-defined flexibility



VTA Hardware Architecture

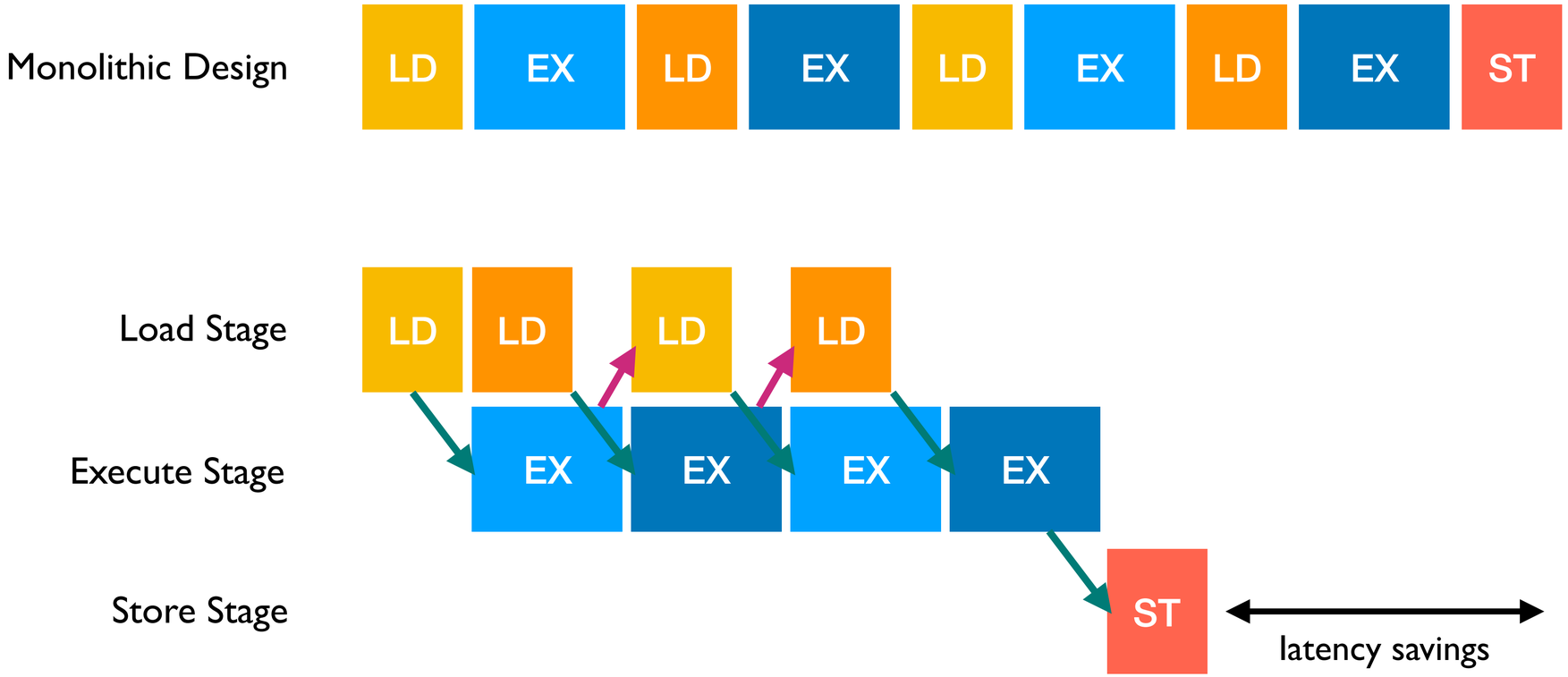


Pipelining Tasks to Hide Memory Latency

Monolithic Design

LD: load
EX: compute
ST: store

Pipelining Tasks to Hide Memory Latency



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

- Use command-level instructions to perform multi-cycle tasks

DENSE | ALU | DMA LOAD | DMA STORE

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

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

VTA RISC Micro-Kernels

multiple micro-ops define a **micro-kernel**,
which can be invoked by a high-level 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?

```
// 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)                                                // EX@T
0x08: POP (EX->ST)                                                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  UF
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   [ 0
0x1D: PUSH(EX->ST)
0x1E: POP (EX->ST)
0x1F: STOR(STBUF[ 0- 7])
// Virtual Thread 3
0x20: POP (EX->LD)                                                // E
0x21: LOAD(ACTIV[25-50])                                          LD
0x22: LOAD(LDBUF[32-63])                                          //  D@T
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

Programming accelerators is hard!!!

```
// 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_ds    = MAX(rf[idx_dst^0], rf[idx_src^0])
x_ds    = MAX(rf[idx_dst^1], rf[idx_src^1])

 bat  norm
r[i  _dst^m] = MUL(rf[idx_dst^m], rf[idx_src^m])
rf  dx_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])
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

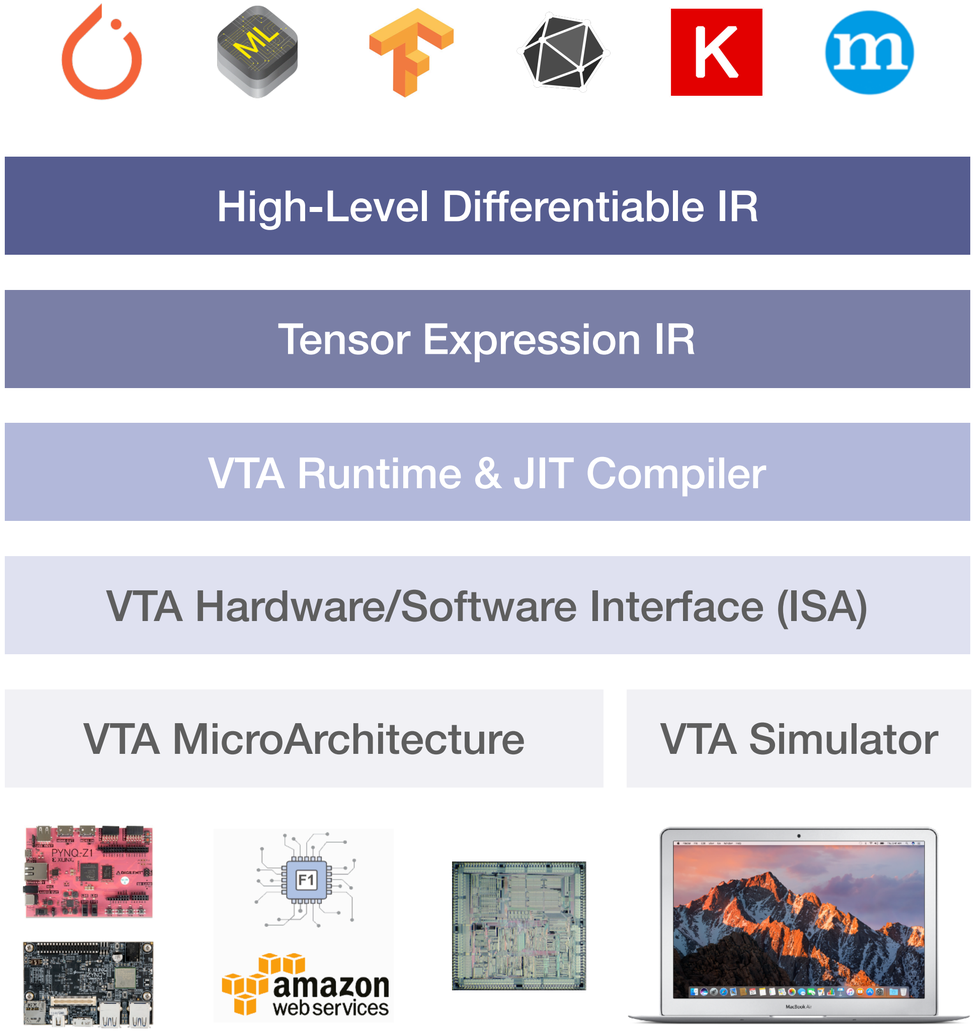
VTA Overview

Flexible Hardware Architecture

Programmability Challenges

Hardware-Software Co-Design

Programmability Challenges

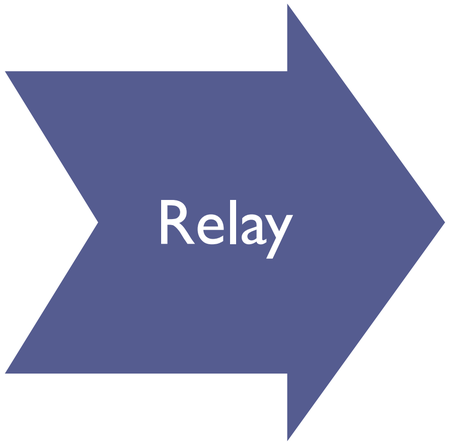


- How does one utilize Relay passes to transform a graph for VTA?
- How do we manipulate tensor expressions to build a library for VTA?
- How does the VTA low-level JIT facilitate code-generation?

