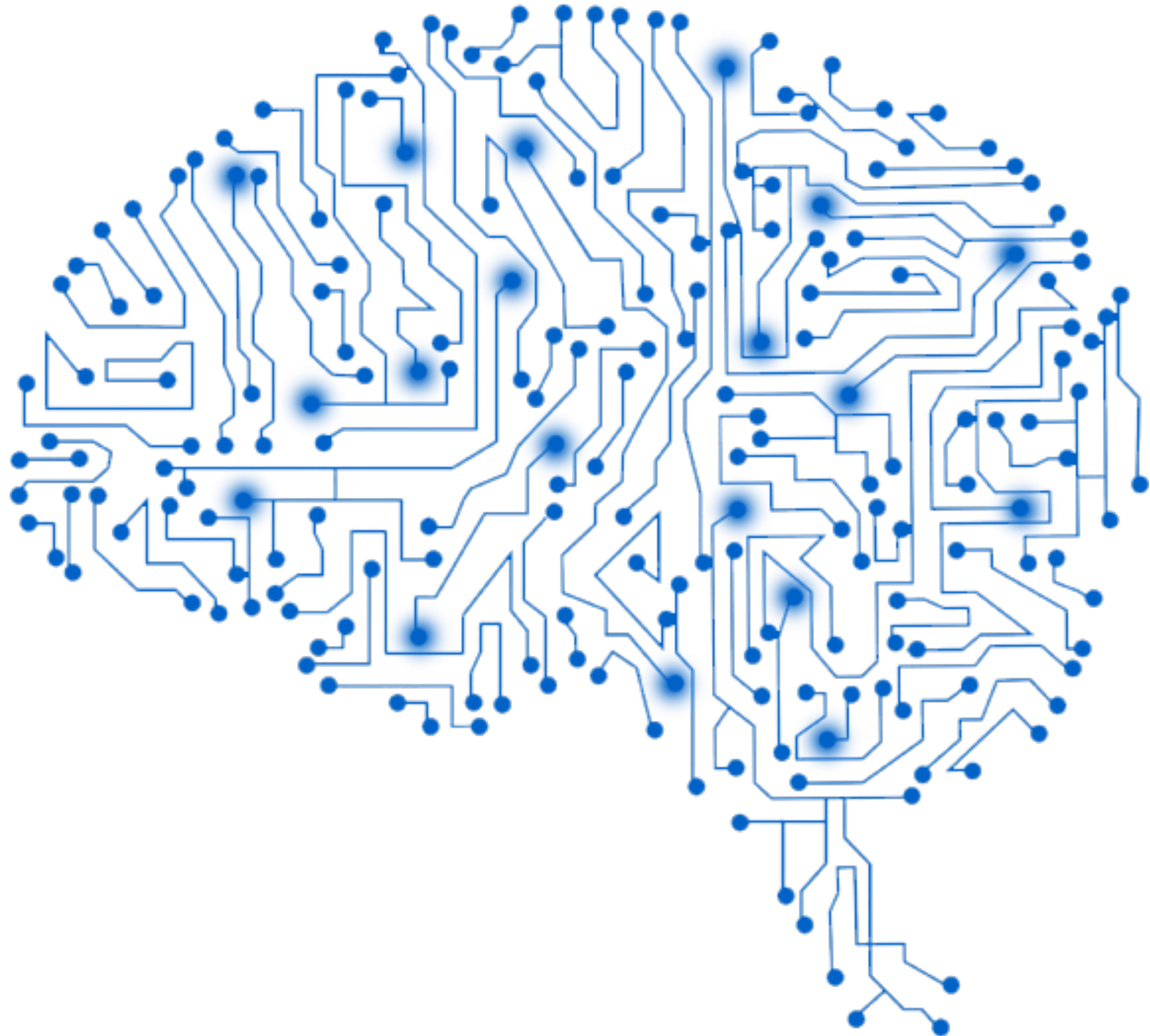
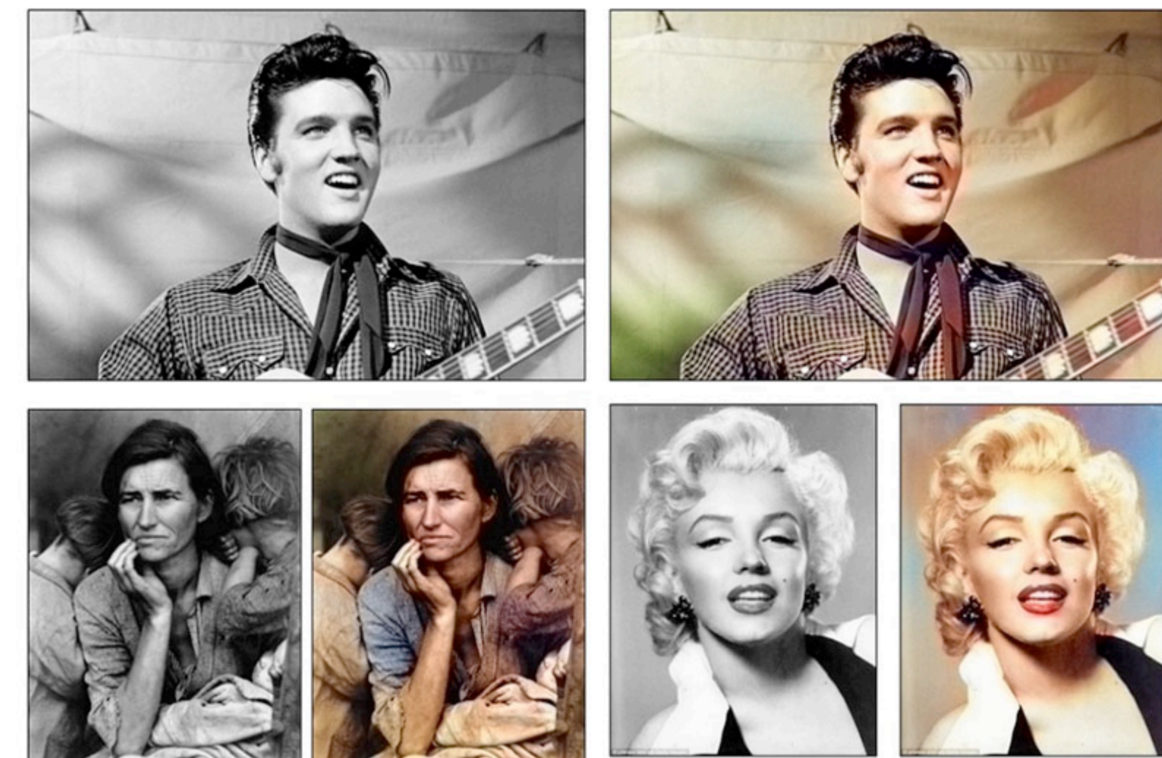
Hardware Specialization in Deep Learning

CSE590W 18Sp, Thursday April 19th 2018

Thierry Moreau

Deep Learning Explosion

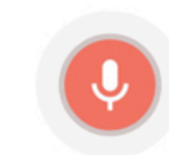




Cortana.

Siri

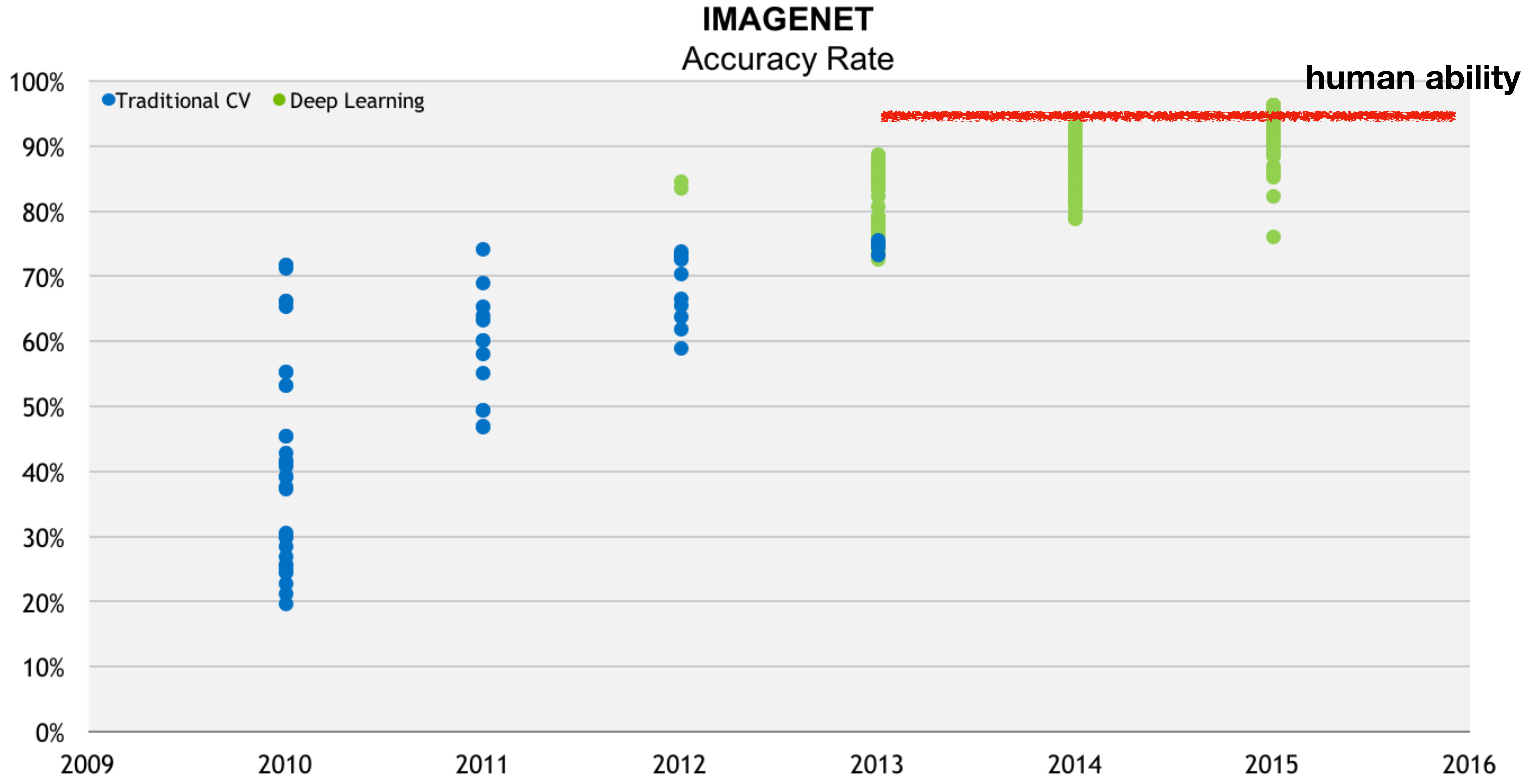
amazon echo



Google now

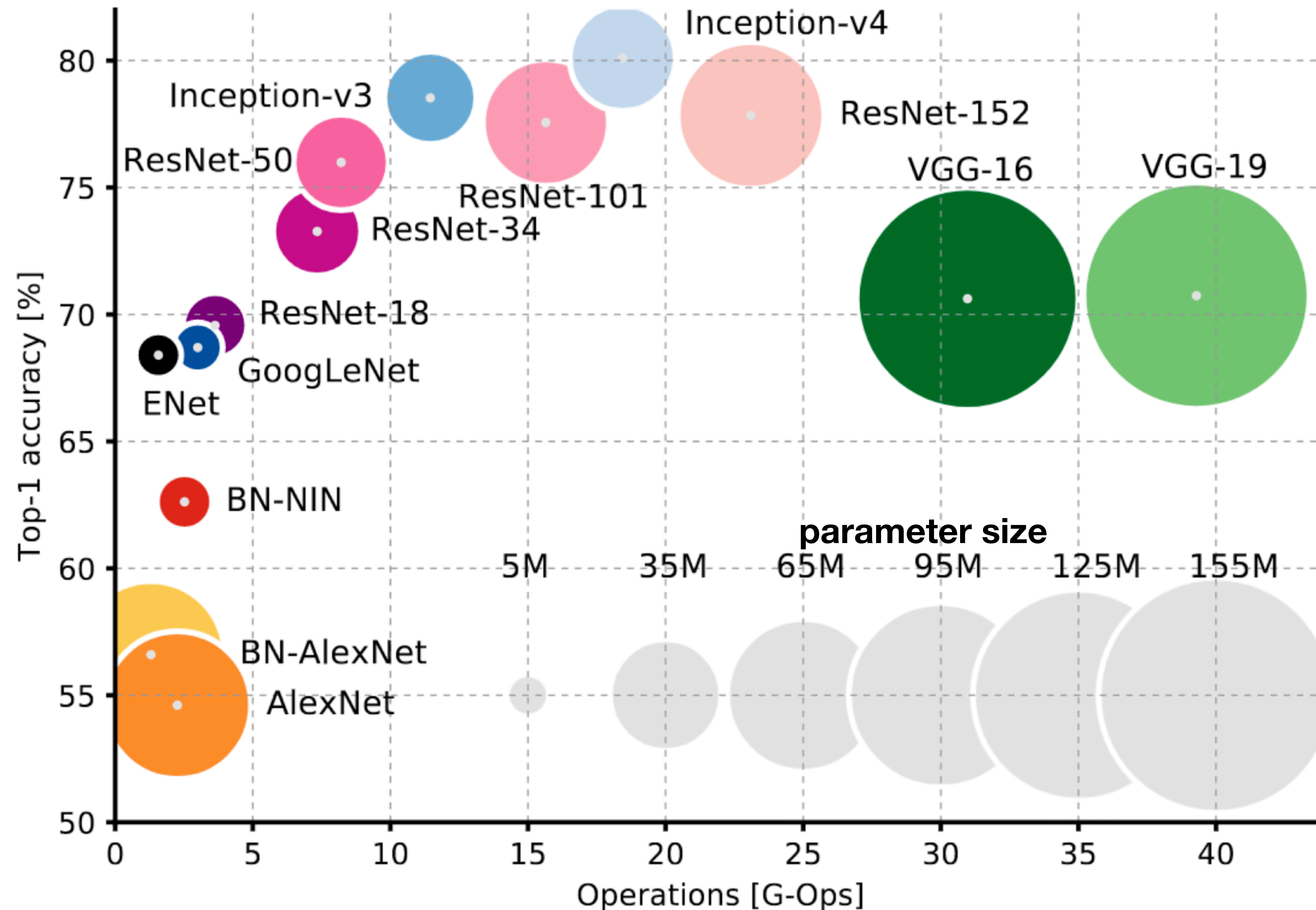
Facebook M

Deep Learning Revolutionizing Computer Vision



Source: NVIDIA blogpost, June 2016

Compute Requirements is Steadily Growing

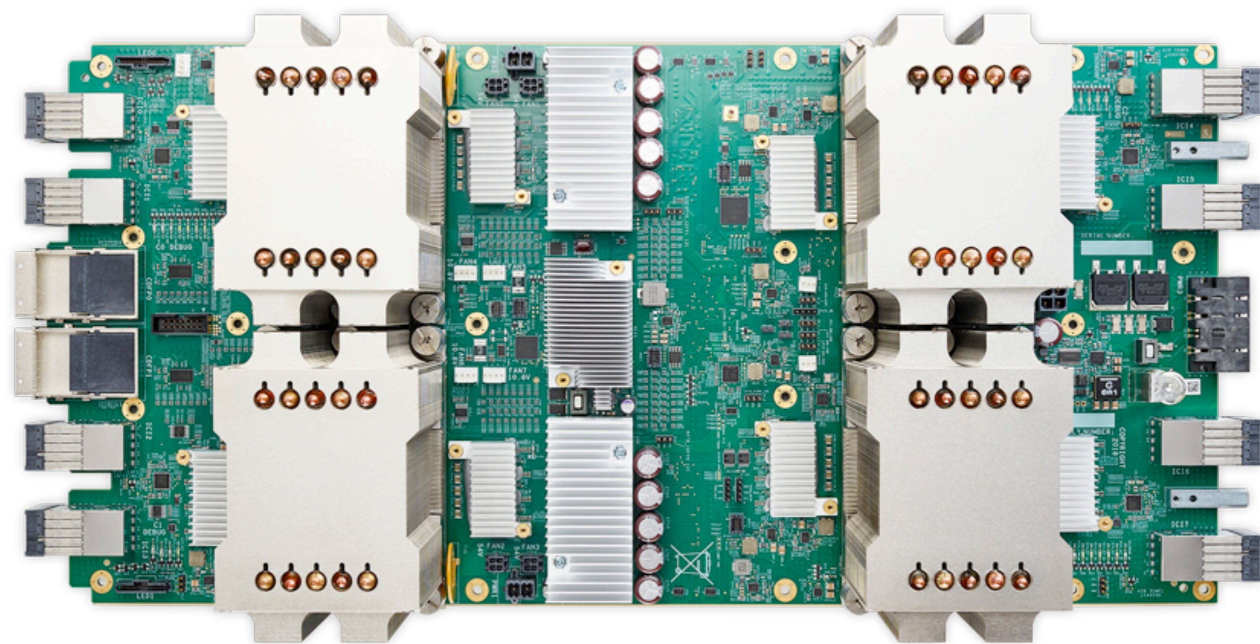


Source: Eugenio Culurciello, An Analysis of Deep Neural Network Models for Practical Applications, arXiv:1605.07678

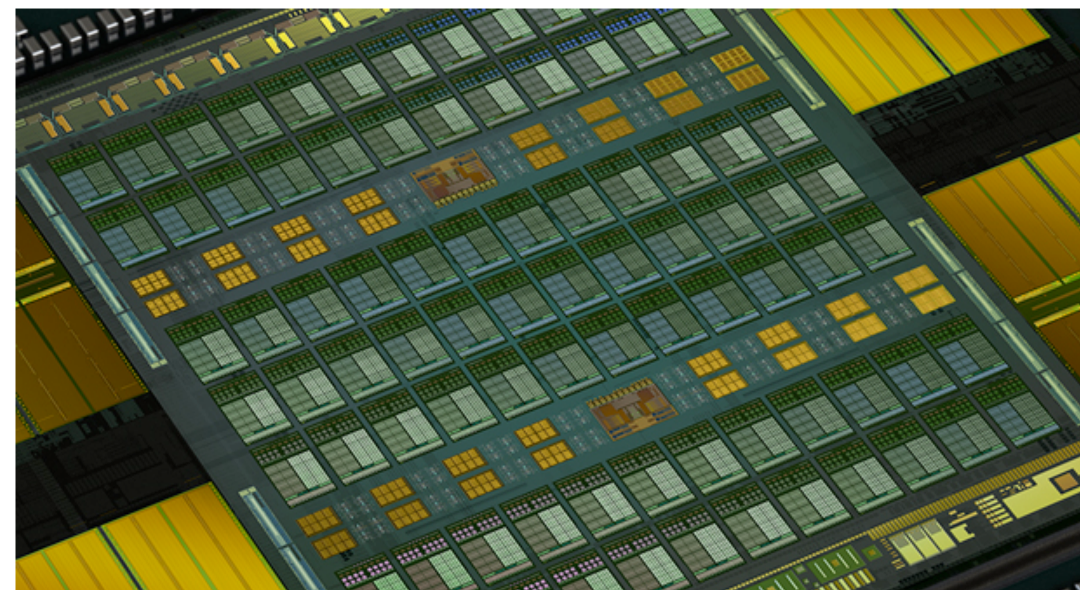
Hardware Specialization

- Idea: tailor your chip architecture to the characteristics of a stable workload

Google Cloud TPU: 180 Tflops



NVIDIA Volta: 100 Tflops

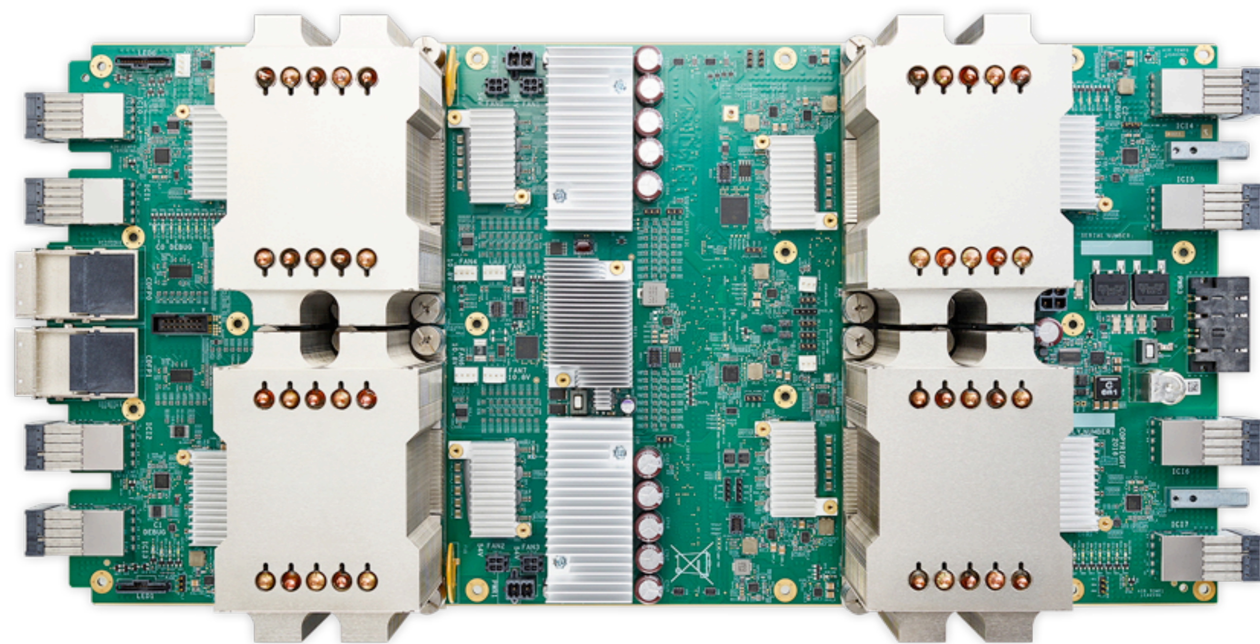


Apple Bionic A11: 0.6 Tflops

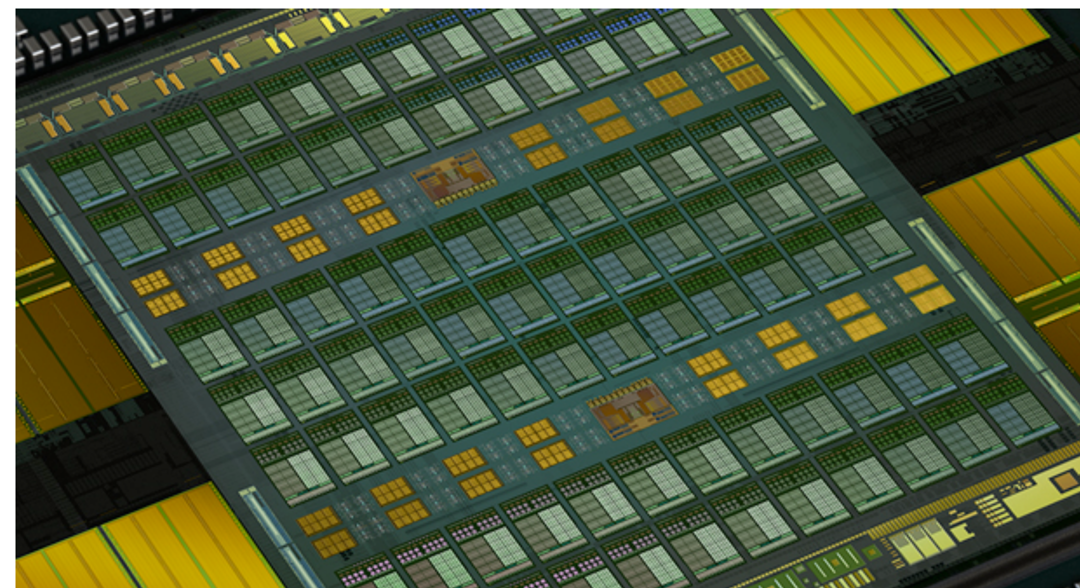
Hardware Specialization

- Idea: tailor your chip architecture to the characteristics of a ***stable*** workload