Compilation Stages

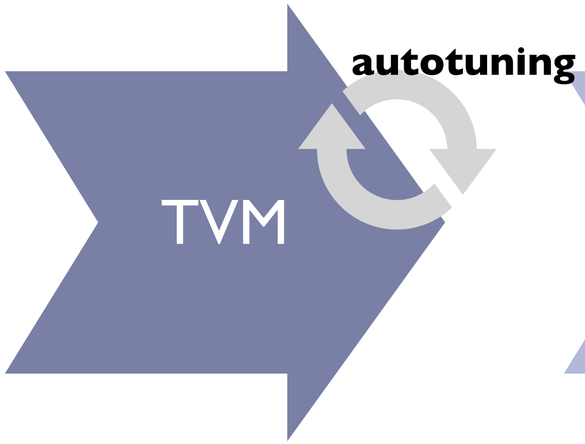
Model from Gluon Zoo

1. Graph Compilation



quantization
re-writing
fusion
partitioning

2. Operator Compilation

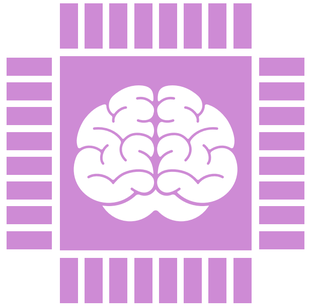


tiling
virtual threads
tensorization
lowering

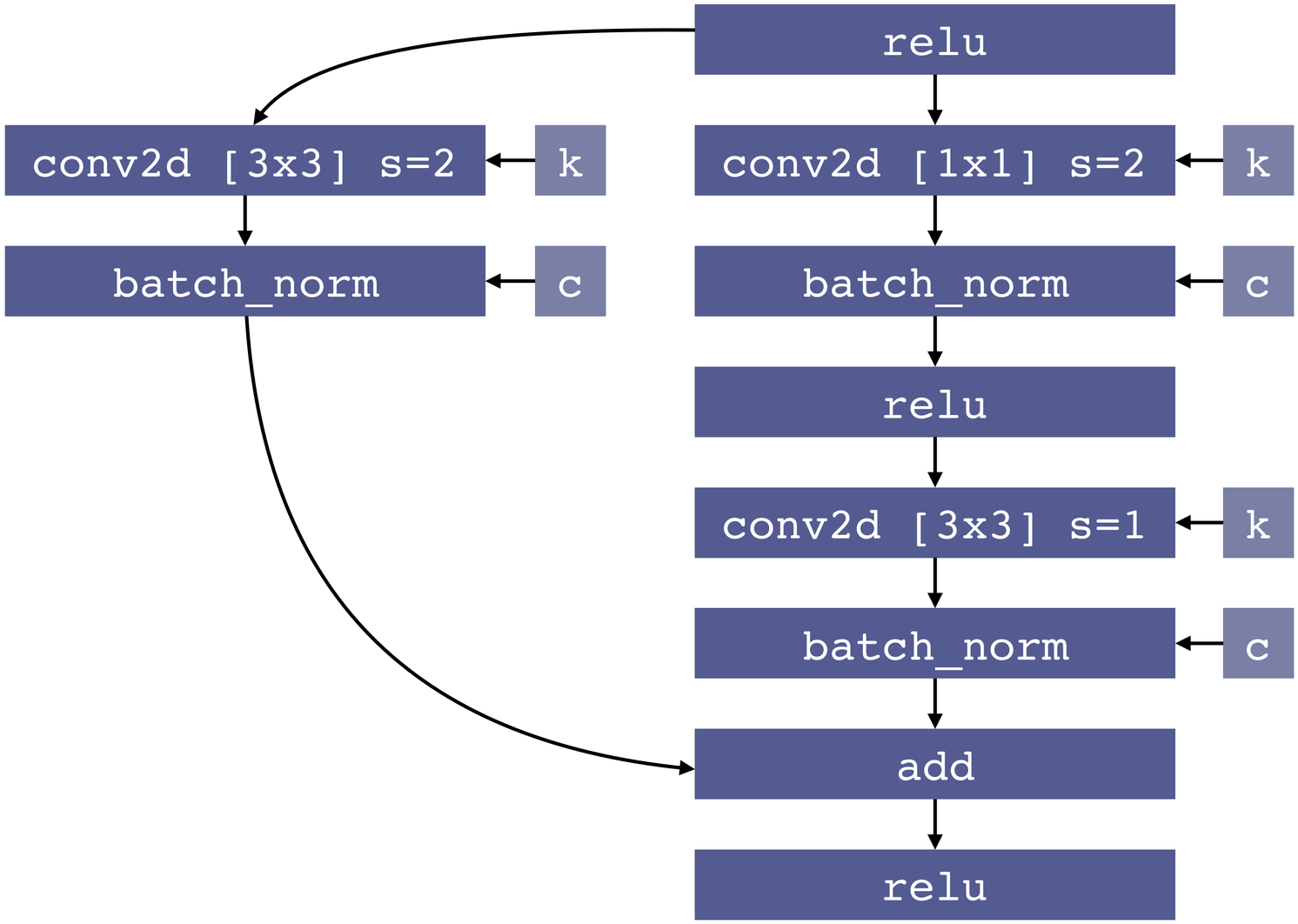
JIT Compilation

code generation to VTA ISA
instruction management

Offload to VTA



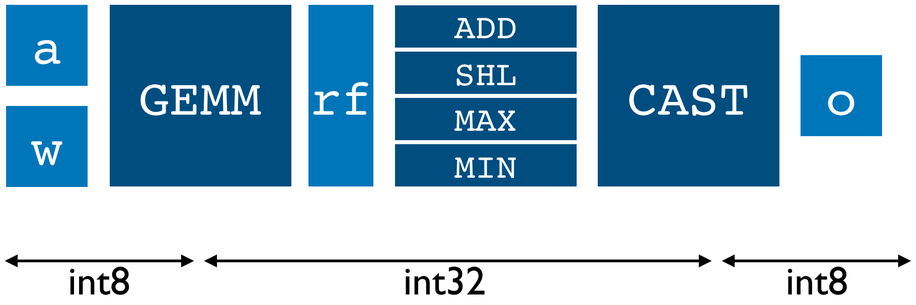
ResNet Compilation Relay Example



Graph Properties

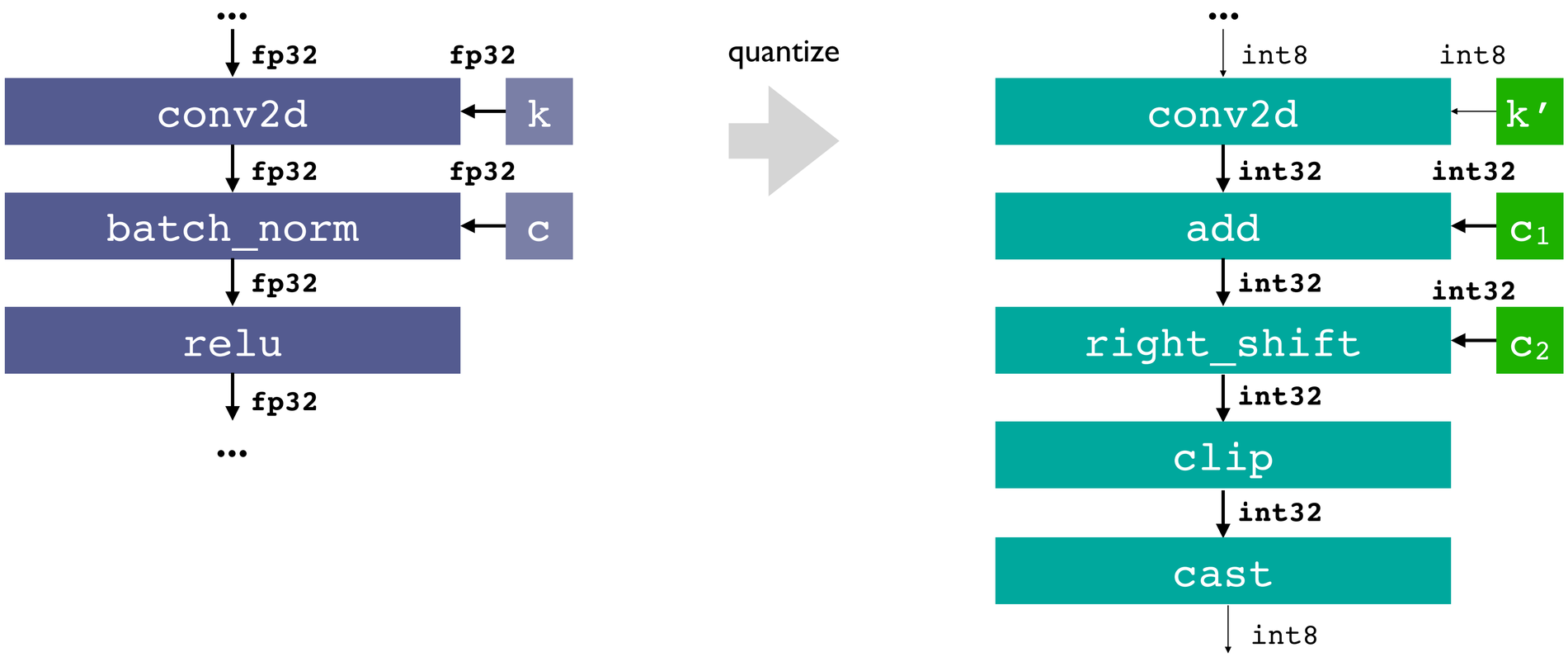
```
dtype:      fp32
activation: NCHW
kernels:    OIHW
```

VTA Pipeline

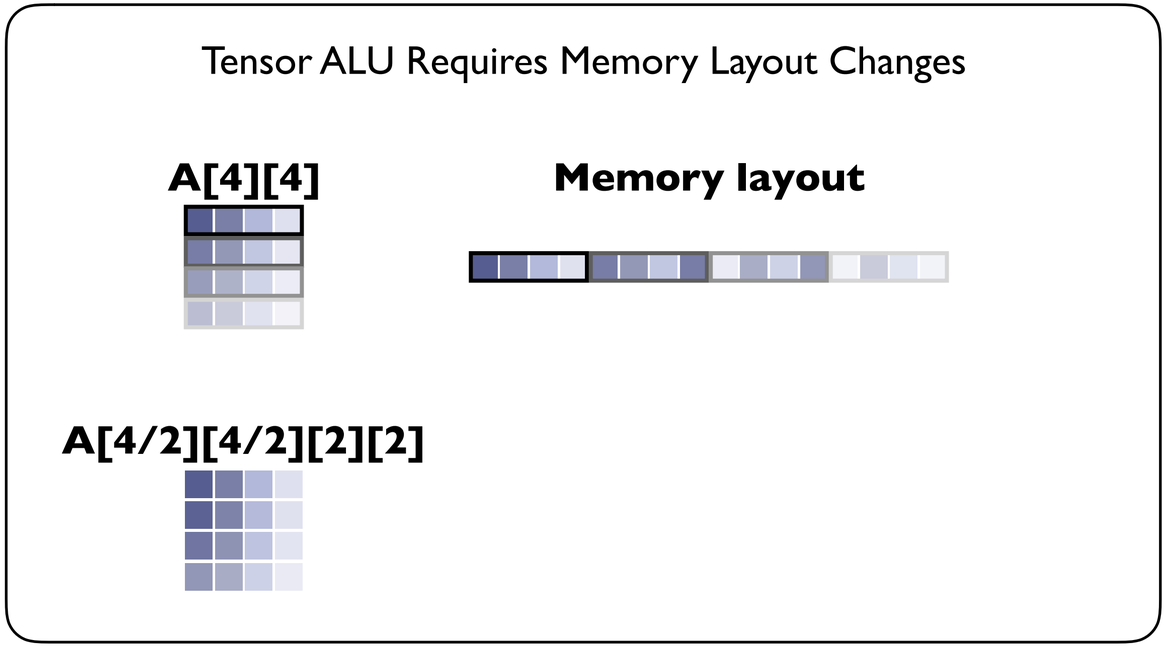


Graph Pass #1: Quantization & Substitutions

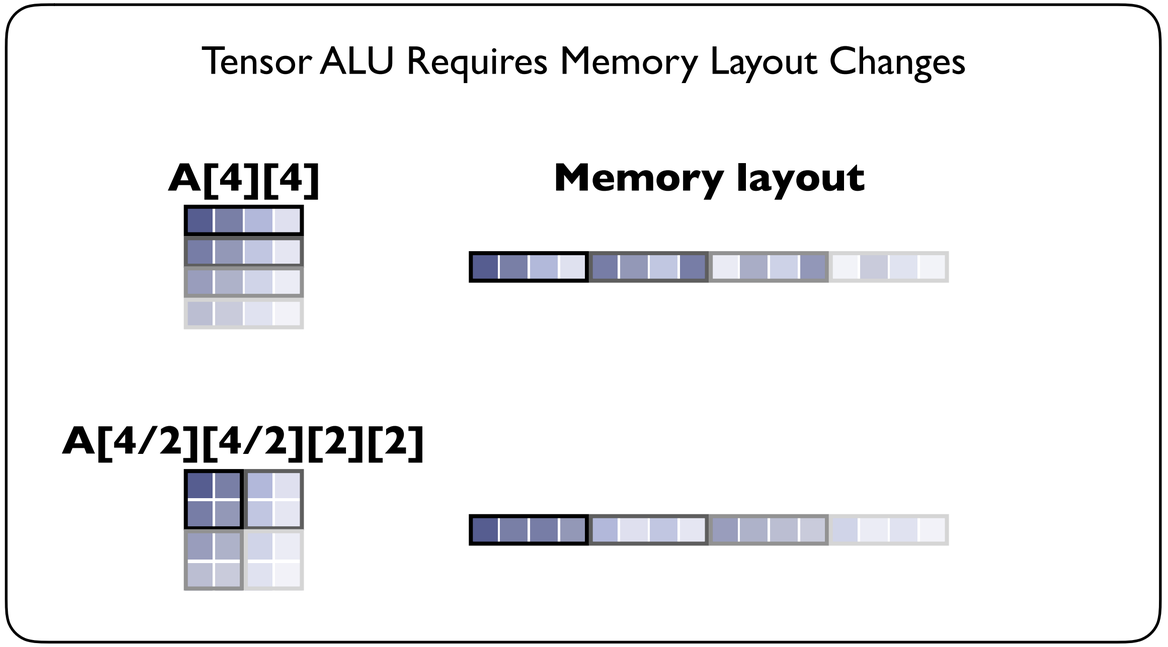
- The goal of quantization is to convert nodes that typically process fp32 data to instead consume 8bit or 32bit integers without significantly degrading accuracy.
- Since VTA has no multipliers, we fold batch normalization constants into the convolution kernels to rely solely on add and shift operations during batch norm.



Graph Pass #2: Data Packing

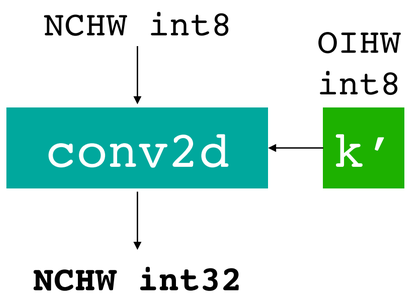


Graph Pass #2: Data Packing

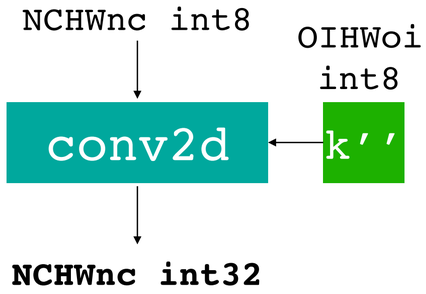


Data Layout

```
activation: NCHW
kernels:    OIHW
```

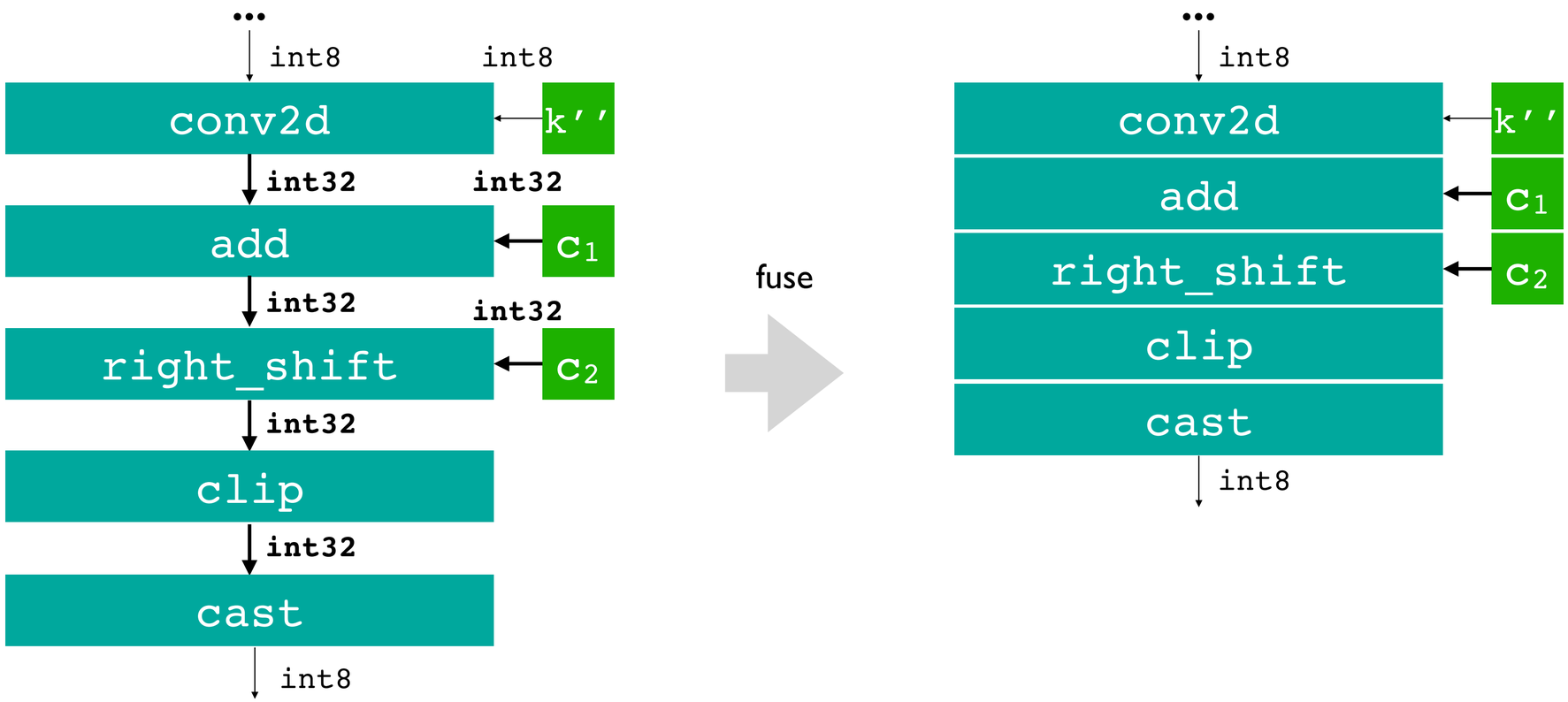



```
activation: NCHWnc
kernels:    OIHWoi
```