Google Cloud TPU: 180 Tflops



NVIDIA Volta: 100 Tflops



Apple Bionic A11: 0.6 Tflops

Data center **Edge**

Evolution of Deep Learning

Matrix Multiplication: `fp32 (A, B) × (B, C)`

Optimization	New problem specification	Reference
quantization	**int4** `(A, B)` × **bool** `(B, C)`	Binary Connect, NIPS 15
knowledge distillation	**fp32** `(a, b)` × `(b, c)`	Fitnets, ICLR 2015
compression, pruning	**sparse int16** `(A, B)` × `(B, C)`	Deep Compression, ICLR 2016
tensor decomposition	**fp32** `(A,r)` x `(r,B)` × `(B,C)`	Compression of deep convolutional neural networks, ICLR 2016

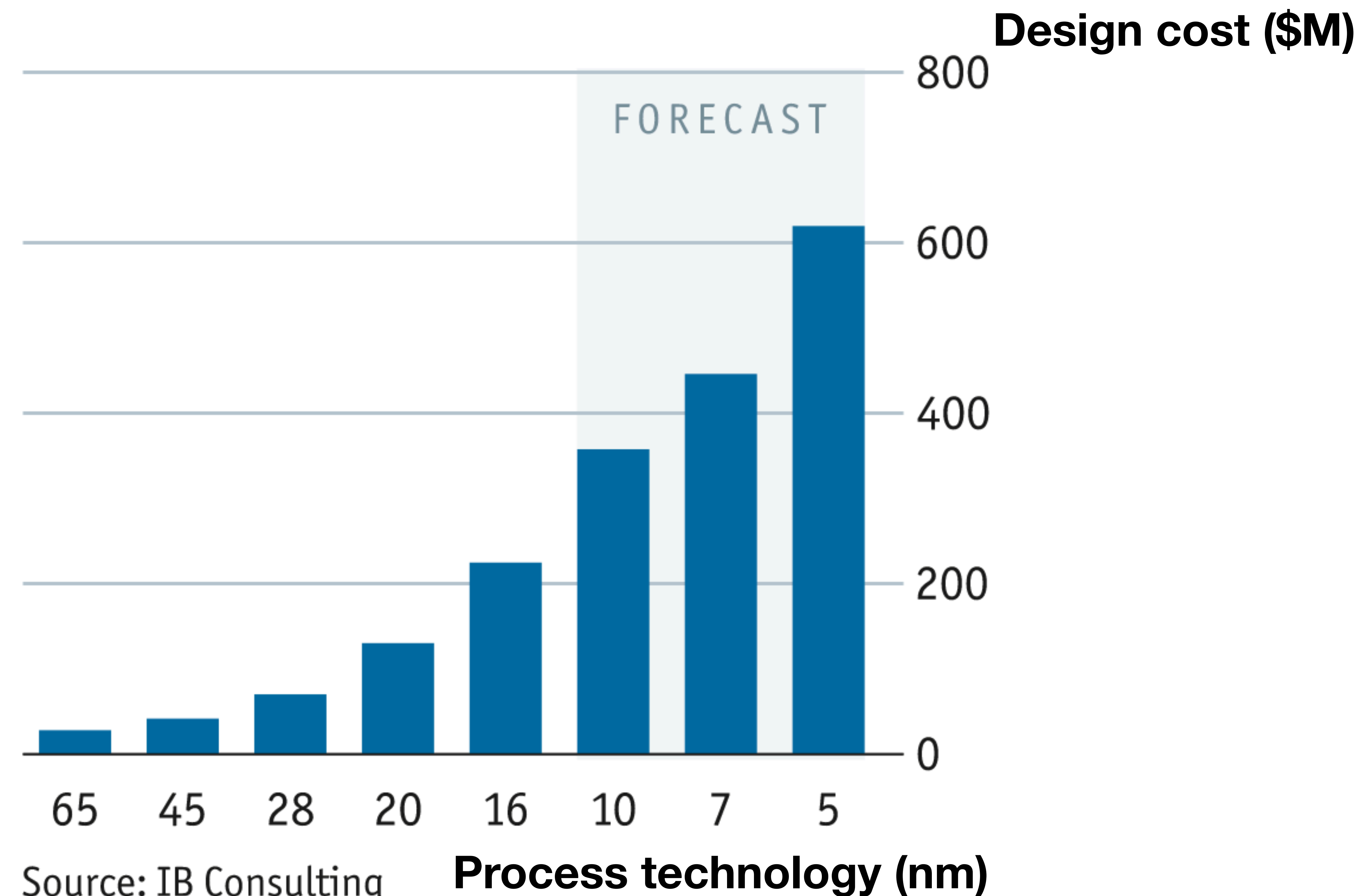
2D Convolution : `fp32 (H, W, Ci)` ⊗ `(K, K, Co, Ci)`

winograd	**fp32 FFT^{-1}(FFT(**`(H, W, Ci)`**)** · **FFT(**`(K, K, Co, Ci)`**)**	Fast Convolutional Nets with fbfft, arXiv:1412.7580 2014
depth wise convolution	fp32 `(H, W, Ci)` ⊗ `(K, K, 1, Ci)` ⊗ `(1, 1, Co, Ci)`	MobileNets, arXiv:1704.04861 2017

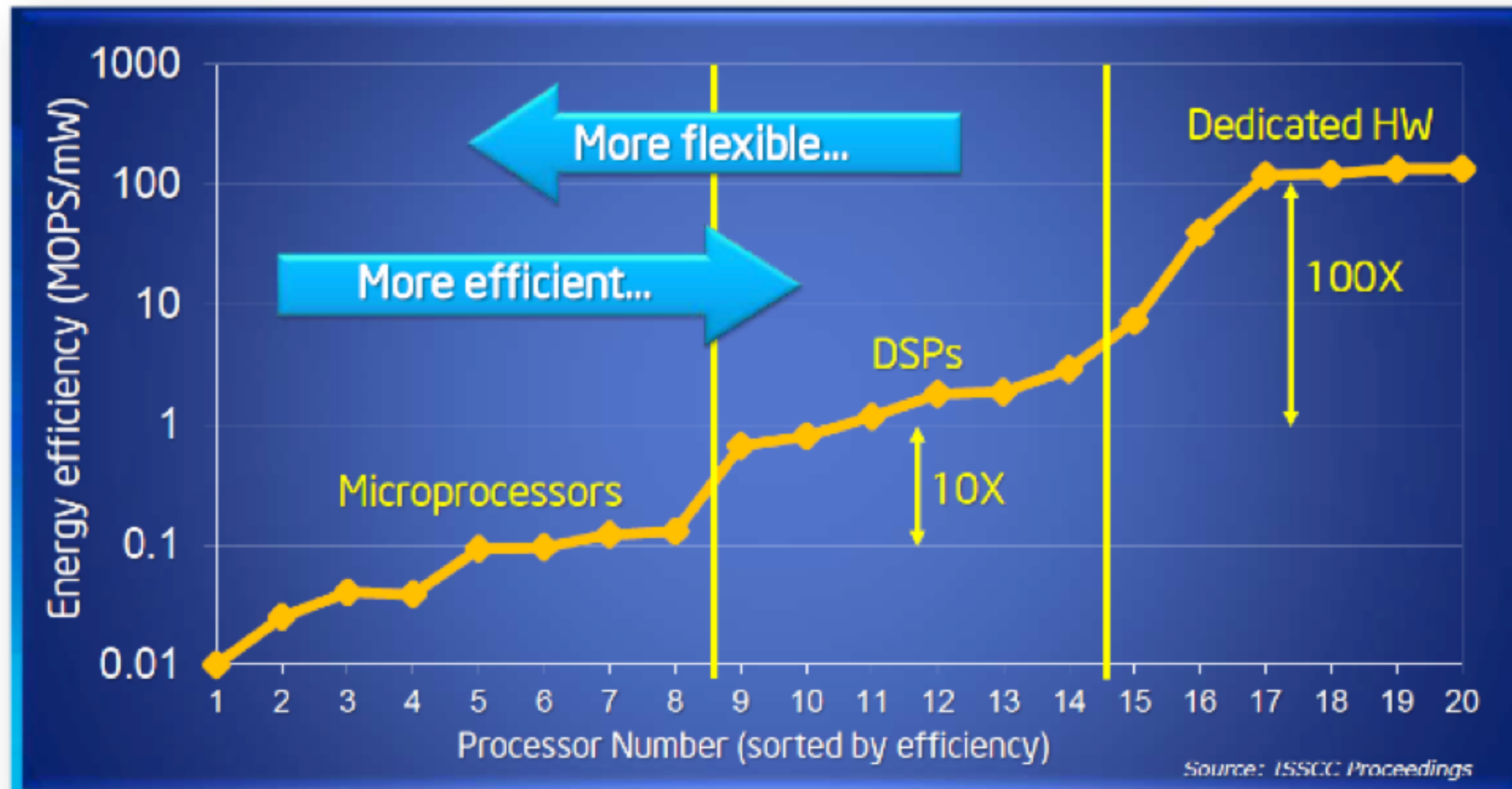
Specialization Challenge

Tape-out costs for ASICs is exorbitant
10x cost gap between 16nm and 65nm

Risky bet to design hardware accelerators for ever-changing applications



Flexibility vs. Efficiency Tradeoffs



Source: Bob Broderson, Berkeley Wireless group

Discussion Break

- Does deep learning constitute a stable workload to justify ASIC-based hardware acceleration?

TPU: Google's Entry in the Deep Learning Acceleration Race

Highlights:

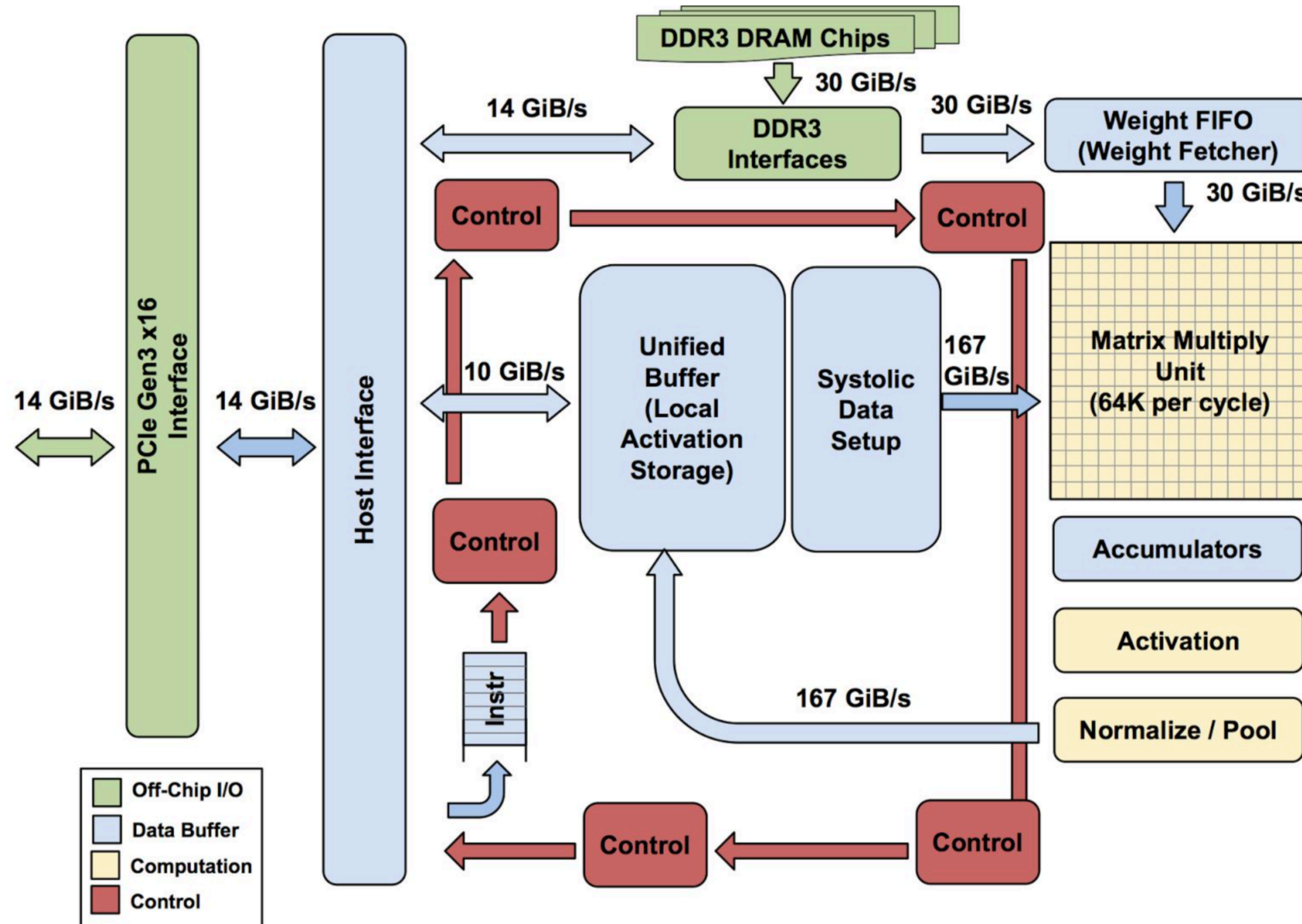
- Custom ASIC deployed in datacenters since 2015
- 65k 8-bit matrix multiply that offers peak throughput of 92 TOPS
- Targets mainstream NN applications (MLPs, CNNs, and LSTMs)
- Shows 30-80x improved TOPS/Watt over K80

Jouppi et al., In-Datacenter Performance Analysis of a Tensor Processing Unit, ISCA 2017

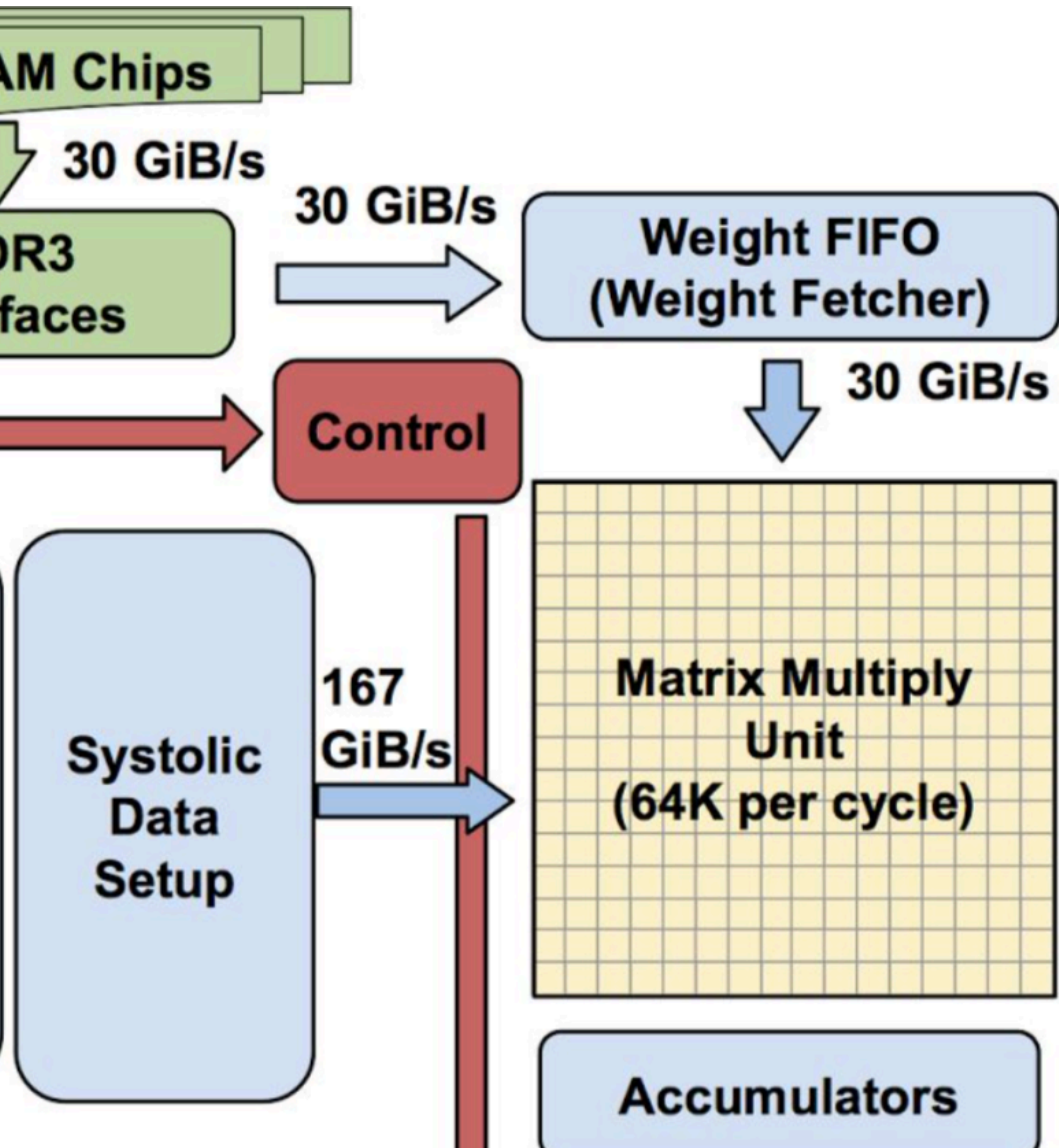
What make TPUs Efficient?

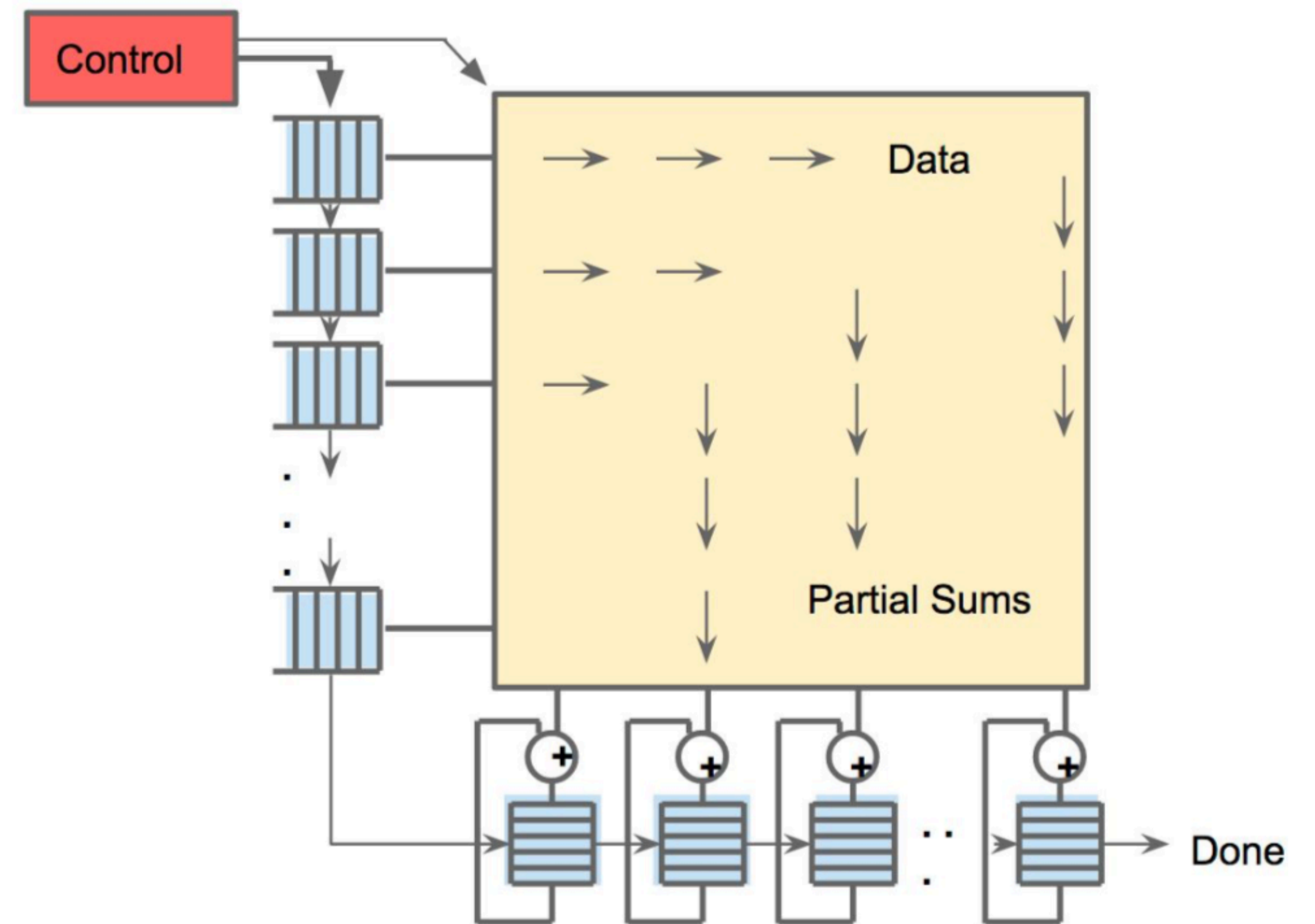
- Integer inference (saves 6-30x energy over 16bit FP)
- Large amount of MACs (25x over K80)
- Large amount of on-chip memory (3.5x over K80)