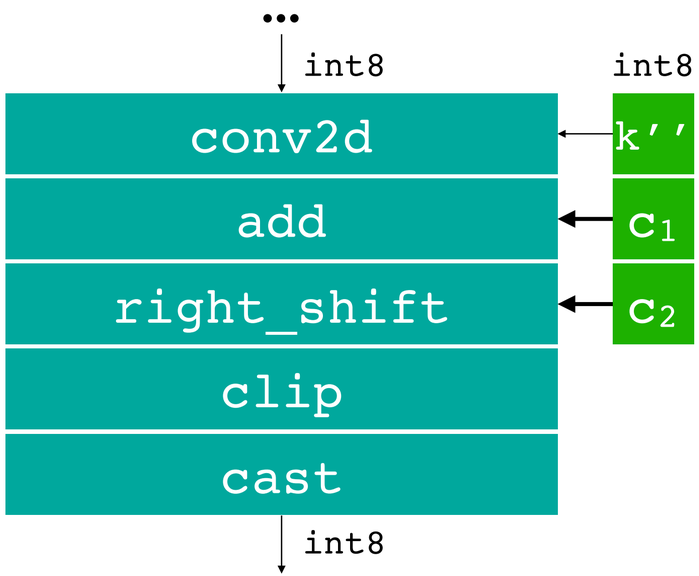
Graph Pass #3: Operator Fusion

- Idea: fuse as many operators to the VTA hardware pipeline to minimize DRAM access