TPU Block Diagram Overview



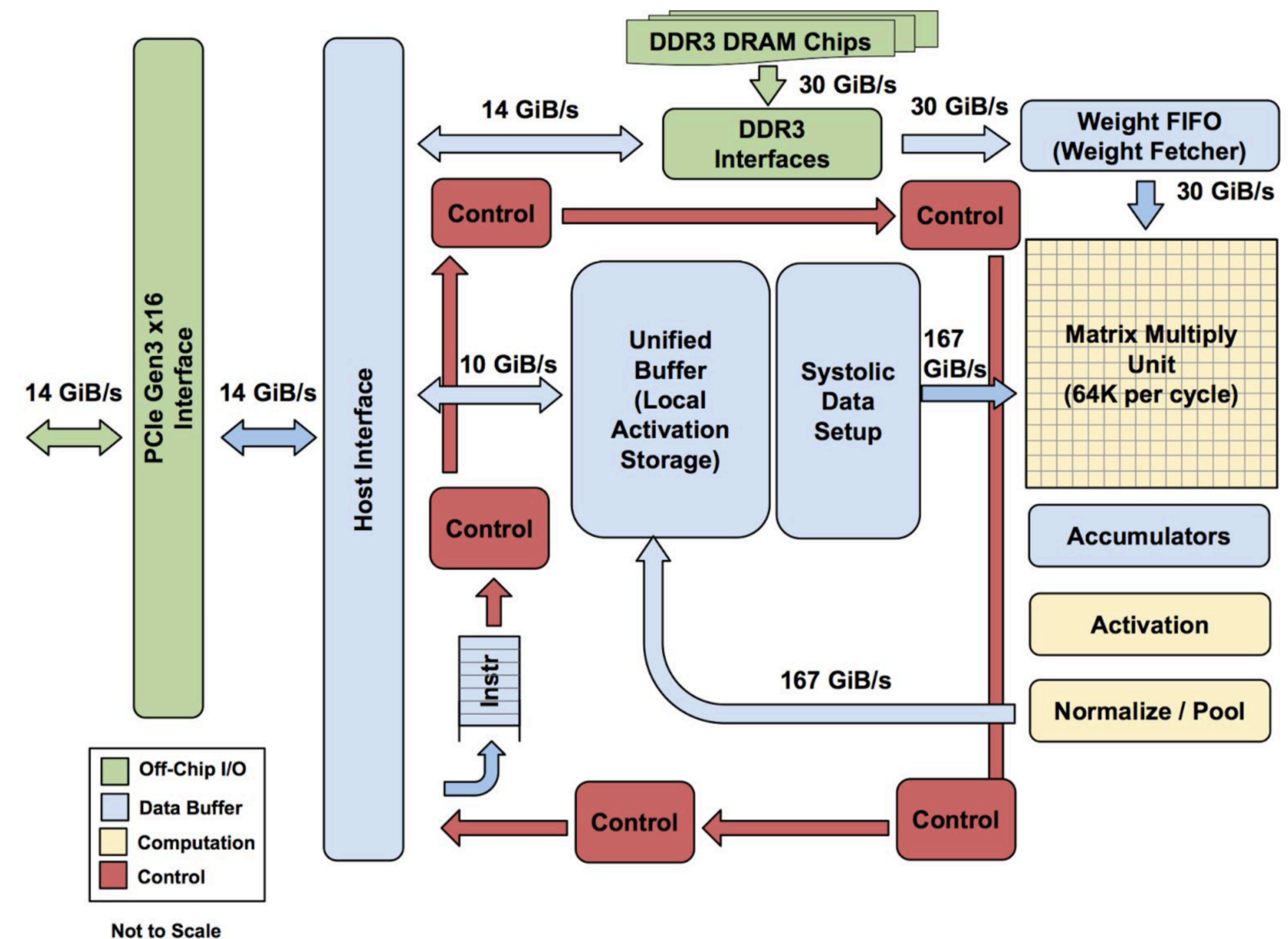
Systolic Data Flow



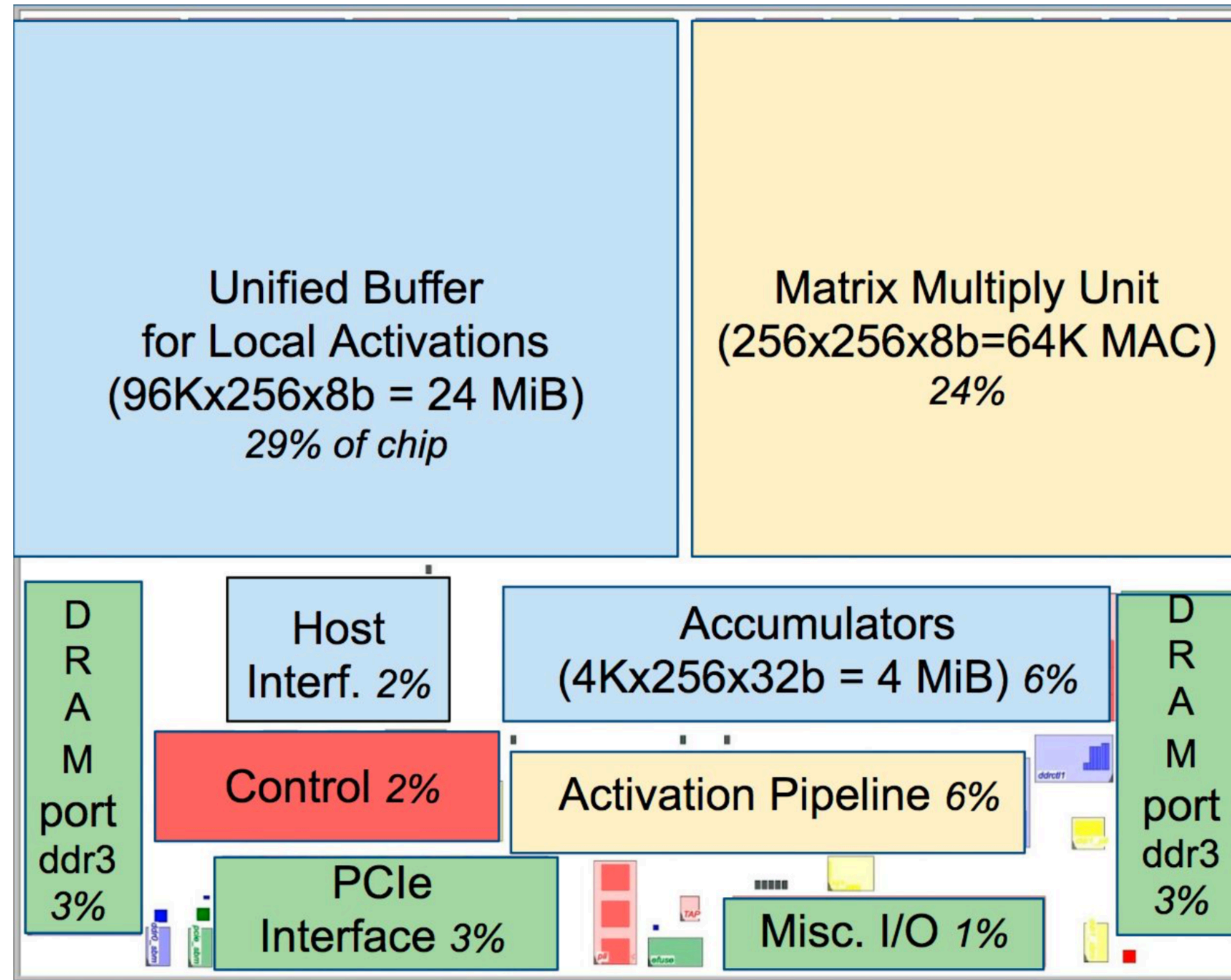


Hardware-Software Interface

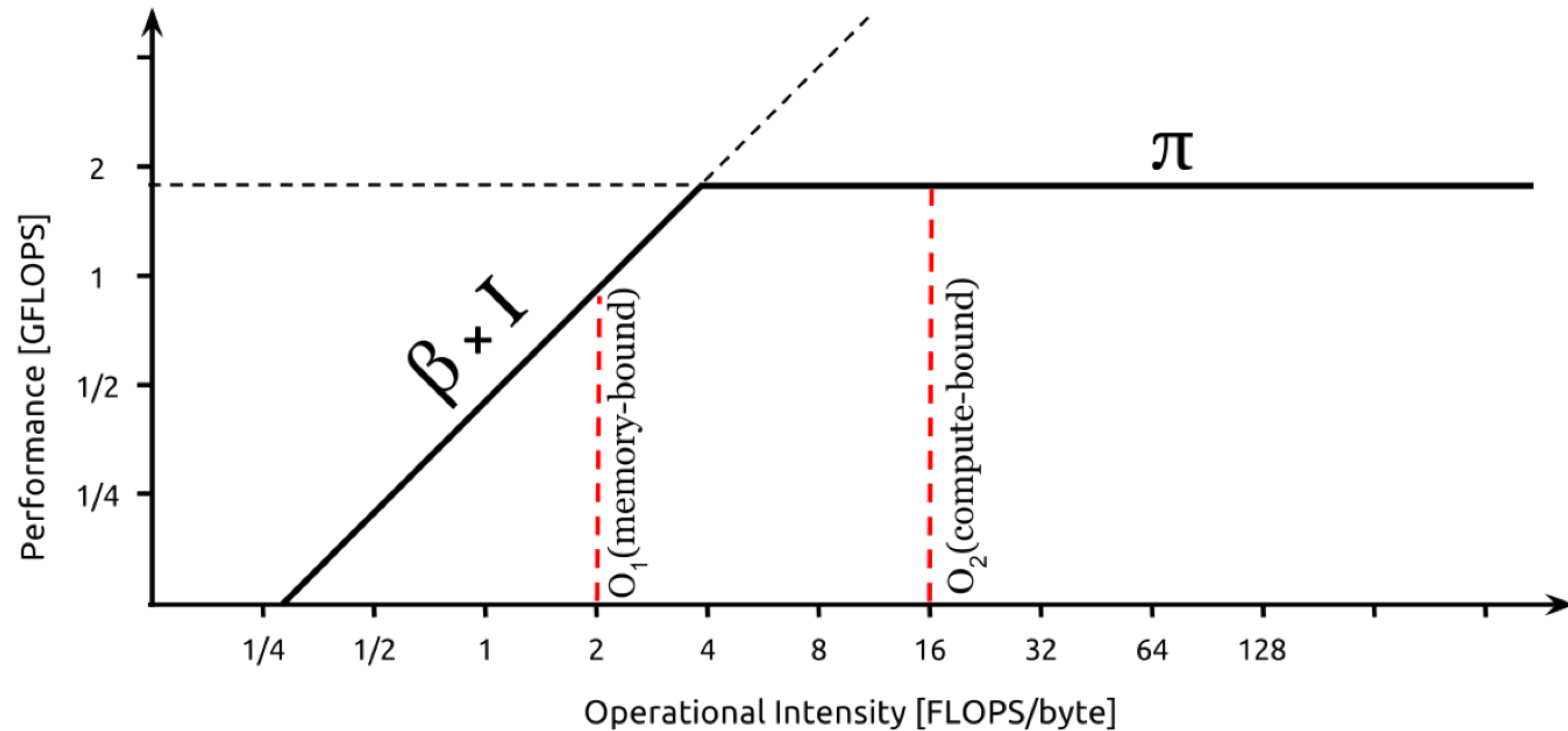
- CISC-like instruction set
  - `Read_Host_Memory`
  - `Read_Weights`
  - `MatrixMultiply/Convolve`
  - `Activate`
  - `Write_Host_Memory`



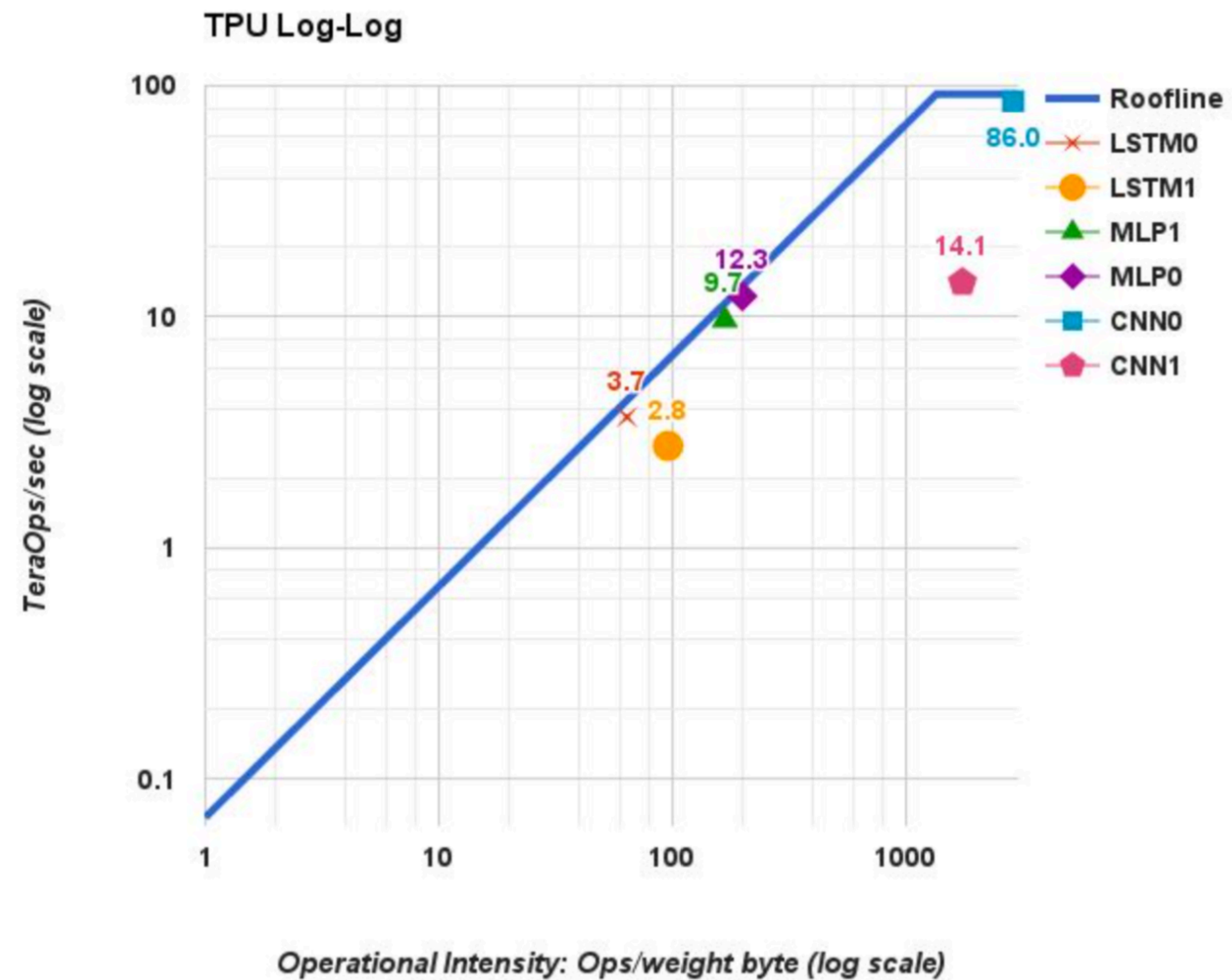
TPU Floor Plan



Roofline Model



TPU Roofline

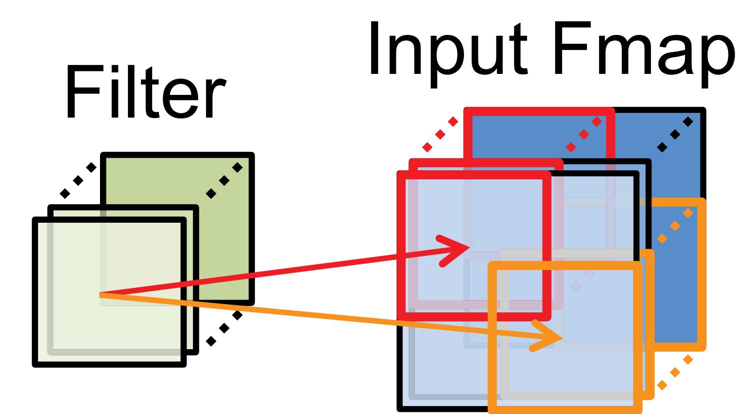


- 1350 Operations per byte of weight memory fetched

Arithmetic Intensity in Convolutional Workloads

Convolutional Reuse

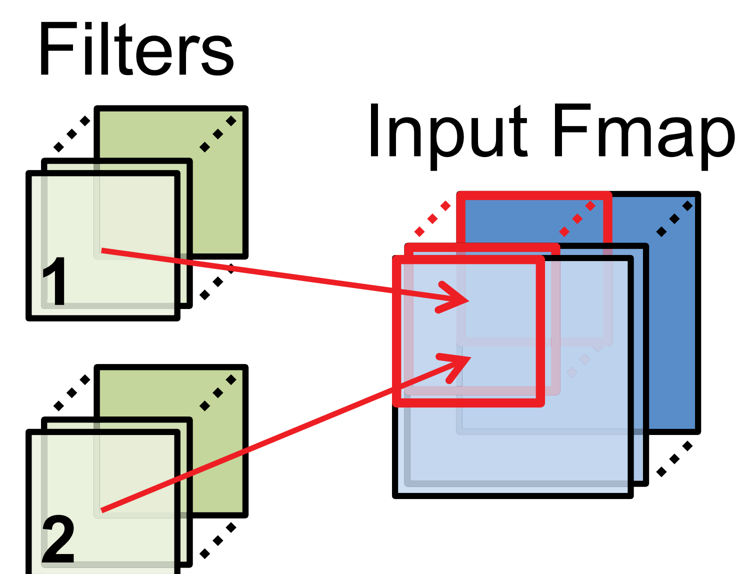
CONV layers only
(sliding window)



Reuse: Activations
Filter weights

Fmap Reuse

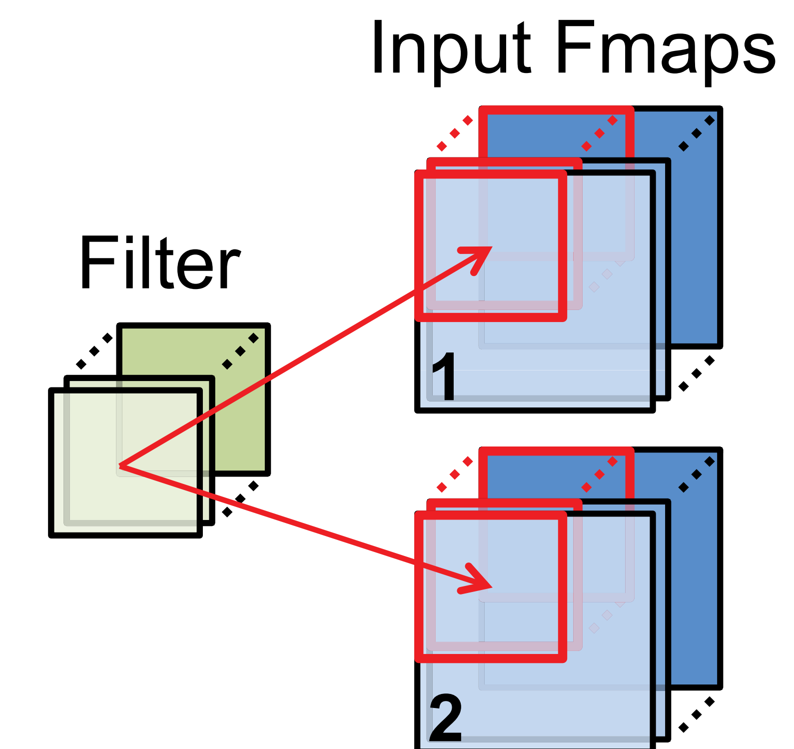
CONV and FC layers



Reuse: Activations

Filter Reuse

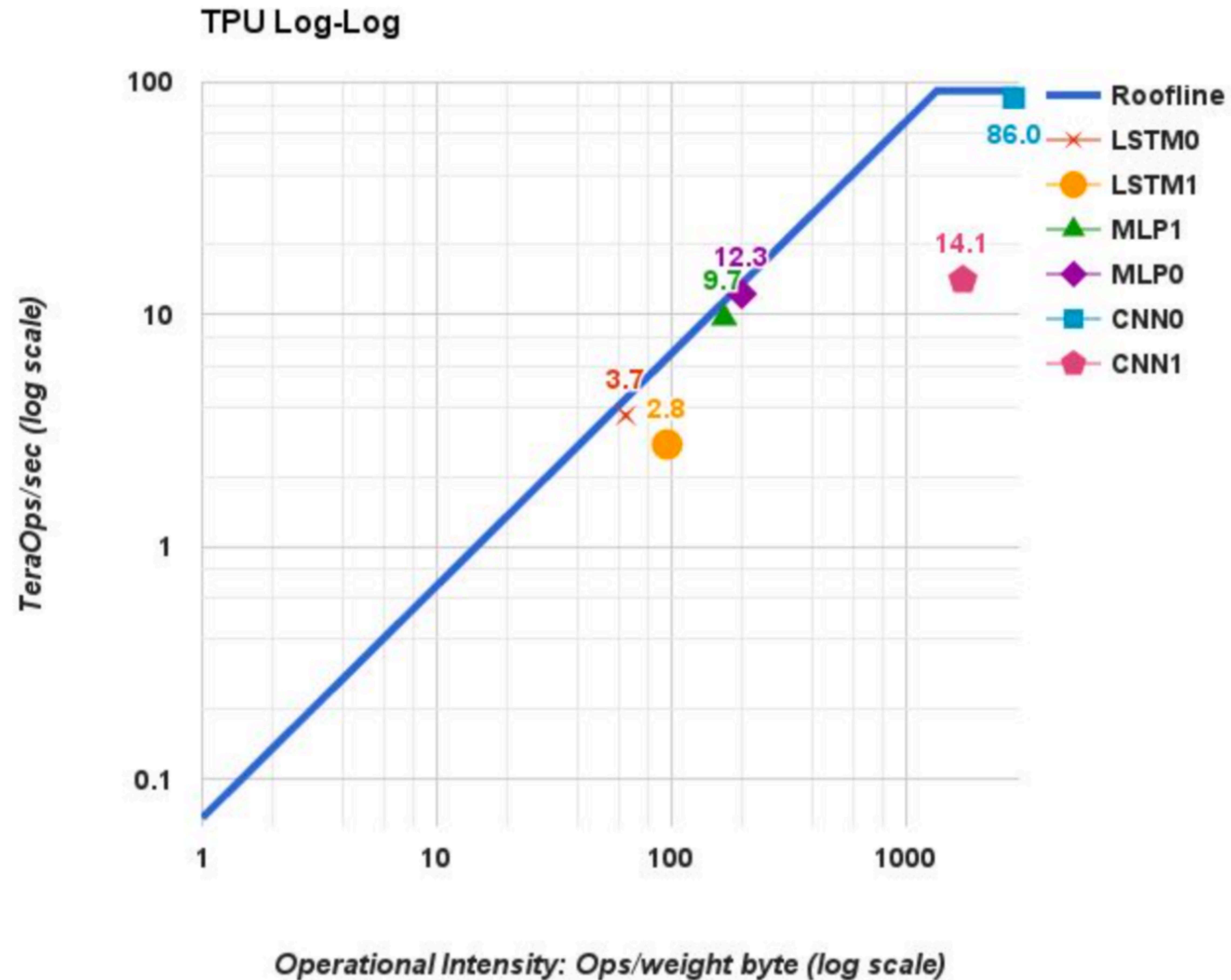
CONV and FC layers
(batch size > 1)



Reuse: Filter weights

Fig. 23. ***Data reuse opportunities in DNNs [82].***

What does the roofline tell us about ways to improve the TPU?