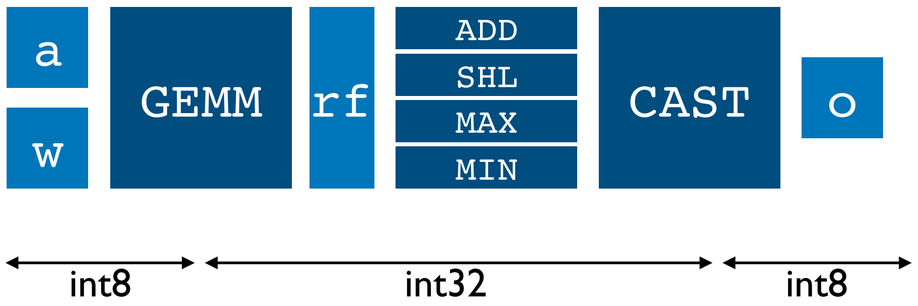
Graph Pass #3: Operator Fusion

Fused Conv-Batch-Relu

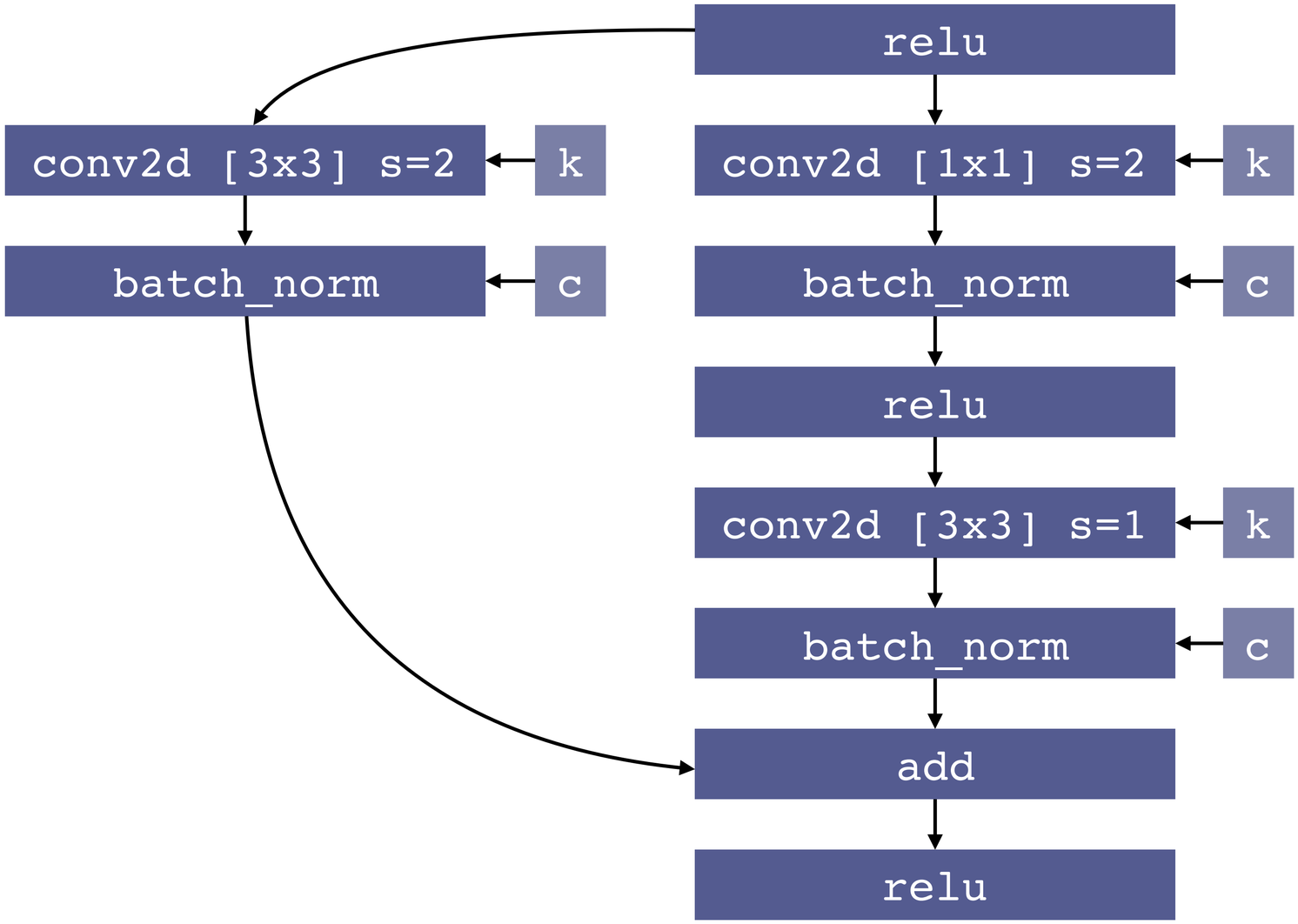


maps

VTA Pipeline



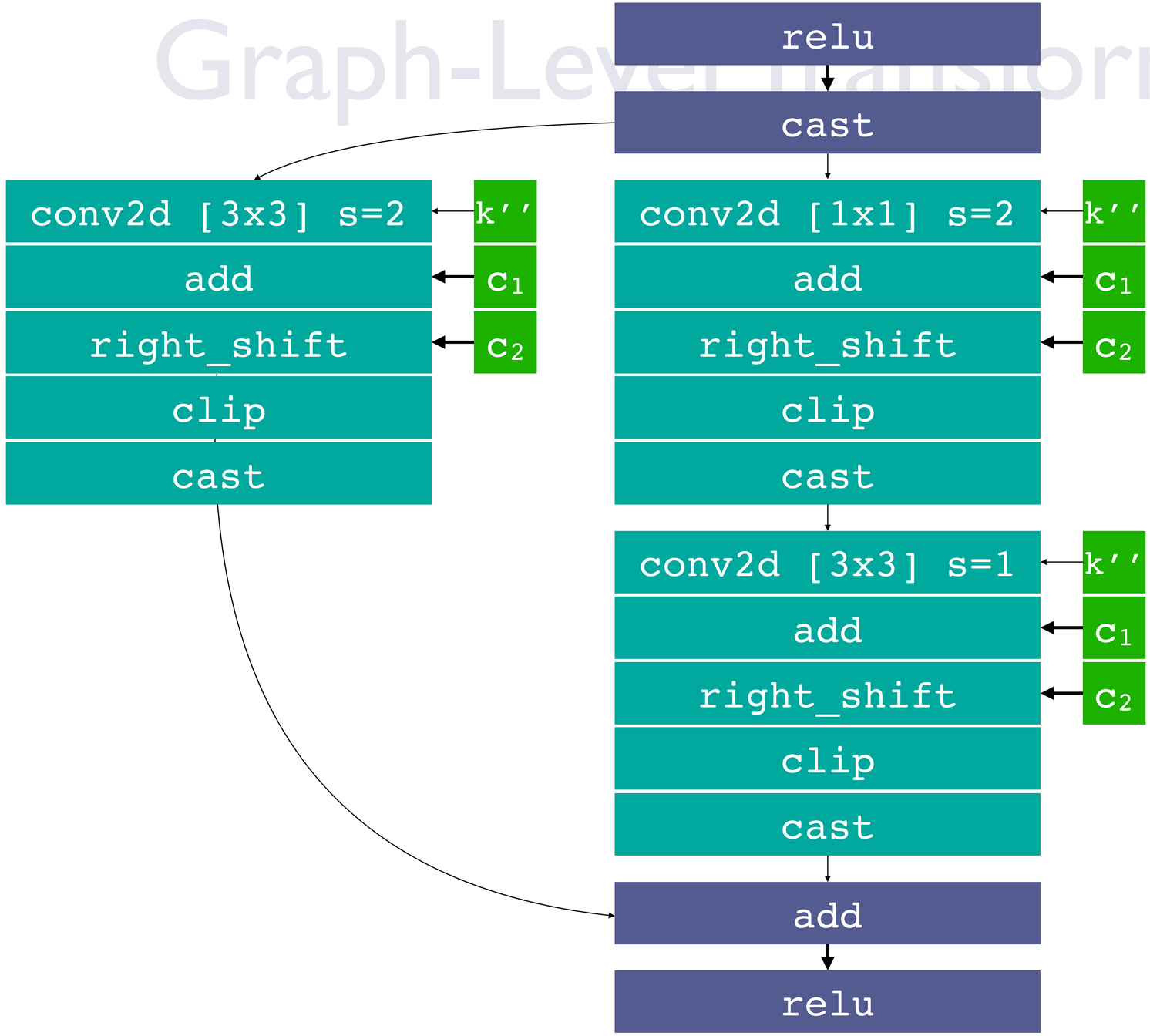
Graph-Level Transformations Recap



Graph Properties

```
dtype:      fp32
activation: NCHW
kernels:    OIHW
```

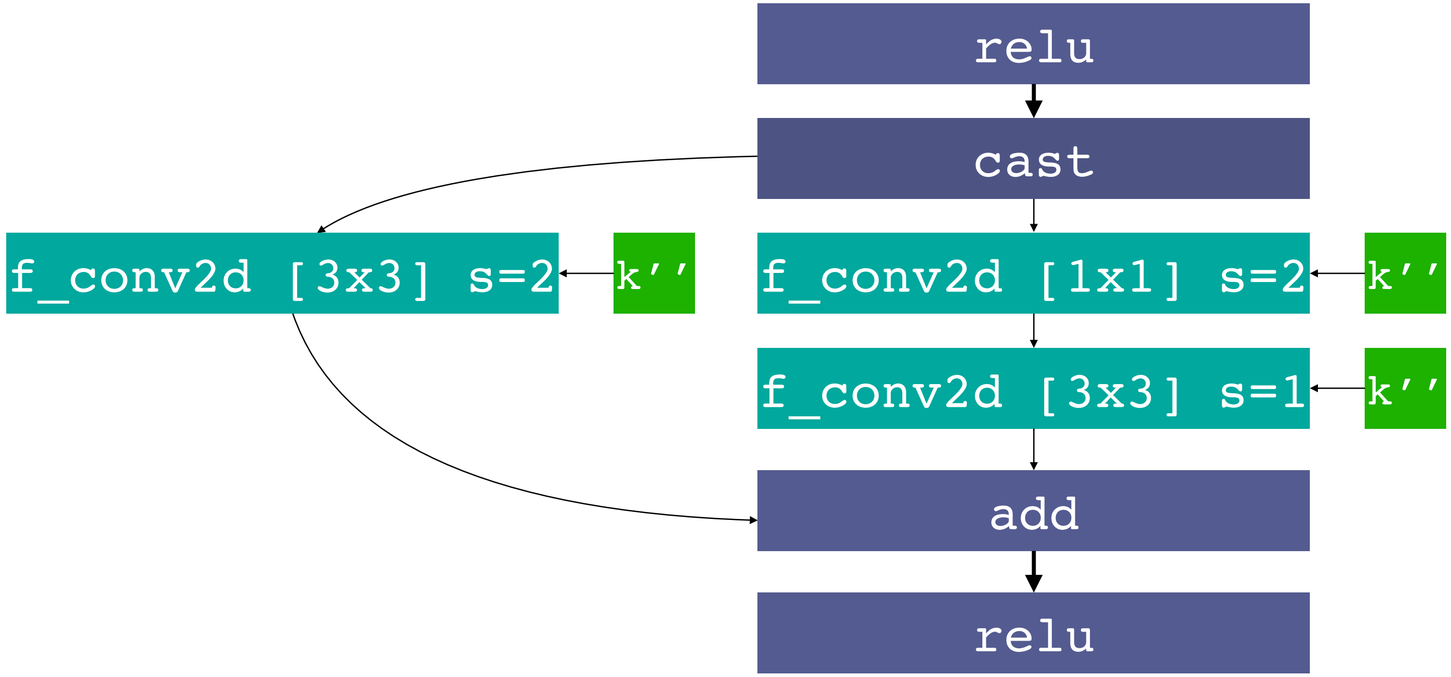
Graph-Level Transformations Recap



Graph Properties

```
dtype:        int8/32
activation:   NCHWnc
kernels:      OIHWoi
```

Graph-Level Transformations Recap



Mixed CPU-VTA Exec

dense fused ops execute on VTA

f_conv2d [3x3] s=2

f_conv2d [1x1] s=2

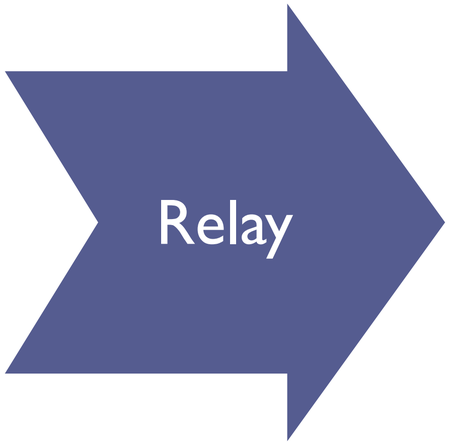
lower intensity ops execute on CPU

add

Compilation Stages

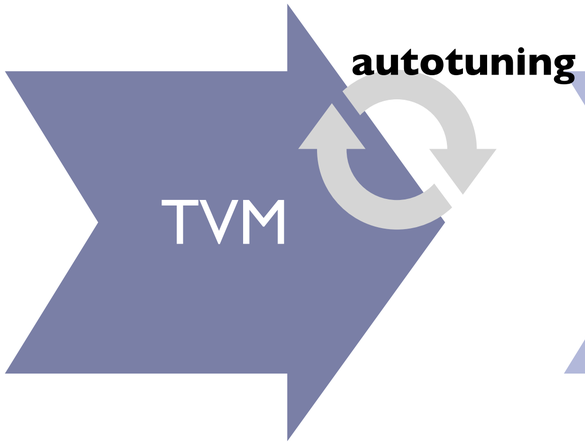
Model from Gluon Zoo

1. Graph Compilation



quantization
re-writing
fusion
partitioning

2. Operator Compilation

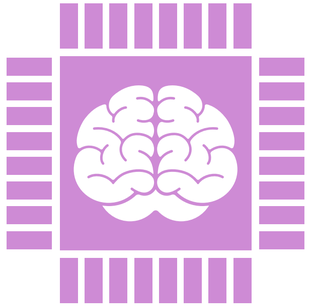


tiling
virtual threads
tensorization
lowering

JIT Compilation

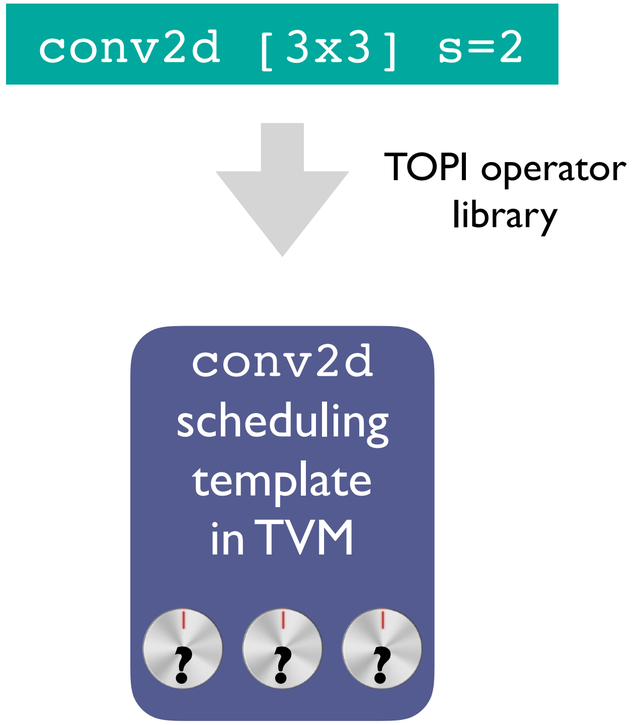
code generation to VTA ISA
instruction management

Offload to VTA



Tensor Operator Library

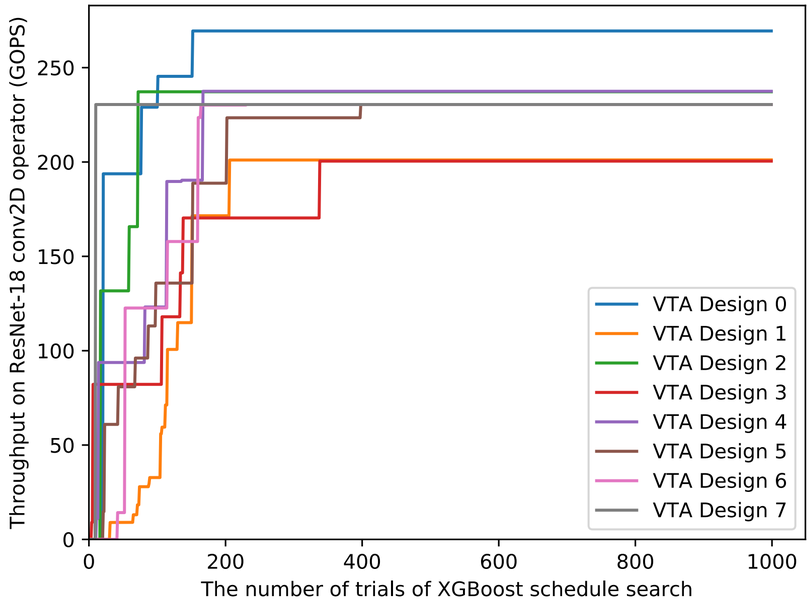
- Now that we have transformed the graph to be more VTA-friendly, we need to generate the corresponding operator libraries.