- What benchmarks would benefit from improvements on clock frequency?
- What benchmarks would benefit from higher memory bandwidth?

NVIDIA's Rebuttal to the TPU

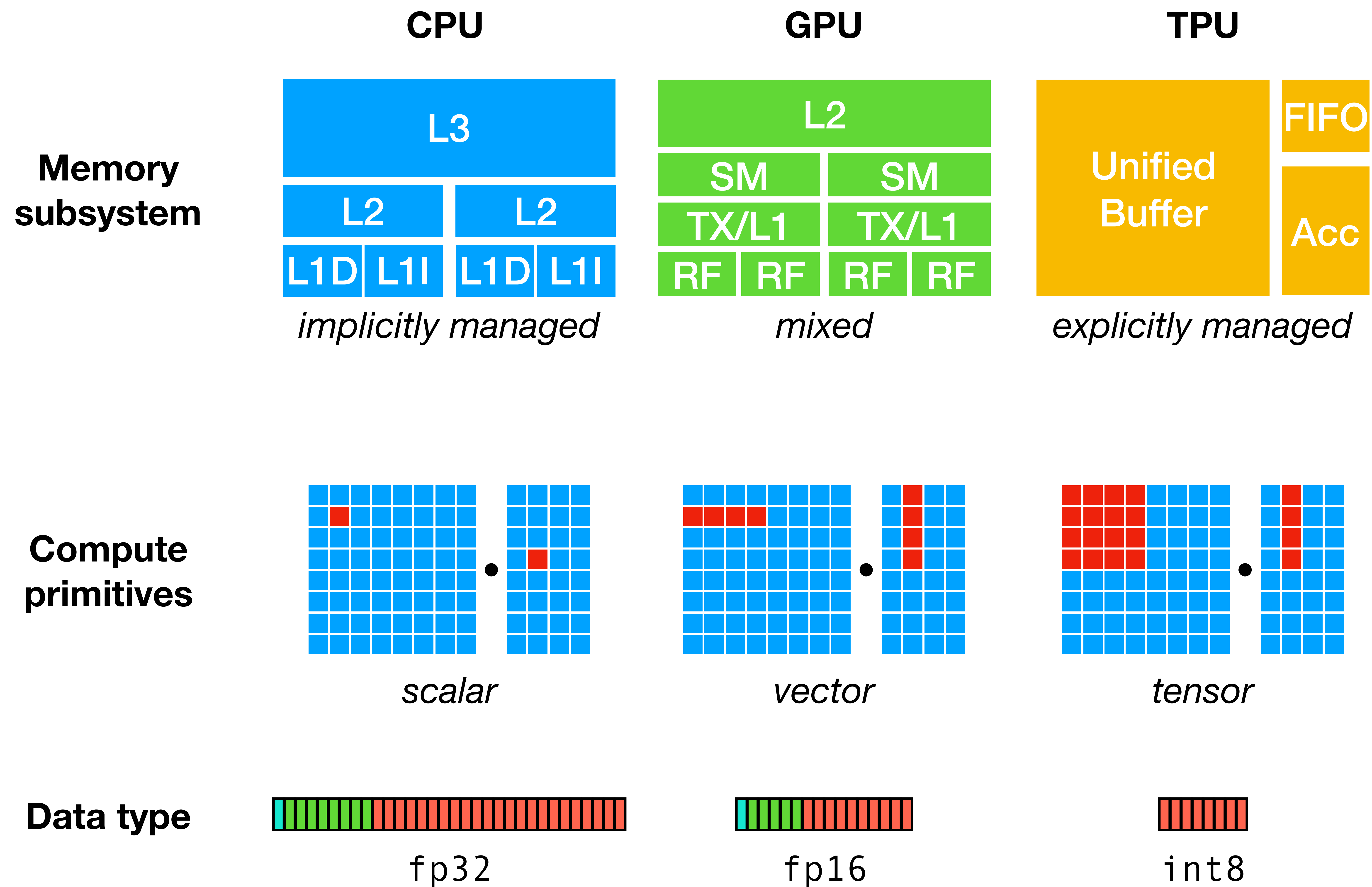
	K80 2012	TPU 2015	P40 2016
Inferences/Sec <10ms latency	$^{1}/_{13}$X	1X	2X
Training TOPS	6 FP32	NA	12 FP32
Inference TOPS	6 FP32	90 INT8	48 INT8
On-chip Memory	16 MB	24 MB	11 MB
Power	300W	75W	250W
Bandwidth	320 GB/S	34 GB/S	350 GB/S

https://blogs.nvidia.com/blog/2017/04/10/ai-drives-rise-accelerated-computing-datacenter/

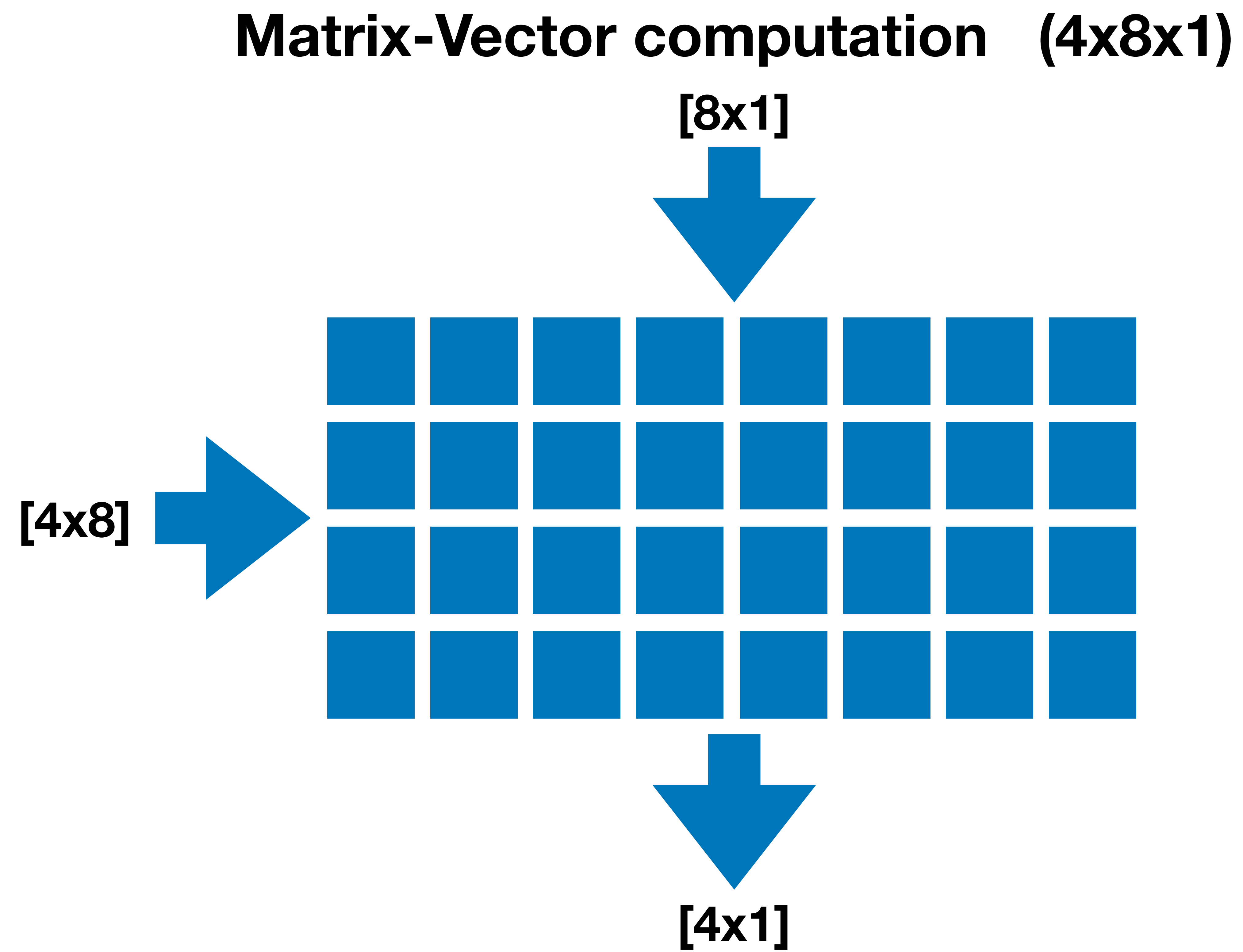
Discussion Break

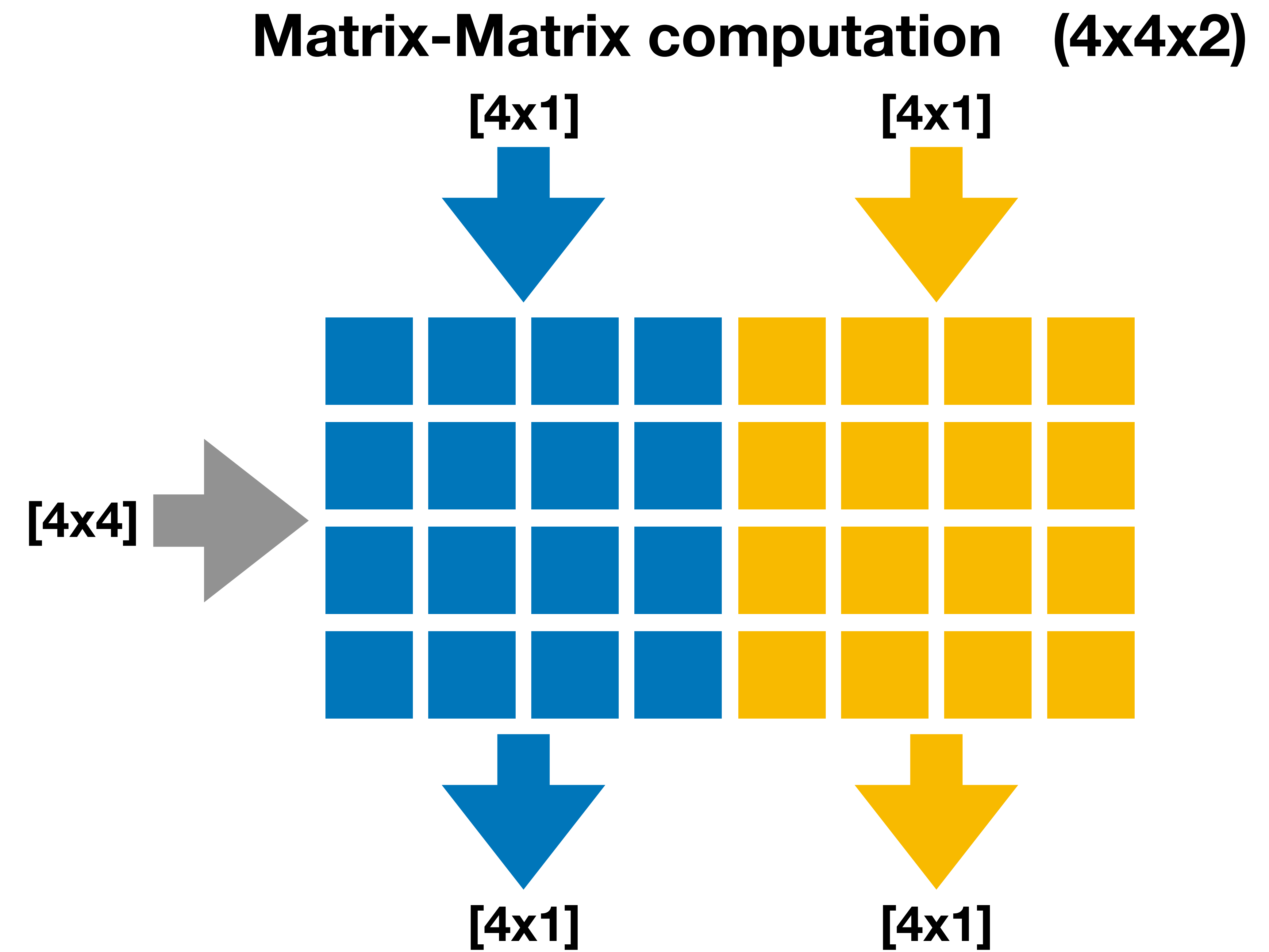
- What makes a specialized accelerator different from a CPU or GPU?

Deep Learning Accelerator Characteristics



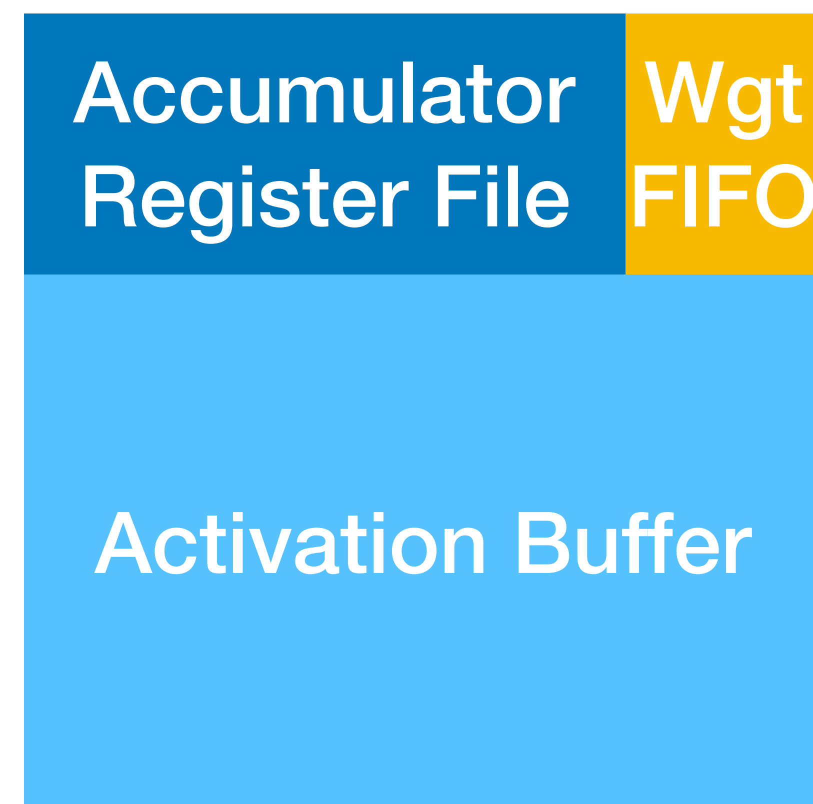
HW/SW Co-Design - #1 Tensorization





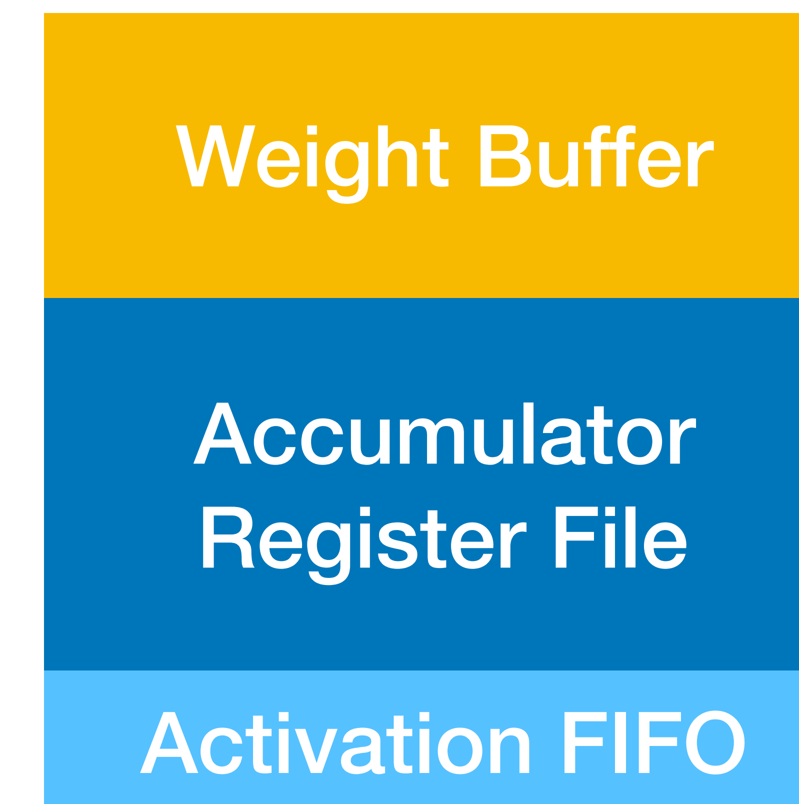
HW/SW Co-Design - #2 Memory Architecting

Convolution-Optimized, no batching



Large activation buffer for spatial reuse
Accumulator-local scheduling
Weight FIFO for single-use weights

GEMM-Optimized, batching



Activation FIFO for single-use activations
Large accumulator storage for GEMM blocking
Weight buffer for batched execution

HW/SW Co-Design - #3 Data Type

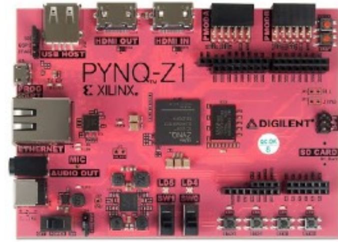
Reducing type width can result in a quadratic increase of compute resources, and linear increase of storage/bandwidth

But it also affects classification accuracy!

VTA: Versatile Tensor Accelerator

- **VTA:** a versatile and extendable deep learning accelerator for software codesign research and the development of next architectures

Addressing the Specialization Challenge



- Targets FPGAs on low-cost edge devices (PYNQ), and high-end datacenter (in progress), allowing for fast prototyping and deployment

- Leverages HLS-C, for code compactness and easy maintainability (<1000 LoC for IP)

- Built for customization, and modularity (extensible pipeline)

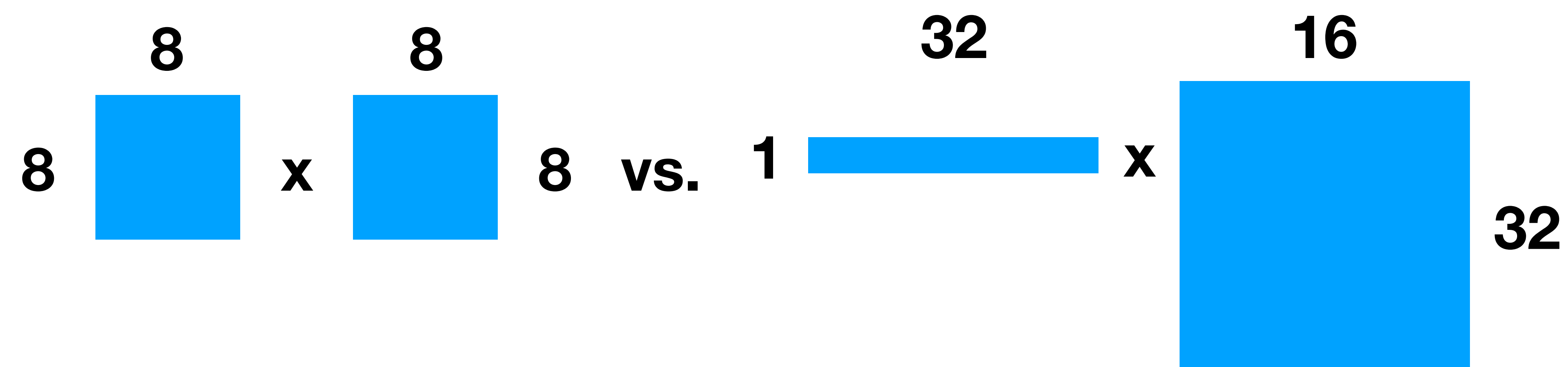
- Community driven (open-sourcing in progress)

VTA Features

- Customizable **tensor core**, **memory subsystem** and **data types** based on bandwidth, storage and accuracy needs
- Flexible **CISC/RISC ISA** for expressive and compact code
- Access-execute decoupling for memory **latency hiding**

Customization

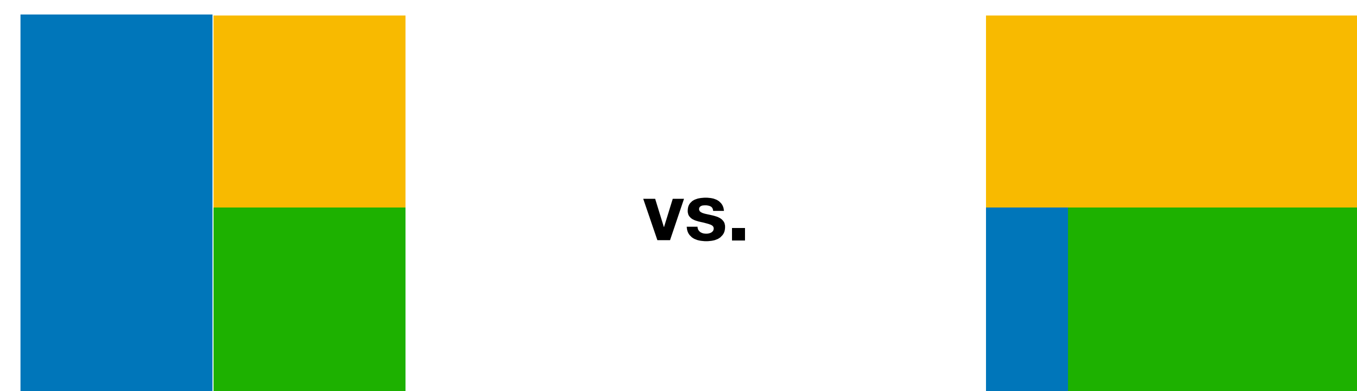
Tensor Intrinsic



Hardware Datatype

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

Memory Subsystem

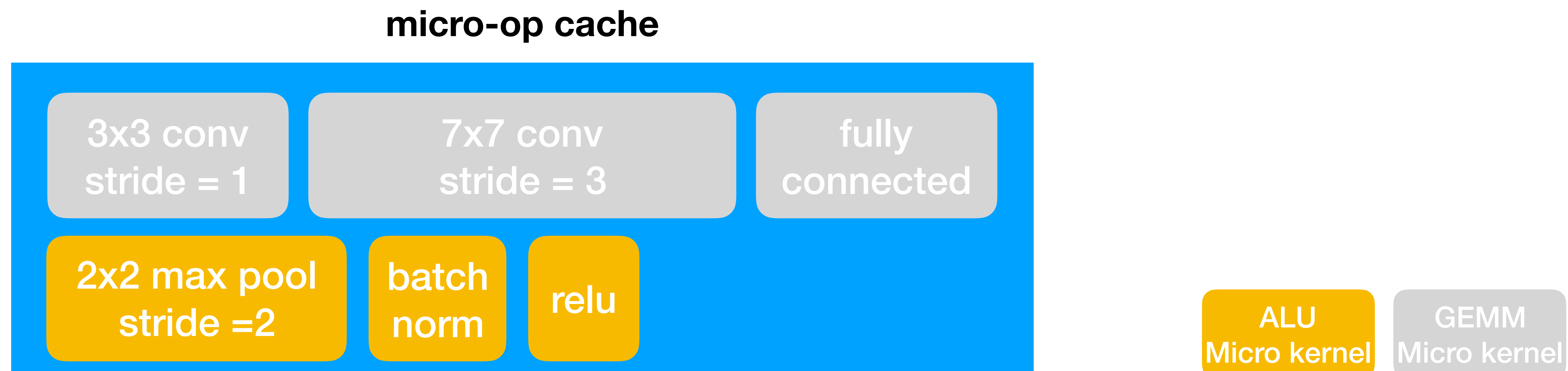


Operator Support

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

CISC/RISC ISA

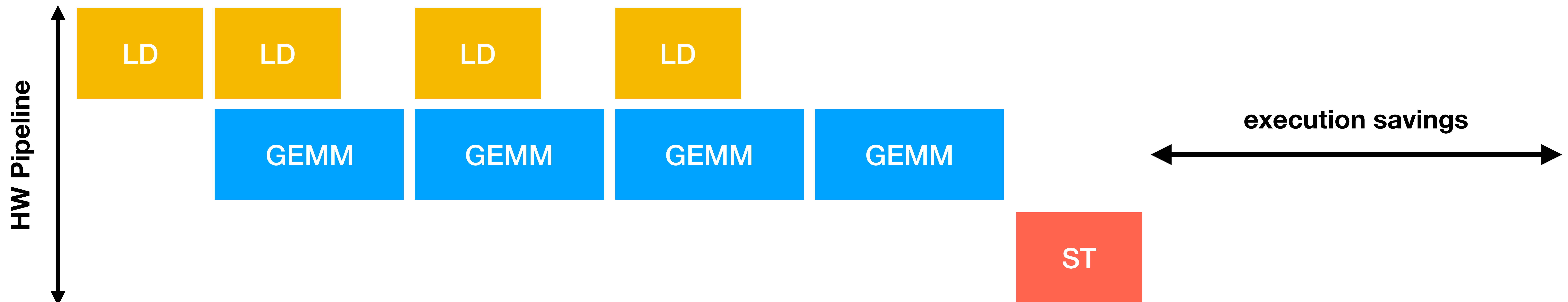
- Goal: Provide the right tradeoff between expressiveness and code compactness
  - Use CISC-ness to describe high-level operation (LD, ST, GEMM, ALU)
  - Use RISC-ness to describe low-level memory access patterns
- Micro-op kernels are stored in a local micro op cache to implement different operators



Latency Hiding

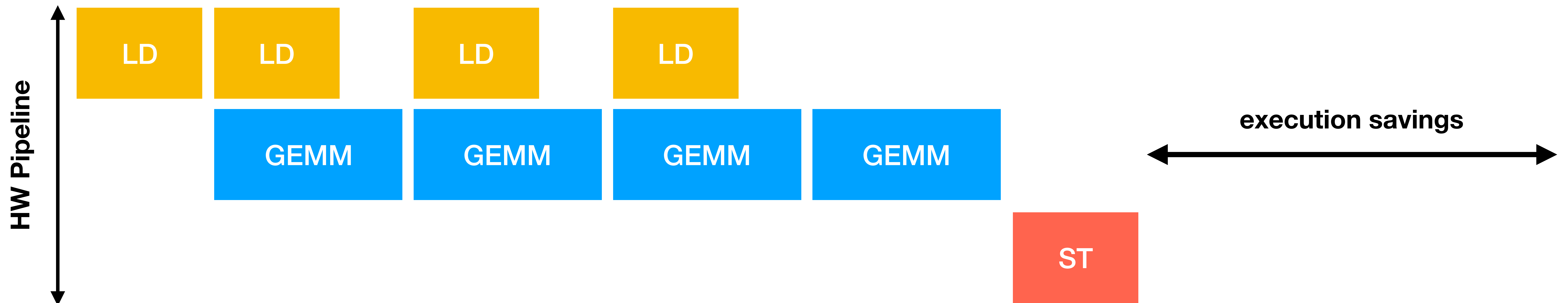
- How do keep computation resources (GEMM) busy:
  - Without latency hiding, we are wasting compute/memory resources

  - By exploiting pipeline parallelism, we can hide memory latency



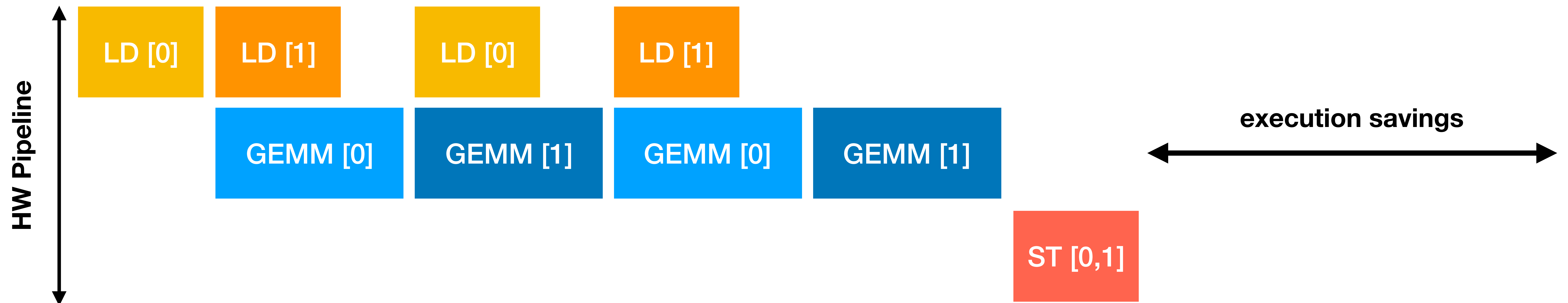
Latency Hiding

- Pipeline parallelism requirements:
    - Concurrent tasks need to access non-overlapping regions of memory



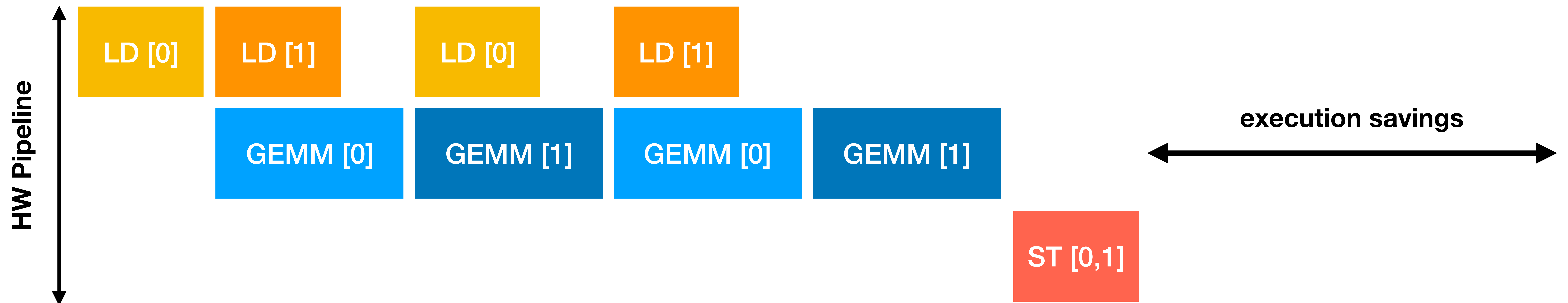
Latency Hiding

- Pipeline parallelism requirements:
  - Concurrent tasks need to access non-overlapping regions of memory



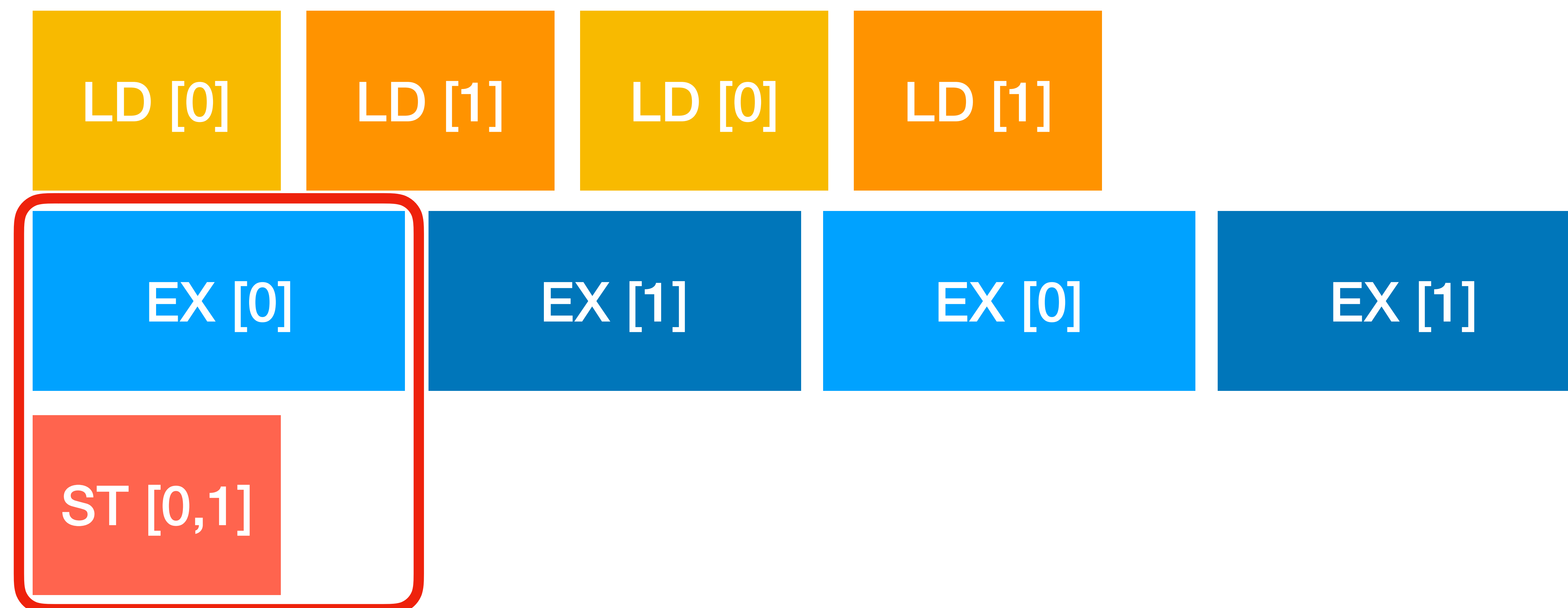
Latency Hiding

- Pipeline parallelism requirements:
  - Concurrent tasks need to access non-overlapping regions of memory
  - Data dependences need to be explicit!



Latency Hiding

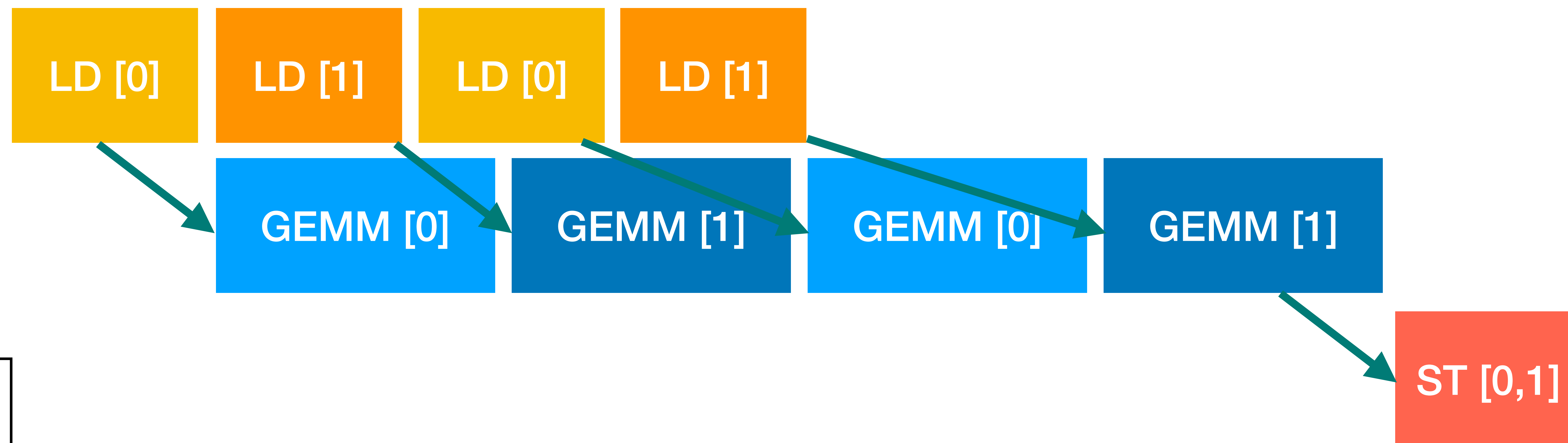
- We want to enforce read-after-write (RAW) dependences



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

Latency Hiding

- We want to enforce read-after-write (RAW) dependences

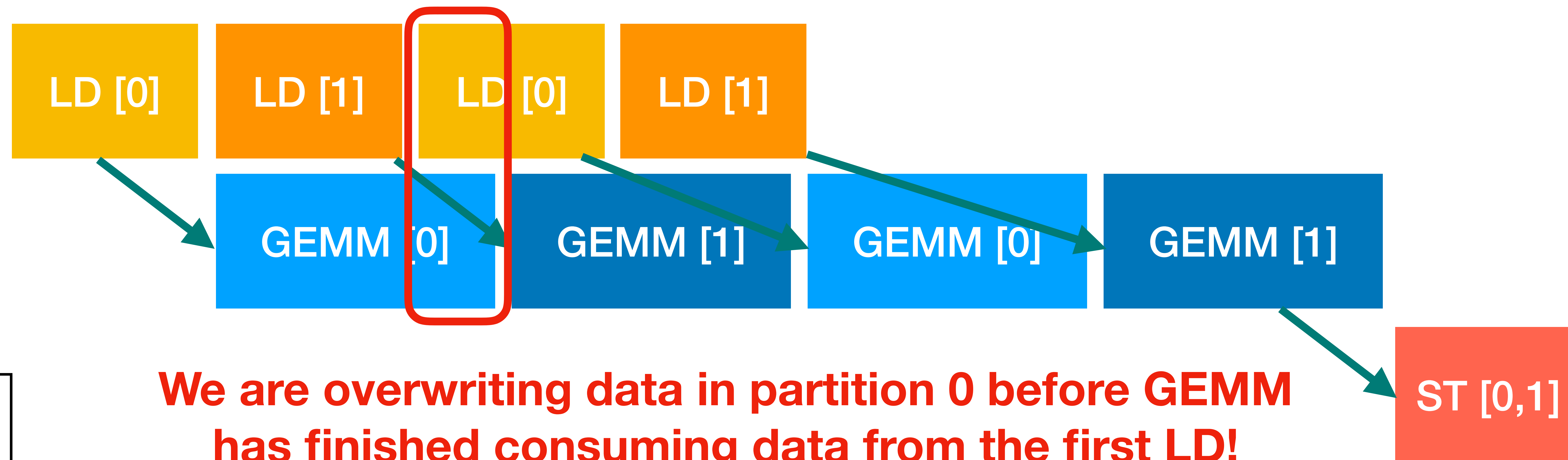


Legend:

RAW dependence:

Latency Hiding

- We want to enforce read-after-write (RAW) dependences
- AND we want to enforce write-after-read (WAR) dependences

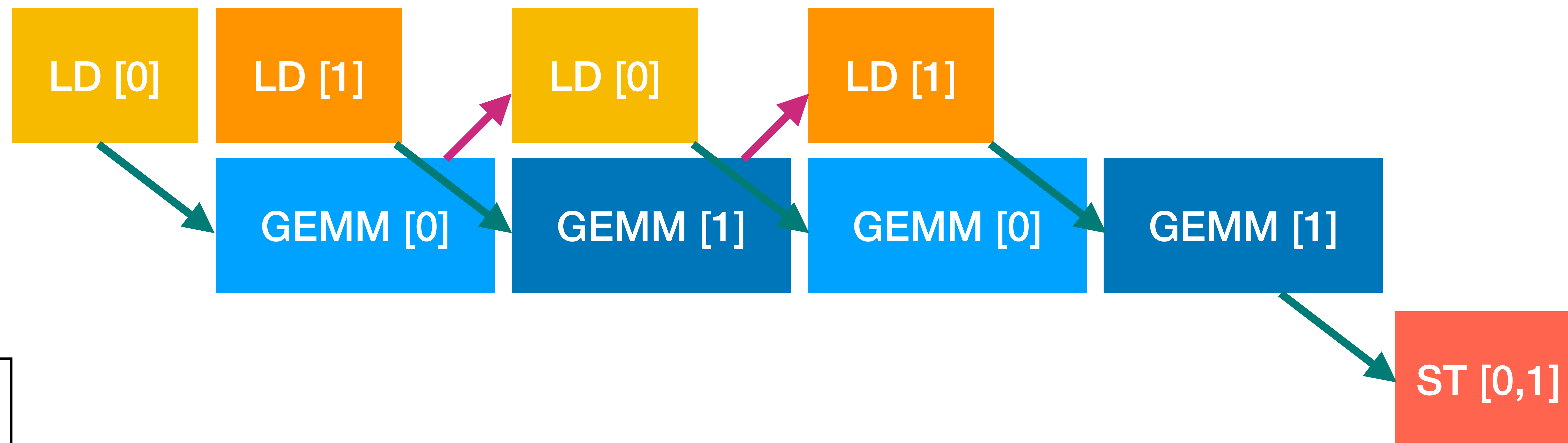


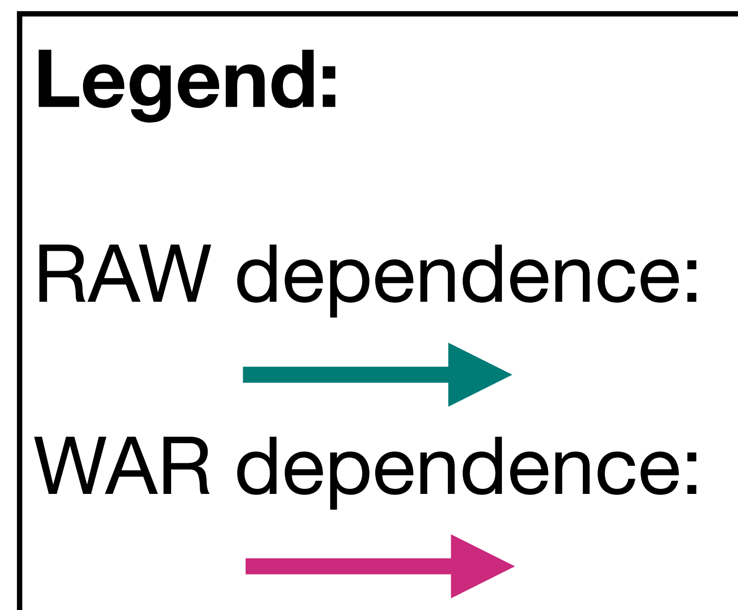
Legend:

RAW dependence:

Latency Hiding

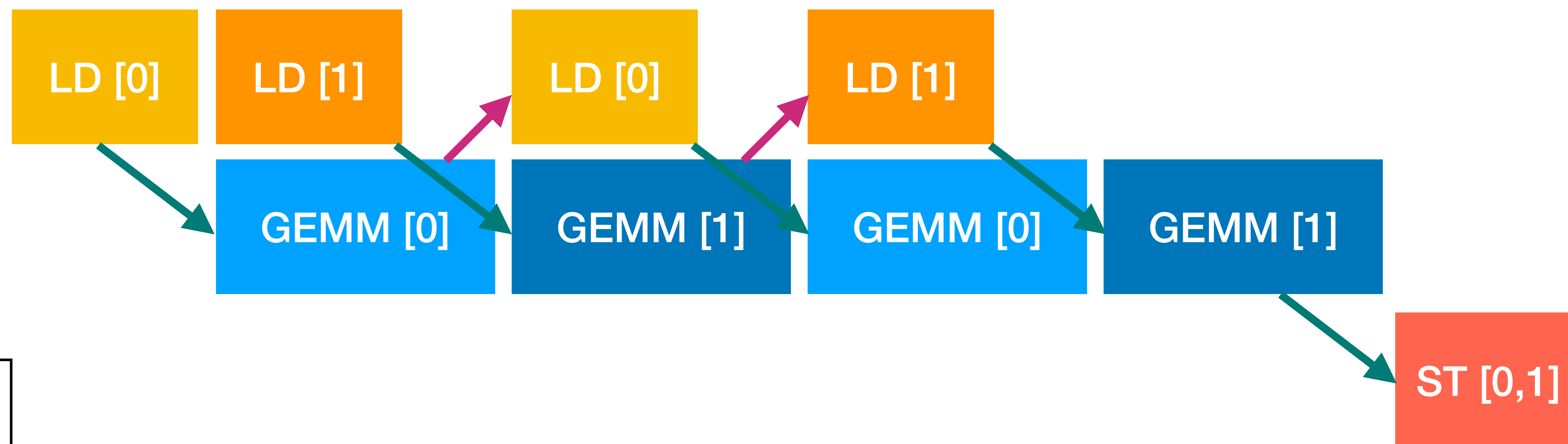
- We want to enforce read-after-write (RAW) dependences
- AND we want to enforce write-after-read (WAR) dependences