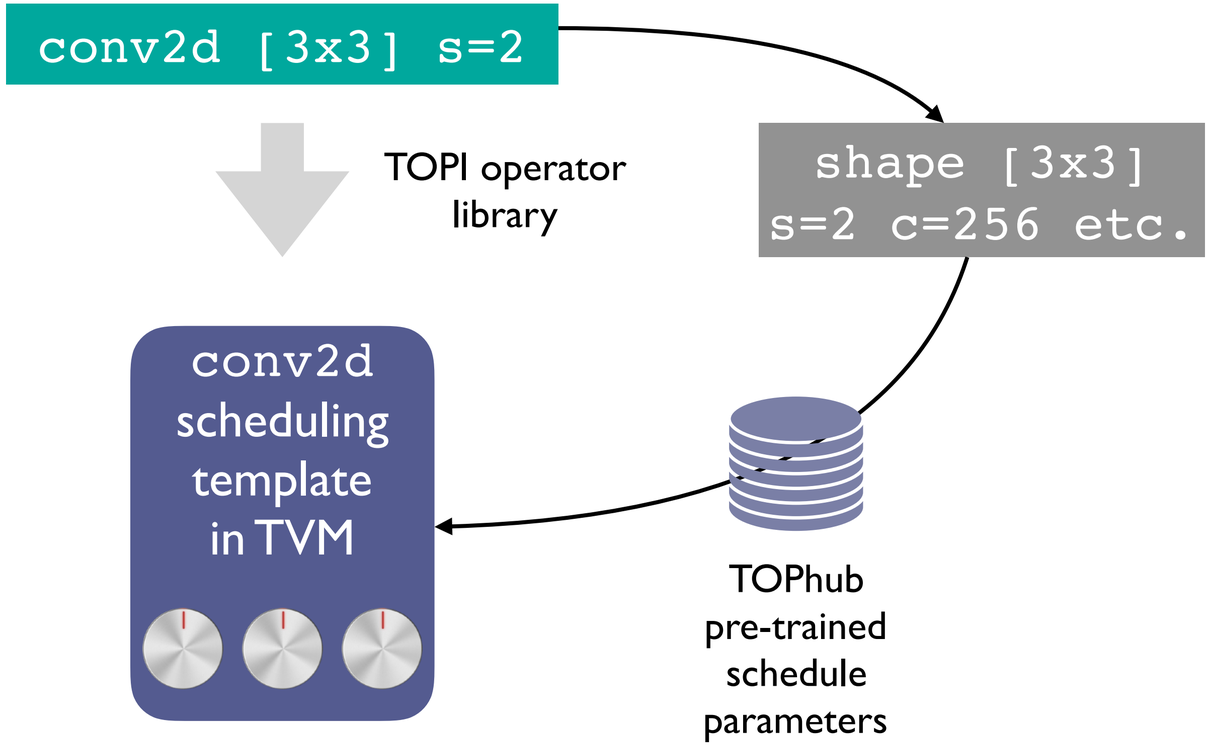


Tensor Operator Library

- Now that we have transformed the graph to be more VTA-friendly, we need to generate the corresponding operator libraries.

Schedule search to populate TOPhub database for the same operator running on different VTA designs

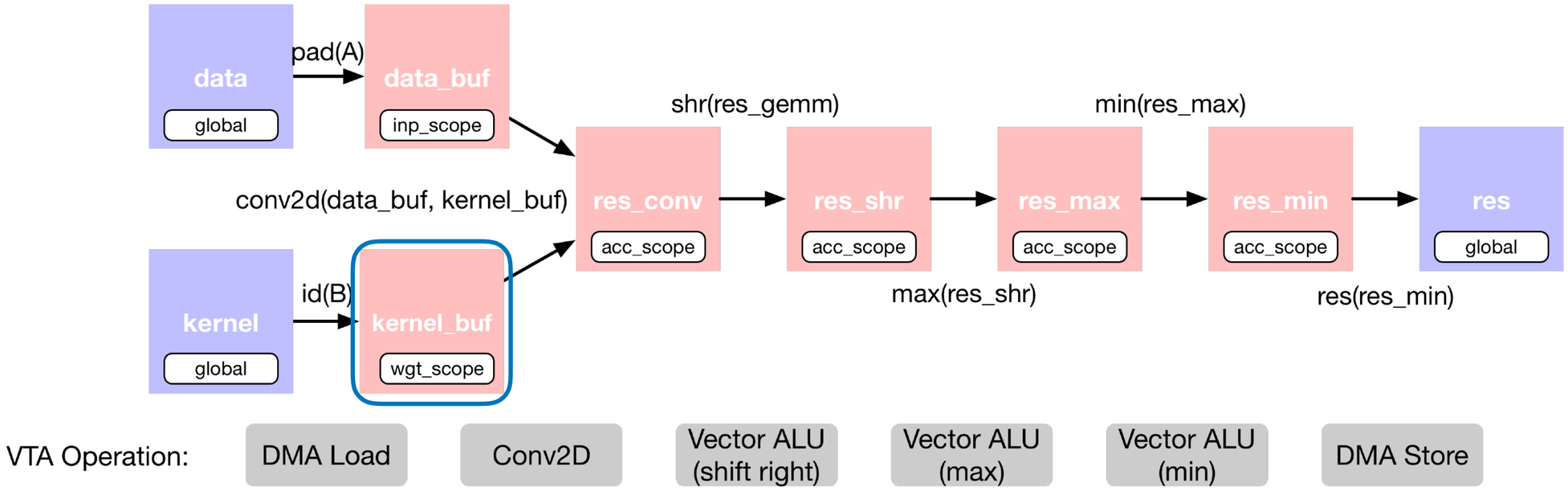




Tensor Expression Template: Staging

- Step 1: Describe computation stages that can be lowered to VTA high-level tasks, and where intermediate data can be assigned to specific SRAM memories

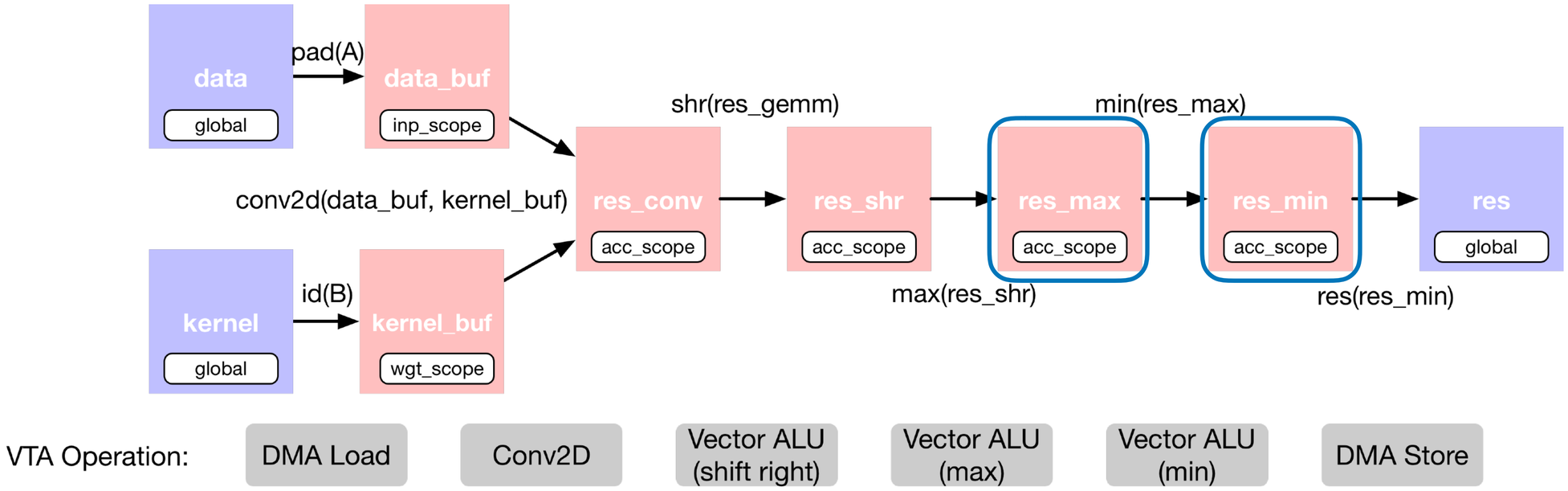
Tensor Expression Template: Staging



We define a kernel buffer with the `cache_read()` schedule primitive

```
kernel_buf = s.cache_read(kernel, env.wgt_scope, ...)
```

Tensor Expression Template: Staging

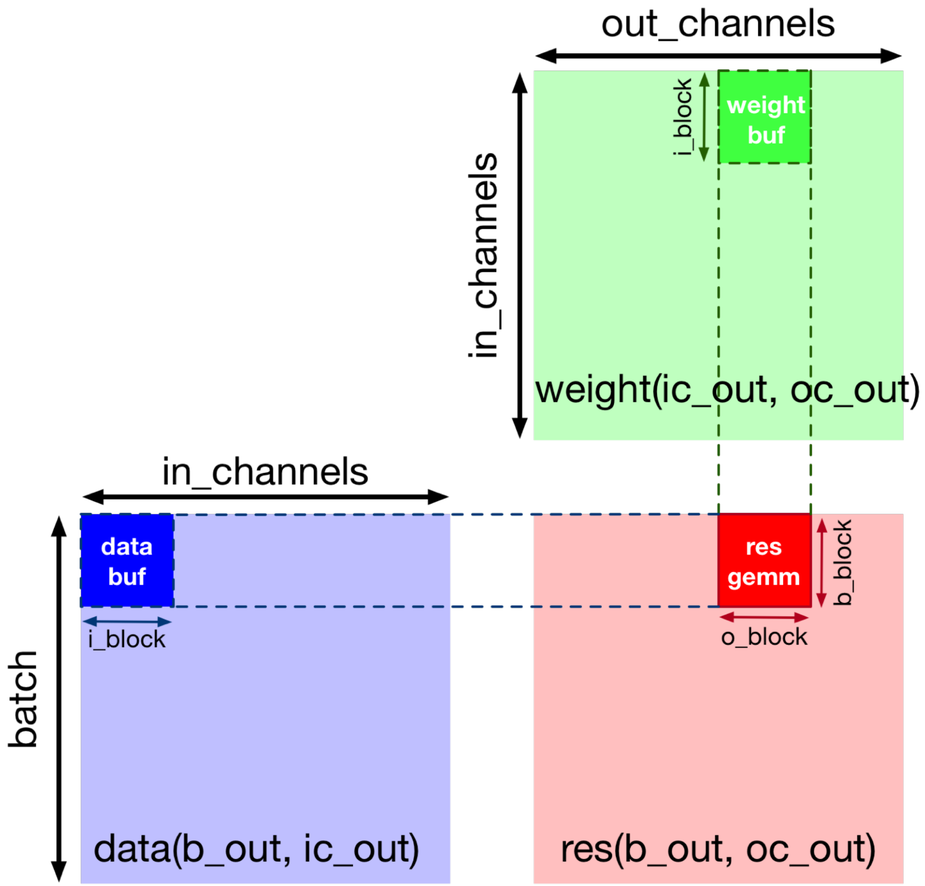


We define the computation stages with `tvm.compute()`, e.g. clip

```
res_max = tvm.compute(output_shape,
                      lambda *i: tvm.max(res_shr(*i), 0),
                      "res_max")
res_min = tvm.compute(output_shape,
                      lambda *i: tvm.min(res_max(*i), 127),
                      "res_min")
```

Tensor Expression Template: Caching

- Step 2: Tile loops to optimize reuse of SRAM (matmult example for simplicity)




```
# Tile loops
b, oc, _, _ = s[res].op.axis
b_out, oc_out, b_inn, oc_inn = s[res].tile(b, oc,
                                           b_block, oc_block)

# Move computation for each stage in the tile
s[res_gemm].compute_at(s[res], oc_out)
...
s[res_min].compute_at(s[res], oc_out)
```

Tensor Expression Template: Lowering

- Step 3: Map load and stores to DMA operations, with dma_copy pragma()

```
# Use DMA copy pragma on DRAM->SRAM operations
s[data_buf].pragma(s[data_buf].op.axis[0], dma_copy)
s[weight_buf].pragma(s[weight_buf].op.axis[0], dma_copy)
```

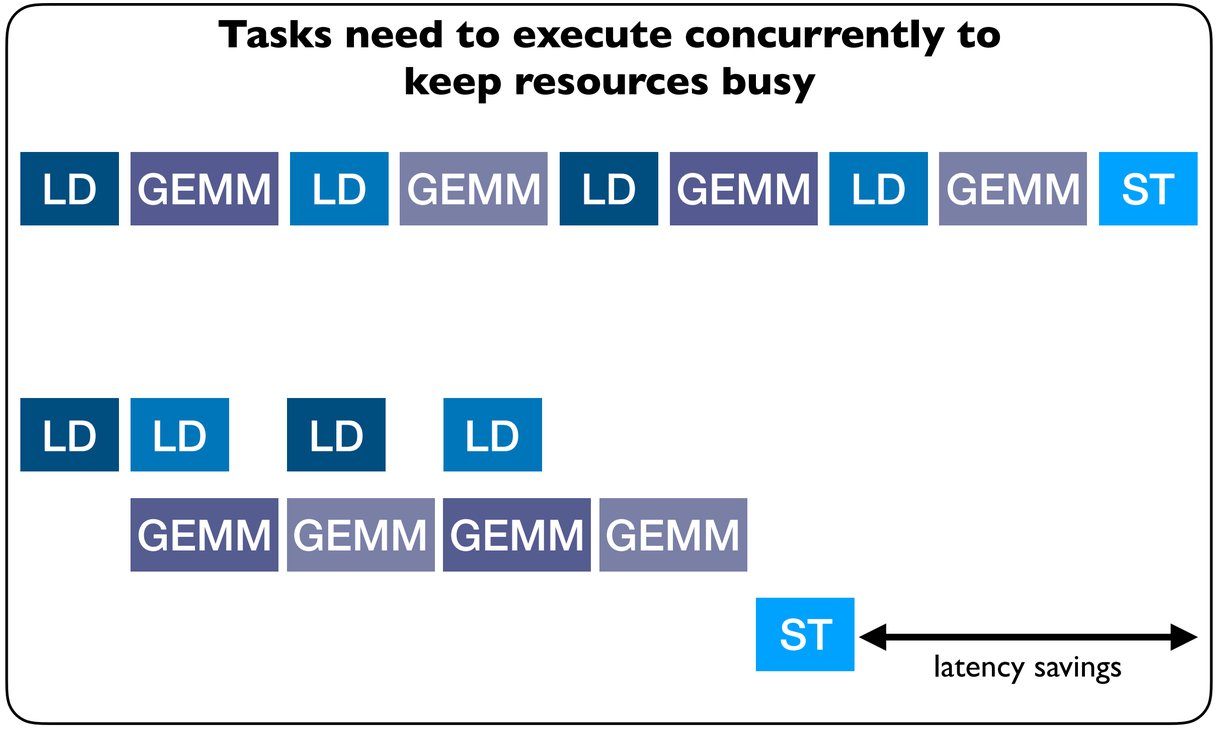
- Step 4: Apply tensorization on schedule to map to GEMM low-level ops

```
# Re-order GEMM computation inner loop to match tensorization constraints
s[res_gemm].reorder(ic_out, b_inn, oc_inn, ic_inn, b_tns, oc_tns, ic_tns)

# Apply tensorization over the batch tensor tile axis
s[res_gemm].tensorize(b_tns, gemm)
```

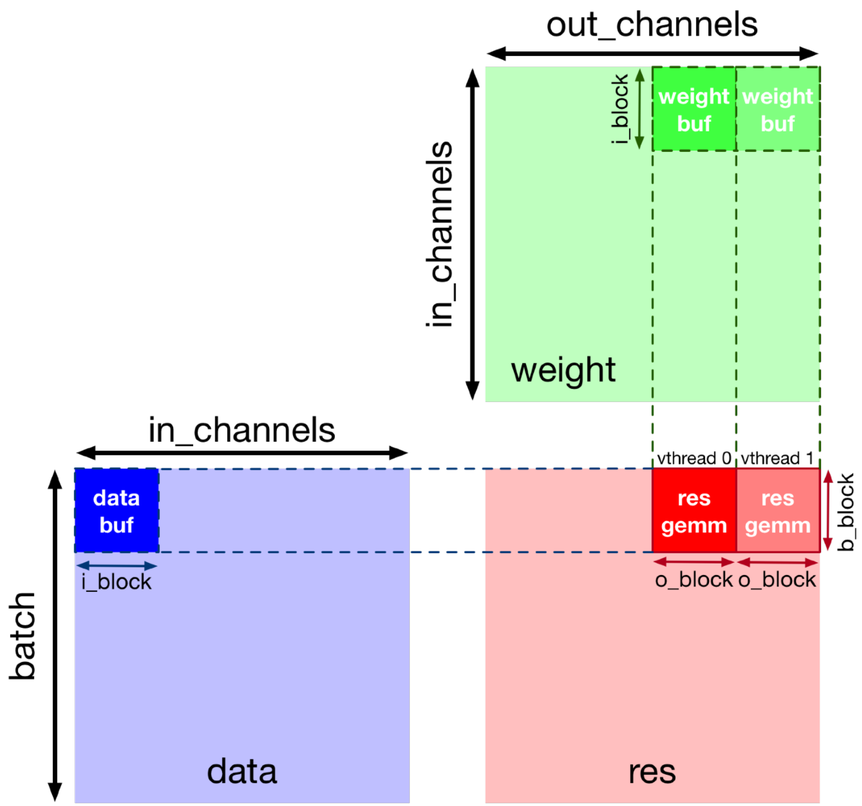
Tensor Expression Template: Virtual Threads

- Step 5: virtual threads allow us to take advantage of architecture-defined task-level pipeline parallelism.