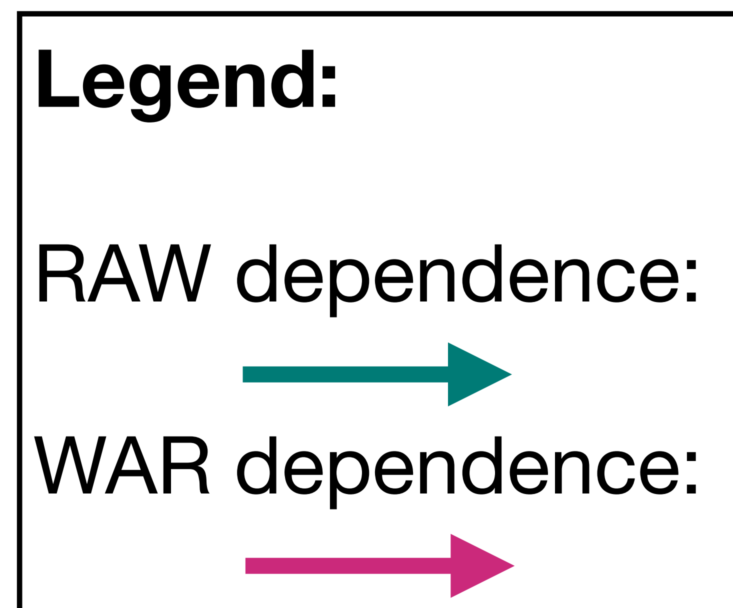




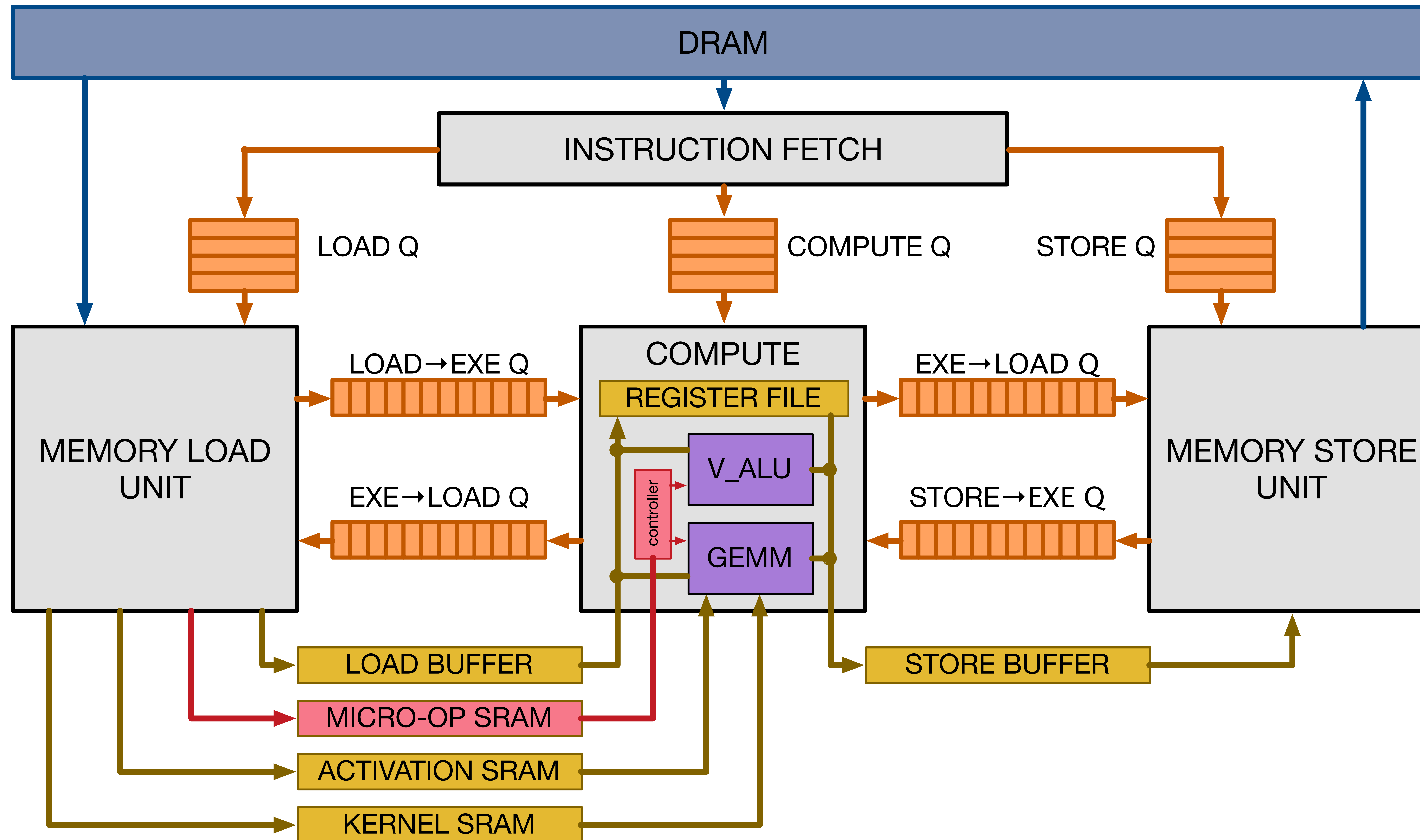
Latency Hiding

Takeaway: work partitioning and explicit dependence graph execution (EDGE) unlocks pipeline parallelism to hide the latency of memory accesses

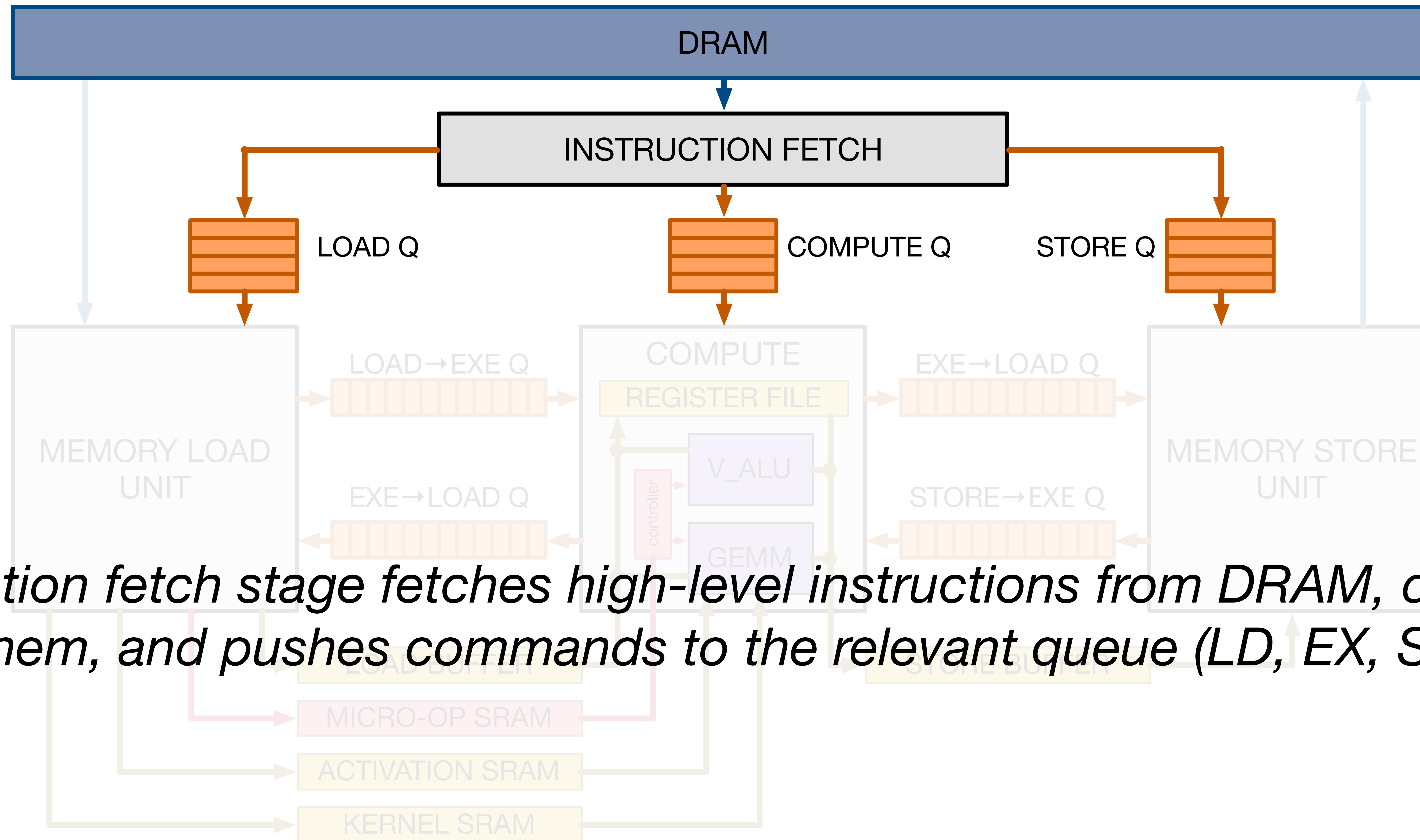




VTA Design Overview

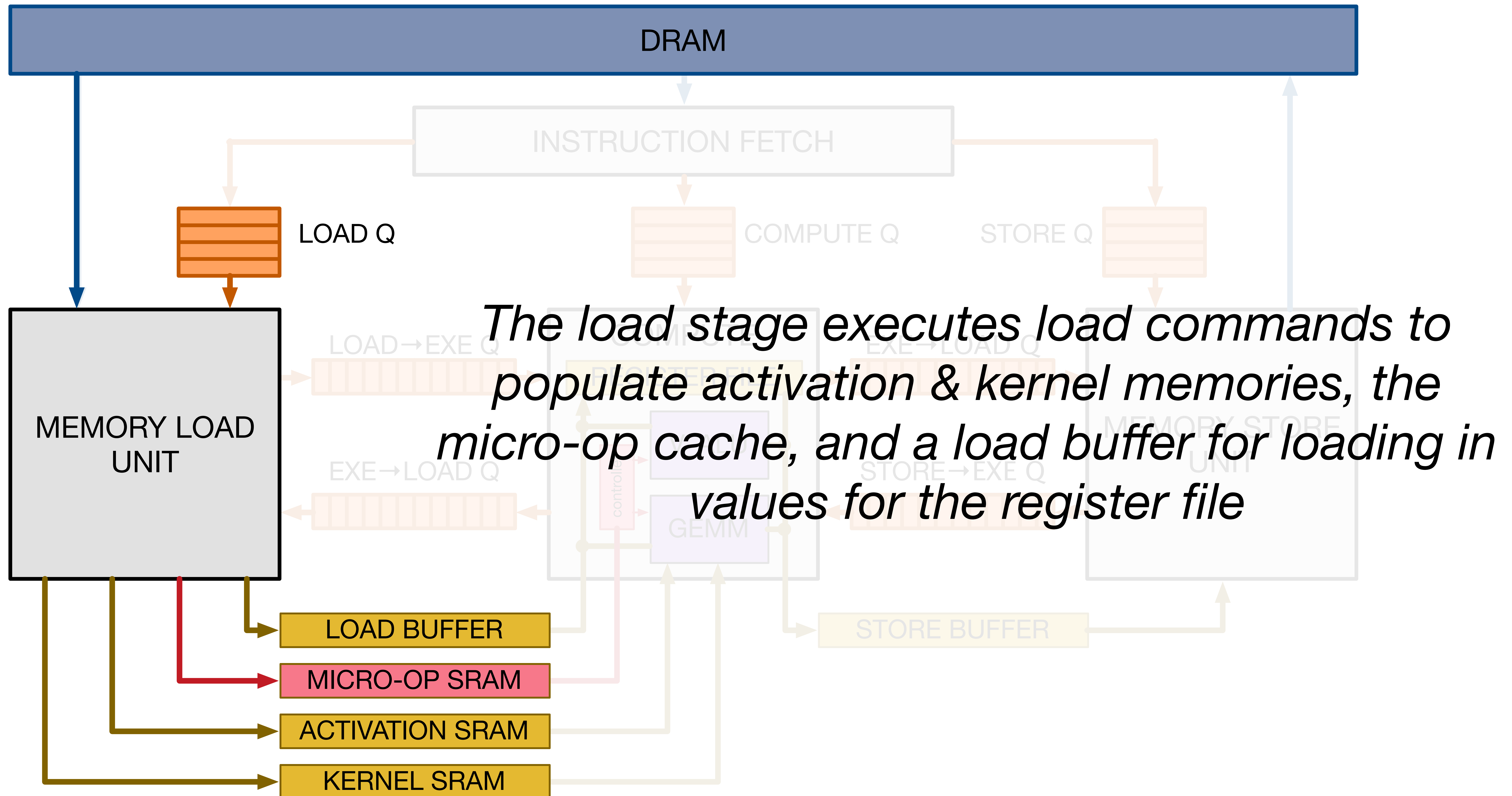


VTA Design



Instruction fetch stage fetches high-level instructions from DRAM, decodes them, and pushes commands to the relevant queue (LD, EX, ST)

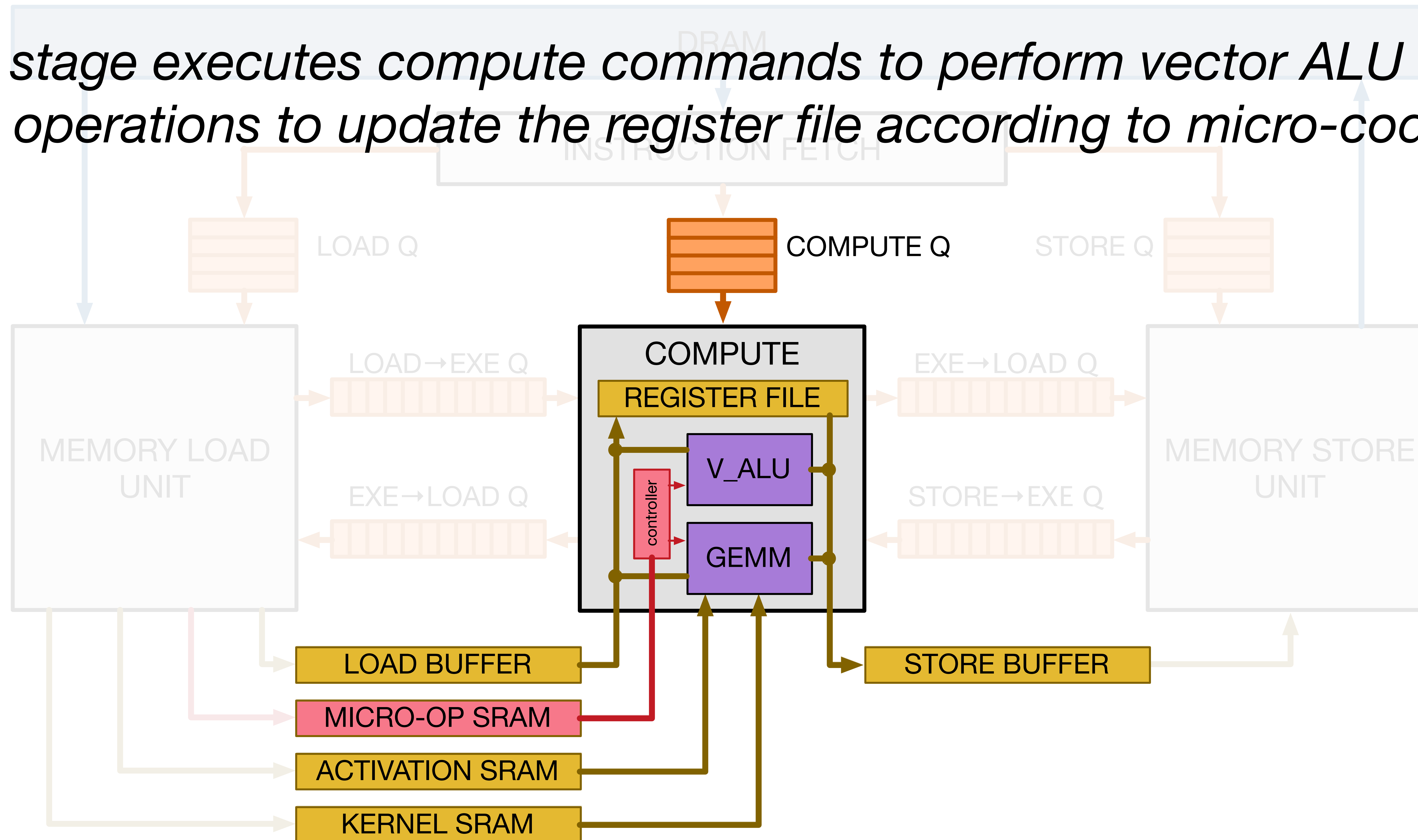
VTA Design



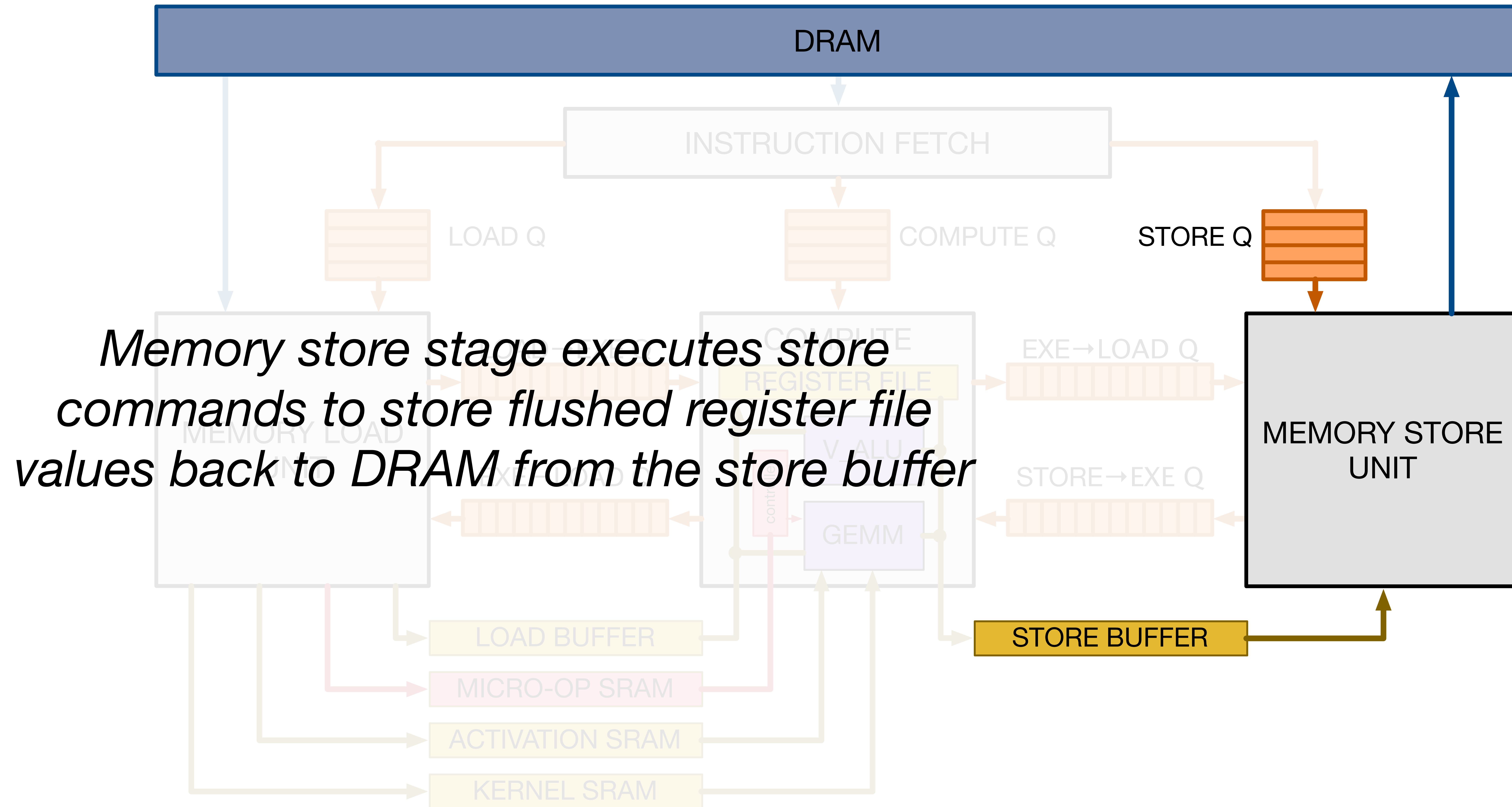
The load stage executes load commands to populate activation & kernel memories, the micro-op cache, and a load buffer for loading in values for the register file

VTA Design

Compute stage executes compute commands to perform vector ALU operations or GEMM operations to update the register file according to micro-coded kernels

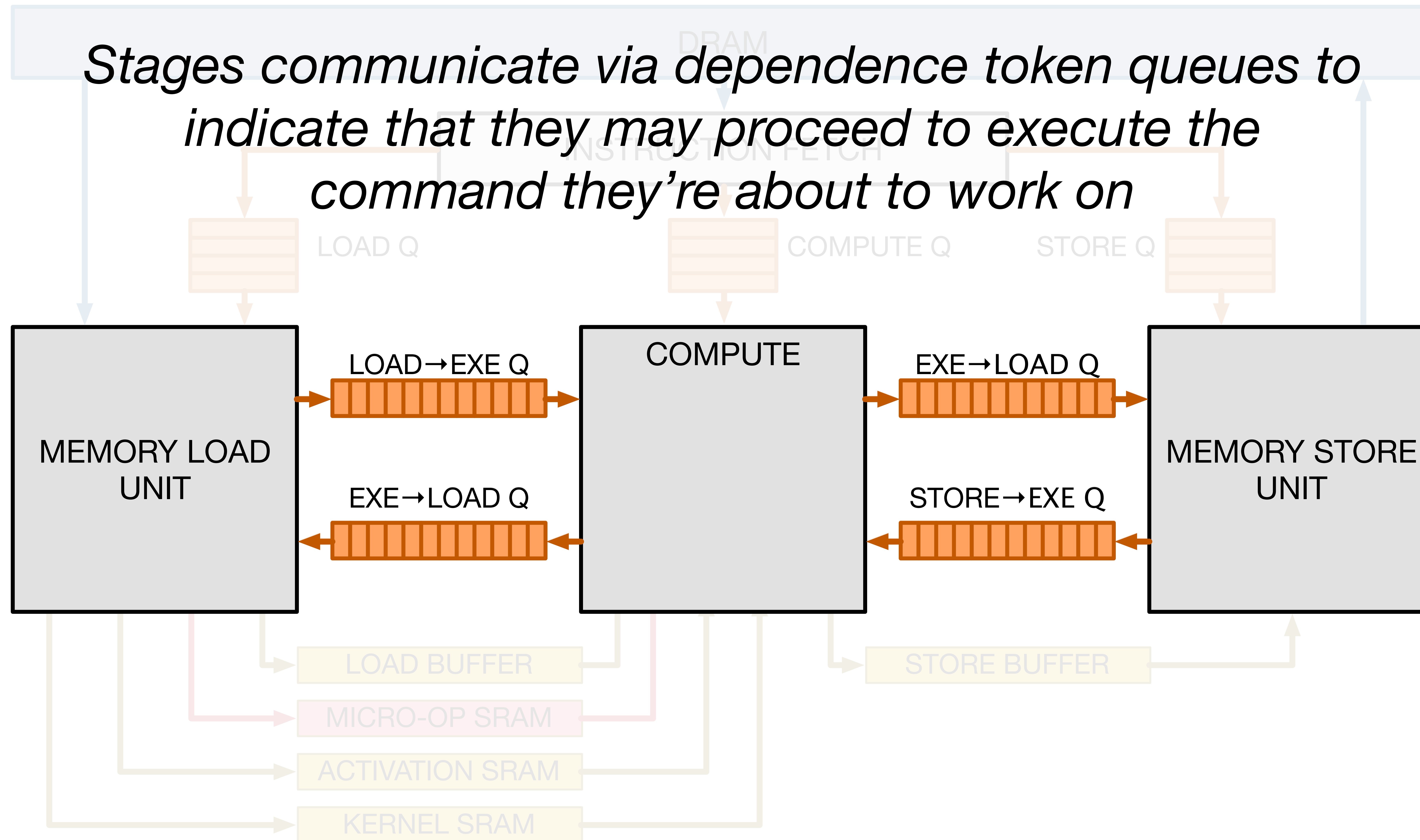


VTA Design



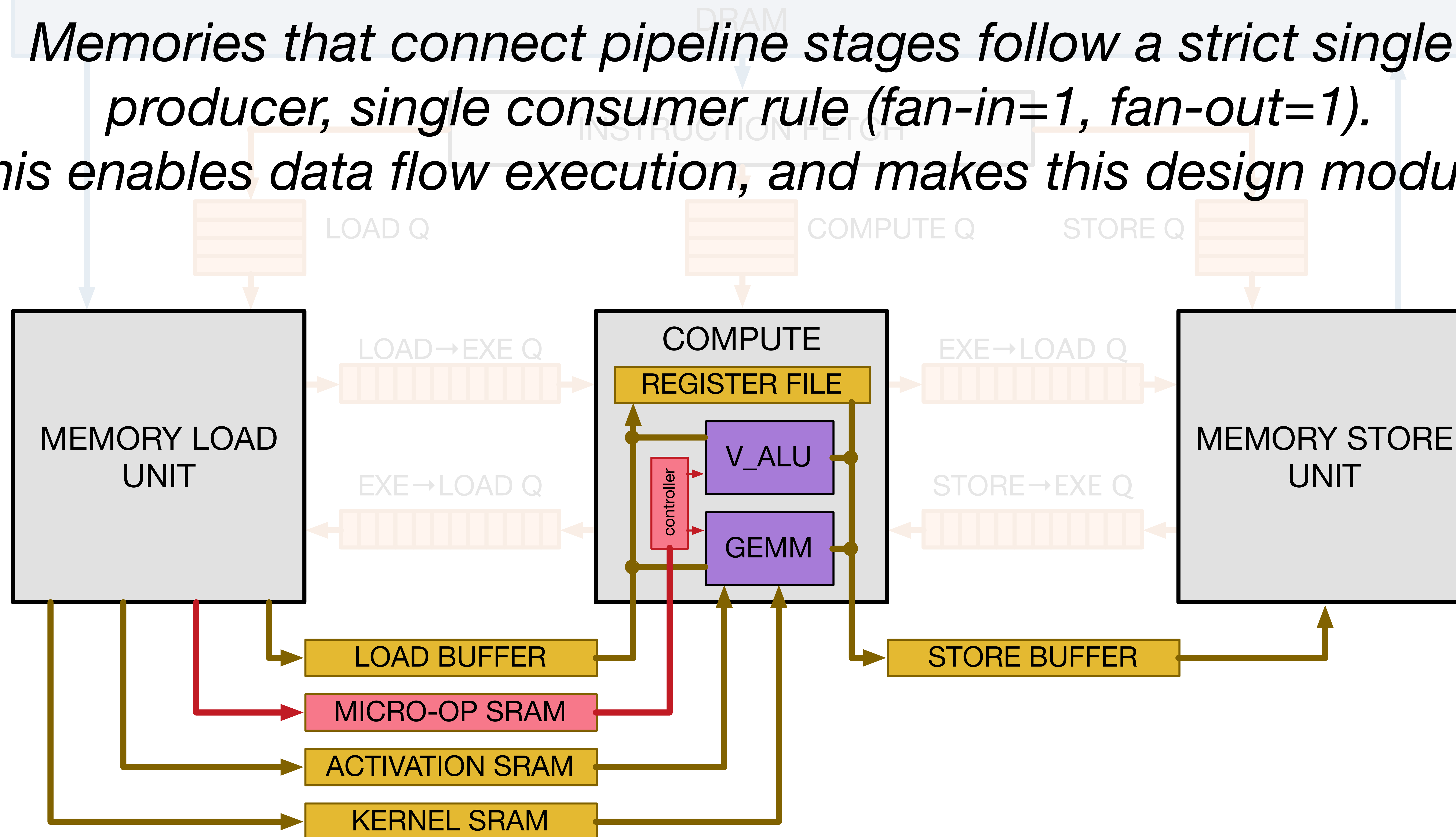
VTA Design

Stages communicate via dependence token queues to indicate that they may proceed to execute the command they're about to work on



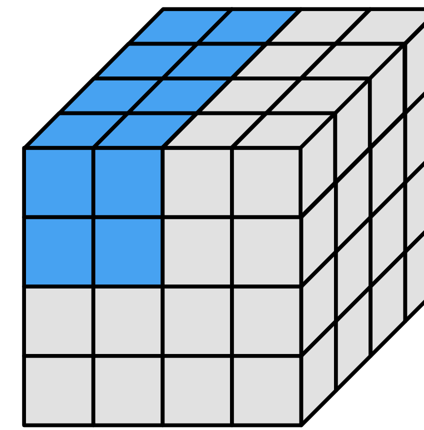
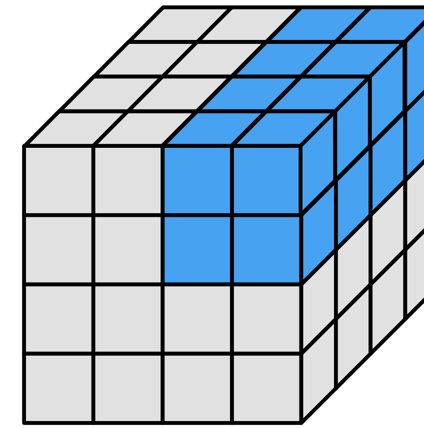
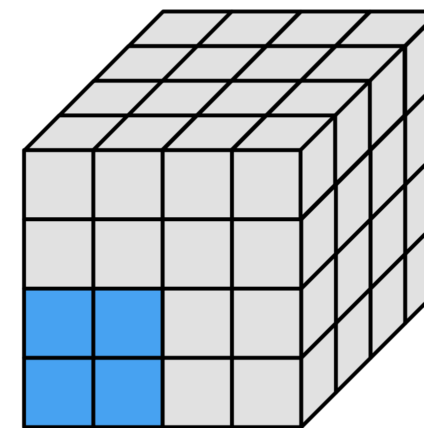
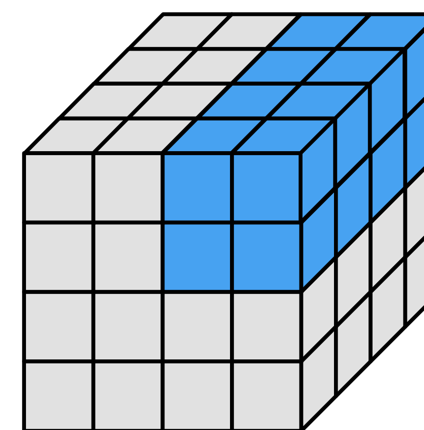
VTA Design

Memories that connect pipeline stages follow a strict single producer, single consumer rule (fan-in=1, fan-out=1). This enables data flow execution, and makes this design modular.



VTA Microprogramming

```
// 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

VTA Microprogramming

```
// Pseudo-code for convolution program for the VIA accelerator
// Virtual Thread 0
0x00: LOAD(PARAM[ 0-71])                                    // LD@TID0
0x01: LOAD(ACTIV[ 0-24])                                    // L
0x02: LOAD(LDBUF[ 0-31])                                    //  @TID0
0x03: PUSH(LD->EX)                                          //  LD@T
0x04: POP (LD->EX)                                          //  EX
0x05: EXE (ACTIV[ 0-24],PARAM[ 0-71],LDBUF[ 0-31],STBUF[ 0- 7  //
0x06: PUSH(EX->LD)                                          //
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], BU
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],STBUF[    7])
0x1D: PUSH(EX->ST)
0x1E: POP (EX->ST)                                          //
0x1F: STOR(STBUF[ 0- 7])                                    // S
// Virtual Thread 3
0x20: POP (EX->LD)                                          // D@TID3
0x21: LOAD(ACTIV[25-50])                                    //
0x22: LOAD(LDBUF[32-63])                                    // LD
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])

    idx_rf^n] += GEVM(act[idx_act^n], par[idx_par^n])
```

(b) onvolution micro-coded program

Microprogramming is a pain!