Tensor Expression Template: Virtual Threads

- Step 5: virtual threads allow us to take advantage of architecture-defined task-level pipeline parallelism using the programmer friendly construct of threads.




```
# VTA only needs 2 virtual threads
v_threads = 2

# Perform split along outer axis
_, tx = s[res].split(oc_out, factor=v_threads)
s[res].bind(tx, tvm.thread_axis("cthread"))
```

Tensor Expression Language: Lowering to Runtime API

Tensor Expression
Compute Declaration & Schedule

build

VTA specific
lowering IR passes

```
vta.coproc_dep_pop(2, 1)
produce A_buf {
  VTALoadBuffer2D(tvm_thread_context(VTATLSCommandHandle()), A, ko, …)
}
produce B_buf {
  VTALoadBuffer2D(tvm_thread_context(VTATLSCommandHandle()), B, ko, …)
}
vta.coproc_dep_push(1, 2)
// attr [iter_var(vta, , vta)] coproc_scope = 2
vta.coproc_dep_pop(1, 2)
// attr [iter_var(vta, , vta)] coproc_uop_scope = "VTAPushGEMMOp"
VTAUopLoopBegin(16, 1, 0, 1)
VTAUopPush(0, 0, 0, 0, 0, 0, 0, 0)
VTAUopLoopEnd()
vta.coproc_dep_push(2, 1)
```

DMALoad()

DMALoad()

wait on DMA

MatMul()

…

Lowered code that calls into the VTA Runtime API

Try our interactive tutorial!

https://sampl.cs.washington.edu/tvmfcrc/

or do an internet search with “TVM FCRC”

VTA Overview

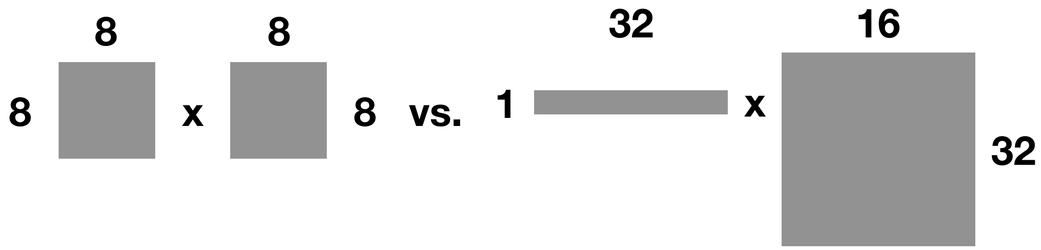
Hardware Architecture Deep Dive

Programmability Challenges

Hardware-Software Co-Design

VTA: General DL Architecture

Tensor Intrinsic



Hardware Datatype

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

Memory Subsystem

Operation Support

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

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

IEEE Micro S.I. 2019

Schedule Exploration with VTA

VTA Candidate Designs

#1 Design AAA @ 307GOPs

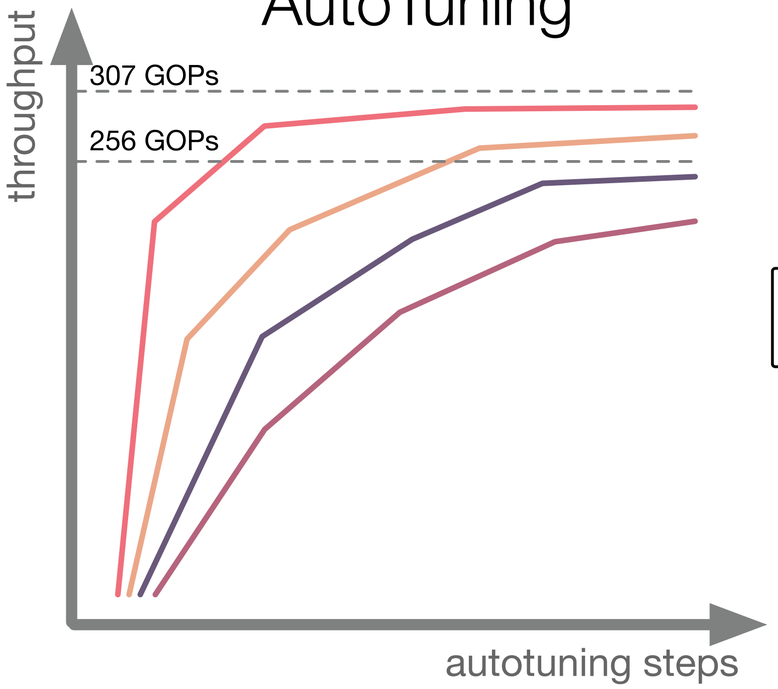
#2 Design BBB @ 307GOPs

#3 Design CCC @ 307GOPs

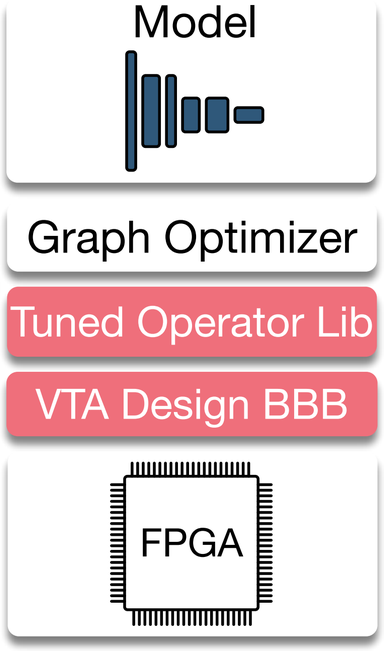
#4 Design DDD @ 256GOPs

Needs to pass place & route and pass timing closure

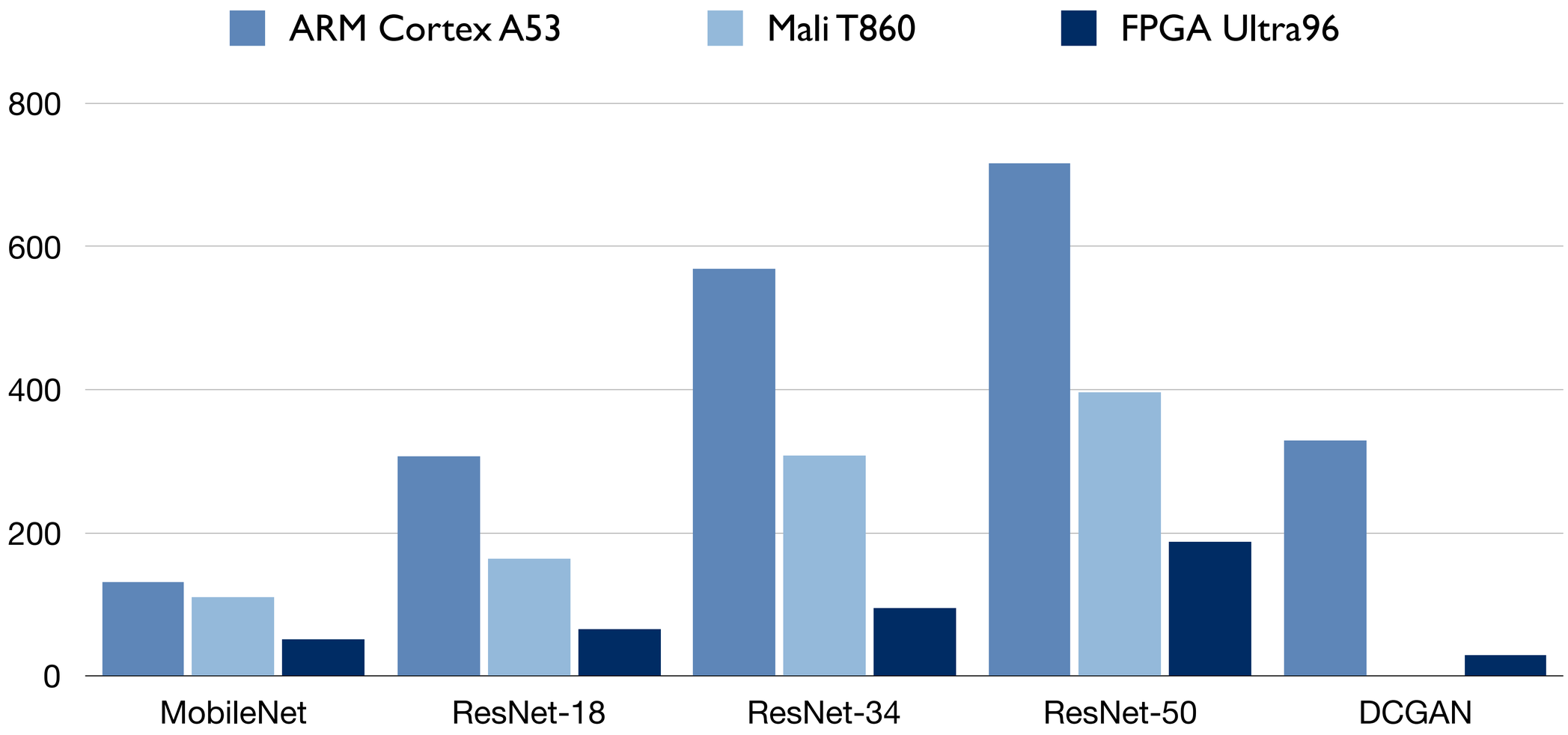
Operator Performance AutoTuning



Deliverable



End-to-end Performance



A Hardware-Software Blueprint for Flexible Deep Learning Specialization

HUAWEI ORACLE amazon intel XILINX ada

CRISP Center for Research in Intelligent Storage and Processing in Memory SRC Semiconductor Research Corporation NSF DARPA Qualcomm

Faculty


Luis Ceze
Professor


Carlos Guestrin
Professor


Arvind Krishnamurthy
Professor


Matthai Philipose
Affiliate Professor


Zachary Tatlock
Assistant Professor

Researchers


Thierry Moreau

Graduate Students


Tianqi Chen


Meghan Cowan


Josh Fromm


Ziheng Jiang


Liang Luo


Steven Lyubomirsky


Pratyush Patel


Jared Roesch


Gus Smith


Luis Vega


Logan Weber


Eddie Yan

sampl

PAUL G. ALLEN SCHOOL OF COMPUTER SCIENCE & ENGINEERING

tvm.ai