```
l, atch normalization and activation function
pa ern dictated by micro-coded program.
gi er index is derived as a 2D affine function.
x_ds  a_dst y+b_dst x+c_dst^0, where c_dst^0 is specified by
cr  op 0 fields.
0…
[ j)
p ling
x_dst  = MAX(rf[idx_dst^0], rf[idx_src^0])
idx  ] = MAX(rf[idx_dst^1], rf[idx_src^1])
…
// tch norm
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])
```

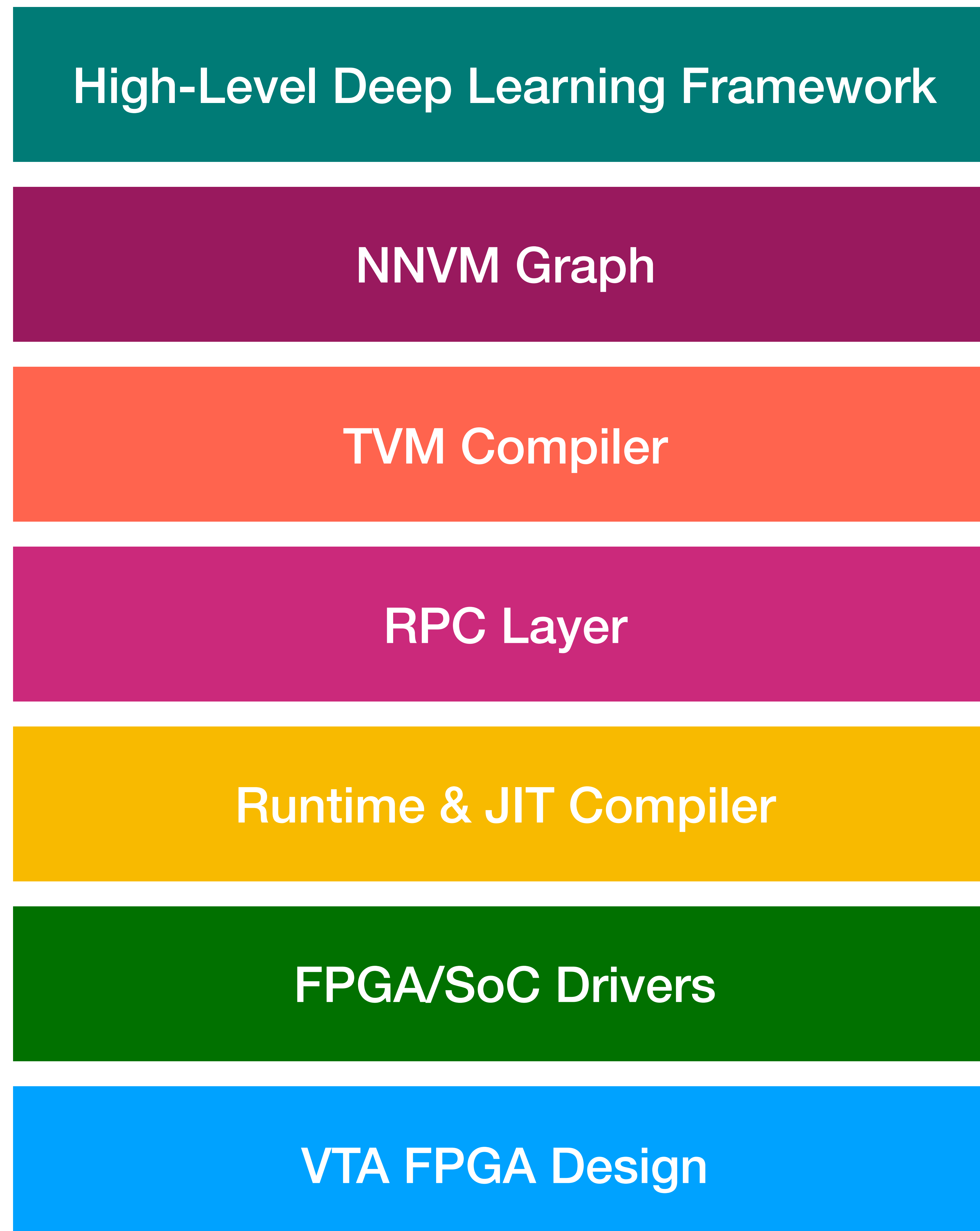
(c) Max pool, batch norm and activation micro-coded program

Building a deep learning accelerator compiler stack in TVM

- **TVM**: An end-to-end compiler & optimization framework for diverse hardware

Chen, Moreau, Jiang, Shen, Yan, Wang, Hu, Ceze, Guestrin and Krishnamurthy
TVM: End-to-end Compilation Stack for Deep Learning
SysML 2018 (1 of 6 invited talk)

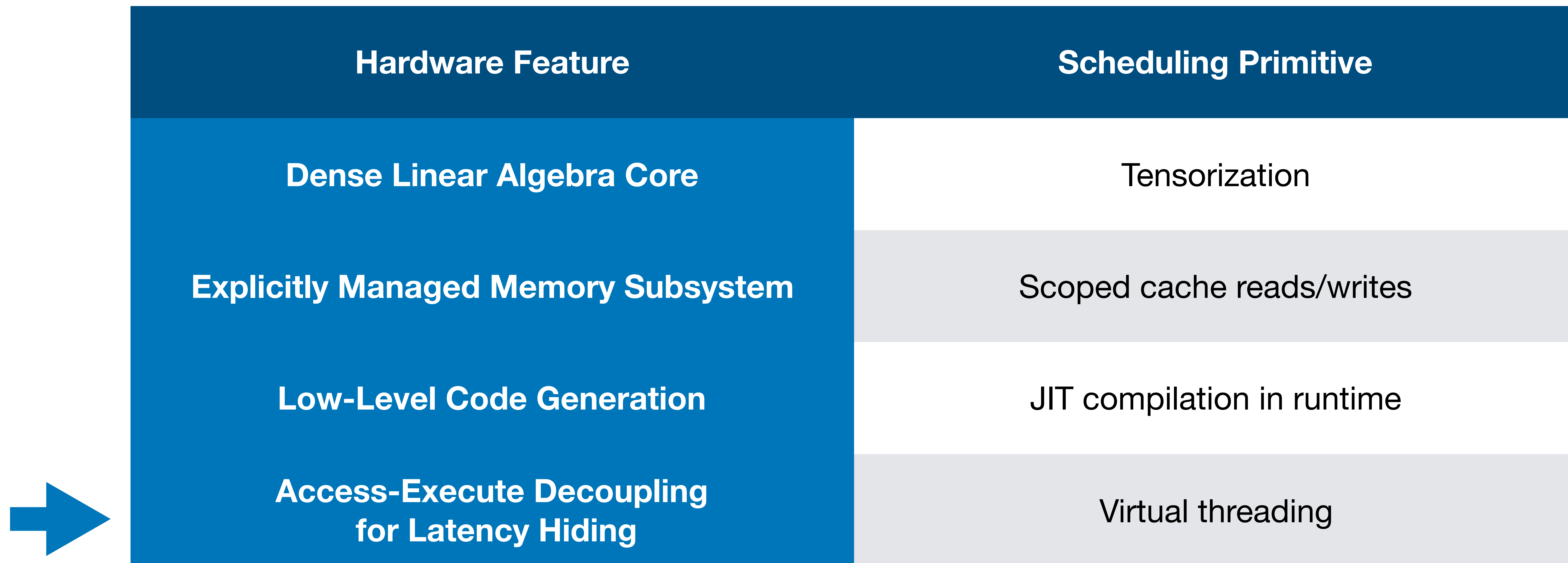
Addressing the Programmability Challenge



TVM DSL allows for separation of schedule and algorithm

Designing Scheduling Primitives for VTA

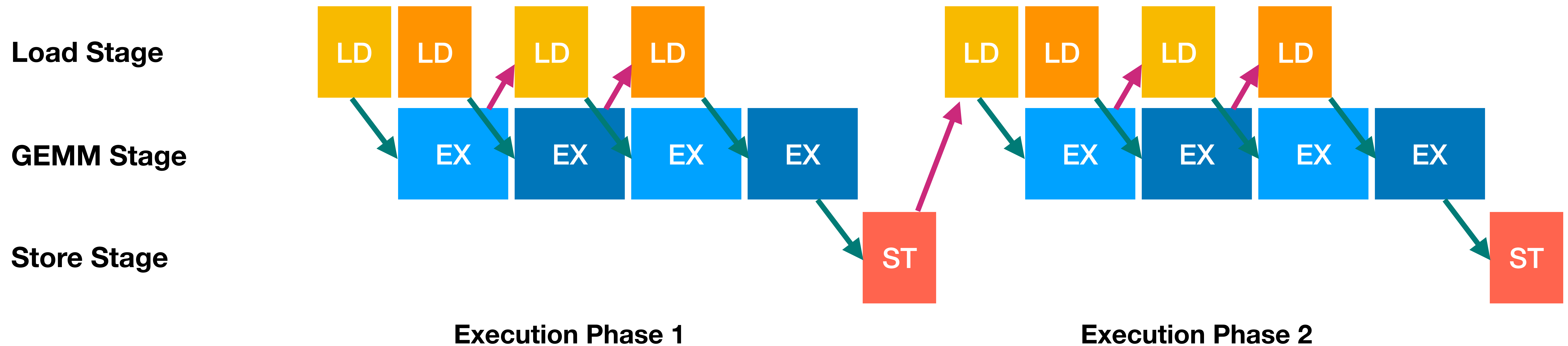
- What does an optimization stack for deep learning accelerators look like?



Hardware Feature	Scheduling Primitive
Dense Linear Algebra Core	Tensorization
Explicitly Managed Memory Subsystem	Scoped cache reads/writes
Low-Level Code Generation	JIT compilation in runtime
Access-Execute Decoupling for Latency Hiding	Virtual threading

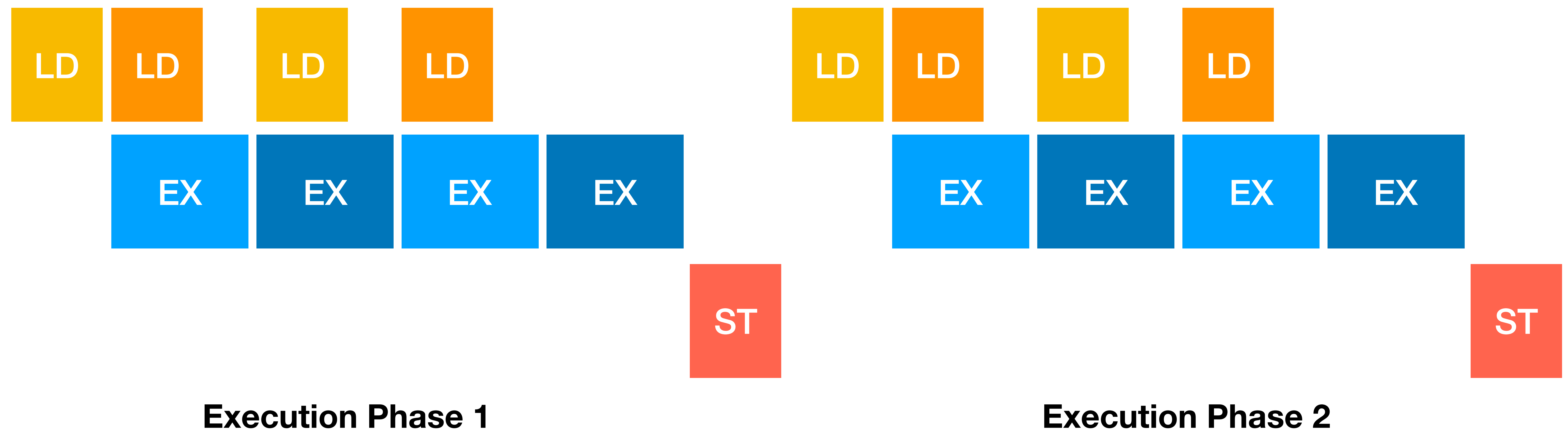
Virtual Threading

- How do we take advantage of pipeline parallelism with virtual threading?



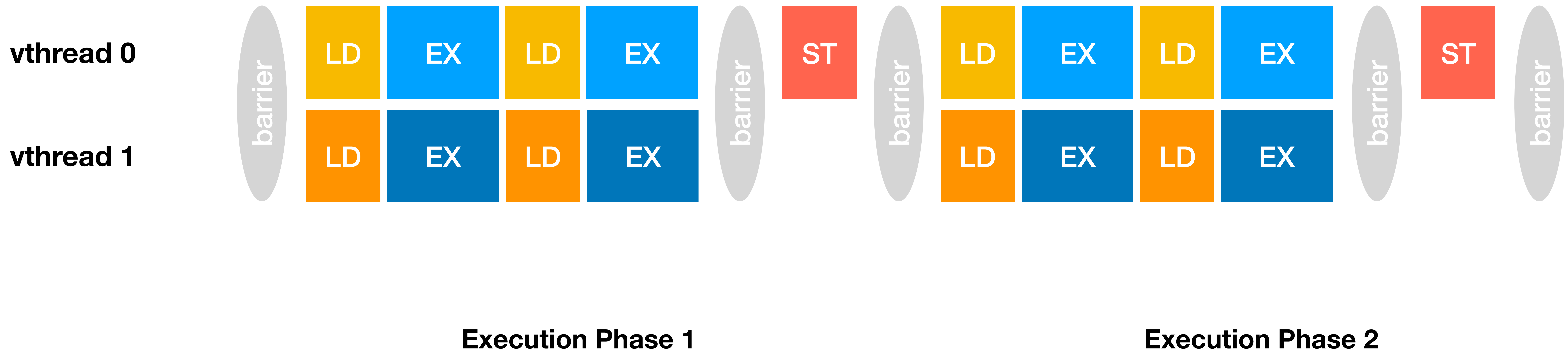
Virtual Threading

- How do we take advantage of pipeline parallelism with virtual threading?



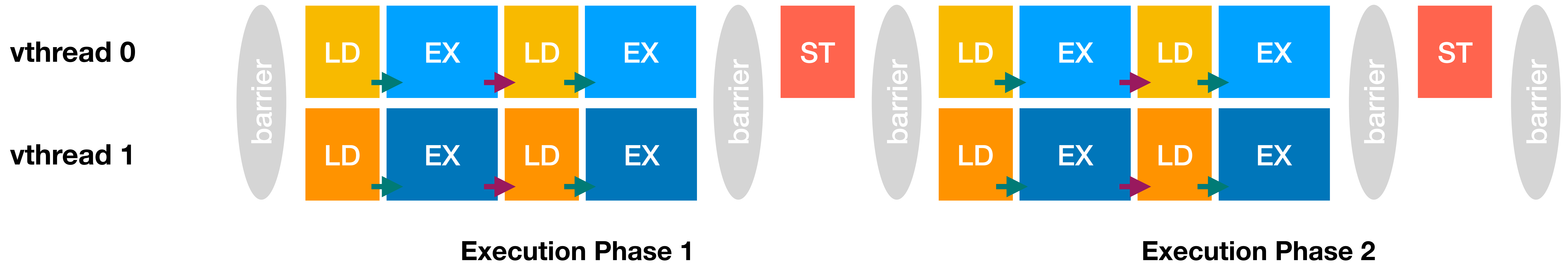
Virtual Threading

- How do we take advantage of pipeline parallelism with virtual threading?



Virtual Threading

Software-centric view: threaded execution



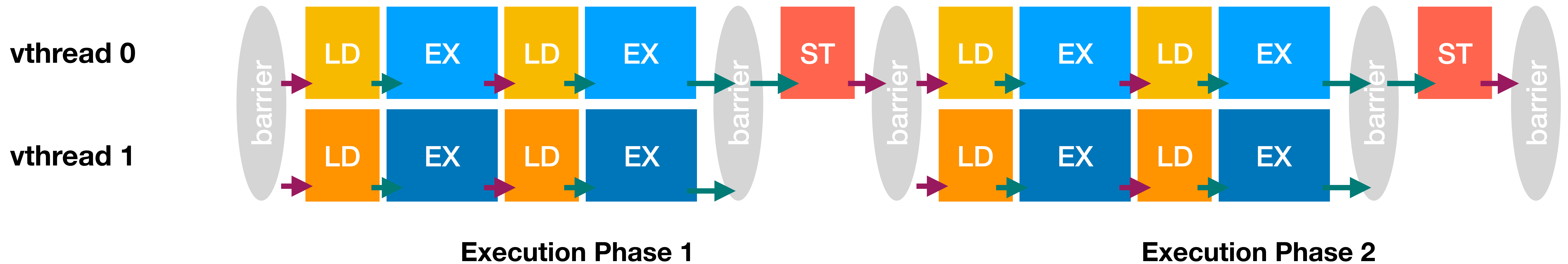
- Benefit #1: dependences are automatically inserted between successive stages within each virtual thread

Legend:

RAW dependence:

WAR dependence:

Virtual Threading



- 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

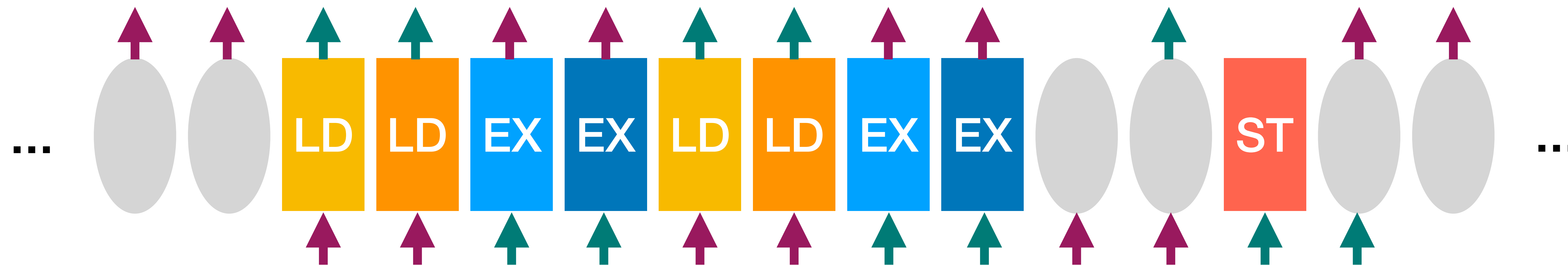
Legend:

RAW dependence:

WAR dependence:

Virtual Threading

Final step: virtual thread lowering into a single instruction stream



Legend

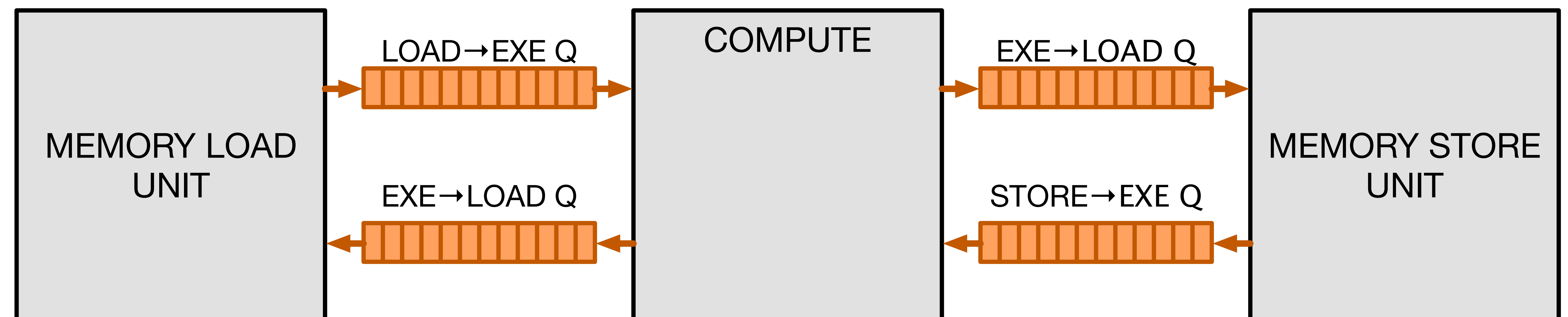
push dependence to consumer stage

push dependence to producer stage

pop dependence from producer stage

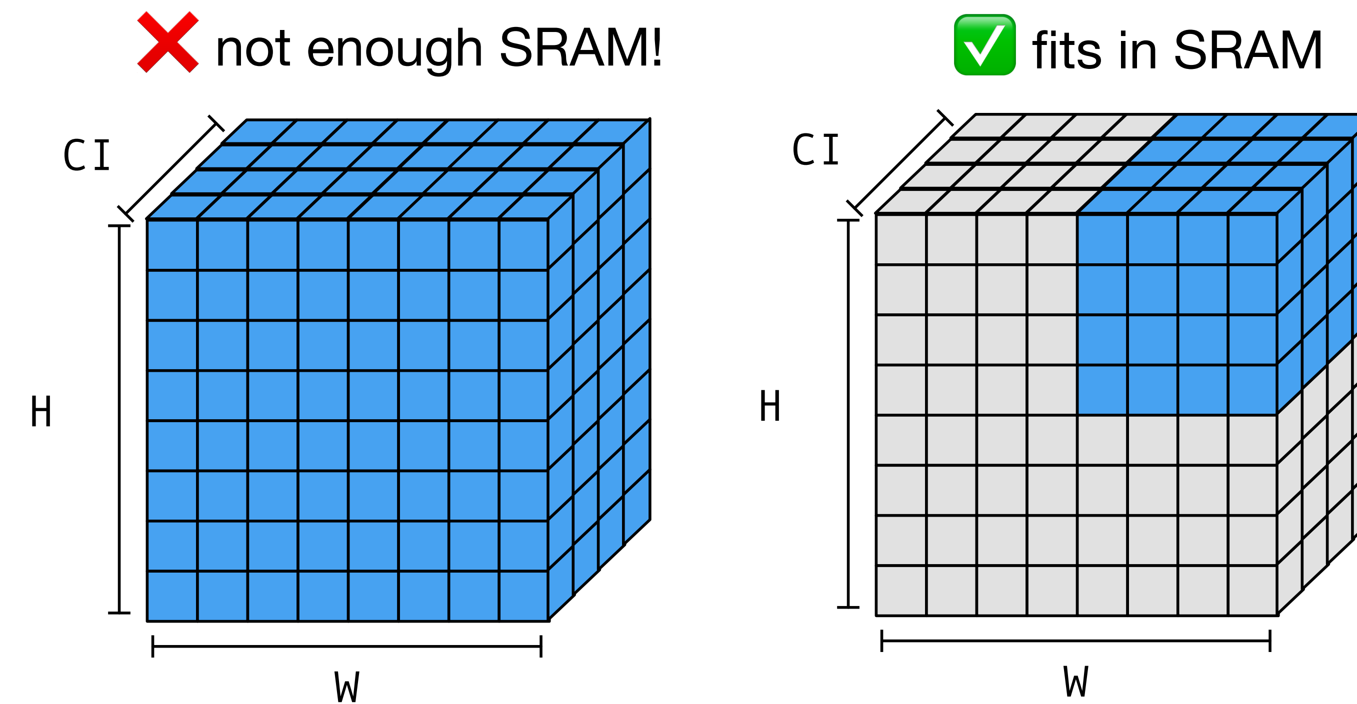
pop dependence from consumer stage

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

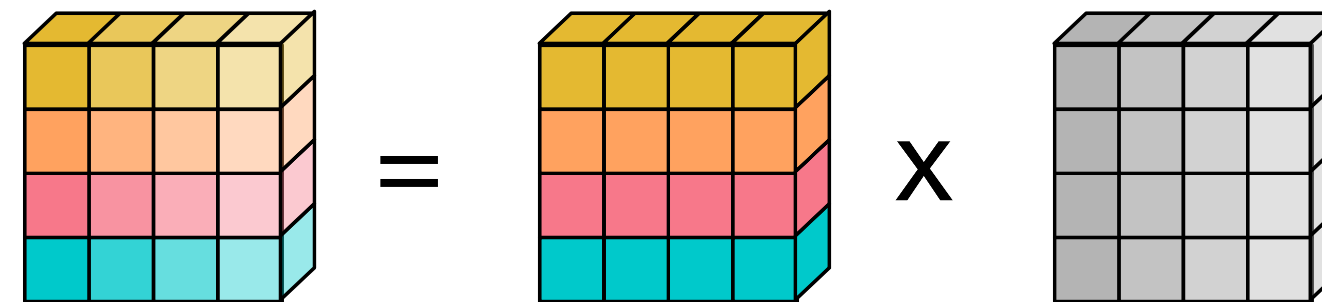


Programming for VTA in TVM

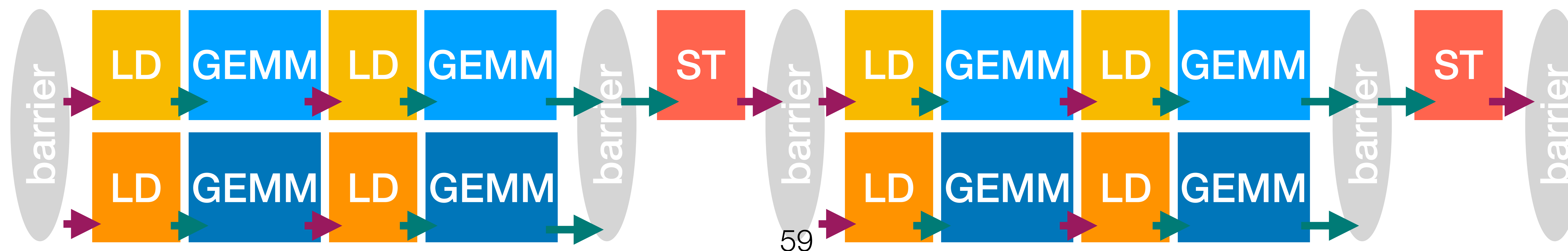
1. How do we partition work and explicitly manage on-chip memories?



2. How do we take advantage of tensorization?



3. How do we take advantage of virtual threading?



TVM Scheduling Primitives

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

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

2. How do we take advantage of tensorization?

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

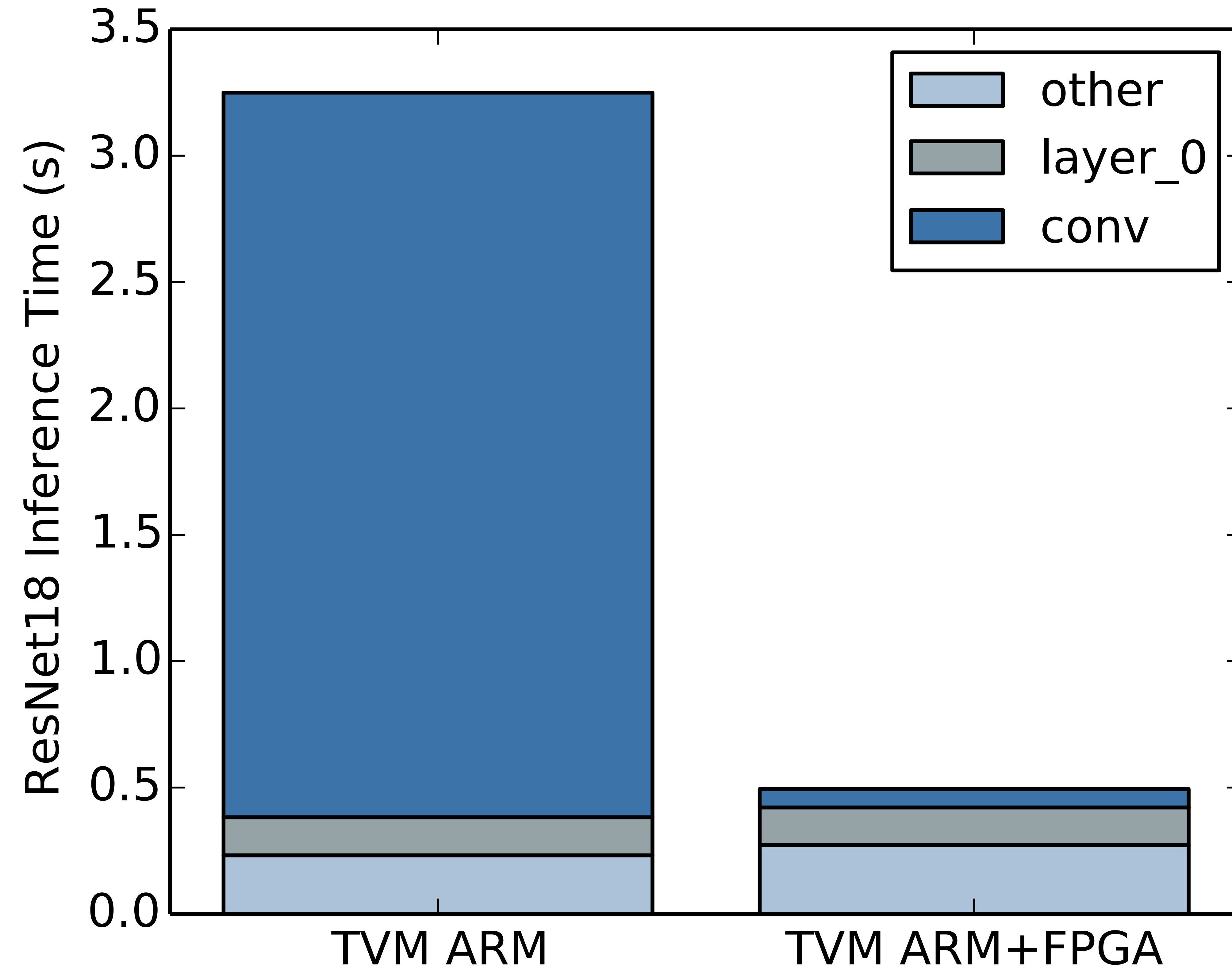
3. How do we take advantage of virtual threading?

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

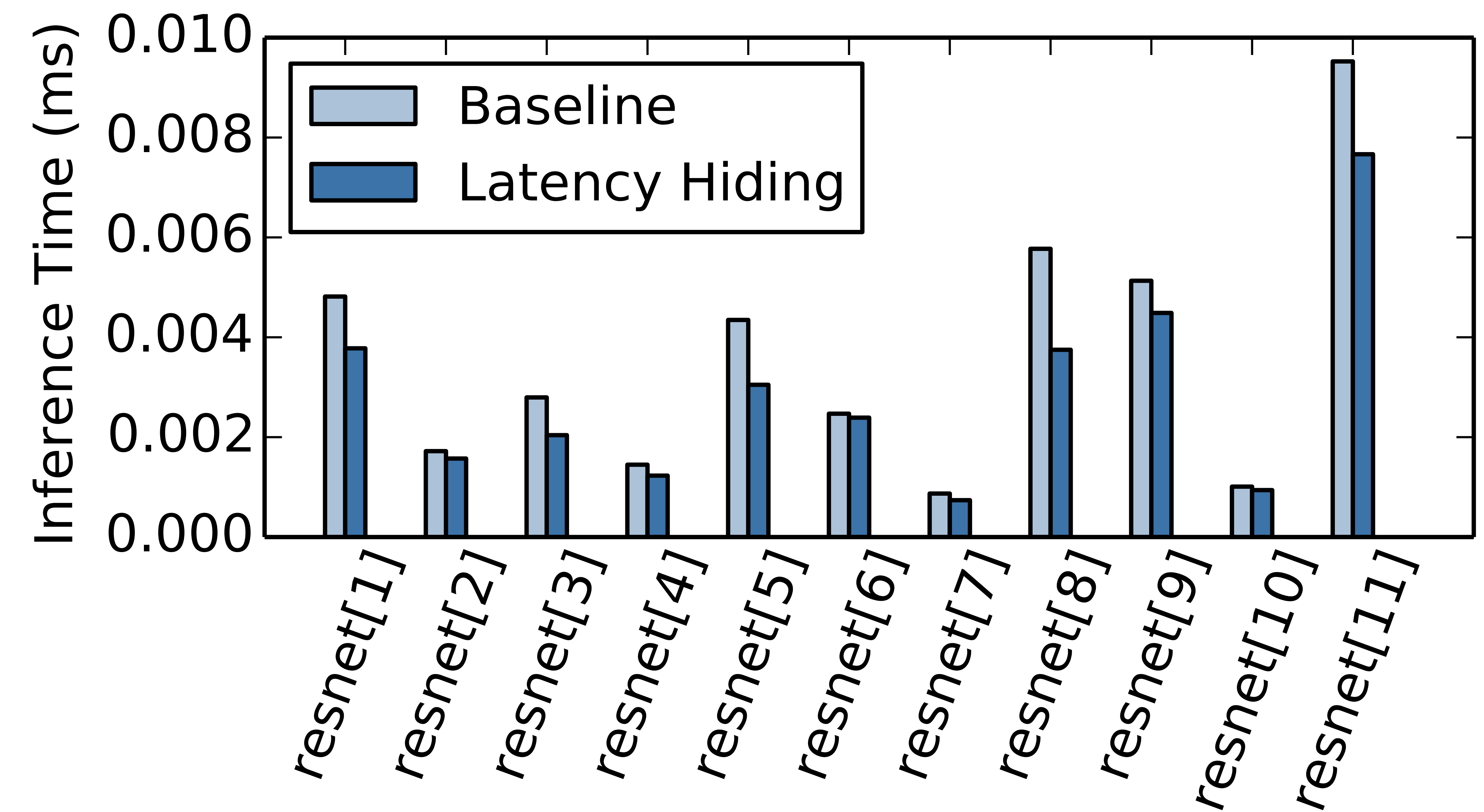
Full Stack Evaluation (TVM)

- Full evaluation on PYNQ FPGA board

TVM can offload most convolution operations to the FPGA (40x speedup on off-loadable layers)



TVM can exploit latency hiding mechanisms to improve throughput. Utilization improves from at best 52% to 74%.



Resources

- Build your own simple VTA accelerator: https://gitlab.cs.washington.edu/cse599s/lab1
- TVM Tutorial for VTA to be released
- Looking for alpha users of the full VTA open source design

moreau@uw.edu

Thank you

moreau@uw.edu